钱进培训是哈法地区资深工程师组成的培训机构，通过各位老师的现身说法，帮助各位学员迅速掌握实战知识，为求职打下坚实的基础。电子邮件：jin.qian.canada@gmail.com钱老师报名、答疑微信号：qianjincanada，或扫描以下二维码添加：

钱老师

加拿大

扫一扫上面的二维码图案，加我微信

Java高级速成班

本课程针对有一定编成基础，希望在短时间内对Java企业级开发有所了解的同学。

通过本课程的学习，学员能够顺利掌握J2EE的体系结构，了解常见的Java框架并熟练使用，具体内容包括:

1. B/s体系结构，HTTP协议栈相关内容（HTML，CSS，JS）
2. 数据库访问的不同方法（Hibernate使用）
3. Spring IOC在企业开发中的应用
4. Restful API/SOAP开发及应用
5. SVN/GIT/MAVEN的应用

第二讲：B/S

作业：

1。JAXB，XML和JavaBean互相转化

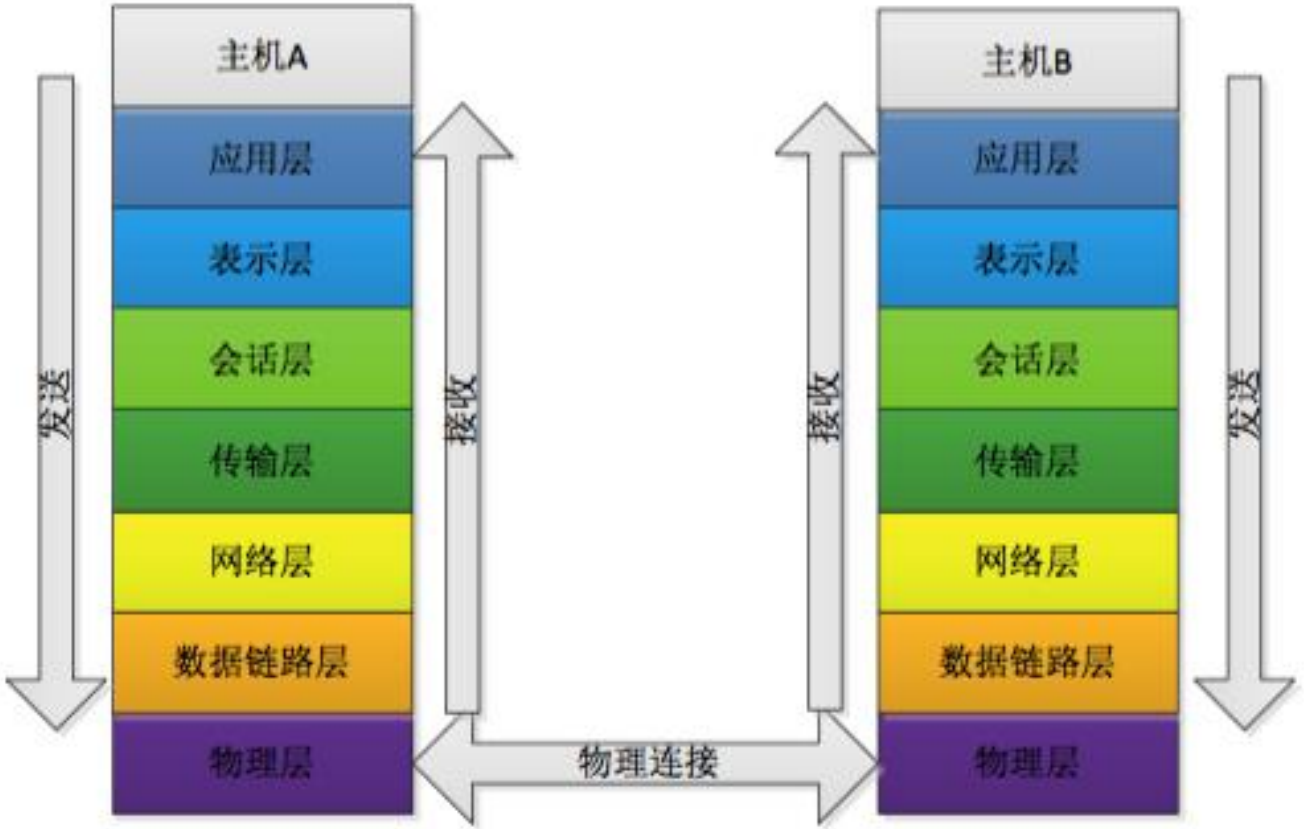
2。给定一个WSDL XSD文件（test.xsd)，自动生成JavaBean

3。创建jar文件，在另外的工程引用这个jar文件里面的类

4。Gson库的使用

5。使用okhttp库联系http协议客户端开发

为了使不同计算机厂家生产的计算机能够相互通信，以便在更大的范围内建立计算机网络，国际标准化组织（ISO）在1978年提出了“开放系统互联参考模型”，即著名的OSI/RM模型（Open System Interconnection/Reference Model）。它将计算机网络体系结构的通信协议划分为七层，自下而上依次为：物理层（Physics Layer）、数据链路层（Data Link Layer）、网络层（Network Layer）、传输层（Transport Layer）、会话层（Session Layer）、表示层（Presentation Layer）、应用层（Application Layer）。其中第四层完成数据传送服务，上面三层面向用户。



物理层：

物理层负责最后将信息编码成电流脉冲或其它信号用于网上传输；

eg：RJ45等将数据转化成0和1；

规定通信设备的机械的、电气的、功能的和过程的特性，用以建立、维护和拆除物理链路连接。具体地讲，机械 特性规定了网络连接时所需接插件的规格尺寸、引脚数量和排列情况等；电气特性规定了在物理连接上传输bit流时线路上信号电平的大小、阻抗匹配、传输速率 距离限制等；功能特性是指对各个信号先分配确切的信号含义，即定义了DTE和DCE之间各个线路的功能；规程特性定义了利用信号线进行bit流传输的一组 操作规程，是指在物理连接的建立、维护、交换信息是，DTE和DCE双放在各电路上的动作系列。在这一层，数据的单位称为比特（bit）。属于物理层定义的典型规范代表包括：EIA/TIA RS-232、EIA/TIA RS-449、V.35、RJ-45等。

数据链路层:

数据链路层通过物理网络链路　供数据传输。不同的数据链路层定义了不同的网络和协议特征,其中包括物理编址、网络拓扑结构、错误校验、数据帧序列以及流控;

可以简单的理解为：规定了0和1的分包形式，确定了网络数据包的形式；

在物理层提供比特流服务的基础上，建立相邻结点之间的数据链路，通过差错控制提供数据帧（Frame）在信道上无差错的传输，并进行各电路上的动作系列。数据链路层在不可靠的物理介质上提供可靠的传输。该层的作用包括：物理地址寻址、数据的成帧、流量控制、数据的检错、重发等。在这一层，数据的单位称为帧（frame）。数据链路层协议的代表包括：SDLC、HDLC、PPP、STP、帧中继等。

网络层

网络层负责在源和终点之间建立连接;

可以理解为，此处需要确定计算机的位置，怎么确定？IPv4，IPv6！

在 计算机网络中进行通信的两个计算机之间可能会经过很多个数据链路，也可能还要经过很多通信子网。网络层的任务就是选择合适的网间路由和交换结点， 确保数据及时传送。网络层将数据链路层提供的帧组成数据包，包中封装有网络层包头，其中含有逻辑地址信息- -源站点和目的站点地址的网络地址。如 果你在谈论一个IP地址，那么你是在处理第3层的问题，这是“数据包”问题，而不是第2层的“帧”。IP是第3层问题的一部分，此外还有一些路由协议和地 址解析协议（ARP）。有关路由的一切事情都在这第3层处理。地址解析和路由是3层的重要目的。网络层还可以实现拥塞控制、网际互连等功能。在这一层，数据的单位称为数据包（packet）。网络层协议的代表包括：IP、IPX、RIP、OSPF等。

传输层

传输层向高层　提供可靠的端到端的网络数据流服务。

可以理解为：每一个应用程序都会在网卡注册一个端口号，该层就是端口与端口的通信！常用的（TCP／IP）协议；

第4层的数据单元也称作数据包（packets）。但是，当你谈论TCP等具体的协议时又有特殊的叫法，TCP的数据单元称为段 （segments）而UDP协议的数据单元称为“数据报（datagrams）”。这个层负责获取全部信息，因此，它必须跟踪数据单元碎片、乱序到达的 数据包和其它在传输过程中可能发生的危险。第4层为上层提供端到端（最终用户到最终用户）的透明的、可靠的数据传输服务。所为透明的传输是指在通信过程中 传输层对上层屏蔽了通信传输系统的具体细节。传输层协议的代表包括：TCP、UDP、SPX等。

会话层

会话层建立、管理和终止表示层与实体之间的通信会话；

建立一个连接（自动的手机信息、自动的网络寻址）；

在会话层及以上的高层次中，数据传送的单位不再另外命名，而是统称为报文。会话层不参与具体的传输，它提供包括访问验证和会话管理在内的建立和维护应用之间通信的机制。如服务器验证用户登录便是由会话层完成的。

表示层:

表示层　供多种功能用于应用层数据编码和转化,以确保以一个系统应用层发送的信息 可以被另一个系统应用层识别;

可以理解为：解决不同系统之间的通信，eg：Linux下的QQ和Windows下的QQ可以通信；

这一层主要解决拥护信息的语法表示问题。它将欲交换的数据从适合于某一用户的抽象语法，转换为适合于OSI系统内部使用的传送语法。即提供格式化的表示和转换数据服务。数据的压缩和解压缩， 加密和解密等工作都由表示层负责。

应用层:

OSI 的应用层协议包括文件的传输、访问及管理协议(FTAM) ,以及文件虚拟终端协议(VIP)和公用管理系统信息(CMIP)等;

规定数据的传输协议；

应用层为操作系统或网络应用程序提供访问网络服务的接口。应用层协议的代表包括：Telnet、FTP、HTTP、SNMP等。

常见的应用层协议：

协议	端口	说明
HTTP	80	超文本传输协议
HTTPS	443	HTTP+SSL,HTTP的安全版
FTP	20,21,990	文件传输协议
POP3	110	邮局协议
SMTP	25	简单邮件传输协议
telnet	23	远程终端协议

开放式系统互联（OSI）模型与TCP/IP协议有什么区别？

开放式系统互联模型是一个参考标准，解释协议相互之间应该如何相互作用。TCP/IP协议是美国国防部发明的，是让互联网成为了目前这个样子的标准之一。开放式系统互联模型中没有清楚地描绘TCP/IP协议，但是在解释TCP/IP协议时很容易想到开放式系统互联模型。两者的主要区别如下：

TCP/IP协议中的应用层处理开放式系统互联模型中的第五层、第六层和第七层的功能。

TCP/IP协议中的传输层并不能总是保证在传输层可靠地传输数据包，而开放式系统互联模型可以做到。TCP/IP协议还提供一项名为UDP（用户数据报协议）的选择。UDP不能保证可靠的数据包传输。

TCP/UDP协议

TCP(Transmission Control Protocol)和UDP(User Datagram Protocol)协议属于传输层协议。其中TCP提供IP环境下的数据可靠传输，它提供的服务包括数据流传送、可靠性、有效流控、全双工操作和多路复用。通过面向连接、端到端和可靠的数据包发送。通俗说，它是事先为所发送的数据开辟出连接好的通道，然后再进行数据发送；而UDP则不为IP提供可靠性、流控或差错恢复功能。一般来说，TCP对应的是可靠性要求高的应用，而UDP对应的则是可靠性要求低、传输经济的应用。

TCP支持的应用协议主要有：Telnet、FTP、SMTP等；UDP支持的应用层协议主要有：NFS（网络文件系统）、SNMP（简单网络管理协议）、DNS（主域名称系统）、TFTP（通用文件传输协议）等。

TCP/IP协议与低层的数据链路层和物理层无关，这也是TCP/IP的重要特点。

IP协议(Internet Protocol)又称互联网协议，是支持网间互连的数据报协议，它与TCP协议（传输控制协议）一起构成了TCP/IP协议族的核心。它提供网间连接的完善功能， 包括IP数据报规定互连网络范围内的IP地址格式。

Internet 上，为了实现连接到互联网上的结点之间的通信，必须为每个结点（入网的计算机）分配一个地址，并且应当保证这个地址是全网唯一的，这便是IP地址。

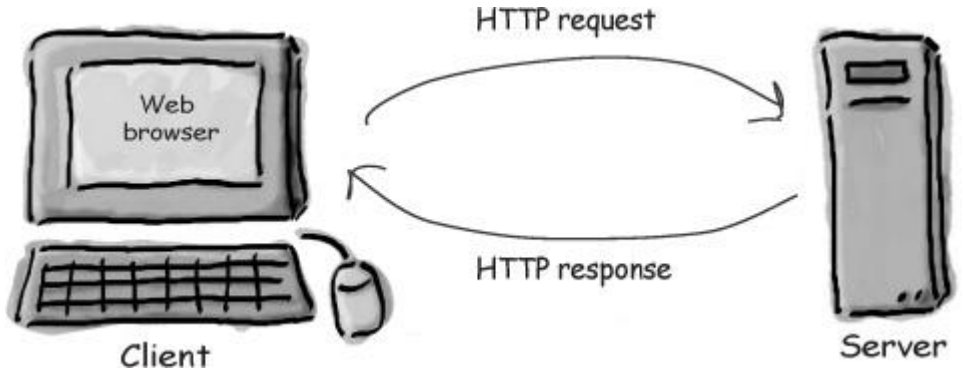
目前的IP地址（IPv4：IP第4版本）由32个二进制位表示，每8位二进制数为一个整数，中间由小数点间隔，如159.226.41.98，整个IP地址空间有4组8位二进制数，由表示主机所在的网络的地址（类似部队的编号）以及主机在该网络中的标识（如同士兵在该部队的编号）共同组成。

HTTP协议

HTTP 协议是互联网的基础协议，也是网页开发的必备知识。HTTP协议是Hyper Text Transfer Protocol（超文本传输协议）的缩写,是用于从万维网（WWW:World Wide Web ）服务器传输超文本到本地浏览器的传送协议。

HTTP是一个基于TCP/IP通信协议来传递数据（HTML 文件, 图片文件, 查询结果等）。

http协议规定了客户端和服务器之间的数据传输格式.



http优点:

简单快速:

http协议简单,通信速度很快;

灵活:

http协议允许传输任意类型的数据;

短连接:

http协议限制每次连接只处理一个请求,服务器对客户端的请求作出响应后,马上断开连接.这种方式可以节省传输时间.

主要特点

1、简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。

2、灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。

3.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

4.无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

HTTP之URL

HTTP使用统一资源标识符（Uniform Resource Identifiers, URI）来传输数据和建立连接。URL是一种特殊类型的URI，包含了用于查找某个资源的足够的信息，URL,全称是UniformResourceLocator, 中文叫统一资源定位符,是互联网上用来标识某一处资源的地址。以下面这个URL为例，介绍下普通URL的各部分组成：

http://www.google.com:8080/news/index.asp?boardID=5&ID=24618&page=1#name

从上面的URL可以看出，一个完整的URL包括以下几部分：

1.协议部分：该URL的协议部分为“http：”，这代表网页使用的是HTTP协议。在Internet中可以使用多种协议，如HTTP，FTP等等本例中使用的是HTTP协议。在"HTTP"后面的“//”为分隔符

2.域名部分：该URL的域名部分为“www.google.com”。一个URL中，也可以使用IP地址作为域名使用

3.端口部分：跟在域名后面的是端口，域名和端口之间使用“:”作为分隔符。端口不是一个URL必须的部分，如果省略端口部分，将采用默认端口

4.虚拟目录部分：从域名后的第一个“/”开始到最后一个“/”为止，是虚拟目录部分。虚拟目录也不是一个URL必须的部分。本例中的虚拟目录是“/news/”

5.文件名部分：从域名后的最后一个“/”开始到“？”为止，是文件名部分，如果没有“?”,则是从域名后的最后一个“/”开始到“#”为止，是文件部分，如果没有“？”和“#”，那么从域名后的最后一个“/”开始到结束，都是文件名部分。本例中的文件名是“index.asp”。文件名部分也不是一个URL必须的部分，如果省略该部分，则使用默认的文件名

6.锚部分：从“#”开始到最后，都是锚部分。本例中的锚部分是“name”。锚部分也不是一个URL必须的部分

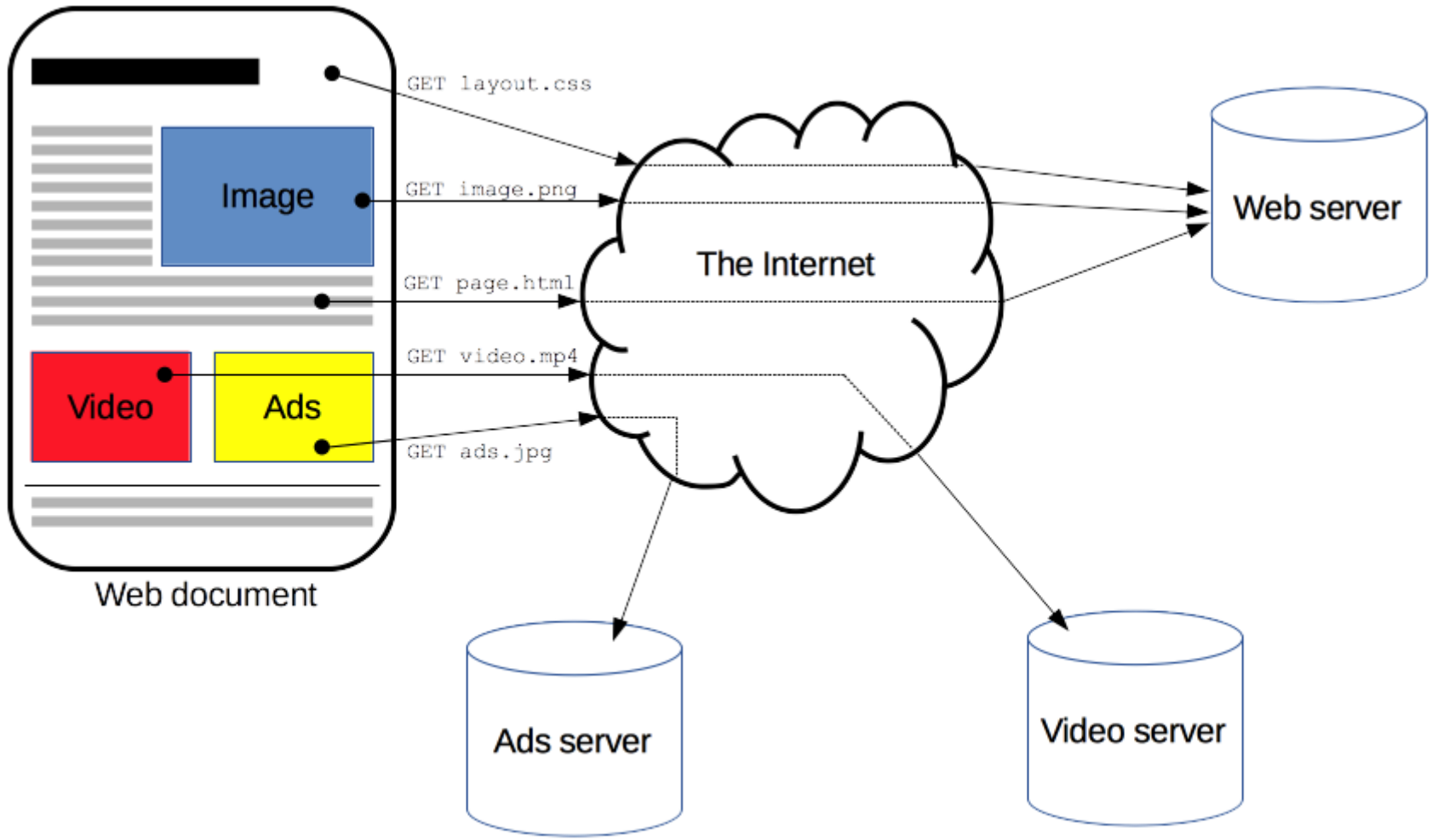
7.参数部分：从“？”开始到“#”为止之间的部分为参数部分，又称搜索部分、查询部分。本例中的参数部分为“boardID=5&ID=24618&page=1”。参数可以允许有多个参数，参数与参数之间用“&”作为分隔符。

两台计算机在使用HTTP通信在一条线路上的必须是一端为客户端，一端为服务器；

HTTP协议规定请求从客户端发出，最后服务器端响应该请求并返回；

HTTP是不保存状态，即无状态协议，于是为了实现保持状态功能引入了Cookie技术；

HTTP是一种能够获取如HTML这样网络资源的协议。它是Web上数据交换的基础，是一种client-server协议，也就是说请求通常是由像浏览器这样的接受方发起的。一个完整的web文档是由不同的子文档重新组建而成的，像是文本、布局描述、图片、视频、脚本等等。


GET layout.css
Image
GET image.png
GET page.html
Video
Ads
GET video.mp4
GET ads.jpg
Web document
The Internet
Web server
Ads server
Video server

http协议的使用

请求:客户端向服务器索要数据.

http协议规定:一个完整的http请求包含'请求行','请求头','请求体'三个部分;

请求行：包含了请求方法,请求资源路径,http协议版本. "GET /resources/images/ HTTP/1.1"

请求头:包含了对客户端的环境描述,客户端请求的主机地址等信息.

Accept: text/html (客户端所能接收的数据类型)

Accept-Language: zh-cn (客户端的语言环境)

Accept-Encoding: gzip(客户端支持的数据压缩格式)

Host: m.baidu.com(客户端想访问的服务器主机地址)

User-Agent: Mozilla/5.0(Macintosh;Intel Mac OS X10.10 rv:37.0) Gecko/20100101Firefox/37.0(客户端的类型,客户端的软件环境)

请求体:客户端发给服务器的具体数据,比如文件/图片等

GET / HTTP/1.0

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5)

Accept: */*

响应:服务器返回客户端想要的数据
http协议规定:一个完整的http响应包含'状态行','响应头','实体内容'三个部分;

状态行:包含了http协议版本,状态吗,状态英文名称.
"HTTP/1.1 200 OK"
响应头:包含了对服务器的描述,对返回数据的描述.
Content-Encoding: gzip(服务器支持的数据压缩格式) Content-Length: 1528(返回数据的长度)
Content-Type:application/xhtml+xml;charset=utf-8(返回数据的类型)
Date: Mon,15Jun201509:06:46GMT(响应的时间) Server: apache (服务器类型)
实体内容:服务器返回给客户端的具体数据(图片/html/文件...).

HTTP/1.0 200 OK
Content-Type: text/plain
Content-Length: 137582
Expires: Thu, 05 Dec 1997 16:00:00 GMT
Last-Modified: Wed, 5 August 1996 15:55:28 GMT
Server: Apache 0.84

<html>
  <body>Hello World</body>
</html>

关于字符的编码，1.0版规定，头信息必须是 ASCII 码，后面的数据可以是任何格式。因此，服务器回应的时候，必须告诉客户端，数据是什么格式，这就是Content-Type字段的作用。

下面是一些常见的Content-Type字段的值。

text/plain

text/html

text/css

image/jpeg

image/png

image/svg+xml

audio/mp4

video/mp4

application/javascript

application/pdf

application/zip

application/atom+xml

这些数据类型总称为MIME type，每个值包括一级类型和二级类型，之间用斜杠分隔。
除了预定义的类型，厂商也可以自定义类型。

application/vnd.debian.binary-package
上面的类型表明，发送的是Debian系统的二进制数据包。
MIME type还可以在尾部使用分号，添加参数。

Content-Type: text/html; charset=utf-8
上面的类型表明，发送的是网页，而且编码是UTF-8。
客户端请求的时候，可以使用Accept字段声明自己可以接受哪些数据格式。

Accept: */*

http协议定义了很多方法对应不同的资源操作,其中最常用的是GET和POST方法。

在请求URL后面以?的形式跟上发给服务器的参数,参数以"参数名"="参数值"的形式拼接,多个参数之间用&分隔;
GET的本质是从服务器得到数据,效率更高.并且GET请求可以被缓存.
注意:GET的长度是有限制的,不同的浏览器有不同的长度限制,一般在2~8K之间;

POST的本质是向服务器发送数据,也可以获得服务器处理之后的结果,效率不如GET.POST请求不可以被缓存,每次刷新之后都需要重新提交表单.
发送给服务器的参数全部放在'请求体'中;
理论上,POST传递的数据量没有限制.
注意:所有涉及到用户隐私的数据(密码/银行卡号等...)都要用POST的方式传递.

在一个网络中。传输数据需要面临三个问题:

1.客户端如何知道所求内容的位置？

2.当客户端知道所求内容的位置后，如何获取所求内容？

3.所求内容以何种形式组织以便被客户端所识别？

对于WEB来说，回答上面三种问题分别采用三种不同的技术，分别为:统一资源定位符(URIs),超文本传输协议(HTTP)和超文本标记语言(HTML)。

学习web前端开发基础技术需要掌握：HTML、CSS、JavaScript语言。

1. HTML是网页内容的载体。内容就是网页制作者放在页面上想要让用户浏览的信息，可以包含文字、图片、视频等。

2. CSS样式是表现(外观控制)。就像网页的外衣。比如，标题字体、颜色变化，或为标题加入背景图片、边框等。所有这些用来改变内容外观的东西称之为表现。

3. JavaScript是用来实现网页上的特效效果。如：鼠标滑过弹出下拉菜单。或鼠标滑过表格的背景颜色改变。还有焦点新闻（新闻图片）的轮换。可以这么理解，有动画的，有交互的一般都是用JavaScript来实现的。

HTML文件是什么？

HTML表示超文本标记语言（Hyper Text Markup Language）。

HTML文件是一个包含标记的文本文件。

这些标记保速浏览器怎样显示这个页面。

HTML文件必须有htm或者html扩展名。

HTML文件可以用一个简单的文本编辑器创建。

HTML是用于创建网页的语言。我们通过使用HTML标记标签创建html文档来创建网页。HTML代表超文本标记语言。HTML是一种标记语言，它具有标记标签的集合。

HTML(HyperText MarkUp Language)超文本标记语言,通过使用标记来描述文档结构和表现形式的一种语言,由浏览器进行解析,然后把结果显示在网页上. 它是网页构成的基础,你见到的所有网页都离不开HTML,所以学习HTML是基础中的基础.

HTML标签是由尖括号（如<html>， <body>)包围的字词。标签通常成对出现，例如<html>和</html>。

一对中的第一个标签是开始标签;第二个标签是结束标签。在上面的示例中，<html>是开始标签，而</html>是结束标签。

我们还可以将开始标签称为起始标签，结束标签称为闭合标签。

HTML 是由各种各样的标签组成，学习 HTML就是学习使用这些标签。


```
1  <html>
2      <head>
3          <title>网页标题</title> <!--这是注释，在网页中不显示-->
4      </head>
5      <body>
6          <h1>内容标题（HTML标签演示页面）</h1>
7          <hr />
8          <p>这里是<b>具体</b>的内容，P标签表示是段落</p>
9          <a href='http://www.adminwang.com'>链接演示</a>
10     </body>
11 </html>
12
```

HTML标签是HTML文档的基本元素,它一般是成对出现的,即有开始标签和对应的结束标签构成. 如 <body></body> <head></head>等,但有些是特殊的单标签,如
 <hr />等. Html HTML语言是弱类型语言,标签不区分大小写<P>和 显示结果是相同的,不过标准推荐使用小写.

l 标签是HTML中最基本单位,也是最重要组成部分。

l HTML标签由开始标签和结束标签组成。

l 某些HTML元素没有结束标签，比如

l 标签是大小写无关的,<body>跟<BODY>表示意思是一样的，标准推荐使用小写.

l 所有的标签之间可以嵌套。例：<head> <title>标签嵌套演示</title></head>

```
<html>
<head>
<title>Title of page</title>
</head>
<body>
This is my first homepage.
<b>This text is bold</b>
</body>
</html>
```

HTML文档中，第一个标签是<html>。这个标签告诉浏览器这是HTML文档的开始。HTML文档的最后一个标签是</html>，这个标签告诉浏览器这是HTML文档的终止。

在<head>和</head>标签之间文本的是头信息。在浏览器窗口中，头信息是不被显示的。

在<title>和</title>标签之间的文本是文档标题，它被显示在浏览器窗口的标题栏。

在<body>和</body>标签之间的文本是正文，会被显示在浏览器中。

在和标签之间的文本会以加粗字体显示。

HTML 标题

HTML 标题（Heading）是通过<h1> - <h6> 标签来定义的.

<h1>这是一个标题</h1>

<h2>这是一个标题</h2>

<h3>这是一个标题</h3>

HTML 段落

HTML 段落是通过标签 来定义的.

这是一个段落。

这是另外一个段落。

HTML 链接

HTML 链接是通过标签 <a> 来定义的.

实例

这是一个链接

提示:在 href 属性中指定链接的地址。

HTML 图像

HTML 图像是通过标签 <img> 来定义的.

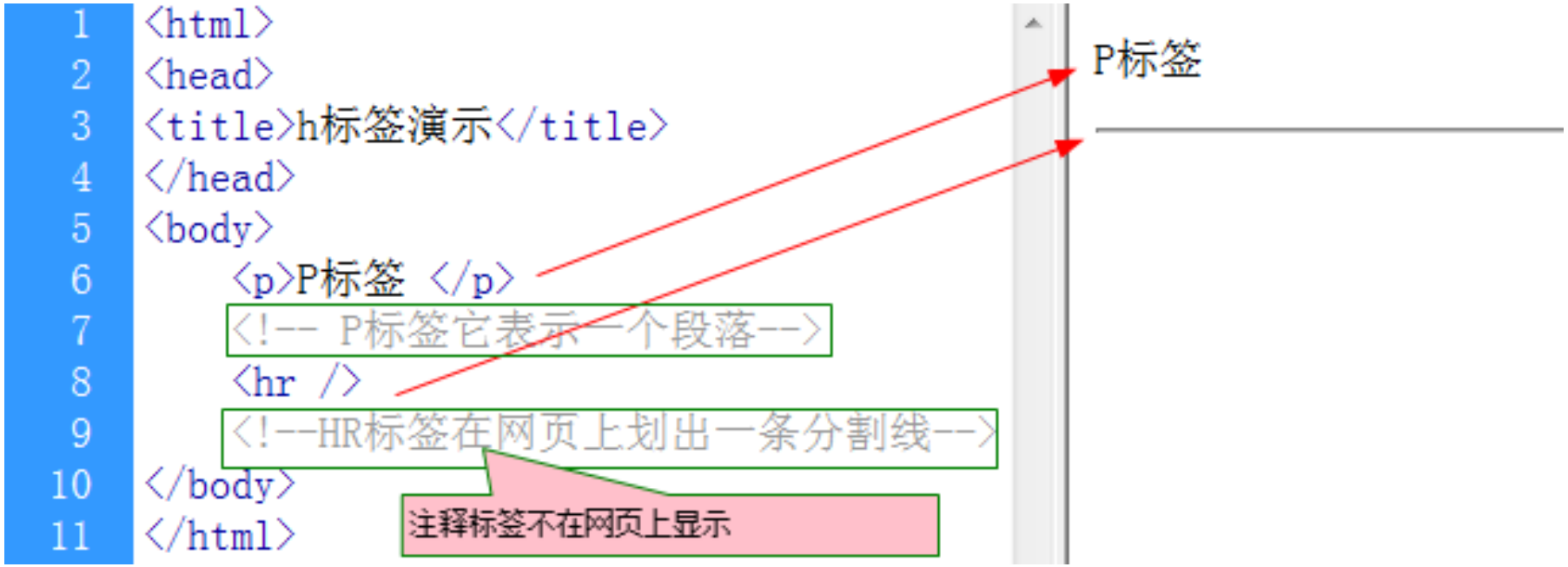
<img src="flyers.png" width="104" height="142">

在实际开发中需要在要在一些代码段做HTML注释,这样做的好处很多,比如:方便查找,方便比对,方便项目组里的其它程序员了解你的代码,而且可以方便以后你对自己代码的理解与修改等等

语法格式：

<!—这里写注释内容 -->

注释：开始括号之后（左边的括号）需要紧跟一个叹号，结束括号之前（右边的括号）不需要。



HTML文档在浏览器里通常是从左到右，从上到下地显示的，到了窗口右边就自动换行。为了实现分栏的效果，很多人使用表格（<table>）进行页面排版（虽然HTML里提供表格的本意不是为了排版）。

<table>标签里通常会包含几个<tr>标签，<tr>代表表格里的一行。<tr>标签又会包含<td>标签，每个<td>代表一个单元格。

点击查看效果

```
<table>
 <tr>
  <td>2000</td><td>悉尼</td>
 </tr>
 <tr>
  <td>2004</td><td>雅典</td>
 </tr>
 <tr>
  <td>2008</td><td>北京</td>
 </tr>
</table>
```

HTML 是由各种各样的标签组成，学习 HTML就是学习使用这些标签。

作业

http://www.halifax.ca/newcomers/WelcomingNewcomers.php#Holidays

抓取出ns省的节日

I/O 问题是任何编程语言都无法回避的问题，可以说 I/O 问题是整个人机交互的核心问题，因为 I/O 是机器获取和交换信息的主要渠道。在当今这个数据大爆炸时代，I/O 问题尤其突出，很容易成为一个性能瓶颈。

Java 的 I/O 操作类在包 java.io 下，大概有将近 80 个类，但是这些类大概可以分成四组，分别是：

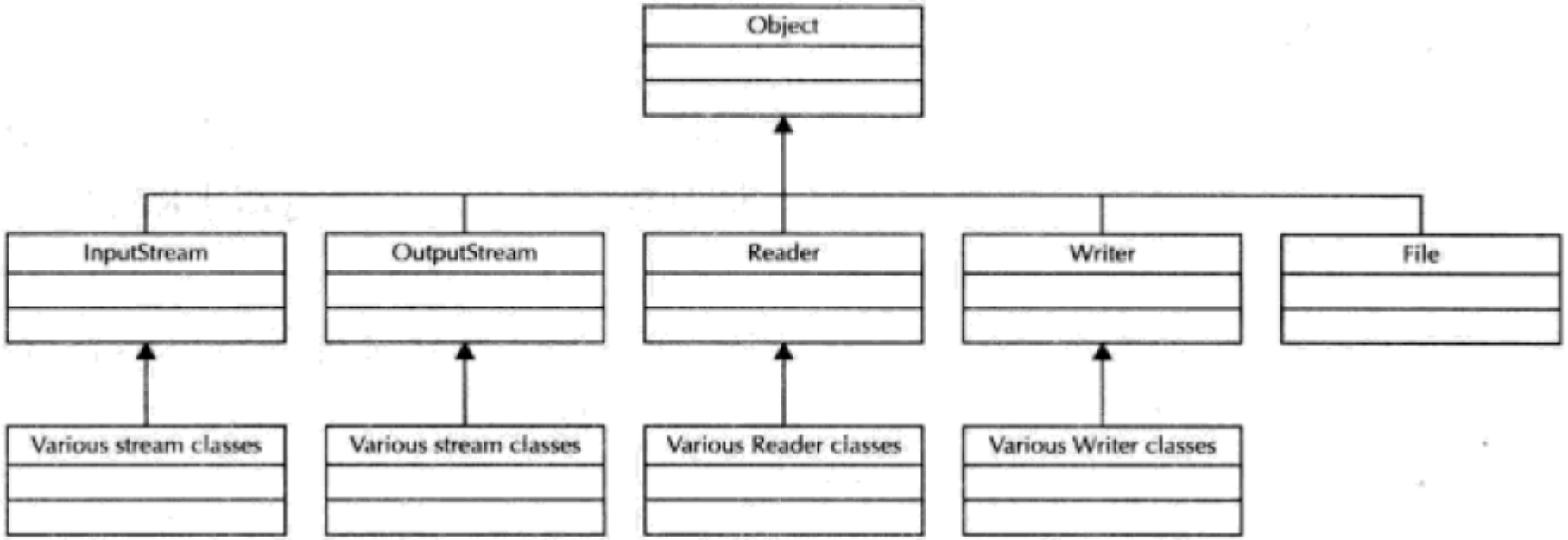
基于字节操作的 I/O 接口：InputStream 和 OutputStream

基于字符操作的 I/O 接口：Writer 和 Reader

基于磁盘操作的 I/O 接口：File

基于网络操作的 I/O 接口：Socket

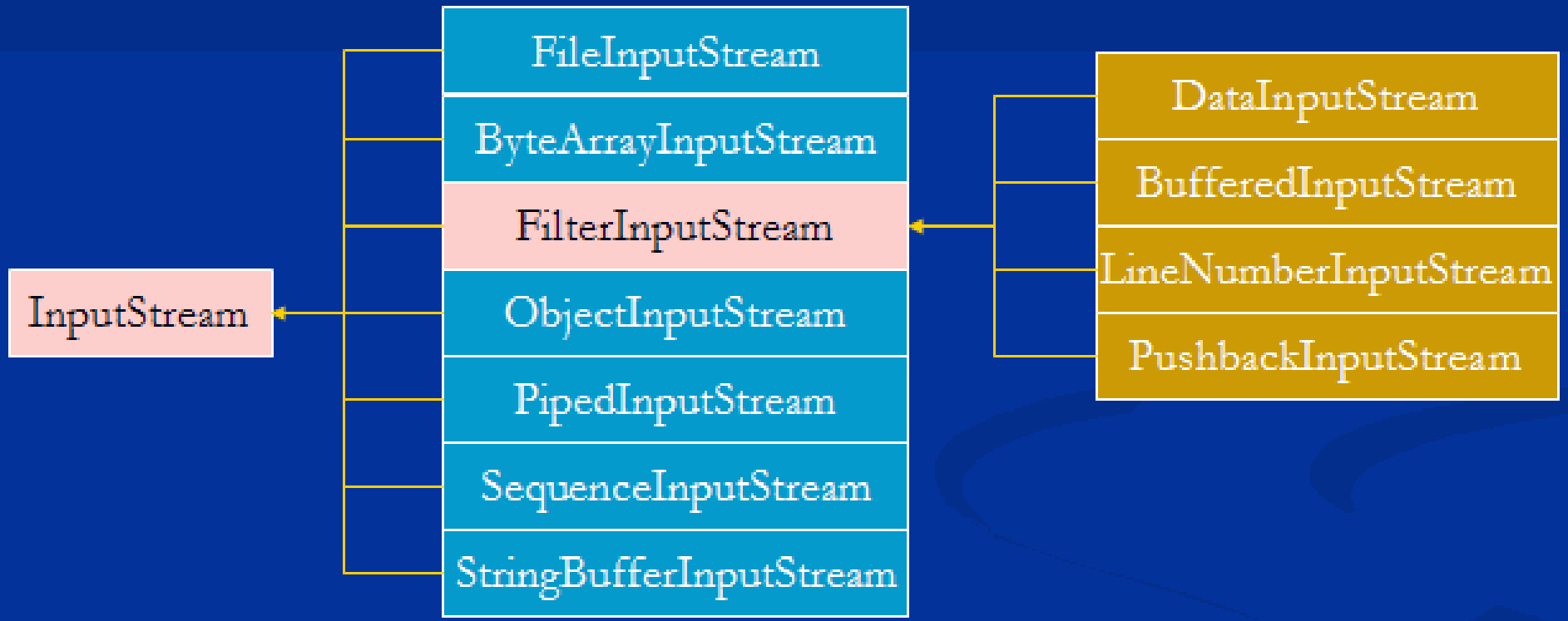
前两组主要是根据传输数据的数据格式，后两组主要是根据传输数据的方式，虽然 Socket 类并不在 java.io 包下



Java中所有的类都继承自Object类，所以各种IO类也不例外。InputStream和OutputStream类操作字节数据。Reader和Writer类工作在字符是上。File类为文件提供接口。

我们很有必要了解IO类的演变历史，如果缺乏历史的眼光，那么我们对什么时候该使用那些类，以及什么时候不该使用那些类而感到迷惑。
JDK1.0的时候，所有与输入相关的类都继承于InputStream，所有与输出相关的类都继承于OutputStream。
JDK1.1的时候，增加了面向字符的IO类，包括Reader和Writer。

java.io包中 InputStream的类层次



字节流

字节流，顾名思义，以字节为单位进行IO操作。字节流最常用的方法是read和write方法。

read方法是从流中读取字节并递增文件指针到一下个位置。在字节流的读取过程中，我们要明白一个重要的概念：文件指针。文件指针指示了流中的当前位置。当文件指针到达文件的末尾时候，读取操作返回-1给调用者。read方法不带任何参数，每次只能读取一个字节。read方法的返回值是读取的字节，并转化为int类型返回。如下所示：

```
FileInputStream reader =new FileInputStream(“filename”);//以字节流格式打开文件

int result =reader.read();
```

read方法还有其他的形式：

int read(byte[] b)，读出的数据被存储在字节数组中，返回读取的字节数，如果提前到达文件的末尾，返回值可能小于数组的长度。

int read(byte[] b, int off, int len)，第一个参数指定数据被存储的字节数组，第二个参数off指定读取的第一个字节将存储在字节数组的偏移量，第三个参数len指定要读取的字节数。

上面三种形式的读取都是直接从文件流里读取的，每次读取只能是一个一个字节的读取。需要说明的是每次读取单个字节或512个字节，需要的io数量都是相同的。为了提高效率，人们设计出了缓存的读取方式：

```
FileInputStream reader =new FileInputStream("filename");
BufferedInputStream bs = new BufferedInputStream(reader);
bs.read();
```

虽然上面仍然读取的是一个字节数据，但是因为缓存的存在，效率已经大大提高了。底层的实现是这样的：BufferedInputStream的实现中有一个用于存储数据的内部缓冲区数组：protected volatile byte[] buf。这个缓冲区数组的作用在于对源进行数据块访问，而不是一字节一字节的访问，也就是进行一次I/O将一块数据存到缓冲区中，再从缓冲区中read，当缓冲区为空时再重新读新的数据块。

注意：

一个是操作数据的方式是可以组合使用的，如这样组合使用

```
OutputStream out = new BufferedOutputStream(new ObjectOutputStream(new FileOutputStream("fileName"))；
```

还有一点是流最终写到什么地方必须要指定，要么是写到磁盘要么是写到网络中

不管是磁盘还是网络传输，最小的存储单元都是字节，而不是字符，所以 I/O 操作的都是字节而不是字符，但是为啥有操作字符的 I/O 接口呢？这是因为我们的程序中通常操作的数据都是以字符形式，为了操作方便当然要提供一个直接写字符的 I/O 接口，如此而已。我们知道字符到字节必须要经过编码转换，而这个编码又非常耗时，而且还会经常出现乱码问题，所以 I/O 的编码问题经常是让人头疼的问题。

字符流

字符流是操作字符文件的。一般情况下我们的使用方式是：

```
FileReader reader =new FileReader("filename");//以字符格式打开文件
BufferedReader buffer = new BufferedReader(reader);//建立缓存
buffer.readLine();//读取缓存中的一行
```

```
/* 读入TXT文件 */
        String pathname = "D:\\twitter\\13_9_6\\dataset\\en\\input.txt"; // 绝对路径或相对路径都可以，这里是绝对路径，写入文件时演示相对路径
        File filename = new File(pathname); // 要读取以上路径的input。txt文件
        InputStreamReader reader = new InputStreamReader(
            new FileInputStream(filename)); // 建立一个输入流对象reader
        BufferedReader br = new BufferedReader(reader); // 建立一个对象，它把文件内容转成计算机能读懂的语言
        String line = "";
        line = br.readLine();
        while (line != null) {
           line = br.readLine(); // 一次读入一行数据
        }

        /* 写入Txt文件 */
        File writename = new File(".\\result\\en\\output.txt"); // 相对路径，如果没有则要建立一个新的output。txt文件
        writename.createNewFile(); // 创建新文件
        BufferedWriter out = new BufferedWriter(new FileWriter(writename));
        out.write("我会写入文件啦\r\n"); // \r\n即为换行
        out.flush(); // 把缓存区内容压入文件
        out.close(); // 最后记得关闭文件
```

协议简介

协议相当于相互通信的程序间达成的一种约定，它规定了分组报文的结构、交换方式、包含的意义以及怎样对报文所包含的信息进行解析。

TCP/IP 协议族有 IP 协议、TCP 协议和 UDP 协议。

TCP 协议和 UDP 协议使用的地址叫做端口号，用来区分同一主机上的不同应用程序。TCP 协议和 UDP 协议也叫端到端传输协议，因为他们将数据从一个应用程序传输到另一个应用程序，而 IP 协议只是将数据从一个主机传输到另一个主机。

在 TCP/IP 协议中，有两部分信息用来确定一个指定的程序：互联网地址和端口号：其中互联网地址由 IP 协议使用，而附加的端口地址信息则由传输协议（TCP 或 UDP 协议）对其进行解析。

现在 TCP/IP 协议族中的主要 socket 类型为流套接字（使用 TCP 协议）和数据报套接字（使用 UDP 协议），其中通过数据报套接字，应用程序一次只能发送最长 65507 个字节长度的信息。

一个 TCP/IP 套接字由一个互联网地址，一个端对端协议（TCP 协议或 UDP 协议）以及一个端口号唯一确定。

TCP与UDP区别：

TCP特点：

1、TCP是面向连接的协议，通过三次握手建立连接，通讯完成时要拆除连接，由于TCP是面向连接协议，所以只能用于点对点的通讯。而且建立连接也需要消耗时间和开销。

2、TCP传输数据无大小限制，进行大数据传输。

3、TCP是一个可靠的协议，它能保证接收方能够完整正确地接收到发送方发送的全部数据。

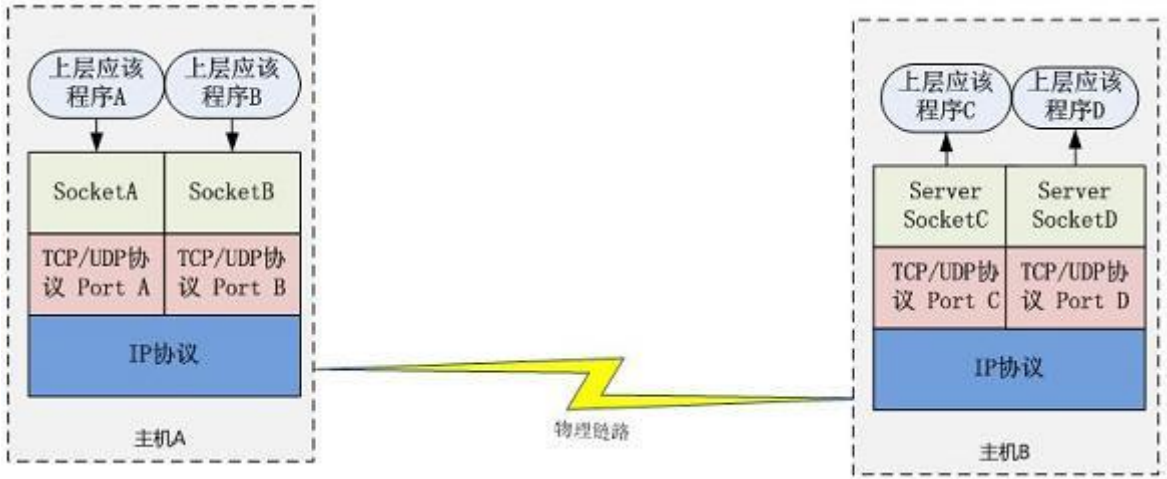
UDP特点：

1、UDP是面向无连接的通讯协议，UDP数据包括目的端口号和源端口号信息，由于通讯不需要连接，所以可以实现广播发送。

2、UDP传输数据时有大小限制，每个被传输的数据报必须限定在64KB之内。

3、UDP是一个不可靠的协议，发送方所发送的数据报并不一定以相同的次序到达接收方。

主机 A 的应用程序要能和主机 B 的应用程序通信，必须通过 Socket 建立连接，而建立 Socket 连接必须需要底层 TCP/IP 协议来建立 TCP 连接。建立 TCP 连接需要底层 IP 协议来寻址网络中的主机。我们知道网络层使用的 IP 协议可以帮助我们根据 IP 地址来找到目标主机，但是一台主机上可能运行着多个应用程序，如何才能与指定的应用程序通信就要通过 TCP 或 UPD 的地址也就是端口号来指定。这样就可以通过一个 Socket 实例唯一代表一个主机上的一个应用程序的通信链路了。



Socket通常也称作"套接字"，用于描述IP地址和端口，是一个通信链的句柄。网络上的两个程序通过一个双向的通讯连接实现数据的交换，这个双向链路的一端称为一个Socket，一个Socket由一个IP地址和一个端口号唯一确定。应用程序通常通过"套接字"向网络发出请求或者应答网络请求。 Socket是TCP/IP协议的一个十分流行的编程界面，但是，Socket所支持的协议种类也不光TCP/IP一种，因此两者之间是没有必然联系的。在Java环境下，Socket编程主要是指基于TCP/IP协议的网络编程。

Socket通讯过程：服务端监听某个端口是否有连接请求，客户端向服务端发送连接请求，服务端收到连接请求向客户端发出接收消息，这样一个连接就建立起来了。客户端和服务端都可以相互发送消息与对方进行通讯。

Socket的基本工作过程包含以下四个步骤：

1、创建Socket；

2、打开连接到Socket的输入输出流；

3、按照一定的协议对Socket进行读写操作；

4、关闭Socket。

Socket可以说是一种针对网络的抽象，应用通过它可以来针对网络读写数据。就像通过一个文件的file handler就可以都写数据到存储设备上一样。根据TCP协议和UDP协议的不同，在网络编程方面就有面向两个协议的不同socket，一个是面向字节流的一个是面向报文的。

对socket的本身组成倒是比较好理解。既然是应用通过socket通信，肯定就有一个服务器端和一个客户端。所以它必然就包含有一个对应的IP地址。另外，在这个地址上server要提供一系列的服务，于是就需要有一系列对应的窗口来提供服务。所以就有一个对应的端口号(Port)。端口号是一个16位的二进制数字，那么范围就是从（0-65535）。IP地址加端口号基本上就构成了socket。

在java.net包下有两个类：Socket和ServerSocket。ServerSocket用于服务器端，Socket是建立网络连接时使用的。在连接成功时，应用程序两端都会产生一个Socket实例，操作这个实例，完成所需的会话。对于一个网络连接来说，套接字是平等的，并没有差别，不因为在服务器端或在客户端而产生不同级别。

TCP 协议提供面向连接的服务，通过它建立的是可靠地连接。Java 为 TCP 协议提供了两个类：Socke 类和 ServerSocket 类。一个 Socket 实例代表了 TCP 连接的一个客户端，而一个 ServerSocket 实例代表了 TCP 连接的一个服务器端，一般在 TCP Socket 编程中，客户端有多个，而服务器端只有一个，客户端 TCP 向服务器端 TCP 发送连接请求，服务器端的 ServerSocket 实例则监听来自客户端的 TCP 连接请求，并为每个请求创建新的 Socket 实例，由于服务端在调用 accept（）等待客户端的连接请求时会阻塞，直到收到客户端发送的连接请求才会继续往下执行代码，因此要为每个 Socket 连接开启一个线程。服务器端要同时处理 ServerSocket 实例和 Socket 实例，而客户端只需要使用 Socket 实例。另外，每个 Socket 实例会关联一个 InputStream 和 OutputStream 对象，我们通过将字节写入套接字的 OutputStream 来发送数据，并通过从 InputStream 来接收数据。

客户端向服务器端发送连接请求后，就被动地等待服务器的响应。典型的 TCP 客户端要经过下面三步操作：

创建一个 Socket 实例：构造函数向指定的远程主机和端口建立一个 TCP 连接；

通过套接字的 I/O 流与服务端通信；

使用 Socket 类的 close 方法关闭连接。

服务端的工作是建立一个通信终端，并被动地等待客户端的连接。

典型的 TCP 服务端执行如下两步操作：

创建一个 ServerSocket 实例并指定本地端口，用来监听客户端在该端口发送的 TCP 连接请求；

重复执行：

调用 ServerSocket 的 accept（）方法以获取客户端连接，并通过其返回值创建一个 Socket 实例；

为返回的 Socket 实例开启新的线程，并使用返回的 Socket 实例的 I/O 流与客户端通信； 通信完成后，使用 Socket 类的 close（）方法关闭该客户端的套接字连接。

```
public class ClientSocket {
  public static void main(String args[]) {
        String host = "127.0.0.1";
        int port = 8919;
        try {
          Socket client = new Socket(host, port);
          Writer writer = new OutputStreamWriter(client.getOutputStream());
          writer.write("Hello From Client");
          writer.flush();
          writer.close();
          client.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
    }

}
```

```
public class ServerClient {
  public static void main(String[] args) {
      int port = 8919;
      try {
         ServerSocket server = new ServerSocket(port);
            Socket socket = server.accept();
         Reader reader = new InputStreamReader(socket.getInputStream());
         char chars[] = new char[1024];
         int len;
         StringBuilder builder = new StringBuilder();
         while ((len=reader.read(chars)) != -1) {
           builder.append(new String(chars, 0, len));
         }
         System.out.println("Receive from client message=: " + builder);
         reader.close();
         socket.close();
         server.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
  }
}
```

```
public class Client1 {
    public static void main(String[] args) throws IOException {
        //客户端请求与本机在20006端口建立TCP连接
        Socket client = new Socket("127.0.0.1", 20006);
        client.setSoTimeout(10000);
        //获取键盘输入
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        //获取Socket的输出流，用来发送数据到服务端
        PrintStream out = new PrintStream(client.getOutputStream());
        //获取Socket的输入流，用来接收从服务端发送过来的数据
        BufferedReader buf =  new BufferedReader(new
InputStreamReader(client.getInputStream()));
```

```
boolean flag = true;
      while(flag){
         System.out.print("输入信息：");
         String str = input.readLine();
         //发送数据到服务端
         out.println(str);
         if("bye".equals(str)){
            flag = false;
         }else{
            try{
               //从服务器端接收数据有个时间限制（系统自设，也可以自己设置），超过了这个时
间，便会抛出该异常
               String echo = buf.readLine();
               System.out.println(echo);
            }catch(SocketTimeoutException e){
               System.out.println("Time out, No response");
            }
         }
      }
      input.close();
```

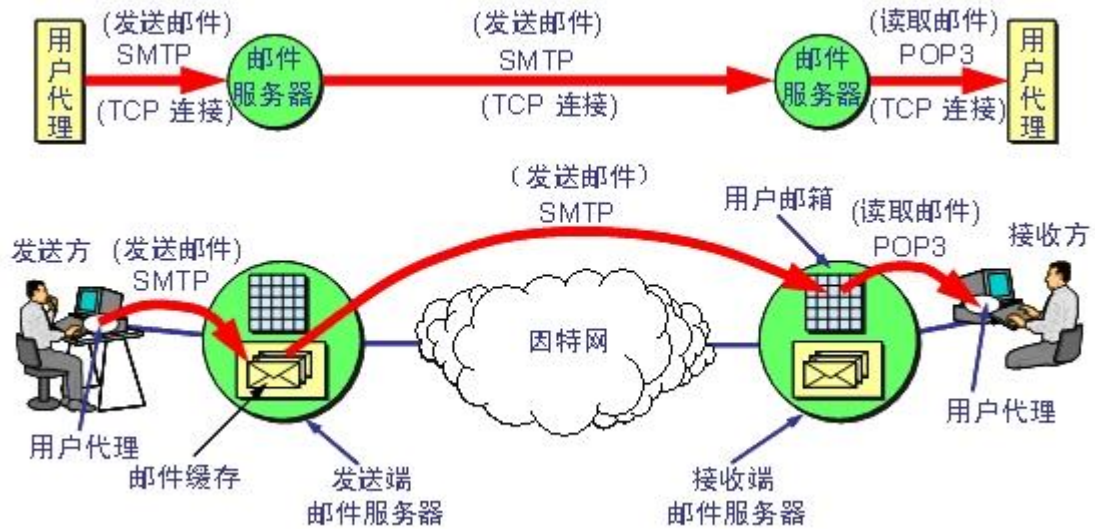
```
public class ServerThread implements Runnable {

    private Socket client = null;
    public ServerThread(Socket client){
        this.client = client;
    }

    @Override
    public void run() {
        try{
            //获取Socket的输出流，用来向客户端发送数据
            PrintStream out = new PrintStream(client.getOutputStream());
            //获取Socket的输入流，用来接收从客户端发送过来的数据
            BufferedReader buf = new BufferedReader(new
InputStreamReader(client.getInputStream()));
            boolean flag =true;
```

```
while(flag){
            //接收从客户端发送过来的数据
            String str =  buf.readLine();
            if(str == null || "".equals(str)){
               flag = false;
            }else{
               if("bye".equals(str)){
                  flag = false;
               }else{
                  //将接收到的字符串前面加上echo，发送到对应的客户端
                  out.println("echo:" + str);
               }
            }
         }
         out.close();
         client.close();
      }catch(Exception e){
         e.printStackTrace();
      }
   }

}
```

```
public class Server1 {
    public static void main(String[] args) throws Exception{
        //服务端在20006端口监听客户端请求的TCP连接
        ServerSocket server = new ServerSocket(20006);
        Socket client = null;
        boolean f = true;
        while(f){
            //等待客户端的连接，如果没有获取连接
            client = server.accept();
            System.out.println("与客户端连接成功！");
            //为每个客户端连接开启一个线程
            new Thread(new ServerThread(client)).start();
        }
        server.close();
    }
}
```

```
public class ClientSocket {
  public static void main(String args[]) {
        String host = "127.0.0.1";
        int port = 8919;
        try {
          Socket client = new Socket(host, port);
          Writer writer = new OutputStreamWriter(client.getOutputStream());
          writer.write("Hello From Client");
          writer.flush();
          writer.close();
          client.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
    }

}
```

SMTP，即简单邮件传送协议，所对应RFC文档为RFC821。同http等多数应用层协议一样，它工作在C/S模式下，用来实现因特网上的邮件传送。SMTP在整个电子邮件通信中所处的位置如图 1所示。



一个具体的SMTP通信（如发送端邮件服务器与接收端服务器的通信）的过程如下。

1)　发送端邮件服务器（以下简称客户端）与接收端邮件服务器（以下简称服务器）的25号端口建立TCP连接。

2)　客户端向服务器发送各种命令，来请求各种服务（如认证、指定发送人和接收人）。

3)　服务器解析用户的命令，做出相应动作并返回给客户端一个响应。

4)　2)和3)交替进行，直到所有邮件都发送完或两者的连接被意外中断。

从这个过程看出，命令和响应是SMTP协议的重点，下面将予以重点讲述。

SMTP的命令不多（14个），它的一般形式是：COMMAND [Parameter] <CRLF>。其中COMMAND是ASCII形式的命令名，Parameter是相应的命令参数，<CRLF>是回车换行符(0DH, 0AH)。

SMTP的响应也不复杂，它的一般形式是：XXX Readable Illustration。XXX是三位十进制数；Readable Illustration是可读的解释说明，用来表明命令是否成功等。XXX具有如下的规律：以2开头的表示成功，以4和5开头的表示失败，以3开头的表示未完成（进行中）。

SMTP命令不区分大小写，但参数区分大小写，有关这方面的详细说明请参考RFC821。常用的命令如下。

HELO <domain> <CRLF>。向服务器标识用户身份发送者能欺骗，说谎，但一般情况下服务器都能检测到。

MAIL FROM: <reverse-path> <CRLF>。<reverse-path>为发送者地址，此命令用来初始化邮件传输，即用来对所有的状态和缓冲区进行初始化。

RCPT TO：<forward-path> <CRLF>。 <forward-path>用来标志邮件接收者的地址，常用在MAIL FROM后，可以有多个RCPT TO。

DATA <CRLF>。将之后的数据作为数据发送，以<CRLF>.<CRLF>标志数据的结尾。

REST <CRLF>。重置会话，当前传输被取消。

NOOP <CRLF>。要求服务器返回OK应答，一般用作测试。

QUIT <CRLF>。结束会话。

VRFY <string> <CRLF>。验证指定的邮箱是否存在，由于安全方面的原因，服务器大多禁止此命令。

EXPN <string> <CRLF>。验证给定的邮箱列表是否存在，由于安全方面的原因，服务器大多禁止此命令。

HELP <CRLF>。查询服务器支持什么命令

常用的响应如下所示，数字后的说明是从英文译过来的。更详细的说明请参考RFC821。

501参数格式错误

502命令不可实现

503错误的命令序列

504命令参数不可实现

211系统状态或系统帮助响应

214帮助信息

220＜domain＞服务就绪

221＜domain＞服务关闭

421＜domain＞服务未就绪，关闭传输信道

250要求的邮件操作完成

251用户非本地，将转发向＜forward-path＞

450要求的邮件操作未完成，邮箱不可用

550要求的邮件操作未完成，邮箱不可用

451放弃要求的操作；处理过程中出错

551用户非本地，请尝试＜forward-path＞

452系统存储不足，要求的操作未执行

552过量的存储分配，要求的操作未执行

553邮箱名不可用，要求的操作未执行

354开始邮件输入，以"."结束

554操作失败

```
SSLSocket socket = (SSLSocket) ((SSLSocketFactory) SSLSocketFactory.getDefault())
.createSocket(“smtp.gmail.com”, 465);
OutputStream socketOut = socket.getOutputStream();
socketOut.write(("HELO " + localhost + "\r\n").getBytes());
socketOut.write(("AUTH LOGIN " + userName + "\r\n").getBytes());
socketOut.write((passWord + "\r\n").getBytes());
socketOut.write(("MAIL FROM:<" + email_from + ">" + "\r\n").getBytes());
socketOut.write(("RCPT TO:<" + email_to + ">" + "\r\n").getBytes());
socketOut.write(("DATA" + "\r\n").getBytes());
socketOut.write(("SUBJECT:" + email_subject + "\r\n").getBytes());
socketOut.write((email_body + "\r\n").getBytes());
socketOut.write(("." + "\r\n").getBytes());
socketOut.write(("QUIT" + "\r\n").getBytes());
```

Exception

Thread

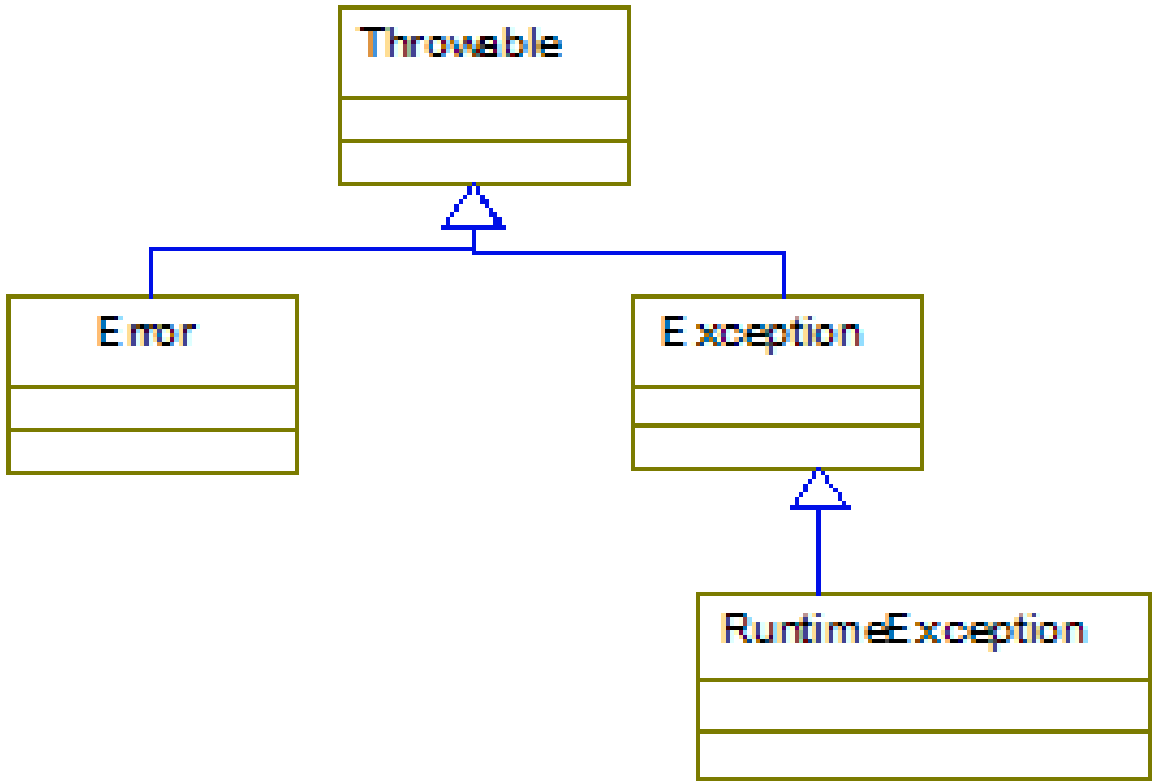
XML

JSON

MAVEN

- 在讲Java异常实践之前，先理解一下什么是异常。到底什么才算是异常呢？其实异常可以看做在我们编程过程中遇到的一些意外情况，当出现这些意外情况时我们无法继续进程正常的逻辑处理，此时我们就可以抛出一个异常。
- 广义的讲，抛出异常分三种不同的情况：
- **编程错误导致的异常**：在这个类别里，异常的出现是由于代码的错误（譬如NullPointerException、IllegalArgumentException、IndexOutOfBoundsException）。代码通常对编程错误没有什么对策，所以它一般是非检查异常。
- **客户端的错误导致的异常**：客户端代码试图违背制定的规则，调用API不支持的资源。如果在异常中显示有效信息的话，客户端可以采取其他的补救方法。例如：解析一个格式不正确的XML文档时会抛出异常，异常中含有有效的信息。客户端可以利用这个有效信息来采取恢复的步骤。
- **资源错误导致的异常**：当获取资源错误时引发的异常。例如，系统内存不足，或者网络连接失败。客户端对于资源错误的反应是视情况而定的。客户端可能一段时间之后重试或者仅仅记录失败然后将程序挂起。

在 Java 程序设计语言中，使用一种异常处理的错误捕获机制。当程序运行过程中发生一些异常情况，程序有可能被中断、或导致错误的结果出现。在这种情况下，程序不会返回任何值，而是抛出封装了错误信息的对象。Java 语言提供了专门的异常处理机制去处理这些异常。如图 1 所示为 Java 异常体系结构：



从上面异常继承树可以看出，所以异常都继承自Throwable，这也意味着所有异常都是可以抛出的。

具体来说，广义的异常可以分为Error和Exception两大类。

Error表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java虚拟机运行错误Virtual MachineError，当 JVM 不再有继续执行操作所需的内存资源时，将出现OutOfMemoryError，虚拟机错误还有StackOverflowError 、InternalError、 UnknownError等。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。经常见到的Error还有LinkageError（结合错误），具体有NoSuchMethodError 、IllegalAccessError 、NoClassDefFoundError。

所以，对于Error我们编程中基本是用不到的，也就是说我们在编程中可以忽略Error错误。所以我们通常所说的异常只的是Exception，而Exception可分为检查异常和非检查异常。

检查异常与非检查异常

通常我们所说的异常指的都是Exception的子类，它们具体可以分为两大类，Exception的子类和RuntimeException的子类，它们分别对应着检查异常和非检查异常。

Checked exception

检查异常，继承自Exception类。对于检查异常，Java强制我们必须进行处理。对于抛出检查异常的API我们有两种处理方式：

对抛出检查异常的API进程try catch

继续把检查异常往上抛

常见的检查异常有：

SQLException

IOException

InterruptedException

Unchecked exception

非检查异常，也称运行时异常RuntimeException ，继承自RuntimeException，所有非检查都有个特点，就是代码不需要处理它们的异常也能通过编译，所以它们称作unchecked exception。RuntimeException 本身也是继承自Exception。

常见的非检查异常有：

NullPointerException

IllegalArgumentException

NumberFormatException

IndexOutOfBoundsException

IllegalStateException

检查 (Checked) 异常与非检查 (Unchecked) 异常

Java 语言规范将派生于 Error 类或 RuntimeException 类的所有异常都称为非检查异常。除非检查异常以外的所有异常都称为检查异常。检查异常对方法调用者来说属于必须处理的异常，当一个应用系统定义了大量或者容易产生很多检查异常的方法调用，程序中会有很多的异常处理代码。

如果一个异常是致命的且不可恢复并且对于捕获该异常的方法根本不知如何处理时，或者捕获这类异常无任何益处，笔者认为应该定义这类异常为非检查异常，由顶层专门的异常处理程序处理；像数据库连接错误、网络连接错误或者文件打不开等之类的异常一般均属于非检查异常。这类异常一般与外部环境相关，一旦出现，基本无法有效地处理。

而对于一些具备可以回避异常或预料内可以恢复并存在相应的处理方法的异常，可以定义该类异常为检查异常。像一般由输入不合法数据引起的异常或者与业务相关的一些异常，基本上属于检查异常。当出现这类异常，一般可以经过有效处理或通过重试可以恢复正常状态。

在Java中如果需要处理异常，必须先对异常进行捕获，然后再对异常情况进行处理。如何对可能发生异常的代码进行异常捕获和处理呢？使用try和catch关键字即可，如下面一段代码所示：

```
try {
  File file = new File("d:/a.txt");
  if(!file.exists())
    file.createNewFile();
} catch (IOException e) {
  // TODO: handle exception
}
```

被try块包围的代码说明这段代码可能会发生异常，一旦发生异常，异常便会被catch捕获到，然后需要在catch块中进行异常处理。

这是一种处理异常的方式。在Java中还提供了另一种异常处理方式即抛出异常，顾名思义，也就是说一旦发生异常，我把这个异常抛出去，让调用者去进行处理，自己不进行具体的处理，此时需要用到throw和throws关键字。

```
public class Main {
    public static void main(String[] args) {
        try {
            createFile();
        } catch (Exception e) {
            // TODO: handle exception
        }
    }

    public static void createFile() throws IOException{
        File file = new File("d:/a.txt");
        if(!file.exists())
            file.createNewFile();
    }
}
```

这段代码和上面一段代码的区别是，在实际的createFile方法中并没有捕获异常，而是用throws关键字声明抛出异常，即告知这个方法的调用者此方法可能会抛出IOException。那么在main方法中调用createFile方法的时候，采用try...catch块进行了异常捕获处理。

当然还可以采用throw关键字手动来抛出异常对象。下面看一个例子：

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] data = new int[]{1,2,3};
            System.out.println(getDataByIndex(-1,data));
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

    }

    public static int getDataByIndex(int index,int[] data) {
        if(index<0||index>=data.length){
            throw new ArrayIndexOutOfBoundsException("数组下标越界");
            System.out.println(“fatal errors!”);
        }
        return data[index];
    }
}
```

也就说在Java中进行异常处理的话，对于可能会发生异常的代码，可以选择三种方法来进行异常处理：

1）对代码块用try..catch进行异常捕获处理；

2）在 该代码的方法体外用throws进行抛出声明，告知此方法的调用者这段代码可能会出现这些异常，你需要谨慎处理。此时有两种情况：

如果声明抛出的异常是非运行时异常，此方法的调用者必须显示地用try..catch块进行捕获或者继续向上层抛出异常。

如果声明抛出的异常是运行时异常，此方法的调用者可以选择地进行异常捕获处理。

3）在代码块用throw手动抛出一个异常对象，此时也有两种情况，跟2）中的类似：

如果抛出的异常对象是非运行时异常，此方法的调用者必须显示地用try..catch块进行捕获或者继续向上层抛出异常。

如果抛出的异常对象是运行时异常，此方法的调用者可以选择地进行异常捕获处理。

- .try,catch,finally
- try关键字用来包围可能会出现异常的逻辑代码，它单独无法使用，必须配合catch或者finally使用。Java编译器允许的组合使用形式只有以下三种形式：
- try...catch...;　try....finally.....;　try....catch...finally...
- 当然catch块可以有多个，注意try块只能有一个,finally块是可选的（但是最多只能有一个finally块）。
- 三个块执行的顺序为try—>catch—>finally。
- 当然如果没有发生异常，则catch块不会执行。但是finally块无论在什么情况下都是会执行的（这点要非常注意，因此部分情况下，都会将释放资源的操作放在finally块中进行）。
- 在有多个catch块的时候，是按照catch块的先后顺序进行匹配的，一旦异常类型被一个catch块匹配，则不会与后面的catch块进行匹配。

- throws和thow关键字
- 1）throws出现在方法的声明中，表示该方法可能会抛出的异常，然后交给上层调用它的方法程序处理，允许throws后面跟着多个异常类型；
- 2）一般会用于程序出现某种逻辑时程序员主动抛出某种特定类型的异常。throw只会出现在方法体中，当方法在执行过程中遇到异常情况时，将异常信息封装为异常对象，然后throw出去。throw关键字的一个非常重要的作用就是 异常类型的转换。
- throws表示出现异常的一种可能性，并不一定会发生这些异常；throw则是抛出了异常，执行throw则一定抛出了某种异常对象。两者都是消极处理异常的方式（这里的消极并不是说这种方式不好），只是抛出或者可能抛出异常，但是不会由方法去处理异常，真正的处理异常由此方法的上层调用处理。

- **Java 内置异常类**
- Java 语言定义了一些异常类在 java.lang 标准包中。
- 标准运行时异常类的子类是最常见的异常类。由于 java.lang 包是默认加载到所有的 Java 程序的，所以大部分从运行时异常类继承而来的异常都可以直接使用。
- Java 根据各个类库也定义了一些其他的异常，下面的表中列出了 Java 的非检查性异常。

ClassCastException	当试图将对象强制转换为不是实例的子类时，抛出该异常。
IndexOutOfBoundsException	指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
NullPointerException	当应用程序试图在需要对象的地方使用 null 时，抛出该异常

Java 定义在 java.lang 包中的检查性异常类。

ClassNotFoundException　应用程序试图加载类时，找不到相应的类，抛出该异常。

NoSuchMethodException　请求的方法不存在

下面的列表是 Throwable 类的主要方法:

序号　　方法及说明

1　　public String getMessage()

返回关于发生的异常的详细信息。这个消息在Throwable 类的构造函数中初始化了。

2　　public Throwable getCause()

返回一个Throwable 对象代表异常原因。

3　　public String toString()

使用getMessage()的结果返回类的串级名字。

4　　public void printStackTrace()

打印toString()结果和栈层次到System.err，即错误输出流。

5　　public StackTraceElement [] getStackTrace()

返回一个包含堆栈层次的数组。下标为0的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。

6　　public Throwable fillInStackTrace()

用当前的调用栈层次填充Throwable 对象栈层次，添加到栈层次任何先前信息中。

多线程

现代操作系统比如Mac OS X，UNIX，Linux，Windows等，都是支持“多任务”的操作系统。

什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听MP3，一边在用Word赶作业，这就是多任务，至少同时有3个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。

现在，多核CPU已经非常普及了，但是，即使过去的单核CPU，也可以执行多任务。由于CPU执行代码都是顺序执行的，那么，单核CPU是怎么执行多任务的呢？

答案就是操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2，任务2执行0.01秒，再切换到任务3，执行0.01秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个Word就启动了一个Word进程。

有些进程还不止同时干一件事，比如Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，像Word这种复杂的进程可以有多个线程，多个线程可以同时执行，多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，真正地同时执行多线程需要多核CPU才可能实现。

CPU+RAM+各种资源（比如显卡，光驱，键盘，GPS, 等等外设）构成我们的电脑，但是电脑的运行，实际就是CPU和相关寄存器以及RAM之间的事情。一个最最基础的事实：CPU太快，太快，太快了，寄存器仅仅能够追的上他的脚步，RAM和别的挂在各总线上的设备完全是望其项背。那当多个任务要执行的时候怎么办呢？轮流着来?或者谁优先级高谁来？不管怎么样的策略，一句话就是在CPU看来就是轮流着来。一个必须知道的事实：执行一段程序代码，实现一个功能的过程介绍，当得到CPU的时候，相关的资源必须也已经就位，就是显卡啊，GPS啊什么的必须就位，然后CPU开始执行。这里除了CPU以外所有的就构成了这个程序的执行环境，也就是我们所定义的程序上下文。当这个程序执行完了，或者分配给他的CPU执行时间用完了，那它就要被切换出去，等待下一次CPU的临幸。在被切换出去的最后一步工作就是保存程序上下文，因为这个是下次他被CPU临幸的运行环境，必须保存。串联起来的事实：前面讲过在CPU看来所有的任务都是一个一个的轮流执行的，具体的轮流方法就是：先加载程序A的上下文，然后开始执行A，保存程序A的上下文，调入下一个要执行的程序B的程序上下文，然后开始执行B,保存程序B的上下文。。。。

进程（Processes ）和线程（Threads）

进程和线程是并发编程的两个基本的执行单元。在 Java 中，并发编程主要涉及线程。

一个计算机系统通常有许多活动的进程和线程。在给定的时间内，每个处理器只能有一个线程得到真正的运行。对于单核处理器来说，处理时间是通过时间切片来在进程和线程之间进行共享的。

现在多核处理器或多进程的电脑系统越来越流行。这大大增强了系统的进程和线程的并发执行能力。但即便是没有多处理器或多进程的系统中，并发仍然是可能的。

进程

进程有一个独立的执行环境。进程通常有一个完整的、私人的基本运行时资源;特别是,每个进程都有其自己的内存空间。

进程往往被视为等同于程序或应用程序。然而,用户将看到一个单独的应用程序可能实际上是一组合作的进程。大多数操作系统都支持进程间通信(Inter Process Communication，简称 IPC)资源,如管道和套接字。IPC 不仅用于同个系统的进程之间的通信，也可以用在不同系统的进程。

大多数 Java 虚拟机的实现作为一个进程运行。

线程

线程有时被称为轻量级进程。进程和线程都提供一个执行环境,但创建一个新的线程比创建一个新的进程需要更少的资源。

线程中存在于进程中,每个进程都至少一个线程。线程共享进程的资源,包括内存和打开的文件。这使得工作变得高效，但也存在了一个潜在的问题——通信。

多线程执行是 Java 平台的一个重要特点。每个应用程序都至少有一个线程,或者几个,如果算上“系统”的线程（负责内存管理和信号处理）那就更多。但从程序员的角度来看,你启动只有一个线程，称为主线程。这个线程有能力创建额外的线程。

Java 给多线程编程提供了内置的支持。一个多线程程序包含两个或多个能并发运行的部分。

在java中要想实现多线程，有两种手段，一种是继承Thread类，另外一种是实现Runable接口.(其实准确来讲，应该有三种，还有一种是实现Callable接口，并与Future、线程池结合使用)

Java 线程类也是一个 object 类，它的实例都继承自 java.lang.Thread 或其子类。 可以用如下方式用 java 中创建一个线程：

```
Tread thread = new Thread();
```

执行该线程可以调用该线程的 start()方法:

```
thread.start();
```

在上面的例子中，我们并没有为线程编写运行代码，因此调用该方法后线程就终止了。

编写线程运行时执行的代码有两种方式：一种是创建 Thread 子类的一个实例并重写 run 方法，第二种是创建类的时候实现 Runnable 接口

java.lang.Thread 类
Thread 类是一个具体的类，即不是抽象类，该类封装了线程的行为。要创建一个线程，程序员必须创建一个从 Thread 类导出的新类。程序员必须覆盖 Thread 的 run() 函数来完成有用的工作。用户并不直接调用此函数；而是必须调用 Thread 的 start() 函数，该函数再调用 run()。

继承Thread类的方法是比较常用的一种，如果说你只是想起一条线程。没有什么其它特殊的要求，那么可以使用Thread.

对于直接继承Thread的类来说，代码大致框架是：

```
class 类名 extends Thread{
方法1;
方法2;
…
public void run(){
// other code…
}
属性1;
属性2;
…

}
```

```
class hello extends Thread {

    public hello() {

    }

    public hello(String name) {
        this.name = name;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(name + "运行     " + i);
        }
    }

    public static void main(String[] args) {
        hello h1=new hello("A");
        hello h2=new hello("B");
        h1.start();
        h2.start();
    }

    private String name;
}
```

因为需要用到CPU的资源，所以每次的运行结果基本是都不一样的

Runnable 接口

此接口只有一个函数，run()，此函数必须由实现了此接口的类实现。

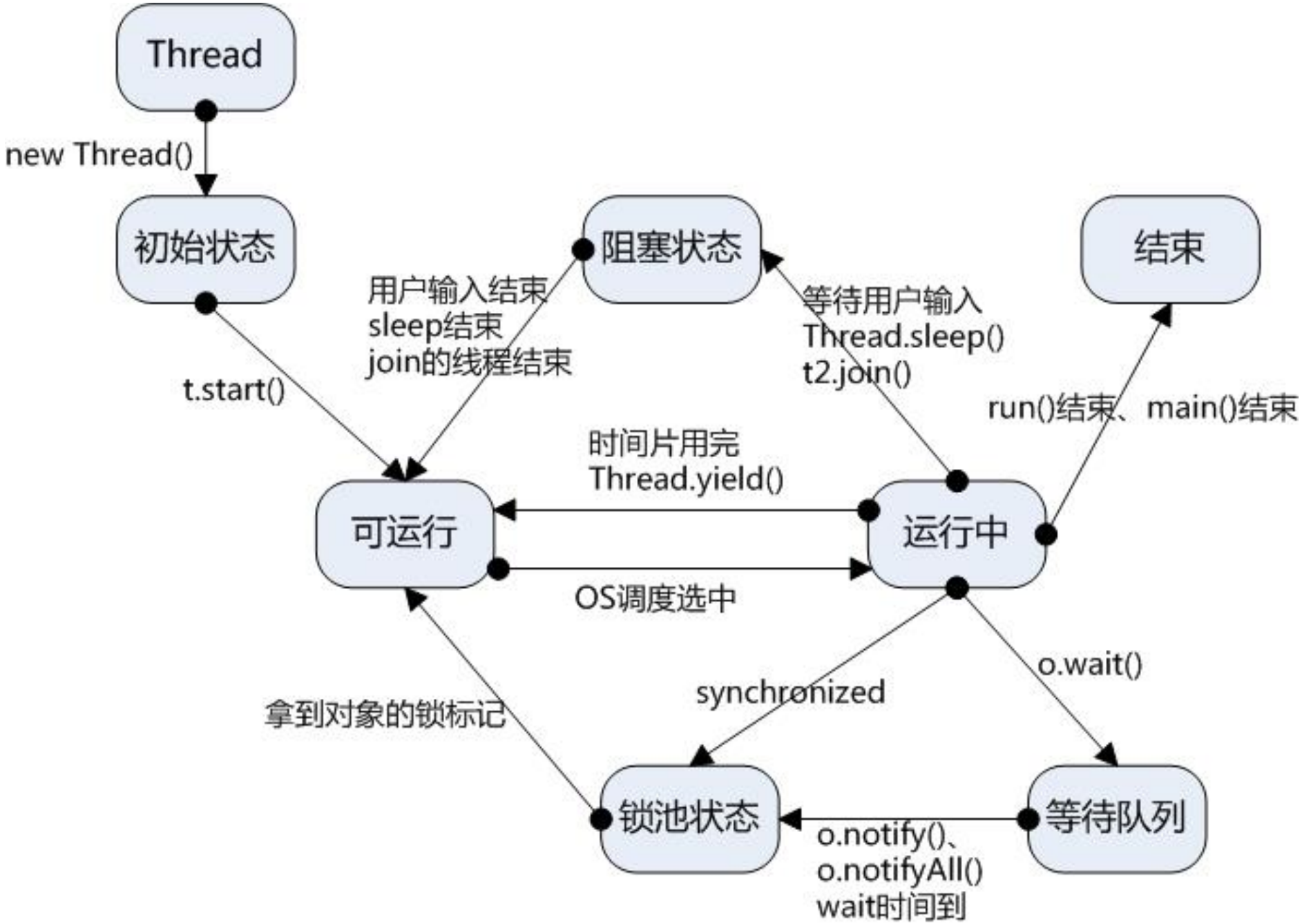
```
class TimePrinter        implements Runnable {
   int pauseTime;
   String name;
   public TimePrinter(int x, String n) {
      pauseTime = x;
      name = n;
   }
   public void run() {
      while(true) {
         try {
            System.out.println(name + ":" + new      Date(System.currentTimeMillis()));
            Thread.sleep(pauseTime);
         } catch(Exception e) {
            System.out.println(e);
         }
      }
   }
```

```
static public void main(String args[]) {
        Thread t1 = new Thread (new TimePrinter(1000, "Fast Guy"));
        t1.start();
        Thread t2 = new Thread (new TimePrinter(3000, "Slow Guy"));
        t2.start();


    }
}
```

实现Runnable接口，使得该类有了多线程类的特征。run（）方法是多线程程序的一个约定。所有的多线程代码都在run方法里面。Thread类实际上也是实现了Runnable接口的类。

在启动的多线程的时候，需要先通过Thread类的构造方法Thread(Runnable target) 构造出对象，然后调用Thread对象的start()方法来运行多线程代码。

实际上所有的多线程代码都是通过运行Thread的start()方法来运行的。因此，不管是扩展Thread类还是实现Runnable接口来实现多线程，最终还是通过Thread的对象的API来控制线程的


Thread
new Thread()
初始状态
t.start()
可运行
用户输入结束
sleep结束
join的线程结束
阻塞状态
等待用户输入
Thread.sleep()
t2.join()
时间片用完
Thread.yield()
OS调度选中
运行中
结束
run()结束、main()结束
拿到对象的锁标记
synchronized
锁池状态
o.wait()
等待队列
o.notify()、
o.notifyAll()
wait时间到

创建 Thread 的子类

创建 Thread 子类的一个实例并重写 run 方法，run 方法会在调用 start()方法之后被执行。例子如下：

```
public class MyThread extends Thread {
  public void run(){
    System.out.println("MyThread running");
  }
}
```

可以用如下方式创建并运行上述 Thread 子类

```
MyThread myThread = new MyThread();
myTread.start();
```

一旦线程启动后 start 方法就会立即返回，而不会等待到 run 方法执行完毕才返回。就好像 run 方法是在另外一个 cpu 上执行一样。当 run 方法执行后，将会打印出字符串 MyThread running。

```
public class MyRunnable implements Runnable {
   public void run(){
    System.out.println("MyRunnable running");
   }
}
```

为了使线程能够执行 run()方法，需要在 Thread 类的构造函数中传入 MyRunnable 的实例对象。示例如下：

```
Thread thread = new Thread(new MyRunnable());
thread.start();
```

当线程运行时，它将会调用实现了 Runnable 接口的 run 方法。上例中将会打印出”MyRunnable running”。

在同一程序中运行多个线程本身不会导致问题，问题在于多个线程访问了相同的资源。如，同一内存区（变量，数组，或对象）、系统（数据库，web services 等）或文件。实际上，这些问题只有在一或多个线程向这些资源做了写操作时才有可能发生，只要资源没有发生变化,多个线程读取相同的资源就是安全的。

多线程同时执行下面的代码可能会出错：

```
public class Counter {
    protected long count = 0;
    public void add(long value){
        this.count = this.count + value;
    }
}
```

想象下线程 A 和 B 同时执行同一个 Counter 对象的 add()方法，我们无法知道操作系统何时会在两个线程之间切换。JVM 并不是将这段代码视为单条指令来执行的，而是按照下面的顺序：

从内存获取 this.count 的值放到寄存器

将寄存器中的值增加 value

将寄存器中的值写回内存

观察线程 A 和 B 交错执行会发生什么：

this.count = 0;

A: 读取 this.count 到一个寄存器 (0)

B: 读取 this.count 到一个寄存器 (0)

B: 将寄存器的值加 2

B: 回写寄存器值(2)到内存. this.count 现在等于 2

A: 将寄存器的值加 3

A: 回写寄存器值(3)到内存. this.count 现在等于 3

两个线程分别加了 2 和 3 到 count 变量上，两个线程执行结束后 count 变量的值应该等于 5。然而由于两个线程是交叉执行的，两个线程从内存中读出的初始值都是 0。然后各自加了 2 和 3，并分别写回内存。最终的值并不是期望的 5，而是最后写回内存的那个线程的值，上面例子中最后写回内存的是线程 A，但实际中也可能是线程 B。如果没有采用合适的同步机制，线程间的交叉执行情况就无法预料。

竞态条件 & 临界区

当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。导致竞态条件发生的代码区称作临界区。上例中 add()方法就是一个临界区,它会产生竞态条件。在临界区中使用适当的同步就可以避免竞态条件。

Java 同步关键字（synchronized）

Java 中的同步块用 synchronized 标记。同步块在 Java 中是同步在某个对象上。所有同步在一个对象上的同步块在同时只能被一个线程进入并执行操作。所有其他等待进入该同步块的线程将被阻塞，直到执行该同步块中的线程退出。

下面是一个同步的实例方法：

```
public synchronized void add(int value){
this.count += value;
}
```

注意在方法声明中同步（synchronized ）关键字。这告诉 Java 该方法是同步的。

Java 实例方法同步是同步在拥有该方法的对象上。这样，每个实例其方法同步都同步在不同的对象上，即该方法所属的实例。只有一个线程能够在实例方法同步块中运行。如果有多个实例存在，那么一个线程一次可以在一个实例同步块中执行操作。一个实例一个线程。

Xml

XML 指可扩展标记语言（eXtensible Markup Language）。

XML 被设计用来传输和存储数据。

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

上面的这条便签具有自我描述性。它包含了发送者和接受者的信息，同时拥有标题以及消息主体。

但是，这个 XML 文档仍然没有做任何事情。它仅仅是包装在 XML 标签中的纯粹的信息。我们需要编写软件或者程序，才能传送、接收和显示出这个文档。

XML 指可扩展标记语言（EXtensible Markup Language）。

XML 是一种很像HTML的标记语言。

XML 的设计宗旨是传输数据，而不是显示数据。

XML 标签没有被预定义。您需要自行定义标签。

XML 被设计为具有自我描述性。

XML 是 W3C 的推荐标准。

XML 仅仅是纯文本

XML 没什么特别的。它仅仅是纯文本而已。有能力处理纯文本的软件都可以处理 XML。

不过，能够读懂 XML 的应用程序可以有针对性地处理 XML 的标签。标签的功能性意义依赖于应用程序的特性。

XML 简化数据共享

在真实的世界中，计算机系统和数据使用不兼容的格式来存储数据。

XML 数据以纯文本格式进行存储，因此提供了一种独立于软件和硬件的数据存储方法。

这让创建不同应用程序可以共享的数据变得更加容易。

XML 的语法规则

XML 文档必须有根元素

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

XML 声明文件的可选部分，如果存在需要放在文档的第一行，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
```

所有的 XML 元素都必须有一个关闭标签

在 HTML 中，某些元素不必有一个关闭标签：

This is a paragraph.

在 XML 中，省略关闭标签是非法的。所有元素都必须有关闭标签：

This is a paragraph.

XML 标签对大小写敏感。标签 <Letter> 与标签 <letter> 是不同的。

必须使用相同的大小写来编写打开标签和关闭标签：

<Message>This is incorrect</message>

<message>This is correct</message>

在 HTML 中，常会看到没有正确嵌套的元素：

This text is bold and italic

在 XML 中，所有元素都必须彼此正确地嵌套：

This text is bold and italic

在上面的实例中，正确嵌套的意思是：由于 元素是在 元素内打开的，那么它必须在 元素内关闭。

与 HTML 类似，XML 元素也可拥有属性（名称/值的对）。

在 XML 中，XML 的属性值必须加引号。

请研究下面的两个 XML 文档。 第一个是错误的，第二个是正确的：

<note date=12/11/2007>

<to>Tove</to>

<to>Tove</to>

<from>Jani</from>

</note>

<note date="12/11/2007">

<to>Tove</to>

<from>Jani</from>

</note>

在第一个文档中的错误是，note 元素中的 date 属性没有加引号。

在 XML 中，一些字符拥有特殊的意义。

如果您把字符 "<" 放在 XML 元素中，会发生错误，这是因为解析器会把它当作新元素的开始。

这样会产生 XML 错误：

<message>if salary < 1000 then</message>

为了避免这个错误，请用实体引用来代替 "<" 字符：

<message>if salary < 1000 then</message>

在 XML 中，有 5 个预定义的实体引用：

<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

在 XML 中编写注释的语法与 HTML 的语法很相似。

<!-- This is a comment -->

Json

JSON 或者 JavaScript 对象表示法是一种轻量级的基于文本的开放标准，被设计用于可读的数据交换。约定使用 JSON 的程序包括 C，C++，Java，Python，Perl 等等。

JSON 是 JavaScript Object Notation 的缩写。

这个格式由 Douglas Crockford 提出。

被设计用于可读的数据交换。

它是从 JavaScript 脚本语言中演变而来。

文件名扩展是 .json。

JSON 的网络媒体类型是 application/json。

SON 特点

JSON 容易阅读和编写。

它是一种轻量级的基于文本的交换格式。

语言无关。

```
{
   "book": [
      {
         "id":"01",
         "language": "Java",
         "edition": "third",
         "author": "Herbert Schildt"
      },
      {
         "id":"07",
         "language": "C++",
         "edition": "second"
         "author": "E.Balagurusamy"
   }]
}
```

JSON 的基本语法

数据使用名/值对表示。
使用大括号保存对象，每个名称后面跟着一个 ':'（冒号），名/值对使用 ,（逗号）分割。
使用方括号保存数组，数组值使用 ,（逗号）分割。

JSON 格式支持以下数据类型：

类型	描述
数字型（Number ）	JavaScript 中的双精度浮点型格式
字符串型（String ）	双引号包裹的 Unicode 字符和反斜杠转义字符
布尔型（Boolean ）	true 或 false
数组（Array ）	有序的值序列
值（Value ）	可以是字符串，数字，true 或 false，null 等等
对象（Object ）	无序的键:值对集合
空格（Whitespace ）	可用于任意符号对之间
null	空

Maven

在进行软件开发的过程中，无论什么项目，采用何种技术，使用何种编程语言，我们 都要重复相同的开发步骤：编码，测试，打包，发布，文档。实际上这些步骤是完全重复性的工作。那为什么让软件开发人员去重复这些工作？开发人员的主要任务 应该是关注商业逻辑并去实现它，而不是把时间浪费在学习如何在不同的环境中去打包，发布，。。。。

Maven正是为了将开发人员从这些任务中解脱出来而诞生的。Apache Maven 是一种用作软件项目管理和理解工具。它基于项目对象模型（POM）的概念， 可以管理一个项目的构建、报告以及从项目核心信息中生成文档。

Maven能够：

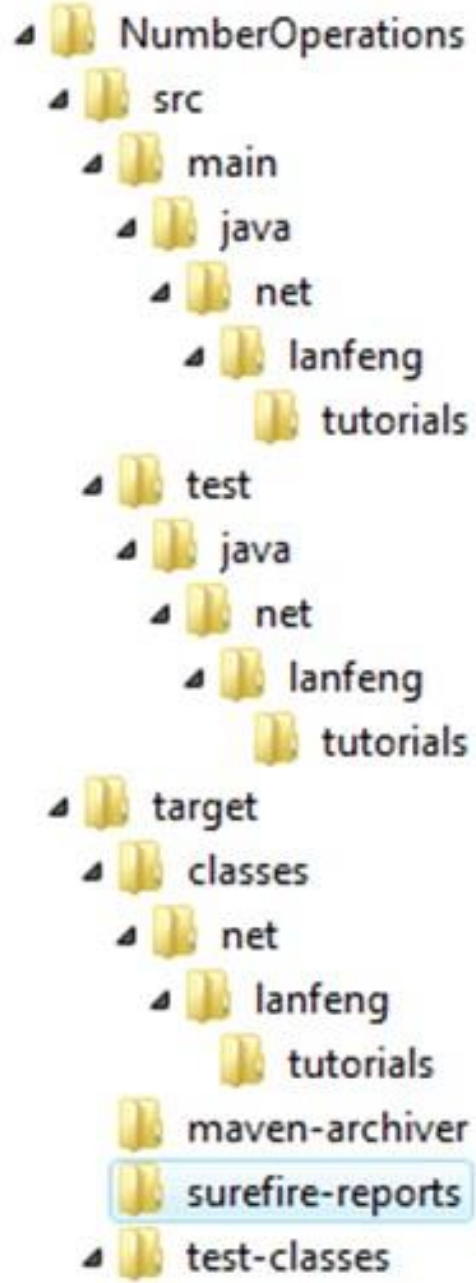
1）理解并管理整个软件开发周期，重用标准的构建过程，比如：编译，测试，打包等。同时Maven还可以通过相应的元数据，重用构建逻辑到一个项目。

2）Maven负责整个项目的构建过程。开发人员只需要描述项目基本信息在一个配置文件中：pom.xml。也就是说，Maven的使用者只需要回答“What”而不是“How”。

Maven设计原则

1）Convention Over Configuration (约定优于配置)。在现实生活中，有很多常识性的东西，地球人都知道。比如说：如何过马路(红灯停绿灯行)，如何开门，关门等。对于这些事情，人们已经有了默认的约定。

在软件开发过程中，道理也是类似的，如果我们事先约定好所有项目的目录结构，标 准开发过程（编译，测试，。。。。），所有人都遵循这个约定。软件项目的管理就会变得简单很多。在现在流行的很多框架中，都使用了这个概念，比如EJB3和 Ruby on Rails。在Maven中默认的目录结构如下：


NumberOperations
src
main
java
net
lanfeng
tutorials
test
java
net
lanfeng
tutorials
target
classes
net
lanfeng
tutorials
maven-archiver
surefire-reports
test-classes

可以看出以下几个标准的Maven目录：

src：源代码目录。所有的源代码都被放在了这个目录下。在这个目录下又包括了：

1) main：所有的源代码放在这里。对于Java项目，还有一个下级子目录：java. 对于Flex项目则是flex，。。。

2) test：所有的单元测试类放在这里。

target：所有编译过的类文件以及生成的打包文件(.jar, .war, ...)放在这里。

2）Reuse Build Logic (重用构建逻辑)：Maven把构建逻辑封装到插件中来达到重用的目的。这样在Maven就有用于编译的插件，单元测试的插件，打包的插件，。。。Maven可以被理解成管理这些插件的框架。

3）Declarative Execution (声明式执行)：Maven中所有的插件都是通过在POM中声明来定义的。Maven会理解所有在POM中的声明，并执行相应的插件。

src/main/java - 存放项目.java文件；

src/main/resources - 存放项目资源文件；

src/test/java - 存放测试类.java文件；

src/test/resources - 存放测试资源文件；

target - 项目输出目录；

pom.xml - Maven核心文件（Project Object Model）；

下载Maven：http://maven.apache.org/

2) 解压缩下载的zip文件到本地目录下，比如：D:\Maven

3) 添加D:\Maven\bin到环境变量PATH中

4) 在命令行下运行:

mvn -version 或者 mvn –v

一些Maven命令

编译: mvn compile

单元测试: mvn test

构建并打包： mvn package

清理： mvn clean

安装 mvn clean install

maven 项目会有一个 pom.xml 文件， 在这个文件里面，只要你添加相应配置，他就会自动帮你下载相应 jar 包，不用你铺天盖地的到处搜索你需要的 jar 包了。

```
<dependency>
  <groupId>junit</groupId> 项目名
  <artifactId>junit</artifactId> 项目模块
  <version>3.8.1</version> 项目版本
  <scope>test</scope>
</dependency>
```

maven都会通过，项目名-项目模块-项目版本来maven在互联网上的代码库中下载相应jar包。

所有的POM文件要求有projec节点和三个必须字段：groupId, artifactId,version。

在仓库中项目的标识为groupId:artifactId:version。

POM.xml的根节点是project并且其下有三个主要的子节点：

节点　　描述

groupId　项目组织的Id。通常在一个项目或者一个组织之中，这个Id是唯一的。例如，某个Id为com.company.bank的银行组织包含所有银行相关的项目。

artifactId 项目的Id，通常是项目的名字。例如，consumer-banking。artifactId与groupId一起定义了仓库中项目构件的路径。

version　项目的版本。它与groupId一起，在项目构件仓库中用作区分不同的版本，例如：

com.company.bank:consumer-banking:1.0

com.company.bank:consumer-banking:1.1.

Eclipse 提供了一个极佳的插件 m2eclipse ， 可以无缝地把 Maven 和 Eclipse 集成在一起。

m2eclipse 的一些特性列出如下：

你可以从 Eclipse 中运行 Maven 目标操作。

你可以使用 Eclipse 自身的控制台查看 Maven 命令的输出。

你可以使用 IDE 更新 Maven 依赖。

你可以从 Eclipse 中启动 Maven 构建。

为基于 Maven pom.xml 文件的 Eclipse 构建路径做依赖管理。

解决来自 Eclipse 工作空间的 Maven 依赖，而无需安装依赖到本地 Maven 仓库中（需要依赖的项目在同一个工作空间中）。

自动从 Maven 远程仓库中下载所需依赖及源码。

提供了向导，可供创建 Maven 新项目和 pom.xml 文件以及为已存在的项目开启 Maven 支持。

提供了对 Maven 远程仓库中依赖的快速搜索。

钱进培训是哈法地区资深工程师组成的培训机构，通过各位老师的现身说法，帮助各位学员迅速掌握实战知识，为求职打下坚实的基础。电子邮件：jin.qian.canada@gmail.com钱老师报名、答疑微信号：qianjincanada，或扫描以下二维码添加：

钱老师

加拿大

扫一扫上面的二维码图案，加我微信