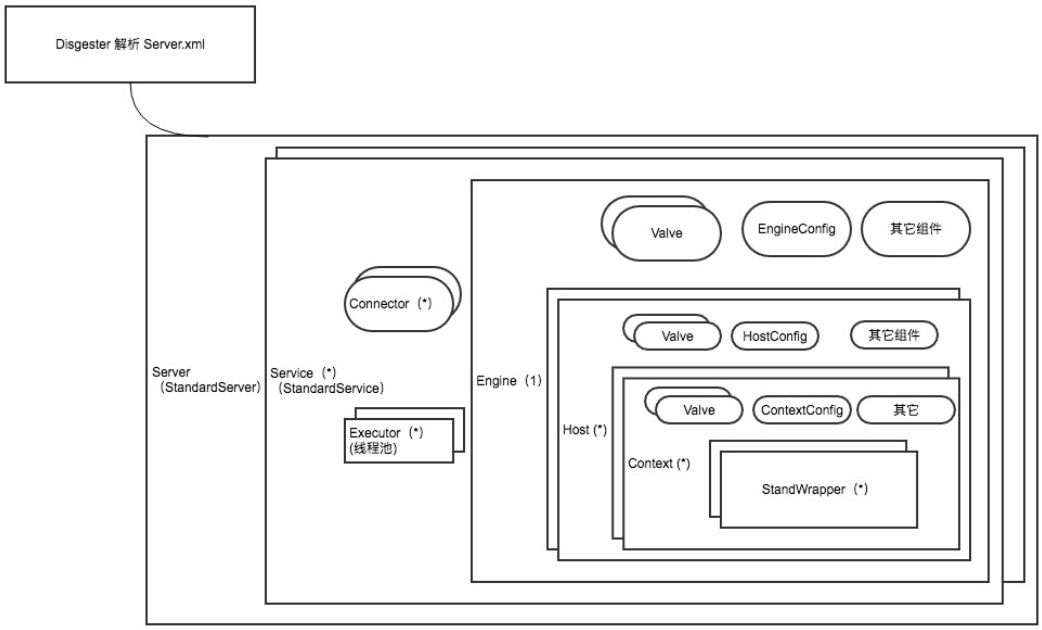
Degister 解析 Server.xml

Degister 通过流来读取 XML文件，识别出特定的XML节点后就会执行特定的动作；或者创建Java对象，或者执行对象的某个方法（常用的setter），核心是：匹配模式和匹配规则；

简单的讲，通过一系列的规则，Disgester 会为创建的对象之间构建父子层级关系；



p30

```
Catalina.load()
Digester digester = createStartDigester();
digester.addSetNext("Server",...
```

Catalina 的Context 配置来源有多处：

Server.xml 中可以有 <Context 的配置，不过大多数情况下 我们不在这里配置；

由 HostConfig 自动扫描部署目录，以Context.xml 文件为基础进行解析；

Catalina 在创建 Context实例的同时，添加了生命周期监听器 ContextConfig，用

于详细配置 ContextConfig，如解析 web.xml 等；此外还会为 之添加一下资源监听，如 WatchedResource 为Context 添加监视资源，当这些资源发生变更时，web应用将会被重新加载，默认为 WEB-INF/web.xml；

同HostConfig一样，ContextConfig也是在Digester解析server.xml的时候添加到StandardContext上的监听器，ContextConfig主要是处理web应用的配置文件;

Servlet 容器：部署web应用 和 将请求映射到具体的Servlet 进行处理；

Web应用加载：

主要由StandardHost、HostConfig、StandardContext、ContextConfig、StandardWrapper 完成；

一、两种加载入口
StandardHost 加载 web应用（StandardContext）的入口有两个：
1、Catalina构造 Server 实例的时候，如果Host 元素存在 Context 子元素（server.xml中），那么Context 元素将作为 Host 容器的子容器添加到 Host 实例中，并在 Host start 过程中 触发 其启动（start）；
<Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true">
    <Context docBase="myApp" path="/myApp" reloadable="true">
</Host>
appBase为web应用部署的基础目录，所有需要部署的web应用均需要复制到此目录下，默认为 $CATALINA_BASE/webapps；
docBase 为web应用根目录的文件路径；
path 为web应用的根请求地址；
如： http://127.0.0.1:8080/myApp 为跟请求地址；

此方式，解析 server.xml 时候一并完成Context 的创建；但是默认情况下，server.xml中并未包含Context 相关配置；

2、由HostConfig自动扫描部署目录，创建Context 实例启动。这是大多数web应用的加载方式；

二、StandardHost的启动加载：

1、为Host 添加 ErrorReportValve；

ErrorReportValve 在服务器处理异常时输出错误页面，如果没有在 web.xml中添加错误处理页面，Tomcat 返回的异常栈页面便是由 ErrorReportValve 生成的；

```
<error-page>
  <error-code>404</error-code>
  <location>/common/error.htm</location>
</error-page>
```

2、触发 父类 ContainerBase 的 startInternal() 方法启动虚拟主机；

2.1、启动 集群组件 Cluster、安全组件Realm等；

2.2、启动子节点（即通过 server.xml 中<Context> 元素创建的StandardContext 实例）；

2.3、启动Host的 pipeline 组件；

2.4、设置Host 状态为 STARTING，触发 START_EVENT 事件，HostConfig 监听改事件，扫描web部署目录，对于部署描述文件、war包、目录 会自动创建 StandardContext 实例，添加到 Host 作为子容器并启动；

2.5、启动 Host 的后台任务线程；

三、HostConfig 部署

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

appBase为web应用部署的基础目录，所有需要部署的web应用均需要复制到此目录下，默认为 $CATALINA_BASE/webapps；

Tomcat 通过 HostConfig 完成该目录下的 web应用的部署；

1、START_EVENT事件处理：

Host 的 start 时候触发，只有当 Host 的 deployOnStartup 属性为 true 时(默认为true)，服务器才会在启动过程部署web应用；

事件处理包括3部分：Context描述文件部署，web目录部署，WAR包部署；

HostConfig.deployApps()：

1.1、Context描述文件部署

通过 独立的 Context 描述文件来配置并启动 web应用，配置方式 同server.xml 中的<Context>；
配置文件的 存储路径： Host 的 xmlBase 属性指定；未指定，默认为 $CATALIAN_BASE/conf/<Engine名称>/<Host名称>，对于Tomcat 的默认Host，描述文件路径为： $CATALIAN_BASE/conf/Catalina/localhost；

如： 创建一个 helloWorld.xml：
<Context docBase="test/helloWorld" path="/helloWorld"
reloadable="false">
    <WatchedResource>WEB-INF/web.xml</WatchedResource>
</Context>
并将 目录名为 helloWorld 的web应用 复制到 test 目录下，Tomcat 启动的时候就会自动部署改 web应用，根请求地址为： http://127.0.0.1:8080/helloWorld ；

1️⃣ 扫描 $CATALIAN_BASE/conf/<Engine名称>/<Host名称> 目录，对每个配置文件 ，由线程池 完成解析部署；（**host.getStartStopExecutor** 事件处理线程池）
2️⃣ 使用Degister 解析配置文件，创建Context实例；更新 Context 实例的名称等；
3️⃣ 为Context 添加ContextConfig 生命周期监听器；
4️⃣ Host.addChild(Context); 会触发 Context 的 start，没有 init 的时候先 init，再start；
5️⃣ 将Context 的描述文件、web应用目录以及 web.xml 等添加到 守护资源，文件发生变更的时候（使用资源文件的上次修改时间判断），重新部署或者加载web应用；

1.2、web目录部署
最常见的方式； 将web应用所有资源文件 、jar包、描述文件（WEB-INF/web.xml） 的目录复制到 Host 指定的 appBase目录下即可；

1、对于 Host的 appBase 目录下（默认为 $CATALIAN_BASE/webapps）下所有符合条件的目录，由线程池完成部署； 每个目录如下操作：
1.1、如果 Host 的deployXML 为true（默认，即通过 Context描述符文件部署），并且存在 **META-INF/context.xml** 文件，则使用 Digester 解析

context.xml 文件创建 Context 对象；

其他情况下，根据 **Host 的 contextClass 属性指定的类型创建 Context 对象；如不指定，默认为 StandardContext；**

1.2、为Context 添加ContextConfig 生命周期监听器；

... 后续添加子容器，以及守护资源 等同；

1.3、WAR包部署

War包部署 和 目录部署基本类似，war包是压缩文件，增加了压缩文件的处理；

1、扫描 Host的 appBase 目录下 的所有符合条件的 war 包，线程池完成部署；

1.1如果 Host 的deployXML 为true 且 war包同名目录（去除扩展名）、war包压缩文件下 存在 **META-INF/context.xml** 文件，则使用 Digester 解析 context.xml 文件创建 Context 对象；

其他情况下，根据 Host 的 contextClass 属性指定的类型创建 Context 对象；如不指定，默认为 StandardContext；

1.2、为Context 添加ContextConfig 生命周期监听器；

... 后续添加子容器，以及守护资源 等同；

ContainerBase.backgroundProcess() : 定时扫描web应用的变更，并进行 重新加载；默认由 Engine 维护后台任务处理线程；（Host的 autoDeploy=true）

重新加载：对同一个 Context对象 的重启；

重新部署：重新创建一个 Context对象；

Catalina 同时守护 两类资源： redeployResources 和 reloadResources ，来区别重新加载还是 部署应用；

如Context 的描述文件变更，则需要重新部署； web.xml 文件变更，只需要重新加载Context即可；

StandardContext

FilterMap 存储 filter-mapping配置；

初始化阶段，Context属性配置：

1、server.xml 中有<Context，解析该文件，更新 Context 实例属性；

2、conf/context.xml（Catalina 容器级默认配置），解析该文件，更新当前Context 实例属性；
3、Host 部署web应用的时候， 如果存在 $CATALIAN_BASE/conf/<Engine名称>/<Host名称>/context.xml.default 文件（Host即默认配置），解析该文件，更新当前Context属性；
4、HostConfig扫描部署web应用的时候， 如果 context 的 configFile属性（META-INF/context.xml） 不为空，解析文件，更新当前Context属性；
属性覆盖优先级： 4 > 3 > 2 > 1;

启动过程：
创建 web应用类加载器（WebappLoader），其启动时创建WebappClassLoader；而且，提供了 backgroundProcess 用于Context后台处理：当检测到 web应用的 类文件、jar包发生变更（文件的修改时间），重新加载Context；
启动 安全组件等；
启动子容器之前，发布CONFIGURE_START_EVENT 事件，其生命周期监听对象**ContextConfig** 监听该事件已完成 Servlet 的创建；
启动子容器；

ContextConfig

初始化事件处理：
将Context 的web资源集合 添加到 ServletContext 属性 resources；
创建实例管理器 InstanceManager，用于创建 对象实例，如 Servlet、Filter等；
实例化 应用监听器，如 ServletContextListener 等生命周期监听器，来源于Context 部署描述文件、web.xml 等；
实例化 FilterConfig（ApplicationFilterConfig）、Filter，并调用 **Filter.init 初始化**；

CONFIGURE_START_EVENT 事件处理：
主要**解析 web.xml**，创建wrapper（servlet）、Filter相关配置、

ServletContextListener等一系列 web容器相关对象，以及请求映射，完成web容器的初始化；

1、解析 web.xml，还有 web-fragment.xml等，并且完成相关 xml 文件的排序和合并；

2、如果 **ignoreAnnotations = false（默认）**，则解析应用程序的 注解配置；

Tomcat 的默认web配置：

Catalina容器级别： conf/web.xml，Host级别：（conf/<Engine名称>/<Host名称>/web.xml.default），web应用级 WEB-INF/web.xml；

优先级： 应用级 > Host > Catalina容器级；

1、解析默认配置，生成 WebXml对象（default）；先解析容器级，再解析Host级配置；对于同名配置，Host级覆盖 容器级；

2、解析 web应用的 WEB-INF/web.xml（默认），析解析结果到 contextWebXml，其他结果均需要合并到此；

3、扫描web应用的 所有 jar包，如果包含 META-INF/web-fragment.xml，则解析文件 创建 WebXml 对象（片段）。

4、处理 WEB-INF/class 下的注解，扫描 javax.servlet.annotation.WebServlet、WebFilter，WebListener 等Servlet 规范注解配置，将结果合并到 第2步的 主 WebXml 中；

5、处理 jar 包内的注解，同样 扫描 servlet 规范注解，结果合并打 第3步的 WebXml 中；

6、配置合并： 片段WebXml 、默认 WebXml 合并到 最终的 主 WebXml（contextWebXml） 中；

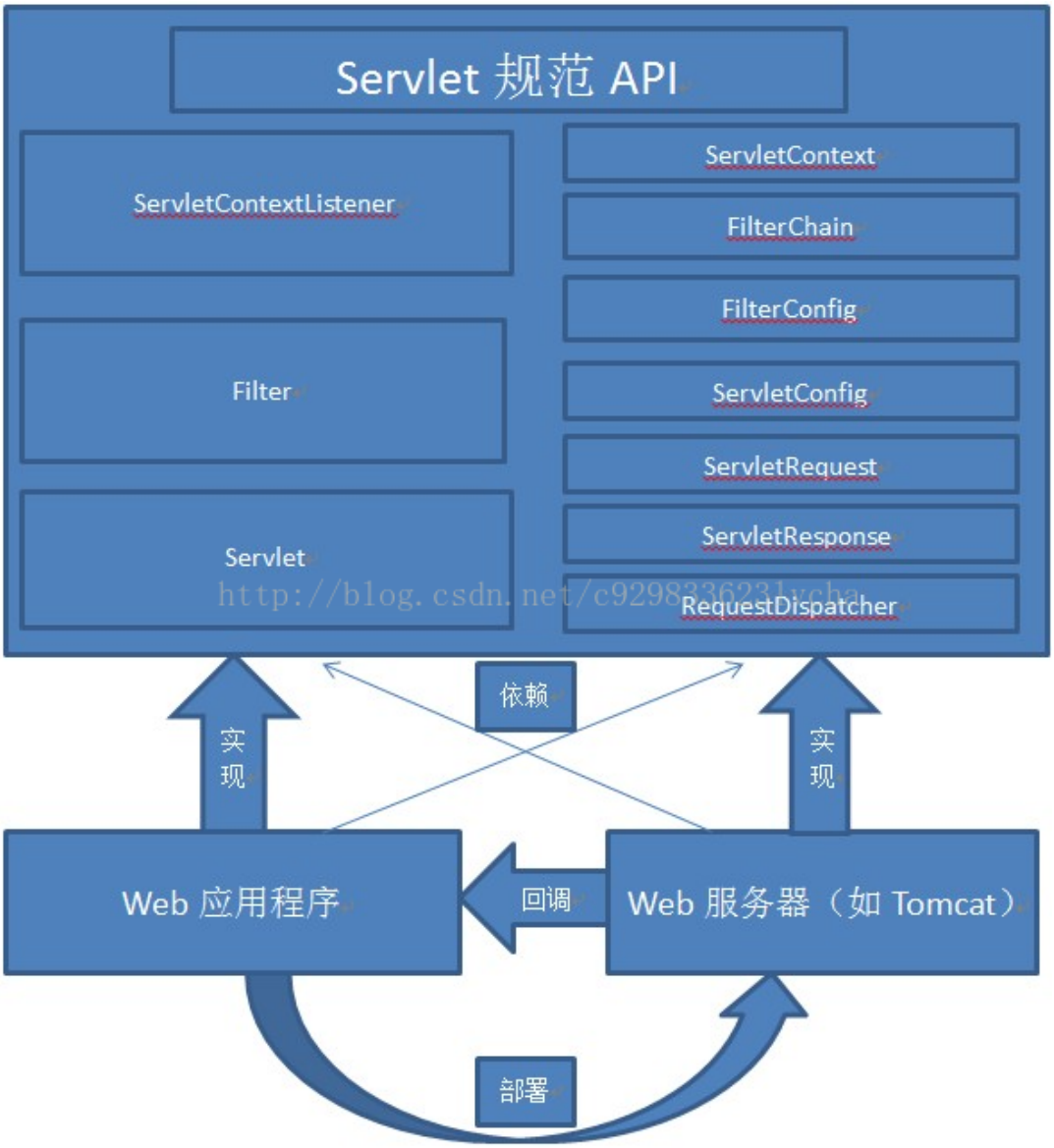
7、配置 jspServlet等；

8、使用 主WebXml 配置当前的 StandardContext，包括 Servlet、Filter、Listener等 Servlet规范组件；

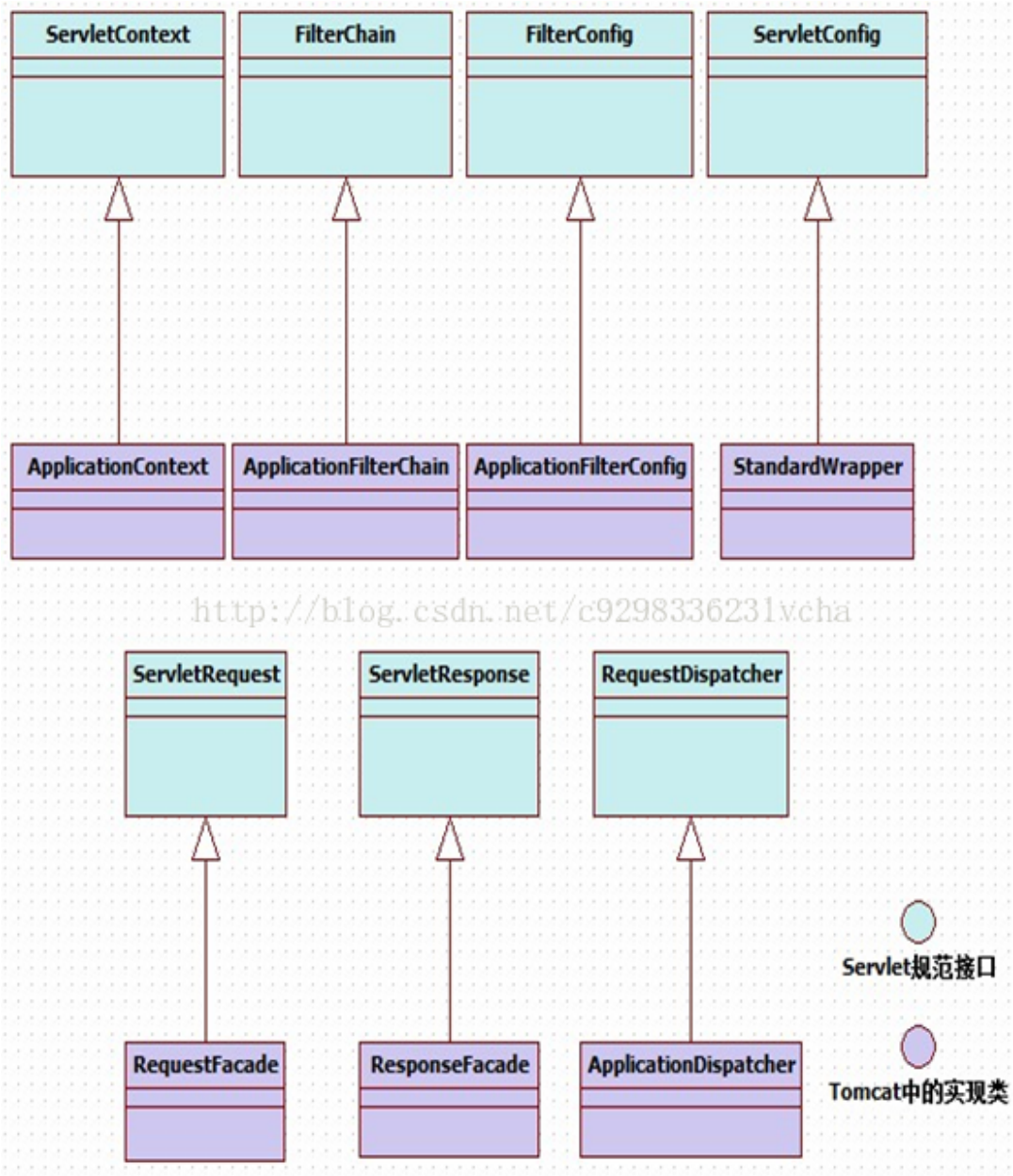
对于 ServletContext 层次的对象，直接由 ServletContext 维护；对于 Servlet，创建 StandardWrapper 对象，并添加到 StandardContext中；

9、查找 jar包中的META-INF/resources/ 下的静态资源，添加到 ServletContext；

完成，在正式启动 StandardWrapper 之前， 完成了 web应用容器的初始化；


Servlet 规范 API
ServletContextListener
Filter
Servlet
ServletContext
FilterChain
FilterConfig
ServletConfig
ServletRequest
ServletResponse
RequestDispatcher
http://blog.csdn.net/c929833623lycha
依赖
实现
实现
Web 应用程序
回调
Web 服务器（如 Tomcat）
部署

p33



p32

StandardWrapper

1、完成了StandardContext 的初始化，触发 StandardWrapper的 start；

2、对于 启动时加载的 Servlet（load-on-startup >= 0），调用 StandardWrapper 的load()，完成 Servlet的加载；

（1）创建 Servlet 实例；

（2）调用 javax.servlet.Servlet.init() 方法进行Servlet 初始化；

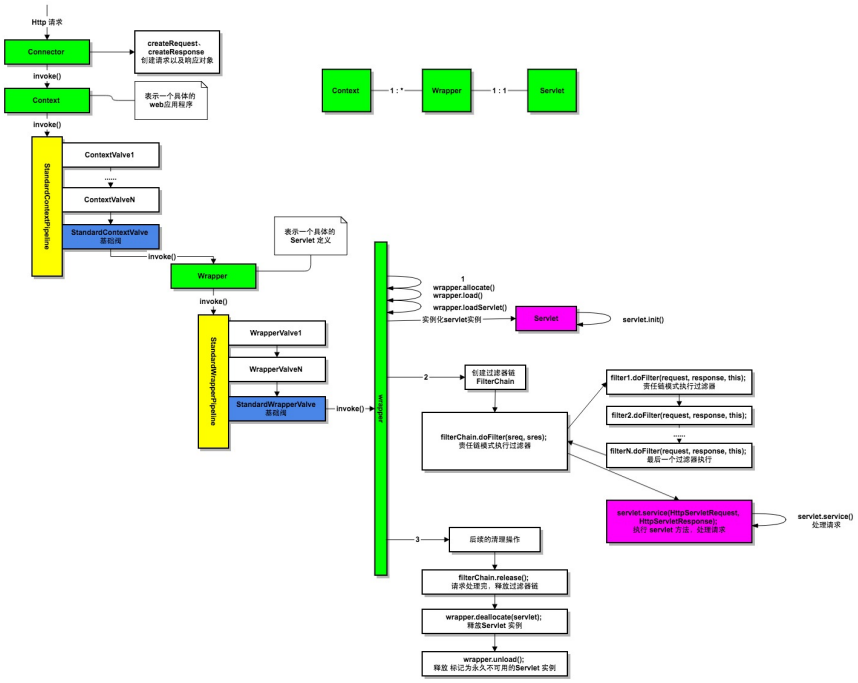
至此，整个 web 应用的加载过程 完成；

http://tomcat.apache.org/tomcat-7.0-doc/config/http.html#Standard_Implementation

---

生命周期组件：

LifeCycle
init();
    initiTERNAL()
start();
stop()
destroy()
addLifecycleListener(LifecycleListener listener);
LifecycleBase



p31

Tomcat的Connector组件

前面也已经经介绍过，Connector组件是Service容器中的一部分。它主要是接收，解析HTTP请求，然后调用本Service下的相关Servlet。由于Tomcat从架构上采用的是一个分层结构，因此根据解析过的HTTP请求，定位到相应Servlet也是一个相对比较复杂的过程，整个Connector实现了从接收Socket到调用Servlet的全过程。先来看一下Connector的功能逻辑：

· **接收Socket**

· **从Socket获取数据包，并解析成HttpServletRequest对象**

· **从Engine容器开始走调用流程，经过各层Valve，最后调用Servlet完成业务逻辑**

· **返回Response，关闭Socket**

可以看出，整个Connector组件是Tomcat运行主干，目前Connector支持的协议是HTTP和AJP，本文主要是针对HTTP协议的Connector进行阐述。先来看一下Connector的配置，在server.xml里；

Mapper

类完全路径org.apache.tomcat.util.http.mapper.Mapper，此对象维护了一个从Host到Wrapper的各级容器的快照。它主要是为了，**当HTTP Request被解析后，能够将HTTP Request绑定到相应的Host,Context,Wrapper(Servlet)，进行业务处理**；

Tomcat运行过程中的线程概况及线程模型

Tomcat在运行过程中会涉及到很多线程，主要的线程有Tomcat启动时的main主线程（Java进程启动时的那个线程）， 子容器启动与关系线程池（用来启动关系子容器），Tomcat后台线程（Tomcat内置的后台线程，比如用来热部署Web应用）， 请求接收线程(用来接收请求的线程)， 请求处理线程池（用来处理请求的线程）。理解了Tomcat运行过程中的主要线程，有助于我们理解整个系统的线程模型。下面是每个线程的源码及功能的详细分析：

(1) **main主线程：**即从main开始执行的线程，在启动Catalina之后就一直在Server的await方法中等待关闭命令，如果未接收到关闭命令就一直等待下去。main线程会一直阻塞着等待关机命令。

启动过程：

Bootstrap.main>>

Catalina.start >>

Catalina.await() >>

StandardServer.await()

(2) **子容器启动与关闭线程：**

ContainerBase的**protected** ThreadPoolExecutor startStopExecutor容器线程池执行器[处理子容器的启动和停止]在ContainerBase.initInternal()方法中进行初始化

作用：在ContainerBase.startInternal()方法中启动子容器，在ContainerBase.stopInternal()方法中停止子容器，即处理子容器的启动和停止事

件。

(3) **容器的后台处理线程：**ContainerBackgroundProcessor是ContainerBase的一个内部类

(4) **请求接收线程：**即运行Acceptor的线程，启动Connector的时候产生
// 默认只有一个接收线程Acceptor

(5) **请求处理线程：**即运行SocketProcessor的线程，从请求处理线程池中产生，org.apache.tomcat.util.net. AbstractEndpoint的成员变量**private** Executor executor用来处理请求，org.apache.tomcat.util.net.JioEndpoint(继承了AbstractEndpoint)在方法startInternal()中调用createExecutor()来创建executor为线程池对象ThreadPoolExecutor

Tomcat所涉及的设计模式

Tomcat虽然代码比较庞大，但是整体还是设计的比较优雅，特别是很多组件化的设计思路，其中涉及到的一些常用的设计模式值得我们学习及借鉴：

责任链模式

Tomcat中有两个地方比较明显的使用了责任链模式，一、Tomcat中的ApplicationFilterChain实现了Filter拦截和实际Servlet的请求，是典型的责任链模式。其他开源框架中类似的设计还有Struts2中的DefaultActionInvocation实现Interceptor拦截和Action的调用。Spring AOP中ReflectiveMethodInvocation实现MethodInceptor方法拦截和target的调用。二、Tomcat中的Pipeline-Valve模式也是责任链模式的一种变种，从Engine到Host再到Context一直到Wrapper都是通过一个链来传递请求。

观察者模式

Tomcat通过LifecycleListener对组件生命周期组件Lifecycle进行监听就是典型的观察者模式，各个组件在其生命期中会有各种各样行为，而这些行为都会触发相应的事件，Tomcat就是通过侦听这些事件达到对这些行为进行扩展的目的。在看组件的init和start过程中会看到大量如：
lifecycle.fireLifecycleEvent(AFTER_START_EVENT,null);这样的代码，这就是对某一类型事件的触发，如果你想在其中加入自己的行为，就只用注册相应类型的

事件即可。

门面模式

门面设计模式在 Tomcat 中有多处使用，在 Request 和 Response 对象封装中(RequestFacade,ResponseFacade)、ApplicationContext 到 ApplicationContextFacade等都用到了这种设计模式。这种设计模式主要用在一个大的系统中有多个子系统组成时，这多个子系统肯定要涉及到相互通信，但是每个子系统又不能将自己的内部数据过多的暴露给其它系统，不然就没有必要划分子系统了。每个子系统都会设计一个门面，把别的系统感兴趣的数据封装起来，通过这个门面来进行访问。

模板方法模式

模板方法模式是我们平时开发当中常用的一种模式，把通用的骨架抽象到父类中，子类去实现特地的某些步骤。Tomcat及Servlet规范API中也大量的使用了这种模式，比如Tomcat中的ContainerBase中对于生命周期的一些方法 init,start,stop和Servlet规范API中的GenericServlet中的service抽象骨架模板方法均使用了模板方法模式。