# Http相关

## 简介

http协议是应用层协议，基于tcp/ip协议，是浏览器与服务器之间的数据传送协议。

注：仅规定了数据传输的格式，与数据如何传输无关（数据传输由tcp完成）。

http协议不涉及数据包传输。HTTP只是一种数据包的格式，数据包具体是由TCP来传输的。

## 特点

### 无状态

http是无状态的。http协议的无状态是针对每一次请求和响应的。对同一个url请求没有上下文关系，每次的请求都是独立的，服务器中没有保存客户端的状态。

### 长连接

HTTP长连接是指建立一次TCP连接后长时间不关闭，供后面的请求使用，本质上是TCP的长连接。http1.1默认长连接，短连接现在用的很少了。

实现：http长连接本质上是基于tcp连接的不关闭实现。而对于两次http请求来说依然没有直接关系的，所以说http是无状态的。

目的：长连接省去了较多的TCP建立、关闭操作，使得每次发送请求不用重新建立tcp连接，减少开销。而保持连接的时间长短由服务端决定，比如Tomcat中的server.xml文件就有相关配置。

保持连接有开销的，因此服务端会给连接设置超时时间，一段时间内连接上没请求数据就会关闭连接；以及设置最大连接数限制。

设置：HTTP1.0中在响应头配置“Connection: keep-alive”。HTTP1.1默认是keep-alive，不需要在服务端配置。http1.0像是买一个信封只能传送一个来回的信。http1.1像是买一个信封可以重复使用，但前提是得等到服务端把这个信封送回来。

连接超时情况：

1、某个TCP连接上一段时间内没有请求，服务端为了释放资源就关闭这个连接。这个超时在服务端设置。

2、浏览器发起一个请求后，在等待服务端的响应时超时，这个超时值是在浏览器端设置。

问1：服务端怎么检测这个tcp链接什么时候可以销毁释放？

服务端会设置连接超时时间，如果TCP连接上超过一段时间没有请求数据，服务端会关闭这个连接。

问2：HTTP的无状态和keep-alive矛盾吗？

connection:keep-alive是告诉tcp在单次请求完成后不要关闭连接，下次再请求时依然使用之前的连接，是依靠tcp协议的特点实现的，和http没关系。

http是应用层协议重点说明接收到的数据要如何解析，本身并不关心之前的请求是什么样，只要根据协议理解本次请求发过来的东西是什么就好。后来因为程序功能需要多个http请求间也能完成相互依赖的操作，就有了cookie和session。即使没有cookie和session，对使用http协议解读请求数据也是没有任何影响的。

长连接和http的无状态说的应该是不同层面上的东西，所以说这两者之间是不矛盾的。

## tcp协议&http协议

http是应用层协议，规定了数据格式；tcp是网络层协议，用于数据传输。

应用层的HTTP协议是无状态的，但可以使用cookie和session来保存请求之间的关系。

一般把“面向连接”来描述传输层的TCP协议，TCP是有状态的，TCP连接过程中有三次握手，因为通信两端需要维护一个TCP的状态机。

主要是两个过程：建立连接和数据传输！建立连接需要经过tcp/tp的三次握手；数据传输需要统一按照http协议规定的格式。

问：如果一个连接里面处理一个长事物，其他的请求会不会排队等待？

在HTTP1.1中，请求是按顺序排队处理的，前面的HTTP请求处理会阻塞后面的HTTP请求，虽然HTTP pipelining对连接请求做了改善，但是复杂度太大，并没有普及，这个问题在HTTP2.0中得到了解决。

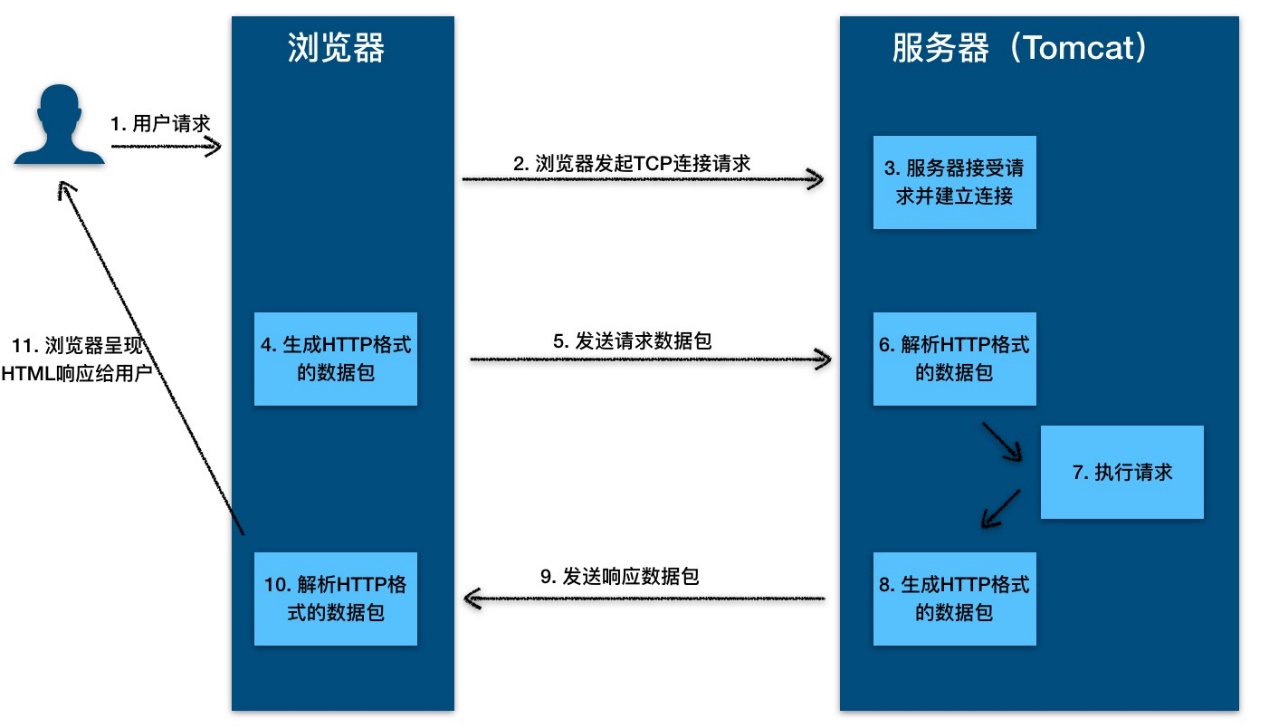
## http请求过程

浏览器从远程http服务器获取一个html文本过程中主要做两件事：

-1） 与服务器建立Socket连接

-2） 生成请求数据并通过Socket发送出去

-3） 解析服务器发送过来的响应数据



说明：

-1）用户通过浏览器访问某网址或点击某a标签；

-2）浏览器向服务器发送tcp连接请求；

-3）服务器接收浏览器的连接请求，并经过tcp三次握手后建立连接；

-4）浏览器将请求数据打包成http协议格式的数据包；

-5）浏览器将该数据包推入网络，数据包经过网络传输，最终达到服务端程序；

-6）服务端程序拿到这个数据包后，同样以http协议格式解包；

-7）服务器通过解包数据执行请求，生成响应结果；

-8）服务端将响应结果（html文本、图片）打包成http协议格式的数据包；

-9）服务器将响应数据包推入网络，数据包经过网络传输最终达到浏览器；

-10）浏览器拿到响应数据包后，以http协议的格式解包，然后解析数据，假设这里的数据是html；

-11）浏览器将html文件展现在页面上。

Tomcat和Jetty作为http服务器，在这个过程中都做了什么？

接收并建立连接、解析请求数据、处理请求、发送响应

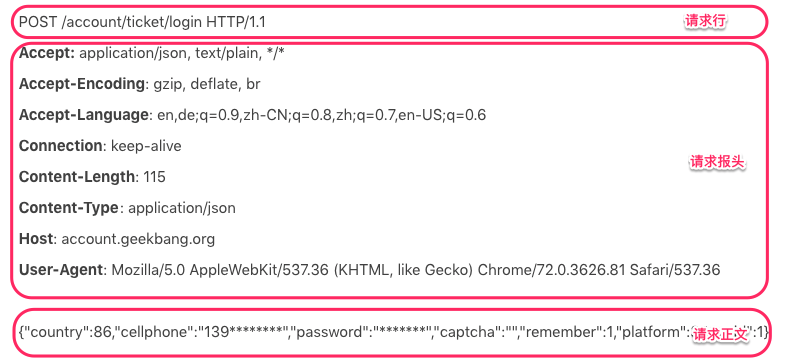
附：若有成千上万的浏览器请求同一个http服务器，因此Tomcat和Jetty为了提高服务的能力和并发度，往往会将自己要做的几个事情并行化，具体使用多线程技术。

## http数据包

HTTP请求数据：

由三部分组成：请求行、请求报头、请求正文。

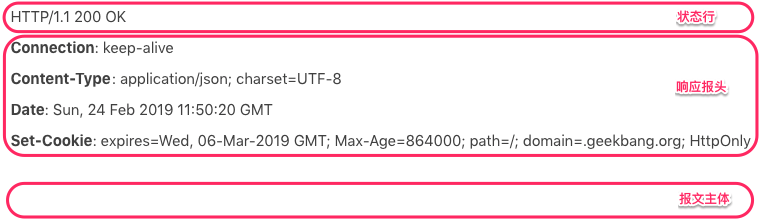
当http请求数据到达Tomcat后，Tomcat会把Http请求数据字节流解析成一个Request对象，这个Request对象封装了Http所有的请求信息。接着Tomcat把这个Request对象交给Web应用去处理，处理完后得到一个Response对象，Tomcat会把这个Response对象转换成Http格式的响应数据并发送给浏览器。



登录请求示例图

HTTP响应数据：

由三部分组成：状态行、响应报头、报文主体。



登录请求的响应示例图

# Cookie&Session&Token理解

## 发展史

详情：https://www.jianshu.com/p/30bd9c229751

1、以前的web基本就是文档的浏览，不需要记录谁在某一段时间里都浏览了什么文档，每次请求都是一个新的http协议（请求+响应）。

2、随着交互式web应用兴起，如在线购物网站需要登录的网站，必须记住那些人登录系统，哪些人在购物车上放商品。http请求是无状态的，所以想到的方法就是给大家发一个会话标识（session id），每个人收到的都不一样，每次大家向我发起http请求时，把这个字符串一并捎过来，这样服务器就能区分谁谁了。

3、可是这样的问题就是：每个人仅保存自己的sessionId，而服务器保存的是所有人的sessionId！这样服务器存储的数据是一个巨大的开销，另一问题就是服务器集群时的各服务器存储sessionId会不一样。

服务器集群时的sessionId不同问题解决：使用memcached或redis。把sessionId集中存储到一个地方，所有机器都来访问这个地方的数据。这样也带来的另一问题：负责session的机器挂了怎么办？解决：负责session的机器也搞出集群，增加可靠性。

4、**如果服务器不保存sessionId，而是验证客户端发过来的sessionId即可**。这里的关键点就是验证！这样就能解除了sessionId的负担！

例如：小A已登录系统，系统给他发一个令牌（token），里面包含了小A的userId，下次小F再次通过http请求访问系统时，把这个token通过http header带过来。

问题：如何防止别人伪造。答：对数据做一个签名，使用hmac-sha256算法加上密钥，对这个签名和数据一起作为token，别人不知道密钥就无法伪造token了。

这个token 我不保存， 当小F把这个token 给我发过来的时候，我再用同样的HMAC-SHA256 算法和同样的密钥，对数据再计算一次签名， 和token 中的签名做个比较， 如果相同， 我就知道小F已经登录过了，并且可以直接取到小F的user id , 如果不相同， 数据部分肯定被人篡改过， 我就告诉发送者： 对不起，没有认证。

## Cookie介绍

Cookie由服务器生成，发送给浏览器，浏览器把cookie以kv形式保存到某个目录下的文本文件内，下一次请求同一网站时会把该cookie发送给服务器。

由于cookie是存在客户端上的，所以浏览器加入了一些限制确保cookie不会被恶意使用，同时不会占据太多磁盘空间，所以每个域的cookie数量是有限的。

### 概念

基本通信流程：设置cookie-->cookie被自动添加到request的header中-->服务器接收cookie。

在console中输入document.cookie能查看到该页面相关的cookie信息

cookie属性：expires、domain、path、secure、HttpOnly

例如："key=name; expires=Thu, 25 Feb 2016 04:18:00 GMT; domain=ppsc.sankuai.com; path=/; secure; HttpOnly"

-1）expires属性：cookie的失效时间；

-2）domain属性：域名。在t.a.b.com下，指定domain为t.a.b.com、a.b.com、b.com都可行。domain=a.b.com和domain=.a.b.com是完全相同的；

-3）path属性：路径。注：domain和path一起来限制cookie能被哪些URL访问

### Cookie版单点登录

相关链接： https://segmentfault.com/a/1190000004556040

在不同的子域名之间实现单点登录的最佳途径是采用Cookie的Forms认证。他的原理简单的来说，就是让Forms Authentication生成的Cookie能够跨域名访问。

跨域访问：

-1）域A为t1.test.com，域B为t2.test.com

-2）在域A生产一个令域A和域B都能访问的cookie，则该cookie的domain设置为.test.com

-3）在域A生产一个令域A能访问而域B不能访问的cookie，则该cookie的domain设置为t1.test.com

### Cookie与Session区别

-1）Cookie是客户端技术，是由服务器将用户的数据写回给用户浏览器。

-2）Session是服务端技术，是把用户数据写到服务器后，为用户所创建的独享的Session对象中，但Session是基于Cookie的。

服务器在运行时可以为每一个用户的浏览器创建一个其独享的session对象，注意是默认情况下，一个浏览器独占一个session，用户把各自数据存放在各自的session中，当用户再去访问服务器的其他web资源时，其他web资源再从用户各自的session中取出数据为用户服务。

## Session介绍

服务器使用session把用户的信息临时保存在了服务器上，用户离开网站后session会被销毁。这种用户信息存储方式相对cookie来说更安全。

session有一个缺陷：如果web服务器做了负载均衡，那么下一个操作请求到了另一台服务器的时候session会丢失。

问1：JESSIONID生成

当你在应用程序调用了Request对象的getSession(boolean create)方法，并且传入的参数是true时，Session才会被创建出来，这样Tomcat才会在Response中添加JESSIONID的这个cookie。

问2：如果我从别人的浏览器中拿到jsessionid，把它放在我自己的请求头中，是不是在服务端对应同一个session，那是不是就可以窃取人家信息了？

是的。虽然敏感的用户信息没有在网络上传输了，但是攻击者拿到sessionid也可以冒充受害者发送请求，这就是为什么我们需要https，加密后攻击者就拿不到sessionid了，另外CSRF也是一种防止session劫持的方式。

问3：spring session与servlet session区别：

spring通过Servlet规范中的Filter机制拦截了所有Servlet请求，偷梁换柱，将标准的Servlet请求对象包装了一下，换成它自己的Request包装类对象，这样当程序员通过包装后的Request对象的getSession方法拿Session时，是通过Spring拿Session，没Web容器什么事了。

## Token介绍

在Web领域基于Token的身份验证随处可见。在大多数使用Web API的互联网公司中，tokens 是多用户下处理认证的最佳方式。

特性会让你在程序中使用基于Token的身份验证：

无状态、可扩展

支持移动设备

跨程序调用

安全

未完成！！！！

https://www.jianshu.com/p/30bd9c229751

# Servlet相关

相关链接：

https://www.cnblogs.com/GtShare/p/8033637.html

https://www.aliyun.com/jiaocheng/306417.html

## Servlet生命周期

声明周期：加载（由web服务器完成）🡪实例化🡪服务🡪销毁

-1）init()方法

在Servlet的生命周期中，仅此执行一次init()方法。用于初始化Servlet对象。

-2）service()方法

Servlet的核心，负责处理和响应客户端的请求。

每当一个客户请求一个HttpServlet对象，该对象的service()方法就要调用，而且传递给这个方法一个请求对象ServletRequest和一个响应对象ServletResponse作为参数。

默认的服务功能是调用与http请求的方法相应的doGet/doPost功能。

-3）destroy()方法

仅执行一次，在服务器端停止且卸载Servlet时执行该方法。用于释放占用的资源。

## Servlet处理多个请求访问

servlet单例：

在web访问中，每多一个用户，会在web容器中多一个用户线程。

多用户同时访问某个servlet的doget方法时，由于方法的执行在线程中是私有的，每个线程执行某个方法不影响。

servlet是单例的，经过测试，init()方法只在servlet加载时执行，destroy()只在销毁时执行。

servlet容器默认采用“单实例多线程”方式处理多个请求：

-1）当web服务器启动时（或客户端第一次请求某个Servlet），Servlet容器将会根据web.xml配置文件实例化该Servlet类（只存在一个Servlet实例）。

-2）容器初始化Servlet主要就是读取配置文件（如tomcat可以通过server.xml的<Connector>设置线程池中线程数目，初始化线程池通过web.xml初始化每个参数值等等）。

-3）当请求到达时，Servlet容器通过调度线程(Dispatcher Thread)调度它管理下线程池中等待执行的线程(Work Thread)给请求者。

-4）线程执行Servlet的service()方法

-5）请求结束，放回线程池，等待被调用

好处：

-1）Servlet单例，减少了创建servlet的开销

-2）通过线程池来响应多个请求，提高了请求的响应时间

-3）servlet容器并不关心到达的Servlet请求访问的是否为同一Servlet，直接分配它一个新的线程。若是同一个Servlet的多个请求，那么该Servlet的service方法将在多线程中并发的执行。

-4）每一个请求由ServletRequest对象来接受请求，由ServletResponse对象来响应该请求。

注：避免使用成员变量

若存在成员变量，可能发生多线程同时访问和操作该变量，照成数据的不一致，因此产生线程安全问题。

## Java的内存模型JMM（Java Memory Model）

JMM主要是为了规定了线程和内存之间的一些关系。

根据JMM的设计，系统存在一个主内存（Main Memory）和多个工作内存（Working Memory）。

主内存：

Java中所有实例变量都存储在主内存中，对于所有线程都是共享的。

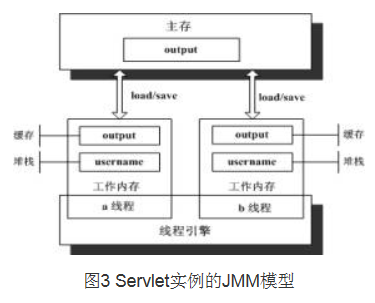
工作内存：

每条线程都有自己的工作内存，工作内存由缓存和堆栈两部分组成。

缓存中保存的是主内存中变量的拷贝，缓存可能并不和主存同步，即缓存中变量的修改可能没有立即写到主存中；

堆栈中保存的是线程的局部变量，线程之间无法相互直接访问堆栈中的变量。

Servlet实例的内存模型抽象：



## 工作者线程和调度线程

工作者线程（Work Thread）：执行代码的一组线程

调度线程（Dispatcher Thread）：每个线程都具有分配给它的线程优先级，线程是根据优先级调度执行的。

Servlet采用多线程来处理多个请求同时访问。Servlet依赖于一个线程池来服务请求，线程池实际上是一系列的工作者线程集合。Servlet使用一个调度线程来管理工作者线程。

当容器收到一个Servlet请求，调度线程从线程池中选出一个工作者线程，将请求传递给该工作者线程，然后由该线程来执行Servlet的service()方法。当这个线程正在执行时，容器收到另一请求，调度线程同样从线程池中选择另一工作者线程来执行Servlet的service()方法。容器并不关心这个请求是否访问的是同一Servlet，当容器同时接收对同一Servlet的多个请求时，该Servlet的service()方法将在多线程中并发执行。

Servlet容器默认采用单实例多线程的方式来处理请求的好处：减少产生Servlet实例的开销，提升了对请求的响应时间。

对于Tomcat可以在server.xml中通过<Connector>元素设置线程池中线程的数目。

## Servlet是非线程安全的

Servlet体系结构是建立在Java多线程机制之上的，其生命周期由Web容器负责。

当客户端第一次请求某个Servlet时，Servlet容器将会根据web.xml配置文件实例化该Servlet类。当有新的客户端请求该Servlet时，一般不会再实例化该Servlet类，即有多个线程在使用这个实例。Servlet容器会自动使用线程池等技术来支持系统的运行。

当两个或多个线程同时访问同一个Servlet时，可能会发生多个线程同时访问同一资源的情况，数据可能会变得不一致。

避免非线程安全问题：

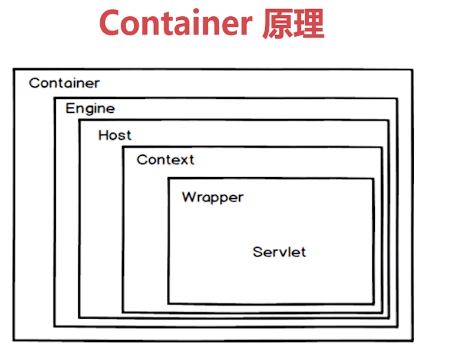
如果一个Servlet实现了SingleThreadModel接口，Servlet引擎将为每个新的请求创建一个单独的Servlet实例，这将引起大量的系统开销。（Servlet2.4不推荐使用）

线程安全问题主要是由实例变量造成的，因此在Servlet中应避免使用实例变量。

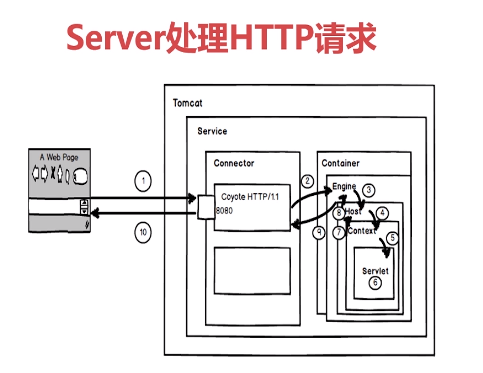
如果应用程序设计无法避免使用实例变量，那么使用同步synchronized来保护要使用的实例变量，但为保证系统的最佳性能，应该同步可用性最小的代码路径。

## Servlet在Tomcat的位置

Container组件：



Server处理Http请求：



# Tomcat相关

问：HTTP服务器、Servlet容器、Web应用它们是怎么一起工作的？

## 简介

Tomcat是由Apache开发的一个servlet容器，实现了对servlet和jsp支持，并提供作为web服务器的一些特有功能，如Tomcat管理和控制平台、安全域管理和Tomcat阈等。

Tomcat的重要目录：

-1）bin：Tomcat脚本存放目录（如启动、关闭脚本）。\*.sh文件用于Unix系统，\*.bat文件用于Windows系统；

-2）conf：Tomcat配置文件目录；

-3）logs：Tomcat默认日志目录

-4）Webapps：webapp运行目录

启动Tomcat：

1）将打包好的war包放在Tomcat安装目录下的webapps目录下

2）执行bin目录下的startup.bat或者startup.sh，Tomcat会自动解压webapps目录下的war包

3）访问<http://localhost:8080/xxx(xxx>是war包文件名)

### Apache、Tomcat、JBoss区别

-1）Apache是Http服务器，Tomcat是web服务器，JBoss是应用服务器

-2）Apache解析静态的html文件；Tomcat可解析jsp动态页面，也可充当servlet容器

## Tomcat总体架构

Tomcat实现2个核心功能：

-1）处理Socket连接，负责网络字节流与Request和Response对象的转化；

-2）加载和管理Servlet，以及具体处理Request请求。

因此Tomcat设计了两个核心组件：连接器（Connector）和容器（Container）。连接器负责对外交流，容器负责内部处理。

Tomcat的整体架构包含了两个核心组件：连接器和容器。连接器负责对外交流，容器负责内部处理。连接器用ProtocolHandler接口来封装通信协议和I/O模型的差异，ProtocolHandler内部又分为EndPoint和Processor模块，EndPoint负责底层Socket通信，Processor负责应用层协议解析。连接器通过适配器Adapter调用容器。

Tomcat支持的IO模型：

NIO：非阻塞I/O，采用Java NIO类库实现；

NIO2：异步I/O，采用JDK7最新的NIO2类库实现；

APR：采用Apache可移植运行库实现，是C/C++编写的本地库。

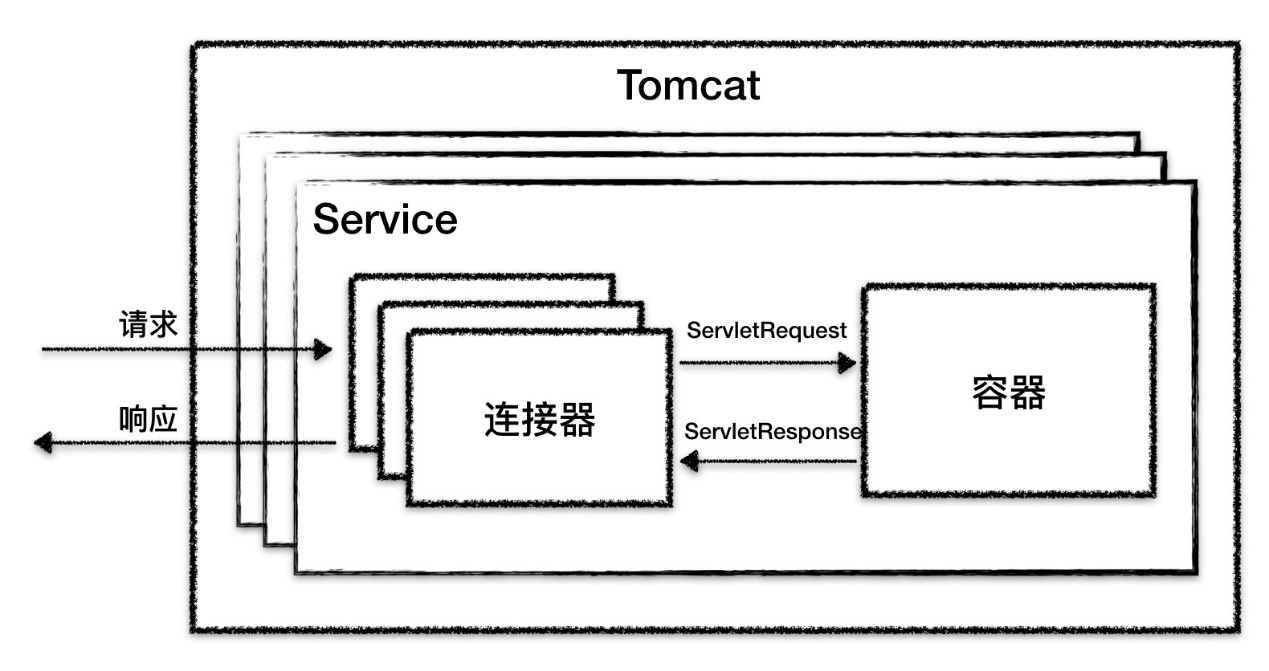
Tomcat支持的应用层协议有：

HTTP/1.1：这是大部分Web应用采用的访问协议；

AJP：用于和Web服务器集成（如Apache）；

HTTP/2.0：HTTP2.0大幅度的提升了Web性能。

Tomcat为了实现支持多种I/O模型和应用层协议，一个容器可能对接多个连接器，把容器和连接器组装在一起组成Service组件。Tomcat内可能有多个Service，通过在Tomcat中配置多个Service，可以实现通过不同的端口号来访问同一台机器上部署的不同应用。



由图片看出，最顶层是Server，这里的Server指的就是一个Tomcat实例。一个Server中有一个或多个Service，一个Service中有多个连接器和一个容器。连接器与容器之间通过标准的ServletRequest和ServletResponse通信。

## 配置文件(server.xml)

相关链接：http://www.cnblogs.com/kismetv/p/7228274.html

Server.xml的整体结构如下：

<Server>

<Service>

<Executor />

<Connector />

<Connector />

<Engine>

<Host>

<Context />

<Context />

</Host>

</Engine>

</Service>

</Server>

### Server元素

Server元素在最顶层，代表整个Tomcat容器，因此它必须是server.xml中唯一一个最外层的元素，它的属性代表了整个servlet容器的特征。

Server的作用是提供一个接口让客户端能够访问这个Service集合，同时维护所有Service的生命周期，包括如何初始化、如何结束服务、如何找到客户端要访问的Service。

格式：

<Server port="8005" shutdown="SHUTDOWN"> ... </Server>

参数说明：

className：默认org.apache.catalina.core.StandardServer

address：服务器等待关机命令的tcp/ip地址。默认localhost

port：表示Server接收shutdown指令的端口。设置为-1以禁用关闭；

shutdown：表示关闭Server的指令；

### Service元素

Service的作用是在Connector和Engine外面包了一层，让它们组装在一起，对外提供服务。

一个Server元素中可以有多个Service元素，不同的Service监听不同的端口。

一个Service可以包含多个连接器Connector，但是只能包含一个引擎组件Engine，多个连接器组件之间共享一个引擎组件。Connector作用是从客户端接收请求，Engine作用是处理接收进来的请求。

格式：

<Service name="Catalina">...</Service>

参数说明：

className：默认org.apache.catalina.core.StandardService

name：此服务的显示名称如果使用标准Catalina组件，将包含在日志消息中。与特定服务器关联的每个服务的名称必须是唯一的

### Executor元素

参见下文线程池Executor！！

### Connector元素

Connector在Service组件中可以存在多个。

Connector作用：从客户端接收连接请求，创建Request和Response对象用于和请求端交换数据。然后分配线程让Engine（即Servlet容器）来处理这个请求，并把产生的Request和Resonse对象传给Engine。当Engine处理完请求后，也会通过Connector将响应返回给客户端。

参见下文连接器Connector！！

### Engine元素

Engine组件在Service组件中有且只有一个；

Engine的作用：接收并处理来自一个或多个连接器Connector的所有请求，并将完成的响应返回给连接器，最终传递客户端。

格式：

<Engine defaultHost="localhost" name="Catalina">…</Engine>

参数说明：

defaultHost：默认主机名，用于标识将处理指向此服务器上主机名称但未在此配置文件中配置的请求的主机。

name：此引擎的逻辑名称，用于日志和错误消息。在同一服务器中使用多个服务元素时，每个引擎必须分配一个唯一的名称。

### Host元素

Host元素表示一个虚拟主机，它是一个服务器的网络名称（如www.xxx.com）与运行的Tomcat的特定服务器的关联。

格式：

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

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

< Context />

</Host>

</Engine>

Engine与Host：

Host是Engine的子容器。Engine组件中可以内嵌1个或多个Host组件，每个Host组件代表Engine中的一个虚拟主机。

Host组件至少有一个，且其中一个的name必须与Engine组件的defaultHost属性相匹配。

Host的作用：

Host虚拟主机的作用，是运行多个Web应用（一个Context代表一个Web应用），并负责安装、展开、启动和结束每个Web应用。

Host组件代表的虚拟主机，对应了服务器中一个网络名实体(如”www.test.com”，或IP地址”116.25.25.25”)；为了使用户可以通过网络名连接Tomcat服务器，这个名字应该在DNS服务器上注册。

### Context元素

Context元素表示一个web应用程序，它在特定的虚拟主机中运行。每个web应用程序都基于web应用程序存档文件（war）、或者war解包。

## Executor元素（线程池）

tomcat自带线程池机制，默认创建了10个线程（可调）

Executor是Service元素的内嵌元素，是Tomcat的线程池，可以被其它组件共享使用（通过executor属性指定）。

一般使用线程池的是Connector组件，因而Executor元素应该放在放在Connector前面。多个Connector共享一个线程池Executor，你可以定义多个不同名称的线程池。

格式

<Service name="xxx">

<Executor name="tomcatThreadPool" namePrefix ="catalina-exec-" maxThreads="1024" minSpareThreads="512" />

<Connector executor="tomcatThreadPool" port="8080" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443" acceptCount="1000" />

</Service>

参数说明：

className：默认org.apache.catalina.core.StandardThreadExecutor

name：线程池名称。要求唯一，供Connector元素的excutor属性引用。

namePrefix：线程名称前缀。线程池中线程名字为：namePrefix+线程编号

maxThreads：最大活跃线程数，默认200

minSpareThreads：最小活跃线程数，相当于初始化的线程数，默认25

maxIdleTime：线程空闲的最大时间。当前活跃线程数大于minSpareThreads时，空闲线程关闭的等待最大时间。默认60000ms（1分钟）

maxQueueSize：最大的等待队列个数，超过则请求拒绝，默认值是Integer.MAX\_VALUE，一般不改变。在某些紧急状态修复问题需要调整。

threadPriority：线程优先级，默认值5

daemon：是否后台线程，默认值true

测试：

--测试tomcat7和tomcat8中分配给Servlet的线程。

@WebServlet("/MyDemoServlet")

public class MyDemoServlet extends HttpServlet {

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

System.out.println(Thread.currentThread().getName());

}

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

doGet(req, resp);

}

}

查看服务器中的连接数和线程数：

查看连接情况：$ netstat -nat | grep 8080

查看进程：$ ps -e | grep java

查看该进程内有多少个线程：$ ps -o nlwp 27989

# nlwp含义是number of light-weight process

# 该进程内部有73个线程,但是73并没有排除处于idle状态的线程

# 获取真正running的线程数：$ ps -eLo pid ,stat | grep 27989 | grep running | wc -l

## Connector元素（连接器）

相关教程：

https://www.toutiao.com/a6699956472178016780

https://gearever.iteye.com/blog/1844203 --未学习！！！

Connector代表连接组件。它的配置和使用对Tomcat的性能有着重要的影响！

默认情况下Tomcat只支持200线程访问，超过这个数量的连接将被等待甚至超时放弃，所以我们需要提高这方面的处理能力。

连接器对Servlet容器屏蔽了协议及I/O模型等的区别，无论是HTTP还是AJP，在容器中获取的都是一个标准的ServletRequest对象。

一个Service中可以部署多个项目呢，一个连接器对应一个监听端口，不是一个请求，一个端口上可以接收多个请求。

格式：

<Service name="Catalina">

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

<Connector port="8009" protocol="AJP/1.3" redirectPort="8443"/>

</Service>

参数说明：

executor：对Executor元素的引用

port：访问需要的端口

protocol：使用的协议。三种协议：HTPP/1.1、HTTP/2.0、AJP

connectionTimeout：连接超时，会报错

redirectPort：重定向端口。需要安全通信的场合，将把客户请求转发至SSL的redirectPort端口。http协议不安全，https协议是安全的，配置好SSL之后可以用https协议通信

### 主要功能&处理流程

主要功能：

-1）接收连接请求，创建Request和Response对象用于和请求端交换数据；

-2）分配线程让Engine（即Servlet容器）来处理请求，并把Request和Response对象传给Engine；

-3）当Engine处理请求后，也会通过Connector将响应返回给客户端。

Connector处理请求流程：

相关链接：https://www.cnblogs.com/kismetv/p/7806063.html

--无论是BIO还是NIO，Connector处理请求的流程大致一样

-1）在accept队列中接收连接（当客户端向服务器发送请求时，若客户端与OS完成三次握手建立连接，则OS将该连接放入accept队列）；

-2）在连接中获取请求的数据，生成Request对象；

-3）调用Engine（Servlet容器）处理请求；

-4）返回Reponse对象。

连接与请求关系：

连接是tcp层面（传输层协议），对应socket；

请求是http层面（应用层协议），其依赖tcp的连接实现；

一个tcp连接可能传输多个http请求。若http请求使用的长连接，意味着一个tcp的socket在当前请求结束后，socket不会立马释放而是等待timeout后再释放，在此期间还能接收新的请求。

长连接与短链接：

HTTP/1.1默认keep-alive为true

### protocol属性

Connector在处理http请求时，会使用不同的protocol协议。包括BIO、NIO和APR（Tomcat7支持3种，Tomcat8增加对NIO2的支持，而Tomcat8.5和Tomcat9.0去掉了对BIO的支持）

指定protocol取值及对应的协议如下：

HTTP/1.1 默认

org.apache.coyote.http11.Http11Protocol：BIO

org.apache.coyote.http11.Http11NioProtocol：NIO

org.apache.coyote.http11.Http11Nio2Protocol：NIO2

org.apache.coyote.http11.Http11AprProtocol：APR

默认值HTTP/1.1，其含义：在Tomcat7中，自动选取使用BIO或APR（如果找到APR需要的本地库，则使用APR，否则使用BIO）；在Tomcat8中，自动选区使用的NIO或APR（如果找到APR需要的本地库，则使用APR，否则使用NIO）。

### BIO实现的Connector

处理请求的主要实体是JioEndpoint对象。

JioEndPoint维护了Acceptor和Worker：Acceptor接收socket，然后从Worker线程池中找出空闲的线程处理socket，如果woker线程池没有空闲线程则Acceptor将阻塞。

Worker是Tomcat自带的线程池，若通过<Executor>配置了其它线程池，原理类似。

### NIO实现的Connector

处理请求的主要实体是NioEndpoint对象。

NioEndpoint中除了包含Acceptor和Worker外，还用了Poller。

流程如下：

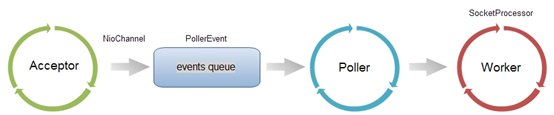
-1）Acceptor接收socket后，不是直接使用Worker中的线程处理请求，而是先将请求发送给Poller，而Poller是实现NIO的关键；

-2）Acceptor向Poller发送请求通过队列实现，使用了典型的生产者-消费者模式；

-3）在Poller中，维护了一个Selector对象；当Poller从Acceptor队列中取出socket后，注册到该Selector中；

-4）然后通过遍历Selector，找出其中可读的socket，并使用Worker中的线程处理相应请求。

与BIO类似，Worker也可以被自定义的线程池代替。



### BIO与NIO比较

在NioEndpoint处理请求的过程中，“Acceptor接收socket”和“线程处理请求”都是阻塞方式；但在“读取socket并交给Worker中的线程”这过程使用非阻塞的NIO实现，这就是NIO模式与BIO模式的最主要区别。而这个区别，在并发量较大的情形下可以带来Tomcat效率的显著提升。

由于多数使用长连接，使得一个tcp的socket在接收当前请求结束后，若没timeout则可以再次接收新的请求。若使用BIO，“读取socket并交给Worker中的线程”这个过程是阻塞的，意味着在socket等待下一个请求或等待释放的过程中，处理这个socket的工作线程会一直被占用，无法释放。因此Tomcat可以同时处理的socket数目不能超过最大线程数，性能受到极大限制。若使用BIO，“读取socket并交给Worker中的线程”这个过程是非阻塞的,当socket在等待下一个请求或等待释放时，并不会占用工作线程，因此Tomcat可以同时处理socket数目远大于最大线程数，并发性能大大提高。

### Connector参数说明

Tomcat同时处理的连接数目是maxConnections，但服务器可以同时接收的连接数是maxConnections+acceptCount。

-1）acceptCount

accept队列的长度。当Acceptor队列中连接的个数达到acceptCount时，队列满，进来的请求一律拒绝。默认100.

acceptCount设置：

-2）maxConnections

最大连接数。当Tomcat接收的连接数达到maxConnections时，Acceptor线程不会读取accept队列中的连接；这时accept队列中的线程会一直阻塞，直到Tomcat接收的连接数小于maxConnections。如果设置为-1，则连接数不受限制。

默认值与连接器使用的协议有关：NIO的默认值是10000，APR/native的默认值是8192，而BIO的默认值为maxThreads（如果配置了Executor，则默认值是Executor的maxThreads）。

maxConnections设置：与Tomcat的运行模式有关。如果tomcat使用的是BIO，那么maxConnections的值应该与maxThreads一致；如果tomcat使用的是NIO，那么类似于Tomcat的默认值，maxConnections值应该远大于maxThreads。

-3）maxThreads

最大线程数。默认200（Tomcat7和8都是）。如果该Connector绑定了Executor，这个值会忽略。

Tomcat通过使用比CPU核心数量多得多的线程数，可以使CPU忙碌起来，大大提高CPU的利用率。

maxThreads设置：与应用的特点和服务器CPU核心数量有关。一般CPU核心数越大，maxThreads应该越大；应用中CPU越不密集（IO越密集），maxThreads应该越大，以便能够充分利用CPU。需注意maxThreads过大时，CPU会花费大量的时间用于线程的切换。

## Tomcat优化

相关链接：https://segmentfault.com/a/1190000015918707

### 线程池Executor优化

conf/server.xml配置如下：

<Executor name="tomcatThreadPool"

namePrefix="catalina-exec-"

maxThreads="1024"

minSpareThreads="512"

prestartminSpareThreads="true" />

### 连接器Connector优化

Connector的运行模式分3种：BIO，NIO，APR。一般使用nio模式，bio效率低，apr对系统配置有比较高的要求。

conf/server.xml配置如下：

<Connector port="14081" protocol="org.apache.coyote.http11.Http11NioProtocol"

connectionTimeout="20000"

executor="tomcatThreadPool"

URIEncoding="UTF-8"

compression="on"

useBodyEncodingForURI="true"

enableLookups="false"

redirectPort="14443" />

配置说明：

-1）protocol

org.apache.coyote.http11.Http11Protocol //阻塞式Java连接器

org.apache.coyote.http11.Http11NioProtocol //不阻塞Java连接器（常用）

org.apache.coyote.http11.Http11AprProtocol //APR/native连接器

-2）enableLookups

若request.getRemoteHost()的调用履行，以便返回的长途客户端的实际主机名的DNS查询，则设置为true。设置为false时跳过DNS查找，并返回字符串的IP地址（从而提高性能）。默认场景下，禁用DNS查找。

-3）compression

设值成on，开启压缩。

-4）禁用AJP连接器

附：使用nginx+tomcat的架构，用不着AJP协议，所以把AJP连接器禁用。

Server.xml注释掉以下配置：

<Connector port="8019" protocol="AJP/1.3" redirectPort="8443" />

## Connector元素（各组件组成）

连接器实现功能细化：

-1） 监听端口号；

-2） 接收网络连接请求；

-3） 读取请求网络字节流；

-4） 根据具体应用层协议（HTTP/AJP）解析字节流，生成统一的Tomcat Request对象；

-5） 将Tomcat Request对象转成标准的ServletRequest对象‘

-6） 调用Servlet容器，得到ServletResponse对象；

-7） 将ServletResponse转成Tomcat Reponse对象；

-8） 将Tomcat Response转成网络字节流；

-9） 将响应字节流回写给浏览器。

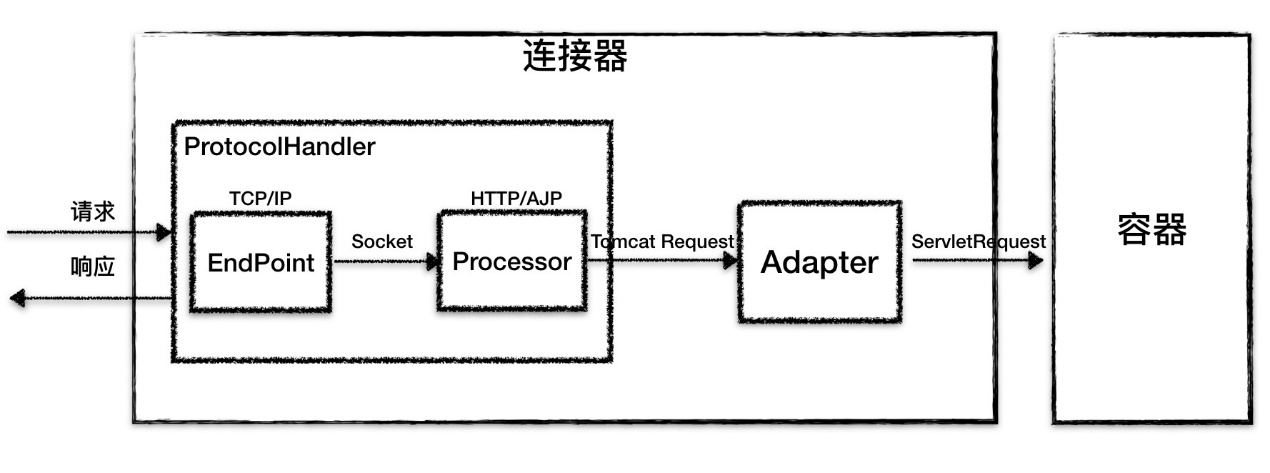
通过分析连接器的详细功能列表，我们发现连接器需要完成3个高内聚功能：

-1） 网络通信

-2） 应用层协议解析

-3） Tomcat Request/Response与ServletRequest/ServletResponse的转化

设计者设计了3个组件来实现者3个功能：EndPoint、Processor和Adapter。EndPoint负责提供字节流给Processor，Processor负责提供Tomcat Request对象给Adapter，Adapter负责提供ServletRequest对象给Servlet容器。其中EndPointh和Processor放在一起抽象成了ProtocolHandler组件。



### ProtocolHanlder组件

EndPointh和Processor放在一起抽象成了ProtocolHandler组件。ProtocolHandler 来处理网络连接和应用层协议。

### EndPoint组件

EndPoint利用Socket接口来将底层传来的数据转化成为HTTP格式的数据。

EndPoint是通信端点，即通信监听的接口，是具体的Socket接收和发送处理器，是对传输层的抽象，因此EndPoint是用来实现TCP/IP协议的。

EndPoint是一个接口，对应抽象实现类是AbstractEndPoint的具体子类，比如在NioEndPoint和Nio2EndPoint中，有两个重要的子组件：Acceptor和SocketProcessor。

其中Acceptor用于监听Socket连接请求。SocketProcessor用于处理接收到的Socket请求，它实现Runnable接口，在run方法里调用协议处理组件Processor进行处理。为了提高处理能力，SocketProcessor被提交到线程池来执行。该线程池叫执行器（Executor）。

### Processor组件

Processor是对应用层协议的抽象。如果说EndPoint是用来实现TCP/IP协议的，那么Processor是用来实现HTTP协议的。

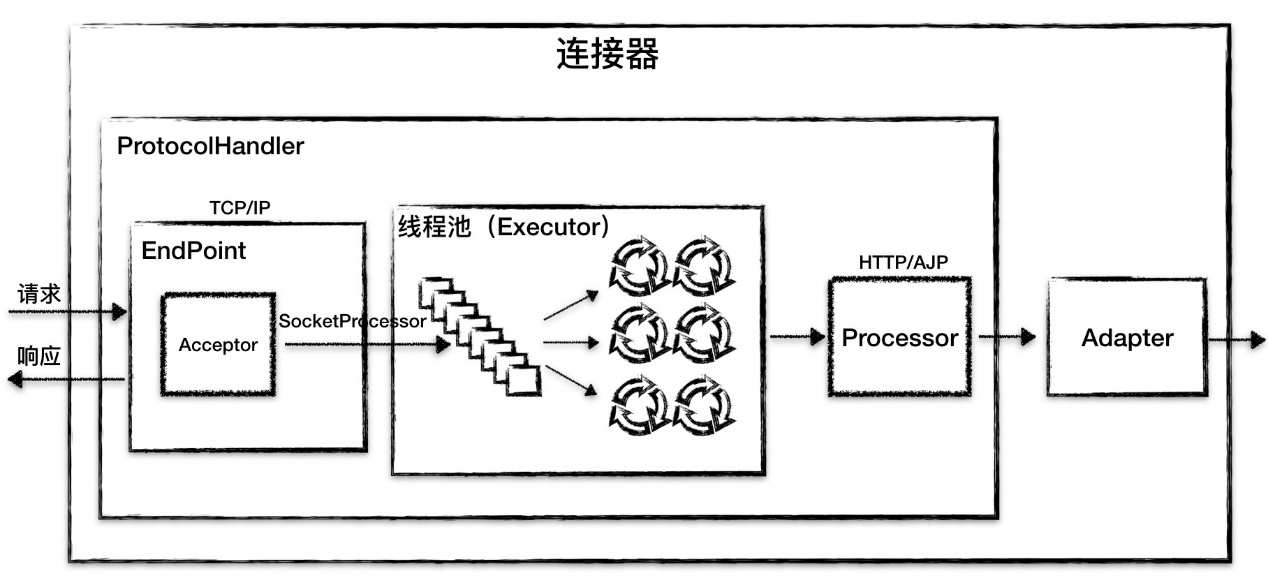
采用何种IO模式（NIO、NIO2、ARP），以及采用何种应用协议（HTTP1.1、AJP、HTTP/2)都是在processor这一层决定的。EndPoint只负责接收连接，并读取网络字节流但是不对字节流本身就进行任何解析。

EndPoint 是用来实现 TCP/IP 协议的，那么 Processor 用来实现 HTTP 协议。可理解成Endpoint负责Socket网络通信，跟TCP/IP协议紧密相关。

Processor接收来自EndPoint的Socket，读取字节流解析成Tomcat Request和Response对象，并通过Adapter将其提交到容器处理。

Processor是一个接口，定义了请求的处理等方法。它的抽象类AbstractProcessor对一些协议共有的属性进行了封装，没有对方法进行实现。具体实现有AJPProcessor、HTTP11Processor等。

连接器组件图：



由图可知，EndPoint接收到Socket连接后，生成一个SocketProcessor任务提交到线程池处理，SocketProcessor的run方法会调用Processor组件去解析应用层协议，Processor通过解析生成Request对象，然后调用Adapter的service方法。

### Adapter组件

由于协议不同，客户端发过来的请求信息也不尽相同，Tomcat定义了自己的Request类来“存放”这些请求信息。ProtocolHandler接口负责解析请求并生成Tomcat Request类。

ProtocolHandler接口负责解析请求并生成Tomcat Request类。但这个Request对象不是标准的ServletRequest，需要转换成ServletRequest对象。Tomcat的解决方案是引入适配器CoyoteAdapter，连接器调用CoyoteAdapter的service方法，将Tomcat Request转换成ServletRequest，再调用容器的service方法。

# Jetty相关

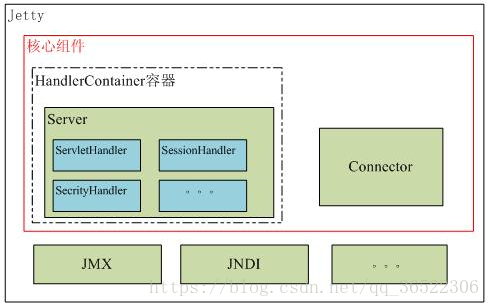
Jetty是一个提供http服务器、http客户端和javax.servlet容器的开源项目。

## Jetty与Tomcat对比

## 基本架构

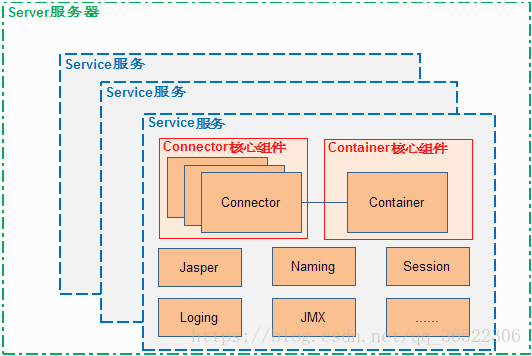
-1）Jetty架构

Jetty的架构比较简单，核心组件主要是由Server和Handler组成。Jetty中所有的组件都是基于Handler来实现的。



-2）Tomcat架构

Server是Tomcat的最顶层元素，包含了多个service。Server控制整个Tomcat的生命周期。Service由一个Container和多个Connector组成，形成一个独立完整的处理单元。



-3）Jetty与Tomcat比较

-1）Jetty比较容易贴合第三方框架，比如你可以直接用Spring配置一个Jetty服务器。

-2）直接可以将Jetty作为提供Http服务的组件，嵌入到应用中

-3）Jetty是面向Handler的架构，而Tomcat是面向容器的架构