

# Tomcat & Jetty

---

## Tomcat & Jetty

基础

系统架构

连接器的设计

容器的设计

Tomcat实现一键式启停

Tomcat的高层组件负责做的事情

Jetty的Connector

Jetty的Handler

组件化设计规范

提高Tomcat启动速度

如何学习源码

连接器

NioEndpoint组件：Tomcat如何实现非阻塞I/O

Nio2Endpoint组件：Tomcat如何实现异步IO

AprEndpoint：Tomcat APR提高IO性能的秘密

Executor

Tomcat如何支持WebSocket

Jetty的EatWhatYouKill线程策略

Tomcat和Jetty中的对象池技术

Tomcat Jetty的高性能 高并发之道

内核如何阻塞与唤醒线程

容器

Host容器 Tomcat如何实现热加载和热部署

Context容器（上）:Tomcat如何打破双亲委托机制

Context容器（中） Tomcat如何隔离Web应用的类

Context容器（下） Tomcat如何实现Servlet规范

Tomcat如何支持异步Servlet

Spring Boot如何使用内嵌式的Tomcat和Jetty

Jetty如何实现具有上下文信息的责任链

Spring中的设计模式

Manager组件：Tomcat的Session管理机制解析

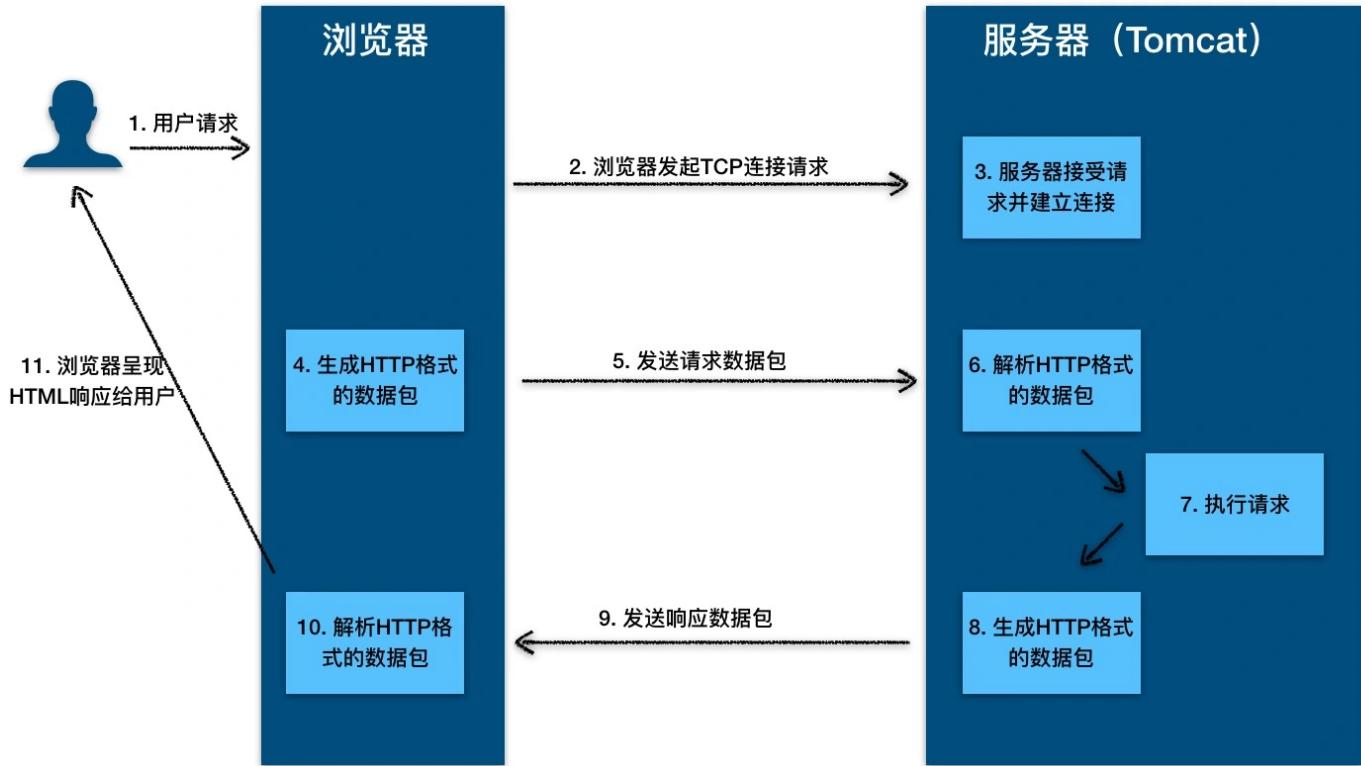
## 基础

---

Web容器：HTTP服务器 + Servlet容器

- HTTP服务器：负责监听、解析、处理HTTP请求、返回HTTP响应
- Servlet容器：负责管理Servlet的生命周期
- Spring框架：是对Servlet的封装，本身就是一个Servlet

HTTP工作流程：

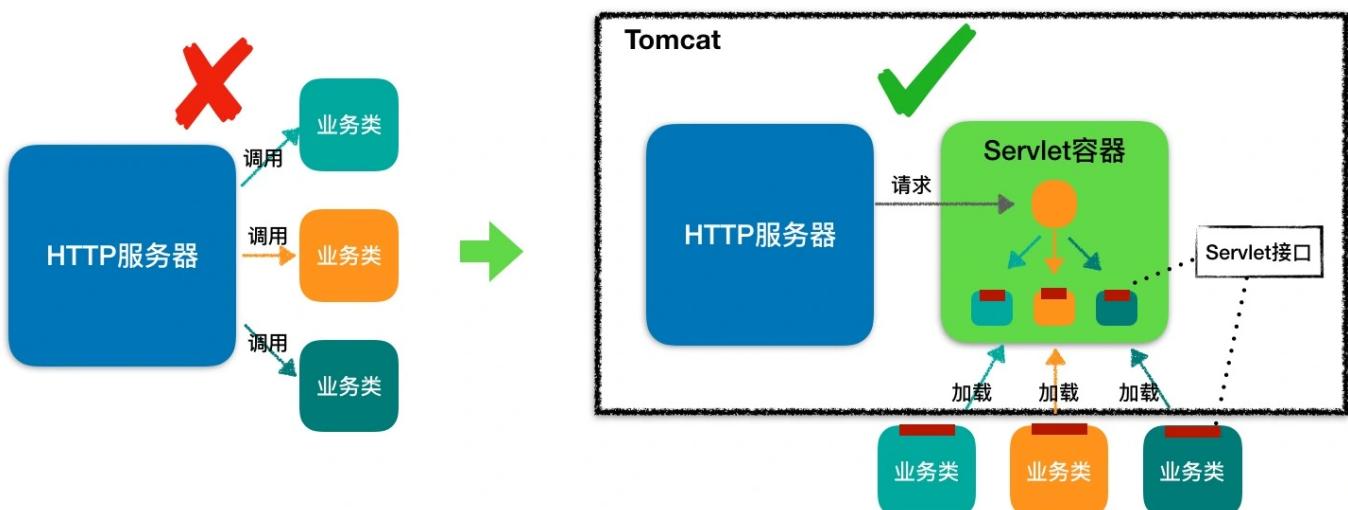


HTTP是无状态的，所以为了保存会话状态引入了Cookie和Session技术

- Cookie: HTTP的请求头，本质上是存储在用户本地的文件，里面包含了每次请求所需传递的信息
- Session: 数据存储在服务端，每个sessionId对应一个session对象

## Servlet规范与Servlet容器

解耦HTTP服务器与Servlet业务类：



Servlet规范：Servlet容器 + Servlet接口

```

// Servlet接口
public interface Servlet {
    void init(ServletConfig config) throws ServletException;
    // 可以拿到web.xml中的设置的参数
    ServletConfig getServletConfig();
    // ServletRequest是对HTTP请求的封装
    // ServletResponse是对HTTP响应的封装 本质上是对通信协议的封装
    void service(ServletRequest req, ServletResponse res) throws ServletException,
    IOException;

    String getServletInfo();

    void destroy();
}

```

工作流：

1. HTTP服务器将请求封装为 `ServletRequest` 对象，将请求对象传递给Servlet容器
2. Servlet容器拿到请求后，根据URL和Servlet的映射关系，找到对应的Servlet，如果该Servlet还没有加载，则利用反射机制创建该类，并调用该类的init方法完成初始化
3. 调用Servlet的service方法处理请求
4. 将 `ServletResponse` 对象返回给HTTP服务器
5. HTTP服务器将响应对象解析为响应报文，发送给客户端

**Web应用目录结构:**

```

| - MyWebApp
|   | - WEB-INF/web.xml      -- 配置文件，用来配置Servlet等
|   | - WEB-INF/lib/          -- 存放Web应用所需各种JAR包
|   | - WEB-INF/classes/     -- 存放你的应用类，比如Servlet类
|   | - META-INF/             -- 目录存放工程的一些信息

```

Servlet规范规定：一个Web应用会对应一个 `ServletContext` 接口，Web应用部署好后，servlet容器在启动时会加载Web应用，并为该应用创建一个唯一的全局对象 `ServletContext`，不同的Servlet可以通过ServletContext共享数据

Servlet规范提供的扩展点：

- Filter
- Listener: 监听Servlet容器中发生的各种事件

三个容器：

- Servlet容器：用于管理Servlet生命周期的

- Spring容器：用于管理Spring bean (service dao) 生命周期的
- Spring MVC容器：用于管理Spring MVC Bean(controller)生命周期的

## Web容器完整启动过程

Tomcat/Jetty启动，对每个WebApp，依次执行初始化工作

1. 每个webapp都会对应一个classLoader和ServletContext
2. ServletContext初始化时，会扫描 web.xml 中的Listener、filter、servlet配置
3. 如果**Listener**中配有Spring的ContextLoaderListener，ContextLoaderListener会获取webapp的状态信息，会在ServletContext初始化时，对Spring IOC容器进行初始化，并将容器存放到ServletContext中
4. 如果Servlet中配有SpringMVC的DispatcherServlet，DispatcherServlet初始化时（即第一次请求到达时），会初始化SpringMVC容器，SpringMVC容器通过ServletContext获取Spring容器，并将其设置为自己的父容器，子容器可以访问父容器(controller可以访问service)，初始化完毕后，DispatcherServlet处理MVC中URL 和servlet的映射关系

## 手动实现一个Servlet

步骤：

1. 下载tomcat
2. 编写一个继承 HttpServlet 的类 重写 doGet doPost 方法

```
public class MyServlet extends HttpServlet {
    // 模板方法设计模式
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
        System.out.println("get method");
        // setContentType要在getWriter前 否则编码格式不起作用
        resp.setContentType("text/html;charset=utf-8");
        PrintWriter writer = resp.getWriter();
        writer.println("<h1>hello daxiao you send a get method</h1>");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
        System.out.println("post method");
        // setContentType要在getWriter前 否则编码格式不起作用
        resp.setContentType("text/html;charset=utf-8");
        PrintWriter writer = resp.getWriter();
        writer.println("<h1>hello daxiao you send a post method</h1>");
    }
}
```

3. 编译Java -> class

```
javac -cp servlet-api.jar MyServlet.java
```

#### 4. 建立web应用的目录结构并配置web.xml

```
<servlet>
    <servlet-name>myServlet</servlet-name>
    <servlet-class>MyServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>myServlet</servlet-name>
    <url-pattern>/myServlet</url-pattern>
</servlet-mapping>
```

#### 5. 部署web应用

tomcat负责实例化servlet

```
/webapp/MyWebApp/WEB-INF/classss/MyServlet.class
```

```
/webapp/MyWebApp/WEB-INF/web.xml
```

#### 6. 启动tomcat

```
./startup.sh
```

#### 7. 浏览器访问 查看tomcat日志

Tomcat目录结构：

- /bin 存放启动或者关闭的脚本文件
- /conf 全局配置文件 最重要的是 server.xml
- /lib 存放Tomcat以及所有Web应用都可以用的jar包
- /log Tomcat日志
- /webapps 应用目录

## 系统架构

先了解需求 -> 设计系统

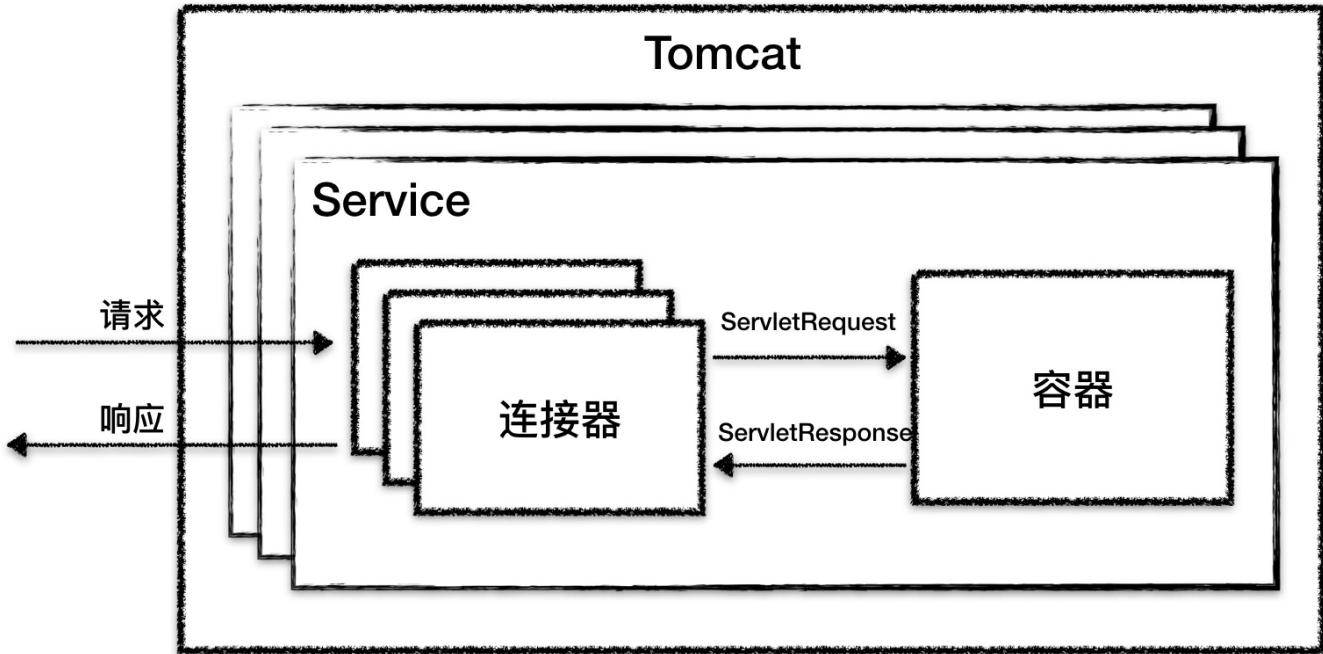
**Web容器的需求：**

- HTTP服务器：处理Socket连接，负责网络字节流和Request Response对象的转化
- Servlet容器：加载和管理Servlet，具体处理Request请求

**系统的两大组成部分：**

- 连接器 Connector

- 容器 Container



## 连接器的设计

连接器为容器屏蔽了IO模型和应用层协议的细节，为容器提供统一的ServletRequest对象

连接器的功能(按步骤来):

1. 监听网络端口
2. 接收网络连接请求
3. 读取网络请求字节流
4. 根据具体的应用层协议 (HTTP/AJP) 解析字节流，生成统一的 `Tomcat Request` 对象
5. 将 `Tomcat Request` 对象转化为 `ServletRequest` 对象
6. 调用 `Servlet 容器`，获取 `ServletResponse`，
7. 将 `ServletResponse` 对象转化为 `Tomcat Response` 对象
8. 将 `Tomcat Response` 对象转化为网络字节流
9. 将响应字节流写回给浏览器

连接器主要分为三个高内聚的模块：

- 网络通信 `Endpoint`

Tomcat支持的IO模型：

- NIO
- NIO.2
- APR

组成部分:

- `Acceptor` 监听Socket连接请求
- `SocketProcessor` 处理接收到的Socket请求, 是`Runnable`的实现子类, 在run方法里调用协议解析组件`Processor`, 被提交给线程池进行处理
- 应用层协议解析 `Processor`

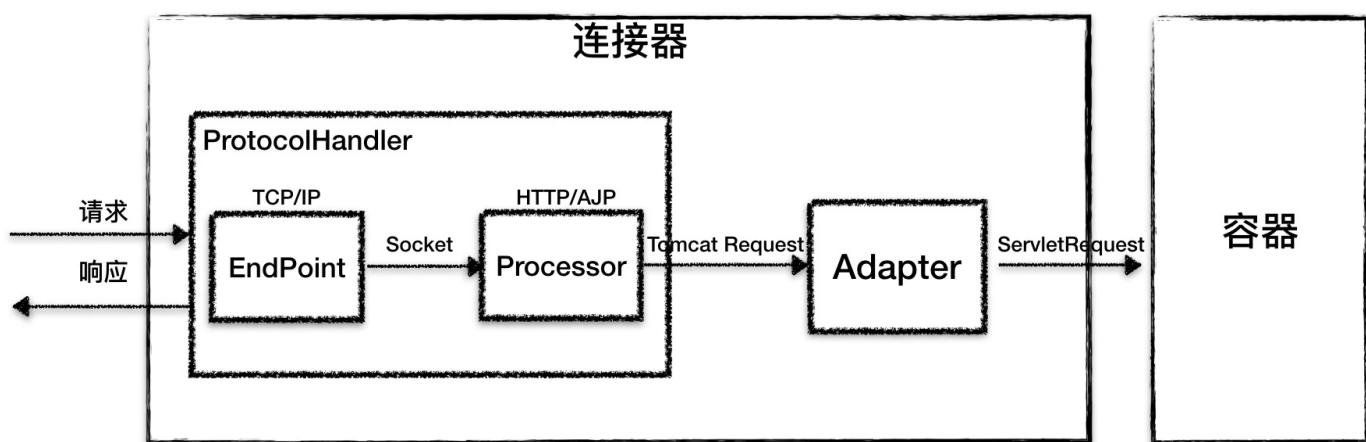
Tomcat支持的应用层协议:

- HTTP/1.1
- AJP
- HTTP/2

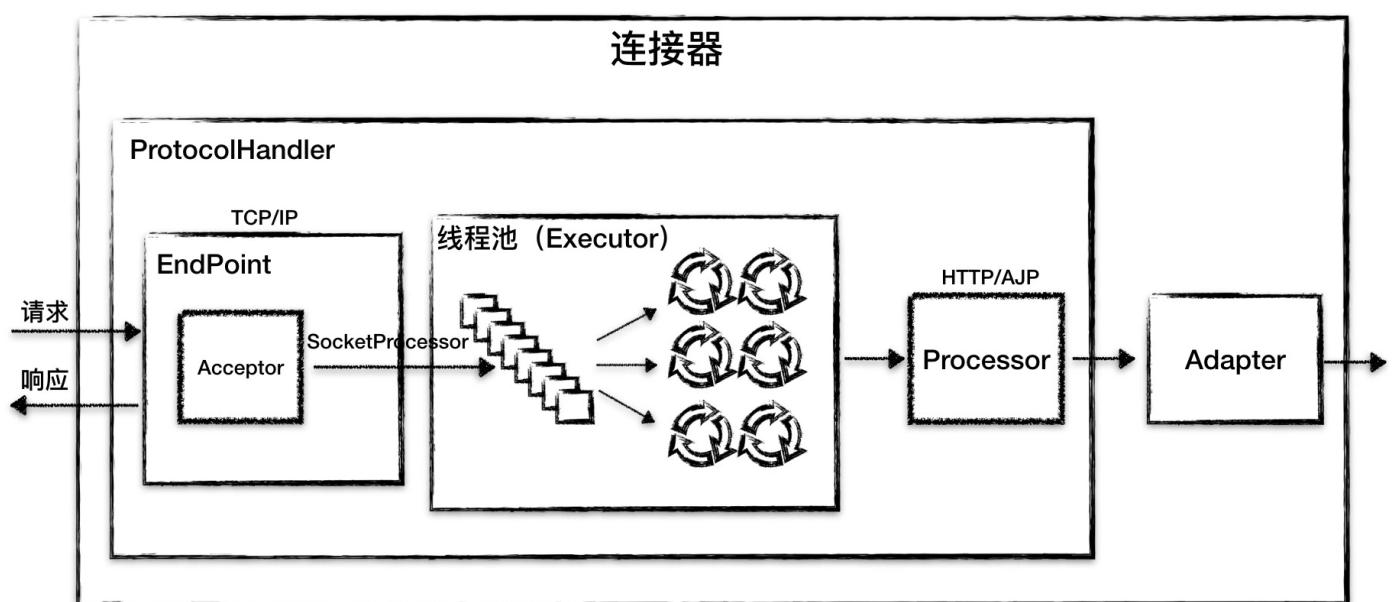
主要功能: 将字节流和Tomcat Request/Response对象的转化

- Tomcat Request/Response 和 Servlet Request/Response的转化 `Adapter`

ProtocolHandler = EndPoint + Processor



连接器细化:



# 容器的设计

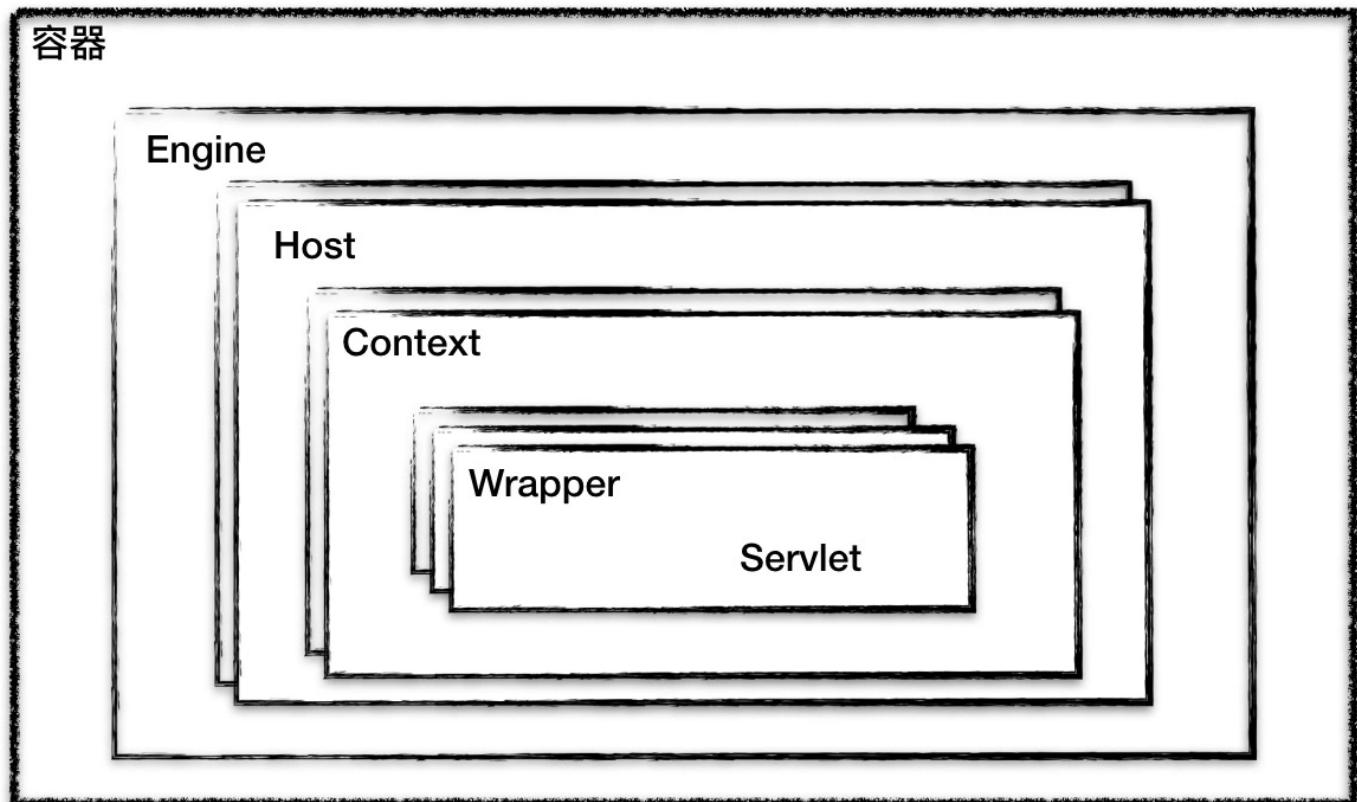
容器的结构：

Engine 对应一个Service

Host 对应一个虚拟主机

Context 对应一个Web应用

Wrapper: Servlet



```
<Server port="8005" shutdown="SHUTDOWN"> 可以包含多个service
  <!-- A "Service" is a collection of one or more "Connectors" that share
      a single "Container". Note: A "Service" is not itself a "Container",
      so you may not define subcomponents such as "Valves" at this level.
      Documentation at /docs/config/service.html
  -->
<Service name="Catalina"> 可以包含一个Engine多个 连接器
  <Connector port="8080" protocol="HTTP/1.1"
             connectionTimeout="20000"
             redirectPort="8443" /> 连接器 通信接口
<Engine name="Catalina" defaultHost="localhost"> 处理Service下的所有请求
  <Host name="localhost" appBase="webapps"
        unpackWARs="true" autoDeploy="true">
    <Context> Web容器
    </Context>
```

```
</Host> 处理特定下host下的请求 可以包含多个Context  
</Engine>  
</Service>  
</Server>
```

Tomcat管理容器的方法：组合模式，一致化树的分支和叶子结点操作

```
// Engine Host Context Wrapper都实现Container接口  
public interface Container extends Lifecycle {  
    public void setName(String name);  
    public Container getParent();  
    public void setParent(Container container);  
    public void addChild(Container child);  
    public void removeChild(Container child);  
    public Container findChild(String name);  
}
```

定位Servlet的过程：通过 Mapper 组件

1. 根据协议和端口号找到Service和Engine
2. 根据域名找到Host
3. 根据URL路径找到Context组件和Wrapper

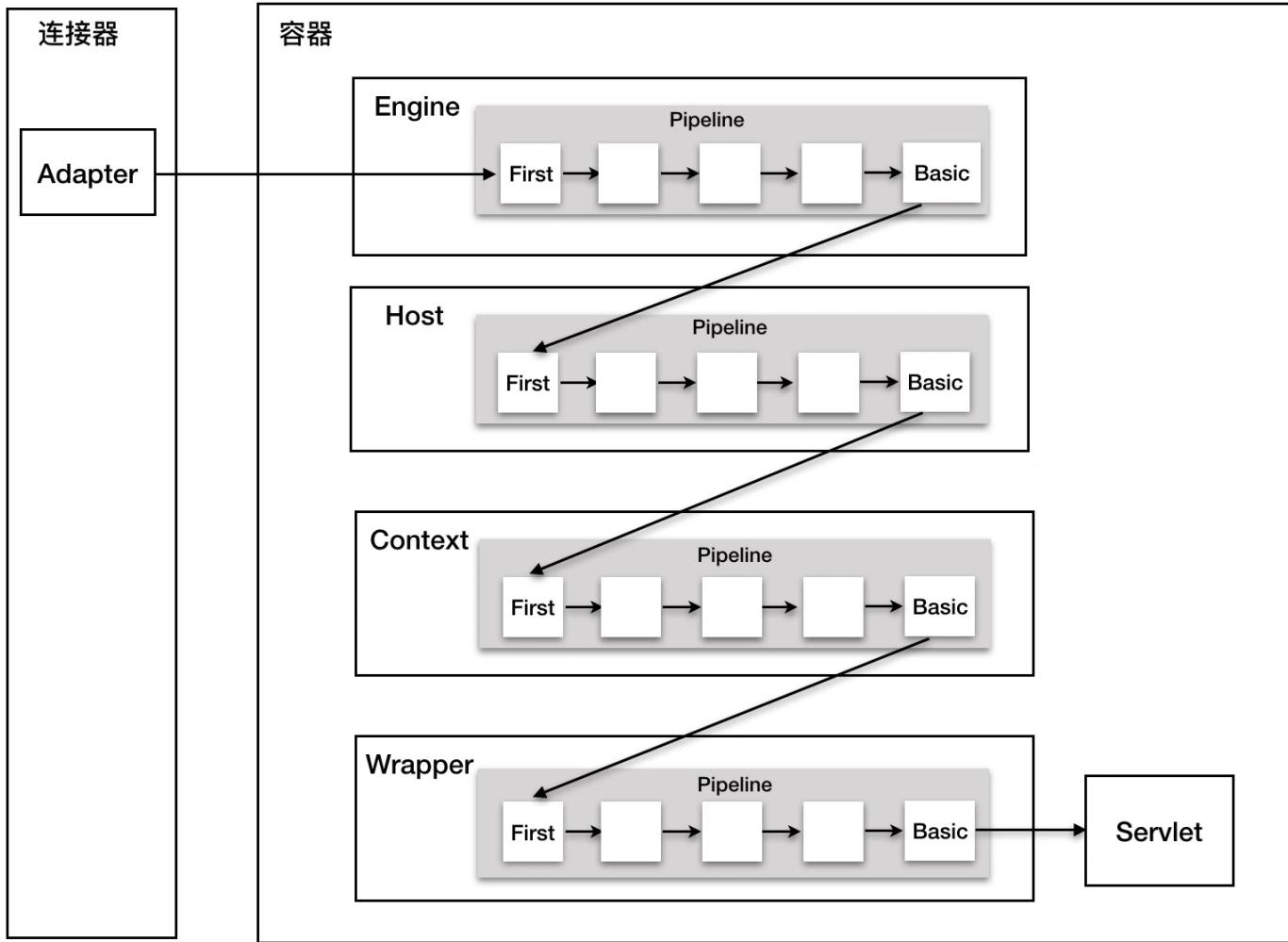
处理请求的过程：父容器处理完之后传递给子容器——责任链模式

链表结点：Valve

```
public interface Valve {  
    public Valve getNext();  
    public void setNext(Valve valve);  
    public void invoke(Request request, Response response)  
}
```

链表：

```
public interface Pipeline extends Contained {  
    public void addValve(Valve valve);  
    public Valve getBasic();  
    public void setBasic(Valve valve);  
    public Valve getFirst();  
}
```



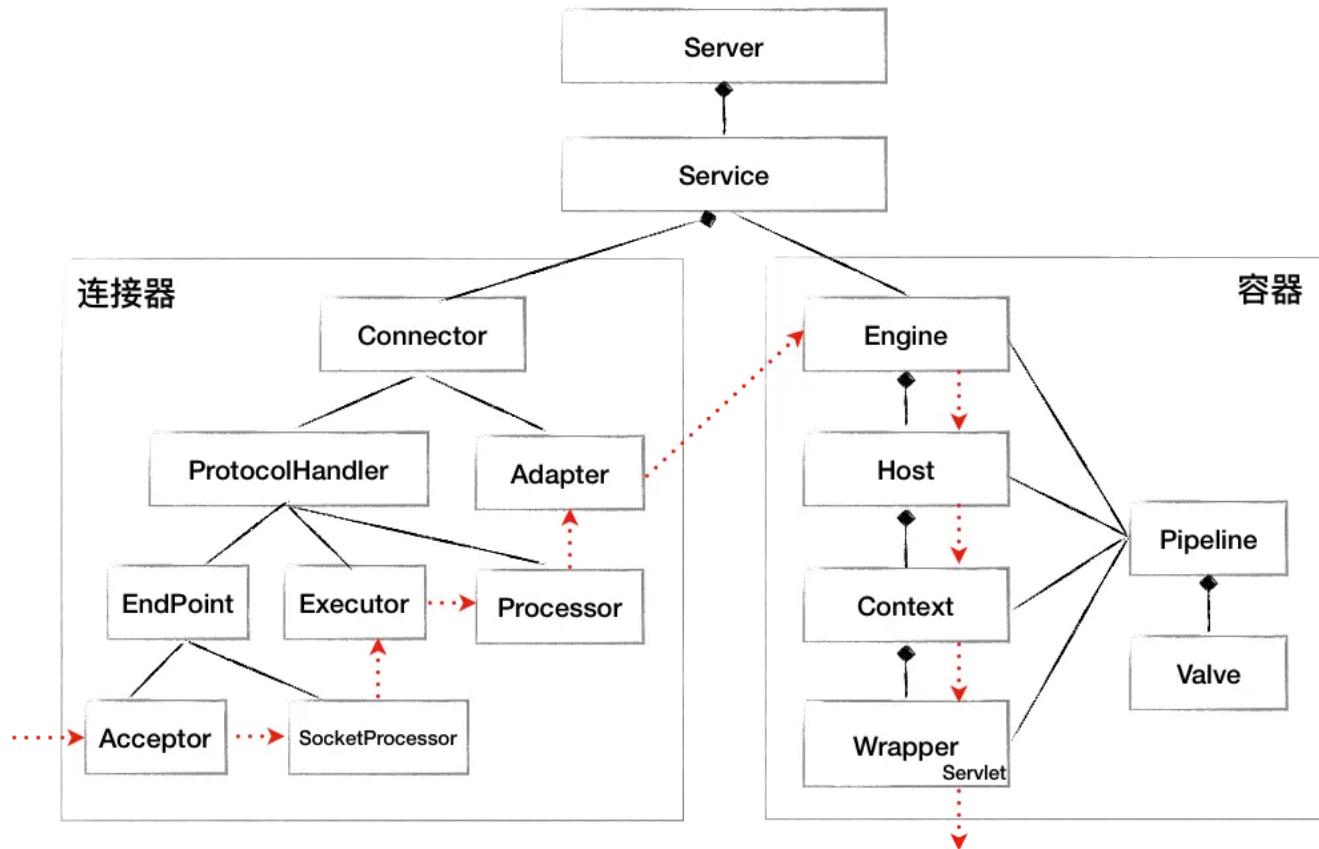
Adapter调用Engine中的第一个Valve

Wrapper中的最后一个Valve会创建一个FilterChain 最终调用到Servlet的Service方法

## Tomcat实现一键式启停

[嵌入式方式启动Tomcat](#)

一个HTTP请求在Tomcat中的流转过程



设计系统时需要解决的问题：

- 如何管理组件的创建 初始化 启动 停止 销毁（生命周期管理）
- 如何方便的添加和删除组件
- 如何做到组件启动和停止不重复 不遗漏

Tomcat组件之间的关系：

- 大小关系：大组件管理小组件 server管理service service管理connector和engine
- 内外关系：外层组件调用内层组件完成功能

组件创建顺序（由组件依赖关系决定）：

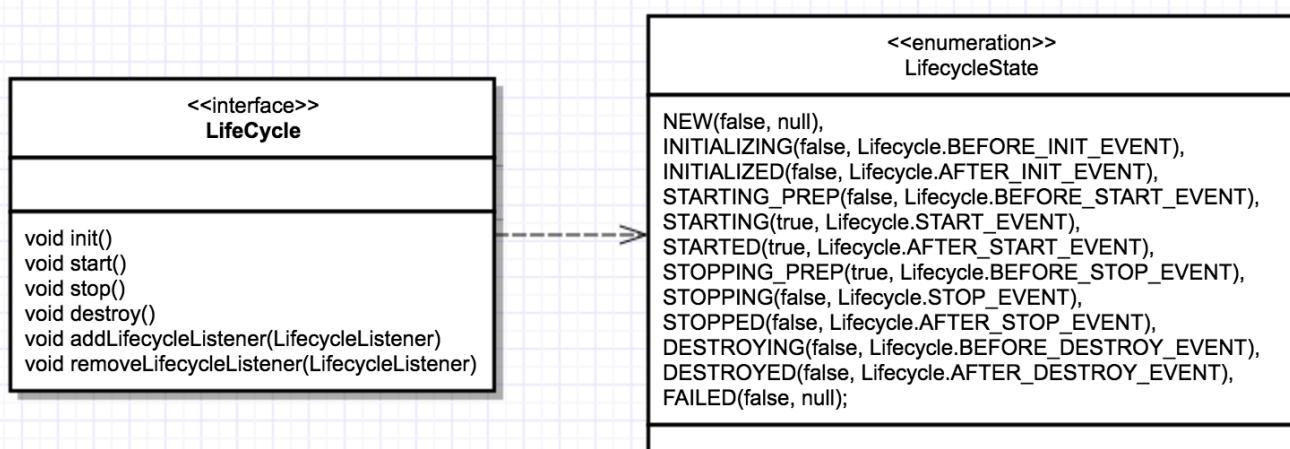
- 先创建子组件 再创建父组件 子组件需要被注入到父组件中
- 先创建内层组件 再创建外层组件 内层组件需要被注入到外层组件中

一键式启停的关键：Lifecycle接口（每个组件都实现该接口）

组合模式：父组件调用 `init()` 方法的时候，会创建子组件，并调用子组件的 `init()` 方法，`start()` 方法也是同理，这样只要 `Server.init() start()` 整个Tomcat就都启动起来了

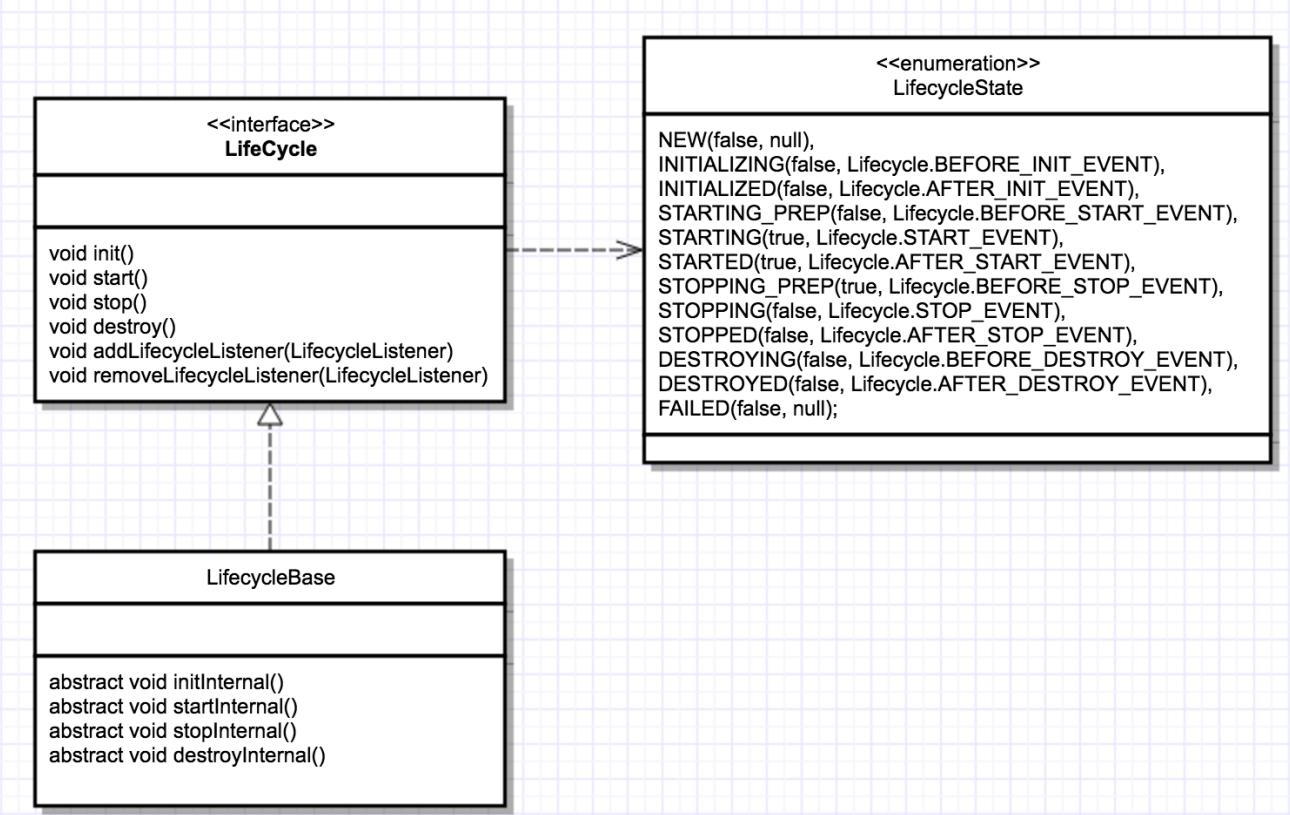
可扩展性：LifeCycle事件（观察者模式）

当我们需要往init或者start等方法里面加逻辑时，会不可避免地违反开闭逻辑，这个时候我们可以把init看成是状态转变的一个事件，给这个事件添加监听器，低耦合，非侵入式地添加功能



代码复用：LifeCycleBase 抽象基类（模板设计模式）

存放公共的逻辑：生命状态的转变和维护、生命事件的触发、监听器的添加和删除



```

// LifeCycleBase
@Override
public final synchronized void init() throws LifecycleException {

```

```
//1. 状态检查
if (!state.equals(LifecycleState.NEW)) {
    invalidTransition(Lifecycle.BEFORE_INIT_EVENT);
}

try {
    //2. 触发INITIALIZING事件的监听器
    setStateInternal(LifecycleState.INITIALIZING, null, false);

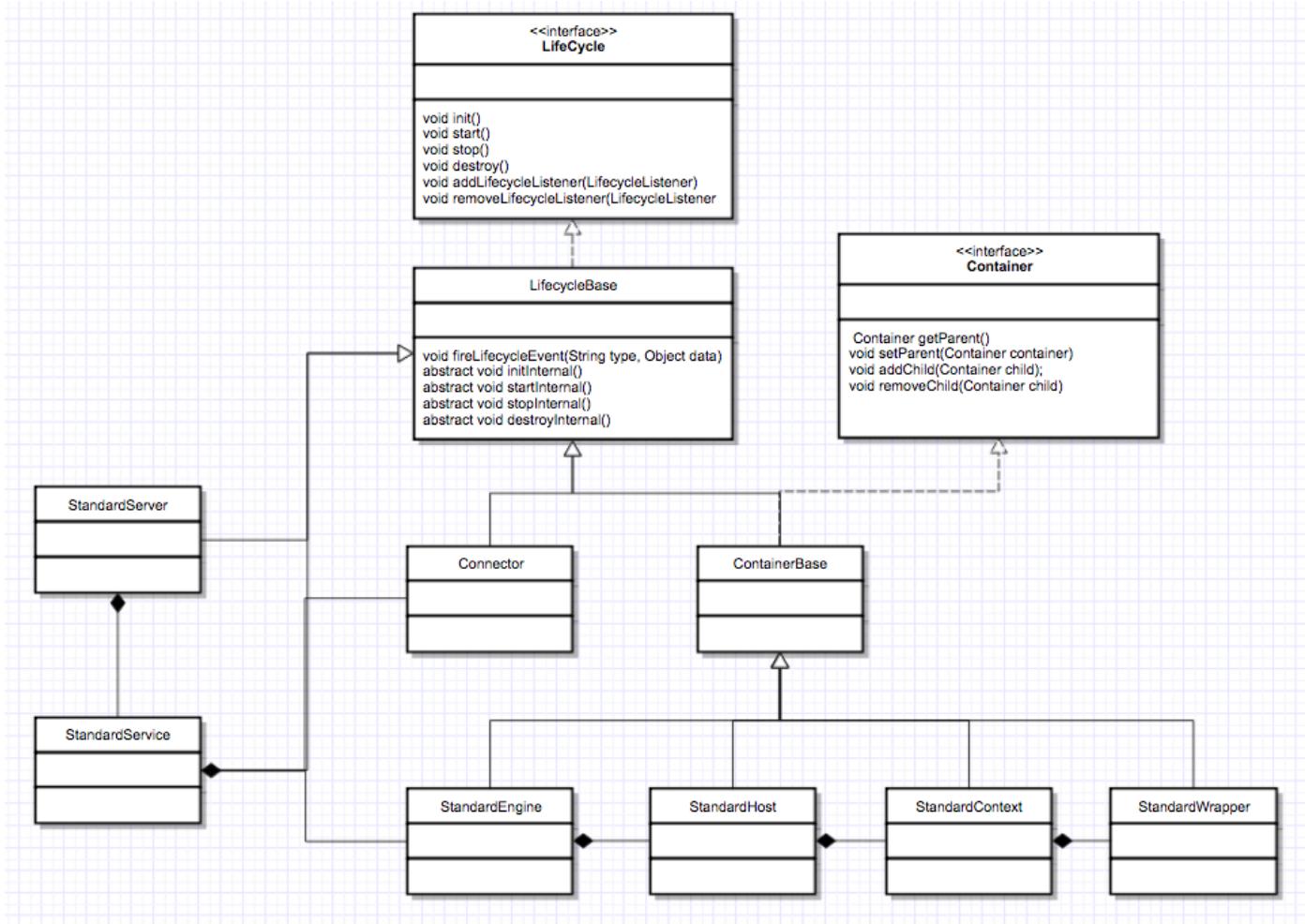
    //3. 调用具体子类的初始化方法
    initInternal();

    //4. 触发INITIALIZED事件的监听器
    setStateInternal(LifecycleState.INITIALIZED, null, false);
} catch (Throwable t) {
    ...
}
```

监听器的注册时间：

- 父组件创建子组件时注册的
- Tomcat启动时会解析server.xml 创建里面的监听器并注册到组件中

生命周期管理图



## Tomcat的高层组件负责做的事情

### Tomcat启动过程

1. `./startup.sh` 启动JVM，并运行Tomcat的启动类 `Bootstrap`
2. `Bootstrap`中会初始化Tomcat自己的类加载器，并创建Catalina
3. Catalina也是一个启动类，会解析 `server.xml` 文件，创建相应的组件，创建Server 并调用其start方法
4. Server管理service 调用service的start方法
5. Service 管理连接器和engine 调用连接器和engine的start方法

### Catalina

主要任务：创建Server，并调用其init和start方法

具体实现：

- 解析`server.xml`,将里面配置的组件创建出来，再调用server的init和start方法
- 创建并注册关闭钩子：观察者模式(注册某个事件对应的动作)，JVM退出时 执行`catalina.stop()`

## Server

主要任务：

- 管理Service，用一个数组存储，且数组长度动态扩增，每次增大一个空间
- await方法中监听8005端口，监测socket中传来的数据是否是 SHUTDOWN，如果是就退出轮询，进入stop流程

## Service

- 管理connector和engine
- MapperListener：监听器，监听容器的变化，并信息更新到Mapper中

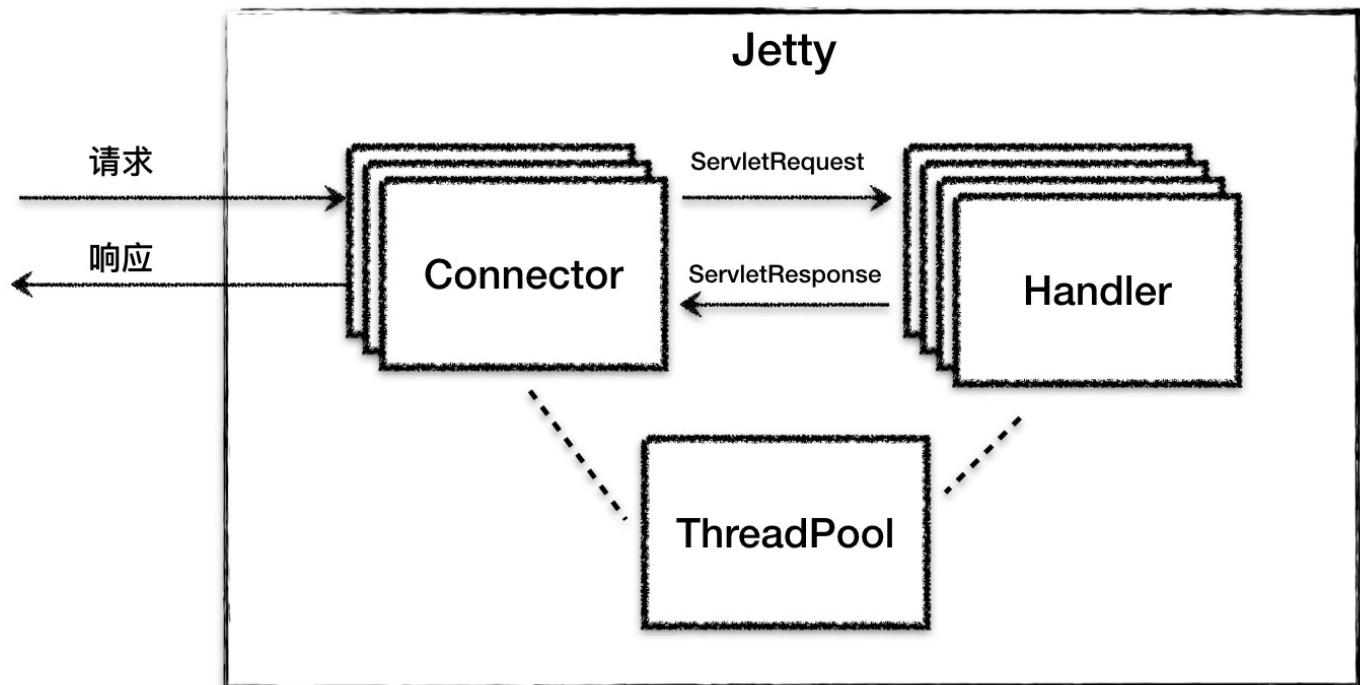
## Engine

- 管理host组件
- 启动host组件时用的是线程池（ContainerBase里面的 所以所有容器都是用线程池）
- 处理请求，将请求转发给某个host处理

# Jetty的Connector

[Jetty源码查看](#)

整体结构



- Connector：实现HTTP服务器的功能

- Handler: 可插拔, 需要支持servlet就配置使用servletHandler, 需要支持session就配置使用sesionhandler
- Threadpool: 全局线程池(Tomcat中每个连接器都有自己的线程池)
- Server: 负责创建并初始化以上三个组件

**Connector:** 用于封装IO模型和应用层协议

服务端IO通信主要做的三件事	Jetty中对应的组件
监听连接	Acceptor
IO事件查询	SelectorManager
数据读写	Connection

## Acceptor

- 是一个Runnable, 通过全局线程池来执行
- 通过阻塞方式接收连接
- 接收连接成功后 设置channel为非阻塞模式 并交给selectorManager处理

## SelectorManager

selectorManager处理channel的流程

1. 从自身的selector数组中选择一个selector来处理这个channel
2. 将channel注册到这个selector上 拿到selectionKey
3. 创建一个endpoint和connection 并将其和selectionkey绑定在一起

## Connection

- 是endpoint的一个内部类, 一个runnable
- 负责具体协议的解析 得到request对象
- 调用handler容器进行处理

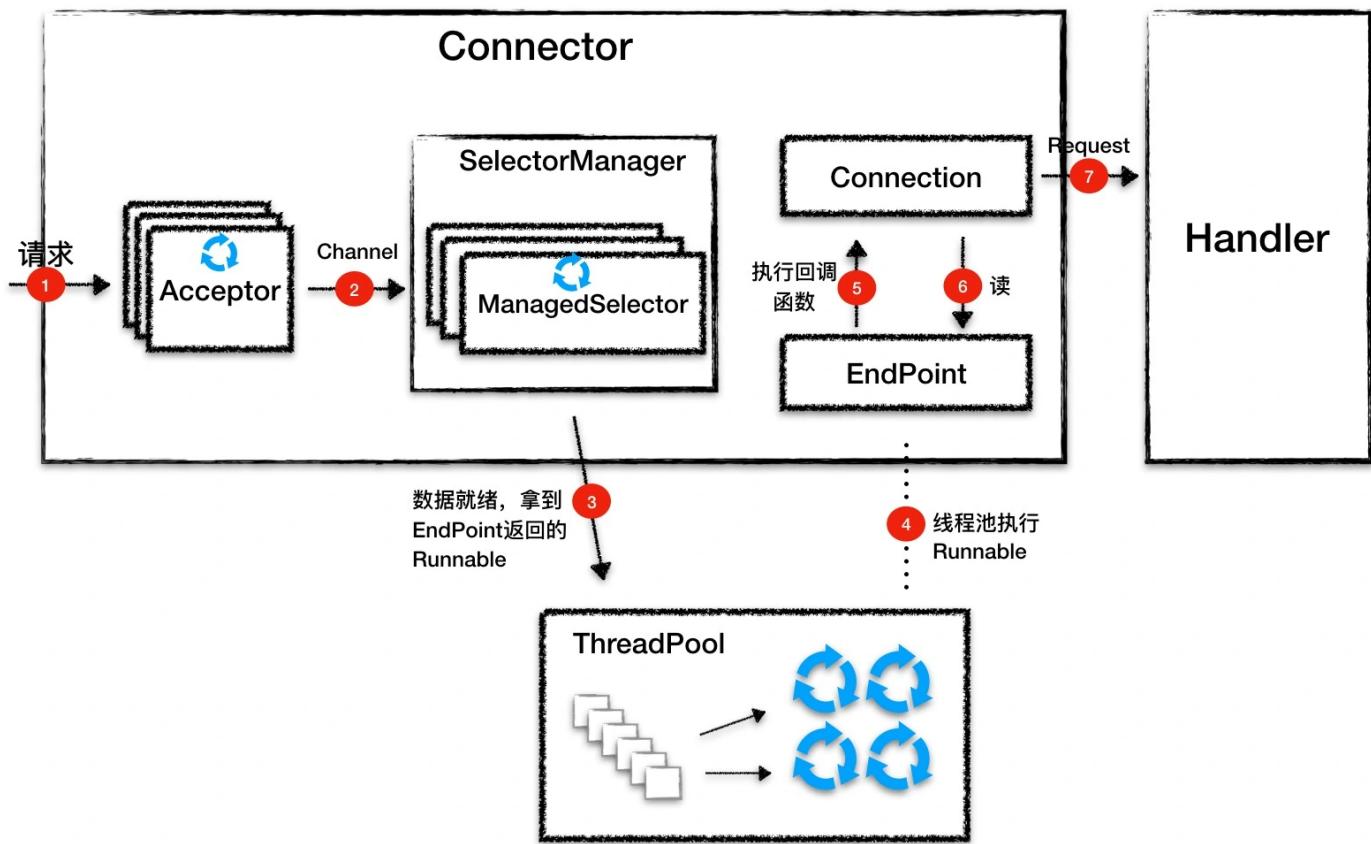
请求的处理:

- 向endpoint中注册回调方法, 告诉endpoint等数据到了就调用这些方法
- 回调方法中, 调用endpint的接口读取数据, 解析HTTP信息, 封装成request对象

响应的处理:

通过endpoint将数据写入到channel中

## Connector工作流程



1. Acceptor监听连接请求，当有连接请求就接收连接，一个连接对应一个channel，acceptor将channel交给managedSelector处理
2. managedSelector将channel注册到selector上，并创建一个endpoint和connection跟channel绑定，接着不断监测IO事件
3. IO事件到了就调用endpoint的方法拿到一个runnable 扔给线程池执行
4. runnable中解析读到的数据，生成request对象交给handler处理

设计特点：使用了回调函数

## Jetty的Handler

Handler接口

```
public interface Handler extends LifeCycle, Destroyable
{
    //处理请求的方法
}
```

```

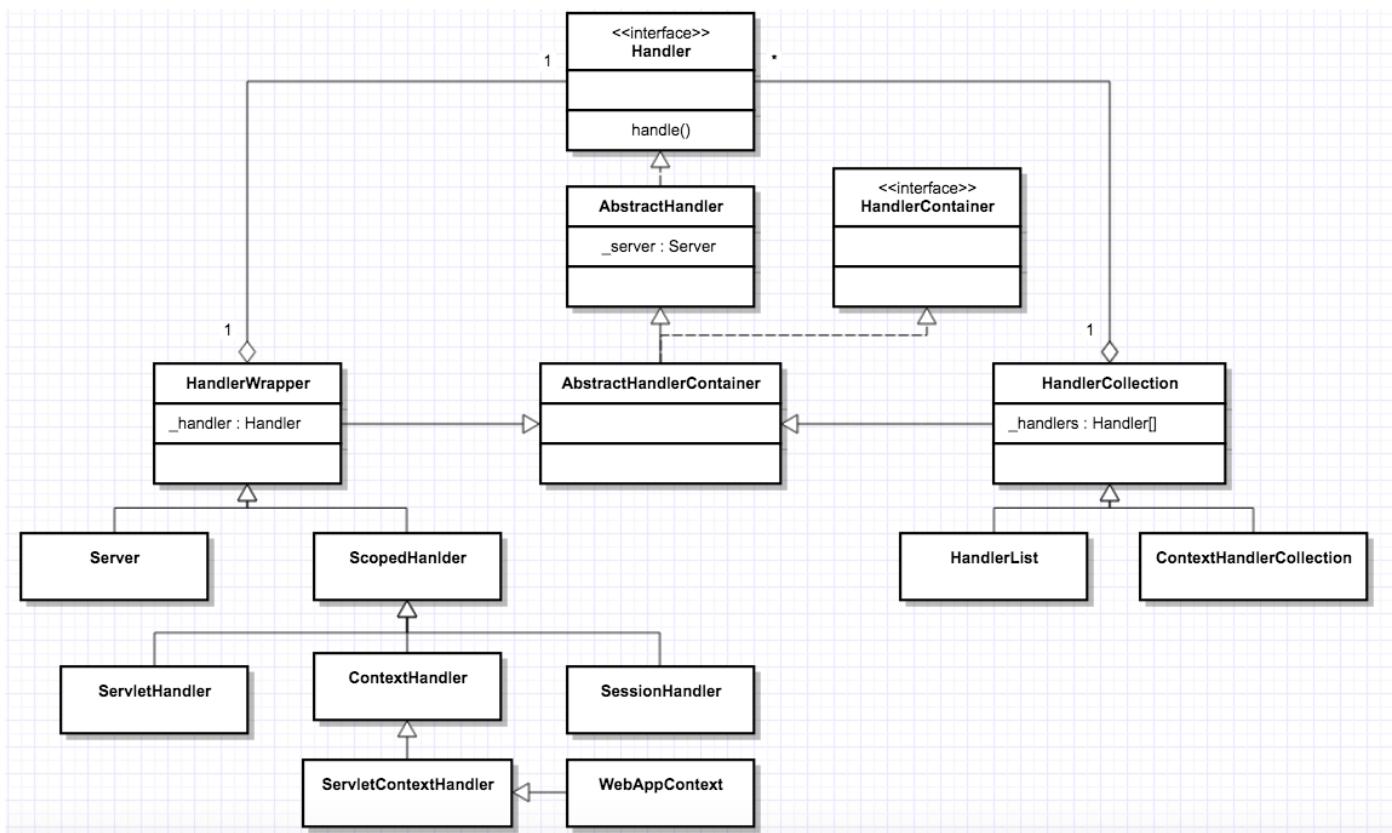
public void handle(String target, Request baseRequest, HttpServletRequest request,
HttpServletRequest response)
throws IOException, ServletException;

//每个Handler都关联一个Server组件，被Server管理
public void setServer(Server server);
public Server getServer();

//销毁方法相关的资源
public void destroy();
}

```

## Handler继承关系



## 关键点解析：

- `AbstractHandlerContainer`：因为handler要把请求传递给其他的handler（实现责任链模式的链式调用）所以要持有其他handler对象的引用，所以叫container，它的两个子类的区别就在于拥有的handler个数不同
- `Server`：Handler模块的入口
- `scopedHandler`：具有上下文信息，其子类用来实现servlet规范(servlet listener filter)
- `HandlerCollection`：维护了一个Handler数组，比如server中就有一个handlercollection，其中的每个handler都对应一个web应用的handler入口

## Handler类型

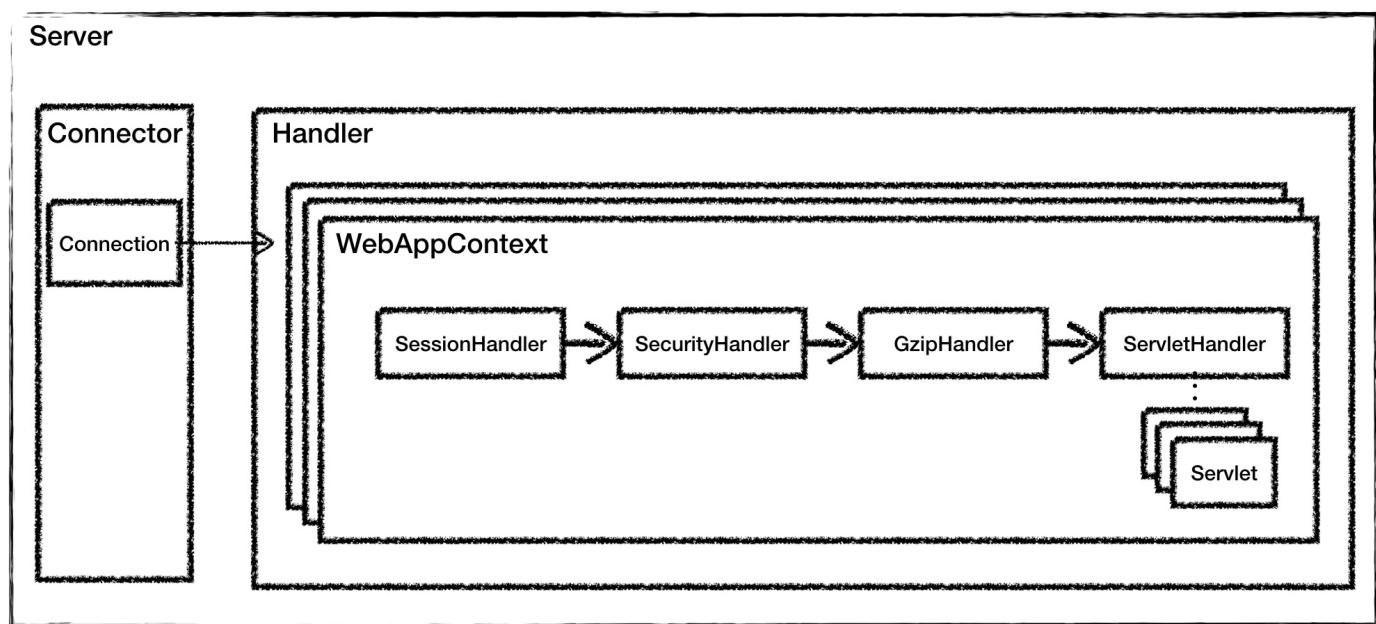
- 协调handler：负责转发请求，如handlerCollection

- 过滤器handler：自己完全处理请求后再转发请求，如handlerWrapper
- 内容handler：会真正调用servlet处理请求，并生成响应的内容

Servlet规范中需要有的组件

- Context
- Servlet
- Listener
- Filter
- Session

调用过程



定制化方法

将通用组件抽象成了Handler，添加或者删除功能时，就在执行链中添加或者删除handler即可（微内核 + 插件的设计思想）

## 组件化设计规范

Web容器如何实现组件化设计

- 根据需求，按照高内聚低耦合的原则将不同的功能进行拆分，每个组件抽象成接口
- 组件的载体(Server) + 组件的连接方式（职责链模式）
- 用户可以通过配置来组装组件

组件的创建

1. 扫描配置
2. 解析配置
3. 反射创建

实例化对象之前Web容器需要组件的类加载到JVM中 Tomcat实现了自己的类加载器  
且给每个Web应用都分配了类加载器 (Spring用的类加载器就是这个)

## 组件的生命周期管理

- 父组件负责子组件的初始化 启停 销毁 一键式启停 (组合模式)
- 定义了生命周期状态，将状态的转化作为观察的事件，从而触发在事件上注册的行为 (观察者模式)

## 组件的骨架抽象类和模板模式

在抽象类中实现通用逻辑，并提供扩展点 (由子类实现的抽象方法)

- 代码复用
- 扩展性

## 提高Tomcat启动速度

- 清理Tomcat
  - 清理不必要的Web应用，每次Tomcat重新启动都会重新部署所有的Web项目
  - 清理xml配置文件，减少解析所需的时间
  - 清理Jar文件
  - 及时清理日志
- 禁止Tomcat TLD扫描
- 关闭Websocket支持
- 关闭JSP支持
- 禁止Servlet扫描注解
- 配置Web-fragment扫描
- 随机数熵源优化
- 并行启动多个Web应用

# 如何学习源码

开源框架和中间件：

- 服务接入层：反向代理Nginx, API网关Node.js
- 业务逻辑层：Web容器Tomcat, Jetty 应用层框架Spring SpringMVC SpringBoot MyBatis
- 数据缓存层：内存数据库Redis 消息中间件Kafka
- 数据存储层：关系型数据库MySQL 非关系型数据库MongoDB 文件存储HDFS，搜索分析引擎Elasticsearch
- RPC框架：Spring Cloud Dubbo
- 网络通信：Netty 分布式协调：ZooKeeper

先学习简单或者熟悉的的框架或中间件，Tomcat Spring，把一个学透然后学其他的就能总结出套路

学习源码的要点：

- 弄清楚中间件的核心功能是什么
- 完成这些核心功能涉及到的代码流程先研究 不要深入到其他细节
- 将源码跑起来 打断点调试看变量的值和调用栈
- 带着问题去学习源码 了解某个功能如何实现的 自己设计这些功能时会怎么做 弄清楚细节后及时记录下来 画流程图或者类图
- 参考官方文档 看大概的设计思路 看网上博客 参考别人的分析

官方文档 -> 使用 -> 带着问题去研究源码

## 连接器

### NioEndpoint组件：Tomcat如何实现非阻塞I/O

UNIX下的IO模型

- 同步阻塞
- 同步非阻塞
- IO多路复用
- 信号驱动
- 异步

IO:

计算机内存与外部设备拷贝数据的过程

IO模型需要解决的问题：

程序通过CPU向外部设备发送一个读指令后，数据到达内存需要一定的时间，这时程序面临的选择有：

- 把CPU让出去（阻塞）
- 让CPU不断轮询数据有没有到达（非阻塞）

## JavaIO模型

当用户线程发起IO操作后，读取网络数据会经历两个步骤

1. 用户线程等待内核将数据从网卡拷贝到内核空间
2. 内核将数据从内核空间拷贝到用户空间

IO模型	实现网络数据读取的方式
同步阻塞IO	用户线程在进行read调用后阻塞了，让出CPU，之后内核等待网卡数据到来，将数据从网卡拷贝到内核空间，再将数据从内核空间拷贝到用户空间，内核唤醒用户线程，read调用返回
同步非阻塞IO	用户线程不断发起read调用，在数据到达内核空间之前一直返回失败（此时用户线程在轮询时可以做其他的事），到达之后，此时read调用就会阻塞，之后的逻辑跟阻塞IO一样
IO多路复用	将用户线程的读取操作分为两个步骤，线程首先发起select调用，向内核中的多个channel查询数据是否准备好了，准备好了之后，再发起read调用，阻塞等待数据从内核空间拷贝到用户空间
异步IO	用户线程发起read调用时，会注册一个回调函数，read立即返回，内核将数据准备好后，会调用注册的回调函数完成处理（整个过程用户线程一直没有阻塞）

## NioEndpoint组件

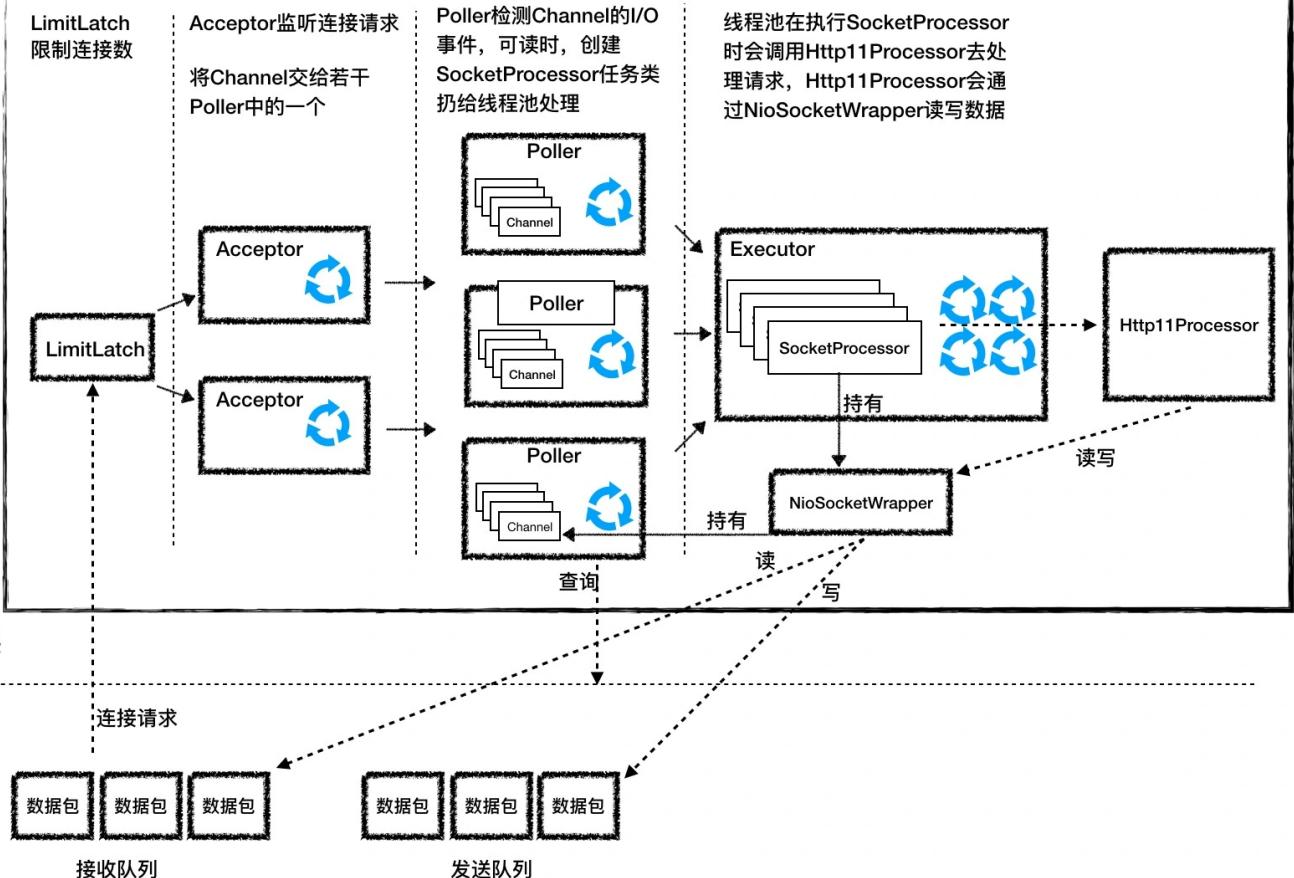
实现了IO多路复用模型

Java中多路复用器的使用步骤：

1. 创建一个Selector，在它身上注册各种感兴趣的事件，调用select，等待感兴趣的事件发生
2. 感兴趣的事件发生了，比如说可读了，那就创建一个新的线程从channel中读数据

组成：

## NioEndPoint



- LimitLatch: 限制连接数
- acceptor: 监听连接请求, 将channel交给一个poller(跑在一个单线程里面, 在一个死循环里面用accept方法接收新请求)
- Poller: 检查channel的IO事件, 可读时创建socketprocessor任务类给线程池处理 (每一个poller就是一个selector, 都维护了一个channel数据, 也是泡在单独的线程里面)
- 任务类run方法中: 调用http11processor处理请求, http11processor通过niosocketwrapper读写数据

## LimitLatch

就类似于一个Semaphore, 用于控制连接个数, 当连接数到达Limit之后操作系统底层还是会接收客户端连接, 但是用户层不再接收

```
public class LimitLatch {
    // AQS是一个基于队列的同步器, 里面维护了一个状态和线程队列
    // 这里的Sync没有用到state作为状态 而是用外部类的count作为状态
    private class Sync extends AbstractQueuedSynchronizer {
        /*
         在AQS的acquireSharedInterruptibly()方法中
         会调用此方法 返回值>0表明该线程获取到了资源
         <0表明需要将该线程挂起 并加入到阻塞队列中
        */
    }
}
```

```

/*
@Override
protected int tryAcquireShared() {
    long newCount = count.incrementAndGet();
    if (newCount > limit) {
        count.decrementAndGet();
        return -1;
    } else {
        return 1;
    }
}

/*
    AQS的releaseShared()中会调用此方法
    返回true表明有资源释放 可以唤醒阻塞队列中的线程
*/
@Override
protected boolean tryReleaseShared(int arg) {
    count.decrementAndGet();
    return true;
}

private final Sync sync;
private final AtomicLong count;
private volatile long limit;

//线程调用这个方法来获得接收新连接的许可，线程可能被阻塞
public void countUpOrAwait() throws InterruptedException {
    sync.acquireSharedInterruptibly(1);
}

//调用这个方法来释放一个连接许可，那么前面阻塞的线程可能被唤醒
public long countDown() {
    sync.releaseShared(0);
    long result = getCount();
    return result;
}
}

```

## Acceptor

实现了 Runnable 接口，一个端口号对应一个 serverSocketChannel，多个acceptor共享一个 serverSocketChannel

```

serverSock = ServerSocketChannel.open();
// acceptCount表示操作系统的ACCEPT等待队列长度
serverSock.socket().bind(addr, getAcceptCount());
// 以阻塞方式接收连接(read系统调用后阻塞线程)
serverSock.configureBlocking(true);

```

主要功能：

- `run` 方法中使用 `serverSocketChannel` 的 `accept()` 方法获取新的连接（socketchannel对象）
- 将socketchannel对象封装在pollereven对象中，将其放入poller的queue中（生产者-消费者模式）

源码解析：

- 启动acceptor线程

```

// NioEndpoint
public void startInternal() throws Exception {

    if (!running) {
        // 调用基类AbstractEndpoint的方法
        startAcceptorThreads();
    }
}

// AbstractEndpoint 维护了一个acceptor数组
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();
    }
}

```

- Run方法逻辑

```

public void run() {

    int errorDelay = 0;

```

```
// Loop until we receive a shutdown command
while (running) {

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

    if (!running) {
        break;
    }
    state = AcceptorState.RUNNING;

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

        SocketChannel socket = null;
        try {
            // 调用该endpoint唯一的ServerSocketChannel的accept方法 获取
socketchannel (连接)
            // Accept the next incoming connection from the server
            // socket
            socket = serverSock.accept();
        } catch (IOException ioe) {
            // we didn't get a socket
            countDownConnection();
            if (running) {
                // 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 (running && !paused) {
            // setSocketOptions() will hand the socket off to
            // an appropriate processor if successful
        }
    }
}
```

```

        // 传递socketchannel给setSocketOptions
        if (!setSocketOptions(socket)) {
            closeSocket(socket);
        }
    } else {
        closeSocket(socket);
    }
} catch (Throwable t) {
    ExceptionUtils.handleThrowable(t);
    log.error(sm.getString("endpoint.accept.fail"), t);
}
}

state = AcceptorState.ENDED;
}

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();
        }
        // 轮询从Poller数组中获取一个poller，并将niochannel注册到poller中(socketchannel放到
了niochannel中)
        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;
}

// 将niochannel封装到niosocketwrapper中 将其封装到PollerEvent中 并放入Poller对象维护的一个同步
队列中
public void register(final NioChannel socket) {
    socket.setPoller(this);
    NioSocketWrapper ka = new NioSocketWrapper(socket, NioEndpoint.this);
    socket.setSocketWrapper(ka);
    ka.setPoller(this);
    ka.setReadTimeout(getSocketProperties().getSoTimeout());
    ka.setWriteTimeout(getSocketProperties().getSoTimeout());
    ka.setKeepAliveLeft(NioEndpoint.this.getMaxKeepAliveRequests());
    ka.setReadTimeout getConnectionTimeout());
    ka.setWriteTimeout getConnectionTimeout());
    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);
}

```

## Poller

主要功能：

- 监听多个channel，当事件发生时，生成一个processor对象交给线程池执行
- 检查连接是否过期

```

public class Poller implements Runnable {
    // 实际上就是一个selector
    private Selector selector;
    // event里面封装了socketchannel 多个acceptor线程可能同时对一个poller的queue进行读取 所以需要同步
    private final SynchronizedQueue<PollerEvent> events =
        new SynchronizedQueue<>();
}

```

```

public void run() {

```

```

// Loop until destroy() is called
while (true) {

    boolean hasEvents = false;

    try {
        if (!close) {
            // events方法是遍历queue, 执行event的run方法, run方法中将socketchannel注册到了
            // 自己的selector上
            hasEvents = events();
            // 检查是否有自己感兴趣的事件
            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();
        iterator.remove();
        NioSocketWrapper socketWrapper = (NioSocketWrapper) sk.attachment();
        // Attachment may be null if another thread has called
    }
}

```

```

        // cancelledKey()
        if (socketWrapper != null) {
            // 处理IO事件
            processKey(sk, socketWrapper);
        }
    }

    // Process timeouts 检查channel是否过期
    timeout(keyCount, hasEvents);
}

getStopLatch().countDown();
}

```

```

public boolean processSocket(SocketWrapperBase<S> socketWrapper,
    SocketEvent event, boolean dispatch) {
    try {
        if (socketWrapper == null) {
            return false;
        }
        // 复用对象作为容器 避免GC
        SocketProcessorBase<S> sc = processorCache.pop();
        if (sc == null) {
            sc = createSocketProcessor(socketWrapper, event);
        } else {
            sc.reset(socketWrapper, event);
        }
        // 生成socketprocessor对象 交给线程池执行
        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;
}

```

## SocketProcessor

主要功能：

调用http11processor处理请求，读取channel的数据来生成servletrequest对象

## Executor

执行socketProcessor的run方法，解析请求并通过容器来处理请求，最终会调用到serlvet

## 高并发思路

- 设计合理的线程模型，不要让线程阻塞，尽量让CPU忙起来
- 有多少任务，就用响应规模的线程去处理

如接受连接 检测IO事件 处理请求 用三个不同的线程组去处理，且线程的数量是可配置的

## IO模型相关问题

- 阻塞与非阻塞：应用程序在发起IO操作后，是等待还是立即返回
- 同步与异步：数据从内核空间到应用空间的拷贝，是由内核主动发起还是由应用程序发起

## Nio2Endpoint组件：Tomcat如何实现异步IO

### 异步IO的工作模式

1. 应用程序调用read API的同时，告诉内核数据准备好了之后应该拷贝到的buffer和处理这些数据需要调用的回调函数
2. 内核接收到read调用后，等待网卡数据到达，到达后产生硬件中断，内核在中断程序中把数据从网卡拷贝到内核空间，进行TCP/IP层面的解包和重组，然后把数据拷贝到指定的buffer，最后调用回调函数

### Java nio2实现异步IO

```
public class Nio2Server {  
  
    void listen(){  
        //1. 创建一个线程池 内核只需要把工作交给线程池就返回了  
        ExecutorService es = Executors.newCachedThreadPool();  
  
        //2. 创建异步通道群组
```

```

    AsynchronousChannelGroup tg = AsynchronousChannelGroup.withCachedThreadPool(es,
1);

    //3. 创建服务端异步通道
    AsynchronousServerSocketChannel assc = AsynchronousServerSocketChannel.open(tg);

    //4. 绑定监听端口
    assc.bind(new InetSocketAddress(8080));

    //5. 监听连接，传入回调类处理连接请求
    assc.accept(this, new AcceptHandler());
}

}

```

```

//AcceptHandler类实现了CompletionHandler接口的completed方法。它还有两个模板参数，第一个是异步通道，第二个就是Nio2Server本身
public class AcceptHandler implements CompletionHandler<AsynchronousSocketChannel,
Nio2Server> {

    //具体处理连接请求的就是completed方法，它有两个参数：第一个是异步通道，第二个就是上面传入的
    NioServer对象
    @Override
    public void completed(AsynchronousSocketChannel asc, Nio2Server attachment) {
        //调用accept方法继续接收其他客户端的请求
        attachment.assc.accept(attachment, this);

        //1. 先分配好Buffer，告诉内核，数据拷贝到哪里去
        ByteBuffer buf = ByteBuffer.allocate(1024);

        //2. 调用read函数读取数据，除了把buf作为参数传入，还传入读回调类
        channel.read(buf, buf, new ReadHandler(asc));
    }
}

```

```

// v: IO调用的返回值 a: 附件类
public interface CompletionHandler<V,A> {

    void completed(V result, A attachment);

    void failed(Throwable exc, A attachment);
}

```

```

public class ReadHandler implements CompletionHandler<Integer, ByteBuffer> {
    //读取到消息后的处理
    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        //attachment就是数据, 调用flip操作, 其实就是把读的位置移动最前面
        attachment.flip();
        //读取数据
        ...
    }

    void failed(Throwable exc, A attachment){
        ...
    }
}

```

## Nio2Endpoint工作步骤

1. limitlatch控制最大连接数
2. Nio2acceptor跑在一个单独的线程中, 也是一个线程组, 获取连接后得到一个 asynchronoussocketchannel, acceptor把这个channel封装成一个nio2socketchannelwrapper, 并创建一个socketprocessor任务类给线程池处理
3. executor在执行任务类的时候, socketprocessor的run方法会调用http11processor处理请求, 解析数据, 读写都是通过niosocketwrapper

异步IO模式下, selector的工作交给内核做了

## Nio2Accepter

监听连接的过程不是在一个死循环里面调用accept方法, 而是通过回调函数完成的

```

// 回调类是acceptor本身
serverSock.accept(null, this);

```

```

// 返回结果是异步的socketchannel
protected class Nio2Accepter extends Acceptor<AsynchronousSocketChannel>
    implements CompletionHandler<AsynchronousSocketChannel, Void> {

    @Override
    public void completed(AsynchronousSocketChannel socket,
        Void attachment) {
        // 运行到这里说内核已经把数据拷贝到用户空间了
        // 所以继续调用accept方法监听下一个请求
        if (isRunning() && !isPaused()) {

```

```

    if (getMaxConnections() == -1) {
        //如果没有连接限制，继续接收新的连接
        serverSock.accept(null, this);
    } else {
        //如果有连接限制，就在线程池里跑run方法，run方法会检查连接数
        getExecutor().execute(this);
    }
    //处理请求
    if (!setSocketOptions(socket)) {
        closeSocket(socket);
    }
}
}

```

## Nio2SocketWrapper

主要作用：

- 封装channel
- 提供接口给http11processor进行读写

Http11processor对nio2socketwrapper进行了两次的read调用

1. 第一个read调用 注册回调类readCompletionHandler read方法立即返回
2. 第二次read调用 数据到达应用层中的buffer后，回调类被调用，在回调方法里面会创建一个socketprocessor 处理请求，而nio2socketwrapper也可以重新拿到（作为回调方法中的附带类）

```

this.readCompletionHandler = new CompletionHandler<Integer,
SocketWrapperBase<Nio2Channel>>() {
    public void completed(Integer nBytes, SocketWrapperBase<Nio2Channel> attachment) {
        ...
        //通过附件类SocketWrapper拿到所有的上下文
        Nio2SocketWrapper.this.getEndpoint().processSocket(attachment,
        SocketEvent.OPEN_READ, false);
    }

    public void failed(Throwable exc, SocketWrapperBase<Nio2Channel> attachment) {
        ...
    }
}

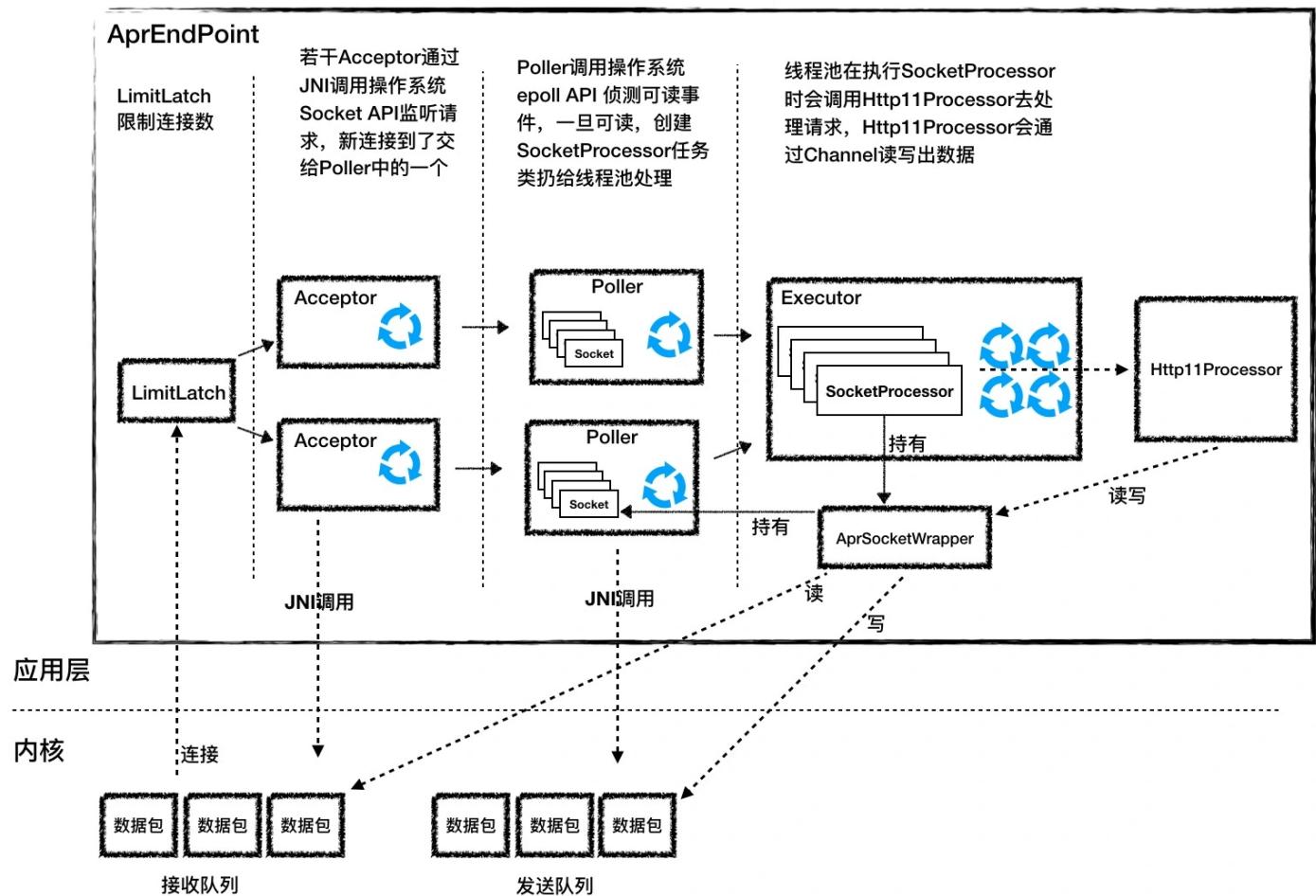
```

# AprEndpoint: Tomcat APR提高IO性能的秘密

APR: apache portable runtime libraries 用C语言实现的，与操作系统和网络交互方面会比较快

JNI: java native interface java调用其他语言编写的程序或者库

工作过程：



与NioEndpoint的区别在Acceptor和Poller的实现

## Acceptor

acceptor的功能：监听连接，接收并建立连接

本质就是调用了操作系统的API：

- socket
- bind
- listen
- accept

## Java通过JNI调用C语言的API

- 封装一个Java类，在里面定义native方法
- 用C实现这些方法

## Poller

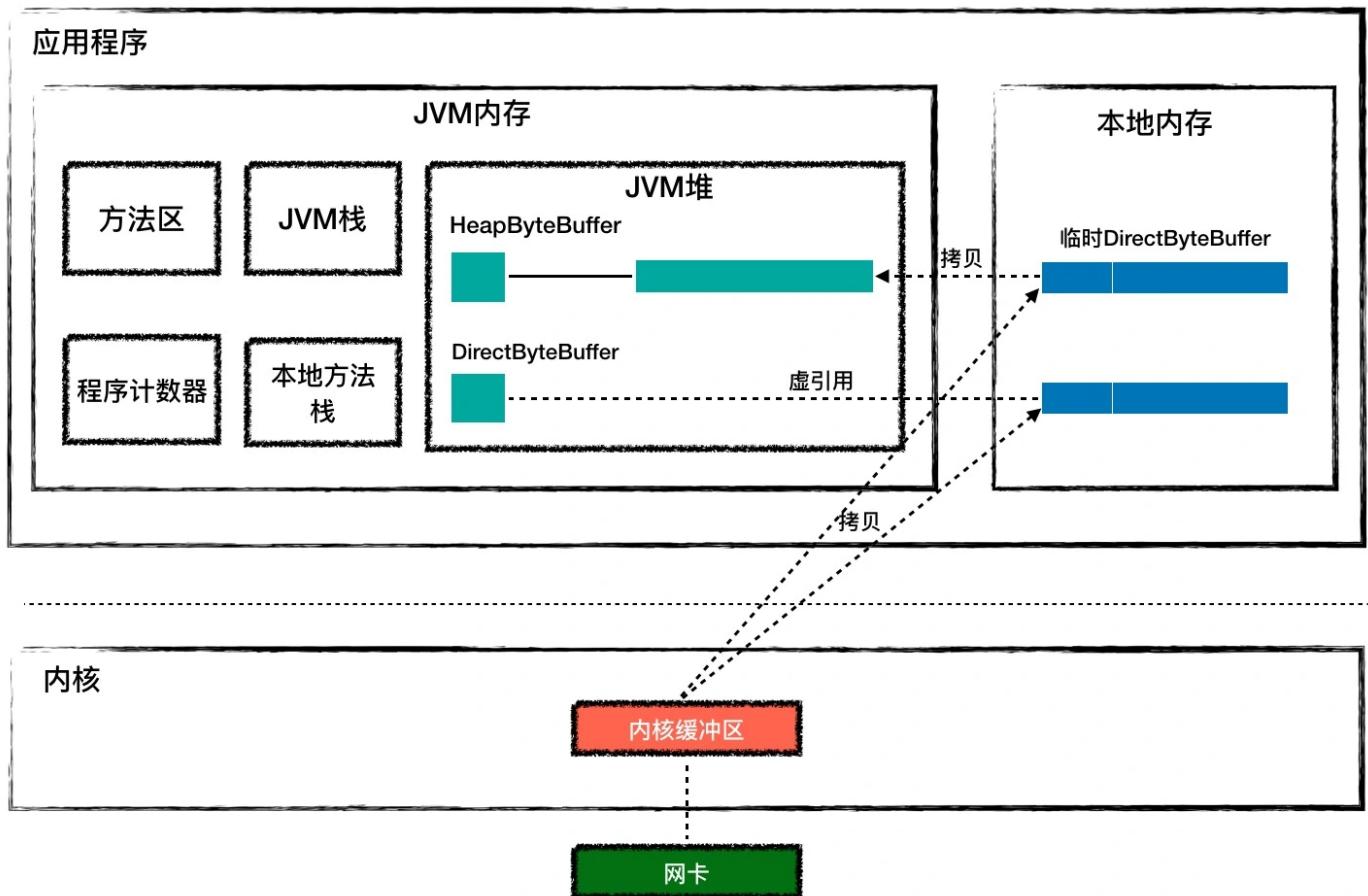
通过JNI调用APR中的poll方法，其中又调用了操作系统的epoll API来实现（检查这个socket的IO事件）

AprEndpoint中的一个参数：`deferAccept`，对应TCP协议中的`TCP_DEFER_ACCEPT`

设置该参数后，TCP客户端有新的连接到达时，TCP服务端先不建立连接，而是等到客户端有请求数据到达后才建立连接，这样就省去了服务端不断用selector查询数据是否准备就绪的环节

## JVM堆 VS 本地内存

C代码运行过程中用到的内存：本地内存



```
//分配HeapByteBuffer  
ByteBuffer buf = ByteBuffer.allocate(1024);  
  
//分配DirectByteBuffer  
ByteBuffer buf = ByteBuffer.allocateDirect(1024);  
  
//将buf作为read函数的参数  
int bytesRead = socketChannel.read(buf);
```

## HeapByteBuffer 与 DirectByteBuffer的区别：以接收网络数据为例

从内核中读取数据时，使用Headbytebuffer需要把数据先拷贝到本地内存，再从本地内存拷贝到堆

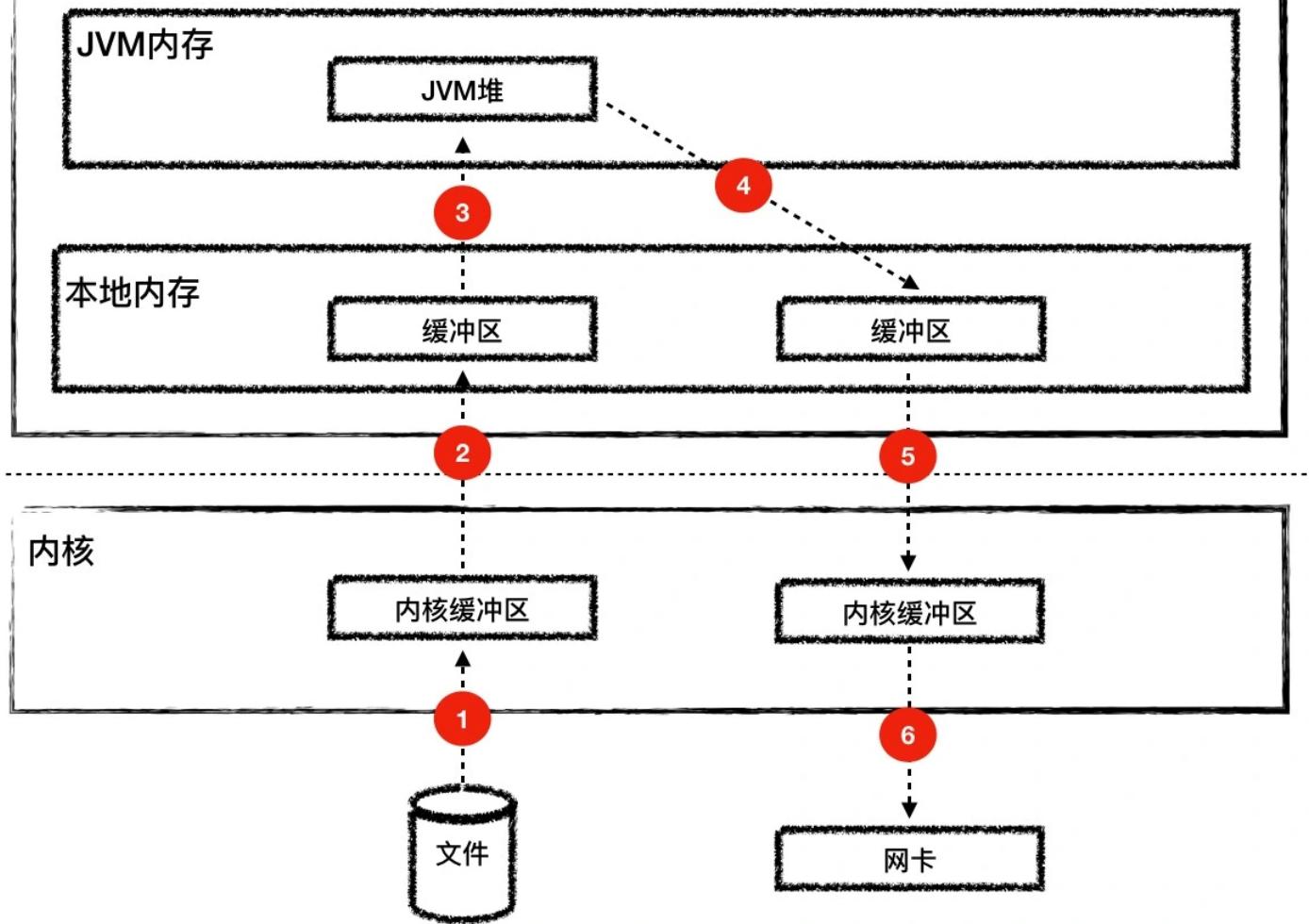
使用directbytebuffer则只需要把数据拷贝到本地内存，堆中持有一个本地内存地址即可（但是本地内存难以管理）

**sendfile**: 浏览器通过Tomcat获取一个Html文件

通过sendfile特性可以避免内核与应用之间的内存拷贝以及用户态和内核态之间的切换

使用HeapByteBuffer的情况

## 应用程序



AprEndpoint直接调用操作系统的sendfile API

## 应用程序

Tomcat

↓ JNI调用

APR

sendFile

## 内核

内核缓冲区

1

2

文件

网卡



# Executor

Java线程池

## ThreadPoolExecutor

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

在提交任务到线程池的时候：

- 如果当前线程数 < 核心线程数，就创建一个新的线程执行该任务
- 如果当前线程数 = 核心线程数，将任务放到任务队列中（线程会不停poll，从任务队列中取任务来执行）
- 如果任务队列已满 `queue.offer()` 返回false，如果当前线程数 < 最大线程数，则创建临时线程执行该任务  
(临时线程有存活时间 到了就会被销毁 临时线程取任务的方法是 `poll(keepAliveTime,unit)`)

否则就执行拒绝策略

## Tomcat线程池

创建一个线程池需要考虑的问题：

- 是否限制线程数量个数
- 是否队列长度

```
// AbstractEndpoint中的方法 供子类调用
public void createExecutor() {
    internalExecutor = true;
    // 自己的任务队列
    TaskQueue taskqueue = new TaskQueue();
    // 自己的线程工厂
    TaskThreadFactory tf = new TaskThreadFactory(getName() + "-exec-", daemon,
getThreadPriority());
    // 自己的线程池
    executor = new ThreadPoolExecutor(getMinSpareThreads(), getMaxThreads(), 60,
TimeUnit.SECONDS,taskqueue, tf);
    taskqueue.setParent( (ThreadPoolExecutor) executor);
}
```

## Tomcat线程池做的改进

在临时线程时也满的时候，不立即执行拒绝策略，而是等待一段时间后，再次尝试把任务放到队列中（因为等待的这段时间内，可以队列中就有任务被取走了空了位置出来）

```
public class ThreadPoolExecutor extends java.util.concurrent.ThreadPoolExecutor {  
    ...  
  
    public void execute(Runnable command, long timeout, TimeUnit unit) {  
        // 已经提交但没有执行完的任务个数  
        submittedCount.incrementAndGet();  
        try {  
            //调用Java原生线程池的execute去执行任务  
            super.execute(command);  
        } catch (RejectedExecutionException rx) {  
            //如果总线程数达到maximumPoolSize, Java原生线程池执行拒绝策略  
            if (super.getQueue() instanceof TaskQueue) {  
                final TaskQueue queue = (TaskQueue)super.getQueue();  
                try {  
                    //继续尝试把任务放到任务队列中去  
                    if (!queue.force(command, timeout, unit)) {  
                        submittedCount.decrementAndGet();  
                        //如果缓冲队列也满了，插入失败，执行拒绝策略。  
                        throw new RejectedExecutionException("...");  
                    }  
                }  
            }  
        }  
    }  
}
```

## Tomcat对任务队列做的改进：

offer方法返回false表示加入队列失败

改进目的：在任务队列长度没有限制的情况下，让线程池有机会可以创建新的线程

```
public class TaskQueue extends LinkedBlockingQueue<Runnable> {  
    ...  
    @Override  
    //线程池调用任务队列的方法时，当前线程数肯定已经大于核心线程数了  
    public boolean offer(Runnable o) {
```

```

//如果线程数已经到了最大值，不能创建新线程了，只能把任务添加到任务队列。
if (parent.getPoolSize() == parent.getMaximumPoolSize())
    return super.offer(o);

//执行到这里，表明当前线程数大于核心线程数，并且小于最大线程数。
//表明是可以创建新线程的，那到底要不要创建呢？分两种情况：

//1. 如果已提交的任务数小于当前线程数，表示还有空闲线程，无需创建新线程
if (parent.getSubmittedCount() <= (parent.getPoolSize()))
    return super.offer(o);

//2. 如果已提交的任务数大于当前线程数，线程不够用了，返回false去创建新线程
if (parent.getPoolSize() < parent.getMaximumPoolSize())
    return false;

//默认情况下总是把任务添加到任务队列
return super.offer(o);
}

}

```

## Tomcat如何支持WebSocket

**Socket:** IP + PORT

是对TCP/IP协议抽象出来的API

### WebSocket工作原理

Websocket是一个应用层协议，通过HTTP完成握手，之后的数据传输都是通过TCP

握手：

- HTTP请求：加上两个请求头

`Connection: Updagnitude`

`Upgrade: websocket`

- HTTP响应也加上这两个响应头

数据传输：以frame的形式传输，会将一个消息分为多个frame

服务端实现代码：

```
//Tomcat端的实现类加上@ServerEndpoint注解，里面的value是URL路径
```

```
@ServerEndpoint(value = "/websocket/chat")
public class ChatEndpoint {

    private static final String GUEST_PREFIX = "Guest";

    //记录当前有多少个用户加入到了聊天室，它是static全局变量。为了多线程安全使用原子变量
    AtomicInteger
    private static final AtomicInteger connectionIds = new AtomicInteger(0);

    //每个用户用一个ChatEndpoint实例来维护，请你注意它是一个全局的static变量，所以用到了线程安全的
    CopyOnWriteArraySet
    private static final Set<ChatEndpoint> connections =
        new CopyOnWriteArraySet<>();

    private final String nickname;
    private Session session;

    public ChatEndpoint() {
        nickname = GUEST_PREFIX + connectionIds.getAndIncrement();
    }

    //新连接到达时，Tomcat会创建一个Session，并回调这个函数
    @OnOpen
    public void start(Session session) {
        this.session = session;
        connections.add(this);
        String message = String.format("* %s %s",
            nickname, "has joined.");
        broadcast(message);
    }

    //浏览器关闭连接时，Tomcat会回调这个函数
    @OnClose
    public void end() {
        connections.remove(this);
        String message = String.format("* %s %s",
            nickname, "has disconnected.");
        broadcast(message);
    }

    //浏览器发送消息到服务器时，Tomcat会回调这个函数
    @OnMessage
    public void incoming(String message) {
        // Never trust the client
        String filteredMessage = String.format("%s: %s",
            nickname, HTMLFilter.filter(message.toString()));
        broadcast(filteredMessage);
    }

    //WebSocket连接出错时，Tomcat会回调这个函数
}
```

```

@OnError
public void onError(Throwable t) throws Throwable {
    log.error("Chat Error: " + t.toString(), t);
}

//向聊天室中的每个用户广播消息
private static void broadcast(String msg) {
    for (ChatEndpoint client : connections) {
        try {
            synchronized (client) {
                client.session.getBasicRemote().sendText(msg);
            }
        } catch (IOException e) {
            ...
        }
    }
}

```

Tomcat主要做的两件事：

- Endpoint加载
- WebSocket请求处理

## Endpoint加载

通过SCI(监听Servlet容器的启动事件，从而执行一些初始化工作)

```

@HandlesTypes({ServerEndpoint.class, ServerApplicationConfig.class, Endpoint.class})
public class WsSci implements ServletContainerInitializer {

    public void onStartup(Set<Class<?>> clazzes, ServletContext ctx) throws
ServletException {
        ...
    }
}

```

Tomcat在启动阶段扫描类的时候，会将HandlersTypes注解中指定的类都扫描出来，作为onStartup的参数

在这个方法中构造一个WebsocketContainer实例，将类注册到这个容器中，并维护URL -> Endpoint的映射关系

## Websocket请求处理

Tomcat先判断有没有ws的请求头，如果有就进行协议升级：

- UpgradeProtocolHandler替换当前的HttpProtocolHandler
- UpgraderProcessor替换processor
- 创建websocket Session实例和endpoint实例与当前websocket连接一一对应，此websocket连接不会立即关闭

## Jetty的EatWhatYouKill线程策略

### Selector编程一般思路

- 启动一个线程在一个死循环里面不断调用select方法 监测channel的IO状态（生产者）
- 一旦IO事件发生 就把该事件和相关数据封装成一个Runnable
- 把Runnable放到新的线程中去执行（消费者）

### 一般思路的缺点

- 无法利用CPU缓存（切换线程时 可能会用另外一个CPU核 这样原来的缓存的数据就用不上了）
- 线程切换也会有开销

Jetty中的Connector：把IO事件的生产和消费放到同一个线程中处理，方便利用CPU缓存

### ManagedSelector

```
public class ManagedSelector extends ContainerLifeCycle implements Dumpable
{
    //原子变量，表明当前的ManagedSelector是否已经启动
    private final AtomicBoolean _started = new AtomicBoolean(false);

    //表明是否阻塞在select调用上
    private boolean _selecting = false;

    //管理器的引用，SelectorManager管理若干ManagedSelector的生命周期
    private final SelectorManager _selectorManager;

    //ManagedSelector不止一个，为它们每人分配一个id
    private final int _id;

    //关键的执行策略，生产者和消费者是否在同一个线程处理由它决定
    private final ExecutionStrategy _strategy;

    //Java原生的Selector
    private Selector _selector;
```

```
//"Selector更新任务"队列
private Deque<SelectorUpdate> _updates = new ArrayDeque<>();
private Deque<SelectorUpdate> _updateable = new ArrayDeque<>();

...
}
```

## SelectUpdate接口

```
/**
 * A selector update to be done when the selector has been woken.
 */
public interface SelectorUpdate
{
    void update(Selector selector);
}
```

用于封装对selector的操作:

- 往selector上注册IO事件
- 绑定channel到selector上

## Selectable接口

```
public interface Selectable
{
    //当某一个Channel的I/O事件就绪后，ManagedSelector会调用的回调函数
    Runnable onSelected();

    //当所有事件处理完了之后ManagedSelector会调的回调函数，我们先忽略。
    void updateKey();
}
```

endpoint组件向managedSelector注册IO事件时，同时传入一个事件对应的附件类（实现Selectable接口），当事件到达的时候，通过 `onSelected()` 方法拿到一个回调方法并执行

## ExecutionStrategy

```
public interface ExecutionStrategy
{
    //只在HTTP2中用到，简单起见，我们先忽略这个方法。
    public void dispatch();
```

```

//实现具体执行策略，任务生产出来后可能由当前线程执行，也可能由新线程来执行
public void produce();

//任务的生产委托给Producer内部接口,
public interface Producer
{
    //生产一个Runnable(任务)
    Runnable produce();
}

}

```

实现类：

- `ProduceConsume`

在一个线程内依次生产和消费IO事件

- `ProduceExecuteConsume`

任务生产者开启新线程来运行任务(生产者单独一个线程 消费者很多线程)

- `ExecuteProduceConsume`

任务的生产和消费都在一个线程内，但是可能会新建一个线程以继续生产和消费

可以利用CPU缓存 但是如果消费IO事件的业务代码执行时间过长 会导致线程大量阻塞

(多个线程对应多组 生产者-消费者)

- `EatWhatYouKill`

线程充足时 生产和消费在同一个线程内

线程不够时 生产一个线程 消费多个线程

`ExecutionStrategy`中通过`Producer`拿到IO事件对应的`Runnable`对象

然后根据线程策略来做出下一步：

- 自己执行
- 交给线程池执行

```

private class SelectorProducer implements ExecutionStrategy.Producer
{
    private Set<SelectionKey> _keys = Collections.emptySet();
    private Iterator<SelectionKey> _cursor = Collections.emptyIterator();

    @Override
    public Runnable produce()

```

```

{
    while (true)
    {
        //如何Channel集合中有I/O事件就绪，调用前面提到的Selectable接口获取Runnable，直接返回
        //给ExecutionStrategy去处理
        Runnable task = processSelected();
        if (task != null)
            return task;

        //如果没有I/O事件就绪，就干点杂活，看看有没有客户提交了更新Selector的任务，就是上面提到的
        //SelectorUpdate任务类。
        processUpdates();
        updateKeys();

        //继续执行select方法，侦测I/O就绪事件
        if (!select())
            return null;
    }
}
}

```

## Tomcat和Jetty中的对象池技术

使用目的：

- 为了避免过大、过复杂的对象的频繁的创建、初始化、GC（会耗费CPU和内存资源）
- 对象数量多 且存在时间比较短

实现方法：

把一个Java对象用完之后将他保存起来，需要再用时再拿出来（清空之前的信息）重复使用（空间换时间）

### Tomcat中的对象池

用了一个同步的栈，需要对象时从栈中去，然后reset对象，清空对象信息

对象使用完毕后将对象压入栈中

```

public class SynchronizedStack<T> {

    //内部维护一个对象数组，用数组实现栈的功能
    private Object[] stack;

    // 在使用完之后 这个方法用来归还对象，用synchronized进行线程同步
    public synchronized boolean push(T obj) {

```

```

    index++;
    if (index == size) {
        if (limit == -1 || size < limit) {
            expand(); //对象不够用了，扩展对象数组
        } else {
            index--;
            return false;
        }
    }
    stack[index] = obj;
    return true;
}

//这个方法用来获取对象
public synchronized T pop() {
    if (index == -1) {
        return null;
    }
    T result = (T) stack[index];
    stack[index--] = null;
    return result;
}

//扩展对象数组长度，以2倍大小扩展
private void expand() {
    int newSize = size * 2;
    if (limit != -1 && newSize > limit) {
        newSize = limit;
    }
    //扩展策略是创建一个数组长度为原来两倍的新数组
    Object[] newStack = new Object[newSize];
    //将老数组对象引用复制到新数组
    System.arraycopy(stack, 0, newStack, 0, size);
    //将stack指向新数组，老数组可以被GC掉了
    stack = newStack;
    size = newSize;
}
}

```

## Jetty的ByteBufferPool

```

public interface ByteBufferPool
{
    // 获取内存 direct表示是否是从本地内存分配
    public ByteBuffer acquire(int size, boolean direct);
    // 释放内存 (表示内存对象可以被再次复用了)
    public void release(ByteBuffer buffer);
}

```

```

public class ArrayByteBufferPool implements ByteBufferPool
{
    private final int _min; //最小size的Buffer长度
    private final int _maxQueue; //Queue最大长度

    //用不同的Bucket(桶)来持有不同size的ByteBuffer对象,同一个桶中的ByteBuffer size是一样的 数组
    + 链表 (链表中的结点是大小相同的内存块)
    private final ByteBufferPool.Bucket[] _direct;
    private final ByteBufferPool.Bucket[] _indirect;

    //ByteBuffer的size增量
    private final int _inc;

    public ArrayByteBufferPool(int minSize, int increment, int maxSize, int maxQueue)
    {
        //检查参数值并设置默认值
        if (minSize<=0) //ByteBuffer的最小长度
            minSize=0;
        if (increment<=0)
            increment=1024; //默认以1024递增
        if (maxSize<0)
            maxSize=64*1024; //ByteBuffer的最大长度默认是64K

        //ByteBuffer的最小长度必须小于增量
        if (minSize>=increment)
            throw new IllegalArgumentException("minSize >= increment");

        //最大长度必须是增量的整数倍
        if ((maxSize%increment)!=0 || increment>=maxSize)
            throw new IllegalArgumentException("increment must be a divisor of
maxSize");

        _min=minSize;
        _inc=increment;

        //创建maxSize/increment个桶,包含直接内存的与heap的
    }
}

```

```

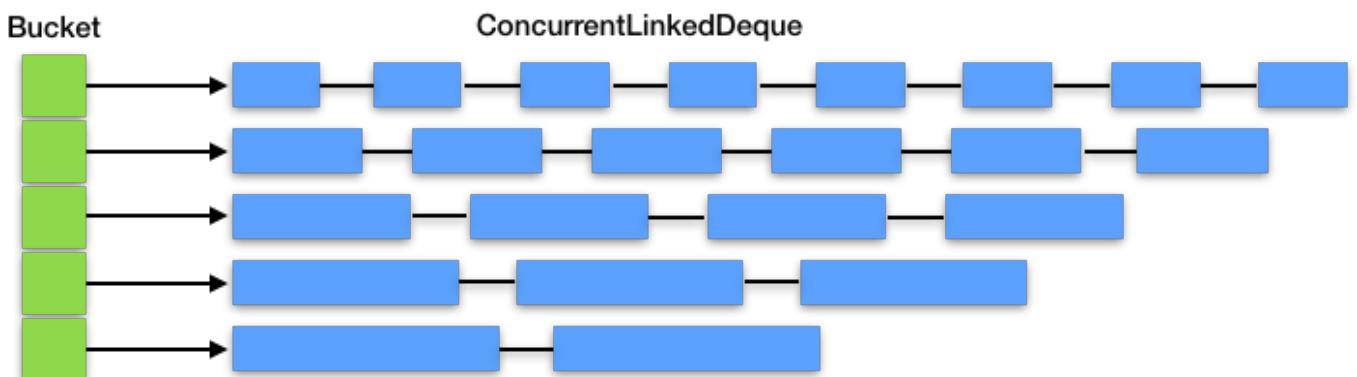
    _direct=new ByteBufferPool.Bucket[maxSize/increment];
    _indirect=new ByteBufferPool.Bucket[maxSize/increment];
    _maxQueue=maxQueue;
    int size=0;
    // 1024 2048 ... 64K
    for (int i=0;i<_direct.length;i++)
    {
        size+=_inc;
        _direct[i]=new ByteBufferPool.Bucket(this,size,_maxQueue);
        _indirect[i]=new ByteBufferPool.Bucket(this,size,_maxQueue);
    }
}
}

```

```

// bucket内部维护了一个链表来存放bytebuffer对象的引用
private final Deque<ByteBuffer> _queue = new ConcurrentLinkedDeque<>();

```



buffer的分配和释放就是堆bucket中的deque作出队和入队操作

```

//分配Buffer
public ByteBuffer acquire(int size, boolean direct)
{
    //找到对应的桶，没有的话创建一个桶
    ByteBufferPool.Bucket bucket = bucketFor(size,direct);
    if (bucket==null)
        return newByteBuffer(size,direct);
    //这里其实调用了Deque的poll方法
    return bucket.acquire(direct);
}

//释放Buffer
public void release(ByteBuffer buffer)

```

```

{
    if (buffer!=null)
    {
        //找到对应的桶
        ByteBufferPool.Bucket bucket = bucketFor(buffer.capacity(), buffer.isDirect());

        //这里调用了Deque的offerFirst方法
        if (bucket!=null)
            bucket.release(buffer);
    }
}

```

## Tomcat Jetty的高性能 高并发之道

### 高性能：

- 高效地利用CPU 内存 磁盘 网络等资源

如何实现：

- 减少资源浪费：避免线程阻塞造成上下文切换 过多的数据拷贝
- 空间换时间（缓存或者对象池） 或者时间换空间（数据压缩后再进行网络传输）
- 在短时间内处理大量请求
  - 短时间：响应时间
  - 大量请求：TPS 每秒处理的事务数

### IO和线程模型

#### IO模型：

- 本质作用：解决CPU与外设之间的速度差
- 目标：尽量减少业务线程因为等待IO而造成的阻塞
- 采用的方法：非阻塞IO 异步IO

#### 线程模型：

- 连接请求由专门的Acceptor线程组来处理
- IO事件探测由专门的Selector线程组来处理
- 具体的协议解析和业务处理可能交给线程池（另外的线程），或者Selector线程自己处理（同一个线程中 可以利用CPU缓存 减少线程的上下文切换）

CPU的核数有限 线程过多反而会造成过多的线程上下文切换

### 减少系统调用

系统调用涉及到用户态到内核态的切换 比较消耗系统资源

- 批处理：用缓冲存储数据 达到一定规模后才进行处理
- 延迟处理或者避免处理：等到真正需要进行操作时才进行系统调用 不需要相关操作就不进行系统调用

## 池化 零拷贝

池化：

- 使用场景：当使用的对象很大 很复杂 且需要创建的数量多 生命周期短时
- 需要注意的点：
  - 对象用完后 归还给对象池
  - 对象池中的对象取出时需要进行重置 否则会有脏数据
  - 对象池需要加锁或者用并发容器保证线程安全
  - 向对象池请求对象时可能出现阻塞 异常 或者null值 都需要在使用时进行额外的处理

## 高效的并发编程

### 缩小锁的范围

```
// 不在方法上加锁而是在内部加锁 多个线程可以并行执行这个方法 只在同时访问某个成员变量时才需要排队等待
protected void startInternal() throws LifecycleException {
    setState(LifecycleState.STARTING);

    // 锁engine成员变量
    if (engine != null) {
        synchronized (engine) {
            engine.start();
        }
    }

    //锁executors成员变量
    synchronized (executors) {
        for (Executor executor: executors) {
            executor.start();
        }
    }

    mapperListener.start();

    //锁connectors成员变量
    synchronized (connectorsLock) {
        for (Connector connector: connectors) {
            // If it has already failed, don't try and start it
            if (connector.getState() != LifecycleState.FAILED) {
                connector.start();
            }
        }
    }
}
```

```
        }
    }
}
}
```

缩小

## 用原子变量和CAS取代锁

```
private boolean startThreads(int threadsToStart)
{
    while (threadsToStart > 0 && isRunning())
    {
        //获取当前已经启动的线程数，如果已经够了就不需要启动了
        int threads = _threadsStarted.get();
        if (threads >= _maxThreads)
            return false;

        //用CAS方法将线程数加一，请注意执行失败走continue，继续尝试
        if (!_threadsStarted.compareAndSet(threads, threads + 1))
            continue;

        boolean started = false;
        try
        {
            Thread thread = newThread(_runnable);
            thread.setDaemon(isDaemon());
            thread.setPriority(getThreadsPriority());
            thread.setName(_name + " - " + thread.getId());
            _threads.add(thread); // _threads并发集合
            _lastShrink.set(System.nanoTime()); // _lastShrink是原子变量
            thread.start();
            started = true;
            --threadsToStart;
        }
        finally
        {
            //如果最终线程启动失败，还需要把线程数减一
            if (!started)
                _threadsStarted.decrementAndGet();
        }
    }
    return true;
}
```

## 并发容器的使用

```
public abstract class LifecycleBase implements Lifecycle {  
  
    //事件监听器集合 适用于读多写少的多线程访问场景  
    private final List<LifecycleListener> lifecycleListeners = new  
CopyOnWriteArrayList<>();  
  
    ...  
}
```

## volatile关键字的使用

```
public abstract class LifecycleBase implements Lifecycle {  
  
    //当前组件的生命状态，用volatile修饰  
    private volatile LifecycleState state = LifecycleState.NEW;  
  
}
```

## 内核如何阻塞与唤醒线程

### 进程与线程：

进程在内核中会对应一个`task_struct`的数据结构，其内容有

- `vm_struct`: 保存了进程的各内存区域的起始和终止地址
- 进程号
- 打开的文件
- 创建的socket
- CPU运行上下文

### 线程：

有自己的`task_struct`和运行栈区，但是其他资源如虚拟地址空间，打开的文件，Socket都是跟父进程共用

## 进程的虚拟地址空间：

32位的机器中，给每个进程都分配了4GB ( $2^{32}$ ) 的虚拟内存地址空间

缺页中断：进程访问到了某个虚拟地址，如果这个地址没有对应的物理内存页，就会产生缺页中断，分配物理内存，MMU（内存管理单元）会维护这个虚拟地址与物理内存页的映射关系

## 分布：

- 低位的3GB属于用户空间
- 高位的1GB属于内核空间

## 组成：由高到低

- 内核空间
- 环境变量
- 栈（向低地址扩大）
- 共享库和mmap映射区

mmap：是一个内存映射函数，可以将文件内容映射到这个内存区域，用户通过读写这段内存，就可以实现对文件的修改，无需通过系统调用read/write，省去了内核空间与用户空间之间的数据拷贝

Java实现：`MappedByteBuffer`

- 堆（向高地址扩大）
- 数据区
- 代码区



不同程序的权限不同：

- 用户程序只能访问用户空间 访问硬件资源需要通过系统调用（内核中的函数）
- 内核程序可以访问整个进程空间 可以直接访问各种硬件资源：网卡 磁盘

### 线程的阻塞与唤醒

内核将线程当做一个进程进行CPU调度，内核中维护了两个队列一个是可运行的 `task_struct` 队列，一个是阻塞的 `task_struct` 队列（本质上就是一个双向链表）

`task_struct` 排队使用CPU时间片，使用完之后重新调用CPU，即从队列中再选出一个`task_struct`，恢复期上下文到CPU的寄存器中，然后执行上下文中指定的下一条指令

阻塞：将`task_struct` 移除运行队列，加入到等待队列，重置进程的状态，重新触发一次CPU调度（选择下一个线程）

唤醒：线程在加入到等待队列的时候，向内核注册了一个回调函数，告诉内核他在等待这个socket的数据，当网卡收到数据，产生硬件中断，内核再通过回调函数唤醒线程，即将线程从等待队列移动到运行队列，同时把数据从内核空间拷贝到用户空间的堆上

## 容器

### Host容器 Tomcat如何实现热加载和热部署

热加载：启动一个后台线程，定期监测类文件的变化，如果有变化就重新加载类

热部署：后台线程定期监测web应用的变化，如果有变化就重新加载整个应用

#### Tomcat的后台线程

```
bgFuture = exec.scheduleWithFixedDelay(  
    new ContainerBackgroundProcessor(), //要执行的Runnable  
    backgroundProcessorDelay, //第一次执行延迟多久  
    backgroundProcessorDelay, //之后每次执行间隔多久  
    TimeUnit.SECONDS); //时间单位
```

```
// 该类为ContainerBase的内部类  
protected class ContainerBackgroundProcessor implements Runnable {  
  
    @Override  
    public void run() {  
        //请注意这里传入的参数是"宿主类"的实例  
        processChildren(ContainerBase.this);  
    }  
  
    protected void processChildren(Container container) {  
        try {
```

```

//1. 调用当前容器的backgroundProcess方法。
container.backgroundProcess();

//2. 遍历所有的子容器，递归调用processChildren,
//这样当前容器的子孙都会被处理
Container[] children = container.findChildren();
for (int i = 0; i < children.length; i++) {
    //这里请你注意，容器基类有个变量叫做backgroundProcessorDelay，如果大于0，表明子容器有自己的后台线程，无需父容器来调用它的processChildren方法。
    if (children[i].getBackgroundProcessorDelay() <= 0) {
        processChildren(children[i]);
    }
}
} catch (Throwable t) { ... }

```

这样的设计（组合模式），只有在顶层容器，如engine，启动一个后台线程定期检查变化，那么所有的子容器的定时检查的方法都能触发

需要定时执行的逻辑放在 `backgroundProcess()` 方法中

```

// ContainerBase的实现
public void backgroundProcess() {

    //1.执行容器中Cluster组件的周期性任务
    Cluster cluster = getClusterInternal();
    if (cluster != null) {
        cluster.backgroundProcess();
    }

    //2.执行容器中Realm组件的周期性任务
    Realm realm = getRealmInternal();
    if (realm != null) {
        realm.backgroundProcess();
    }

    //3.执行容器中Valve组件的周期性任务
    Valve current = pipeline.getFirst();
    while (current != null) {
        current.backgroundProcess();
        current = current.getNext();
    }

    //4. 触发容器的"周期事件"，Host容器的监听器HostConfig就靠它来调用
    fireLifecycleEvent(Lifecycle.PERIODIC_EVENT, null);
}

```

## Tomcat热加载

就是在StandardContext里面的backgroundProcess()方法里实现的

在 context.xml 文件中设置来开启

```
<Context reloadable="true" />

public void backgroundProcess() {

    //WebappLoader周期性的检查WEB-INF/classes和WEB-INF/lib目录下的类文件
    Loader loader = getLoader();
    if (loader != null) {
        // 里面会调用context的stop和start方法 每个Context都对应一个类加载器
        // 会创建一个新的类加载器加载文件
        loader.backgroundProcess();
    }

    //Session管理器周期性的检查是否有过期的session
    Manager manager = getManager();
    if (manager != null) {
        manager.backgroundProcess();
    }

    //周期性的检查静态资源是否有变化
    WebResourceRoot resources = getResources();
    if (resources != null) {
        resources.backgroundProcess();
    }

    //调用父类ContainerBase的backgroundProcess方法
    super.backgroundProcess();
}
```

## Tomcat热部署

部署：销毁掉整个Context对象及其关联的所有资源

实现：通过一个事件监听器 public class HostConfig implements LifecycleListener

其中的

```
public void lifecycleEvent(LifecycleEvent event) {
```

```

// 执行check方法。
if (event.getType().equals(Lifecycle.PERIODIC_EVENT)) {
    check();
}

// 检查webapps目录下的所有web目录 如果多了目录就部署响应的web应用 少了目录就将相应的Context容器销毁掉
protected void check() {

    if (host.getAutoDeploy()) {
        // 检查这个Host下所有已经部署的Web应用
        DeployedApplication[] apps =
            deployed.values().toArray(new DeployedApplication[0]);

        for (int i = 0; i < apps.length; i++) {
            //检查Web应用目录是否有变化
            checkResources(apps[i], false);
        }
    }

    //执行部署
    deployApps();
}
}

```

## Context容器（上）：Tomcat如何打破双亲委托机制

### JVM的类加载器

- 过程：
  1. 把 .class 文件加载到JVM的方法区 (findClass)
  2. 在JVM的堆区创建一个 java.lang.Class 对象的实例(defineClass)，用来封装Java类相关的数据和方法
- 主体：类加载器
- 加载时机：只有在运行中用到了这个类才去加载

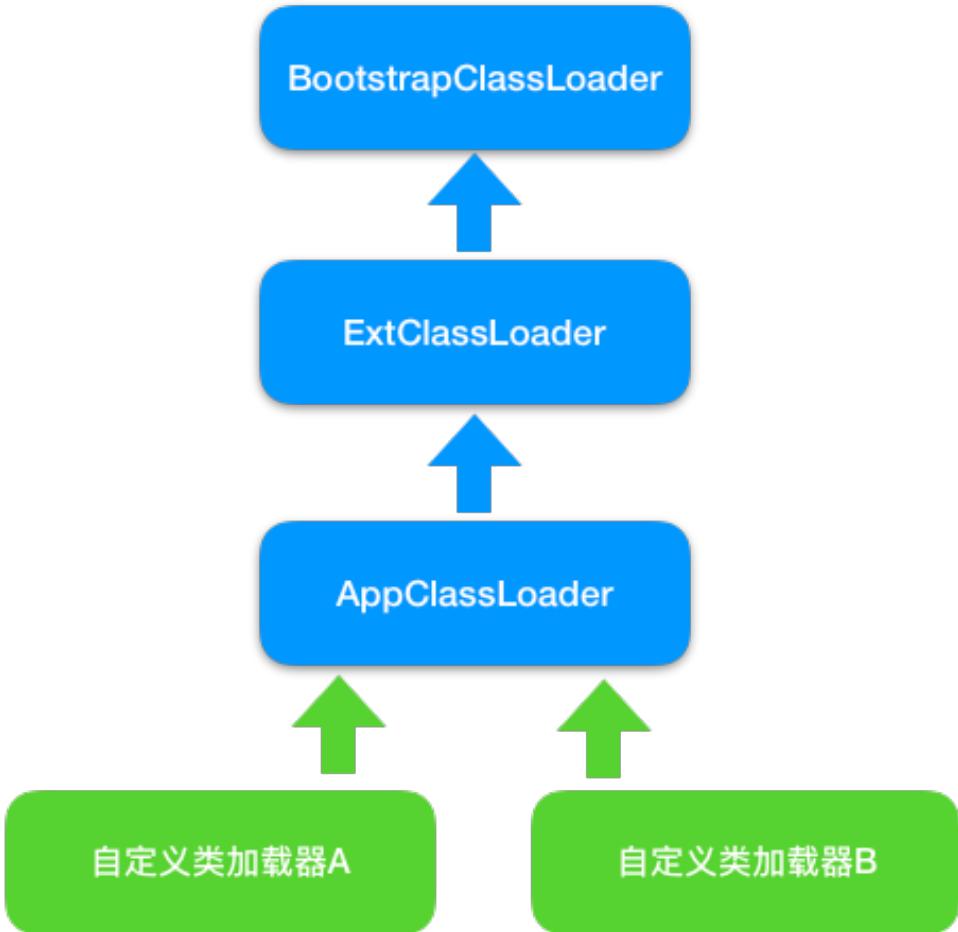
```

public abstract class ClassLoader {

    //每个类加载器都有个父加载器 for delegation
    private final ClassLoader parent;
}

```

```
public Class<?> loadClass(String name) {  
  
    //查找一下这个类是不是已经加载过了  
    Class<?> c = findLoadedClass(name);  
  
    //如果没有加载过  
    if( c == null ) {  
        //先委托给父加载器去加载, 注意这是个递归调用  
        if (parent != null) {  
            c = parent.loadClass(name);  
        }else {  
            // 如果父加载器为空, 查找Bootstrap加载器是不是加载过了  
            c = findBootstrapClassOrNull(name);  
        }  
    }  
  
    // 如果父加载器没加载成功, 调用自己的findClass去加载  
    if (c == null) {  
        c = findClass(name);  
    }  
  
    return c;  
}  
  
protected Class<?> findClass(String name){  
    //1. 根据传入的类名name, 到在特定目录下去寻找类文件, 把.class文件读入内存  
    ...  
  
    //2. 调用defineClass将字节数组转成Class对象  
    return defineClass(buf, off, len);  
}  
  
// 将字节码数组解析成一个Class对象(堆中), 用native方法实现  
protected final Class<?> defineClass(byte[ ] b, int off, int len){  
    ...  
}
```



类加载器的层级（工作原理相同 只是加载路径不同 父子关系通过组合 + 委托来实现）

- 启动类加载器：用来加载JVM启动时所需的核心类 /jre/lib
- 扩展类加载器：加载 /jre/lib/ext
- 应用程序类加载器：用来加载classpath下的类 `Class.forName()` 就是用这个类加载器进行加载
- 自定义类加载器：用来加载自定义路径下的类（重写 `findClass` 方法即可）

## Tomcat的类加载器

Servlet规范建议：

Web应用中的全路径类名与应用程序类（Tomcat的JVM的classpath下的类）同名，优先加载Web应用自己定义的类

首先尝试自己去加载某个类（在试着交给扩展类加载器加载之后），找不到再交给应用程序类加载器

实现：重写 `findClass` 和 `loadClass` 方法

### `findClass`方法

```

public Class<?> findClass(String name) throws ClassNotFoundException {
    ...
}

```

```

Class<?> clazz = null;
try {
    //1. 先在Web应用目录下查找类
    clazz = findClassInternal(name);
} catch (RuntimeException e) {
    throw e;
}

if (clazz == null) {
try {
    //2. 如果在本地目录没有找到, 交给父加载器去查找
    clazz = super.findClass(name);
} catch (RuntimeException e) {
    throw e;
}

//3. 如果父类也没找到, 抛出ClassNotFoundException
if (clazz == null) {
    throw new ClassNotFoundException(name);
}

return clazz;
}

```

## loadClass方法

```

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

synchronized (getClassLoadingLock(name)) {

Class<?> clazz = null;

//1. 先在本地cache查找该类是否已经加载过 就是一个CurrentHashMap
clazz = findLoadedClass0(name);
if (clazz != null) {
    if (resolve)
        resolveClass(clazz);
    return clazz;
}

//2. 从系统类加载器的cache中查找是否加载过
clazz = findLoadedClass(name);
if (clazz != null) {
    if (resolve)
        resolveClass(clazz);
}

```

```
        return clazz;
    }

// 3. 尝试用ExtClassLoader类加载器类加载, 为什么?
// 为了不让自己的类跟JRE中类发生冲突 (全类名一致的)
ClassLoader javaseLoader = getJavaseClassLoader();
try {
    clazz = javaseLoader.loadClass(name);
    if (clazz != null) {
        if (resolve)
            resolveClass(clazz);
        return clazz;
    }
} catch (ClassNotFoundException e) {
    // Ignore
}

// 4. 尝试在本地目录搜索class并加载
try {
    clazz = findClass(name);
    if (clazz != null) {
        if (resolve)
            resolveClass(clazz);
        return clazz;
    }
} catch (ClassNotFoundException e) {
    // Ignore
}

// 5. 尝试用系统类加载器(也就是AppClassLoader)来加载
try {
    clazz = Class.forName(name, false, parent);
    if (clazz != null) {
        if (resolve)
            resolveClass(clazz);
        return clazz;
    }
} catch (ClassNotFoundException e) {
    // Ignore
}
}

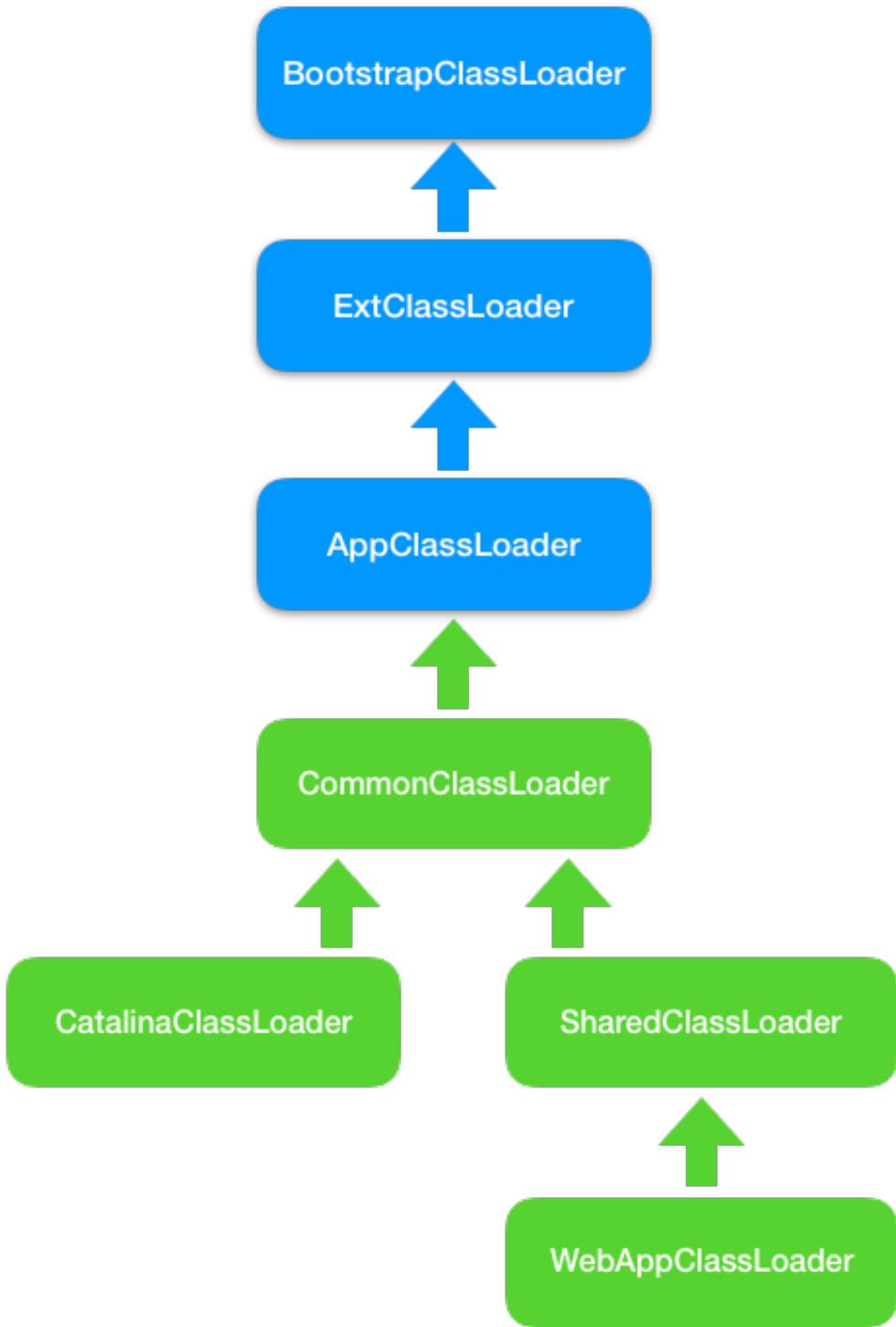
//6. 上述过程都加载失败, 抛出异常
throw new ClassNotFoundException(name);
}
```

## Context容器（中） Tomcat如何隔离Web应用的类

Tomcat的类加载器需要考虑的问题：

- 需要隔离不同Web应用加载的类  
(不同Web应用需要可以加载同名的类)
- 多个Web应用之间需要共享类
- 需要隔离Tomcat自身的类和Web应用的类

Tomcat的类加载器的层次



如何隔离不同Web应用的类：

每个Context容器维护一个WebAppClassLoader的实例，不同加载器实例加载的类被认为是不同的类

Web应用之间如何共享类：

不同的WebAppClassLoader的parent都设置为一个SharedClassLoader，当自己没有加载到该类时，就会委托父加载器

共享类放在父加载器的加载路径下即可

如何隔离Tomcat自身的类和Web应用的类：

用 `CatalinaClassLoader` 专门加载Tomcat自身的类，给 `CatalinaClassLoader` 和 `SharedClassLoader` 加一个父加载器 `CommonClassLoader`

需要共享的类放在 `CommonClassLoader` 的加载路径下

## Spring的加载问题

Spring是调用 `Class.forName` 来加载业务类的，但是它的实现是会用Spring的加载器来加载业务类

Spring由多个Web应用共享 自身的加载器是 `SharedClassLoader` 但是它的加载路径下没有业务类

需要拿到业务类对应WebAppClassLoader实例

```
public static Class<?> forName(String className) {
    Class<?> caller = Reflection.getCallerClass();
    return forName0(className, true, ClassLoader.getClassLoader(caller), caller);
}
```

解决方法：

Tomcat给启动Web应用的线程里面设置了线程上下文加载器（把WebappClassLoader类加载器的实例放到线程私有数据中）

```
originalClassLoader = Thread.currentThread().getContextClassLoader();
Thread.currentThread().setContextClassLoader(webApplicationClassLoader);

// 中间就是Spring的初始化过程

// 启动方法结束后恢复上下文加载器
Thread.currentThread().setContextClassLoader(originalClassLoader);
```

Spring启动的时候从线程私有数据中取出来

```
cl = Thread.currentThread().getContextClassLoader();
```

## Context容器（下） Tomcat如何实现Servlet规范

Servlet容器最重要的任务：实例化并调用Servlet

### Servlet管理

Context通过Wrapper来管理Servlet

实例化：

```
// wrapper持有一个servlet对象 和 URL映射 初始化参数等其他元信息
protected volatile Servlet instance = null;

// 初始化方法 懒加载 在用到servlet的时候 才会初始化servlet
public synchronized Servlet loadServlet() throws ServletException {
    Servlet servlet;

    //1. 创建一个Servlet实例
    servlet = (Servlet) instanceManager.newInstance(servletClass);

    //2.调用了Servlet的init方法，这是Servlet规范要求的
    initServlet(servlet);

    return servlet;
}
```

调用：

请求到来时，会调用到Wrapper的BasicValve（即链表中的最后一个结点 standardWrapperValve）

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

    //1.实例化Servlet
    servlet = wrapper.allocate();

    //2.给当前请求创建一个Filter链 每一个请求一个FilterChain
    // 因为请求路径可能不同 所以会对应不同的Filter
    ApplicationFilterChain filterChain =
        ApplicationFilterFactory.createFilterChain(request, wrapper, servlet);

    //3. 调用这个Filter链，Filter链中的最后一个Filter会调用Servlet
    filterChain.doFilter(request.getRequest(), response.getResponse());

}
```

```
//  
public final class ApplicationFilterChain implements FilterChain {  
  
    //Filter链中有Filter数组，这个好理解  
    private ApplicationFilterConfig[] filters = new ApplicationFilterConfig[0];  
  
    //Filter链中的当前的调用位置  
    private int pos = 0;  
  
    //总共有多少个Filter  
    private int n = 0;  
  
    //每个Filter链对应一个Servlet，也就是它要调用的Servlet  
    private Servlet servlet = null;  
  
    public void doFilter(ServletRequest req, ServletResponse res) {  
        internalDoFilter(req, res);  
    }  
  
    private void internalDoFilter(ServletRequest req,  
                                  ServletResponse res){  
  
        // 每个Filter链在内部维护了一个Filter数组  
        if (pos < n) {  
            ApplicationFilterConfig filterConfig = filters[pos++];  
            Filter filter = filterConfig.getFilter();  
  
            filter.doFilter(req, res, this);  
            return;  
        }  
  
        servlet.service(req, res);  
    }  
  
    // Filter中会回调FilterChain的doFilter方法  
    public void doFilter(ServletRequest request, ServletResponse response,  
                        FilterChain chain){  
  
        ...  
  
        //调用Filter的方法  
        chain.doFilter(request, response);  
    }
```

```
}
```

## Filter管理

Filter的作用域是整个Web应用，所以在Context中

```
private Map<String, FilterDef> filterDefs = new HashMap<>();
```

## Listener管理

监听容器内部发生的事件

事件类型：

- 生命状态的变化 Context容器启停 Session创建和销毁
- 属性的变化 Context容器的某个属性值变了 Session的某个属性值变了

```
//监听属性值变化的监听器
private List<Object> applicationEventListenersList = new CopyOnWriteArrayList<>();

//监听生命事件的监听器
private Object applicationLifecycleListenersObjects[] = new Object[0];
```

```
//1.拿到所有的生命周期监听器
Object instances[] = getApplicationLifecycleListeners();

for (int i = 0; i < instances.length; i++) {
    //2. 判断Listener的类型是不是ServletContextListener
    if (!(instances[i] instanceof ServletContextListener))
        continue;

    //3.触发Listener的方法
    ServletContextListener lr = (ServletContextListener) instances[i];
    lr.contextInitialized(event);
}
```

## Tomcat如何支持异步Servlet

问题：

一个请求到来，需要从Tomcat线程池中拿一个线程出来处理请求，在线程中会调用Web应用，应用处理完之后，才会输出响应



```

// 在Request中创建异步上下文
if (asyncContext == null) {
    asyncContext = new AsyncContextImpl(this);
}
// 将request response传入异步上下文
// request用于获取请求相关信息
// response用于发送响应
asyncContext.setStarted(getContext(), request, response,
    request==getRequest() && response==getResponse().getResponse());
// 异步请求超时时间为30s
asyncContext.setTimeout(getConnector().getAsyncTimeout());

return asyncContext;
}

// setStarted()
this.request.getCoyoteRequest().action(ActionCode.ASYNC_START, this);

// 最终会修改AsyncStateMachine 状态机中的状态 并传入异步上下文
public synchronized void asyncStart(AsyncContextCallback asyncCtxt) {
    if (state == AsyncState.DISPATCHED) {
        updateState(AsyncState.STARTING);
        this.asyncCtxt = asyncCtxt;
    }
}

```

```

// CoyoteAdapter
public void service(org.apache.coyote.Request req, org.apache.coyote.Response res) {

//调用容器的service方法处理请求
connector.getService().getContainer().getPipeline().
    getFirst().invoke(request, response);

//如果是异步Servlet请求，仅仅设置一个标志，
//否则说明是同步Servlet请求，就将响应数据刷到浏览器
if (request.isAsync()) {
    async = true;
} else {
    request.finishRequest();
    response.finishResponse();
}

//如果不是异步Servlet请求，就清空Request对象和Response对象
if (!async) {
    request.recycle();
    response.recycle();
}

```

```
}
```

```
asyncContext.complete()
```

```
public void complete() {
    //调用Request对象的action方法，其实就是通知连接器，这个异步请求处理完了
    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) {
    // 从Request中的ActionHook对象就是Processor对象 里面有Socket对象
    SocketWrapperBase<?> socketWrapper = getSocketWrapper();
    if (socketWrapper != null) {
        socketWrapper.processSocket(event, dispatch);
    }
}
```

```
public boolean processSocket(SocketWrapperBase<S> socketWrapper,
    SocketEvent event, boolean dispatch) {

    if (socketWrapper == null) {
        return false;
    }
    // 在sc中的run方法中执行response的写出和 监听新的IO事件
    SocketProcessorBase<S> sc = processorCache.pop();
    if (sc == null) {
        sc = createSocketProcessor(socketWrapper, event);
    } else {
        sc.reset(socketWrapper, event);
    }
    //线程池运行
    Executor executor = getExecutor();
    if (dispatch && executor != null) {
```

```
        executor.execute(sc);
    } else {
        sc.run();
    }
}
```

设计思想：将不同的业务处理所使用的资源隔离开

## Spring Boot如何使用内嵌式的Tomcat和Jetty

### Spring Boot中与Web容器相关的接口

```
// 对Tomcat jetty netty 服务器的接口
public interface WebServer {
    void start() throws WebServerException;
    void stop() throws WebServerException;
    int getPort();
}
```

```
public interface ServletWebServerFactory {
    WebServer getWebServer(ServletContextInitializer... initializers);
}
```

```
// getWebServer方法会调用初始化器的onStartup方法 在Servlet容器启动的时候做一些事情
public interface ServletContextInitializer {
    void onStartup(ServletContext servletContext) throws ServletException;
}
```

### 内嵌式Web容器的创建和启动

```
// 通过重写ApplicationContext的抽象子类的onRefresh方法
@Override
protected void onRefresh() {
    super.onRefresh();
    try {
        //重写onRefresh方法， 调用createWebServer创建和启动Tomcat
        createWebServer();
    }
}
```

```
        catch (Throwable ex) {
    }
}

//createWebServer的具体实现
private void createWebServer() {
    //这里WebServer是Spring Boot抽象出来的接口，具体实现类就是不同的Web容器
    WebServer webServer = this.webServer;
    ServletContext servletContext = this.getServletContext();

    //如果Web容器还没创建
    if (webServer == null && servletContext == null) {
        //通过Web容器工厂来创建
        ServletWebServerFactory factory = this.getWebServerFactory();
        //注意传入了一个"SelfInitializer"
        this.webServer = factory.getWebServer(new ServletContextInitializer[]
{this.getSelfInitializer()});
    }

    } else if (servletContext != null) {
        try {
            this.getSelfInitializer().onStartup(servletContext);
        } catch (ServletException var4) {
            ...
        }
    }

    this.initPropertySources();
}

// 通过调用Tomcat的API来创建组件
public WebServer getWebServer(ServletContextInitializer... initializers) {
    //1.实例化一个Tomcat，可以理解为Server组件。
    Tomcat tomcat = new Tomcat();

    //2. 创建一个临时目录
    File baseDir = this.baseDirectory != null ? this.baseDirectory :
this.createTempDir("tomcat");
    tomcat.setBaseDir(baseDir.getAbsolutePath());

    //3.初始化各种组件
    Connector connector = new Connector(this.protocol);
    tomcat.getService().addConnector(connector);
    this.customizeConnector(connector);
    tomcat.setConnector(connector);
    tomcat.getHost().setAutoDeploy(false);
    this.configureEngine(tomcat.getEngine());

    //4. 创建定制版的"Context"组件。
}
```

```
        this.prepareContext(tomcat.getHost(), initializers);
        return this.getTomcatWebServer(tomcat);
    }
```

## Web容器的定制

通过特定的Web容器的工厂来进行对特定的容器的配置

```
// 如：给Tomcat增加一个Valve 用于在请求头中添加traceId 用于分布式追踪
class TraceValve extends ValveBase {
    @Override
    public void invoke(Request request, Response response) throws IOException,
ServletException {

        request.getCoyoteRequest().getMimeHeaders().
        addValue("traceid").setString("1234xxxxabcd");

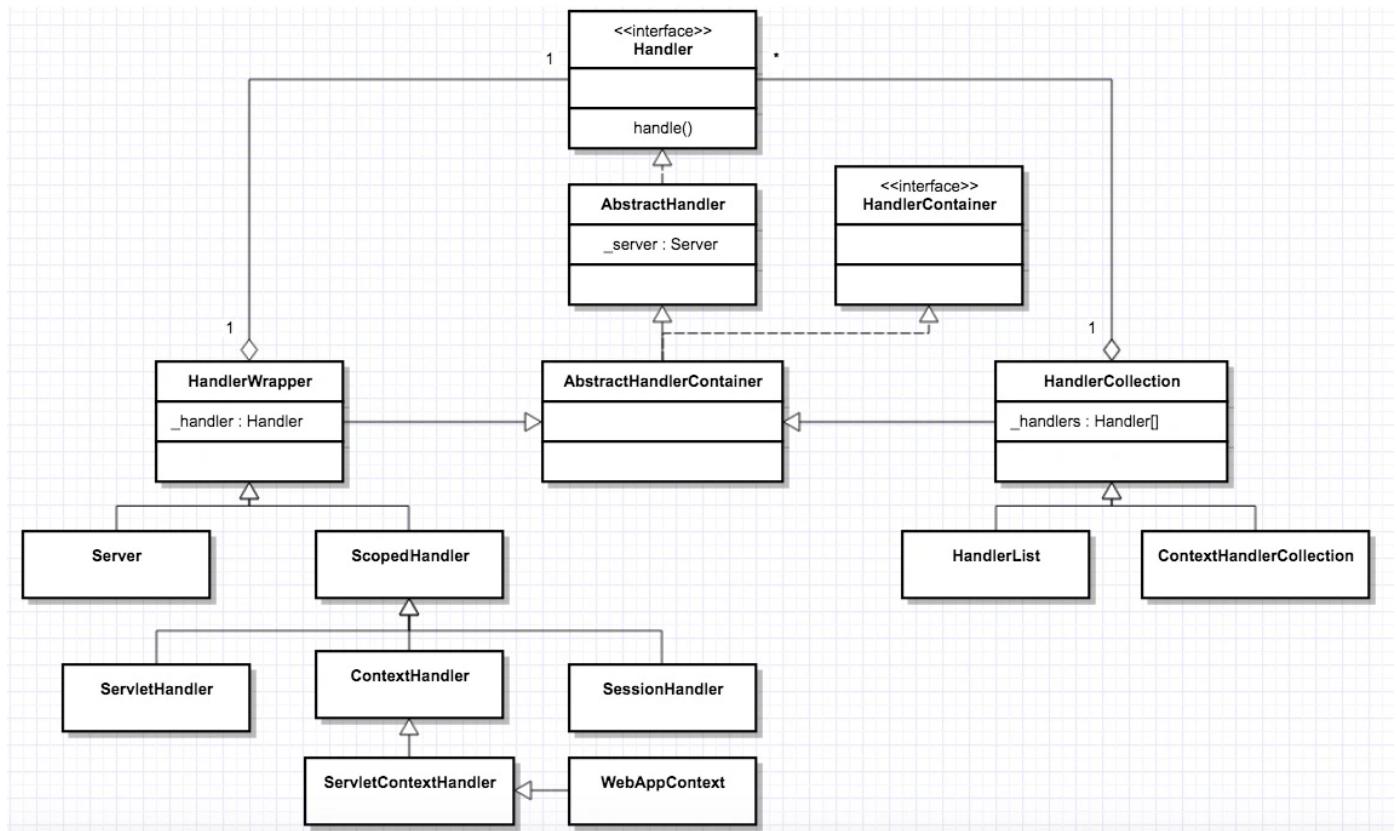
        Valve next = getNext();
        if (null == next) {
            return;
        }

        next.invoke(request, response);
    }
}

// 添加一个定制器
@Component
public class MyTomcatCustomizer implements
    WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    @Override
    public void customize(TomcatServletWebServerFactory factory) {
        factory.setPort(8081);
        factory.setContextPath("/hello");
        factory.addEngineValves(new TraceValve());
    }
}
```

# Jetty如何实现具有上下文信息的责任链



`scopedHandler` : 所有与Servlet规范相关的Handler都是其子类

## ScopedHandler链式调用的实现

```
public class HandlerWrapper extends AbstractHandlerContainer
{
    protected Handler _handler;

    @Override
    public void handle(String target,
                       Request baseRequest,
                       HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException
    {
        Handler handler=_handler;
        if (handler!=null)
            handler.handle(target,baseRequest, request, response);
    }
}
```

```
// scopedHandler
```

```
// 头结点
protected ScopedHandler _outerScope;
// 下一个scopedHandler结点
protected ScopedHandler _nextScope;

// 线程私有数据 因为有的信息需要在函数调用中传递
// 参数中没有 但是又要用到 所以用线程私有数据
private static final ThreadLocal<ScopedHandler> __outerScope= new
ThreadLocal<ScopedHandler>();

public final void handle(String target,
                         Request baseRequest,
                         HttpServletRequest request,
                         HttpServletResponse response)
                         throws IOException, ServletException
{
    if (isStarted())
    {
        // 若为头结点则使用doScope方法 进行初始化操作
        if (_outerScope==null)
            doScope(target,baseRequest,request, response);
        else
            doHandle(target,baseRequest,request, response);
    }
}

// 设置_nextScope _outerScope是为了确保先执行完所有的doScope方法
// 再执行所有的doHandle方法
public void doScope(String target,
                     Request baseRequest,
                     HttpServletRequest request,
                     HttpServletResponse response)
                     throws IOException, ServletException
{
    nextScope(target,baseRequest,request,response);
}

public final void nextScope(String target,
                            Request baseRequest,
                            HttpServletRequest request,
                            HttpServletResponse response)
                            throws IOException, ServletException
{
    if (_nextScope!=null)
        _nextScope.doScope(target,baseRequest,request, response);
    else if (_outerScope!=null)
        _outerScope.doHandle(target,baseRequest,request, response);
    else

```

```

        doHandle(target,baseRequest,request, response);
    }

public abstract void doHandle(String target,
                               Request baseRequest,
                               HttpServletRequest request,
                               HttpServletResponse response)
    throws IOException, ServletException;

public final void nextHandle(String target,
                             final Request baseRequest,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws IOException, ServletException
{
    if (_nextScope!=null && _nextScope==_handler)
        _nextScope.doHandle(target,baseRequest,request, response);
    else if (_handler!=null)
        super.handle(target,baseRequest,request,response);
}

```

```

// ScopedHandler启动的时候 会调用该方法 设置_outScope和_nextScope
@Override
protected void doStart() throws Exception
{
    try
    {
        //请注意_outScope是一个实例变量，而__outerScope是一个全局变量。先读取全局的线程私有变量
        __outerScope到_outerScope中
        __outerScope=__outerScope.get();

        //如果全局的__outerScope还没有被赋值，说明执行doStart方法的是头节点
        if (_outerScope==null)
            //handler链的头节点将自己的引用填充到__outerScope
            __outerScope.set(this);

        //调用父类HandlerWrapper的doStart方法
        super.doStart();
        //各Handler将自己的_nextScope指向下一个ScopedHandler
        _nextScope= getChildHandlerByClass(ScopedHandler.class);
    }
    finally
    {
        // 回调头结点 将线程私有数据清空 因为线程可能会被重用
        if (_outerScope==null)
            __outerScope.set(null);
    }
}

```

```
    }  
}
```

## Spring中的设计模式

简单工厂：静态工厂方法，根据传入的参数来动态决定创建哪个产品类(反射创建Bean)

```
public interface BeanFactory {  
    Object getBean(String name) throws BeansException;  
    <T> T getBean(String name, Class<T> requiredType);  
    Object getBean(String name, Object... args);  
    <T> T getBean(Class<T> requiredType);  
    <T> T getBean(Class<T> requiredType, Object... args);  
    boolean containsBean(String name);  
    boolean isSingleton(String name);  
    boolean isPrototype(String name);  
    boolean isTypeMatch(String name, ResolvableType typeToMatch);  
    boolean isTypeMatch(String name, Class<?> typeToMatch);  
    Class<?> getType(String name);  
    String[] getAliases(String name);  
}
```

工厂方法模式：一个工厂只对应一个相应的对象

```
public interface FactoryBean<T> {  
    T getObject();  
    Class<?> getObjectType();  
    boolean isSingleton();  
}
```

单例模式：单例注册表

```
public class DefaultSingletonBeanRegistry {  
  
    //使用了线程安全容器ConcurrentHashMap，保存各种单实例对象  
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>;  
  
    protected Object getSingleton(String beanName) {
```

```

//先到HashMap中拿Object
Object singletonObject = singletonObjects.get(beanName);

//如果没拿到通过反射创建一个对象实例，并添加到HashMap中
if (singletonObject == null) {
    singletonObjects.put(beanName,
        Class.forName(beanName).newInstance());
}

//返回对象实例
return singletonObjects.get(beanName);
}
}

```

代理模式：动态代理

## Manager组件：Tomcat的Session管理机制解析

### Session的创建

Context容器中通过Manager管理Session

```
protected Map<String, Session> sessions = new ConcurrentHashMap<>();
```

```

// 添加和删除Session load unload来序列化和反序列化session(持久化)
public interface Manager {
    public Context getContext();
    public void setContext(Context context);
    public SessionIdGenerator getSessionIdGenerator();
    public void setSessionIdGenerator(SessionIdGenerator sessionIdGenerator);
    public long getSessionCounter();
    public void setSessionCounter(long sessionCounter);
    public int getMaxActive();
    public void setMaxActive(int maxActive);
    public int getActiveSessions();
    public long getExpiredSessions();
    public void setExpiredSessions(long expiredSessions);
    public int getRejectedSessions();
    public int getSessionMaxAliveTime();
    public void setSessionMaxAliveTime(int sessionMaxAliveTime);
    public int getSessionAverageAliveTime();
    public int getSessionCreateRate();
    public int getSessionExpireRate();
}
```

```
public void add(Session session);
public void changeSessionId(Session session);
public void changeSessionId(Session session, String newId);
public Session createEmptySession();
public Session createSession(String sessionId);
public Session findSession(String id) throws IOException;
public Session[] findSessions();
public void load() throws ClassNotFoundException, IOException;
public void remove(Session session);
public void remove(Session session, boolean update);
public void addPropertyChangeListener(PropertyChangeListener listener);
public void removePropertyChangeListener(PropertyChangeListener listener);
public void unload() throws IOException;
public void backgroundProcess();
public boolean willAttributeDistribute(String name, Object value);
}
```

```
request.getSession()
```

如果Cookie中有JSESSIONID 则通过这个ID获取session对象 否则就创建一个新的

```
public class Request implements HttpServletRequest {}

// 实际上获取的是一个包装类 用于避免细节暴露给使用者 装饰者模式
public class RequestFacade implements HttpServletRequest {
    protected Request request = null;

    public HttpSession getSession(boolean create) {
        return request.getSession(create);
    }
}
```

```
getSession()
```

```
// request类
public HttpSession getSession(boolean create) {
    Session session = doGetSession(create);
    if (session == null) {
        return null;
    }
    // 这个方法中创建了request的包装类 并返回包装类
    return session.getSession();
}
```

```
// request
protected Session doGetSession(boolean create) {
    // There cannot be a session if no context has been assigned yet
    Context context = getContext();
    if (context == null) {
        return null;
    }
    // Return the current session if it exists and is valid
    if ((session != null) && !session.isValid()) {
        session = null;
    }
    // request自身持有的Session一级缓存 如果该request之前调用过这个方法 就会有
    if (session != null) {
        return session;
    }
    // Return the requested session if it exists and is valid
    Manager manager = context.getManager();
    if (manager == null) {
        return null;          // Sessions are not supported
    }
    if (requestedSessionId != null) {
        // 根据sessionId到map中找 map可以看作是二级缓存
        session = manager.findSession(requestedSessionId);
        if ((session != null) && !session.isValid()) {
            session = null;
        }
        if (session != null) {
            // 刷新访问时间
            session.access();
            return session;
        }
    }
    // map中没有找到 通过manager创建
    session = manager.createSession(sessionId);
    // Creating a new session cookie based on that session
    if (session != null && trackModesIncludesCookie) {
        Cookie cookie = ApplicationSessionCookieConfig.createSessionCookie(
            context, session.getIdInternal(), isSecure());
        response.addSessionCookieInternal(cookie);
    }
    if (session == null) {
        return null;
    }
    session.access();
    return session;
}

// ManagerBase
```

```
@Override
public Session createSession(String sessionId) {
    //首先判断session数量是不是到了最大值，最大session数可以通过参数设置
    if ((maxActiveSessions >= 0) &&
        (getActiveSessions() >= maxActiveSessions)) {
        rejectedSessions++;
        throw new TooManyActiveSessionsException(
            sm.getString("managerBase.createSession.ise"),
            maxActiveSessions);
    }

    // 重用或者创建一个新的Session对象，请注意在Tomcat中就是StandardSession
    // 它是 HttpSession 的具体实现类，而 HttpSession 是 Servlet 规范中定义的接口
    Session session = createEmptySession();

    // 初始化新Session的值
    session.setNew(true);
    session.setValid(true);
    session.setCreationTime(System.currentTimeMillis());
    session.setMaxInactiveInterval(getContext().getSessionTimeout() * 60);
    String id = sessionId;
    if (id == null) {
        id = generateSessionId();
    }
    // 这里会将Session添加到ConcurrentHashMap中 同时通知监听器
    session.setId(id);
    sessionCounter++;
    //将创建时间添加到LinkedList中，并且把最先添加的时间移除
    //主要还是方便清理过期Session
    SessionTiming timing = new SessionTiming(session.getCreationTime(), 0);
    synchronized (sessionCreationTiming) {
        sessionCreationTiming.add(timing);
        sessionCreationTiming.poll();
    }
    return session
}

// StandardSession
public void setId(String id, boolean notify) {
    // 创建session的时候传入了一个Manager
    if ((this.id != null) && (manager != null))
        manager.remove(this);
    this.id = id;
    if (manager != null)
        manager.add(this);
    if (notify) {
        // 通知监听器 session 创建事件发生
        tellNew();
    }
}
```

```

    }

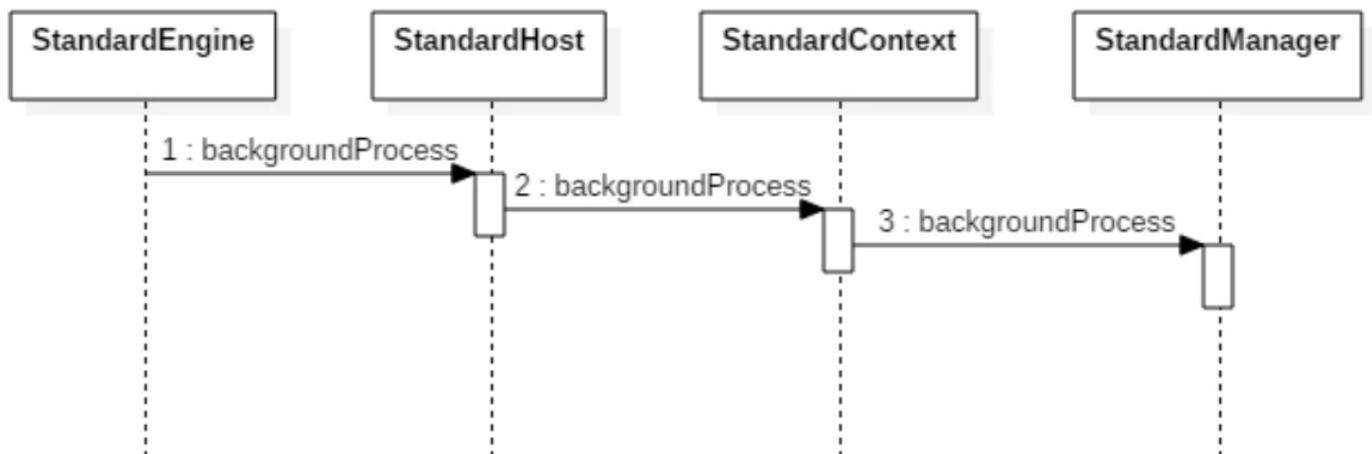
}

public class StandardSession implements HttpSession, Session, Serializable {
    protected ConcurrentHashMap<String, Object> attributes = new ConcurrentHashMap<>();
    protected long creationTime = 0L;
    protected transient volatile boolean expiring = false;
    // 对外暴露的是包装类
    protected transient StandardSessionFacade facade = null;
    protected String id = null;
    protected volatile long lastAccessedTime = creationTime;
    protected transient ArrayList<SessionListener> listeners = new ArrayList<>();
    protected transient Manager manager = null;
    protected volatile int maxInactiveInterval = -1;
    protected volatile boolean isNew = false;
    protected volatile boolean isValid = false;
    protected transient Map<String, Object> notes = new Hashtable<>();
    protected transient Principal principal = null;
}

```

## Session的清理

Tomcat的定时任务线程 ContainerBackgroundProcessor 会调用容器自身的 backgroundProcess 方法和子容器的后台处理方法



StandardContext中的backgroundProcess会调用Manager的backgroundProcess方法

```

// StandardManager
public void backgroundProcess() {
    // processExpiresFrequency 默认值为6, 而backgroundProcess默认每隔10s调用一次, 也就是说除了
    // 任务执行的耗时, 每隔 60s 执行一次
}

```

```

        count = (count + 1) % processExpiresFrequency;
        if (count == 0) // 默认每隔 60s 执行一次 Session 清理
            processExpires();
    }

/**
 * 单线程处理，不存在线程安全问题
 */
public void processExpires() {
    // 获取所有的 Session
    Session sessions[] = findSessions();
    int expireHere = 0 ;
    for (int i = 0; i < sessions.length; i++) {
        // Session 的过期是在isValid()方法里处理的
        if (sessions[i] !=null && !sessions[i].isValid()) {
            expireHere++;
        }
    }
}

```

## Session事件通知

```

public interface HttpSessionListener extends EventListener {
    //Session创建时调用
    public default void sessionCreated(HttpSessionEvent se) {
    }

    //Session销毁时调用
    public default void sessionDestroyed(HttpSessionEvent se) {
    }
}

public void tellNew() {

    // 通知org.apache.catalina.SessionListener
    fireSessionEvent(Session.SESSION_CREATED_EVENT, null);

    // 获取Context内部的LifecycleListener并判断是否为HttpSessionListener
    Context context = manager.getContext();
    Object listeners[] = context.getApplicationLifecycleListeners();
    if (listeners != null && listeners.length > 0) {

        //创建HttpSessionEvent
        HttpSessionEvent event = new HttpSessionEvent(getSession());
        for (int i = 0; i < listeners.length; i++) {

```

```
//判断是否是HttpSessionListener
if (!(listeners[i] instanceof HttpSessionListener))
    continue;

HttpSessionListener listener = (HttpSessionListener) listeners[i];
//注意这是容器内部事件
context.fireContainerEvent("beforeSessionCreated", listener);
//触发Session Created 事件
listener.sessionCreated(event);

//注意这也是容器内部事件
context.fireContainerEvent("afterSessionCreated", listener);

}

}

}
```

```
java -Xmx32m -Xss400k -verbose:gc -
Xloggc:/Users/yangsiping/programme/Java/designpattern/gc.log -XX:+PrintGCTimeStamps -
XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:+UseGCLogFileRotation -
XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=1024k -jar
```