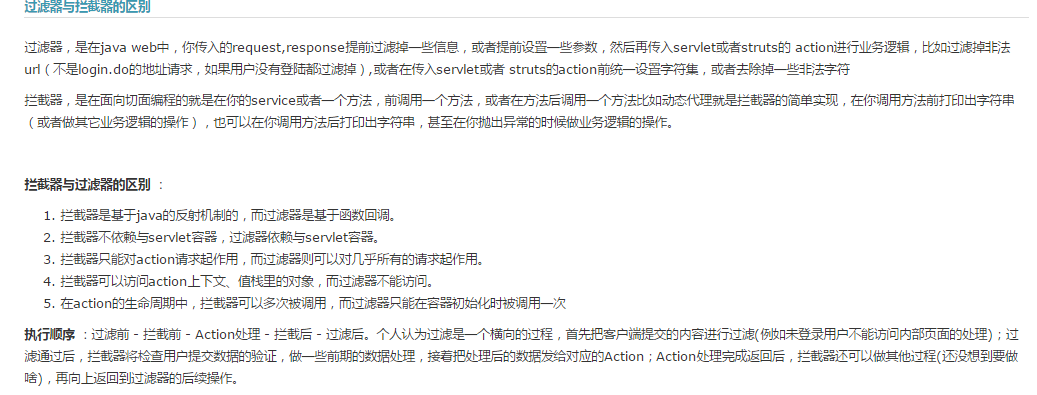
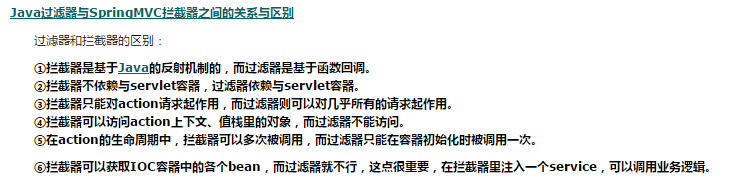
# Servlet

## filter servlet listener interceptor区别

<http://www.cnblogs.com/doit8791/p/4209442.html>

<http://www.cnblogs.com/zhangshitong/p/5995552.html>



 实现filter接口中的doFilter方法就是回调函数。

1 过滤器是基于函数回调，实现filter接口的doFilter方法就是回调函数，而拦截器是基于java反射机制的

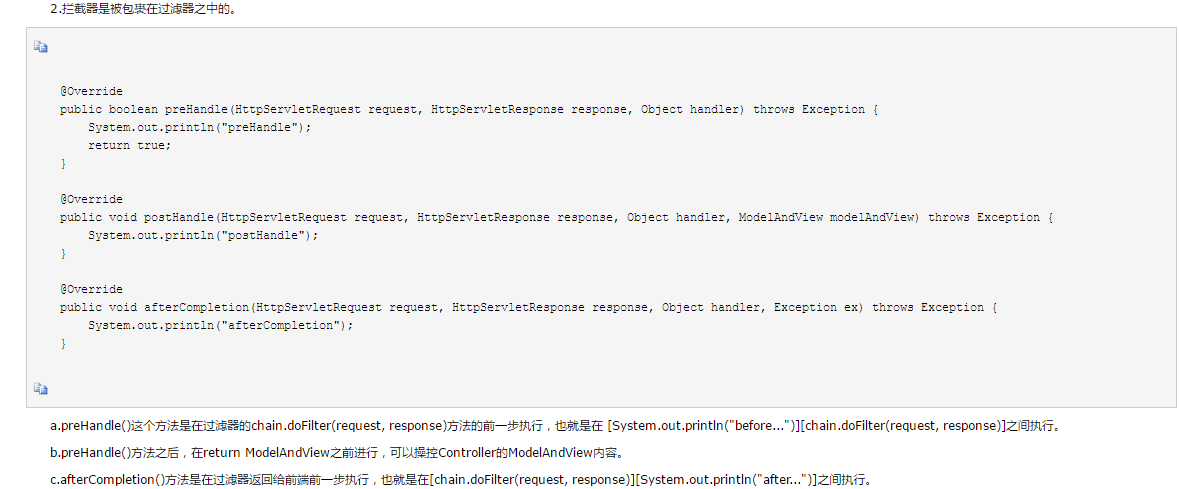
2 过滤器依赖servlet容器，没有servlet容器就无法回调doFilter方法

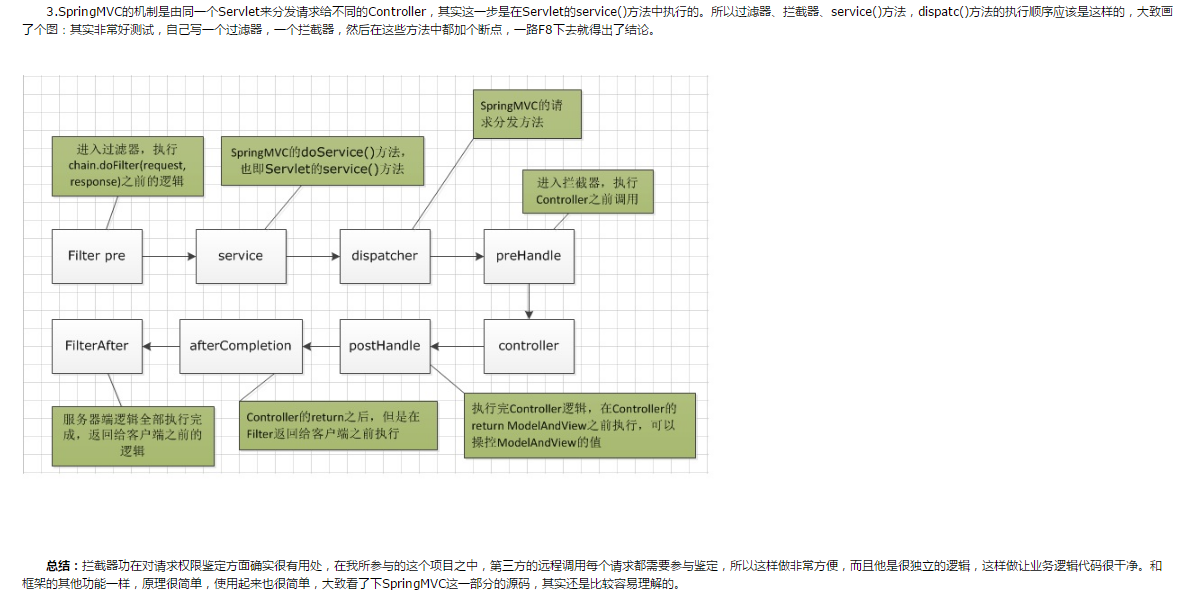
3 过滤器可以对所有的请求起作用，拦截器只能对action请求起作用。Filter的过滤范围比interceptor大，filter除了过滤请求外，通过通配符可以保护页面 图片 文件等。

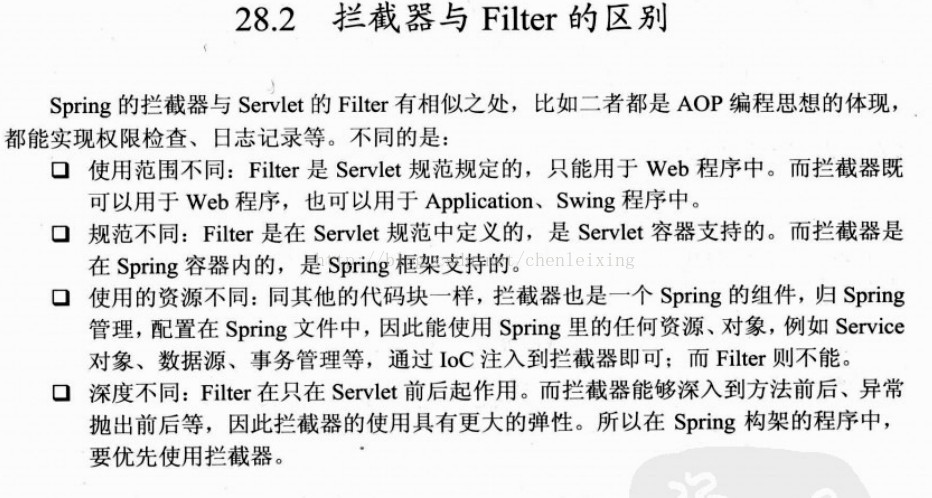
4 拦截器可以访问action的上下文 值栈里的对象，而过滤器不可以

5 在action的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化的时候被调用一次。

6 拦截器可以获取IOC中的bean，过滤器不可以，在拦截器中注入一个service，可以调用业务逻辑。







 所谓回调，就是客户程序C调用服务程序S中的某个函数A，然后S又在某个时候反过来调用C中的某个函数B，对于C来说，这个B便叫做回调函数。例如Win32下的窗口过程函数就是一个典型的回调函数。一般说来，C不会自己调用B，C提供B的目的就是让S来调用它，而且是C不得不提供。由于S并不知道C提供的B姓甚名谁，所以S会约定B的接口规范（函数原型），然后由C提前通过S的一个函数R告诉S自己将要使用B函数，这个过程称为回调函数的注册，R称为注册函数。Web Service以及[**Java**](http://lib.csdn.net/base/java)的RMI都用到回调机制，可以访问远程服务器程序。

## ServletConfig

在servlet配置文件中，可以使用<init-param>标签为servlet配置一些初始化参数。

当servlet配置了初始化参数后，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，并调用servlet的init方法时候，将servletConfig对象传递给Servlet，从而通过ServletConfig对象可以得到当前Servlet的初始化参数的信息。

String getInitParameter(String name)       得到参数

Enumeration getInitParameterNames()      得到所有参数

ServletContext getServletContext()           得到上下文对象

String getServletName()                          得到Servlet名称

可以获得配置文件中配置编码信息（防止硬编码） 数据库连接信息和配置文件信息

## ServletContext

ServletContext是一个Servlet上下文对象，该对象表示当前的web应用环境的信息，一个web应用只会创建一个ServletContext对象。Web容器启动时，ServletContext被创建，web容器关闭时，ServletContext被销毁。一个用户可以有多个request，只有一个session。但是对于ServletContext而言，所有用户共用一个。

在一个web应用中，所有servlet共享一个ServletContext对象，故多个Servlet可以通过ServletContext对象实现数据共享。

ServletContext对象是作为ServletConfig对象成员变量传入到servlet中。通过ServletConfig的getServletContext()方法就可以得到ServletContext对象。ServletConfig对象是在调用init方法之前创建的，在ServletContext对象创建之前。

换言之，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。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.以 /项目名 请求路径（称为上下文路径）开始的所有请求被发送到与此ServletContext关联的Web应用程序。一个ServletContext对象表示了一个Web应用程序的上下文。

Servlet上下文：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。Servlet可能需要共享他们之间的共有信息。运行于同一服务器的Servlet有时会共享资源，如JSP页面、文件和其他Servlet。

例如：做一个购物类的网站，要从数据库中提取物品信息，如果用session保存这些物品信息，每个用户都访问一便数据库，效率就太低了；所以要用来Servlet上下文来保存，在服务器开始时，就访问数据库，将物品信息存入Servlet上下文中，这样，每个用户只用从上下文中读入物品信息就行了。

this.getServletContext(); 直接获取

this.getServletConfig.getServleyContext(); 间接获取

ServletContext的作用：

1.获取web的上下文路径

　　String getContextPath();

2.获取全局的参数

　　String getInitParameter(String name);

　　Enumeration getInitParameterNames();

3.和域对象相关的

　　void setAttribute(String name,Onject object);

　　Object getAttribute(String name);

　　void removeAttribute(String name);

　　域对象（域对象就是在不同资源之前来共享数据，保存数据，获取数据）

　　ServletContext是我们学习的第一个域对象(Servlet共有三个域对象ServletContext、HttpServletRequest、HttpSession)

4. 请求转发的

　　RequestDispatcher getRequestDispatcher(String path);

　　在Servlet跳转页面:

　　4.1请求重定向（你找我借钱，我没有，你自己去找他借钱）

　　　　1.地址栏**会改变**，变成重定向到的地址

　　　　2.**可以**跳转到项目内的资源，**也可以**跳转项目外的资源

　　　　3.浏览器向服务器发出**两次请求**，那么不能使用请求来作为域对象来共享数据。

　　4.2请求转发（你找我借钱，我没有，我帮你去向他借钱）

　　　　1.地址栏**不会改变**

　　　　2.**只能**跳转到项目内的资源，**不能**跳转项目外的资源。

　　　　3.浏览器向服务器发出**一次请求**，那么可以使用请求作为域对象共享数据。

5.读取web项目的资源文件

　　String getRealPath(String path);

　　InputStream getResourceAsStream(String path);

　　URL getResource(String path);

## ServletContextListener

**ServletContext 被Servlet 程序用来与 Web 容器通信**。例如写日志，转发请求。**每一个 Web 应用程序含有一个Context，被Web应用内的各个程序共享**。因为Context可以用来保存资源并且共享，所以我所知道的 ServletContext 的最大应用是Web缓存----把不经常更改的内容读入内存，所以服务器响应请求的时候就不需要进行慢速的磁盘I/O了。

   ServletContextListener 是ServletContext 的监听者，如果 ServletContext 发生变化，ServletContextListener 就可以监听到，如服务器启动时 ServletContext 被创建，服务器关闭时 ServletContext 将要被销毁。

在JSP文件中，application 是 ServletContext 的实例，由JSP容器默认创建。

Servlet 中调用 getServletContext()方法得到 ServletContext 的实例。

我们使用缓存的思路大概是：

       1. 服务器启动时，ServletContextListener 的contextInitialized()方法被调用，所以在里面创建好缓存。可以从文件中或者从**数据库**中读取取缓存内容生成类，用 ServletContext.setAttribute()方法将缓存类保存在ServletContext 的实例中。

       2. 程序使用 ServletContext.getAttribute()读取缓存。

**如果是 JSP，使用application.getAttribute()**。

**如果是 Servlet，使用 getServletContext().getAttribute()**。

如果缓存发生变化(如访问计数)，你可以同时更改缓存和文件/数据库。或者你等 变化积累到一定程序再保存，也可以在下一步保存。

       3. **服务器将要关闭时，ServletContextListener 的 contextDestroyed()方法被调用**，所以在里面保存缓存的更改。将更改后的缓存保存回文件或者数据库，更新原来的内容。

1. **import** User; //my own
2. classimport DatabaseManager; // my own class
3. **import** javax.servlet.ServletContext;
4. **import** javax.servlet.ServletContextListener;
5. **public** **class** MyContextListener  **implements** ServletContextListener {
6. **private** ServletContext context = **null**;
7. **public** **void** contextInitialized(ServletContextEvent event) {
8. context = event.getServletContext();
9. User user = DatabaseManager.getUserById(1);
10. context.setAttribute("user1", user);
11. }
12. **public** **void** contextDestroyed(ServletContextEvent event) {
13. User user = (User)context.getAttribute("user1");
14. DatabaseManager.updateUserData(user);
15. **this**.context = **null**;
16. }
17. }

## servlet的生命周期

<http://blog.csdn.net/javaloveiphone/article/details/8154791>

**servlet的四个生命周期方法**

1 构造方法：在创建servlet对象的时候调用，只会调用一次。单例

2 init方法：创建完对象后会调用该方法进行初始化操作，只调用一次。

3 service方法：每次请求servlet时，都会调用。

4 destroy方法：在servlet对象销毁后，调用该方法，只调用一次。（服务器重启或者web应用重启）

servlet服务器是由web服务器调用，web服务器收到客户端的请求之后，web服务器首先检查是否已经装载并创建了该servlet实例对象。如果没有，则装载并创建该servlet的实例对象，调用servlet实例对象的init方法对其进行初始化。然后创建一个用于封装http请求消息的httpservletrequest对象和一个代表http响应的httpservletresponse对象，然后调用service()方法并将请求和响应作为参数传递进去。最后当tomcat重启或者应用重启时，会销毁该servlet对象（destroy方法）。

**servlet何时被创建？**

1 在第一次访问某个servlet时，web容器会创建该servlet实例对象。

2 在web.xml中，如果<servlet></servlet>元素中配置了一个<load-on-start>子元素，则容器在启动的时候就会自动加载这些servlet并调用init方法，完成一些全局性的初始化工作。

## servlet的单例多线程

struts2的action是多实例的，就是对于每个请求都会产生一个action对象。因为页面上的数据会存放在action中的成员变量中，如果是单例，那么这些数据在多线程的环境下就会影响。但是用spring管理的struts2的action时，默认是单例的。

在servlet中，采取的是单例模式。servlet容器维护一个线程池，里面存放着工作者线程来响应相应的请求，同时会有一个调度者线程来管理工作者线程。

当容器收到一个servlet请求后，调度线程就会从线程池中取出一个工作者线程，让工作者线程处理这个请求即调用service方法。当这个线程执行的时候，收到另外一个请求，调度线程就会从线程池中取出另外一个工作者线程来响应新的请求。容器不关心请求的是否访问同一个servlet，当多个请求同时访问同一个servlet时，这个servlet的service方法将在多线程中并发执行。并发执行会出现线程安全问题。

**如何同时处理多个请求？**

工作者线程Work Thread:执行代码的一组线程   
调度线程Dispatcher Thread：每个线程都具有分配给它的线程优先级,线程是根据优先级调度执行的   
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，对于Tomcat可以在server.xml中通过元素设置线程池中线程的数目。   
就实现来说：   
调度者线程类所担负的责任是线程的调度，只需要利用自己的属性完成自己的责任。所以该类是承担了责任的，并且该类的责任又集中到唯一的单体对象中。   
而其他对象又依赖于该特定对象所承担的责任，我们就需要得到该特定对象。那该类就是一个单例模式的实现了。工作者线程Work Thread:执行代码的一组线程   
调度线程Dispatcher Thread：每个线程都具有分配给它的线程优先级,线程是根据优先级调度执行的   
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，对于Tomcat可以在server.xml中通过元素设置线程池中线程的数目。   
就实现来说：   
调度者线程类所担负的责任是线程的调度，只需要利用自己的属性完成自己的责任。所以该类是承担了责任的，并且该类的责任又集中到唯一的单体对象中。   
而其他对象又依赖于该特定对象所承担的责任，我们就需要得到该特定对象。那该类就是一个单例模式的实现了。

**如何做到线程安全？**

1变量的线程安全：这里的变量指字段和共享数据(如表单参数值)。

* 1. 将 参数变量本地化。多线程并不共享局部变量.所以我们要尽可能的在servlet中使用局部变量。 例如：String user =   
     “”;   
     user = request.getParameter(“user”);   
     b，使用同步块Synchronized，防止可能异步调用的代码块。这意味着线程需要排队处理。 在使用同步块的时候要尽可能的缩小同步代码的范围，不要直接在sevice方法和响应方法上使用同步，这样会严重影响性能。

2,属性的线程安全：ServletContext，HttpSession，ServletRequest对象中属性   
ServletContext：（线程是不安全的）   
ServletContext是可以多线程同时读/写属性的，线程是不安全的。要对属性的读写进行同步处理或者进行深度Clone()。   
所以在Servlet上下文中尽可能少量保存会被修改（写）的数据，可以采取其他方式在多个Servlet中共享，比方我们可以使用单例模式来处理共享数据。   
HttpSession：（线程是不安全的）   
HttpSession对象在用户会话期间存在，只能在处理属于同一个Session的请求的线程中被访问，因此Session对象的属性访问理论上是线程安全的。   
当用户打开多个同属于一个进程的浏览器窗口，在这些窗口的访问属于同一个Session，会出现多次请求，需要多个工作线程来处理请求，可能造成同时多线程读写属性。   
这时我们需要对属性的读写进行同步处理：使用同步块Synchronized和使用读/写器来解决。   
ServletRequest：（线程是安全的）   
对于每一个请求，由一个工作线程来执行，都会创建有一个新的ServletRequest对象，所以ServletRequest对象只能在一个线程中被访问。ServletRequest是线程安全的。   
注意：ServletRequest对象在service方法的范围内是有效的，不要试图在service方法结束后仍然保存请求对象的引用。

3 使用同步的集合类，如Hash Table等的。

4不要在Servlet中创建自己的线程来完成某个功能。

Servlet本身就是多线程的，在Servlet中再创建线程，将导致执行情况复杂化，出现多线程安全问题。

5在多个servlet中对外部对象(比方文件)进行修改操作一定要加锁，做到互斥的访问。

6 javax.servlet.SingleThreadModel接口是一个标识接口，如果一个Servlet实现了这个接口，那Servlet容器将保证在一个时刻仅有一个线程可以在给定的servlet实例的service方法中执行。将其他所有请求进行排队。如果一个Servlet实现了SingleThreadModel接口，Servlet引擎将为每个新的请求创建一个单独的Servlet实例，这将引起大量的系统开销。SingleThreadModel在Servlet2.4中已不再提倡使用

注意：服务器可以使用多个实例来处理请求，代替单个实例的请求排队带来的效益问题。服务器创建一个Servlet类的多个Servlet实例组成的实例池，对于每个请求分配Servlet实例进行响应处理，之后放回到实例池中等待下此请求。这样就造成并发访问的问题。   
此时,局部变量(字段)也是安全的，但对于全局变量和共享数据是不安全的，需要进行同步处理。而对于这样多实例的情况SingleThreadModel接口并不能解决并发访问问题。 SingleThreadModel接口在servlet规范中已经被废弃了。

# Web服务器 容器 应用服务器

# Tomcat对静态资源处理

对于tomcat来说，资源分为3种：

1 静态资源，如css html js jpg

2 servlet

3 jsp

对于静态资源，tomcat最终会交给DefaultServlet类进行处理。

对于servlet，tomcat最终会交给InvokerServlet类进行处理。

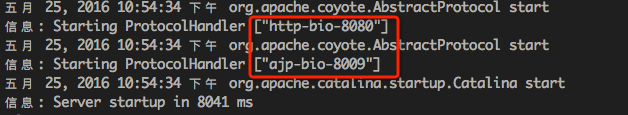
对于jsp，tomcat最终交给JspServlet类进行处理。

# Tomcat

## 三种工作模式

1 BIO模式

阻塞式I/O操作，表示Tomcat使用的是传统**Java**I/O操作(即**Java**.io包及其子包)。Tomcat7及其以下版本默认情况下是以bio模式运行的，由于每个请求都要创建一个线程来处理，线程开销较大，不能处理高并发的场景，在三种模式中性能也最低。启动tomcat看到如下日志，表示使用的是BIO模式：



2 NIO模式

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.nio包及其子包)。是一个基于缓冲区、并能提供非阻塞I/O操作的Java API，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模式来运行比较简单，只需要在Tomcat安装目录/conf/server.xml文件中将如下配置：

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

connectionTimeout="20000"

redirectPort="8443" />

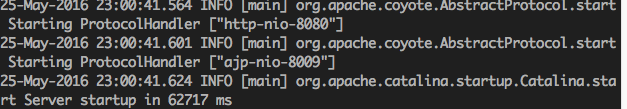
**修改为**

<Connector port="8080" protocol="org.apache.coyote.http11.Http11NioProtocol"

connectionTimeout="20000"

redirectPort="8443" />

注意：Tomcat8以上版本，默认使用的就是NIO模式，不需要额外修改



3 apr模式

从操作系统级别解决异步IO问题，大幅度的提高服务器的处理和响应性能，也是Tomcat运行高并发应用的首选模式。

APR(Apache Portable Runtime/Apache可移植运行时)，是Apache HTTP服务器的支持库。你可以简单地理解为: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，从而大大地提高 Tomcat对静态文件的处理性能。

1 安装依赖库

因为**apr模式本质是使用JNI技术调用操作系统IO接口**，需要用到相关API的头文件

yum install apr-devel

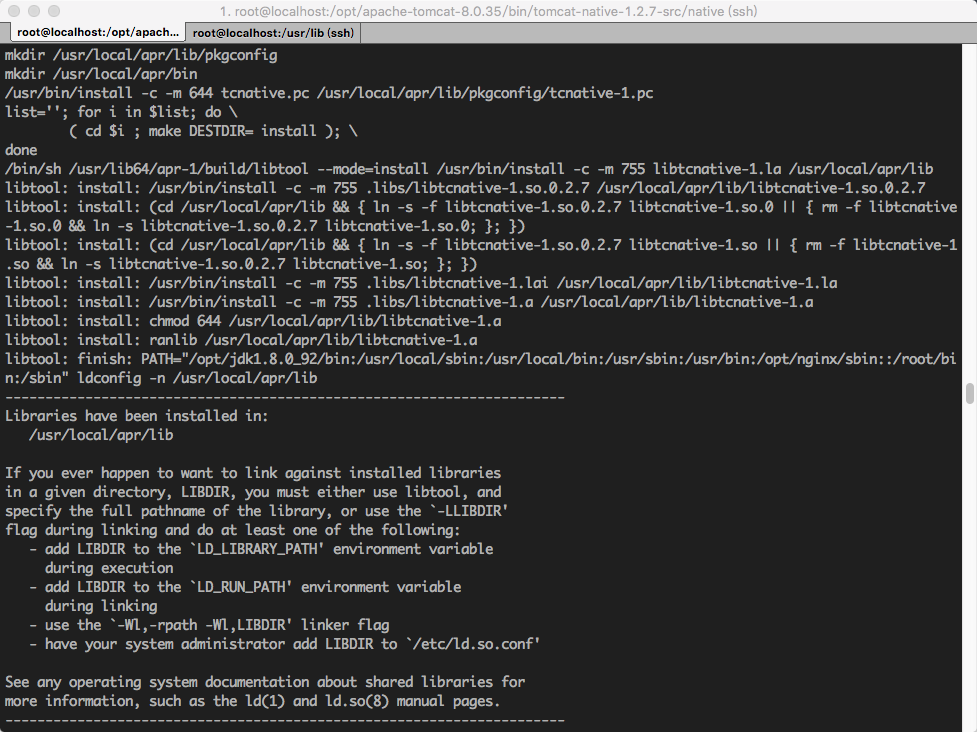
yum install openssl-devel

yum install gcc

yum install make

2 安装apr动态库

进入tomcat的bin目录，解压tomcat-native.tar.gz文件，并进入tomcat-native-1.2.7-src/native目录，执行./configure && make && make install 命令，动态库默认安装在/usr/local/apr/lib目录下，如下图所示：



3 配置APR本地库到系统共享库搜索路径中

方式1 设置LD\_LIBRARY\_PATH和LD\_RUN\_PATH环境变量，指向/usr/local/apr/lib目录，可配置到$HOME/.profile文件中

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/apr/lib

export LD\_RUN\_PATH=$LD\_RUN\_PATH:/usr/local/apr/lib

方式2 拷贝/usr/local/apr/lib目录下所有动态库到/usr/lib或/lib系统共享库搜索目录下即可。

copy /usr/local/apr/lib/libtcnative\* /usr/lib/

方式3（推荐）

编辑$TOMCAT\_HOME/bin/catalina.sh文件，在虚拟机启动参数JAVA\_OPTS中添加java.library.path参数，指定apr库的路径

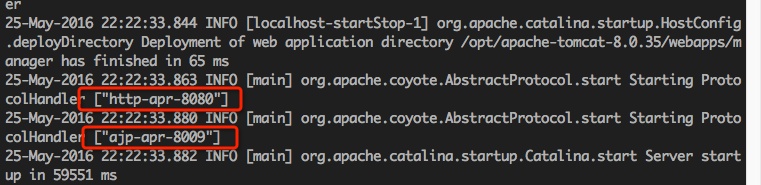
JAVA\_OPTS="$JAVA\_OPTS -Djava.library.path=/usr/local/apr/lib"

Tomcat8以下版本，需要指定运行模式，将protocol从HTTP/1.1改成org.apache.coyote.http11.Http11AprProtocol

<Connector port="8080" protocol="org.apache.coyote.http11.Http11AprProtocol"

connectionTimeout="20000"

redirectPort="8443" />



### APR详解

APR(Apache Portable Runtime)是一个高可移植库，它是Apache HTTP Server 2.x的核心。APR有很多用途，包括访问高级 IO功能(例如sendfile,epoll和OpenSSL)，OS级别功能(随机数生成，系统状态等等)，本地进程管理(共享内存，NT管道和UNIX sockets)。这些功能可以使Tomcat作为一个通常的前台WEB服务器，能更好地和其它本地web技术集成，总体上让Java更有效率作为一个高性能web服务器平台而不是简单作为后台容器。   
        APR的目的如其名称一样，主要为上层的应用程序提供一个可以跨越多操作系统平台使用的底层支持接口库。在早期的Apache版本中，应用程序本身必须能够处理各种具体操作系统平台的细节，并针对不同的平台调用不同的处理函数。随着Apache的进一步开发，Apache组织决定将这些通用的函数独立出来并发展成为一个新的项目。这样，APR的开发就从Apache中独立出来，Apache仅仅是使用APR而已。目前APR主要还是由Apache使用，不过由于APR的较好的移植性，因此一些需要进行移植的C程序也开始使用APR。   
        APR使得平台细节的处理进行下移。对于应用程序而言，它们根本就不需要考虑具体的平台，不管是Unix、Linux还是Window，应用程序执行的接口基本都是统一一致的。因此对于APR而言，可移植性和统一的上层接口是其考虑的一个重点。而APR最早的目的并不是如此，它最早只是希望将Apache中用到的所有代码合并为一个通用的代码库，然而这不是一个正确的策略，因此后来APR改变了其目标。有的时候使用公共代码并不是一件好事，比如如何将一个请求映射到线程或者进程是平台相关的，因此仅仅一个公共的代码库并不能完成这种区分。APR的目标则是希望安全合并所有的能够合并的代码而不需要牺牲性能。

## 优化

如果想要Tomcat达到最优的效果，首先要争取使得[**操作系统**](http://lib.csdn.net/base/operatingsystem)以及网络资源达到最优，并且最好使用高版本的JDK。对于有大量静态页面的系统，采用Apache集成Tomcat的方式，把静态页面交由Apache处理，动态部分交由Tomcat处理，能极大解放Tomcat的处理能力。使用ARP库也能极大的提高Tomcat对静态文件的处理能力。对于并发要求较高的系统，采用Apache加Tomcat集群的方式，将负载分别分担到多个Tomcat上，能很大的提高系统的性能，充分利用硬件资源。同时需要对Tomcat自身进行优化，包括增大内存、调节并发线程数等。

Windows 每个进程中线程数不超过2000

Linux 每个进程中线程数不超过1000

在java中每开启一个线程需要消耗1MB的JVM内存空间用于线程栈

Tomcat的最大并发数与硬件性能和CPU有关。Tomcat默认（7之前）的http是阻塞式的，每个请求都需要创建一个线程去处理。这种模式下并发量受到了线程数的限制。

### 提高tomcat并发能力的方法

1、Apache + Tomcat 结合起来用Apache 负责静态页面，Tomcat负责动态页面，同时减少connectionTimeout的时间，以应对并发量大线程回收来不及的情况。  
2、压力过大的问题，可以做负载均衡，一个TOMCAT无论如何也不可能担当如此多的线程负载，而且JVM过大，其内存管理成本将显著加大。2G的内存，做3-4个TOMCAT实例（512RAM\*4），更为科学合理。  
3、数据库连接池，不少人，都推荐使用C3P0，能提高访问数据库的并发性能好几倍。（有博文称使用tomcat自带的jdbc-pool更好，还没试过）  
4、采用Tomcat集群可以最大程度的发挥服务器的性能，可以在配置较高的服务器上部署多个Tomcat，也可以在多台服务器上分别部署 Tomcat，Apache和Tomcat整合的方式还是JK方式。经过验证，系统对大用户量使用的响应方面，Apache+3Tomccat集群> Apache+2Tomcat集群> Apache集成Tomcat >单个Tomcat。并且采用Apache+多Tomcat集群的部署方式时，如果一个Tomcat出现宕机，系统可以继续使用，所以在硬件系统性能足够优越的情况下，需要尽量发挥软件的性能，可以采用增加Tomcat集群的方式。  
5. 打开KeepAlive支持  
KeepAlive on, KeepAliveTimeout 15 MaxKeepAliveRequests 1000  
根据实际经验，通过Apache和Tomcat集群的方式提高系统性能的效果十分明显，这种方式可以最大化的利用硬件资源，通过多个Tomcat的处理来分担单Tomcat时的压力。  
web server允许的最大连接数还受制于操作系统的内核参数设置，通常Windows是2000个左右，Linux是1000个左右。

## Tomcat优化

<http://itindex.net/detail/53789-tomcats>

http://itindex.net/detail/53998-tomcat-%E4%BC%98%E5%8C%96

### 内存优化

内存优化主要是针对tomcat启动参数优化。

在 tomcat 的启动脚本 catalina.sh 中设置 java\_OPTS 参数。   
　　JAVA\_OPTS参数说明   
　　-server 启用jdk 的 server 版；   
　　-Xms java虚拟机初始化时的最小堆内存；   
　　-Xmx java虚拟机可使用的最大堆内存；   
　　-XX: PermSize 内存永久保留区域   
　　-XX:MaxPermSize 内存最大永久保留区域

Tomcat默认使用的内存为128M

linux修改TOMCAT\_HOME/bin/catalina.sh，在前面加入

JAVA\_OPTS="-XX:PermSize=64M -XX:MaxPermSize=128m -Xms512m -Xmx1024m

-Duser.timezone=Asia/Shanghai"

windows修改TOMCAT\_HOME/bin/catalina.bat，在前面加入

set JAVA\_OPTS=-XX:PermSize=64M -XX:MaxPermSize=128m -Xms512m -Xmx1024m

初始化堆内存为512M，可使用的最大内存1024M

另外，虚拟机堆的大小决定了虚拟机花费在收集垃圾上的时间和频率。如果堆的空间很大，那么一次full gc所消耗的时间会很长，但是频率比较低。调整堆的大小目的是最小化垃圾收集的时间，以在特定时间内最大化处理客户请求。将最小堆内存和最大堆内存设置为同一值，这样可以避免浪费用于调整堆大小所需的vm资源。

### 并发优化

**1 tomcat连接相关参数**

server.xml中：

<Connector port="9027"

　　protocol="HTTP/1.1"

　　maxHttpHeaderSize="8192"

　　minProcessors="100"

　　maxProcessors="1000"

　　acceptCount="1000"

　　redirectPort="8443"

　　disableUploadTimeout="true"/>

**2 调整连接器connector的并发处理能力**

参数说明

maxThreads tomcat可以创建的最大线程数（也是可以处理的同时请求的最大数目，并发数） tomcat使用线程来处理接收的每个请求 maxThreads并不是配置越大越好，这个最大值是受操作系统及相关硬件制约的，并且最大值不一定是最优值。

 QPS（Query Per Second）：每秒查询率QPS是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。我们常常使用 QPS值来衡量一个服务器的性能。   
        QPS = 并发数 / 平均响应时间   
        或者   
        并发数 = QPS \* 平均响应时间

一个系统吞吐量通常由QPS、并发数两个因素决定，每套系统的这两个值都有一个相对极限值，在应用场景访问压力下，只要某一项达到系统最高值，系统的吞吐量就上不去了，如果压力继续增大，系统的吞吐量反而会下降，原因是系统超负荷工作，上下文切换、内存等等其它消耗导致系统性能下降。所谓吞吐量这里可以理解为每秒能处理请求的次数。   
        所以选择一个合理的 maxThreads值，其实并不是那么容易的事。因为过多的线程只会造成，更多的内存开销，更多的CPU开销，但是对提升QPS确毫无帮助；找到最佳线程数后通过简单的设置，可以让web系统更加稳定，得到最高，最稳定的QPS输出。

我们可以通过以下几种方式来获取 maxThreads的最佳值：   
        （1）通过线上系统不断使用和用户的不断增长来进行性能测试，观察QPS，响应时间，这种方式会在爆发式增长时系统崩溃，如双12等。   
        （2）根据公式计算，服务器端最佳线程数量=((线程等待时间+线程cpu时间)/线程cpu时间) \* cpu数量，这种方式有时会被误导，因为某些系统处理环节可能会耗时比较长，从而影响公式的结果。   
        （3）单、多用户压力测试，查看CPU的消耗，然后直接乘以百分比，再进行压测，一般这个值的附近应该就是最佳线程数量，这种方式理想场景比较适用，实际情况会比这个复杂的多。   
        （4）根据系统的自身情况调整，如硬件限制，系统限制，程序处理能力限制等。   
        （5）定期修改为不同的 maxThreads值，看服务器响应结果及用户反应。

 QPS和线程数的关系   
        （1）在最佳线程数量之前，QPS和线程是互相递增的关系，线程数量到了最佳线程之后，QPS持平，不在上升，甚至略有下降，同时相应时间持续上升。   
        （2）同一个系统而言，支持的线程数越多（最佳线程数越多而不是配置的线程数越多），QPS越高。

QPS和响应时间的关系   
        （1）对于一般的web系统，响应时间一般有CPU执行时间+IO等待时间组成。   
        （2）CPU的执行时间减少，对QPS有实质的提升，IO时间的减少，对QPS提升不明显。如果要想明显提升QPS，优化系统的时候要着重优化CPU消耗大户。   
        所以想要找出 maxThreads的最优值可并不容易，没有最好只有更好，更好的值只能通过时间来显现，如果你不想考虑那么多，一般情况下设置成1000即可。   
　　minSpareThreads Tomcat初始化时创建的 socket 线程数 。也是线程最小运行数目。这些线程始终保持运行。即使没有人使用，也开这么多空线程等待。  
　　maxSpareThreads Tomcat连接器的最大空闲 socket 线程数 （一旦创建的线程数超过这个值，tomcat就会关闭不再需要的socket线程）  
　　enableLookups 若设为true, 则支持域名解析，可把 ip 地址解析为主机名 设为false  
　　redirectPort 在需要基于安全通道的场合，把客户请求转发到基于SSL 的

redirectPort 端口   
　　acceptAccount 监听端口队列最大数，满了之后客户请求会被拒绝（当线程数达到maxThreads后，后续的请求会被放入到一个等待队列中，这个acceptCount是这个队列的大小，如果这个队列也满了，直接拒绝连接）

　　connectionTimeout 连接超时   
　　minProcessors 服务器创建时的最小处理线程数   
　　maxProcessors 服务器同时最大处理线程数   
　　URIEncoding URL统一编码

配置示例

<Connector port="9027"

　　protocol="HTTP/1.1"

　　maxHttpHeaderSize="8192"

　　maxThreads="1000"

　　minSpareThreads="100"

　　maxSpareThreads="1000"

　　minProcessors="100"

　　maxProcessors="1000"

　　enableLookups="false"

　　URIEncoding="utf-8"

　　acceptCount="1000"

　　redirectPort="8443"

　　disableUploadTimeout="true"/>

**3 tomcat缓存优化**

参数说明

compression 打开压缩功能   
　　compressionMinSize 启用压缩的输出内容大小，这里面默认为2KB   
　　compressableMimeType 压缩类型   
　　connectionTimeout 定义建立客户连接超时的时间. 如果为 -1, 表示不限制建立客户连接的时间

配置示例

<Connector port="9027"

　　protocol="HTTP/1.1"

　　maxHttpHeaderSize="8192"

　　maxThreads="1000"

　　minSpareThreads="100"

　　maxSpareThreads="1000"

　　minProcessors="100"

　　maxProcessors="1000"

　　enableLookups="false"

　　compression="on"

　　compressionMinSize="2048"

　　compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain"

　　connectionTimeout="20000"

　　URIEncoding="utf-8"

　　acceptCount="1000"

　　redirectPort="8443"

　　disableUploadTimeout="true"/>

HTTP 压缩可以大大提高浏览网站的速度，它的原理是，在客户端请求网页后，从服务器端将网页文件压缩，再下载到客户端，由客户端的浏览器负责解压缩并浏览。相对于普通的浏览过程HTML,CSS,Javascript , Text ，它可以节省40%左右的流量。更为重要的是，它可以对动态生成的，包括CGI、PHP , JSP , ASP , Servlet,SHTML等输出的网页也能进行压缩，压缩效率惊人。

### 线程池

Executor代表了一个线程池，可以在Tomcat组件之间共享。使用线程池的好处在于减少了创建销毁线程的相关消耗，而且可以提高线程的使用效率。   
        要想使用线程池，首先需要在 Service标签中配置 Executor，如下：

<Executor name="tomcatThreadPool"     
         namePrefix="catalina-exec-"     
         maxThreads="1000"     
         minSpareThreads="100"    
         maxIdleTime="60000"    
         maxQueueSize="Integer.MAX\_VALUE"    
         prestartminSpareThreads="false"    
         threadPriority="5"    
         className="org.apache.catalina.core.StandardThreadExecutor"/>    
  ....

name：线程池名称，用于 Connector中指定。   
        namePrefix：所创建的每个线程的名称前缀，一个单独的线程名称为 namePrefix+threadNumber。   
        maxThreads：池中最大线程数。   
        minSpareThreads：活跃线程数，也就是核心池线程数，这些线程不会被销毁，会一直存在。   
        maxIdleTime：线程空闲时间，超过该时间后，空闲线程会被销毁，默认值为6000（1分钟），单位毫秒。   
        maxQueueSize：在被执行前最大线程排队数目，默认为Int的最大值，也就是广义的无限。除非特殊情况，这个值不需要更改，否则会有请求不会被处理的情况发生。   
        prestartminSpareThreads：启动线程池时是否启动 minSpareThreads部分线程。默认值为false，即不启动。   
        threadPriority：线程池中线程优先级，默认值为5，值从1到10。   
        className：线程池实现类，未指定情况下，默认实现类为org.apache.catalina.core.StandardThreadExecutor。如果想使用自定义线程池首先需要实现 org.apache.catalina.Executor接口。   
        线程池配置完成后需要在 Connector中指定：

<Connector executor="tomcatThreadPool"    
...

### Listener

另一个影响Tomcat 性能的因素是内存泄露。Server标签中可以配置多个Listener，其中 JreMemoryLeakPreventionListener是用来预防JRE内存泄漏。此Listener只需在Server标签中配置即可，默认情况下无需配置，已经添加在 Server中。

<Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />

### 禁用DNS查询

当web应用程序向要记录客户端的信息时，它也会记录客户端的IP地址或者通过域名服务器查找机器名 转换为IP地址。

DNS查询需要占用网络，并且包括可能从很多很远的服务器或者不起作用的服务器上去获取对应的IP的过程，这样会消耗一定的时间。

修改server.xml文件中的Connector元素，修改属性enableLookups参数值: enableLookups="false"

如果为true，则可以通过调用request.getRemoteHost()进行DNS查询来得到远程客户端的实际主机名，若为false则不进行DNS查询，而是返回其ip地址

### 设置session过期时间

conf\web.xml中通过参数指定：

<session-config>

<session-timeout>180</session-timeout>

</session-config>

单位为分钟。

### Apr插件提高tomcat性能

Tomcat可以使用APR来提供超强的可伸缩性和性能，更好地集成本地服务器技术.

  APR(Apache Portable Runtime)是一个高可移植库，它是Apache HTTP Server 2.x的核心。APR有很多用途，包括访问高级IO功能(例如sendfile,epoll和OpenSSL)，OS级别功能(随机数生成，系统状态等 等)，本地进程管理(共享内存，NT管道和UNIX sockets)。这些功能可以使Tomcat作为一个通常的前台WEB服务器，能更好地和其它本地web技术集成，总体上让Java更有效率作为一个高 性能web服务器平台而不是简单作为后台容器。

  在产品环境中，特别是直接使用Tomcat做WEB服务器的时候，应该使用Tomcat Native来提高其性能

  要测APR给tomcat带来的好处最好的方法是在慢速网络上（模拟Internet），将Tomcat线程数开到300以上的水平，然后模拟一大堆并发请求。   
  如果不配APR，基本上300个线程狠快就会用满，以后的请求就只好等待。但是配上APR之后，并发的线程数量明显下降，从原来的300可能会马上下降到只有几十，新的请求会毫无阻塞的进来。   
  在 局域网环境测，就算是400个并发，也是一瞬间就处理/传输完毕，但是在真实的Internet环境下，页面处理时间只占0.1%都不到，绝大部分时间都 用来页面传输。如果不用APR，一个线程同一时间只能处理一个用户，势必会造成阻塞。所以生产环境下用apr是非常必要的。

## Tomcat与apache的连接方式

1、JK方式

这是最常见的方式。JK 是通过 AJP 协议与 Tomcat 服务器进行通讯的，Tomcat 默认的 AJP Connector 的端口是 8009。JK 本身提供了一个监控以及管理的页面 jkstatus，通过 jkstatus 可以监控 JK 目前的工作状态以及对到 tomcat 的连接进行设置

2、http\_proxy 方式

这是利用 Apache 自带的 mod\_proxy 模块使用代理技术来连接 Tomcat。在配置之前请确保是否使用的是 2.2.x 版本的 Apache 服务器。因为 2.2.x 版本对这个模块进行了重写，大大的增强了其功能和稳定性。

http\_proxy 模式是基于 HTTP 协议的代理，因此它要求 Tomcat 必须提供 HTTP 服务，也就是说必须启用 Tomcat 的 HTTP Connector。  
3、ajp\_proxy 方式

ajp\_proxy 连接方式其实跟 http\_proxy 方式一样，都是由 mod\_proxy 所提供的功能。配置也是一样，只需要把 http:// 换成 ajp:// ，同时连接的是 Tomcat 的 AJP Connector 所在的端口。

三者比较

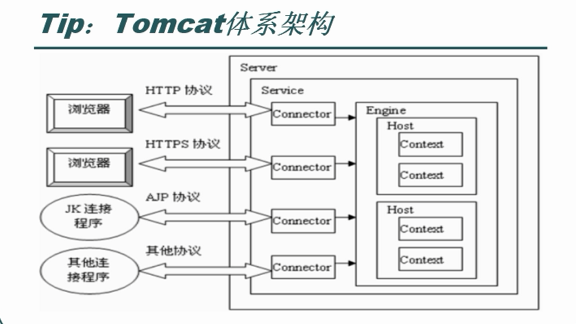
相对于 JK 的连接方式，后两种在配置上是比较简单的，灵活性方面也一点都不逊色。但就稳定性而言就不像 JK 这样久经考验，毕竟 Apache 2.2.3 推出的时间并不长，采用这种连接方式的网站还不多，因此，如果是应用于关键的互联网网站，还是建议采用 JK 的连接方式。  
配置比 JK 简单多了，而且它也可以通过一个页面来监控集群运行的状态，并做一些简单的维护设置。

**tomcat与apache为什么要整合起来，有什么意义？**

对于静态页面的数据，Apache的处理速度比Tomcat要快很多。  
如果客户端请求的是静态页面，则只需要Apache服务器响应请求，  
如果客户端请求动态页面，则是Tomcat服务器响应请求，  
因为jsp是服务器端解释代码的，这样整合就可以减少Tomcat的服务开销，提升对静态文件的处理性能，  
可以利用 Web 服务器来做负载均衡以及容错，无缝的升级应用程序。

## Tomcat完整的http请求

<http://www.cnblogs.com/hggen/p/6264475.html>



1) 请求被发送到本机端口8080，被在那里侦听的Coyote HTTP/1.1 Connector获得   
2) Connector把该请求交给它所在的Service的Engine来处理，并等待来自Engine的回应   
3) Engine获得请求localhost/yy/index.jsp，匹配它所拥有的所有虚拟主机Host   
4) Engine匹配到名为localhost的Host（即使匹配不到也把请求交给该Host处理，因为该Host被定义为该Engine的默认主机）   
5) localhost Host获得请求/yy/index.jsp，匹配它所拥有的所有Context   
6) Host匹配到路径为/yy的Context（如果匹配不到就把该请求交给路径名为”“的Context去处理）   
7) path=”/yy”的Context获得请求/index.jsp，在它的mapping table中寻找对应的servlet   
8) Context匹配到URL PATTERN为\*.jsp的servlet，对应于JspServlet类   
9) 构造HttpServletRequest对象和HttpServletResponse对象，作为参数调用JspServlet的doGet或doPost方法   
10)Context把执行完了之后的HttpServletResponse对象返回给Host   
11)Host把HttpServletResponse对象返回给Engine   
12)Engine把HttpServletResponse对象返回给Connector   
13)Connector把HttpServletResponse对象返回给客户browser

# Hibernate

## Hibernate和mybatis区别

## Hibernate中的缓存机制以及为什么使用

## Hibernate中的session和httpsession区别

## Sql注入问题

## 如果不使用参数绑定，怎么防止sql注入

## OpenSessionInView

Spring解决了Hibernate中的Session的关闭和开启问题。

Hibernate允许对关联对象 属性进行延迟加载，但是必须保证延迟加载的操作限于同一个Hibernate Session范围之内。如果Service层返回了一个启用延迟加载功能的领域对象给Web层，当Web层访问到那些需要延迟加载的数据时，由于加载领域对象的session已经关闭，这时会报异常LazyInitializationException。

Spring提供了OpenSessionInViewFilter过滤器，可以把一个Hibernate Session和一次完整的请求过程对应的线程绑定。也就是在web层开启了session，请求结束关闭session。这样，就可以允许在事务提交之后延迟加载所需要显示的对象。

OpenSessionInViewFilter 过滤器将 Hibernate Session 绑定到请求线程中，它将自动被 Spring 的事务管理器探测到。所以 OpenSessionInViewFilter 适用于 Service 层使用HibernateTransactionManager 或 JtaTransactionManager 进行事务管理的环境，也可以用于非事务只读的数据操作中。

<filter>  
        <filter-name>Spring OpenSessionInViewFilter</filter-name>  
        <filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-class>  
     <init-param>  
      
<!--   
指定org.springframework.orm.hibernate3.LocalSessionFactoryBean在spring配置文件中的名称,默认值为sessionFactory  
     如果LocalSessionFactoryBean在spring中的名称不是sessionFactory,该参数一定要指定,否则会出现找不到sessionFactory的例外  
-->  
     <param-name>sessionFactoryBean</param-name>  
   <param-value>sessionFactory</param-value>  
  </init-param>   
    </filter>  
    <filter-mapping>  
        <filter-name>Spring OpenSessionInViewFilter</filter-name>  
        <url-pattern>/\*</url-pattern>  
    </filter-mapping>

# Java web

## Memcache

**memcache**是一套**分布式的高速缓存系统**，由LiveJournal的Brad Fitzpatrick开发，但目前被许多网站使用以提升网站的访问速度，尤其对于一些大型的、需要频繁访问数据库的网站访问速度提升效果十分显著。这是一套开放源代码软件，以BSD license授权发布。

MemCache的工作流程如下：先检查客户端的请求数据是否在memcached中，如有，直接把请求数据返回，不再对数据库进行任何操作；如果请求的数据不在memcached中，就去查数据库，把从数据库中获取的数据返回给客户端，同时把数据缓存一份到memcached中（memcached客户端不负责，需要程序明确实现）；每次更新数据库的同时更新memcached中的数据，保证一致性；当分配给memcached内存空间用完之后，会使用**LRU（Least Recently Used，最近最少使用）策略加上到期失效策略，失效数据首先被替换，然后再替换掉最近未使用的数据。**

Memcache是一个**高性能的分布式的内存对象缓存系统**，通过**在内存里维护一个统一的巨大的hash表**，它能够用来存储各种格式的数据，包括图像、视频、文件以及数据库检索的结果等。简单的说就是**将数据调用到内存中，然后从内存中读取，从而大大提高读取速度。**

Memcache是danga的一个项目，最早是LiveJournal 服务的，最初为了加速 LiveJournal 访问速度而开发的，后来被很多大型的网站采用。

**Memcached是以守护程序(监听)方式运行于一个或多个服务器中，随时会接收客户端的连接和操作。**

在 Memcached中可以保存的item数据量是没有限制的，只要内存足够 。  
　　Memcached单进程在32位系统中**最大使用内存为2G**，若在64位系统则没有限制,这是由于**32位系统限制单进程最多可使用2G内存**,要使用更多内存，可以分多个端口开启多个Memcached进程 ,最大30天的数据过期时间,设置为永久的也会在这个时间过期，常量REALTIME\_MAXDELTA 60\*60\*24\*30控制 　　**最大键长为250字节，大于该长度无法存储，常量KEY\_MAX\_LENGTH 250控制**　　**单个item最大数据是1MB，超过1MB数据不予存储**，常量POWER\_BLOCK 1048576进行控制，它是默认的slab大小 　　最大同时连接数是200，通过 conn\_init()中的freetotal进行控制，最大软连接数是1024，通过settings.maxconns = 1024 进行控制 　　跟空间占用相关的参数：settings.factor=1.25, settings.chunk\_size=48, 影响slab的数据占用和步进方式

memcached是一种**无阻塞的socket通信方式服务**，基于libevent库，由于无阻塞通信，对内存读写速度非常之快。　　memcached分服务器端和客户端，可以配置多个服务器端和客户端，应用于分布式的服务非常广泛。　　memcached作为**小规模的数据分布式平台**是十分有效果的。

**memcached是键值一一对应，key默认最大不能超过128个字 节，value默认大小是1M**，也就是一个slabs，如果要存2M的值（连续的），不能用两个slabs，因为两个slabs不是连续的，无法在内存中 存储，故需要修改slabs的大小，多个key和value进行存储时，即使这个slabs没有利用完，那么也不会存放别的数据。

memcached已经可以支持C/C++、Perl、PHP、Python、Ruby、Java、C#、Postgres、Chicken Scheme、Lua、MySQL和Protocol等语言客户端。

memcache 可以应对任意多个连接，使用非阻塞的网络IO。由于它的工作机制是在内存中开辟一块空间，然后建立一个**HashTable**，Memcached自管理这些HashTable。

为什么会有Memcache和memcached两种名称？

其实Memcache是这个项目的名称，而memcached是它服务器端的主程序文件名，知道我的意思了吧~~~~。一个是项目名称，一个是主程序文件名，在网上看到了很多人不明白，于是混用了。