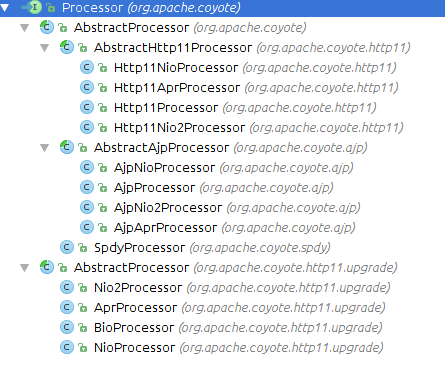
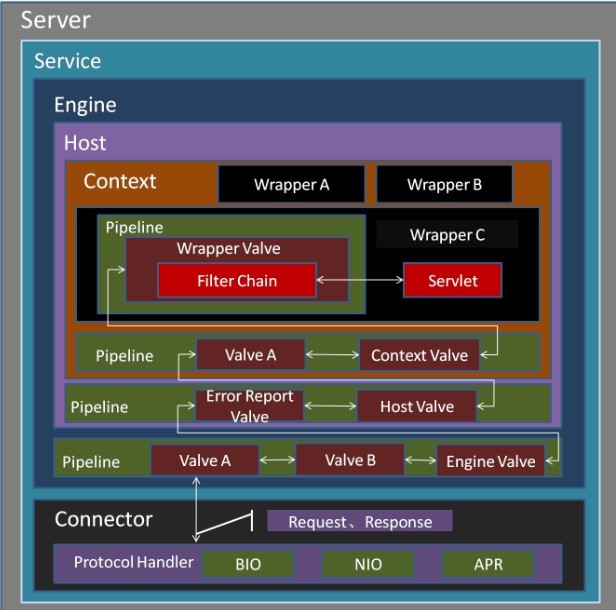
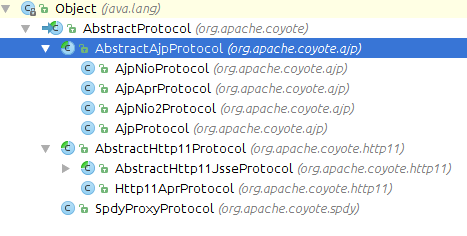
# Tomcat8分析

### 整体框架:





## 初始化过程:

Bootstrap.main()

Bootstrap.init()

-> public void init() throws Exception {

initClassLoaders();

//初始化类加载器commonLoader，catalinaLoader,sharedLoader

-> private void initClassLoaders() {

try {

//创建以/lib目录和/lib/\*.jar为仓库类的加载器

commonLoader = createClassLoader("common", null);

}

if( commonLoader == null ) {

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);

}

<- }

Thread.currentThread().setContextClassLoader(catalinaLoader);

SecurityClassLoad.securityClassLoad(catalinaLoader);

//使用catalina加载器载入Catalina类,之后实例化,再调用其setParentClassLoader方法

Class<?> startupClass =

catalinaLoader.loadClass

("org.apache.catalina.startup.Catalina");

Object startupInstance = startupClass.newInstance();

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;

daemon.load(args);//调用catalina.load() 初始化各个容器和其组件

daemon.start(); //调用catalina.start()启动各个容器和其组件

## 主要容器和组件的构造函数:

在Catalina.load()中digester.parse()会构造StandardServer,StandardEngine,StandardHost,Connector

以下是他们的构造函数

public Connector(String protocol) {

//根据server.xml的属性设置协议setProtocolHandlerClassName的值

setProtocol(protocol);

ProtocolHandler p = null;

Class<?> clazz = Class.forName(protocolHandlerClassName);

//实例化Connector中的ProtocolHandler

//TOMCAT8默认为"org.apache.coyote.http11.Http11NioProtocol"

p = (ProtocolHandler) clazz.newInstance();

-> public Http11NioProtocol() {

//在初始化Http11NioProtocol时候会设置endpoint为NioEndPoint()

endpoint=new NioEndpoint();

cHandler = new Http11ConnectionHandler(this);

((NioEndpoint) endpoint).setHandler(cHandler);

<- }

this.protocolHandler = p;

}

StandardServer,将globalNamingResouces容器设置为server

StandardEngine设置基础阀,并且辅助 backgroundProcessorDelay = 10;

所以Tomcat启动解析xml时碰到一个Engine节点就会对应产生一个后台处理线程。

StandardHost,StandardContext设置基础阀

## Catalina.load()

-> public void load() {

//创建一些规则来解析xml,之后通过digester.parse(file)按照规则解析

Digester digester = createStartDigester();

//放入栈顶进行相关容器，子孩子相关联

digester.push(this);

//解析server.xml，按照其规则创建StandardService,等..

digester.parse(inputSource);

getServer().init();

-> StandardServer.initInternal()

for (int i = 0; i < services.length; i++) {

services[i].init();

-> StandardService.initInternal(){

container.init(); //初始化container,此处为StandardEngine.init()

for (Connector connector : connectors) {

//一个server可以有多个connector,此处为HTTP/1.1 Connector

connector.init();

-> initInternal(){

adapter = new CoyoteAdapter(this); protocolHandler.setAdapter(adapter);

//AbstractHttp11JsseProtocol.init()通过super调用

//AbstractProtocol.init()在其中会调用

//endpoint.init();//此方法在Connector设置Protocol的endpoint中

//在endpoint中会调用bind()进行serversocket的初始化

protocolHandler.init();

}

}

}

}

## Catalina.start()//加载服务配置,启动服务,注册关闭钩子,监听关闭事件

Public start(){

getServer().start();

-> StandardServer.startInternal() {

for (int i = 0; i < services.length; i++) {

services[i].start();

-> StandardService.startInternal()

//Engine的ContainerBase.startInternal()

container.start();

//在Engine的ContainerBase.startInternal()内通过一个startStopExecutor

//来提交子容器初始化任务;在此处只有StandardHost

//之后调用StandardHost的startInternal()之后在最后会调用

//super.startInternal()即ContainerBase.startInternal()

//该函数会触发消息setState(LifecycleState.STARTING)，

//之后在createDigest中设置的HostConfig监听到该事件进行部署

//在HostConfig中deployDirectories()中提交部署线程来部署各个WEB目录

//在部署线程deployDirectory中会实例化StandardContext并且设置监听器ContextConfig

//deployDirectory中调用host.addChild(context)会调用addChildInternal其中会调用child.start()

//即StandardContext的start方法

//StandardContext.startInternal()会fireLifecycleEvent(Lifecycle.CONFIGURE\_START\_EVENT, null);

//之后其监听器ContextConfig根据事件调用configureStart()

//再调用webConfig()中调用configureContext()

//其中会进行一些关于过滤器，监听器,Servler的Wrapper相关信息设置.回到

//StandardContext.startInternal()

//继续执行会执行listenerStart()，filterStart()，loadOnStartup(),对监听器，过滤器，自启动

//servlet实例化

-> ContainerBase.startInternal(){

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])));

}

//engine会启动该后台线程

//并且启动子容器的后台线程backgroundProcess

//会对Web文件变动检测之后进行发布消息部署等功能

threadStart();

<- }

for (Connector connector: connectors) {

connector.start();

-> Connector.startInternal(){

//在Catalina.load()中digest.parse()解析规则中有一个Connector规则,

//在创建Connector的时候根据protocol="HTTP/1.1"

//初始化一个org.apache.coyote.http11.Http11NioProtocol，

//Tomcat8默认为org.apache.coyote.http11.Http11NioProtocol

//之后会根据Http11NioProtocol.start(){super.start()} -> AbstractProtocol.start(){endpoint.start();} ->

//AbstractEndpoint.start(){startInternal()}->NioEndpoint.startInternal()中构造线程池

//和acceptor和poller线程

protocolHandler.start();

<- }

<- }

<- }

if (await) {

await();

stop();

}

}

## Jio分析:

Connector.startInternal(){

//Http11Protocol.start() {super.start();}->AbstractProtocol.start() {endpoint.start();}->

//AbstractEndpoint.start(){ startInternal();}->JioEndpoint.startInternal()

//在JioEndpoint中会通过startAcceptorThreads()开启Acceptor线程,AsyncTimeout线程

//和创建一个线程池

protocolHandler.start();

}

Acceptor线程分析:

protected class Acceptor extends AbstractEndpoint.Acceptor {

socket = serverSocketFactory.acceptSocket(serverSocket);

if (!processSocket(socket)) {

-> protected boolean processSocket(Socket socket) {

// Process the request from this socket

SocketWrapper<Socket> wrapper = new SocketWrapper<>(socket);

wrapper.setKeepAliveLeft(getMaxKeepAliveRequests());

wrapper.setSecure(isSSLEnabled());

//把socket包装成SocketWrapper用来初始化SocketProcessor传递给线程池

getExecutor().execute(new SocketProcessor(wrapper));

return true;

<- }

}

protected class SocketProcessor implements Runnable {.

//调用Http11Protocol.ConnectionHandler.process(...){super.process(socket, status);}

//->AbstractProtocol.AbstractConnectionHandler.process(SocketWrapper<S> wrapper,

// SocketStatus status)

//接下面

state = handler.process(socket, SocketStatus.OPEN\_READ);

-> 接上面AbstractConnectionHandler.process(){

processor = createProcessor();

processor.process(wrapper);

-> AbstractHttp11Processor.process(SocketWrapper<S> socketWrapper){

getAdapter().service(request, response);

//以下方法会将请求转换成Tomcat中流转的形式

-> CoyotoAdapter.service(){

//通过postParseRequest方法的调用请求对象

//内保存了关于本次请求的具体要执行的 //Host、Context、Wrapper组件的引用。

boolean postParseSuccess = postParseRequest(req, request, res, response);

if(postParseSuccess){

//之后会调用pipeline链,Engine最后调用Host的

//pipeline.Host最后调用Context的pipeline.Context

//再调用Wrapper的pipeline

//最后调用到StandardWrapperValve.invoker()阀之后会调用

//过滤器链之后在最后过滤器的internalDoFileter()中调用

// servlet.service(request, response);

connector.getService().getContainer().getPipeline().

getFirst().invoke(request, response);

}

<- }

<- }

}

}

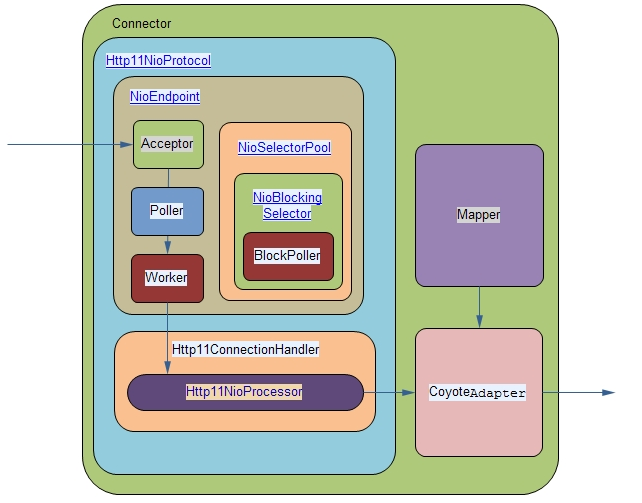
StandardWrapperValve阀简略过程:

请求来临时候最终调用StandardWrapperValve.invoke()会allocate()一个servlet,在allocate()方法中会调用loadSerlet()->initServlet(servlet)->servlet.init(facade);调用servlet本身的init()

,之后利用该wrapper和servler创建一个filterChain并且调用其doFilter方法

在DoFilter又会调用internalDoFilter(request,response)会在其中通过filters变量获取filter，之后再调用filter.doFilter(request, response, this)调用过滤器的chain.doFilter(req, resp);调用链中的下一个过滤器,调用完过滤器链的过滤器,就调用servlet.service(request, response);

## Bio分析:



public void startInternal() throws Exception {

// 构造线程池，用于后续执行SocketProcessor线程，这就是Worker。

createExecutor();

// 根据处理器数量构造一定数目的轮询器，即Poller

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();

}

//开启Acceptor线程

startAcceptorThreads();

-> protected final void startAcceptorThreads() {

int count = getAcceptorThreadCount();

acceptors = new Acceptor[count];

for (int i = 0; i < count; i++) {

acceptors[i] = createAcceptor();

String threadName = getName() + "-Acceptor-" + i;

acceptors[i].setThreadName(threadName);

Thread t = new Thread(acceptors[i], threadName);

t.setPriority(getAcceptorThreadPriority());

t.setDaemon(getDaemon());

t.start();

}

<- }

}

}

## NioEndpoint.Acceptor

// --------------------------------------------------- Acceptor Inner Class

/\*\*后台线程，用于监听TCP/IP连接以及将它们分发给相应的调度器处理。

\*/

protected class Acceptor extends AbstractEndpoint.Acceptor {

public void run() {

SocketChannel socket = null;

// 接收连接，这里用的阻塞模式。

// Accept the next incoming connection from the server

// socket

socket = serverSock.accept();

//把上面接收到的SocketChannel设置些参数,非阻塞模式 ，发送接受缓存大小,等等，

//然后通过 getPoller0().register(channel);

//将SocketChannel注册到Poller中添加到轮询器Poller中这里是events缓存队列

if (running && !paused) {

if (!setSocketOptions(socket)) {

countDownConnection();

closeSocket(socket);

}

}

## NioEndpoint.Poller

public void run() {

// Loop until destroy() is called

while (true) {

//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;

// 根据向selector中注册的key遍历channel中已经就绪的keys，并处理这些key

// Walk through the collection of ready keys and dispatch

// any active event.

while (iterator != null && iterator.hasNext()) {

SelectionKey sk = iterator.next();

// 这里的attachment方法返回的就是在register()方法中注册的

// 而KeyAttachment对象是对socket的包装

KeyAttachment attachment = (KeyAttachment)sk.attachment();

attachment.access();

iterator.remove();

// 具体处理通道的逻辑

processKey(sk, attachment);

-> processKey(){processSocket()}

//在processKey()会调用调用方法processSocket()

-> processSocket()

sc = new SocketProcessor(attachment, status);

Executor executor = getExecutor();

// 将有事件发生的socket交给Worker处理

executor.execute(sc);

<-

<-

}//while

protected class SocketProcessor implements Runnable {

public void run() {

NioChannel socket = ka.getSocket();

// 从socket中获取SelectionKey

SelectionKey key = socket.getIOChannel().keyFor(

socket.getPoller().getSelector());

doRun(key, ka);

private void doRun(SelectionKey key, KeyAttachment ka) {

NioChannel socket = ka.getSocket();

// 最关键的代码，这里将KeyAttachment(实际就是socket)交给Handler处理请求

state = handler.process(ka, SocketStatus.OPEN\_READ);

->HttpNioProtocol.process()->AbstractProtocol.process()->

->AbstractHttp11Processor->getAdapter().service(request, response);

//和Jio一样调用coyote.service()...

}