2019 版

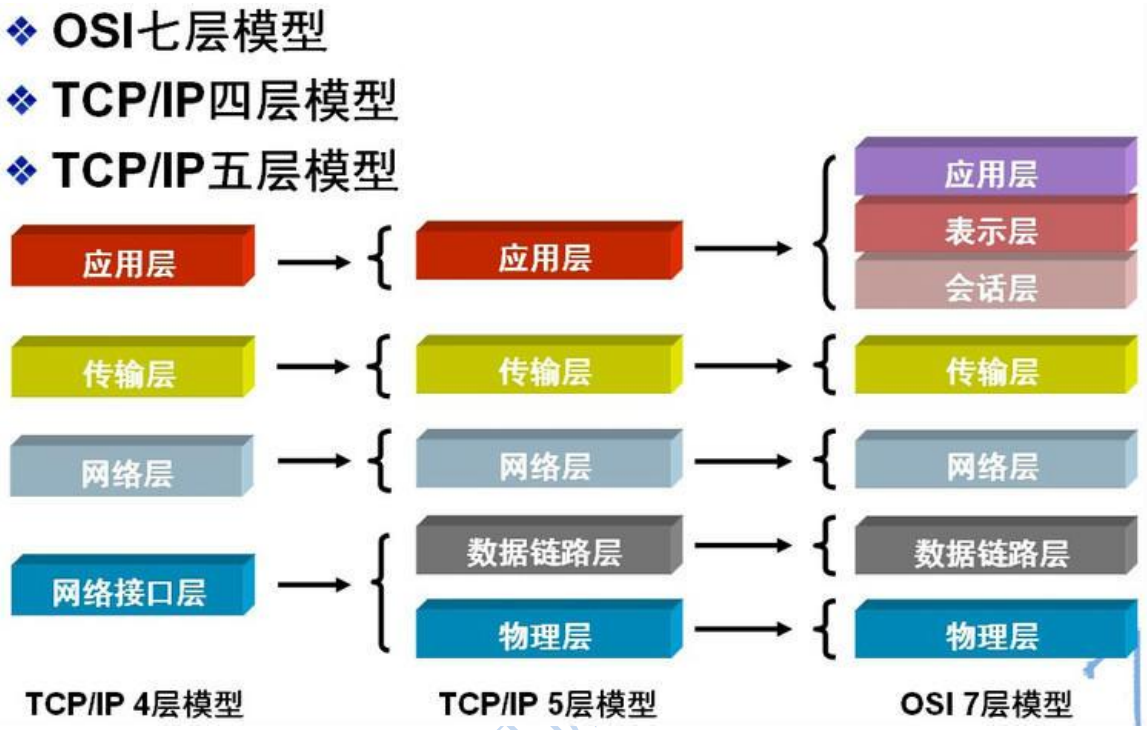
治愈系 Java 工程师面试指导课程

[Part2・网络基础和 JavaWeb]

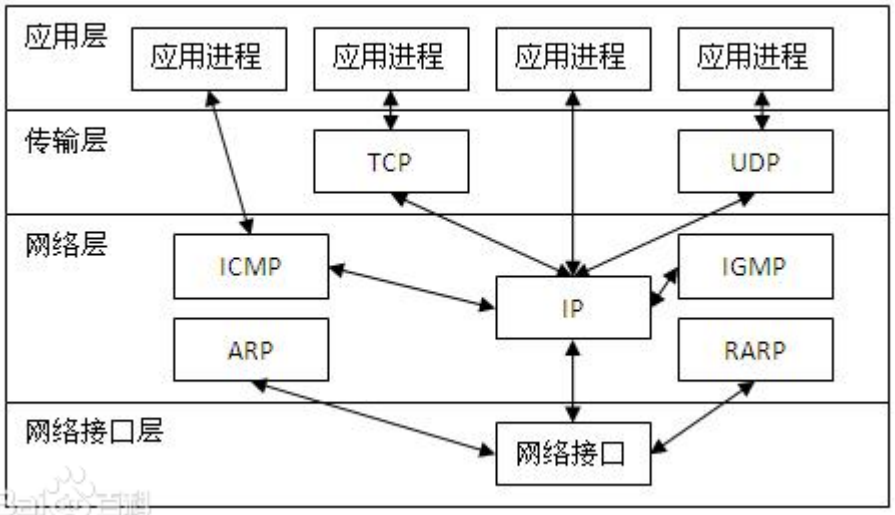
[本教程涵盖了 JavaWeb 部分高频面试题并介绍了计算机网络的相关理论知识，建议大家将目录结构打开，对照目录结构做一个复习提纲准备面试。]

1 TCP/IP

1.1 三种网络分层模型



TCP/IP 协议不是 TCP 和 IP 这两个协议的合称，而是指因特网整个 TCP/IP 协议族。从协议分层模型方面来讲，TCP/IP 由四个层次组成：网络接口层、网络层、传输层、应用层。



1.2 TCP、UDP 协议的区别

- UDP 在**传送数据之前不需要先建立连接**，远地主机在收到 UDP 报文后，**不需要给出任何确认**。虽然 UDP 不提供可靠交付，但在某些情况下 UDP 确是一种最有效的工作方式（一般用于即时通信），比如： QQ 语音、QQ 视频 、直播等
- TCP 提供面向连接的服务。在传**送数据之前必须先建立连接**，数据传送结束后要释放连接。 TCP **不提供广播或多播服务**。由于 TCP 要提供可靠的，面向连接的运输服务（TCP 的可靠体现在 TCP 在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用来节约系统资源），这样难以避免地增加了许多开销，如确认，流量控制，计时器以及连接管理等。这不仅使协议数据单元的首部增大很多，还要占用许多处理机资源。TCP 一般用于文件传输、发送和接收邮件、远程登录等场景。

类型	特点			性能		应用场景	首部字节
	是否面向连接	传输可靠性	传输形式	传输效率	所需资源		
TCP	面向连接	可靠	字节流	慢	多	要求通信数据可靠 （如文件传输、邮件传输）	20-60
UDP	无连接	不可靠	数据报文段	快	少	要求通信速度高 （如域名转换）	8个字节 （由4个字段组成）

1.3 在浏览器中输入 url 地址 ->> 显示主页的过程

总体来说分为以下几个过程:

1. DNS 解析
2. TCP 连接
3. 发送 HTTP 请求
4. 服务器处理请求并返回 HTTP 报文
5. 浏览器解析渲染页面
6. 连接结束

过程	使用的协议
1. 浏览器查找域名的IP地址 (DNS查找过程：浏览器缓存、路由器缓存、DNS 缓存)	DNS：获取域名对应IP
2. 浏览器向web服务器发送一个HTTP请求 (cookies会随着请求发送给服务器)	• TCP：与服务器建立TCP连接 • IP：建立TCP协议时，需要发送数据，发送数据在网络层使用IP协议 • OPSF：IP数据包在路由器之间，路由选择使用OPSF协议 • ARP：路由器在与服务器通信时，需要将ip地址转换为MAC地址，需要使用ARP协议 • HTTP：在TCP建立完成后，使用HTTP协议访问网页
3. 服务器处理请求 (请求 处理请求 & 它的参数、cookies、生成一个HTML响应)	
4. 服务器发回一个HTML响应	
5. 浏览器开始显示HTML	

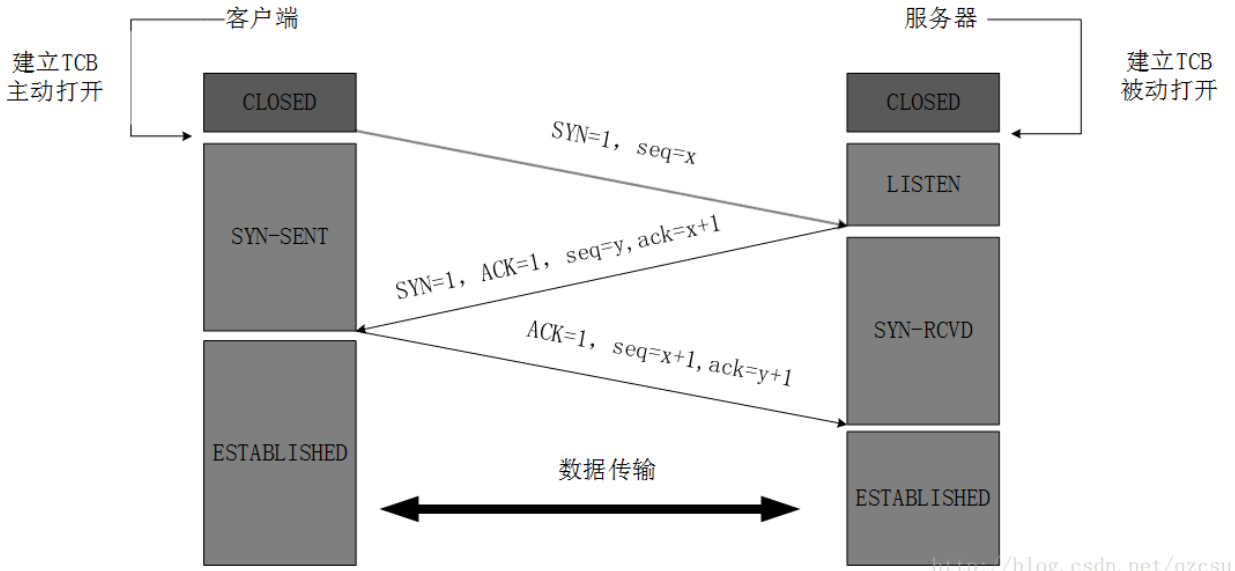
1.4 URI 和 URL 的区别是什么?

- URI(Uniform Resource Identifier) 是同一资源标志符，可以唯一标识一个资源。
- URL(Uniform Resource Location) 是同一资源定位符，可以提供该资源的路径。它是一种具体的 URI，即 URL 可以用来标识一个资源，而且还指明了如何 locate 这个资源。
- URI 的作用像身份证号一样，URL 的作用更像家庭住址一样。URL 是一种具体的 URI，它不仅唯一标识资源，而且还提供了定位该资源的信息。

2 TCP 三次握手/四次挥手

TCP 在传输之前会进行三次沟通，一般称为“三次握手”，传完数据断开的时候要进行四次沟通，一般 称为“四次挥手”。

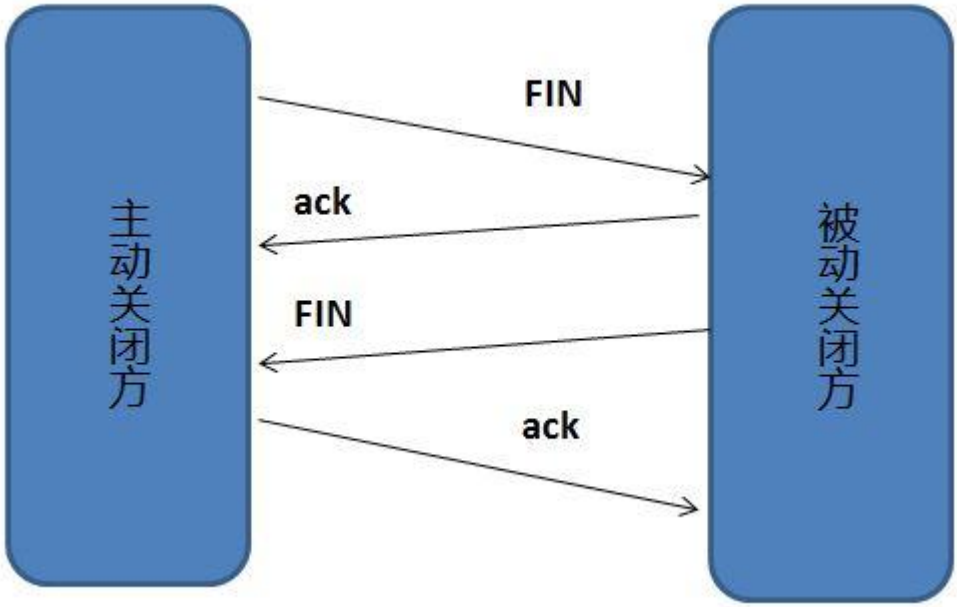
2.1 三次握手【说清楚】



- **准备阶段：**TCP 服务器进程先创建传输控制块 TCB，时刻准备接受客户进程的连接请求，此时服务器就进入了 LISTEN（监听）状态；
- **第一次握手：**TCP 客户进程也是先创建传输控制块 TCB，然后向服务器发出连接请求报文，这是报文首部中的同部位 SYN=1，同时选择一个初始序列号 seq=x ，此时，TCP 客户端进程进入了 SYN-SENT（同步已发送状态）状态。TCP 规定，SYN 报文段（SYN=1 的报文段）不能携带数据，但需要消耗掉一个序号。
- **第二次握手：**TCP 服务器收到请求报文后，如果同意连接，则发出确认报文。确认报文中应该 ACK=1，SYN=1，确认号是 ack=x+1，同时也要为自己初始化一个序列号 seq=y，此时，TCP 服务器进程进入了 SYN-RCVD（同步收到）状态。这个报文也不能携带数据，但是同样要消耗一个序号。
- **第三次握手：**TCP 客户进程收到确认后，还要向服务器给出确认。确认报文的 ACK=1，ack=y+1，自己的序列号 seq=x+1，此时， TCP 连接建立，客户端进入 ESTABLISHED（已建立连接）状态。TCP 规定，ACK 报文段可以携带数据，但是如果不携带数据则不消耗序号。
- **结束阶段：**当服务器收到客户端的确认后也进入 ESTABLISHED 状态，此后双方就可以开始通信了。

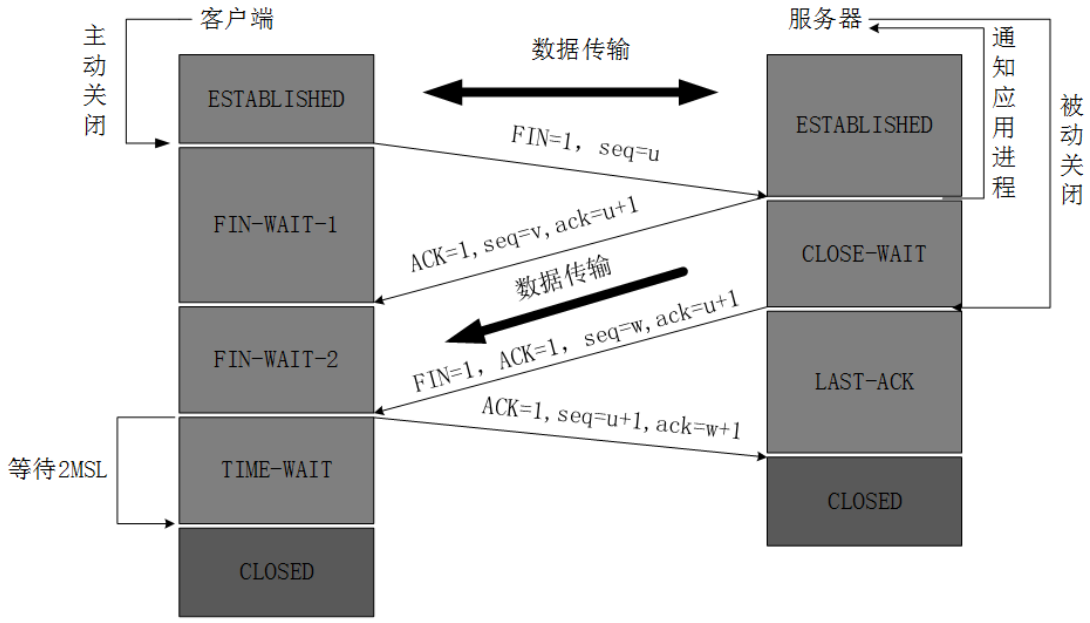
2.2 四次挥手简版

断开一个 TCP 连接则需要“四次挥手”：



- 客户端-发送一个 FIN，用来关闭客户端到服务器的数据传送
- 服务器-收到这个 FIN，它发回一个 ACK，确认序号为收到的序号加 1 。和 SYN 一样，一个 FIN 将占用一个序号
- 服务器-关闭与客户端的连接，发送一个 FIN 给客户端
- 客户端-发回 ACK 报文确认，并将确认序号设置为收到序号加 1

2.3 四次挥手详解



- **准备阶段：**客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为 seq=u（等于前面已经传送过来的数据的最后一个字节的序号加 1），此时，客户端进入 FIN-WAIT-1（终止等待 1）状态。 TCP 规定，FIN 报文段即使不携带数据，也要消耗一个序号。
- **第一次挥手：**服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号 seq=v，此时，服务端就进入了 CLOSE-WAIT（关闭等待）状态。TCP 服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个 CLOSE-WAIT 状态持续的时间。
- **第二次挥手：**客户端收到服务器的确认请求后，此时，客户端就进入 FIN-WAIT-2（终止等待 2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。
- **第三次挥手：**服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为 seq=w，此时，服务器就进入了 LAST-ACK（最后确认）状态，等待客户端的确认。
- **第四次挥手：**客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是 seq=u+1，此时，客户端就进入了 TIME-WAIT（时间等待）状态。注意此时 TCP 连接还没有释放，必须经过 2* **MSL（最长报文段寿命）的时间后，当客户端撤销相应的 TCB 后，才进入 CLOSED 状态。
- **结束阶段：**服务器只要收到了客户端发出的确认，立即进入 CLOSED 状态。同样，撤销 TCB 后，就结束了这次的 TCP 连接。可以看到，服务器结束 TCP 连接的时间要比客户端早一些。

3 HTTP 协议详解

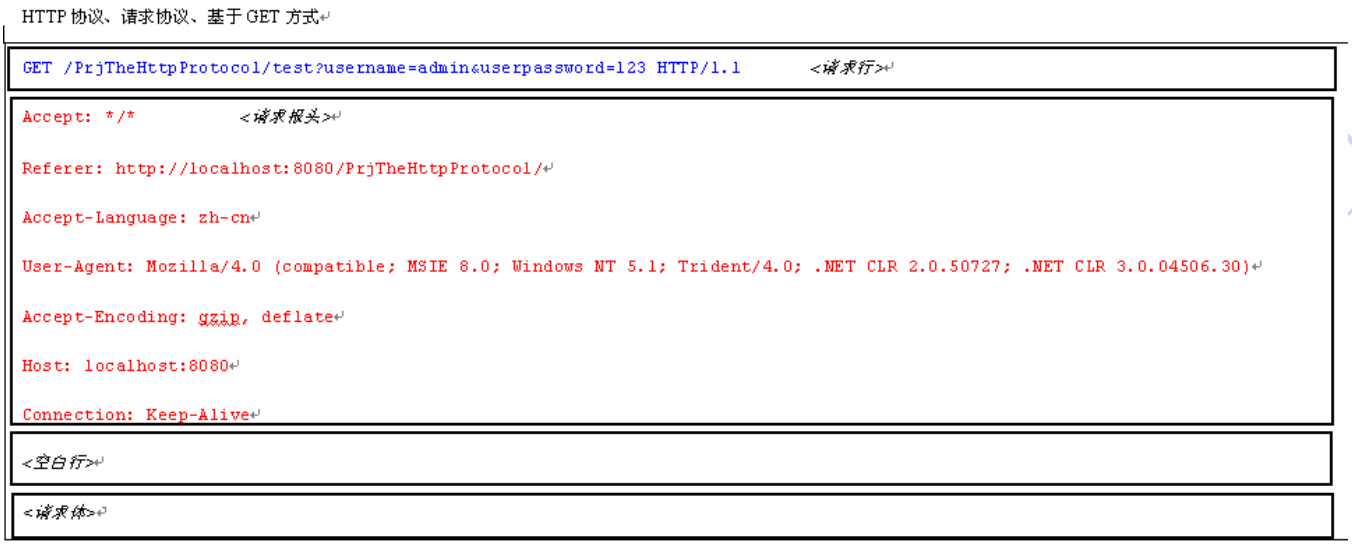
HTTP 协议：超文本传输协议（HTTP，HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议。所有的 WWW 文件都必须遵守这个标准。

3.1 工作原理

由 HTTP 客户端发起一个请求，建立一个到服务器指定端口（默认是 80 端口）的 TCP 连接。连接 HTTP 服务器则在那个端口监听客户端发送过来的请求。一旦收到请求， 请求服务器（向客户端）发回一个状态行和（响应的）消息，消息的消息体可能是请求的文件、错误消息、或者其它一些信息。响应客户端接收服务器所返回的信息通过浏览器显示在用户的显示屏上，然后客户机与服务器断开连接

https://www.cnblogs.com/yumo1627129/p/7941220.html
https://www.cnblogs.com/lauhp/p/8979393.html

3.2 Http 请求 Request



3.2.1 请求行

- 请求方法 （见 2.4）
- 请求资源的 URI
- 协议/版本（见 2.5）

3.2.2 请求报头

每个头域由一个域名，冒号（:）和域值三部分组成。域名是大小写无关的，域值前可以添加任何数量的空格符，头域可以被扩展为多行，在每行开始处，使用至少一个空格或制表符。

- 告诉 web 服务器浏览器接收的语言版本
- 请求 web 服务器的 IP 地址和端口号
- Cookies 等信息。

3.2.3 空白行

（分割请求报头和请求体的专用行）

3.2.4 请求体

GET 请求方式请求体中不传送任何数据，Post 请求时可以传送数据

3.3 Http 的响应 Response

HTTP 协议、响应协议

```
HTTP/1.1 200 OK          <状态行>
Server: Apache-Coyote/1.1    <响应报头>
Content-Type: text/html;charset=GB18030
Content-Length: 132
Date: Wed, 21 Aug 2013 02:09:09 GMT
<空白行>
<html>              <响应体>
  <head>
    <title>Response From WebServer</title>
  </head>
  <body>
      Response from WebServer!
  </body>
</html>
```

3.3.1 状态行

- HTTP1.1 ： HTTP 协议版本号
- **200 ： 响应状态号**
  - ✓ **状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：**
    - **1xx：指示信息--表示请求已接收，继续处理**
    - **2xx：成功--表示请求已被成功接收、理解、接受**
    - **3xx：重定向--要完成请求必须进行更进一步的操作**
    - **4xx：客户端错误--请求有语法错误或请求无法实现**
    - **5xx：服务器端错误--服务器未能实现合法的请求**
  - ✓ **常见状态代码、状态描述、说明：**
    - **200 OK 客户端请求成功**
    - **400 Bad Request 客户端请求有语法错误，不能被服务器所理解**
    - **401 Unauthorized 请求未经授权**
    - **403 Forbidden 服务器收到请求，但是拒绝提供服务**
    - **404 Not Found 请求资源不存在**
    - **500 Internal Server Error 服务器发生不可预期的错误**
    - **503 Server Unavailable 服务器当前不能处理客户端的请求，一段时间后可能恢复正常**
    - **405 浏览器客户端发送的请求和底层的方法 doPost/doGet 不一致导致的。**
- OK ： 对响应结果的描述

3.3.2 响应报头

- WEB 服务器版本信息
- 内容类型以及字符编码方式

- 内容长度，响应回来的总字符数
- 响应时间
- Cookies 等信息。

3.3.3 空白行

（分割响应报头和响应正文的专用行）

3.3.4 响应正文

（从 WEB 服务器端响应回来的 HTML 代码）

3.4 请求方法

3.4.1 8 种请求方法

- GET 请求获取 Request-URI 所标识的资源
- POST 在 Request-URI 所标识的资源后附加新的数据
- HEAD 请求获取由 Request-URI 所标识的资源的响应消息报头
- PUT 请求服务器存储一个资源，并用 Request-URI 作为其标识
- DELETE 请求服务器删除 Request-URI 所标识的资源
- TRACE 请求服务器回送收到的请求信息，主要用于测试或诊断
- CONNECT 保留将来使用
- OPTIONS 请求查询服务器的性能，或者查询与资源相关的选项和需求

3.4.1 GET 和 POST 请求区别

1、GET 请求通过 URL（请求行）提交数据，在 URL 中可以看到所传参数。POST 通过“请求体”传递数据，参数不会在 url 中显示 。但是这个也不能说 GET 请求不安全。其实，HTTP 协议中提到 GET 是安全的方法（safe method），其意思是说 GET 方法不会改变服务器端数据，所以不会产生副作用。这是建立在 Web 开发人员正确使用 GET 方法的基础上的，如果修改数据的请求却使用了 GET 方法，显然是非常危险的。
2、GET 请求提交的数据有长度限制（1024 或 2048），POST 请求没有限制（或限制 80KB）。
3、GET 请求返回的内容可以被浏览器缓存起来。而每次提交的 POST，浏览器在你按下 F5 的时候会跳出确认框，浏览器不会缓存 POST 请求返回的内容。
4、GET 和 POST 的本质区别是使用场景的区别，简单的说，GET 是只读，POST 是写。

3.5 Request 的请求行中的协议\版本详解

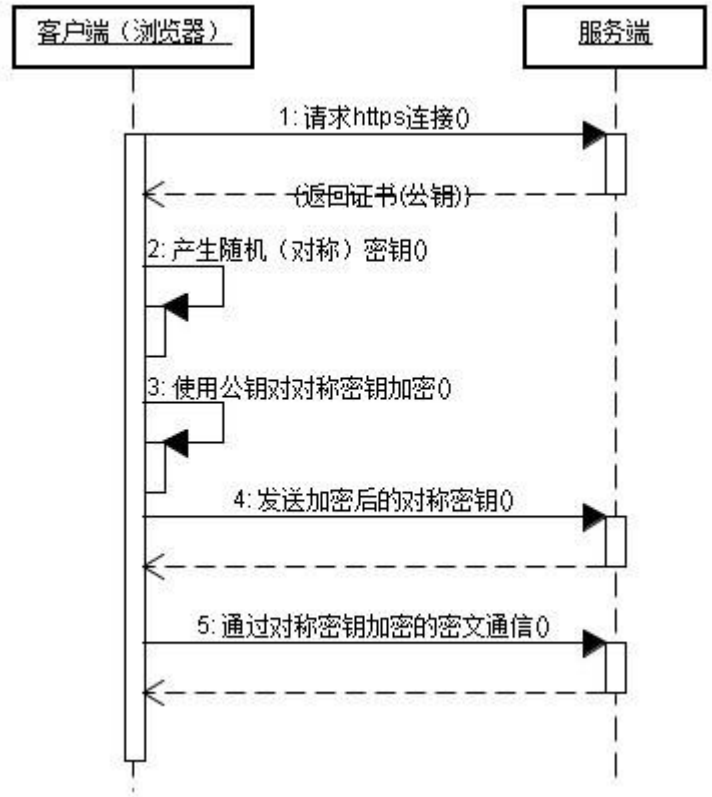
3.5.1 HTTP 1.0 和 HTTP 1.1

HTTP1.0 最早在网页中使用是在 1996 年，那个时候只是使用一些较为简单的网页上和网络请求上，而 HTTP1.1 则在 1999 年才开始广泛应用于现在的各大浏览器网络请求中，同时 HTTP1.1 也是当前使用最为广泛的 HTTP 协议。主要区别主要体现在：

- **长连接** : 1.0 是短连接，1.1 默认是长连接响应头加入 **Connection： keep-alive**
- **错误状态响应码** :在 HTTP1.1 中新增了 24 个错误状态响应码，如 409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
- **缓存处理** :在 HTTP1.0 中主要使用 header 里的 If-Modified-Since,Expires 来做为缓存判断的标准，HTTP1.1 则引入了更多的缓存控制策略例如 Entity tag，If-Unmodified-Since, If-Match, If-None-Match 等更多可供选择的缓存头来控制缓存策略。
- **带宽优化及网络连接的使用** :HTTP1.0 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1 则在请求头引入了 range 头域，它允许只请求资源的某个部分，即返回码是 206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。

3.5.2 HTTPS

HTTPS（全称：Hypertext Transfer Protocol over Secure Socket Layer），是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版。即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL。其所用 的端口号是 443。 过程大致如下：



3.5.2.1 HTTP 和 HTTPS 的区别？

- **端口**：HTTP 的 URL 由“http://”起始且默认使用端口 80，而 HTTPS 的 URL 由“https://”起始且默认使用端口 443。
- **安全性和资源消耗：** HTTP 协议运行在 TCP 之上，所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。HTTPS 是运行在 SSL/TLS 之上的 HTTP 协议，SSL/TLS 运行在 TCP 之上。所有传输的内容都经过加密，**加密采用对称加**密，并且**对称加密的密钥用服务器方的证书进行了非对称加密**。所以说，HTTP 安全性没有 HTTPS 高，但是 HTTPS 比 HTTP 耗费更多服务器资源。

3.5.2.2 对称加密和非对称加密

- 对称加密：密钥只有一个，加密解密为同一个密码，且加解密速度快，典型的对称加密算法有 DES、AES 等；
  - 非对称加密：密钥成对出现（且根据公钥无法推知私钥，根据私钥也无法推知公钥），加密解密使用不同密钥（公钥加密需要私钥解密，私钥加密需要公钥解密），相对对称加密速度较慢，典型的非对称加密算法有 RSA、DSA 等。

3.5.3 HTTP 的长连接、短连接

- 在 HTTP/1.0 中默认使用短连接。也就是说，客户端和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接。HTTP 是基于 TCP/IP 协议的，每一次建立或者断开连接都需要三次握手四次挥手的开销，如果每次请求都要这样的话，开销会比较大。
- 从 HTTP/1.1 起，默认使用长连接，用以保持连接特性。使用长连接的 HTTP 协议，会在响应头加入这行代码： Connection:keep-alive 。
- HTTP/1.1 的持续连接有非流水线方式和流水线方式 。流水线方式是客户在收到 HTTP 的响应报文之前就能接着发送新的请求报文。与之相对应的非流水线方式是客户在收到前一个响应后才能发送下一个请求。
- 在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输 HTTP 数据的 TCP 连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive 不会永久保持连接，它有一个保持时 间，可以在不同的服务器软件（如 Apache）中设定这个时间。
- 实现长连接需要客户端和服务端都支持长连接。
- HTTP 协议的长连接和短连接，实质上是 TCP 协议的长连接和短连接。

4 Jsp 和 servlet 详解

4.1 Servlet 的本质

Servlet 是服务器端的小 java 程序，这个小 java 程序不能随意编写，必须实现 SUN 制定的 javax.servlet.Servlet 接口，实现其中的方法。Servlet 是一个满足 java 规范的 java 类。Servlet 既然满足 Servlet 规范，Tomcat 服务器我们可以叫做“WEB 容器（Container）”，那么 Servlet 就可以叫做容器中的“组件（Component）”

4.2 Servlet 对象的生命周期

Servlet 的生命周期始于将它装入的内存时，并在终止或重新装入 Servlet 时结束。包括加载和实例化、初始化、处理请求以及服务结束。这个生存期由 javax.servlet.Servlet 接口的 init,service 和 destroy 方法表达。

- Servlet 对象只创建一次（构造函数只执行一次）
- Servlet 对象的 init 方法在对象创建之后马上执行（只执行一次完成初始化操作）

  init 方法是 SUN 规范中为程序员提供的一个对象初始化时机，这是一个特定的时刻，有的时候我们需要在这个特定的时刻执行一段特定的程序，此时将该程序写入 init 方法，例如：项目经理要求在 Servlet 对象创建时刻记录日志，请将该程序编写到 init 方法中。对象第一次创建的时候执行记录日志，并且只执行一次，记录一次日志信息。（init 方法一般很少用）
- Servlet 对象 service 方法，只要用户访问一次则执行一次。

  Service 方法是 Servlet 的核心业务方法，核心业务都在该方法中完成，只要编写一个 Servlet，service 方法是一定要编写代码的，完成业务的处理。（常用）
- Servlet 对象的 destroy 方法，也是只执行一次，执行之前对象没有销毁，即将销毁。

4.3 Servlet 的线程安全

servlet 是单例的,对于所有请求都使用一个实例,所以如果有全局变量被多线程使用的时候,就会出现线程安全问题。解决这个问题有三种方案:

1. 实现 singleThreadModel 接口,这样对于每次请求都会创建一个新的 servlet 实列,这样就会消耗服务端内存,降低性能,但是这个接口已经过时,不推荐使用.
2. 可以通过加锁(synchronized 关键字)来避免线程安全问题.这个时候虽然还是单列,但是对于多线程的访问,每次只能有一个请求进行方法体内执行,只有执行完毕后,其他线程才允许访问,降低吞吐量.
3. 避免使用全局变量,使用局部变量可以避免线程安全问题,强烈推荐使用此方法来解决.

Servlet 的线程安全问题只有在大量的并发访问时才会显现出来，并且很难发现，因此在编写 Servlet 程序时要特别注意。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。

4.4 Servlet 总结

在 Java Web 程序中，**Servlet** 主要负责接收用户请求 **HttpServletRequest**,在 **doGet(),doPost()**中做相应的处理，并将回应 **HttpServletResponse** 反馈给用户。Servlet 可以设置初始化参数，供 Servlet 内部使用。一个 Servlet 类只会有一个实例，在它初始化时调用 **init()**方法，销毁时调用 **destroy()**方法。**Servlet 需要在 web.xml 中配置**，一个 **Servlet 可以设置多个 URL 访问**。**Servlet 不是线程安全**，因此要谨慎使用类变量。

4.5 filter 过滤器

filter 是一个过滤器，用来在请求前和响应后进行数据的处理。 filter 的生命周期是:实例化--->初始化(init)-->进行过滤(doFilter)--->销毁(destroy)-->释放资源

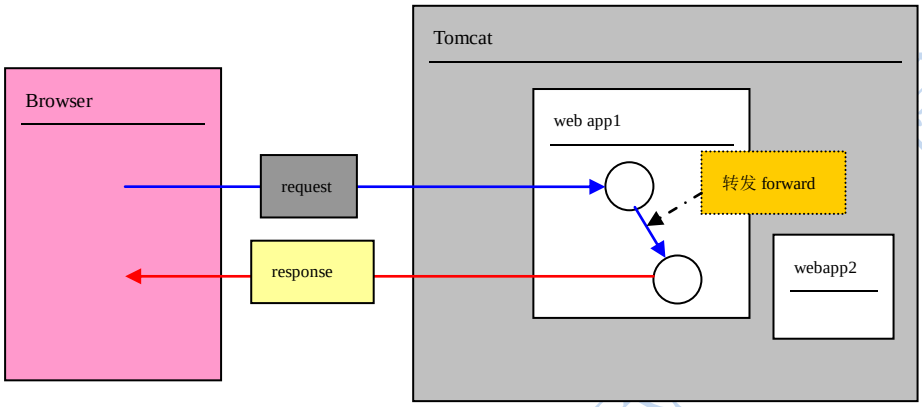
一个 Filter 必须实现 javax.servlet.Filter 接口 在项目中我们通常通过 filter 进行编码转换， 进行安全验证，进行重复提交的判断。

4.6 web.xml 中的 listener、filter、servlet 加载顺序

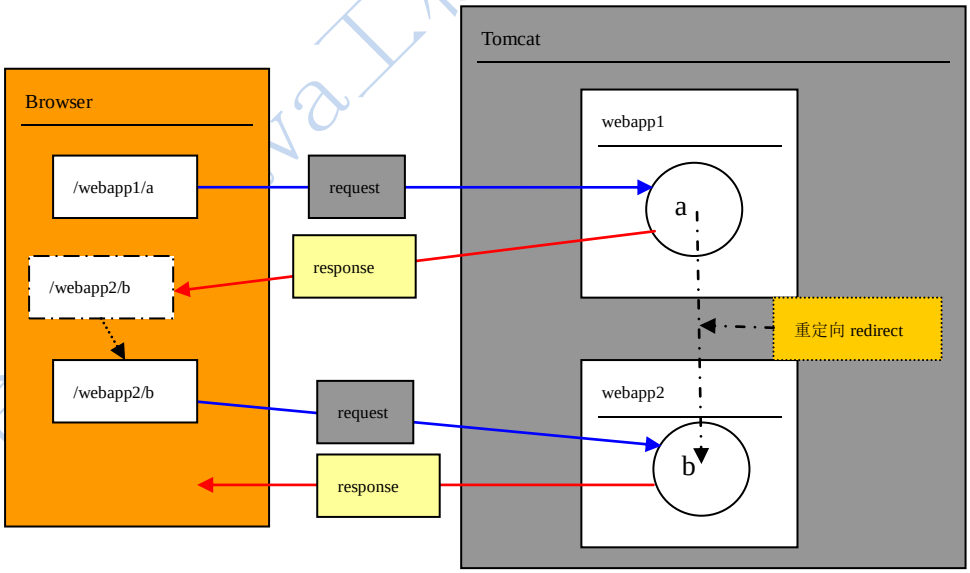
真正的加载顺序为：context-param -> listener -> filter -> servlet

4.7 Forward 和 Redirect

转发图解



重定向图解



- **转发是服务器行为，重定向是客户端行为。**

- **转发**：通过 RequestDispatcher 对象的 forward（HttpServletRequest request,HttpServletResponse response）方法实现的。RequestDispatcher 可以通过 HttpServletRequest 的 getRequestDispatcher()方法获得。
- **重定向**是利用服务器返回的状态码来实现的。客户端浏览器请求服务器的时候，服务器会返回一个状态码。服务器通过 HttpServletRequestResponse 的 setStatus(int status)方法设置状态码。如果服务器返回 301 或者 302，则浏览器会到新的网址重新请求该资源。

- **从地址栏显示来说**
  - forward 是服务器请求资源,服务器直接访问目标地址的 URL,把那个 URL 的响应内容读取过来,然后把这些内容再发给浏览器.浏览器根本不知道服务器发送的内容从哪里来的,所以它的地址栏还是原来的地址.
  - redirect 是服务端根据逻辑,发送一个状态码,告诉浏览器重新去请求那个地址.所以地址栏显示的是新的 URL.
- **从数据共享来说**
  - forward:转发页面和转发到的页面可以共享 request 里面的数据.
  - redirect:不能共享数据.
- **从运用地方来说**
  - forward:一般用于用户登陆的时候,根据角色转发到相应的模块.
  - redirect:一般用于用户注销登陆时返回主页面和跳转到其它的网站等
- 从效率来说
  - forward:高.
  - redirect:低.

4.8 JSP 和 Servlet 是什么关系

Servlet 是一个特殊的 Java 程序，它运行于服务器的 JVM 中，能够依靠服务器的支持向浏览器提供显示内容。JSP 本质上是 Servlet 的一种简易形式，JSP 会被服务器处理成一个类似于 Servlet 的 Java 程序，可以简化页面内容的生成。Servlet 和 JSP 最主要的不同点在于，Servlet 的应用逻辑是在 Java 文件中，并且完全从表示层中的 HTML 分离开来。而 JSP 的情况是 Java 和 HTML 可以组合成一个扩展名为.jsp 的文件。有人说，Servlet 就是在 Java 中写 HTML，而 JSP 就是在 HTML 中写 Java 代码，当然这个说法是很片面且不够准确的。JSP 侧重于视图，Servlet 更侧重于控制逻辑，在 MVC 架构模式中，JSP 适合充当视图（view）而 Servlet 适合充当控制器（controller）。

4.9 JSP 的九大内置对象

- **request**
  表示 HttpServletRequest 对象。它包含了有关浏览器请求的信息，并且提供了几个用于获取 cookie, header, 和 session 数据的有用的方法。
- **response**
  表示 HttpServletResponse 对象，并提供了几个用于设置送回 浏览器的响应的方法（如 cookies,头信息等）
- **out**
  out 对象是 javax.jsp.JspWriter 的一个实例，并提供了几个方法使你能用于向浏览器回送输出结果。
- **pageContext**
  javax.servlet.jsp.PageContext 对象。它是用于方便存取各种范围的名字空间、servlet 相关的对象的 API，并且包装了通用的 servlet 相关功能的方法。
- **session**
  表示一个请求的 javax.servlet.http.HttpSession 对象。 Session 可以存贮用户的状态信息
- **applicaton**

javax.servle.ServletContext 对象。这有助于查找有关 servlet 引擎和 servlet 环境的信息

- **config**
  javax.servlet.ServletConfig 对象。该对象用于存取 servlet 实例的初始化参数。
- **page**
  表示从该页面产生的一个 servlet 实例
- **exception**
  表示异常对象

4.10 JSP 四大作用域

四个作用域从大到小：appliaction>session>request>page

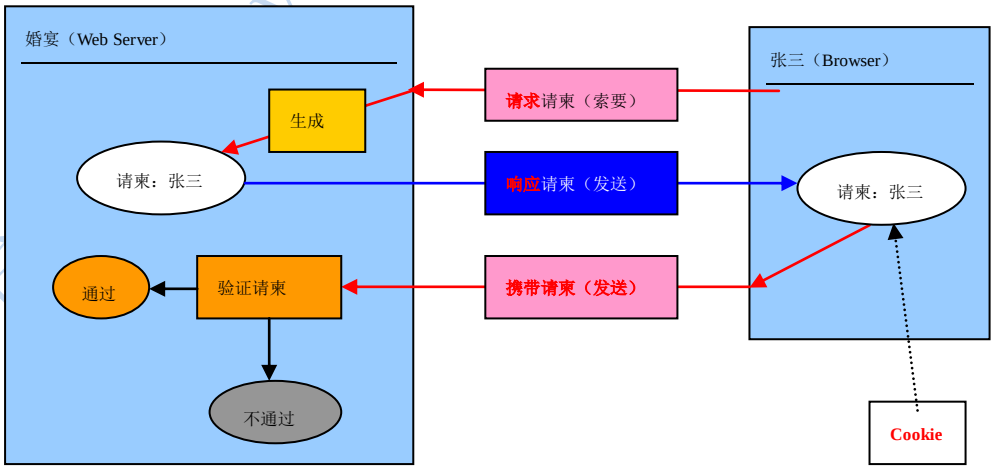
- application：全局作用范围，整个应用程序共享. 生命周期为应用程序启动到停止。
- session：会话作用域，当用户首次访问时，产生一个新的会话，以后服务器就可以记住这个会话状态。
- request：请求作用域，就是客户端的一次请求。
- page：一个 JSP 页面。 以上作用范围使越来越小，

request 和 page 的生命周期都是短暂的，他们之间的区别就是一个 request 可以包含多个 page 页(include，forward)。

4.11 Cookie 和 Session【重点】

4.11.1 Cookie 概述

Cookie 是由服务器端生成并储存在浏览器客户端上的数据。在 javaweb 开发中 Cookie 被当做 java 对象在 web 服务器端创建，并由 web 服务器发送给特定浏览器客户端，并且 WEB 服务器可以向同一个浏览器客户端上同时发送多个 Cookie，每一个 Cookie 对象都由 name 和 value 组成，name 和 value 只能是字符串类型，浏览器接收到来自服务器的 Cookie 数据之后默认将其保存在浏览器缓存中（如果浏览器关闭，缓存消失，Cookie 数据消失），只要浏览器不关闭，当我们下一次发送“特定”请求的时候，浏览器负责将 Cookie 数据发送给 WEB 服务器。我们还可以使用特殊的方法，将 Cookie 保存在客户端的硬盘上。永久性保存。这样关闭浏览器 Cookie 还是存在的，不会消失，比如：实现两周内自动登录。



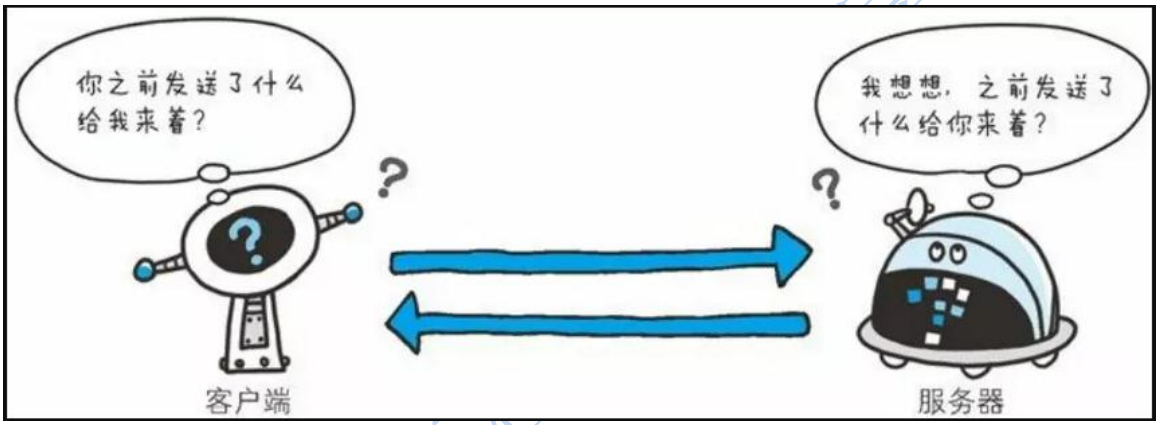
4.11.2 Cookie 有效时间

- 服务器端默认创建的 Cookie，发送到浏览器之后，浏览器默认将其保存在缓存中，当浏览器关闭之后 Cookie 消失，但是可以通过以下方法设置 cookie 保存到客户端中一段时间，而不是关闭浏览器就消失
- 服务器创建 Cookie 对象之后，调用 setMaxAge 方法设置 Cookie 的有效时间，
  - ✓ 如果这个有效时间 >0，则该 Cookie 对象发送给浏览器之后浏览器将其保存到硬盘文件中。
  - ✓ 如果这个有效时间 <0，则该 Cookie 对象也是被保存在浏览器缓存中，待浏览器关闭 Cookie 消失。
  - ✓ 如果这个有效时间 =0，则该 Cookie 从服务器端发过来的时候就已经是一个已过时的 Cookie。

4.11.1 浏览器禁用 Cookie

当浏览器禁用 Cookie 之后，服务器还是仍然会将 Cookie 发送给浏览器，只不过这次浏览器选择了不接收。现在有很多网站的使用都是需要开启接收 Cookie 的。例如 126 邮箱。

如果浏览器被禁用了，最常用的就是利用 URL 重写把 Session ID 直接附加在 URL 路径的后面。



4.11.2 Cookie 和请求路径之间的关系

每一个 Cookie 和请求路径是绑定在一起的，只有特定的路径才可以发送特定的 Cookie。实际上浏览器是这样做的：浏览器在向 web 服务器发送请求的时候先去对应的请求路径下搜索是否有对应的 Cookie，如果有 Cookie，并且 Cookie 没有失效，则发送该 Cookie 或者多个 Cookie 到服务器端。将来发送 cookie 的路径包含获取 cookie 时的路径、获取 cookie 时路径的同级目录以及子目录不过我们也可以在创建 Cookie 对象的时候设置 Cookie 的关联路径 cookie.setPath()。

4.11.3 session 概述

在 java web 中 session 是一个存储在 WEB 服务器端的 java 对象，该对象代表用户和 WEB 服务器的一次会话（一次会话指的就是用户在浏览某个网站时，从进入网站到浏览器关闭所经过的这段时间，也就是用户浏览这个网站所

花费的时间），通过 session 对象可以完成数据的存取，而放在 session 对象中的数据都是用户相关的。也就说张三访问 WEB 服务器，服务器会生成一个张三的 session 对象，李四去访问 WEB 服务器，服务器就会生成一个李四的 session 对象。 系统为每个访问者都设立一个独立的 session 对象，用以存取数据，并且各个访问者的 session 对象互不干扰。

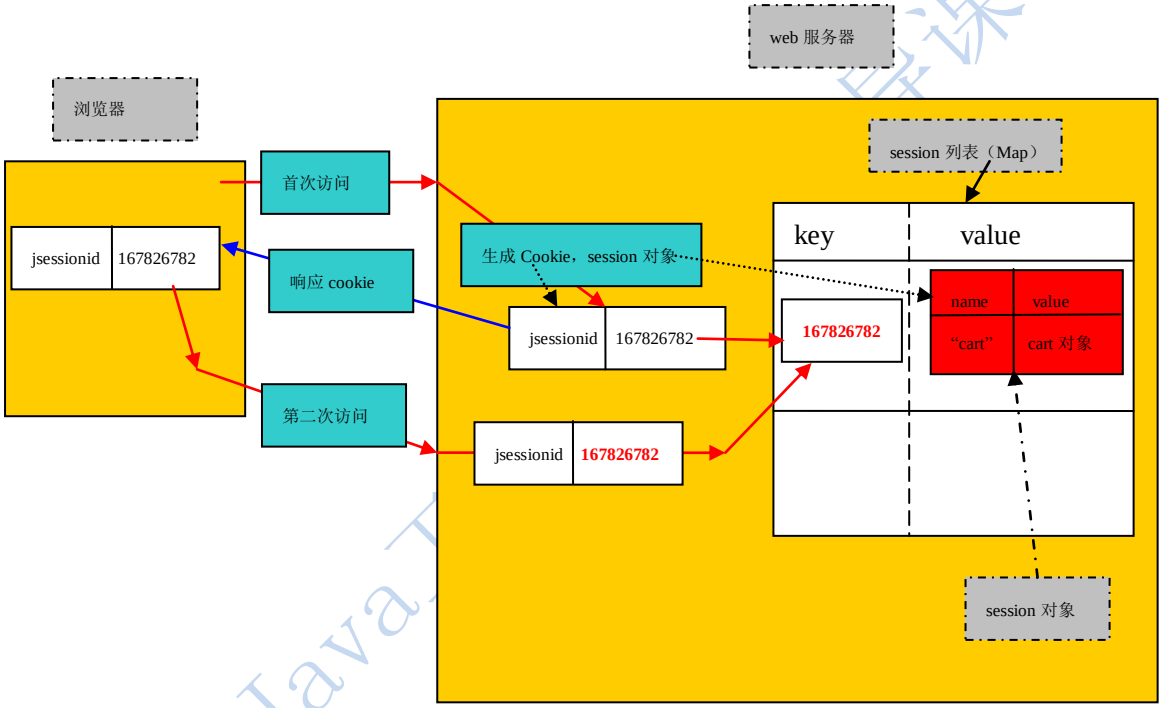
session 与 cookie 是紧密相关的。session 的使用要求用户浏览器必须支持 cookie，如果浏览器不支持使用 cookie，或者设置为禁用 cookie，那么将不能使用 Session。除非服务器使用了 URL 重写机制（uri;jsessionid=xxxxxx）。

4.11.4 session 的工作原理

■ 分析以下案例

假设有两个用户，一个是北京的张三，一个是南京的李四，都在访问京东商城购物网站，那么在京东 WEB 服务器中一定会有两个购物车，一个是张三的购物车，一个是属于李四的购物车，大家思考：一个 WEB 服务器，两个浏览器客户端，为什么张三在购物的时候向购物车中放入的商品一定是放到张三的购物车中，而不会存放到李四的购物车中，也就是说 session 是怎么实现和某个特定用户绑定的？下面使用图形描述 session 的工作原理

【HTTP 是不保存状态的协议,如何保存用户状态?】



- 当用户第一次访问 web 服务器的时候，web 服务器会为该用户分配一个 session 对象，并且同时生成一个 cookie 对象（jsessionid=xxxx），然后 web 服务器将 cookie 对象的值和 session 对象以键值对的方式存储在 web 服务器端的 session 列表（Map）中，服务器负责将该 cookie 数据发送给浏览器，浏览器将 cookie 信息存储在浏览器的缓存中
- 只要浏览器不关闭，用户第二次访问服务器的时候会自动发送缓存中存储的 cookie 数据给服务器，服务器收到 cookie 数据之后获取 cookie 的值，然后通过该值在 session 列表（Map）中搜索对应的 session 对象。需要注意的是，当浏览器关闭之后，缓存中的 cookie 消失，这样客户端下次再访问服务器的时候就无法获取到服务器端的 session 对象了。这就意味着会话已经结束。但是这并不代表服务器端的 session 对象马上被回收，session 对象仍然在 session 列表中存储，当长时间没有用户再访问这个 session 对象了，我们称作 session 超时，此时 web 服务器才会回收 session 对象。
- 什么情况下一次会话结束？

浏览器关闭，缓存中的 Cookie 消失，会话不一定结束，因为服务器端的 session 对象还没有被销毁。我们还可

以通过 URL 重写机制继续访问 Session 对象。

浏览器没关闭，但是由于长时间没有访问 web 服务器，服务器判定 session 超时，将 session 对象销毁。此时浏览器虽然没关闭，但是这次会话已经结束。

- session 对象所关联的这个 Cookie 的 name 有点特殊，这个 name 必须是 jsessionid,必须全部小写，这是 HTTP 协议中规定的。
- 浏览器禁用了 Cookie，可以采用 URL 重写机制（这样编码的代价比较大，所以一般网站都是不允许禁用 Cookie 的）http://ip:port/webapp/servlet/accessSys**;jsessionid=xxxxxx**

注意：B/S 架构采用 HTTP 协议通信，该协议是一个无连接协议，浏览器向服务器发送请求的瞬间是连接状态，等请求结束之后，B 和 S 的连接断开了，服务器是无法监测浏览器的（浏览器关闭服务器是无法监测的）。

4.11.5 HttpSession 对象的相关方法

- 获取 session 对象：request.getSession(true); request.getSession();如果获取不到则新建
- 获取 session 对象：request.getSession(false);如果获取不到则返回 null
- 向 session 中存储数据：session.setAttribute("name",ObjectValue);
- 从 session 中获取数据：Object value = session.getAttribute("name");
- 删除 session 中某个数据：session.removeAttribute("name");
- 使 session 失效：session.invalidate();

4.11.6 session 对象的超时包括三种设置方式

1. 在 web 容器中设置（此处以 tomcat 为例）

在 tomcat-5.0.28\conf\web.xml 中设置，以下是 tomcat 5.0 中的默认配置：

```
<session-config>
        <session-timeout>30</session-timeout>
</session-config>
```

Tomcat 默认 session 超时时间为 30 分钟，可以根据需要修改，负数或 0 为不限制 session 失效时间。

2. 在工程的 web.xml 中设置 设置方式通在 tomcat 中设置一样
3. 通过 java 代码设置 session.setMaxInactiveInterval（30*60）； //以秒为单位

三种方式优先级 3>2>1

4.11.7 session 和 cookie 的区别

- **session 是存储在服务器端， cookie 是存储在客户端的**，所以安全来讲 session 的安全性要比 cookie 高。又由于 session 是存放在服务器的内存中，所以 session 里的东西不断增加会造成服务器的负担，所以会把很重要的信息存储在 session 中，而把一些次要东西存储在客户端的 cookie 里
- cookie 确切的说分为两大类分为会话 cookie 和持久化 cookie

  会话 cookie 确切的说是存放在客户端浏览器的内存中，所以说他的生命周期和浏览器是一致的，浏览器关了会话 cookie 也就消失了，然而持久化 cookie 是存放在客户端硬盘中， 而持久化 cookie 的生命周期就是我们在设置 cookie 时候设置的那个保存时间
- 我们获取 session 里的信息是通过存放在会话 cookie 里的 sessionid 获取的。Cookie 禁用的情况下，使用所有的访问路径 url 后面，带一个 sessionid 的参数

5 Netty【了解】

5.1 原理

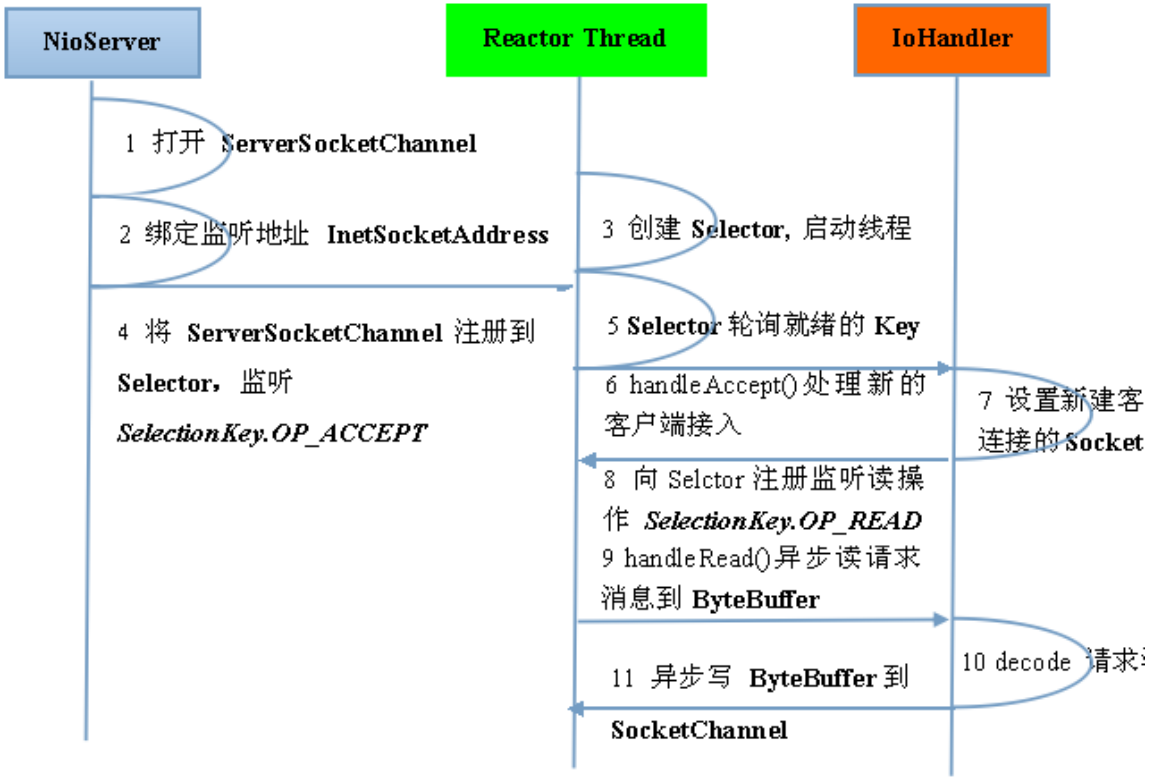
Netty 是一个高性能、异步事件驱动的 NIO 框架，基于 JAVA NIO 提供的 API 实现。它提供了对 TCP、UDP 和文件传输的支持，作为一个异步 NIO 框架，Netty 的所有 IO 操作都是异步非阻塞 的，通过 Future-Listener 机制，用户可以方便的主动获取或者通过通知机制获得 IO 操作结果

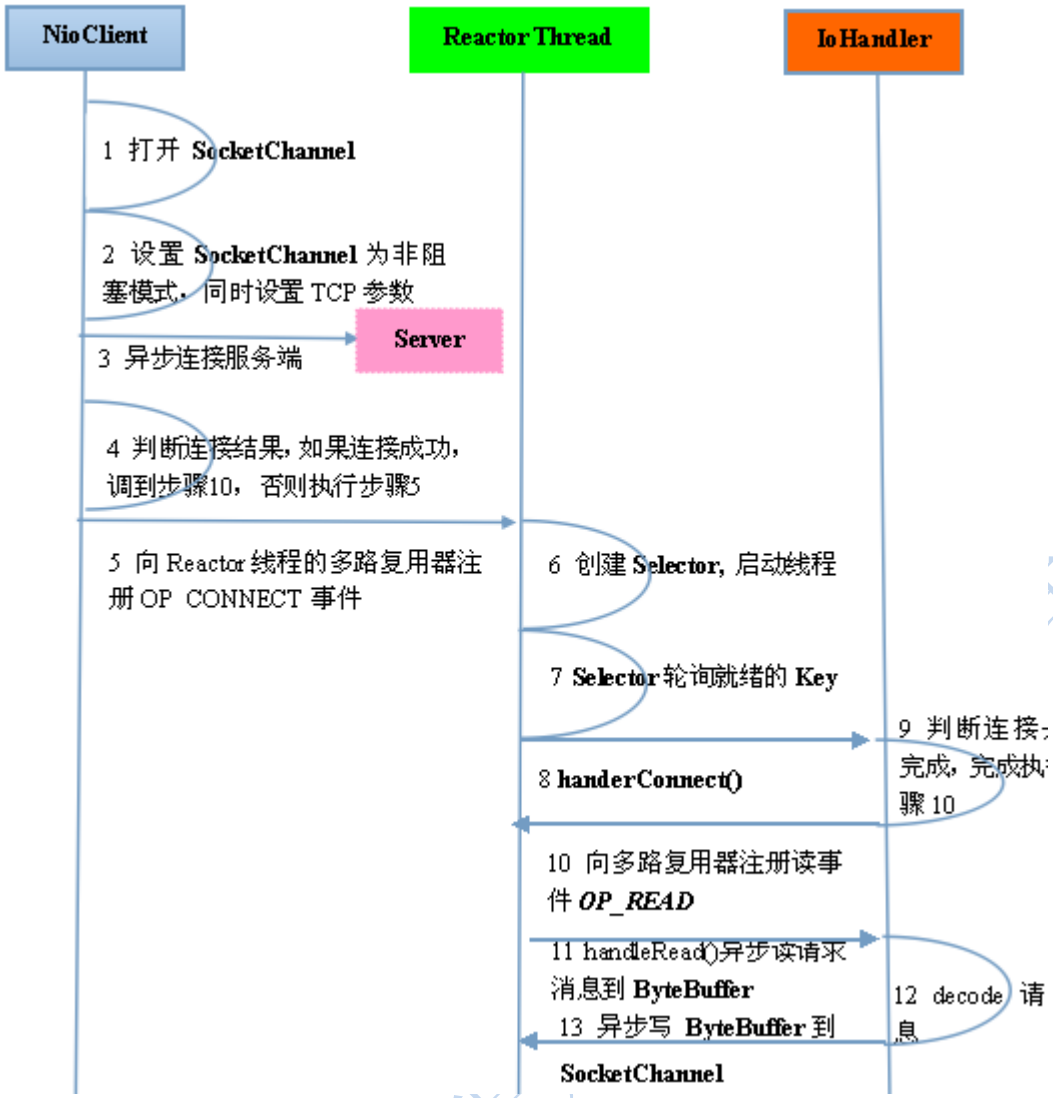
5.2 Netty 高性能

在 IO 编程过程中，当需要同时处理多个客户端接入请求时，可以利用多线程或者 IO 多路复用技术 进行处理。IO 多路复用技术通过把多个 IO 的阻塞复用到同一个 select 的阻塞上，从而使得系统在 单线程的情况下可以同时处理多个客户端请求。与传统的多线程/多进程模型比，I/O 多路复用的 最大优势是系统开销小，系统不需要创建新的额外进程或者线程，也不需要维护这些进程和线程 的运行，降低了系统的维护工作量，节省了系统资源。与 Socket 类和 ServerSocket 类相对应，NIO 也提供了 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现。

5.2.1 多路复用通讯方式

Netty 架构按照 Reactor 模式设计和实现，它的服务端通信序列图如下：





Netty 的 IO 线程 NioEventLoop 由于聚合了多路复用器 Selector，可以同时并发处理成百上千个 客户端 Channel，由于读写操作都是非阻塞的，这就可以充分提升 IO 线程的运行效率，避免由于 频繁 IO 阻塞导致的线程挂起。

5.2.2 异步通讯 NIO

由于 Netty 采用了异步通信模式，一个 IO 线程可以并发处理 N 个客户端连接和读写操作，这从根 本上解决了传统同步阻塞 IO 一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极 大的提升。

5.2.3 零拷贝（DIRECT BUFFERS 使用堆外直接内存）

1. Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进行 Socket 读写， 不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行 Socket 读写， JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后才写入 Socket 中。相比于堆外直接内存， 消息在发送过程中多了一次缓冲区的内存拷贝。

2. Netty 提供了组合 Buffer 对象，可以聚合多个 ByteBuffer 对象，用户可以像操作一个 Buffer 那样 方便的对组合 Buffer 进行操作，避免了传统通过内存拷贝的方式将几个小 Buffer 合并成一个大的 Buffer。

3. Netty 的文件传输采用了 transferTo 方法，它可以直接将文件缓冲区的数据发送到目标 Channel， 避免了传

统通过循环 write 方式导致的内存拷贝问题

5.2.4 内存池（基于内存池的缓冲区重用机制）

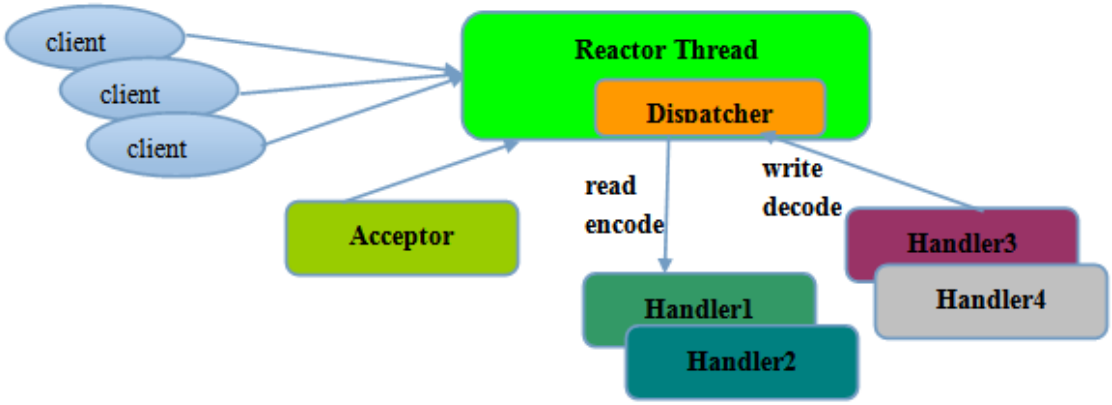
随着 JVM 虚拟机和 JIT 即时编译技术的发展，对象的分配和回收是个非常轻量级的工作。但是对于缓 冲区 Buffer，情况却稍有不同，特别是对于堆外直接内存的分配和回收，是一件耗时的操作。为了尽 量重用缓冲区，Netty 提供了基于内存池的缓冲区重用机制。

5.2.5 高效的 Reactor 线程模型

常用的 Reactor 线程模型有三种，Reactor 单线程模型, Reactor 多线程模型，主从 Reactor 多线程模 型。

5.2.5.1 Reactor 单线程模型

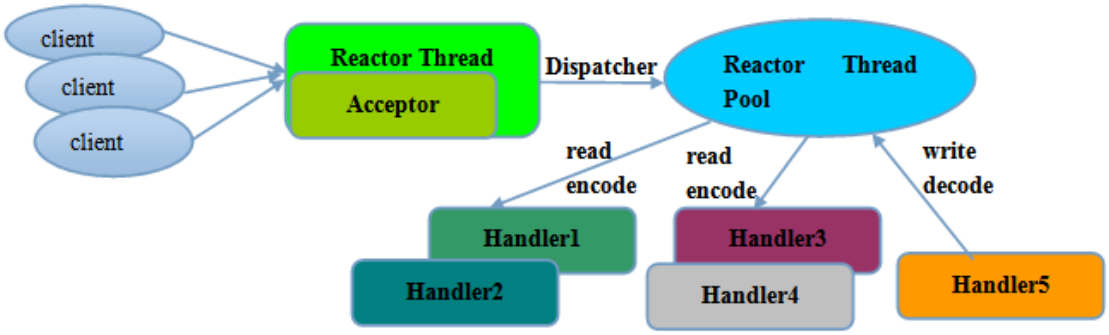
Reactor 单线程模型，指的是所有的 IO 操作都在同一个 NIO 线程上面完成，NIO 线程的职责如下： 1) 作为 NIO 服务端，接收客户端的 TCP 连接；
2) 作为 NIO 客户端，向服务端发起 TCP 连接；
3) 读取通信对端的请求或者应答消息；
4) 向通信对端发送消息请求或者应答消息。



由于 Reactor 模式使用的是异步非阻塞 IO，所有的 IO 操作都不会导致阻塞，理论上一个线程可以独 立处理所有 IO 相关的操作。从架构层面看，一个 NIO 线程确实可以完成其承担的职责。例如，通过 Acceptor 接收客户端的 TCP 连接请求消息，链路建立成功之后，通过 Dispatch 将对应的 ByteBuffer 派发到指定的 Handler 上进行消息解码。用户 Handler 可以通过 NIO 线程将消息发送给客户端。

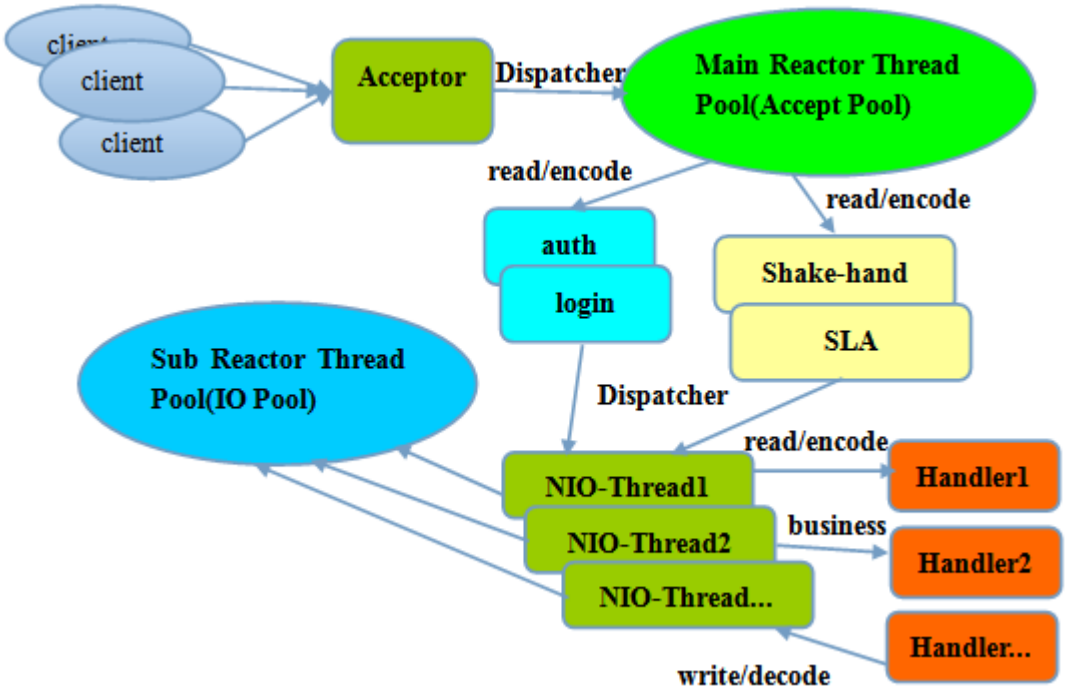
5.2.5.2 Reactor 多线程模型

Rector 多线程模型与单线程模型最大的区别就是有一组 NIO 线程处理 IO 操作。 有专门一个 NIO 线程 -Acceptor 线程用于监听服务端，接收客户端的 TCP 连接请求； 网络 IO 操作-读、写 等由一个 NIO 线程池负责，线程池可以采用标准的 JDK 线程池实现，它包含一个任务队列和 N 个可用的线程，由这些 NIO 线程负责消息的读取、解码、编码和发送；



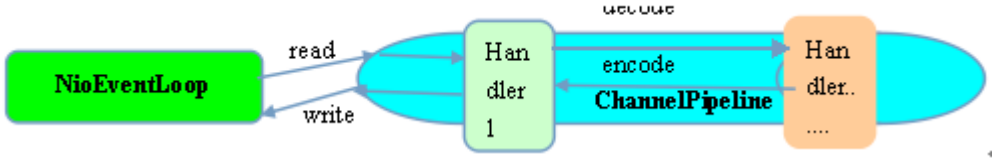
5.2.5.3 主从 Reactor 多线程模型

服务端用于接收客户端连接的不再是个 1 个单独的 NIO 线程，而是一个独立的 NIO 线程池。 Acceptor 接收到客户端 TCP 连接请求处理完成后（可能包含接入认证等），将新创建的 SocketChannel 注册到 IO 线程池（sub reactor 线程池）的某个 IO 线程上，由它负责 SocketChannel 的读写和编解码工作。Acceptor 线程池仅仅只用于客户端的登陆、握手和安全 认证，一旦链路建立成功，就将链路注册到后端 subReactor 线程池的 IO 线程上，由 IO 线程负 责后续的 IO 操作。



5.2.6 无锁设计、线程绑定

Netty 采用了串行无锁化设计，在 IO 线程内部进行串行操作，避免多线程竞争导致的性能下降。 表面上看，串行化设计似乎 CPU 利用率不高，并发程度不够。但是，通过调整 NIO 线程池的线程 参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列- 多个工作线程模型性能更优。



Netty 的 NioEventLoop 读取到消息之后，直接调用 ChannelPipeline 的 fireChannelRead(Object msg)，只要用户不主动切换线程，一直会由 NioEventLoop 调用 到用户的 Handler，期间不进行线程切换，这种串行化处理方式避免了多线程操作导致的锁 的竞争，从性能角度看是最优的。

5.2.7 高性能的序列化框架

Netty 默认提供了对 Google Protobuf 的支持，通过扩展 Netty 的编解码接口，用户可以实现其它的 高性能序列化框架，例如 Thrift 的压缩二进制编解码框架。

1. SO_RCVBUF 和 SO_SNDBUF：通常建议值为 128K 或者 256K。

   小包封大包，防止网络阻塞

2. SO_TCPNODELAY：NAGLE 算法通过将缓冲区内的小封包自动相连，组成较大的封包，阻止大量 小封包的发送阻塞网络，从而提高网络应用效率。但是对于时延敏感的应用场景需要关闭该优化算 法。

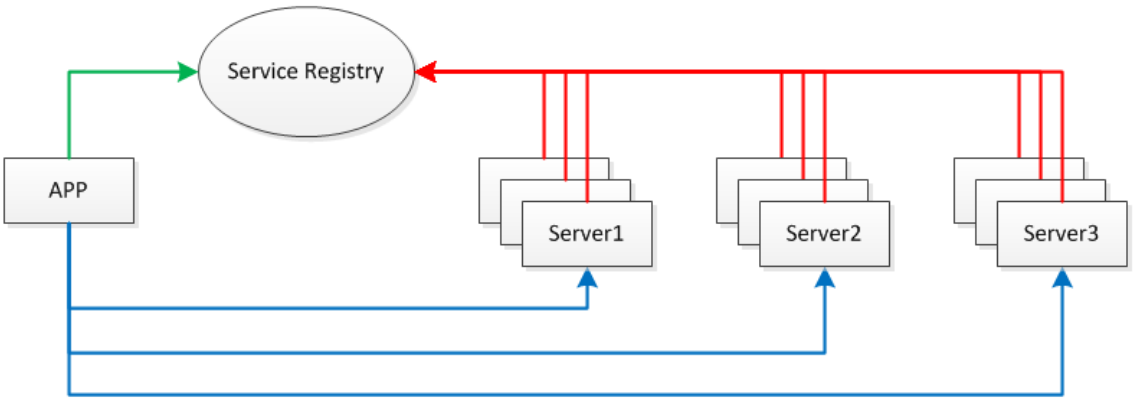
   软中断 Hash 值和 CPU 绑定

3. 软中断：开启 RPS 后可以实现软中断，提升网络吞吐量。RPS 根据数据包的源地址，目的地址以 及目的和源端口，计算出一个 hash 值，然后根据这个 hash 值来选择软中断运行的 cpu，从上层 来看，也就是说将每个连接和 cpu 绑定，并通过这个 hash 值，来均衡软中断在多个 cpu 上，提升 网络并行处理性能。

5.3 Netty RPC 实现

5.3.1 概念

RPC，即 Remote Procedure Call（远程过程调用），调用远程计算机上的服务，就像调用本地服务一 样。RPC 可以很好的解耦系统，如 WebService 就是一种基于 Http 协议的 RPC。这个 RPC 整体框架 如下：



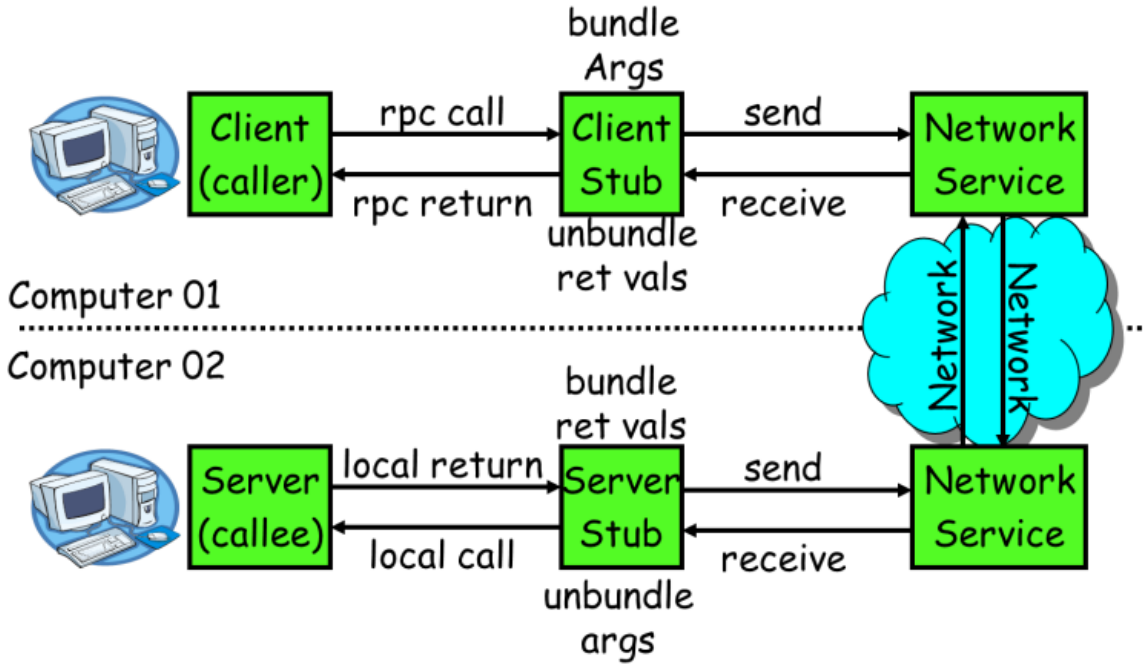
5.3.2 关键技术

1. 服务发布与订阅：服务端使用 Zookeeper 注册服务地址，客户端从 Zookeeper 获取可用的服务 地址。
2. 通信：使用 Netty 作为通信框架。
3. Spring：使用 Spring 配置服务，加载 Bean，扫描注解。
4. 动态代理：客户端使用代理模式透明化服务调用。
5. 消息编解码：使用 Protostuff 序列化和反序列化消息。

5.3.3 核心流程

1. 服务消费方（client）调用以本地调用方式调用服务；
2. client stub 接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体；
3. client stub 找到服务地址，并将消息发送到服务端；
4. server stub 收到消息后进行解码；
5. server stub 根据解码结果调用本地的服务；
6. 本地服务执行并将结果返回给 server stub；
7. server stub 将返回结果打包成消息并发送至消费方；
8. client stub 接收到消息，并进行解码；
9. 服务消费方得到最终结果。

RPC 的目标就是要 2~8 这些步骤都封装起来，让用户对这些细节透明。JAVA 一般使用动态代 理方式实现远程调用。



5.3.4 消息编解码

5.3.4.1 息数据结构（接口名称+方法名+参数类型和参数值+超时时间+ requestID）

客户端的请求消息结构一般需要包括以下内容：

1. 接口名称：在我们的例子里接口名是“HelloWorldService”，如果不传，服务端就不知道调用哪 个接口了；
2. 方法名：一个接口内可能有很多方法，如果不传方法名服务端也就不知道调用哪个方法；
3. 参数类型和参数值：参数类型有很多，比如有 bool、int、long、double、string、map、list， 甚至如 struct（class）；以及相应的参数值；
4. 超时时间：
5. requestID，标识唯一请求 id，在下面一节会详细描述 requestID 的用处。
6. 服务端返回的消息 ： 一般包括以下内容。返回值+状态 code+requestID

5.3.4.2 序列化

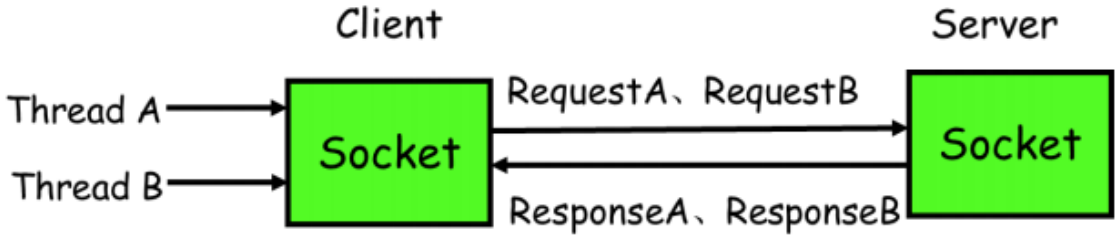
目前互联网公司广泛使用 Protobuf、Thrift、Avro 等成熟的序列化解决方案来搭建 RPC 框架，这 些都是久经考验的解决方案。

5.3.5 通讯过程

核心问题(线程暂停、消息乱序)

如果使用 netty 的话，一般会用 channel.writeAndFlush()方法来发送消息二进制串，这个方 法调用后对于整个远程调用(从发出请求到接收到结果)来说是一个异步的，即对于当前线程来说， 将请求发送出来后，线程就可以往后执行了，至于服务端的结果，是服务端处理完成后，再以消息 的形式发送给客户端的。于是这里出现以下两个问题：

1. 怎么让当前线程“暂停”，等结果回来后，再向后执行？
2. 如果有多个线程同时进行远程方法调用，这时建立在 client server 之间的 socket 连接上 会有很多双方发送的消息传递，前后顺序也可能是随机的，server 处理完结果后，将结 果消息发送给 client，client 收到很多消息，怎么知道哪个消息结果是原先哪个线程调用 的？如下图所示，线程 A 和线程 B 同时向 client socket 发送请求 requestA 和 requestB， socket 先后将 requestB 和 requestA 发送至 server，而 server 可能将 responseB 先返回，尽管 requestB 请求到达时间更晚。我们需要一种机制保证 responseA 丢给 ThreadA，responseB 丢给 ThreadB。



通讯流程

requestID 生成-AtomicLong

1. client 线程每次通过 socket 调用一次远程接口前，生成一个唯一的 ID，即 requestID （requestID 必需保证在一个 Socket 连接里面是唯一的），一般常常使用 AtomicLong 从 0 开始累计数字生成唯一 ID；

存放回调对象 callback 到全局 ConcurrentHashMap

2. 将处理结果的回调对象 callback，存放到全局 ConcurrentHashMap 里面 put(requestID, callback)；

synchronized 获取回调对象 callback 的锁并自旋 wait

2. 当线程调用 channel.writeAndFlush()发送消息后，紧接着执行 callback 的 get()方法试图获取远程返回的结果。在 get()内部，则使用 synchronized 获取回调对象 callback 的锁，再先检测是否已经获取到结果，如果没有，然后调用 callback 的 wait()方法，释放 callback 上的锁，让当前线程处于等待状态。

   监听消息的线程收到消息，找到 callback 上的锁并唤醒

3. 服务端接收到请求并处理后，将 response 结果（此结果中包含了前面的 requestID）发送给客户端，客户端 socket 连接上专门监听消息的线程收到消息，分析结果，取到 requestID，再从前面的 ConcurrentHashMap 里面 get(requestID)，从而找到 callback 对象，再用 synchronized 获取 callback 上的锁，将方法调用结果设置到 callback 对象里，再调用 callback.notifyAll()唤醒前面处于等待状态的线程。

```
public Object get() {
    synchronized (this) { // 旋锁
        while (true) { // 是否有结果了
            If (!isDone) {
                wait(); //没结果释放锁，让当前线程处于等待状态
            }else{//获取数据并处理
            }
        }
    }
}
private void setDone(Response res) {
    this.res = res;
    isDone = true;
    synchronized (this) { //获取锁，因为前面 wait()已经释放了 callback 的锁了
notifyAll(); // 唤醒处于等待的线程
    }
}
```

5.4 RMI 实现方式

Java 远程方法调用，即 Java RMI（Java Remote Method Invocation）是 Java 编程语言里，一种用于实现远程过程调用的应用程序编程接口。它使客户机上运行的程序可以调用远程服务器上的对象。远程方法调用特性使 Java 编程人员能够在网络环境中分布操作。RMI 全部的宗旨就是尽可能简化远程接口对象的使用。

5.4.1 实现步骤

1. 编写远程服务接口，该接口必须继承 java.rmi.Remote 接口，方法必须抛出 java.rmi.RemoteException 异常；

2. 编写远程接口实现类，该实现类必须继承 java.rmi.server.UnicastRemoteObject 类；3. 运行 RMI 编译器（rmic），创建客户端 stub 类和服务端 skeleton 类;

4. 启动一个 RMI 注册表，以便驻留这些服务;

5. 在 RMI 注册表中注册服务；

6. 客户端查找远程对象，并调用远程方法；

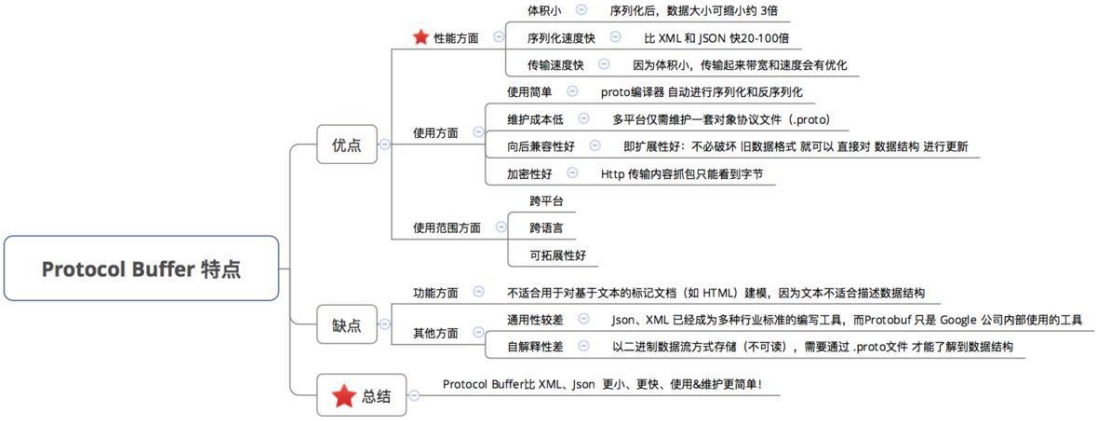
```
//1: 创建远程接口，    继承 java.rmi.Remote 接口
public interface GreetService extends java.rmi.Remote {
    String sayHello(String name) throws RemoteException
}
//2: 实现远程接口，继承  java.rmi.server.UnicastRemoteObject 类
public class GreetServiceImpl extends java.rmi.server.UnicastRemoteObject implements
GreetService {
    private static final long serialVersionUID = 3434060152387200042L;
    public GreetServiceImpl() throws RemoteException {
        super();
    }
    @Override
    public String sayHello(String name) throws RemoteException {
        return "Hello " + name;
    }
}
//3: 生成  Stub 和  Skeleton;
//4: 执行  rmiregistry 命令注册服务
//5: 启动服务
LocateRegistry.createRegistry(1098);
Naming.bind("rmi://10.108.1.138:1098/GreetService", new GreetServiceImpl());

//6.客户端调用
GreetService greetService =
(GreetService)Naming.lookup("rmi://10.108.1.138:1098/GreetService");
System.out.println(greetService.sayHello("Jobs"));
```

5.5 Protoclol Buffer

protocol buffer 是 google 的一个开源项目,它是用于结构化数据串行化的灵活、高效、自动的方法， 例如 XML，不过它比 xml 更小、更快、也更简单。你可以定义自己的数据结构，然后使用代码生成器 生成的代码来读写这个数据结构。你甚至可以在无需重新部署程序的情况下更新数据结构。

5.5.1 特点



Protocol Buffer 的序列化 & 反序列化简单 & 速度快的原因是：

1. 编码 / 解码 方式简单（只需要简单的数学运算 = 位移等等）
2. 采用 Protocol Buffer 自身的框架代码 和 编译器 共同完成

Protocol Buffer 的数据压缩效果好（即序列化后的数据量体积小）的原因是：

1. a. 采用了独特的编码方式，如 Varint、Zigzag 编码方式等等
2. b. 采用 T - L - V 的数据存储方式：减少了分隔符的使用 & 数据存储得紧凑

5.6 Thrift

Apache Thrift 是 Facebook 实现的一种高效的、支持多种编程语言的远程服务调用的框架。本文将从 Java 开发人员角度详细介绍 Apache Thrift 的架构、开发和部署，并且针对不同的传输协议和服务类 型给出相应的 Java 实例，同时详细介绍 Thrift 异步客户端的实现，最后提出使用 Thrift 需要注意的事 项。

目前流行的服务调用方式有很多种，例如基于 SOAP 消息格式的 Web Service，基于 JSON 消息格式 的 RESTful 服务等。其中所用到的数据传输方式包括 XML，JSON 等，然而 XML 相对体积太大，传输 效率低，JSON 体积较小，新颖，但还不够完善。本文将介绍由 Facebook 开发的远程服务调用框架 Apache Thrift，它采用接口描述语言定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码 生成引擎可以在多种语言中，如 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk 等创建高效的、无缝的服务，其传输数据采用二进制格式，相对 XML 和 JSON 体积更小， 对于高并发、大数据量和多语言的环境更有优势。本文将详细介绍 Thrift 的使用，并且提供丰富的实例 代码加以解释说明，帮助使用者快速构建服务。

为什么要 Thrift：

1、多语言开发的需要 2、性能问题