# 一、Tomcat 介绍

## 1.1 Tomcat目录结构

|--**bin**：用于存放Tomcat的启动、停止等批处理脚本和Shell脚本

|--startup.sh/bat：用于启动Tomcat

|--shutdown.sh/bat：用于停止Tomcat

|--conf：用于存放Tomcat的相关配置文件

|--Catalina：用于存储针对每个虚拟机的Context配置

|--context.xml：用于定义所有Web应用均需要加载的Context配置，如果Web应用制定了自己的context.xml，那么改文件的配置将会被覆盖

|--catalina.properties：Tomcat环境变量配置

|--catalinna.policy：当Tomcat在安全模式下运行时，此文件为默认的安全策略配置

|--logging.properties：Tomcat日志配置文件，可通过改文件修改Tomcat日志级别以及日志路径

|--**server.xml**：Tomcat服务器核心配置文件，用于配置Tomcat的连接器，监听端口，处理请求的虚拟主机的等。可以说，Tomcat主要根据该问价你的配置信息创建服务器实例

|--tomca-users.xml：用于定义Tomcat默认用户及角色映射信息，Tomcat的Manager模块即用改文件中定义的用户进行安全认证

|--web.xml：Tomcat中所有应用默认的部署描述文件，主要定义了基础Servlet和MIME映射。如果应用中不包含web.xml，那么Tomcat将使用此文件初始化部署描述，反之，Tomcat会启动时将默认部署描述与自定义配置进行合并

|--lib：Tomcat服务器依赖库目录，包含Tomcat服务器运行环境依赖Jar包

|--logs：Tomcat默认的日志存放路径

|--**webapps**：Tomcat默认的Web应用部署目录

|--work：Web应用JSP代码生成和编译临时目录

# 二、Tomcat总体架构

## 2.1 Server

概念：表示整个Servlet容器，因此Tomcat运行环境中只有唯一一个Server示例

一个完整的服务器要完成的基本功能：

1. 接受客户端发来的请求，并进行解析
2. 完成相关业务处理
3. 将响应结果返回给客户端

Server元素代表着整个Catalinna servlet容器。

Servlet 容器的定义：能够监听请求，处理协议并处理请求的容器。

## 2.2 Connector 和 Engine

因为网络协议有多种，但是处理请求却是相同的，所以要实现网络协议与请求处理进行**抽象**和**分离。**

Connector 的作用：

1. 开启Socket并监听客户端请求，并将请求封装成Request、Response
2. 将请求传递给Engine
3. 返回响应数据

Engine的作用：负责具体的请求处理

Connector与 Engine相互配合，但如何知道Engine要处理来自哪个Connector的请求的？两者之间需要维护一个映射关系来进行更好地工作配合。

### 2.2.1 Connector

概念：是Tomcat链接器，用于监听并且转化Socket请求，同事将读取的Socket请求交给Engine处理，支持不同的协议以及不同的IO方式。

Connector的功能：

1. **监听**服务器端口，读取来自客户端的请求
2. 将请求数据按照指定的协议进行**解析**
3. 根据请求地址**匹配**正确的容器进行处理
4. 将**响应**返回客户端

Tomcat 支持多协议（HTTP、HTTP2、AJP），支持多种IO方式（BIO、NIO、APR）

ProtocolHandler表示一个协议处理器，针对**不同协议**和**IO方式**，提供了不同实现。（Http11NioProtocol表示基于NIO的HTTP协议的处理器）

ProtocolHandler：

1. Endpoint：用于启动Socket监听。
2. Processor：用于按照指定的协议读取数据，将请求交给容器处理（如Http11NioProcessor表示在NIO的方式下HTTP请求的处理类）

### 2.2.2 Engine

概念：Engine表示整个Servlet引擎。在Tomcat中，Engine为最高层级的容器对象。它是目标容器的入口。

## 2.3 Service

概念：Service表示一个或者多个Connector的集合，这些Connector共享同一个Engine来处理请求，在同一个Tomcat实例内可以包含任意多个Service实例，它们彼此独立。

一个Service可以维护多个Connector和一个Engine，来自这多个 Connector的不同的request都会给这一个Engine进行处理。当然，一个Server包含多个Service，这样的映射关系还有很多。

应用服务器是用来部署并且运行Web应用的，是一个运行环境，而不是一个独立的业务处理系统。因此，需要在Engine容器中支持管理Web应用，当接收到Connector的处理请求时，Engine容器能够找到一个合适的Web应用来处理。

### 2.3.1 Mapper 和MapperListener

当Processor读取客户端请求后，需要按照请求地址映射到具体的容器进行处理，这个过程称为**请求映射。**Tomcat通过Mapper和MapperListener实现。

Mapper：维护容器映射信息，按照映射规则查找容器

MapperListener：实现了ContainerListner和LifecycleListener，用于在容器组件状态发生变更时，注册或者取消对应的容器映射信息。

Tomcat通过适配器模式实现了Connector、Mapper、Engine三者的解耦。Mapper由Service来维护（因为Service从概念上的作用就是维护Connector与Engine之间的关系，所以Mapper是它的核心）

## 2.4 Context

概念：

使用Context来表示一个Web应用，并且一个Engine可以包含多个Context。当接受到来自多个Connector中不同的请求时，Engine就把这些请求分别分配给处理它们的Context中。它用于表示ServletContext。在Servlet规范中，一个ServletContext就表示一个独立的Web应用。

## 2.5 Host

### 2.5.1 引入Host

设想我们由一台主机，它承担了多个域名的服务，如：jd.com、taobao.com、tmall.com均由该主机处理。我们该如何实现？

1. 在该主机上运行多个Tomcat实例，每个实例对应一个域名
2. 使用Tomcat中的Host进行配置，每个Host配置一个域名（每一个Host作为一个虚拟主机）

概念：Host作为一类容器，表示Servlet引擎（Engine）的虚拟机，与一个服务器的域名有关。

### 2.5.2 如何理解一个域名对应一个Host

比如我们要在Tomcat中设置三个Host，其中他们分别是：

1. host\_1：jd.com
2. host\_2：taobao.com
3. host\_3：tmall.com

这三个Host分别处理不同的购物请求（用户到底是用京东来购物还是淘宝来购物）

每个Host中包含多个Context（Web应用）：

比如：jd.com对应的host\_1中包含三个应用，可以用应用名来分：

1. context\_1：user-app（/user/\*）
2. context\_2：order-app（/order/\*）
3. context\_3：good-app（/good/\*）

## 2.6 Wrapper（Servlet）

在一个Web应用中，可以包含多个Servlet示例来处理不同链接的请求，在Tomcat中，Servlet定义被称为Wrapper。

## 2.7 Container

Container 是一个抽象概念，它代表了一类组件。这类组件的作用就是处理来自上层组件的请求并且返回响应数据。Engine、Host、Context、Wrapper都是Container。

1. 接受来自上层组件的请求
2. 把请求委派给下层组件处理（如果是最底层，则自己处理）
3. 将处理后的结果返回给上层组件

## 2.8 Lifecycle

所有Container组件均存在启动、停止等生命周期方法，拥有生命周期管理的特性，所以对这些特性进行抽取，得到了Lifecycle接口。该接口定义了生命周期管理的核心方法：

1. init（）
2. start（）
3. stop（）
4. destroy（）

## 2.9 Pipeline 和 Valve

Tomcat采用了职责链模式来实现客户端的请求处理——请求处理也是职责链模式典型的应用场景之一。

Tomcat定义了Pipelin（管道）和Valve（阀）。前者用于苟泽职责链，后者代表职责链上的每一个处理器。来自客户端的请求就像是流经管道的水一样，经过每个阀进行处理。

每个层级的容器（Engine、Host、Context、Wrapper）均由对应的基础Valve实现，同事未获了一个Pipeline实例。

## Executor

Executor提供了一个可以在组件之间共享的线程池（默认使用了JDK5提供的线程池技术）Executor由Service来维护，因此同一个Service中的组件可以共享一个线程池。默认情况下，Endpoint会自动创建线程池来监听监听Socket端口，当接收到客户端的请求后，会创建请求处理对象，从线程池中获取一条线程，这条线程会把这个请求通过Mapper的映射，传递给匹配的处理者去处理。由此支持并发处理客户端请求。

# 三、Catalina

概念：Catailna 作为整个Servlet容器的正名，它包含了之前的所有容器组件。它是Tomcat的核心。

## Disgester

Digester是一款用于将XML转换成Java对象的事件驱动型工具，它来负责解析Tomcat中的server.xml，并创建应用服务器。

Digester以及SAX的事件驱动，简而言之，就是通过流去读取XML文件，当识别出特定的XML节点后，会执行特定的动作（创建Java对象、执行对象的某个方法等）。它的核心是**匹配模式**和**处理规则**。

## Web 应用的加载

Web应用的加载是由StandardHost来完成的。加载方式包含两类：

* 直接配置context.xml
* 通过HostConfig这个监听器来自动扫描Web应用目录或者War包

### 在server.xml 直接配置Context标签

|  |
| --- |
| <Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true">  <Context docBase="C:\App\project\open-platform\open-platform\open-report\open-report-business\open-report-web\target\open-report-web"  path="myReport" reloadable="true"></Context>  </Host> |

docBase：open-report编译后的根目录（WEB-INF文件夹的上一层，也就是Web应用根目录的文件路径）

path：在浏览器中进行访问的路径名称（Web应用的根请求路径）

reloadable：暂时没有实验

访问路径：<http://localhost:8080/myReport/>

这种情况,Context元素将会作为Host容器的子容器,添加到Host实例当中,当Host启动时候,有生命周期管理接口的start( )方法启动

如果部署的Web应用路径相对固定，且每个应用需要分别在特定的目录下进行管理，可以使用该方式，

### HostConfig自动扫描部署

自动扫描中，HostConfig处理的生命周期事件包括：

1. START\_EVENT
2. PERIODIC\_EVENT（periodic：周期的，定期的）
3. STOP\_EVENT

其中，前两者与Web应用部署有关，第三者用于Host停止时注销其对应的MBean。

#### 1、START\_EVENT：

该事件在Host启动时触发完成服务器启动过程中的Web应用部署（只有当Host的deployOnStartup属性为true的时候，服务器才会在启动的时候部署Web应用。默认为true）

这个过程包含了三个部分：

* Context描述文件部署
* Web目录部署
* War包部署

**1.1 通过Context描述文件部署**

Tomcat支持通过一个独立的Context描述文件来配置并且启动Web应用。（有待试验）

**1.2 通过Web应用的目录部署**

做法：

把Web应用编译后的目录（WEB-INF文件夹的上一层）复制到webapps下面，并修改为ROOT，启动tomcat。

访问路径：<http://localhost:8080>

**过程原理：**

对于Host的appBase目录（默认为$CATALINA\_BASE/webapps）下所有符合条件的目录，由线程池来进行部署。

1. 根据Host的contextClass属性指定的类型创建Context对象。如果没有指定，则默认使用StandardContext。此时，所有的Context属性均采用默认的配置，而name，path，webappVersion、docBase会根据Tomcat的webapps目录下面的Web应用目录的路径及名称进行设置。
2. 为Context实例添加ContextConfig生命周期监听器
3. 将Context实例添加到Host实例中，并且启动
4. Web应用目录及web.xml等添加到守护资源以便文件发生变更时重新部署或者加载Web应用。

结论：Context的来源是webapps下面，要部署的Web应用中包含的信息（如：web.xml）

**1.3 通过War包部署**

通过War包部署与Web应用目录部署基本一致

**过程原理：**

对于Host的appBase目录（默认为$CATALINA\_BASE/webapps）下所有符合条件的war包，由线程池来进行部署。

1. 根据Host的contextClass属性指定的类型创建Context对象。如果没有指定，则默认使用StandardContext。此时，所有的Context属性均采用默认的配置，而name，path，webappVersion、docBase会根据Tomcat的webapps目录下面的war包的路径及名称进行设置。
2. 为Context实例添加ContextConfig生命周期监听器
3. 将Context实例添加到Host实例中，并且启动
4. Web应用目录及web.xml等添加到守护资源以便文件发生变更时重新部署或者加载Web应用。

#### 2、PERIODIC\_EVENT

Catalina容器支持定期执行自身及子容器的后台处理过程，具体过程在容器中的backgroundProcess（）中定义。该机制常用于定时扫描Web应用的变更，并进行重新加载。后台任务处理完成后，会触发PERIODIC\_EVENT事件。

HostConfig中通过DeployedApplication维护两个守护资源列表：

* redeployResources：用于守护导致应用重新部署的资源（重新创建了这个Context对象，如：context描述文件改变，需要重新部署）
* reloadResources：用于守护导致应用重新加载的资源（针对同一个Context对象进行重启，如：Web应用中web.xml改变，需要重新加载）

过程原理：

1. 对于每一个已经部署的Web应用，如果资源发生变更，则重新加载Context对象
2. 部署Web应用

**实践：**

1. **在server.xml配置Context标签**

|  |
| --- |
| <Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true">  <Context docBase="C:\App\project\open-platform\open-platform\open-report\open-report-business\open-report-web\target\open-report-web"  path="myReport" reloadable="true"></Context>  </Host> |

docBase：open-report编译后的根目录（WEB-INF文件夹的上一层）

path：在浏览器中进行访问的路径名称

reloadable：暂时没有实验

访问路径：http://localhost:8080/myReport/

1. **通过Web应用的目录部署**

把open-capital编译后的目录（WEB-INF文件夹的上一层）复制到webapps下面，并修改为ROOT，启动tomcat。

访问路径：<http://localhost:8080>

1. **通过WAR包进行部署**

把open-risk-web-1.0-SNAPSHOT.war 直接复制到webapps下面，启动tocmat。

访问路径：http://localhost:8080/open-risk-web-1.0-SNAPSHOT/

**结论：**

这三种方式可以同时使用，互相独立。

证明理论：这三个包在tomcat启动时都会创建Context，并且加载到当前的惟一的Host中。

此次测试的是三个项目，分支点在Context上。三个Context共享一个Host，也就是同一个域名。

### 3.2.3 StandardContext

对于StandardHost和HostConfig来说，完成的仅仅是根据不同的情况（部署描述文件、部署目录、部署War包）创建并启动Context对象。并不包含*具体的Web应用初始化以及启动工作。该部分工作由StandardContext来完成*。

**启动过程原理：**

1. 初始化当前Context使用的WebResourceRoot并启动（WebResourceRoot维护了Web应用中所有的资源集合，包括Class文件，Jar包以及其他资源文件，找客户要用于**类的加载**和**按照路径查找资源**）。
2. 创建**WebappLoader** 这个Web应用类加载器。
3. 如果没有设置Cookie管理器，则默认创建Rfc6265CookieProcessor
4. 设置字符集映射（CharsetMapper），用于根据Locale获取字符集编码
5. 初始化临时目录（默认为$CATALINA\_BASE/work/${Engine名称}/${Host名称}/${Context名称}）
6. 启动WebappLoader（WebappLoader.start），创建WebappClassLoader这个Web应用类加载器。
7. 发布CONFIGURATION\_START\_EVENT事件，**ContextConfig监听该事件，来完成Servlet的创建**
8. 启动Context的子节点（Wrapper）
9. 启动Context维护的Pipeline
10. 将Context的Web资源集合（也就是WebResourceRoot）设置为ServletContext的属性
11. 创建实例管理器（InstanceManager），用于创建Servlet、Filter对象
12. 将Jar包扫描器（JarScanner）添加到ServletContextx属性
13. 合并ServletContext的初始化参数和Context组件中的ApplicationParameter（在context.xml描述文件中配置）
14. 启动当前Context的ServletContainerInitializer，该类主要用于添加Web应用的设置，如Servlet、Filter
15. 实例化监听器
    1. 事件监听器
       1. ServletContextAttributeListener
       2. ServletRequestAttributeListener
       3. ServletRequestListener
       4. HttpSessionIdListener
       5. HttpSessionAttributeListener
    2. 生命周期监听器
       1. HttpSessionListener
       2. ServletContextListener
16. 实例化FilterConfig，、Filter，并调用Filter.init方法
17. 对于loadOnStartup >= 0 的Wrapper，调用Wrapper.load，该方法负责实例化Servlet，并调用Servlet.init
18. 设置Context状态为STARTING

上述过程仅包含StandardContext的启动过程，并不包含如何解析Web.xml中的Servlet、请求映射、Filter等相关配置。

### ContextConfig

Context创建时（在HostConfig中创建的）会添加一个监听器ContextConfig，一共监听6个状态，此处只说比较核心的三个：AFTER\_INIT\_EVENT、BEFORE\_START\_EVENT、CONFIGURE\_START\_EVENT。

ContextConfig工作内容：

* Web应用中Context的属性配置
* 解析web.xml，完成web容器初始化

1. **AFTER\_INIT\_EVENT**

该事件属于Context的初始化阶段，**主要用于Context的属性配置工作**。

Context的来源有以下几个：

* 在实例化Server时，解析server.xml文件中的Context元素创建。
* 在HostConfig部署Web应用时，解析Web应用（Web目录或者War包）根目录下的META-INF/context.xml文件创建。如果不存在该文件，则自动创建一个Context实例，设置默认参数。
* 在Host部署Web应用时，解析$CATALINA\_BASE/conf/${Engine名称}/${Host名称}下的Context部署文件创建。

除了Context创建时的属性配置，Tomcat还提供了针对Context的默认配置

* Catalina级默认配置：如果存在 conf/context.xml，那么解析该文件，将文件中的属性配置更新到Context实例中
* Host级默认配置：如果存在conf/${Engin名称}/${Host名称}/context.xml.default文件，那么解析该文件，将文件中的属性配置更新到Context实例中
* Web级配置：configFile（META-INF/context.xml），解析该文件，并将文件属性更新到Context实例中。覆盖之前的属性值。

通过以上描述，总结出Context实例配置内容的优先级：Web级>Host级>Catalina级。

1. **BEFORE\_START\_EVENT**

在Context启动之前触发，用于**更新Context的docBase和解决Web目录锁的问题**

更新Context的docBase属性针对的是部署War包的情况。当Web应用为一个部署War包且需要解压部署时，***docBase属性指向的是解压后的文件夹目录***。

1. **CONFIGURE\_START\_EVENT**

Context在***启动子节点之前***，触发了CONFIGURE\_START\_EVENT事件.Context正是通过该事件**解析web.xml**，创建Wrapper（Servlet）、Filter、ServletContextListener等，**完成Web容器的初始化的**。

ContextConfig在处理CONFIGURE\_START\_EVENT事件所做的主要工作：

* 1. Web容器初始化：
     1. Web应用部署描述来源：
        1. Servlet规范中提到的描述：
           1. WEB-INF/web.xml
           2. Web应用的Jar包中的META-INF/web-fragment.xml（web-fragment.xml可以看做是wen.xml的片段。通过将其置于Jar包的META-INF下，可以将web应用的配置拆解到各个模块中，有利于Web应用的可插拔和模块化）
           3. Web应用的Jar包中的 META下的services/javax.servlet.ServletContainerInitializer。
        2. Tomcat的默认配置
           1. Catalina级别：conf/web.xml
           2. Host级别：conf/${Engine名称}/${Host名称}/web.xml.default
     2. Tomcat初始化Web应用容器过程：
        1. 首先解析Tomcat的默认配置，生成WebXml对象。先解析Catalina级的，再解析Host级的。此时它是“默认WebXml对象”。
        2. 解析Web应用中的web.xml（WEB-INF/web.xml），创建一个WebXml对象，此时它是“主WebXml对象”。
        3. 扫描所有Jar中的web-fragment.xml，解析并创建多个WebXml对象，此时它们是“片段WebXml对象”。对它们按照Servlet规范进行排序，将排序的结果保存到ServletContext中，这个顺序决定了Filter等的执行顺序。
        4. 加载WEB-INF/lib下面所有的Jar包
        5. 将“默认WebXml”和“片段WebXml”合并到“主WebXml”中。
        6. 配置JspServlet
        7. 使用“主WebXml”来配置当前ServletContext，包括Servlet、Filter、Listener等。并交给StandardContext。对于Servlet，则创建StandardWrapper，并添加到StandardContext中。
        8. 查找Jar包“META-INF/resources”下的静态资源，添加到StandardContext中。

Tomcat初始化Web容器的过程，就是StandardContext正是启动StandardWrapper之前的准备。

### 3.2.5 StardardWrapper

StandardWrapper 维护了Servlet实例，StandardContext启动过程中，StandardWrapper的处理过程如下：

1. 当ContextConfig完成Web容器的初始化时，会调用StandardWrapper.start方法，StandardWrapper的状态会变成STARTED。
2. 对于启动时加载Servlet（loadOnStartup >= 0），调用StandardWrapper.load方法。
   1. 创建Servlet实例，如果添加JNDI资源注解，将进行依赖注入。
   2. 读取MultipartConfig注解配置，以用于multipart/form-data请求处理。
   3. 读取ServletSecurity配置，添加Servlet安全。
   4. 调用Servlet.init进行初始化。

## 3.3 Web请求处理

### 3.3.1 总体过程

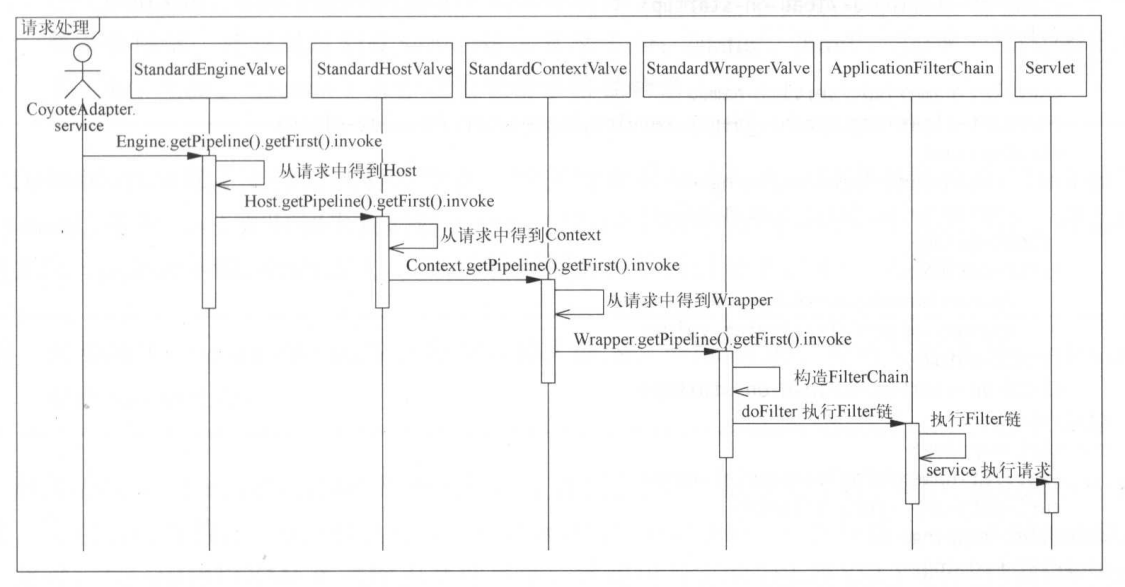
1. CoyoteAdapter将Connector与Mapper、Engine联系起来。当Connector接收到请求后，首先读取请求数据，然后调用CoyoteAdapter.service完成请求处理。
2. 根据Connector的请求对象（org.apache.coyote.Request）和响应对象（org.apache.coyote..Response），创建Servlet的请求对象（org.apache.catalina.connector.Request）和响应对象（org.apache.catalina.connector.Response）。
3. 转换请求参数并完成映射
4. 得到Engine的第一个Valve，并执行，已完成客户端请求处理

### Catalina中处理请求的过程

Tomcat采用责任链的形式来处理客户端需求：

Pipeline：构造职责链

Valve：职责链上的处理器。它可以拦截请求传递过程中的请求，然后做需要做的处理。Tomcat的每一级容器都提供了基础的Valve实现以完成当前容器的请求处理过程。



只需要获取Engine的Pipeline，然后获取到第一个Valve，就可以完成整个请求处理。

**Pipeline与Valve的作用是为了进行请求处理的。**

# 四、Coyote

## 4.1 Coyote简介

Coyote是Tomcat中的连接器框架，是Tomcat提供给客户端访问的接口。，客户端通过Coyote与服务端建立连接、发送请求并且接受相应。

在Coyote中，请求的状态：

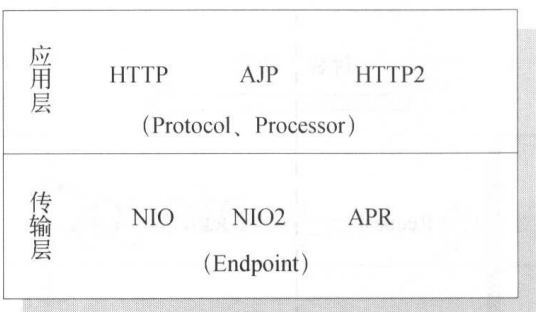
流 -> Socket -> SocketWrapper -> Request -> 解析处理 -> Response 输入流

Tomcat支持三种传输协议：

* HTTP1.1
* AJP
* HTTP2.0

Tomcat支持三种I/O传输方案

* NIO
* NIO2
* APR



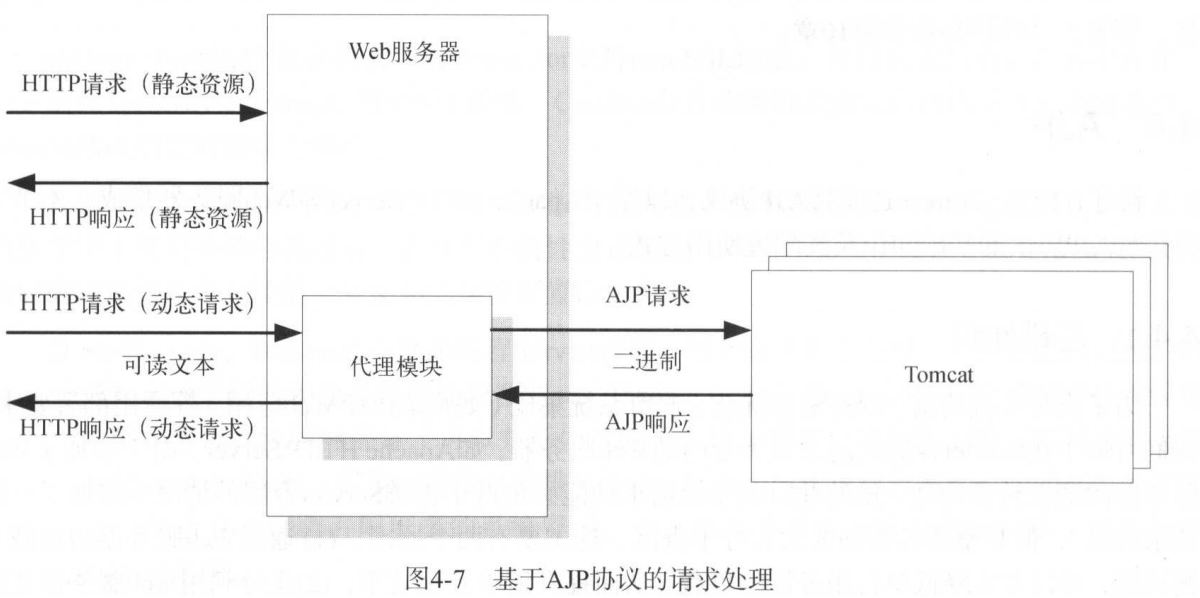
## 4.2 Web处理请求

### 4.2.1 Connector中的核心概念

* Endpoint：Coyote的通信监听接口。是具体的Socket接受类。*是对传输层的抽象。*
* Processor：Coyote的协议处理接口。负责构造Request和Response对象。是对应用层的抽象。
* ProtocolHandler：Coyote协议接口。把Endpoint和Processor都封装其中，共同完成请求的具体协议处理过程。通过将协议类型和传输类型进行组合，形成不同的实现类：
  + Http11NioProtocol
  + Http11AprProtocol
  + Http11Nio2Protocol
  + Ajp11NioProtocol
  + Ajp11AprProtocol
  + Ajp11Nio2Protocol

## 4.3 AJP

AJP是一种采用二进制格式进行传输的协议。



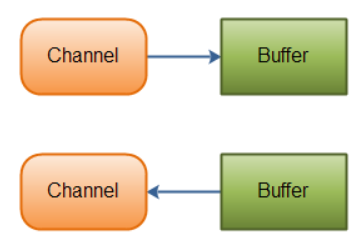
AJP是在动态请求下，Web服务器与Tomcat之间进行交流的协议。

## I/O

### Channel

NIO与流有所不同，它的特点是：

* 既可以从通道中读取数据，又可以把数据写到相同的通道中去；但是流是单向的
* 通道可以异步的读写
* 只能通过Buffer来间接操作Channel的读写



Channel的实现：

* FileChannel：针对文件
* DatagramChannel：针对UDP
* SocketChannel：通过TCP读写网络数据
* ServerSocketChannel：可以监听新进来的TCP链接，像Web服务器那样，对每一个新进来的链接都会创建一个SocketChannel

### 4.4.2 Buffer

Java NIO中的Buffer用于和NIO通道进行交互。如你所知，数据是从通道读入缓冲区，从缓冲区写入到通道中的。

#### 4.4.2.1 Buffer用法

Buffer的基本用法：

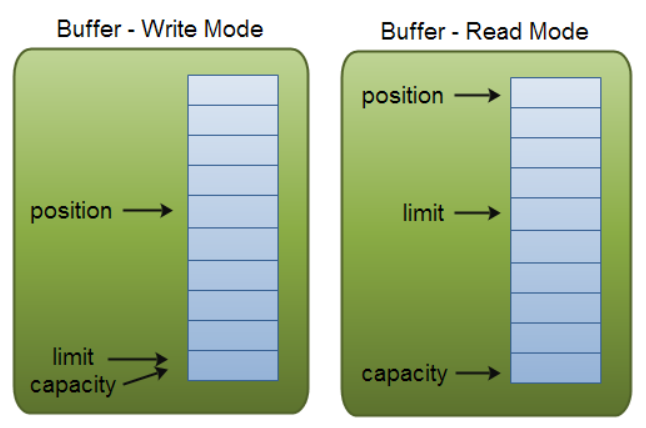
1. 从Channel中把数据写入到Buffer中
2. 调用filp方法，将Buffer从写模式切换到读模式。调用flip()方法会将position设回0，并将limit设置成之前position的值。
3. 从Buffer中读取数据，可以打印，可以写到另一个Channel中等
4. 调用clear方法清空Buffer，重复1过程

当向buffer写入数据时，buffer会记录下写了多少数据。一旦要读取数据，需要通过flip()方法将Buffer从写模式切换到读模式。在读模式下，可以读取之前写入到buffer的所有数据。

一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用clear()或compact()方法。clear()方法会清空整个缓冲区。compact()方法只会清除已经读过的数据。任何未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。

#### 4.4.2.2 Buffer的结构

Buffer本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成NIO Buffer对象，并且提供了一组方法，用来方便管理和操作这块内存。



* capacity：Buffer的固定值大小，在初始化Buffer的时候进行指定
* position：当前位置指针，当向Buffer中写入数据的时候，position会向前移动相应长度。
* limit：
  + 写模式：表示你最多能往Buffer中写多少数据，在这里limit = capacity
  + 读模式：表示最多能从Buffer中读取多少数据，在这里limit = position，因为从0到position这段是有值的，是可读的；position到capacity之间没有数据，没什么可读的。

#### 4.4.2.3 Buffer的类型

ByteBuffer

MappedByteBuffer

CharBuffer

DoubleBuffer

FloatBuffer

IntBuffer

LongBuffer

ShortBuffer

#### Buffer 的方法

* filp：
  + flip方法将Buffer从写模式切换到读模式。调用flip()方法会将position设回0，并将limit设置成之前position的值。
* clear：
  + position会重置为0，limit会重置为capacity。Buffer的所有属性重新归位了，但数据还在，如果在读之前调用clear方法，则Buffer中的数据会被覆盖。
* compact：
  + position会放到未读数据的后面，limit重置为capacity，这样会续写，不会覆盖上一次的写入Buffer中的数据。

Ps：对于Buffer来说，从buffer中读取数据叫“读过程”；把数据写入Buffer中叫“写过程”。*先写后读*。

### 4.4.3 Channel与Buffer使用的综合案例

|  |
| --- |
| String sourceFile = "F:\\mmnd128.avi"; String targetFile = "G:\\mmnd128.avi";  FileInputStream fileInputStream = new FileInputStream(sourceFile); FileOutputStream fileOutputStream = new FileOutputStream(targetFile);  FileChannel InputChannel = fileInputStream.getChannel(); FileChannel outputChannel = fileOutputStream.getChannel();  ByteBuffer buffer = ByteBuffer.allocate(1024); while (true) {  buffer.clear();  int read = InputChannel.read(buffer);  if (read == -1) {  break;  }  buffer.flip();  outputChannel.write(buffer); } |

### Selector

Selector可以在一条线程内同时管理多个channel，这样节省了多线程内存的开销。

#### 向Selector注册通道

为了将Channel和Selector配合使用，*必须将channel注册到selector上*。通过SelectableChannel.register()方法来实现。

|  |
| --- |
| channel.configureBlocking(**false**);  SelectionKey key = channel.register(selector, Selectionkey.OP\_READ); |

与Selector一起使用时，Channel必须处于非阻塞模式下。这意味着不能将FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式。而套接字通道都可以。

#### Selector的四种监听

register()方法的第二个参数是一个“interest集合”，***意思是在通过Selector监听Channel时对什么事件感兴趣***。可以监听四种不同类型的事件：

* Connect
* Accept
* Read
* Write

#### SelectionKey

*通道触发了一个事件意思是该事件已经就绪*。所以，某个channel成功连接到另一个服务器称为“连接就绪”。一个server socket channel准备好接收新进入的连接称为“接收就绪”。一个有数据可读的通道可以说是“读就绪”。等待写数据的通道可以说是“写就绪”。

这四种事件用SelectionKey的四个常量来表示：

* SelectionKey.OP\_CONNECT
* SelectionKey.OP\_ACCEPT
* SelectionKey.OP\_READ
* SelectionKey.OP\_WRITE

可以使用以下四个方法，来检测channel中什么事件或操作已经就绪：

|  |
| --- |
| selectionKey.isAcceptable();  selectionKey.isConnectable();  selectionKey.isReadable();  selectionKey.isWritable(); |

从SelectionKey访问Channel和Selector很简单。如下：

|  |
| --- |
| Channel channel = selectionKey.channel();  Selector selector = selectionKey.selector(); |

#### 通过Selector选择通道

一旦想Selector注册了一个或者多个channel，就可以调用select方法（或者select的重载方法）。select方法是阻塞方法，如果Selector中注册过的任何一个channel已经就绪了某个状态（四种状态中的一个），则会发生阻塞。然后可以通过调用selector的selectedKeys()方法，访问“已选择键集（selected key set）”中的就绪通道。

|  |
| --- |
| Set selectedKeys = selector.selectedKeys(); |

可以遍历这个已选择的键集合来访问就绪的通道。如下：

|  |
| --- |
| Set selectedKeys = selector.selectedKeys();  Iterator keyIterator = selectedKeys.iterator();  while(keyIterator.hasNext()) {  SelectionKey key = keyIterator.next();  if(key.isAcceptable()) {  // a connection was accepted by a ServerSocketChannel.  } else if (key.isConnectable()) {  // a connection was established with a remote server.  } else if (key.isReadable()) {  // a channel is ready for reading  } else if (key.isWritable()) {  // a channel is ready for writing  }  keyIterator.remove();  } |

# 五、Tomcat配置管理

## 5.1 JVM配置

# 六、Web服务器集成

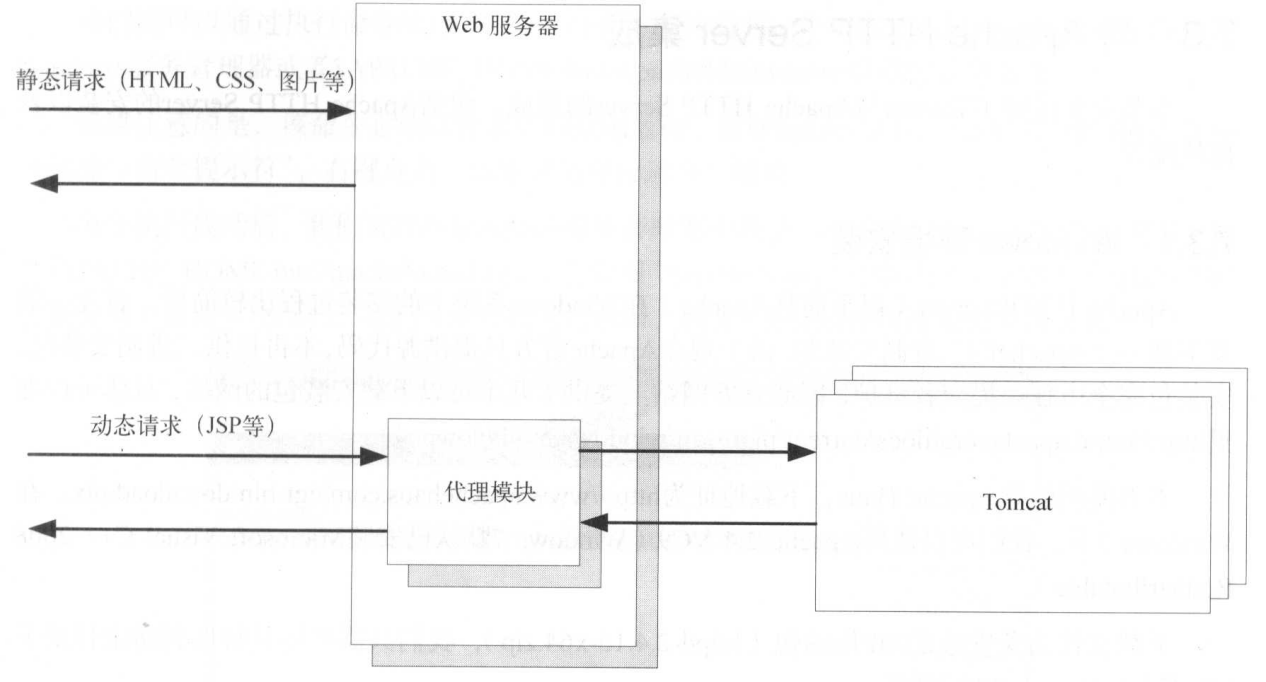
## 6.1 Web服务器与应用服务器

* Web服务器：
  + Web服务器是一个处理HTTP请求的计算机系统，主要功能是存储、处理、传送Web页面到达客户端。
  + Web服务器更侧重于系统的吞吐量，并发量的支持。
  + Web服务器提供了反向代理，用于做负载均衡。
  + 提供了静态文件缓存服务，因此对于静态文件的请求性能要好于应用服务器。
* 应用服务器：
  + 应用服务器的功能致力于过程（程序、脚本）的处理效率以支撑其应用。

两者的长处不同，所以最佳实践场景不同。*Web服务器侧重于HTTP请求的处理；而应用服务器侧重于构建业务系统组件的支撑，以简化复杂系统的构建工作*。

Web服务器技术实际上是应用服务器的一个子集，它仅仅提供了基本的面向文档的处理，并不支持复杂的业务过程处理。但是应用服务器并不完全包含Web服务器所涉及到的各种技术。

所以，我们在部署架构中可以将Web服务器置于应用服务器的前端，用于处理来自客户的**静态请求**（如js脚本，图片等）；而应用服务器则专注于处理业务逻辑，也就是**动态请求**。以此来提高系统的整体性能。



## 6.2 集成Nginx

Nginx集中解决Web服务器高性能、高并发及低内存消耗的问题，同时提供了负载均衡、缓存、访问控制、带宽控制及高效的应用整合能力。

### 6.2.1 高并发

造成服务端积累大量并发链接的场景来自于两点：

* 客户端：由于移动终端和新的应用架构导致，通常这些应用基于持久链接提供实时新闻、微博消息等。
* 持久链接：现代浏览器为了加快页面加载速度，会同时发送多个请求。

### 6.2.2 反向代理

正向代理：

我们如果由于技术需要要去访问国外的某些网站，此时你会发现位于国外的某网站我们通过浏览器是没有办法访问的，此时大家可能都会用一个操作FQ进行访问，FQ的方式主要是找到一个可以访问国外网站的代理服务器，我们将请求发送给代理服务器，代理服务器去访问国外的网站，然后将访问到的数据传递给我们！

上述这样的代理模式称为正向代理，正向代理最大的特点是客户端非常明确要访问的服务器地址；服务器只清楚请求来自哪个代理服务器，而不清楚来自哪个具体的客户端；正向代理模式屏蔽或者隐藏了真实客户端信息。

反向代理：

多个客户端给服务器发送的请求，nginx服务器接收到之后，按照一定的规则分发给了后端的业务处理服务器进行处理了。此时~请求的来源也就是客户端是明确的，但是请求具体由哪台服务器处理的并不明确了，nginx扮演的就是一个反向代理角色。反向代理，主要用于服务器集群分布式部署的情况下，反向代理隐藏了服务器的信息

### 6.2.3 负载均衡

我们已经明确了所谓代理服务器的概念，那么接下来，nginx扮演了反向代理服务器的角色，它是以依据什么样的规则进行请求分发的呢？不用的项目应用场景，分发的规则是否可以控制呢？这里提到的客户端发送的、nginx反向代理服务器接收到的请求数量，就是我们说的负载量请求数量按照一定的规则进行分发到不同的服务器处理的规则，就是一种均衡规则。所以~将服务器接收到的请求按照规则分发的过程，称为负载均衡。

nginx支持的负载均衡调度算法方式如下：

1. weight轮询（默认）：

接收到的请求按照顺序逐一分配到不同的后端服务器，即使在使用过程中，某一台后端服务器宕机，nginx会自动将该服务器剔除出队列，请求受理情况不会受到任何影响。 这种方式下，可以给不同的后端服务器设置一个权重值（weight），用于调整不同的服务器上请求的分配率；权重数据越大，被分配到请求的几率越大；该权重值，主要是针对实际工作环境中不同的后端服务器硬件配置进行调整的。

1. ip\_hash：

每个请求按照发起客户端的ip的hash结果进行匹配，这样的算法下一个固定ip地址的客户端总会访问到同一个后端服务器，这也在一定程度上解决了集群部署环境下session共享的问题。

1. fair：

智能调整调度算法，动态的根据后端服务器的请求处理到响应的时间进行均衡分配，响应时间短处理效率高的服务器分配到请求的概率高，响应时间长处理效率低的服务器分配到的请求少；结合了前两者的优点的一种调度算法。但是需要注意的是nginx默认不支持fair算法，如果要使用这种调度算法，请安装upstream\_fair模块

1. url\_hash：

按照访问的url的hash结果分配请求，每个请求的url会指向后端固定的某个服务器，可以在nginx作为静态服务器的情况下提高缓存效率。同样要注意nginx默认不支持这种调度算法，要使用的话需要安装nginx的hash软件包

### 6.2.4 nginx配置

如上述配置文件所示，主要由6个部分组成：

* main：用于进行nginx全局信息的配置
* events：用于nginx工作模式的配置
* http：用于进行http协议信息的一些配置
* server：用于进行服务器访问信息的配置
* location：用于进行访问路由的配置
* upstream：用于进行负载均衡的配置

#### main模块

|  |
| --- |
| # user nobody nobody;  worker\_processes 2;  # error\_log logs/error.log  # error\_log logs/error.log notice  # error\_log logs/error.log info  # pid logs/nginx.pid  worker\_rlimit\_nofile 1024; |

* **user**用来指定nginx worker进程运行*用户以及用户组*，默认nobody账号运行
* **worker\_processes**指定nginx要开启的*子进程数量*，运行过程中监控每个进程消耗内存(一般几M~几十M不等)根据实际情况进行调整，通常数量是*CPU内核数量的整数倍*
* **error\_log**定义*错误日志文件的位置及输出级别*【debug / info / notice / warn / error / crit】
* **pid**用来指定*进程id的存储文件的位置*
* **worker\_rlimit\_nofile**用于指定*一个进程可以打开最多文件数量*的描述

#### event 模块

|  |
| --- |
| event {  worker\_connections 1024;  multi\_accept on;  use epoll;  } |

上述配置是针对nginx服务器的工作模式的一些操作配置

* **worker\_connections** 指定*最大可以同时接收的连接数量*，这里一定要注意，*最大连接数量是和worker processes共同决定的*。
* **multi\_accept** 配置指定nginx在收到一个新连接通知后尽可能多的接受更多的连接（什么意思）
* **use epoll** 配置指定了*线程轮询的方法*，如果是linux2.6+，使用epoll，如果是BSD如Mac请使用Kqueue（线程轮询的方法是什么）

#### http模块

作为web服务器，http模块是nginx最核心的一个模块，配置项也是比较多的，项目中会设置到很多的实际业务场景，需要根据硬件信息进行适当的配置，常规情况下，使用默认配置即可

|  |
| --- |
| http {  ##  # 基础配置  ##  sendfile on;  tcp\_nopush on;  tcp\_nodelay on;  keepalive\_timeout 65;  types\_hash\_max\_size 2048;  # server\_tokens off;  # server\_names\_hash\_bucket\_size 64;  # server\_name\_in\_redirect off;  include /etc/nginx/mime.types;  default\_type application/octet-stream;  ##  # SSL证书配置  ##  ssl\_protocols TLSv1 TLSv1.1 TLSv1.2; # Dropping SSLv3, ref: POODLE  ssl\_prefer\_server\_ciphers on;  ##  # 日志配置  ##  access\_log /var/log/nginx/access.log;  error\_log /var/log/nginx/error.log;  ##  # Gzip 压缩配置  ##  gzip on;  gzip\_disable "msie6";  # gzip\_vary on;  # gzip\_proxied any;  # gzip\_comp\_level 6;  # gzip\_buffers 16 8k;  # gzip\_http\_version 1.1;  # gzip\_types text/plain text/css application/json application/javascript  text/xml application/xml application/xml+rss text/javascript;  ##  # 虚拟主机配置  ##  include /etc/nginx/conf.d/\*.conf;  include /etc/nginx/sites-enabled/\*; |

1. 基础配置（基本没看懂）

* **sendfile on**：配置on让sendfile发挥作用，将文件的回写过程交给数据缓冲去去完成，而不是放在应用中完成，这样的话在性能提升有有好处
* **tc\_nopush on**：让nginx在一个数据包中发送所有的头文件，而不是一个一个单独发
* **tcp\_nodelay on**：让nginx不要缓存数据，而是一段一段发送，如果数据的传输有实时性的要求的话可以配置它，发送完一小段数据就立刻能得到返回值，但是不要滥用哦
* **keepalive\_timeout 10**：给客户端分配连接超时时间，服务器会在这个时间过后关闭连接。一般设置时间较短，可以让nginx工作持续性更好
* **client\_header\_timeout 10**：设置请求头的超时时间
* **client\_body\_timeout 10**:设置请求体的超时时间
* **send\_timeout 10**：指定客户端响应超时时间，如果客户端两次操作间隔超过这个时间，服务器就会关闭这个链接
* **limit\_conn\_zone $binary\_remote\_addr zone=addr:5m** ：设置用于保存各种key的共享内存的参数，
* **limit\_conn addr 100**: 给定的key设置最大连接数
* **server\_tokens**：虽然不会让nginx执行速度更快，但是可以在错误页面关闭nginx版本提示，对于网站安全性的提升有好处哦
* **include /etc/nginx/mime.types**：指定在当前文件中包含另一个文件的指令
* **default\_type application/octet-stream**：指定默认处理的文件类型可以是二进制
* **type\_hash\_max\_size 2048**：混淆数据，影响三列冲突率，值越大消耗内存越多，散列key冲突率会降低，检索速度更快；值越小key，占用内存较少，冲突率越高，检索速度变慢

1. 日志配置

* **access\_log logs/access.log**：设置存储*访问记录的日志*
* **error\_log logs/error.log**：设置存储记*录错误发生的日志*

1. SSL证书加密（SSL证书是啥呀）

* **ssl\_protocols**：指令用于启动特定的加密协议，nginx在1.1.13和1.0.12版本后默认是ssl\_protocols SSLv3 TLSv1 TLSv1.1 TLSv1.2，TLSv1.1与TLSv1.2要确保OpenSSL >= 1.0.1 ，SSLv3 现在还有很多地方在用但有不少被攻击的漏洞。
* **ssl prefer server ciphers**：设置协商加密算法时，优先使用我们服务端的加密套件，而不是客户端浏览器的加密套件

1. 压缩配置

* **gzip** 是告诉nginx*采用gzip压缩的形式发送数据（发送给谁？减少了哪一块流程的数据量？）*。这将会减少我们发送的数据量。
* **gzip\_disable** 为指定的客户端禁用gzip功能。我们设置成IE6或者更低版本以使我们的方案能够广泛兼容。
* **gzip\_static** 告诉nginx在压缩资源之前，先查找是否有预先gzip处理过的资源。这要求你预先压缩你的文件（在这个例子中被注释掉了），从而允许你使用最高压缩比，这样nginx就不用再压缩这些文件了（想要更详尽的gzip\_static的信息，请点击这里）。
* **gzip\_proxied** 允许或者禁止压缩基于请求和响应的响应流。我们设置为any，意味着将会压缩所有的请求。
* **gzip\_min\_length** *设置对数据启用压缩的最少字节数*。如果一个请求小于1000字节，我们最好不要压缩它，因为压缩这些小的数据会降低处理此请求的所有进程的速度。
* **gzip\_comp\_level** *设置数据的压缩等级*。这个等级可以是1-9之间的任意数值，9是最慢但是压缩比最大的。我们设置为4，这是一个比较折中的设置。
* **gzip\_type** *设置需要压缩的数据格式*。上面例子中已经有一些了，你也可以再添加更多的格式。

1. 文件缓存配置

* **open\_file\_cache** 打开缓存的同时也指定了缓存最大数目，以及缓存的时间。我们可以设置一个相对高的最大时间，这样我们可以在它们不活动超过20秒后清除掉。
* **open\_file\_cache\_valid** 在open\_file\_cache中指定检测正确信息的间隔时间。
* **open\_file\_cache\_min\_uses** 定义了open\_file\_cache中指令参数*不活动时间期间里最小的文件数*。
* **open\_file\_cache\_errors** 指定了当搜索一个文件时是否缓存错误信息，也包括再次给配置中添加文件。我们也包括了服务器模块，这些是在不同文件中定义的。如果你的服务器模块不在这些位置，你就得修改这一行来指定正确的位置。

#### server模块

srever模块配置是http模块中的一个子模块，用来定义一个虚拟访问主机，也就是一个虚拟服务器的配置信息

|  |
| --- |
| server {  listen 80;  server\_name localhost 192.168.1.100;  root /nginx/www;  index index.php index.html index.html;  charset utf-8;  access\_log logs/access.log;  error\_log logs/error.log;  ......  } |

* **server**：一个虚拟主机的配置，一个http中可以配置多个server
* **server\_name**：用来指定ip地址或者域名，多个配置之间用空格分隔
* **root**：表示整个server虚拟主机内的根目录，所有当前主机中web项目的根目录（啥意思）
* **index**：用户访问web网站时的全局首页
* **charset**：用于设置www/路径中配置的网页的默认编码格式
* **access\_log**：用于指定该虚拟主机服务器中的访问记录日志存放路径
* **error\_log**：用于指定该虚拟主机服务器中访问错误日志的存放路径

#### location模块

location模块是nginx配置中出现最多的一个配置，主要用于配置路由访问信息在路由访问信息配置中关联到反向代理、负载均衡等等各项功能，所以location模块也是一个非常重要的配置模块。

1、基本配置

|  |
| --- |
| location / {  root /nginx/www;  index index.php index.html index.htm;  } |

* **location /**：表示匹配访问根目录
* **root**：用于指定访问根目录时，访问虚拟主机的web目录
* **index**：在不指定访问具体资源时，默认展示的资源文件列表

2、反向代理配置方式

通过反向代理代理服务器访问模式，通过proxy\_set配置让客户端访问透明化

|  |
| --- |
| location / {  proxy\_pass http://localhost:8888;  proxy\_set\_header X-real-ip $remote\_addr;  proxy\_set\_header Host $http\_host;  } |

（这么配置啥意思啊？）

#### upstream模块

upstream模块主要负责负载均衡的配置，通过默认的轮询调度方式来分发请求到后端服务器

|  |
| --- |
| upstream name {  ip\_hash;  server 192.168.1.100:8000;  server 192.168.1.100:8001 down;  server 192.168.1.100:8002 max\_fails=3;  server 192.168.1.100:8003 fail\_timeout=20s;  server 192.168.1.100:8004 max\_fails=3 fail\_timeout=20s;  } |

* **ip\_hash**：指定请求调度算法，默认是weight权重轮询调度，可以指定
* **server host:port**：分发服务器的列表配置
  + **down**：表示该主机暂停服务
  + **max\_fails**：表示失败最大次数，超过失败最大次数暂停服务
  + **fail\_timeout**：表示如果请求受理失败，暂停指定的时间之后重新发起请求

### 6.2.5 虚拟主机

例如将[www.aaa.com](http://www.aaa.com/" \t "_blank)和[www.bbb.com](http://www.bbb.com/" \t "_blank)两个网站部署在同一台服务器上，两个域名解析到同一个IP地址，但是用户通过两个域名却可以打开两个完全不同的网站，互相不影响，就像访问两个服务器一样，所以叫两个虚拟主机。

|  |
| --- |
| server {  listen 80 default\_server;  server\_name \_;  return 444; # 过滤其他域名的请求，返回444状态码  }  server {  listen 80;  server\_name www.aaa.com; # www.aaa.com域名  location / {  proxy\_pass http://localhost:8080; # 对应端口号8080  }  }  server {  listen 80;  server\_name www.bbb.com; # www.bbb.com域名  location / {  proxy\_pass http://localhost:8081; # 对应端口号8081  }  } |

在服务器8080和8081分别开了一个应用，客户端通过不同的域名访问，根据server\_name可以反向代理到对应的应用服务器。