技术文档

目录

[1. Java 3](#_Toc482001267)

[1.1 IO 3](#_Toc482001268)

[**1.1.1** 流操作相关类、接口 3](#_Toc482001269)

[**1.1.2** 流 4](#_Toc482001270)

[1.2 JVM机制 6](#_Toc482001271)

[1.3 MyBatis 7](#_Toc482001272)

[**1.3.1** SqlMapConfig.xml 7](#_Toc482001273)

[1.4 Spring 8](#_Toc482001274)

[**1.4.1** 概述 8](#_Toc482001275)

[**1.4.2** 配置实例 9](#_Toc482001276)

[**1.4.3** 注解 9](#_Toc482001277)

[1.5 Spring MVC 10](#_Toc482001278)

[**1.5.1** Spring Web MVC 简介 10](#_Toc482001279)

[**1.5.2** spring Web MVC架构 10](#_Toc482001280)

[**1.5.3** DispatcherServlet 12](#_Toc482001281)

[**1.5.4** Servlet 14](#_Toc482001282)

[**1.5.5** 文件上传 14](#_Toc482001283)

[1.6 通信 15](#_Toc482001284)

[**1.6.1** DataBase 15](#_Toc482001285)

[1.7 线程（同步、异步方法） 17](#_Toc482001286)

[**1.7.1** 创建、启动、周期 17](#_Toc482001287)

[**1.7.2** 管理、死锁 17](#_Toc482001288)

[**1.7.3** 同步 19](#_Toc482001289)

[**1.7.4** 线程通信 20](#_Toc482001290)

[**1.7.5** 线程池 23](#_Toc482001291)

[**1.7.6** 相关类 28](#_Toc482001292)

[**1.7.7** java 锁的种类及辨析 30](#_Toc482001293)

[**1.7.8** 关键属性功能 32](#_Toc482001294)

[**1.7.9** thread Dump 33](#_Toc482001295)

[1.8 整合 35](#_Toc482001296)

[**1.8.1** spring+springmvc+mybatis 35](#_Toc482001297)

[1.9 过时方法 40](#_Toc482001298)

[1.10 二十三种设计模式 40](#_Toc482001299)

[**1.10.1** 三大类 40](#_Toc482001300)

[2. WEB 48](#_Toc482001301)

[2.1 javascript 48](#_Toc482001302)

[2.2 jquery 49](#_Toc482001303)

[3. Linux 50](#_Toc482001304)

[3.1 Sys 50](#_Toc482001305)

[3.1.1 Command 50](#_Toc482001306)

[3.1.2 Operate 50](#_Toc482001307)

[3.2 Vim 57](#_Toc482001308)

[3.2.1 文本编辑 57](#_Toc482001309)

[3.2.2 多行注释、取消注释 58](#_Toc482001310)

[3.2.3 文本查看 58](#_Toc482001311)

[3.3 Nginx 59](#_Toc482001312)

[3.4 Redis 59](#_Toc482001313)

[3.4.1 安装 59](#_Toc482001314)

[3.4.2 自带工具 60](#_Toc482001315)

[3.4.3 Redis的配置 61](#_Toc482001316)

[3.5 Svn 62](#_Toc482001317)

[3.6 MySQL 62](#_Toc482001318)

[3.6.1 RDBMS 62](#_Toc482001319)

[3.6.2 安装、配置 63](#_Toc482001320)

[3.6.3 数据操作 66](#_Toc482001321)

[3.6.4 sql 67](#_Toc482001322)

[3.6.5 异常错误 67](#_Toc482001323)

[3.6.6 存储过程 68](#_Toc482001324)

[4. WIN 74](#_Toc482001325)

[4.1 Cmd 74](#_Toc482001326)

[5. Tools 74](#_Toc482001327)

[5.1 Eclipse 74](#_Toc482001328)

[5.1.1 配置 74](#_Toc482001329)

[5.1.2 插件引用 75](#_Toc482001330)

[5.1.3 preferences设置 79](#_Toc482001331)

[5.1.4 项目设置 81](#_Toc482001332)

[5.1.5 快捷键 81](#_Toc482001333)

[5.2 Idea 82](#_Toc482001334)

[5.2.1 永久激活方法 82](#_Toc482001335)

[5.2.2 配置文件修改 83](#_Toc482001336)

[5.2.3 工具优化 84](#_Toc482001337)

[5.2.4 手动添加tomcat 86](#_Toc482001338)

[5.2.5 手动添加SVN 87](#_Toc482001339)

[5.2.6 代码提示快捷键 89](#_Toc482001340)

[5.3 Maven 92](#_Toc482001341)

[5.3.1 坐标 92](#_Toc482001342)

[5.3.2 依赖 92](#_Toc482001343)

[5.4 Svn 93](#_Toc482001344)

[5.5 Tomcat 93](#_Toc482001345)

[5.5.1 ProjectName.XML 93](#_Toc482001346)

[5.5.2 项目发布目录设置 94](#_Toc482001347)

[6. Topic 94](#_Toc482001348)

[6.1 Global 94](#_Toc482001349)

[6.2 SpringMvc 97](#_Toc482001350)

[6.3 WEB 99](#_Toc482001351)

[6.4 Computing 99](#_Toc482001352)

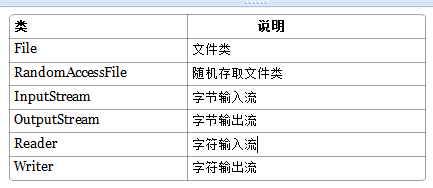
[6.5 Java 100](#_Toc482001353)

[6.6 JavaThread 102](#_Toc482001354)

# Java

## IO

* + 1. 流操作相关类、接口



**File类** 对文件系统中文件以及文件夹进行封装的对象，可以通过对象的思想来操作文件和文件夹

注：File类保存文件或目录的各种元数据信息，包括文件名、文件长度、最后修改时间、是否可

读、获取当前文件的路径名，判断指定文件是否存在、获得当前目录中的文件列表，创建、删除文件和目录等方法

**RandomAccessFile类** 封装了字节流，同时还封装了一个缓冲区（字符数组），通过内部的指针

来操作字符数组中的数据

**特点：** 1、只能操作文件，所以构造函数接收两种类型的参数：字符串文件路径、File对

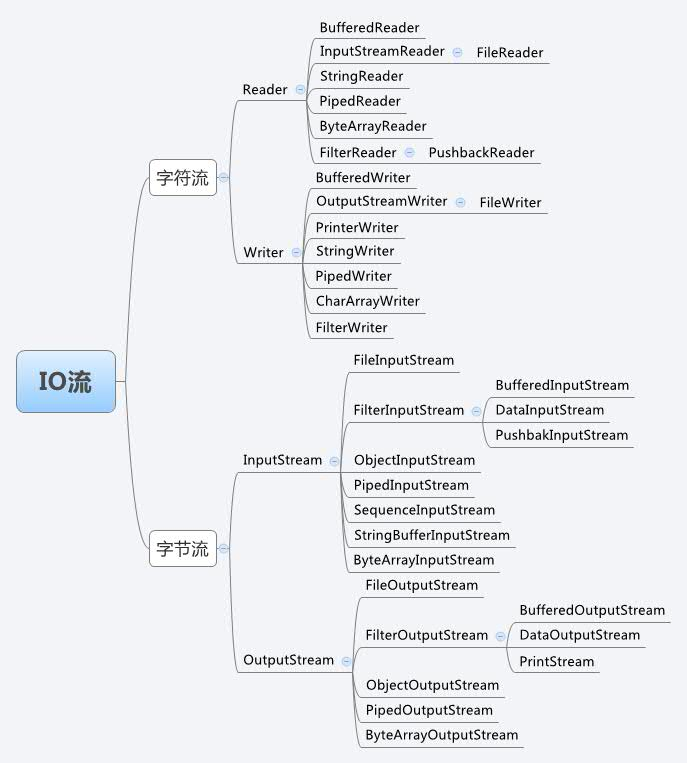
象（可为String）

2、既可以对文件进行读操作，也能进行写操作，在进行对象实例化时可指定操作

模式(r,rw)

注：该对象在实例化时，如果要操作的文件不存在，会自动创建；如果文件存在，写数据未指定位

置，会从头开始写，即覆盖原有的内容。 可以用于多线程下载或多个线程同时写数据到文件

* + 1. 流

**概念**：一组有顺序的，有起点和终点的字

节集合，是对数据传输的总称或抽象。即数据

在两设备间的传输称为流，流的本质是数据传输

**作用：**方便更直观的进行数据操作。

**分类**：根据数据类型的不同分为：字符流和字节流；根据数据流向不同分为：输入流和输出流

**字符流的由来**： 因为数据编码不同，有了对字符进行高效操作的流对象。本质是基于字节流读取时，查了指定的码表

**字节流和字符流的区别：1、**读写单位不同：字节流以字节(8bit)为单位，字符流以字符为单位，根据码表映射字符，一次可能读多个字节2、处理对象不同：字节流能处理所有类型的数据（如图片、avi等），而字符流只能处理字符类型的数据

结论：只要是处理纯文本数据，就优先考虑使用字符流。 除此之外都使用字节流

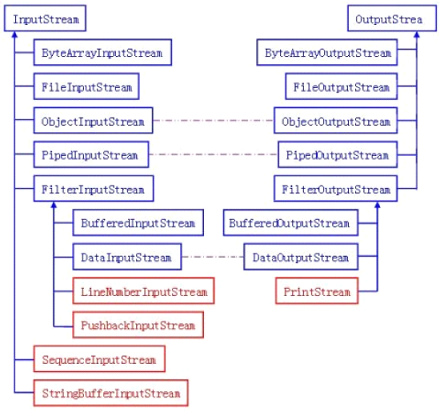
**输入流和输出流的区别：**输入流只能进行读操作，输出流只能进行写操作

**字符流与字节流转换**

|  |  |
| --- | --- |
| 转换流的特点 | 可对读取到的字节数据经过指定编码转换成字符 or 字节 |
| 何时使用转换流 | 1、当字节和字符之间需要转换时  2、流操作的数据需要编码或解码时 |
| 具体的对象体现 | 1、InputStreamReader 字节到字符的桥梁  2、OutputStreamWriter 字符到字节的桥梁 |

注：本身是字符流，所以在构造的时候需要传入字节流对象进来

* + - 1. 字节输入、输出流

蓝色为对应部分、红色为不对应部分、紫色虚线代表这些流一般要搭配使用

LineNumberInputStream 主要完成从流中读取数据时，会得到相应的行号，至于什么时候分行、在哪里分行是由该类确定，并不是在原始中有这样一个行号。

注：在输出部分没有对应的部分，我们可以自己建立一个LineNumberOutputStream，在最初写入时会有一个基准的行号，以后每次遇到换行时会在下一行添加一个行号

PushbackInputStream 的功能是查看最后一个字节，不满意就放入缓冲区。主要用在编译器的语法、词法分析部分。输出部分的BufferedOutputStream 实现相近的功能

注：可以使用unread()将数据推回流，给用户第二次读的机会

StringBufferInputStream 已经被Deprecated。本身就不应该出现在InputStream 部分，主要因为String 应该属于字符流的范围。已经被废弃了，还允许它存在只是为了保持版本的向下兼容

SequenceInputStream将两个或者多个输入流当成一个输入流依次读取

PrintStream向其他输出流、FileInputStream 写入数据。System.out 是PrintStream 的实例

**输入字节流InputStream：**

1. InputStream 是所有的输入字节流的父类，它是一个抽象类
2. ByteArrayInputStream、StringBufferInputStream、FileInputStream三种基本的介质流，分别

从Byte 数组、StringBuffer、本地文件中读取数据。PipedInputStream 是从与其它线程共用的管道中读取数据

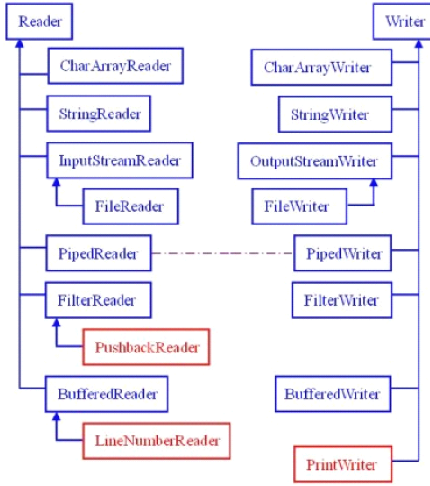
1. ObjectInputStream 和FilterInputStream 的子类都是装饰流（装饰器模式的主角）

**输出字节流OutputStream**

1. OutputStream 是所有的输出字节流的父类，它是一个抽象类
2. ByteArrayOutputStream、FileOutputStream 两种基本的介质流，它们分别向Byte 数组、本地

文件中写入数据。PipedOutputStream 是向与其它线程共用的管道中写入数据

1. ObjectOutputStream 和所有FilterOutputStream 的子类都是装饰流。
   * + 1. 字符输入、输出流

**Reader**所有的输入字符流的父类，它是一个抽象类

1、CharArrayReader、StringReader 两种基本的介质流，分别从Char 数组、String中读取数据。PipedReader 从与其它线程共用的管道中读取数据

2、BufferedReader 一个装饰器，它和其子类负责装饰

其它Reader 对象

3、FilterReader 所有自定义装饰流的父类，其子类

PushbackReader 对Reader 对象进行装饰，会增加一个行号

4、InputStreamReader 是一个连接字节流和字符流的桥梁，它将字节流转变为字符流。FileReader是一个达到此功能、常用的工具类，在其源代码中使用了将FileInputStream 转变为Reader 的方法

**Writer** 是所有的输出字符流的父类，它是一个抽象类

1、CharArrayWriter、StringWriter 两种基本的介质流，分别向Char 数组、String 中写入数据。PipedWriter 向与其它线程共用的管道中写入数据

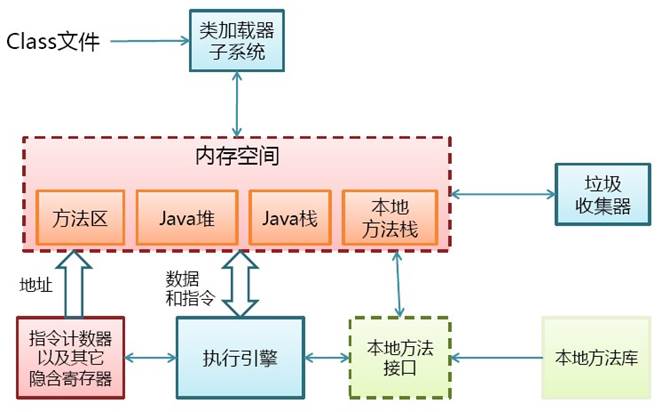
2、BufferedWriter 是一个装饰器为Writer 提供缓冲功能

3、PrintWriter （字符16bit）和PrintStream（字节8bit） 功能和使用非常相似

4、OutputStreamWriter 是OutputStream 到Writer 转换的桥梁，子类FileWriter 其实是一个实现

此功能的具体类

## JVM机制



**内存回收：**

1. 标记计数器（方法区）：内存对象中引用一次加1，释放一次减1，当这个对象的标记计数器为0时，就可以被GC回收了
2. 根搜索算法（执行引擎、本地方法接口）：对象引用状态分为强引用、弱引用、软引用、虚引用，引用不可达即无引用时回收
3. 回收时间会在虚拟机空闲、内存紧张时执行。注：java耗内存原因之一

**减少GC开销：**

1. 减少System.gc()的调用，此函数是建议JVM执行主GC，增加了主GC的执行频率，也增加了系统间歇停顿的次数
2. 减少临时对象的使用，对象跳出应用后成为垃圾，少量临时变量可以减少垃圾的产生，从而延长触发主GC的时间
3. 对象不使用时可置为NULL，一般NULL对象被认为垃圾，有利于GC收集器的判定，从而提高GC效率
4. 减少静态变量的使用，静态变量作为全局变量不会被GC收集，应用结束后才会被收集
5. 字符串累加时尽量使用StringBuffer减少String的使用，String是固定长度对象，每次累加都会产生新的对象，之前成为垃圾对象；StringBuffer为可变长度对象，可在原有对象基础进行扩增，不会产生中间的垃圾对象
6. 尽量使用基本类型，减少使用引用类型，基本类型占用内存比引用类型占用的很少
7. 分散对象创建或删除，突然大量创建对象需要大量内存，突然删除对象出现大量垃圾，这都会增加JVM强制执行主GC机会

**怎样确定哪些是垃圾：**

**引用计数** 对象引用一次计数器加1，失效一次减1

注：不能判断循环引用，计数器会增加额外的内存开销，JDK1.1以后不再使用

**根搜索** 1、虚拟机栈中引用的对象（栈帧中本地变量表）

2、本地方法中类静态属性引用的对象

3、方法区域中变量引用的对象

4、本地方法栈JNI中引用的对象

**垃圾回收机制中的算法：**

1. 标记-清除算法（Mark-Sweep），标记阶段确定需要回收的对象，清除阶段就是清除哪些带有标记的不可用对象

注：效率不高，清除后存在大量不连续的内存空间

1. 标记-复制算法Coping，将内存空间分成两部分，将存活的的对象复制到另一部分，然后清除当前内存块

注：算法简单、效率高，只适用一般内存，利用率低。新生代占用比例大，约8：1

1. 标记-整理算法（Mark-Compact），将对象向内存的一端移动，然后清除边界以外内存。

注：充分利用内存，有利于老年代对象收集

1. 分代收集算法（Generational Collection），根据对象存活时间划分为新生代、年轻代、老年代。新生代和年轻代采用标记-复制算法，老年代采用标记-整理算法

## MyBatis

* + 1. SqlMapConfig.xml

|  |  |
| --- | --- |
| **配置内容和顺序如下（顺序不能乱）** | |
| Properties | 属性 |
| Settings | 全局参数设置 |
| typeAliases | 类型别名 |
| typeHandlers | 类型处理器 |
| objectFactory | 对象工厂 |
| plugins | 插件 |
| environments | 环境信息集合 |
| environment | 单个环境信息 |
| transactionManager | 事物 |
| dataSource | 数据源 |
| mappers | 映射器 |

**transactionManager节点**：定义ibatis的事务管理器

* JDBC：通过传统JDBC Connection.commit/rollback实现事务支持
* JTA： 使用容器提供的JTA服务实现全局事务管理.
* EXTERNAL：外部事务管理,如在EJB中使用ibatis，通过EJB的部署配置即可实现自动的事务管理机制。此时ibatis将把所有事务委托给外部容器进行管理。此外,通过[spring](http://lib.csdn.net/base/javaee)等轻量级容量实现事务的配置化管理

**dataSource 节点**：从属于 transactionManager, 用于设定ibatis运行期使用DataSource属性

**type** **属性**：dataSource 元素的 type 属性指定 dataSource 的实现类型

* SIMPLE：ibatis 内置的 dataSource 实现,其中实现了一个简单的数据库连接池i机制,对应 ibatis 实现类为com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory
* JDBC：数据库自己的事务（局部事务），connect.beginTranstion(), connect.commit()等
* DBCP：基于Apache DBCP连接池组件实现的DataSource封装,无容器提供DataSource服务时，建议使用该选项,对应ibatis实现类为

com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory

* JTA：jta 事务管理器管理事务（全局事务），使用userTranstion对象
* JNDI：J2EE容器提供的DataSource实现, DataSource将通过指定的JNDI Name从容器中获取.对应ibatis实现类为com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory

注：Ibatis 不控制事务，事务交由外部控制，一般在CTM,或spring托管事务中使用。

JNDI的配置大部分都在应用服务器中进行，所以在ibatis中的配置相对简单

jdbc事务控制的JNDI配置

**<** **transctionManager**  type ="JDBC" **>**

**<** **dataSource**  type ="JNDI" **>**

**<** **property**  name ="DataSource"  value ="java:comp/env/jdbc/myDataSource" **/>**

**</** **dataSource** **>**

**</** **transctionManager** **>**

JTA事务控制的JNDI配置

**<** **transctionManager**  type ="JTA" **>**

**<** **property**  name ="UserTransaction"  value ="java:/ctx/con/UserTransaction" **/>**

**<** **dataSource**  type ="JNDI" **>**

**<** **property**  name ="DataSource"  value ="java:comp/env/jdbc/myDataSource" **/>**

**</** **dataSource** **>**

**</** **transctionManager** **>**

**mappers（映射器）**

* 相对路径 如：<mapper resource="sqlmap/User.xml" />
* 绝对路径 如：<mapper url="file:///D:\workspace\_spingmvc\mybatis\_01\config\sqlmap\User.xml" />

**mapper接口的限定名** 如：<mapper class="cn.itcast.mybatis.mapper.UserMapper"/>

注：此种方法要求mapper接口和mapper映射文件要名称相同，且放到同一个目录下

**注册指定包下的所有映射文件** 如：<package name="cn.itcast.mybatis.mapper"/>

注：此种方法要求mapper接口和mapper映射文件要名称相同，且放到同一个目录下

## Spring

* + 1. 概述

**Spring优点**

1、方便解耦，简化开发：通过Spring提供的IoC容器，可以将对象之间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合

注：IoC全称Iversion of Controller，控制反转：这概念是说你不用创建对象，而只需要描述它如何被创建。你不在代码里直接组装你的组件和服务，但是要在配置文件里描述哪些组件需要哪些服务，IOC容器负责把他们组装起来

    2、AOP编程的支持

注：AOP全称：Aspect-Oriented Programming，面向切面编程：把可重用的功能提取出来，然后将这些通用功能在合适的时候织入到应用程序中，比如事务管理、权限控制、日志记录、性能统计等

    3、声明式事务的支持

    4、方便集成各种优秀框架

    5、降低Java EE API的使用难度：如对JDBC，JavaMail，远程调用等提供了简便封装

**AOP优点：**AOP是一种更好的办法，能够用更少的工作量来解决现有的一些问题，使得系统更加健壮，可维护性更好

**Spring的事务管理：**对一系列的数据库操作（比如插入多条数据）进行统一的提交或回滚操作，如果插入成功，那么一起成功，如果中间有一条出现异常，那么回滚之前的所有操作。这样可以防止出现脏数据，防止数据库数据出现问题

* 声明式事务通常是指在配置文件中对事务进行配置声明，其中包括了很多声明属性，它是通过Spring Proxy帮你做代理，自己不用额外的写代码，只要在Spring配置文件中声明即可；通常用在数据库的操作里面；

注：不必去处理获得连接、关闭连接、事务提交和回滚等这些操作，也无需我们在与事务相关的方法中处理大量的try…catch…finally代码

* 编程式事务就是指通过硬编码的方式做事务处理，这种处理方式需要写代码，事务中的逻辑可以自己定制；可以是数据库，也可以是其他的操作
* Spring自己的事务管理机制，一般是使用TransactionMananger进行管理，可以通过Spring的注入来完成此功能

**Spring框架支持以下五种bean的作用域：**

* singleton : 默认值，bean在每个Spring ioc 容器中只有一个实例
* prototype：一个bean的定义可以有多个实例。
* request：每次http请求都会创建一个bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。
* session：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
* global-session：在一个全局的HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效

**注解映射器**

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping

**注解适配器**

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

spring3.1之后使用 <mvc:annotation-driven /> 代替上边注解映射器和注解适配器配置

**Spring事务的传播属性：**当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。

如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行

|  |  |
| --- | --- |
| PROPAGATION\_REQUIRED | 如果没有事务，就新建一个事务；如果已经存在一个事务，加入到这个事务中 |
| PROPAGATION\_SUPPORTS | 支持当前事务，如果当前没有事务，就以非事务方式执行 |
| PROPAGATION\_MANDATORY | 使用当前的事务，如果当前没有事务，就抛出异常 |
| PROPAGATION\_REQUIRES\_NEW | 新建事务，如果当前存在事务，把当前事务挂起 |
| PROPAGATION\_NOT\_SUPPORTED | 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起 |
| PROPAGATION\_NEVER | 以非事务方式执行，如果当前存在事务，则抛出异常 |
| PROPAGATION\_NESTED | 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作 |

* + 1. 配置实例

**启用注解**

<context:annotation-config/> 或者

<context:component-scan base-package="com.tgb.web.controller.annotation">

</context:component-scan>

* + 1. 注解

**@RequestMapping** 1). url映射      2). uri请求映射      3). http请求方法

@Controller //标注UserController成为处理请求的控制器

@RequestMapping("/owner/{ownerId}") //处理来自/owner URI的请求

public class UserController {

@CookieValue(value="sessionId",required=false)String sessionId

@SessionAttributes("user") // 将模型属性自动保存到HttpSession中 @RequestMapping("user/{userId}",method = RequestMethod.POST) //处理 POST 请求

//将表单值映射到User对象中 @PathVariable 将URL中的占位符参数绑定到控制器处理方法的入参中，类和方法处的占位符都可以绑定到处理方法的入参中

public ModelAndView cteateUser(@PathVariable("ownerId")String ownerId,

@PathVariable("userId")String userId,User user) {

//此处是 org.springframework.web.servlet 包下 ModelAndView

ModelAndView mav = new ModelAndView();

mav.setViewName("menu"); // menu.html

mav.addObject("user", user);

return mav;

return "redirect:handle72.html";  //向handle72发起一个新的请求

}

}

## Spring MVC

* + 1. Spring Web MVC 简介

一种基于Java的实现了Web MVC设计模式的请求驱动类型的轻量级Web框架，即使用了MVC架构模式的思想，将web层进行职责解耦，基于请求驱动指的就是使用请求-响应模型，框架的目的就是帮助我们简化开发，Spring Web MVC也是要简化我们日常Web开发的。

**优点：** √ 与Spring框架集成（如IoC容器、AOP等）；

√ 提供强大的约定大于配置的契约式编程支持；

√ 能简单的进行Web层的单元测试；

√ 支持灵活的URL到页面控制器的映射；

√ 非常容易与其他视图技术集成，如Velocity、FreeMarker等等，因为模型数据不放在特定的API里，而是放在一个Model里（Map数据结构实现，因此很容易被其他框架使用）；

√ 非常灵活的数据验证、格式化和数据绑定机制，能使用任何对象进行数据绑定，不必实现特定框架的API；

√ 提供一套强大的JSP标签库，简化JSP开发；

√ 支持灵活的本地化、主题等解析；

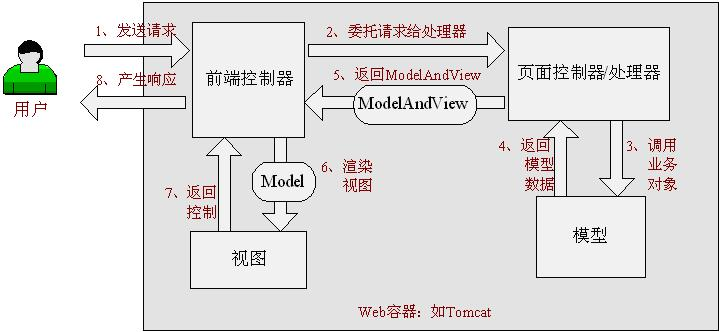
√ 更加简单的异常处理；

√ 对静态资源的支持；

√ 支持Restful风格

注：Spring Web MVC也是服务到工作者模式的实现，但进行可优化。前端控制器是DispatcherServlet；应用控制器其实拆为处理器映射器(Handler Mapping)进行处理器管理和视图解析器(View Resolver)进行视图管理；页面控制器/动作/处理器为Controller接口（仅包含ModelAndView handleRequest(request, response) 方法）的实现（也可以是任何的POJO类）；支持本地化（Locale）解析、主题（Theme）解析及文件上传等；提供了非常灵活的数据验证、格式化和数据绑定机制；提供了强大的约定大于配置（惯例优先原则）的契约式编程支持

* + 1. spring Web MVC架构

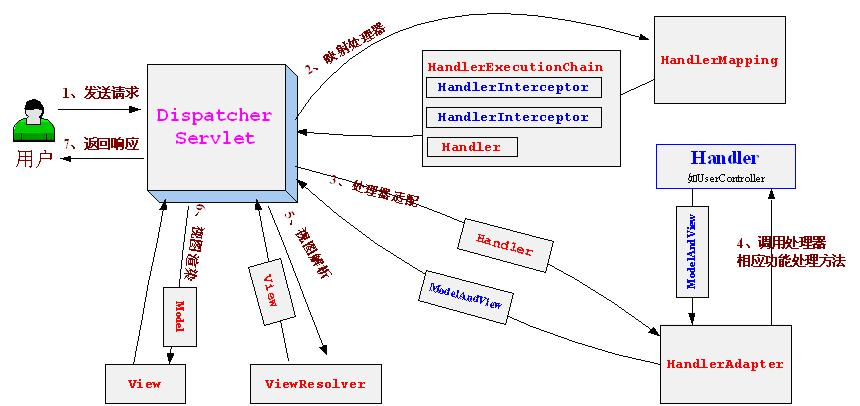


1、用户发送请求——>前端控制器，前端控制器根据请求信息（如URL）来决定选择哪一个页面控制器进行处理并把请求委托给它，即以前的控制器的控制逻辑部分；1、2步骤；

2、页面控制器接收到请求后，首先需要收集和绑定请求参数到一个对象，这个对象在Spring Web MVC中叫命令对象，并进行验证，然后将命令对象委托给业务对象进行处理；处理完毕后返回一个ModelAndView（模型数据和逻辑视图名）；3、4、5步骤；

3、前端控制器收回控制权，然后根据返回的逻辑视图名，选择相应的视图进行渲染，并把模型数据传入以便视图渲染；6、7步骤；

4、前端控制器再次收回控制权，将响应返回给用户，步骤8



1、用户发送请求——>DispatcherServlet，前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为统一访问点，进行全局的流程控制；

2、DispatcherServlet——>HandlerMapping， HandlerMapping将会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象，通过这种策略模式，很容易添加新的映射策略；

3、DispatcherServlet——>HandlerAdapter，HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；

4、HandlerAdapter——>处理器功能处理方法的调用，HandlerAdapter将会根据适配的结果调用真正的处理器的功能处理方法，完成功能处理；并返回一个ModelAndView对象（包含模型数据、逻辑视图名）；

5、ModelAndView的逻辑视图名——> ViewResolver， ViewResolver将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；

6、View——>渲染，View会根据传进来的Model模型数据进行渲染，此处的Model实际是一个Map数据结构，因此很容易支持其他视图技术；

7、返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

* + 1. DispatcherServlet

//前端控制器分派方法

**protected** **void** doDispatch(HttpServletRequest request, HttpServletResponse response) **throws** Exception {        HttpServletRequest processedRequest = request;

        HandlerExecutionChain mappedHandler = **null**;

**int** interceptorIndex = -1;

**try** {

           ModelAndView mv;

**boolean** errorView = **false**;

**try** { //检查是否是请求是否是multipart（如文件上传），如果是将通过MultipartResolver解析

              processedRequest = checkMultipart(request);

              //步骤2、请求到处理器（页面控制器）的映射，通过HandlerMapping进行映射

              mappedHandler = getHandler(processedRequest, **false**);

**if** (mappedHandler == **null** || mappedHandler.getHandler() == **null**) {

               noHandlerFound(processedRequest, response);

**return**;

              }

              //步骤3、处理器适配，即将我们的处理器包装成相应的适配器（从而支持多种类型的处器）

              HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

              // 304 Not Modified缓存支持  //省略代码

              // 执行处理器相关的拦截器的预处理（HandlerInterceptor.preHandle） //省略代码

              // 步骤4、由适配器执行处理器（调用处理器相应功能处理方法）

              mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

              // Do we need view name translation?

**if** (mv != **null** && !mv.hasView())

              mv.setViewName(getDefaultViewName(request));

              // 执行处理器相关的拦截器的后处理（HandlerInterceptor.postHandle） //省略代码

            } **catch** (ModelAndViewDefiningException ex) {

              logger.debug("ModelAndViewDefiningException encountered", ex);

              mv = ex.getModelAndView();

            } **catch** (Exception ex) {

              Object handler = (mappedHandler!=**null** ? mappedHandler.getHandler():**null**);

              mv = processHandlerException(processedRequest, response, handler, ex);

              errorView = (mv != **null**);

            }

//步骤5 由ViewResolver解析View（viewResolver.resolveViewName(viewName, locale)）

//步骤6 视图渲染把Model传入（view.render(mv.getModelInternal(),request,response);）

**if** (mv != **null** && !mv.wasCleared()) {

              render(mv, processedRequest, response);

**if** (errorView)

               WebUtils.clearErrorRequestAttributes(request);

            } **else** {

**if** (logger.isDebugEnabled())

                 logger.debug("Null ModelAndView returned to DispatcherServlet '" + getServletName() + "': assuming HandlerAdapter completed request handling");

            }

            // 处理器相关的拦截器的完成后处理（HandlerInterceptor.afterCompletion）//省略代码

        } **catch** (Exception ex) {

            // Trigger after-completion for thrown exception.

            triggerAfterCompletion(mappedHandler,interceptorIndex,processedRequest,response, ex);            **throw** ex;

        } **catch** (Error err) {

            ServletException ex = **new** NestedServletException("Handler failed",err);

            // Trigger after-completion for thrown exception.

            triggerAfterCompletion(mappedHandler,interceptorIndex,processedRequest,response, ex);            **throw** ex;

        } **finally** {

            // Clean up any resources used by a multipart request.

**if** (processedRequest != request)

                cleanupMultipart(processedRequest);

        }

    }

1、  请求如何给前端控制器？这个应该在web.xml中进行部署描述

2、  前端控制器如何根据请求信息选择页面控制器进行功能处理？ 我们需要配置HandlerMapping进行映射

3、  如何支持多种页面控制器呢？配置HandlerAdapter从而支持多种类型的页面控制器

4、  如何页面控制器如何使用业务对象？可以预料到，肯定利用Spring IoC容器的依赖注入功能

5、  页面控制器如何返回模型数据？使用ModelAndView返回

6、  前端控制器如何根据页面控制器返回的逻辑视图名选择具体的视图进行渲染？ 使用ViewResolver进行解析

7、  不同的视图技术如何使用相应的模型数据？ 因为Model是一个Map数据结构，很容易支持其他视图技术

**核心开发步骤：**

1、  DispatcherServlet在web.xml中的部署描述，从而拦截请求到Spring Web MVC

2、  HandlerMapping的配置，从而将请求映射到处理器

3、  HandlerAdapter的配置，从而支持多种类型的处理器

4、  ViewResolver的配置，从而将逻辑视图名解析为具体视图技术

5、处理器（页面控制器）的配置，从而进行功能处理

* + 1. Servlet

Servlet是运行在Servlet容器中的，常用的tomcat、jboss、weblogic都是Servlet容器，其生命周期是由容器来管理。Servlet的生命周期通过java.servlet.Servlet接口中的init（）、service（）、和destroy（）方法表示。

**Servlet的生命周期** 有四个阶段：加载并实例化、初始化、请求处理、销毁

加载并实例化：Servlet容器启动时，创建Servlet实例。Servlet容器启动后，Servlet通过类加载器来加载Servlet类，加载完成后再new一个Servlet对象来完成实例化

初始化：调用init（）方法，并传递实现ServletConfig接口的对象。在init（）方法中，Servlet可以读取配置参数，或者执行任何其他一次性活动

注：在Servlet的整个生命周期类，init（）方法只被调用一次

就绪/调用：有请求到达容器，容器调用servlet对象的service()方法。HttpServlet的service()方法，会依据请求方式来调用doGet()或者doPost()方法。但是，这两个do方法默认情况下，会抛出异常，需要子类去override

销毁：容器依据自身的算法，将不再需要的servlet对象删除掉。在删除之前，会调用servlet对象的destroy()方法释放资源

注：在servlet的整个生命周期当中，init,destroy只会执行一次，而service方法会执行多次

**Servlet的调用过程**

1.客户端通过发送请求给Tomcat，Tomcat发送客户端的请求页面给客户端

2.用户对请求页面进行相关操作后将页面提交给Tomcat,Tomcat将其封装成一个HttpRequest对象，然后对请求进行处理

3.Tomcat截获请求，根据action属性值查询xml文件中对应的servlet-name，再根据servlet-name查询到对应的java类（如果是第一次，Tomcat则会将servlet编译成java类文件，所以如果servlet有很多的话第一次运行的时候程序会比较慢）。

4.Tomcat实例化查询到的java类，注意该类只实例化一次。

5.调用java类对象的service()方法（如果不对service()方法进行重写则根据提交的方式来决定执行doPost()方法还是doGet()方法）。

6.通过request对象取得客户端传过来的数据，对数据进行处理后通过response对象将处理结果写回客户端。

* + 1. 文件上传

**spring.xml文件配置**

<bean id=**"multipartResolver"** class=**"org.springframework.web.multipart.commons.CommonsMultipartResolver"**> <property name=**"defaultEncoding"** value=**"UTF-8"** />

<property name=**"maxUploadSize"** value=**"209715200"** /><!-- 字节单位 209715200 即200M -->

<property name=**"maxInMemorySize"** value=**"2048"** /><!-- 2M -->

</bean>

/\*\* @description client上传至server \*/

@RequestMapping**(**"/upload.do"**)**

public void upload**(**HttpServletRequest request**,** HttpServletResponse response**)** **{**

// 创建一个通用的多部分解析器

CommonsMultipartResolver multipartResolver **=** **new** CommonsMultipartResolver**(**request**.**getSession**().**getServletContext**());**

**if** **(**multipartResolver**.**isMultipart**(**request**))** **{**// 判断 request 是否有文件上传,即多部分请求

MultipartHttpServletRequest multiRequest **=** **(**MultipartHttpServletRequest**)** request**;** // 转换成多部分request

Iterator**<**String**>** iter **=** multiRequest**.**getFileNames**();**// 取得request中的所有文件名

List**<**MultipartFile**>** list **=** multiRequest**.**getFiles**(**"file"**);**//取得request中的所有文件对象

**for** **(**MultipartFile multipartFile **:** list**)** **{**

int pre **=** **(**int**)** System**.**currentTimeMillis**();**// 上传起始时间

MultipartFile file **=** multiRequest**.**getFile**(**iter**.**next**());**// 取得上传文件

**if** **(**multipartFile **!=** **null)** **{**

String myFileName **=** file**.**getOriginalFilename**();**// 取得当前上传文件的文件名称

// 如果名称不为"",说明该文件存在，否则说明该文件不存在,trim()去掉字符串首尾的空格

**if** **(**myFileName**.**trim**()** **!=** ""**)**

multipartFile**.**transferTo**(new** File**(**"D:/filePath"**+** myFileName**));**//将上传文件保存到一个目标文件中

**}**

System**.**out**.**println**((**int**)** System**.**currentTimeMillis**()-** pre**);** // 上传该文件后的时间 - 上传起始时间

**}**

**}**

**}**注：byte[] getBytes() 获取文件数据

String getContentType() 获取文件MIME类型

InputStream getInputStream() 获取文件流

String getName() 获取表单中文件组件的名字

String getOriginalFilename() 获取上传文件的原名

long getSize() 获取文件的字节大小，单位byte

boolean isEmpty() 是否有上传的文件

void transferTo(File dest) 将上传文件保存到一个目标文件中

<form action=**"**<%=request.getContextPath()%>**/file/upload.do"** enctype=**"multipart/form-data"** method=**"post"**>

<input class='fileName' type=**"file"** name=**"file" multiple**>

<input class=**"confirm"** type=**"submit"** value=**"提交"**>

</form>

## 通信

* + 1. DataBase

**参数配置**

maxRequests：同时执行 SQL 的最大线程数。大于这个值的线程将阻塞直到另一个线程执行完成。这个值应该至少是 maxTransactions 的 10 倍，总是大于 maxSessions 和maxTranactions。减小这个参数值可以提高性能。

如：maxRequests=“256”  缺省值：512

maxSessions：同一时间内活动的最大 session 数。一个 session 可以maxSessions是代码请求的显式 session，也可以是当线程使用SqlMapClient 实例（即执行一条语句）自动获得的session。它应该总是大于或等于 maxTransactions 并小于 maxRequests。减小这个参数值可以减少内存使用

如：maxSessions=“64”  缺省值：128

maxTransactions：同时进入 SqlMapClient.startTransaction()的最大线程maxTransactions 数。大于这个值的线程将阻塞直到另一个线程退出。不同的 DBMS 有不同的限制值，但任何数据库都有这些限制。这个参数值应该总是小于或等于maxSessions 并总是远远小于 maxRequests。减小这个参数值通常能提高性能。

如：maxTransactions=“16”  缺省值：32

cacheModelsEnabled：全局性地启用或禁用 SqlMapClient 的所有缓存cacheModelsEnabled model。调试程序时使用。

如：cacheModelsEnabled=“true”  缺省值：true（启用）

lazyLoadingEnabled：全局性地启用或禁用SqlMapClient的所有延迟加载，调试程序时使用

如：lazyLoadingEnabled=“true”  缺省值：true（启用）

enhancementEnabled：全局性地启用或禁用运行时字节码增强，以优化访问Java Bean属性的性能，同时优化延迟加载的性能

如：enhancementEnabled=“true”  缺省值：false（禁用）

useStatementNamespaces：启用本属性，必须使用全限定名来引用 Mapped statement

Mapped statement 的全限定名由 sql-map 的名称和 mapped-statement 的名称合成

如：queryForObject("sqlMapName.statementName")

如：useStatementNamespaces=“false”  缺省值：false（禁用）

defaultStatementTimeout：此设置值必须为一个整数，作为JDBC连接超时的时间，这个值可以被任意一个映射文件的statement属性来重新设置，如果没有指明，查询将无超时时间限制除非在映射文件中设置statement属性值。被设置的值以秒为单位等待连接失效

**事务简介**：一系列的动作，它们被当作一个单独的工作单元。这些动作要么全部完成，要么全部不起作用。

作用：用来确保数据的完整性和一致性

四个关键属性(ACID)

① 原子性(atomicity):事务是一个原子操作，有一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用

② 一致性(consistency):一旦所有事务动作完成，事务就被提交。数据和资源就处于一种满足业务规则的一致性状态中

③ 隔离性(isolation):可能有许多事务会同时处理相同的数据，因此每个事物都应该与其他事务隔离开来，防止数据损坏

④ 持久性(durability):一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响。通常情况下，事务的结果被写到持久化存储器中

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重复读 | 幻象读 | 第一类丢失更新 | 第二类丢失更新 |
| READ UNCOMMITED | 允许 | 允许 | 允许 | 不允许 | 允许 |
| READ COMMITED | 不允许 | 允许 | 允许 | 不允许 | 允许 |
| REPEATABLE READ | 不允许 | 不允许 | 允许 | 不允许 | 不允许 |
| SERIALIZABLE | 不允许 | 不允许 | 不允许 | 不允许 | 不允许 |

SqlServer2008R2的默认隔离级别是 READ COMMITED ，MySQL的默认隔离级别是 REPEATABLE READ

**并发事务所导致的问题：**多个事务在同一个数据集上并发执行时，可能会出现许多意外的问题

① 脏读：脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的

② 不可重复读：不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间更新了数据

③ 幻读：幻读与不可重复读类似。它发生在一个事务(T1)读取了几行数据，接着另一个并发事务(T2)插入了一些数据时。在随后的查询中，第一个事务(T1)就会发现多了一些原本不存在的记录

## 线程（同步、异步方法）

* + 1. 创建、启动、周期

**Thread：** a）重写run()

b) 创建Thread对象实例即线程对象

c) 调用线程对象的start()方法启动线程

**Runnable：** a) 重写run()

b) 创建Runnable实现类实例

c) 以创建Runnable实现类实例对象作为Thread的Target对象即线程对象

d) 调用线程对象的start()

**Callable：** a) 创建实现Callable接口的类对象 Callable oneCallable = new SomeCallable();

b) 由Callable创建一个FutureTask对象

         FutureTask oneTask = new FutureTask(oneCallable);

c) 由FutureTask创建一个Thread对象 Thread oneThread = new Thread(oneTask);

d) 调用线程对象即启动线程 oneThread.start();

**线程生命周期**

1、新建：创建线程对象开辟内存空间

2、就绪：调用start()方法处加入线程就绪队列（可运行池），使线程对象具备运行条件，

3、运行：获得cpu调度，并调用run()方法开始执行程序

4、阻塞： a） 重复调用start()方法会出现[Java](http://lib.csdn.net/base/17).lang.IllegalThreadStateException异常（非法线程状态）

b） 运行状态变阻塞状态的5中场景：

①  线程调用sleep方法主动放弃所占用的系统资源

②  线程调用一个阻塞式IO方法，在该方法返回前，该线程被阻塞

③  线程视图获取一个同步监视器，但该监视器正在被其它线程持有

④  线程在等待某个通知（notify）

注：阻塞状态下的线程不能进入就绪队列，需先消除阻塞。

5、死亡：1 当线程run()方法执行完 1 出现异常 1 强制终止interrupt()

* + 1. 管理、死锁

**线程睡眠** – Thread.sleep(long millis)：线程睡眠并进入阻塞状态（依然持有对象锁，只有睡眠时间结束才能进入就绪状态）。由于受系统控制，我们不能精准的干涉它，所以睡眠时间可能会大于设定时间。 注：sleep方法声明后会抛出InterruptedException

**线程让步** - Thread.yield()：线程暂停并进入就绪状态，优先级与当前线程优先级相同或可能更高。可能刚进入就绪状态又立即被调度到运行状态。

**线程合并** - join：等待当前线程执行完后执行另一个线程

void join()

当前线程等该加入该线程后面，等待该线程终止。 void join(long millis)

当前线程等待该线程终止的时间最长为 millis 毫秒。 如果在millis时间内，该线程没有执行完，那么当前线程进入就绪状态，重新等待cpu调度 void join(long millis,int nanos)

等待该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。如果在millis时间内，该线程没有执行完，那么当前线程进入就绪状态，重新等待cpu调度

**设置线程优先级**：

Thread类提供了setPriority(int newPriority)和getPriority()方法来设置和返回一个指定线程的优先级

setPriority方法的参数是一个整数，范围是1~10之间，也可以使用Thread类提供的三个静态常量： MAX\_PRIORITY =10 MIN\_PRIORITY =1 NORM\_PRIORITY =5

虽然Java提供了10个优先级别，但这些优先级别需要操作系统的支持。不同的操作系统的优先级并不相同，而且也不能很好的和Java的10个优先级别对应。所以我们应该使用MAX\_PRIORITY、MIN\_PRIORITY和NORM\_PRIORITY三个静态常量来设定优先级，这样才能保证程序最好的可移植性。

**后台（守护）进程**

setDaemon(boolean on) ：守护线程并非虚拟机内部可以提供，用户也可以自行的设定守护线程，不用去关心结束。

on - 如果为 true，则将该线程标记为守护线程。

a)、thread.setDaemon(true)必须在thread.start()之前设置，首先调用该线程的 checkAccess 方法且不带任何参数。从安全管理器中判断是否有修改该线程的权限，若不允许修改则抛出 SecurityException，然后判断如果该线程处于运行状态则会抛出IllegalThreadStateException异常。

注：这点与守护进程有着明显的区别，守护进程是创建后，让进程摆脱原会话的控制+让进程摆脱原进程组的控制+让进程摆脱原控制终端的控制；所以说寄托于虚拟机的语言机制跟系统级语言有着本质上面的区别

b)、在Daemon线程中产生的新线程也是Daemon的。

注：守护进程fork()出来的子进程不再是守护进程，尽管它把父进程的进程相关信息复制过去了，但是子进程的父进程不是init进程，所谓的守护进程本质上说就是“父进程挂掉，init收养，然后文件0,1,2都是/dev/null，当前目录到/”）

c)、不是所有的应用都可以分配给Daemon线程来进行服务，比如读写操作或者计算逻辑。因为在Daemon Thread还没来的及进行操作时，虚拟机可能已经退出了。 IllegalThreadStateException - 如果该线程处于活动状态。 SecurityException - 如果当前线程无法修改该线程。

**正确结束线程**

正常执行完run()方法后结束，通过标识符等保证run()方法正常结束

Thread.stop()、Thread.suspend()、Thread.resume()、Runtime.runFinalizersOnExit(boolean value)这些终止线程运行的方法已经被废弃了，使用它们是极端不安全的！

**结束当前线程的方法**

interrupt()只能解决InterruptedException的阻塞的线程，IO阻塞异常需要使用套接close方法

**死锁的四个必要条件**

① 互斥条件：资源不能被共享，一个资源一次只能由一个进程使用，如果有其它进程申请使用该资源，申请进程必须等待直到所申请的资源被释放。

② 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放

③ 非剥夺条件：已经分配的资源不能从相应的进程中被强制剥夺

④ 循环等待条件：系统中若干进程组成环路，该环路中每个进程都在等待相邻进程

占用的资源。例：进程A中包含资源A,进程B中包含资源B，A的下一步需要资源B，B的下一步需要资源A，所以它们就互相等待对方占有的资源释放，所以也就产生了一个循环等待死锁。

**处理死锁的方法**

* 忽略该问题，也即鸵鸟算法。当发生问题时，不管它，直接跳过，无视它；
* 检测死锁并恢复；
* 资源进行动态分配；
* 破除上面的四种死锁条件之一

由来：传说中鸵鸟看到危险就把头埋在地底下。当你对某一件事情没有一个很好的解决方法时，那就忽略它，就像鸵鸟面对危险时会把它深埋在沙砾中，装作看不到。这样的算法称为“鸵鸟算法“。例如在计算机操作系统中，当死锁真正发生且影响系统正常运行时，手动干预—重新启动

* + 1. 同步

**作用：**java允许多线程并发控制，当多个线程同时操作一个共享资源时（数据库增、删、改、查），将会导致数据不准缺，相互之前产生冲突。因此加入同步锁可以保证该线程在没有完成操作之前，被其它线程的调用，保证数据的唯一性和准确性

**同步方法：**synchronized关键字修饰的方法。由于java的每个对象都有一个内置锁，用此关键字修饰非静态方法时，内置锁会保护整个方法；修饰静态方法时会锁住整个类。在调用该方法前，需要获得内置锁，否则就处于阻塞状态

public synchronized void save(){}

**同步块：**有synchronized关键字修饰的语句块。被该关键字修饰的语句块会自动被加上内置锁，从而实现同步

public void save(){synchronized (this) { System.out.println("同步块");   }}

注：同步是一种高开销的操作，因此应该尽量减少同步的内容。通常没有必要同步整个方法，使用synchronized代码块同步关键代码即可。

**使用特殊域变量(volatile)** a) 每次使用该域就要重新计算，而不是使用寄存器中的值；

b) volatile仅实现了Synchronized的可见性，未提供任何原子操作

c) volatile和final不能同时使用，二者功能冲突

注：volatile是一个类型修饰符，它是被设计用来修饰被不同线程访问和修改的变量，可以被异步的线程所修改；final必须对它赋予初值并且不能修改它。

**使用重入锁（Lock）实现线程同步：**在JavaSE5.0中新增了一个java.util.concurrent包来支持同步。ReentrantLock类是可重入、互斥、实现了Lock接口的锁，它与使用synchronized方法和快具有相同的基本行为和语义，并且扩展了其能力。ReenreantLock类的常用方法有：

//声明锁 private Lock lock = new ReentrantLock(false); //不公平锁，不考虑排序（default）

注：不公平情况下，是无序状态允许插队，jvm会自动计算如何处理更快速来调度插队。

private Lock lock = new ReentrantLock(true); //公平锁，保证线程队列顺序

// 获得锁 lock.lock()

// 释放锁 lock.unlock():

// 是否被锁 lock.tryLock()

// 中断锁 lock.lockInterruptibly()

注：相比synchornized有更高的伸缩性，ReentrantLock不公平锁比synchornized公平锁的性能高

**场景分析：**

1. 如果发现该操作已经在执行中则不再执行（有状态执行）

private ReentrantLocklock **=** **new** ReentrantLock**();**

**if(**lock**.**tryLock**()){** //如果已经被lock，则立即返回false不会等待，达到忽略操作的效果

**try{** //操作

**}finally{**lock**.**unlock**();}**

**}**

a、用在定时任务时，如果任务执行时间超过下次计划执行时间，确保该有状态任务只有一个正在执行（忽略重复触发）

b、用在界面交互时点击执行较长时间请求操作时，防止多次点击导致后台重复执行（忽略重复触发）

注：多用于进行非重要任务防止重复执行，（如：清除无用临时文件，检查某些资源的可用性，数据备份操作等）

1. 如果发现该操作已经在执行，等待一个一个执行（同步执行，类似synchronized）

private ReentranLock lock **=** **new** ReentrantLock**(true);** //参数默认false，不公平锁

**try{** lock**.**lock**();**//如果被其它资源锁定，会在此等待锁释放，达到暂停的效果

//操作

**}finally{**lock**.**unlock**();}**

1. 如果发现该操作已经在执行，则尝试等待一段时间，等待超时则不执行（尝试等待执行）

**try{**

**if(**lock**.**tryLock**(**5**,**TimeUnit**,**SECONDS**)){** //如果已经被lock，尝试等待5s，看是否可以

**try{** //操作

**}finally{** lock**.**unlock**();** **}**

**}**

**}catch(**InterruptedException e**){**

e**.**printStackTrace**();**//当前线程被中断时(interrupt)，会抛出InterruptedExcetpion

**}**

用来防止由于资源处理不当长时间占用导致死锁情况（大家都在等待资源，导致线程队列溢出）

1. 如果发现该操作已经在执行，等待执行。这时可中断正在进行的操作立刻释放锁继续下一操作。

**try{** lock**.**lockInterruptibly**();**

//操作

**}** **catch(**InterruptedException e**){**

e**.**printStackTrace**();**

**}** **finally{** lock**.**unlock**();** **}**

synchronized与Lock在默认情况下是不会响应中断(interrupt)操作，会继续执行完。

主要用于取消某些操作对资源的占用。如：（取消正在同步运行的操作，来防止不正常操作长时间占用造成的阻塞）

* + 1. 线程通信

**Object类的wait()、notify()和notifyAll()**

  wait()放弃运行资格，处于冻结状态；

     线程运行时，内存中会建立一个线程池，冻结状态的线程都存在于线程池中，notify()执行时唤醒的也是线程池中的线程，线程池中有多个线程时唤醒第一个被冻结的线程。

    notifyall(), 唤醒线程池中所有线程。

注： （1） wait(), notify(),notifyall()都用在同步里面，因为这3个函数是对持有锁的线程进行操作，而只有同步才有锁，所以要使用在同步中；

       （2） wait(),notify(),notifyall() 在使用时必须标识它们所操作的线程持有的锁，因为等待和唤醒必须是同一锁下的线程；而锁可以是任意对象，所以这3个方法都是Object类中的方法。

class Resource{

private String name;

private boolean flag=false;

public synchronized void set(String name){ //生产者

while(flag) /\*while持续判断flag，等待生产者线程从冻结状态醒来\*/

try{ wait(); }catch(Exception e){}

flag=true;

this.notifyAll();/\*原先是notity(), 现在改成notifyAll(),这样生产者线程生产完一个商品后可以将等待中的消费者线程唤醒，否则只将上面改成while后，可能出现所有生产者和消费者都在wait()的情况。\*/

}

public synchronized void out(){ //消费者

while(!flag)

try{wait();}catch(Exception e){}

flag=false; this.notifyAll();

}

}

public class ProducerConsumerDemo{

public static void main(String[] args){

Resource r=new Resource();

Producer pro=new Producer(r); Consumer con=new Consumer(r);

Thread t1=new Thread(pro); Thread t2=new Thread(con); Thread t3=new Thread(pro); Thread t4=new Thread(con);

t1.start(); t2.start(); t3.start(); t4.start();

}

}

**Condition控制**

（1）将同步synchronized替换为显式的Lock操作；

    （2）将Object类中的wait(), notify(),notifyAll()替换成了Condition对象，该对象可以通过Lock锁对象获取;

    （3）一个Lock对象上可以绑定多个Condition对象，这样实现了本方线程只唤醒对方线程，而jdk1.5之前，一个同步只能有一个锁，不同的同步只能用锁来区分，且锁嵌套时容易死锁。

class Resource{

private boolean flag=false;

private Lock lock = new ReentrantLock();/\*Lock接口，ReentrantLock是该接口的直接子类\*/

private Condition condition\_pro=lock.newCondition(); /\*创建生产者的Condition对象\*/

private Condition condition\_con=lock.newCondition(); /\*使用同一个锁，创建消费者的Condition对象\*/

public void set(String name){ lock.lock();//锁住此语句与lock.unlock()之间的代码

try{

while(flag)

condition\_pro.await(); //生产者线程在conndition\_pro对象上等待 System.out.println(Thread.currentThread().getName()+"...生产者...");

flag=true; condition\_con.signalAll(); //\*唤醒所有在condition\_pro对象下等待的线程，也就是唤醒所有消费者线程\*/

} finally{

lock.unlock(); //unlock()要放在finally块中。

}

}

public void out(){

lock.lock();

try{

while(!flag)

condition\_con.await();

System.out.println(Thread.currentThread().getName()+"...消费者...");

flag=false;

condition\_pro.signalAll();

} finally{ lock.unlock(); }

}

}

**阻塞队列（BlockingQueue）控制**

BlockingQueue接口 extends Queue的子接口。

当生产者线程试图向BlockingQueue中放入元素时，如果该队列已满，则线程被阻塞；当消费者线程试图从BlockingQueue中取出元素时，如果队列已空，则该线程阻塞。

程序的两个线程通过交替向BlockingQueue中放入元素、取出元素，即可很好地控制线程的通信

阻塞方法：

（1）put(E e)：尝试把Eu元素放如BlockingQueue中，如果该队列的元素已满，则阻塞该线程。

（2）take()：尝试从BlockingQueue的头部取出元素，如果该队列的元素已空，则阻塞该线程。

BlockingQueue继承的Queue接口中方法：

  （1）队尾插入元素，包括add（E e）、offer（E e）、put（E e）方法，当该队列已满时，这三个方法分别会抛出异常、返回false、阻塞队列。

（2）队头部删除并返回删除的元素。包括remove（）、poll（）、和take（）方法，当该队列已空时，这三个方法分别会抛出异常、返回false、阻塞队列。

（3）队头部取出但不删除元素。包括element（）和peek（）方法，当队列已空时，这两个方法分别抛出异常、返回false。

BlockingQueue接口的5个实现类：

ArrayBlockingQueue ：基于数组实现的队列。 LinkedBlockingQueue：基于链表实现的队列。

PriorityBlockingQueue：它并不是保准的阻塞队列，该队列调用remove（）、poll（）、take()等方法提取出元素时，并不是取出队列中存在时间最长的元素，而是队列中最小的元素。

它判断元素的大小即可根据元素（实现Comparable接口）的本身大小来自然排序，也可使用Comparator进行定制排序。 SynchronousQueue：同步队列。对该队列的存、取操作必须交替进行。

DelayQueue：特殊的BlockingQueue，底层基于PriorityBlockingQueue实现，不过，DelayQueue要求集合元素都实现Delay接口（该接口里只有一个long getDelay（）方法），根据集合元素的getDalay（）方法的返回值进行排序。

**public class** BlockingQueueTest{

**public static void** main(String[] args)**throws** Exception{

//创建一个容量为1的BlockingQueue

BlockingQueue<String> b = **new** ArrayBlockingQueue<>(1);

//启动2个生产者线程

**new** Producer(b).start(); **new** Producer(b).start();

//启动1个消费者线程

**new** Consumer(b).start();

}

}

**class** Producer **extends** Thread{

**private** BlockingQueue<String> **b**;

**public** Producer(BlockingQueue<String> b){ **this**.**b**=b; }

**public synchronized void** run(){

String [] str=**new** String[]{"java","struts","Spring"};

**for**(**int** i=0;i<9999;i++){

**try**{

**b**.put(str[i%3]); //尝试放入元素，如果队列已满，则线程被阻塞

}**catch**(Exception e){System.***out***.println(e);}

}

}

}

**class** Consumer **extends** Thread{

**private** BlockingQueue<String> **b**;

**public** Consumer(BlockingQueue<String> b){ **this**.**b**=b; }

**public synchronized void** run(){

**while**(**true**){

**try**{

**b**.take(); //尝试取出元素，如果队列已空，则线程被阻塞

}**catch**(Exception e){System.***out***.println(e);}

}

}

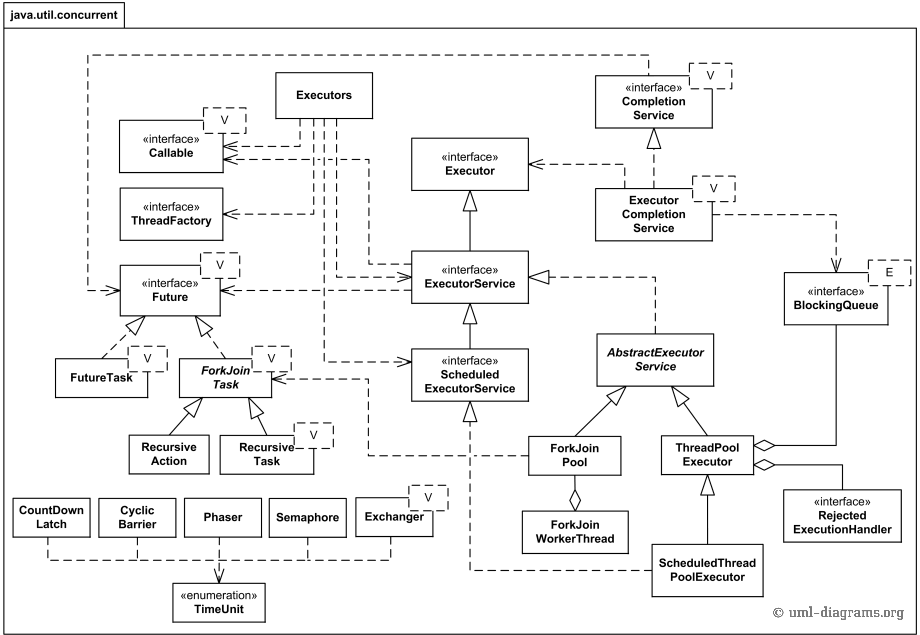
}

* + 1. 线程池

合理利用线程池的三个好处：

1. 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
2. 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。
3. 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控
   * + 1. Executors工厂类

  Executor线程池框架的最大优点是把任务的提交和执行解耦。客户端将要执行的任务封装成Task，然后提交即可。而Task如何执行客户端则是透明的。具体点讲，提交一个Callable对象给ExecutorService（如最常用的线程池ThreadPoolExecutor），将得到一个Future对象，调用Future对象的get方法等待执行结果。线程池实现原理类结构图如下：



注：1、Executor接口和Executors工厂类 2、Executor接口只有一个execute()方法

**Executors执行多线程任务的步骤**：

  • 调用Executors类的静态工厂方法创建一个ExecutorService对象，该对象代表一个线程池（例：newSingleThreadExecutor()）；

  • 创建Runnable实现类或Callable实现类的实例，作为线程执行任务；

  • 调用ExecutorService对象的submit()方法来提交Runnable实例或Callable实例；

  • 不想提交任务时，调用ExecutorService对象的shutdown()方法关闭线程池；

注：仅关闭线程池并未关闭停止以运行的线程

**Executors的静态工厂类创建线程池的方法：**

① newFixedThreadPool() ：

作用：该方法返回一个固定线程数量的线程池，该线程池中的线程数量始终不变，即不会再创建新的线程，也不会销毁已经创建好的线程，自始自终都是那几个固定的线程在工作，所以该线程池可以控制线程的最大并发数。

例子：新任务提交时，线程池中如果有空闲的线程则立即使用空闲线程来处理任务，如果没有，则会把这个新任务存在一个任务队列中，一旦有线程空闲了，则按FIFO（先进先出）方式处理任务队列中的任务。

② newCachedThreadPool() ：

作用：该方法返回一个可以根据实际情况调整线程池中线程的数量的线程池。即该线程池中的线程数量不确定，是根据实际情况动态调整的。

例子：假如该线程池中的所有线程都正在工作，而此时有新任务提交，那么将会创建新的线程，或者之前有一些线程完成了任务，不会创建新线程，而是复用空闲的线程去处理新任务。

提问：该线程池的线程会不会随着新任务的提交越集越多？不会，因为线程池中的线程都有一个“保持活动时间”的参数，通过配置它，如果线程池中的空闲线程的空闲时间超过该“保存活动时间”则立刻停止该线程，而该线程池默认的“保持活动时间”为60s

③ newSingleThreadExecutor() ：

作用：该方法返回一个只有一个线程的线程池，即每次只能执行一个线程任务，多余的任务会保存到一个任务队列中，等待这一个线程空闲，当这个线程空闲了再按FIFO方式顺序执行任务队列中的任务。

④ newScheduledThreadPool() ：

作用：该方法返回一个可以控制线程池内线程定时或周期性执行某任务的线程池。

⑤ newSingleThreadScheduledExecutor() ：

作用：该方法返回一个可以控制线程池内线程定时或周期性执行某任务的线程池。只不过和上面的区别是该线程池大小为1，而上面的可以指定线程池的大小。

注：Executors只是一个工厂类，它所有的方法返回的都是ThreadPoolExecutor、

ScheduledThreadPoolExecutor这两个类的实例。

**ExecutorService方法：**

执行方法

**void** execute(Runnable command); //该方法接收Runnable实例，并异步执行

提交任务方法

<T> Future<T> submit(Callable<T> task);

<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);

注：1、submit和execute区别是前者可以返回一个Future对象，通过返回的Future

对象，我们可以检查提交的任务是否执行完毕。

2、任务执行完成，Runnable任务执行run()方法future.get()返回null，Callable

任务执行call()方法future.get()返回执行结果。注意，future.get()方法会产生阻塞。

接收一个Callable的集合，执行这个方法不会返回Future，但是会返回所有Callable

任务中其中一个任务的执行结果。无法保证返回的是哪个任务的执行结果，反正是其中的某一个

<T> T invokeAny(Collection<? **extends** Callable<T>> tasks)

**throws** InterruptedException, ExecutionException;

<T> T invokeAny(Collection<? **extends** Callable<T>> tasks,**long** timeout, TimeUnit unit)

**throws** InterruptedException, ExecutionException, TimeoutException;

与 invokeAny 类似，也是接收一个Callable集合，但是invokeAll执行之后会返回一个Future的List，其中对应着每个Callable任务执行后的Future对象

<T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks)

**throws** InterruptedException;

<T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks,

**long** timeout, TimeUnit unit) **throws** InterruptedException;

**ExecutorService关闭方法**

**void** shutdown();

     当我们使用完成ExecutorService之后应该关闭它，否则它里面的线程会一直处于运行状态。举个例子，如果的应用程序是通过main()方法启动的，在这个main()退出之后，如果应用程序中的ExecutorService没有关闭，这个应用将一直运行。之所以会出现这种情况，是因为ExecutorService中运行的线程会阻止JVM关闭

    在调用shutdown()方法之后，ExecutorService不会立即关闭，但是它不再接收新的任务，直到当前所有线程执行完成才会关闭，所有在shutdown()执行之前提交的任务都会被执行。

List<Runnable> shutdownNow(); 立即关闭ExecutorService。这个动作将跳过所有正在执行的任务和被提交还没有执行的任务。但是对于正在执行的任务，有可能它们都会停止，也有可能执行完成

* + - 1. ForkJoinPool

  在Java 8中，引入了自动并行化的概念。它能够让一部分Java代码自动地以并行的方式执行，前提是使用了ForkJoinPool。

  ForkJoinPool同ThreadPoolExecutor一样，也实现了Executor和ExecutorService接口。它使用了一个无限队列来保存需要执行的任务，而线程的数量则是通过构造函数传入，如果没有向构造函数中传入希望的线程数量，那么当前计算机可用的CPU数量会被设置为线程数量作为默认值。

  ForkJoinPool主要用来使用分治法(Divide-and-Conquer Algorithm)来解决问题。典型的应用比如快速排序算法。这里的要点在于，ForkJoinPool需要使用相对少的线程来处理大量的任务。比如要对100万个数据进行排序，那么会将这个任务分割成两个50万的排序任务和一个针对这两组50万数据的合并任务。以此类推，对于50万的数据也会做出同样的分割处理，到最后会设置一个阈值来规定当数据规模到多少时，停止这样的分割处理。比如，当元素的数量小于10时，会停止分割，转而使用插入排序对它们进行排序。那么到最后，所有的任务加起来会有大概2000000+个。问题的关键，对于一个任务而言，只有当它所有的子任务完成之后，它才能够被执行。ThreadPoolExecutor中的线程无法像任务队列中再添加一个任务并且在等待该任务完成之后再继续执行，所以不能使用分治法。而使用ForkJoinPool时，就能够让其中的线程创建新的任务，并挂起当前的任务，此时线程就能够从队列中选择子任务执行。

**static class** CalTask **extends** RecursiveTask<Integer>{ //继承RecursiveTask来实现"可分解"的任务

**private static final int *THRESHOLD*** = 20; // 每个“小任务”只最多只累加20个数

**private int arr**[], **start**, **end**;

**public** CalTask(**int**[] arr , **int** start, **int** end) { // 累加从start到end的数组元素

**this**.**arr** = arr; **this**.**start** = start; **this**.**end** = end;

}

@Override

**protected** Integer compute(){

**int** sum = 0;

**if**(**end** - **start** < ***THRESHOLD***) { // 当end与start之间的差小于THRESHOLD时，开始进行实际累加

**for** (**int** i = **start** ; i < **end** ; i++ ) { sum += **arr**[i]; }

**return** sum;

}**else**{ // 如果当end与start之间的差大于THRESHOLD时，即要打印的数超过20个

**int** middle = (**start** + **end**) /2; // 将大任务分解成两个小任务*。*

CalTask left = **new** CalTask(**arr** , **start**, middle);

CalTask right = **new** CalTask(**arr** , middle, **end**);

// 并行执行两个“小任务”

left.fork(); right.fork();

// 把两个“小任务”累加的结果合并起来

**return** left.join() + right.join();

}

}

}

**public static void** main(String[] args) **throws** Exception {

**int**[] arr = **new int**[100];

Random rand = **new** Random();

**int** total = 0;

// 初始化100个数字元素

**for** (**int** i = 0 , len = arr.**length**; i < len ; i++ ) {

**int** tmp = rand.nextInt(20);

total += (arr[i] = tmp); // 对数组元素赋值，并将数组元素的值添加到total总和中

}

ForkJoinPool pool = **new** ForkJoinPool();

Future<Integer> future = pool.submit(**new** CalTask(arr , 0 , arr.**length**)); // 提交可分解的CalTask任务

System.***out***.println(future.get());

pool.shutdown(); // 关闭线程池

}

注：关键是fork()和join()方法。在ForkJoinPool使用的线程中，会使用一个内部队列来对需要执行的任务以及子任务进行操作来保证它们的执行顺序

**ThreadPoolExecutor和ForkJoinPool的性能差异：**

  （1）首先，使用ForkJoinPool能够使用数量有限的线程来完成非常多的具有父子关系的任务，比如使用4个线程来完成超过200万个任务。但是，使用ThreadPoolExecutor时，是不可能完成的，因为ThreadPoolExecutor中的Thread无法选择优先执行子任务，需要完成200万个具有父子关系的任务时，也需要200万个线程，显然这是不可行的。

（2）ForkJoinPool能够实现工作窃取(Work Stealing)，在该线程池的每个线程中会维护一个队列来存放需要被执行的任务。当线程自身队列中的任务都执行完毕后，它会从别的线程中拿到未被执行的任务并帮助它执行。因此，提高了线程的利用率，从而提高了整体性能。

（3）对于ForkJoinPool，还有一个因素会影响它的性能，就是停止进行任务分割的那个阈值。比如在之前的快速排序中，当剩下的元素数量小于10的时候，就会停止子任务的创建。

结论：

1. 当需要处理递归分治算法时，考虑使用ForkJoinPool；
2. 仔细设置不再进行任务划分的阈值，这个阈值对性能有影响；
3. Java 8中的一些特性会使用到ForkJoinPool中的通用线程池。在某些场合

下，需要调整该线程池的默认的线程数量

* + 1. 相关类

**ThreadLocal**:每个线程创建对象，而不是引用

它并不是一个线程，而是一个可以在每个线程中存储数据的数据存储类，通过它可以在指定的线程中存储数据，数据存储之后，只有在指定线程中可以获取到存储的数据，对于其他线程来说则无法获取到该线程的数据。 即多个线程通过同一个ThreadLocal获取到的东西是不一样的，就算有的时候出现的结果是一样的（偶然性，两个线程里分别存了两份相同的东西），但他们获取的本质是不同的。使用这个工具类可以简化多线程编程时的并发访问，很简洁的隔离多线程程序的竞争资源

 对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而

ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

get方法会返回一个当前线程的变量值，如果数组不存在就会创建一个新的。另外，对于“当前线程”和“数 组”，数组对于每个线程来说都是不同的 values.table。而values是通过当前线程获取到的一个Values对象，因此这个数组是每个线程唯一的，不能共用，而下面的几句话也更直接了，获取一个索引，再返回通过这个索引找到数组中对应的值。这也就解释了为什么多个线程通过同一个ThreadLocal返回的是不同的东西。

**Java中为什么要这么设置**

* ThreadLocal在日常开发中使用到的地方较少，但是在某些特殊的场景下，通过

ThreadLocal可以轻松实现一些看起来很复杂的功能。一般来说，当某些数据是以线程为作用域并且不同线程具有不同的数据副本的时候，就可以考虑使用ThreadLocal。例如在Handler和Looper中。对于Handler来说，它需要获取当前线程的Looper，很显然Looper的作用域就是线程并且不同的线程具有不同的Looper，这个时候通过ThreadLocal就可以轻松的实现Looper在线程中的存取。如果不采用ThreadLocal，那么系统就必须提供一个全局的哈希表供Handler查找指定的Looper，这样就比较麻烦了，还需要一个管理类。

* ThreadLocal的另一个使用场景是复杂逻辑下的对象传递，比如监听器的传递，

有些时候一个线程中的任务过于复杂，就可能表现为函数调用栈比较深以及代码入口的多样性，这种情况下，我们又需要监听器能够贯穿整个线程的执行过程。这个时候就可以使用到ThreadLocal，通过ThreadLocal可以让监听器作为线程内的全局对象存在，在线程内通过get方法就可以获取到监听器。如果不采用的话，可以使用参数传递，但是这种方式在设计上不是特别好，当调用栈很深的时候，通过参数来传递监听器这个设计太糟糕。而另外一种方式就是使用static静态变量的方式，但是这种方式存在一定的局限性，拓展性并不是特别的强。比如有10个线程在执行，就需要提供10个监听器对象。

注：ThreadLocal和其他所有的同步机制一样，都是为了解决多线程中对于同一变量的访问冲突。值普通的同步机制中，通过对象加锁来实现多线程对同一变量的安全访问，且该变量是多线程共享的，所有需要使用这种同步机制来明确分开是在什么时候对变量进行读写，在什么时候需要锁定该对象。此种情况下，系统并没有将这个资源复制多份，而是采取安全机制来控制访问而已。ThreadLocal只是从另一个角度解决多线程的并发访问，即将需要并发访问的资源复制多份，每个线程拥有一份资源，每个线程都有自己的资源副本。

总结：若多个线程之间需要共享资源，以达到线程间的通信时，就使用同步机制；若仅仅需要隔离多线程之间的关系资源，则可以使用ThreadLocal。

**public class** ThreadLocalTest { //创建一个Integer型的线程本地变量

**public static final** ThreadLocal<Integer> ***local*** = **new** ThreadLocal<Integer>() {

@Override

**protected** Integer initialValue() { **return** 0; }

};

**public static void** main(String[] args) **throws** InterruptedException {

Thread[] threads = **new** Thread[5];

**for** (**int** j = 0; j < 5; j++) {

threads[j] = **new** Thread(**new** Runnable() {

@Override

**public void** run() {

**int** num = ***local***.get(); //获取当前线程的本地变量，然后累加5次

**for** (**int** i = 0; i < 5; i++) { num++; }

***local***.set(num); //重新设置累加后的本地变量

System.***out***.println(Thread.*currentThread*().getName() + **" : "**+ ***local***.get());

}

}, **"Thread-"** + j);

}

**for** (Thread thread : threads) { thread.start(); }

}

}

源码解析：

**private** T setInitialValue() { //获取初始化值，initialValue 就是我们之前覆盖的方法

T value = initialValue();

Thread t = Thread.*currentThread*();

ThreadLocalMap map = getMap(t); //如果map不为空，将初始化值放入到当前线程的ThreadLocalMap对象中

**if** (map != **null**)

map.set(**this**, value);

**else**

createMap(t, value); //当前线程第一次使用本地线程变量，需要对map进行初始化工作

**return** value; //返回初始化值

}

**public** T get() { //获取当前执行线程

Thread t = Thread.*currentThread*(); //取得当前线程的ThreadLocalMap实例

ThreadLocalMap map = getMap(t); //如果map不为空，说明该线程已经有了一个ThreadLocalMap实例

**if** (map != **null**) { //map中保存线程的所有的线程本地变量，我们要去查找当前线程本地变量

ThreadLocalMap.Entry e = map.getEntry(**this**); //如果当前线程本地变量存在这个map中，则返回其对应的值

**if** (e != **null**)

**return** (T)e.value;

}

**return** setInitialValue(); //如果map不存在或者map中不存在当前线程本地变量，返回初始值

}

Thread对象都有一个ThreadLocalMap类型的属性threadLocals，这个属性是专门用于保存自己所有的线程本地变量的。这个属性在线程对象初始化的时候为null。所以对一个线程对象第一次使用线程本地变量的时候，需要对这个threadLocals属性进行初始化操作。注意要区别 “线程第一次使用本地线程变量”和“第一次使用某一个线程本地线程变量”

ThreadLocalMap getMap(Thread t) { **return** t.**threadLocals**; } //直接返回线程对象的threadLocals属性

* + 1. java 锁的种类及辨析

**作用：**锁作为并发共享数据，保证一致性的工具，JAVA平台有多种实现( synchronized 和 ReentrantLock等等 )

**锁的种类：**

1、自旋锁 2、自旋锁的其他种类 3、阻塞锁 4、可重入锁

5、读写锁 6、互斥锁 7、悲观锁 8、乐观锁

9、公平锁 10、非公平锁 11、偏向锁 12、对象锁

13、线程锁 14、锁粗化 15、轻量级锁 16、锁消除

17、锁膨胀 18、信号量

1、自旋锁：不公平，让当前线程不停地的在循环体内执行（忙循环），当循环的条件被其他线程改变时 才能进入临界区(跳出)

2、自旋锁的其他种类：公平，TicketLock 解决访问顺序，根据服务编号、CLHlock 和MCSlock以链表形式排序，前者为隐式的队列，没有真实的后继结点属性；后者为显式的队列，有真实的后继结点属性。

3、阻塞锁：让线程进入阻塞状态进行等待，当获得相应的信号（唤醒，时间） 时，才可以进入线程的准备就绪状态，准备就绪状态的所有线程，通过竞争，进入运行状态。

注：1、JAVA中，能够进入\退出、阻塞状态或包含阻塞锁的方法有 ，synchronized 关键字（其中的重量锁），ReentrantLock，Object.wait()\notify(),LockSupport.park()/unpart()(j.u.c经常使用) 2、阻塞锁的优势在于，阻塞的线程不会占用cpu时间， 不会导致 CPu占用率过高，但进入时间以及恢复时间都要比自旋锁略慢。

在竞争激烈的情况下 阻塞锁的性能要明显高于自旋锁。理想的情况则是; 在线程竞争不激烈的情况下，使用自旋锁，竞争激烈的情况下使用，阻塞锁。

4、可重入锁：广义上的可重入锁，而不是单指JAVA下的ReentrantLock；可重入锁，也叫做递归锁，指的是同一线程 外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。在JAVA环境下 ReentrantLock 和synchronized 都是 可重入锁

注：可重入锁最大的作用是避免死锁

5、读写锁：共享-独占锁.，维护了一对相关的锁，一个用于只读操作，一个用于写入操作。只要没有writer，读取锁可以由多个reader线程同时保持。写入锁是独占的。

注：当读写锁在读加锁状态时, 所有试图以读模式对它进行加锁的线程都可以得到访问权, 但是如果线程希望以写模式对此锁进行加锁, 它必须直到所有的线程释放锁.

通常, 当读写锁处于读模式锁住状态时, 如果有另外线程试图以写模式加锁, 读写锁通常会阻塞随后的读模式锁请求, 这样可以避免读模式锁长期占用, 而等待的写模式锁请求长期阻塞.

6、互斥锁：一次只允许一个线程访问共享数据，哪怕进行的是只读操作

注：读写锁与互斥锁相比，性能取决于读写操作期间读取数据相对于修改数据的频率，以及数据的争用——即在同一时间试图对该数据执行读取或写入操作的线程数。

7、悲观锁：操作之前先上锁。每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block（阻止）直到它拿到锁

8、乐观锁：每次去拿数据的时候不会上锁，但在更新的时候会判断一下在此期间别的线程有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量

注：如果经常产生冲突，上层应用会不断的进行retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

9、公平锁：并发环境中，每个线程在获取锁时会先查看此锁维护的等待队列，如果为空或当前线程线程是等待队列的第一个，就占有锁；否则就会加入到等待队列中，以后会按照FIFO的规则从队列中取到

10、非公平锁：直接尝试占有锁，如果尝试失败，就再采用类似公平锁那种方式

11、偏向锁：锁对象的对象头中有个Thread Id字段，这个字段如果是空的，第一次获取锁的时候，就将自身的ThreadId写入到锁的ThreadId字段内，将锁头内的是否偏向锁的状态位置为1.这样下次获取锁的时候，如果ThreadId是否和自身线程Id一致，则认为当前线程已经获取了锁，因此不需再次获取锁，略过了轻量级锁和重量级锁的加锁阶段。提高了效率。

注：偏向锁也有一个问题，就是当锁有竞争关系的时候，需要解除偏向锁，使锁进入竞争的状态。

12、对象锁： 所有对象都自动含有单一的锁

注：JVM负责跟踪对象被加锁的次数。如果一个对象被解锁，其计数变为0。在任务（线程）第一次给对象加锁的时候，计数变为1。每当这个相同的任务（线程）在此对象上获得锁时，计数会递增。

只有首先获得锁的任务（线程）才能继续获取该对象上的多个锁。

每当任务离开一个synchronized方法，计数递减，当计数为0的时候，锁被完全释放，此时别的任务就可以使用此资源。

13、线程锁：~~线程内部对象加锁导致线程锁定~~

14、锁粗化：如果虚拟机探测到有这样一串零碎的操作都对同一个对象加锁，将会把加锁同步的范围扩展到整个操作序列的外部，这样就只需要加锁一次就够了

15、轻量级锁：一种快速的锁定方法，在进入互斥之前，使用CAS操作来尝试加锁，尽量不要用操作系统层面的互斥，提高了性能

注：偏向锁失败时，轻量锁步骤

1) 将对象头的Mark指针保存到锁对象中（这里的对象指的就是锁住的对象，比如synchronized (this){}，this就是这里的对象）。

2) 将对象头设置为指向锁的指针（在线程栈空间中）。

lock位于线程栈中。所以判断一个线程是否持有这把锁，只要判断这个对象头指向的空间是否在这个线程栈的地址空间当中

如果轻量级锁失败，表示存在竞争，升级为重量级锁（常规锁），就是操作系统层面的同步方法。在没有锁竞争的情况，轻量级锁减少传统锁使用OS互斥量产生的性能损耗。在竞争非常激烈时（轻量级锁总是失败），轻量级锁会多做很多额外操作，导致性能下降。

16、锁消除：虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。锁消除主要判定依据来源于逃逸分析的数据支持

17、锁膨胀：等于锁粗化

18、信号量：仅有3种操作，且它们全部是原子的：初始化、增加和减少acquire()和release()

注：默认是不公平的，可在初始化时修改 Semaphore(**int** permits, **boolean** fair)

* + 1. 关键属性功能

**synchronized 、 ReentrantLock**

sychronized：jvm层面，最基本同步互斥的手段,可以修饰代码块,方法,类.

① 在修饰代码块的时候需要一个reference引用对象作为锁的对象.

② 在修饰方法的时候默认是当前对象作为锁的对象.

③ 在修饰类时候默认是当前类的Class对象作为锁的对象.

synchronized会在进入同步块的前后分别形成monitor enter和monitor exit

字节码指令.在执行monitorenter指令时会尝试获取对象的锁,如果此没对象没有被锁,或者此对象已经被当前线程锁住,那么锁的计数器加一,每当monitorexit被锁的对象的计数器减一.直到为0就释放该对象的锁.由此synchronized是可重入的,不会出现自己把自己锁死.

ReentrantLock：jdk层面，以对象的方式来操作对象锁.相对于sychronized需要在finally中去释放锁

区别：除了synchronized的功能,多了三个高级功能，等待可中断、公平锁、绑定多个Condition.

1.等待可中断：持有锁的线程长时间不释放锁的时候,等待的线程可以选择放弃等待tryLock(long timeout, TimeUnit unit)

2.公平锁：按照申请锁的顺序来一次获得锁称为公平锁.synchronized的是非公平锁,ReentrantLock可以通过构造函数实现公平锁.    new RenentrantLock(boolean fair)

3.绑定多个Condition：通过多次newCondition可以获得多个Condition对象,可以简单的实现比较复杂的线程同步的功能.通过await(),signal();

使用场景

① 资源竞争不是很激烈的情况使用synchronized性能高于synchronized。

但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍。而ReentrantLock确还能维持常态。

② 需要用到比synchronized多出的功能时

**FutureTask**

可取消的异步计算，FutureTask 实现了Future的基本方法，提供 start、cancel

操作，可以查询计算是否已经完成，并且可以获取计算的结果。结果只可以在计算完成之后获取，get方法会阻塞当计算没有完成的时候，一旦计算已经完成，那么计算就不能再次启动或是取消。一个FutureTask 可以用来包装一个 Callable 或是一个runnable对象。因为FurtureTask实现了Runnable方法，所以一个FutureTask可以提交(submit)给一个Excutor执行(excution)

FutureTask 有两个很重要的属性 分别是 state、runner ，futureTask之所以可以

支持cancel操作 就是因为这两个属性

state 枚举值

|  |  |  |
| --- | --- | --- |
| State | 状态 | 枚举值 |
| NEW | 新建 | 0 |
| COMPLETING | 执行中 | 1 |
| NORMAL | 正常 | 2 |
| EXCEPTIONAL | 异常 | 3 |
| CANCELLED | 取消 | 4 |
| INTERRUPTING | 中断中 | 5 |
| INTERRUNPED | 被中断 | 6 |

state 四种状态变化

NEW -> COMPLETING -> NORMAL 正常完成的流程

NEW -> COMPLETING -> EXCEPTIONAL 出现异常的流程

NEW -> CANCELED 被取消

NEW -> INTERRUNPING-> INTERRRUNPTED 被中断

**使用场景**

1、FutureTask执行多任务计算时，利用FutureTask和ExecutorService，可以用多线程的方式提交计算任务，主线程继续执行其他任务，当主线程需要子线程的计算结果时，在异步获取子线程的执行结果。

2、ConcurrentHashMap<String,FutureTask<Connection>> 高并发环境下确保任务只执行一次。假设有一个带key的连接池，当key存在时，即直接返回key对应的对象；当key不存在时，则创建连接。对于这样的应用场景，通常采用的方法为使用一个Map对象来存储key和连接池对应的对应关系

* + 1. thread Dump

UNIX中 可以使用kill-3 pid >filename 注：tomcat会记录在catalina.out

windows中 可以使用”CTRL+Break”。

**jstack Dump 日志文件中的线程状态**

死锁 Deadlock（重点关注）

执行中 Runnable

等待资源 Waiting on condition（重点关注）

等待获取监视器 Waiting on monitor entry（重点关注）

暂停 Suspended

对象等待中 Object.wait() 或 TIMED\_WAITING

阻塞 Blocked（重点关注）

停止 Parked

* + - 1. Waiting to lock、Blocked

"RMI TCP Connection(267865)-172.16.5.25" daemon prio=10 tid=0x00007fd508371000 nid=0x55ae**waiting for monitor entry**[0x00007fd4f8684000]

   java.lang.Thread.State: **BLOCKED (on object monitor)**

at org.apache.log4j.Category.callAppenders(Category.java:201)

- **waiting to lock <0x00000000acf4d0c0>** (a org.apache.log4j.Logger)

at org.apache.log4j.Category.forcedLog(Category.java:388)

at org.apache.log4j.Category.log(Category.java:853)

at org.apache.commons.logging.impl.Log4JLogger.warn(Log4JLogger.java:234)

at com.tuan.core.common.lang.cache.remote.SpyMemcachedClient.get(SpyMemcachedClient.java:110)

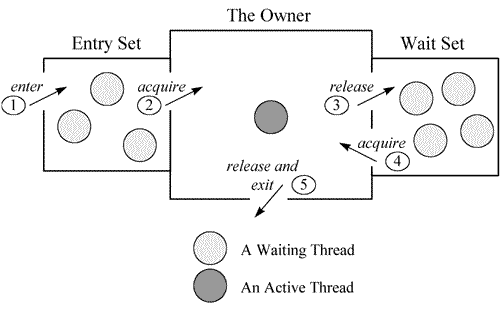
1）线程状态是 **Blocked**，阻塞状态。说明线程等待资源超时

2） waiting to lock <0x00000000acf4d0c0>线程在等待给 0x00000000acf4d0c0 地址上锁（英文可描述为：trying to obtain  0x00000000acf4d0c0 lock）

3）在 dump 日志里查找字符串 0x00000000acf4d0c0，发现有大量线程都在等待给这个地址上锁。如果能在日志里找到谁获得了这个锁（如locked < 0x00000000acf4d0c0 >），就可以顺藤摸瓜了

4）**waiting for monitor entry**此线程通过 synchronized(obj) {……} 申请进入了临界区，从而进入了下图中的 Entry Set 队列，但该 obj 对应的 monitor 被其他线程拥有，所以本线程在 Entry Set 队列中等待。

5）第一行里 RMI TCP Connection(267865)-172.16.5.25 是 Thread Name 。tid 指 Java Thread id。nid 指native线程的id。prio是线程优先级。[0x00007fd4f8684000] 是线程栈起始地址



* + - 1. Waiting on condition、TIMED\_WAITING

"RMI TCP Connection(idle)" daemon prio=10 tid=0x00007fd50834e800 nid=0x56b2 **waiting on condition** [0x00007fd4f1a59000]

   java.lang.Thread.State: **TIMED\_WAITING (parking)**

at sun.misc.Unsafe.park(Native Method)

- **parking to wait for  <0x00000000acd84de8>** (a java.util.concurrent.SynchronousQueue$TransferStack)

at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:198)

at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill(SynchronousQueue.java:424)

at java.util.concurrent.SynchronousQueue$TransferStack.transfer(SynchronousQueue.java:323)

at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:874)

at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:945)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:907)

at java.lang.Thread.run(Thread.java:662)

1）**TIMED\_WAITING (parking)** 中的 timed\_waiting 指等待状态，但这里指定了时间，到达指定的时间后自动退出等待状态；parking指线程处于挂起中

2）**waiting on condition** 需要与堆栈中的 **parking to wait for  <0x00000000acd84de8>** (a java.util.concurrent.SynchronousQueue$TransferStack) 结合来看。首先，本线程肯定是在等待某个条件的发生，来把自己唤醒。其次，SynchronousQueue 并不是一个队列，只是线程之间移交信息的机制，当我们把一个元素放入到 SynchronousQueue 中时必须有另一个线程正在等待接受移交的任务，这就是本线程在等待的条件

* + - 1. in Obejct.wait()、TIMED\_WAITING

"RMI RenewClean-[172.16.5.19:28475]" daemon prio=10 tid=0x0000000041428800 nid=0xb09 **in Object.wait()** [0x00007f34f4bd0000]

   java.lang.Thread.State: **TIMED\_WAITING (on object monitor)**

at java.lang.Object.wait(Native Method)

- **waiting on <0x00000000aa672478> (a java.lang.ref.ReferenceQueue$Lock)**

at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)

- **locked <0x00000000aa672478> (a java.lang.ref.ReferenceQueue$Lock)**

at sun.rmi.transport.DGCClient$EndpointEntry$RenewCleanThread.run(DGCClient.java:516)

at java.lang.Thread.run(Thread.java:662)

1）**TIMED\_WAITING (on object monitor)** 对于本例而言，是因为本线程调用了 java.lang.Object.wait(long timeout) 而进入等待状态

2）Wait Set 中等待的线程状态就是 **in Object.wait()** 。当线程获得了 Monitor，进入了临界区之后，如果发现线程继续运行的条件没有满足，它则调用对象（一般就是被 synchronized 的对象）的 wait() 方法，放弃了 Monitor，进入 Wait Set 队列。只有当别的线程在该对象上调用了 notify() 或者 notifyAll() ， Wait Set 队列中线程才得到机会去竞争，但是只有一个线程获得对象的 Monitor，恢复到运行态

3）RMI RenewClean 是 DGCClient 的一部分。[DGC](http://doc.java.sun.com/DocWeb/api/java.rmi.dgc.DGC?lang=zh_cn&mode=Read) 指的是 Distributed GC，即分布式垃圾回收

4）注意，先 **locked <0x00000000aa672478>**，后 **waiting on <0x00000000aa672478>**，之所以先锁再等同一个对象

例：即，线程的执行中，先用 synchronized 获得了这个对象的 Monitor（对应于  **locked <0x00000000aa672478>** ）；当执行到 lock.wait(timeout);线程就放弃了 Monitor 的所有权，进入 Wait Set 队列（对应于  **waiting on <0x00000000aa672478>** ）。

5）从堆栈信息看，是正在清理 remote references to remote objects ，引用的租约到了，分布式垃圾回收在逐一清理

## 整合

* + 1. spring+springmvc+mybatis
       1. 整合过程

① 整合dao层：通过spring管理mapper接口，使用mapper的扫描器自动扫描mapper接口在spring中进行注册

  ② 整合service层：通过spring管理 service接口，使用配置方式将service接口配置在spring配置文件中，实现事务控制

  ③整合springmvc：由于springmvc是spring的模块，不需要整合。

* + - 1. 主要配置

1) mybatis配置文件sqlMapConfig.xml配置别名自动扫描(实体类)

2) mapper扫描器(接口，数据库访问接口)

3) 数据库连接池配置

4)  声明式事务配置

5) 启用注解扫描：<context:component-scan base-

package="cn.itcast.ssm.controller"></context:component-scan>

6) 配置注解映射器和适配器： <mvc:annotation-driven></mvc:annotation-driven>

7) 视图解析器：<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

8) 配置控制类： DispatcherServlet前端控制器

9) 配置spring配置文件加载类：ClassLoadListener

* + - * 1. web.xml

<!--系统欢迎页 -->

<welcome-file-list><welcome-file>**index.html**</welcome-file></welcome-file-list>

**&&**

<servlet> <!--核心控制器 DispatcherServlet -->

<servlet-name>**springMVC**</servlet-name>

<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>

<init-param> <!--加载配置文件 -->

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

<param-value>**config/springMVC.xml**</param-value>

</init-param>

<load-on-startup>**1**</load-on-startup>

</servlet>

**&&**

<servlet-mapping> <!--Servlet拦截匹配规则 \*.action 或者 /(拦截了静态文件)-->

<servlet-name>**springMVC**</servlet-name>

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

</servlet-mapping>

**&&**

<context-param><!-- 指定业务层Spring容器的配置文件，多个用逗号隔开 -->

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

<param-value>**classpath:config/spring-config.xml**</param-value>

</context-param>

**&&**

<listener><!-- 通过contextConfigLocation指定的配置文件启动业务层的Spring容器，上下文监听器 -->

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

</listener>

**&&**

<servlet-mapping><!-- 设置\*.css不被拦截，DispatcherServlet拦截"/"，为了实现REST风格，拦截了所有的请求 -->

<servlet-name>**spring**</servlet-name>

<url-pattern>**\*.css**</url-pattern>

</servlet-mapping>

* + - * 1. spring-mvc.xml

<context:component-scan base-package=**"com.jiang.web.controller"**/><!-- 让Spring扫描包下标注注解的类生效 -->

**&&**

<mvc:annotation-driven /> <!-- 开启注解 -->

**&&**

<!-- 视图解析器 ，视图模式配置-->

<bean id=**"viewResolver"** class=**"org.springframework.web.servlet.view.InternalResourceViewResolver"**><!-- 自动给后面action的方法return的字符串加上前缀和后缀，变成一个 可用的url地址 -->

<property name=**"prefix"** value=**"/"** /> <!-- 前缀 -->

<property name=**"suffix"** value=**""** /> <!-- 后缀，可为空 -->

<property name=**"viewClass"** value=**"org.springframework.web.servlet.view.JstlView"** />

</bean>

* + - * 1. SqlMapConfig.xml

**<settings** cacheModelsEnabled ="true" <!--是否启用缓存机制-- >

**&&**

lazyLoadingEnabled="true"   <!-- 是否启用延迟加载机制 -->

**or**

aggressiveLazyLoading="true"   <!-- 将积极加载修改为消极加载，即延迟加载 -->

**&&**

enhancementEnabled="true"   <!-- 是否启用字节码增强机制 -->

errorTracingEnabled="true"   <!-- 是否启用错误处理机制 -->

cacheEnabled="true"   <!-- 是否开启二级缓存 -->

maxRequests="32"   <!-- 最大并发请求数 -->

maxSessions="10"   <!-- 最大Session数 -->

maxTransactions="5"   <!-- 最大并发事务数 -->

useStatementNamespaces="true"**/>**   <!-- 是否启用名称空间 -->

**<** **transactionManager**  type ="JDBC" **>**    <!-- 事务管理器-->

**<** **dataSource**  type ="SIMPLE" **>**    <!-- 数据源的链接类型 -->

**<** **property**  name ="JDBC.Driver"  value ="com.mysql.jdbc.Driver" **/>**

**<** **property**  name ="JDBC.ConnectionURL" value ="jdbc:mysql://localhost/aa" **/>**

**<** **property**  name ="JDBC.Username"  value ="root" **/>**

**<** **property**  name ="JDBC.Password"  value ="root" **/>**

**<** **property**  name ="Pool.MaximumActiveConnections" value ="10"**/>** <!--连接池维持的最大容量-->

**<** **property**  name ="Pool.MaximumIdleConnections" value ="5"**/>**<!--连接池允许挂起的最大连接-->

**<** **property**  name ="Pool.MaximumCheckoutTime" value ="120000" **/>** <!-- 连接被某个任务所允许

占用的最大时间 -->

**<** **property**  name ="TimeToWait" value ="500" **/>**   <!-- 线程等待的最大时间 -->

**</** **dataSource** **>**

**</** **transactionManager** **>**

**&&**

<!-- 针对单个别名定义 type:类型的路径; alias:别名 -->

<typeAliases> <typeAlias type="com.mybatis.entity.User" alias="user"/> </typeAliases>

UserMapper.xml引用别名:

<select id="findUserById" parameterType="int" resultType="user"> select \* from t\_user where id=#{id} </select>

**or**

<!-- 批量别名的定义:package:指定包名，mybatis会自动扫描包中的pojo类，自动定义别名，别名就是类名(首字母大写或小写都可以) -->

<typeAliases> <package name="com.mybatis.entity"/> <package name="其它包"/> </typeAliases>

**&&**

<!--sqlMap节点指定映射文件的位置，可以定义多个sqlMap元素,以指定项目内的所有映射文件-->

**<sqlMap**  resource ="com/wyq/map/userMap.xml" **/>**

**or**

<mappers> <!-- 加载映射文件 -->

<mapper resource="sqlmap/User.xml"/> <!--通过resource方法一次加载一个映射文件 -->

</mappers>

**or**

<!-- 通过mapper接口加载单个映射配置文件，遵循一定的规范：需要将mapper接口类名和mapper.xml映射文件名称保持一致，且在一个目录中-> <mapper class="com.mybatis.mapper.UserMapper"/>

**or**

<!-- 批量加载映射配置文件，mybatis自动扫描包下面的mapper接口进行加载，遵循一定的规范：需要将mapper接口类名和mapper.xml映射文件名称保持一致，且在一个目录中 -->

<package name="com.mybatis.mapper"/> **applicationContext-transaction.xml**

<!-- 数据源 -->

**<bean** id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>**

**<property** name="driverClassName" value="com.mysql.jdbc.Driver" **/>**

**<property** name="url" value="jdbc:mysql://localhost:3306/test" **/>**

**<property** name="username" value="root" **/>**

**<property** name="password" value="root" **/>**

**</bean>**

**&&**

<!--配置一个JdbcTemplate实例，并将这个“共享的”，“安全的”实例注入到不同的DAO类中去 -->

**<bean** id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>**

**<property** name="dataSource" ref="dataSource" **/>**

**</bean>**

**&&**

<!-- 声明事务管理器 -->

**<bean** id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>**

**<property** name="dataSource" ref="dataSource" **/>**

**</bean>**

**&&**

<!-- 需要实施事务增强的目标业务Bean -->

**<bean** id="libraryTarget" class="com.mucfc.dao.LibraryDaoImpl" p:jdbcTemplate-ref="jdbcTemplate" **/>**

**&&**

<!-- 使用事务代理工厂类为目标业务Bean提供事务增强 -->

**<bean** id="libraryFactory" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"

p:transactionManager-ref="txManager" p:target-ref="libraryTarget"**>**

**<property** name="transactionAttributes"**>**  <!-- 事务属性配置 -->

**<props>**

**<prop** key="get\*"**>**PROPAGATION\_REQUIRED,readOnly**</prop>**  <!--以get开头的方法采用只读型事务控制类型-->

**<prop** key="addBook"**>**PROPAGATION\_REQUIRED**</prop>** <!-- 所有方法均进行事务控制，如果当前没有事务，则新建一个事务 -->

**</props>**

**</property>**

**</bean>**

**or**

**<aop:config>**  <!-- 使用tx/aop来配置 -->

**<aop:pointcut** id="serviceMethod" expression="execution(\* com.mucfc.dao.\*(..))"**/>** <!-- 通过aop定义事务增强切面 -->

**<aop:advisor** pointcut-ref="serviceMethod" advice-ref="txAdvice" **/>**  <!-- 引用事务增强 -->

**</aop:config>**

**<tx:advice** id="txAdvice" transaction-manager="txManager"**>**  <!--事务增强 -->

**<tx:attributes>**  <!-- 事务属性定义 -->

**<tx:method** name="get\*" read-only="false" **/>**

**<tx:method** name="add\*" rollback-for="Exception" **/>**

**<tx:method** name="del\*" **/>**

**</tx:attributes>**

**</tx:advice>**

（1） tx:attribute 标签所配置的是作为事务的方法的命名类型

   如 <tx:method name="save\*" propagation="REQUIRED"/>

   其中\*为通配符，即代表以save为开头的所有方法，即表示符合此命名规则的方法作为一个事务

（2） aop:pointcut标签配置参与事务的类，由于是在Service中进行数据库业务操作，配的应该是包含那些作为事务的方法的Service类

   expression="execution(\* com.test.testAda.test.model.service.\*.\*(..))"

   其中第一个\*代表返回值，第二\*代表service下子包，第三个\*代表方法名，“（..）”代表方法参数

（3） aop:advisor 标签就是把上面我们所配置的事务管理两部分属性整合起来作为整个事务管理

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 是否需要 | 默认值 | 描述 |
| name | 是 |  | 与事务属性关联的方法名。通配符（\*）可以用来指定一批关联到相同的事务属性的方法。如：'get\*'、'handle\*'、'on\*Event'等等 |
| propagation | 不 | REQUIRED | 事务传播行为（配置文章开头） |
| isolation | 不 | DEFAULT | 事务隔离级别（配置看文章开头）propagation="REQUIRED"代表支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择 |
| timeout | 不 | -1 | 事务超时的时间（以秒为单位） |
| read-only | 不 | false | 事务是否只读 |
| rollback-for | 不 |  | 将被触发进行回滚的 Exception(s)；以逗号分开。 如：'com.foo.MyBusinessException,ServletException' |
| no-rollback-for | 不 |  | 不 被触发进行回滚的Exception(s)；以逗号分开。 如：'com.foo.MyBusinessException,ServletException' |

**&&**

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="mapperLocations" value="classpath:com/mybatis/mapper/\*.xml"/>

<property name="typeAliasesPackage" value="com.mybatis.model" />

</bean>

**&&**

<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">

<property name="mapperInterface" value="com.dao.UserMapper" />

<property name="sqlSessionFactory" ref="sqlSessionFactory" />

</bean>

<bean id="managerMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">

<property name="mapperInterface" value="com.dao.ManagerMapper" />

<property name="sqlSessionFactory" ref="sqlSessionFactory" />

</bean>

**or**

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="basePackage" value="com.dao"/> <!--com.dao是mapper类所在的包-->

  <property name="sqlSessionFactory" ref="sqlSessionFactory"/> <!--下面这个省略掉的话也不会出错，建议保留它-->

</bean>

## 过时方法

**Suspend()**、**resume()** 容易导致死锁 **异常** SecurityException -如果当前线程不能修改这个线程。

**destroy()**容易造成死锁 **异常** NoSuchMethodError -永远

**stop()**、[**stop**](mk:@MSITStore:C:\Users\rbk\Desktop\jdk1.8.CHM::/java/lang/Thread.html#stop-java.lang.Throwable-)**(**[**Throwable**](mk:@MSITStore:C:\Users\rbk\Desktop\jdk1.8.CHM::/java/lang/Throwable.html)**obj)** 本质上不安全的 **异常** SecurityException

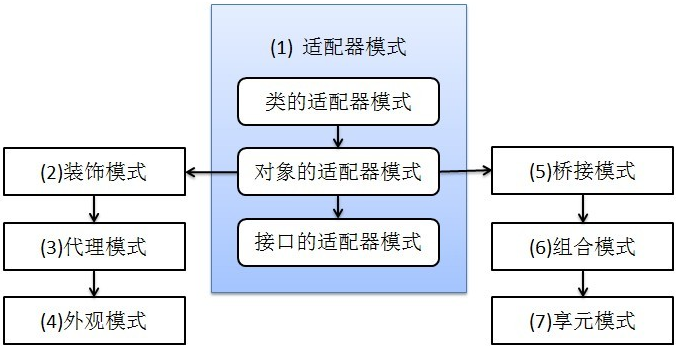
**Runtime.runFinalizersOnExit(boolean value)** 终结器，导致不稳定行为或死锁

## 二十三种设计模式

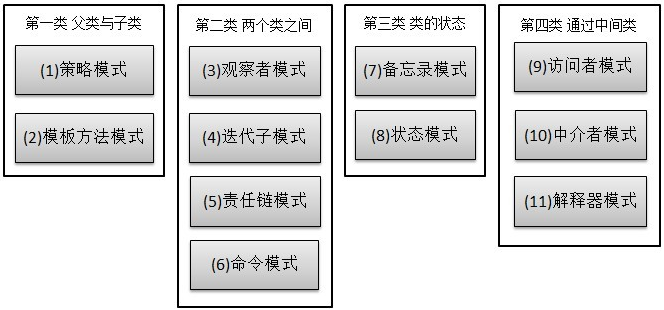
* + 1. 三大类

**5种创建型模式：**工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式

**7种结构型模式：**适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式



**11种行为型模式：**策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介模式、解释器模式



* + - 1. 六大原则

**开关闭原则** Close Open Principle：对外扩展开发，禁止内部修改

**里氏代换原则** Liskov Substitution Principle：子类具有父类所有的方法，父类返回值包含子类返回值属性

**依赖倒转原则** Dependence Inversion Principle：面向接口编程，依赖抽象而不依赖具体

**接口隔离原则** Inerface Segregation Principle：针对不同场景划分多个接口，达到降低依赖、耦合的目的

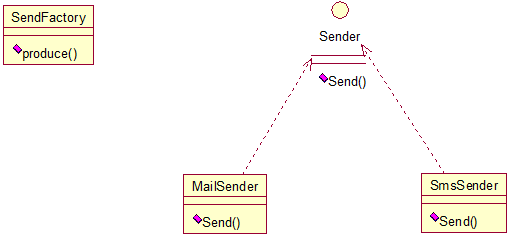
**迪米特原则** 实体类相互独立，持久对象

**合成复用原则** 合成/聚合方式，非继承

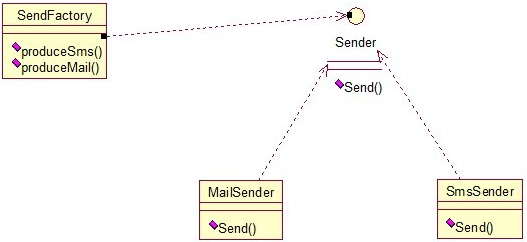
* + - 1. 详解

**（1） 工厂方法模式**

普通工厂模式：一个工厂方法实现对同一接口一些类实例的创建



多个工厂方法模式：多个工厂方法分别创建对象



静态工厂方法模式：将工厂方法设置为静态，不需要创建实例，直接调用即可

注：对实现了同一接口的类进行实例创建，依赖工厂类违背闭包原则

**（2） 抽象工厂模式：**创建多个工厂类（区别于工厂方法，一个工厂类多个方法对象）



**（3） 单例模式：**

① 私有化构造方法

② private static封装当前类静态实例对象

③ 静态方法返回当前类实例对象；

懒汉、饿汉、双重锁、枚举、静态内部类

线程安全，效率高，内存占用低（调用静态方法时创建对象），变量可见性，多线程操作原子性;

同步块保证多线程访问时，对象第一次创建后不再重复创建

Public class Singleton｛

Private Singleton（）｛｝

Private static volatile Singleton instance；

Public static Singleton getInstance（）｛

**if(**instance **==** **null){**

Sychronized**(**Singleton**.**class**){**

If**(**instance **==** **null)**

Instance **=** New Singleton**();**

**}**

**}**

**return** instance**;**

**}**

**}**

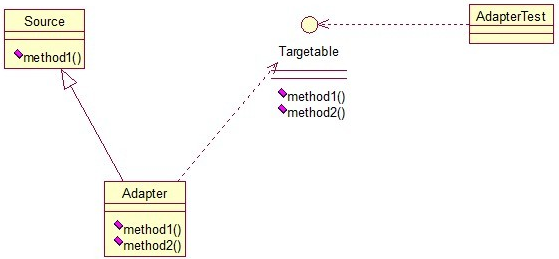
**（4） 建造者模式：**复杂对象的构建与表示分离

Product 复杂对象、builder 抽象接口、concreatebuilder 接口实现、director 组织复杂对象之间逻辑

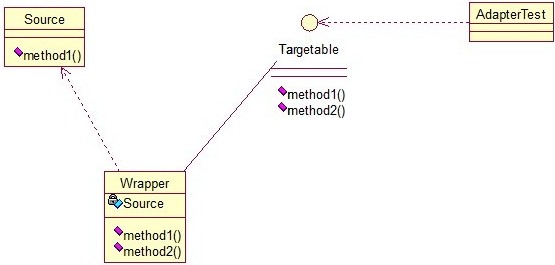
**（5） 原型模式：**以一个对象作为原型，实现对基本变量、引用类型的复制、克隆

**（6） 适配器模式：**以原始类、对象、接口为基础生成新的类、对象、接口

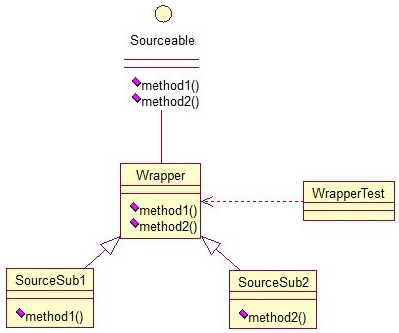
类适配器



对象适配器



接口适配器



**（7）装饰器模式：**

1、具有和真实对象相同的接口，保证客户端能以和真实对象交互方式和装饰对象交互

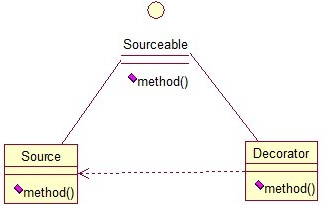
2、装饰对象包含一个真实对象的引用reference

3、装饰对象接收客户端的所有请求，并转发给真实对象

4、装饰对象可以在真实对象引用的前后增加附加功能，保证程序运行时不用修改真实

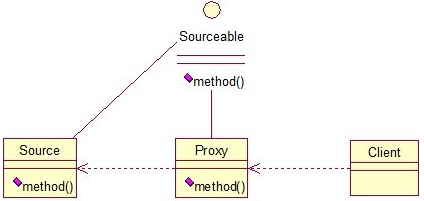
对象结构就可以对真实对象提供扩展功能；

注：面向对象程序设计中，继承功能是静态的，不能动态修改、删除



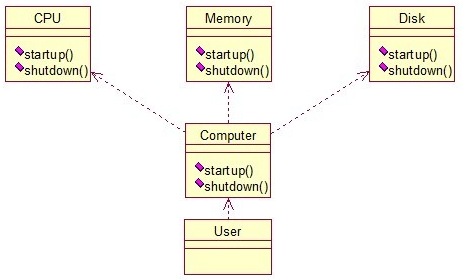
**（8） 代理模式：**多出一个代理类替原对象进行一些操作

注：区别装饰器模式，无需转发客户请求

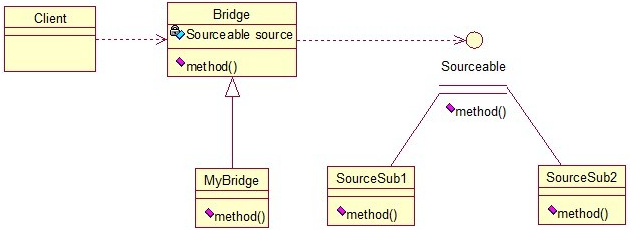


**（9） 外观模式：**spring是将类和类的关系配置到文件中，外观模式则是将类之间的关系放在facade类中，来降低类之间的耦合

注：该模式未涉及接口



**（10） 桥接模式：**将抽象化和实现化脱耦，是抽象类和实现类可以独立变化

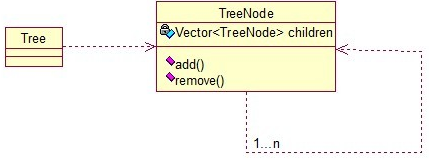


**（11） 组合模式：**将对象组合成树形结构，以表示“部分-整体”的层次结构。单个对象和组合对象的使用具有一致性

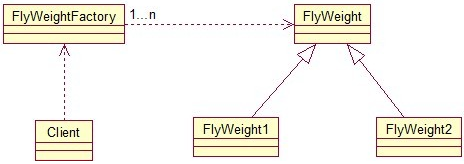
Component：对象声明接口，及所有类共有接口的默认行为。

Leaf：叶子节点对象，无子对象。

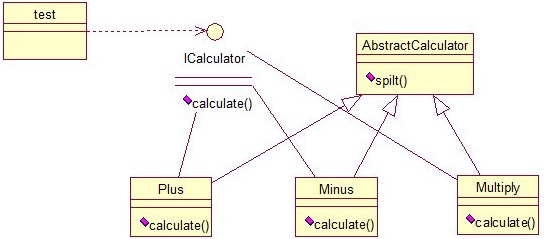
Composite：定义枝节点行为，用于存储子部件。component中实现与子节点的操作



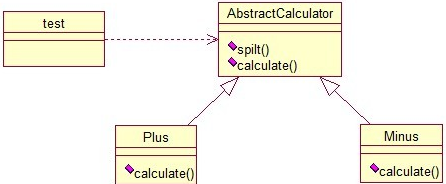
**（12） 享元模式：**实现对象的共享、共享池。对象多的时候可以减少内存的开销。通常和工厂模式一起使用



**（13） 策略模式：**定义一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。接口提供统一方法，多个实现类实现该方法

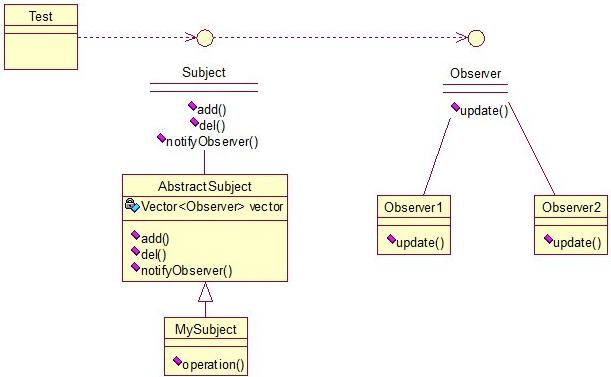


**（14） 模板方法模式：**通过调用抽象类实现对子类的调用



**（15） 观察者模式：**一个对象变化时，其他依赖对象都会收到通知，并随着变化。对象之间是一对多的关系

注：一个接口监控另一个接口的操作，不涉及继承

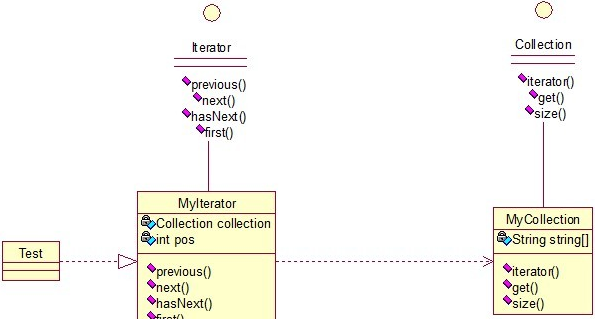


**（16） 迭代子模式：**顺序访问聚集中的对象

一、需要遍历的对象，即聚集对象

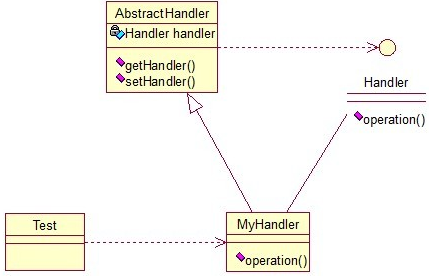
二、迭代对象，用于对聚集对象进行遍历访问

注：一个接口监控另一个接口的操作，不涉及继承



**（17） 责任链模式：**有多个对象，每个对象持有下一个对象的引用，这样形成一条链（树、环），请求在这条链上传递，知道某个对象开始处理这个请求。只允许由一个对象传给另一个对象

注：发出者不清楚最后由哪个对象处理请求，所以可以在隐藏客户端的情况下对系统进行调整一个接口监控另一个接口的操作，不涉及继承



**（18） 命令模式：**客户发出请求，系统将请求传递至相应模块处理并返回结果。客户只需等待结果不需关注具体的执行过程；实现了请求和执行的解耦。

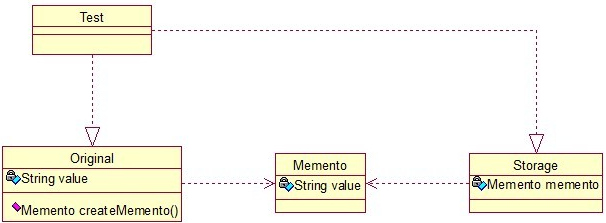
注：一个接口监控另一个接口的操作，不涉及继承



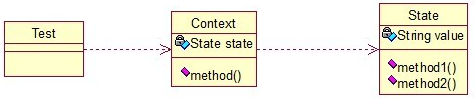
**（19） 备忘录模式：**保存一个对象的某个状态，在适当的时候恢复

A类对象属性、B类备忘录保存对象状态、C类储存备忘录类，只能存储不能修改

注：一个接口监控另一个接口的操作，不涉及继承



**（20） 状态模式**：修改状态的同时改变行为。1、改变状态获取不同的行为；2、外部可以看到状态的改变



**（21） 访问者模式：**数据据结构和作用于结构上的操作解耦，适用于结构稳定算法易于变化的系统

优点：增加操作容易；新操作意味着增加新的访问者，访问者模式将有关行为集中到一个访问者对象中，其 改变不影响系统数据结构

缺点：增加新的数据结构困难

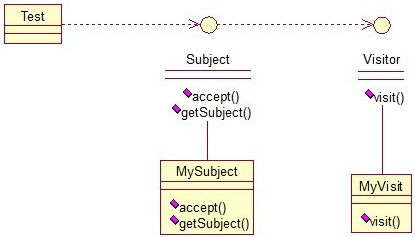
涉及角色：Visitor抽象访问者角色，为对象结构中具体元素提供一个访问操作接口。该接口名字和参数标识了发送给具体访问者的具体角色，这样访问者就可以通过的特定的接口直接访问；

ConcreateVisitor具体访问者角色接口实现类

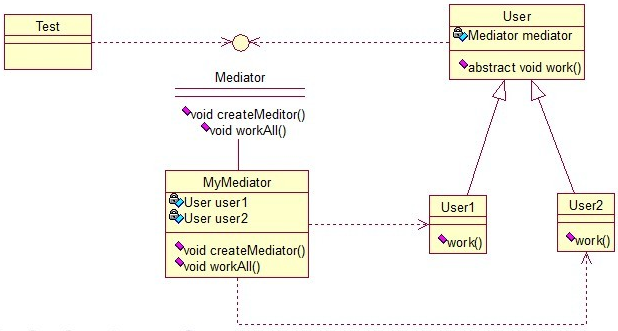
Element接受访问操作，以一个访问者作为参数

ConcreateElement具体元素，实现抽象元素所定义的接受操作接口

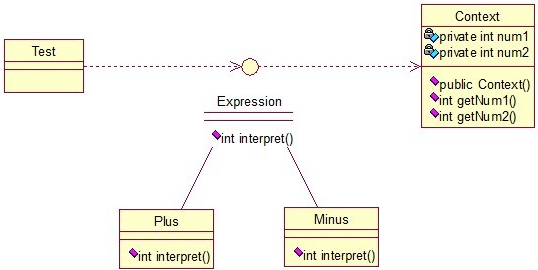
ObjectStructure结构对象角色,控制访问者访问相应角色



**（22） 中介者模式：**通过一个对象来一系列对象之间的交互，中介对象是各对象之间不需要显示的相互引用，从而实现耦合松散。而且可以独立改变他们之间的交互



**（23） 解释器模式：**给定一个语言，定义它语法规则的一种表示；定义一个解释器，这个解释器用来表解释语言中的句子



# WEB

## javascript

window.location.href 获取当前页面的url

indexOf("?") 设置参数字符串开始的位置

length 返回字符串长度，不是函数，不需要括号

indexOf(a)  返回字符串中一个子串a第一次出现的索引，如果没有匹配则返回-1

lastIndexOf(a) 返回字符串中一个子串a最后一次出现的索引，如果没有匹配则返回-1

match(reg) 检查一个字符串是否匹配一个正则表达式reg

substring(begin,end) 返回字符串一个子串，参数包含两头

substr(start,length)  返回从字符串中抽取从start下标开始的指定数目的字符

split(separator,howmany) separator为字符串或正则表达式（必填），howmany 指定返回的数组长度（可选）

replace(regexp/substr, replacement) 替换字符

toLowerCase() 将整个字符串转成小写字母

toUpperCase() 将整个字符串转成大写字母

**Array 对象**

Array 对象用于在变量中存储多个值:

var cars = ["Saab", "Volvo", "BMW"];

第一个数组元素的索引值为 0，第二个索引值为 1，以此类推

## jquery

jQuery.post( url, [data], [callback], [type] ) 使用POST方式来进行异步请求

**参数：** url (String) : 发送请求的URL地址

data (Map) : (可选)发送给服务器的数据，以Key/value的键值对形式表示

callback (Function) : (可选) 载入成功时回调函数

type (String) : Type of data to be sent，其实应该为客户端请求的类型

(JSON,XML,等等)

$**.**post**(** "/myurl"**,**

**{** uname**:** "change"**,** **password:** "me" **},**

**function** **(**data**,** textStatus**){** // data 可以是 xmlDoc, jsonObj, html, text, 等等.

// this;

// 这个Ajax请求的选项配置信息，请参考jQuery.get()说到的this

console.log(data);//object

console.log(JSON.stringify(data));//json

**},**

"json"

**);**

# Linux

## Sys

### Command

**rpm** 命令是RPM软件包的管理工具。原是Red Hat Linux发行版专门用来管理Linux各项套件的程序，由于它遵循GPL规则且功能强大方便，因而广受欢迎。逐渐受到其他发行版的采用。RPM套件管理方式的出现，让Linux易于安装，升级，间接提升了Linux的适用度。

语法rpm(选项)(参数，即软件包)

选项

-a 查询所有套件；

-b <完成阶段><套件档>+或-t <完成阶段><套件档>+：设置包装套件的完成阶段，并指定套件档的文件名称；

-c 只列出组态配置文件，本参数需配合"-l"参数使用；

-d 只列出文本文件，本参数需配合"-l"参数使用；

-e <套件档>或--erase<套件档> 删除指定的套件；

-f <文件>+ 查询拥有指定文件的套件；

-h或--hash 套件安装时列出标记；

-i 显示套件的相关信息； -i<套件档>或--install<套件档>：安装指定的套件档；

-l 显示套件的文件列表；

-p<套件档>+ 查询指定的RPM套件档；

-q 使用询问模式，当遇到任何问题时，rpm指令会先询问用户；

-R 显示套件的关联性信息；

-s 显示文件状态，本参数需配合"-l"参数使用；

-U<套件档> 或 --upgrade<套件档> 升级指定的套件档；

-v 显示指令执行过程；

-vv 详细显示指令执行过程，便于排错。

### Operate

#### 时间、时区

查看当前时区 date -R

修改设置Linux服务器时区

tzselect

timeconfig 仅限于RedHat Linux 和 CentOS

dpkg-reconfigure tzdata 适用于Debian

cp /usr/share/zoneinfo/$主时区/$次时区 /etc/localtime 复制相应的时区文件，

替换系统时区文件；或者创建链接文件

例如：在设置中国时区使用亚洲/上海（+8）

cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

查看本月月历 cal

查看时间和日期 date

设置时间和日期

将系统日期设定成2009年11月3日的命令

date -s 11/03/2009

将系统时间设定成下午5点55分55秒的命令

date -s 17:55:55

将当前时间和日期写入BIOS，避免重启后失效 hwclock -w

修改系统时区并立即生效

[root@localhost ~]# cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

cp: overwrite `/etc/localtime'? y

[root@localhost ~]# date

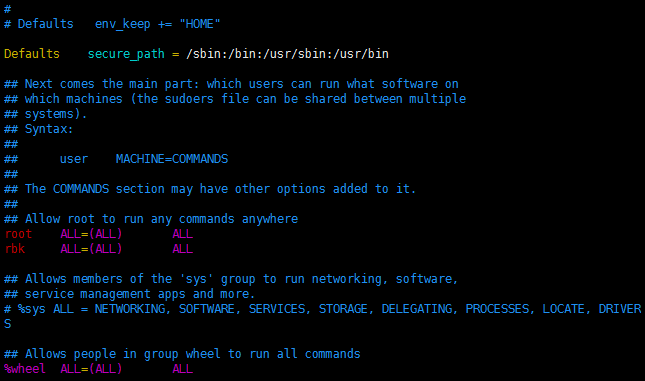
Sat Feb 20 16:04:43 CST 2010

[root@localhost ~]# hwclock

Sat 20 Feb 2010 04:05:12 PM CST -0.474966 seconds

#### 权限修改

当前用户变为管理员权限



**chmod** 权限管理命令change the permissions mode of a file的缩写。

u代表所有者，x代表执行权限。 + 表示增加权限。

chmod u+x \*.sh 就表示对当前目录下的.sh结尾文件的所有者增加可执行权限

chmod 777 文件夹地址 给文件夹最高权限，解决文件不能上传问题

注：r--4、w--2、x—1、rwxr-xr--  754、rw-r-x—x  651

chmod u+权限 文件或目录-------给所属者增加权限

chmod u+权限 文件或目录-------给所属者减少权限

  chmod g=权限 文件或目录-------给所属组赋予权限

  chmod o ------------------------同u，为加减权限

  用数字表示权限，直接赋值就可以  chmod 641 文件或目录

su - 用户名 切换用户

chown 所有者 文件或目录 change file ownership 改变所有者

chgrp 所属组 文件或目录 change file group ownership 改变所属组

umask 查看创建文件的默认权限，改变默认权限 umask 权限掩码值

注：特殊权限位 022-用户权限位，权限掩码值777、022、755、022 表示755

  SetUID 4、SetGID 2

Linux权限规则：空文件没有x权限

**添加用户信息** useradd 用户名 添加用户名

passwd 密码 设置密码

who 显示当前登录用户，登录时间和登录IP地址

w 按照标题显示：当前时间、机器运行时间、登录用户数量、系统负载信息

#### 系统操作

cat /proc/version or uname -a or uname -r 当前centos 版本与redhat对应的版本的命令

**系统版本**

lsb\_release -a 列出所有版本信息，适用于所有的linux，包括RedHat、SUSE、Debian等发行版

　　cat /etc/redhat-release or cat /etc/issue CentOS

**查看系统是64位还是32位**

　　getconf LONG\_BIT or getconf WORD\_BITorfile /bin/ls or lsb\_release -a or vim /ect/issue

rm -f /var/run/yum.pid 强制关闭yum进程

**防火墙开启端口访问权限**

vi /etc/sysconfig/iptables增加

-A INPUT -p tcp -m state --state NEW -m tcp --dport 端口号 -j ACCEPT

**查看网络管理器服务的状态** systemctl status NetworkManager.service

**检查受网络管理器管理的网络接口**

nmcli dev status or nmcli connection show 注：当前连接

**重启网络服务** service network restart or systemctl restart network.service

**查看IP接口** ip add or ifconfig

**静态IP配置（虚拟机需开启桥接模式）**

BOOTPROTO="static"  #dhcp改为static

ONBOOT="yes"  #开机启用本配置

IPADDR=192.168.7.106  #静态IP

GATEWAY=192.168.7.1  #默认网关

NETMASK=255.255.255.0  #子网掩码

DNS1=192.168.7.1  #DNS 配置，多个DNS网址的话再增加（增加）

BROADCAST=192.168.20.255广播地址，可以忽略，系统会根据IP地址和网关或者网络地址自动设定

NM\_CONTROLLED=yes 决定了你的eth0是否可以由NNetwork Manager托管

nm-connection-editor 调出网络设置界面窗口

**ps aux** 是用BSD的格式来显示 java这个进程

显示的项目有：USER , PID , %CPU , %MEM , VSZ , RSS , TTY , STAT , START , TIME , COMMAND

**ps -ef** 是用标准的格式显示java这个进程

显示的项目有：UID , PID , PPID , C , STIME , TTY , TIME , CMD

#### 文件、文件夹操作

**移动文件**  mv 文件名 移动目的地文件名

**重命名文件**  mv 文件名 修改后的文件名

**删除文件** rm 文件名

**删除文件夹** rmdir 文件夹名

注：rmdir不能删除非空的文件夹，rm -rf 非空文件夹名

-r 就是向下递归，不管有多少级目录，一并删除。

-f 就是直接强行删除，不作任何提示的意思。

**删除文件命令** rm -f 文件名 注：此命令是不会放到回收站的

**查看文件、文件夹** ls list 目录文件

ls -ld 文件目录及属性

ls -a all显示所有文件，包括隐藏文件

ls -l long详细信息显示

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| drwxr-xr-x | 2 | root | root | 4096 | Feb 14 11:20 |
| 每三个表示权限 | 表示硬连接数 | 所有者 | 所属组 | 文件大小 | 创建时间 |

d  目录directory

pwd print working directory 查看当前目录

touch 创建空文件夹

mkdir make directories 创建空目录

cp copy 复制文件 注：-R 复制目录

mv move 移动文件，改名

rm remove 删除

rm -f 不提示，强制删除，写脚本时用，因为yes没有交互 r删除、f强制

more 分页查看，空格或f 显示下一页 enter 显示下一行 q或Q 退出

head -num 查看文件前num行

tail -num 查看文件尾num行 -f 动态查看文件尾部信息

ln 文件名 目的目录 link 创建硬连接

注：类似于copy，文件大小相同，不同于copy的是，它是同步更新的还有一个不同的是，拷贝的时间是不同，时间是创建的时间，一个i节点可以映射到多个文件，所以硬链接可以同步更新

ln -s 文件名 目的目录 -s soft软连接

注：软连接文件类型是l，所有人都有权限，但能不能访问取决于源文件软连接时间值 是创建软连接的时间，相当于快捷方式

which 命令 查询命令所在的绝对路径

注：命令也是文件，whereis除了绝对路径，还包括帮助文档

find [搜索路径] -name [搜索关键字] 目录

\*匹配任意字符 init\*、

?匹配单个字符 init???

find [搜索路径] -user [搜索关键字] 用户

find [搜索路径] -name [搜索关键字] -a -type f 查找二进制文件

find [搜索路径] -name test -exec ls -l {} \; 查找test，并显示详细信息

find [搜索路径] -name a.soft -exec rm {} \; 查找a.soft文件并删除

locate  [搜索关键字] list files in databases 查找所有跟file相关的文件，速度快

locate指令是根据数据库查找的，如果数据库中文件没有更新，那么就会找不到

  注：定期updatedb，则可以直接locate找到

updatedb update the slocate database 建立整个系统目录文件的数据库

grep [指定字串][源文件] 文件中搜寻字串匹配的行，并输出

#### cat命令的使用

**cat :** concatenate files and print on the standard output

**查看文件内容**

cat f1.txt 查看 f1.txt 文件的内容

cat -n f1.txt 查看 f1.txt 文件的内容，由 1 开始对所有输出行进行编号

cat -b f1.txt 查看 f1.txt 文件的内容，用法与-n相似，对空白行不编号

cat -s f1.txt 连续两行或两行以上的空白行，代换为一行的空白行

cat -e f1.txt 每一行后面加一个$符号。

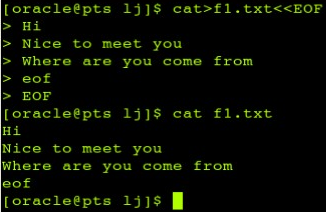
cat f1.txt f2.txt 同时显示f1.txt和f2.txt文件内容

cat -n f1.txt>f2.txt f1.txt文件中每一行加上行号后覆盖到f2.txt中，文件不存

在则创建

cat -n f1.txt>>f2.txt f1.txt文件中每一行加上行号后追加到f2.txt中，文件不

存在则创建它

**创建文件以及写入文件内容**

注：创建文件的时候要设置文件结束标志，也就是<<EOF，可以把EOF换成别的字符，注意是大小写敏感的，当文件内容写完之后要输入结束标志EOF，这时命令会正确结束，表示成功创建文件并且写进内容

**追加文件内容** 与创建文件内容不同的是符号单边号>变成了双边号>>

**文件合并**

注：把文件f2.txt,f3.txt,f4.txt的文件内容写入到f1.txt中，如果f1.txt文件以前有内容，则先会清除它们然后再写入合并后的内容；如果不想清除文件内容，则可以把单边号>变成了双边号>>

#### 系统中文支持

**CentOs7**

查看更改后的系统语言变量 locale

查看语言包 locale -a

安装中文语言包 yum install kde-l10n-Chinese yum reinstall glibc-common

#### 帮助命令

man [命令或配置文件] manual，man 5查看配置文件

注：man ls  使用more查看，空格翻页，enter下一行

infoinfo [任何关键字] information 获取帮助信息

whatis apropos[任何关键字] 获得索引的简短说明信息

如：whatis whereis

help 内置命令 如：ls --help

#### 压缩解压命令

windows可解压Linux所有，但相反不可共同：zip 但压缩比不高，如果小文件建议使用

**gzip** gzip 选项[文件] or gzip –d 压缩文件

压缩后文件格式: .gz

只能压缩文件，不能压缩目录，不保留源文件

gunzip 压缩文件

tar –zcf 打包后的名字 要打包的文件名

**tar 选项[cvf][目录]**

-c 产生.tar打包文件

-v 显示详细信息

  -f 指定压缩后的文件名

  -z 打包同时压缩

注：压缩后文件格式： .tar.gz

tar -zxf 包名 **解压缩**

  -x 解包.tar文件

  -v 显示详细信息

  -f 指定解压文件

  -z 解压缩

zip 选项[-r][压缩后文件名称][文件或目录] 保留源文件，-r  压缩目录

unzip [压缩文件] 解压.zip的压缩文件

bzip2 选项[-k][文件] -k 产生压缩文件后保留源文件，不加k则和gzip相同，但压缩比很高

#### 网络通信指令

write 用户名 当用户登陆的时候，可以进行信息交互，以Ctrl+D作为结束

wall 登陆的所有人都会收到

ping 网络地址 和windows不同的是，Linux如果不ctrl+c终止就会一直进行下去

ping -c 次数 网络地址 像windows一样定义ping的次数

ping -s 数据包大小 网络地址 默认64字节，改变数据包大小

注：丢包率：packet loss

如果ping不通，先ping 127.0.0.1 回环地址，检查自己的设置。

不一定是网络不通，可能防火墙等屏蔽了

ifconfig 选项[-a][网卡设备标识] 查看网络设置信息

-a 显示所有网卡信息

例：ifconfig –a

eth0 本机物理网卡

lo 回环网卡地址

ifconfig eth0 IP地址 修改ip，仅本次生效,只有写入配置文件才会永久生效

shutdown 只有root可以执行， shutdown -h now 最好不要使用，立马关机

reboot 重启

#### Shell应用技巧

**命令补全** Tab 如果是唯一的则直接补全，如果不唯一则按两下之后出现候选命令

**清屏** clear or Ctrl+l

**清除光标前字符** Ctrl+u

**查看以前的命令** history

!123 执行第123条命令

注：方向键上下可以找到历史命令

**定义别名** alias系统定义的别名

如：alias copy=cp alias xrm=”rm –r”组合要括起来

**清除别名** unalias copy

**输入、输出重定向** 标准I/O一样，Shell对于每一个进程预先定义3个文件描述字

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| (STDIN)标准输入 | (STDOUT)标准输出 | (STDERR)标准错误输出 |

|  |  |  |
| --- | --- | --- |
| >或>> | < | 2> |
| 输出重定向 | 输入重定向 | 错误除数重定向 |

**写一个脚本，然后自动输出**

[root@localhost]# wall < /etc/issue

Broadcast message from root (Thu Feb 26 17:08:07 2015):

CentOS release 5.5 (Final)

**错误除数重定向**

[root@localhost]# cp -R /usr /backup/usr.bak 2> /bak.error

[root@localhost]# ls /kdls

ls: /kdls: No such file or directory

[root@localhost]# ls /kdls 2> ls.err   将错误信息写入，这个在写自动化脚本时非常有用

[root@localhost]# cat ls.err

ls: /kdls: No such file or directory

[root@localhost test]# ls /tmp 2> ls.err  如果不错则没有

**管道** 将一个命令的输出传送给另一个命令，作为另一个命令的输入

ls -l /etc | more 将ls -l /etc的输出传送给more

ls -l /etc | grep init 包含init的行

wc -l /etc/services wc 可以查看行数 –l表示line行

ls -l /etc | grep init | wc –l 多个管道 ：查看etc下包含init多少行

**命令连接符** ; ; 间隔的各命令按顺序依次执行

&& 前后命令的执行存在逻辑与关系，&&前面的命令执行成功后，后面的命

令才被执行

|| 前后命令的执行存在逻辑或关系，||前面的命令执行失败后，它后面的命

令才被执行

**命令替换符** 将一个命令的输出作为另一个命令的参数，键盘1左边的符号`

[root@localhost test]# which touch  查看touch所在

/bin/touch

[root@localhost test]# ls -l /bin/touch 查看详情

-rwxr-xr-x 1 root root 42284 Mar  1  2010 /bin/touch

[root@localhost test]# ls -l `which touch` 可以直接做参数

-rwxr-xr-x 1 root root 42284 Mar  1  2010 /bin/touch

## Vim

启动vim后，vim处于Normal 模式 , ESC从编辑模式返回Normal 模式

### 文本编辑

**复制粘贴**

v 可视化选择 y 复制 p 粘贴 d 剪切

注：复制和剪切的命令后面都可以接一个移动命令，表示将光标移动区间内的文本复制过来

i Insert 模式，按 ESC 回到 Normal 模式.

x    删当前光标所在的一个字符q

dd   删除当前行，并把删除的行存到剪贴板里

p    粘贴剪贴板 注：p 与 P 的区别是贴到光标前后

:w  保存文件但不退出

:w file 将修改另外保存到file中，不退出

:w!  强制保存，不推出

:wq!  强制保存文件，并退出

q:  不保存文件，退出

:q!  不保存文件，强制退出

:qa!  强行退出所有的正在编辑的文件

:x 或 ZZ 或 :wq  保存并退出 (:x 表示仅在需要时保存，ZZ不需要输

入冒号并回车)

:e!  放弃所有修改，从上次保存文件开始再编辑

gU 变大写

gu  变小写

yy  拷贝当前行当行于 ddP

u undo，撤销

ctrl+r redo，反撤销

**各种插入模式** a  → 在光标后插入

o  → 在当前行后插入一个新行

O  → 在当前行前插入一个新行

cw → 替换从光标所在位置后到一个单词结尾的字符

**简单的移动光标**

0  → 数字零，到行头

^  → 到本行第一个不是blank字符的位置（所谓blank字符就是空

格，tab，换行，回车等）

$  → 到本行行尾

g\_  → 到本行最后一个不是blank字符的位置。

/pattern  → 搜索 pattern 的字符串（n到下一个，N到上一

个）

**打开/保存/改变文件(Buffer)**

:e <path/to/file>  打开一个文件

:saveas <path/to/file> 另存为 <path/to/file>

:bn 和 :bp  同时打开很多文件时，使用这两个命令来切换下一个或上

一个文件

.  (小数点) 可以重复上一次的命令

N<command> 重复某个命令N次 如： 2dd → 删除2行、 3p → 粘贴文本3次

### 多行注释、取消注释

1、CTRL+V进入可视化模式（VISUAL BLOCK）

2、移动光标上移或者下移，选中多行的开头

3、选择完毕后，按大写的的I键，此时下方会提示进入“insert”模式，输入你要插入的注释

4、ESC键，你就会发现多行代码已经被注释了

注：删除多行注释，Ctrl+v 进入列选择模式，移到光标把注释符选中，按下d，注释就被删除

### 文本查看

grep -v "^#" /etc/inittab | more   可以把以#开头的行排除

hjkl (强例推荐使用其移动光标，但不必需) →你也可以使用光标键 (←↓↑→)

注：j 就像下箭头

NG  到第 N 行 （陈皓注：注意命令中的G是大写的，另我一般使用 : N 到第N行

Gg 到第一行。（陈皓注：相当于1G，或 :1）

G 到最后一行

w  到下一个单词的开头，W 一个以空格为分隔的单词

s  一个句字

p  一个段落

e  到下一个单词的结尾

%    匹配括号移动，包括 (, {, [ 注：需要把光标先移到括号上

\* 和 #  匹配光标当前所在的单词，移动光标到下一个（或上一个）匹配单词（\*是下一个，#是上一个）

0  到行头

^   到本行的第一个非blank字符

$  到行尾

g\_   到本行最后一个不是blank字符的位置。

Fa 到下一个为a的字符处，你也可以fs到下一个为s的字符。

t,  到逗号前的第一个字符。逗号可以变成其它字符。

3fa 在当前行查找第三个出现的a

dt" 删除所有的内容，直到遇到双引号—— "

注：F 和 T → 和 f 和 t 一样，只不过是相反方向

:split 创建分屏 (:vsplit创建垂直分屏)

## Nginx

## Redis

### 安装

**下载** wget http://download.redis.io/releases/redis-3.2.8.tar.gz

**解压** tar xzf redis-3.2.8.tar.gz

cd redis-3.2.8

**编译** make or 使用root用户执行 make install

**整理可执行文件** Src目录下，四个可执行文件redis-server、redis-benchmark、redis-cli、redis.conf

**直接启动Redis服务** redis-server

**or**

**通过指定配置文件启动** Redis根目录下redis.conf

daemonize yes 默认以后台程序方式运行（省略手动使用&号强制后台运行） port 6379 修改默认监听端口

logfile "/home/futeng/logs/redis.log" 修改生成默认日志文件位置 dir /home/futeng/data/redisData 配置持久化文件存放位置

redis-server ./redis.conf

**or**

**使用Redis启动脚本设置开机自启动** Redis的 /utils/ 目录下

redis\_init\_script

注：redis启动脚本，习惯性用监听的端口名作为配置文件等命名 redis服务器

监听的端口 REDISPORT=6379

服务端所处位置，在make install后默认存放于/usr/local/bin/redis-server

注：如果未make install则需要修改该路径 EXEC=/usr/local/bin/redis-

server

客户端位置 CLIEXEC=/usr/local/bin/redis-cli

Redis的PID文件位置 PIDFILE=/var/run/redis\_${REDISPORT}.pid

配置文件位置，需要修改 CONF="/etc/redis/${REDISPORT}.conf"

1、将修改好的配置文件以端口为名复制一份到指定目录，需使用root用户

mkdir /etc/redis

cp redis.conf /etc/redis/6379.conf

2、将启动脚本复制到/etc/init.d目录下（通常都以d结尾表示是后台自启动服

务）

cp redis\_init\_script /etc/init.d/redisd

错误： service redisd does not support chkconfig

在启动脚本开头添加如下两行注释以修改其运行级别：

#!/bin/sh

# chkconfig: 2345 90 10

# description: Redis is a persistent key-value database

设置为开机自启动服务器 chkconfig redisd on

打开服务 service redisd start

关闭服务 service redisd stop

**客户端测试**

检测后台进程是否存在 ps -ef |grep redis

检测6379端口是否在监听 netstat -lntp | grep 6379

使用`redis-cli`客户端检测连接是否正常 ./redis-cli

如果更改了端口，使用`redis-cli`客户端连接时，也需要指定端口，

如： ./redis-cli -p 6380

127.0.0.1:6379> keys \*

(empty list or set)

127.0.0.1:6379> set key "hello world"

OK

127.0.0.1:6379> get key

"hello world"

**停止** redis-cli shutdownorkill -9 PID or pkill redis-server

### 自带工具

redis/src目录 redis-benchmark redis性能测试工具

　　 redis-check-aof 检查aof日志的工具

　　redis-check-dump 检查rdb日志的工具

　　redis-cli 连接用的客户端

　　redis-server redis服务进程

### Redis的配置

**redis目录** **redis.conf**

daemonize yes 为在后台运行

pdifile pid文件存放地址

bind 只接收来自该IP的请求，不设置将处理所有请求，生产环境需设置

port 监听端口，默认为6379

timeout 设置客户端连接时的超时时间，单位为秒

loglevel 等级分4级 debug、revbose、notice、warning 生产环境一般开启notice

logfile 配置log文件地址，默认使用标准输出，即打印在命令行终端的端口上

database 数据库的个数，默认使用的数据库是0

save redis进行数据库镜像的频率

rdbcompression 在进行镜像备份时，是否进行压缩

dbfilename 镜像备份文件的文件名

dir 数据库镜像备份的文件放置的路径

slaveof 该数据库为其他数据库的从数据库

masterauth 主数据库连接需要密码验证时，在这里设定

requirepass 客户端连接后进行任何其他指定前需要使用的密码

maxclients 限制同时连接的客户端数量

maxmemory redis能够使用的最大内存

appendonly 开启appendonly模式后，redis会把每一次所接收到的写操作都追加到

**appendonly.aof**文件中，当redis重新启动时，会从该文件恢复出之前的状态

appendfsync appendonly.aof文件进行同步的频率

vm\_enabled 是否开启虚拟内存支持

vm\_swap\_fil 虚拟内存的交换文件的路径

vm\_max\_momery 开启虚拟内存后，redis将使用的最大物理内存的大小，默认为0

vm\_page\_size 虚拟内存页的大小

vm\_pages 交换文件的总的page数量

vm\_max\_thrrads vm IO同时使用的线程数量

**redis数据结构** redis 五种数据类型 string,list,set、zset(sorted set)、Hash

**string** 与**Memcached**的操作类似，但它的功能更丰富。

**list** 链表结构，主要功能是push、pop、获取一个范围的所有值等等。操作中key理解为链表的名字

**set** 对集合的操作有添加删除元素，有对多个集合求交并差等操作。操作中key理解为集合的名字。

**zset** set升级版本，在set的基础上增加了一个顺序属性，这一属性在添加修改元素的时候可以指定，每次指定后，zset会自动重新按新的值调整顺序。可以理解了有两列的mysql表，一列存value，一列存顺序

**Hash** 数据类型允许用户用Redis存储对象类型,Hash数据类型的一个重要优点是,当存储的数据对象只有很少几个key值时,数据存储的内存消耗会很小

**redis数据存储** 分为内存存储、磁盘存储、log文件三部分，配置文件中有三个参数对其进行配置。

save seconds updates save配置，指出在多长时间内，有多少次更新操作，将数据同步到数据文件

appendonly yes/no  appendonly配置，指出是否在每次更新操作后进行日志记录，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为redis本身同步数据文件是按上面的save条件来同步的，所以有的数据会在一段时间内只存在于内存中。

appendfsync no/always/everysec ，appendfsync配置，no表示等操作系统进行数据缓存同步到磁盘，always表示每次更新操作后手动调用fsync()将数据写到磁盘，everysec表示每秒同步一次

**redis主从配置：**redis支持master-slave的主从配置，配置方法是在从机的配置文件中指定slaveof参数为主机的ip和port即可

## Svn

**启动svn服务** svnserve -d -r $svn目录

## MySQL

### RDBMS

**术语**

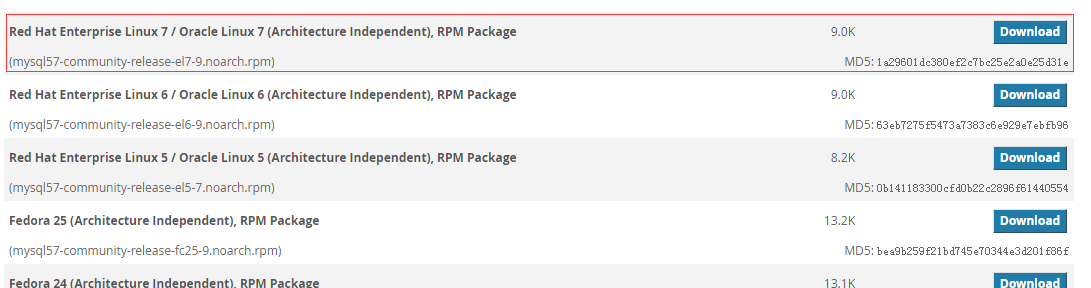
* 数据库: 数据库是一些关联表的集合。
* 数据表: 表是数据的矩阵。在一个数据库中的表看起来像一个简单的电子表格。列: 一列(数据元素) 包含了相同的数据, 例如邮政编码的数据。
* 行：一行（=元组，或记录）是一组相关的数据，例如一条用户订阅的数据。
* 冗余：存储两倍数据，冗余降低了性能，但提高了数据的安全性。
* 主键：主键是唯一的。一个数据表中只能包含一个主键。你可以使用主键来查询数据。
* 外键：外键用于关联两个表。
* 复合键：复合键（组合键）将多个列作为一个索引键，一般用于复合索引。
* 索引：使用索引可快速访问数据库表中的特定信息。索引是对数据库表中一列或多列的值进行排序的一种结构。类似于书籍的目录。
* 参照完整性: 参照的完整性要求关系中不允许引用不存在的实体。与实体完整性是关系模型必须满足的完整性约束条件，目的是保证数据的一致性。

### 安装、配置

#### 配置YUM源

**MySQL官网YUM源**rpm安装包：http://dev.mysql.com/downloads/repo/yum/

注：左侧 MySQL Yum Repository



**下载mysql源安装包**

wget https://dev.mysql.com/get/mysql57-community-release-el7-9.noarch.rpm

**安装mysql源**

yum localinstall mysql57-community-release-el7-9.noarch.rpm

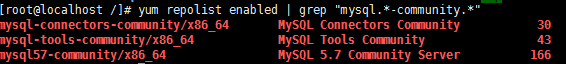
注：卸载 rpm -e 包名

**检查mysql源是否安装成功** rpm -qa|grep mysql

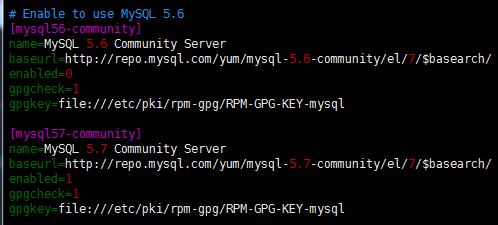
D://Downloads/renbaokun_mail@163.com/f8f34861ce9d46efae9f4e934510bc59/clipboard.png

**or**

yum repolist enabled | grep "mysql.\*-community.\*"



**修改**vim /etc/yum.repos.d/mysql-community.repo源，改变默认安装的mysql版本。enabled=0关闭enabled=1启用



#### 安装、启动、状态

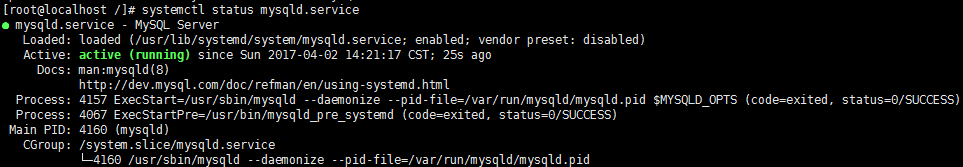
**安装** yum install mysql-community-server

**设置为开机启动** systemctl enable mysqld

systemctl daemon-reload

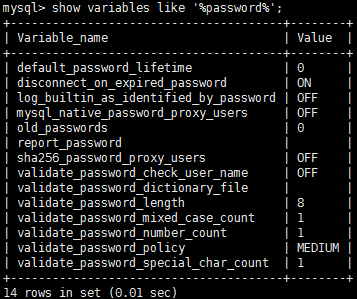
**启动服务** systemctl start mysqld

**查看启动状态** systemctl status mysqld



#### 修改root本地登录密码

mysql5.7默认安装了密码安全检查插件（validate\_password），默认密码检查策略要求密码必须包含：大小写字母、数字和特殊符号，并且长度不能少于8位。否则会提示ERROR 1819 (HY000): Your password does not satisfy the current policy requirements错误

show variables like '%password%'; 通过msyql环境变量可以查看密码策略的相关信息：

validate\_password\_policy：密码策略，默认为MEDIUM策略

validate\_password\_dictionary\_file：密码策略文件，策略为STRONG才需要

validate\_password\_length：密码最少长度

validate\_password\_mixed\_case\_count：大小写字符长度，至少1个

validate\_password\_number\_count ：数字至少1个

validate\_password\_special\_char\_count：特殊字符至少1个

*上述参数是默认策略MEDIUM的密码检查规则。*

|  |  |
| --- | --- |
| **策略** | **检查规则** |
| 0 or LOW | Length |
| 1 or MEDIUM | Length; numeric, lowercase/uppercase, and special characters |
| 2 or STRONG | Length; numeric, lowercase/uppercase, and special characters; dictionary file |

**修改密码策略**

/etc/my.cnf文件添加validate\_password\_policy配置，指定密码策略,选择0（LOW），1（MEDIUM），2（STRONG）其中一种，选择2需要提供密码字典文件

validate\_password\_policy=0

validate\_password = off 不需要密码策略添加

systemctl restart mysqld 重新启动mysql服务使配置生效：

mysql安装完成之后，在/var/log/mysqld.log文件中给root生成了一个默认密码。通过下面的方式找到root默认密码，然后登录mysql进行修改：

grep 'temporary password' /var/log/mysqld.log

D://Downloads/renbaokun_mail@163.com/b9ec0e1b0fde4802ad1e87dfd204a8c5/clipboard.png

mysql -uroot -p

ALTER USER 'root'@'localhost' IDENTIFIED BY 'MyNewPass4!';

或者

set password for 'root'@'localhost'=password('MyNewPass4!');

#### 添加远程登录用户

默认只允许root帐户在本地登录，如果要在其它机器上连接mysql，必须修改root允许远程连接，或者添加一个允许远程连接的帐户，为了安全起见，我添加一个新的帐户：

GRANT ALL PRIVILEGES ON \*.\* TO 'yangxin'@'%' IDENTIFIED BY 'Yangxin0917!' WITH GRANT OPTION;

#### 配置默认编码

**默认配置文件路径：**

配置文件： /etc/my.cnf

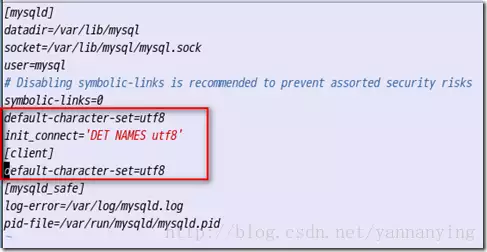
日志文件： /var/log//var/log/mysqld.log

服务启动脚本： /usr/lib/systemd/system/mysqld.service

socket文件： /var/run/mysqld/mysqld.pid

通过rpm安装的，center os 系统在/etc/目录下、Ubuntu系统系统在/etc/mysql/目录下

数据库的配置文件my.cnf文件



server修改默认编码

default-character-set=utf8

init\_connect='SET NAMES utf8'

client的默认编码

[client]

default-character-set=utf8

**重启mysql服务**

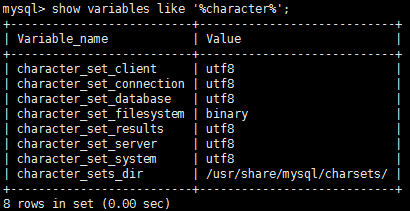
service mysqld restart

**登录mysql**

mysql –uroot –p123456（123456为密码，或者可以去掉，隐藏密码输入）

**查看默认编码**

show variables like '%character%';



### 数据操作

**查看当前数据库信息** status;

**执行sql脚本**  use yourdatabasename;

source sql文件;

**显示数据库**  show databases;

**选择数据库** use 数据库名;

**显示数据库中的表** show tables;

**显示数据表的结构**  describe 表名;

**显示数据表的属性、属性类型、主键等** SHOW COLUMNS FROM 表名

**显示数据表的详细索引信息，包括PRIMARY KEY（主键）** SHOW INDEX FROM 表名

**显示数据库 RUNOOB 中所有表的信息** SHOW TABLE STATUS FROM RUNOOB;

**表名以runoob开头的表的信息** SHOW TABLE STATUS from RUNOOB LIKE 'runoob%';

**加上 \G，查询结果按列打印** SHOW TABLE STATUS from RUNOOB LIKE 'runoob%'\G;

**建库** create databse 库名; set names utf8;

**建表** create table 表名 (字段设定列表)；

mysql> create table name(

-> id int auto\_increment not null primary key ,

-> uname char(8),

-> gender char(2),

-> birthday date );

注： auto\_increment 自增 、primary key 主键

**删除表** drop table 表名

**删除库** drop database 库名;

**备份数据库**  mysqldump -u root -p --opt 数据库名>备份名; //进入到库目录

**恢复** mysql -u root -p 数据库名<备份名; //恢复时数据库必须存在

**数据库授权** grant select on 数据库.\* to 用户名@登录主机 identified by "密码"

例：用户user1密码为123456，让他可以在任何主机上登录，并对所有数据库有查询、插入、修改、删除的权限

grant select,insert,update,delete on \*.\* to user1@"%" Identified by "123456";

  用户user2密码为123456,让此用户只可以在localhost上登录,也可以设置指定IP，并可以对数据库test进行查询、插入、修改、删除的操作 (localhost指本地主机，即MySQL数据库所在的那台主机）

grant select,insert,update,delete on test.\* to user2@localhost

identified by "123456";

注： 其次也可以采用修改表的方式，处理用户的登录方式：

数据库： Mysql

表:   User

修改：  User表中的Host列的值来现实登录入口

**导出mysql数据库**

导出数据和表结构 mysqldump -u用户名 -p密码 数据库名 > 数据库名.sql

只导出表结构 mysqldump -u用户名 -p密码 -d 数据库名 > 数据库名.sql

注：导出至 mysql的data目录

**导出mysql数据库**

方法一：

（1）选择数据库 mysql>use abc;

（2）设置数据库编码 mysql>set names utf8;

（3）导入数据（注意sql文件的路径） mysql>source /home/abc/abc.sql;

方法二： mysql -u用户名 -p密码 数据库名 < 数据库名.sql

建议使用第二种方法导入。

注：有命令行模式，有sql命令

### sql

**小数格式化**

SELECT CONVERT(4545.1366,DECIMAL(10,2));

10是精度，表示保存值的位数。2是标度，表示小数点保留位数四舍五入

SELECT FORMAT(12562.6655,2);

保留2位整数，部分超过三位的时候以逗号分割，并且返回的结果是string类型

SELECT TRUNCATE(4545.1366,2);

小数点保留位数，不四舍五入

**分页**

/\*创建测试表\*/

DROP TABLE IF EXISTS pagetest;

create table pagetest(

id int NOT NULL PRIMARY KEY AUTO\_INCREMENT ,

col01 int NULL,

col02 nvarchar(50) NULL,

col03 datetime NULL

)

**select** **\*** **from** **table** **limit** **(start**-1**)\*limit,limit;** start 是页码，limit 是每页显示的条数

### 异常错误

[Err] 1055 - Expression #1 of ORDER BY clause is not in GROUP BY clause and contains nonaggregated

column 'information\_schema.PROFILING.SEQ' which is not functionally dependent on columns in GROUP BY clause; this is incompatible with sql\_mode=only\_full\_group\_by

**ONLY\_FULL\_GROUP\_BY：**对于GROUP BY聚合操作，若select中的列没有在group by中出现，那么这句SQL是不合法的。

**解决办法**：my.cnf中添加以下几行

[mysqld] sql\_mode='STRICT\_TRANS\_TABLES,NO\_ZERO\_IN\_DATE,NO\_ZERO\_DATE,ERROR\_FOR\_DIVISION\_BY\_ZERO,NO\_AUTO\_CREATE\_USER,NO\_ENGINE\_SUBSTITUTION'

sql\_mode 中去掉only\_full\_group\_by

然后重启MySQL Server即可

### 存储过程

**概念：**存储过称可以说是一个记录集，它是由一些T-SQL语句组成的代码块，这些T-SQL语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他

**好处：** 1、数据库执行动作时，是先编译后执行的。存储过程是一个编译后的代码块，所以效率要比T-SQL语句高

1. 网络中交互时可以替代大堆的T-SQL语句，所以也能降低网络的通信量，提高

通信速率

1. 系统管理员通过执行某一存储过程的权限进行限制，能够实现对相应的数据的

访问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全

1. 增强了SQL语言的功能和灵活性。可以用流控制语句编写，有很强的灵活性，

可以完成复杂的判断和运算

1. 允许标准组件是编程。被创建后，可以在程序中被多次调用，而不必重新编写

该存储过程的SQL语句。而且数据库专业人员可以随时对存储过程进行修改，对应用程序源代码毫无影响

**创建的格式：** CREATE PROCEDURE 存储过程名 (参数列表) BEGIN SQL语句代码块 END

例：DELIMITER **//**

**CREATE** **PROCEDURE** proc1**(OUT** s **int)**

**BEGIN**

**SELECT** **COUNT(\*)** **INTO** s **FROM** **user;**

**END** **//**

DELIMITER **;**

注：（1）这里需要注意的是DELIMITER //和DELIMITER ;两句，DELIMITER

是分割符的意思，因为MySQL默认以";"为分隔符，如果我们没有声明分割符，那么编译器会把存储过程当成SQL语句进行处理，则存储过程的编译过程会报错，所以要事先用DELIMITER关键字申明当前段分隔符，这样MySQL才会将";"当做存储过程中的代码，不会执行这些代码，用完了之后要把分隔符还原

用MySQL的Administrator管理工具时，可以直接创建，不再需要声明

1. 存储过程根据需要可能会有输入、输出、输入输出参数，这里有一

个输出参数s，类型是int型，如果有多个参数用","分割开

1. SQL语句代码块的开始与结束使用BEGIN与END进行标识

#### 参数变量

**参数：**三种参数类型 IN、OUT、INOUT

形式如： CREATE PROCEDURE([[IN |OUT |INOUT ] 参数名 数据类形...])

IN 输入参数:表示该参数的值必须在调用存储过程时指定，在存储过程中修改该参数的值不能被返回，为默认值

OUT 输出参数:该值可在存储过程内部被改变，并可返回

INOUT 输入输出参数:调用时指定，并且可被改变和返回

**IN参数例子** 同理 换成 OUT 、INOUT

DELIMITER **//**

**CREATE** **PROCEDURE** demo\_in\_parameter**(IN** p\_in **int)**

**BEGIN**

**SELECT** p\_in**;**

**SET**  p\_in**=**2**;**

**SELECT** p\_in**;**

**END** **//**

DELIMITER **;**

**CALL** demo\_in\_parameter**(**@p\_in**);**

**SELECT** @p\_in**;**

**变量：** DECLARE variable\_name [,variable\_name...] datatype [DEFAULT value];

注：datatype为MySQL的数据类型，如：int, float, date, varchar(length),datetime

例：**DECLARE** l\_int **int** unsigned **default** 4000000**;**

**DECLARE** l\_numeric **number(**8**,**2**)** **DEFAULT** 9.95**;**

**DECLARE** l\_date **date** **DEFAULT** '1999-12-31'**;**

**DECLARE** l\_datetime **datetime** **DEFAULT** '1999-12-31 23:59:59'**;**

**DECLARE** l\_varchar **varchar(**255**)** **DEFAULT** 'This will not be padded'**;**

**变量赋值：** SET 变量名 = 表达式值 [,variable\_name = expression ...]

**用户变量：** MySQL客户端使用用户变量

**SELECT** 'Hello World' **into** @x**;**   **SELECT** @x**;**

**SET** @y**=**'Goodbye Cruel World'**;**   **SELECT** @y**;**

**SET** @z**=**1**+**2**+**3**;**   **SELECT** @z**;**

存储过程中使用用户变量

**CREATE** **PROCEDURE** GreetWorld**(** **)** **SELECT** **CONCAT(**@greeting**,**' World'**);**

**SET** @greeting**=**'Hello'**;**

**CALL** GreetWorld**(** **);**

存储过程间传递全局范围的用户变量

**CREATE** **PROCEDURE** p1**()**   **SET** @last\_procedure**=**'p1'**;**

**CREATE** **PROCEDURE** p2**()** **SELECT** **CONCAT(**'Last procedure was '**,**@last\_proc**);**

**CALL** p1**(** **);**

**CALL** p2**(** **);**

注：①用户变量名一般以@开头 ②滥用用户变量会导致程序难以理解及管理

#### 语法

**存储过程的查询**

查看某个数据库下面的存储过程

**select** name **from** mysql**.**proc **where** db**=**’数据库名’**;**

或者

**select** routine\_name **from** information\_schema**.**routines **where** routine\_schema**=**'数据库名'**;**

或者

show **procedure** status **where** db**=**'数据库名'**;**

某个存储过程的详细

SHOW **CREATE** **PROCEDURE** 数据库**.**存储过程名**;**

**控制语句** if-then -else语句

DELIMITER **//**

**CREATE** **PROCEDURE** proc2**(IN** **parameter** **int)**

**begin**

**declare** var **int;**

**set** var**=parameter**+1**;**

**if** var**=**0 **then**

**insert** **into** t **values(**17**);**

**end** **if;**

**if** **parameter=**0 **then**

**update** t **set** s1**=**s1**+**1**;**

**else**

**update** t **set** s1**=**s1**+**2**;**

**end** **if;**

**end;** **//**

DELIMITER **;**

case语句：

DELIMITER **//**

**CREATE** **PROCEDURE** proc3 **(in** **parameter** **int)**

**begin**

**declare** var **int;**

**set** var**=parameter**+1**;**

**case** var

**when** 0 **then**

**insert** **into** t **values(**17**);**

**when** 1 **then**

**insert** **into** t **values(**18**);**

**else**

**insert** **into** t **values(**19**);**

**end** **case;**

**end;** **//**

DELIMITER **;**

while ···· end while 执行前进行检查

DELIMITER **//**

**CREATE** **PROCEDURE** proc4**()**

**begin**

**declare** var **int;**

**set** var**=**0**;**

**while** var**<**6 **do**

**insert** **into** t **values(**var**);**

**set** var**=**var+1**;**

**end** **while;**

**end;** **//**

DELIMITER **;**

repeat···· end repeat 执行操作后检查结果

DELIMITER **//**

**CREATE** **PROCEDURE** proc5 **()**

**begin**

**declare** v **int;**

**set** v**=**0**;**

**repeat**

**insert** **into** t **values(**v**);**

**set** v**=**v+1**;**

until v**>=**5

**end** **repeat;**

**end;** **//**

DELIMITER **;**

loop ·····end loop 不需要初始条件，这点和while 循环相似，同时

和repeat循环一样不需要结束条件, leave语句的意义是离开循环。

DELIMITER **//**

**CREATE** **PROCEDURE** proc6 **()**

**begin**

**declare** v **int;**

**set** v**=**0**;**

 LOOP\_LABLE**:loop**

**insert** **into** t **values(**v**);**

**set** v**=**v+1**;**

**if** v **>=**5 **then**

 leave LOOP\_LABLE**;**

**end** **if;**

**end** **loop;**

**end;** **//**

DELIMITER **;**

ITERATE: 通过引用复合语句的标号,来从新开始复合语句

DELIMITER **//**

**CREATE** **PROCEDURE** proc10 **()**

**begin**

**declare** v **int;**

**set** v**=**0**;**

LOOP\_LABLE**:loop**

**if** v**=**3 **then**

**set** v**=**v+1**;**

**ITERATE** LOOP\_LABLE**;**

**end** **if;**

**insert** **into** t **values(**v**);**

**set** v**=**v+1**;**

**if** v**>=**5 **then**

leave LOOP\_LABLE**;**

**end** **if;**

**end** **loop;**

**end;** **//**

DELIMITER ;

#### 基本函数

**字符串类**

CHARSET(str) //返回字串字符集

CONCAT (string2 [,... ]) //连接字串

INSTR (string ,substring ) //返回substring首次在string中出现的位置,不存在返回0

LCASE (string2 ) //转换成小写

LEFT (string2 ,length ) //从string2中的左边起取length个字符

LENGTH (string ) //string长度

LOAD\_FILE (file\_name ) //从文件读取内容

LOCATE (substring , string [,start\_position ] ) //同INSTR,但可指定开始位置

LPAD (string2 ,length ,pad ) //重复用pad加在string开头,直到字串长度为length

LTRIM (string2 ) //去除前端空格

REPEAT (string2 ,count ) //重复count次

REPLACE (str ,search\_str ,replace\_str ) //在str中用replace\_str替换search\_str

RPAD (string2 ,length ,pad) //在str后用pad补充,直到长度为length

RTRIM (string2 ) //去除后端空格

STRCMP (string1 ,string2 ) //逐字符比较两字串大小,

SUBSTRING (str , position [,length ])//从str的position开始,取length个字符,

注：处理字符串时，默认第一个字符下标为1，即参数position必须大于等于1

select substring(abcd',0,2);

TRIM([[BOTH|LEADING|TRAILING] [padding] FROM]string2)//去除指定位置指定字符

UCASE (string2 ) //转换成大写

RIGHT(string2,length) //取string2最后length个字符

SPACE(count) //生成count个空格

**数学类**

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

注：HEX()中可传入字符串，则返回其ASC-11码，如HEX('DEF')返回4142143，也可以传入十进制整数，返回其十六进制编码，如HEX(25)返回19

LEAST (number , number2 [,..]) //求最小值

MOD (numerator ,denominator ) //求余

POWER (number ,power ) //求指数

RAND([seed]) //随机数

ROUND (number [,decimals ]) //四舍五入,decimals为小数位数]

注：返回类型并非均为整数，如：

(1)默认变为整形值 select round(1.26);

(2)可以设定小数位数，返回浮点型数据 select round(1.567,2);

SIGN (number2 )

**日期时间类**

ADDTIME (date2 ,time\_interval ) //将time\_interval加到date2

CONVERT\_TZ (datetime2 ,fromTZ ,toTZ ) //转换时区

CURRENT\_DATE ( ) //当前日期

CURRENT\_TIME ( ) //当前时间

CURRENT\_TIMESTAMP ( ) //当前时间戳

DATE (datetime ) //返回datetime的日期部分

DATE\_ADD (date2 , INTERVAL d\_value d\_type ) //在date2中加上日期或时间

DATE\_FORMAT (datetime ,FormatCodes ) //使用formatcodes格式显示datetime

DATE\_SUB (date2 , INTERVAL d\_value d\_type ) //在date2上减去一个时间

DATEDIFF (date1 ,date2 ) //两个日期差

DAY (date ) //返回日期的天

DAYNAME (date ) //英文星期

DAYOFWEEK (date ) //星期(1-7) ,1为星期天

DAYOFYEAR (date ) //一年中的第几天

EXTRACT (interval\_name FROM date ) //从date中提取日期的指定部分

MAKEDATE (year ,day ) //给出年及年中的第几天,生成日期串

MAKETIME (hour ,minute ,second ) //生成时间串

MONTHNAME (date ) //英文月份名

NOW ( ) //当前时间

SEC\_TO\_TIME (seconds ) //秒数转成时间

STR\_TO\_DATE (string ,format ) //字串转成时间,以format格式显示

TIMEDIFF (datetime1 ,datetime2 ) //两个时间差

TIME\_TO\_SEC (time ) //时间转秒数]

WEEK (date\_time [,start\_of\_week ]) //第几周

YEAR (datetime ) //年份

DAYOFMONTH(datetime) //月的第几天

HOUR(datetime) //小时

LAST\_DAY(date) //date的月的最后日期

MICROSECOND(datetime) //微秒

MONTH(datetime) //月

MINUTE(datetime) //分返回符号,正负或0

SQRT(number2) //开平方

# WIN

## Cmd

netstat -ano 列出所有端口的情况

netstat -aon|findstr "端口号" 查看被占用端口的情况

tasklist|findstr "PID编号" 查看占用该端口号的进程

taskkill /f /t /im 进程名称 结束该进程

copy \*.sql test.sql 合并多个sql文件的内容至test.sql文件中

# Tools

## Eclipse

### 配置

**ecplise.ini 配置**

jdk引用

-vm

D:/applay/jdk1.8/bin/javaw.exe

-server 让JVM 以 server模式运行,加重JIT的优化作用,由于eclipse是经常

开着不关,在server模式下,JIT会随着运行的时间,把字节码更深刻的变成成机器代码.加快运行速度

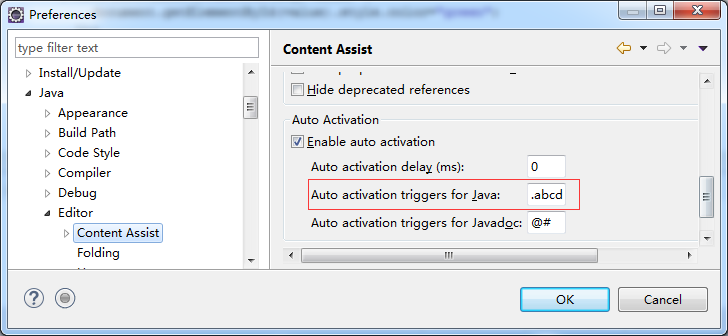
-Xverify:none 跳过对字节码的验证

-XX:PermSize=128m PermSize永生带设置为128M,堆的初始大小设置为256M,新生

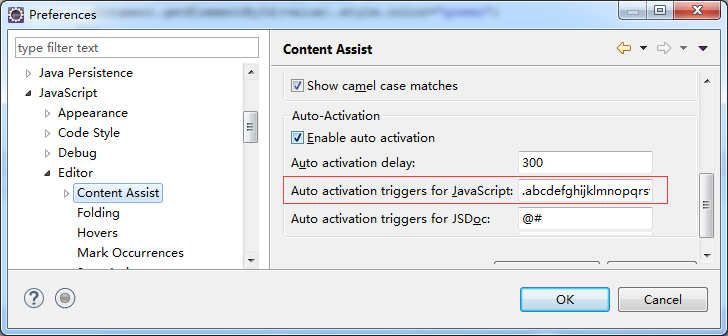
代站了40M. 每个线程栈大小设为2M

**代码提示**

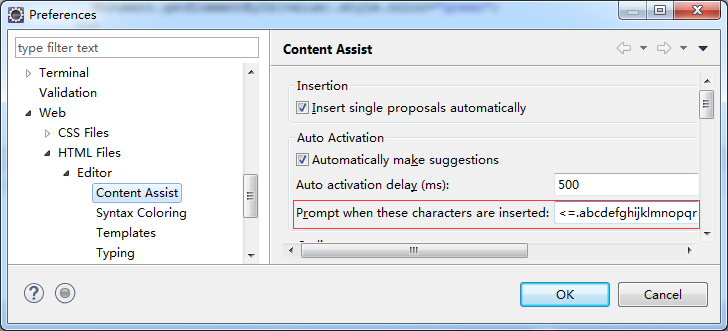
.abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ



.abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ



<=.abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

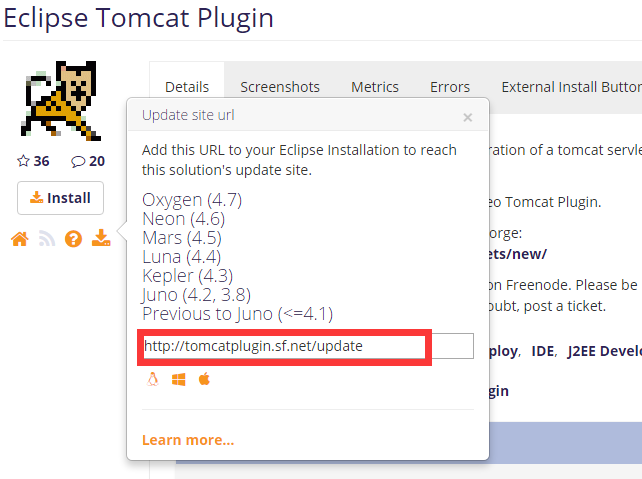


### 插件引用

**clean重启Eclipse** 进入eclipse安装目录，执行eclipse.exe -clean

IMG_259

**eclipse插件市场** http://marketplace.eclipse.org/



**JS插件**

1、http://www.spket.com/download.html 下载 Plugin

2、解压，如 D:\\EclipseNeon\\dropins\\spket-1.6.23

3、eclipse下新建subclipse.link文件，内容是 path=${your eclipse-plugin path}

4、重新启动eclipse，可以在eclipse的菜单"Help"-->"About Eclipse

。SDK"-->"Feature Details" 和"Plug-in Details"中看到新安装的插件。

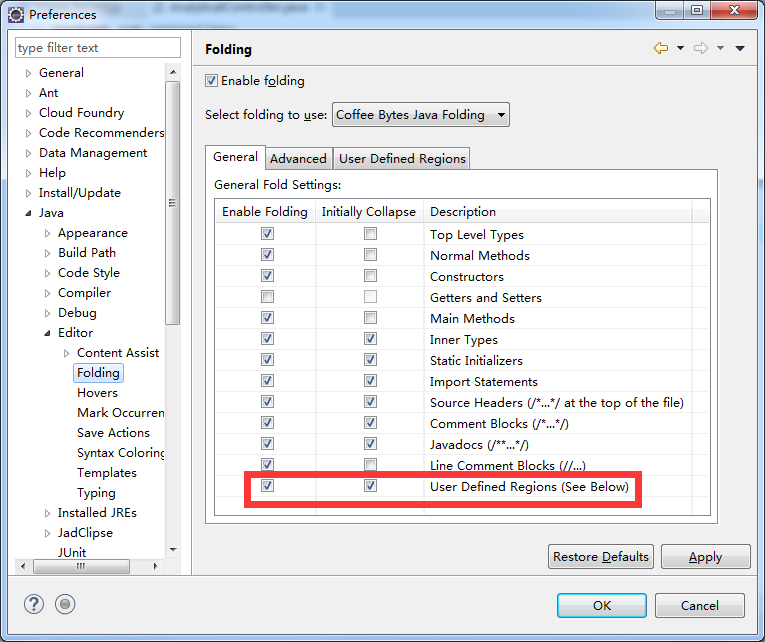
<spket-1.6.23>

**SVN插件**

http://subclipse.tigris.org/servlets/ProjectDocumentList 下载eclipse插件

<site-1.10.13-1.9.x>

**代码块折叠** //[start] 和 //[end] 之间的部分 com.cb.eclipse.folding\_1.0.6.jar



**Properties在线更新** http://propedit.sourceforge.jp/eclipse/updates/

**jadClipse查看class文件**

1、adClipse的jar包 sourceforge.net/projects/jadclipse/

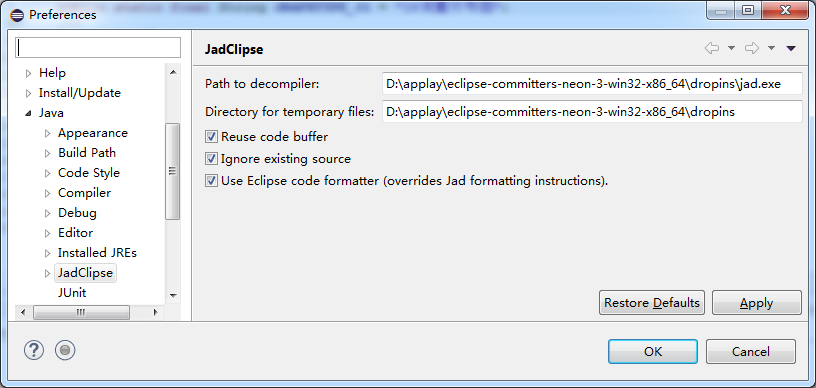
2、将net.sf.jadclipse\_3.3.0.jar拷贝到eclipse的dropins目录下，并添加 .link 路径

3、删除eclipse的configuration目录下org.eclipse.update文件

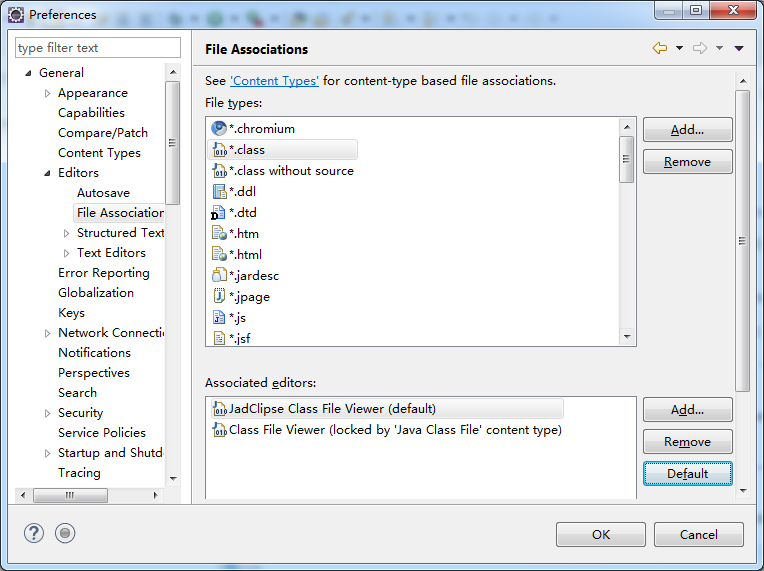
4、如果eclipse是开着的，点击菜单栏中File->Restart

5、下载jad的可执行文件 http://varaneckas.com/jad/ ，解压后放在某一磁盘中

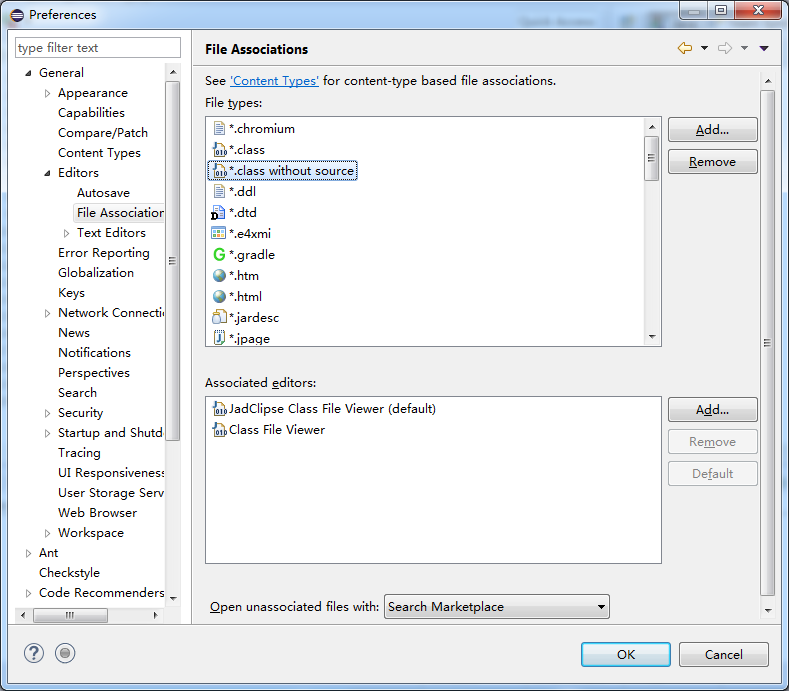
6、设置jad的可执行文件路径以及生成的临时文件路径



7、设置\*.class文件类型默认打开方式，设置default



8、设置\*.class without source文件类型打开方式，设置default



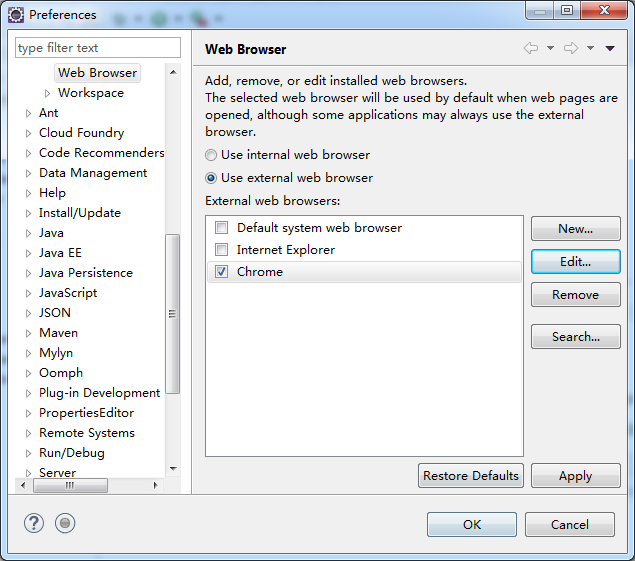
**tomcat插件** http://www.eclipsetotale.com/tomcatPlugin.html

注：与trotoise版本一致

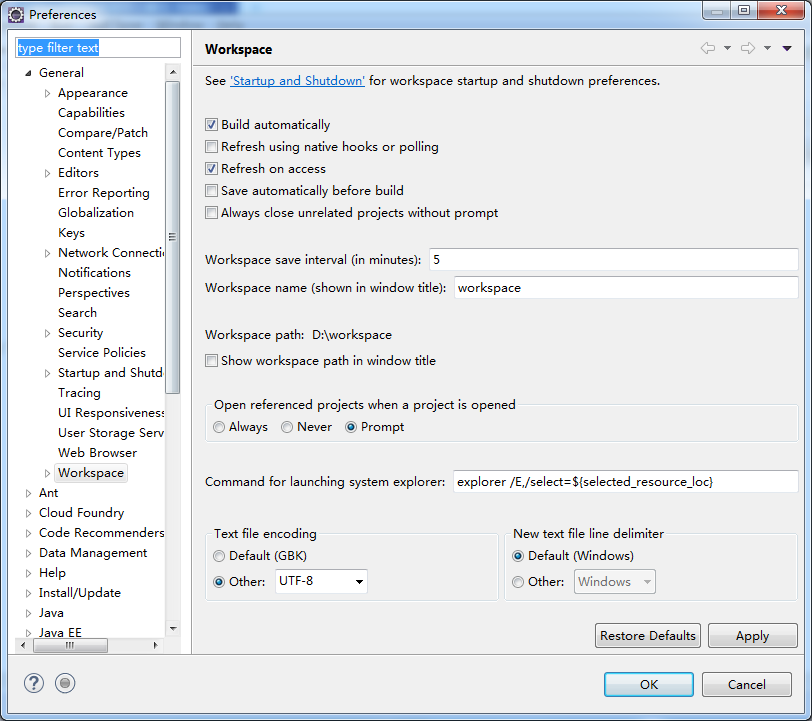
### preferences设置

**当前参数设置（背景颜色、提示颜色等）** 当前对象右键 -> preferences

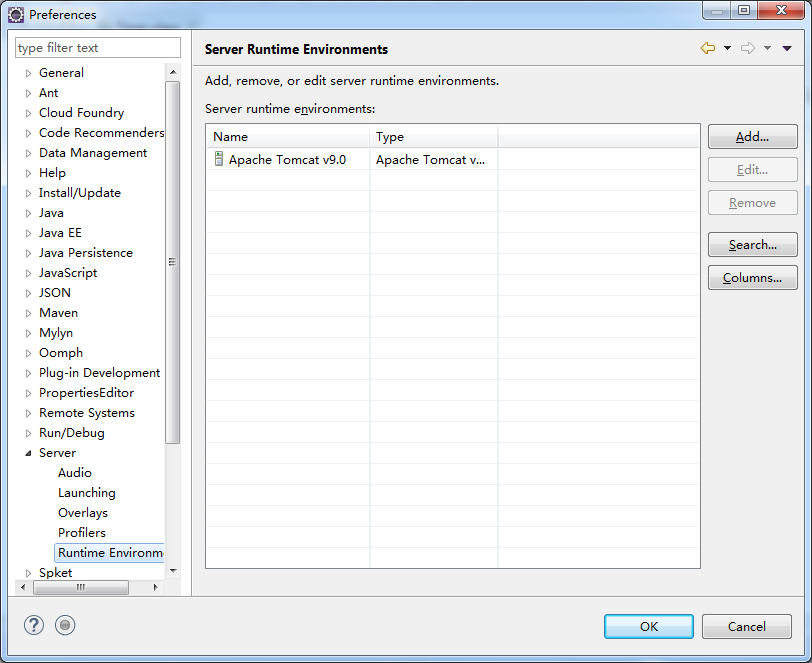
**修改默认浏览器**



**控制台编码格式**



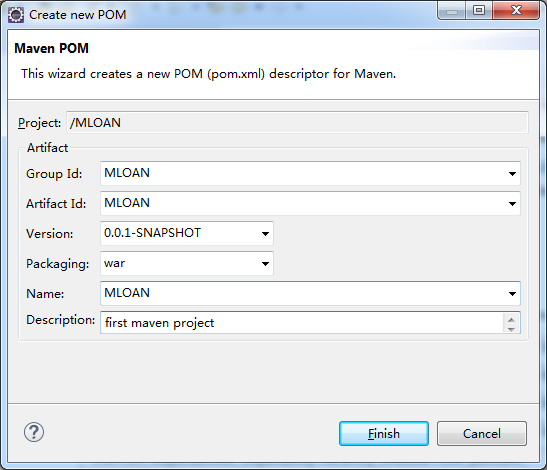
**集成Tomcat**



### 项目设置

**Java Web工程转换为Maven工程**

选中该工程右键 -> configure -> Convert to Maven project:



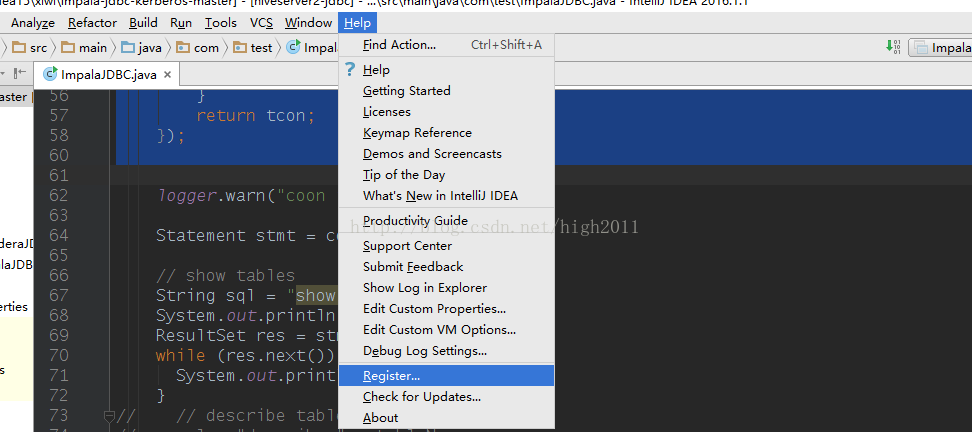
### 快捷键

**Alt+Shift+A** 进入列编辑模式，再按一次退出

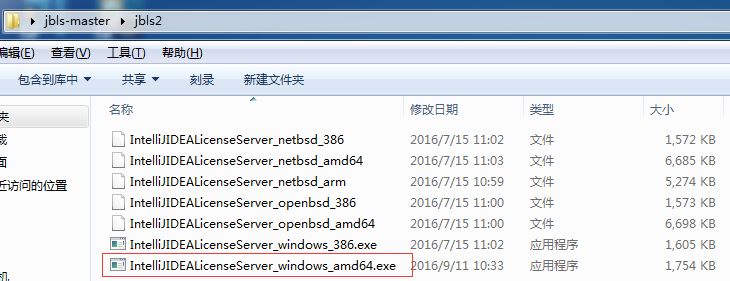
## Idea

### 永久激活方法

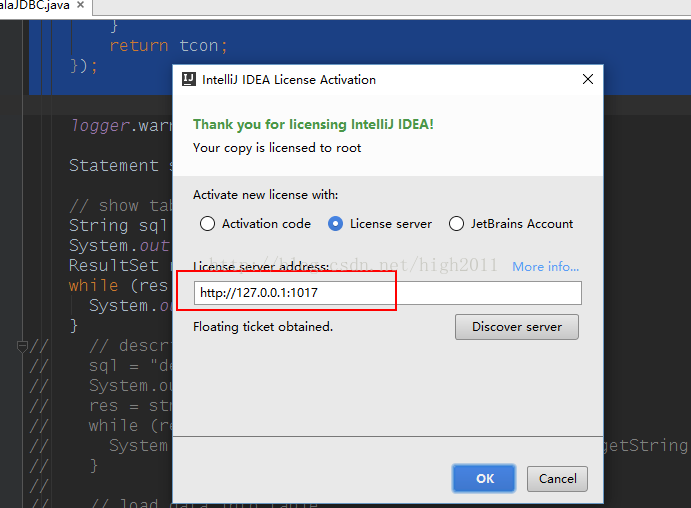
选择Help--------》Register



[操作系统](http://lib.csdn.net/base/operatingsystem)选择解压后的exe文件 <jbls-master>



输入（本机）注册机地址：http://127.0.0.1:1017



### 配置文件修改

**执行文件的 JVM 配置文件内容**

32 位的系统修改 idea.exe.vmoptions 文件里面的内容，但是由于 32 位系统内存一般都是 2G 左右的，所以也没有多大空间可以调整，一般无需调整

64 位的系统修改 idea64.exe.vmoptions

注：修改的原则主要是根据自己机器的内存情况来判断的，8G 以下的机子或是静态页面开发者无需修改的。

常修改的 4 个参数

-Xms128m，16 G 内存的机器可尝试设置为 -Xms512m

-Xmx750m，16 G 内存的机器可尝试设置为 -Xmx1500m

-XX:MaxPermSize=350m，16G 内存的机器可尝试设置为 -XX:MaxPermSize=500m

-XX:ReservedCodeCacheSize=225m，16G 内存的机器可尝试设置为

-XX:ReservedCodeCacheSize=500m

**属性配置**：没有 32 位和 64 位之分，idea.properties

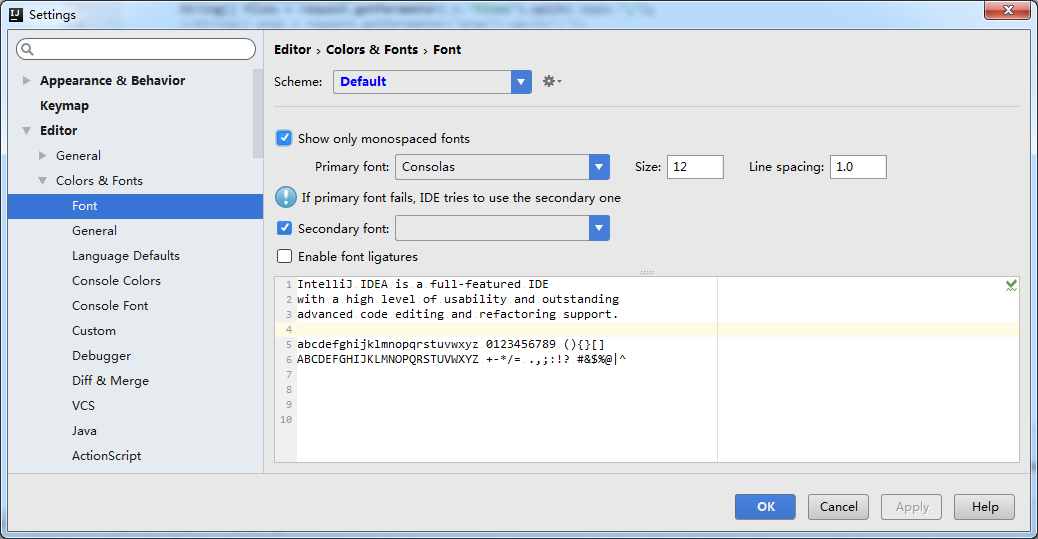
* idea.config.path=${user.home}/.IntelliJIdea/config，指向 IntelliJ IDEA 的个性化配置目录，默认是被注释，这里需要特别注意的是斜杠方向，正斜杠。
* idea.system.path=${user.home}/.IntelliJIdea/system，指向 IntelliJ IDEA 的系统文件目录，默认是被注释，这里需要特别注意的是斜杠方向，正斜杠。如果你的项目很多，则该目录会很大，如果你的 C 盘空间不够的时候，还是建议把该目录转移到其他盘符下。
* idea.max.intellisense.filesize=2500，该属性主要用于提高在编辑大文件时候的代码帮助。IntelliJ IDEA 在编辑大文件的时候还是很容易卡顿的。
* idea.cycle.buffer.size=1024，该属性主要用于控制控制台输出缓存。有遇到一些项目开启很多输出，控制台很快就被刷满了没办法再自动输出后面内容，这种项目建议增大该值或是直接禁用掉，禁用语句 idea.cycle.buffer.size=disabled

### 工具优化

**字体设置**

* 勾选的 Show only monospaced fonts 筛选显示系统上的等宽字体。取消勾选之后，可以显示系统上所有已安装的字体。
* 编码字体有第一字体（Primary font） 和 第二字体（Secondary font）之分。当有些字符在第一字体支持不了的时候，会去使用第二字体进行支持。

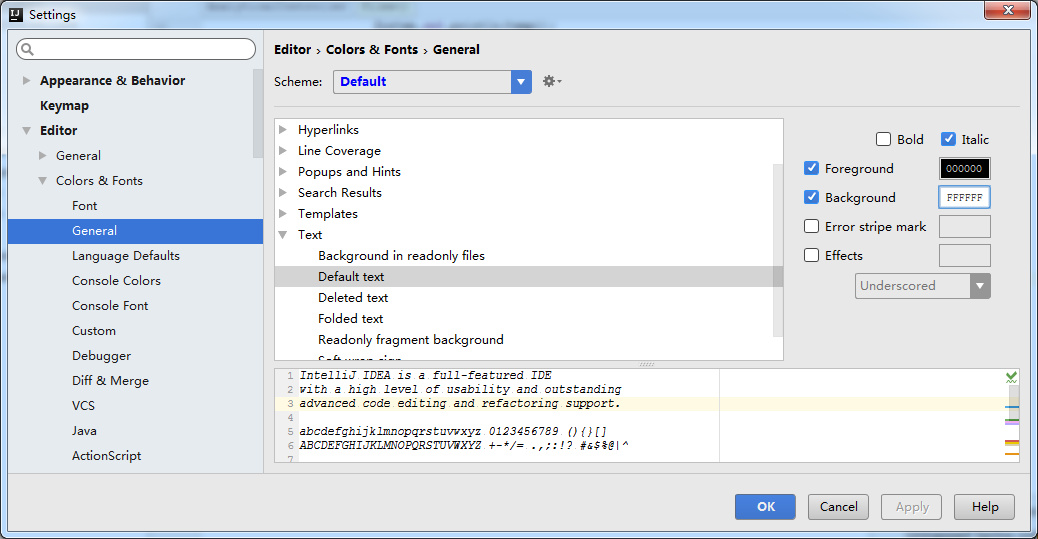
推荐：英文字体使用 Monaco或Courier new，由于此字体不支持中文，所以我把这个设置为第一字体，第二字体使用 Yahei Consolas Hybrid 进行支持，该字体含有中文。这两个字体都不是系统自带的，需要自行下载安装。



**主题细节：字体、背景等**

General 区，我常修改的有：

* Default text，默认代码文本
* Caret row，光标所在行
* Vertical indent guide，垂直缩进线，我一般会修改其 Foreground 属性
* Identifier under caret，光标所在位置的相同标识符呈现什么效果
* Text search result，在查找模式下，匹配字符的样式



**文件编码**

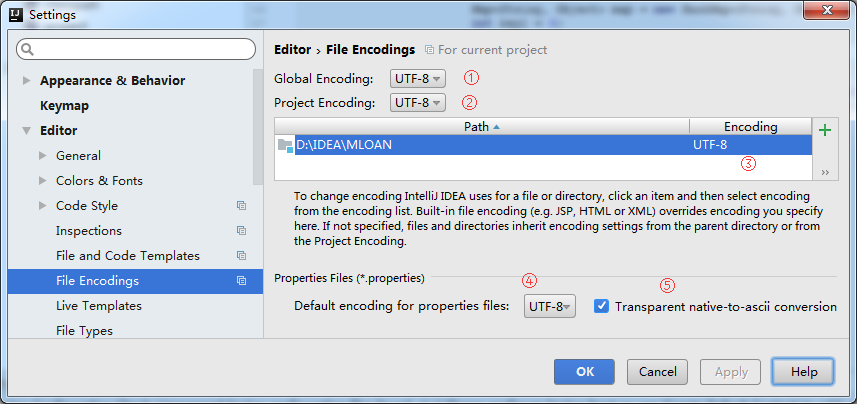
① IDE 的编码默认是 UTF-8，

② Project Encoding 默认是 GBK，建议修改为 UTF-8。

③ 对目录、文件进行编码设置。会出现需要 Convert 编码的弹出操作选择，在转换之前做好文件备份，不然可能出现转换过程变成乱码，无法还原。

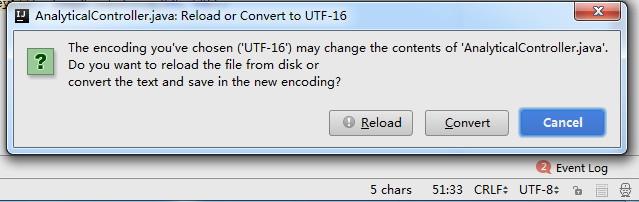
④ IntelliJ IDEA 可以对 Properties 文件进行专门的编码设置，建议改为 UTF-8，其中有一个重点就是属性 Transparent native-to-ascii conversion

⑤ 对于 Properties 文件，重要属性 Transparent native-to-ascii conversion 主要用于转换 ascii，一般都要勾选，不然 Properties 文件中的注释显示的都不会是中文。

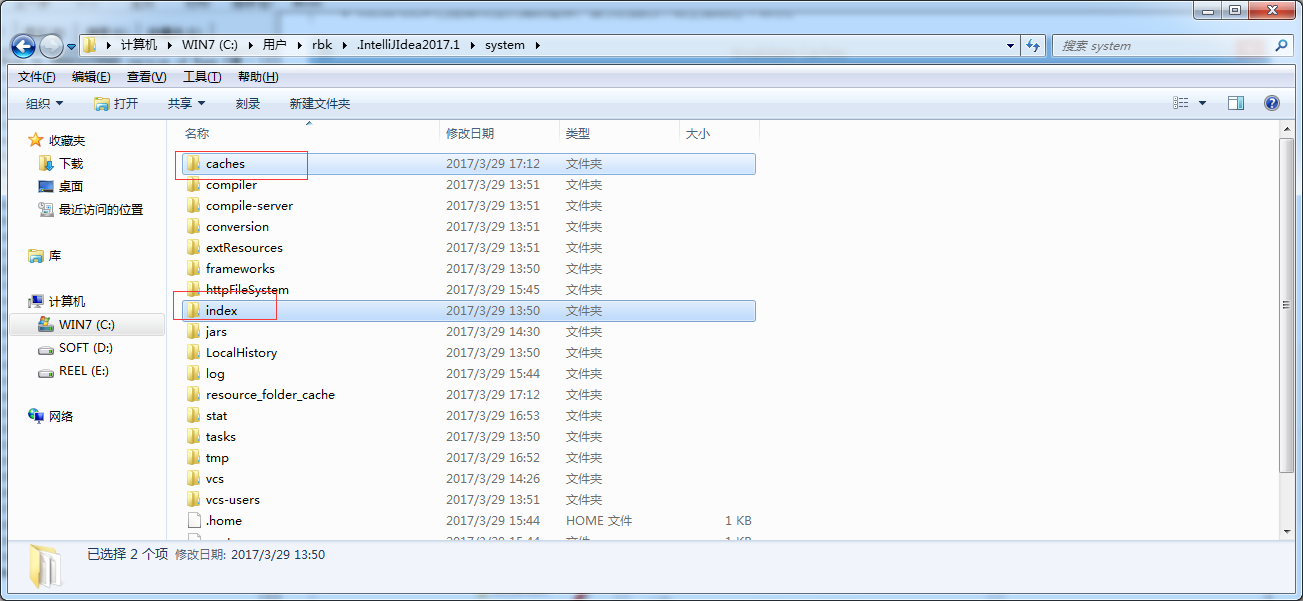


**文件编码转换**

* Reload 使用新编码重新加载，新编码不会保存到文件中，重新打开此文件，旧编码是什么依旧还是什么。
* Convert 使用新编码进行转换，新编码会保存到文件中，重新打开此文件，新编码是什么则是什么。
* 含有中文的代码文件，Convert 之后可能会使中文变成乱码，所以在转换成请做好备份，不然可能出现转换过程变成乱码，无法还原。

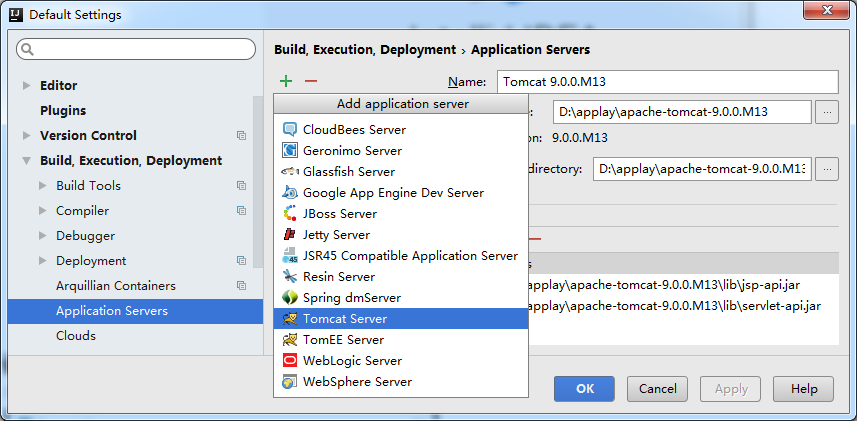


**手动清除缓存、索引**



### 手动添加tomcat

1. 进入项目之前或者close project，Configure -> Plugins
2. 在搜索框输入tomcat，并选中，点击OK
3. 返回后 Configure -> Plugins
4. 如下图，此时点击加号便有了tomcat选项

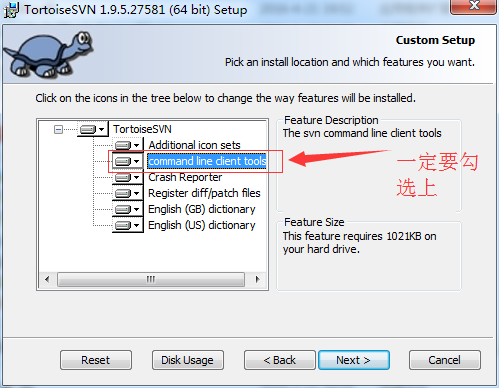


### 手动添加SVN

**插件地址** https://sliksvn.com/download/

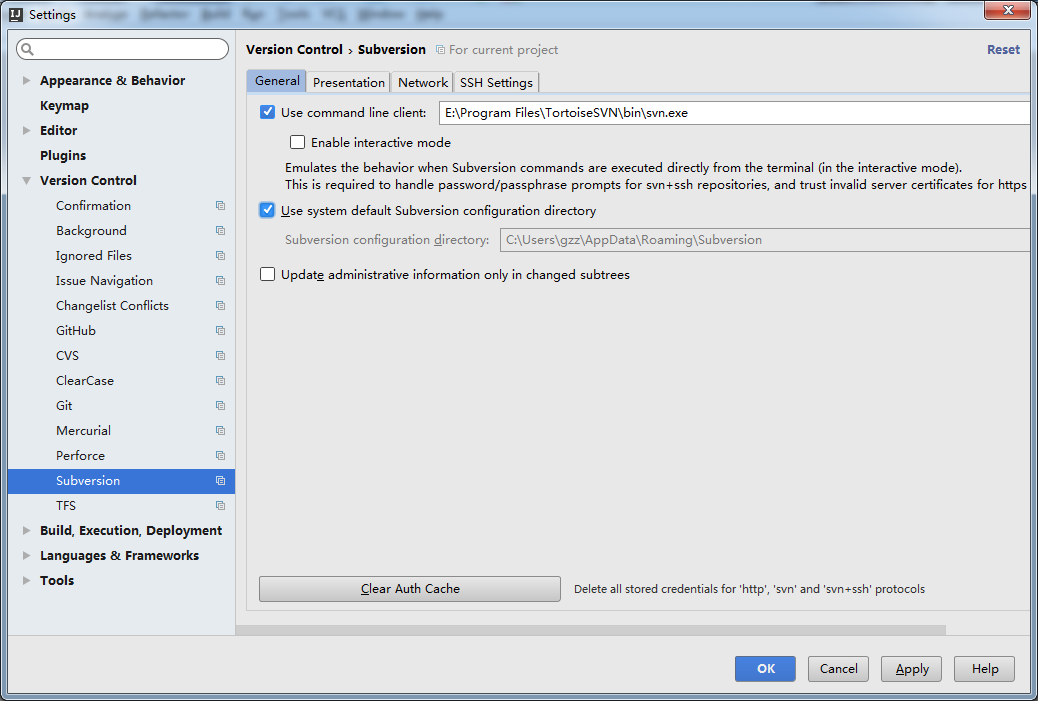
Slik-Subversion-1.9.5-x64.msi

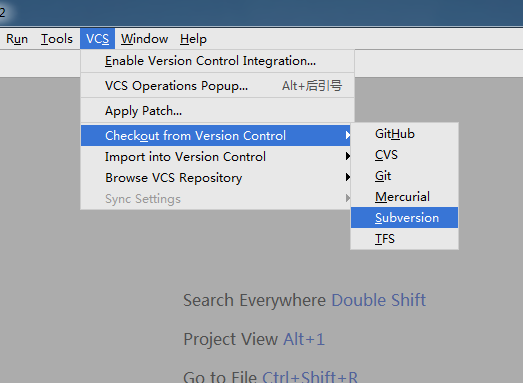
**安装注意事项**



**setting配置**

file - setting 按钮打开设置界面 或者（Ctrl + Alt + S）快捷键







### 代码提示快捷键

File（文件） -> Setting（设置），KeyMap，Default Copy，Main menu -> Code -> Completion -> Basic

**【IntelliJ Idea 常用快捷键列表】**

Ctrl+Shift + Enter 语句完成

“！” 否定完成，输入表达式时按 “！”键

Ctrl+E 最近的文件

Ctrl+Shift+E 最近更改的文件

Shift+Click 关闭文件

Ctrl+[ OR ] 跑到大括号的开头与结尾

Ctrl+F12 显示当前文件的结构

Ctrl+F7 查询当前元素在当前文件中的引用，然后按 F3 可以选择

Ctrl+N 快速打开类

Ctrl+Shift+N 快速打开文件

Alt+Q 看到当前方法的声明

Ctrl+P 显示参数信息

Ctrl+Shift+Insert 选择剪贴板内容并插入

Alt+Insert 生成构造器/Getter/Setter等

Ctrl+Alt+V 引入变量。例如：new String(); 自动导入变量定义

Ctrl+Alt+T 把代码包在一个块内，例如：try/catch

Ctrl+Enter 导入包，自动修正

Ctrl+Alt+L 格式化代码

Ctrl+Alt+I 将选中的代码进行自动缩进编排，在编辑 JSP 文件时也可以工作

Ctrl+Alt+O 优化导入的类和包

Ctrl+R 替换文本

Ctrl+F 查找文本

Ctrl+Shift+Space 自动补全代码

Ctrl+空格 代码提示（与系统输入法快捷键冲突）

Ctrl+Shift+Alt+N 查找类中的方法或变量

Alt+Shift+C 最近的更改

Alt+Shift+Up/Down 上/下移一行

Shift+F6 重构 – 重命名

Ctrl+X 删除行

Ctrl+D 复制行

Ctrl+/或Ctrl+Shift+/ 注释（//或者/\*\*/）

Ctrl+J 自动代码（例如：serr）

Ctrl+Alt+J 用动态模板环绕

Ctrl+H 显示类结构图（类的继承层次）

Ctrl+Q 显示注释文档

Alt+F1 查找代码所在位置

Alt+1 快速打开或隐藏工程面板

Ctrl+Alt+left/right 返回至上次浏览的位置

Alt+left/right 切换代码视图

Alt+Up/Down 在方法间快速移动定位

Ctrl+Shift+Up/Down 向上/下移动语句

F2 或 Shift+F2 高亮错误或警告快速定位

Tab 代码标签输入完成后，按 Tab，生成代码

Ctrl+Shift+F7 高亮显示所有该文本，按 Esc 高亮消失

Alt+F3 逐个往下查找相同文本，并高亮显示

Ctrl+Up/Down 光标中转到第一行或最后一行下

Ctrl+B/Ctrl+Click 快速打开光标处的类或方法（跳转到定义处）

Ctrl+Alt+B 跳转到方法实现处

Ctrl+Shift+Backspace 跳转到上次编辑的地方

Ctrl+O 重写方法

Ctrl+Alt+Space 类名自动完成

Ctrl+Alt+Up/Down 快速跳转搜索结果

Ctrl+Shift+J 整合两行

Alt+F8 计算变量值

Ctrl+Shift+V 将最近使用的剪贴板内容选择插入到文本

Ctrl+Alt+Shift+V 简单粘贴

Shift+Esc 把焦点移到编辑器上，隐藏当前（或最后活动的）工具窗口

F12 把焦点从编辑器移到最近使用的工具窗口

Shift+F1 要打开编辑器光标字符处使用的类或者方法 Java 文档的浏览器

Ctrl+W 可以选择单词继而语句继而行继而函数

Ctrl+Shift+W 取消选择光标所在词

Alt+F7 查找整个工程中使用地某一个类、方法或者变量的位置

Ctrl+I 实现方法

Ctrl+Shift+U 大小写转化

Ctrl+Y 删除当前行

Shift+Enter 向下插入新行

psvm/sout，main/System.out.println(); Ctrl+J，查看更多

Ctrl+Shift+F 全局查找

Ctrl+F，查找/Shift+F3，向上查找/F3，向下查找

Ctrl+Shift+S 高级搜索

Ctrl+U 转到父类

Ctrl+Alt+S 设置对话框

Alt+Shift+Inert 开启/关闭列选择模式

Ctrl+Alt+Shift+S 打开当前项目/模块属性

Ctrl+G 定位行

Alt+Home 跳转到导航栏

Ctrl+Enter 上插一行

Ctrl+Backspace 按单词删除

Ctrl+”+/-” 当前方法展开、折叠

Ctrl+Shift+”+/-” 全部展开、折叠

**【调试部分、编译】**

Ctrl+F2 停止

Alt+Shift+F9 选择 Debug

Alt+Shift+F10 选择 Run

Ctrl+Shift+F9 编译

Ctrl+Shift+F10 运行

Ctrl+Shift+F8 查看断点

F8 步过

F7 步入

Shift+F7 智能步入

Shift+F8 步出

Alt+Shift+F8 强制步过

Alt+Shift+F7 强制步入

Alt+F9 运行至光标处

Ctrl+Alt+F9 强制运行至光标处

F9 恢复程序

Alt+F10 定位到断点

Ctrl+F8 切换行断点

Ctrl+F9 生成项目

Alt+1 项目

Alt+2 收藏

Alt+6 TODO

Alt+7 结构

Ctrl+Shift+C 复制路径

Ctrl+Alt+Shift+C 复制引用，必须选择类名

Ctrl+Alt+Y 同步

Ctrl+~ 快速切换方案（界面外观、代码风格、快捷键映射等菜单）

Shift+F12 还原默认布局

Ctrl+Shift+F12 隐藏/恢复所有窗口

Ctrl+F4 关闭

Ctrl+Shift+F4 关闭活动选项卡

Ctrl+Tab 转到下一个拆分器

Ctrl+Shift+Tab 转到上一个拆分器

**【重构】**

Ctrl+Alt+Shift+T 弹出重构菜单

Shift+F6 重命名

F6 移动

F5 复制

Alt+Delete 安全删除

Ctrl+Alt+N 内联

**【查找】**

Ctrl+F 查找

Ctrl+R 替换

F3 查找下一个

Shift+F3 查找上一个

Ctrl+Shift+F 在路径中查找

Ctrl+Shift+R 在路径中替换

Ctrl+Shift+S 搜索结构

Ctrl+Shift+M 替换结构

Alt+F7 查找用法

Ctrl+Alt+F7 显示用法

Ctrl+F7 在文件中查找用法

Ctrl+Shift+F7 在文件中高亮显示用法

**【基本操作】**

Ctrl+z 撤销快捷键

Ctrl + Shift + Z 恢复Ctrl+z 掉的内容

## Maven

### 坐标

**maven** 坐标通常用冒号作为分割符来书写,像这样的格式：groupId：artifactId：

packaging:version

**groupId** 团体，公司，小组，组织，项目，或者其他团体。团体标识的约定是，它以创建这个项目的组织名称的逆向域名（reverse domain name）开头

**artifactId** 在groupId下的表示一个单独项目的唯一标识符。

**version** 一个项目的特定版本，不能同时有一个同样的groupId，artifactId和version标识的项目

**packaging** 项目的类型，默认是jar，描述了项目打包后的输出。类型为jar的项目产生一个jar文件，类型为war的项目产生一个web应用

### 依赖

传递性依赖（transitive dependencies）。假如你的项目依赖于一个库，而这个库又依赖其他，maven自动加入这些，也可以排除一些特定的传递性依赖。

一个依赖不仅仅是一个jar，他是一个pom（project object model）文件，这个pom可能声明了对其他构建的依赖。

提供了不同的依赖范围（dependency scope）。simple项目包含了一个依赖：--junit：junit：jar:3.8.1---范围test。当一个依赖的范围是test的时候，说明它在compiler插件运行compile目标的时候是不可用的。它只有在运行compiler:testCompile和surefire:test目标的时候才会被加入到classpath中。

当为项目创建jar文件的时候，它的依赖不会捆绑在生成的构件中，他们只是用来编译。当maven来创建war或jar，你可以配置maven让它在生成的构件中捆绑依赖，也可以用provided范围，让它排除war文件中特定的依赖。provided范围告诉maven一个依赖在编译的时候需要，但是它不应该被捆绑在构件的输出中。当你开发web应用的时候provided范围变得十分有用，你需要Servlet API来编译你的代码，但是你不希望Servlet API的jar文件包含在你的web应用的web-inf/lib目录中

compile(编译范围)如果没有提供一个范围，编译范围依赖在所有的classpath中可用，同时他们也会被打包

provided（已提供范围）provided依赖只有在当jdk或者一个容器已提供该依赖之后才使用。例如，如果你开发了一个web应用，你可能在编译classpath中需要可用的Servlet api；这个servlet api jar由你的应用服务器或者servlet容器提供。已提供范围的依赖在编译classpath（不是运行时）可以用。他们不是传递性的，也不会被打包。

runtime(运行时范围)依赖在运行和测试系统的时候需要。但在编译的时候不需要。比如，你可能在编译的时候只需要jdbc api jar，而只有在运行的时候才需要jdbc驱动实现。

test(测试范围)依赖在一般的编译和运行时都不需要，他们只有在测试编译和测试运行阶段可用。

system(系统范围)依赖与provided类似，但是你必须显示的提供一个对于本地系统中jar文件的路径。这么做事为了允许基于本地对象编译，而这些对象是系统类库的一部分。这样的构件应该是一直可用的，maven也不会在仓库中寻找它。如果你将一个依赖范围设置成系统范围，你必须同时提供一个systemPath元素。这一该范围是不推荐使用的

<dependency>

<groupId>**org.apache.poi**</groupId>

<artifactId>**poi-ooxml**</artifactId>

<version>**3.6**</version>

<scope>**system**</scope>

<systemPath>**${basedir}/src/main/webapp/WEB-INF/lib/poi-ooxml-3.6.jar**</systemPath>

</dependency>

## Svn

## Tomcat

### ProjectName.XML

<Context path="/MLOAN" reloadable="false" docBase="D:\workspace\MLOAN\WebRoot" workDir="D:\applay\apache-tomcat-9\webapps"/>

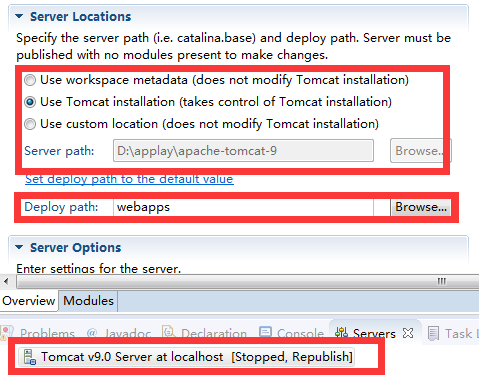
**workDir**：jsp运行时都要先转换成servlet，使用tomcat时会在tomcat安装目录下的work生成一系列的文件夹和.java文件和.class文件

我想更改默认的work目录为其他的，可以做到吗？比如象虚拟目录那样的，我的jsp文件放在d:\jsp里,我想让.java文件和.class文件自动存放到D:\jspclass里

path 位虚拟目录　docBase 为实际目录

注：context.xml 文件中 配置的设置将为整个服务器所共享

### 项目发布目录设置



# Topic

## Global

**1、什么是中间件**：是程序中 可植入的、可重用的、与业务逻辑无关的 各种组件

分类：数据访问中间件，远程调用中间件，消息中间件，交易中间件，对象中间件

例： 1 、 RMI （ Remote Method Invocations, 远程调用）

2 、 Load Balancing( 负载均衡，将访问负荷分散到各个服务器中 )

3 、 Transparent Fail-over( 透明的故障切换 )

4 、Clustering( 集群 , 用多个小的服务器代替大型机）

5 、 Back-end-Integration( 后端集成，用现有的、新开发的系统如何去集

成遗留的系统 )

6 、 T ransaction 事务（全局 / 局部）全局事务，控制在同一数据库联接

内的事务

7 、 Dynamic Redeployment ( 动态重新部署 , 在不停止原系统的情况下，

部署新的系统）

8 、 System Management( 系统管理 )

9 、 Threading( 多线程处理 )

10 、 Message-oriented Middleware 面向消息的中间件（异步的调用编程）

11 、 Component Life Cycle( 组件的生命周期管理 )

12 、 Resource pooling （资源池）

13 、 Security （安全）

14 、 Caching （缓存）

**2、软件分层有什么好处：**使代码便于维护，设计明确，各层独立，专注各自擅长的领域

**3、名词解释**

API Application Programming Interface 应用程序编程接口

CRM Customer Relationship Management 客户关系管理

ERP Enterprise Resource Planning 企业资源规划

OSS Operation Support System 运营支撑系统

BSS Business Support System 数据定义语言

BOSS Business Operation Support System 数据定义语言

OA Office Automatizat ion 办公自动化

HTTP OOP SOA GUI

DDL Data Definit ion Language 数据定义语言

DML Data Manipulat ion Language 数据操作语言

WYSIWYG What You See is What You Get 所见即所得

CMP CMT B2B C2C B2C IDE DOM Client/Server CMM ORM MIS OLE

CORBA 公共对象请求代理结构 (Common Object Request Broker Architecture) ，由 对象管理组织 (Object Management Group ，缩写为 OMG) 标准化。它的组成是接口定义语言(IDL), 语言绑定 (binding: 也译为联编 ) 和允许应用程序间互操作的协议。 其目的为： 用不同的程序设计语言书写在不同的进程中运行为不同的操作系统开发。

UML 标准建模语言，包含用例图 , 静态图 (类图、对象图、包图 ), 行为图 , 交互图 ( 顺 序图 , 合作图 ), 实现图等。

XML CMMI JRE

J2EE Sun 公司提出的多层 (multi-diered), 分布式 (distributed), 基于组件 (component-base )的企业级应用模型(enterpriese application model). 在这样的一个应用系统中，可按照功能划分为不同的组件，这些组件又可在不同计算机上，并且处于相应的层次 (tier) 中。所属层次包 括客户层 (clietn tier) 组件 ,web 层和组件 ,Business 层和组件 , 企业信息系统 (EIS) 层

JDK AOP OO Container W3C JMS Domain POJO JVM JNDI JTA

SOAP Simple Object Access Protocol 简单对象访问协议

WSDL

JDO Java 对象持久化的新的规范，为 Java Data Object 的简称 , 也是一个用于存取某种 数据仓库中的对象的标准化 API 。 JDO 提供了透明的对象存储，因此对开发人员来说，存储 数据对象完全不需要额外的代码（如 JDBC API 的使用）。这些繁琐的例行工作已经转移到 J DO产品提供商身上，使开发人员解脱出来，从而集中时间和精力在业务逻辑上。另外， JDO很灵活，因为它可以在任何数据底层上运行。 JDBC 只是面向关系数据库（ [RDBMS](#RMDBS))JDO 更通用，提供到任何数据底层的存储功能，比如关系数据库、文件、 XM L 以及对象数据库（ ODBMS ）等等，使得应用可移植性更强

TDD DAO

IOC Inversion of Control 控制反转

RMI

DNS Internet Domain Name System 因特网域名系统

URL URI

RDBMS Relational Database Management System 关系数据库管理系统,是建立实体之间的联系,最后得到的是关系表  
ORDBMS 对象关系数据库在实质上还是关系数据库

OODBMS 面向对象数据库管理系统,将所有实体都看着对象,并将这些对象类进行封装,

对象之间的通信通过消息

注：面向对象的数据库只在某些特定的领域应用到,最典型的就是地理信息系统

**4、谈谈 SSH 整合：** Struts （表示层） +Spring （业务层） +Hibernate （持久层）

Struts ：一个表示层框架，主要作用是界面展示，接收请求，分发请求

注：在 MVC 框架中， Struts 属于 VC 层次，负责界面表现，负责 MVC 关系的分发。（ Vie w ：沿用 JSP ， HTTP ， Form ， Tag ， Resourse ； Controller ：ActionServlet ， struts-config.xml ， A ction ）

Hibernate ：一个持久层框架，它只负责与关系数据库的操作。

Spring ：一个业务层框架，是一个整合的框架，能够很好地关联表示层与持久层。

**5、 应用服务器与 Web Server 的区别**

Web服务器只负责处理HTTP协议，只能发送静态页面的内容

JSP，ASP，PHP等动态内容需要通过CGI、FastCGI、ISAPI等接口交给应用服务器去处理

如：Web服务器包括Nginx，Apache，IIS等。

应用服务器包括WebLogic，JBoss，Tomcat等。应用服务器一般也支持HTTP协议，因此界限没这么清晰。但是应用服务器的HTTP协议部分仅仅是支持，一般不会做特别优化，所以很少有见Tomcat直接暴露给外面，而是和Nginx、Apache等配合，只让Tomcat处理JSP和Servlet部分

**6、什么是 WebService ：**一种跨编程语言和跨[操作系统](http://lib.csdn.net/base/operatingsystem)平台的远程调用技术，可用于XML数据传递

注：提供标准的协议和接口，让不同的程序集成的一种 SOA（面向服务的编程）架构

优点 (1) 可以让异构的程序相互访问（跨平台）

(2) 松耦合

(3) 基于标准协议（通用语言，允许其他程序访问）

基本原理 (1) Service Provider 采用 WSDL 描述服务

1. Service Provider 采用 UDDI 将服务的描述文件发布到 UDDI 服务

器（ Register server ）

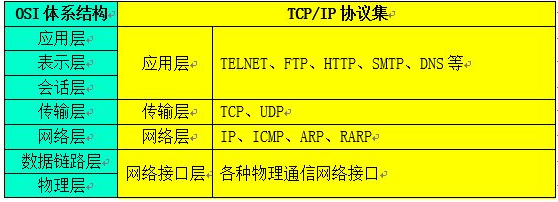
(3) Service Requestor 在 UDDI 服务器上查询并 获取 WSDL 文件

(4) Service requestor 将请求绑定到 SOAP ，并访问相应的服务

**7、Memcached**

高性能的分布式内存对象缓存系统，用于动态Web应用以减轻数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高动态、数据库驱动网站的速度。Memcached基于一个存储键/值对的hashmap。其守护进程（daemon ）是用C写的，但是客户端可以用任何语言来编写，并通过memcached协议与守护进程通信。

**8、网络 7 层协议， tcp/ip4 层协议**



**OSI七层模型**

1、物理层：网卡、网线、集线器、中继器、调制解调器等物理设备

2、数据链路层：在网络层实体间提供数据发送和接收的功能和过程；提供数据链

路的流控。网桥、交换机

1. 网络层：控制分组传送系统的操作、路由选择、拥护控制、网络互连等功能。

路由器

1. 传输层：提供建立、维护和拆除传送连接的功能；选择网络层提供最合适的服

务；在系统之间提供可靠的透明的数据传送，提供端到端的错误恢复和流量控制

1. 会话层：提供两进程之间建立、维护和结束会话连接的功能；提供交互会话的

管理功能，如三种数据流方向的控制，即一路交互、两路交替和两路同时会话模式 。

6、表示层：代表应用进程协商数据表示；完成数据转换、格式化和文本压缩

7、应用层：提供OSI用户服务，例如事务处理程序、文件传送协议和网络管理等。

**TCP/IP模型实际上是OSI模型的一个浓缩版本，只有四个层次**

1. 应用层：为用户提供所需要的各种应用协议服务，如：FTP、Telnet、DNS、

SMTP

1. 传输层：为应用层实体提供端到端的通信功能，保证了数据包的顺序传送及数

据的完整性

注：TCP协议(Transmission Control Protocol)提供可靠的数据流运输服务，UDP

协议(Use Datagram Protocol)提供不可靠的用户数据报服务

3、互联层：主要解决主机到主机的通信问题。负责数据的包装、寻址和路由

注：网际协议（IP）、互联网组管理协议（IGMP）、互联网控制报文协议（ICMP）、

[ARP](http://baike.baidu.com/item/ARP)(Address Resolution Protocol）地址转换协议、RARP(Reverse ARP)反向地址转换协议

1. 网络接口层：负责监视数据在主机和网络之间的交换。事实上，TCP/IP本身并

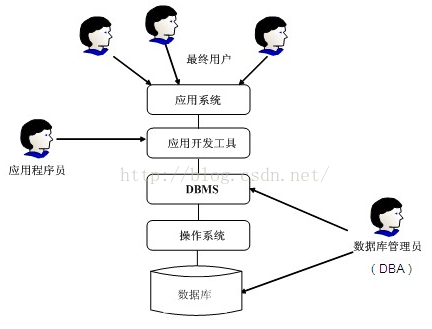
未定义该层的协议，保持原有的物理层和数据链路层

### 数据库

**1、数据库连接池的工作机制：**连接池的实现是以空间换时间

J2EE服务器启动时会建立一定数量的池连接，并一直维持不少于此数目的池连接。客

户端程序需要连接时，池驱动程序会返回一个未使用的池连接并将其表记为忙。如果当前没有空闲连接，池驱动程序就新建一定数量的连接，新建连接的数量有配置参数决定。当使用的池连接调用完成后，池驱动程序将此连接表记为空闲，其他调用就可以使用这个连接

**2、**[**数据库**](http://lib.csdn.net/base/mysql)**即存储数据的仓库**

数据库系统（DataBase System，DBS）一般由数据库、数据库管理系统（DBMS）、应用系统、数据库管理员和用户构成。DBMS是数据库系统的基础和核心

**3、数据模型**由数据结构、数据操作、完整性约束3部分组成，

数据结构：对系统静态特征的描述，包括数据的类型、内容、性质、数据间的相互关系；

数据操作：对系统动态特征的描述，对数据库各种对象实例的操作；

完整性约束：完整性规则的集合，定义了给定的数据模型中数据及其联系的所具有的

制约和依存关系。

**常见的数据库数据模型：**

层次模型是一种用树形结构描述实体及其之间关系的数据模型。

在这种结构中，每一个记录类型都是用节点表示，记录类型之间的联系则用结点

之间的有向线段来表示。每一个双亲结点可以有多个子节点但是每一个子节点只能有一个双亲结点。这种结构决定了采用层次模型作为数据组织方式的层次数据库系统只能处理一对多的实体联系。

网状模型允许一个结点可以同时拥有多个双亲结点和子节点。因而同层次模型相

比，网状结构更具有普遍性，能够直接地描述现实世界的实体。也可以认为层次模型是网状模型的一个特例。

关系模型是采用二维表格结构表达实体类型及实体间联系的数据模型，它的基本

假定是所有数据都表示为数学上的关系。关系模型数据结构简单、清晰、具有很高的数据独立性，因此是目前主流的数据库数据模型。

**关系模型的基本术语**共有十三个，它们分别是：

1、关系：一个关系对应着一个二维表，二维表就是关系名。

2、属性和值域：在二维表中的列，称为属性。属性的个数称为关系的元或

度。列的值称为属性值；属性值的取值范围为值域。

3、关系模式：在二维表中的行定义，即对关系的描述称为关系模式

4、元组：在二维表中的一行，称为一个元组。

5、分量：元组中的一个属性值

6、键或者码：如果在一个关系中存在这样的一个属性，使得在该关系的任何

一个关系状态中的两个元组，在该属性上的值的组合都不同，即这些属性的值都能够用来唯一标识该关系的元组，则称这些属性为该关系的键或者码。

7、超键或者超码：如果在关系的一个键中移去某个属性，它仍然是这个关系

的键，则称这样的键为关系的超键或者超码。

8、候选键或者候选码：如果在关系的一个键中不能移去任何一个属性，否则

它就不是这个关系的键，则称这个被指定的候选键为该关系的候选键或者候选码。

9、主键或者主码：在一个关系的若干候选键中指定一个用来唯一标识该关系

的元组，则称这个被指定的候选键为该关系的主键或者主码。

10、全键或者全码：一个关系模式中的所有属性的集合。

11、主属性和非主属性：关系中包含在任何一个候选键中的属性称为主属

性，不包含在任何一个候选键中的属性为非主属性。

12、外键或者外码：关系中的某个属性虽然不是这个关系的主键，或者只是

主键的，但它却是另外一个关系的主键时，则称之为外键或者外码。

13、参照关系与被参照关系：以外键相互联系的两个关系，可以相互转化。

**关系数据库的规范化**为了解决数据库中数据的插入、删除、修改异常等问题的一

组规则。分为5个等级：第一范式(1NF)~第五范式(5NF)，NF 指Normal Form

* 1. 第一范式(1NF)：每一个列（属性）只有一个，没有重复。同一列中不能

有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。

* 1. 第二范式(1NF)：数据表里的所有数据都要和该数据表的主键有完全依赖

系；如果有哪些数据只和主键的一部份有关的话，它就不符合第二范式。同时可以得出：如果一个数据表的主键只有单一一个字段的话，它就一定符合第二范式(前提是该数据表符合第一范式)。

**不符合第二范式的例子**

|  |  |  |  |
| --- | --- | --- | --- |
| 货物类型 | 货物ID | 货物名称 | 注意事项 |
| 瓷碗 | 1 | 白色瓷碗 | 易碎品 |
| 瓷碗 | 2 | 青花瓷碗 | 易碎品 |
| 瓷碗 | 3 | 雕花瓷碗 | 易碎品 |
| 三合板 | 1 | 普通三合板 | 易燃物品，注意防火 |

在该表中有两个主键，即（货物类型，货物ID），货物名称字段完全依

赖于这两个主键，换句话说，货物的名称完全是取决于这两个主键的值的。但”注意事项“这一列，仅依赖于一个主键”货物类型“。简单地说，第二范式要求每个非键属性完全依赖于主键。

那么，既然表中存在一个对主键不是完全依赖的字段，那么我们就可以

确定，该表不符合第二范式。

**符合第二范式的例子**

|  |  |  |  |
| --- | --- | --- | --- |
| 货物类型 | 货物ID | 货物名称 | 注意事项 |
| 瓷碗 | 1 | 白色瓷碗 | 易碎品 |
| 瓷碗 | 2 | 青花瓷碗 | 易碎品 |
| 瓷碗 | 3 | 雕花瓷碗 | 易碎品 |
| 三合板 | 1 | 普通三合板 | 易燃物品，注意防火 |

在该表中的主键依然是（货物类型、货物ID），非主键字段“货物名

称”，完全依赖于这两个主键，那么我们就可以说，该表是符合数据库第二范式的。

* 1. 第三范式(3NF)：要求一个数据库表中不包含已在其它表中已包含的非主关

键字信息。例如，存在一个部门信息表，其中每个部门有部门编号（dept\_id）、部门名称、部门简介等信息。那么在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余。简而言之，第三范式就是属性不依赖于其它非主属性。 满足第三范式（3NF）必须先满足第二范式（2NF）。

一般把数据规范到第三范式标准就可以满足需要了。

**4、实体**是指客观存在并可相互区别的事物，可以是具体的人事物，也可以是抽象的概念、联系。

**实体间有3种关系**：一对一关系、1对多关系、多对多关系

**5、数据库3级模式结构：**面向用户或应用程序员的用户级、面向建立和维护数据库人员的概念级、面向系统程序员的物理级。

模式（逻辑、概念模式）、内模式均只有一个；外模式在一个数据库中可以有多个。

**模式（逻辑、概念模式）**：数据库中全体数据的逻辑结构和特征的描述，是所有

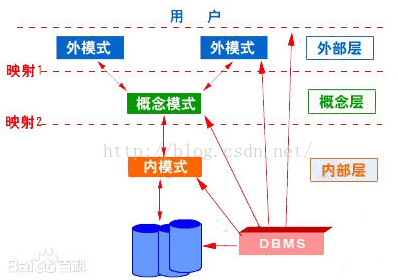
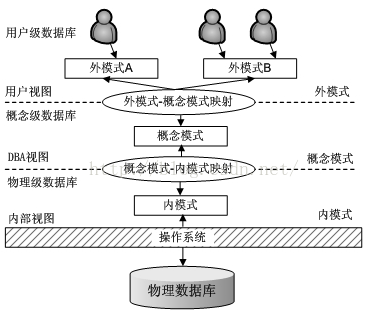
用户的公共数据视图；

**内模式（存储模式）：**数据物理结构和存储方式的描述，是数据在数据库内部的

表示方式。

**外模式（用户模式）：**是模式的子集，数据库用户（应用程序员和最终用户）能

够看见和使用的局部数据的逻辑结构和特征的描述，是数据库用户的数据视图。



## SpringMvc

**1、对 MVC 的理解**

MVC（Model-View-Controller），即把一个应用的输入、处理、输出流程按照Model、View、Controller的方式进行分离，分成三个层——模型层、视图层、控制层

视图(View)：代表用户交互界面，视图上数据的采集和处理，以及用户的请求

注：不包括在视图上的业务流程的处理

模型(Model)：业务流程/状态的处理以及业务规则的制定

注：业务流程的处理过程对其它层是黑箱操作，模型接受视图请求的数据，并返回最终的处理结果

控制(Controller)：从用户接收请求, 将模型与视图匹配在一起，共同完成用户的请求

注：不做任何的数据处理

**优点**

1. 耦合性低：模型与控制器和视图相分离，所以很容易改变应用程序的数据层和

业务规则

1. 重用性高：同样的构件能被不同的界面使用，降低了开发、维护接口的技术含

量

1. 部署快（开发周期短）：Java开发人员集中精力于业务逻辑，前端程序员集

中精力于表现形式上

1. 可维护性高：结构分层使应用更易于维护和修改
2. 有利软件工程化管理：每一层不同的应用具有某些相同的特征，有利于通过工

程化、工具化管理程序

**缺点**

1. 不适合小型，中等规模的应用程序：大量时间将MVC应用到规模中小应用程序

通常会得不偿失

1. 增加系统结构和实现的复杂性：对于简单的界面，严格遵循MVC，使模型、视

图与控制器分离，会增加结构的复杂性，并可能产生过多的更新操作，降低运行效率

3、视图与控制器间的过于紧密的连接：视图不能跳过控制器独立重用

4、视图对模型数据的低效率访问：依据模型操作接口的不同，视图可能需要多次

调用才能获得全部显示数据。对未变化数据的不必要的频繁访问，导致操作性能效率低

**2、结合 Struts ，说明在一个 Web 应用如何去使用**

Model：Struts 没有提供 model 类。业务逻辑由 Web 应用程序的开发者以

JavaBean 或 EJB 的形式提供

View ： Struts 提供了 action form 创建 form bean, 用于在 controlle r 和

view 间传输数据。Struts 提供了自定义 JSP 标签库，辅助开发者用 JSP 创建交互式的以表单为基础的应用程序 ，应用程序资源文件保留了一些文本常量和错误消息，可转变为其它语言，可用于 JSP 中

Controller ： Struts 提供了一个核心的控制器 ActionServlet ，通过

ActionServlet 核心控制器来调用自定义控制器 Action 。自定义 Action 需要符合 Struts 的自定义 Action 规范，还需要在 struts-config.xml 的配置文件中进行配置，

接收 JSP 输入字段形成 Actionform ，才可以调用一个 Action 控制器。 Action 控制器中提供了 model 的逻辑接口

**3、Spring MVC 和 Struts2 对比**

 1、Struts2是类级别的拦截， 一个类对应一个request上下文，SpringMVC是方法级别的拦截，一个方法对应一个request上下文，而方法同时又跟一个url对应

 2、SpringMVC的方法之间基本上独立的，独享request response数据，请求数据通过参数获取，处理结果

通过ModelMap交回给框架，方法之间不共享变量，而Struts2搞的就比较乱，虽然方法 之间也是独立的，但其所有Action变量是共享的，这不会影响程序运行，却给我们编码 读程序时带来麻烦，每次来了请求就创建一个Action，一个Action对象对应一个request上下文

 3、Struts2需要针对每个request进行封装，把request，session等servlet生命周期的变量封装成一个

一个Map，供给每个Action使用，并保证线程安全，所以在原则上，是比较耗费内存的

  4、SpringMVC集成了Ajax，使用非常方便，只需一个注解@ResponseBody就可以实现，然后直接返回

响应文本即可，而Struts2拦截器集成了Ajax，在Action中处理时一般必须安装插件或者自己写代码集成进去，使用起来也相对不方便。

  5、springmvc面向方法开发的（更接近service接口的开发方式），struts2面向类开发。

  6、springmvc可以单例开发，struts2只能是多例开发

## WEB

1. **请写出 JSP 的几个隐含内置 对象、组件**

|  |  |  |  |
| --- | --- | --- | --- |
| JSP 中隐含内置对象 | | | 与ASP的6种组件对应 |
| 名称 | 类型 | 注释和范围 | 含义 |
| request | javax.servlet.http.HttpServletRequest | request | 用户端请求，包含来自 GET/POST 请求的参数 |
| response | javax.servlet.http.HttpServletResponse | response | 网页传回用户端的回应 |
| page | javax.lang.Object | page | JSP 网页本身 |
| Exception | java.lang.Throwable | page | 针对错误网页，未捕捉的例外 |
| pageContext | javax.servlet.jsp.PageContext | page | 管理网页的属性 |
| session | javax.servlet.http.HttpSession | session | 与请求有关的会话期 |
| application | javax.servlet.ServletContext | ServletContext | servlet正在执行的内容 |
| out | javax.servlet.jsp.JspWriter Output | Stream | 用来传送回应的输出 |
| config | javax.servlet.ServletConf ig | ServletConfig | Servlet的构架部件 |

## Computing

**1、什么是阿里云，需要知道云是做什么用的（对资源的管理，快速生成）；**

1、阿里云： 数据、网站、产品统一管理分配的多元化平台

2、云是做什么的： 以高效快速方式实现资源共享、虚拟平台构建。

**2、项目中遇到了哪些问题**

1、多线程并发遍历读取 HashMap<>() 时导致数据重复读取

2、前端请求后台数据丢失

3、系统在高配置服务上运行依然卡顿

你是怎么解决的

1、将线程方法封装为对象类，通过构造方法传参

2、tomcat 中 server.xml 配置 maxPostSize 最大传输字节

3、调整 jvm 内存，服务器配置 tomcat/bin/catalina.sh 的 JAVA\_OPTS 参数

做这个项目最大的收获是什么

1、学习到 OSI 七层参考模型，深入了解 tcp/ip4 层协议

2、更清晰了解一个系统的架构

3、在独立负责这个项目的时候，发现自己需要提升的方面。（业务的熟悉和自主

分析、技术的逻辑严谨、可能存在隐患的预测能力）

## Java

**1、什么是 OOP** ：Object-Oriented Programming ，面向对象编程

OOP 相对于面向过程编程有哪些优点

1、OOP 关注对象和角色，也就是事物的本质

2、OOP 把客观世界中的对象抽象成对应的类

3、通过类构造实例；

4、通过依赖、继承、实现等形式建立对象间的通信关系

5、OOP 易于扩展，增加或改变业务的功能，无需大幅改动改变源代码

6、OOP 易于建模，可以很好地理解和建立起计算机中的抽象模型

**2、Overload 和 Override 的区别**

方法的重写 Overriding 和重载 Overloading 是 Java 多态性的不同表现

重写 Overriding是父类与子类之间多态性的一种表现，重载 Overloading 是一个类中多态性的一种表现

Overriding：子类中定义与其父类有相同的方法名和参数

Overloading： 方法名相同，不同的参数个数 或 不同的参数类型。Overloaded 的可以改变返回值的类型

**3、谈谈 HashMap 和 Hashtable 的区别**

都属于 Map 接口的类，实现了将惟一键映射到特定的值上

1. HashMap 轻量级集合（线程不安全的，效率高的），Hashtable 重量级集合（线程安

全的，效率低的）

1. HashMap 类没有分类或者排序。允许一个 null 键、多个 null 值；Hashtable 类似

于 HashMap ，不允许 null 键、 null 值。它也比 HashMap 慢 ，因为它是同步的

**4、请问 abstract class 和 interface 有什么区别**

abstract class：含有abstract修饰符的class为抽象类

1、abstract 类不能创建的实例对象。

2、含有abstract方法的类必须定义为abstract class

3、abstract class类中的方法不必是抽象的

4、abstract class类中抽象方法必须在子类中全部实现，所以，不能有抽象构造方法、抽象静态方法

注：如果的子类没有实现抽象父类中的所有抽象方法，那么子类也必须定义为abstract

类型

interface：接口（interface）可以说成是抽象类的一种特例，接口中的所有方法都必

须是抽象的

注：接口中的方法定义默认为public abstract类型，接口中的成员变量类型默认为

public static final

1、接口没有任何的实现 ；抽象类它可以有部分的实现也可以没有

2、抽象类可以复用、共享部分代码（由子类补充父类没有的功能）

**5、类的核心特性有哪些：**封装性、继承性（inheritance ）、多态性

封装性：类的成员提供公有、缺省、保护和私有等多级访问权限，隐藏类中的变量、方法的实现细节

继承性：使一个新类自动拥有被继承类（父类）的全部成员

多态性：提供类中方法执行的多样性，多态性有两种表现形式：重载、覆盖（重写）

**6、请问类与对象有什么区别**

类：某一种事物的一般性的集合体，是相同或相似的各个事物共同特性的一种抽象

对象：对象（ Object ）是类的实例（ instance ）。对象与类的关系就像变量与数据类型的关系一样

1. **请说出几个常用的异常类**

|  |  |
| --- | --- |
| **异常** | **含义** |
| NullpointException | 空指针异常 |
| ClassNotFoundException | 类找不到异常 |
| ClassCastException | 类型转换异常 |
| IllegalArgumentException | 非法参数异常 |
| ArithmeticException | 算术异常 |
| NumberFormatException | 数据格式异常 |
| IndexOutOfBoundsException | 数组下标越界异常 |
| IllegalStateException | 非法语句异常 |

**8、 Java 面向对象的四大特点：**封装、抽象、继承和多态

抽象：抽象就是将一类实体的共同特性抽象出来，封装在一个抽象类中

注：抽象类不能实例化

① 具体的类才可以实例化

② 抽象类实例化后没有任何可以实现的，所以是无意义的

**9、 public, private, protected ， default 访问权限（可见性）**



public 全局可见

protected 继承体系结构之间可见

default （或不写）：同包可见

private 本类可见

**10、Java Servlet API 中 forward() 与 redirect() 的区别**

forward() 容器中控制权的转向，在客户端浏览器地址栏中不会显示出转向后的地址；

redirect() 完全的跳转，浏览器将会得到跳转的地址，并重新发送请求链接。这样，从浏览器的地址栏中可以看到跳转后的链接地址

前者更加高效，在前者可以满足需要时，尽量使用 forward() 方法，并且，这样也有助于隐藏实际的链接。在有些情况下，比如，需要跳转到一个其它服务器上的资源，则必须使用 sendRedirect() 方法

**11、写一个简单的 C/S 结构程序， Java 的通信编程，用 JAVA SOCKET编程，读服务器几个字符**

Server 端程序 :

public static void main**(**String**[]** args**)** **throws** IOException**{**

Socket socket **=** **null;** BufferedReader in **=** **null;** PrintWriter out **=** **null;**

**try** **{**

ServerSocket ss **=** **new** ServerSocket**(**10000**);** //1、指定绑定的端口，并启动端口监听

**while** **(true)** **{**

socket **=** ss**.**accept**();** //2、accept()方法开始监听并阻塞等待客户请求，等待客户端的连接

System**.**out**.**println**(**"A client come in!IP:" **+** socket**.**toString**());**

//3、收到请求产生一个Socket对象，并继续执行

in **=** **new** BufferedReader**(new** InputStreamReader**(**socket**.**getInputStream**()));**

System**.**out**.**println**(**"Cleint send is :" **+** in**.**readLine**());**

**}**

**}** **catch** **(**IOException e**)** **{** System**.**err**.**println**(**"Server Wrong"**); }finally** **{**

**if(**out **!=** **null)** out**.**close**();**

**if(**in **!=** **null)** in**.**close**();**

**if(**socket **!=** **null)** socket**.**close**();**

**}**

**}**

Client 端程序 :

public static void main**(**String**[]** args**)** **throws** IOException **{**

Socket socket **=** **null;** BufferedReader in **=** **null;** PrintWriter out **=** **null;**

**try** **{**

socket **=** **new** Socket**(**"127.0.0.1"**,** 10000**);**//1、创建客户端Socket，指定服务器地址和端口

System**.**out**.**println**(**"Please enter some Character:"**);**

//2、 由系统标准输入设备构造BufferedReader对象

BufferedReader line **=** **new** BufferedReader**(new** InputStreamReader**(**System**.**in**));**

out **=** **new** PrintWriter**(**socket**.**getOutputStream**());**

out**.**println**(**line**.**readLine**());**

**}** **catch** **(**IOException e**)** **{** System**.**err**.**println**(**"Client Wrong"**); }finally{**

**if(**out **!=** **null)** out**.**close**();**

**if(**in **!=** **null)** in**.**close**();**

**if(**socket **!=** **null)** socket**.**close**();**

**}**

**}**

## JavaThread

**1、多线程什么时候使用：**程序需要同时做一个以上的任务时

**2、多线程使用场景：**数据库连接池、web线程池实现多线程单例serlvet、多任务并行

**3、实现多线程三种方式：**

继承Thread重写run()，实现runnable实现run()方法，使用Callable、Future实现接口；

注：使用场景结合单继承多实现考虑Thread和runnable，Callable可以返回结果

**4、进程和多线程的区别：**

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,资源分配和调度的一个独立单位

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位

a) 进程之间不能共享数据，线程可以

b) 系统创建进程需要为进程重新分配资源，线程代价较小

c)  java语言内置多线程功能支持，简化多线程编程

d）地址空间:进程内的一个执行单元;进程至少有一个线程;它们共享进程的地址空间;而进程有自己独立的地址空间;

e）资源拥有:进程是资源分配和拥有的单位,同一个进程内的线程共享进程的资源

注：二者均可并发执行

**5、wait()方法和sleep()方法有什么不同：**

a) wait()会释放锁；sleep()一直持有线程锁，通常用于暂停，仍占用cpu

b) sleep是Thread类的静态方法，wait是Object方法。

c) wait、notify、notifyAll只能在同步方法、同步块里面使用，sleep可以在任何地方使用

d) sleep必须捕获异常，而 wait，notify、notifyAll不需要捕获异常

**6、T1、T2、T3三个线程，怎样保证T2在T1执行完后执行，T3在T2执行完后执行？**

join 或者 Future的get结果

注：join方法的功能是使异步执行的线程变成同步执行。即调用线程实例的start方法后，该方法会立即返回，使用join方法后，直到这个线程退出，程序才会往下执行。

**7、在Java中Lock接口比synchronized块的优势是什么？**

1、lock接口在多线程和并发编程中最大的优势是它们为读和写分别提供了锁，类似ConcurrentHashMap这样的高性能数据结构和有条件的阻塞；

读写锁ReadWriteLock拥有更加强大的功能，它可细分为读锁和解锁。

读锁可以允许多个进行读操作的线程同时进入，但不允许写进程进入；写锁只允许一个写进程进入，在这期间任何进程都不能再进入

注：因为每个读写锁都有挂锁和解锁，最好将每一对挂锁和解锁操作都用try、finally来套入中间的代码，这样就会防止因异常的发生而造成死锁得情况。

2、相比synchornized有更高的伸缩性，ReentrantLock的不公平锁比synchornized公平锁的性能高

3、在资源竞争不是很激烈的情况使用synchronized性能高于ReentrantLock。但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍。而ReentrantLock确还能维持常态。

4、比synchronized的功能多

你需要实现一个高效的缓存，它允许多个用户读，但只允许一个用户写，以此来保持它的完整性，你会怎样去实现它？

/\*\*使用ReentrantLock 分别控制读、写操作\*/

class ReadWriteLockTest **{**

public static void main**(**String**[]** args**)** **{**

final TheData myData **=** **new** TheData**();** // 这是各线程的共享数据

**for** **(**int i **=** 0**;** i **<** 3**;** i**++)** **{** // 开启3个读线程

**new** Thread**(new** Runnable**()** **{**

public void run**()** **{** myData**.**get**();}}).**start**();**

**}**

**for** **(**int i **=** 0**;** i **<** 3**;** i**++)** **{** // 开启3个写线程

**new** Thread**(new** Runnable**()** **{**

public void run**()** **{** myData**.**put**(new** Random**());** **}}).**start**();**

**}**

**}**

**}**

class TheData **{**

private Object data **=** **null;**

private ReadWriteLock rwl **=** **new** ReentrantReadWriteLock**();**

public void get**()** **{** rwl**.**readLock**().**lock**();** // 读锁开启，读线程均可进入

**try** **{** // 用try finally来防止因异常而造成的死锁

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()** **+** "have read date" **+** data**);**

**}** **finally** **{** rwl**.**readLock**().**unlock**();** // 读锁解锁

**}**

**}**

public void put**(**Object data**)** **{** rwl**.**writeLock**().**lock**();** // 写锁开启，这时只有一个写线程进入

**try** **{ this.**data **=** data**;**

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()** **+** "have write date" **+** data**);**

**}** **finally** **{** rwl**.**writeLock**().**unlock**();** // 写锁解锁

**}**

**}**

**}**

**8、用Java实现阻塞队列 or 生产者——消费者问题**（支持2个及以上附加操作的队列）

1、wait()和notify()

2、new ArrayBlockingQueue<Object>(3); // 有界限阻塞队列，可用于控制并发程序最多3个

public class BigPlate **{** /\*\* 定义一个盘子类，可以放鸡蛋和取鸡蛋 \*/

/\*\* 装鸡蛋的盘子，大小为5 \*/

private static BlockingQueue**<**Object**>** eggs **=** **new** ArrayBlockingQueue**<**Object**>(**3**);**

static class AddThread **extends** Thread **{** /\*\* 放鸡蛋线程 \*/

public void run**()** **{** /\*\* 放鸡蛋 \*/

**try** **{** eggs**.**put**(**0**);** // 向盘子末尾放一个鸡蛋，如果盘子满了，当前线程阻塞

**}** **catch** **(**InterruptedException e**)** **{** e**.**printStackTrace**();** **}**

System**.**out**.**println**(**"放入鸡蛋"**);**// 下面输出有时不准确，因为与put操作不是一个原子操作

**}**

**}**

static class GetThread **extends** Thread **{** /\*\* 取鸡蛋线程 \*/

public void run**()** **{** /\*\* 取鸡蛋 \*/

**try** **{** eggs**.**take**();** // 从盘子开始取一个鸡蛋，如果盘子空了，当前线程阻塞

**}** **catch** **(**InterruptedException e**)** **{** e**.**printStackTrace**();** **}**

System**.**out**.**println**(**"拿到鸡蛋"**);** // 下面输出有时不准确，因为与take操作不是一个原子操作

**}**

**}**

public static void main**(**String**[]** args**)** **{**

**for(**int i **=** 0**;** i **<** 10**;** i**++)** **{** // 先启动10个放鸡蛋线程

**new** Thread**(new** AddThread**()).**start**();** **}**

**for(**int i **=** 0**;** i **<** 10**;** i**++)** **{** // 再启动10个取鸡蛋线程

**new** Thread**(new** GetThread**()).**start**();** **}**

**}**

**}**

**9、用Java编程一个会导致死锁的程序，你将怎么解决？**

1、死锁的四个必要条件

① 互斥条件 ② 请求与保持条件 ③ 非剥夺条件 ④ 循环等待条件

1. 处理死锁的方法

* 忽略该问题，也即鸵鸟算法。当发生了什么问题，不管他，直接跳过，无视它；
* 检测死锁并恢复；
* 资源进行动态分配；
* 破除上面的四种死锁条件之一

**10、什么是原子操作**

定义 1、如果这个操作所处的层(layer)的更高层不能发现其内部实现与结构，那么这个操作是一个原子(atomic)操作

2、原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序不可以被打乱，也不可以被切割而只执行其中的一部分

3、将整个操作视作一个整体是原子性的核心特征

**Java中的原子操作是什么**

jdk1.5的包为java.util.concurrent.atomic这个包里面提供了一组原子类。其基本特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性

即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像锁一样，一直等到该方法执行完成，才由JVM从等待队列中选择另一个线程进入，这只是一种逻辑上的理解。实际上是借助硬件的相关指令来实现的，但不会阻塞线程（synchronized 会把别的等待的线程挂起，或者说只是在硬件级别上阻塞了）

注：原子操作(atomic operation)是不需要synchronized

**11、Java中的volatile关键是什么作用**

volatile修饰的变量，jvm虚拟机保证从主内存加载到线程工作内存的值是最新的

**怎样使用它**

a) 状态标志（布尔状态），还需要某些原子状态转换机制

b) 一次性安全发布，单利模式的双重检查锁定中获取最新的实例对象

c) 独立观察，定期发布观察结果供程序内部使用

d) “volatile bean” 模式，为易变数据的持有者（例如 HttpSession）提供

了容器，但是放入这些容器中的对象必须是线程安全的。

注：JavaBean 的所有数据成员都是 volatile 类型的，并且 getter 和 setter

方法必须非常普通——即不包含约束！

e) 开销较低的“读－写锁”策略

注：锁一次只允许一个线程访问值，volatile 允许多个线程执行读操作

**Java中它跟synchronized方法有什么不同**

Volatile 变量具有 synchronized 的可见性特性，但是不具备原子性。如果读操作远远大于写操作，volatile 变量还可以提供优于锁的性能优势

**12、什么是竞争条件**

1、多个线程或者进程在读写一个共享数据时结果依赖于它们执行的相对[时间](http://baike.baidu.com/item/%E6%97%B6%E9%97%B4/25651)的情形

2、发生在当多个进程或者线程在读写数据时，其最终的的结果依赖于多个进程的指令执行顺序

**怎样发现和解决竞争（会产生互斥量）**

synchronized 和 Lock的ReentrantLock 修饰共享资源

注：线程安全的集合HashMap<>()、StringBuffer、Vector<>()

**13、如何使用thread dump**

在UNIX中 可以使用kill-3 pid >filename 注：tomcat会记录在catalina.out

windows中 可以使用”CTRL+Break”。

**如何分析Thread dump**

1、识别线程的8中状态

a) 死锁，Deadlock（重点关注）

b) 执行中，Runnable

c) 等待资源，Waiting on condition（重点关注）

d) 等待获取监视器，Waiting on monitor entry（重点关注）

e) 暂停，Suspended

f) 对象等待中，Object.wait() 或 TIMED\_WAITING

g) 阻塞，Blocked（重点关注）

h) 停止，Parked

2、16进制线程地址、nid为native线程的id、prio是线程优先级、tid为线程地址、SynchronousQueue 存放线程间移交临时元素的容器

注：通过以上属性结合描述判断线程做出的动作

**14、为什么调用start()方法时会执行run()方法**

调用start()方法时你将创建新的线程，并且执行在run()方法里的代码。

**为什么不能直接调用run()方法**

如果直接调用run()方法，它不会创建新的线程也不会执行调用线程的代码

**15、怎样唤醒一个阻塞的线程**

IO阻塞

1、创建线程时，加一个数量的阈值，超过该值后则不再创建

2、为每个线程设置标志变量标志该线程是否已经束，

3、直接加入线程组去管理

注：因为调用 wait()、sleep()、join()方法而导致的阻塞，理论上可以通过中断线程，并抛出InterruptedException来唤醒它，实际不一定有效

**16、CycliBarriar和CountdownLatch有什么区别**（JDK5中的并发包）

1. CyclicBarrier可以多次使用，CountDownLatch只能用一次（为0后不可变）

2. CyclicBarrier是等待指定数量线程到达再继续处理；Latch是等待指定事件变为指定状态后发生再继续处理，对于CountDownLatch是计数减为0的事件

3. CyclicBarrier等待指定数量任务完成，CountDownLatch等待其他任务完成指定状态的改变再继续

注：CountdownLatch: 一个线程(或者多个)，等待另外N个线程完成某个事情之后才能执行。

CycliBarriar: N个线程相互等待，任何一个线程完成之前，所有的线程都必须等待。

场景：用于等待其他现象，满足一定条件时同时执行

**17、什么是不可变对象**

不可变对象（Immutable object）是一种对象,在被创造之后，状态就不可以被改变

**它对写并发应用有什么帮助**

由于不可更改，并发时不需要其他额外的同步保证，故相比其他的锁同步等方式的并发性能要好

**为什么String是不可变的**

1. 字符串常量池(String pool, String intern pool, String保留池) 是Java

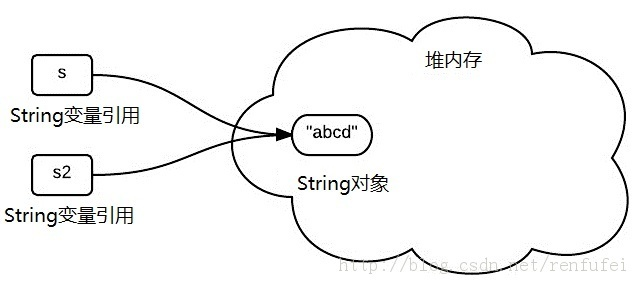
堆内存中一个特殊的存储区域, 当创建一个String对象时,假如此字符串值已经存在于常量池中,则不会创建一个新的对象,而是引用已经存在的对象

1. 允许String对象缓存HashCode：ava中String对象的哈希码被频繁地使用, 比

如在hashMap 等容器中，字符串不变性保证了hash码的唯一性

1. 安全性：被许多的Java类(库)用来当做参数,例如 网络连接地址URL、文件路

径path、反射机制所需要的String参数等, 假若String不是固定不变的,将会引起各种安全隐患



**18、多线程环境中遇到的常见的问题？你是怎么解决它的**

多线程和并发程序中常遇到的有 竞争条件（12题）、死锁（9题）、活锁、饥饿

活锁(livelock)：事物1可以使用资源，但它让其他事物先使用资源；事物2可以使用资源，但它也让其他事物先使用资源，事务之间一直谦让，导致都无法使用资源；活锁有一定几率解开，而死锁（deadlock）是无法解开的。

解决：避免活锁的简单方法是采用先来先服务的策略。当多个事务请求封锁同一数据对象时，封锁子系统按请求封锁的先后次序对事务排队，数据对象上的锁一旦释放就批准申请队列中第一个事务获得锁。

饥饿：如果事务T1封锁了数据R,事务T2又请求封锁R，于是T2等待。T3也请求封锁R，当T1释放了R上的封锁后，系统首先批准了T3的请求，T2仍然等待。然后T4又请求封锁R，当T3释放了R上的封锁之后，系统又批准了T4的请求......T2可能永远等待

解决：公平锁：每一个调用lock()的线程都会进入一个队列，当解锁后，只有队列里的

第一个线程被允许锁住Farlock实例，所有其它的线程都将处于等待状态，直到都处于队列

**19、绿色线程和本地线程区别**

绿色线程执行用户级别的线程，且一次只使用一个OS（操作系统）线程

本地线程用的是OS线程系统，在每个JAVA线程中使用一个OS线程

注：在执行java时，可通过使用-green 或 -native标志来选择所用线程是绿色还是本地。

**20、什么是多线程中的上下文切换**

1、一个进程停止，另一个进程开始活动

2、线程被抢占时，持有资源的线程，资源被抢占后被迫暂停，获得资源的线程活跃

**21、Java中用到的线程调度算法是什么**

java虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用CPU，

如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用CPU。处于运行状态的线程会一直运行，直至它不得不放弃CPU。

**22、Java中什么是线程调度：**按照特定机制为多个线程分配CPU的使用权

**23、线程中怎么处理不可捕捉异常**

1、把线程的错误捕捉到，往上抛

2、通过线程池工厂，把异常捕捉到

**24、什么是线程组：**ThreadGroup线程组表示一个线程的集合

注：线程组也可以包含其他线程组，线程组构成一棵树，在树中，除了初始线程组外，

每个线程组都有一个父线程组

允许线程访问有关自己的线程组的信息，但是不允许它访问有关其线程组的父线程组

或其他任何线程组的信息

**为什么在Java中不推荐使用**：节省频繁创建和销毁线程的开销，提升线程使用效率

**线程组和线程池的区别在哪里**

线程池：创建一定量的线程，批量处理一类任务，任务执行完毕后，线程又可以

执行其他的任务。等不再需要线程的时候，就销毁。省去了频繁创建和销毁线程的麻烦。

线程组：线程执行完一个任务后就会销毁

**25、为什么使用Executor框架比使用应用创建和管理线程好？**

应用程序：以执行任务（task）为基本单位进行管理的，会为每个任务单独创建一个线程来执行

Executor 框架：

1、可以用于异步任务执行，而且支持很多不同类型的任务执行策略。

2、提供了对生命周期的支持和hook（钩子）函数（挂载机制），可以添加如统计收集、应用程序管理机制和监视器等扩展。

3、可以重用已存在的线程，免除创建新的线程，减少线程创建、消亡的开销。同时，在任务到达时，工作线程通常已经存在，提高了响应性。适当的调整线程池的大小，在得到足够多的线程以保持处理器忙碌的同时，还可以防止过多的线程相互竞争资源，导致应用程序在线程管理上耗费过多的资源

**26、在Java中Executor和Executors的区别**

Executor是接口：是用来执行 Runnable 任务的；它只定义一个方法- execute(Runnable command)

Executors是类：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

**27、Windows上查找哪个线程使用的CPU占用率最高：**任务管理器查看进程

**Linux上查找哪个线程使用的CPU占用率最高：**

1、top查看进程

2、shift+h查看CPU占用率最高的

3、java详细信息查看 jstack 进程id（jps查看）

jstack 进程id > ps.txt

jstack -l 进程id （窗口打印）

## Oracle

1. **对数据库SQL2005、ORACLE熟悉吗**  
      SQL2005是微软公司的数据库产品。是一个[RDBMS](#RMDBS)数据库，一般应用在一些中型数据库的应用，不能跨平台。

ORACLE是ORACLE公司的数据产品，支持分布式布暑，支持多用户，跨平台，数据安全完整性控制性能优越，是一个ORDBMS，一般用在大型公司。

1. **如何只显示重复数据，或不显示重复数据**

显示重复： **select** **\*** **from** tablename **group** **by** id **having** **count(\*)>**1**;**

不显示重复：**select** **\*** **from** tablename **group** **by** id **having** **count(\*)=**1**;**

**3、什么是数据库的映射：** 将数据库的表与字段对应到模型层，类名与属性的过程

**5、写分页有哪些方法，你一般用什么方法？用SQL语句写一个分页？**

**如何用存储过程写分页？**

[MySQL分页](#MySQL分页)

SQLSERVER使用TOP分页，使用TOP：

**select** top 20**,**n**.\*** **from** n **minus** **select** top 10**,**m**.\*** **from** m

ORACLE中用ROWNUM，或分析函数ROW\_NUMBER over(order by COLUMN\_NAME)

  使用分析函数：

**select** **\*** **from** **(select** n**.\*,row\_number()** over**(order** **by** columnname**)** num **from** n**) where** num**>=**10 **and** num **<=**20**;**

使用过程时，只要将分页的范围用两个参数就可以实现。在ORACLE中，要将过程封装在包里，还要用动态游标变量才能实现数据集的返回。

**6、ORACLE中左连接与右连接**  
*左连接：LEFT JOIN  右连接：RIGHT JOIN*  
*select n.column,m.column from tablename1 n left join tablename2 m*  
*on n.columnname=m.columnname*  
*用WHERE实现：*  
*select n.column,m.column from tablename1 n， tablename2 m*  
*where n.columnname(+)=m.columnname*

**7、什么是反射、序列化、反序列化？事务有几种级别？**  
    反射是在程序运行时动态访问DDL的一种方式。序列化是将对象对二进制、XML等方式直接向文件的存储。反序列化是将存储到文件的对象取出的过程。事务的级别的三种：页面级、应用程序级、数据库级。

**8、数据测试如何测试？**  
   在PLSQL里对过程或函数可能通过专用的测试工具，通过对

**9、用事务的时候，如果在业务逻辑层中，调用数据库访问层中的方法，访问层中有很多类，类又有很多方法，每个方法都要实现，那么如何处理？**  
    通用数据访问层的实现

**10、什么时候会用到触发器**  
   A安全管理、B日志管理、C复杂业务逻辑实现

**11、如何在数据库中显示树控制？**  
   用父ID与子ID来实现

**12、如何实现数据库的优化？**  
    A、调整数据结构的设计。这一部分在开发信息系统之前完成，程序员需要考虑是否使用ORACLE数据库的分区功能，对于经常访问的数据库表是否需要建立索引等。

    B、调整应用程序结构设计。这一部分也是在开发信息系统之前完成，程序员在这一步需要考虑应用程序使用什么样的体系结构，是使用传统的Client/Server两层体系结构，还是使用Browser/Web/Database的三层体系结构。不同的应用程序体系结构要求的数据库资源是不同的。

    C、调整数据库SQL语句。应用程序的执行最终将归结为数据库中的SQL语句执行，因此SQL语句的执行效率最终决定了ORACLE数据库的性能。ORACLE公司推荐使用ORACLE语句优化器（Oracle Optimizer）和行锁管理器（row-level manager）来调整优化SQL语句。

    D、调整服务器内存分配。内存分配是在信息系统运行过程中优化配置的，数据库管理员可以根据数据库运行状况调整数据库系统全局区（SGA区）的数据缓冲区、日志缓冲区和共享池的大小；还可以调整程序全局区（PGA区）的大小。需要注意的是，SGA区不是越大越好，SGA区过大会占用操作系统使用的内存而引起虚拟内存的页面交换，这样反而会降低系统。

    E、调整硬盘I/O，这一步是在信息系统开发之前完成的。数据库管理员可以将组成同一个表空间的数据文件放在不同的硬盘上，做到硬盘之间I/O负载均衡。

    F、调整操作系统参数，例如：运行在UNIX操作系统上的ORACLE数据库，可以调整UNIX数据缓冲池的大小，每个进程所能使用的内存大小等参数。