学习笔记

# Java基础学习笔记

## 1.1多线程理解

### 1.1.1 初步理解进程，线程：

进程是正在执行的程序，线程就是进程中正在执行的一段代码。线程是进程的一个执行顺序。线程是进程的一个执行单元。

执行顺序是不是就是类似一个方法的代码 在顺序执行 或者是按照条件顺序执行的那个顺序。 进程是正在执行中的程序，线程就是进程中正在执行的一段代码。 线程就是一个的进程的一条执行路径

多线程就是一个进程中存在多条的执行路径。(生活中最常见的例子就是：迅雷下载，他并不是将一段资源从头到尾顺序下载，而是将一个资源分成几份，每一份都有一个下载程序去下载，最后再将他们拼接到一起。这样就加快了速度)

多线程出现的意义就是让我们的程序出现同时运行的效果。

1。单进程单线程：一个人在一个桌子上吃菜。  
2。单进程多线程：多个人在同一个桌子上一起吃菜。  
3。多进程单线程：多个人每个人在自己的桌子上吃菜。

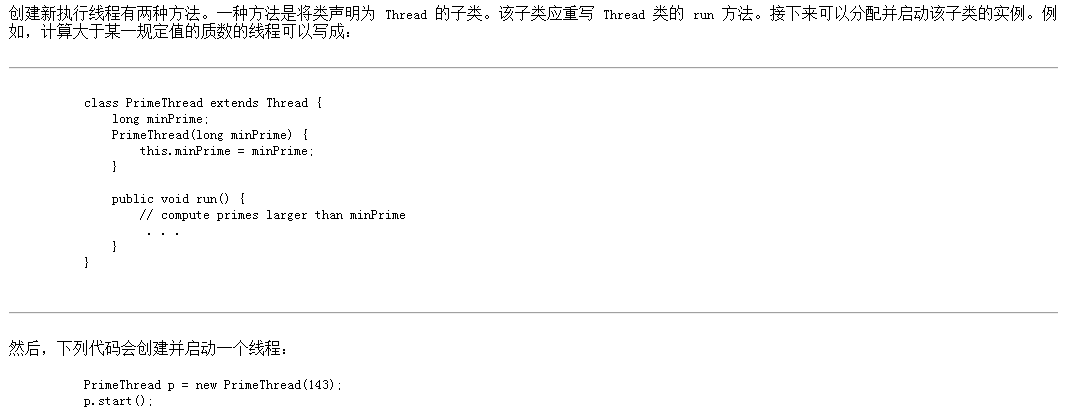
多线程的问题是多个人同时吃一道菜的时候容易发生争抢，例如两个人同时夹一个菜，一个人刚伸出筷子，结果伸到的时候已经被夹走菜了。。。此时就必须等一个人夹一口之后，在还给另外一个人夹菜，也就是说资源共享就会发生冲突争抢。

1。对于 Windows 系统来说，【开桌子】的开销很大，因此 Windows 鼓励大家在一个桌子上吃菜。因此 Windows 多线程学习重点是要大量面对资源争抢与同步方面的问题。

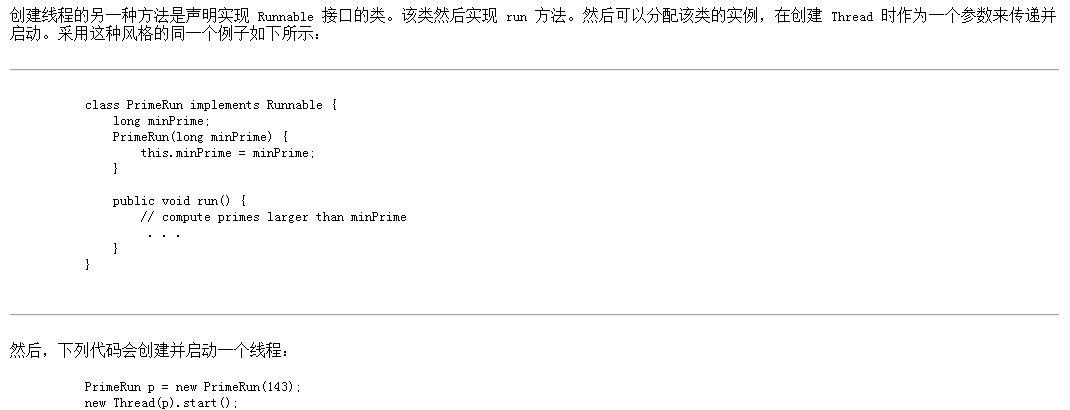
2。对于 Linux 系统来说，【开桌子】的开销很小，因此 Linux 鼓励大家尽量每个人都开自己的桌子吃菜。这带来新的问题是：坐在两张不同的桌子上，说话不方便。因此，Linux 下的学习重点大家要学习进程间通讯的方法。

### 1.1.2 自己创建线程

线程这个东西别Java封装成了对象。(线程的真正创建不是由我们创建，而是：系统帮助我们创建进程，也帮助我们创建线程)

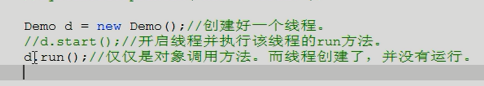


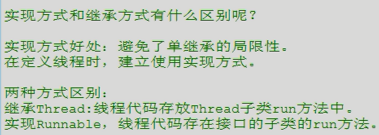
这里的start()方法只是主线程开启了一个控制单元，多了一条执行路径。（将run{}中的代码编程一个线程，让他的线程状态变成就绪态，并不是让他立即执行，他的执行要看他什么时候抢到cpu的执行权。）



线程定义的目的就是要让线程去执行一些代码，所以Java在定义线程的时候就定义了存储我们需要线程去帮我们执行的代码 的位置，就是run{}。（主线程定义的要执行代码的存储空间就是main方法）

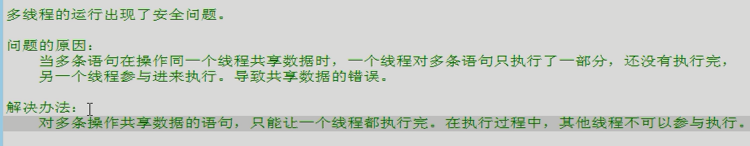
注意：

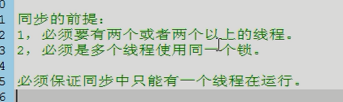




(使用实现方式，他还将线程中的资源独立出来了，因为他的调用方式决定了他在堆内存只建立了一次，那么线程中的成员属性也只new了一次，这样就变向的让资源共享了）。

多线程存在线程安全问题。(对于共享的资源操作需要使用synchronized 同步关键字)





### 1.1.3 多线程之间的通信

**什么是多线程之间的通信**：

多线程之间的通信：就是多个线程在操作同一资源，但是操作的动作是不同的。

(多线程通信：顾名思义就是一个线程在进行某些操作的时候需要和其他线程进行交流告诉其他线程一些信息，让其他线程做一些操作)

**为什么要进行多线程通信**：

我们在操作共享资源的时候，比如对一个共享资源进行(一个线程)输入和(一个线程)获取。这是不同的操作，如果没有进行线程通信，那么会存在下列弊端，资源没有了还在获取，或者是放满了还在放。如果线程之间可以进行线程通信，那么如果没有资源了告诉获取线程停止获取，如果资源没有了，告诉输入线程输入

线程之间进行通信，可以有效的避免死锁（等安全问题），和提高资源的利用率

**怎样进行多线程的通信**：

多线程的实现主要依靠Object类方法wait(),notify(),notifyAll()；

线程执行wait()后，就放弃了运行资格，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。

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。

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。

wait和sleep区别：从执行权和锁上来分析这2个方法

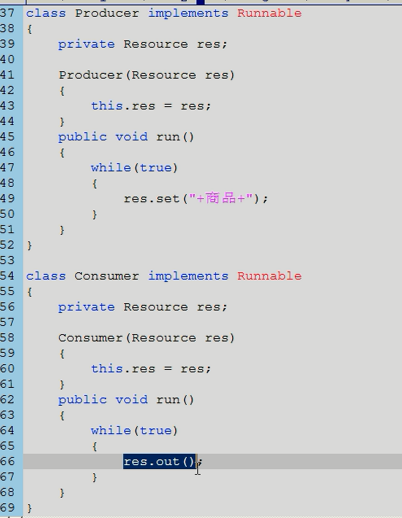
wait()：可以指定时间也可以不指定时间，不指定时间时，只能由对应的notify()或notifyAll()来唤醒。

sleep()：必须指定时间，时间到自动从冻结状态转入运行状态或临时阻塞状态。

wait()：线程会**释放执行权，并释放锁**。

sleep()：线程**会释放执行权，但是并不释放锁**。

示例：

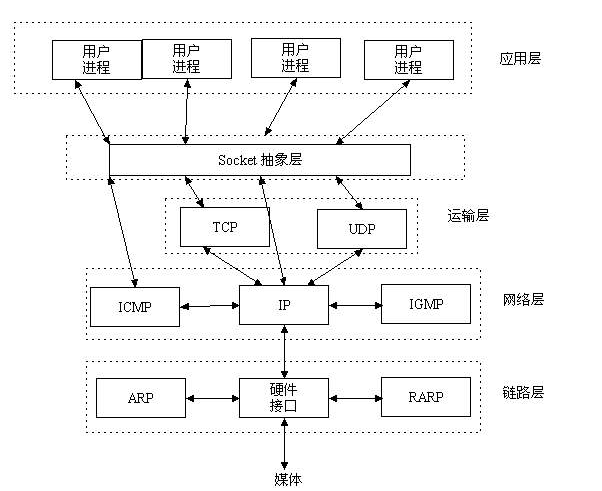


## 1.2 socket(网络编程)

**什么是socket(套接字)**：

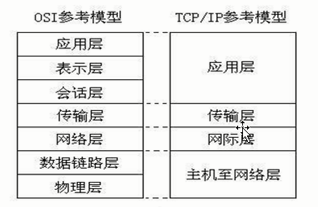
Socket是Java的网络编程。(简而言之就是Java对于 主机连入网络 的编程。 我认为Socket的实质就是本地主机和网络上其他地方的主机之间的通信，我们知道两个进城之间进行通信的前提是获得对方的**唯一**标识。我们知道IP层的ip地址可以唯一标示主机，而TCP层协议和端口号可以唯一标示主机的一个进程，这样我们可以利用ip地址＋协议＋端口号唯一标示网络中的一个进程。)

TCP/IP传输控制协议/网间协议，定义了主机如何连入因特网及数据如何再它们之间传输的标准。



socket是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。

Socket是基于网络模型中的传输层和网际层（Javaweb是基于应用层，Socket是相对于底层的，Javaweb 的本质就是基于Socket的。传输层主要使用的是TCP，UDP，网际层主要使用的是IP）



**为什么需要Socket**：

我们以前做的都是在本地进行操作，所做的都是单机版，他无法满足我们想和网络上其他主机进行交流的需求。

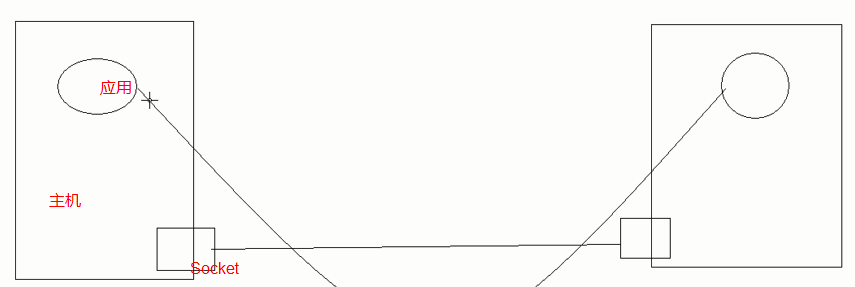
**如何使用Socket**：



socket起源于UNIX，在Unix一切皆文件哲学的思想下，socket是一种"打开—读/写—关闭"模式的实现，服务器和客户端各自维护一个"文件"，在建立连接打开后，可以向自己文件写入内容供对方读取或者读取对方内容，通讯结束时关闭文件。 数据通信的原理就是数据传输。

**对于Socket的简单理解**：

Socket是网络上两台主机之间数据交流的码头。



Socket是通信端点。但是每个传输协议不一样。每个传输协议都有自己的一套建立通信端点的规则。

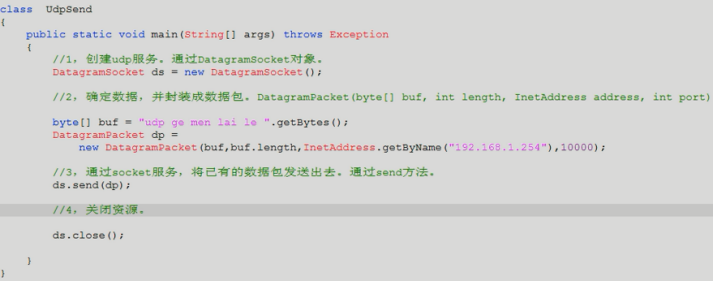
UDP：



要进行通讯就必须有两个端点，发送端和接收端(这两个端点都是独立的程序)。

Demo：

发送端：



接收端：



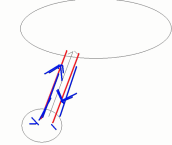
接收端的receive方法是阻塞机制的，在没有接收到数据的时候，我们他会阻塞，接收到数据之后会notify.

TCP传输：



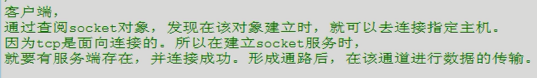
和UDP连接之间的传输不同的是：TCP是面向连接的需要在一开始的时候就建立连接(所以在客户端启动之前，服务端就必须先启动起来)。

当Socket一建立，这时在客户端和服务端中就已经建立了一个通路（就已经有了一个输入流和一个输出流，可以直接从里面获取，不需要我们自己再去new），在这个通路中数据有来有往。

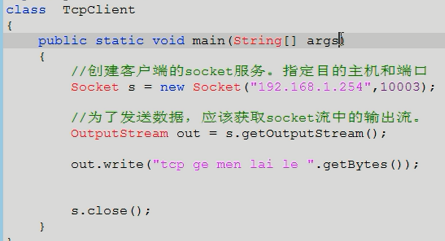




客户端：

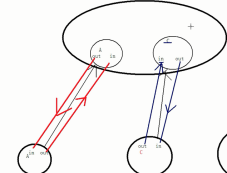


Demo:

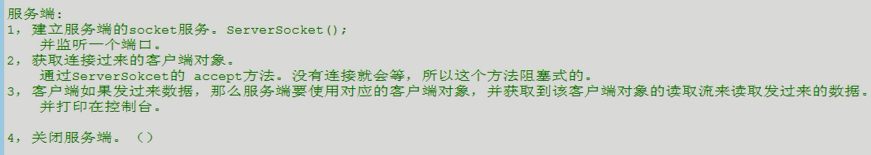


服务端：

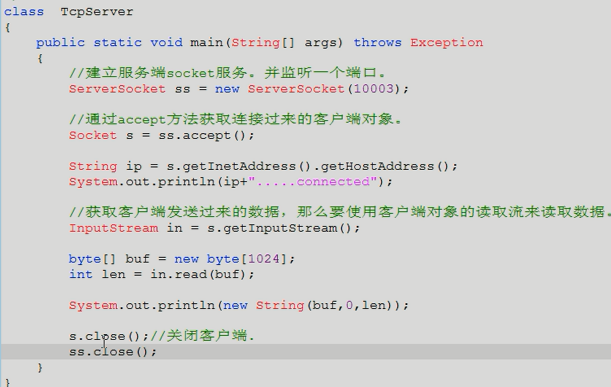
服务端为了避免将连入的A客户端的内容写到连入的C客户端中，所以服务端会拿到连入的客户端中的对象。



服务端建立serviceSocket(),并监听一个端口，来等待连接。

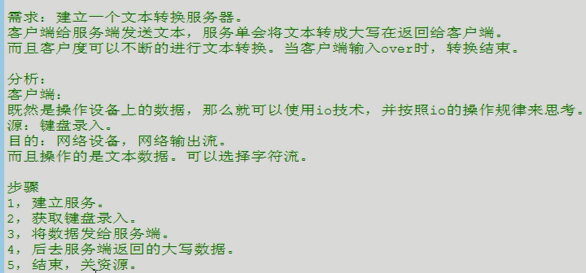


Demo:



**总结**：

我们在进行Socket编程的时候以一种“IO 源和目的”的思想来思考。我们首先需要直达搜的是源是什么？ 目的是什么？



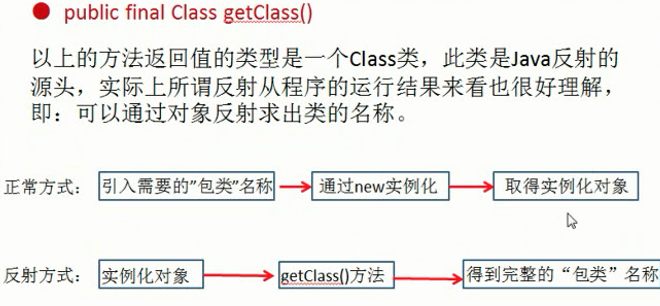
## 1.3 Java反射机制

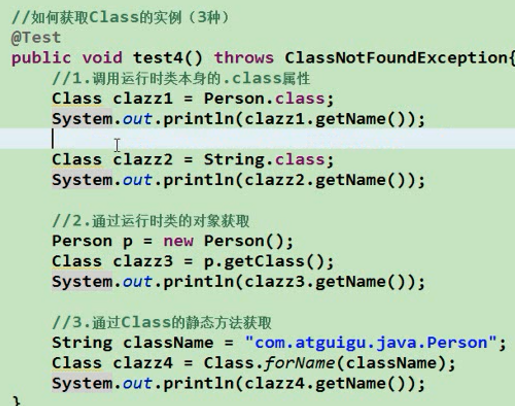
**什么是反射**：

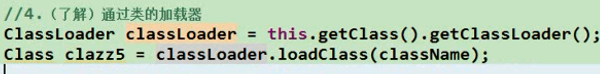
反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。(反射是**动态**的关键。反射允许程序在运行的时候借助Reflation API 获取任何类的内部信息，并能直接操作任意对象的内部属性和方法。)



Java.lang.class 是反射的源头。 一个class对象实质上就是一个指针，它指向的是对应的类。







**反射能做什么**?

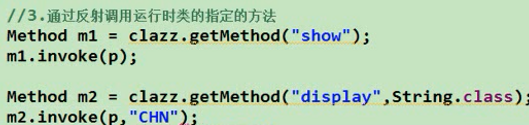
1. 在运行时判断任意一个对象所属的类；
2. 在运行时构造任意一个类的对象；
3. 在运行时判断任意一个类所具有的成员变量和方法；
4. 在运行时调用任意一个对象的方法；
5. 生成动态代理。

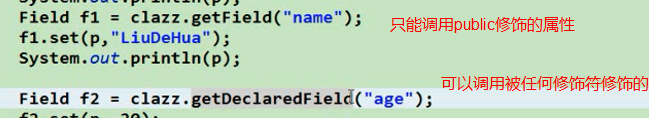
**为什么需要反射**：

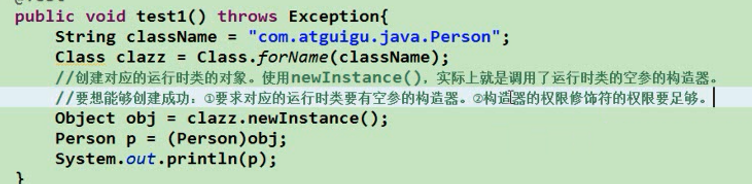
反射最大的作用就在于我们可以不在编译时知道某个对象的类型，而在运行时得到。同时我们只需要得到我们想得到的类的名字即可(如果不在一个包，必须写完整的名字包括包名)。

**怎么使用反射**：

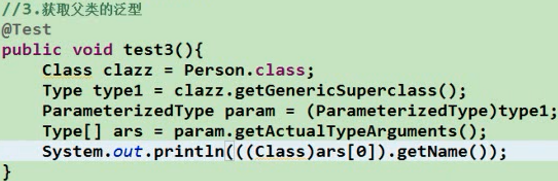


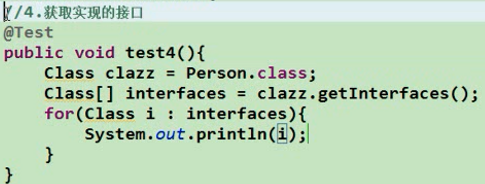


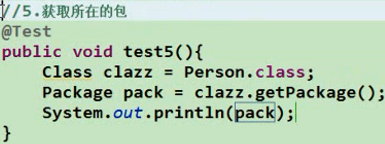


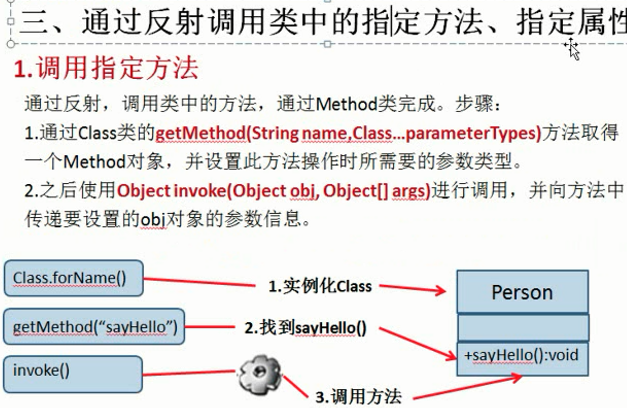












注意： 反射之后，封装性依然在。私有的一半无法获取。

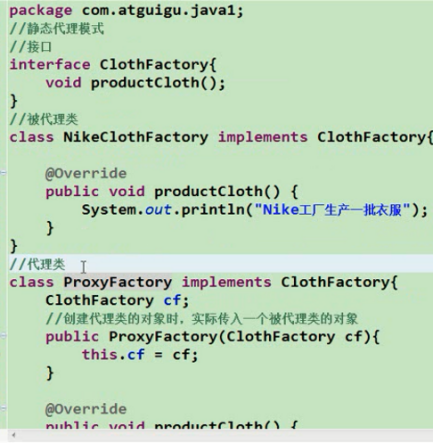
**动态代理**：

**什么是代理**：

代理就是为其他对象提供一种代理以控制对这个对象的访问。 说白了就是，在一些情况下客户不想或者不能直接引用一个对象，而代理对象可以在客户和目标对象之间起到中介作用，去掉客户不能看到的内容和服务或者增添客户需要的额外服务。

**什么是动态代理**：

客户通过代理类来调用其他对象的方法，并且是在程序运行时根据需要动态创建目标类的代理对象。







## 1.4 I/O

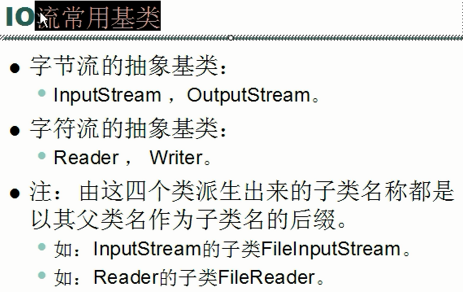
**什么是I/O**:

I/O流就是用来处理设备之间的数据传输的。 I/o就是输入和输出的简写。I/O主要被用来处理设备上已有的数据(文件)。



数据的本质就是字节(字符流的出现是为了更好的操作文本)而其他格式的文件例如MP3，视屏等都是字节。

对数据的操作就只有两种读和写。向外面写是输出流output。往里读是输入流Input。



(字符流读写所使用的默认编码是系统的默认编码(win7是GBK) )

理解流这个概念：流相当于内存中的一个管道(这个管道中有一个缓冲，暂时存储放入其中的数据)。

**为什么使用I/O**:

对数据进行操作。

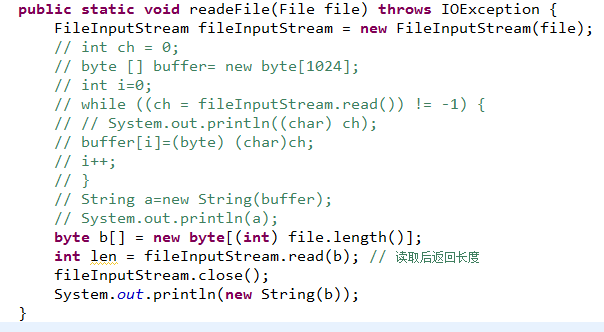
**怎么使用I/O**：

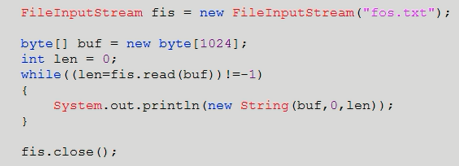
**问题记录**：

我思考的问题是： FileInputStream 的无参read（）方法，每次只能读到的一个字节。但是如果我们拿着这个字节流的方法去读取一个字符文件，使用FileInputStream我们如何能读取字符文件,因为在Java中UTF-8编码对汉字的编码是以3个自己，GBK对字符的编码是2个字节。



还可以：定义一个字节数组，每次流读取了之后，追加写到这个字节数组，读完了之后，再把这个字节数组转成字符串。(但是这个方法需要注意，new String()方法的默认编码是UTF-8,如果源文件的编码不是UTF-8，我们需要自己在这个方法中指定编码)





这个方法的本事依旧是缓冲之后转换。

## 1.5 Java的集中简单的设计模式

http://www.jianshu.com/p/61b67ca754a3

**什么是设计模式**：

设计模式是一套别长期使用并进行总结的经验(使用设计模式是为了让我们利用这些经验来重用代码)。

### 几种常用的设计模式及demo

**单例设计模式**：

单例: 顾名思义就是单个实例。

单例设计模式简单说就是无论程序如何运行，采用单例设计模式的类（Singleton类）永远只会有一个实例化对象产生。

为什么会有单列设计模式：

单例模式主要是为了避免因为创建了多个实例造成资源的浪费，且多个实例由于多次调用容易导致结果出现错误，而使用单例模式能够保证整个应用中有且只有一个实例。单例的出现就是为了保证在内存中一个单例类的示例对象是唯一的。

什么时候需要使用单例：

根据我们对单例的了解：单例模式是为了保证在内存中类的实例对象是唯一的，那么什么时候需要确保类的实例对象在内存中是唯一的？当这个类的实例对象中的数据在内存中被其他类的对象所共享，那么这个时候就需要确保单例类的实例对象是唯一的。

**如何确保类对象的唯一**：

实现步骤：

1. 将采用单例设计模式的类的构造方法私有化（采用private修饰）)不允许其他程序用new对象。
2. 在其内部产生该类的实例化对象，并将其封装成private static类型。  
   既然不允许其他类来new我们的单例对象，且我们的构造方法是私有的(只能在我们的本类中调用)，那么我们就只能在本类中自己new一个对象。

因为要在第3部中对外提供一个外部可以访问的静态方法。静态方法只能访问静态的成员属性。

1. 对外提供一个可以让其他程序获取该对象的方法，定义一个静态方法返回该类的实例。

  因为对象是在本类中创建的，所以需要提供一个方法让其它的类获取这个对象。且外部

不能new我们的本类对象，那么就只能今天调用了。

单例模式分为饿汉式(获取的对象是一开始就准备好的)和懒汉式（当我们需要的时候才去准备对象，但是懒汉式线程不安全一般不推荐使用）。

**public** **class** SingelDemoTest {

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

// 如何测试我们定义的类是单例的，在单例类中定义一个数据，我们访问后修改之后再次获取

SingleDemo singleDemo=SingleDemo.*getSingel*();

System.*out*.println(singleDemo.num);

singleDemo.num=4;

System.*out*.println(SingleDemo.*getSingel*().num);

}

}

// 在下面写定义一个个单例类，使用饿汉式

**class** SingleDemo{

**public** **int** num=3;

**private** **static** SingleDemo *singleDemo*=**new** SingleDemo();

**private** SingleDemo() {};

**public** **static** SingleDemo getSingel() {

**return** *singleDemo*;

}

}

**工厂设计模式**：

工厂设计模式的作用就是为了减少代码之间的**耦合度**。让需求的修改不必去修改我们的核心代码。

工厂模式分类：

1、静态工厂模式

2、简单工厂模式

3、工厂方法模式

4、抽象工厂模式

1）还没有工厂时代：假如还没有工业革命，如果一个客户要一款宝马车,一般的做法是客户去创建一款宝马车，然后拿来用。

2）简单工厂模式：后来出现工业革命。用户不用去创建宝马车。因为客户有一个工厂来帮他创建宝马.想要什么车，这个工厂就可以建。比如想要320i系列车。工厂就创建这个系列的车。即工厂可以创建产品。

3）工厂方法模式时代：为了满足客户，宝马车系列越来越多，如320i，523i,30li等系列一个工厂无法创建所有的宝马系列。于是由单独分出来多个具体的工厂。每个具体工厂创建一种系列。即具体工厂类只能创建一个具体产品。但是宝马工厂还是个抽象。你需要指定某个具体的工厂才能生产车出来。

4）抽象工厂模式时代：随着客户的要求越来越高，宝马车必须配置空调。而且这空调必须对应给系列车才能使用。于是这个工厂开始生产宝马车和需要的空调。

最终是客户只要对宝马的销售员说：我要523i空调车，销售员就直接给他523i空调车了。而不用自己去创建523i空调车宝马车.

这就是工厂模式

<http://blog.csdn.net/lmj623565791/article/details/24460585>

**什么场景需要使用到工厂设计模式**？

当需求会经常性的改变，导致核心代码也会跟着改变的时候，这时候就需要使用工厂设计模式来将经常改变的东西从核心代码中抽取成一个工厂，降低代码的耦合性，这样需求改表时就不用去修改核心代码，直接修改工厂中代码就可以了。

**package** com.zhao.day4;

/\*\*

\* 这里写一个测试来帮助理解工厂模式

\*

\* 需求：产品的更新换代。 1）还没有工厂时代：假如还没有工业革命，如果一个客户要一款宝马车,一般的做法是客户去创建一款宝马车，然后拿来用。

\* 2）简单工厂模式

\* ：后来出现工业革命。用户不用去创建宝马车。因为客户有一个工厂来帮他创建宝马.想要什么车，这个工厂就可以建。比如想要320i系列车。工厂就创建这个系列的车

\* 。即工厂可以创建产品。  3）工厂方法模式时代：为了满足客户，宝马车系列越来越多，如320i，523i,30li等系列一个工厂无法创建所有的宝马系列。

\* 于是由单独分出来多个具体的工厂

\* 。每个具体工厂创建一种系列。即具体工厂类只能创建一个具体产品。但是宝马工厂还是个抽象。你需要指定某个具体的工厂才能生产车出来。

\* 4）抽象工厂模式时代：随着客户的要求越来越高，宝马车必须配置空调。而且这空调必须对应给系列车才能使用。于是这个工厂开始生产宝马车和需要的空调。

\*

\* **@author** Administrator

\*

\*/

**public** **class** FactoryTest {

// 这里的需求就只有一个，生成一个一辆宝马车

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

// **TODO** Auto-generated method stub

// 1.没有工厂 ：

BMW bmw=**new** BMW(); //这样写会有一个问题，如果宝马车有设计更新，那么我们就需要去修改生成宝马的源代码。每更新一次，修改一次。

// 2.简单工厂模式

BMWFctory facFctory = **new** BMWFctory("320i");

facFctory.createCar(); //这样做就将变化的抽取出来，不用每次变化了就去修改核心代码

// 3）工厂方法模式时代

CarFactory carFactory=**new** Factoty523();

carFactory.createCar();

// 4）抽象工厂模式时代 这个就是用来创建其他的额外功能的工厂的

}

}

/\*\*

\* 这里是一个宝马车的类，下面为了模仿需求的变更，设计几个类似的变化的类

\* **@author** Administrator

\*

\*/

**class** BMW{

}

**class** BMW320i **extends** BMW{

**public** BMW320i() {

// **TODO** Auto-generated constructor stub

System.*out*.println("这是一辆320i的宝马车");

}

}

/\*\*

\* 这是一个创建宝马车的工厂，这里可以创建多种样式的宝马

\* **@author** Administrator

\*

\*/

**class** BMWFctory{

**private** String cartype;

**public** BMWFctory(String type) {

// 这里传入需要生产的车的型号

**this**.cartype=type;

}

**public** BMW createCar() {

**if** ("320i".equals(cartype)) {

**return** **new** BMW320i();

}

**return** **null**;

}

}

// 下面是关于工厂方法模式 的

// 这里是一个工厂无法创建所有的宝马系列 ，就需要弄几个工厂，这时就要将原来的工厂进行抽象。 其他的分工厂来基础这个基础工厂

**abstract** **class** CarFactory{

**private** String cartype;

**public** **abstract** BMW createCar();

}

**class** Factoty523 **extends** CarFactory {

@Override

**public** BMW createCar() {

// **TODO** Auto-generated method stub

**return** **null**;

}

}

**class** Factoty30l **extends** CarFactory {

@Override

**public** BMW createCar() {

// **TODO** Auto-generated method stub

**return** **null**;

}

}

// 抽象工厂 ： 需要实现给车添加空调

**interface** airFactory{

// 给车添加空调

**public** **void** addAir();

}

**class** Factoty524 **extends** CarFactory **implements** airFactory {

@Override

**public** **void** addAir() {

// **TODO** Auto-generated method stub

System.*out*.println("添加了空调");

}

@Override

**public** BMW createCar() {

// **TODO** Auto-generated method stub

**return** **null**;

}

}

**模版方法设计模式**：

<http://blog.csdn.net/lmj623565791/article/details/26276093>

我们定义好做一系列事情的已经固定了的步骤，将不确定的少部分方法抽象出来交给子类，让他自己去具体实现。 (这就类似写网页去套模板一样)

**装饰者模式设计模式**：

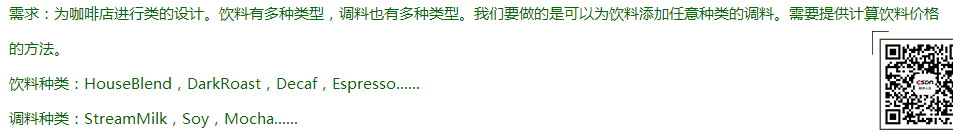
<http://blog.csdn.net/lmj623565791/article/details/24269409>

装饰者模式：若要扩展功能，装饰者提供了比集成更有弹性的替代方案，动态地将责任附加到对象上。 (我们所熟知的I/O中的流体系就是使用的装饰者设计模式，缓冲流装饰着普通流)

**什么时候需要使用到装饰者模式?**当我们设计好了一个类，我们需要给这个类添加一些辅助的功能，并且不希望改变这个类的代码的时候。

这里还体现了一个原则：类应该对扩展开放，对修改关闭。

DEMO：



**package** com.zhao.day4;

/\*\*

\* 这里是对装饰者模式的一个测试

\* 需求：

\* 为咖啡店进行类的设计。饮料有多种类型，调料也有多种类型。我们要做的是可以为饮料添加任意种类的调料。需要提供计算饮料价格的方法。

\* 饮料种类：DarkRoast，Decaf

\* 调料种类：StreamMilk，Soy

\* **@author** Administrator

\*

\*/

**public** **class** DecorateTestDemo {

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

// 这里尝试在DarkRoast 中加入这两个调料

Drink drink=**new** StreamMilk(**new** Soy(**new** DarkRoast()));

System.*out*.println(drink.description()+ drink.caculatePrice());

}

}

// 为了解决这个需求，因为计算加了调料后的价格与饮料本身无关，所以将其设计成接口

//-----------------------------------饮料----------------------------------------------

**interface** Drink{

/\*\*

\* 计算价格

\* **@return**

\*/

**public** **int** caculatePrice();

/\*\*

\* 饮料描述

\* **@return**

\*/

**public** String description();

}

/\*\*

\* 饮料DarkRoast

\* **@author** Administrator

\*

\*/

**class** DarkRoast **implements** Drink{

@Override

**public** **int** caculatePrice() {

// **TODO** Auto-generated method stub

**return** 10;

}

@Override

**public** String description() {

// **TODO** Auto-generated method stub

**return** "this is DarkRoast";

}

}

//-----------------------------------饮料

//--------------------------------------调料

// 饮料与调料的关系是 1对多 ，我们在多的这边维护关系

// 调料这个接口的功能定义与饮料是重合的，且调料也是可以与饮用的，可以看出饮料这个接口的子类

**interface** Seasoning **extends** Drink{

}

/\*\*

\* StreamMilk 调料

\* **@author** Administrator

\*

\*/

**class** StreamMilk **implements** Seasoning{

**private** Drink drink;

StreamMilk(Drink drink)

{

**this**.drink=drink;

}

@Override

**public** **int** caculatePrice() {

// **TODO** Auto-generated method stub

**return** 4+drink.caculatePrice();

}

@Override

**public** String description() {

// **TODO** Auto-generated method stub

**return** "StreamMilk 调料"+drink.description();

}

}

/\*\*

\* Soy 调料

\* **@author** Administrator

\*

\*/

**class** Soy **implements** Seasoning{

**private** Drink drink;

Soy(Drink drink)

{

**this**.drink=drink;

}

@Override

**public** **int** caculatePrice() {

// **TODO** Auto-generated method stub

**return** 3+drink.caculatePrice();

}

@Override

**public** String description() {

// **TODO** Auto-generated method stub

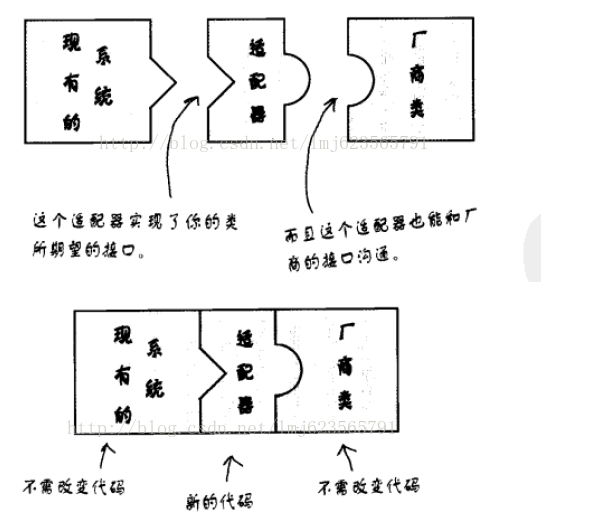
**return** "Soy 调料"+drink.description();

}

}

**适配器设计模式**：

定义：将一个类的接口转换成客户期望的另一个接口，适配器让原本接口不兼容的类可以相互合作。这个定义还好，说适配器的功能就是把一个接口转成另一个接口。



Demo：

1. /\*\*
2. \* 提供5V电压的一个接口
3. \* @author zhy
4. \*
5. \*/
6. **public** **interface** V5Power
7. {
8. **public** **int** provideV5Power();
9. }
10. **public** **class** Mobile
11. {
12. /\*\*
13. \* 充电
14. \* @param power
15. \*/
16. **public** **void** inputPower(V5Power power)
17. {
18. **int** provideV5Power = power.provideV5Power();
19. System.out.println("手机（客户端）：我需要5V电压充电，现在是-->" + provideV5Power + "V");
20. }
21. }
22. /\*\*
23. \* 家用220V交流电
24. \* @author zhy
25. \*
26. \*/
27. **public** **class** V220Power
28. {
29. /\*\*
30. \* 提供220V电压
31. \* @return
32. \*/
33. **public** **int** provideV220Power()
34. {
35. System.out.println("我提供220V交流电压。");
36. **return** 220 ;
37. }
38. }
39. /\*\*
40. \* 适配器，把220V电压变成5V
41. \* @author zhy
42. \*
43. \*/
44. **public** **class** V5PowerAdapter **implements** V5Power
45. {
46. /\*\*
47. \* 组合的方式
48. \*/
49. **private** V220Power v220Power ;
51. **public** V5PowerAdapter(V220Power v220Power)
52. {
53. **this**.v220Power = v220Power ;
54. }
56. @Override
57. **public** **int** provideV5Power()
58. {
59. **int** power = v220Power.provideV220Power() ;
60. //power经过各种操作-->5
61. System.out.println("适配器：我悄悄的适配了电压。");
62. **return** 5 ;
63. }
65. }

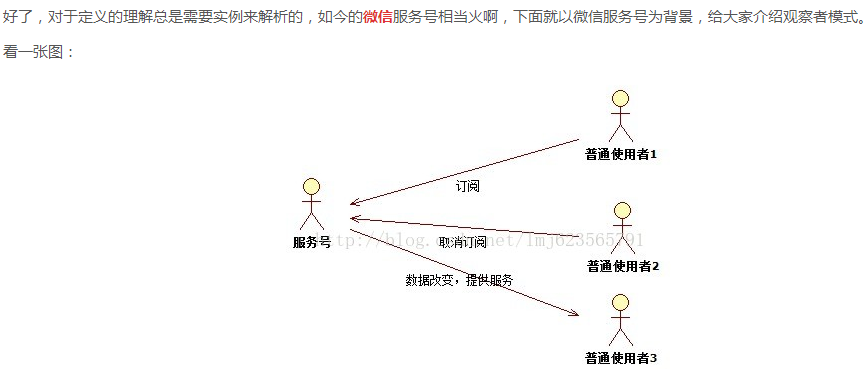
测试：

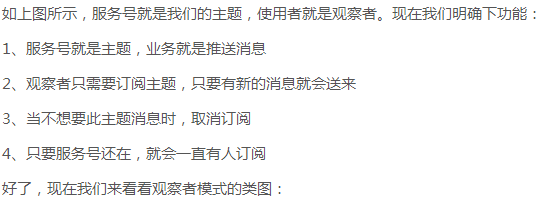
1. **public** **class** Test
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. Mobile mobile = **new** Mobile();
6. V5Power v5Power = **new** V5PowerAdapter(**new** V220Power()) ;
7. mobile.inputPower(v5Power);
8. }
9. }

**观察者设计模式**：

定义了对象之间的一对多的依赖，这样一来，当一个对象改变时，它的所有的依赖者都会收到通知并自动更新。

<http://blog.csdn.net/lmj623565791/article/details/24179699>





检查者模式就是类似于监听器。

**外观设计模式**：

提供一个统一的接口，用来访问子系统中的一群接口，外观定义了一个高层的接口，让子系统更容易使用。其实就是为了方便客户的使用，把一群操作，封装成一个方法。

外观模式：一般用于需要简化一个很大的接口，或者一群复杂的接口的时候。

# JavaWeb 基础

## 2.0 web服务器

Web服务器一般指网站服务器，是指驻留于因特网上某种类型计算机的程序。可以向浏览器等Web客户端提供文档，[1] 也可以放置网站文件，让全世界浏览；可以放置数据文件，让全世界下载。

WEB服务器也称为WWW(WORLD WIDE WEB)服务器，主要功能是提供网上信息浏览服务。

**WEB服务器工作原理**

Web服务器的工作原理:

一般可分成如下4个步骤：

连接过程:

Web服务器和其浏览器之间所建立起来的一种连接

请求过程

Web的浏览器运用socket这个文件向其服务器而提出各种请求

应答过程

运用HTTP协议把在请求过程中所提出来的请求传输到Web的服务器，进而实施任

务处理，然后运用HTTP协议把任务处理的结果传输到Web的浏览器，同时在Web

的浏览器上面展示上述所请求之界面

关闭连接

应答过程完成以后，Web服务器和其浏览器之间断开连接之过程.

## 2.1 http协议

**http 超文本传输协议**：

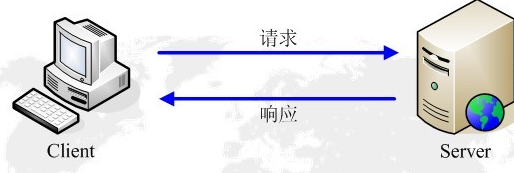
服务器使用HTTP（超文本传输协议）与客户机浏览器进行信息交流。

HTTP协议是用于从服务器传输超文本到本地浏览器的传送协议。

HTTP是一个应用层协议，由请求和响应构成，是一个标准的客户端服务器模型。HTTP是一个**无状态的协议**。

默认HTTP的端口号为80，HTTPS的端口号为443。

HTTP协议永远都是客户端发起请求，服务器回送响应

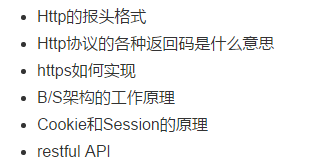


（这样就限制了使用HTTP协议，无法实现在客户端没有发起请求的时候，服务器将消息推送给客户端）

HTTP协议是一个无状态，无连接的协议，同一个客户端的这次请求和上次请求是没有对应关系。(因为服务器没有维护请求)

http（超文本传输协议）是一个基于请求与响应模式的、无状态的、应用层的协议，常基于TCP的连接方式。

我们在开发中需要掌握的



**http的报头格式**：



请求行：

由请求方法字段、URL字段和HTTP协议版本字段，组成，它们用空格分隔，例如：GET /index.html HTTP/1.1

GET：当client要从server中读取文档时，使用GET方法。GET方法要求服务器将URL定位的资源放在响应报文的数据部分，回送给client。使用GET方法时，请求参数和对应的值附加在URL后面，利用一个问号（"?"）代表URL的结尾与请求参数的开始，传递参数长度受限制，例如： /index.jsp?id=100&op=bind

POST:当client给服务器提供信息较多时， 使用POST方法。POST方法将请求参数封装在**HTTP请求数据**中，以key/value的形式出现，可以传递大量数据，可用来传递文件

消息头部：

请求头部由key/value键值对组成，每行一对，key和value用冒号":"分隔，**请求头部通知服务器有关于client端的请求信息**，典型的请求头：

User-Agent：产生请求的浏览器类型

Accept：client端可识别的内容类型列表

Host：请求的主机名，允许多个域名同处一个ip地址，即虚拟主机

空行：

最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器请求头结束。

**对于一个完整的http请求来说空行是必须的**，否则服务器会认为本次请求的数据尚未完全发送到server，处于等待状态。(从上面的报文构成图也可以看出空行是分割报文内容的分割符，服务器用此来区分报文的各个部分)

请求正文：

请求数据不在GET方法中使用，而是在**POST中使用**。POST方法适用于需要client填写表单的场合，与请求数据相关的最常用的请求头是Content-Type 和Content-Length

**http协议中各种常见的返回码**：

200 – 服务器成功返回网页

404 – 请求的网页不存在

503 – 服务器暂时不可用

HTTP的状态码及其含义：

1xx——信息提示

这些状态代码表示临时的响应。客户端在收到常规响应之前，应准备接收一个或多个 1xx 响应。

100——Continue 初始的请求已经接受，客户应当继续发送请求的其余部分

101——Switching Protocols 服务器将遵从客户的请求转换到另外一种协议

2xx——成功

这类状态代码表明服务器成功地接受了客户端请求。

200——OK 一切正常，对GET和POST请求的应答文档跟在后面

3xx——重定向

客户端浏览器必须采取更多操作来实现请求。例如，浏览器可能不得不请求服务器上的不同的页面，或通过代理服务器重复该请求。

4xx——客户端错误

发生错误，客户端似乎有问题。例如，客户端请求不存在的页面，客户端未提供有效的身份验证信息。

400——Bad Request 请求出现语法错误

404——Not Found 无法找到指定位置的资源。这也是一个常用的应答。

405——Method Not Allowed 请求方法（GET、POST、HEAD、DELETE、PUT、TRACE等）对指定的资源不适用，用来访问本页面的HTTP 谓词不被允许（方法不被允许）

5xx——服务器错误(服务器由于遇到错误而不能完成该请求)

500——Internal Server Error 服务器遇到了意料不到的情况，不能完成客户的请求

503——Service Unavailable 服务不可用，服务器由于维护或者负载过重未能应答(请求服务器没有应答)

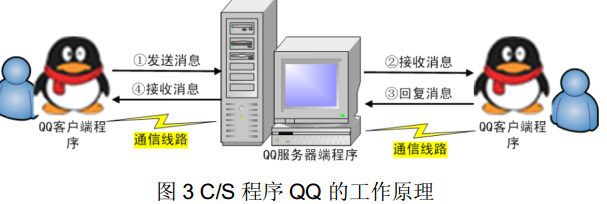
**https如何实现**：

HTTPS实际是SSL over HTTP, 该协议通过SSL在发送方把原始数据进行加密，在接收方解密。HTTPS就是在http的基础上对数据进行了加密。

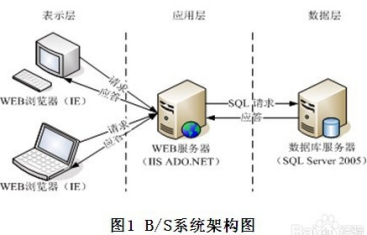
**C/S ,B/S架构的工作原理**：

**C/S** 全称client/sever，即客户端/服务器。 使用的话需要先安装客户端

C/S架构就像qq一样分为客户机和服务器两层。

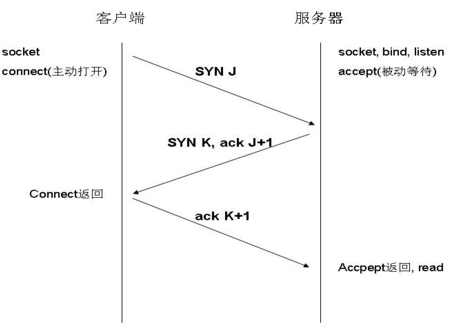


B/S 全称Browser/Server，即浏览器/服务器。只要有浏览器就可以打开。



http的三次握手:

TCP传输控制协议.TCP是主机对主机层的传输控制协议，提供**可靠**的连接服务，采用**三次握手**确认建立一个连接:



第一次握手：主机A发送位码为syn＝1,随机产生seq number=1234567的数据包到服务器，主机B由SYN=1知道，A要求建立联机；

第二次握手：主机B收到请求后要确认联机信息，向A发送ack number=(主机A的seq+1),syn=1,ack=1,随机产生seq=7654321的包

第三次握手：主机A收到后检查ack number是否正确，即第一次发送的seq number+1,以及位码ack是否为1，若正确，主机A会再发送ack number=(主机B的seq+1),ack=1，主机B收到后确认seq值与ack=1则连接建立成功。

完成三次握手，主机A与主机B开始传送数据。

## 2.2 Servlet

**什么是Servlet**：

Servlet是用Java编写的服务器端程序。主要功能在于交互式地浏览和修改数据，生成动态Web内容。广义的Servlet是指任何实现了这个Servlet接口的类。Servlet用来响应基于HTTP协议的请求。

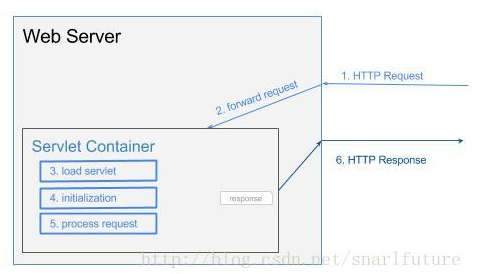
即： Servlet是运行于服务端的java程序。我们一般自己实现的Servlet都是从HttpServlet类继承，然后实现自己的init | doGet | doPost | service方法。

Java Servlet 是运行在 Web 服务器或应用服务器上的程序，它是作为来自 **Web 浏览器或其他 HTTP 客户端的请求**和 HTTP **服务器上的数据库或应用程序**之间的中间层。

**Servlet容器和web服务器之间的区别与联系**

我们常说的Servlet容器在我看来指的就是服务器，就我的开发环境来看就是本地的Tomcat。但是这是不正确的。

WEB服务器与Servlet容器之间的关系如下图所示：



servlet容器只是web服务器的一部分，一个组件。也是web服务器处理请求的核心。

WEB服务器的作用就是来接收http请求，但是处理是Servlet容器来处理的。

**生命周期**：

客户端请求该 Servlet；

加载 Servlet 类到内存；

实例化并调用init()方法初始化该 Servlet；

service()（根据请求方法不同调用doGet() 或者 doPost()，此外还有doHead()、doPut()、doTrace()、doDelete()、doOptions()、destroy()。

加载和实例化 Servlet。这项操作一般是动态执行的。然而，Server 通常会提供一个管理的选项，用于在 Server 启动时强制装载和初始化特定的 Servlet。

Server 创建一个 Servlet的实例

第一个客户端的请求到达 Server

Server 调用 Servlet 的 init() 方法（可配置为 Server 创建 Servlet 实例时调用，在 web.xml 中 <servlet> 标签下配置 <load-on-startup> 标签，配置的值为整型，值越小 Servlet 的启动优先级越高）

一个客户端的请求到达 Server

Server 创建一个请求对象，处理客户端请求

Server 创建一个响应对象，响应客户端请求

Server 激活 Servlet 的 service() 方法，传递请求和响应对象作为参数

service() 方法获得关于请求对象的信息，处理请求，访问其他资源，获得需要的信息

service() 方法使用响应对象的方法，将响应传回Server，最终到达客户端。service()方法可能激活其它方法以处理请求，如 doGet() 或 doPost() 或程序员自己开发的新的方法。

对于更多的客户端请求，Server 创建新的请求和响应对象，仍然激活此 Servlet 的 service() 方法，将这两个对象作为参数传递给它。如此重复以上的循环，但无需再次调用 init() 方法。一般 Servlet 只初始化一次(只有一个对象)，当 Server 不再需要 Servlet 时（一般当 Server 关闭时），Server 调用 Servlet 的 destroy() 方法。

( 自己理解：

在web容器的启动的时候，会根据web.xml来init()Servlet容器，在初始化的时候加载了web.xml中<load-on-startup> 标签中的启动项和<init-param>中配置的初始化参数。初始化后建立了Servlet容器的实例，Servlet容器实例建立后开始接受并处理浏览器发出请求(通过调用service()方法来处理请求，每次接受请求都通过servic()方法来处理，不会重新初始化)，最后结束的时候调用destroy方法结束Servlet容器。

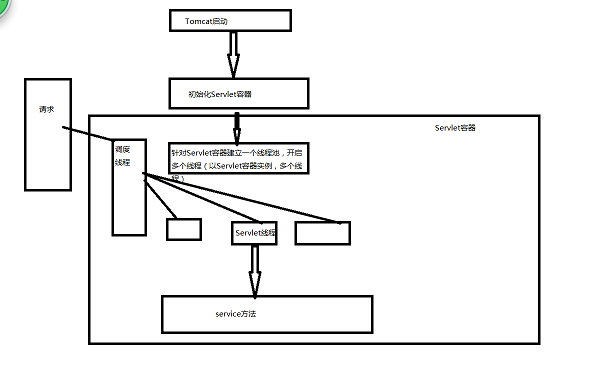
)

**[Servlet容器如何同时来处理多个请求](http://www.cnblogs.com/liangxiaofeng/p/5838696.html)**

Servlet采用多线程来处理多个请求同时访问。

servlet依赖于一个线程池来处理服务请求。线程池实际上是一系列的工作者线程集合。Servlet使用一个调度线程来管理工作者线程.当容器收到一个Servlet请求，调度线程从线程池中选出一个工作者线程,将请求传递给该工作者线程，然后由该线程来执行Servlet的service方法。当这个线程正在执行的时候,容器收到另外一个请求,调度线程同样从线程池中选出另一个工作者线程来服务新的请求,容器并不关心这个请求是否访问的是同一个Servlet.当容器同时收到对同一个Servlet的多个请求的时候，那么这个Servlet的service()方法将在多线程中并发执行。Servlet容器默认采用**单实例多线程**的方式来处理请求，这样减少产生Servlet实例的开销，提升了对请求的响应时间。

所以： 整个流程如下：



(但是需要注意的是: 每一个http请求是一个独立的整体，他的请求到服务器，服务器对其作出响应。这个请求与同一浏览器的后面的请求没有必然的关系(http是无连接协议)。正因为每个http请求是独立的，所以他们在Servlet容器中处理也是独立的，他与浏览器无关

)

## 2.3 ServletConfig对象



通过查看API，我们知道了ServletConfig是一个接口，在eclipse中查看他的声明

public abstract interface javax.servlet.ServletConfig

也可以看出他是抽象的接口(因为接口的方法都是默认的是public abstract修饰的，那么接口也可以说成是抽象的)。因为是抽象的接口，那么我们就不能直接new ,再看文档我们知道他的实现类是GenericServlet (它基本上是所有Servlet的父类)，这同时说明我们无法直接获取ServletConfig对象。如果我们想获取他就只能去new Servlet，在他的实现类中去获取。

ServletConfig接口提供了以下的方法



在Servlet 的配置文件中，可以用一个或多个<init-param>标签为servlet配置一些初始化参数(就是在这里调用ServletConfig中最常用的一个方法getInitParameterNames)。

当servlet配置了初始化参数之后，web容器在创建servlet实例对象时，会自动将这些初始化参数封装到ServletConfig对象中，并在调用servlet的init方法时，将ServletConfig对象传递给Servlet。进而，程序员通过ServletConfig对象得到当前servlet的初始化参数信息。



开发中ServletConfig的常用来获取字符集编码。

## 2.4 ServletContext



从Servlet的API文档声明中可以看出ServletContext是一个抽象的接口。

从上面的ServletConfig接口的getServletContext()方法可以获取我们的ServletContext。

ServletContext按照中文意思来接收就是Servlet的上下文。

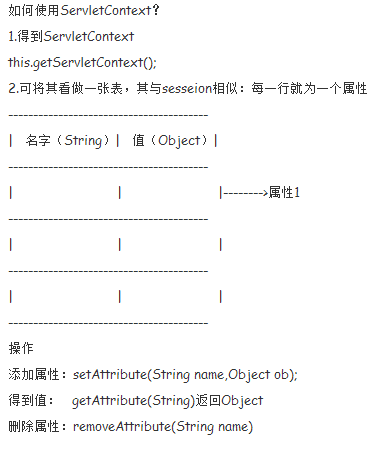
ServletContext是Servlet容器(服务器)开辟的是一个**全局**的存储信息的空间。服务器一开启的时候就已经存在，服务器关闭的时候才销毁。运行在Java虚拟机中的每一个Web应用程序都有一个与之相关的Servlet上下文。

request，一个用户可有多个；session，一个用户一个；而servletContext，所有用户共用一个。所以，为了节省空间，提高效率，ServletContext中，要放所有用户需要共享的线程又是安全的一些必须重要信息。

ServletContext对象是Web服务器中的一个已知路径的根，Servlet上下文被定位于http://localhost:8080/项目名.

Servlet上下文提供对应用程序中所有Servlet所共有的各种资源和功能的访问。

Servlet上下文API用于设置应用程序中**所有Servlet共有**的信息。



## 2.5 HttpServletRequest 和HttpServletResponse

由来：

Web服务器收到一个http请求，会针对每个请求创建一个HttpServletRequest和HttpServletResponse对象，向客户端发送数据找HttpServletResponse,从客户端取数据找HttpServletRequest.





通过JDK定义可以发现他们也是一个接口。他们分别继承自ServletRequest和ServletResponse，这个接口只是封装了提供http协议的方法。而ServletRequest和ServletResponse对象分别代表的是一次代表一个HTTP请求和一次http响应。

HttpServletRequest接口最常用的方法就是获得请求中的参数，这些参数一般是客户端表单中的数据。

在Servlet中，当服务器响应客户端的一个请求时，就要用到HttpServletResponse接口。设置响应的类型可以使用setContentType()方法。发送字符数据，可以使用getWriter()返回一个对象。

HttpServletResponse继承了ServletResponse接口，并提供了与Http协议有关的方法，这些方法的主要功能是设置[HTTP状态码](http://baike.baidu.com/view/1790469.htm)和管理Cookie。HttpServletResponse对象代表服务器的响应。这个对象中封装了向客户端发送数据、发送响应头，发送响应状态码的方法

HttpServletResponse对象可以向客户端发送三种类型的数据:

a.**响应头(**Response headers**)//**addHeader(String name,String value)  将指定的名字和值加入到响应的头信息中,setHeader(String name,String value)  将给出的名字和值设置响应的头部,常用的还有 response.setContentType("text/html;charset=utf-8");

**b.状态码(**Protocol—Status code—Description**) //**setStatus(int sc)  给当前响应设置状态码

**c.实体数据(**Entity body **)**

**出现乱码的原因和解决**

系统会把在java程序中的中文字符按照某种字符集编码的方式转换成字节数组，再通过浏览器输出，浏览器在输出的时候要进行解码，只有在这两种方式一样的情况下，才不会出现乱码。

1. 字符编码是用reponse对象去设置的。

设置编码的两种方式：

response.setContentType("text/html;charset=utf-8"); //设置响应头中信息，告诉浏览器用什么编码来解析服务器返回的字节数组

request.setCharacterEncoding("utf-8");//设置对客户端请求进行重新编码的编码。

指定对浏览器发送过来的数据进行重新编码

必须是在out.println之前使用，要不会出现错误，会抛找不到设置的字符编码而出错。

**解决办法**：

Post方式提交出现乱码

request.setCharacterEncoding("UTF-8");

请求中之所以会产生乱码，就是因为服务器和客户端沟通的编码不一致造成的，因此解决的办法是：在客户端和服务器之间设置一个统一的编码，之后就按照此编码进行数据的传输和接收。

**Get方式提交出现乱码**

对于以get方式传输的数据，request即使设置了以指定的编码接收数据也是无效的,默认的还是使用ISO8859-1这个字符编码来接收数据，客户端以UTF-8的编码传输数据到服务器端，而服务器端的request对象使用的是ISO8859-1这个字符编码来接收数据，服务器和客户端沟通的编码不一致因此才会产生中文乱码的。

（既然我们不能设置get方式的解析编码，那么我们就只能在接收之后来自己解析。）

解决办法：在接收到数据后，先获取request对象以ISO8859-1字符编码接收到的原始数据的字节数组，然后通过字节数组以指定的编码构建字符串，解决乱码问题。代码如下：

String name = request.getParameter("name");//接收数据

name =new String(name.getBytes("ISO8859-1"), "UTF-8") ;//获取request对象以ISO8859-1字符编码接收到的原始数据的字节数组，然后通过字节数组以指定的编码构建字符串，解决乱码问题

## 2.6 cookie和session

Cookie和session出现的原因：

HTTP协议是无状态的协议。一旦数据交换完毕，客户端与服务器端的连接就会关闭，再次交换数据需要建立新的连接。这就意味着**服务器无法从连接上跟踪会话**。即用户A购买了一件商品放入购物车内，当再次购买商品时服务器已经无法判断该购买行为是属于用户A的会话还是用户B的会话了。要跟踪该会话，必须引入一种机制。

我们需要一种机制来弥补HTTP协议无状态的不足，也就是确保服务器知道哪几个连接属于哪个浏览器发出的。

由于HTTP是一种无状态的协议，服务器单从网络连接上无从知道客户身份。怎么办呢？就给客户端们颁发一个通行证吧，每人一个，无论谁访问都必须携带自己通行证。这样服务器就能从通行证上确认客户身份了。这就是Cookie的工作原理。

**Cookie**实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用response向客户端浏览器颁发一个Cookie。**客户端浏览器会把Cookie保存起来**。当浏览器再请求该网站时，浏览器把请求的网址连同该Cookie一同提交给服务器。服务器检查该Cookie，以此来辨认用户状态。服务器还可以根据需要修改Cookie的内容。

(这就告诉我们)

当我们需要查看某个网站颁发的Cookie很简单。在浏览器地址栏输入**[JavaScript](http://lib.csdn.net/base/javascript" \o "JavaScript知识库" \t "http://blog.csdn.net/fangaoxin/article/details/6952954/_blank):alert (document. cookie)**就可以了（需要有网才能查看）。**[javascript](http://lib.csdn.net/base/javascript" \o "JavaScript知识库" \t "http://blog.csdn.net/fangaoxin/article/details/6952954/_blank)**脚本会弹出一个对话框显示本网站颁发的所有Cookie的内容

**Session**是另一种记录客户状态的机制，不同的是Cookie保存在客户端浏览器中，而Session保存在服务器上。客户端浏览器访问服务器的时候，**服务器把客户端信息以某种形式记录在服务器上**。这就是Session。客户端浏览器再次访问时只需要从该Session中查找该客户的状态就可以了。

**如果说Cookie机制是通过检查客户身上的“通行证”来确定客户身份的话，那么Session机制就是通过检查服务器上的“客户明细表”来确认客户身份**。Session相当于程序在服务器上建立的一份客户档案，客户来访的时候只需要查询客户档案表就可以了。

Session机制决定了当前客户只会获取到自己的Session，而不会获取到别人的Session。各客户的Session也彼此独立，互不可见。Session的使用比Cookie方便，但是过多的Session存储在服务器内存中，会对服务器造成压力。

Session的生命周期：

Session在用户第一次访问服务器的时候自动创建。需要注意只有访问JSP、Servlet等程序时才会创建Session，只访问HTML、IMAGE等静态资源并不会创建Session。如果尚未生成Session，也可以使用request.getSession(true)强制生成Session。

Session生成后，只要用户继续访问，服务器就会更新Session的最后访问时间，并维护该Session。用户每访问服务器一次，无论是否读写Session，服务器都认为该用户的Session“活跃（active）”了一次。

由于会有越来越多的用户访问服务器，因此Session也会越来越多。为防止内存溢出，服务器会把长时间内没有活跃的Session从内存删除。这个时间就是Session的超时时间。如果超过了超时时间没访问过服务器，Session就自动失效了。

Session的超时时间为maxInactiveInterval属性，可以通过对应的getMaxInactiveInterval()获取，通过setMaxInactiveInterval(longinterval)修改。

Session的超时时间也可以在web.xml中修改。



另外，通过调用Session的invalidate()方法可以使Session失效。

具体来说cookie机制采用的是在客户端保持状态的方案，而session机制采用的是在服务器端保持状态的方案。

由于采用服务器端保持状态的方案在客户端也需要保存一个标识，所以session机制可能需要借助于cookie机制来达到保存标识的目的。

当程序需要为某个客户端的请求创建一个session时，**服务器首先检查这个客户端的请求里是否已包含了一个session标识------------称为session id，**如果已包含则说明以前已经为此客户端创建过session，服务器就按照session id把这个session检索出来使用（检索不到，会新建一个），如果客户端请求不包含session id，则为此客户端创建一个session并且生成一个与此session相关联的session id，session id的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个session id将被在本次响应中返回给客户端保存。

保存这个session id的方式可以采用cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发挥给服务器。一般这个cookie的名字都是类似于SEEESIONID。但cookie可以被人为的禁止，则必须有其他机制以便在cookie被禁止时仍然能够把session id传递回服务器。经常被使用的一种技术叫做URL重写，就是把session id直接附加在URL路径的后面。还有一种技术叫做表单隐藏字段。就是服务器会自动修改表单，添加一个隐藏字段，以便在表单提交时能够把session id传递回服务器。

每个访问者服务器会分配一个id给你，那就是sessionid，session没什么很神秘或者奇异的地方，它的实质就是服务器端保存的一份数据而已，这份数据都有一个独一无二的ID，这个ID服务器会通过set cookie的方式告诉给客户端。然后客户端请求的时候把这个ID带上，服务端的session就可以和客户端关联起来了。

## 2.7 Servlet Filter （Servlet 过滤器）

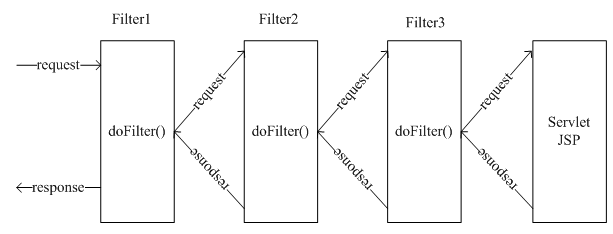
**什么是Filter**：

过滤器位于客户端和web应用程序之间，用于检查和修改两者之间流过的请求和响应。

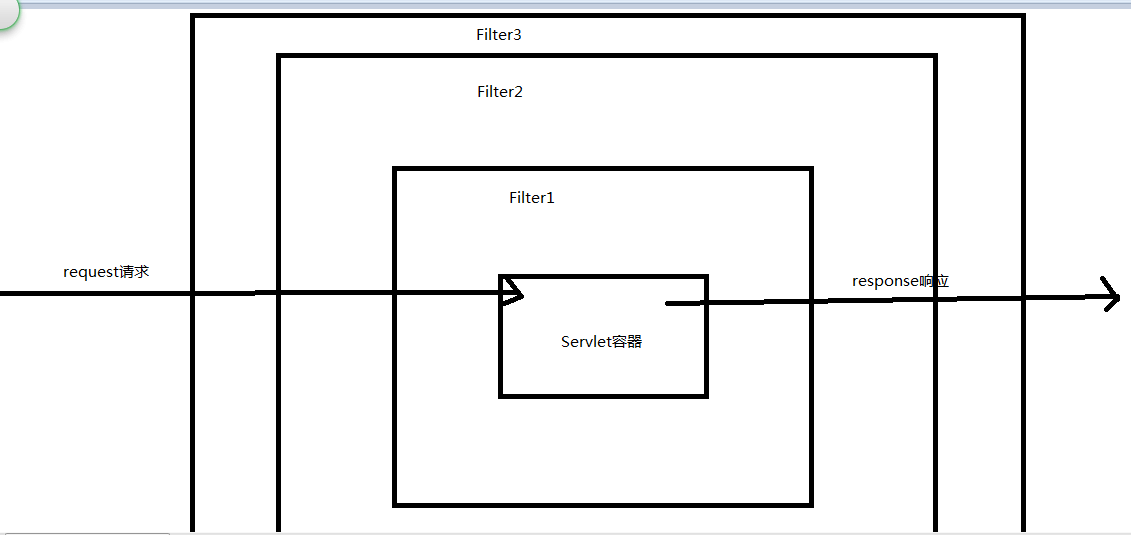
在请求到达Servlet/JSP之前，过滤器截获请求。在响应送给客户端之前，过滤器截获响应。

多个过滤器形成一个过滤器链，过滤器链中不同过滤器的先后顺序由部署文件web.xml中过滤器映射<filter-mapping>的顺序决定。

最先截获客户端请求的过滤器将最后截获Servlet/JSP的响应信息



我的理解是： 过滤器就是在Servlet的外面包裹的一层盒子如下图。



请求如上图一样，过滤器就像一层一层盒子套在Servlet这个盒子上，请求与响应就贯穿这些盒子的管道一样。越是套在外面的盒子越早接收到request请求，但是他越晚接收到response响应。

可以为一个Web应用组件部署多个过滤器，这些过滤器组成一个过滤器链，每个过滤器只执行某个特定的操作或者检查。这样请求在到达被访问的目标之前，需要经过这个过滤器链。

**实现过滤器**：

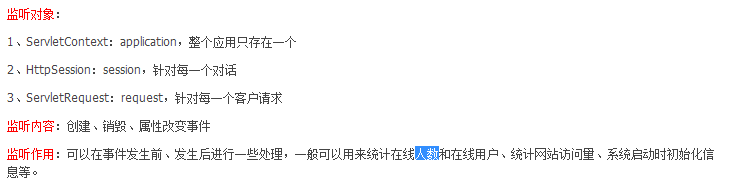
在Web应用中使用过滤器需要**实现javax.servlet.Filter接口**，实现Filter接口中所定义的方法，并**在web.xml中部署过滤器**。

## 2.7 ServletListener（Servlet监听器）

Servlet监听器是Servlet规范中定义的一种特殊类，用于监听ServletContext、HttpSession和ServletRequest等域对象的创建与销毁事件，以及监听这些域对象中属性发生修改的事件。

监听器Listener就是在application,session,request三个对象创建、销毁或者往其中添加修改删除属性时自动执行代码的功能组件。

**Listener是Servlet的监听器，可以监听客户端的请求，服务端的操作等**。



监听器的基本使用：

创建步骤：

1、创建一个实现监听器接口的类

2、配置web.xml文件，注册监听器(现在也可可以使用@WebListener注解来直接配置监听器)



各种方法：查看

<http://www.cnblogs.com/hellojava/archive/2012/12/26/2833840.html>

Session实例demo：

/\*\*

\* 这个监听器的目的就是为了统计在线的用户人数

\* 这里我们要获取所有的在线人数，这就要使用跨用户的存储空间(Servlet上下文)来存储

\* 而一个用户，就使用一个会话的创建来表示。用户的退出，就用session的销毁来判断

\*

\*/

@WebListener

**public** **class** FirstImpl **implements** HttpSessionListener {

/\*\*

\* **@see** HttpSessionListener#sessionCreated(HttpSessionEvent)

\* 这里的就是session创建的时候

\* 我们首先要获取到上下文这个存储空间

\*/

**public** **void** sessionCreated(HttpSessionEvent event) {

// **TODO** Auto-generated method stub

ServletContext context= event.getSession().getServletContext();

**int** count=0;

//第一个用户登录系统的时候，ServletContext中没有信息，获取就会变空

Object onLineCount=context.getAttribute("onLineCount");

**if** (onLineCount==**null**) {

context.setAttribute("onLineCount", count);

}

count=Integer.*parseInt*(onLineCount.toString());

//在这个方法是当session创建的的时候触发的，所以在这里执行这段代码就表示有用户登录了，所以记录数要加1

count++;

context.setAttribute("onLineCount", count);

}

/\*\*

\* @see HttpSessionListener#sessionDestroyed(HttpSessionEvent)

\* 这里是session失效

\*/

public void sessionDestroyed(HttpSessionEvent event) {

// TODO Auto-generated method stub

ServletContext context = event.getSession().getServletContext();

int count = Integer.parseInt(context.getAttribute("onLineCount").toString());

count--;

context.setAttribute("onLineCount", count);

}

使用实例二：

Spring使用ContextLoaderListener加载ApplicationContext配置信息

ContextLoaderListener的作用就是启动Web容器时，自动装配ApplicationContext的配置信息。因为它实现了ServletContextListener这个接口，在web.xml配置这个监听器，启动容器时，就会默认执行它实现的方法。

Spring使用Log4jConfigListener配置Log4j日志：

## 2.8 Servlet的文件上传

1. 文件上传需要使用的jar包：commons-fileupload-1.3.2、commons-io-2.5.jar。
2. 要使用文件上传页面表单要写成

<form method="post" action="/TomcatTest/UploadServlet" enctype="multipart/form-data">

<input type="file" name="uploadFile" />

1. 编写接收上传文件的逻辑Servlet。

Demo;

**private** **void** fileControl(HttpServletRequest request,

HttpServletResponse response) **throws** ServletException {

// 要接收文件，需要进行以下步骤：(主要分两步： 解析请求，对请求的信息就行判断与处理)

// 1.在解析请求之前先判断请求类型是否为文件上传类型

Boolean isMultipart = ServletFileUpload.*isMultipartContent*(request);

// 2.文件上传处理工厂

FileItemFactory factory = **new** DiskFileItemFactory();

// 3.创建文件上传处理器

ServletFileUpload upload = **new** ServletFileUpload(factory);

// 开始解析请求信息

List items = **null**;

**try** {

upload.parseRequest(request);

} **catch** (FileUploadException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

// 对所有请求信息进行判断

Iterator iter = items.iterator();

**while** (iter.hasNext()) {

FileItem item = (FileItem) iter.next();

// 在这里面取出每条信息判断是否是文件格式，或者是普通文件格式

**if** (item.isFormField()) {

String fieldName = item.getFieldName();

String value = item.getString();

request.setAttribute(fieldName, value);

} **else** {

// 这里对接收到的文件进行处理

item.getName(); // 获取文件上传时的文件名

// 我们需要构建上传文件的真正保存文件构建一个Flie，

// 当构建后，可以调用FileItem的write()方法来将上传的文件写入到磁盘中

}

}

## 2.9 JSP 技术

### 2.9.1 JSP语法详解

1. 确认Servlet与JSP的区别

Servlet的作用： 用java语言开发动态资源的技术！！！

Jsp的作用：用java语言（+html语言）开发动态资源的技术！！！

Jsp就是servlet！ Servlet容器在编译阶段会将JSP翻译成Servlet。

1. JSP的特点

1）jsp的运行必须交给tomcat服务器！！！！

tomcat的work目录： tomcat服务器存放jsp运行时的临时文件

2）jsp页面既可以写html代码，也可以写java代码。(使用**<% JAVA代码 %>**来书写)

（html页面不能写java代码 。而jsp页面可以写java代码）

可以把jsp页面当做html页面在tomcat中访问！！！

1. Jsp的执行过程

问题： 访问http://localhost:8080/day12/01.hello.jsp  如何显示效果？

1）访问到01.hello.jsp页面，tomcat扫描到jsp文件，在%tomcat%/work把jsp文件翻译成java源文件

(01.hello.jsp   ->   \_01\_hello\_jsp.java) （翻译）

2）tomcat服务器把java源文件编译成class字节码文件 （编译）

（\_01\_hello\_jsp.java  ->  \_01\_hello\_jsp.class）

3）tomcat服务器构造\_01\_hello\_jsp类对象

4）tomcat服务器调用\_01\_hello\_jsp类里面方法，返回内容显示到浏览器。

1. 问题： 为什么Jsp就是servlet？

jsp翻译的java文件：

public final **class \_01\_hello\_jsp**extends org.apache.jasper.runtime.**HttpJspBase**

    implements org.apache.jasper.runtime.**JspSourceDependent** {

HttpJspBase类：

public abstract class org.apache.jasper.runtime.**HttpJspBase** extends javax.servlet.http.**HttpServlet** implements javax.servlet.jsp.**HttpJspPage** {

结论： Jsp就是一个servlet程序！！！

servlet的技术可以用在jsp程序中

jsp的技术并不是全部适用于servlet程序！

1. **JSP的基本语法**

(要注意这些JSP的基本语法的实质，必须记住的JSP的本质就是Servlet，这些JSP语法会按照一定的规则翻译成Servlet中的Java代码。我们需要从Servlet的视角去看JSP)

**表达式**：

语法：<%=变量或表达式%>

作用： 向浏览器输出变量的值或表达式计算的结果

注意：

1）表达式的原理就是翻译成out.print(“变量” );通过该方法向浏览器写出内容

2）表达式后面不需要带分号结束。

**Jsp脚本**：

语法：<%java代码 %>

作用： 执行java代码

注意：

1. 原理把脚本中java代码原封不动拷贝到\_jspService方法中执行。

**Jsp的声明**：

语法：<%! 变量或方法 %>

作用： 声明jsp的变量或方法

注意:

1. 变量翻译成成员变量，方法翻译成成员方法，声明在对应的Servlet中。

**Jsp的注释**：

语法： <%!-- jsp注释 --%>

注意;

1. html的注释会被翻译和执行。而jsp的注释不能被翻译和执行。
2. **JSP的三大指令**

include指令：

作用： 在当前页面用于包含其他页面

语法： <%@include file="common/header.jsp"%>

注意：

1）原理是把被包含的页面（header.jsp）的内容翻译到包含页面(index.jsp)中,合并成翻译成一 个java源文件，再编译运行！！，这种包含叫静态包含（源码包含）

2）如果使用静态包含，被包含页面中不需要出现全局的html标签了！！！（如html、head、 body）

page指令

作用： 告诉tomcat服务器如何翻译jsp文件

<%@ page

language="java"   --告诉服务器使用什么动态语言来翻译jsp文件

import="java.util.\*" --告诉服务器java文件使用什么包

导入包，多个包之间用逗号分割

pageEncoding="utf-8"  --告诉服务器使用什么编码翻译jsp文件（成java文件）

 contentType="text/html; charset=utf-8" 服务器发送浏览器的数据类型和内容编码

注意：在开发工具中，以后只需要设置pageEncoding即可解决中文乱码问题

taglib指令

taglib指令是用来在当前jsp页面中导入第三方的标签库

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

prefix：指定标签前缀，这个东西可以随意起名

uri：指定第三方标签库的uri（唯一标识）

当然，需要先把第三方标签库所需jar包放到类路径中。

**7 JSP的九大内置对象**

什么是内置对象？

在jsp开发中，会频繁使用到一些对象

。例如HttpSession,ServletContext,ServletContext,HttpServletRequet。如果我们每次要使用这些对象都去创建这些对象就显示非常麻烦。所以Sun公司设计Jsp时，在jsp页面加载完毕之后就会自动帮开发者创建好这些对象，而开发者只需要直接使用这些对象调用方法即可！，这些创建好的对象就叫内置对象！！！！

简单来说就是在JSP页面中我们课可以直接使用的，由Servlet容器已经提供好的一些对象。



<http://www.cnblogs.com/handsomecui/p/6151602.html>

这里面需要注意的是：

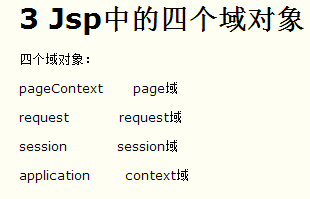
pageContext对象的类型是PageContext，叫jsp的上下文对象

我们可以使用pageContext对象来获取其他八个内置对象。

**8 JSP的四大域对象**

域对象作用：

保存数据 和 获取数据 ，用于数据共享。



域对象方法：

setAttribute("name",Object) 保存数据

getAttribute("name")  获取数据

removeAttribute("name") 清除数据

域对象作用范围：

page域： 只能在当前jsp页面中使用（当前页面）

request域： 只能在同一个请求中使用（转发）

session域： 只能在同一个会话（session对象）中使用（私有的）

context域： 只能在同一个web应用中使用。（全局的）

## 3.0 EL 表达式

1. EL的作用：

jsp的核心语法： jsp表达式 <%=%>和 jsp脚本<% %>。

但是在JSP页面中书写Java，看着很混乱。开发jsp的原则： 尽量在jsp页面中少写甚至不写java代码。使用EL表达式替换掉jsp表达式

EL表达式作用： 向浏览器输出**域对象中的变量值**或表达式计算的结果！！！(这里就需要使用到JSP的四个域对象)

**语法：** **${变量或表达式}**

### 3.0.1 EL语法

1. 输出基本数据类型变量

从四个域获取： (最常用)

${name}

指定域获取

${pageScope.name}

域范围： pageScoep / requestScope / sessionScope / applicationScope

2）输出对象的属性值

${stu.name}

1. 输出集合对象

List 和 Map

1. EL表达式计算

# Java 框架

首先需要知道的就是框架的底层都是Servlet和JSP

## 1.0 struts2 工作原理及使用

### 1.0.0 struts的helloWord

一个基本的struts程序，由action和struts的配置文件组成。

当然使用struts框架必须要加入核心jar包

(在struts的开发jar包中解压app下的struts2-blank.war 可以直接获取核心jar包)

基本步骤：

1. 建立动态web工程，在web.xml中加入

<filter>

<filter-name>Struts2</filter-name>

<filter-class>

org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter

</filter-class>

</filter>

<filter-mapping>

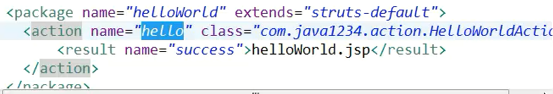
<filter-name>Struts2</filter-name>

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

　　</filter-mapping>

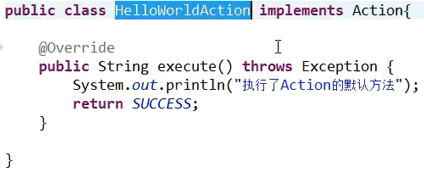
在web.xml中配置struts的过滤器，来接收，处理请求。

1. 配置struts的配置文件struts.xml



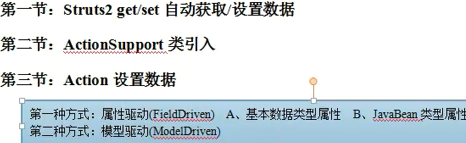
在这里配置action .在第一步中配置的过滤器会根据请求来 struts.xml中的配置文件来查找对应的action。然后会根据action下的result来响应请求页面

1. 配置文件中配置的action，类似于我们的Servlet。要在其中书写我们的处理逻辑。



### 1.0.1 struts的核心知识

(详细笔记进入“http://lib.csdn.net/my/structure/Javaweb” 我的**知识库**查看)



自动获取和设置数据：

<http://blog.csdn.net/feidie436/article/details/70266912>

总结下程序的流程：我们在处理类HelloWorldAction中定义的一个String类型的名为name变量，并生成了name变量的get和set函数。当我们在浏览器中请求hello，并带着了一个名为name的参数值，拦截器拦截到我们的请求后根据配置把给请求交给HelloWorldAction处理，请求中带了参数值，拦截器会调用HelloWorldAction中同名变量（也就是name变量）的set函数，从而把传入的值赋值给类中的变量。HelloWorldAction类执行默认的execute函数并返回success，根据配置跳转到helloWorld.jsp页面。这个页面上使用$表达式取name变量的值，这里拦截器会执行处理类中同名变量（也就是name变量）的get函数，得到处理类中对应变量的值。

继承了ActionSupport类：

ActionSupport类 中实现了众多功能接口，继承这个类有助于我们action的开发

Action设置数据：

常用上面的第一种方式中的方法。

第二种方式只能接受一种类型的Javabean 。

### 1.0.2 struts处理传入多个值



## 2.0 Spring 的工作原理及使用

## 3.0 Hibernate的工作原理及使用

是打发