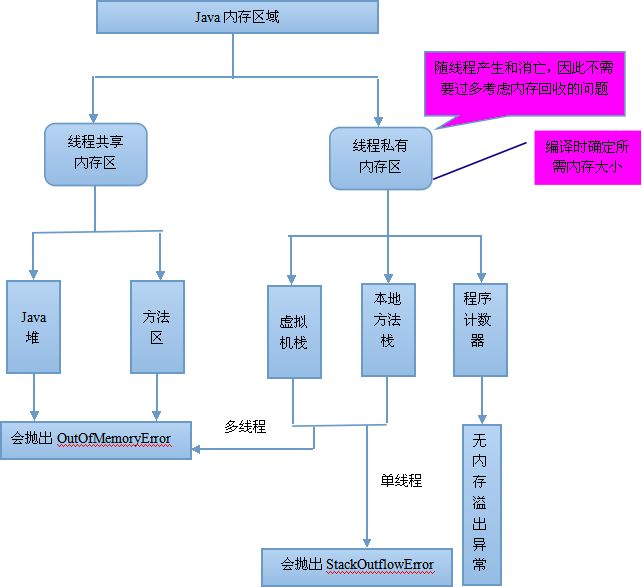
**JAVA堆栈**

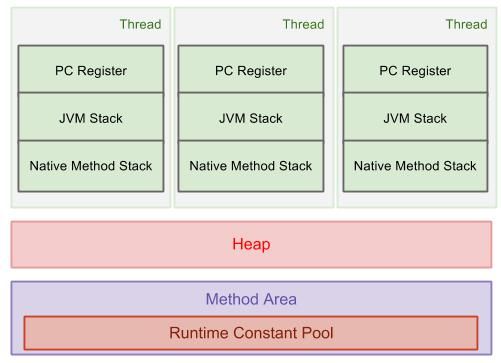
**栈内存用来存储局部变量和方法调用**

**堆内存用来存储Java中的对象**。无论是成员变量，局部变量，还是类变量。它们指向的对象都存储在堆内存中

栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即**栈内存可以理解成线程的私有内存**

堆内存中的对象对所有线程可见，**堆内存中的对象可以被所有线程访问**





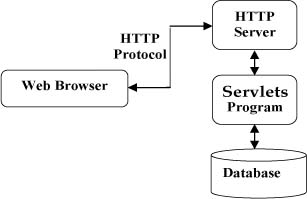
**Servlet简介**

Java Servlet是运行在WEB服务器或者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服务器上的数据或应用程序之间的中间层

使用Servlet，您可以收集来自网页表单的用户输入，呈现来自数据库或者其他源的记录，还可以动态创建网页

Java Servlet通常情况下使用CGI实现的程序可以达到异曲同工的效果，但是相比于CGI，**Servlet有以下几点优势:**

1. 性能明显更好
2. Servlet在Web服务器的地址空间内执行，这样它就没有必要再创建一个单独的进程来处理每个客户端请求
3. Servlet是独立于平台的，因为它们是用Java编写的
4. 服务器上的Java安全管理器执行了一系列限制，以保护服务器计算机上的资源。因此Servlet是可信的
5. Java类库的全部功能对Servlet来说都是可用的。他可以通过sockets和RMI机制与applets、数据库或其他软件进行交互



**Servlet执行以下主要任务：**

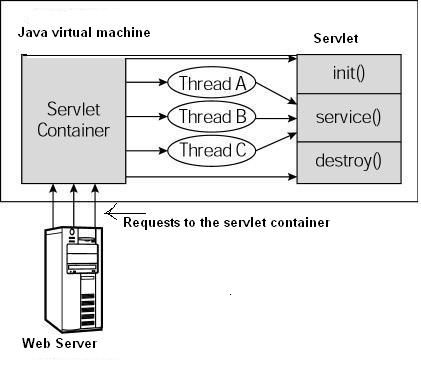
1. 读取客户端发送的显示的数据，这包括网页上的HTML表单，或者也可以是来自applet或自定义的HTTP客户端程序的表单
2. 读取客户端发送的隐式的HTTP请求数据，这包括cookies、媒体类型和浏览器能理解的压缩格式等等
3. 处理数据并生成结果。这个过程可能需要访问数据库，执行RMO或者CORBA调用，调用Web服务，或者直接计算得出对应的响应
4. 发送显示的数据(即文档)到客户端(浏览器)。该文档的格式可以是多种多样的，包括文本文件(HTML/XML)、二进制文件(GIF图像)、EXECL等
5. 发送隐式的HTTP响应到客户端(浏览器)。这包括高速浏览器或其他客户端被返回的文档类型(例如HTML)，设置cookies和缓存参数，已经其他类型的任务

**Servlet包**

Java Servlet是运行在带有支持Java Servlet规范的解释器的web服务器上的java类

Servlet可以使用javax.servlet和javax.servlet.http包创建，它是Java企业版的标准组成部分，Java企业版是支持大型开发项目的Java类库的扩展版本

Java Servlet就像任何其他的Java类一样已经被创建和编译。在您安装Servlet包并把它们添加到您的计算机上的ClassPath类路径中之后，您就可以通过JDK的Java编译器或任何其他编译器来编译Servlet



**JSP介绍**

JSP全称Java Server Pages，是一种动态网页开发技术。它使用JSP标签在HTML网页中插入java代码。标签通常以<%开头，以%>结束

JSP是一种Java Servlet，主要用于实现Java web应用程序的用户界面部分。网页开发者们通过HTML代码、XHTML代码、XML元素以及嵌入JSP操作和命令来编写JSP

JSP通过网页表单获取用户输入数据、服务数据库及其他数据源，然后动态地创建网页

JSP标签有多种功能，比如访问数据库、记录用户选择信息、服务JavaBeans组件等。还可以在不同的网页中传递控制信息和共享信息

**为什么使用JSP:**

1. 性能更加优越，因为JSP可以直接在HTML网页中动态嵌入元素而不需要单独引用CGI文件
2. 服务器调用的是已经编译好的JSP文件，而不像CGI/Perl那样必须先载入解释器和目标脚本
3. JSP基于Java Servlet API，因此JSP拥有各种强大的企业级Java API，包括JDBC、JNDI、EJB、JAXP等待
4. JSP页面可以与处理业务逻辑的Servlet一起使用，这种模式被Java Servlet模板引擎所支持

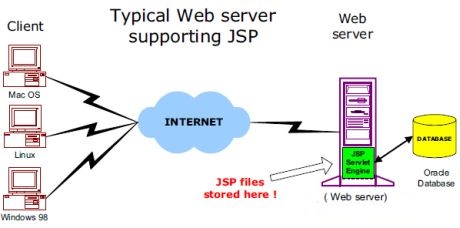
JSP是Java EE不可或缺的一部分，是一个完整的企业级应用平台。这意味着JSP可以用最简单的方式来实现最复杂的应用

**JSP优势:**

1. 与APS相比，JSP有两大优势。首先，动态部分用Java编写，而不是VB或其他MS专用语言。所以更加强大与易用。第二点就是JSP易于移植到非MS平台上
2. 与纯Servlet相比；JSP可以很方便的编写或修改HTML页面而不用去面对大量的println语句
3. 与SSI相比:SSI无法使用表单数据、无法进行数据库链接
4. 与JavaScript相比；虽然JavaScript可以在客户端动态生成HTML，但是很难与服务器交换，因此不能提供复杂的服务，比如访问数据库和图像处理等等
5. 与静态HTML相比；静态HTML不包含动态信息

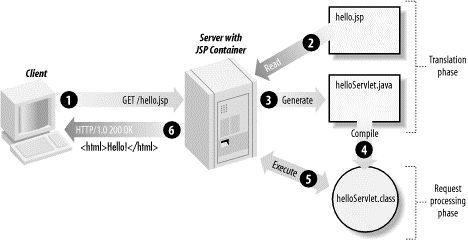
**JSP结构**

网络服务器需要一个JSP引擎，也就是一个容器来处理JSP页面。容器负责截取对JSP页面的请求。JSP容器与Web服务器协调合作，为JSP的正常运行提供必要的运行环境和其他服务，并且能真确识别专属于JSP网页的特色元素



**JSP处理步骤：**

1. 浏览器发生http请求给服务器
2. WEB服务器识别出是一个对JSP网页的请求，并且将该请求传递给JSP引擎。通过使用URL或者.jsp文件来完成
3. JSP引擎从磁盘中载入JSP文件，然后将它们转化为Servlet。这种转化只是简单地将所有模板文本该用println()语句，并且将所有的JSP元素转化成Java代码
4. JSP引擎将Servlet编译成可执行类，并且将原始请求传递给Servlet引擎
5. Web服务器的某组件将会调用Servlet引擎。然后载入并执行Servlet类。在执行过程中，Servlet产生HTML格式的输出并将其内嵌于HTTP response中交给Web服务器
6. Web服务器以静态HTML网页的形式将HTTP response返回到您的浏览器中
7. 最终，Web浏览器处理HTTP response中动态产生的HTML网页，就好像在处理静态页面一样



一般情况下，JSP引擎会检查JSP文件对应的Servlet是否已经存在，并且检查JSP文件的修改日期是否早于Servlet。如果JSP文件的修改日期早于对应的Servlet，那么容器就可以确定JSP文件没有被修改过并且Servlet有效。

总的来说，JSP我要就是用另一种方式来编写Servlet而不用成为Java编程高手。除了解释阶段外，JSP我们几乎可以被当成一个普通的Servlet来对待

**JSP生命周期**

JSP生命周期就是从创建到销毁的整个过程，类似于Servlet生命周期，区别在于JSP生命周期还包括将JSP文件编译成servlet

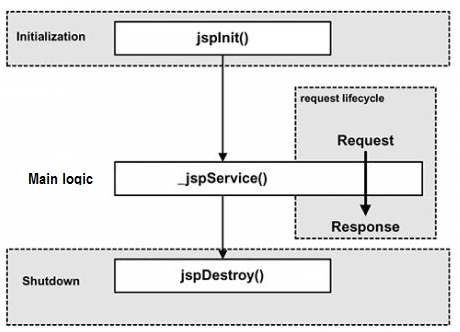
JSP生命周期主要以下几个过程：

**编译阶段：**servlet容器编译servlet源文件，生成servlet类

**初始化阶段：**加载与JSP对应的servlet类，创建其实例，并调用它的初始化方法

**执行阶段：**调用与JSP对应的servlet实例的服务方法

**销毁阶段：**调用与JSP对应的servlet实例的销毁方法，然后销毁servlet实例



**JSP编译**

当浏览器请求JSP页面时，JSP引擎首先去检查需要编译这个文件。如果这个文件没有被编译过，或者再上次编译后被更改过，则编译这个JSP文件

编译过程包括三个步骤：

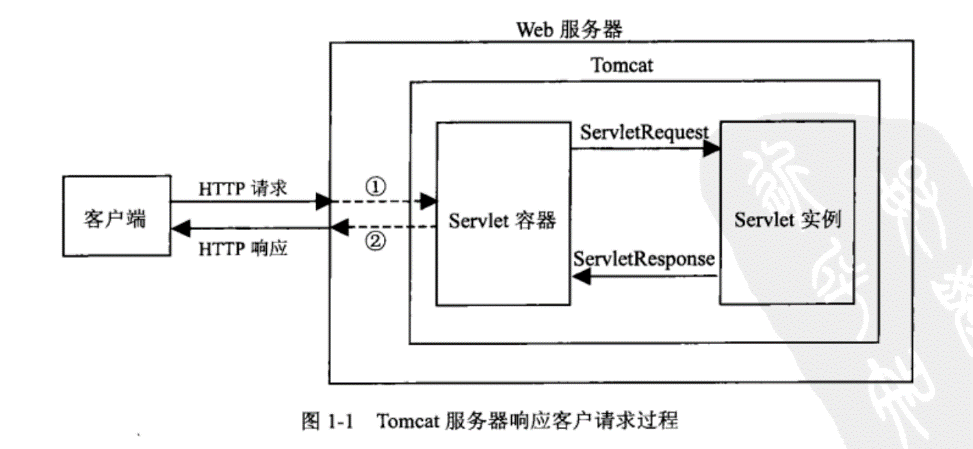
解析JSP文件

将JSP文件转为servlet

编译servlet

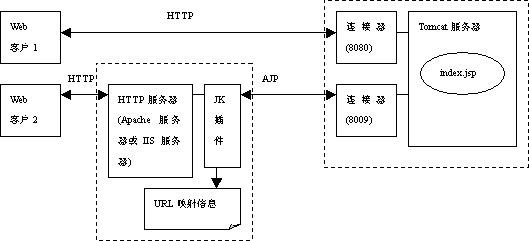
**Tomcat介绍**

Tomcat是Web应用服务器，是一个Servlet/JSP容器。Tomcat作为Servlet容器，负责处理客户请求，把请求传递给Servlet，并将Servlet的响应传回给客户。而Servlet是一种运行在支持Java语言的服务器上的组件。Servlet最常见的用途是扩展Java Web服务器功能，提供非常安全的，可移植的，易于使用的CGI替代品



1. Tomcat将http请求文件接收并解析，然后封装成HttpServletResponse类型的request对象。所有的HTTP头数据可以通过request对象调用对应的方法查询到
2. Tomcat同时会要响应的信息封装为HttpServletResponse类型的response对象，通过设置response属性就可以控制要输出到浏览器的内容，然后将response交给tomcat，tomcat就会将其变成响应文本的格式发生给浏览器

Java Servlet API是Servlet容器(tomcat)和servlet之间的接口，它定义了servlet的各种方法。还定义了Servlet容器传送给Servlet的对象类，其中最重要的就是ServletRequest和ServletResopne。



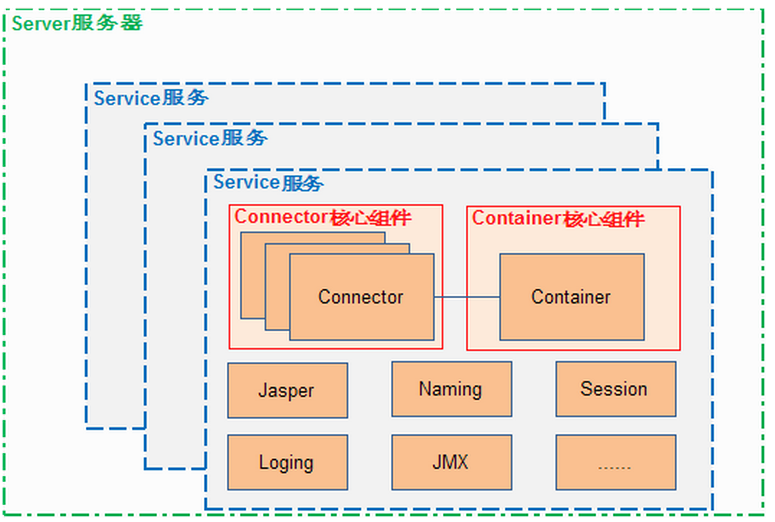
**Tomcat架构**

Tomcat架构核心内容是**Connector**和**Container**组件

Server元素在最顶层，代表整个Tomcat容器；一个Server元素中可以有一个或多个Service元素。

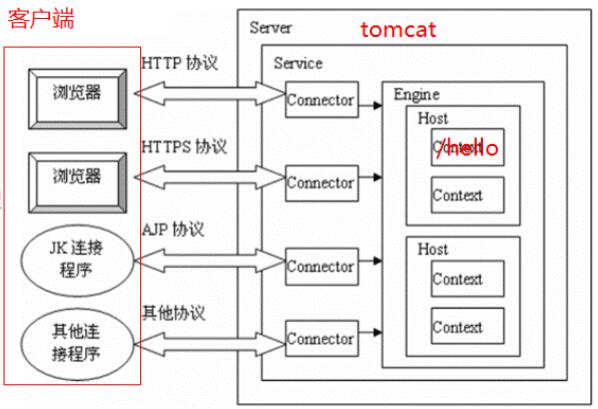
Service在Connector和Engine外面包了一层，把它们组装在一起，对外提供服务。一个Service可以包含多个Connector，但是只能包含一个Engine；Connector接收请求，Engine处理请求。

Engine、Host和Context都是容器，且 Engine包含Host，Host包含Context。每个Host组件代表Engine中的一个虚拟主机；每个Context组件代表在特定Host上运行的一个Web应用。

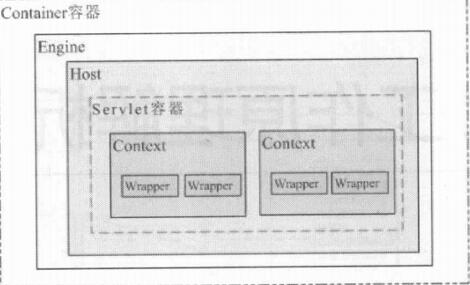


1. **Connector**

当从浏览器传过来http请求后，web容器回去创建一个request和response对象，在Tomcat中，负责该创建工作的是Connector组件



1. **Container**



Container容器有四个子容器:Engine、Host、Context和Wrapper，他们之间是父子包含关系

1. **Engine**
2. **Host(虚拟主机)**
3. **Context(所属的web应用上下文)**
4. **Wrapper(针对的是每个具体的Servlet，他是最直接和Servlet打交道的)**

**Tomcat Server.xml配置文件**

<?xml version="1.0" encoding="UTF-8"?>

<!--Server-->

<Server port="8005" shutdown="SHUTDOWN">

<Service name="Catalina">

<!-- A "Connector" represents an endpoint by which requests are received

and responses are returned. Documentation at :

Java HTTP Connector: /docs/config/http.html (blocking & non-blocking)

Java AJP Connector: /docs/config/ajp.html

APR (HTTP/AJP) Connector: /docs/apr.html

Define a non-SSL HTTP/1.1 Connector on port 8080

-->

<Connector URIEncoding="GB18030" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>

<!-- A "Connector" using the shared thread pool-->

<!--

<Connector executor="tomcatThreadPool"

port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" />

-->

<!-- Define a SSL HTTP/1.1 Connector on port 8443

This connector uses the JSSE configuration, when using APR, the

connector should be using the OpenSSL style configuration

described in the APR documentation -->

<!--

<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"

maxThreads="150" scheme="https" secure="true"

clientAuth="false" sslProtocol="TLS" />

-->

<!-- Define an AJP 1.3 Connector on port 8009 -->

<Connector port="8009" protocol="AJP/1.3" redirectPort="8443"/>

<!-- An Engine represents the entry point (within Catalina) that processes

every request. The Engine implementation for Tomcat stand alone

analyzes the HTTP headers included with the request, and passes them

on to the appropriate Host (virtual host).

Documentation at /docs/config/engine.html -->

<Engine defaultHost="localhost" name="Catalina">

<Host appBase="webapps" autoDeploy="true" name="localhost" unpackWARs="true">

<!-- Access log processes all example.

Documentation at: /docs/config/valve.html

Note: The pattern used is equivalent to using pattern="common" -->

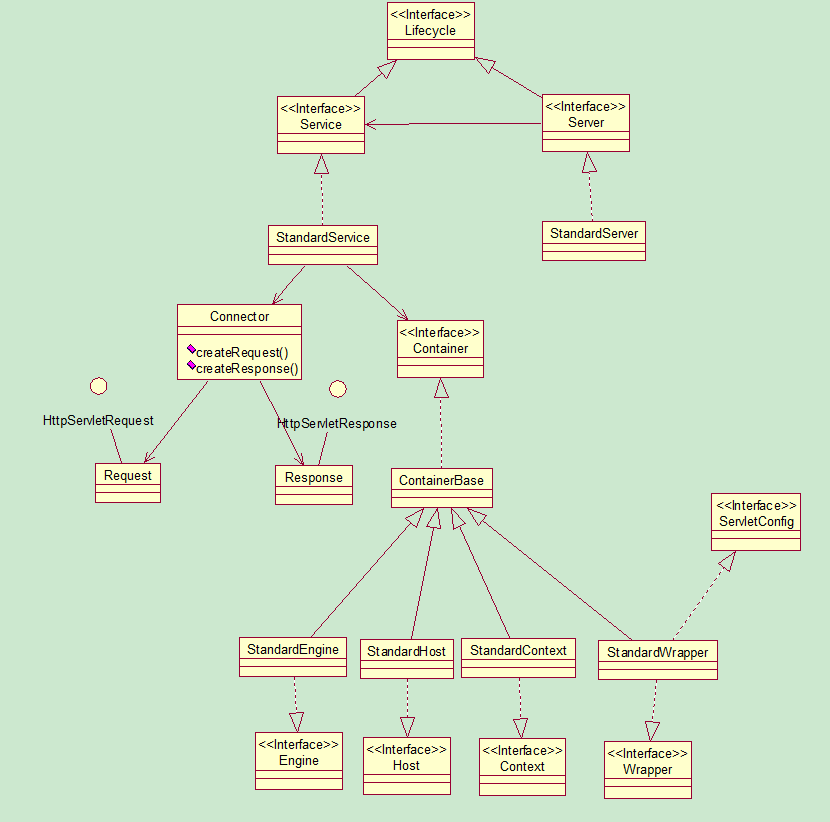
<Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs" pattern="%h %l %u %t &quot;%r&quot; %s %b" prefix="localhost\_access\_log." suffix=".txt"/>

<Context docBase="shop" path="/shop" reloadable="true" source="org.eclipse.jst.jee.server:shop"/></Host>

</Engine>

</Service>

</Server>



**Tomcat设计思想**

1. **职责链模式**

设计Container时使用到了职责链模式，我们可以很明显的看到，Engine、host、context和Wrapper连成一个链。当请求到来时，通过该链传递，最终达到处理请求的子容器。

多个对象有机会处理请求，发出请求和处理请求两者之间耦合关系减弱。对象连成一条链，沿着该链传递请求，直到有对象处理它为止

1. **命令模式**

在设计Connector和Container关系时用到了命令模式

Tomcat中可以有多个Connector，多个Connector和一个Container来构成一个Service。Connector是请求者。而Container是接受命令并且执行的人。

将请求封装成一个对象，我们可以用参数封装各种请求，并且进行请求排队完成及时撤销功能

1. **观察者模式**

这个模式是比较常用的，经常被用来实现”异步处理”。在Tomcat中有很多地方都用到了该模式，比如:Servlet实例的创建、Session管理、Lifecycle等

观察者模式，也叫事件监听机制，也叫发布-订阅模式

**Tomcat连接数与线程池**

在使用tomcat时，经常会遇到连接数、线程数之类的配置问题，要真正理解这些概念，必须先了解Tomcat的连接器（Connector）。

Connector的主要功能，是接收连接请求，创建Request和Response对象用于和请求端交换数据；然后分配线程让Engine（也就是Servlet容器）来处理这个请求，并把产生的Request和Response对象传给Engine。当Engine处理完请求后，也会通过Connector将响应返回给客户端。

可以说，Servlet容器处理请求，是需要Connector进行调度和控制的，**Connector是Tomcat处理请求的主干**，因此Connector的配置和使用对Tomcat的性能有着重要的影响。这篇文章将从Connector入手，讨论一些与Connector有关的重要问题，包括NIO/BIO模式、线程池、连接数等。

根据协议的不同，Connector可以分为HTTP Connector、AJP Connector等，本文只讨论HTTP Connector。

**Tomcat三种高级运行模型**

Connector在处理HTTP请求时，会使用不同的protocol。不同的Tomcat版本支持的protocol不同，其中最典型的protocol包括BIO、NIO和APR（Tomcat7中支持这3种，Tomcat8增加了对NIO2的支持，而到了Tomcat8.5和Tomcat9.0，则去掉了对BIO的支持）。

BIO是Blocking IO，顾名思义是阻塞的IO；NIO是Non-blocking IO，则是非阻塞的IO。而APR是Apache Portable Runtime，是Apache可移植运行库，利用本地库可以实现高可扩展性、高性能；Apr是在Tomcat上运行高并发应用的首选模式，但是需要安装apr、apr-utils、tomcat-native等包。

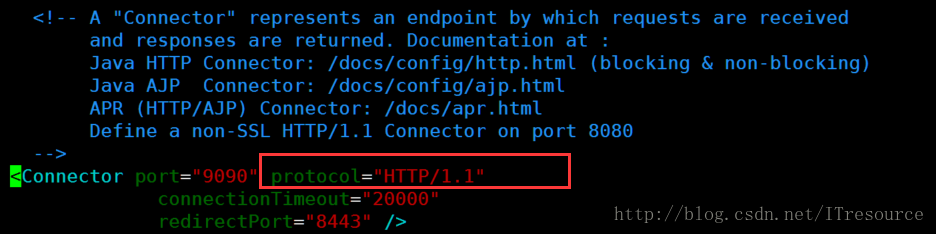
如果没有指定protocol，则使用默认值HTTP/1.1，其含义如下：在Tomcat7中，自动选取使用BIO或APR（如果找到APR需要的本地库，则使用APR，否则使用BIO）；在Tomcat8中，自动选取使用NIO或APR（如果找到APR需要的本地库，则使用APR，否则使用NIO）

1. **BIO（blocking I/O）**

阻塞式I/O操作，表示Tomcat使用的是传统的Java I/O操作(即java.io包及子包)。是基于JAVA的HTTP/1.1连接器，Tomcat7以下版本在默认情况下是以bio模式运行的、一般而言。Bio模式是三种运行模式中性能最低的一种。我们可以通过Tomcat Manager来查看服务器的当前状态

一个线程处理一个请求，缺点:并发量高时，线程数较多，浪费资源

**Server.xml文件中的配置**



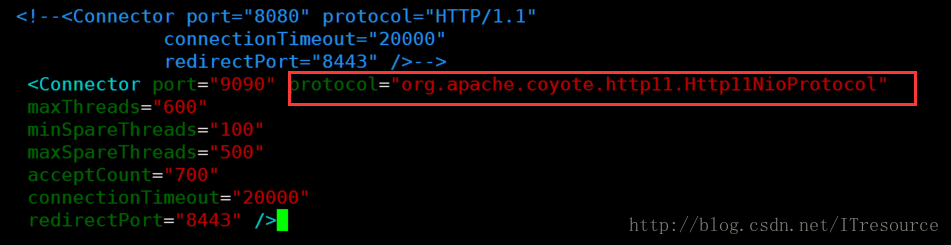
1. **NIO（new I/O）**

是Java SE 1.4及后续版本提供的一种新的I/O操作方式(即java.io包及子包)。Java nio是基于缓冲区、并能提供非阻塞I/O操作的Java API，因此nio也被看错是non-blocking I/O的缩写。它拥有比传统I/O操作(bio)更好的并发运行性能。要让Tomcat以nio模式来运行只需要在Tomcat安装目录/conf/server.xml中将对应的protocol的属性改为org.apache.coyote.http11.Http11NioProtocol即可

利用JAVA的异步请求IO处理，可以通过少量的线程处理大量的请求

**注意:**Tomcat8以上版本在Linux系统中，默认使用的就是NIO模式，不需要额外修改，Tomcat7必须修改Connector配置来启动

**Server.xml文件中修改**



1. **APR（Apache Portable Runtime/Apache 可移植运行时）(安装配置过程相当复杂)**

Tomcat将以JNI的形式调用Apache HTTP服务器的核心动态链接库来处理文件读取或网络传输操作。从而大大提高Tomcat对静态文件的处理能力。Tomcat apr也是在Tomcat上运行高并发应用的首选模式。从操作系统级别来解决异步的IO问题

APR是使用原生C语言编写的非阻塞I/O，利用了操作系统的网络连接功能，速度很快。但是需先按照apr和native，若直接启动就知道apr，能大幅度提升性能。



**将Web项目War包部署到Tomcat服务器**

War包一般是在进行Web开发时，通常是一个网站Project下的所有源码的集合，里面包含前台HTML/CSS/JS的代码，也包含Java的代码。当开发人员在自己的开发机器上调试所有代码并通过后，为了交给测试人员测试和未来进行产品发布，都需要将开发人员的源码打包成War进行发布。War包可以放在Tomcat下的webapps或者word目录下，随着tomcat服务器的启动，它可以自动被解压。