Tomcat

Tomcat 参数调优

默认值：

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

修改配置：

```
<Connector port="8080" protocol="org.apache.coyote.http11.Http11Nio2Protocol"
        connectionTimeout="20000"
        redirectPort="8443"
        executor="TomcatThreadPool"
        enableLookups="false"
        acceptCount="100"
        maxPostSize="10485760"
        compression="on"
        disableUploadTimeout="true"
        compressionMinSize="2048"
        noCompressionUserAgents="gozilla, traviata"
        acceptorThreadCount="2"
compressableMimeType="text/html,text/xml,text/plain,text/css,text/javascript,application/javascript"
  URIEncoding="utf-8"/>
```

Connector

Tomcat 有一个 acceptor 线程来 accept socket 连接，然后有工作线程来进行业务处理。对于 client 端的一个请求进来，流程是这样的：tcp 的三次握手建立连接，建立连接的过程中，OS 维护了半连接队列(syn 队列)以及完全连接队列(accept 队列)，在第三次握手之后，server 收到了 client 的 ack，则进入 establish 的状态，然后该连接由 syn 队列移动到 accept 队列。

Tomcat 的 acceptor 线程则负责从 accept 队列中取出该 connection，接受该 connection，然后交给工作线程去处理(读取请求参数、处理逻辑、返回响应等等；如果该连接不是 keep alived 的话，则关闭该连接，然后该工作线程释放回线程池，如果是 keep alived 的话，则等待下一个数据包的到来直到 keepAliveTimeout，然后关闭该连接释放回线程池)，
然后自己接着去 accept 队列取 connection(当当前 socket 连接超过 maxConnections 的时候，acceptor 线程自己会阻塞等待，等连接降下去之后，才去处理 accept 队列的下一个连接)。acceptCount 指的就是这个 accept 队列的大小。

protocol（IO 方式）

Tomcat 8 设置 nio2 更好：org.apache.coyote.http11.Http11Nio2Protocol（如果这个用不了，就用下面那个）
Tomcat 6、7 设置 nio 更好：org.apache.coyote.http11.Http11NioProtocol
apr：调用 httpd 核心链接库来读取或文件传输，从而提高 tomat 对静态文件的处理性能。Tomcat APR 模式也是 Tomcat 在高并发下的首选运行模式

URIEncoding（URL 编码）

URIEncoding=”UTF-8”

使得 Tomcat 可以解析含有中文名的文件的 url

Executor（启用 Worker 线程池）

```
<Executor name="TomcatThreadPool" namePrefix="catalina-exec-"
                  maxThreads="150" minSpareThreads="100"
             prestartminSpareThreads="true" maxQueueSize="100"/>
```

minSpareThreads（初始化时创建的线程数，类似于 corePoolSize）

最小备用线程数，Tomcat 启动时的初始化的线程数。

maxThreads（最大并发数，类似于 maxPoolSize）

maxThreads Tomcat 使用线程来处理接收的每个请求。这个值表示 Tomcat 可创建的最大的线程数，即最大并发数。
默认设置 200，一般建议在 500 ~ 800，根据硬件设施和业务来判断。
虽然 client 的 socket 连接上了，但是可能都在 Tomcat 的 task queue 里头，等待 worker 线程处理返回响应。

maxQueueSize（Task 队列大小）

指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理，默认设置 100。

connectionTimeout（超时时间）

connectionTimeout 为网络连接超时时间毫秒数。

enableLookups（是否允许 DNS 查询）

enableLookups="false" 为了消除 DNS 查询对性能的影响我们可以关闭 DNS 查询，方式是修改 server.xml 文件中的 enableLookups 参数值。

maxConnections（接收的最大连接数）

这个值表示最多可以有多少个 socket 连接到 Tomcat 上。NIO 模式下默认是 10000.
当连接数达到最大值后，系统会继续接收连接但不会超过 acceptCount 的值。

acceptCount（accept 队列大小）

当 accept 队列满了之后，即使 client 继续向 server 发送 ACK 的包，也会不被响应，此时，server 通过/proc/sys/net/ipv4/tcp_abort_on_overflow 来决定如何返回，0 表示直接丢丢弃该 ACK，1 表示发送 RST 通知 client；相应的，client 则会分别返回 read timeout 或者 connection reset by peer。

acceptCount 在源码里对应的是 backlog 参数。backlog 参数提示内核监听队列的最大长度。监听队列的长度如果超过 backlog，服务器将不受理新的客户连接，客户端也将收到 ECONNREFUSED 错误信息。Linux 自内核版本 2.2 之后，它只表示处于完全连接状态的 socket 的上限，处于半连接状态的 socket 的上限则由/proc/sys/net/ipv4/tcp_max_syn_backlog 内核参数定义。

client 端的 socket 等待队列：
当第一次握手，建立半连接状态：client 通过 connect 向 server 发出 SYN 包时，client 会维护一个 socket 队列，如果 socket 等待队列满了，而 client 也会由此返回 connection time out，只要是 client 没有收到 第二次握手 SYN+ACK，3s 之后，client 会再次发送，如果依然没有收到，9s 之后会继续发送。

server 端的半连接队列(syn 队列)：
此时 server 会维护一个 SYN 队列，半连接 syn 队列的长度为 max(64, /proc/sys/net/ipv4/tcp_max_syn_backlog) ，在机器的 tcp_max_syn_backlog 值在 /proc/sys/net/ipv4/tcp_max_syn_backlog 下配置，当 server 收到 client 的 SYN 包后，会进行第二次握手发送 SYN＋ACK 的包加以确认，client 的 TCP 协议栈会唤醒 socket 等待队列，发出 connect 调用。

server 端的完全连接队列(accpet 队列)：
当第三次握手时，当 server 接收到 ACK 报之后， 会进入一个新的叫 accept 的队列，该队列的长度为 min(backlog, somaxconn)，默认情况下，somaxconn 的值为 128，表示最多有 129 的 ESTAB 的连接等待 accept()，而 backlog 的值则应该是由 int listen(int sockfd, int backlog) 中的第二个参数指定，listen 里面的 backlog 可以有我们的应用程序去定义的。

acceptorThreadCount（用于接收请求的线程数）

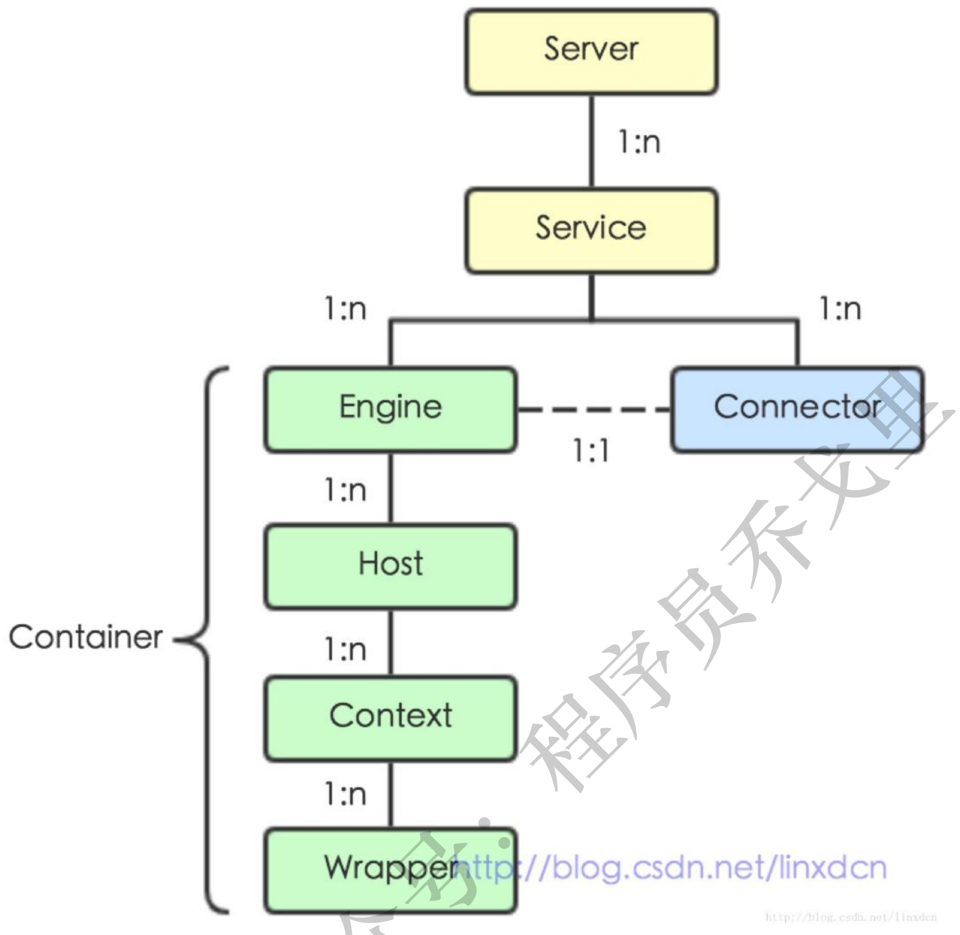
用于接收连接的线程的数量，默认值是 1。一般这个指需要改动的时候是因为该服务器是一个多核 CPU，如果是多核 CPU 一般配置为 2。

HTTP 压缩

```
compression="on" compressionMinSize="2048"
compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain"
```

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。
1)compression="on" 打开压缩功能
2)compressionMinSize="2048" 启用压缩的输出内容大小，这里面默认为 2KB
3)noCompressionUserAgents="gozilla, traviata" 对于以下的浏览器，不启用压缩
4)compressableMimeType="text/html,text/xml" 压缩类型

组件与框架


Server
1:n
Service
1:n
1:n
Engine
Connector
1:1
1:n
Host
Container
1:n
Context
1:n
Wrapper
http://blog.csdn.net/linxdcn
http://blog.csdn.net/linxdcn

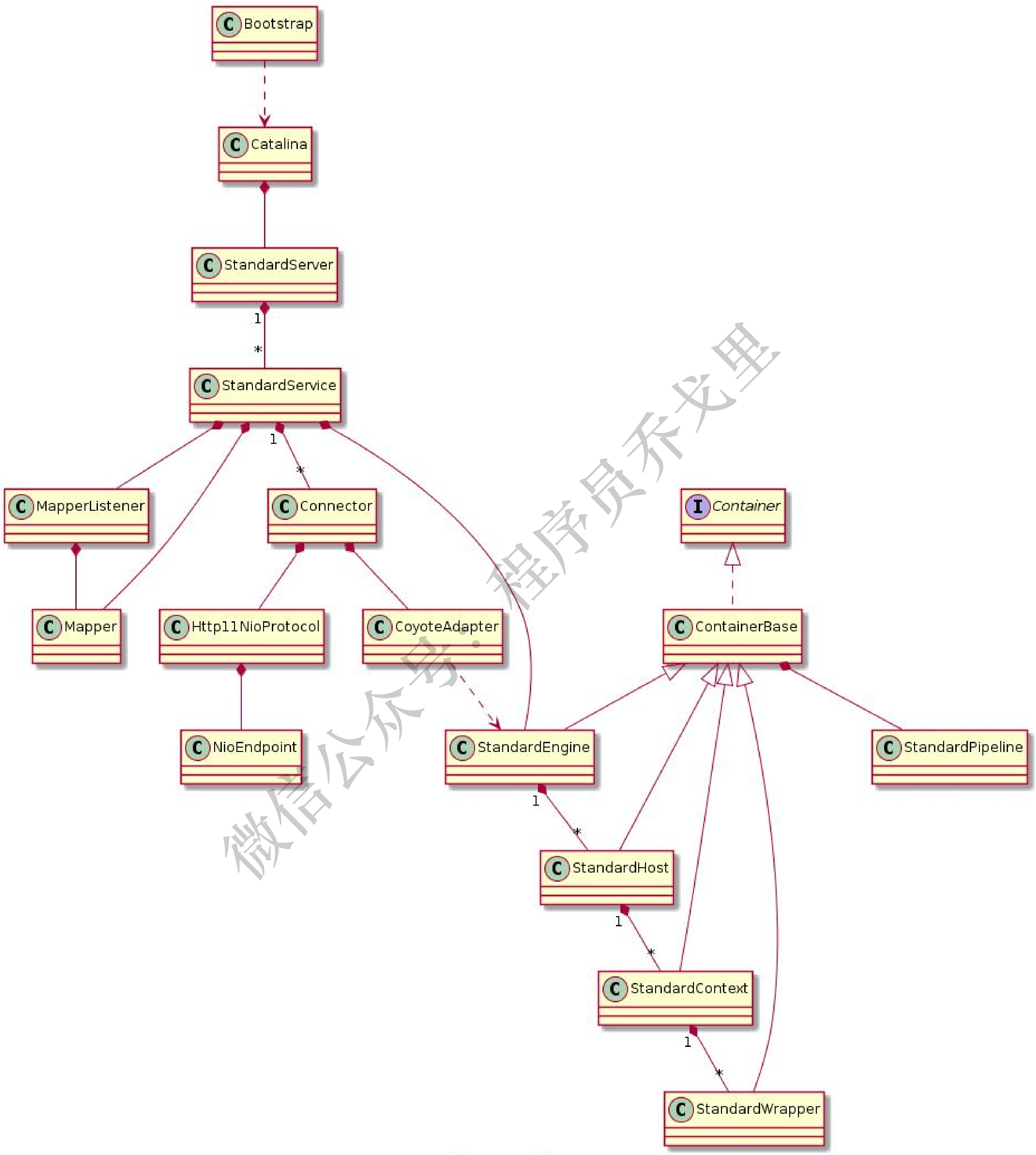


图 1 Tomcat architecture

- Bootstrap：作为 Tomcat 对外界的启动类,在 $CATALINA_BASE/bin 目录下，它通过反

射创建 Catalina 的实例并对其进行初始化及启动。

- Catalina：解析 $CATALINA_BASE/conf/server.xml 文件并创建 StandardServer、StandardService、StandardEngine、StandardHost 等
- Server：代表整个 Catalina Servlet 容器，可以包含一个或多个 Service
- Service：包含一个或多个 **Connector**，和一个 **Engine**，Connector 和 Engine 都是在解析 conf/server.xml 文件时创建的，Engine 在 Tomcat 的标准实现是 StandardEngine
- Connector：实现某一协议的连接器，用来处理客户端发送来的协议，如默认的实现协议有 HTTP、HTTPS、AJP。
  主要作用有：
  - 根据不同的协议解析客户端的请求
  - 将解析完的请求转发给 Connector 关联的 Engine 容器处理

  1. Mapper 维护了 URL 到容器的映射关系。当请求到来时会根据 Mapper 中的映射信息决定将请求映射到哪一个 Host、Context、Wrapper。
  2. Http11NioProtocol 用于处理 HTTP/1.1 的请求
  3. NioEndpoint 是连接的端点，在请求处理流程中该类是核心类，会重点介绍。
  4. CoyoteAdapter 用于将请求从 Connctor 交给 Container 处理，**使 Connctor 和 Container 解耦。**
- MapperListener 实现了 LifecycleListener 和 ContainerListener 接口用于监听容器事件和生命周期事件。该监听器实例监听所有的容器，包括 StandardEngine、StandardHost、StandardContext、StandardWrapper，当容器有变动时，注册容器到 Mapper。
- Engine：代表的是 Servlet 引擎，接收来自不同 Connector 请求，处理后将结果返回给 Connector。Engine 是一个逻辑容器，包含一个或多个 Host。默认实现是 StandardEngine，主要有以下模块：
  - Cluster：实现 Tomcat 管理
  - Realm：实现用户权限管理模块
  - Pipeline 和 Valve（阀门）：处理 Pipeline 上的各个 Valve，是一种责任链模式。只是简单的将 Connector 传过来的变量传给 Host 容器

- Host：虚拟主机，即域名或网络名，用于部署该虚拟主机上的应用程序。通常包含多个 Context (Context 在 Tomcat 中代表应用程序)。Context 在 Tomcat 中的标准实现是 StandardContext。
- Context：部署的**具体 Web 应用**，每个请求都在是相应的上下文里处理，如一个 war 包。默认实现是 StandardContext，通常包含多个 Wrapper 主要有以下模块：
  - Realm：实现用户权限管理模块
  - Pipeline 和 Valve：处理 Pipeline 上的各个 Valve，是一种责任链模式
  - Manager: 它主要是应用的 session 管理模块
  - Resources: 它是每个 web app 对应的部署结构的封装
  - Loader：它是对每个 web app 的自有的 classloader 的封装
  - Mapper：它封装了请求资源 URI 与每个相对应的处理 wrapper 容器的映射关系
- Wrapper：对应定义的 Servlet，一一对应。默认实现是 StandardWrapper，主要有以下模块：

    - Pipeline 和 Valve：处理 Pipeline 上的各个 Valve，是一种责任链模式
    - Servlet 和 Servlet Stack：保存 Wrapper 包装的 Servlet
- StandardPipeline 组件代表一个流水线，与 Valve（阀）结合，用于处理请求。StandardPipeline 中含有多个 Valve， 当需要处理请求时，会逐一调用 Valve 的 invoke 方法对 Request 和 Response 进行处理。特别的，其中有一个特殊的 Valve 叫 basicValve,每一个标准容器都有一个指定的 BasicValve，他们做的是最核心的工作。
    - StandardEngine 的是 StandardEngineValve，他用来将 Request 映射到指定的 Host;
    - StandardHost 的是 StandardHostValve，他用来将 Request 映射到指定的 Context;
    - StandardContext 的是 StandardContextValve，它用来将 Request 映射到指定的 Wrapper；
    - StandardWrapper 的是 StandardWrapperValve，他用来加载 Rquest 所指定的 Servlet,并调用 Servlet 的 Service 方法。

由上可知，Catalina 中有两个主要的模块：连接器（Connector）和容器（Container）、

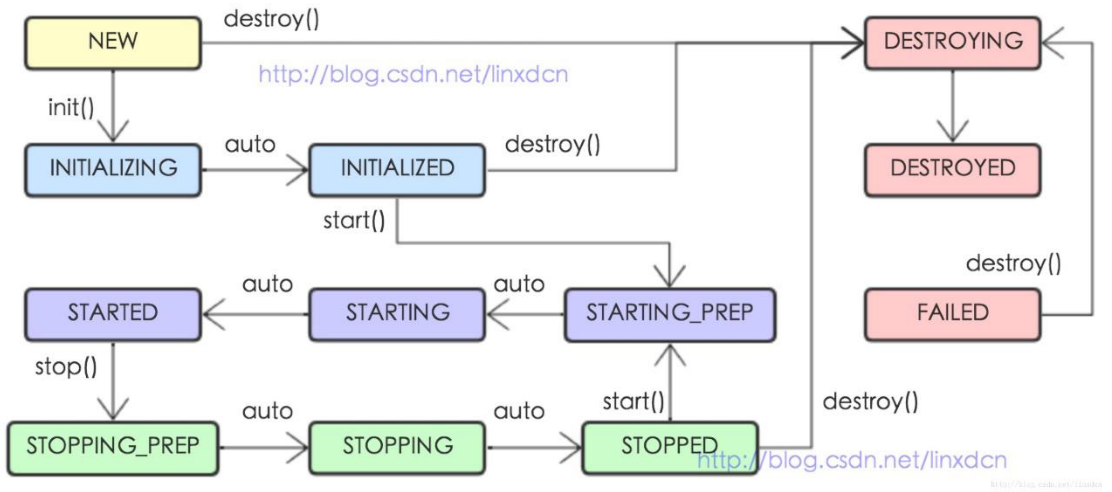
以 Tomcat 为例，它的主线流程大致可以分为 3 个：启动、部署、请求处理。入口点就是 Bootstrap 类和 接受请求的 Acceptor 类！

生命周期

在 Tomcat 启动时，会读取 server.xml 文件创建 Server, Service, Connector, Engine, Host, Context, Wrapper 等组件。

Lifestyle

Tomcat 中的所有组件都继承了 Lifecycle 接口，Lifecycle 接口定义了一整套生命周期管理的函数，从组件的新建、初始化完成、启动、停止、失败到销毁，都遵守同样的规则，Lifecycle 组件的状态转换图如下。



正常的调用顺序是 init()->start()->destroy()，父组件的 init()和 start()会触发子组件的 init()和 start()，所以 Tomcat 中只需调用 Server 组件的 init()和 start()即可。
每个实现组件都继承自 LifecycleBase，LifecycleBase 实现了 Lifecycle 接口，当容器状态发生变化时，都会调用 fireLifecycleEvent 方法，生成 LifecycleEvent，并且交由此容器的事件监听器处理。

启动

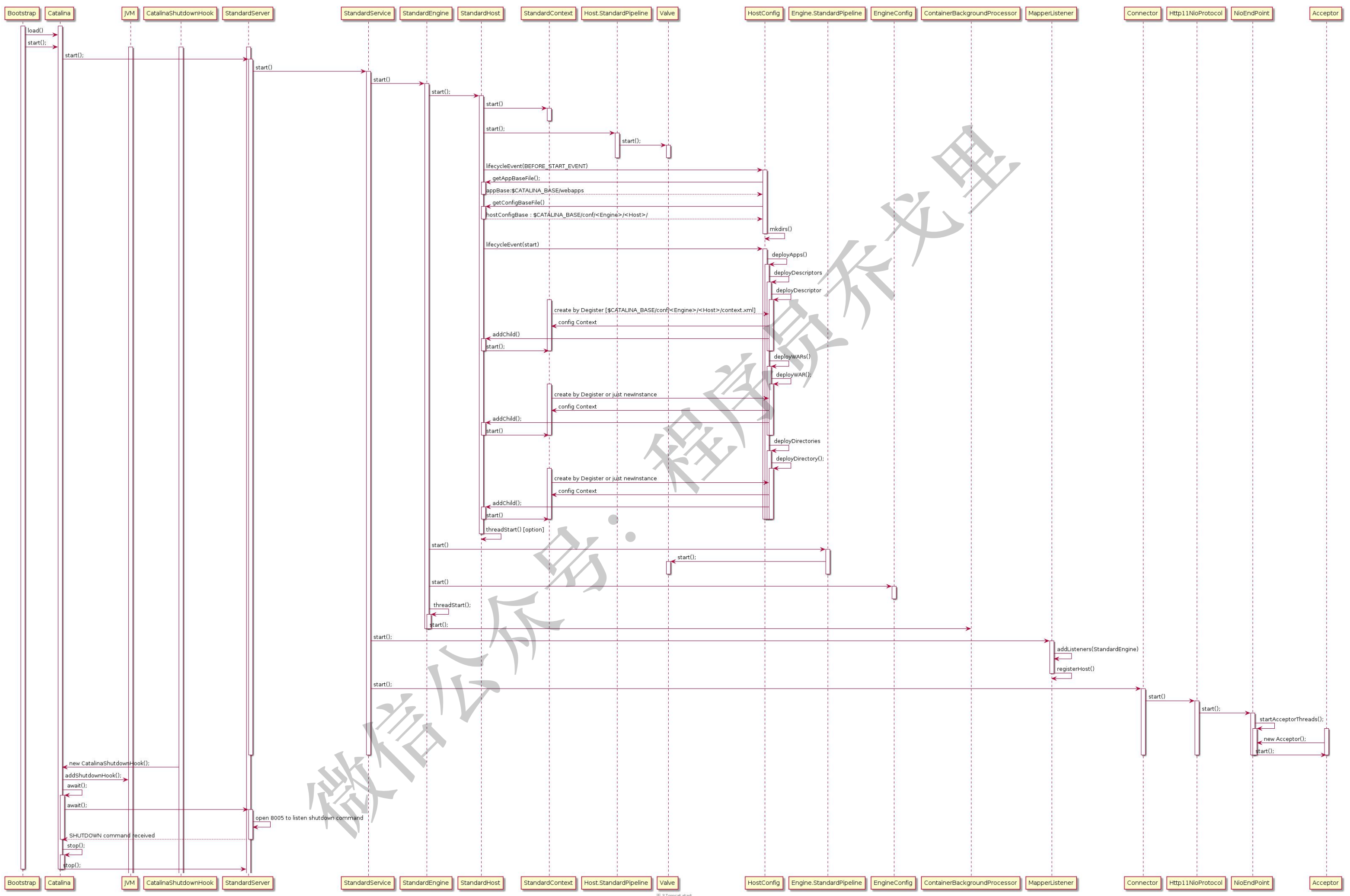


图 3 Tomcat start

tomcat/bin/startup.sh 脚本是启动了 org.apache.catalina.startup.Bootstra 类的 main 方法，并传入 start 参数。

主要步骤如下：

1. 新建 Bootstrap 对象 daemon，并调用其 init()方法
2. 初始化 Tomcat 的类加载器（init）
3. 用反射实例化 org.apache.catalina.startup.Catalina 对象 catalinaDaemon（init）
4. 调用 daemon 的 load 方法，实质上调用了 catalinaDaemon 的 load 方法（load）
5. 加载和解析 server.xml 配置文件（load）
6. 调用 daemon 的 start 方法，实质上调用了 catalinaDaemon 的 start 方法 （start）
7. 启动 Server 组件，Server 的启动会带动其他组件的启动，如 Service, Container, Connector（start）
8. 调用 catalinaDaemon 的 await 方法循环等待接收 Tomcat 的 shutdown 命令

BootStrap#main

```java
public static void main(String args[]) {

    if (daemon == null) {
        // Don't set daemon until init() has completed
        Bootstrap bootstrap = new Bootstrap();
        try {
            bootstrap.init();
        } catch (Throwable t) {
            handleThrowable(t);
            t.printStackTrace();
            return;
        }
        daemon = bootstrap;
    } else {
        // When running as a service the call to stop will be on a new
        // thread so make sure the correct class loader is used to prevent
        // a range of class not found exceptions.
        Thread.currentThread().setContextClassLoader(daemon.catalinaLoader);
    }

    try {
        String command = "start";
        if (args.length > 0) {
            command = args[args.length - 1];
        }

        if (command.equals("startd")) {
            args[args.length - 1] = "start";
            daemon.load(args);
            daemon.start();
        } else if (command.equals("stopd")) {
            args[args.length - 1] = "stop";
            daemon.stop();
        } else if (command.equals("start")) {
            // 设置 Catalina 的 await 属性为 true。在 Start 阶段尾部，若该属性为true，Tomcat 会在 main 线程中监听 SHUTDOWN 命令，默认端口是 8005.当收到该命令后执行 Catalina 的 stop() 方法关闭 Tomcat 服务器。
            daemon.setAwait(true);
            daemon.load(args);
            daemon.start();
        } else if (command.equals("stop")) {
            daemon.stopServer(args);
        } else if (command.equals("configtest")) {
            daemon.load(args);
            if (null==daemon.getServer()) {
                System.exit(1);
            }
            System.exit(0);
        } else {
            log.warn("Bootstrap: command \"" + command + "\" does not exist.");
        }
    } catch (Throwable t) {
        // Unwrap the Exception for clearer error reporting
        if (t instanceof InvocationTargetException &&
                t.getCause() != null) {
            t = t.getCause();
        }
        handleThrowable(t);
        t.printStackTrace();
        System.exit(1);
    }

}
```

1) BootStrap#init

必须使用反射去实例化 Catalina 对象，此时可以使用 Tomcat 自己的 Classloader。否则会使用 Java 的 Classloader 去加载 Catalina 对象。

```java
public void init() throws Exception {
    // 初始化 commonLoader、catalinaLoader 和 sharedLoader;
    initClassLoaders();
    // 将 catalinaLoader 设置为 Tomcat 主线程的线程上下文类加载器;
    Thread.currentThread().setContextClassLoader(catalinaLoader);

    SecurityClassLoad.securityClassLoad(catalinaLoader);
    // Load our startup class and call its process() method
    if (log.isDebugEnabled())
        log.debug("Loading startup class");
    Class<?> startupClass =
catalinaLoader.loadClass("org.apache.catalina.startup.Catalina");
    Object startupInstance = startupClass.getConstructor().newInstance();

    // Set the shared extensions class loader
    if (log.isDebugEnabled())
        log.debug("Setting startup class properties");
    String methodName = "setParentClassLoader";
    Class<?> paramTypes[] = new Class[1];
    paramTypes[0] = Class.forName("java.lang.ClassLoader");
    Object paramValues[] = new Object[1];
    paramValues[0] = sharedLoader;
    Method method =
        startupInstance.getClass().getMethod(methodName, paramTypes);
    method.invoke(startupInstance, paramValues);

    catalinaDaemon = startupInstance;

}
```

1.1) BootStrap#initClassLoaders()

commonLoader、catalinaLoader 和 sharedLoader 是在 Tomcat 容器初始化的的过程刚刚开始（即调用 Bootstrap 的 init 方法时）创建的。catalinaLoader 会被设置为 Tomcat 主线程的线程上下文类加载器，并且使用 catalinaLoader 加载 Tomcat 容器自身的 class。

```
private void initClassLoaders() {
    try {
        commonLoader = createClassLoader("common", null);
        if( commonLoader == null ) {
            // no config file, default to this loader - we might be in a 'single'
env.
            commonLoader=this.getClass().getClassLoader();
        }
        catalinaLoader = createClassLoader("server", commonLoader);
        sharedLoader = createClassLoader("shared", commonLoader);
    } catch (Throwable t) {
        handleThrowable(t);
        log.error("Class loader creation threw exception", t);
        System.exit(1);
    }
}
```

1.1.1) createClassLoader

createClassLoader 的处理步骤如下：

定位资源路径与资源类型；
使用 ClassLoaderFactory 创建类加载器 org.apache.catalina.loader.StandardClassLoader，这**个 StandardClassLoader 仅仅继承了 URLClassLoader 而没有其他更多改动。**
Tomcat 默认只会指定 commonLoader， catalinaLoader 和 sharedLoader 实际也是 commonLoader。(在 catalina.properties 配置文件中，我们可以看到 common 属性默认值为 {catalina.base}/lib/.jar,{catalina.home}/lib/.jar， 如下配置所示， 属性 catalina.home 默认为 Tomcat 的根目录。)

common.loader=${catalina.home}/lib,${catalina.home}/lib/*.jar

```java
private ClassLoader createClassLoader(String name, ClassLoader parent)
    throws Exception {

    String value = CatalinaProperties.getProperty(name + ".loader");
    if ((value == null) || (value.equals("")))
        return parent;

    value = replace(value);

    List<Repository> repositories = new ArrayList<>();

    String[] repositoryPaths = getPaths(value);

    for (String repository : repositoryPaths) {
        // Check for a JAR URL repository
        try {
            @SuppressWarnings("unused")
            URL url = new URL(repository);
            repositories.add(
                    new Repository(repository, RepositoryType.URL));
            continue;
        } catch (MalformedURLException e) {
            // Ignore
        }

        // Local repository
        if (repository.endsWith("*.jar")) {
            repository = repository.substring
```

```
               (0, repository.length() - "*.jar".length());
           repositories.add(
                   new Repository(repository, RepositoryType.GLOB));
       } else if (repository.endsWith(".jar")) {
           repositories.add(
                   new Repository(repository, RepositoryType.JAR));
       } else {
           repositories.add(
                   new Repository(repository, RepositoryType.DIR));
       }
   }

   return ClassLoaderFactory.createClassLoader(repositories, parent);
}
```

1.1.1.1) ClassLoaderFactory#createClassLoader

```
public static ClassLoader createClassLoader(List<Repository> repositories,
                                            final ClassLoader parent)
    throws Exception {

    if (log.isDebugEnabled())
        log.debug("Creating new class loader");

    // Construct the "class path" for this class loader
    Set<URL> set = new LinkedHashSet<>();

    if (repositories != null) {
        for (Repository repository : repositories)  {
            if (repository.getType() == RepositoryType.URL) {
                URL url = buildClassLoaderUrl(repository.getLocation());
                if (log.isDebugEnabled())
                    log.debug("  Including URL " + url);
                set.add(url);
            } else if (repository.getType() == RepositoryType.DIR) {
                File directory = new File(repository.getLocation());
                directory = directory.getCanonicalFile();
                if (!validateFile(directory, RepositoryType.DIR)) {
                    continue;
                }
                URL url = buildClassLoaderUrl(directory);
                if (log.isDebugEnabled())
                    log.debug("  Including directory " + url);
                set.add(url);
            } else if (repository.getType() == RepositoryType.JAR) {
                File file=new File(repository.getLocation());
                file = file.getCanonicalFile();
                if (!validateFile(file, RepositoryType.JAR)) {
                    continue;
                }
                URL url = buildClassLoaderUrl(file);
                if (log.isDebugEnabled())
                    log.debug("  Including jar file " + url);
                set.add(url);
            } else if (repository.getType() == RepositoryType.GLOB) {
                File directory=new File(repository.getLocation());
                directory = directory.getCanonicalFile();
                if (!validateFile(directory, RepositoryType.GLOB)) {
                    continue;
                }
                if (log.isDebugEnabled())
                    log.debug("  Including directory glob "
                        + directory.getAbsolutePath());
                String filenames[] = directory.list();
                if (filenames == null) {
                    continue;
                }
                for (int j = 0; j < filenames.length; j++) {
                    String filename = filenames[j].toLowerCase(Locale.ENGLISH);
                    if (!filename.endsWith(".jar"))
                        continue;
                    File file = new File(directory, filenames[j]);
                    file = file.getCanonicalFile();
                    if (!validateFile(file, RepositoryType.JAR)) {
                        continue;
                    }
                    if (log.isDebugEnabled())
                        log.debug("    Including glob jar file "
                            + file.getAbsolutePath());
                    URL url = buildClassLoaderUrl(file);
                    set.add(url);
                }
            }
        }
    }

    // Construct the class loader itself
    final URL[] array = set.toArray(new URL[set.size()]);
```

```
    if (log.isDebugEnabled())
        for (int i = 0; i < array.length; i++) {
            log.debug("  location " + i + " is " + array[i]);
        }

    return AccessController.doPrivileged(
            new PrivilegedAction<URLClassLoader>() {
                @Override
                public URLClassLoader run() {
                    if (parent == null)
                        return new URLClassLoader(array);
                    else
                        return new URLClassLoader(array, parent);
                }
            });
}
```

2) BootStrap#load

```
private void load(String[] arguments)
    throws Exception {

    // Call the load() method
    String methodName = "load";
    Object param[];
    Class<?> paramTypes[];
    if (arguments==null || arguments.length==0) {
        paramTypes = null;
        param = null;
    } else {
        paramTypes = new Class[1];
        paramTypes[0] = arguments.getClass();
        param = new Object[1];
        param[0] = arguments;
    }
    Method method =
        catalinaDaemon.getClass().getMethod(methodName, paramTypes);
    if (log.isDebugEnabled())
        log.debug("Calling startup class " + method);
    method.invoke(catalinaDaemon, param);

}
```

2.1) Catalina#load

```java
public void load() {

    if (loaded) {
        return;
    }
    loaded = true;

    long t1 = System.nanoTime();

    initDirs();

    // Before digester - it may be needed
    initNaming();

    // Create and execute our Digester
    Digester digester = createStartDigester();

    InputSource inputSource = null;
    InputStream inputStream = null;
    File file = null;
    try {
        try {
            file = configFile();
            inputStream = new FileInputStream(file);
            inputSource = new InputSource(file.toURI().toURL().toString());
        } catch (Exception e) {
            if (log.isDebugEnabled()) {
                log.debug(sm.getString("catalina.configFail", file), e);
            }
        }
        if (inputStream == null) {
            try {
                inputStream = getClass().getClassLoader()
                    .getResourceAsStream(getConfigFile());
                inputSource = new InputSource
                    (getClass().getClassLoader()
                     .getResource(getConfigFile()).toString());
            } catch (Exception e) {
                if (log.isDebugEnabled()) {
                    log.debug(sm.getString("catalina.configFail",
                            getConfigFile()), e);
                }
            }
        }

        // This should be included in catalina.jar
        // Alternative: don't bother with xml, just create it manually.
        if (inputStream == null) {
            try {
                inputStream = getClass().getClassLoader()
                        .getResourceAsStream("server-embed.xml");
                inputSource = new InputSource
                (getClass().getClassLoader()
                        .getResource("server-embed.xml").toString());
            } catch (Exception e) {
                if (log.isDebugEnabled()) {
                    log.debug(sm.getString("catalina.configFail",
                            "server-embed.xml"), e);
                }
            }
        }


        if (inputStream == null || inputSource == null) {
            if  (file == null) {
                log.warn(sm.getString("catalina.configFail",
                        getConfigFile() + "] or [server-embed.xml]"));
            } else {
                log.warn(sm.getString("catalina.configFail",
                        file.getAbsolutePath()));
                if (file.exists() && !file.canRead()) {
                    log.warn("Permissions incorrect, read permission is not 
allowed on the file.");
                }
            }
            return;
        }

        try {
            inputSource.setByteStream(inputStream);
            digester.push(this);
            digester.parse(inputSource);
        } catch (SAXParseException spe) {
            log.warn("Catalina.start using " + getConfigFile() + ": " +
                    spe.getMessage());
            return;
        } catch (Exception e) {
            log.warn("Catalina.start using " + getConfigFile() + ": " , e);
            return;
        }
    } finally {
        if (inputStream != null) {
            try {
                inputStream.close();
            } catch (IOException e) {
                // Ignore
            }
        }
    }

    getServer().setCatalina(this);
    getServer().setCatalinaHome(Bootstrap.getCatalinaHomeFile());
    getServer().setCatalinaBase(Bootstrap.getCatalinaBaseFile());

    // Stream redirection
    initStreams();

    // Start the new server
    try {
        getServer().init();
    } catch (LifecycleException e) {
        if
(Boolean.getBoolean("org.apache.catalina.startup.EXIT_ON_INIT_FAILURE")) {
            throw new java.lang.Error(e);
        } else {
            log.error("Catalina.start", e);
        }
    }

    long t2 = System.nanoTime();
    if(log.isInfoEnabled()) {
        log.info("Initialization processed in " + ((t2 - t1) / 1000000) + " ms");
    }
}
```

2.1.1) Digester#parse（配置文件解析，创建子容器）

```java
public Object parse(InputSource input) throws IOException, SAXException {
    configure();
    getXMLReader().parse(input);
    return root;
}
```
org.apache.commons.digester
该包提供了基于规则的，可任意处理 XML 文档的类

org.apache.commons.digester.Digester 是 Digester 类库的主类，该类可用于解析 XML 文档。
解析过程分为两步:
定义好模式(定义要匹配的标签)
将模式与规则(定义匹配到标签后的行为的对象)相关联

解析过程中会调用 startElement 方法，会按照既定的一些规则，在读取的同时去创建对象。
比如：
```
digester.addRule("Server/Service/Connector",
                 new ConnectorCreateRule());
```

2.1.1.1) ConnectorCreateRule#begin

```
public void begin(String namespace, String name, Attributes attributes)
        throws Exception {
    Service svc = (Service)digester.peek();
    Executor ex = null;
    if ( attributes.getValue("executor")!=null ) {
        ex = svc.getExecutor(attributes.getValue("executor"));
    }
    Connector con = new Connector(attributes.getValue("protocol"));
    if (ex != null) {
        setExecutor(con, ex);
    }
    String sslImplementationName =
attributes.getValue("sslImplementationName");
    if (sslImplementationName != null) {
        setSSLImplementationName(con, sslImplementationName);
    }
    digester.push(con);
}
```

2.1.1.1.1) Connector#constructor（从 Connector 开始的初始化）

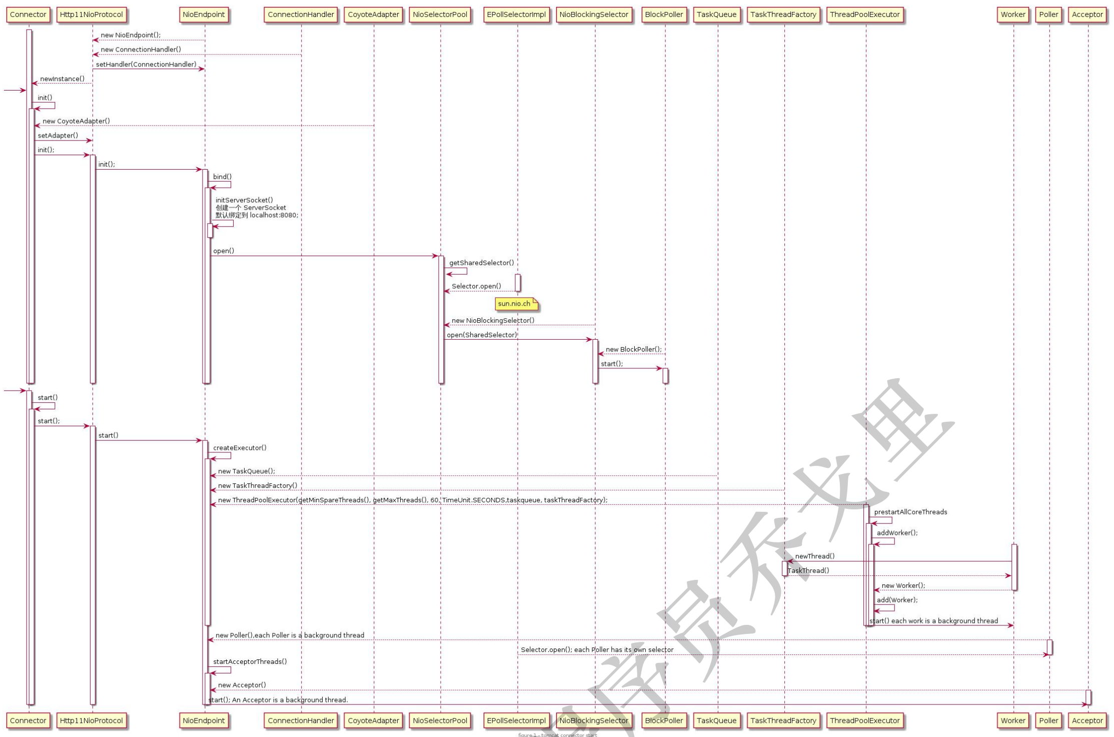


figure 1 - tomcat connector start

```
public Connector(String protocol) {
    boolean aprConnector = AprLifecycleListener.isAprAvailable() &&
            AprLifecycleListener.getUseAprConnector();

    if ("HTTP/1.1".equals(protocol) || protocol == null) {
        if (aprConnector) {
            protocolHandlerClassName =
"org.apache.coyote.http11.Http11AprProtocol";
        } else {
            protocolHandlerClassName =
"org.apache.coyote.http11.Http11NioProtocol";
        }
    } else if ("AJP/1.3".equals(protocol)) {
        if (aprConnector) {
            protocolHandlerClassName = "org.apache.coyote.ajp.AjpAprProtocol";
        } else {
            protocolHandlerClassName = "org.apache.coyote.ajp.AjpNioProtocol";
        }
    } else {
        protocolHandlerClassName = protocol;
    }

    // Instantiate protocol handler
    ProtocolHandler p = null;
    try {
        // 反射创建 Http11NioProtocol
        Class<?> clazz = Class.forName(protocolHandlerClassName);
        p = (ProtocolHandler) clazz.getConstructor().newInstance();
    } catch (Exception e) {
        log.error(sm.getString(
                "coyoteConnector.protocolHandlerInstantiationFailed"), e);
    } finally {
        this.protocolHandler = p;
    }

    // Default for Connector depends on this system property

setThrowOnFailure(Boolean.getBoolean("org.apache.catalina.startup.EXIT_ON_I
NIT_FAILURE"));
}
```

2.1.1.1.1.1) Http11NioProtocol#constructor

```
public Http11NioProtocol() {
    super(new NioEndpoint());
}
```

```
public AbstractHttp11JsseProtocol(AbstractJsseEndpoint<S,?> endpoint) {
    super(endpoint);
}
```

```
public AbstractHttp11Protocol(AbstractEndpoint<S,?> endpoint) {
    super(endpoint);
    setConnectionTimeout(Constants.DEFAULT_CONNECTION_TIMEOUT);
    ConnectionHandler<S> cHandler = new ConnectionHandler<>(this);
    setHandler(cHandler);
    getEndpoint().setHandler(cHandler);
}
```

```
public AbstractProtocol(AbstractEndpoint<S,?> endpoint) {
    this.endpoint = endpoint;
    setConnectionLinger(Constants.DEFAULT_CONNECTION_LINGER);
    setTcpNoDelay(Constants.DEFAULT_TCP_NO_DELAY);
}
```

2.1.1.1.1.1.1) ConnectionHandler#constructor

```java
public ConnectionHandler(AbstractProtocol<S> proto) {
    this.proto = proto;
}
```

2.1.2) StandardServer#init

模板方法模式，调用的是自己重写的 initInternal。

```java
protected void initInternal() throws LifecycleException {

    super.initInternal();

    // Register global String cache
    // Note although the cache is global, if there are multiple Servers
    // present in the JVM (may happen when embedding) then the same cache
    // will be registered under multiple names
    onameStringCache = register(new StringCache(), "type=StringCache");

    // Register the MBeanFactory
    MBeanFactory factory = new MBeanFactory();
    factory.setContainer(this);
    onameMBeanFactory = register(factory, "type=MBeanFactory");

    // Register the naming resources
    globalNamingResources.init();

    // Populate the extension validator with JARs from common and shared
    // class loaders
    if (getCatalina() != null) {
        ClassLoader cl = getCatalina().getParentClassLoader();
        // Walk the class loader hierarchy. Stop at the system class loader.
        // This will add the shared (if present) and common class loaders
        while (cl != null && cl != ClassLoader.getSystemClassLoader()) {
            if (cl instanceof URLClassLoader) {
                URL[] urls = ((URLClassLoader) cl).getURLs();
                for (URL url : urls) {
                    if (url.getProtocol().equals("file")) {
                        try {
                            File f = new File (url.toURI());
                            if (f.isFile() &&
                                    f.getName().endsWith(".jar")) {
                                ExtensionValidator.addSystemResource(f);
                            }
                        } catch (URISyntaxException e) {
                            // Ignore
                        } catch (IOException e) {
                            // Ignore
                        }
                    }
                }
            }
            cl = cl.getParent();
        }
    }
    // Initialize our defined Services
    for (int i = 0; i < services.length; i++) {
        services[i].init();
    }
}
```

初始化 StandardService

2.1.2.1) StandardService#init

```
protected void initInternal() throws LifecycleException {

    super.initInternal();

    if (engine != null) {
        engine.init();
    }

    // Initialize any Executors
    for (Executor executor : findExecutors()) {
        if (executor instanceof JmxEnabled) {
            ((JmxEnabled) executor).setDomain(getDomain());
        }
        executor.init();
    }

    // Initialize mapper listener
    mapperListener.init();
    // Initialize our defined Connectors
    synchronized (connectorsLock) {
        for (Connector connector : connectors) {
            connector.init();
        }
    }
}
```

2.1.2.1.1) Connector#init

```
protected void initInternal() throws LifecycleException {

    super.initInternal();

    if (protocolHandler == null) {
        throw new LifecycleException(

sm.getString("coyoteConnector.protocolHandlerInstantiationFailed"));
    }

    // Initialize adapter
    adapter = new CoyoteAdapter(this);
    // protocolHandler 即 Http11NioProtocol
    protocolHandler.setAdapter(adapter);

    // Make sure parseBodyMethodsSet has a default
    if (null == parseBodyMethodsSet) {
        setParseBodyMethods(getParseBodyMethods());
    }

    if (protocolHandler.isAprRequired()
&& !AprLifecycleListener.isAprAvailable()) {
        throw new
LifecycleException(sm.getString("coyoteConnector.protocolHandlerNoApr",
                getProtocolHandlerClassName()));
    }
    if (AprLifecycleListener.isAprAvailable() &&
AprLifecycleListener.getUseOpenSSL() &&
            protocolHandler instanceof AbstractHttp11JsseProtocol) {
        AbstractHttp11JsseProtocol<?> jsseProtocolHandler =
                (AbstractHttp11JsseProtocol<?>) protocolHandler;
        if (jsseProtocolHandler.isSSLEnabled() &&
                jsseProtocolHandler.getSslImplementationName() == null) {
```

```
           // OpenSSL is compatible with the JSSE configuration, so use it if
APR is available

jsseProtocolHandler.setSslImplementationName(OpenSSLImplementation.class.ge
tName());
       }
    }

    try {
        protocolHandler.init();
    } catch (Exception e) {
        throw new LifecycleException(

sm.getString("coyoteConnector.protocolHandlerInitializationFailed"), e);
    }
}
```

2.1.2.1.1.1) AbstractHttp11Protocol#init

```
public void init() throws Exception {
    // Upgrade protocols have to be configured first since the endpoint
    // init (triggered via super.init() below) uses this list to configure
    // the list of ALPN protocols to advertise
    for (UpgradeProtocol upgradeProtocol : upgradeProtocols) {
        configureUpgradeProtocol(upgradeProtocol);
    }

    super.init();
}
```

AbstractProtocol#init

```
public void init() throws Exception {
    if (getLog().isInfoEnabled()) {
        getLog().info(sm.getString("abstractProtocolHandler.init",
getName()));
    }

    if (oname == null) {
        // Component not pre-registered so register it
        oname = createObjectName();
        if (oname != null) {
            Registry.getRegistry(null, null).registerComponent(this, oname,
null);
```

```
        }
    }

    if (this.domain != null) {
        rgOname = new ObjectName(domain + ":type=GlobalRequestProcessor,name="
+ getName());
        Registry.getRegistry(null, null).registerComponent(
                getHandler().getGlobal(), rgOname, null);
    }

    String endpointName = getName();
    endpoint.setName(endpointName.substring(1, endpointName.length()-1));
    endpoint.setDomain(domain);

    endpoint.init();
}
```

2.1.2.1.1.1.1) AbstractEndPoint#init

```
public final void init() throws Exception {
    if (bindOnInit) {
        bind();
        bindState = BindState.BOUND_ON_INIT;
    }
    if (this.domain != null) {
        // Register endpoint (as ThreadPool - historical name)
        oname = new ObjectName(domain + ":type=ThreadPool,name=\"" + getName()
+ "\"");
        Registry.getRegistry(null, null).registerComponent(this, oname, null);

        for (SSLHostConfig sslHostConfig : findSslHostConfigs()) {
            registerJmx(sslHostConfig);
        }
    }
}
```

2.1.2.1.1.1.1.1) NioEndpoint#init

```
public void bind() throws Exception {
    initServerSocket();

    // Initialize thread count defaults for acceptor, poller
    if (acceptorThreadCount == 0) {
        // FIXME: Doesn't seem to work that well with multiple accept threads
        acceptorThreadCount = 1;
    }
    if (pollerThreadCount <= 0) {
        //minimum one poller thread
        pollerThreadCount = 1;
    }
    setStopLatch(new CountDownLatch(pollerThreadCount));

    // Initialize SSL if needed
    initialiseSsl();

    selectorPool.open();
}
```

2.1.2.1.1.1.1.1) NioEndpoint#initServerSocket（创建阻塞的 ServerSocket）

```
protected void initServerSocket() throws Exception {
    serverSock = ServerSocketChannel.open();
    socketProperties.setProperties(serverSock.socket());
    InetSocketAddress addr = (getAddress()!=null?new
InetSocketAddress(getAddress(),getPort()):new InetSocketAddress(getPort()));
    serverSock.socket().bind(addr,getAcceptCount());
    serverSock.configureBlocking(true); //mimic APR behavior
}
```

打开一个 ServerSocket，默认绑定到 8080 端口，默认的连接等待队列长度是 100，当超过 100 个时会拒绝服务。我们可以通过配置 conf/server.xml 中 Connector 的 acceptCount 属性对其进行定制。

2.1.2.1.1.1.1.2) NioSelectorPool#open（辅助 selector）

```
protected static final boolean SHARED =

Boolean.parseBoolean(System.getProperty("org.apache.tomcat.util.net.NioSele
ctorShared", "true"));
```

```
public void open() throws IOException {
    enabled = true;
    getSharedSelector();
    if (SHARED) {
        blockingSelector = new NioBlockingSelector();
        blockingSelector.open(getSharedSelector());
    }
}
```

2.1.2.1.1.1.1.1.2.1) NioSelectorPool#getSharedSelector （开启 selector）

```
protected Selector getSharedSelector() throws IOException {
    if (SHARED && SHARED_SELECTOR == null) {
        synchronized ( NioSelectorPool.class ) {
            if ( SHARED_SELECTOR == null )  {
                SHARED_SELECTOR = Selector.open();
                log.info("Using a shared selector for servlet write/read");
            }
        }
    }
    return  SHARED_SELECTOR;
}
```

2.1.2.1.1.1.1.1.2.2) NioBlockingSelector#open （启动 blockPoller 线程）

```
public void open(Selector selector) {
    sharedSelector = selector;
    poller = new BlockPoller();
    poller.selector = sharedSelector;
    poller.setDaemon(true);
    poller.setName("NioBlockingSelector.BlockPoller-"+(++threadCounter));
    poller.start();
}
```

3) BootStrap#start

```
public void start()
    throws Exception {
    if( catalinaDaemon==null ) init();

    Method method = catalinaDaemon.getClass().getMethod("start", (Class
[] )null);
    method.invoke(catalinaDaemon, (Object [])null);

}
```

3.1) Catalina#start

```
public void start() {

    if (getServer() == null) {
        load();
    }

    if (getServer() == null) {
        log.fatal("Cannot start server. Server instance is not configured.");
        return;
    }

    long t1 = System.nanoTime();

    // Start the new server
    try {
        getServer().start();
    } catch (LifecycleException e) {
        log.fatal(sm.getString("catalina.serverStartFail"), e);
        try {
            getServer().destroy();
        } catch (LifecycleException e1) {
            log.debug("destroy() failed for failed Server ", e1);
        }
        return;
    }

    long t2 = System.nanoTime();
    if(log.isInfoEnabled()) {
        log.info("Server startup in " + ((t2 - t1) / 1000000) + " ms");
    }

    // Register shutdown hook
    if (useShutdownHook) {
        if (shutdownHook == null) {
            shutdownHook = new CatalinaShutdownHook();
        }
        Runtime.getRuntime().addShutdownHook(shutdownHook);

        // If JULI is being used, disable JULI's shutdown hook since
        // shutdown hooks run in parallel and log messages may be lost
        // if JULI's hook completes before the CatalinaShutdownHook()
        LogManager logManager = LogManager.getLogManager();
        if (logManager instanceof ClassLoaderLogManager) {
            ((ClassLoaderLogManager) logManager).setUseShutdownHook(
                    false);
        }
    }

    if (await) {
        await();
        stop();
    }
}
```

3.1.1) StandardServer#start

start 同样也是模板方法模式。

```
protected void startInternal() throws LifecycleException {

    fireLifecycleEvent(CONFIGURE_START_EVENT, null);
    setState(LifecycleState.STARTING);

    globalNamingResources.start();

    // Start our defined Services
    synchronized (servicesLock) {
        for (int i = 0; i < services.length; i++) {
            services[i].start();
        }
    }
}
```

3.1.1.1) StandardService#start

```java
protected void startInternal() throws LifecycleException {

    if(log.isInfoEnabled())
        log.info(sm.getString("standardService.start.name", this.name));
    setState(LifecycleState.STARTING);

    // Start our defined Container first
    if (engine != null) {
        synchronized (engine) {
            engine.start();
        }
    }

    synchronized (executors) {
        for (Executor executor: executors) {
            executor.start();
        }
    }

    mapperListener.start();

    // Start our defined Connectors second
    synchronized (connectorsLock) {
        for (Connector connector: connectors) {
            // If it has already failed, don't try and start it
            if (connector.getState() != LifecycleState.FAILED) {
                connector.start();
            }
        }
    }
}
```

3.1.1.1.1) StandardEngine#start

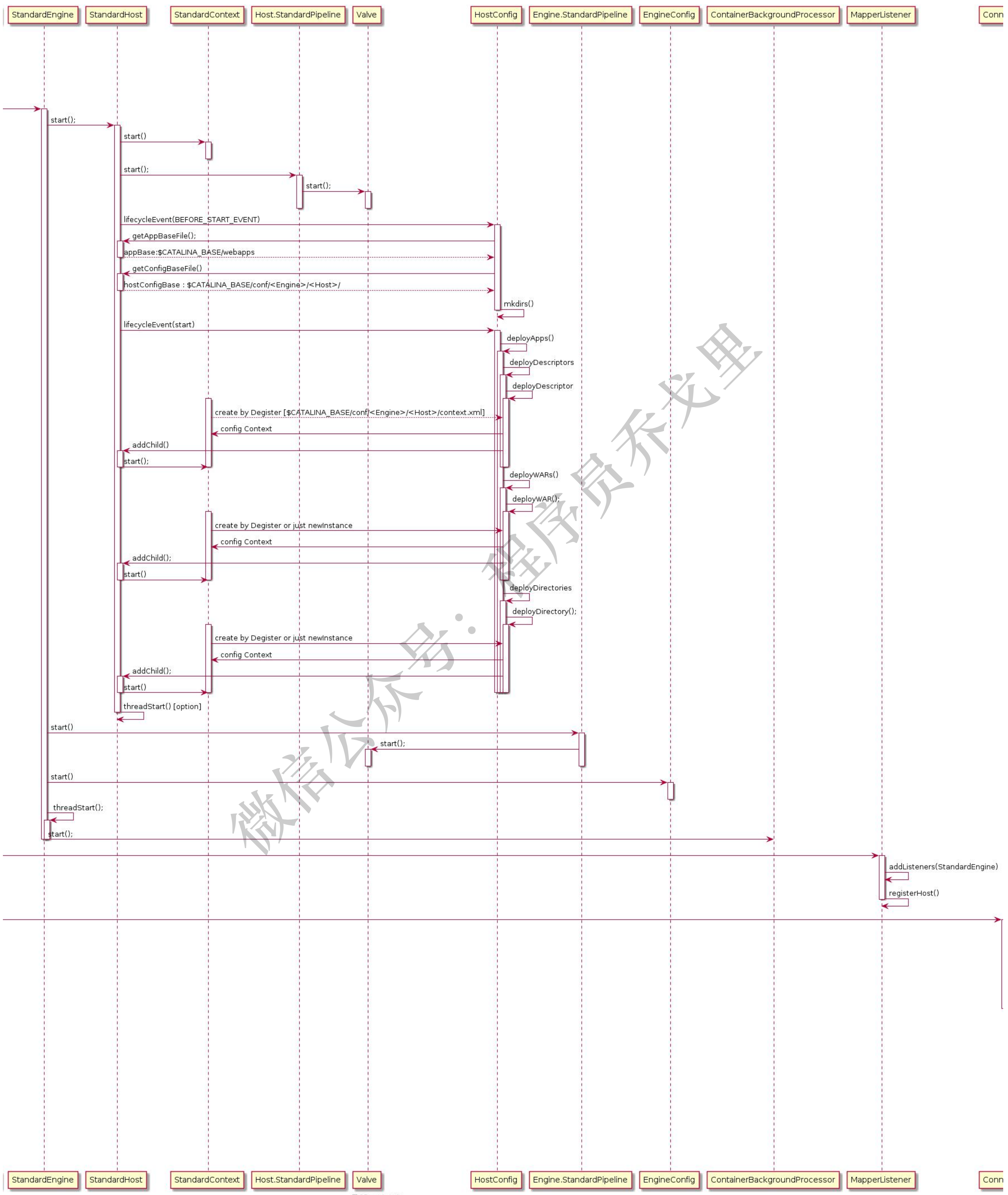


图 3 Tomcat start

```
protected synchronized void startInternal() throws LifecycleException {

    // Log our server identification information
    if(log.isInfoEnabled())
        log.info( "Starting Servlet Engine: " + ServerInfo.getServerInfo());

    // Standard container startup
    super.startInternal();
}
```

3.1.1.1.1.1) ContainerBase#startInternal

```
protected synchronized void startInternal() throws LifecycleException {

    // Start our subordinate components, if any
    logger = null;
    getLogger();
    Cluster cluster = getClusterInternal();
    if (cluster instanceof Lifecycle) {
        ((Lifecycle) cluster).start();
    }
    Realm realm = getRealmInternal();
    if (realm instanceof Lifecycle) {
        ((Lifecycle) realm).start();
    }

    // Start our child containers, if any
    Container children[] = findChildren();
    List<Future<Void>> results = new ArrayList<>();
    for (int i = 0; i < children.length; i++) {
        results.add(startStopExecutor.submit(new StartChild(children[i])));
    }

    boolean fail = false;
    for (Future<Void> result : results) {
        try {
            result.get();
        } catch (Exception e) {
            log.error(sm.getString("containerBase.threadedStartFailed"), e);
            fail = true;
        }
    }
    if (fail) {
        throw new LifecycleException(
                sm.getString("containerBase.threadedStartFailed"));
    }

    // Start the Valves in our pipeline (including the basic), if any
    if (pipeline instanceof Lifecycle)
        ((Lifecycle) pipeline).start();


    setState(LifecycleState.STARTING);

    // Start our thread
    threadStart();

}
```

3.1.1.1.1.1.1) StandardHost#start

```
protected synchronized void startInternal() throws LifecycleException {

    // Set error report valve
    String errorValve = getErrorReportValveClass();
    if ((errorValve != null) && (!errorValve.equals(""))) {
        try {
            boolean found = false;
            Valve[] valves = getPipeline().getValves();
            for (Valve valve : valves) {
                if (errorValve.equals(valve.getClass().getName())) {
                    found = true;
                    break;
                }
            }
            if(!found) {
                Valve valve =
                    (Valve)
Class.forName(errorValve).getConstructor().newInstance();
                getPipeline().addValve(valve);
            }
        } catch (Throwable t) {
            ExceptionUtils.handleThrowable(t);
            log.error(sm.getString(
                    "standardHost.invalidErrorReportValveClass",
                  errorValve), t);
        }
    }
    super.startInternal();
}
```

3.1.1.1.1.1.1.1) StandardContext#start（会初始化 loadOnStartup 的 servlet）

```
protected synchronized void startInternal() throws LifecycleException {

    if(log.isDebugEnabled())
        log.debug("Starting " + getBaseName());

    // Send j2ee.state.starting notification
    if (this.getObjectName() != null) {
        Notification notification = new Notification("j2ee.state.starting",
                this.getObjectName(), sequenceNumber.getAndIncrement());
        broadcaster.sendNotification(notification);
    }

    setConfigured(false);
    boolean ok = true;

    // Currently this is effectively a NO-OP but needs to be called to
    // ensure the NamingResources follows the correct lifecycle
    if (namingResources != null) {
        namingResources.start();
    }

    // Post work directory
    postWorkDirectory();

    // Add missing components as necessary
    if (getResources() == null) {   // (1) Required by Loader
        if (log.isDebugEnabled())
            log.debug("Configuring default Resources");

        try {
            setResources(new StandardRoot(this));
        } catch (IllegalArgumentException e) {
            log.error(sm.getString("standardContext.resourcesInit"), e);
            ok = false;
        }
    }
    if (ok) {
        resourcesStart();
    }
    // 初始化 WebappLoader
    if (getLoader() == null) {
        WebappLoader webappLoader = new WebappLoader(getParentClassLoader());
        webappLoader.setDelegate(getDelegate());
        setLoader(webappLoader);
    }

    // An explicit cookie processor hasn't been specified; use the default
    if (cookieProcessor == null) {
        cookieProcessor = new Rfc6265CookieProcessor();
    }

    // Initialize character set mapper
    getCharsetMapper();

    // Validate required extensions
    boolean dependencyCheck = true;
    try {
        dependencyCheck = ExtensionValidator.validateApplication
            (getResources(), this);
    } catch (IOException ioe) {
        log.error(sm.getString("standardContext.extensionValidationError"),
ioe);
        dependencyCheck = false;
    }

    if (!dependencyCheck) {
        // do not make application available if dependency check fails
        ok = false;
    }

    // Reading the "catalina.useNaming" environment variable
    String useNamingProperty = System.getProperty("catalina.useNaming");
    if ((useNamingProperty != null)
        && (useNamingProperty.equals("false"))) {
        useNaming = false;
    }

    if (ok && isUseNaming()) {
        if (getNamingContextListener() == null) {
            NamingContextListener ncl = new NamingContextListener();
            ncl.setName(getNamingContextName());
            ncl.setExceptionOnFailedWrite(getJndiExceptionOnFailedWrite());
            addLifecycleListener(ncl);
            setNamingContextListener(ncl);
        }
    }

    // Standard container startup
    if (log.isDebugEnabled())
        log.debug("Processing standard container startup");


    // Binding thread
    ClassLoader oldCCL = bindThread();

    try {
        if (ok) {
            // Start our subordinate components, if any
            Loader loader = getLoader();
            if (loader instanceof Lifecycle) {
                // 启动 WebappClassLoader
                ((Lifecycle) loader).start();
            }

            // since the loader just started, the webapp classloader is now
            // created.
            setClassLoaderProperty("clearReferencesRmiTargets",
                    getClearReferencesRmiTargets());
            setClassLoaderProperty("clearReferencesStopThreads",
                    getClearReferencesStopThreads());
            setClassLoaderProperty("clearReferencesStopTimerThreads",
                    getClearReferencesStopTimerThreads());

setClassLoaderProperty("clearReferencesHttpClientKeepAliveThread",
                    getClearReferencesHttpClientKeepAliveThread());

            // By calling unbindThread and bindThread in a row, we setup the
            // current Thread CCL to be the webapp classloader
            unbindThread(oldCCL);
            oldCCL = bindThread();

            // Initialize logger again. Other components might have used it
            // too early, so it should be reset.
            logger = null;
            getLogger();

            Realm realm = getRealmInternal();
            if(null != realm) {
                if (realm instanceof Lifecycle) {
                    ((Lifecycle) realm).start();
                }

                // Place the CredentialHandler into the ServletContext so
                // applications can have access to it. Wrap it in a "safe"
                // handler so application's can't modify it.
                CredentialHandler safeHandler = new CredentialHandler() {
                    @Override
                    public boolean matches(String inputCredentials, String
storedCredentials) {
                        return
getRealmInternal().getCredentialHandler().matches(inputCredentials,
storedCredentials);
                    }

                    @Override
                    public String mutate(String inputCredentials) {
                        return
getRealmInternal().getCredentialHandler().mutate(inputCredentials);
                    }
                };
                context.setAttribute(Globals.CREDENTIAL_HANDLER, safeHandler);
            }

            // Notify our interested LifecycleListeners
            fireLifecycleEvent(Lifecycle.CONFIGURE_START_EVENT, null);

            // Start our child containers, if not already started
            for (Container child : findChildren()) {
                if (!child.getState().isAvailable()) {
                    child.start();
                }
            }

            // Start the Valves in our pipeline (including the basic),
            // if any
            if (pipeline instanceof Lifecycle) {
                ((Lifecycle) pipeline).start();
            }
            // Acquire clustered manager
            Manager contextManager = null;
            Manager manager = getManager();
            if (manager == null) {
                if (log.isDebugEnabled()) {

log.debug(sm.getString("standardContext.cluster.noManager",
                            Boolean.valueOf((getCluster() != null)),
                            Boolean.valueOf(distributable)));
                }
                if ( (getCluster() != null) && distributable) {
                    try {
                        contextManager = getCluster().createManager(getName());
                    } catch (Exception ex) {
                        log.error("standardContext.clusterFail", ex);
                        ok = false;
                    }
                } else {
                    contextManager = new StandardManager();
                }
            }

            // Configure default manager if none was specified
            if (contextManager != null) {
                if (log.isDebugEnabled()) {
                    log.debug(sm.getString("standardContext.manager",
                            contextManager.getClass().getName()));
                }
                setManager(contextManager);
            }

            if (manager!=null && (getCluster() != null) && distributable) {
                //let the cluster know that there is a context that is
distributable
                //and that it has its own manager
                getCluster().registerManager(manager);
            }
        }

        if (!getConfigured()) {
            log.error(sm.getString("standardContext.configurationFail"));
            ok = false;
        }
        // We put the resources into the servlet context
        if (ok)
            getServletContext().setAttribute
                (Globals.RESOURCES_ATTR, getResources());

        if (ok ) {
            if (getInstanceManager() == null) {
                javax.naming.Context context = null;
                if (isUseNaming() && getNamingContextListener() != null) {
                    context = getNamingContextListener().getEnvContext();
                }
                Map<String, Map<String, String>> injectionMap =
buildInjectionMap(
                        getIgnoreAnnotations() ? new NamingResourcesImpl():
getNamingResources());
                setInstanceManager(new DefaultInstanceManager(context,
                        injectionMap, this, this.getClass().getClassLoader()));
            }
            getServletContext().setAttribute(
                    InstanceManager.class.getName(), getInstanceManager());
            InstanceManagerBindings.bind(getLoader().getClassLoader(),
getInstanceManager());
        }

        // Create context attributes that will be required
        if (ok) {
            getServletContext().setAttribute(
                    JarScanner.class.getName(), getJarScanner());
        }

        // Set up the context init params
        mergeParameters();

        // Call ServletContainerInitializers
        for (Map.Entry<ServletContainerInitializer, Set<Class<?>>> entry :
            initializers.entrySet()) {
            try {
                entry.getKey().onStartup(entry.getValue(),
                        getServletContext());
            } catch (ServletException e) {
                log.error(sm.getString("standardContext.sciFail"), e);
                ok = false;
                break;
```

```java
            }
        }

        // Configure and call application event listeners
        if (ok) {
            if (!listenerStart()) {
                log.error(sm.getString("standardContext.listenerFail"));
                ok = false;
            }
        }

        // Check constraints for uncovered HTTP methods
        // Needs to be after SCIs and listeners as they may programmatically
        // change constraints
        if (ok) {
            checkConstraintsForUncoveredMethods(findConstraints());
        }

        try {
            // Start manager
            Manager manager = getManager();
            if (manager instanceof Lifecycle) {
                ((Lifecycle) manager).start();
            }
        } catch(Exception e) {
            log.error(sm.getString("standardContext.managerFail"), e);
            ok = false;
        }

        // Configure and call application filters
        if (ok) {
            if (!filterStart()) {
                log.error(sm.getString("standardContext.filterFail"));
                ok = false;
            }
        }

        // Load and initialize all "load on startup" servlets
        if (ok) {
            if (!loadOnStartup(findChildren())){
                log.error(sm.getString("standardContext.servletFail"));
                ok = false;
            }
        }
        // Start ContainerBackgroundProcessor thread
        super.threadStart();
    } finally {
        // Unbinding thread
        unbindThread(oldCCL);
    }

    // Set available status depending upon startup success
    if (ok) {
        if (log.isDebugEnabled())
            log.debug("Starting completed");
    } else {
        log.error(sm.getString("standardContext.startFailed", getName()));
    }

    startTime=System.currentTimeMillis();

    // Send j2ee.state.running notification
    if (ok && (this.getObjectName() != null)) {
        Notification notification =
            new Notification("j2ee.state.running", this.getObjectName(),
                             sequenceNumber.getAndIncrement());
        broadcaster.sendNotification(notification);
    }

    // The WebResources implementation caches references to JAR files. On
    // some platforms these references may lock the JAR files. Since web
    // application start is likely to have read from lots of JARs, trigger
    // a clean-up now.
    getResources().gc();

    // Reinitializing if something went wrong
    if (!ok) {
        setState(LifecycleState.FAILED);
    } else {
        setState(LifecycleState.STARTING);
    }
}
```

3.1.1.1.1.1.1.1.1) StandardWrapper#start

```
protected synchronized void startInternal() throws LifecycleException {
    // Send j2ee.state.starting notification
    if (this.getObjectName() != null) {
        Notification notification = new Notification("j2ee.state.starting",
                                                        this.getObjectName(),
                                                        sequenceNumber++);
        broadcaster.sendNotification(notification);
    }

    // Start up this component
    super.startInternal();

    setAvailable(0L);

    // Send j2ee.state.running notification
    if (this.getObjectName() != null) {
        Notification notification =
            new Notification("j2ee.state.running", this.getObjectName(),
                             sequenceNumber++);
        broadcaster.sendNotification(notification);
    }

}
```

3.1.1.1.1.1.1.1.2) WebappLoader#constructor

```
    if (getLoader() == null) {
        WebappLoader webappLoader = new WebappLoader(getParentClassLoader());
        webappLoader.setDelegate(getDelegate());
        setLoader(webappLoader);
    }
```

```
/**
 * Construct a new WebappLoader with the specified class loader
 * to be defined as the parent of the ClassLoader we ultimately create.
 *
 * @param parent The parent class loader
 */
public WebappLoader(ClassLoader parent) {
    super();
    this.parentClassLoader = parent;
}
```

```
@Override
public void setLoader(Loader loader) {

    Lock writeLock = loaderLock.writeLock();
    writeLock.lock();
    Loader oldLoader = null;
    try {
        // Change components if necessary
        oldLoader = this.loader;
        if (oldLoader == loader)
            return;
        this.loader = loader;

        // Stop the old component if necessary
        if (getState().isAvailable() && (oldLoader != null) &&
            (oldLoader instanceof Lifecycle)) {
            try {
                ((Lifecycle) oldLoader).stop();
            } catch (LifecycleException e) {
                log.error("StandardContext.setLoader: stop: ", e);
            }
        }

        // Start the new component if necessary
        if (loader != null)
            loader.setContext(this);
        if (getState().isAvailable() && (loader != null) &&
            (loader instanceof Lifecycle)) {
            try {
                ((Lifecycle) loader).start();
            } catch (LifecycleException e) {
                log.error("StandardContext.setLoader: start: ", e);
            }
        }
    } finally {
        writeLock.unlock();
    }

    // Report this property change to interested listeners
    support.firePropertyChange("loader", oldLoader, loader);
}
```

3.1.1.1.1.1.1.1.3) WebappLoader#start

```
protected void startInternal() throws LifecycleException {

    if (log.isDebugEnabled())
        log.debug(sm.getString("webappLoader.starting"));

    if (context.getResources() == null) {
        log.info("No resources for " + context);
        setState(LifecycleState.STARTING);
        return;
    }

    // Construct a class loader based on our current repositories list
    try {

        classLoader = createClassLoader();
        classLoader.setResources(context.getResources());
        classLoader.setDelegate(this.delegate);

        // Configure our repositories
        setClassPath();

        setPermissions();

        classLoader.start();

        String contextName = context.getName();
        if (!contextName.startsWith("/")) {
            contextName = "/" + contextName;
        }
        ObjectName cloname = new ObjectName(context.getDomain() + ":type=" +
                classLoader.getClass().getSimpleName() + ",host=" +
                context.getParent().getName() + ",context=" + contextName);
        Registry.getRegistry(null, null)
            .registerComponent(classLoader, cloname, null);

    } catch (Throwable t) {
        t = ExceptionUtils.unwrapInvocationTargetException(t);
        ExceptionUtils.handleThrowable(t);
        log.error( "LifecycleException ", t );
        throw new LifecycleException("start: ", t);
    }
    setState(LifecycleState.STARTING);
}
```

3.1.1.1.1.1.1.1.3.1) WebappLoader#createClassLoader

```
private WebappClassLoaderBase createClassLoader()
    throws Exception {

    Class<?> clazz = Class.forName(loaderClass);
    WebappClassLoaderBase classLoader = null;
    // parentClassLoader 实际就是 sharedLoader，即
org.apache.catalina.loader.StandardClassLoader
    if (parentClassLoader == null) {
        parentClassLoader = context.getParentClassLoader();
    }
    Class<?>[] argTypes = { ClassLoader.class };
    Object[] args = { parentClassLoader };
    Constructor<?> constr = clazz.getConstructor(argTypes);
    classLoader = (WebappClassLoaderBase) constr.newInstance(args);

    return classLoader;
}
```

3.1.1.1.1.1.2) StandardPipeline#start

```
protected synchronized void startInternal() throws LifecycleException {

    // Start the Valves in our pipeline (including the basic), if any
    Valve current = first;
    if (current == null) {
        current = basic;
    }
    while (current != null) {
        if (current instanceof Lifecycle)
            ((Lifecycle) current).start();
        current = current.getNext();
    }

    setState(LifecycleState.STARTING);
}
```

3.1.1.1.1.1.2) ContainerBase#threadStart（启动后台线程，检查 session 过期）

```java
/**
 * Start the background thread that will periodically check for
 * session timeouts.
 */
protected void threadStart() {

    if (thread != null)
        return;
    if (backgroundProcessorDelay <= 0)
        return;

    threadDone = false;
    String threadName = "ContainerBackgroundProcessor[" + toString() + "]";
    thread = new Thread(new ContainerBackgroundProcessor(), threadName);
    thread.setDaemon(true);
    thread.start();

}
```

3.1.1.1.2) Connector#start

```java
protected void startInternal() throws LifecycleException {

    // Validate settings before starting
    if (getPort() < 0) {
        throw new LifecycleException(sm.getString(
                "coyoteConnector.invalidPort", Integer.valueOf(getPort())));
    }

    setState(LifecycleState.STARTING);

    try {
        protocolHandler.start();
    } catch (Exception e) {
        throw new LifecycleException(
                sm.getString("coyoteConnector.protocolHandlerStartFailed"),
e);
    }
}
```

3.1.1.1.2.1) AbstractProtocol#start

```
public void start() throws Exception {
    if (getLog().isInfoEnabled()) {
        getLog().info(sm.getString("abstractProtocolHandler.start",
getName()));
    }

    endpoint.start();

    // Start async timeout thread
    asyncTimeout = new AsyncTimeout();
    Thread timeoutThread = new Thread(asyncTimeout, getNameInternal() +
"-AsyncTimeout");
    int priority = endpoint.getThreadPriority();
    if (priority < Thread.MIN_PRIORITY || priority > Thread.MAX_PRIORITY) {
        priority = Thread.NORM_PRIORITY;
    }
    timeoutThread.setPriority(priority);
    timeoutThread.setDaemon(true);
    timeoutThread.start();
}
```

3.1.1.1.2.1.1) NioEndpoint#start

```
public void startInternal() throws Exception {

    if (!running) {
        running = true;
        paused = false;

        processorCache = new
SynchronizedStack<>(SynchronizedStack.DEFAULT_SIZE,
                socketProperties.getProcessorCache());
        eventCache = new SynchronizedStack<>(SynchronizedStack.DEFAULT_SIZE,
                        socketProperties.getEventCache());
        nioChannels = new SynchronizedStack<>(SynchronizedStack.DEFAULT_SIZE,
                socketProperties.getBufferPool());

        // Create worker collection
        if ( getExecutor() == null ) {
            createExecutor();
        }

        initializeConnectionLatch();

        // Start poller threads
        pollers = new Poller[getPollerThreadCount()];
        for (int i=0; i<pollers.length; i++) {
            pollers[i] = new Poller();
            Thread pollerThread = new Thread(pollers[i], getName() +
"-ClientPoller-"+i);
            pollerThread.setPriority(threadPriority);
            pollerThread.setDaemon(true);
            pollerThread.start();
        }

        startAcceptorThreads();
    }
}
```

3.1.1.1.2.1.1.1) NioEndpoint#createExecutor（创建 Worker 线程池）

用于创建 Worker 线程池。默认会启动 10 个 Worker 线程，Tomcat 处理请求过程中，Woker 最多不超过 200 个。我们可以通过配置 conf/server.xml 中 Connector 的 minSpareThreads 和 maxThreads 对这两个属性进行定制。

```
public void createExecutor() {
    internalExecutor = true;
    TaskQueue taskqueue = new TaskQueue();
    TaskThreadFactory tf = new TaskThreadFactory(getName() + "-exec-", daemon,
getThreadPriority());
    executor = new ThreadPoolExecutor(getMinSpareThreads(), getMaxThreads(),
60, TimeUnit.SECONDS,taskqueue, tf);
    taskqueue.setParent( (ThreadPoolExecutor) executor);
}
```

3.1.1.1.2.1.1.1.1) ThreadPoolExecutor#constructor（启动 Worker）

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long
keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory
threadFactory) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
threadFactory, new RejectHandler());
    prestartAllCoreThreads();
}
```

```
public int prestartAllCoreThreads() {
    int n = 0;
    while (addWorker(null, true))
```

```
        ++n;
    return n;
}
```

```
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;

        for (;;) {
            int wc = workerCountOf(c);
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get();  // Re-read ctl
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }

    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                // Recheck while holding lock.
                // Back out on ThreadFactory failure or if
                // shut down before lock acquired.
                int rs = runStateOf(ctl.get());

                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    if (t.isAlive()) // precheck that t is startable
                        throw new IllegalThreadStateException();
                    workers.add(w);
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
                }
            } finally {
                mainLock.unlock();
            }
            if (workerAdded) {
                t.start();
                workerStarted = true;
            }
        }
    } finally {
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

每个 Worker 线程启动是一个后台线程完成的。

3.1.1.1.1.1.1.2) Poller(Runnable)#run（启动 Poller）

以守护线程的方式运行。
用于检测已就绪的 Socket。 默认最多不超过 2 个, **Math.min(2,Runtime.getRuntime().availableProcessors());** **。 我 们 可 以 通 过 配 置 pollerThreadCount 来定制。**

3.1.1.1.1.1.1.3) NioEndpoint#startAcceptorThreads（启动 Acceptors）

Acceptors 以后台线程方式运行
用于接受新连接。默认是 1 个。我们可以通过配置 acceptorThreadCount 对其进行定制。

```
protected final void startAcceptorThreads() {
    int count = getAcceptorThreadCount();
    acceptors = new ArrayList<>(count);
    for (int i = 0; i < count; i++) {
        Acceptor<U> acceptor = new Acceptor<>(this);
        String threadName = getName() + "-Acceptor-" + i;
        acceptor.setThreadName(threadName);
        acceptors.add(acceptor);
        Thread t = new Thread(acceptor, threadName);
        t.setPriority(getAcceptorThreadPriority());
        t.setDaemon(getDaemon());
        t.start();
    }
}
```

3.1.2) StandardServer#await

```
public void await() {
    // Negative values - don't wait on port - tomcat is embedded or we just don't
like ports
    if( port == -2 ) {
        // undocumented yet - for embedding apps that are around, alive.
        return;
    }
    if( port==-1 ) {
        try {
            awaitThread = Thread.currentThread();
            while(!stopAwait) {
                try {
                    Thread.sleep( 10000 );
                } catch( InterruptedException ex ) {
                    // continue and check the flag
                }
            }
        } finally {
            awaitThread = null;
        }
        return;
    }

    // Set up a server socket to wait on
    try {
        awaitSocket = new ServerSocket(port, 1,
                InetAddress.getByName(address));
    } catch (IOException e) {
        log.error("StandardServer.await: create[" + address
                    + ":" + port
                    + "]: ", e);
        return;
    }

    try {
        awaitThread = Thread.currentThread();

        // Loop waiting for a connection and a valid command
        while (!stopAwait) {
            ServerSocket serverSocket = awaitSocket;
            if (serverSocket == null) {
                break;
            }

            // Wait for the next connection
            Socket socket = null;
            StringBuilder command = new StringBuilder();
            try {
                InputStream stream;
                long acceptStartTime = System.currentTimeMillis();
                try {
                    socket = serverSocket.accept();
                    socket.setSoTimeout(10 * 1000);  // Ten seconds
                    stream = socket.getInputStream();
                } catch (SocketTimeoutException ste) {
                    // This should never happen but bug 56684 suggests that
                    // it does.
                    log.warn(sm.getString("standardServer.accept.timeout",
                            Long.valueOf(System.currentTimeMillis() -
acceptStartTime)), ste);
                    continue;
                } catch (AccessControlException ace) {
                    log.warn("StandardServer.accept security exception: "
                            + ace.getMessage(), ace);
                    continue;
                } catch (IOException e) {
                    if (stopAwait) {
                        // Wait was aborted with socket.close()
                        break;
                    }
                    log.error("StandardServer.await: accept: ", e);
                    break;
                }
```

```java
            // Read a set of characters from the socket
            int expected = 1024; // Cut off to avoid DoS attack
            while (expected < shutdown.length()) {
                if (random == null)
                    random = new Random();
                expected += (random.nextInt() % 1024);
            }
            while (expected > 0) {
                int ch = -1;
                try {
                    ch = stream.read();
                } catch (IOException e) {
                    log.warn("StandardServer.await: read: ", e);
                    ch = -1;
                }
                // Control character or EOF (-1) terminates loop
                if (ch < 32 || ch == 127) {
                    break;
                }
                command.append((char) ch);
                expected--;
            }
        } finally {
            // Close the socket now that we are done with it
            try {
                if (socket != null) {
                    socket.close();
                }
            } catch (IOException e) {
                // Ignore
            }
        }

        // Match against our command string
        boolean match = command.toString().equals(shutdown);
        if (match) {
            log.info(sm.getString("standardServer.shutdownViaPort"));
            break;
        } else
            log.warn("StandardServer.await: Invalid command '"
                    + command.toString() + "' received");
    }
} finally {
        ServerSocket serverSocket = awaitSocket;
        awaitThread = null;
        awaitSocket = null;

        // Close the server socket and return
        if (serverSocket != null) {
            try {
                serverSocket.close();
            } catch (IOException e) {
                // Ignore
            }
        }
    }
}
```

停止

catalinaDaemon 调用 await 等待停止命令，我们一般是通过执行 tomcat/bin/shutdown.sh 来关闭 Tomcat，等价于执行 org.apache.catalina.startup.Bootstra 类的 main 方法，并传入 stop 参数。

逻辑：

1. 新建 Bootstrap 对象 daemon，并调用其 init()方法
2. 初始化 Tomcat 的类加载器
3. 用反射实例化 org.apache.catalina.startup.Catalina 对象 catalinaDaemon
4. **调用 daemon 的 stopServer 方法，实质上调用了 catalinaDaemon 的 stopServer 方法**
5. 解析 server.xml 文件，构造出 Server 容器
6. 获取 Server 的 socket 监听端口和地址，创建 Socket 对象连接启动 Tomcat 时创建的 ServerSocket，最后向 ServerSocket 发送 SHUTDOWN 命令
7. 运行中的 Server 调用 stop 方法停止

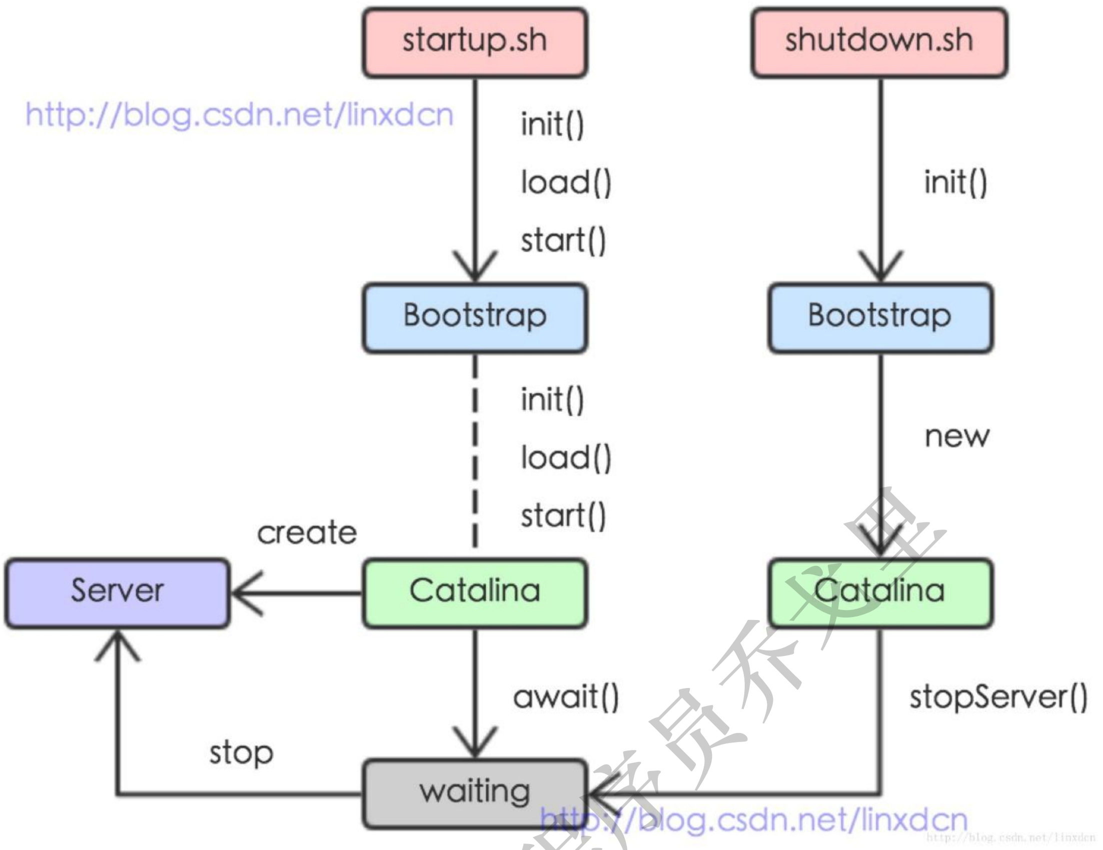
BootStrap#stopServer

```
public void stopServer(String[] arguments)
    throws Exception {

    Object param[];
    Class<?> paramTypes[];
    if (arguments==null || arguments.length==0) {
        paramTypes = null;
        param = null;
    } else {
        paramTypes = new Class[1];
        paramTypes[0] = arguments.getClass();
        param = new Object[1];
        param[0] = arguments;
    }
    Method method =
        catalinaDaemon.getClass().getMethod("stopServer", paramTypes);
    method.invoke(catalinaDaemon, param);

}
```

1) Catalina#stopServer

```
public void stopServer() {
    stopServer(null);
}
```

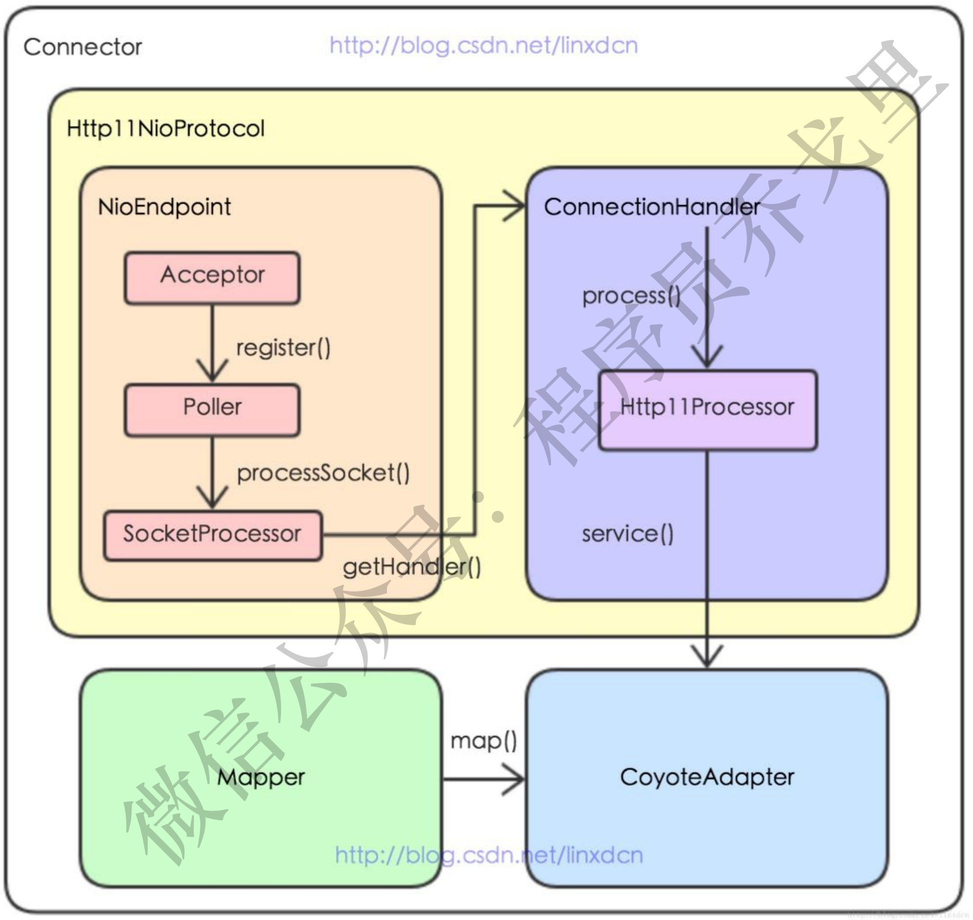
```
public void stopServer(String[] arguments) {

    if (arguments != null) {
        arguments(arguments);
    }

    Server s = getServer();
    if (s == null) {
        // Create and execute our Digester
        Digester digester = createStopDigester();
        File file = configFile();
        try (FileInputStream fis = new FileInputStream(file)) {
            InputSource is =
                new InputSource(file.toURI().toURL().toString());
            is.setByteStream(fis);
            digester.push(this);
            digester.parse(is);
        } catch (Exception e) {
            log.error("Catalina.stop: ", e);
            System.exit(1);
        }
    } else {
        // Server object already present. Must be running as a service
        try {
            s.stop();
        } catch (LifecycleException e) {
            log.error("Catalina.stop: ", e);
        }
        return;
    }

    // Stop the existing server
    s = getServer();
    if (s.getPort()>0) {
        try (Socket socket = new Socket(s.getAddress(), s.getPort());
                OutputStream stream = socket.getOutputStream()) {
            String shutdown = s.getShutdown();
            for (int i = 0; i < shutdown.length(); i++) {
                stream.write(shutdown.charAt(i));
            }
            stream.flush();
        } catch (ConnectException ce) {
            log.error(sm.getString("catalina.stopServer.connectException",
                                   s.getAddress(),
                                   String.valueOf(s.getPort())));
            log.error("Catalina.stop: ", ce);
            System.exit(1);
        } catch (IOException e) {
            log.error("Catalina.stop: ", e);
            System.exit(1);
        }
    } else {
        log.error(sm.getString("catalina.stopServer"));
        System.exit(1);
    }
}
```


startup.sh
shutdown.sh
http://blog.csdn.net/linxdcn
init()
load()
start()
init()
Bootstrap
Bootstrap
init()
load()
start()
new
create
Server
Catalina
Catalina
await()
stopServer()
stop
waiting
http://blog.csdn.net/linxdcn

请求处理

Connector

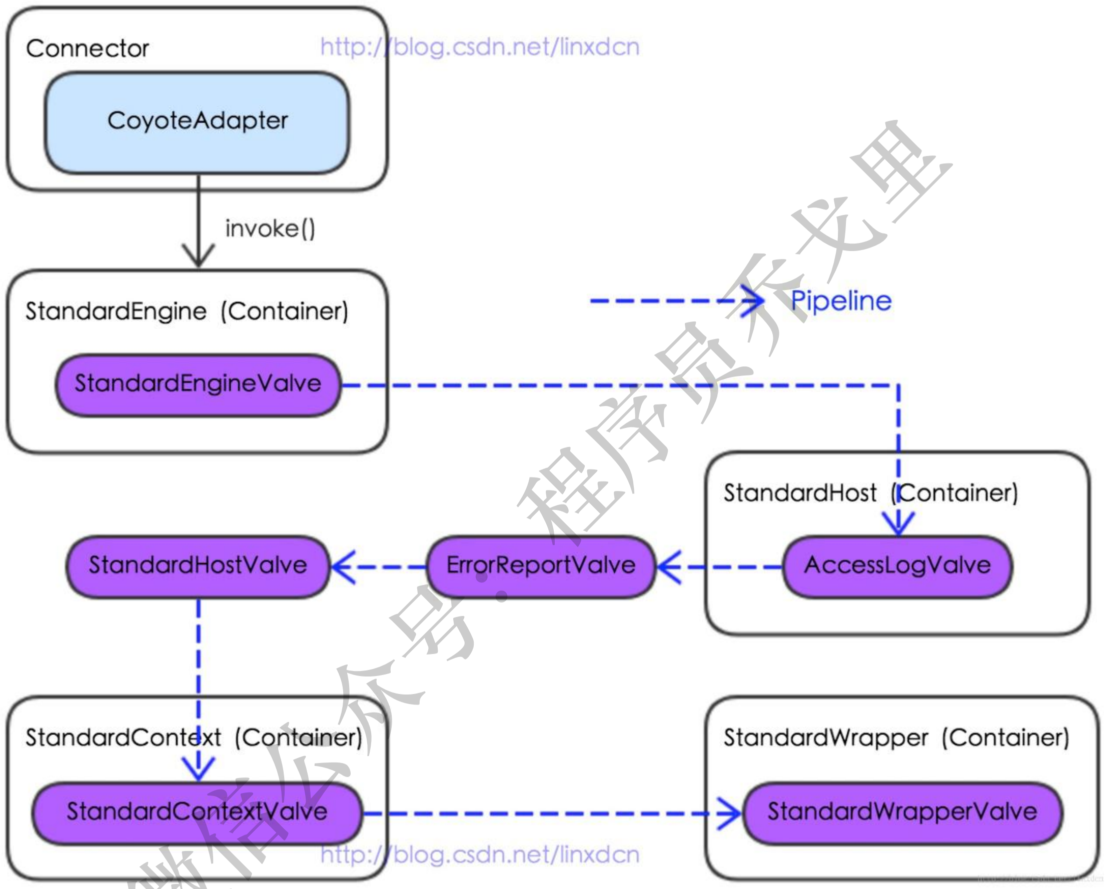
在 Tomcat9 中， Connector 支持的协议是 HTTP 和 AJP， 协议处理类分别对应 org.apache.coyote.http11.Http11NioProtocol 和 org.apache.coyote.http11.Http11AprProtocol（已经取消 BIO 模式)。
Connector 主要包含三个模块：Http11NioProtocol, Mapper, CoyoteAdapter， http 请求在 Connector 中的流程如下：



1. Acceptor 为监听线程， 调用 serverSocketAccept() 阻塞， 本质上调用 ServerSocketChannel.accept()
2. Acceptor 将接收到的 Socket 添加到 Poller 池中的一个 Poller
3. Poller 通过 worker 线程把 socket 包装成 SocketProcessor
4. SocketProcessor 调用 getHandler()获取对应的 ConnectionHandler
5. ConnectionHandler 把 socket 交由 Http11Processor 处理，解析 http 的 Header 和 Body
6. Http11Processor 调用 service()把包装好的 request 和 response 传给 CoyoteAdapter
7. CoyoteAdapter 会通过 Mapper 把请求对应的 session、servlet 等关联好，准备传给 Container

Container

有 4 个 Container，采用了责任链的设计模式。
Pipeline 就像是每个容器的逻辑总线，在 Pipeline 上按照配置的顺序，加载各个 Valve。通过 Pipeline 完成各个 Valve 之间的调用，各个 Valve 实现具体的应用逻辑。
每个请求在 pipeline 上流动，经过每个 Container（对应着一个或多个 Valve 阀门），各个 Container 按顺序处理请求，最终在 Wrapper 结束。



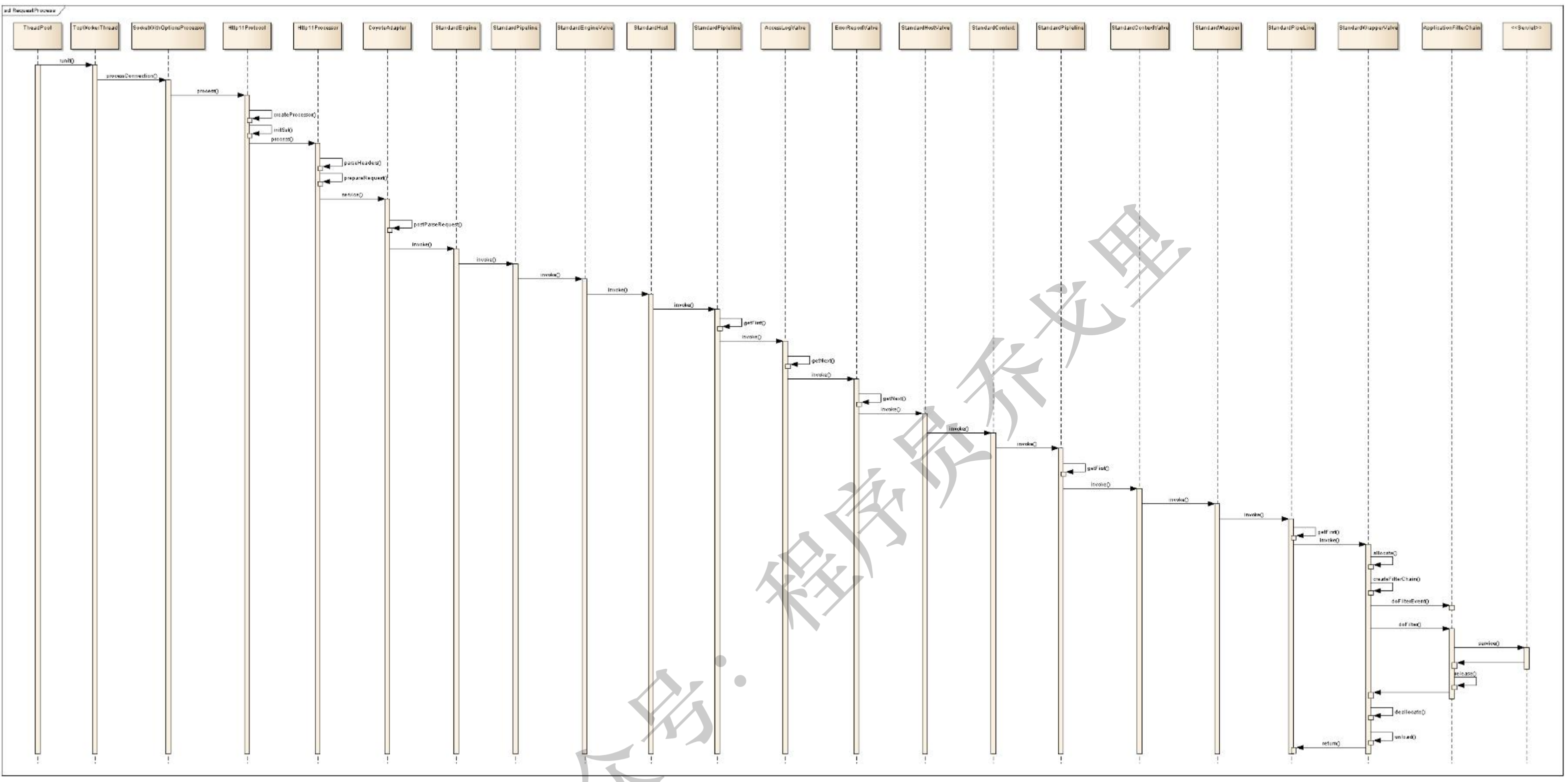
Connector 中的 CoyoteAdapter 会调用 invoke()把 request 和 response 传给 Container，Container 中依次调用各个 Valve，每个 Valve 的作用如下：

1. StandardEngineValve：StandardEngine 中的唯一阀门，主要用于从 request 中**选择其 host 映射的 Host 容器 StandardHost**
2. AccessLogValve：StandardHost 中的第一个阀门，主要用于管道执行结束之后**记录日志信息**
3. ErrorReportValve：StandardHost 中紧跟 AccessLogValve 的阀门，主要用于管道执行结束后，**从 request 对象中获取异常信息，并封装到 response 中**以便将问题展现给访问者
4. StandardHostValve： StandardHost 中最后的阀门，主要用于从 request 中选择**其 context 映射的 Context 容器 StandardContext** 以及访问 request 中的 Session 以更新会话的最后访问时间
5. StandardContextValve：StandardContext 中的唯一阀门，主要作用是**禁止任何对**

WEB-INF 或 META-INF 目录下资源的重定向访问，对应用程序热部署功能的实现，从 request 中获得 StandardWrapper

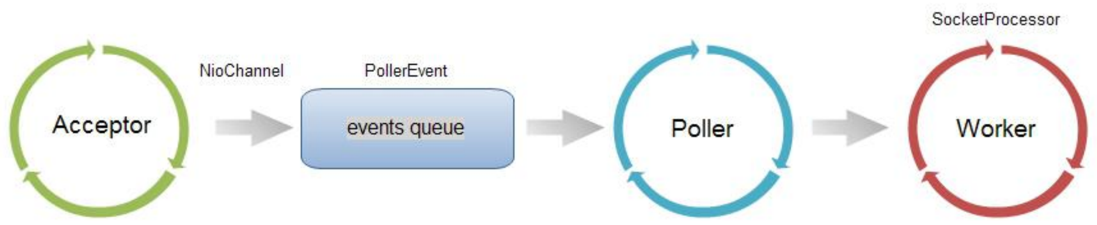
6. StandardWrapperValve：StandardWrapper 中的唯一阀门，主要作用包括**调用 StandardWrapper 的 loadServlet 方法生成 Servlet 实例和调用 ApplicationFilterFactory 生成 Filter 链**

最终将 Response 返回给 Connector 完成一次 http 的请求。



NioEndPoint 职责

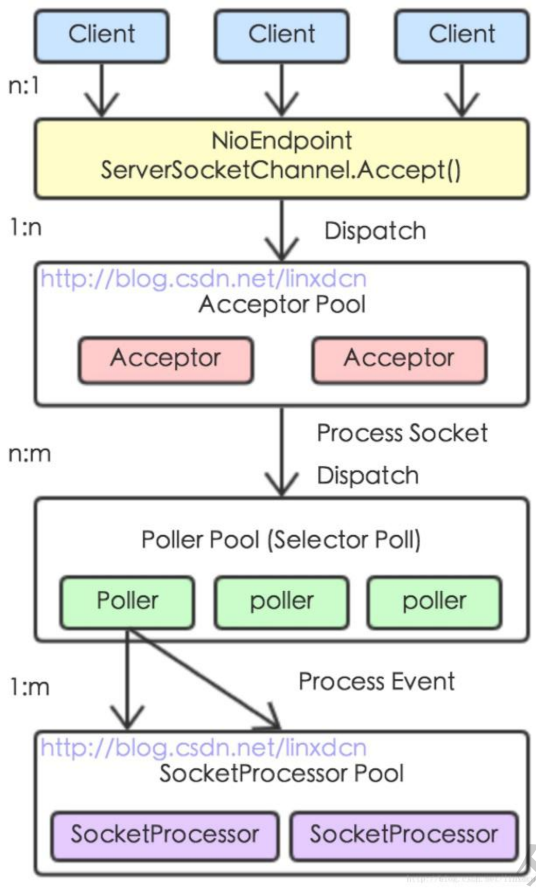
在 Tomcat 中 Endpoint 主要用来接收网络请求，处理则由 ConnectionHandler 来执行。



包含了三个组件：
Acceptor：后台线程，负责监听请求，将接收到的 Socket 请求放到 Poller 队列中
Poller：后台线程，当 Socket 就绪时，将 Poller 队列中的 Socket 交给 Worker 线程池处理
SocketProcessor（Worker）：处理 socket，本质上委托 ConnectionHandler 处理

Connector 启动以后会启动一组线程用于不同阶段的请求处理过程。
Acceptor 线程组。用于接受新连接，并将新连接封装一下，选择一个 Poller 将新连接添加到 Poller 的事件队列中。
Poller 线程组。用于监听 Socket 事件，当 Socket 可读或可写等等时，将 Socket 封装一下添加到 worker 线程池的任务队列中。
worker 线程组。用于对请求进行处理，包括分析请求报文并创建 Request 对象，调用容器的 pipeline 进行处理。

Acceptor、Poller、worker 所在的 ThreadPoolExecutor 都维护在 NioEndpoint 中。



这种模式类似于 Reactor 的主从多线程方式。

1) Acceptor#run（BIO,阻塞接收 Socket 连接,mainReactor）

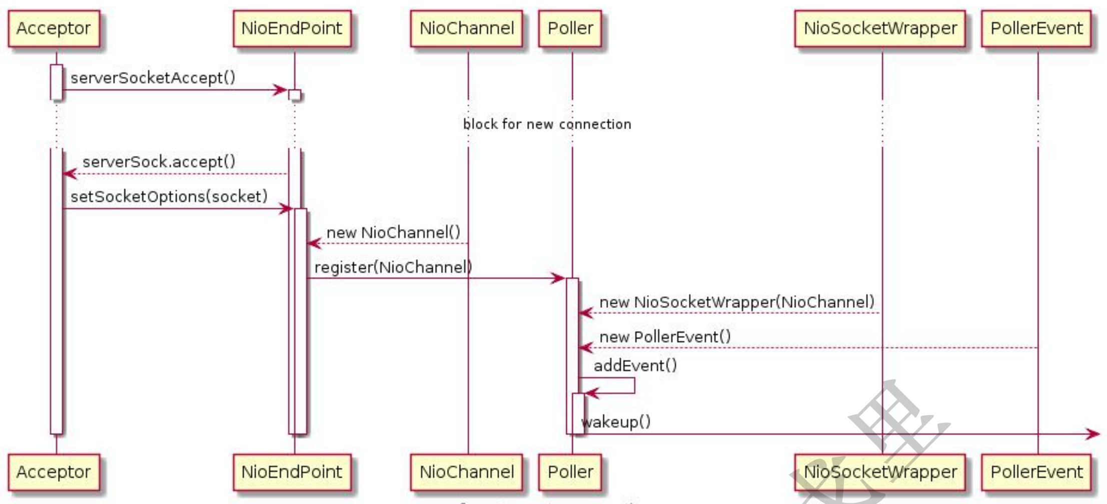


figure 2 - accept new connection

1. Acceptor 在启动后会阻塞在 ServerSocketChannel.accept(); 方法处，当有新连接到达时，该方法返回一个 SocketChannel。
2. 配置完 Socket 以后将 Socket 封装到 NioChannel 中，并注册到 Poller,值的一提的是，我们一开始就启动了多个 Poller 线程，注册的时候，连接是公平的分配到每个 Poller 的。NioEndpoint 维护了一个 Poller 数组，当一个连接分配给 pollers[index] 时，下一个连接就会分配给 pollers[(index+1)%pollers.length].
3. addEvent() 方法会将 Socket 添加到该 Poller 的 PollerEvent 队列中。到此 Acceptor 的任务就完成了。

持有 Endpoint

private final AbstractEndpoint<?,U> endpoint;

在启动后会阻塞在 ServerSocketChannel.accept(); 方法处，当有新连接到达时，该方法返回一个 SocketChannel。

```java
public void run() {

    int errorDelay = 0;

    // Loop until we receive a shutdown command
    while (endpoint.isRunning()) {

        // Loop if endpoint is paused
        while (endpoint.isPaused() && endpoint.isRunning()) {
            state = AcceptorState.PAUSED;
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                // Ignore
            }
        }
```

```
if (!endpoint.isRunning()) {
    break;
}
state = AcceptorState.RUNNING;

try {
    //if we have reached max connections, wait
    endpoint.countUpOrAwaitConnection();

    // Endpoint might have been paused while waiting for latch
    // If that is the case, don't accept new connections
    if (endpoint.isPaused()) {
        continue;
    }

    U socket = null;
    try {
        // Accept the next incoming connection from the server
        // socket
        socket = endpoint.serverSocketAccept();
    } catch (Exception ioe) {
        // We didn't get a socket
        endpoint.countDownConnection();
        if (endpoint.isRunning()) {
            // Introduce delay if necessary
            errorDelay = handleExceptionWithDelay(errorDelay);
            // re-throw
            throw ioe;
        } else {
            break;
        }
    }
    // Successful accept, reset the error delay
    errorDelay = 0;

    // Configure the socket
    if (endpoint.isRunning() && !endpoint.isPaused()) {
        // setSocketOptions() will hand the socket off to
        // an appropriate processor if successful
        if (!endpoint.setSocketOptions(socket)) {
            endpoint.closeSocket(socket);
        }
    } else {
```

```
                endpoint.destroySocket(socket);
            }
        } catch (Throwable t) {
            ExceptionUtils.handleThrowable(t);
            String msg = sm.getString("endpoint.accept.fail");
            // APR specific.
            // Could push this down but not sure it is worth the trouble.
            if (t instanceof Error) {
                Error e = (Error) t;
                if (e.getError() == 233) {
                    // Not an error on HP-UX so log as a warning
                    // so it can be filtered out on that platform
                    // See bug 50273
                    log.warn(msg, t);
                } else {
                    log.error(msg, t);
                }
            } else {
                    log.error(msg, t);
            }
        }
    }
    state = AcceptorState.ENDED;
}
```

1.1) NioEndpoint#setSocketOptions（处理 Socket）

配置完 Socket 以后将 Socket 封装到 NioChannel 中，并注册到 Poller,值的一提的是，我们一开始就启动了多个 Poller 线程，注册的时候，连接是公平的分配到每个 Poller 的。NioEndpoint 维护了一个 Poller 数组，当一个连接分配给 pollers[index] 时，下一个连接就会分配给 pollers[(index+1)%pollers.length].

```
protected boolean setSocketOptions(SocketChannel socket) {
    // Process the connection
    try {
        //disable blocking, APR style, we are gonna be polling it
        socket.configureBlocking(false);
        Socket sock = socket.socket();
        socketProperties.setProperties(sock);

        NioChannel channel = nioChannels.pop();
        if (channel == null) {
            SocketBufferHandler bufhandler = new SocketBufferHandler(
                    socketProperties.getAppReadBufSize(),
                    socketProperties.getAppWriteBufSize(),
                    socketProperties.getDirectBuffer());
            if (isSSLEnabled()) {
                channel = new SecureNioChannel(socket, bufhandler, selectorPool,
this);
            } else {
                channel = new NioChannel(socket, bufhandler);
            }
        } else {
            channel.setIOChannel(socket);
            channel.reset();
        }
        getPoller0().register(channel);
    } catch (Throwable t) {
        ExceptionUtils.handleThrowable(t);
        try {
            log.error("",t);
        } catch (Throwable tt) {
            ExceptionUtils.handleThrowable(tt);
        }
        // Tell to close the socket
        return false;
    }
    return true;
}
```

```
public Poller getPoller0() {
    int idx = Math.abs(pollerRotater.incrementAndGet()) % pollers.length;
    return pollers[idx];
}
```

1.1.1) Poller#register（将 Socket 放入 Poller 队列）

addEvent() 方法会将 Socket 添加到该 Poller 的 PollerEvent 队列中。到此 Acceptor 的任务就完成了。

```
public void register(final NioChannel socket) {
    socket.setPoller(this);
    NioSocketWrapper ka = new NioSocketWrapper(socket, NioEndpoint.this);
    socket.setSocketWrapper(ka);
    ka.setPoller(this);
    ka.setReadTimeout(getConnectionTimeout());
    ka.setWriteTimeout(getConnectionTimeout());
    ka.setKeepAliveLeft(NioEndpoint.this.getMaxKeepAliveRequests());
    ka.setSecure(isSSLEnabled());
    PollerEvent r = eventCache.pop();
    ka.interestOps(SelectionKey.OP_READ);//this is what OP_REGISTER turns into.
    if ( r==null) r = new PollerEvent(socket,ka,OP_REGISTER);
    else r.reset(socket,ka,OP_REGISTER);
    addEvent(r);
}
```

1.1.1.1) NioSocketWrapper#constructor（持有 NioEndpoint 的 SelectorPool）

```
public NioSocketWrapper(NioChannel channel, NioEndpoint endpoint) {
    super(channel, endpoint);
    pool = endpoint.getSelectorPool();
    socketBufferHandler = channel.getBufHandler();
}
```

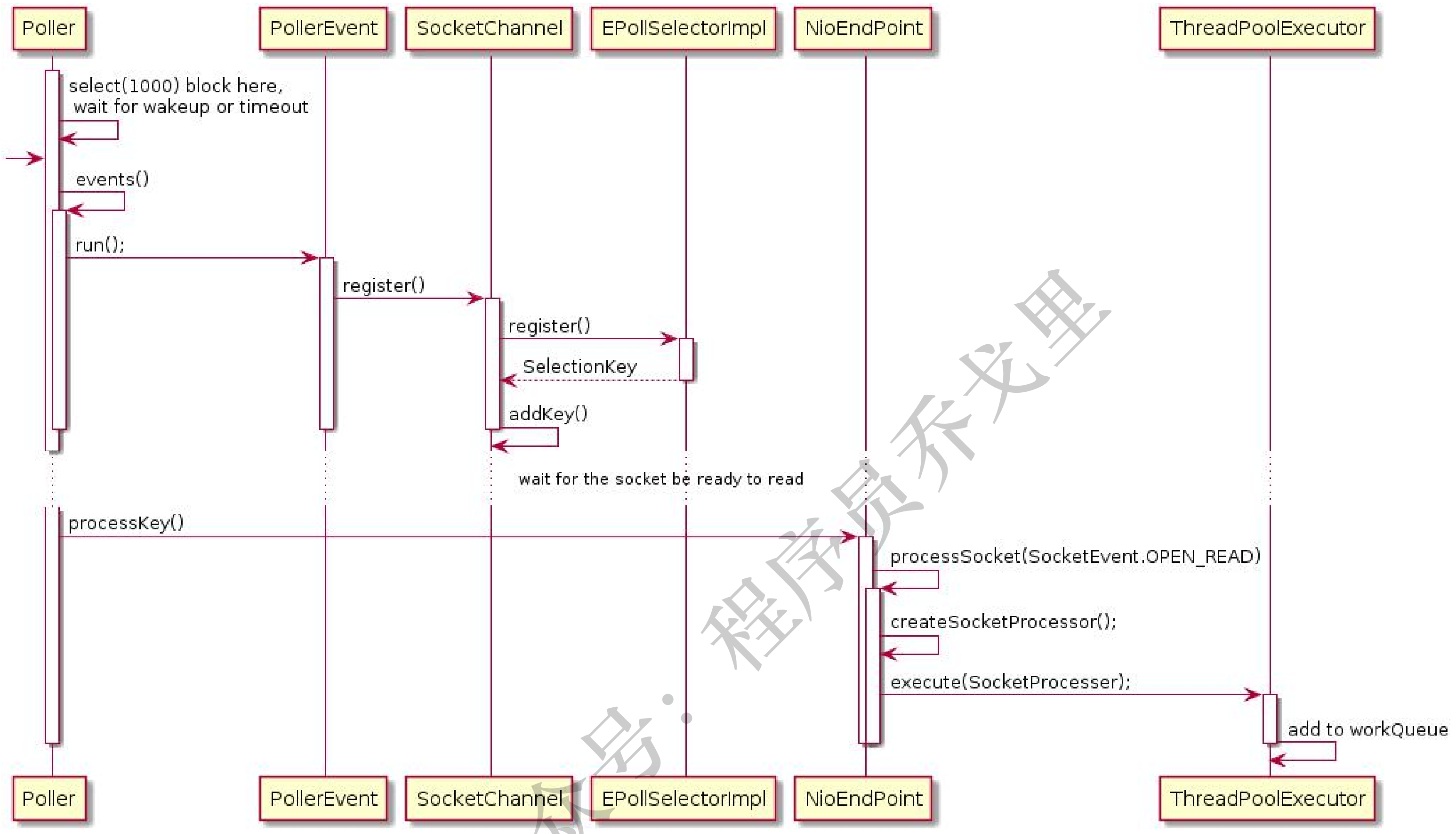
```
public SocketWrapperBase(E socket, AbstractEndpoint<E> endpoint) {
    this.socket = socket;
    this.endpoint = endpoint;
    ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    this.blockingStatusReadLock = lock.readLock();
    this.blockingStatusWriteLock = lock.writeLock();
}
```

1.1.1.2) Poller#addEvent

```
private void addEvent(PollerEvent event) {
    events.offer(event);
    if ( wakeupCounter.incrementAndGet() == 0 ) selector.wakeup();
}
```

```
private final SynchronizedQueue<PollerEvent> events =
        new SynchronizedQueue<>();
```

2) Poller#run （NIO,把队列中的就绪的 Socket 封装为 SocketProcessor 交给 Worker 线程池,subReactor）



1. selector.select(1000)。当 Poller 启动后因为 selector 中并没有已注册的 Channel，所以当执行到该方法时只能阻塞。所有的 Poller 共用一个 Selector，其实现类是 sun.nio.ch.EPollSelectorImpl
2. **events() 方法会将通过 addEvent() 方法添加到事件队列中的 Socket 注册到 EPollSelectorImpl，当 Socket 可读时，Poller 才对其进行处理**
3. createSocketProcessor() 方法将 Socket 封装到 SocketProcessor 中，SocketProcessor 实现了 Runnable 接口。worker 线程通过调用其 run() 方法来对 Socket 进行处理。
4. execute(SocketProcessor) 方法将 SocketProcessor 提交到线程池，放入线程池的 workQueue 中。workQueue 是 BlockingQueue 的实例。到此 Poller 的任务就完成了。

- 调用 selector 的 select()函数，监听就绪事件
- 根据向 selector 中注册的 key 遍历 channel 中已经就绪的 keys，并处理 key
- 处理 key 对应的 channel，调用 NioEndPoint 的 processSocket()
- 从 SocketProcessor 池中取出空闲的 SocketProcessor，关联 socketWrapper，提交运行 SocketProcessor

```
public Poller() throws IOException {
    this.selector = Selector.open();
}
```

它的 selector 是初始化时开启的，每个 Poller 对应着自己的 Selector，监听该 Poller 对应的 SocketChannel 的 Read 事件。当 Poller 队列中加入新的 Socket 时，会将 Socket 注册在 selector 上，这样 selector 就可以监测 socket 就绪事件了。

```
public void run() {
    // Loop until destroy() is called
    while (true) {

        boolean hasEvents = false;

        try {
            if (!close) {
                hasEvents = events();
                if (wakeupCounter.getAndSet(-1) > 0) {
                    //if we are here, means we have other stuff to do
                    //do a non blocking select
                    keyCount = selector.selectNow();
                } else {
                    keyCount = selector.select(selectorTimeout);
                }
                wakeupCounter.set(0);
            }
            if (close) {
                events();
                timeout(0, false);
                try {
                    selector.close();
                } catch (IOException ioe) {
                    log.error(sm.getString("endpoint.nio.selectorCloseFail"),
ioe);
                }
                break;
            }
        } catch (Throwable x) {
            ExceptionUtils.handleThrowable(x);
            log.error("",x);
            continue;
        }
        //either we timed out or we woke up, process events first
        if ( keyCount == 0 ) hasEvents = (hasEvents | events());

        Iterator<SelectionKey> iterator =
            keyCount > 0 ? selector.selectedKeys().iterator() : null;
        // Walk through the collection of ready keys and dispatch
        // any active event.
        while (iterator != null && iterator.hasNext()) {
            SelectionKey sk = iterator.next();
            NioSocketWrapper attachment = (NioSocketWrapper)sk.attachment();
            // Attachment may be null if another thread has called
            // cancelledKey()
            if (attachment == null) {
                iterator.remove();
            } else {
                // 有 Socket 出现读事件
                iterator.remove();
                processKey(sk, attachment);
            }
        }//while

        //process timeouts
        timeout(keyCount,hasEvents);
    }//while

    getStopLatch().countDown();
}
```

2.1) Poller#events（将队列中的 Socket 注册到 Selector）

events() 方法会将通过 addEvent() 方法添加到事件队列中的 Socket 注册到 EPollSelectorImpl，当 Socket 可读时，Poller 才对其进行处理。

```
public boolean events() {
    boolean result = false;

    PollerEvent pe = null;
    for (int i = 0, size = events.size(); i < size && (pe = events.poll()) !=
null; i++ ) {
        result = true;
        try {
            pe.run();
            pe.reset();
            if (running && !paused) {
                eventCache.push(pe);
            }
        } catch ( Throwable x ) {
            log.error("",x);
        }
    }

    return result;
}
```

2.1.1) PollerEvent#run（注册到 Selector）

```
public void run() {
    if (interestOps == OP_REGISTER) {
        try {
            socket.getIOChannel().register(
                    socket.getPoller().getSelector(), SelectionKey.OP_READ,
socketWrapper);
        } catch (Exception x) {
            log.error(sm.getString("endpoint.nio.registerFail"), x);
        }
    } else {
        final SelectionKey key =
socket.getIOChannel().keyFor(socket.getPoller().getSelector());
        try {
            if (key == null) {
                // The key was cancelled (e.g. due to socket closure)
                // and removed from the selector while it was being
                // processed. Count down the connections at this point
                // since it won't have been counted down when the socket
                // closed.
                socket.socketWrapper.getEndpoint().countDownConnection();
            } else {
                final NioSocketWrapper socketWrapper = (NioSocketWrapper)
key.attachment();
                if (socketWrapper != null) {
                    //we are registering the key to start with, reset the fairness
counter.
                    int ops = key.interestOps() | interestOps;
                    socketWrapper.interestOps(ops);
                    key.interestOps(ops);
                } else {
                    socket.getPoller().cancelledKey(key);
                }
            }
        } catch (CancelledKeyException ckx) {
            try {
                socket.getPoller().cancelledKey(key);
            } catch (Exception ignore) {}
        }
    }
}
```

2.2) Poller#processKey（将就绪的 Socket 交给线程池）

```
protected void processKey(SelectionKey sk, NioSocketWrapper attachment) {
    try {
        if ( close ) {
            cancelledKey(sk);
        } else if ( sk.isValid() && attachment != null ) {
            if (sk.isReadable() || sk.isWritable() ) {
                if ( attachment.getSendfileData() != null ) {
                    processSendfile(sk,attachment, false);
                } else {
                    unreg(sk, attachment, sk.readyOps());
                    boolean closeSocket = false;
                    // Read goes before write
                    if (sk.isReadable()) {
                        if (!processSocket(attachment, SocketEvent.OPEN_READ,
true)) {
                            closeSocket = true;
                        }
                    }
                    if (!closeSocket && sk.isWritable()) {
                        if (!processSocket(attachment, SocketEvent.OPEN_WRITE,
true)) {
                            closeSocket = true;
                        }
                    }
                    if (closeSocket) {
                        cancelledKey(sk);
                    }
                }
            }
        } else {
            //invalid key
            cancelledKey(sk);
        }
    } catch ( CancelledKeyException ckx ) {
        cancelledKey(sk);
    } catch (Throwable t) {
        ExceptionUtils.handleThrowable(t);
        log.error("",t);
    }
}
```

2.2.1) AbstractEndpoint#processSocket

createSocketProcessor() 方法将 Socket 封装到 SocketProcessor 中，SocketProcessor 实现了 Runnable 接口。worker 线程通过调用其 run() 方法来对 Socket 进行处理。

```
public boolean processSocket(SocketWrapperBase<S> socketWrapper,
        SocketEvent event, boolean dispatch) {
    try {
        if (socketWrapper == null) {
            return false;
        }
        SocketProcessorBase<S> sc = processorCache.pop();
        if (sc == null) {
            sc = createSocketProcessor(socketWrapper, event);
        } else {
            sc.reset(socketWrapper, event);
        }
        Executor executor = getExecutor();
        if (dispatch && executor != null) {
            executor.execute(sc);
        } else {
            sc.run();
        }
    } catch (RejectedExecutionException ree) {
        getLog().warn(sm.getString("endpoint.executor.fail", socketWrapper) ,
ree);
        return false;
    } catch (Throwable t) {
        ExceptionUtils.handleThrowable(t);
        // This means we got an OOM or similar creating a thread, or that
        // the pool and its queue are full
        getLog().error(sm.getString("endpoint.process.fail"), t);
        return false;
    }
    return true;
}
```

2.2.1.1) NioEndpoint#createSocketProcessor

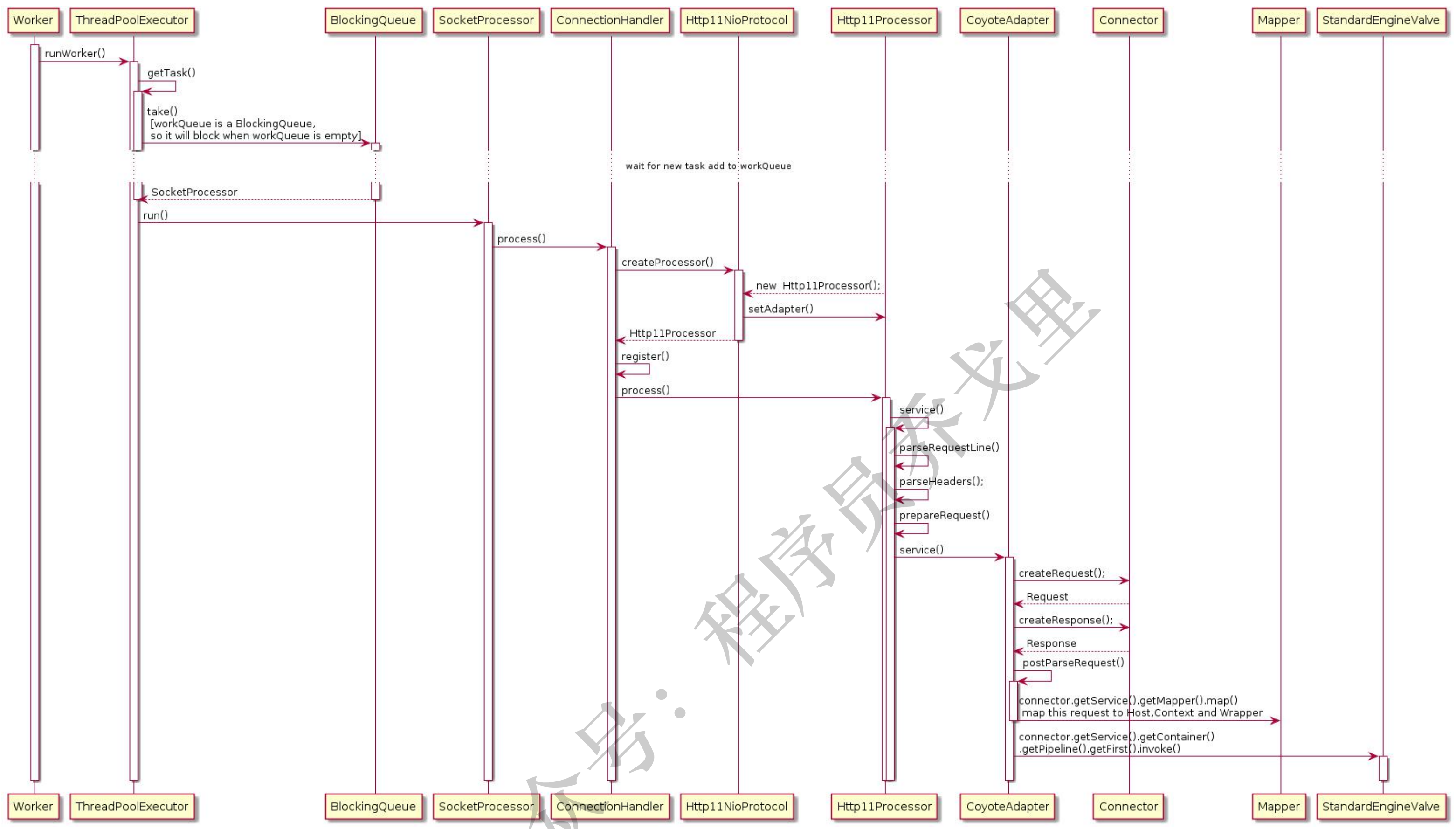
```
protected SocketProcessorBase<NioChannel> createSocketProcessor(
        SocketWrapperBase<NioChannel> socketWrapper, SocketEvent event) {
    return new SocketProcessor(socketWrapper, event);
}
```

```
public SocketProcessor(SocketWrapperBase<NioChannel> socketWrapper,
SocketEvent event) {
    super(socketWrapper, event);
}
```

```
public SocketProcessorBase(SocketWrapperBase<S> socketWrapper, SocketEvent
event) {
    reset(socketWrapper, event);
}
```

```
public void reset(SocketWrapperBase<S> socketWrapper, SocketEvent event) {
    Objects.requireNonNull(event);
    this.socketWrapper = socketWrapper;
    this.event = event;
}
```

3) Worker#run （将 SocketProcessor 封装为 Request,IO Handler）



1. worker 线程被创建以后就执行 ThreadPoolExecutor 的 runWorker() 方法，试图从 workQueue 中取待处理任务，但是一开始 workQueue 是空的，所以 worker 线程会阻塞在 workQueue.take() 方法。
2. 当新任务添加到 workQueue 后，workQueue.take() 方法会返回一个 Runnable，通常是 SocketProcessor,然后 worker 线程调用 SocketProcessor 的 run() 方法对 Socket 进行处理。
3. createProcessor() 会创建一个 Http11Processor，它用来解析 Socket，将 Socket 中的内容封装到 Request 中。注意这个 Request 是临时使用的一个类，它的全类名是 org.apache.coyote.Request，
4. postParseRequest() 方法封装一下 Request，并处理一下映射关系(从 URL 映射到相应的 Host、Context、Wrapper)。
5. CoyoteAdapter 将 Rquest 提交给 Container 处理之前，并将 org.apache.coyote.Request 封装到 org.apache.catalina.connector.Request，传递给 Container 处理的 Request 是 org.apache.catalina.connector.Request。
6. connector.getService().getMapper().map()，用来在 Mapper 中查询 URL 的映射关系。映射关系会保留到 org.apache.catalina.connector.Request 中，Container 处理阶段

request.getHost() 是使用的就是这个阶段查询到的映射主机，以此类推 request.getContext()、request.getWrapper() 都是。

7. connector.getService().getContainer().getPipeline().getFirst().invoke() 会将请求传递到 Container 处理，当然了 Container 处理也是在 Worker 线程中执行的，但是这是一个相对独立的模块，所以单独分出来一节。

```java
/** Delegates main run loop to outer runWorker  */
public void run() {
    runWorker(this);
}
```

```java
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        while (task != null || (task = getTask()) != null) {
            w.lock();
            // If pool is stopping, ensure thread is interrupted;
            // if not, ensure thread is not interrupted.  This
            // requires a recheck in second case to deal with
            // shutdownNow race while clearing interrupt
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt();
            try {
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
                    task.run();
                } catch (RuntimeException x) {
                    thrown = x; throw x;
                } catch (Error x) {
                    thrown = x; throw x;
                } catch (Throwable x) {
                    thrown = x; throw new Error(x);
                } finally {
                    afterExecute(task, thrown);
                }
            } finally {
                task = null;
                w.completedTasks++;
                w.unlock();
            }
        }
        completedAbruptly = false;
    } finally {
        processWorkerExit(w, completedAbruptly);
    }
}
```

task 是 SocketProcessor 类型

3.1) SocketProcessor#run

```
public final void run() {
    synchronized (socketWrapper) {
        // It is possible that processing may be triggered for read and
        // write at the same time. The sync above makes sure that processing
        // does not occur in parallel. The test below ensures that if the
        // first event to be processed results in the socket being closed,
        // the subsequent events are not processed.
        if (socketWrapper.isClosed()) {
            return;
        }
        doRun();
    }
}
```

```
protected void doRun() {
    NioChannel socket = socketWrapper.getSocket();
    SelectionKey key =
socket.getIOChannel().keyFor(socket.getPoller().getSelector());

    try {
        int handshake = -1;

        try {
            if (key != null) {
                if (socket.isHandshakeComplete()) {
                    // No TLS handshaking required. Let the handler
                    // process this socket / event combination.
                    handshake = 0;
                } else if (event == SocketEvent.STOP || event ==
SocketEvent.DISCONNECT ||
                        event == SocketEvent.ERROR) {
                    // Unable to complete the TLS handshake. Treat it as
                    // if the handshake failed.
                    handshake = -1;
                } else {
                    handshake = socket.handshake(key.isReadable(),
key.isWritable());
                    // The handshake process reads/writes from/to the
                    // socket. status may therefore be OPEN_WRITE once
                    // the handshake completes. However, the handshake
                    // happens when the socket is opened so the status
                    // must always be OPEN_READ after it completes. It
                    // is OK to always set this as it is only used if
                    // the handshake completes.
                    event = SocketEvent.OPEN_READ;
                }
            }
        } catch (IOException x) {
            handshake = -1;
            if (log.isDebugEnabled()) log.debug("Error during SSL handshake",x);
        } catch (CancelledKeyException ckx) {
            handshake = -1;
        }
        if (handshake == 0) {
            SocketState state = SocketState.OPEN;
            // Process the request from this socket
            if (event == null) {
                state = getHandler().process(socketWrapper,
SocketEvent.OPEN_READ);
            } else {
                state = getHandler().process(socketWrapper, event);
            }
            if (state == SocketState.CLOSED) {
                close(socket, key);
            }
        } else if (handshake == -1 ) {
            close(socket, key);
        } else if (handshake == SelectionKey.OP_READ){
            socketWrapper.registerReadInterest();
        } else if (handshake == SelectionKey.OP_WRITE){
            socketWrapper.registerWriteInterest();
        }
    } catch (CancelledKeyException cx) {
        socket.getPoller().cancelledKey(key);
    } catch (VirtualMachineError vme) {
        ExceptionUtils.handleThrowable(vme);
    } catch (Throwable t) {
        log.error("", t);
        socket.getPoller().cancelledKey(key);
    } finally {
        socketWrapper = null;
        event = null;
        //return to cache
        // keep-alive 的实现
        if (running && !paused) {
            processorCache.push(this);
        }
    }
}
```

3.1.1) AbstractProtocol#process

先试图从 connections 中获取当前 Socket 对应的 Processor，如果没有找到的话从 recycledProcessors 中获取，也就是已经处理过连接但是没有被销毁的 Processor，这样做的好处是避免频繁地创建和销毁对象。**processor 还是为空的话，那就使用 createProcessor 创建。**

```
public SocketState process(SocketWrapperBase<S> wrapper, SocketEvent status)
{
    if (getLog().isDebugEnabled()) {
        getLog().debug(sm.getString("abstractConnectionHandler.process",
                wrapper.getSocket(), status));
    }
    if (wrapper == null) {
        // Nothing to do. Socket has been closed.
        return SocketState.CLOSED;
    }

    S socket = wrapper.getSocket();

    Processor processor = connections.get(socket);
    if (getLog().isDebugEnabled()) {

getLog().debug(sm.getString("abstractConnectionHandler.connectionsGet",
                processor, socket));
    }
    if (processor != null) {
        // Make sure an async timeout doesn't fire
        getProtocol().removeWaitingProcessor(processor);
    } else if (status == SocketEvent.DISCONNECT || status == SocketEvent.ERROR)
{
        // Nothing to do. Endpoint requested a close and there is no
        // longer a processor associated with this socket.
        return SocketState.CLOSED;
    }

    ContainerThreadMarker.set();

    try {
        if (processor == null) {
            String negotiatedProtocol = wrapper.getNegotiatedProtocol();
            if (negotiatedProtocol != null) {
                UpgradeProtocol upgradeProtocol =

getProtocol().getNegotiatedProtocol(negotiatedProtocol);
                if (upgradeProtocol != null) {
                    processor = upgradeProtocol.getProcessor(
                            wrapper, getProtocol().getAdapter());
                } else if (negotiatedProtocol.equals("http/1.1")) {
                    // Explicitly negotiated the default protocol.
                    // Obtain a processor below.
                } else {
                    // TODO:
                    // OpenSSL 1.0.2's ALPN callback doesn't support
                    // failing the handshake with an error if no
                    // protocol can be negotiated. Therefore, we need to
                    // fail the connection here. Once this is fixed,
                    // replace the code below with the commented out
                    // block.
                    if (getLog().isDebugEnabled()) {
                        getLog().debug(sm.getString(

"abstractConnectionHandler.negotiatedProcessor.fail",
                            negotiatedProtocol));
                    }
                    return SocketState.CLOSED;
                    /*
                     * To replace the code above once OpenSSL 1.1.0 is
                     * used.
                    // Failed to create processor. This is a bug.
                    throw new IllegalStateException(sm.getString(

"abstractConnectionHandler.negotiatedProcessor.fail",
                            negotiatedProtocol));
                    */
                }
            }
        }
        if (processor == null) {
            processor = recycledProcessors.pop();
            if (getLog().isDebugEnabled()) {

getLog().debug(sm.getString("abstractConnectionHandler.processorPop",
                        processor));
            }
        }
        if (processor == null) {
            processor = getProtocol().createProcessor();
            register(processor);
        }

        processor.setSslSupport(

wrapper.getSslSupport(getProtocol().getClientCertProvider()));

        // Associate the processor with the connection
        connections.put(socket, processor);

        SocketState state = SocketState.CLOSED;
        do {
            state = processor.process(wrapper, status);

            if (state == SocketState.UPGRADING) {
                // Get the HTTP upgrade handler
                UpgradeToken upgradeToken = processor.getUpgradeToken();
                // Retrieve leftover input
                ByteBuffer leftOverInput = processor.getLeftoverInput();
                if (upgradeToken == null) {
                    // Assume direct HTTP/2 connection
                    UpgradeProtocol upgradeProtocol =
getProtocol().getUpgradeProtocol("h2c");
                    if (upgradeProtocol != null) {
                        processor = upgradeProtocol.getProcessor(
                                wrapper, getProtocol().getAdapter());
                        wrapper.unRead(leftOverInput);
                        // Associate with the processor with the connection
                        connections.put(socket, processor);
                    } else {
                        if (getLog().isDebugEnabled()) {
                            getLog().debug(sm.getString(

"abstractConnectionHandler.negotiatedProcessor.fail",
                                    "h2c"));
                        }
                        return SocketState.CLOSED;
                    }
                } else {
                    HttpUpgradeHandler httpUpgradeHandler =
upgradeToken.getHttpUpgradeHandler();
                    // Release the Http11 processor to be re-used
                    release(processor);
                    // Create the upgrade processor
                    processor = getProtocol().createUpgradeProcessor(wrapper,
upgradeToken);
                    if (getLog().isDebugEnabled()) {

getLog().debug(sm.getString("abstractConnectionHandler.upgradeCreate",
                                processor, wrapper));
                    }
                    wrapper.unRead(leftOverInput);
                    // Mark the connection as upgraded
                    wrapper.setUpgraded(true);
                    // Associate with the processor with the connection
                    connections.put(socket, processor);
                    // Initialise the upgrade handler (which may trigger
                    // some IO using the new protocol which is why the lines
                    // above are necessary)
                    // This cast should be safe. If it fails the error
                    // handling for the surrounding try/catch will deal with
                    // it.
                    if (upgradeToken.getInstanceManager() == null) {
                        httpUpgradeHandler.init((WebConnection) processor);
                    } else {
                        ClassLoader oldCL =
upgradeToken.getContextBind().bind(false, null);
                        try {
                            httpUpgradeHandler.init((WebConnection) processor);
                        } finally {
                            upgradeToken.getContextBind().unbind(false, oldCL);
                        }
                    }
                }
            }
        } while ( state == SocketState.UPGRADING);

        if (state == SocketState.LONG) {
            // In the middle of processing a request/response. Keep the
            // socket associated with the processor. Exact requirements
            // depend on type of long poll
            longPoll(wrapper, processor);
            if (processor.isAsync()) {
                getProtocol().addWaitingProcessor(processor);
            }
        } else if (state == SocketState.OPEN) {
            // In keep-alive but between requests. OK to recycle
            // processor. Continue to poll for the next request.
            connections.remove(socket);
            release(processor);
            wrapper.registerReadInterest();
        } else if (state == SocketState.SENDFILE) {
            // Sendfile in progress. If it fails, the socket will be
            // closed. If it works, the socket either be added to the
            // poller (or equivalent) to await more data or processed
            // if there are any pipe-lined requests remaining.
        } else if (state == SocketState.UPGRADED) {
            // Don't add sockets back to the poller if this was a
            // non-blocking write otherwise the poller may trigger
            // multiple read events which may lead to thread starvation
            // in the connector. The write() method will add this socket
            // to the poller if necessary.
            if (status != SocketEvent.OPEN_WRITE) {
                longPoll(wrapper, processor);
            }
        } else if (state == SocketState.SUSPENDED) {
            // Don't add sockets back to the poller.
            // The resumeProcessing() method will add this socket
            // to the poller.
        } else {
            // Connection closed. OK to recycle the processor. Upgrade
            // processors are not recycled.
            connections.remove(socket);
            if (processor.isUpgrade()) {
                UpgradeToken upgradeToken = processor.getUpgradeToken();
                HttpUpgradeHandler httpUpgradeHandler =
upgradeToken.getHttpUpgradeHandler();
                InstanceManager instanceManager =
upgradeToken.getInstanceManager();
                if (instanceManager == null) {
                    httpUpgradeHandler.destroy();
                } else {
                    ClassLoader oldCL =
upgradeToken.getContextBind().bind(false, null);
                    try {
                        httpUpgradeHandler.destroy();
                    } finally {
                        try {

instanceManager.destroyInstance(httpUpgradeHandler);
                        } catch (Throwable e) {
                            ExceptionUtils.handleThrowable(e);

getLog().error(sm.getString("abstractConnectionHandler.error"), e);
                        }
                        upgradeToken.getContextBind().unbind(false, oldCL);
                    }
                }
            } else {
                release(processor);
            }
        }
        return state;
    } catch(java.net.SocketException e) {
        // SocketExceptions are normal
        getLog().debug(sm.getString(
                "abstractConnectionHandler.socketexception.debug"), e);
    } catch (java.io.IOException e) {
        // IOExceptions are normal
        getLog().debug(sm.getString(
                "abstractConnectionHandler.ioexception.debug"), e);
    } catch (ProtocolException e) {
        // Protocol exceptions normally mean the client sent invalid or
        // incomplete data.
        getLog().debug(sm.getString(
                "abstractConnectionHandler.protocolexception.debug"), e);
    }
    // Future developers: if you discover any other
    // rare-but-nonfatal exceptions, catch them here, and log as
    // above.
    catch (Throwable e) {
        ExceptionUtils.handleThrowable(e);
        // any other exception or error is odd. Here we log it
        // with "ERROR" level, so it will show up even on
        // less-than-verbose logs.
        getLog().error(sm.getString("abstractConnectionHandler.error"), e);
    } finally {
        ContainerThreadMarker.clear();
    }

    // Make sure socket/processor is removed from the list of current
    // connections
    connections.remove(socket);
    release(processor);
    return SocketState.CLOSED;
}
```

3.1.1.1) AbstractHttp11Protocol#createProcessor

createProcessor() 会创建一个 Http11Processor，它用来解析 Socket，将 Socket 中的内容封装到 Request 中。注意这个 Request 是临时使用的一个类，它的全类名是 org.apache.coyote.Request。

```
protected Processor createProcessor() {
    Http11Processor processor = new Http11Processor(this, adapter);
    return processor;
}
```

3.1.1.1.1) Http11Processor#constructor（创建 req 和 resp 缓冲区）

```
public Http11Processor(AbstractHttp11Protocol<?> protocol, Adapter adapter) {
    super(adapter);
    this.protocol = protocol;

    userDataHelper = new UserDataHelper(log);

    inputBuffer = new Http11InputBuffer(request,
protocol.getMaxHttpHeaderSize(),
           protocol.getRejectIllegalHeaderName());
    request.setInputBuffer(inputBuffer);

    outputBuffer = new Http11OutputBuffer(response,
```

```java
protocol.getMaxHttpHeaderSize());
    response.setOutputBuffer(outputBuffer);

    // Create and add the identity filters.
    inputBuffer.addFilter(new
IdentityInputFilter(protocol.getMaxSwallowSize()));
    outputBuffer.addFilter(new IdentityOutputFilter());

    // Create and add the chunked filters.
    inputBuffer.addFilter(new
ChunkedInputFilter(protocol.getMaxTrailerSize(),
            protocol.getAllowedTrailerHeadersInternal(),
protocol.getMaxExtensionSize(),
            protocol.getMaxSwallowSize()));
    outputBuffer.addFilter(new ChunkedOutputFilter());

    // Create and add the void filters.
    inputBuffer.addFilter(new VoidInputFilter());
    outputBuffer.addFilter(new VoidOutputFilter());

    // Create and add buffered input filter
    inputBuffer.addFilter(new BufferedInputFilter());

    // Create and add the chunked filters.
    //inputBuffer.addFilter(new GzipInputFilter());
    outputBuffer.addFilter(new GzipOutputFilter());

    pluggableFilterIndex = inputBuffer.getFilters().length;
}
```

```java
public AbstractProcessor(Adapter adapter) {
    this(adapter, new Request(), new Response());
}
```

```java
protected AbstractProcessor(Adapter adapter, Request coyoteRequest, Response
coyoteResponse) {
    this.adapter = adapter;
    asyncStateMachine = new AsyncStateMachine(this);
    request = coyoteRequest;
    response = coyoteResponse;
    response.setHook(this);
    request.setResponse(response);
    request.setHook(this);
}
```

3.1.1.1.1.1) Http11InputBuffer#constructor（存放解析后的 Request 信息）

```
public Http11InputBuffer(Request request, int headerBufferSize,
       boolean rejectIllegalHeaderName) {

    this.request = request;
    headers = request.getMimeHeaders();

    this.headerBufferSize = headerBufferSize;
    this.rejectIllegalHeaderName = rejectIllegalHeaderName;

    filterLibrary = new InputFilter[0];
    activeFilters = new InputFilter[0];
    lastActiveFilter = -1;

    parsingHeader = true;
    parsingRequestLine = true;
    parsingRequestLinePhase = 0;
    parsingRequestLineEol = false;
    parsingRequestLineStart = 0;
    parsingRequestLineQPos = -1;
    headerParsePos = HeaderParsePosition.HEADER_START;
    swallowInput = true;

    inputStreamInputBuffer = new SocketInputBuffer();
}
```

3.1.1.2) ConnectionHandler#register（注册 Http11Processor）

```
protected void register(Processor processor) {
    if (getProtocol().getDomain() != null) {
        synchronized (this) {
            try {
                long count = registerCount.incrementAndGet();
                RequestInfo rp =
                    processor.getRequest().getRequestProcessor();
                rp.setGlobalProcessor(global);
                ObjectName rpName = new ObjectName(
                        getProtocol().getDomain() +
                        ":type=RequestProcessor,worker="
                        + getProtocol().getName() +
                        ",name=" + getProtocol().getProtocolName() +
                        "Request" + count);
                if (getLog().isDebugEnabled()) {
                    getLog().debug("Register " + rpName);
                }
                Registry.getRegistry(null, null).registerComponent(rp,
                        rpName, null);
                rp.setRpName(rpName);
            } catch (Exception e) {
                getLog().warn("Error registering request");
            }
        }
    }
}
```

3.1.1.3) AbstractProcessorLight#process（Http11Processor 进行处理）

```
public SocketState process(SocketWrapperBase<?> socketWrapper, SocketEvent
status)
        throws IOException {

    SocketState state = SocketState.CLOSED;
    Iterator<DispatchType> dispatches = null;
    do {
        if (dispatches != null) {
            DispatchType nextDispatch = dispatches.next();
            state = dispatch(nextDispatch.getSocketStatus());
        } else if (status == SocketEvent.DISCONNECT) {
            // Do nothing here, just wait for it to get recycled
        } else if (isAsync() || isUpgrade() || state == SocketState.ASYNC_END)
{
            state = dispatch(status);
            if (state == SocketState.OPEN) {
                // There may be pipe-lined data to read. If the data isn't
                // processed now, execution will exit this loop and call
                // release() which will recycle the processor (and input
                // buffer) deleting any pipe-lined data. To avoid this,
                // process it now.
                state = service(socketWrapper);
            }
        } else if (status == SocketEvent.OPEN_WRITE) {
            // Extra write event likely after async, ignore
            state = SocketState.LONG;
        } else if (status == SocketEvent.OPEN_READ){
            state = service(socketWrapper);
        } else {
            // Default to closing the socket if the SocketEvent passed in
            // is not consistent with the current state of the Processor
            state = SocketState.CLOSED;
        }

        if (state != SocketState.CLOSED && isAsync()) {
            state = asyncPostProcess();
        }

        if (getLog().isDebugEnabled()) {
            getLog().debug("Socket: [" + socketWrapper +
                    "], Status in: [" + status +
                    "], State out: [" + state + "]");
        }

        if (dispatches == null || !dispatches.hasNext()) {
            // Only returns non-null iterator if there are
            // dispatches to process.
            dispatches = getIteratorAndClearDispatches();
        }
    } while (state == SocketState.ASYNC_END ||
            dispatches != null && state != SocketState.CLOSED);

    return state;
}
```

3.1.1.3.1) （service 骨架）Http11Processor#service（包含 servlet 后续处理，keep-alive 的实现）

1，org.apache.coyote.Request 是 tomcat 内部使用用于存放关于 request 消息的数据结构
2，org.apache.tomcat.util.buf.MessageBytes 用于存放消息，在 org.apache.coyote.Request 中大量用于存放解析后的 byte 字符
3，org.apache.tomcat.util.buf.ByteChunk 真正用于存放数据的数据结构，存放的是 byte[],org.apache.tomcat.util.buf.MessageBytes 使用它。

Request 存放着解析后的 Request 信息，其数据来自于 InputBuffer。
http 消息通过 inputBuffer 解析后放到 Request 中，Request 把它放到相应的 MessageBytes，最后 MessageBytes 把它存到 ByteChunk 里。

```
public SocketState service(SocketWrapperBase<?> socketWrapper)
    throws IOException {
```

```
    RequestInfo rp = request.getRequestProcessor();
    rp.setStage(org.apache.coyote.Constants.STAGE_PARSE);

    // Setting up the I/O
    setSocketWrapper(socketWrapper);
    inputBuffer.init(socketWrapper);
    outputBuffer.init(socketWrapper);

    // Flags
    keepAlive = true;
    openSocket = false;
    readComplete = true;
    boolean keptAlive = false;
    SendfileState sendfileState = SendfileState.DONE;

    while (!getErrorState().isError() && keepAlive && !isAsync() &&
upgradeToken == null &&
            sendfileState == SendfileState.DONE && !protocol.isPaused()) {

        // Parsing the request header
        try {
            if (!inputBuffer.parseRequestLine(keptAlive,
protocol.getConnectionTimeout(),
                    protocol.getKeepAliveTimeout())) {
                if (inputBuffer.getParsingRequestLinePhase() == -1) {
                    return SocketState.UPGRADING;
                } else if (handleIncompleteRequestLineRead()) {
                    break;
                }
            }

            if (protocol.isPaused()) {
                // 503 - Service unavailable
                response.setStatus(503);
                setErrorState(ErrorState.CLOSE_CLEAN, null);
            } else {
                keptAlive = true;
                // Set this every time in case limit has been changed via JMX

request.getMimeHeaders().setLimit(protocol.getMaxHeaderCount());
                if (!inputBuffer.parseHeaders()) {
                    // We've read part of the request, don't recycle it
                    // instead associate it with the socket
                    openSocket = true;
                    readComplete = false;
                    break;
                }
                if (!protocol.getDisableUploadTimeout()) {

socketWrapper.setReadTimeout(protocol.getConnectionUploadTimeout());
                }
            }
        } catch (IOException e) {
            if (log.isDebugEnabled()) {
                log.debug(sm.getString("http11processor.header.parse"), e);
            }
            setErrorState(ErrorState.CLOSE_CONNECTION_NOW, e);
            break;
        } catch (Throwable t) {
            ExceptionUtils.handleThrowable(t);
            UserDataHelper.Mode logMode = userDataHelper.getNextMode();
            if (logMode != null) {
                String message = sm.getString("http11processor.header.parse");
                switch (logMode) {
                    case INFO_THEN_DEBUG:
                        message += sm.getString("http11processor.fallToDebug");
                        //$FALL-THROUGH$
                    case INFO:
                        log.info(message, t);
                        break;
                    case DEBUG:
                        log.debug(message, t);
                }
            }
            // 400 - Bad Request
            response.setStatus(400);
            setErrorState(ErrorState.CLOSE_CLEAN, t);
            getAdapter().log(request, response, 0);
        }

        // Has an upgrade been requested?
        Enumeration<String> connectionValues =
request.getMimeHeaders().values("Connection");
        boolean foundUpgrade = false;
        while (connectionValues.hasMoreElements() && !foundUpgrade) {
            foundUpgrade = connectionValues.nextElement().toLowerCase(
                    Locale.ENGLISH).contains("upgrade");
        }
```

```java
        if (foundUpgrade) {
            // Check the protocol
            String requestedProtocol = request.getHeader("Upgrade");

            UpgradeProtocol upgradeProtocol =
protocol.getUpgradeProtocol(requestedProtocol);
            if (upgradeProtocol != null) {
                if (upgradeProtocol.accept(request)) {
                    // TODO Figure out how to handle request bodies at this
                    // point.

response.setStatus(HttpServletResponse.SC_SWITCHING_PROTOCOLS);
                    response.setHeader("Connection", "Upgrade");
                    response.setHeader("Upgrade", requestedProtocol);
                    action(ActionCode.CLOSE,  null);
                    getAdapter().log(request, response, 0);

                    InternalHttpUpgradeHandler upgradeHandler =
                            upgradeProtocol.getInternalUpgradeHandler(
                                    socketWrapper, getAdapter(),
cloneRequest(request));
                    UpgradeToken upgradeToken = new UpgradeToken(upgradeHandler,
null, null);
                    action(ActionCode.UPGRADE, upgradeToken);
                    return SocketState.UPGRADING;
                }
            }
        }

        if (!getErrorState().isError()) {
            // Setting up filters, and parse some request headers
            rp.setStage(org.apache.coyote.Constants.STAGE_PREPARE);
            try {
                prepareRequest();
            } catch (Throwable t) {
                ExceptionUtils.handleThrowable(t);
                if (log.isDebugEnabled()) {
                    log.debug(sm.getString("http11processor.request.prepare"),
t);
                }
                // 500 - Internal Server Error
                response.setStatus(500);
                setErrorState(ErrorState.CLOSE_CLEAN, t);
                getAdapter().log(request, response, 0);
            }
        }

        int maxKeepAliveRequests = protocol.getMaxKeepAliveRequests();
        if (maxKeepAliveRequests == 1) {
            keepAlive = false;
        } else if (maxKeepAliveRequests > 0 &&
                socketWrapper.decrementKeepAlive() <= 0) {
            keepAlive = false;
        }

        // Process the request in the adapter
        if (!getErrorState().isError()) {
            try {
                rp.setStage(org.apache.coyote.Constants.STAGE_SERVICE);
getAdapter().service(request, response);
                // Handle when the response was committed before a serious
                // error occurred.  Throwing a ServletException should both
                // set the status to 500 and set the errorException.
                // If we fail here, then the response is likely already
                // committed, so we can't try and set headers.
                if(keepAlive && !getErrorState().isError() && !isAsync() &&
                        statusDropsConnection(response.getStatus())) {
                    setErrorState(ErrorState.CLOSE_CLEAN, null);
                }
            } catch (InterruptedIOException e) {
                setErrorState(ErrorState.CLOSE_CONNECTION_NOW, e);
            } catch (HeadersTooLargeException e) {
                log.error(sm.getString("http11processor.request.process"), e);
                // The response should not have been committed but check it
                // anyway to be safe
                if (response.isCommitted()) {
                    setErrorState(ErrorState.CLOSE_NOW, e);
                } else {
                    response.reset();
                    response.setStatus(500);
                    setErrorState(ErrorState.CLOSE_CLEAN, e);
                    response.setHeader("Connection", "close"); // TODO: Remove
                }
            } catch (Throwable t) {
                ExceptionUtils.handleThrowable(t);
                log.error(sm.getString("http11processor.request.process"), t);
                // 500 - Internal Server Error
                response.setStatus(500);
                setErrorState(ErrorState.CLOSE_CLEAN, t);
                getAdapter().log(request, response, 0);
            }
        }

        // Finish the handling of the request
        rp.setStage(org.apache.coyote.Constants.STAGE_ENDINPUT);
        if (!isAsync()) {
            // If this is an async request then the request ends when it has
            // been completed. The AsyncContext is responsible for calling
            // endRequest() in that case.
            endRequest();
        }
        rp.setStage(org.apache.coyote.Constants.STAGE_ENDOUTPUT);

        // If there was an error, make sure the request is counted as
        // and error, and update the statistics counter
        if (getErrorState().isError()) {
            response.setStatus(500);
        }

        if (!isAsync() || getErrorState().isError()) {
            request.updateCounters();
            if (getErrorState().isIoAllowed()) {
                inputBuffer.nextRequest();
                outputBuffer.nextRequest();
            }
        }

        if (!protocol.getDisableUploadTimeout()) {
            int connectionTimeout = protocol.getConnectionTimeout();
            if(connectionTimeout > 0) {
                socketWrapper.setReadTimeout(connectionTimeout);
            } else {
                socketWrapper.setReadTimeout(0);
            }
        }

        rp.setStage(org.apache.coyote.Constants.STAGE_KEEPALIVE);

        sendfileState = processSendfile(socketWrapper);
    }
```

```
    rp.setStage(org.apache.coyote.Constants.STAGE_ENDED);

    if (getErrorState().isError() || protocol.isPaused()) {
        return SocketState.CLOSED;
    } else if (isAsync()) {
        return SocketState.LONG;
    } else if (isUpgrade()) {
        return SocketState.UPGRADING;
    } else {
        if (sendfileState == SendfileState.PENDING) {
            return SocketState.SENDFILE;
        } else {
            if (openSocket) {
                if (readComplete) {
                    return SocketState.OPEN;
                } else {
                    return SocketState.LONG;
                }
            } else {
                return SocketState.CLOSED;
            }
        }
    }
}
```

3.1.1.3.1.1) Http11inputBuffer#init（初始化 InputBuffer）

```
void init(SocketWrapperBase<?> socketWrapper) {

    wrapper = socketWrapper;
    wrapper.setAppReadBufHandler(this);

    int bufLength = headerBufferSize +
            wrapper.getSocketBufferHandler().getReadBuffer().capacity();
    if (byteBuffer == null || byteBuffer.capacity() < bufLength) {
        byteBuffer = ByteBuffer.allocate(bufLength);
        byteBuffer.position(0).limit(0);
    }
}
```

3.1.1.3.1.2) Http11inputBuffer#parseRequestLine（解析请求行）

将 SocketBufferHandler 中的 readBuffer 的部分数据填充到 byteBuffer 中，读取 byteBuffer，

解析，将结果存入 Request

```
boolean parseRequestLine(boolean keptAlive) throws IOException {

    // check state
    if (!parsingRequestLine) {
        return true;
    }
    //
    // Skipping blank lines
    //
    if (parsingRequestLinePhase < 2) {
        byte chr = 0;
        do {

            // Read new bytes if needed
            if (byteBuffer.position() >= byteBuffer.limit()) {
                if (keptAlive) {
                    // Haven't read any request data yet so use the keep-alive
                    // timeout.

wrapper.setReadTimeout(wrapper.getEndpoint().getKeepAliveTimeout());
                }
                if (!fill(false)) {
                    // A read is pending, so no longer in initial state
                    parsingRequestLinePhase = 1;
                    return false;
                }
                // At least one byte of the request has been received.
                // Switch to the socket timeout.
                wrapper.setReadTimeout(wrapper.getEndpoint().getSoTimeout());
            }
            if (!keptAlive && byteBuffer.position() == 0 &&
byteBuffer.limit() >= CLIENT_PREFACE_START.length - 1) {
                boolean prefaceMatch = true;
                for (int i = 0; i < CLIENT_PREFACE_START.length && prefaceMatch;
i++) {
                    if (CLIENT_PREFACE_START[i] != byteBuffer.get(i)) {
                        prefaceMatch = false;
                    }
                }
                if (prefaceMatch) {
                    // HTTP/2 preface matched
                    parsingRequestLinePhase = -1;
                    return false;
                }
            }
            // Set the start time once we start reading data (even if it is
            // just skipping blank lines)
            if (request.getStartTime() < 0) {
                request.setStartTime(System.currentTimeMillis());
            }
            chr = byteBuffer.get();
        } while ((chr == Constants.CR) || (chr == Constants.LF));
        byteBuffer.position(byteBuffer.position() - 1);

        parsingRequestLineStart = byteBuffer.position();
        parsingRequestLinePhase = 2;
        if (log.isDebugEnabled()) {
            log.debug("Received ["
                    + new String(byteBuffer.array(), byteBuffer.position(),
byteBuffer.remaining(), StandardCharsets.ISO_8859_1) + "]");
        }
    }
    if (parsingRequestLinePhase == 2) {
        //
        // Reading the method name
        // Method name is a token
        //
        boolean space = false;
        while (!space) {
            // Read new bytes if needed
            if (byteBuffer.position() >= byteBuffer.limit()) {
                if (!fill(false)) // request line parsing
                    return false;
            }
            // Spec says method name is a token followed by a single SP but
            // also be tolerant of multiple SP and/or HT.
            int pos = byteBuffer.position();
            byte chr = byteBuffer.get();
            if (chr == Constants.SP || chr == Constants.HT) {
                space = true;
                request.method().setBytes(byteBuffer.array(),
parsingRequestLineStart,
                        pos - parsingRequestLineStart);
            } else if (!HttpParser.isToken(chr)) {
                byteBuffer.position(byteBuffer.position() - 1);
                throw new
IllegalArgumentException(sm.getString("iib.invalidmethod"));
```

```java
        }
    }
    parsingRequestLinePhase = 3;
}
if (parsingRequestLinePhase == 3) {
    // Spec says single SP but also be tolerant of multiple SP and/or HT
    boolean space = true;
    while (space) {
        // Read new bytes if needed
        if (byteBuffer.position() >= byteBuffer.limit()) {
            if (!fill(false)) // request line parsing
                return false;
        }
        byte chr = byteBuffer.get();
        if (!(chr == Constants.SP || chr == Constants.HT)) {
            space = false;
            byteBuffer.position(byteBuffer.position() - 1);
        }
    }
    parsingRequestLineStart = byteBuffer.position();
    parsingRequestLinePhase = 4;
}
if (parsingRequestLinePhase == 4) {
    // Mark the current buffer position

    int end = 0;
    //
    // Reading the URI
    //
    boolean space = false;
    while (!space) {
        // Read new bytes if needed
        if (byteBuffer.position() >= byteBuffer.limit()) {
            if (!fill(false)) // request line parsing
                return false;
        }
        int pos = byteBuffer.position();
        byte chr = byteBuffer.get();
        if (chr == Constants.SP || chr == Constants.HT) {
            space = true;
            end = pos;
        } else if (chr == Constants.CR || chr == Constants.LF) {
            // HTTP/0.9 style request
            parsingRequestLineEol = true;
```

```
                space = true;
                end = pos;
            } else if (chr == Constants.QUESTION && parsingRequestLineQPos ==
-1) {
                parsingRequestLineQPos = pos;
            } else if (HttpParser.isNotRequestTarget(chr)) {
                throw new
IllegalArgumentException(sm.getString("iib.invalidRequestTarget"));
            }
        }
        if (parsingRequestLineQPos >= 0) {
            request.queryString().setBytes(byteBuffer.array(),
parsingRequestLineQPos + 1,
                    end - parsingRequestLineQPos - 1);
            request.requestURI().setBytes(byteBuffer.array(),
parsingRequestLineStart,
                    parsingRequestLineQPos - parsingRequestLineStart);
        } else {
            request.requestURI().setBytes(byteBuffer.array(),
parsingRequestLineStart,
                    end - parsingRequestLineStart);
        }
        parsingRequestLinePhase = 5;
    }
    if (parsingRequestLinePhase == 5) {
        // Spec says single SP but also be tolerant of multiple and/or HT
        boolean space = true;
        while (space) {
            // Read new bytes if needed
            if (byteBuffer.position() >= byteBuffer.limit()) {
                if (!fill(false)) // request line parsing
                    return false;
            }
            byte chr = byteBuffer.get();
            if (!(chr == Constants.SP || chr == Constants.HT)) {
                space = false;
                byteBuffer.position(byteBuffer.position() - 1);
            }
        }
        parsingRequestLineStart = byteBuffer.position();
        parsingRequestLinePhase = 6;

        // Mark the current buffer position
        end = 0;
    }
    if (parsingRequestLinePhase == 6) {
        //
        // Reading the protocol
        // Protocol is always "HTTP/" DIGIT "." DIGIT
        //
        while (!parsingRequestLineEol) {
            // Read new bytes if needed
            if (byteBuffer.position() >= byteBuffer.limit()) {
                if (!fill(false)) // request line parsing
                    return false;
            }

            int pos = byteBuffer.position();
            byte chr = byteBuffer.get();
            if (chr == Constants.CR) {
                end = pos;
            } else if (chr == Constants.LF) {
                if (end == 0) {
                    end = pos;
                }
                parsingRequestLineEol = true;
            } else if (!HttpParser.isHttpProtocol(chr)) {
                throw new
IllegalArgumentException(sm.getString("iib.invalidHttpProtocol"));
            }
        }

        if ((end - parsingRequestLineStart) > 0) {
            request.protocol().setBytes(byteBuffer.array(),
parsingRequestLineStart,
                    end - parsingRequestLineStart);
        } else {
            request.protocol().setString("");
        }
        parsingRequestLine = false;
        parsingRequestLinePhase = 0;
        parsingRequestLineEol = false;
        parsingRequestLineStart = 0;
        return true;
    }
    throw new IllegalStateException(
            "Invalid request line parse phase:" + parsingRequestLinePhase);
}
```

3.1.1.3.1.2.1) Http11InputBuffer#fill（）

```
/**
 * Attempts to read some data into the input buffer.
 *
 * @return <code>true</code> if more data was added to the input buffer
 *         otherwise <code>false</code>
 */
private boolean fill(boolean block) throws IOException {

    if (parsingHeader) {
        if (byteBuffer.limit() >= headerBufferSize) {
            throw new
IllegalArgumentException(sm.getString("iib.requestheadertoolarge.error"));
        }
    } else {
        byteBuffer.limit(end).position(end);
    }

    byteBuffer.mark();
    if (byteBuffer.position() < byteBuffer.limit()) {
        byteBuffer.position(byteBuffer.limit());
    }
    byteBuffer.limit(byteBuffer.capacity());
    int nRead = wrapper.read(block, byteBuffer);
    byteBuffer.limit(byteBuffer.position()).reset();
    if (nRead > 0) {
        return true;
    } else if (nRead == -1) {
        throw new EOFException(sm.getString("iib.eof.error"));
    } else {
        return false;
    }

}
```

3.1.1.3.1.2.1.1) NioEndpoint#read

```
public int read(boolean block, ByteBuffer to) throws IOException {
    int nRead = populateReadBuffer(to);
    if (nRead > 0) {
        return nRead;
        /*
         * Since more bytes may have arrived since the buffer was last
         * filled, it is an option at this point to perform a
         * non-blocking read. However correctly handling the case if
         * that read returns end of stream adds complexity. Therefore,
         * at the moment, the preference is for simplicity.
         */
    }

    // The socket read buffer capacity is socket.appReadBufSize
    int limit = socketBufferHandler.getReadBuffer().capacity();
    // 如果 to 的剩余可用比 read buffer 还要大。那么直接从 socketchannel 读到 to
    if (to.remaining() >= limit) {
        to.limit(to.position() + limit);
        nRead = fillReadBuffer(block, to);
        updateLastRead();
    } else {
        // Fill the read buffer as best we can.
        nRead = fillReadBuffer(block);
        updateLastRead();

        // Fill as much of the remaining byte array as possible with the
        // data that was just read
        if (nRead > 0) {
            nRead = populateReadBuffer(to);
        }
    }
    return nRead;
}
```

3.1.1.3.1.2.1.1.1) SocketWrapperBase#populateReadBuffer （将 SocketBufferHandler 中的 ByteBuffer 拷贝到 Http11InputBuffer 中的 ByteBuffer）

```
protected int populateReadBuffer(ByteBuffer to) {
    // Is there enough data in the read buffer to satisfy this request?
    // Copy what data there is in the read buffer to the byte array
    socketBufferHandler.configureReadBufferForRead();
    int nRead = transfer(socketBufferHandler.getReadBuffer(), to);

    if (log.isDebugEnabled()) {
        log.debug("Socket: [" + this + "], Read from buffer: [" + nRead + "]");
    }
    return nRead;
}
```

```
protected static int transfer(ByteBuffer from, ByteBuffer to) {
    int max = Math.min(from.remaining(), to.remaining());
    if (max > 0) {
        int fromLimit = from.limit();
        from.limit(from.position() + max);
        to.put(from);
        from.limit(fromLimit);
    }
    return max;
}
```

3.1.1.3.1.2.1.1.2) NioEndpoint#fillReadBuffer（从 channel 或者 selectorPool 中读到 ByteBuffer 中）

```
private int fillReadBuffer(boolean block, ByteBuffer to) throws IOException {
    int nRead;
    NioChannel channel = getSocket();
    if (block) {
        Selector selector = null;
        try {
            selector = pool.get();
        } catch (IOException x) {
            // Ignore
        }
        try {
            NioEndpoint.NioSocketWrapper att = (NioEndpoint.NioSocketWrapper)
channel
                    .getAttachment();
            if (att == null) {
                throw new IOException("Key must be cancelled.");
            }
            nRead = pool.read(to, channel, selector, att.getReadTimeout());
        } finally {
            if (selector != null) {
                pool.put(selector);
            }
        }
    } else {
        nRead = channel.read(to);
        if (nRead == -1) {
            throw new EOFException();
        }
    }
    return nRead;
}
```

3.1.1.3.1.3) Http11inputBuffer#parseHeaders（解析请求头）

读取 byteBuffer，解析，将结果存入 Request

3.1.1.3.1.4) prepareRequest（封装 InputFilter）

```
private void prepareRequest() {

    http11 = true;
    http09 = false;
    contentDelimitation = false;

    if (protocol.isSSLEnabled()) {
        request.scheme().setString("https");
    }
    MessageBytes protocolMB = request.protocol();
    if (protocolMB.equals(Constants.HTTP_11)) {
        http11 = true;
        protocolMB.setString(Constants.HTTP_11);
    } else if (protocolMB.equals(Constants.HTTP_10)) {
        http11 = false;
        keepAlive = false;
        protocolMB.setString(Constants.HTTP_10);
    } else if (protocolMB.equals("")) {
        // HTTP/0.9
        http09 = true;
        http11 = false;
        keepAlive = false;
    } else {
        // Unsupported protocol
        http11 = false;
        // Send 505; Unsupported HTTP version
        response.setStatus(505);
        setErrorState(ErrorState.CLOSE_CLEAN, null);
        if (log.isDebugEnabled()) {
            log.debug(sm.getString("http11processor.request.prepare")+
                      " Unsupported HTTP version \""+protocolMB+"\"");
        }
    }
```

```
MimeHeaders headers = request.getMimeHeaders();

// Check connection header
MessageBytes connectionValueMB = headers.getValue(Constants.CONNECTION);
if (connectionValueMB != null) {
    ByteChunk connectionValueBC = connectionValueMB.getByteChunk();
    if (findBytes(connectionValueBC, Constants.CLOSE_BYTES) != -1) {
        keepAlive = false;
    } else if (findBytes(connectionValueBC,
                         Constants.KEEPALIVE_BYTES) != -1) {
        keepAlive = true;
    }
}

if (http11) {
    MessageBytes expectMB = headers.getValue("expect");
    if (expectMB != null) {
        if (expectMB.indexOfIgnoreCase("100-continue", 0) != -1) {
            inputBuffer.setSwallowInput(false);
            request.setExpectation(true);
        } else {

response.setStatus(HttpServletResponse.SC_EXPECTATION_FAILED);
            setErrorState(ErrorState.CLOSE_CLEAN, null);
        }
    }
}

// Check user-agent header
Pattern restrictedUserAgents = protocol.getRestrictedUserAgentsPattern();
if (restrictedUserAgents != null && (http11 || keepAlive)) {
    MessageBytes userAgentValueMB = headers.getValue("user-agent");
    // Check in the restricted list, and adjust the http11
    // and keepAlive flags accordingly
    if(userAgentValueMB != null) {
        String userAgentValue = userAgentValueMB.toString();
        if (restrictedUserAgents.matcher(userAgentValue).matches()) {
            http11 = false;
            keepAlive = false;
        }
    }
}
```

```java
// Check host header
MessageBytes hostValueMB = null;
try {
    hostValueMB = headers.getUniqueValue("host");
} catch (IllegalArgumentException iae) {
    // Multiple Host headers are not permitted
    // 400 - Bad request
    response.setStatus(400);
    setErrorState(ErrorState.CLOSE_CLEAN, null);
    if (log.isDebugEnabled()) {
        log.debug(sm.getString("http11processor.request.multipleHosts"));
    }
}
if (http11 && hostValueMB == null) {
    // 400 - Bad request
    response.setStatus(400);
    setErrorState(ErrorState.CLOSE_CLEAN, null);
    if (log.isDebugEnabled()) {
        log.debug(sm.getString("http11processor.request.prepare")+
                  " host header missing");
    }
}

// Check for a full URI (including protocol://host:port/)
ByteChunk uriBC = request.requestURI().getByteChunk();
if (uriBC.startsWithIgnoreCase("http", 0)) {

    int pos = uriBC.indexOf("://", 0, 3, 4);
    int uriBCStart = uriBC.getStart();
    int slashPos = -1;
    if (pos != -1) {
        pos += 3;
        byte[] uriB = uriBC.getBytes();
        slashPos = uriBC.indexOf('/', pos);
        int atPos = uriBC.indexOf('@', pos);
        if (slashPos == -1) {
            slashPos = uriBC.getLength();
            // Set URI as "/"
            request.requestURI().setBytes
                (uriB, uriBCStart + pos - 2, 1);
        } else {
            request.requestURI().setBytes
                (uriB, uriBCStart + slashPos,
                     uriBC.getLength() - slashPos);
            }
            // Skip any user info
            if (atPos != -1) {
                pos = atPos + 1;
            }
            if (http11) {
                // Missing host header is illegal but handled above
                if (hostValueMB != null) {
                    // Any host in the request line must be consistent with
                    // the Host header
                    if (!hostValueMB.getByteChunk().equals(
                            uriB, uriBCStart + pos, slashPos - pos)) {
                        if (protocol.getAllowHostHeaderMismatch()) {
                            // The requirements of RFC 2616 are being
                            // applied. If the host header and the request
                            // line do not agree, the request line takes
                            // precedence
                            hostValueMB = headers.setValue("host");
                            hostValueMB.setBytes(uriB, uriBCStart + pos,
slashPos - pos);
                        } else {
                            // The requirements of RFC 7230 are being
                            // applied. If the host header and the request
                            // line do not agree, trigger a 400 response.
                            response.setStatus(400);
                            setErrorState(ErrorState.CLOSE_CLEAN, null);
                            if (log.isDebugEnabled()) {

log.debug(sm.getString("http11processor.request.inconsistentHosts"));
                            }
                        }
                    }
                }
            } else {
                // Not HTTP/1.1 - no Host header so generate one since
                // Tomcat internals assume it is set
                hostValueMB = headers.setValue("host");
                hostValueMB.setBytes(uriB, uriBCStart + pos, slashPos - pos);
            }
        }
    }

    // Input filter setup
    InputFilter[] inputFilters = inputBuffer.getFilters();

    // Parse transfer-encoding header
    if (http11) {
        MessageBytes transferEncodingValueMB =
headers.getValue("transfer-encoding");
        if (transferEncodingValueMB != null) {
            String transferEncodingValue = transferEncodingValueMB.toString();
            // Parse the comma separated list. "identity" codings are ignored
            int startPos = 0;
            int commaPos = transferEncodingValue.indexOf(',');
            String encodingName = null;
            while (commaPos != -1) {
                encodingName = transferEncodingValue.substring(startPos,
commaPos);
                addInputFilter(inputFilters, encodingName);
                startPos = commaPos + 1;
                commaPos = transferEncodingValue.indexOf(',', startPos);
            }
            encodingName = transferEncodingValue.substring(startPos);
            addInputFilter(inputFilters, encodingName);
        }
    }

    // Parse content-length header
    long contentLength = request.getContentLengthLong();
    if (contentLength >= 0) {
        if (contentDelimitation) {
            // contentDelimitation being true at this point indicates that
            // chunked encoding is being used but chunked encoding should
            // not be used with a content length. RFC 2616, section 4.4,
            // bullet 3 states Content-Length must be ignored in this case -
            // so remove it.
            headers.removeHeader("content-length");
            request.setContentLength(-1);
        } else {
            inputBuffer.addActiveFilter
                    (inputFilters[Constants.IDENTITY_FILTER]);
            contentDelimitation = true;
        }
    }

    parseHost(hostValueMB);
    if (!contentDelimitation) {
        // If there's no content length
        // (broken HTTP/1.0 or HTTP/1.1), assume
        // the client is not broken and didn't send a body
        inputBuffer.addActiveFilter
                (inputFilters[Constants.VOID_FILTER]);
        contentDelimitation = true;
    }

    if (getErrorState().isError()) {
        getAdapter().log(request, response, 0);
    }
}
```

3.1.1.3.1.4)（service 骨架） CoyoteAdapter#service（将 coyote 的 req 和 resp 转为 catalina 的 req 和 resp）

```
public void service(org.apache.coyote.Request req, org.apache.coyote.Response
res)
        throws Exception {

    Request request = (Request) req.getNote(ADAPTER_NOTES);
    Response response = (Response) res.getNote(ADAPTER_NOTES);

    if (request == null) {
        // Create objects
        request = connector.createRequest();
        request.setCoyoteRequest(req);
        response = connector.createResponse();
        response.setCoyoteResponse(res);

        // Link objects
        request.setResponse(response);
        response.setRequest(request);

        // Set as notes
        req.setNote(ADAPTER_NOTES, request);
        res.setNote(ADAPTER_NOTES, response);

        // Set query string encoding
        req.getParameters().setQueryStringCharset(connector.getURICharset());
    }
```

```java
    if (connector.getXpoweredBy()) {
        response.addHeader("X-Powered-By", POWERED_BY);
    }

    boolean async = false;
    boolean postParseSuccess = false;

    req.getRequestProcessor().setWorkerThreadName(THREAD_NAME.get());

    try {
        // Parse and set Catalina and configuration specific
        // request parameters
        postParseSuccess = postParseRequest(req, request, res, response);
        if (postParseSuccess) {
            //check valves if we support async
            request.setAsyncSupported(

connector.getService().getContainer().getPipeline().isAsyncSupported());
            // Calling the container
           // 加入到 pipeline 中进行调用
connector.getService().getContainer().getPipeline().getFirst().invoke(
                    request, response);
        }
        if (request.isAsync()) {
            async = true;
            ReadListener readListener = req.getReadListener();
            if (readListener != null && request.isFinished()) {
                // Possible the all data may have been read during service()
                // method so this needs to be checked here
                ClassLoader oldCL = null;
                try {
                    oldCL = request.getContext().bind(false, null);
                    if (req.sendAllDataReadEvent()) {
                        req.getReadListener().onAllDataRead();
                    }
                } finally {
                    request.getContext().unbind(false, oldCL);
                }
            }

            Throwable throwable =
                    (Throwable)
request.getAttribute(RequestDispatcher.ERROR_EXCEPTION);
```

```
            // If an async request was started, is not going to end once
            // this container thread finishes and an error occurred, trigger
            // the async error process
            if (!request.isAsyncCompleting() && throwable != null) {
                request.getAsyncContextInternal().setErrorState(throwable,
true);
            }
        } else {
            request.finishRequest();
            response.finishResponse();
        }

    } catch (IOException e) {
        // Ignore
    } finally {
        AtomicBoolean error = new AtomicBoolean(false);
        res.action(ActionCode.IS_ERROR, error);

        if (request.isAsyncCompleting() && error.get()) {
            // Connection will be forcibly closed which will prevent
            // completion happening at the usual point. Need to trigger
            // call to onComplete() here.
            res.action(ActionCode.ASYNC_POST_PROCESS,  null);
            async = false;
        }

        // Access log
        if (!async && postParseSuccess) {
            // Log only if processing was invoked.
            // If postParseRequest() failed, it has already logged it.
            Context context = request.getContext();
            // If the context is null, it is likely that the endpoint was
            // shutdown, this connection closed and the request recycled in
            // a different thread. That thread will have updated the access
            // log so it is OK not to update the access log here in that
            // case.
            if (context != null) {
                context.logAccess(request, response,
                        System.currentTimeMillis() - req.getStartTime(), false);
            }
        }

        req.getRequestProcessor().setWorkerThreadName(null);
        // Recycle the wrapper request and response
        if (!async) {
            request.recycle();
            response.recycle();
        }
    }
}
```

3.1.1.3.1.4.1)（Mapper#map） CoyoteAdapter#postParseRequest（req 和 resp 的转换）

postParseRequest() 方法封装一下 Request，并处理一下映射关系(从 URL 映射到相应的 Host、Context、Wrapper)。
CoyoteAdapter 将 Rquest 提交给 Container 处理之前，并将 org.apache.coyote.Request 封装到 org.apache.catalina.connector.Request，传递给 Container 处理的 Request 是 org.apache.catalina.connector.Request。
connector.getService().getMapper().map()，用来在 Mapper 中查询 URL 的映射关系。映射关系会保留到 org.apache.catalina.connector.Request 中，Container 处理阶段 request.getHost() 是使用的就是这个阶段查询到的映射主机，以此类推 request.getContext()、request.getWrapper() 都是。

```
protected boolean postParseRequest(org.apache.coyote.Request req, Request
request,
        org.apache.coyote.Response res, Response response) throws IOException,
ServletException {

    // If the processor has set the scheme (AJP does this, HTTP does this if
    // SSL is enabled) use this to set the secure flag as well. If the
    // processor hasn't set it, use the settings from the connector
    if (req.scheme().isNull()) {
        // Use connector scheme and secure configuration, (defaults to
        // "http" and false respectively)
        req.scheme().setString(connector.getScheme());
        request.setSecure(connector.getSecure());
    } else {
        // Use processor specified scheme to determine secure state
        request.setSecure(req.scheme().equals("https"));
    }

    // At this point the Host header has been processed.
    // Override if the proxyPort/proxyHost are set
    String proxyName = connector.getProxyName();
    int proxyPort = connector.getProxyPort();
    if (proxyPort != 0) {
        req.setServerPort(proxyPort);
    } else if (req.getServerPort() == -1) {
        // Not explicitly set. Use default ports based on the scheme
        if (req.scheme().equals("https")) {
            req.setServerPort(443);
        } else {
            req.setServerPort(80);
        }
    }
    if (proxyName != null) {
        req.serverName().setString(proxyName);
    }

    MessageBytes undecodedURI = req.requestURI();

    // Check for ping OPTIONS * request
    if (undecodedURI.equals("*")) {
        if (req.method().equalsIgnoreCase("OPTIONS")) {
            StringBuilder allow = new StringBuilder();
            allow.append("GET, HEAD, POST, PUT, DELETE, OPTIONS");
            // Trace if allowed
            if (connector.getAllowTrace()) {
                allow.append(", TRACE");
            }
            // Always allow options
            res.setHeader("Allow", allow.toString());
            // Access log entry as processing won't reach AccessLogValve
            connector.getService().getContainer().logAccess(request, response,
0, true);
            return false;
        } else {
            response.sendError(400, "Invalid URI");
        }
    }

    MessageBytes decodedURI = req.decodedURI();

    if (undecodedURI.getType() == MessageBytes.T_BYTES) {
        // Copy the raw URI to the decodedURI
        decodedURI.duplicate(undecodedURI);

        // Parse the path parameters. This will:
        //   - strip out the path parameters
        //   - convert the decodedURI to bytes
```

```java
        parsePathParameters(req, request);

        // URI decoding
        // %xx decoding of the URL
        try {
            req.getURLDecoder().convert(decodedURI, false);
        } catch (IOException ioe) {
            response.sendError(400, "Invalid URI: " + ioe.getMessage());
        }
        // Normalization
        if (!normalize(req.decodedURI())) {
            response.sendError(400, "Invalid URI");
        }
        // Character decoding
        convertURI(decodedURI, request);
        // Check that the URI is still normalized
        if (!checkNormalize(req.decodedURI())) {
            response.sendError(400, "Invalid URI");
        }
    } else {
        /* The URI is chars or String, and has been sent using an in-memory
         * protocol handler. The following assumptions are made:
         * - req.requestURI() has been set to the 'original' non-decoded,
         *   non-normalized URI
         * - req.decodedURI() has been set to the decoded, normalized form
         *   of req.requestURI()
         */
        decodedURI.toChars();
        // Remove all path parameters; any needed path parameter should be set
        // using the request object rather than passing it in the URL
        CharChunk uriCC = decodedURI.getCharChunk();
        int semicolon = uriCC.indexOf(';');
        if (semicolon > 0) {
            decodedURI.setChars(uriCC.getBuffer(), uriCC.getStart(),
semicolon);
        }
    }

    // Request mapping.
    MessageBytes serverName;
    if (connector.getUseIPVHosts()) {
        serverName = req.localName();
        if (serverName.isNull()) {
            // well, they did ask for it
            res.action(ActionCode.REQ_LOCAL_NAME_ATTRIBUTE, null);
        }
    } else {
        serverName = req.serverName();
    }

    // Version for the second mapping loop and
    // Context that we expect to get for that version
    String version = null;
    Context versionContext = null;
    boolean mapRequired = true;

    if (response.isError()) {
        // An error this early means the URI is invalid. Ensure invalid data
        // is not passed to the mapper. Note we still want the mapper to
        // find the correct host.
        decodedURI.recycle();
    }

    while (mapRequired) {
        // 使用 Mapper 将当前 request 映射到 Host、Context、Wrapper
        // This will map the the latest version by default
        connector.getService().getMapper().map(serverName, decodedURI,
                version, request.getMappingData());

        // If there is no context at this point, either this is a 404
        // because no ROOT context has been deployed or the URI was invalid
        // so no context could be mapped.
        if (request.getContext() == null) {
            // Don't overwrite an existing error
            if (!response.isError()) {
                response.sendError(404, "Not found");
            }
            // Allow processing to continue.
            // If present, the error reporting valve will provide a response
            // body.
            return true;
        }

        // Now we have the context, we can parse the session ID from the URL
        // (if any). Need to do this before we redirect in case we need to
        // include the session id in the redirect
        String sessionID;
        if (request.getServletContext().getEffectiveSessionTrackingModes()
        .contains(SessionTrackingMode.URL)) {

    // Get the session ID if there was one
    sessionID = request.getPathParameter(
            SessionConfig.getSessionUriParamName(
                    request.getContext()));
    if (sessionID != null) {
        request.setRequestedSessionId(sessionID);
        request.setRequestedSessionURL(true);
    }
}

// Look for session ID in cookies and SSL session
parseSessionCookiesId(request);
parseSessionSslId(request);

sessionID = request.getRequestedSessionId();

mapRequired = false;
if (version != null && request.getContext() == versionContext) {
    // We got the version that we asked for. That is it.
} else {
    version = null;
    versionContext = null;

    Context[] contexts = request.getMappingData().contexts;
    // Single contextVersion means no need to remap
    // No session ID means no possibility of remap
    if (contexts != null && sessionID != null) {
        // Find the context associated with the session
        for (int i = contexts.length; i > 0; i--) {
            Context ctxt = contexts[i - 1];
            if (ctxt.getManager().findSession(sessionID) != null) {
                // We found a context. Is it the one that has
                // already been mapped?
                if (!ctxt.equals(request.getMappingData().context)) {
                    // Set version so second time through mapping
                    // the correct context is found
                    version = ctxt.getWebappVersion();
                    versionContext = ctxt;
                    // Reset mapping
                    request.getMappingData().recycle();
                    mapRequired = true;
                    // Recycle cookies and session info in case the
```

```
                        // correct context is configured with different
                        // settings
                        request.recycleSessionInfo();
                        request.recycleCookieInfo(true);
                    }
                    break;
                }
            }
        }
    }

    if (!mapRequired && request.getContext().getPaused()) {
        // Found a matching context but it is paused. Mapping data will
        // be wrong since some Wrappers may not be registered at this
        // point.
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // Should never happen
        }
        // Reset mapping
        request.getMappingData().recycle();
        mapRequired = true;
    }
}

// Possible redirect
MessageBytes redirectPathMB = request.getMappingData().redirectPath;
if (!redirectPathMB.isNull()) {
    String redirectPath = URLEncoder.DEFAULT.encode(
            redirectPathMB.toString(), StandardCharsets.UTF_8);
    String query = request.getQueryString();
    if (request.isRequestedSessionIdFromURL()) {
        // This is not optimal, but as this is not very common, it
        // shouldn't matter
        redirectPath = redirectPath + ";" +
                SessionConfig.getSessionUriParamName(
                    request.getContext()) +
            "=" + request.getRequestedSessionId();
    }
    if (query != null) {
        // This is not optimal, but as this is not very common, it
        // shouldn't matter
        redirectPath = redirectPath + "?" + query;
        }
        response.sendRedirect(redirectPath);
        request.getContext().logAccess(request, response, 0, true);
        return false;
    }

    // Filter trace method
    if (!connector.getAllowTrace()
            && req.method().equalsIgnoreCase("TRACE")) {
        Wrapper wrapper = request.getWrapper();
        String header = null;
        if (wrapper != null) {
            String[] methods = wrapper.getServletMethods();
            if (methods != null) {
                for (int i=0; i<methods.length; i++) {
                    if ("TRACE".equals(methods[i])) {
                        continue;
                    }
                    if (header == null) {
                        header = methods[i];
                    } else {
                        header += ", " + methods[i];
                    }
                }
            }
        }
        res.addHeader("Allow", header);
        response.sendError(405, "TRACE method is not allowed");
        // Safe to skip the remainder of this method.
        return true;
    }

    doConnectorAuthenticationAuthorization(req, request);

    return true;
}
```

3.1.1.3.1.4.2) （->4)）Valve#invoke

```
public void invoke(Request request, Response response)
    throws IOException, ServletException;
```

connector.getService().getContainer().getPipeline().getFirst().invoke() 会将请求传递到 Container 处理，当然了 Container 处理也是在 Worker 线程中执行的，但是这是一个相对

独立的模块，所以单独分出来一节。

第一个 Container#Valve 是 StandardEngineValve。
按照这样的顺序：engine->host->context->wrapper。

3.1.1.3.1.4.3) Request#finishRequest（非异步 Servlet 被调用）

```
public void finishRequest() throws IOException {
    if (response.getStatus() ==
HttpServletResponse.SC_REQUEST_ENTITY_TOO_LARGE) {
        checkSwallowInput();
    }
}
```

3.1.1.3.1.4.4) Response#finishResponse（非异步 Servlet 被调用）

```
public void finishResponse() throws IOException {
    // Writing leftover bytes
    outputBuffer.close();
}
```

3.1.1.3.1.4.5) Request#recycle（非异步 Servlet 被调用，释放资源，待被复用）

```
/**
 * Release all object references, and initialize instance variables, in
 * preparation for reuse of this object.
 */
public void recycle() {

    internalDispatcherType = null;
    requestDispatcherPath = null;

    authType = null;
    inputBuffer.recycle();
    usingInputStream = false;
    usingReader = false;
    userPrincipal = null;
    subject = null;
    parametersParsed = false;
    if (parts != null) {
        for (Part part: parts) {
            try {
                part.delete();
            } catch (IOException ignored) {
                // ApplicationPart.delete() never throws an IOEx
            }
        }
        parts = null;
    }
    partsParseException = null;
    locales.clear();
    localesParsed = false;
    secure = false;
    remoteAddr = null;
    remoteHost = null;
    remotePort = -1;
    localPort = -1;
    localAddr = null;
    localName = null;

    attributes.clear();
    sslAttributesParsed = false;
    notes.clear();

    recycleSessionInfo();
    recycleCookieInfo(false);

    if (Globals.IS_SECURITY_ENABLED || Connector.RECYCLE_FACADES) {
        parameterMap = new ParameterMap<>();
    } else {
        parameterMap.setLocked(false);
        parameterMap.clear();
    }

    mappingData.recycle();
    applicationMapping.recycle();

    applicationRequest = null;
    if (Globals.IS_SECURITY_ENABLED || Connector.RECYCLE_FACADES) {
        if (facade != null) {
            facade.clear();
            facade = null;
        }
        if (inputStream != null) {
            inputStream.clear();
            inputStream = null;
        }
        if (reader != null) {
            reader.clear();
```

```
            reader = null;
        }
    }

    asyncSupported = null;
    if (asyncContext!=null) {
        asyncContext.recycle();
    }
    asyncContext = null;
}
```

3.1.1.3.1.4.6) Response#recycle（非异步 Servlet 被调用，释放资源，待被复用）

```
/**
 * Release all object references, and initialize instance variables, in
 * preparation for reuse of this object.
 */
public void recycle() {

    cookies.clear();
    outputBuffer.recycle();
    usingOutputStream = false;
    usingWriter = false;
    appCommitted = false;
    included = false;
    isCharacterEncodingSet = false;

    applicationResponse = null;
    if (Globals.IS_SECURITY_ENABLED || Connector.RECYCLE_FACADES) {
        if (facade != null) {
            facade.clear();
            facade = null;
        }
        if (outputStream != null) {
            outputStream.clear();
            outputStream = null;
        }
        if (writer != null) {
            writer.clear();
            writer = null;
        }
    } else if (writer != null) {
        writer.recycle();
    }
}
```

3.1.1.3.1.5) endRequest（非异步 Servlet 被调用）

```
/*
 * No more input will be passed to the application. Remaining input will be
 * swallowed or the connection dropped depending on the error and
 * expectation status.
 */
private void endRequest() {
    if (getErrorState().isError()) {
        // If we know we are closing the connection, don't drain
        // input. This way uploading a 100GB file doesn't tie up the
        // thread if the servlet has rejected it.
        inputBuffer.setSwallowInput(false);
    } else {
        // Need to check this again here in case the response was
        // committed before the error that requires the connection
        // to be closed occurred.
        checkExpectationAndResponseStatus();
    }

    // Finish the handling of the request
    if (getErrorState().isIoAllowed()) {
        try {
            inputBuffer.endRequest();
        } catch (IOException e) {
            setErrorState(ErrorState.CLOSE_CONNECTION_NOW, e);
        } catch (Throwable t) {
            ExceptionUtils.handleThrowable(t);
            // 500 - Internal Server Error
            // Can't add a 500 to the access log since that has already been
            // written in the Adapter.service method.
            response.setStatus(500);
            setErrorState(ErrorState.CLOSE_NOW, t);
            log.error(sm.getString("http11processor.request.finish"), t);
        }
    }
    if (getErrorState().isIoAllowed()) {
        try {
            action(ActionCode.COMMIT, null);
            outputBuffer.end();
        } catch (IOException e) {
            setErrorState(ErrorState.CLOSE_CONNECTION_NOW, e);
        } catch (Throwable t) {
            ExceptionUtils.handleThrowable(t);
            setErrorState(ErrorState.CLOSE_NOW, t);
            log.error(sm.getString("http11processor.response.finish"), t);
        }
    }
}
```

3.1.1.3.1.5.1) Http11InputBuffer#endRequest

```
void endRequest() throws IOException {

    if (swallowInput && (lastActiveFilter != -1)) {
        int extraBytes = (int) activeFilters[lastActiveFilter].end();
        byteBuffer.position(byteBuffer.position() - extraBytes);
    }
}
```

3.1.1.3.1.5.2) AbstractProcessor#action(COMMIT)

```
case COMMIT: {
    if (!response.isCommitted()) {
        try {
            // Validate and write response headers
            prepareResponse();
        } catch (IOException e) {
            setErrorState(ErrorState.CLOSE_CONNECTION_NOW, e);
        }
    }
    break;
}
```

Http11Processor#prepareResponse

```
protected final void prepareResponse() throws IOException {

    boolean entityBody = true;
    contentDelimitation = false;

    OutputFilter[] outputFilters = outputBuffer.getFilters();

    if (http09 == true) {
        // HTTP/0.9
outputBuffer.addActiveFilter(outputFilters[Constants.IDENTITY_FILTER]);
        outputBuffer.commit();
        return;
    }

    int statusCode = response.getStatus();
    if (statusCode < 200 || statusCode == 204 || statusCode == 205 ||
            statusCode == 304) {
        // No entity body
        outputBuffer.addActiveFilter
            (outputFilters[Constants.VOID_FILTER]);
        entityBody = false;
        contentDelimitation = true;
        if (statusCode == 205) {
            // RFC 7231 requires the server to explicitly signal an empty
            // response in this case
            response.setContentLength(0);
        } else {
            response.setContentLength(-1);
        }
    }

    MessageBytes methodMB = request.method();
    if (methodMB.equals("HEAD")) {
        // No entity body
        outputBuffer.addActiveFilter
            (outputFilters[Constants.VOID_FILTER]);
        contentDelimitation = true;
    }

    // Sendfile support
    if (protocol.getUseSendfile()) {
        prepareSendfile(outputFilters);
    }

    // Check for compression

    boolean useCompression = false;
    if (entityBody && sendfileData == null) {
        useCompression = protocol.useCompression(request, response);
    }

    MimeHeaders headers = response.getMimeHeaders();
```

```
    // A SC_NO_CONTENT response may include entity headers
    if (entityBody || statusCode == HttpServletResponse.SC_NO_CONTENT) {
        String contentType = response.getContentType();
        if (contentType != null) {
            headers.setValue("Content-Type").setString(contentType);
        }
        String contentLanguage = response.getContentLanguage();
        if (contentLanguage != null) {
            headers.setValue("Content-Language")
                .setString(contentLanguage);
        }
    }

    long contentLength = response.getContentLengthLong();
    boolean connectionClosePresent = false;
    if (http11 && response.getTrailerFields() != null) {
        // If trailer fields are set, always use chunking

outputBuffer.addActiveFilter(outputFilters[Constants.CHUNKED_FILTER]);
        contentDelimitation = true;

headers.addValue(Constants.TRANSFERENCODING).setString(Constants.CHUNKED);
    } else if (contentLength != -1) {
        headers.setValue("Content-Length").setLong(contentLength);

outputBuffer.addActiveFilter(outputFilters[Constants.IDENTITY_FILTER]);
        contentDelimitation = true;
    } else {
        // If the response code supports an entity body and we're on
        // HTTP 1.1 then we chunk unless we have a Connection: close header
        connectionClosePresent = isConnectionClose(headers);
        if (http11 && entityBody && !connectionClosePresent) {

outputBuffer.addActiveFilter(outputFilters[Constants.CHUNKED_FILTER]);
            contentDelimitation = true;

headers.addValue(Constants.TRANSFERENCODING).setString(Constants.CHUNKED);
        } else {

outputBuffer.addActiveFilter(outputFilters[Constants.IDENTITY_FILTER]);
        }
    }

    if (useCompression) {
        outputBuffer.addActiveFilter(outputFilters[Constants.GZIP_FILTER]);
    }

    // Add date header unless application has already set one (e.g. in a
    // Caching Filter)
    if (headers.getValue("Date") == null) {
        headers.addValue("Date").setString(
                FastHttpDateFormat.getCurrentDate());
    }

    // FIXME: Add transfer encoding header

    if ((entityBody) && (!contentDelimitation)) {
        // Mark as close the connection after the request, and add the
        // connection: close header
        keepAlive = false;
    }

    // This may disabled keep-alive to check before working out the
    // Connection header.
    checkExpectationAndResponseStatus();

    // If we know that the request is bad this early, add the
    // Connection: close header.
    if (keepAlive && statusDropsConnection(statusCode)) {
        keepAlive = false;
    }
    if (!keepAlive) {
        // Avoid adding the close header twice
        if (!connectionClosePresent) {
            headers.addValue(Constants.CONNECTION).setString(
                    Constants.CLOSE);
        }
    } else if (!http11 && !getErrorState().isError()) {

headers.addValue(Constants.CONNECTION).setString(Constants.KEEPALIVE);
    }

    // Add server header
    String server = protocol.getServer();
    if (server == null) {
        if (protocol.getServerRemoveAppProvidedValues()) {
            headers.removeHeader("server");
        }
    } else {
        // server always overrides anything the app might set
        headers.setValue("Server").setString(server);
    }

    // Build the response header
    try {
        outputBuffer.sendStatus();

        int size = headers.size();
        for (int i = 0; i < size; i++) {
            outputBuffer.sendHeader(headers.getName(i), headers.getValue(i));
        }
        outputBuffer.endHeaders();
    } catch (Throwable t) {
        ExceptionUtils.handleThrowable(t);
        // If something goes wrong, reset the header buffer so the error
        // response can be written instead.
        outputBuffer.resetHeaderBuffer();
        throw t;
    }

    outputBuffer.commit();
}
```

Http11OutputBuffer#commit

```
protected void commit() throws IOException {
    response.setCommitted(true);

    if (headerBuffer.position() > 0) {
        // Sending the response header buffer
        headerBuffer.flip();
        try {
            socketWrapper.write(isBlocking(), headerBuffer);
        } finally {
            headerBuffer.position(0).limit(headerBuffer.capacity());
        }
    }
}
```

3.1,1,3,1,5,3) Http11OutputBuffer#end

```
public void end() throws IOException {
    if (responseFinished) {
        return;
    }

    if (lastActiveFilter == -1) {
        outputStreamOutputBuffer.end();
    } else {
        activeFilters[lastActiveFilter].end();
    }

    responseFinished = true;
}
```

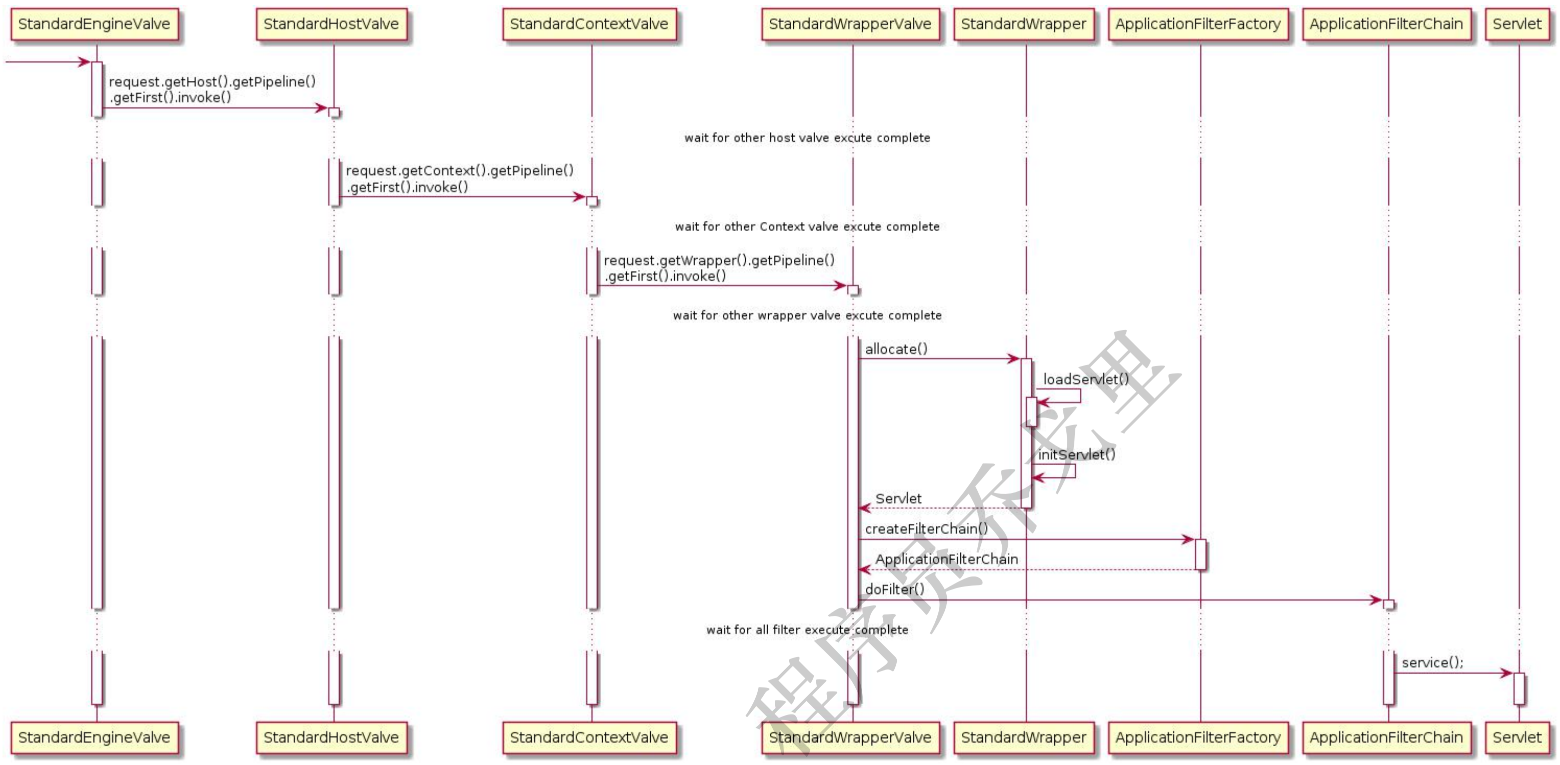
3.1.1.3.2) asyncPostProcess（异步 Servlet）

```
public SocketState asyncPostProcess() {
    return asyncStateMachine.asyncPostProcess();
}
```

3.1.1.3.2.1) AsyncStateMachine#asyncPostProcess

```
synchronized SocketState asyncPostProcess() {
    if (state == AsyncState.COMPLETE_PENDING) {
        doComplete();
        return SocketState.ASYNC_END;
    } else if (state == AsyncState.DISPATCH_PENDING) {
        doDispatch();
        return SocketState.ASYNC_END;
    } else  if (state == AsyncState.STARTING || state ==
AsyncState.READ_WRITE_OP) {
        state = AsyncState.STARTED;
        return SocketState.LONG;
    } else if (state == AsyncState.MUST_COMPLETE || state ==
AsyncState.COMPLETING) {
        asyncCtxt.fireOnComplete();
        state = AsyncState.DISPATCHED;
        return SocketState.ASYNC_END;
    } else if (state == AsyncState.MUST_DISPATCH) {
        state = AsyncState.DISPATCHING;
        return SocketState.ASYNC_END;
    } else if (state == AsyncState.DISPATCHING) {
        state = AsyncState.DISPATCHED;
        return SocketState.ASYNC_END;
    } else if (state == AsyncState.STARTED) {
        // This can occur if an async listener does a dispatch to an async
        // servlet during onTimeout
        return SocketState.LONG;
    } else {
        throw new IllegalStateException(
                sm.getString("asyncStateMachine.invalidAsyncState",
                        "asyncPostProcess()", state));
    }
}
```

4) Container#Valve#invoke（在 Worker 线程池中执行）



1. 需要注意的是，基本上每一个容器的 StandardPipeline 上都会有多个已注册的 Valve，我们只关注每个容器的 Basic Valve。其他 Valve 都是在 Basic Valve 前执行。
2. request.getHost().getPipeline().getFirst().invoke() 先获取对应的 StandardHost，并执行其 pipeline。
3. request.getContext().getPipeline().getFirst().invoke() 先获取对应的 StandardContext,并执行其 pipeline。
4. request.getWrapper().getPipeline().getFirst().invoke() 先获取对应的 StandardWrapper，并执行其 pipeline。
5. 最值得说的就是 StandardWrapper 的 Basic Valve，StandardWrapperValve
6. allocate() 用来加载并初始化 Servlet，值的一提的是 Servlet 并不都是单例的，当 Servlet 实现了 SingleThreadModel 接口后，StandardWrapper 会维护一组 Servlet 实例，这是享元模式。当然了 SingleThreadModel 在 Servlet 2.4 以后就弃用了。
7. createFilterChain() 方法会从 StandardContext 中获取到所有的过滤器，然后将匹配 Request URL 的所有过滤器挑选出来添加到 filterChain 中。
8. doFilter() 执行过滤链,当所有的过滤器都执行完毕后调用 Servlet 的 service() 方法。

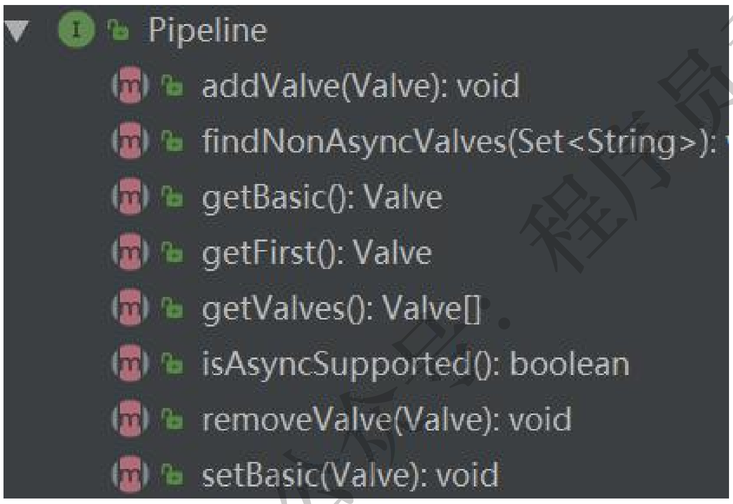
第一个 Container#Valve 是 StandardEngineValve。
按照这样的顺序：engine->host->context->wrapper。
这四个容器都继承自 ContainerBase。

ContainerBase

```
public abstract class ContainerBase extends LifecycleMBeanBase
        implements Container {
/**
 * The Pipeline object with which this Container is associated.
 */
protected final Pipeline pipeline = new StandardPipeline(this);
}
```

持有一个 StandardPipeline 对象。

Pipeline（一个 pipeline 只能与一个 Container 关联，多对一）



```
Pipeline
    addValve(Valve): void
    findNonAsyncValves(Set<String>):
    getBasic(): Valve
    getFirst(): Valve
    getValves(): Valve[]
    isAsyncSupported(): boolean
    removeValve(Valve): void
    setBasic(Valve): void
```

StandardPipeline 组件代表一个流水线，与 Valve（阀）结合，用于处理请求。StandardPipeline 中含有多个 Valve，当需要处理请求时，会逐一调用 Valve 的 invoke 方法对 Request 和 Response 进行处理。特别的，其中有一个特殊的 Valve 叫 basicValve，每一个标准容器都有一个指定的 BasicValve，他们做的是最核心的工作。

```
public class StandardPipeline extends LifecycleBase implements Pipeline {

    private static final Log log = LogFactory.getLog(StandardPipeline.class);

    // ------------------------------------------------------------
Constructors


    /**
     * Construct a new StandardPipeline instance with no associated Container.
     */
    public StandardPipeline() {
        this(null);

    }


    /**
     * Construct a new StandardPipeline instance that is associated with the
     * specified Container.
     *
     * @param container The container we should be associated with
     */
    public StandardPipeline(Container container) {

        super();
        setContainer(container);

    }


    // ----------------------------------------------------- Instance
Variables


    /**
     * The basic Valve (if any) associated with this Pipeline.
     */
    protected Valve basic = null;


    /**
     * The Container with which this Pipeline is associated.
     */
    protected Container container = null;


    /**
     * The first valve associated with this Pipeline.
     */
    protected Valve first = null;
}
```

Valve（一个 pipeline 对应着多个 Valve，一对多，链表结构）

Valve 是一个接口，其基本实现的 BaseValve 类。

```
public abstract class ValveBase extends LifecycleMBeanBase implements Contained,
Valve {

    protected static final StringManager sm =
StringManager.getManager(ValveBase.class);


    //------------------------------------------------------ Constructor

    public ValveBase() {
        this(false);
    }


    public ValveBase(boolean asyncSupported) {
        this.asyncSupported = asyncSupported;
    }


    //------------------------------------------------------ Instance
Variables

    /**
     * Does this valve support Servlet 3+ async requests?
     */
    protected boolean asyncSupported;


    /**
     * The Container whose pipeline this Valve is a component of.
     */
    protected Container container = null;


    /**
     * Container log
     */
    protected Log containerLog = null;
    /**
     * The next Valve in the pipeline this Valve is a component of.
     */
    protected Valve next = null;
}
```

4.1) StandardEngineValve#invoke

```
public final void invoke(Request request, Response response)
    throws IOException, ServletException {

    // Select the Host to be used for this Request
    Host host = request.getHost();
    if (host == null) {
        response.sendError
            (HttpServletResponse.SC_BAD_REQUEST,
             sm.getString("standardEngine.noHost",
                          request.getServerName()));
        return;
    }
    if (request.isAsyncSupported()) {
        request.setAsyncSupported(host.getPipeline().isAsyncSupported());
    }

    // Ask this Host to process this request
    host.getPipeline().getFirst().invoke(request, response);

}
```

4.1.1) StandardHostValve#invoke

```
public final void invoke(Request request, Response response)
    throws IOException, ServletException {

    // Select the Context to be used for this Request
    Context context = request.getContext();
    if (context == null) {
        return;
    }
    if (request.isAsyncSupported()) {
        request.setAsyncSupported(context.getPipeline().isAsyncSupported());
    }

    boolean asyncAtStart = request.isAsync();

    try {
        context.bind(Globals.IS_SECURITY_ENABLED, MY_CLASSLOADER);

        if (!asyncAtStart
&& !context.fireRequestInitEvent(request.getRequest())) {
            // Don't fire listeners during async processing (the listener
            // fired for the request that called startAsync()).
            // If a request init listener throws an exception, the request
            // is aborted.
            return;
        }

        // Ask this Context to process this request. Requests that are in
        // async mode and are not being dispatched to this resource must be
        // in error and have been routed here to check for application
        // defined error pages.
        try {
            if (!response.isErrorReportRequired()) {
                context.getPipeline().getFirst().invoke(request, response);
            }
        } catch (Throwable t) {
            ExceptionUtils.handleThrowable(t);
            container.getLogger().error("Exception Processing " +
request.getRequestURI(), t);
            // If a new error occurred while trying to report a previous
            // error allow the original error to be reported.
            if (!response.isErrorReportRequired()) {
                request.setAttribute(RequestDispatcher.ERROR_EXCEPTION, t);
                throwable(request, response, t);
            }
        }

        // Now that the request/response pair is back under container
        // control lift the suspension so that the error handling can
        // complete and/or the container can flush any remaining data
        response.setSuspended(false);

        Throwable t = (Throwable)
request.getAttribute(RequestDispatcher.ERROR_EXCEPTION);

        // Protect against NPEs if the context was destroyed during a
        // long running request.
        if (!context.getState().isAvailable()) {
            return;
        }

        // Look for (and render if found) an application level error page
        if (response.isErrorReportRequired()) {
            if (t != null) {
                throwable(request, response, t);
            } else {
                status(request, response);
            }
        }

        if (!request.isAsync() && !asyncAtStart) {
            context.fireRequestDestroyEvent(request.getRequest());
        }
    } finally {
        // Access a session (if present) to update last accessed time, based
        // on a strict interpretation of the specification
        if (ACCESS_SESSION) {
            request.getSession(false);
        }

        context.unbind(Globals.IS_SECURITY_ENABLED, MY_CLASSLOADER);
    }
}
```

4.1.1.1) StandardContextValve#invoke

```
public final void invoke(Request request, Response response)
    throws IOException, ServletException {

    // Disallow any direct access to resources under WEB-INF or META-INF
    MessageBytes requestPathMB = request.getRequestPathMB();
    if ((requestPathMB.startsWithIgnoreCase("/META-INF/", 0))
            || (requestPathMB.equalsIgnoreCase("/META-INF"))
            || (requestPathMB.startsWithIgnoreCase("/WEB-INF/", 0))
            || (requestPathMB.equalsIgnoreCase("/WEB-INF"))) {
        response.sendError(HttpServletResponse.SC_NOT_FOUND);
        return;
    }

    // Select the Wrapper to be used for this Request
    Wrapper wrapper = request.getWrapper();
    if (wrapper == null || wrapper.isUnavailable()) {
        response.sendError(HttpServletResponse.SC_NOT_FOUND);
        return;
    }

    // Acknowledge the request
    try {
        response.sendAcknowledgement();
    } catch (IOException ioe) {
        container.getLogger().error(sm.getString(
                "standardContextValve.acknowledgeException"), ioe);
        request.setAttribute(RequestDispatcher.ERROR_EXCEPTION, ioe);
        response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        return;
    }

    if (request.isAsyncSupported()) {
        request.setAsyncSupported(wrapper.getPipeline().isAsyncSupported());
    }
    wrapper.getPipeline().getFirst().invoke(request, response);
}
```

4.1.1.1.1) （调用 Servlet）StandardWrapperValve#invoke

StandardWrapperValve

1. allocate() 用来加载并初始化 Servlet，值的一提的是 Servlet 并不都是单例的，当 Servlet 实现了 SingleThreadModel 接口后，StandardWrapper 会维护一组 Servlet 实例，这是享元模式。当然了 SingleThreadModel 在 Servlet 2.4 以后就弃用了。
2. createFilterChain() 方法会从 StandardContext 中获取到所有的过滤器，然后将匹配 Request URL 的所有过滤器挑选出来添加到 filterChain 中。
3. doFilter() 执行过滤链,当所有的过滤器都执行完毕后调用 Servlet 的 service() 方法。

```
public final void invoke(Request request, Response response)
    throws IOException, ServletException {

    // Initialize local variables we may need
    boolean unavailable = false;
    Throwable throwable = null;
    // This should be a Request attribute...
    long t1=System.currentTimeMillis();
    requestCount.incrementAndGet();
    StandardWrapper wrapper = (StandardWrapper) getContainer();
    Servlet servlet = null;
    Context context = (Context) wrapper.getParent();

    // Check for the application being marked unavailable
    if (!context.getState().isAvailable()) {
        response.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE,
                     sm.getString("standardContext.isUnavailable"));
        unavailable = true;
    }

    // Check for the servlet being marked unavailable
    if (!unavailable && wrapper.isUnavailable()) {

container.getLogger().info(sm.getString("standardWrapper.isUnavailable",
              wrapper.getName()));
        long available = wrapper.getAvailable();
        if ((available > 0L) && (available < Long.MAX_VALUE)) {
            response.setDateHeader("Retry-After", available);
            response.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE,
                   sm.getString("standardWrapper.isUnavailable",
                          wrapper.getName()));
        } else if (available == Long.MAX_VALUE) {
            response.sendError(HttpServletResponse.SC_NOT_FOUND,
                   sm.getString("standardWrapper.notFound",
                          wrapper.getName()));
        }
        unavailable = true;
    }

    // Allocate a servlet instance to process this request
    try {
        if (!unavailable) {
            servlet = wrapper.allocate();
        }
    } catch (UnavailableException e) {
        container.getLogger().error(
               sm.getString("standardWrapper.allocateException",
                      wrapper.getName()), e);
        long available = wrapper.getAvailable();
        if ((available > 0L) && (available < Long.MAX_VALUE)) {
            response.setDateHeader("Retry-After", available);
```

```
           response.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE,
                     sm.getString("standardWrapper.isUnavailable",
                                 wrapper.getName()));
       } else if (available == Long.MAX_VALUE) {
           response.sendError(HttpServletResponse.SC_NOT_FOUND,
                     sm.getString("standardWrapper.notFound",
                                 wrapper.getName()));
       }
   } catch (ServletException e) {

container.getLogger().error(sm.getString("standardWrapper.allocateException
",
                      wrapper.getName()), StandardWrapper.getRootCause(e));
       throwable = e;
       exception(request, response, e);
   } catch (Throwable e) {
       ExceptionUtils.handleThrowable(e);

container.getLogger().error(sm.getString("standardWrapper.allocateException
",
                      wrapper.getName()), e);
       throwable = e;
       exception(request, response, e);
       servlet = null;
   }

   MessageBytes requestPathMB = request.getRequestPathMB();
   DispatcherType dispatcherType = DispatcherType.REQUEST;
   if (request.getDispatcherType()==DispatcherType.ASYNC) dispatcherType =
DispatcherType.ASYNC;
   request.setAttribute(Globals.DISPATCHER_TYPE_ATTR,dispatcherType);
   request.setAttribute(Globals.DISPATCHER_REQUEST_PATH_ATTR,
           requestPathMB);
   // Create the filter chain for this request
   ApplicationFilterChain filterChain =
           ApplicationFilterFactory.createFilterChain(request, wrapper,
servlet);

   // Call the filter chain for this request
   // NOTE: This also calls the servlet's service() method
   try {
       if ((servlet != null) && (filterChain != null)) {
           // Swallow output if needed
           if (context.getSwallowOutput()) {
                try {
                    SystemLogHandler.startCapture();
                    if (request.isAsyncDispatching()) {

request.getAsyncContextInternal().doInternalDispatch();
                    } else {
                        filterChain.doFilter(request.getRequest(),
                                response.getResponse());
                    }
                } finally {
                    String log = SystemLogHandler.stopCapture();
                    if (log != null && log.length() > 0) {
                        context.getLogger().info(log);
                    }
                }
            } else {
                if (request.isAsyncDispatching()) {
                    request.getAsyncContextInternal().doInternalDispatch();
                } else {
                    filterChain.doFilter
                        (request.getRequest(), response.getResponse());
                }
            }

        }
    } catch (ClientAbortException e) {
        throwable = e;
        exception(request, response, e);
    } catch (IOException e) {
        container.getLogger().error(sm.getString(
                "standardWrapper.serviceException", wrapper.getName(),
                context.getName()), e);
        throwable = e;
        exception(request, response, e);
    } catch (UnavailableException e) {
        container.getLogger().error(sm.getString(
                "standardWrapper.serviceException", wrapper.getName(),
                context.getName()), e);
        //              throwable = e;
        //              exception(request, response, e);
        wrapper.unavailable(e);
        long available = wrapper.getAvailable();
        if ((available > 0L) && (available < Long.MAX_VALUE)) {
            response.setDateHeader("Retry-After", available);
            response.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE,
                       sm.getString("standardWrapper.isUnavailable",
                                    wrapper.getName()));
        } else if (available == Long.MAX_VALUE) {
            response.sendError(HttpServletResponse.SC_NOT_FOUND,
                        sm.getString("standardWrapper.notFound",
                                    wrapper.getName()));
        }
        // Do not save exception in 'throwable', because we
        // do not want to do exception(request, response, e) processing
    } catch (ServletException e) {
        Throwable rootCause = StandardWrapper.getRootCause(e);
        if (!(rootCause instanceof ClientAbortException)) {
            container.getLogger().error(sm.getString(
                    "standardWrapper.serviceExceptionRoot",
                    wrapper.getName(), context.getName(), e.getMessage()),
                    rootCause);
        }
        throwable = e;
        exception(request, response, e);
    } catch (Throwable e) {
        ExceptionUtils.handleThrowable(e);
        container.getLogger().error(sm.getString(
                "standardWrapper.serviceException", wrapper.getName(),
                context.getName()), e);
        throwable = e;
        exception(request, response, e);
    }

    // Release the filter chain (if any) for this request
    if (filterChain != null) {
        filterChain.release();
    }

    // Deallocate the allocated servlet instance
    try {
        if (servlet != null) {
            wrapper.deallocate(servlet);
        }
    } catch (Throwable e) {
        ExceptionUtils.handleThrowable(e);

container.getLogger().error(sm.getString("standardWrapper.deallocateExcepti
on",
                        wrapper.getName()), e);
        if (throwable == null) {
            throwable = e;
            exception(request, response, e);
        }
    }

    // If this servlet has been marked permanently unavailable,
    // unload it and release this instance
    try {
        if ((servlet != null) &&
            (wrapper.getAvailable() == Long.MAX_VALUE)) {
            wrapper.unload();
        }
    } catch (Throwable e) {
        ExceptionUtils.handleThrowable(e);

container.getLogger().error(sm.getString("standardWrapper.unloadException",
                        wrapper.getName()), e);
        if (throwable == null) {
            throwable = e;
            exception(request, response, e);
        }
    }
    long t2=System.currentTimeMillis();

    long time=t2-t1;
    processingTime += time;
    if( time > maxTime) maxTime=time;
    if( time < minTime) minTime=time;

}
```

4.1.1.1.1.1) StandardWrapper#allocate（创建 servlet 实例）

```
public Servlet allocate() throws ServletException {

    // If we are currently unloading this servlet, throw an exception
    if (unloading) {
        throw new ServletException(sm.getString("standardWrapper.unloading",
getName()));
    }
```

```java
    boolean newInstance = false;

    // If not SingleThreadedModel, return the same instance every time
    if (!singleThreadModel) {
        // Load and initialize our instance if necessary
        if (instance == null || !instanceInitialized) {
            synchronized (this) {
                if (instance == null) {
                    try {
                        if (log.isDebugEnabled()) {
                            log.debug("Allocating non-STM instance");
                        }

                        // Note: We don't know if the Servlet implements
                        // SingleThreadModel until we have loaded it.
                        instance = loadServlet();
                        newInstance = true;
                        if (!singleThreadModel) {
                            // For non-STM, increment here to prevent a race
                            // condition with unload. Bug 43683, test case
                            // #3
                            countAllocated.incrementAndGet();
                        }
                    } catch (ServletException e) {
                        throw e;
                    } catch (Throwable e) {
                        ExceptionUtils.handleThrowable(e);
                        throw new
ServletException(sm.getString("standardWrapper.allocate"), e);
                    }
                }
                if (!instanceInitialized) {
                    initServlet(instance);
                }
            }
        }

        if (singleThreadModel) {
            if (newInstance) {
                // Have to do this outside of the sync above to prevent a
                // possible deadlock
                synchronized (instancePool) {
                    instancePool.push(instance);
                    nInstances++;
                }
            }
        } else {
            if (log.isTraceEnabled()) {
                log.trace("  Returning non-STM instance");
            }
            // For new instances, count will have been incremented at the
            // time of creation
            if (!newInstance) {
                countAllocated.incrementAndGet();
            }
            return instance;
        }
    }

    synchronized (instancePool) {
        while (countAllocated.get() >= nInstances) {
            // Allocate a new instance if possible, or else wait
            if (nInstances < maxInstances) {
                try {
                    instancePool.push(loadServlet());
                    nInstances++;
                } catch (ServletException e) {
                    throw e;
                } catch (Throwable e) {
                    ExceptionUtils.handleThrowable(e);
                    throw new
ServletException(sm.getString("standardWrapper.allocate"), e);
                }
            } else {
                try {
                    instancePool.wait();
                } catch (InterruptedException e) {
                    // Ignore
                }
            }
        }
        if (log.isTraceEnabled()) {
            log.trace("  Returning allocated STM instance");
        }
        countAllocated.incrementAndGet();
        return instancePool.pop();
    }
}
```

4.1.1.1.1.1.1) StandardWrapper#loadServlet

```java
public synchronized Servlet loadServlet() throws ServletException {

    // Nothing to do if we already have an instance or an instance pool
    if (!singleThreadModel && (instance != null))
        return instance;

    PrintStream out = System.out;
    if (swallowOutput) {
        SystemLogHandler.startCapture();
    }

    Servlet servlet;
    try {
        long t1=System.currentTimeMillis();
        // Complain if no servlet class has been specified
        if (servletClass == null) {
            unavailable(null);
            throw new ServletException
                (sm.getString("standardWrapper.notClass", getName()));
        }

        InstanceManager instanceManager =
((StandardContext)getParent()).getInstanceManager();
        try {
            servlet = (Servlet) instanceManager.newInstance(servletClass);
        } catch (ClassCastException e) {
            unavailable(null);
            // Restore the context ClassLoader
            throw new ServletException
                (sm.getString("standardWrapper.notServlet", servletClass), e);
        } catch (Throwable e) {
            e = ExceptionUtils.unwrapInvocationTargetException(e);
            ExceptionUtils.handleThrowable(e);
            unavailable(null);

            // Added extra log statement for Bugzilla 36630:
            // http://bz.apache.org/bugzilla/show_bug.cgi?id=36630
            if(log.isDebugEnabled()) {
                log.debug(sm.getString("standardWrapper.instantiate",
servletClass), e);
```

```java
            }

            // Restore the context ClassLoader
            throw new ServletException
                (sm.getString("standardWrapper.instantiate", servletClass), e);
        }

        if (multipartConfigElement == null) {
            MultipartConfig annotation =
                    servlet.getClass().getAnnotation(MultipartConfig.class);
            if (annotation != null) {
                multipartConfigElement =
                        new MultipartConfigElement(annotation);
            }
        }

        // Special handling for ContainerServlet instances
        // Note: The InstanceManager checks if the application is permitted
        //       to load ContainerServlets
        if (servlet instanceof ContainerServlet) {
            ((ContainerServlet) servlet).setWrapper(this);
        }

        classLoadTime=(int) (System.currentTimeMillis() -t1);

        if (servlet instanceof SingleThreadModel) {
            if (instancePool == null) {
                instancePool = new Stack<>();
            }
            singleThreadModel = true;
        }

        initServlet(servlet);

        fireContainerEvent("load", this);

        loadTime=System.currentTimeMillis() -t1;
    } finally {
        if (swallowOutput) {
            String log = SystemLogHandler.stopCapture();
            if (log != null && log.length() > 0) {
                if (getServletContext() != null) {
                    getServletContext().log(log);
                } else {
                    out.println(log);
                }
            }
        }
    }
    return servlet;

}
```

4.1.1.1.1.1.1.1) StandardWrapper#initServlet

```
private synchronized void initServlet(Servlet servlet)
        throws ServletException {

    if (instanceInitialized && !singleThreadModel) return;

    // Call the initialization method of this servlet
    try {
        if( Globals.IS_SECURITY_ENABLED) {
            boolean success = false;
            try {
                Object[] args = new Object[] { facade };
                SecurityUtil.doAsPrivilege("init",
                                           servlet,
                                           classType,
                                           args);
                success = true;
            } finally {
                if (!success) {
                    // destroy() will not be called, thus clear the reference now
                    SecurityUtil.remove(servlet);
                }
            }
        } else {
            servlet.init(facade);
        }

        instanceInitialized = true;
    } catch (UnavailableException f) {
        unavailable(f);
        throw f;
    } catch (ServletException f) {
        // If the servlet wanted to be unavailable it would have
        // said so, so do not call unavailable(null).
        throw f;
    } catch (Throwable f) {
        ExceptionUtils.handleThrowable(f);
        getServletContext().log("StandardWrapper.Throwable", f );
        // If the servlet wanted to be unavailable it would have
        // said so, so do not call unavailable(null).
        throw new ServletException
            (sm.getString("standardWrapper.initException", getName()), f);
    }
}
```

4.1.1.1.1.2) ApplicationFilterFactory#createFilterChain

```
public static ApplicationFilterChain createFilterChain(ServletRequest request,
        Wrapper wrapper, Servlet servlet) {

    // If there is no servlet to execute, return null
    if (servlet == null)
        return null;

    // Create and initialize a filter chain object
    ApplicationFilterChain filterChain = null;
    if (request instanceof Request) {
        Request req = (Request) request;
        if (Globals.IS_SECURITY_ENABLED) {
            // Security: Do not recycle
            filterChain = new ApplicationFilterChain();
        } else {
            filterChain = (ApplicationFilterChain) req.getFilterChain();
            if (filterChain == null) {
                filterChain = new ApplicationFilterChain();
                req.setFilterChain(filterChain);
            }
        }
    } else {
        // Request dispatcher in use
        filterChain = new ApplicationFilterChain();
    }

    filterChain.setServlet(servlet);
    filterChain.setServletSupportsAsync(wrapper.isAsyncSupported());
    // Acquire the filter mappings for this Context
    StandardContext context = (StandardContext) wrapper.getParent();
    FilterMap filterMaps[] = context.findFilterMaps();

    // If there are no filter mappings, we are done
    if ((filterMaps == null) || (filterMaps.length == 0))
        return filterChain;

    // Acquire the information we will need to match filter mappings
    DispatcherType dispatcher =
            (DispatcherType)
request.getAttribute(Globals.DISPATCHER_TYPE_ATTR);

    String requestPath = null;
    Object attribute =
request.getAttribute(Globals.DISPATCHER_REQUEST_PATH_ATTR);
    if (attribute != null){
        requestPath = attribute.toString();
    }

    String servletName = wrapper.getName();

    // Add the relevant path-mapped filters to this filter chain
    for (int i = 0; i < filterMaps.length; i++) {
        if (!matchDispatcher(filterMaps[i] ,dispatcher)) {
            continue;
        }
        if (!matchFiltersURL(filterMaps[i], requestPath))
            continue;
        ApplicationFilterConfig filterConfig = (ApplicationFilterConfig)
            context.findFilterConfig(filterMaps[i].getFilterName());
        if (filterConfig == null) {
            // FIXME - log configuration problem
            continue;
        }
        filterChain.addFilter(filterConfig);
    }

    // Add filters that match on servlet name second
    for (int i = 0; i < filterMaps.length; i++) {
        if (!matchDispatcher(filterMaps[i] ,dispatcher)) {
            continue;
        }
        if (!matchFiltersServlet(filterMaps[i], servletName))
```

```
            continue;
        ApplicationFilterConfig filterConfig = (ApplicationFilterConfig)
            context.findFilterConfig(filterMaps[i].getFilterName());
        if (filterConfig == null) {
            // FIXME - log configuration problem
            continue;
        }
        filterChain.addFilter(filterConfig);
    }

    // Return the completed filter chain
    return filterChain;
}
```

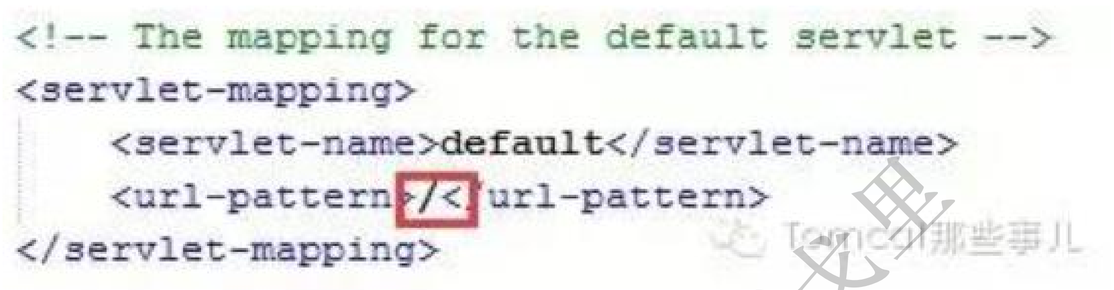
4.1.1.1.1.3) ApplicationFilterChain#doFilter

```
public void doFilter(ServletRequest request, ServletResponse response)
    throws IOException, ServletException {

    if( Globals.IS_SECURITY_ENABLED ) {
        final ServletRequest req = request;
        final ServletResponse res = response;
        try {
            java.security.AccessController.doPrivileged(
                new java.security.PrivilegedExceptionAction<Void>() {
                    @Override
                    public Void run()
                        throws ServletException, IOException {
                        internalDoFilter(req,res);
                        return null;
                    }
                }
            );
        } catch( PrivilegedActionException pe) {
            Exception e = pe.getException();
            if (e instanceof ServletException)
                throw (ServletException) e;
            else if (e instanceof IOException)
                throw (IOException) e;
            else if (e instanceof RuntimeException)
                throw (RuntimeException) e;
            else
                throw new ServletException(e.getMessage(), e);
        }
    } else {
        internalDoFilter(request,response);
    }
}
```

4.1.1.1.1.3.1) ApplicationFilterChain#internalDoFilter（这里是起个头，后续 doFilter 是在用户 Filter 中调用的）

```
private void internalDoFilter(ServletRequest request,
                              ServletResponse response)
    throws IOException, ServletException {

    // Call the next filter if there is one
    if (pos < n) {
        ApplicationFilterConfig filterConfig = filters[pos++];
        try {
            Filter filter = filterConfig.getFilter();

            if (request.isAsyncSupported() && "false".equalsIgnoreCase(
                    filterConfig.getFilterDef().getAsyncSupported())) {
                request.setAttribute(Globals.ASYNC_SUPPORTED_ATTR,
Boolean.FALSE);
            }
            if( Globals.IS_SECURITY_ENABLED ) {
                final ServletRequest req = request;
                final ServletResponse res = response;
                Principal principal =
                    ((HttpServletRequest) req).getUserPrincipal();

                Object[] args = new Object[]{req, res, this};
                SecurityUtil.doAsPrivilege ("doFilter", filter, classType, args,
principal);
            } else {
                filter.doFilter(request, response, this);
            }
        } catch (IOException | ServletException | RuntimeException e) {
            throw e;
        } catch (Throwable e) {
            e = ExceptionUtils.unwrapInvocationTargetException(e);
            ExceptionUtils.handleThrowable(e);
            throw new ServletException(sm.getString("filterChain.filter"), e);
        }
        return;
    }

    // We fell off the end of the chain -- call the servlet instance
    try {
        if (ApplicationDispatcher.WRAP_SAME_OBJECT) {
            lastServicedRequest.set(request);
            lastServicedResponse.set(response);
        }

        if (request.isAsyncSupported() && !servletSupportsAsync) {
            request.setAttribute(Globals.ASYNC_SUPPORTED_ATTR,
                    Boolean.FALSE);
        }
        // Use potentially wrapped request from this point
        if ((request instanceof HttpServletRequest) &&
                (response instanceof HttpServletResponse) &&
                Globals.IS_SECURITY_ENABLED ) {
            final ServletRequest req = request;
            final ServletResponse res = response;
            Principal principal =
                ((HttpServletRequest) req).getUserPrincipal();
            Object[] args = new Object[]{req, res};
            SecurityUtil.doAsPrivilege("service",
                                       servlet,
                                       classTypeUsedInService,
                                       args,
                                       principal);
        } else {
            servlet.service(request, response);
        }
    } catch (IOException | ServletException | RuntimeException e) {
        throw e;
    } catch (Throwable e) {
        e = ExceptionUtils.unwrapInvocationTargetException(e);
        ExceptionUtils.handleThrowable(e);
        throw new ServletException(sm.getString("filterChain.servlet"), e);
    } finally {
        if (ApplicationDispatcher.WRAP_SAME_OBJECT) {
            lastServicedRequest.set(null);
            lastServicedResponse.set(null);
        }
    }
}
```

4.1.1.1.1.3.1.1) DefaultServlet#service（处理静态资源，如果任何 servlet 都无法匹配，则转向该 servlet）



```
<!-- The mapping for the default servlet -->
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

```
protected void service(HttpServletRequest req, HttpServletResponse resp)
       throws ServletException, IOException {

    if (req.getDispatcherType() == DispatcherType.ERROR) {
        doGet(req, resp);
    } else {
        super.service(req, resp);
    }
}
```

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
    throws IOException, ServletException {

    // Serve the requested resource, including the data content
    serveResource(request, response, true, fileEncoding);

}
```

4.1.1.1.1.3.1.1.1) DefaultServlet#serveResource

首先会判断要请求的资源是否存在，文件是否可读，之后，根据资源的类型，设置响应头的 content-type，判断文件的时间，设置超时时间等，最终是流的读写。

```
/**
 * Serve the specified resource, optionally including the data content.
 *
 * @param request  The servlet request we are processing
```

```java
 * @param response The servlet response we are creating
 * @param content  Should the content be included?
 * @param encoding The encoding to use if it is necessary to access the
 *                 source as characters rather than as bytes
 *
 * @exception IOException if an input/output error occurs
 * @exception ServletException if a servlet-specified error occurs
 */
protected void serveResource(HttpServletRequest request,
                          HttpServletResponse response,
                          boolean content,
                          String encoding)
    throws IOException, ServletException {

    boolean serveContent = content;

    // Identify the requested resource path
    String path = getRelativePath(request, true);

    if (debug > 0) {
        if (serveContent)
            log("DefaultServlet.serveResource:  Serving resource '" +
                path + "' headers and data");
        else
            log("DefaultServlet.serveResource:  Serving resource '" +
                path + "' headers only");
    }

    if (path.length() == 0) {
        // Context root redirect
        doDirectoryRedirect(request, response);
        return;
    }

    WebResource resource = resources.getResource(path);
    boolean isError = DispatcherType.ERROR == request.getDispatcherType();

    if (!resource.exists()) {
        // Check if we're included so we can return the appropriate
        // missing resource name in the error
        String requestUri = (String) request.getAttribute(
                RequestDispatcher.INCLUDE_REQUEST_URI);
        if (requestUri == null) {
            requestUri = request.getRequestURI();
        } else {
            // We're included
            // SRV.9.3 says we must throw a FNFE
            throw new FileNotFoundException(sm.getString(
                    "defaultServlet.missingResource", requestUri));
        }

        if (isError) {
            response.sendError(((Integer) request.getAttribute(
                    RequestDispatcher.ERROR_STATUS_CODE)).intValue());
        } else {
            response.sendError(HttpServletResponse.SC_NOT_FOUND, requestUri);
        }
        return;
    }

    if (!resource.canRead()) {
        // Check if we're included so we can return the appropriate
        // missing resource name in the error
        String requestUri = (String) request.getAttribute(
                RequestDispatcher.INCLUDE_REQUEST_URI);
        if (requestUri == null) {
            requestUri = request.getRequestURI();
        } else {
            // We're included
            // Spec doesn't say what to do in this case but a FNFE seems
            // reasonable
            throw new FileNotFoundException(sm.getString(
                    "defaultServlet.missingResource", requestUri));
        }

        if (isError) {
            response.sendError(((Integer) request.getAttribute(
                    RequestDispatcher.ERROR_STATUS_CODE)).intValue());
        } else {
            response.sendError(HttpServletResponse.SC_FORBIDDEN, requestUri);
        }
        return;
    }

    // If the resource is not a collection, and the resource path
    // ends with "/" or "\", return NOT FOUND
    if (resource.isFile() && (path.endsWith("/") || path.endsWith("\\"))) {
        // Check if we're included so we can return the appropriate
```

```
        // missing resource name in the error
        String requestUri = (String) request.getAttribute(
                RequestDispatcher.INCLUDE_REQUEST_URI);
        if (requestUri == null) {
            requestUri = request.getRequestURI();
        }
        response.sendError(HttpServletResponse.SC_NOT_FOUND, requestUri);
        return;
    }

    boolean included = false;
    // Check if the conditions specified in the optional If headers are
    // satisfied.
    if (resource.isFile()) {
        // Checking If headers
        included = (request.getAttribute(
                RequestDispatcher.INCLUDE_CONTEXT_PATH) != null);
        if (!included && !isError && !checkIfHeaders(request, response,
resource)) {
            return;
        }
    }

    // Find content type.
    String contentType = resource.getMimeType();
    if (contentType == null) {
        contentType = getServletContext().getMimeType(resource.getName());
        resource.setMimeType(contentType);
    }

    // These need to reflect the original resource, not the potentially
    // precompressed version of the resource so get them now if they are going
to
    // be needed later
    String eTag = null;
    String lastModifiedHttp = null;
    if (resource.isFile() && !isError) {
        eTag = resource.getETag();
        lastModifiedHttp = resource.getLastModifiedHttp();
    }


    // Serve a precompressed version of the file if present
    boolean usingPrecompressedVersion = false;
    if (compressionFormats.length > 0 && !included && resource.isFile() &&
            !pathEndsWithCompressedExtension(path)) {
        List<PrecompressedResource> precompressedResources =
                getAvailablePrecompressedResources(path);
        if (!precompressedResources.isEmpty()) {
            Collection<String> varyHeaders = response.getHeaders("Vary");
            boolean addRequired = true;
            for (String varyHeader : varyHeaders) {
                if ("*".equals(varyHeader) ||
                        "accept-encoding".equalsIgnoreCase(varyHeader)) {
                    addRequired = false;
                    break;
                }
            }
            if (addRequired) {
                response.addHeader("Vary", "accept-encoding");
            }
            PrecompressedResource bestResource =
                    getBestPrecompressedResource(request,
precompressedResources);
            if (bestResource != null) {
                response.addHeader("Content-Encoding",
bestResource.format.encoding);
                resource = bestResource.resource;
                usingPrecompressedVersion = true;
            }
        }
    }

    ArrayList<Range> ranges = null;
    long contentLength = -1L;

    if (resource.isDirectory()) {
        if (!path.endsWith("/")) {
            doDirectoryRedirect(request, response);
            return;
        }

        // Skip directory listings if we have been configured to
        // suppress them
        if (!listings) {
            response.sendError(HttpServletResponse.SC_NOT_FOUND,
                               request.getRequestURI());
            return;
        }
        contentType = "text/html;charset=UTF-8";
    } else {
        if (!isError) {
            if (useAcceptRanges) {
                // Accept ranges header
                response.setHeader("Accept-Ranges", "bytes");
            }

            // Parse range specifier
            ranges = parseRange(request, response, resource);

            // ETag header
            response.setHeader("ETag", eTag);

            // Last-Modified header
            response.setHeader("Last-Modified", lastModifiedHttp);
        }

        // Get content length
        contentLength = resource.getContentLength();
        // Special case for zero length files, which would cause a
        // (silent) ISE when setting the output buffer size
        if (contentLength == 0L) {
            serveContent = false;
        }
    }

    ServletOutputStream ostream = null;
    PrintWriter writer = null;

    if (serveContent) {
        // Trying to retrieve the servlet output stream
        try {
            ostream = response.getOutputStream();
        } catch (IllegalStateException e) {
            // If it fails, we try to get a Writer instead if we're
            // trying to serve a text file
            if (!usingPrecompressedVersion &&
                    ((contentType == null) ||
                            (contentType.startsWith("text")) ||
                            (contentType.endsWith("xml")) ||
                            (contentType.contains("/javascript")))
                    ) {
```

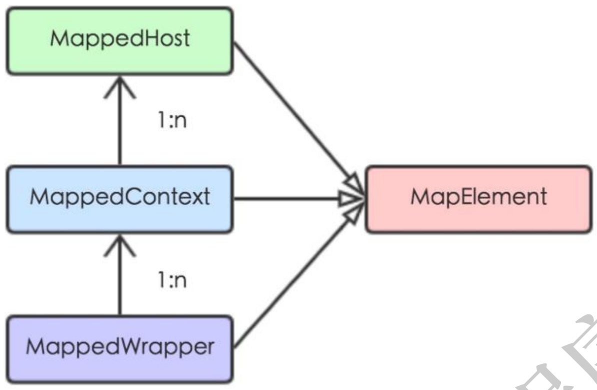
```
            writer = response.getWriter();
            // Cannot reliably serve partial content with a Writer
            ranges = FULL;
        } else {
            throw e;
        }
    }
}

// Check to see if a Filter, Valve of wrapper has written some content.
// If it has, disable range requests and setting of a content length
// since neither can be done reliably.
ServletResponse r = response;
long contentWritten = 0;
while (r instanceof ServletResponseWrapper) {
    r = ((ServletResponseWrapper) r).getResponse();
}
if (r instanceof ResponseFacade) {
    contentWritten = ((ResponseFacade) r).getContentWritten();
}
if (contentWritten > 0) {
    ranges = FULL;
}

if (resource.isDirectory() ||
        isError ||
        ( (ranges == null || ranges.isEmpty())
                && request.getHeader("Range") == null ) ||
        ranges == FULL ) {

    // Set the appropriate output headers
    if (contentType != null) {
        if (debug > 0)
            log("DefaultServlet.serveFile:  contentType='" +
                contentType + "'");
        response.setContentType(contentType);
    }
    if (resource.isFile() && contentLength >= 0 &&
            (!serveContent || ostream != null)) {
        if (debug > 0)
            log("DefaultServlet.serveFile:  contentLength=" +
                contentLength);
        // Don't set a content length if something else has already
        // written to the response.
```

```
            if (contentWritten == 0) {
                response.setContentLengthLong(contentLength);
            }
        }

        if (serveContent) {
            try {
                response.setBufferSize(output);
            } catch (IllegalStateException e) {
                // Silent catch
            }
            InputStream renderResult = null;
            if (ostream == null) {
                // Output via a writer so can't use sendfile or write
                // content directly.
                if (resource.isDirectory()) {
                    renderResult = render(getPathPrefix(request), resource,
encoding);
                } else {
                    renderResult = resource.getInputStream();
                }
                copy(resource, renderResult, writer, encoding);
            } else {
                // Output is via an InputStream
                if (resource.isDirectory()) {
                    renderResult = render(getPathPrefix(request), resource,
encoding);
                } else {
                    // Output is content of resource
                    if (!checkSendfile(request, response, resource,
                            contentLength, null)) {
                        // sendfile not possible so check if resource
                        // content is available directly
                        byte[] resourceBody = resource.getContent();
                        if (resourceBody == null) {
                            // Resource content not available, use
                            // inputstream
                            renderResult = resource.getInputStream();
                        } else {
                            // Use the resource content directly
                            ostream.write(resourceBody);
                        }
                    }
                }
            // If a stream was configured, it needs to be copied to
            // the output (this method closes the stream)
            if (renderResult != null) {
                copy(resource, renderResult, ostream);
            }
        }
    }

} else {

    if ((ranges == null) || (ranges.isEmpty()))
        return;

    // Partial content response.

    response.setStatus(HttpServletResponse.SC_PARTIAL_CONTENT);

    if (ranges.size() == 1) {

        Range range = ranges.get(0);
        response.addHeader("Content-Range", "bytes "
                           + range.start
                           + "-" + range.end + "/"
                           + range.length);
        long length = range.end - range.start + 1;
        response.setContentLengthLong(length);

        if (contentType != null) {
            if (debug > 0)
                log("DefaultServlet.serveFile:  contentType='" +
                    contentType + "'");
            response.setContentType(contentType);
        }

        if (serveContent) {
            try {
                response.setBufferSize(output);
            } catch (IllegalStateException e) {
                // Silent catch
            }
            if (ostream != null) {
                if (!checkSendfile(request, response, resource,
                        range.end - range.start + 1, range))
                    copy(resource, ostream, range);
                } else {
                    // we should not get here
                    throw new IllegalStateException();
                }
            }
        } else {
            response.setContentType("multipart/byteranges; boundary="
                                    + mimeSeparation);
            if (serveContent) {
                try {
                    response.setBufferSize(output);
                } catch (IllegalStateException e) {
                    // Silent catch
                }
                if (ostream != null) {
                    copy(resource, ostream, ranges.iterator(), contentType);
                } else {
                    // we should not get here
                    throw new IllegalStateException();
                }
            }
        }
    }
}
```

Mapper

在 Tomcat 中，当一个请求到达时，该请求最终由哪个 Servlet 来处理呢？这个任务是由 Mapper 路由映射器完成的。Mapper 是由 Service 管理。

存储结构



MapElement（基类）

```java
protected abstract static class MapElement<T> {

    public final String name;
    public final T object;

    public MapElement(String name, T object) {
        this.name = name;
        this.object = object;
    }
}
```

MappedHost

```java
protected static final class MappedHost extends MapElement<Host> {

    public volatile ContextList contextList;

    /**
     * Link to the "real" MappedHost, shared by all aliases.
     */
    private final MappedHost realHost;

    /**
     * Links to all registered aliases, for easy enumeration. This field
     * is available only in the "real" MappedHost. In an alias this field
     * is <code>null</code>.
     */
    private final List<MappedHost> aliases;

    /**
     * Constructor used for the primary Host
     *
     * @param name The name of the virtual host
     * @param host The host
     */
    public MappedHost(String name, Host host) {
        super(name, host);
        realHost = this;
        contextList = new ContextList();
        aliases = new CopyOnWriteArrayList<>();
    }
}
```

MappedContext

```
protected static final class MappedContext extends MapElement<Void> {
    public volatile ContextVersion[] versions;

    public MappedContext(String name, ContextVersion firstVersion) {
        super(name, null);
        this.versions = new ContextVersion[] { firstVersion };
    }
}
```

其中 ContextVersion 包含了 Context 下的所有 Servlet，有多种映射方式，如精确的 map，通配符的 map，扩展名的 map，如下：

```
protected static final class ContextVersion extends MapElement<Context> {
    public final String path;
    public final int slashCount;
    public final WebResourceRoot resources;
    public String[] welcomeResources;
    public MappedWrapper defaultWrapper = null;
    // 精确匹配
    public MappedWrapper[] exactWrappers = new MappedWrapper[0];
    // 通配符匹配
    public MappedWrapper[] wildcardWrappers = new MappedWrapper[0];
    // 基于扩展名的匹配
    public MappedWrapper[] extensionWrappers = new MappedWrapper[0];
    public int nesting = 0;
    private volatile boolean paused;

    public ContextVersion(String version, String path, int slashCount,
            Context context, WebResourceRoot resources,
            String[] welcomeResources) {
        super(version, context);
        this.path = path;
        this.slashCount = slashCount;
        this.resources = resources;
        this.welcomeResources = welcomeResources;
    }
}
```

MappedWrapper

```
protected static class MappedWrapper extends MapElement<Wrapper> {

    public final boolean jspWildCard;
    public final boolean resourceOnly;

    public MappedWrapper(String name, Wrapper wrapper, boolean jspWildCard,
            boolean resourceOnly) {
        super(name, wrapper);
        this.jspWildCard = jspWildCard;
        this.resourceOnly = resourceOnly;
    }
}
```

Mapper

简单地说，Mapper 中以数组的形式保存了 host, context, wrapper，且他们在数组中有序的，Mapper 可以通过请求的 url，通过二分法查找定位到 wrapper。

```
public final class Mapper {


    private static final Log log = LogFactory.getLog(Mapper.class);

    private static final StringManager sm =
StringManager.getManager(Mapper.class);

    // ----------------------------------------------------- Instance
Variables


    /**
     * Array containing the virtual hosts definitions.
     */
    // Package private to facilitate testing

    // host 数组，host 里面又包括了 context 和 wrapper 数组
    volatile MappedHost[] hosts = new MappedHost[0];


    /**
     * Default host name.
     */
    private String defaultHostName = null;
    private volatile MappedHost defaultHost = null;


    /**
     * Mapping from Context object to Context version to support
     * RequestDispatcher mappings.
     */
    private final Map<Context, ContextVersion>
contextObjectToContextVersionMap =
            new ConcurrentHashMap<>();

}
```

Mapper#addHost

```
public synchronized void addHost(String name, String[] aliases,
                                  Host host) {
    name = renameWildcardHost(name);
    MappedHost[] newHosts = new MappedHost[hosts.length + 1];
```

```java
    MappedHost newHost = new MappedHost(name, host);
    if (insertMap(hosts, newHosts, newHost)) {
        hosts = newHosts;
        if (newHost.name.equals(defaultHostName)) {
            defaultHost = newHost;
        }
        if (log.isDebugEnabled()) {
            log.debug(sm.getString("mapper.addHost.success", name));
        }
    } else {
        MappedHost duplicate = hosts[find(hosts, name)];
        if (duplicate.object == host) {
            // The host is already registered in the mapper.
            // E.g. it might have been added by addContextVersion()
            if (log.isDebugEnabled()) {
                log.debug(sm.getString("mapper.addHost.sameHost", name));
            }
            newHost = duplicate;
        } else {
            log.error(sm.getString("mapper.duplicateHost", name,
                    duplicate.getRealHostName()));
            // Do not add aliases, as removeHost(hostName) won't be able to
            // remove them
            return;
        }
    }
    List<MappedHost> newAliases = new ArrayList<>(aliases.length);
    for (String alias : aliases) {
        alias = renameWildcardHost(alias);
        MappedHost newAlias = new MappedHost(alias, newHost);
        if (addHostAliasImpl(newAlias)) {
            newAliases.add(newAlias);
        }
    }
    newHost.addAliases(newAliases);
}
```

Mapper#addContextVersion

```java
public void addContextVersion(String hostName, Host host, String path,
        String version, Context context, String[] welcomeResources,
        WebResourceRoot resources, Collection<WrapperMappingInfo> wrappers) {
```

```java
    hostName = renameWildcardHost(hostName);

    MappedHost mappedHost  = exactFind(hosts, hostName);
    if (mappedHost == null) {
        addHost(hostName, new String[0], host);
        mappedHost = exactFind(hosts, hostName);
        if (mappedHost == null) {
            log.error("No host found: " + hostName);
            return;
        }
    }
    if (mappedHost.isAlias()) {
        log.error("No host found: " + hostName);
        return;
    }
    int slashCount = slashCount(path);
    synchronized (mappedHost) {
        ContextVersion newContextVersion = new ContextVersion(version,
                path, slashCount, context, resources, welcomeResources);
        if (wrappers != null) {
            addWrappers(newContextVersion, wrappers);
        }

        ContextList contextList = mappedHost.contextList;
        MappedContext mappedContext = exactFind(contextList.contexts, path);
        if (mappedContext == null) {
            mappedContext = new MappedContext(path, newContextVersion);
            ContextList newContextList = contextList.addContext(
                    mappedContext, slashCount);
            if (newContextList != null) {
                updateContextList(mappedHost, newContextList);
                contextObjectToContextVersionMap.put(context,
newContextVersion);
            }
        } else {
            ContextVersion[] contextVersions = mappedContext.versions;
            ContextVersion[] newContextVersions = new
ContextVersion[contextVersions.length + 1];
            if (insertMap(contextVersions, newContextVersions,
                    newContextVersion)) {
                mappedContext.versions = newContextVersions;
                contextObjectToContextVersionMap.put(context,
newContextVersion);
            } else {
                // Re-registration after Context.reload()
                // Replace ContextVersion with the new one
                int pos = find(contextVersions, version);
                if (pos >= 0 && contextVersions[pos].name.equals(version)) {
                    contextVersions[pos] = newContextVersion;
                    contextObjectToContextVersionMap.put(context,
newContextVersion);
                }
            }
        }
    }

}
```

Mapper#addWrapper

```
public void addWrapper(String hostName, String contextPath, String version,
                       String path, Wrapper wrapper, boolean jspWildCard,
                       boolean resourceOnly) {
    hostName = renameWildcardHost(hostName);
    ContextVersion contextVersion = findContextVersion(hostName,
            contextPath, version, false);
    if (contextVersion == null) {
        return;
    }
    addWrapper(contextVersion, path, wrapper, jspWildCard, resourceOnly);
}
```

```
protected void addWrapper(ContextVersion context, String path,
        Wrapper wrapper, boolean jspWildCard, boolean resourceOnly) {

    synchronized (context) {
        if (path.endsWith("/*")) {
            // Wildcard wrapper
            String name = path.substring(0, path.length() - 2);
            MappedWrapper newWrapper = new MappedWrapper(name, wrapper,
                    jspWildCard, resourceOnly);
            MappedWrapper[] oldWrappers = context.wildcardWrappers;
            MappedWrapper[] newWrappers = new MappedWrapper[oldWrappers.length
+ 1];
            if (insertMap(oldWrappers, newWrappers, newWrapper)) {
                context.wildcardWrappers = newWrappers;
                int slashCount = slashCount(newWrapper.name);
```

```java
                if (slashCount > context.nesting) {
                    context.nesting = slashCount;
                }
            }
        } else if (path.startsWith("*.")) {
            // Extension wrapper
            String name = path.substring(2);
            MappedWrapper newWrapper = new MappedWrapper(name, wrapper,
                    jspWildCard, resourceOnly);
            MappedWrapper[] oldWrappers = context.extensionWrappers;
            MappedWrapper[] newWrappers =
                new MappedWrapper[oldWrappers.length + 1];
            if (insertMap(oldWrappers, newWrappers, newWrapper)) {
                context.extensionWrappers = newWrappers;
            }
        } else if (path.equals("/")) {
            // Default wrapper
            MappedWrapper newWrapper = new MappedWrapper("", wrapper,
                    jspWildCard, resourceOnly);
            context.defaultWrapper = newWrapper;
        } else {
            // Exact wrapper
            final String name;
            if (path.length() == 0) {
                // Special case for the Context Root mapping which is
                // treated as an exact match
                name = "/";
            } else {
                name = path;
            }
            MappedWrapper newWrapper = new MappedWrapper(name, wrapper,
                    jspWildCard, resourceOnly);
            MappedWrapper[] oldWrappers = context.exactWrappers;
            MappedWrapper[] newWrappers = new MappedWrapper[oldWrappers.length
+ 1];
            if (insertMap(oldWrappers, newWrappers, newWrapper)) {
                context.exactWrappers = newWrappers;
            }
        }
    }
}
```

Mapper#find（查找 MapElement）

// 根据 name，查找一个 MapElement（host, context，或者 wrapper)

```java
/**
 * Find a map element given its name in a sorted array of map elements.
 * This will return the index for the closest inferior or equal item in the
 * given array.
 */
private static final <T> int find(MapElement<T>[] map, CharChunk name) {
    return find(map, name, name.getStart(), name.getEnd());
}
```

```java
/**
 * Find a map element given its name in a sorted array of map elements.
 * This will return the index for the closest inferior or equal item in the
 * given array.
 */
private static final <T> int find(MapElement<T>[] map, CharChunk name,
                                  int start, int end) {

    int a = 0;
    int b = map.length - 1;

    // Special cases: -1 and 0
    if (b == -1) {
        return -1;
    }

    if (compare(name, start, end, map[0].name) < 0 ) {
        return -1;
    }
    if (b == 0) {
        return 0;
    }

    int i = 0;
    while (true) {
        i = (b + a) >>> 1;
        int result = compare(name, start, end, map[i].name);
        if (result == 1) {
            a = i;
        } else if (result == 0) {
            return i;
        } else {
            b = i;
        }
        if ((b - a) == 1) {
            int result2 = compare(name, start, end, map[b].name);
            if (result2 < 0) {
                return a;
            } else {
                return b;
            }
        }
    }
}
```

```java
/**
 * Compare given char chunk with String.
 * Return -1, 0 or +1 if inferior, equal, or superior to the String.
 */
private static final int compare(CharChunk name, int start, int end,
                                 String compareTo) {
    int result = 0;
    char[] c = name.getBuffer();
    int len = compareTo.length();
    if ((end - start) < len) {
        len = end - start;
    }
    for (int i = 0; (i < len) && (result == 0); i++) {
        if (c[i + start] > compareTo.charAt(i)) {
            result = 1;
        } else if (c[i + start] < compareTo.charAt(i)) {
            result = -1;
        }
    }
    if (result == 0) {
        if (compareTo.length() > (end - start)) {
            result = -1;
        } else if (compareTo.length() < (end - start)) {
            result = 1;
        }
    }
    return result;
}
```

Mapper#exactFind（精确查找 MapElement）

```
private static final <T, E extends MapElement<T>> E exactFind(E[] map,
        String name) {
    int pos = find(map, name);
    if (pos >= 0) {
        E result = map[pos];
        if (name.equals(result.name)) {
            return result;
        }
    }
    return null;
}
```

Mapper#map

```
public void map(MessageBytes host, MessageBytes uri, String version,
             MappingData mappingData) throws IOException {

    if (host.isNull()) {
        host.getCharChunk().append(defaultHostName);
    }
    host.toChars();
    uri.toChars();
    internalMap(host.getCharChunk(), uri.getCharChunk(), version,
            mappingData);
}
```

MappingData 是 Request 中的域

internalMap（查找 host 和 context）

```
private final void internalMap(CharChunk host, CharChunk uri,
        String version, MappingData mappingData) throws IOException {

    if (mappingData.host != null) {
        // The legacy code (dating down at least to Tomcat 4.1) just
        // skipped all mapping work in this case. That behaviour has a risk
        // of returning an inconsistent result.
        // I do not see a valid use case for it.
        throw new AssertionError();
```

```java
}

// Virtual host mapping
MappedHost[] hosts = this.hosts;
MappedHost mappedHost = exactFindIgnoreCase(hosts, host);
if (mappedHost == null) {
    // Note: Internally, the Mapper does not use the leading * on a
    //       wildcard host. This is to allow this shortcut.
    int firstDot = host.indexOf('.');
    if (firstDot > -1) {
        int offset = host.getOffset();
        try {
            host.setOffset(firstDot + offset);
            mappedHost = exactFindIgnoreCase(hosts, host);
        } finally {
            // Make absolutely sure this gets reset
            host.setOffset(offset);
        }
    }
    if (mappedHost == null) {
        mappedHost = defaultHost;
        if (mappedHost == null) {
            return;
        }
    }
}
// 设置 host
mappingData.host = mappedHost.object;

if (uri.isNull()) {
    // Can't map context or wrapper without a uri
    return;
}

uri.setLimit(-1);

// Context mapping
ContextList contextList = mappedHost.contextList;
MappedContext[] contexts = contextList.contexts;
int pos = find(contexts, uri);
if (pos == -1) {
    return;
}
```

```java
int lastSlash = -1;
int uriEnd = uri.getEnd();
int length = -1;
boolean found = false;
MappedContext context = null;
while (pos >= 0) {
    context = contexts[pos];
    if (uri.startsWith(context.name)) {
        length = context.name.length();
        if (uri.getLength() == length) {
            found = true;
            break;
        } else if (uri.startsWithIgnoreCase("/", length)) {
            found = true;
            break;
        }
    }
    if (lastSlash == -1) {
        lastSlash = nthSlash(uri, contextList.nesting + 1);
    } else {
        lastSlash = lastSlash(uri);
    }
    uri.setEnd(lastSlash);
    pos = find(contexts, uri);
}
uri.setEnd(uriEnd);

if (!found) {
    if (contexts[0].name.equals("")) {
        context = contexts[0];
    } else {
        context = null;
    }
}
if (context == null) {
    return;
}

mappingData.contextPath.setString(context.name);

ContextVersion contextVersion = null;
ContextVersion[] contextVersions = context.versions;
final int versionCount = contextVersions.length;
if (versionCount > 1) {
        Context[] contextObjects = new Context[contextVersions.length];
        for (int i = 0; i < contextObjects.length; i++) {
            contextObjects[i] = contextVersions[i].object;
        }
        mappingData.contexts = contextObjects;
        if (version != null) {
            contextVersion = exactFind(contextVersions, version);
        }
    }
    if (contextVersion == null) {
        // Return the latest version
        // The versions array is known to contain at least one element
        contextVersion = contextVersions[versionCount - 1];
    }
    mappingData.context = contextVersion.object;
    mappingData.contextSlashCount = contextVersion.slashCount;

    // Wrapper mapping
    if (!contextVersion.isPaused()) {
        internalMapWrapper(contextVersion, uri, mappingData);
    }

}
```

internalMapWrapper（查找 Wrapper）

```
private final void internalMapWrapper(ContextVersion contextVersion,
                                      CharChunk path,
                                      MappingData mappingData) throws
IOException {

    int pathOffset = path.getOffset();
    int pathEnd = path.getEnd();
    boolean noServletPath = false;

    int length = contextVersion.path.length();
    if (length == (pathEnd - pathOffset)) {
        noServletPath = true;
    }
    int servletPath = pathOffset + length;
    path.setOffset(servletPath);

    // Rule 1 -- Exact Match
```

```java
MappedWrapper[] exactWrappers = contextVersion.exactWrappers;
internalMapExactWrapper(exactWrappers, path, mappingData);

// Rule 2 -- Prefix Match
boolean checkJspWelcomeFiles = false;
MappedWrapper[] wildcardWrappers = contextVersion.wildcardWrappers;
if (mappingData.wrapper == null) {
    internalMapWildcardWrapper(wildcardWrappers, contextVersion.nesting,
                               path, mappingData);
    if (mappingData.wrapper != null && mappingData.jspWildCard) {
        char[] buf = path.getBuffer();
        if (buf[pathEnd - 1] == '/') {
            /*
             * Path ending in '/' was mapped to JSP servlet based on
             * wildcard match (e.g., as specified in url-pattern of a
             * jsp-property-group.
             * Force the context's welcome files, which are interpreted
             * as JSP files (since they match the url-pattern), to be
             * considered. See Bugzilla 27664.
             */
            mappingData.wrapper = null;
            checkJspWelcomeFiles = true;
        } else {
            // See Bugzilla 27704
            mappingData.wrapperPath.setChars(buf, path.getStart(),
                                              path.getLength());
            mappingData.pathInfo.recycle();
        }
    }
}

if(mappingData.wrapper == null && noServletPath &&
        contextVersion.object.getMapperContextRootRedirectEnabled()) {
    // The path is empty, redirect to "/"
    path.append('/');
    pathEnd = path.getEnd();
    mappingData.redirectPath.setChars
        (path.getBuffer(), pathOffset, pathEnd - pathOffset);
    path.setEnd(pathEnd - 1);
    return;
}

// Rule 3 -- Extension Match
MappedWrapper[] extensionWrappers = contextVersion.extensionWrappers;
```

```java
if (mappingData.wrapper == null && !checkJspWelcomeFiles) {
    internalMapExtensionWrapper(extensionWrappers, path, mappingData,
            true);
}

// Rule 4 -- Welcome resources processing for servlets
if (mappingData.wrapper == null) {
    boolean checkWelcomeFiles = checkJspWelcomeFiles;
    if (!checkWelcomeFiles) {
        char[] buf = path.getBuffer();
        checkWelcomeFiles = (buf[pathEnd - 1] == '/');
    }
    if (checkWelcomeFiles) {
        for (int i = 0; (i < contextVersion.welcomeResources.length)
                 && (mappingData.wrapper == null); i++) {
            path.setOffset(pathOffset);
            path.setEnd(pathEnd);
            path.append(contextVersion.welcomeResources[i], 0,
                    contextVersion.welcomeResources[i].length());
            path.setOffset(servletPath);

            // Rule 4a -- Welcome resources processing for exact macth
            internalMapExactWrapper(exactWrappers, path, mappingData);

            // Rule 4b -- Welcome resources processing for prefix match
            if (mappingData.wrapper == null) {
                internalMapWildcardWrapper
                    (wildcardWrappers, contextVersion.nesting,
                     path, mappingData);
            }

            // Rule 4c -- Welcome resources processing
            //            for physical folder
            if (mappingData.wrapper == null
                && contextVersion.resources != null) {
                String pathStr = path.toString();
                WebResource file =
                        contextVersion.resources.getResource(pathStr);
                if (file != null && file.isFile()) {
                    internalMapExtensionWrapper(extensionWrappers, path,
                                                mappingData, true);
                    if (mappingData.wrapper == null
                        && contextVersion.defaultWrapper != null) {
                        mappingData.wrapper =
                            contextVersion.defaultWrapper.object;
                        mappingData.requestPath.setChars
                            (path.getBuffer(), path.getStart(),
                             path.getLength());
                        mappingData.wrapperPath.setChars
                            (path.getBuffer(), path.getStart(),
                             path.getLength());
                        mappingData.requestPath.setString(pathStr);
                        mappingData.wrapperPath.setString(pathStr);
                    }
                }
            }
        }

        path.setOffset(servletPath);
        path.setEnd(pathEnd);
    }

}

/* welcome file processing - take 2
 * Now that we have looked for welcome files with a physical
 * backing, now look for an extension mapping listed
 * but may not have a physical backing to it. This is for
 * the case of index.jsf, index.do, etc.
 * A watered down version of rule 4
 */
if (mappingData.wrapper == null) {
    boolean checkWelcomeFiles = checkJspWelcomeFiles;
    if (!checkWelcomeFiles) {
        char[] buf = path.getBuffer();
        checkWelcomeFiles = (buf[pathEnd - 1] == '/');
    }
    if (checkWelcomeFiles) {
        for (int i = 0; (i < contextVersion.welcomeResources.length)
                 && (mappingData.wrapper == null); i++) {
            path.setOffset(pathOffset);
            path.setEnd(pathEnd);
            path.append(contextVersion.welcomeResources[i], 0,
                        contextVersion.welcomeResources[i].length());
            path.setOffset(servletPath);
            internalMapExtensionWrapper(extensionWrappers, path,
                                        mappingData, false);
        }
```

```
      path.setOffset(servletPath);
      path.setEnd(pathEnd);
    }
  }


  // Rule 7 -- Default servlet
  if (mappingData.wrapper == null && !checkJspWelcomeFiles) {
    if (contextVersion.defaultWrapper != null) {
      mappingData.wrapper = contextVersion.defaultWrapper.object;
      mappingData.requestPath.setChars
        (path.getBuffer(), path.getStart(), path.getLength());
      mappingData.wrapperPath.setChars
        (path.getBuffer(), path.getStart(), path.getLength());
      mappingData.matchType = MappingMatch.DEFAULT;
    }
    // Redirection to a folder
    char[] buf = path.getBuffer();
    if (contextVersion.resources != null && buf[pathEnd -1 ] != '/') {
      String pathStr = path.toString();
      WebResource file;
      // Handle context root
      if (pathStr.length() == 0) {
        file = contextVersion.resources.getResource("/");
      } else {
        file = contextVersion.resources.getResource(pathStr);
      }
      if (file != null && file.isDirectory() &&
          contextVersion.object.getMapperDirectoryRedirectEnabled())
{
        // Note: this mutates the path: do not do any processing
        // after this (since we set the redirectPath, there
        // shouldn't be any)
        path.setOffset(pathOffset);
        path.append('/');
        mappingData.redirectPath.setChars
          (path.getBuffer(), path.getStart(), path.getLength());
      } else {
        mappingData.requestPath.setString(pathStr);
        mappingData.wrapperPath.setString(pathStr);
      }
    }
  }
```

```
    path.setOffset(pathOffset);
    path.setEnd(pathEnd);
}
```

internalMapExactWrapper（URL 精确匹配）

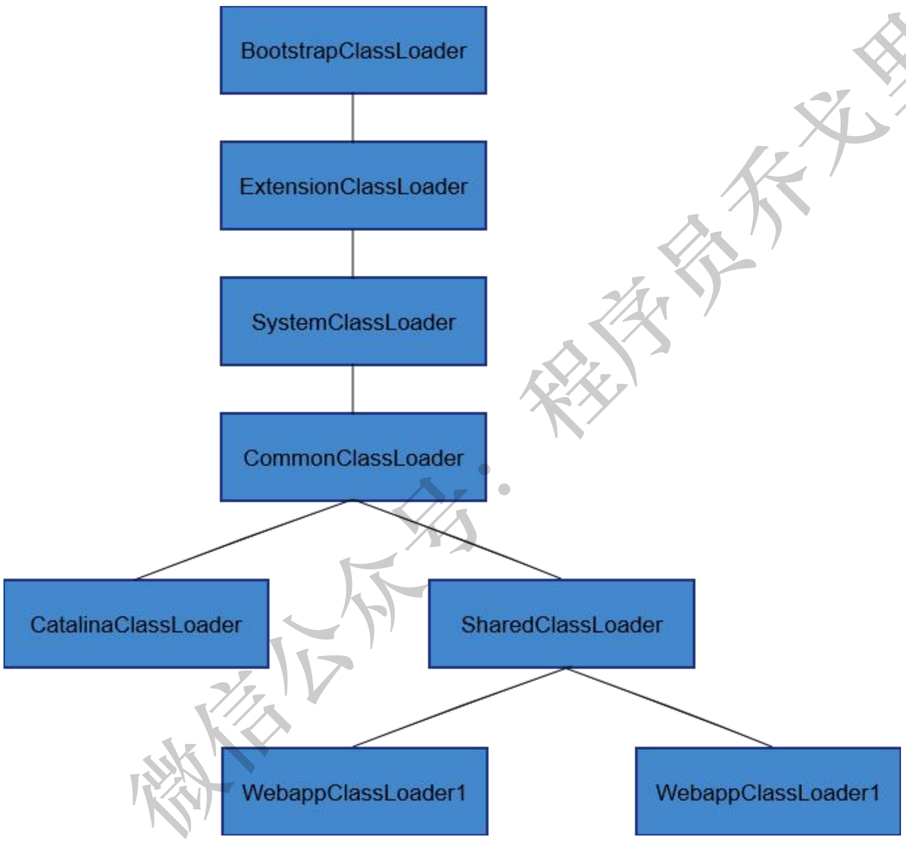
```
private final void internalMapExactWrapper
    (MappedWrapper[] wrappers, CharChunk path, MappingData mappingData) {
    MappedWrapper wrapper = exactFind(wrappers, path);
    if (wrapper != null) {
        mappingData.requestPath.setString(wrapper.name);
        mappingData.wrapper = wrapper.object;
        if (path.equals("/")) {
            // Special handling for Context Root mapped servlet
            mappingData.pathInfo.setString("/");
            mappingData.wrapperPath.setString("");
            // This seems wrong but it is what the spec says...
            mappingData.contextPath.setString("");
            mappingData.matchType = MappingMatch.CONTEXT_ROOT;
        } else {
            mappingData.wrapperPath.setString(wrapper.name);
            mappingData.matchType = MappingMatch.EXACT;
        }
    }
}
```

```
private static final <T, E extends MapElement<T>> E exactFind(E[] map,
        CharChunk name) {
    int pos = find(map, name);
    if (pos >= 0) {
        E result = map[pos];
        if (name.equals(result.name)) {
            return result;
        }
    }
    return null;
}
```

Tomcat 类加载器

Tomcat 不能直接使用系统的类加载器，必须要实现自定义的类加载器。servlet 应该只允许加载 WEB-INF/classes 目录及其子目录下的类，和从部署的库到 WEB-INF/lib 目录加载类，实现不同的应用之间的隔离。另一个要实现自定义类加载器的原因是，为了提供热加载的功能。如果 WEB-INF/classes 或 WEB-INF/lib 目录下的类发生变化时，Tomcat 应该会重新加载这些类。在 Tomcat 的类加载中，类加载使用一个额外的线程，不断检查 servlet 类和其他类的文件的时间戳。Tomcat 所有类加载器必须实现 Loader 接口，支持热加载的还需要实现 Reloader 接口。

Tomcat 类加载器



commonLoader、catalinaLoader 和 sharedLoader 是在 Tomcat 容器初始化时创建的。catalinaLoader 会被设置为 Tomcat 主线程的线程上下文类加载器，并且使用 catalinaLoader 加载 Tomcat 容器自身的 class。

它们三个都是 URLClassLoader 类的一个实例，只是它们的类加载路径不一样，在 tomcat/conf/catalina.properties 配置文件中配置 (common.loader,server.loader,shared.loader).

应用隔离

对于每个 webapp 应用，都会对应唯一的 StandContext，在 StandContext 中会引用 **WebappLoader，该类又会引用 WebappClassLoader**，WebappClassLoader 就是真正加载

webapp 的 classloader。

WebappClassLoader 加载 class 的步骤如下：

1. **先检查 webappclassloader 的缓存是否有该类**
2. **为防止 webapp 覆盖 java se 类，尝试用 application classloader（应用类加载器）加载**
3. **尝试 WebappClassLoader 自己加载 class**
4. **最后无条件地委托给父加载器 common classloader，加载 CATALINA_HOME/lib 下的类**
5. **如果都没有加载成功，则抛出 ClassNotFoundException 异常**

WebappClassLoader#loadClass

不同的 StandardContext 有不同的 WebappClassLoader，那么不同的 webapp 的类加载器就是不一致的。加载器的不一致带来了名称空间不一致，所以 webapp 之间是相互隔离的。

```
public Class<?> loadClass(String name) throws ClassNotFoundException {
    return loadClass(name, false);
}
```

```
public Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException {

    synchronized (getClassLoadingLock(name)) {
        if (log.isDebugEnabled())
            log.debug("loadClass(" + name + ", " + resolve + ")");
        Class<?> clazz = null;

        // Log access to stopped class loader
        checkStateForClassLoading(name);

        // (0) Check our previously loaded local class cache
        clazz = findLoadedClass0(name);
        if (clazz != null) {
            if (log.isDebugEnabled())
                log.debug("  Returning class from cache");
            if (resolve)
                resolveClass(clazz);
            return clazz;
        }

        // (0.1) Check our previously loaded class cache
        clazz = findLoadedClass(name);
        if (clazz != null) {
            if (log.isDebugEnabled())
                log.debug("  Returning class from cache");
            if (resolve)
                resolveClass(clazz);
            return clazz;
        }

        // (0.2) Try loading the class with the system class loader, to prevent
        //       the webapp from overriding Java SE classes. This implements
        //       SRV.10.7.2
        String resourceName = binaryNameToPath(name, false);

        ClassLoader javaseLoader = getJavaseClassLoader();
        boolean tryLoadingFromJavaseLoader;
        try {
            // Use getResource as it won't trigger an expensive
            // ClassNotFoundException if the resource is not available from
            // the Java SE class loader. However (see
            // https://bz.apache.org/bugzilla/show_bug.cgi?id=58125 for
            // details) when running under a security manager in rare cases
            // this call may trigger a ClassCircularityError.
            // See https://bz.apache.org/bugzilla/show_bug.cgi?id=61424 for
            // details of how this may trigger a StackOverflowError
            // Given these reported errors, catch Throwable to ensure any
            // other edge cases are also caught
            tryLoadingFromJavaseLoader =
(javaseLoader.getResource(resourceName) != null);
        } catch (Throwable t) {
            // Swallow all exceptions apart from those that must be re-thrown
            ExceptionUtils.handleThrowable(t);
            // The getResource() trick won't work for this class. We have to
            // try loading it directly and accept that we might get a
            // ClassNotFoundException.
            tryLoadingFromJavaseLoader = true;
        }
        // 使用 System ClassLoader 加载 J2SE 的类
        if (tryLoadingFromJavaseLoader) {
            try {
                clazz = javaseLoader.loadClass(name);
                if (clazz != null) {
                    if (resolve)
                        resolveClass(clazz);
                    return clazz;
                }
```

```
    } catch (ClassNotFoundException e) {
        // Ignore
    }
}

// (0.5) Permission to access this class when using a SecurityManager
if (securityManager != null) {
    int i = name.lastIndexOf('.');
    if (i >= 0) {
        try {
            securityManager.checkPackageAccess(name.substring(0,i));
        } catch (SecurityException se) {
            String error = "Security Violation, attempt to use " +
                "Restricted Class: " + name;
            log.info(error, se);
            throw new ClassNotFoundException(error, se);
        }
    }
}

boolean delegateLoad = delegate || filter(name, true);

// (1) Delegate to our parent if requested
if (delegateLoad) {
    if (log.isDebugEnabled())
        log.debug("  Delegating to parent classloader1 " + parent);
    try {
        clazz = Class.forName(name, false, parent);
        if (clazz != null) {
            if (log.isDebugEnabled())
                log.debug("  Loading class from parent");
            if (resolve)
                resolveClass(clazz);
            return clazz;
        }
    } catch (ClassNotFoundException e) {
        // Ignore
    }
}

// (2) Search local repositories
if (log.isDebugEnabled())
    log.debug("  Searching local repositories");
try {
```

```
            clazz = findClass(name);
            if (clazz != null) {
                if (log.isDebugEnabled())
                    log.debug("  Loading class from local repository");
                if (resolve)
                    resolveClass(clazz);
                return clazz;
            }
        } catch (ClassNotFoundException e) {
            // Ignore
        }

        // (3) Delegate to parent unconditionally
        if (!delegateLoad) {
            if (log.isDebugEnabled())
                log.debug("  Delegating to parent classloader at end: " + parent);
            try {
                clazz = Class.forName(name, false, parent);
                if (clazz != null) {
                    if (log.isDebugEnabled())
                        log.debug("  Loading class from parent");
                    if (resolve)
                        resolveClass(clazz);
                    return clazz;
                }
            } catch (ClassNotFoundException e) {
                // Ignore
            }
        }
    }

    throw new ClassNotFoundException(name);
}
```

热部署

后台的定期检查，该定期检查是 StandardContext 的一个后台线程，会做 **reload** 的 check，过期 **session** 清理等等，这里的 modified 实际上调用了 **WebappClassLoader** 中的方法以判断这个 class 是不是已经修改。注意到它调用了 StandardContext 的 reload 方法。

StandardContext#backgroundProcess

```java
public void backgroundProcess() {

    if (!getState().isAvailable())
        return;

    Loader loader = getLoader();
    if (loader != null) {
        try {
            loader.backgroundProcess();
        } catch (Exception e) {
            log.warn(sm.getString(
                    "standardContext.backgroundProcess.loader", loader), e);
        }
    }
    Manager manager = getManager();
    if (manager != null) {
        try {
            manager.backgroundProcess();
        } catch (Exception e) {
            log.warn(sm.getString(
                    "standardContext.backgroundProcess.manager", manager),
                    e);
        }
    }
    WebResourceRoot resources = getResources();
    if (resources != null) {
        try {
            resources.backgroundProcess();
        } catch (Exception e) {
            log.warn(sm.getString(
                    "standardContext.backgroundProcess.resources",
                    resources), e);
        }
    }
    InstanceManager instanceManager = getInstanceManager();
    if (instanceManager != null) {
        try {
            instanceManager.backgroundProcess();
        } catch (Exception e) {
            log.warn(sm.getString(
                    "standardContext.backgroundProcess.instanceManager",
                  resources), e);
        }
    }
    super.backgroundProcess();
}
```

WebappLoader#backgroundProcess

```
public void backgroundProcess() {
    if (reloadable && modified()) {
        try {
            Thread.currentThread().setContextClassLoader
                (WebappLoader.class.getClassLoader());
            if (context != null) {
                context.reload();
            }
        } finally {
            if (context != null && context.getLoader() != null) {
                Thread.currentThread().setContextClassLoader
                    (context.getLoader().getClassLoader());
            }
        }
    }
}
```

StandardContext#reload

Tomcat lifecycle 中标准的启停方法 stop 和 start，别忘了，start 方法会重新造一个 WebappClassLoader 并且重复 loadOnStartup 的过程，从而重新加载了 webapp 中的类，注意到一般应用很大时，热部署通常会报 outofmemory: permgen space not enough 之类的，这是由于之前加载进来的 class 还没有清除而方法区内存又不够的原因

```
public synchronized void reload() {

    // Validate our current component state
    if (!getState().isAvailable())
        throw new IllegalStateException
            (sm.getString("standardContext.notStarted", getName()));

    if(log.isInfoEnabled())
        log.info(sm.getString("standardContext.reloadingStarted",
              getName()));

    // Stop accepting requests temporarily.
    setPaused(true);

    try {
        stop();
    } catch (LifecycleException e) {
        log.error(
            sm.getString("standardContext.stoppingContext", getName()), e);
    }

    try {
        start();
    } catch (LifecycleException e) {
        log.error(
            sm.getString("standardContext.startingContext", getName()), e);
    }

    setPaused(false);

    if(log.isInfoEnabled())
        log.info(sm.getString("standardContext.reloadingCompleted",
                getName()));

}
```

异步 Servlet

入口点是 Request#startAsync

Request#startAsync（开启异步上下文，之后 Tomct 回收 Worker 线程）

```
public AsyncContext startAsync() {
    return startAsync(getRequest(),response.getResponse());
}
```

```
public AsyncContext startAsync(ServletRequest request,
        ServletResponse response) {
    if (!isAsyncSupported()) {
        IllegalStateException ise =
                new
IllegalStateException(sm.getString("request.asyncNotSupported"));
        log.warn(sm.getString("coyoteRequest.noAsync",
                StringUtils.join(getNonAsyncClassNames())), ise);
        throw ise;
    }

    if (asyncContext == null) {
        asyncContext = new AsyncContextImpl(this);
    }

    asyncContext.setStarted(getContext(), request, response,
            request==getRequest() && response==getResponse().getResponse());
    asyncContext.setTimeout(getConnector().getAsyncTimeout());

    return asyncContext;
}
```

1) AsyncContextImpl#construactor

成员变量
Tomcat 工作线程在 Request#startAsync 之后，把该异步 servlet 的后续代码执行完毕后，Tomcat 工作线程直接就结束了，也就是返回线程池中了，相当于线程根本不会保存记录信息。

```
public class AsyncContextImpl implements AsyncContext, AsyncContextCallback {

    private static final Log log = LogFactory.getLog(AsyncContextImpl.class);

    protected static final StringManager sm =
        StringManager.getManager(Constants.Package);

    /* When a request uses a sequence of multiple start(); dispatch() with
     * non-container threads it is possible for a previous dispatch() to
     * interfere with a following start(). This lock prevents that from
     * happening. It is a dedicated object as user code may lock on the
     * AsyncContext so if container code also locks on that object deadlocks
may
     * occur.
     */
    private final Object asyncContextLock = new Object();

    private volatile ServletRequest servletRequest = null;
    private volatile ServletResponse servletResponse = null;
    private final List<AsyncListenerWrapper> listeners = new ArrayList<>();
    private boolean hasOriginalRequestAndResponse = true;
    private volatile Runnable dispatch = null;
    private Context context = null;
    // Default of 30000 (30s) is set by the connector
    private long timeout = -1;
    private AsyncEvent event = null;
    private volatile Request request;
    private volatile InstanceManager instanceManager;
}
```

```
public AsyncContextImpl(Request request) {
    if (log.isDebugEnabled()) {
        logDebug("Constructor");
    }
    this.request = request;
}
```

2) AsyncContextImpl#setStarted

```
public void setStarted(Context context, ServletRequest request,
        ServletResponse response, boolean originalRequestResponse) {

    synchronized (asyncContextLock) {
        this.request.getCoyoteRequest().action(
                ActionCode.ASYNC_START, this);

        this.context = context;
        this.servletRequest = request;
        this.servletResponse = response;
        this.hasOriginalRequestAndResponse = originalRequestResponse;
        this.event = new AsyncEvent(this, request, response);

        List<AsyncListenerWrapper> listenersCopy = new ArrayList<>();
        listenersCopy.addAll(listeners);
        listeners.clear();
        for (AsyncListenerWrapper listener : listenersCopy) {
            try {
                listener.fireOnStartAsync(event);
            } catch (Throwable t) {
                ExceptionUtils.handleThrowable(t);
                log.warn("onStartAsync() failed for listener of type [" +
                        listener.getClass().getName() + "]", t);
            }
        }
    }
}
```

AbstractProcessor#action

```
case ASYNC_START: {
    asyncStateMachine.asyncStart((AsyncContextCallback) param);
    break;
}
```

AsyncStateMachine#asyncStart

```
synchronized void asyncStart(AsyncContextCallback asyncCtxt) {
    if (state == AsyncState.DISPATCHED) {
        state = AsyncState.STARTING;
        this.asyncCtxt = asyncCtxt;
        lastAsyncStart = System.currentTimeMillis();
    } else {
        throw new IllegalStateException(
                sm.getString("asyncStateMachine.invalidAsyncState",
                        "asyncStart()", state));
    }
}
```

3) AsyncContextImpl#setTimeout

```
public void setTimeout(long timeout) {
    check();
    this.timeout = timeout;
    request.getCoyoteRequest().action(ActionCode.ASYNC_SETTIMEOUT,
            Long.valueOf(timeout));
}
```

AsyncContext#complete（结束）

```
public void complete() {
    if (log.isDebugEnabled()) {
        logDebug("complete   ");
    }
    check();
    request.getCoyoteRequest().action(ActionCode.ASYNC_COMPLETE, null);
}
```

```
case ASYNC_COMPLETE: {
    clearDispatches();
    if (asyncStateMachine.asyncComplete()) {
        processSocketEvent(SocketEvent.OPEN_READ, true);
    }
    break;
}
```

```
protected void processSocketEvent(SocketEvent event, boolean dispatch) {
    SocketWrapperBase<?> socketWrapper = getSocketWrapper();
    if (socketWrapper != null) {
        socketWrapper.processSocket(event, dispatch);
    }
}
```

```
public void processSocket(SocketEvent socketStatus, boolean dispatch) {
    endpoint.processSocket(this, socketStatus, dispatch);
}
```

见 2.2.1) AbstractEndpoint#processSocket
相当于重新开启一个工作线程，这个工作线程带着 SocketWrapper，又来一遍容器的流程，而这一遍的流程，因为 Servlet 已经处理过，所以会略过 servlet 的执行直接将后续的处理走完，包括最后 response 的收尾，对象的清空等等。
但是异步 Servlet 此时不会重新跑一次 Servlet，直接跳到 response 收尾。

AsyncContext#dispatch（转发）