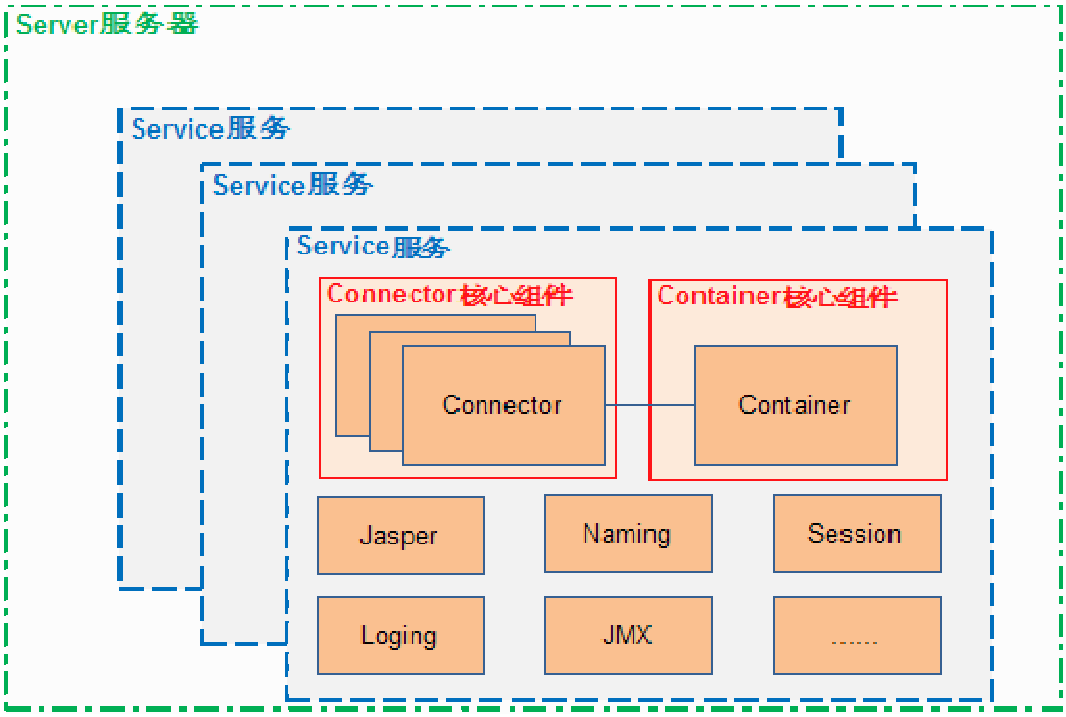
Tomcat 系统架构与设计模式 工作原理

本文以 Tomcat 5 为基础，也兼顾最新的 Tomcat 6 和 Tomcat 4。Tomcat 的基本设计思路和架构是具有一定连续性的。

Tomcat 总体结构

Tomcat 的结构很复杂，但是 Tomcat 也非常的模块化，找到了 Tomcat 最核心的模块，您就抓住了 Tomcat 的“七寸”。下面是 Tomcat 的总体结构图：

图 1.Tomcat 的总体结构

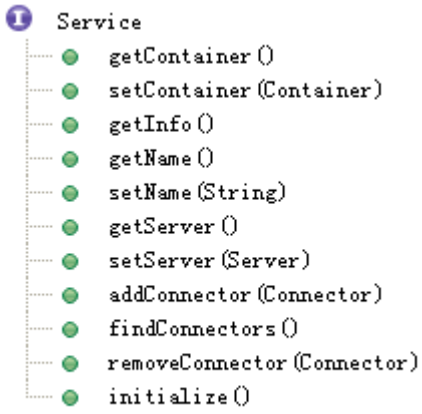


从上图中可以看出 Tomcat 的心脏是两个组件：Connector 和 Container，关于这两个组件将在后面详细介绍。Connector 组件是可以被替换，这样可以提供给服务器设计者更多的选择，因为这个组件是如此重要，不仅跟服务器的设计的本身，而且和不同的应用场景也十分相关，所以一个 Container 可以选择对应多个 Connector。多个 Connector 和一个 Container 就形成了一个 Service，Service 的概念大家都很熟悉了，有了 Service 就可以对外提供服务了，但是 Service 还要一个生存的环境，必须要有人能够给她生命、掌握其生死大权，那就非 Server 莫属了。所以整个 Tomcat 的生命周期由 Server 控制。

以 Service 作为“婚姻”

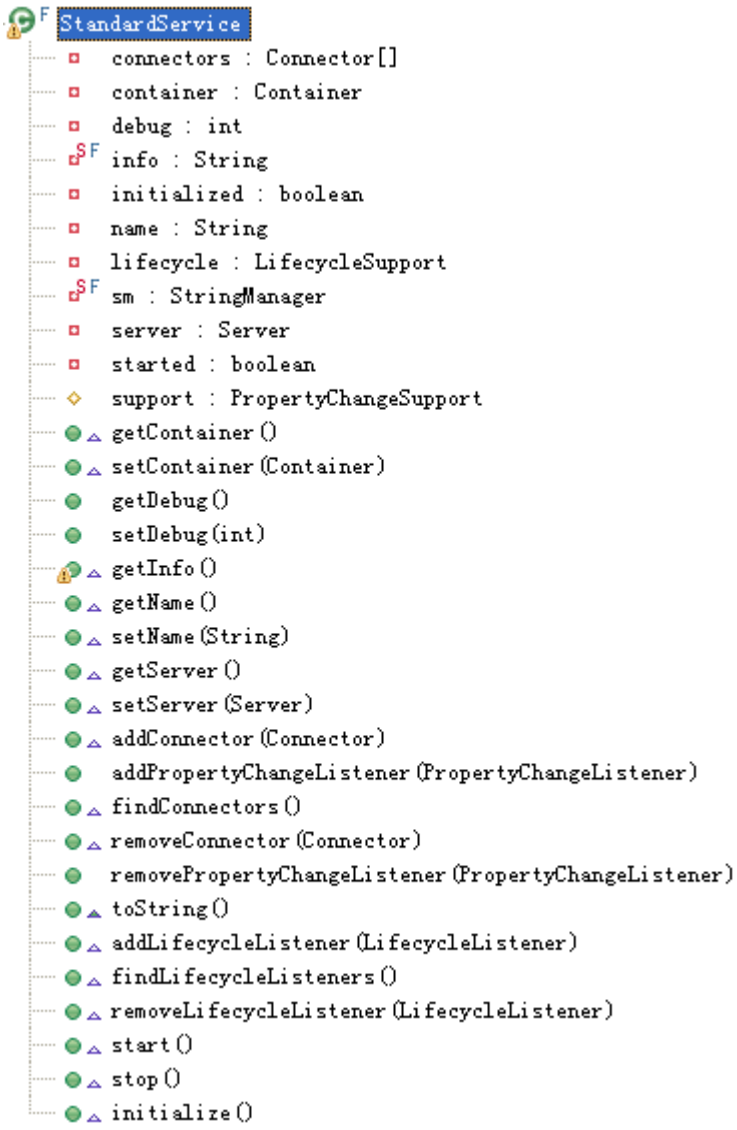
我们将 Tomcat 中 Connector、Container 作为一个整体比作一对情侣的话，Connector 主要负责对外交流，可以比作为 Boy，Container 主要处理 Connector 接受的请求，主要是处理内部事务，可以比作为 Girl。那么这个 Service 就是连接这对男女的结婚证了。是 Service 将它们连接在一起，共同组成一个家庭。当然要组成一个家庭还要很多其它的元素。
说白了，Service 只是在 Connector 和 Container 外面多包一层，把它们组装在一起，向外面提供服务，一个 Service 可以设置多个 Connector，但是只能有一个 Container 容器。这个 Service 接口的方法列表如下：

图 2. Service 接口



从 Service 接口中定义的方法中可以看出，它主要是为了关联 Connector 和 Container，同时会初始化它下面的其它组件，注意接口中它并没有规定一定要控制它下面的组件的生命周期。所有组件的生命周期在一个 Lifecycle 的接口中控制，这里用到了一个重要的设计模式，关于这个接口将在后面介绍。
Tomcat 中 Service 接口的标准实现类是 StandardService 它不仅实现了 Service 借口同时还实现了 Lifecycle 接口，这样它就可以控制它下面的组件的生命周期了。StandardService 类结构图如下：

图 3. StandardService 的类结构图



从上图中可以看出除了 Service 接口的方法的实现以及控制组件生命周期的 Lifecycle 接口的实现，还有几个方法是用于在事件监听的方法的实现，不仅是这个 Service 组件，Tomcat 中其它组件也同样有这几个方法，这也是一个典型的设计模式，将在后面介绍。

下面看一下 StandardService 中主要的几个方法实现的代码，下面是 setContainer 和 addConnector 方法的源码：

清单 1. StandardService. SetContainer

```
public void setContainer(Container container) {
    Container oldContainer = this.container;
    if ((oldContainer != null) && (oldContainer instanceof Engine))
        ((Engine) oldContainer).setService(null);
    this.container = container;
```

```
    if ((this.container != null) && (this.container instanceof Engine))
        ((Engine) this.container).setService(this);
    if (started && (this.container != null) && (this.container instanceof Lifecycle))
{
        try {
            ((Lifecycle) this.container).start();
        } catch (LifecycleException e) {
            ;
        }
    }
    synchronized (connectors) {
        for (int i = 0; i < connectors.length; i++)
            connectors[i].setContainer(this.container);
    }
    if (started && (oldContainer != null) && (oldContainer instanceof Lifecycle)) {
        try {
            ((Lifecycle) oldContainer).stop();
        } catch (LifecycleException e) {
            ;
        }
    }
    support.firePropertyChange("container", oldContainer, this.container);
}
```

这段代码很简单，其实就是先判断当前的这个 Service 有没有已经关联了 Container，如果已经关联了，那么去掉这个关联关系——oldContainer.setService(null)。如果这个 oldContainer 已经被启动了，结束它的生命周期。然后再替换新的关联、再初始化并开始这个新的 Container 的生命周期。最后将这个过程通知感兴趣的事件监听程序。这里值得注意的地方就是，修改 Container 时要将新的 Container 关联到每个 Connector，还好 Container 和 Connector 没有双向关联，不然这个关联关系将会很难维护。

清单 2. StandardService. addConnector

```
public void addConnector(Connector connector) {
    synchronized (connectors) {
        connector.setContainer(this.container);
        connector.setService(this);
        Connector results[] = new Connector[connectors.length + 1];
        System.arraycopy(connectors, 0, results, 0, connectors.length);
        results[connectors.length] = connector;
        connectors = results;
        if (initialized) {
            try {
                connector.initialize();
```

```
            } catch (LifecycleException e) {
                e.printStackTrace(System.err);
            }
        }
        if (started && (connector instanceof Lifecycle)) {
            try {
                ((Lifecycle) connector).start();
            } catch (LifecycleException e) {
                ;
            }
        }
        support.firePropertyChange("connector", null, connector);
    }
}
```

上面是 addConnector 方法，这个方法也很简单，首先是设置关联关系，然后是初始化工作，开始新的生命周期。这里值得一提的是，注意 Connector 用的是数组而不是 List 集合，这个从性能角度考虑可以理解，有趣的是这里用了数组但是并没有向我们平常那样，一开始就分配一个固定大小的数组，它这里的实现机制是：重新创建一个当前大小的数组对象，然后将原来的数组对象 copy 到新的数组中，这种方式实现了类似的动态数组的功能，这种实现方式，值得我们以后拿来借鉴。

最新的 Tomcat6 中 StandardService 也基本没有变化，但是从 Tomcat5 开始 Service、Server 和容器类都继承了 MBeanRegistration 接口，Mbeans 的管理更加合理。

以 Server 为“居”

前面说一对情侣因为 Service 而成为一对夫妻，有了能够组成一个家庭的基本条件，但是它们还要有个实体的家，这是它们在社会上生存之本，有了家它们就可以安心的为人民服务了，一起为社会创造财富。

Server 要完成的任务很简单，就是要能够提供一个接口让其它程序能够访问到这个 Service 集合、同时要维护它所包含的所有 Service 的生命周期，包括如何初始化、如何结束服务、如何找到别人要访问的 Service。还有其它的一些次要的任务，如您住在这个地方要向当地政府去登记啊、可能还有要配合当地公安机关日常的安全检查什么的。

Server 的类结构图如下：

图 4. Server 的类结构图

它的标准实现类 StandardServer 实现了上面这些方法，同时也实现了 Lifecycle、MbeanRegistration 两个接口的所有方法，下面主要看一下 StandardServer 重要的一个方法 addService 的实现：

清单 3. StandardServer.addService

```
public void addService(Service service) {
   service.setServer(this);
   synchronized (services) {
      Service results[] = new Service[services.length + 1];
      System.arraycopy(services, 0, results, 0, services.length);
      results[services.length] = service;
      services = results;
      if (initialized) {
         try {
            service.initialize();
         } catch (LifecycleException e) {
            e.printStackTrace(System.err);
         }
      }
      if (started && (service instanceof Lifecycle)) {
         try {
            ((Lifecycle) service).start();
         } catch (LifecycleException e) {
            ;
         }
      }
      support.firePropertyChange("service", null, service);
   }
```

```
}
```

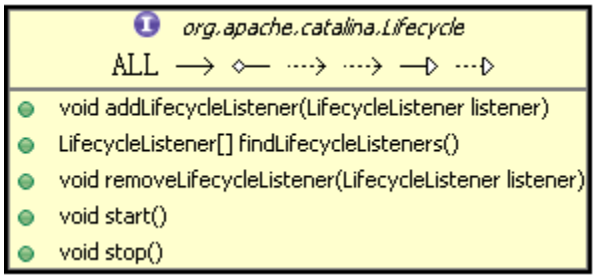
从上面第一句就知道了 Service 和 Server 是相互关联的，Server 也是和 Service 管理 Connector 一样管理它，也是将 Service 放在一个数组中，后面部分的代码也是管理这个新加进来的 Service 的生命周期。Tomcat6 中也是没有什么变化的。

组件的生命线“Lifecycle”

前面一直在说 Service 和 Server 管理它下面组件的生命周期，那它们是如何管理的呢？

Tomcat 中组件的生命周期是通过 Lifecycle 接口来控制的，组件只要继承这个接口并实现其中的方法就可以统一被拥有它的组件控制了，这样一层一层的直到一个最高级的组件就可以控制 Tomcat 中所有组件的生命周期，这个最高的组件就是 Server，而控制 Server 的是 Startup，也就是您启动和关闭 Tomcat。下面是 Lifecycle 接口的类结构图：

图 5. Lifecycle 类结构图



除了控制生命周期的 Start 和 Stop 方法外还有一个监听机制，在生命周期开始和结束的时候做一些额外的操作。这个机制在其它的框架中也被使用，如在 Spring 中。关于这个设计模式会在后面介绍。

Lifecycle 接口的方法的实现都在其它组件中，就像前面中说的，组件的生命周期由包含它的父组件控制，所以它的 Start 方法自然就是调用它下面的组件的 Start 方法，Stop 方法也是一样。如在 Server 中 Start 方法就会调用 Service 组件的 Start 方法，Server 的 Start 方法代码如下：

清单 4. StandardServer.Start

```
public void start() throws LifecycleException {
   if (started) {
      log.debug(sm.getString("standardServer.start.started"));
      return;
   }
   lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
   lifecycle.fireLifecycleEvent(START_EVENT, null);
   started = true;
```

```
    synchronized (services) {
        for (int i = 0; i < services.length; i++) {
            if (services[i] instanceof Lifecycle)
                ((Lifecycle) services[i]).start();
        }
    }
    lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}
```

监听的代码会包围 Service 组件的启动过程，就是简单的循环启动所有 Service 组件的 Start 方法，但是所有 Service 必须要实现 Lifecycle 接口，这样做会更加灵活。

Server 的 Stop 方法代码如下：

清单 5. StandardServer.Stop

```
public void stop() throws LifecycleException {
    if (!started)
        return;
    lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
    lifecycle.fireLifecycleEvent(STOP_EVENT, null);
    started = false;
    for (int i = 0; i < services.length; i++) {
        if (services[i] instanceof Lifecycle)
            ((Lifecycle) services[i]).stop();
    }
    lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}
```

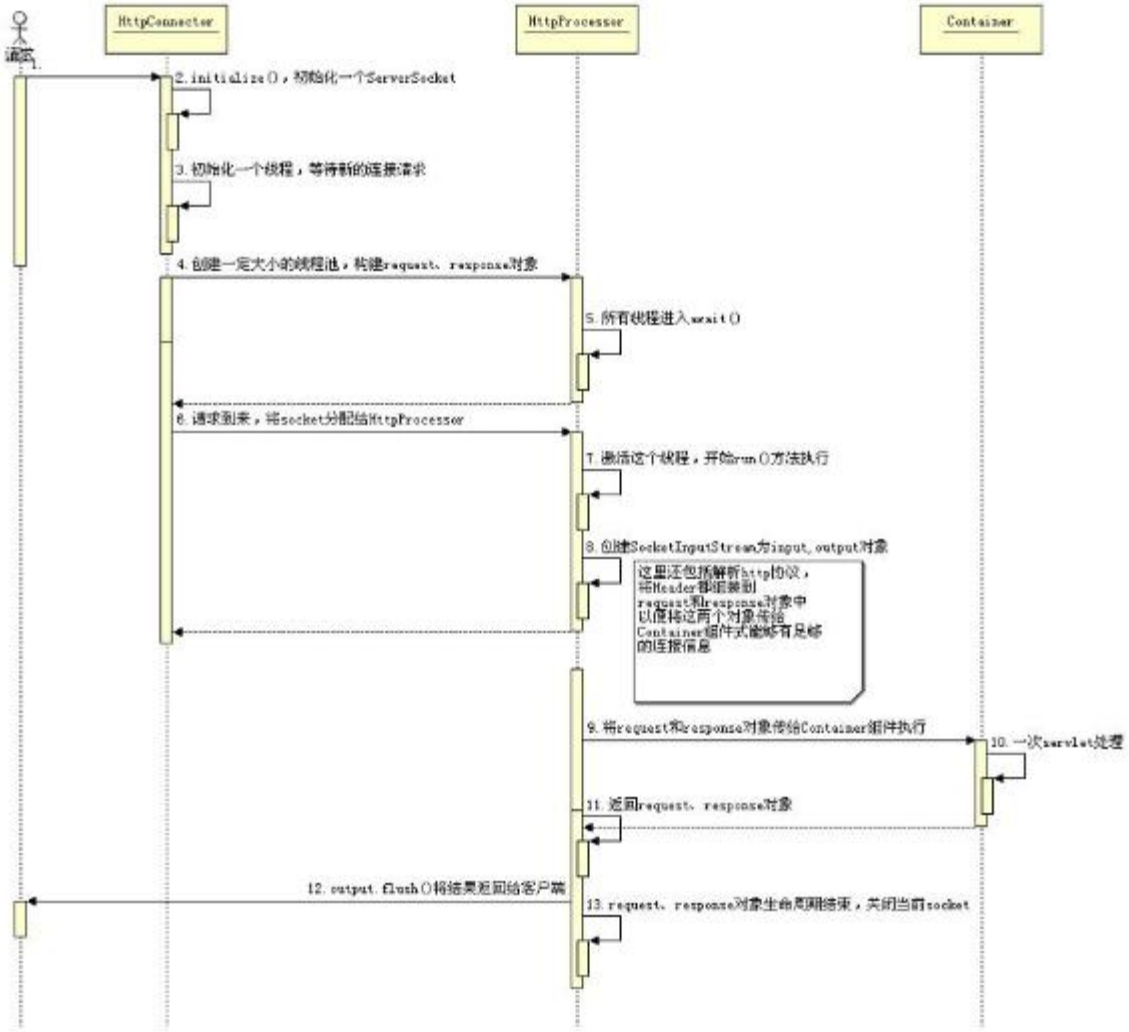
它所要做的事情也和 Start 方法差不多。

Connector 组件

Connector 组件是 Tomcat 中两个核心组件之一，它的主要任务是负责接收浏览器的发过来的 tcp 连接请求，创建一个 Request 和 Response 对象分别用于和请求端交换数据，然后会产生一个线程来处理这个请求并把产生的 Request 和 Response 对象传给处理这个请求的线程，处理这个请求的线程就是 Container 组件要做的事了。

由于这个过程比较复杂，大体的流程可以用下面的顺序图来解释：

图 6. Connector 处理一次请求顺序图

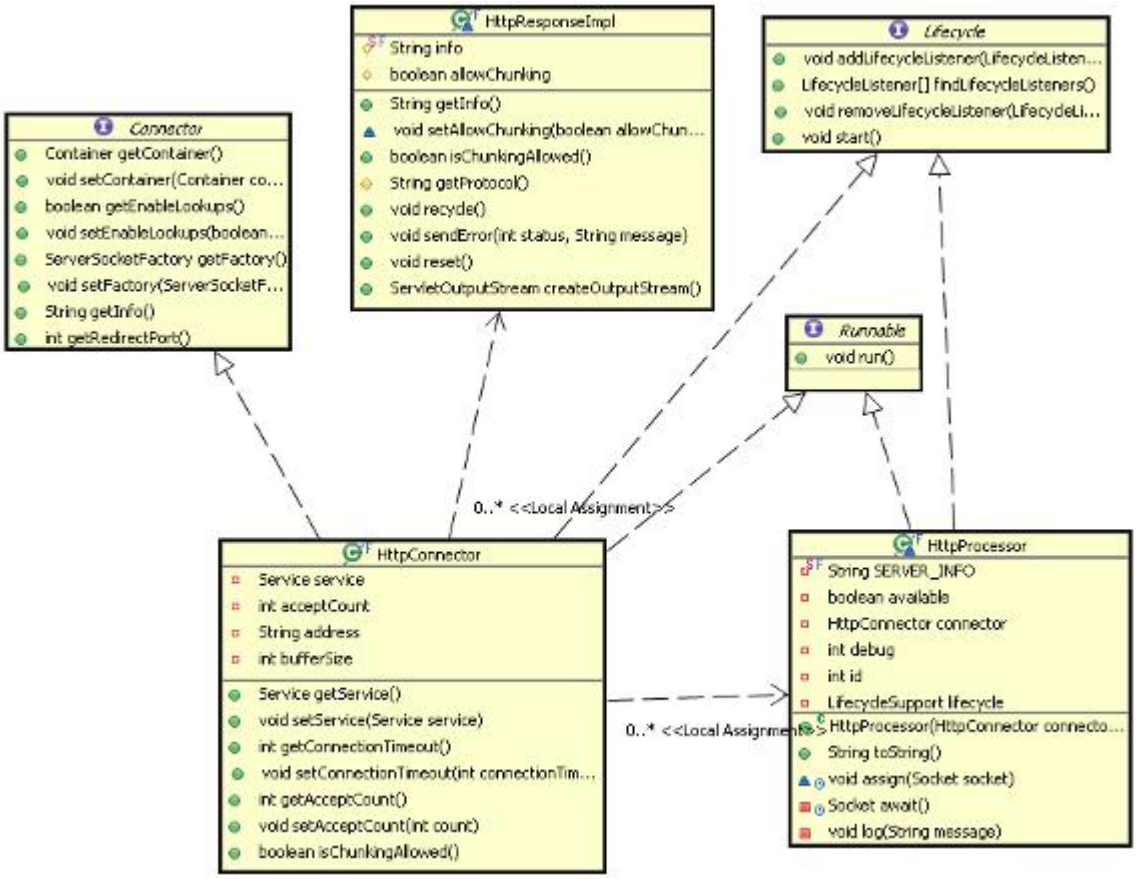


（查看清晰大图）

Tomcat5 中默认的 Connector 是 Coyote，这个 Connector 是可以选择替换的。Connector 最重要的功能就是接收连接请求然后分配线程让 Container 来处理这个请求，所以这必然是多线程的，多线程的处理是 Connector 设计的核心。Tomcat5 将这个过程更加细化，它将 Connector 划分成 Connector、Processor、Protocol, 另外 Coyote 也定义自己的 Request 和 Response 对象。

下面主要看一下 Tomcat 中如何处理多线程的连接请求，先看一下 Connector 的主要类图：

图 7. Connector 的主要类图



（查看清晰大图）

看一下 HttpConnector 的 Start 方法：

清单 6. HttpConnector.Start

```
public void start() throws LifecycleException {
   if (started)
      throw new LifecycleException
         (sm.getString("httpConnector.alreadyStarted"));
   threadName = "HttpConnector[" + port + "]";
   lifecycle.fireLifecycleEvent(START_EVENT, null);
   started = true;
   threadStart();
   while (curProcessors < minProcessors) {
      if ((maxProcessors > 0) && (curProcessors >= maxProcessors))
         break;
      HttpProcessor processor = newProcessor();
      recycle(processor);
   }
}
```

threadStart() 执行就会进入等待请求的状态，直到一个新的请求到来才会激活它继续执行，这个激活是在 HttpProcessor 的 assign 方法中，这个方法是代码如下：

清单 7. HttpProcessor.assign

```
synchronized void assign(Socket socket) {
   while (available) {
      try {
         wait();
      } catch (InterruptedException e) {
      }
   }
   this.socket = socket;
   available = true;
   notifyAll();
   if ((debug >= 1) && (socket != null))
      log(" An incoming request is being assigned");
}
```

创建 HttpProcessor 对象是会把 available 设为 false，所以当请求到来时不会进入 while 循环，将请求的 socket 赋给当期处理的 socket，并将 available 设为 true，当 available 设为 true 是 HttpProcessor 的 run 方法将被激活，接下去将会处理这次请求。

Run 方法代码如下：

清单 8. HttpProcessor.Run

```
public void run() {
   while (!stopped) {
      Socket socket = await();
      if (socket == null)
         continue;
      try {
         process(socket);
      } catch (Throwable t) {
         log("process.invoke", t);
      }
      connector.recycle(this);
   }
   synchronized (threadSync) {
      threadSync.notifyAll();
   }
}
```

解析 socket 的过程在 process 方法中，process 方法的代码片段如下：

清单 9. HttpProcessor.process

```
 private void process(Socket socket) {
    boolean ok = true;
    boolean finishResponse = true;
    SocketInputStream input = null;
    OutputStream output = null;
    try {
       input = new
SocketInputStream(socket.getInputStream(),connector.getBufferSize());
    } catch (Exception e) {
       log("process.create", e);
       ok = false;
    }
    keepAlive = true;
    while (!stopped && ok && keepAlive) {
       finishResponse = true;
       try {
          request.setStream(input);
          request.setResponse(response);
          output = socket.getOutputStream();
          response.setStream(output);
          response.setRequest(request);
          ((HttpServletResponse) response.getResponse())
                                    .setHeader("Server", SERVER_INFO);
       } catch (Exception e) {
          log("process.create", e);
          ok = false;
       }
       try {
          if (ok) {
             parseConnection(socket);
             parseRequest(input, output);
             if (!request.getRequest().getProtocol().startsWith("HTTP/0"))
                parseHeaders(input);
             if (http11) {
                ackRequest(output);
                if (connector.isChunkingAllowed())
                   response.setAllowChunking(true);
             }
          }
       。。。。。。
       try {
          ((HttpServletResponse) response).setHeader
```

```
            ("Date", FastHttpDateFormat.getCurrentDate());
         if (ok) {
            connector.getContainer().invoke(request, response);
         }
         。。。。。。
      }
      try {
         shutdownInput(input);
         socket.close();
      } catch (IOException e) {
         ;
      } catch (Throwable e) {
         log("process.invoke", e);
      }
   socket = null;
}
```

当 Connector 将 socket 连接封装成 request 和 response 对象后接下来的事情就交给 Container 来处理了。

Servlet 容器“Container”

Container 是容器的父接口，所有子容器都必须实现这个接口，Container 容器的设计用的是典型的责任链的设计模式，它有四个子容器组件构成，分别是：Engine、Host、Context、Wrapper，这四个组件不是平行的，而是父子关系，Engine 包含 Host,Host 包含 Context，Context 包含 Wrapper。通常一个 Servlet class 对应一个 Wrapper，如果有多个 Servlet 就可以定义多个 Wrapper，如果有多个 Wrapper 就要定义一个更高的 Container 了，如 Context，Context 通常就是对应下面这个配置：

清单 10. Server.xml

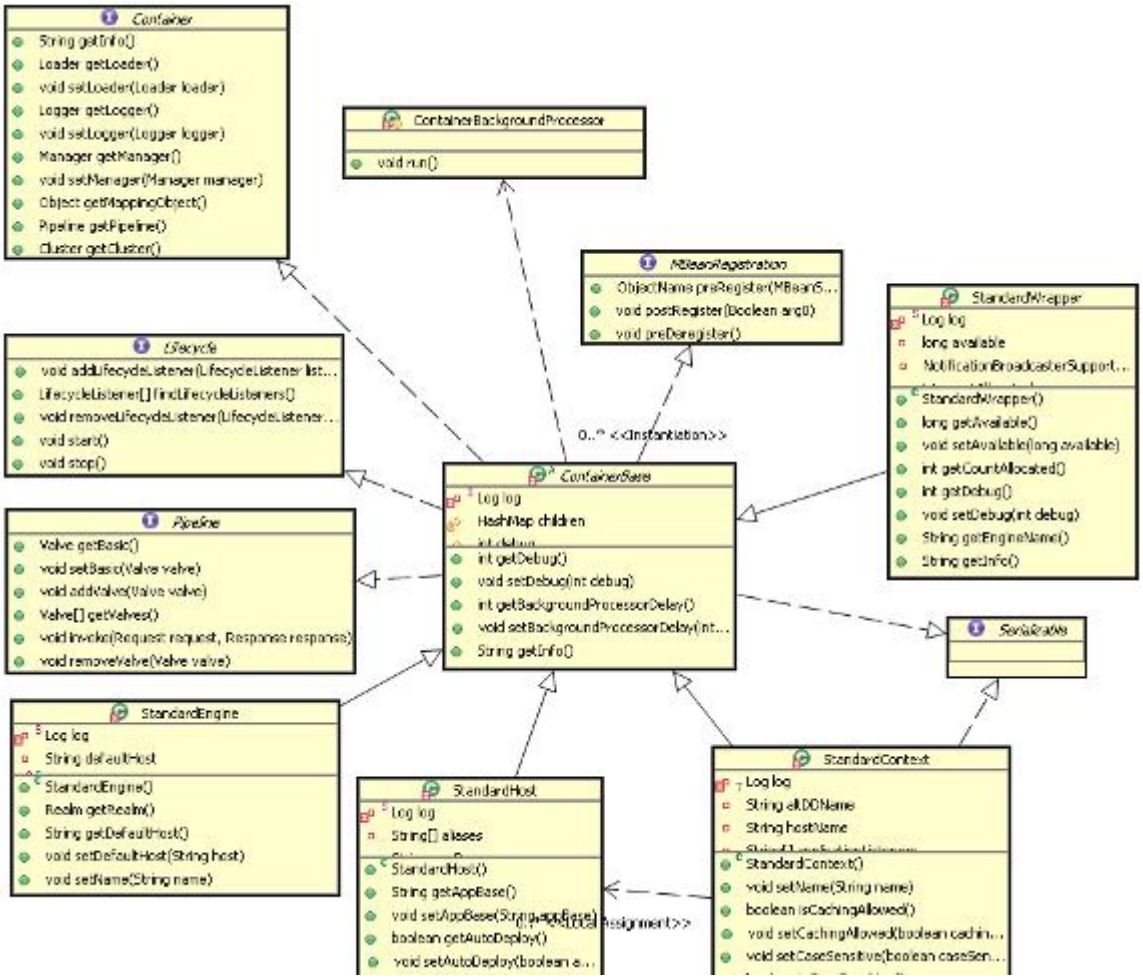
```
<Context
   path="/library"
   docBase="D:\projects\library\deploy\target\library.war"
   reloadable="true"
/>
```

容器的总体设计

Context 还可以定义在父容器 Host 中，Host 不是必须的，但是要运行 war 程序，就必须要 Host，因为 war 中必有 web.xml 文件，这个文件的解析就需要 Host 了，如果要有多个 Host 就要定义一个 top 容器 Engine 了。而 Engine 没有父容器了，一个 Engine 代表一个完整的 Servlet 引擎。

那么这些容器是如何协同工作的呢？先看一下它们之间的关系图：

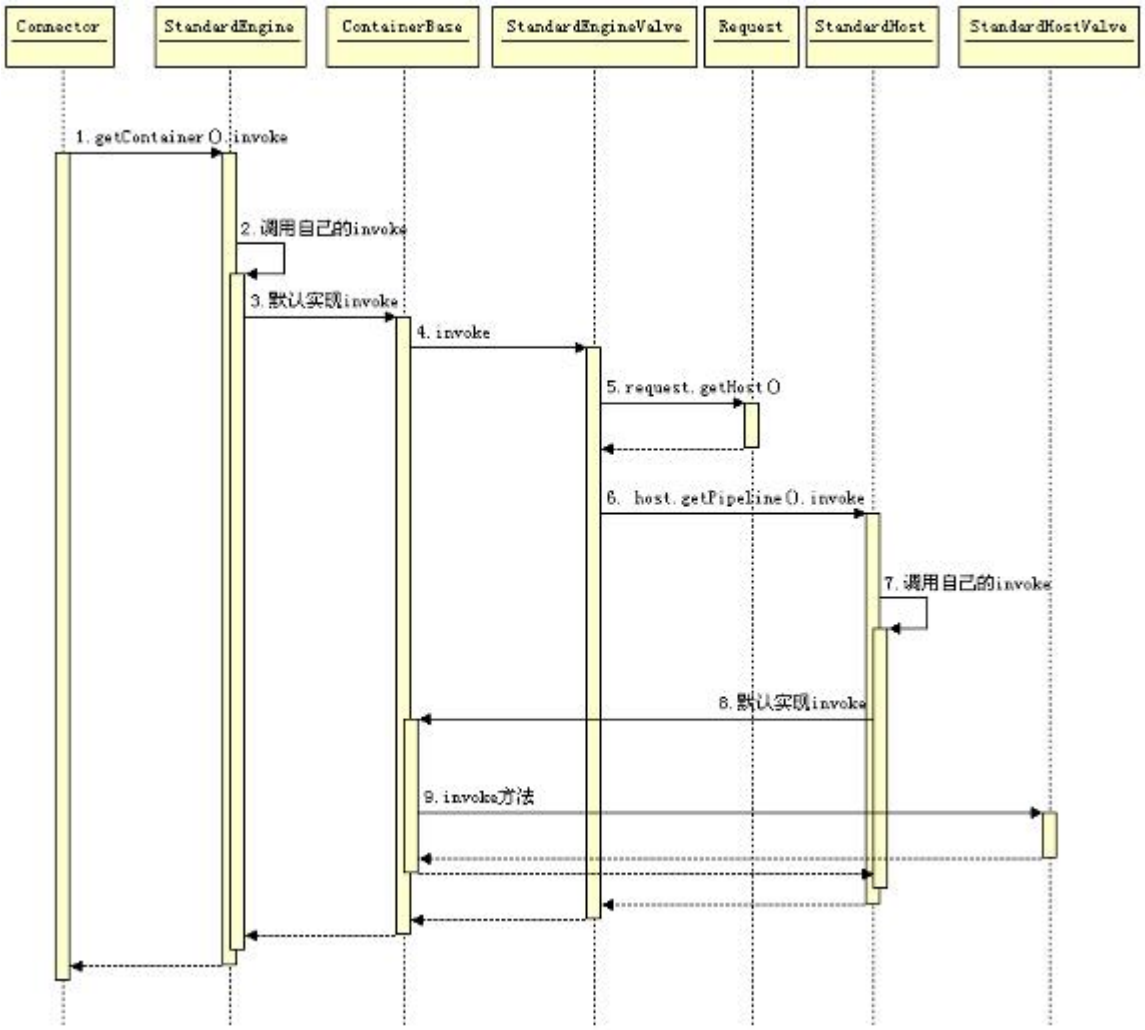
图 8. 四个容器的关系图



（查看清晰大图）

当 Connector 接受到一个连接请求时，将请求交给 Container，Container 是如何处理这个请求的？这四个组件是怎么分工的，怎么把请求传给特定的子容器的呢？又是如何将最终的请求交给 Servlet 处理。下面是这个过程的时序图：

图 9. Engine 和 Host 处理请求的时序图



（查看清晰大图）

这里看到了 Valve 是不是很熟悉，没错 Valve 的设计在其他框架中也有用的，同样 Pipeline 的原理也基本是相似的，它是一个管道，Engine 和 Host 都会执行这个 Pipeline，您可以在这个管道上增加任意的 Valve，Tomcat 会挨个执行这些 Valve，而且四个组件都会有自己的一套 Valve 集合。您怎么才能定义自己的 Valve 呢？在 server.xml 文件中可以添加，如给 Engine 和 Host 增加一个 Valve 如下：

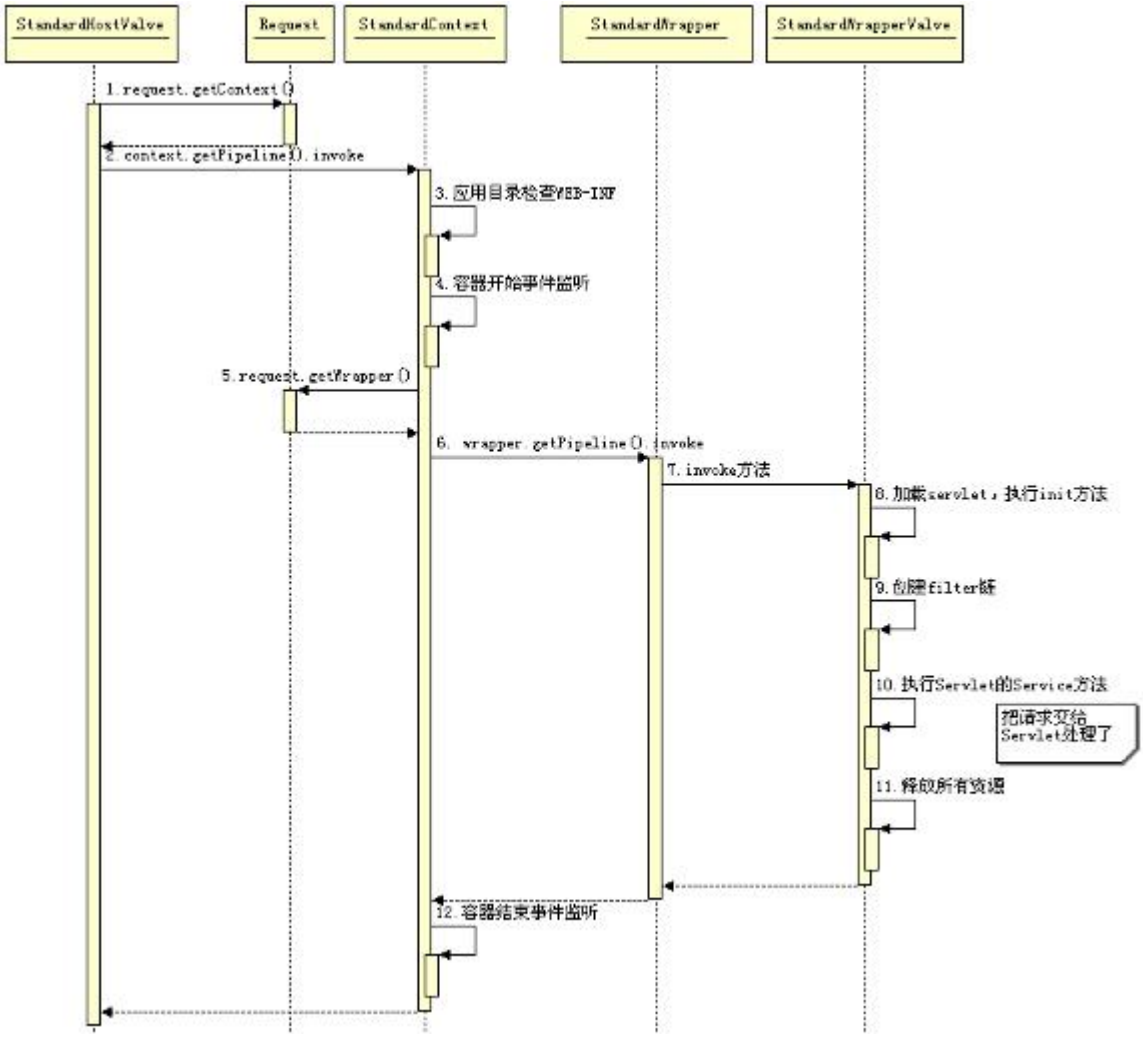
清单 11. Server.xml

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

    <Valve className="org.apache.catalina.valves.RequestDumperValve"/>
    ……
    <Host appBase="webapps" autoDeploy="true" name="localhost" unpackWARs="true"
            xmlNamespaceAware="false" xmlValidation="false">
```

```
      <Valve className="org.apache.catalina.valves.FastCommonAccessLogValve"
         directory="logs"  prefix="localhost_access_log." suffix=".txt"
         pattern="common" resolveHosts="false"/>
   ……
   </Host>
</Engine>
```

StandardEngineValve 和 StandardHostValve 是 Engine 和 Host 的默认的 Valve，它们是最后一个 Valve 负责将请求传给它们的子容器，以继续往下执行。前面是 Engine 和 Host 容器的请求过程，下面看 Context 和 Wrapper 容器时如何处理请求的。下面是处理请求的时序图：

图 10. Context 和 wrapper 的处理请求时序图



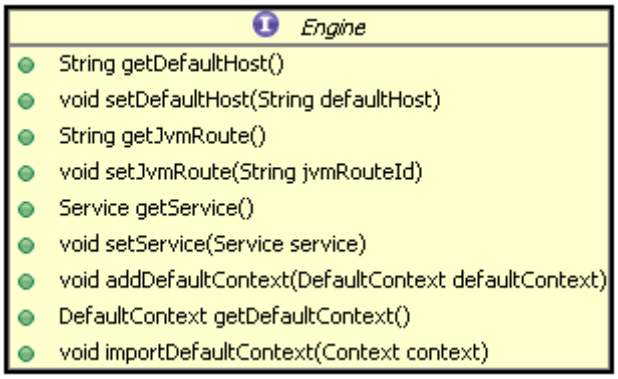
（查看清晰大图）

从 Tomcat5 开始，子容器的路由放在了 request 中，request 中保存了当前请求正在处理的 Host、Context 和 wrapper。

Engine 容器

Engine 容器比较简单，它只定义了一些基本的关联关系，接口类图如下：

图 11. Engine 接口的类结构



它的标准实现类是 StandardEngine，这个类注意一点就是 Engine 没有父容器了，如果调用 setParent 方法时将会报错。添加子容器也只能是 Host 类型的，代码如下：

清单 12. StandardEngine. addChild

```
public void addChild(Container child) {
   if (!(child instanceof Host))
      throw new IllegalArgumentException
         (sm.getString("standardEngine.notHost"));
   super.addChild(child);
}

public void setParent(Container container) {
   throw new IllegalArgumentException
      (sm.getString("standardEngine.notParent"));
}
```

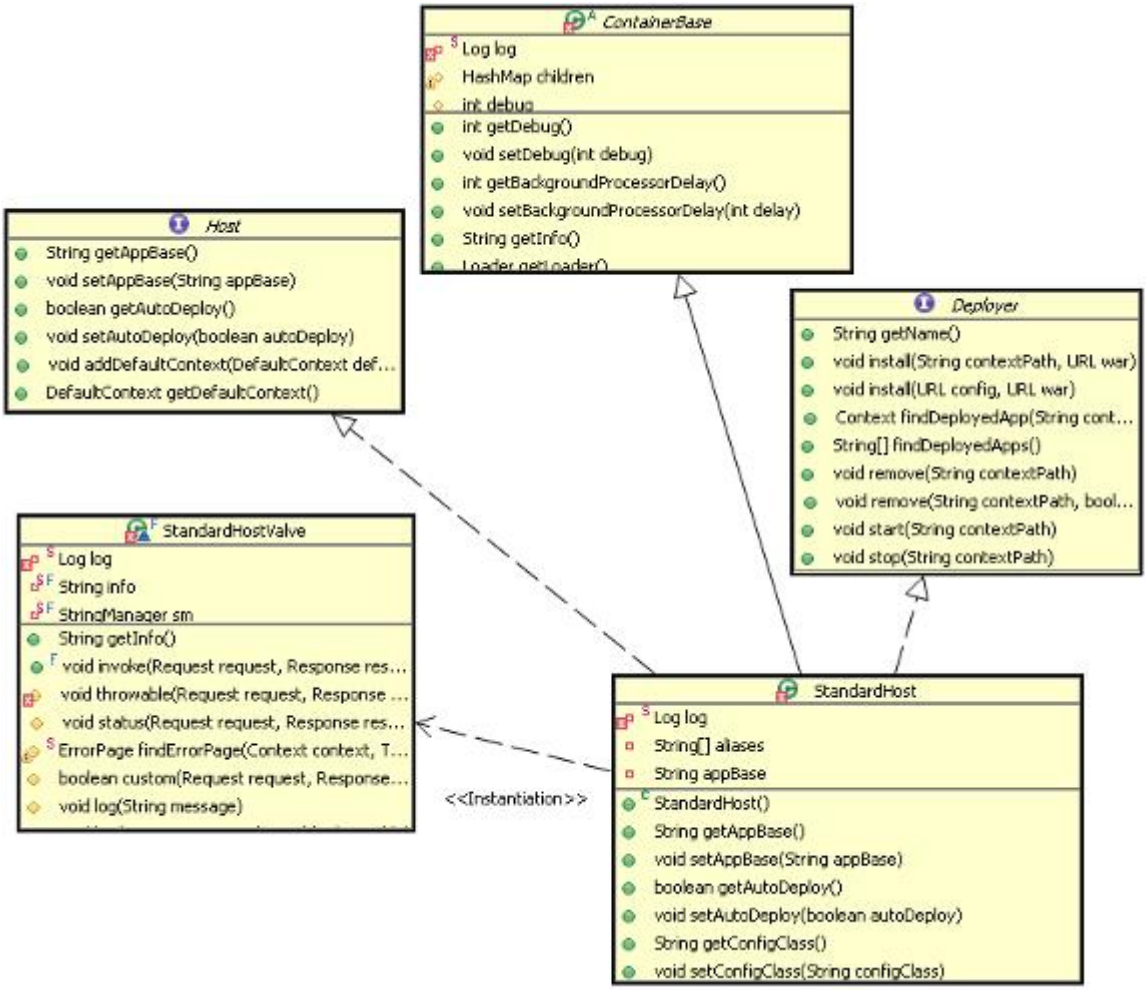
它的初始化方法也就是初始化和它相关联的组件，以及一些事件的监听。

Host 容器

Host 是 Engine 的字容器，一个 Host 在 Engine 中代表一个虚拟主机，这个虚拟主机的作用就是运行多个应用，它负责安装和展开这些应用，并且标识这个应用以便能够区分它们。它的子容器通常是 Context，它除了关联子容器外，还有就是保存一个主机应该有的信息。

下面是和 Host 相关的类关联图：

图 12. Host 相关的类图



（查看清晰大图）

从上图中可以看出除了所有容器都继承的 ContainerBase 外，StandardHost 还实现了 Deployer 接口，上图清楚的列出了这个接口的主要方法，这些方法都是安装、展开、启动和结束每个 web application。

Deployer 接口的实现是 StandardHostDeployer，这个类实现了的最要的几个方法，Host 可以调用这些方法完成应用的部署等。

Context 容器

Context 代表 Servlet 的 Context，它具备了 Servlet 运行的基本环境，理论上只要有 Context 就能运行 Servlet 了。简单的 Tomcat 可以没有 Engine 和 Host。

Context 最重要的功能就是管理它里面的 Servlet 实例，Servlet 实例在 Context 中是以 Wrapper 出现的，还有一点就是 Context 如何才能找到正确的 Servlet 来执行它呢？ Tomcat5 以前是通过一个 Mapper 类来管理的，

Tomcat5 以后这个功能被移到了 request 中，在前面的时序图中就可以发现获取子容器都是通过 request 来分配的。
Context 准备 Servlet 的运行环境是在 Start 方法开始的，这个方法的代码片段如下：

清单 13. StandardContext.start

```
public synchronized void start() throws LifecycleException {
   ……
   if( !initialized ) {
      try {
         init();
      } catch( Exception ex ) {
         throw new LifecycleException("Error initializaing ", ex);
      }
   }

        ……
   lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
   setAvailable(false);
   setConfigured(false);
   boolean ok = true;
   File configBase = getConfigBase();
   if (configBase != null) {
      if (getConfigFile() == null) {
         File file = new File(configBase, getDefaultConfigFile());
         setConfigFile(file.getPath());
         try {
            File appBaseFile = new File(getAppBase());
            if (!appBaseFile.isAbsolute()) {
               appBaseFile = new File(engineBase(), getAppBase());
            }
            String appBase = appBaseFile.getCanonicalPath();
            String basePath =
               (new File(getBasePath())).getCanonicalPath();
            if (!basePath.startsWith(appBase)) {
               Server server = ServerFactory.getServer();
               ((StandardServer) server).storeContext(this);
            }
         } catch (Exception e) {
            log.warn("Error storing config file", e);
         }
      } else {
         try {
```

```
            String canConfigFile =  (new
File(getConfigFile())).getCanonicalPath();
            if (!canConfigFile.startsWith (configBase.getCanonicalPath())) {
                File file = new File(configBase, getDefaultConfigFile());
                if (copy(new File(canConfigFile), file)) {
                    setConfigFile(file.getPath());
                }
            }
         } catch (Exception e) {
            log.warn("Error setting config file", e);
         }
      }
   }

   ……
   Container children[] = findChildren();
   for (int i = 0; i < children.length; i++) {
      if (children[i] instanceof Lifecycle)
         ((Lifecycle) children[i]).start();
   }

        if (pipeline instanceof Lifecycle)
      ((Lifecycle) pipeline).start();
   ……

}
```

它主要是设置各种资源属性和管理组件，还有非常重要的就是启动子容器和 Pipeline。

我们知道 Context 的配置文件中有个 reloadable 属性，如下面配置：

清单 14. Server.xml

```
<Context
   path="/library"
   docBase="D:\projects\library\deploy\target\library.war"
   reloadable="true"
/>
```

当这个 reloadable 设为 true 时，war 被修改后 Tomcat 会自动的重新加载这个应用。如何做到这点的呢 ？ 这个功能是在 StandardContext 的 backgroundProcess 方法中实现的，这个方法的代码如下：

清单 15. StandardContext. backgroundProcess

```
public void backgroundProcess() {
   if (!started) return;
```

```
    count = (count + 1) % managerChecksFrequency;
    if ((getManager() != null) && (count == 0)) {
        try {
            getManager().backgroundProcess();
        } catch ( Exception x ) {
            log.warn("Unable to perform background process on manager",x);
        }
    }
    if (getLoader() != null) {
        if (reloadable && (getLoader().modified())) {
            try {
                Thread.currentThread().setContextClassLoader
                    (StandardContext.class.getClassLoader());
                reload();
            } finally {
                if (getLoader() != null) {
                    Thread.currentThread().setContextClassLoader
                        (getLoader().getClassLoader());
                }
            }
        }
        if (getLoader() instanceof WebappLoader) {
            ((WebappLoader) getLoader()).closeJARs(false);
        }
    }
}
```

它会调用 reload 方法，而 reload 方法会先调用 stop 方法然后再调用 Start 方法，完成 Context 的一次重新加载。可以看出执行 reload 方法的条件是 reloadable 为 true 和应用被修改，那么这个 backgroundProcess 方法是怎么被调用的呢？

这个方法是在 ContainerBase 类中定义的内部类 ContainerBackgroundProcessor 被周期调用的，这个类是运行在一个后台线程中，它会周期的执行 run 方法，它的 run 方法会周期调用所有容器的 backgroundProcess 方法，因为所有容器都会继承 ContainerBase 类，所以所有容器都能够在 backgroundProcess 方法中定义周期执行的事件。

Wrapper 容器

Wrapper 代表一个 Servlet，它负责管理一个 Servlet，包括的 Servlet 的装载、初始化、执行以及资源回收。Wrapper 是最底层的容器，它没有子容器了，所以调用它的 addChild 将会报错。

Wrapper 的实现类是 StandardWrapper，StandardWrapper 还实现了拥有一个 Servlet 初始化信息的 ServletConfig，由此看出 StandardWrapper 将直接和 Servlet 的各种信息打交道。
下面看一下非常重要的一个方法 loadServlet，代码片段如下：

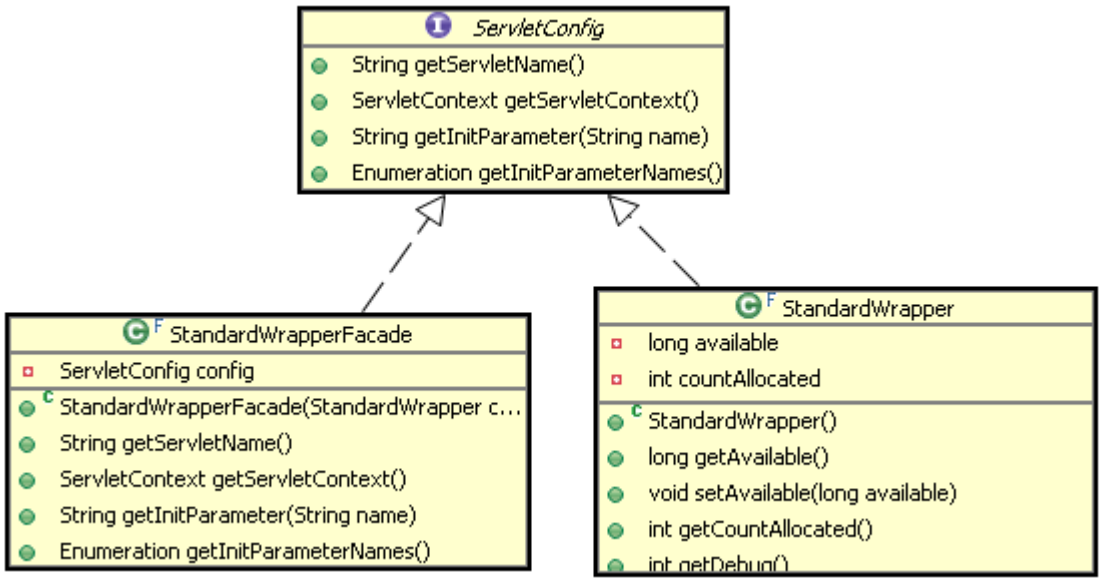
清单 16. StandardWrapper.loadServlet

```
public synchronized Servlet loadServlet() throws ServletException {
   ……
   Servlet servlet;
   try {
      ……
      ClassLoader classLoader = loader.getClassLoader();
      ……
      Class classClass = null;
      ……
      servlet = (Servlet) classClass.newInstance();
      if ((servlet instanceof ContainerServlet) &&
         (isContainerProvidedServlet(actualClass) ||
         ((Context)getParent()).getPrivileged() )) {
            ((ContainerServlet) servlet).setWrapper(this);
      }
      classLoadTime=(int) (System.currentTimeMillis() -t1);
      try {

instanceSupport.fireInstanceEvent(InstanceEvent.BEFORE_INIT_EVENT,servlet);
         if( System.getSecurityManager() != null) {
            Class[] classType = new Class[]{ServletConfig.class};
            Object[] args = new Object[]{((ServletConfig)facade)};
            SecurityUtil.doAsPrivilege("init",servlet,classType,args);
         } else {
            servlet.init(facade);
         }
         if ((loadOnStartup >= 0) && (jspFile != null)) {
            ……
            if( System.getSecurityManager() != null) {
               Class[] classType = new Class[]{ServletRequest.class,
                  ServletResponse.class};
               Object[] args = new Object[]{req, res};
               SecurityUtil.doAsPrivilege("service",servlet,classType,args);
            } else {
               servlet.service(req, res);
            }
         }
```

```
instanceSupport.fireInstanceEvent(InstanceEvent.AFTER_INIT_EVENT,servlet);
            ……

          return servlet;
}
```

它基本上描述了对 Servlet 的操作，当装载了 Servlet 后就会调用 Servlet 的 init 方法，同时会传一个 StandardWrapperFacade 对象给 Servlet，这个对象包装了 StandardWrapper，ServletConfig 与它们的关系图如下：

图 13. ServletConfig 与 StandardWrapperFacade 、StandardWrapper 的关系



Servlet 可以获得的信息都在 StandardWrapperFacade 封装，这些信息又是在 StandardWrapper 对象中拿到的。所以 Servlet 可以通过 ServletConfig 拿到有限的容器的信息。

当 Servlet 被初始化完成后，就等着 StandardWrapperValve 去调用它的 service 方法了，调用 service 方法之前要调用 Servlet 所有的 filter。

Tomcat 中其它组件

Tomcat 还有其它重要的组件，如安全组件 security、logger 日志组件、session、mbeans、naming 等其它组件。这些组件共同为 Connector 和 Container 提供必要的服务。

设计模式分析

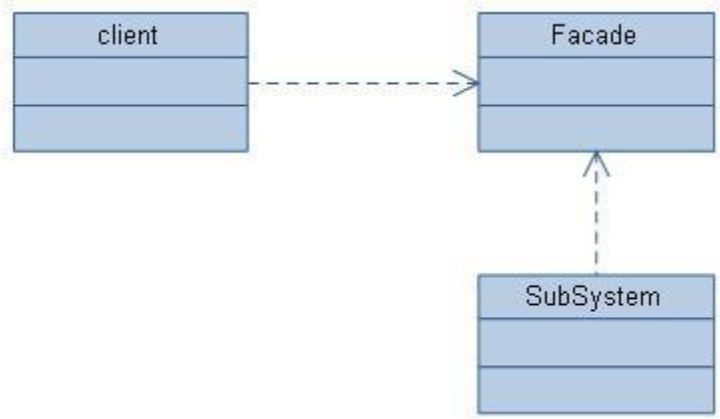
门面设计模式

门面设计模式在 Tomcat 中有多处使用，在 Request 和 Response 对象封装中、Standard Wrapper 到 ServletConfig 封装中、ApplicationContext 到 ServletContext 封装中等都用到了这种设计模式。

门面设计模式的原理

这么多场合都用到了这种设计模式，那这种设计模式究竟能有什么作用呢？顾名思义，就是将一个东西封装成一个门面好与人家更容易进行交流，就像一个国家的外交部一样。
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。这就是门面设计模式存在的意义。
门面设计模式示意图如下：

图 1. 门面示意图

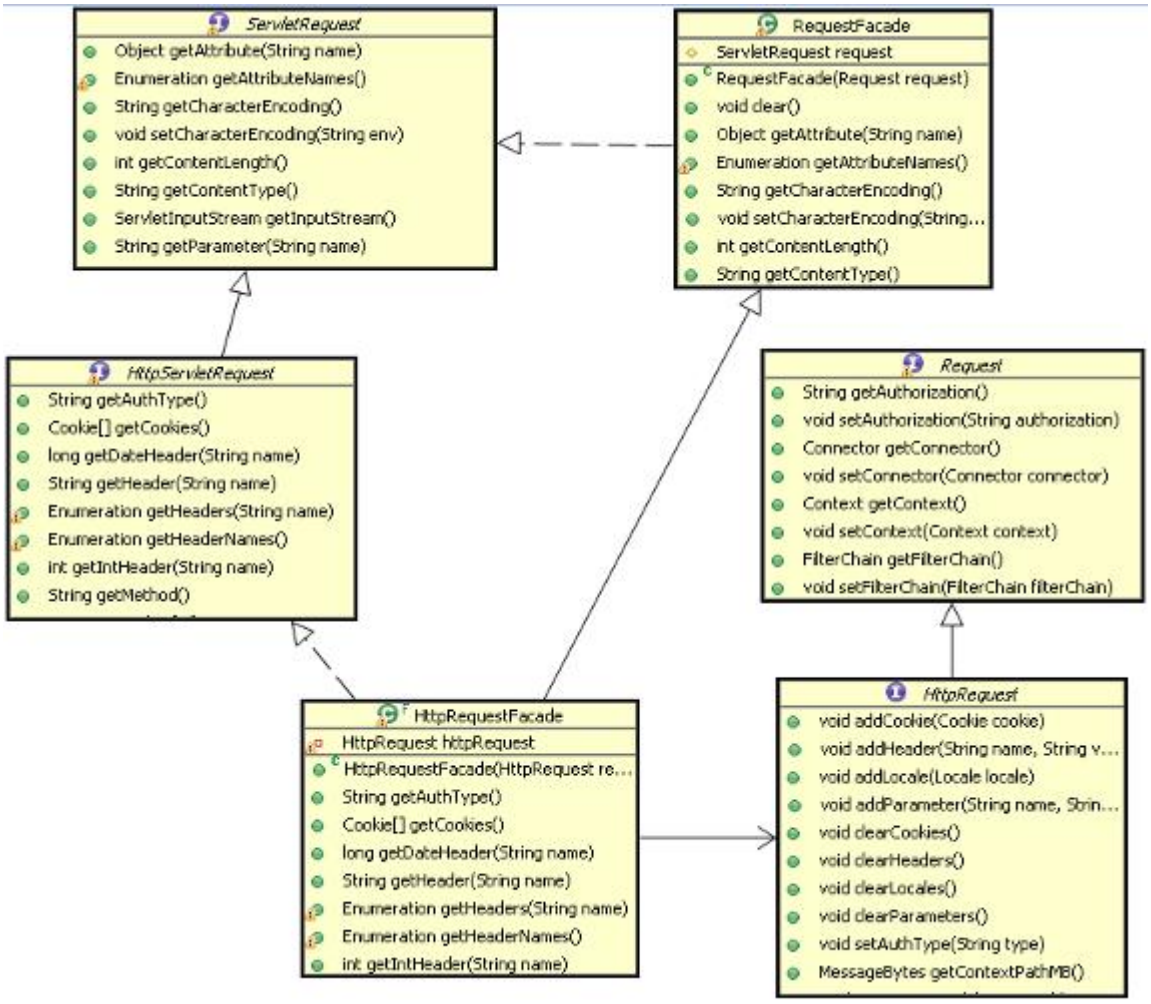


Client 只能访问到 Façade 中提供的数据是门面设计模式的关键，至于 Client 如何访问 Façade 和 Subsystem 如何提供 Façade 门面设计模式并没有规定死。

Tomcat 的门面设计模式示例

Tomcat 中门面设计模式使用的很多，因为 Tomcat 中有很多不同组件，每个组件要相互交互数据，用门面模式隔离数据是个很好的方法。
下面是 Request 上使用的门面设计模式：

图 2. Request 的门面设计模式类图



从图中可以看出 HttpRequestFacade 类封装了 HttpRequest 接口能够提供数据，通过 HttpRequestFacade 访问到的数据都被代理到 HttpRequest 中，通常被封装的对象都被设为 Private 或者 Protected 访问修饰，以防止在 Façade 中被直接访问。

观察者设计模式

这种设计模式也是常用的设计方法通常也叫发布 - 订阅模式，也就是事件监听机制，通常在某个事件发生的前后会触发一些操作。

观察者模式的原理

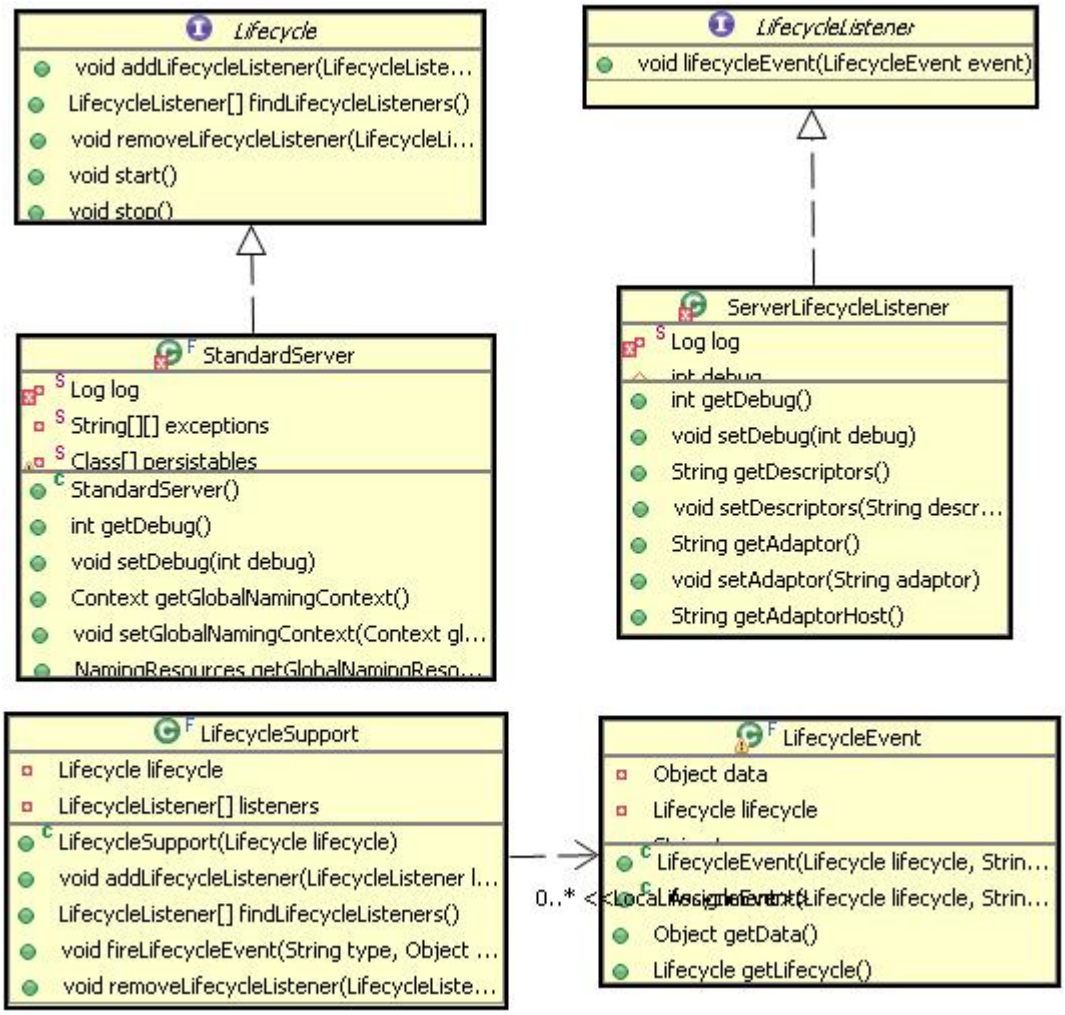
观察者模式原理也很简单，就是你在做事的时候旁边总有一个人在盯着你，当你做的事情是它感兴趣的时候，它就会跟着做另外一些事情。但是盯着你的人必须要到你那去登记，不然你无法通知它。观察者模式通常包含下面这几个角色：

- Subject 就是抽象主题：它负责管理所有观察者的引用，同时定义主要的事件操作。
- ConcreteSubject 具体主题：它实现了抽象主题的所有定义的接口，当自己发生变化时，会通知所有观察者。
- Observer 观察者：监听主题发生变化相应的操作接口。

Tomcat 的观察者模式示例

Tomcat 中观察者模式也有多处使用，前面讲的控制组件生命周期的 Lifecycle 就是这种模式的体现，还有对 Servlet 实例的创建、Session 的管理、Container 等都是同样的原理。下面主要看一下 Lifecycle 的具体实现。
Lifecycle 的观察者模式结构图：

图 3. Lifecycle 的观察者模式结构图



上面的结构图中，LifecycleListener 代表的是抽象观察者，它定义一个 lifecycleEvent 方法，这个方法就是当主题变化时要执行的方法。ServerLifecycleListener 代表的是具体的观察者，它实现了 LifecycleListener 接口的方法，就是这个具体的观察者具体的实现方式。Lifecycle 接口代表的是抽象主题，它定义了管理观察者的方法和它要所做的其它方法。而 StandardServer 代表的是具体主题，它实现了抽象主题的所有方法。这里 Tomcat 对观察者做了扩展，增加了另外两个类：LifecycleSupport、LifecycleEvent，它们作为辅助类扩展了观察者的功能。LifecycleEvent 使得可以定义事件类别，不同的事件可区别处理，更加灵活。LifecycleSupport 类代理了主题对多观察者的管理，将这个管理抽出来统一实现，以后如果修改只要修改 LifecycleSupport 类就可以了，不需要去修改所有具体主题，因为所有具体主题的对观察者的操作都被代理给 LifecycleSupport 类了。这可以认为是观察者模式的改进版。

LifecycleSupport 调用观察者的方法代码如下：

清单 1. LifecycleSupport 中的 fireLifecycleEvent 方法

```
public void fireLifecycleEvent(String type, Object data) {
   LifecycleEvent event = new LifecycleEvent(lifecycle, type, data);
   LifecycleListener interested[] = null;
   synchronized (listeners) {
      interested = (LifecycleListener[]) listeners.clone();
   }
   for (int i = 0; i < interested.length; i++)
      interested[i].lifecycleEvent(event);
}
```

主题是怎么通知观察者呢？看下面代码：

清单 2. 容器中的 start 方法

```
public void start() throws LifecycleException {
   lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
   lifecycle.fireLifecycleEvent(START_EVENT, null);
   started = true;
   synchronized (services) {
      for (int i = 0; i < services.length; i++) {
         if (services[i] instanceof Lifecycle)
            ((Lifecycle) services[i]).start();
         }
      }
   lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}
```

命令设计模式

前面把 Tomcat 中两个核心组件 Connector 和 Container，比作一对夫妻。男的将接受过来的请求以命令的方式交给女主人。对应到 Connector 和 Container，Connector 也是通过命令模式调用 Container 的。

命令模式的原理

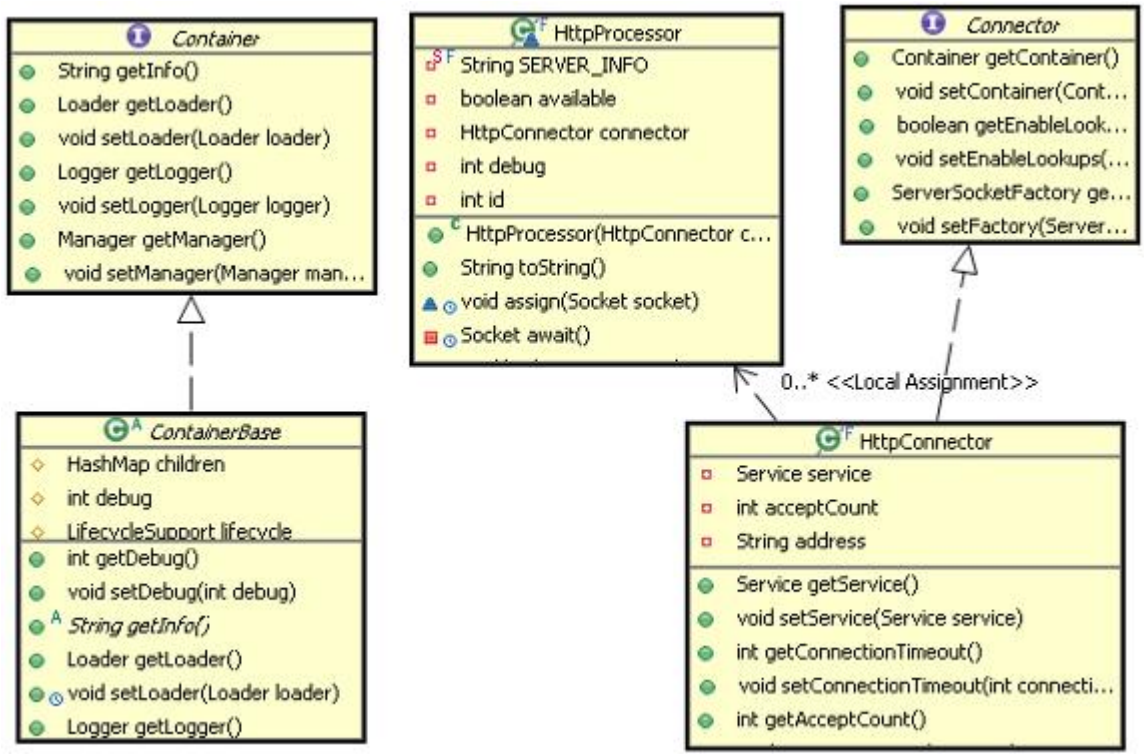
命令模式主要作用就是封装命令，把发出命令的责任和执行命令的责任分开。也是一种功能的分工。不同的模块可以对同一个命令做出不同解释。
下面是命令模式通常包含下面几个角色：

- Client：创建一个命令，并决定接受者
- Command 命令：命令接口定义一个抽象方法
- ConcreteCommand：具体命令，负责调用接受者的相应操作
- Invoker 请求者：负责调用命令对象执行请求
- Receiver 接受者：负责具体实施和执行一次请求

Tomcat 中的命令模式的示例

Tomcat 中命令模式在 Connector 和 Container 组件之间有体现，Tomcat 作为一个应用服务器，无疑会接受到很多请求，如何分配和执行这些请求是必须的功能。
下面看一下 Tomcat 是如何实现命令模式的，下面是 Tomcat 命令模式的结构图：

图 4. Tomcat 命令模式的结构图



Connector 作为抽象请求者，HttpConnector 作为具体请求者。HttpProcessor 作为命令。Container 作为命令的抽象接受者，ContainerBase 作为具体的接受者。客户端就是应用服务器 Server 组件了。Server 首先创建命令请求者 HttpConnector 对象，然后创建命令 HttpProcessor 命令对象。再把命令对象交给命令接受者 ContainerBase 容器来处理命令。命令的最终是被 Tomcat 的 Container 执行的。命令可以以队列的方式进来，Container 也可以以不同的方式来处理请求，如 HTTP1.0 协议和 HTTP1.1 的处理方式就会不同。

责任链模式

Tomcat 中一个最容易发现的设计模式就是责任链模式，这个设计模式也是 Tomcat 中 Container 设计的基础，整个容器的就是通过一个链连接在一起，这个链一直将请求正确的传递给最终处理请求的那个 Servlet。

责任链模式的原理

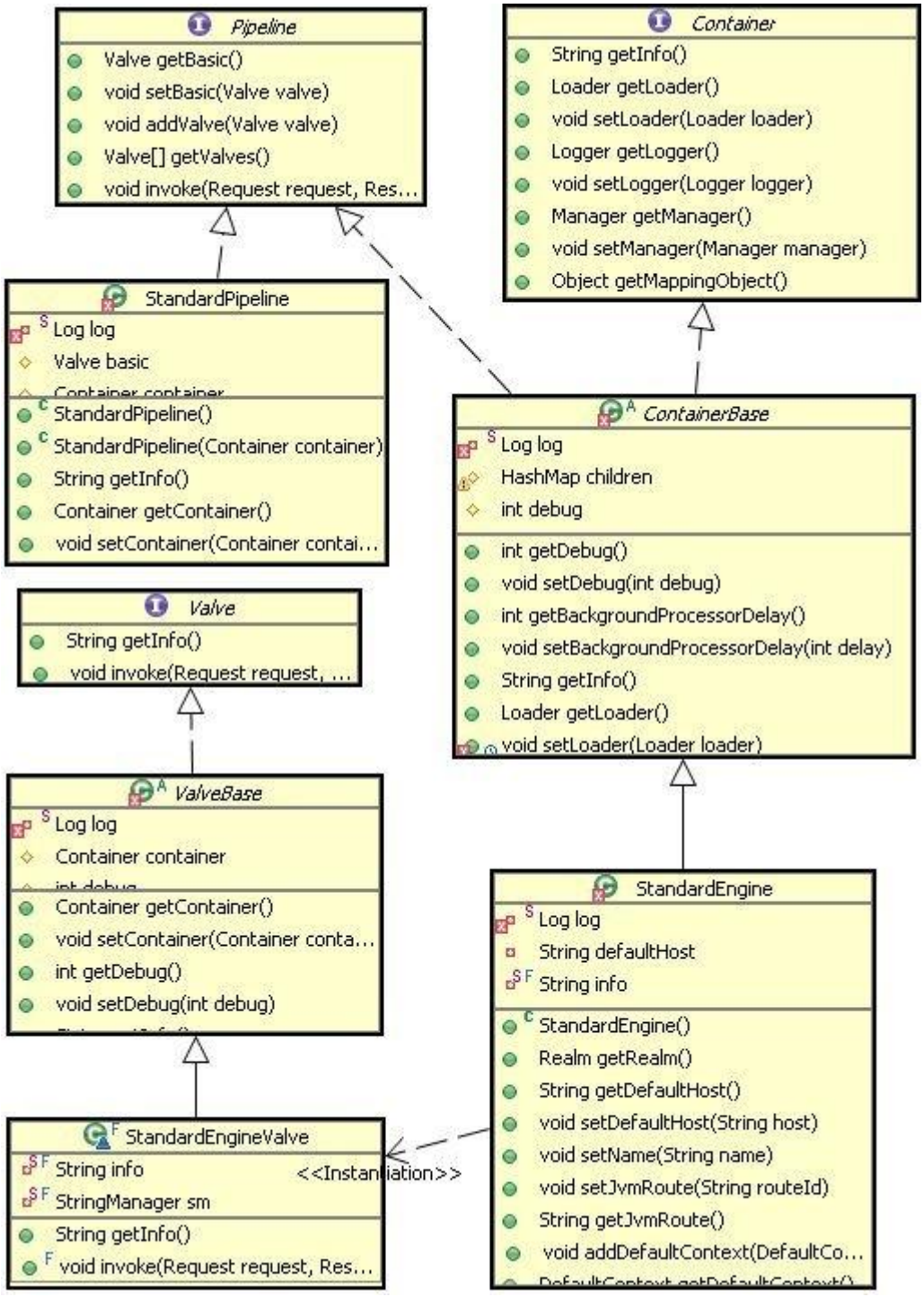
责任链模式，就是很多对象有每个对象对其下家的引用而连接起来形成一条链，请求在这条链上传递，直到链上的某个对象处理此请求，或者每个对象都可以处理请求，并传给下一家，直到最终链上每个对象都处理完。这样可以不影响客户端而能够在链上增加任意的处理节点。
通常责任链模式包含下面几个角色：

- Handler（抽象处理者）：定义一个处理请求的接口
- ConcreteHandler（具体处理者）：处理请求的具体类，或者传给下家

Tomcat 中责任链模式示例

在 tomcat 中这种设计模式几乎被完整的使用，tomcat 的容器设置就是责任链模式，从 Engine 到 Host 再到 Context 一直到 Wrapper 都是通过一个链传递请求。
Tomcat 中责任链模式的类结构图如下：

图 5. Tomcat 责任链模式的结构图



上图基本描述了四个子容器使用责任链模式的类结构图，对应的责任链模式的角色，Container 扮演抽象处理者角色，具体处理者由 StandardEngine 等子容器扮演。与标准的责任链不同的是，这里引入了 Pipeline 和 Valve 接口。他们有什么作用呢？

实际上 Pipeline 和 Valve 是扩展了这个链的功能，使得在链往下传递过程中，能够接受外界的干预。Pipeline 就是连接每个子容器的管子，里面传递的

Request 和 Response 对象好比管子里流的水，而 Valve 就是这个管子上开的一个个小口子，让你有机会能够接触到里面的水，做一些额外的事情。
为了防止水被引出来而不能流到下一个容器中，每一段管子最后总有一个节点保证它一定能流到下一个子容器，所以每个容器都有一个 StandardXXXValve。只要涉及到这种有链式是处理流程这是一个非常值得借鉴的模式。