

# 优极限

“极限教育，挑战极限”

[www.yjxxt.com](http://www.yjxxt.com)

极限教育，挑战极限。优极限是一个让 95% 的学生年薪过 18 万的岗前培训公司，让我们的学员具备优秀的互联网技术和职业素养，勇攀高薪，挑战极限。公司位于上海浦东，拥有两大校区，共万余平。累计培训学员超 3 万名。我们的训练营就业平均月薪 19000，最高年薪 50 万。

核心理念：让学员学会学习，拥有解决问题的能力，拿到高薪职场的钥匙。

项目驱动式团队协作、一对一服务、前瞻性思维、教练式培养模型-培养你成为就业明星。首创的老学员项目联盟给学员充分的项目、技术支撑，利用优极限平台这根杠杆，不断挑战极限，勇攀高薪，开挂人生。

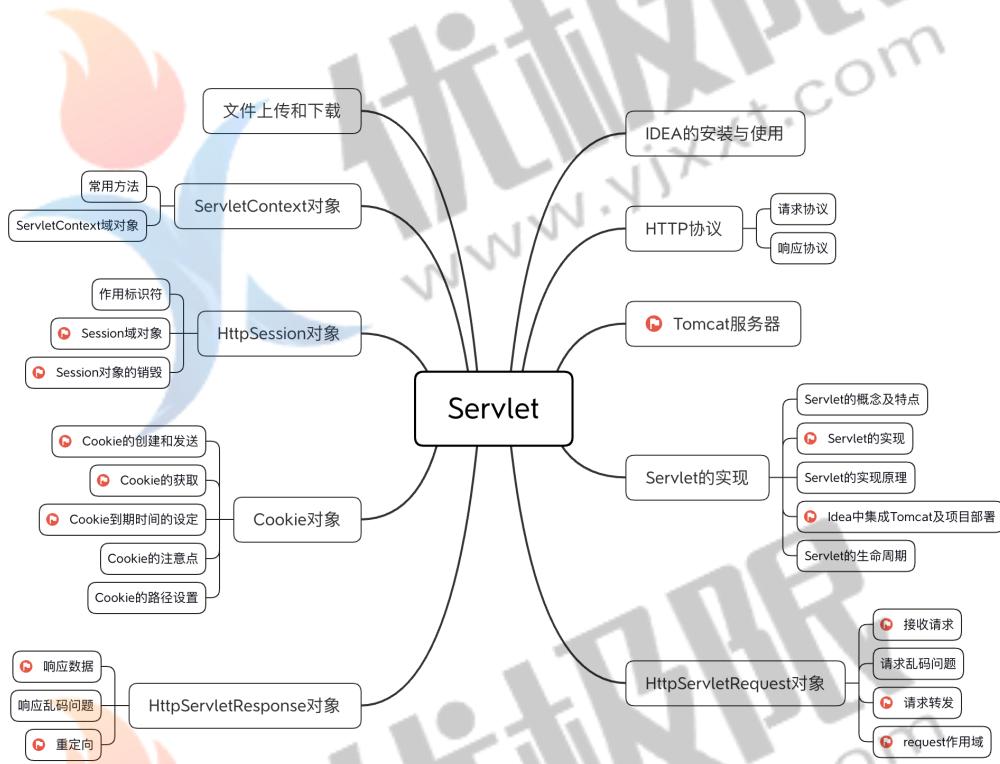
扫码关注优极限微信公众号：

(获取最新技术相关资讯及更多源码笔记)



# Servlet

## 主要内容



## IDEA的安装与使用

IDEA 全称 IntelliJ IDEA，由 JetBrains 公司开发，是 Java 编程语言开发的集成环境。在业界被公认为最好的 Java 开发工具，尤其在智能代码助手、代码自动提示、重构、J2EE 支持、各类版本工具(git、svn 等)、JUnit、CVS 整合、代码分析、创新的 GUI 设计等方面的功能可以说是超常的。

## IDEA的特色功能

- 智能选取
- 丰富的导航模式
- 历史记录功能
- 编码辅助
- 灵活的排版功能
- 代码检查
- 完美的自动代码完成
- 版本控制的支持

.....

## IDEA的下载

1. 在浏览器中IntelliJ IDEA百度一下，打开如下官网

[IntelliJ IDEA: The Java IDE for Professional Developers by...](https://www.jetbrains.com/idea/)

查看此网页的中文翻译，请点击[翻译此页](#)

A Capable and Ergonomic Java IDE for Enterprise Java, Scala, Kotlin and much more...

<https://www.jetbrains.com/idea/> - 百度快照

2. 进入官网，单击DOWNLOAD



3. 选择指定版本，点击DOWNLOAD



4. 点击保存，进行下载



5. 下载之后的文件

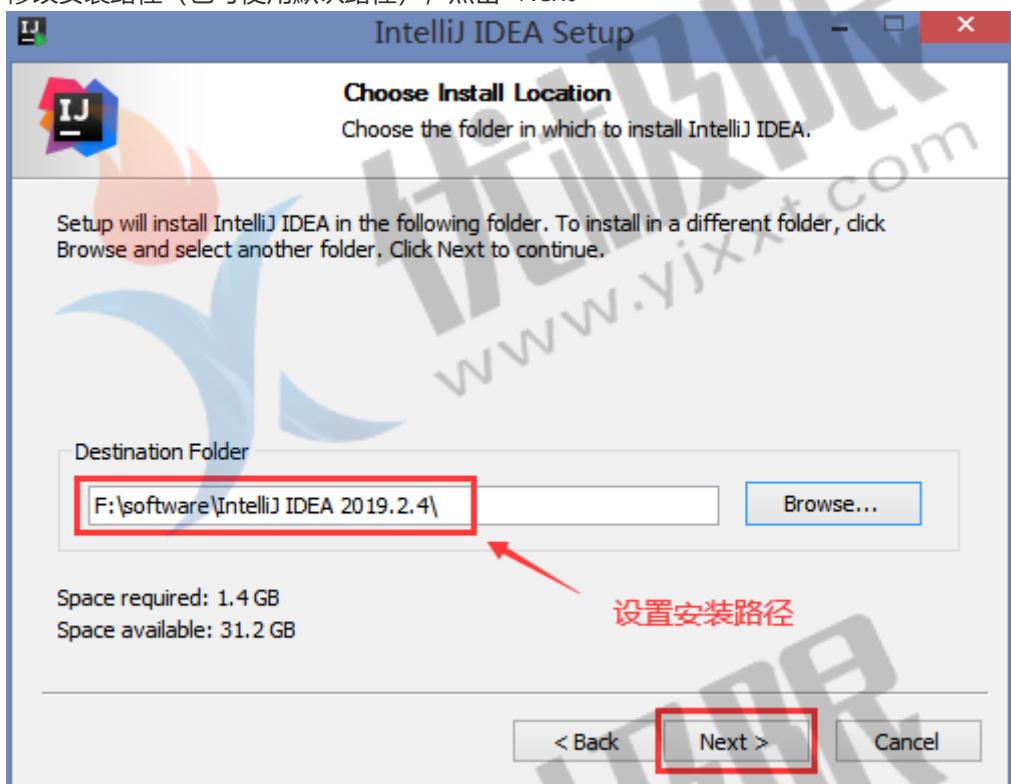
ideaIU-2019.3.3.exe

## IDEA的安装

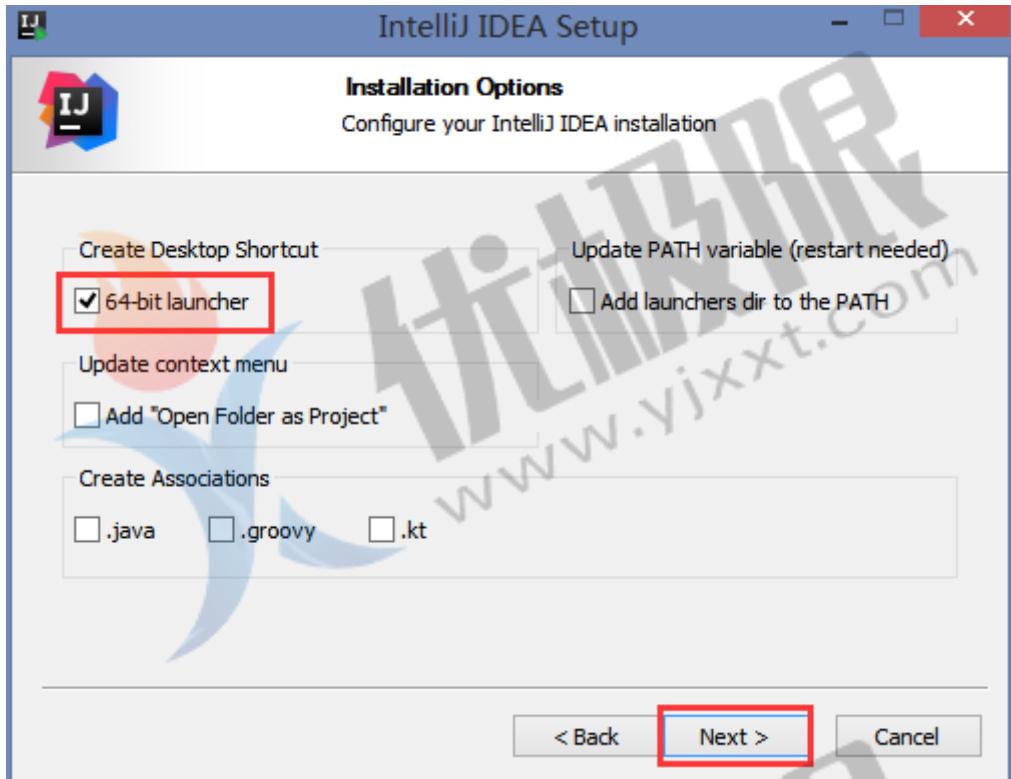
1. 双击运行安装程序，点击 "Next" 下一步



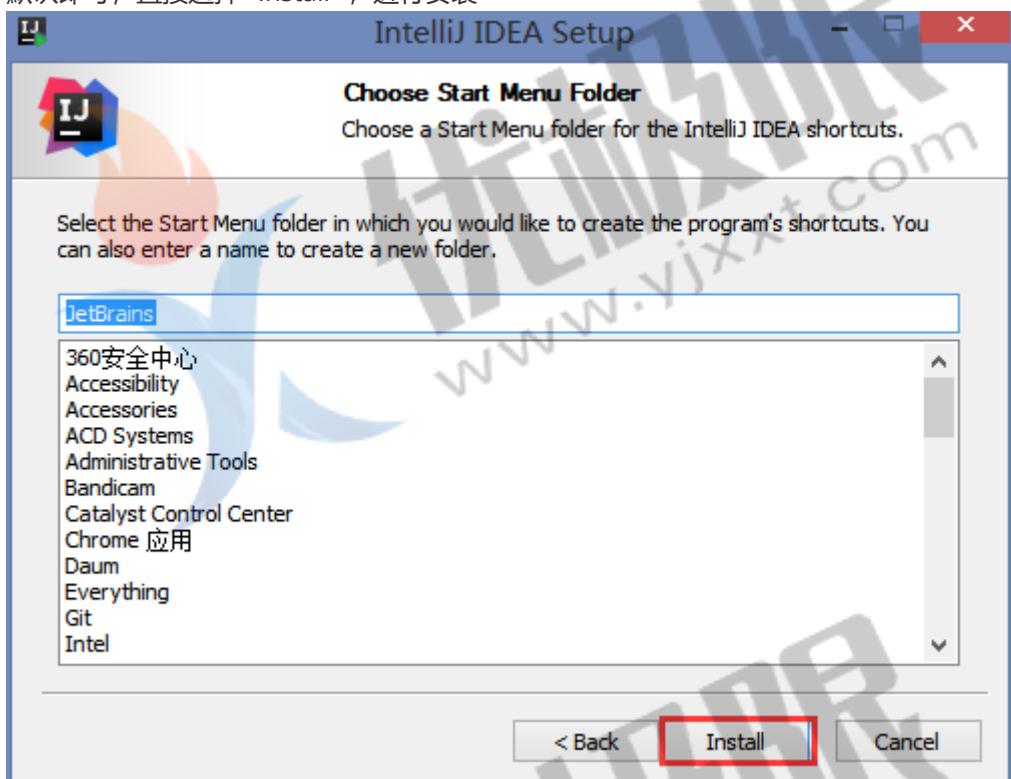
2. 修改安装路径（也可使用默认路径），点击 "Next"



3. 根据自己电脑的操作系统，来进行相应地选择



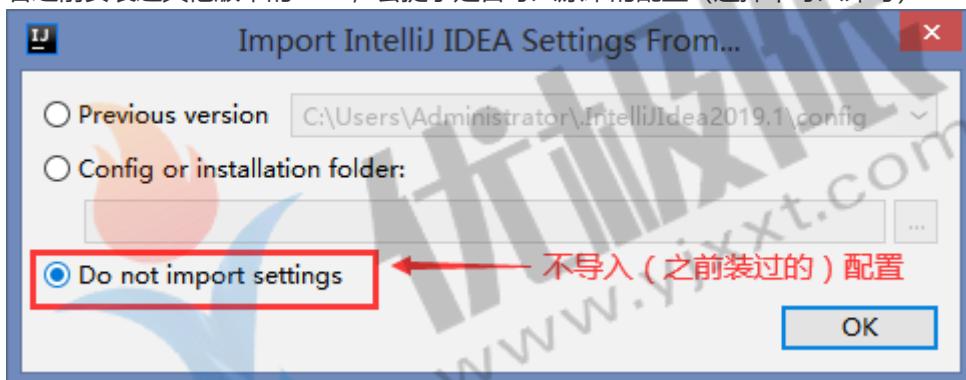
4. 默认即可，直接选择 "Install"，进行安装



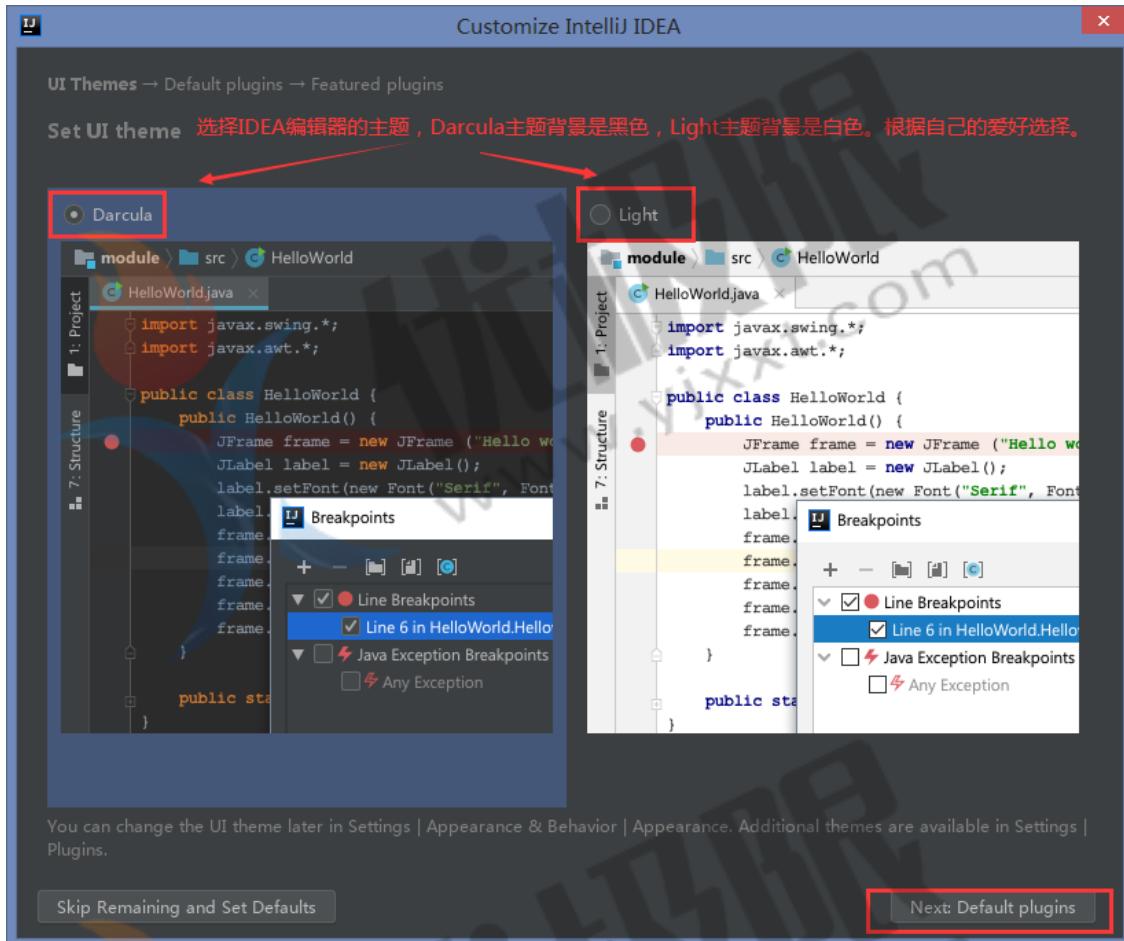
5. 安装成功，可选择运行IDEA，点击 "Finish" 完成安装



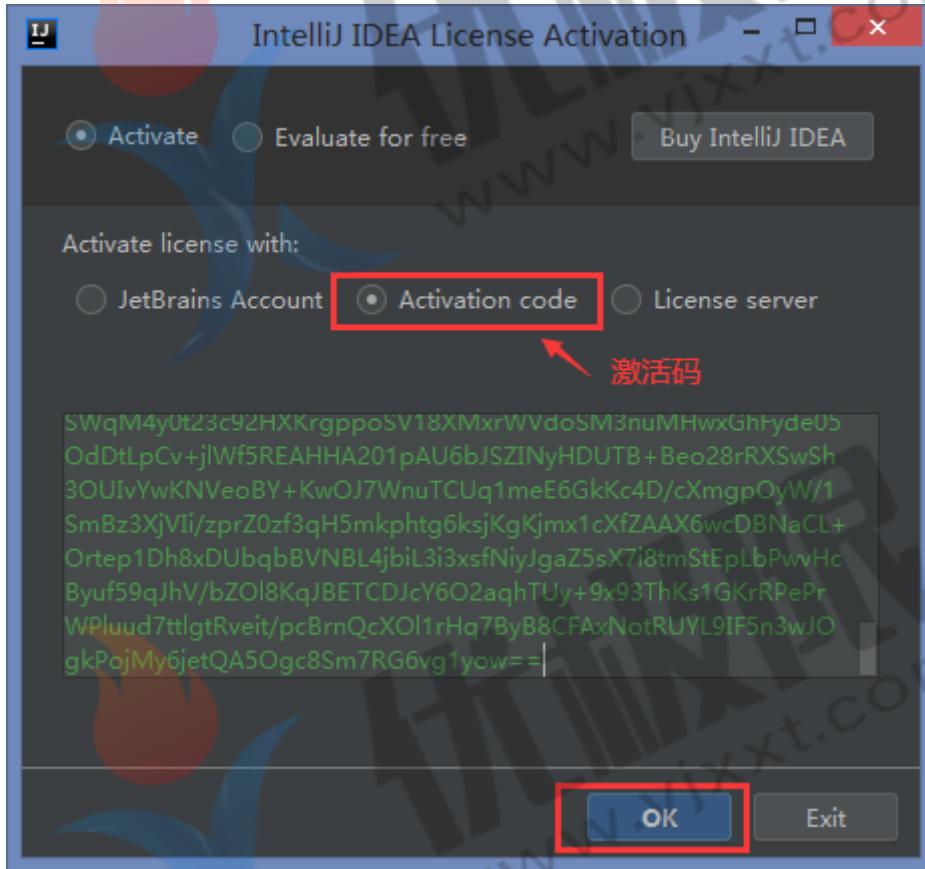
6. 若之前安装过其他版本的IDEA，会提示是否导入原来的配置（选择不导入即可）



7. 选择自己喜欢的主题，然后一直选择 "Next"



8. 选择 "Activation code" 方式，输入激活码，点击 "OK"



9. 出现如下画面，则表示激活成功



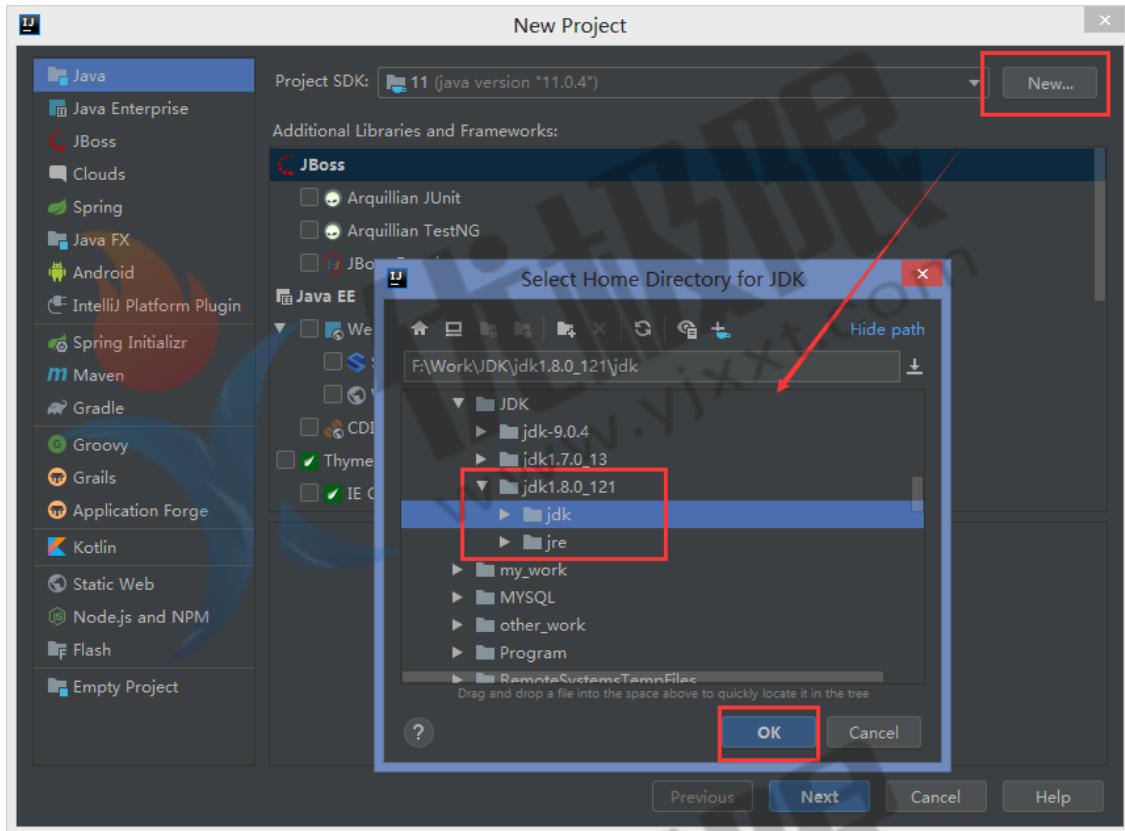
## IDEA创建项目

### 创建Java项目

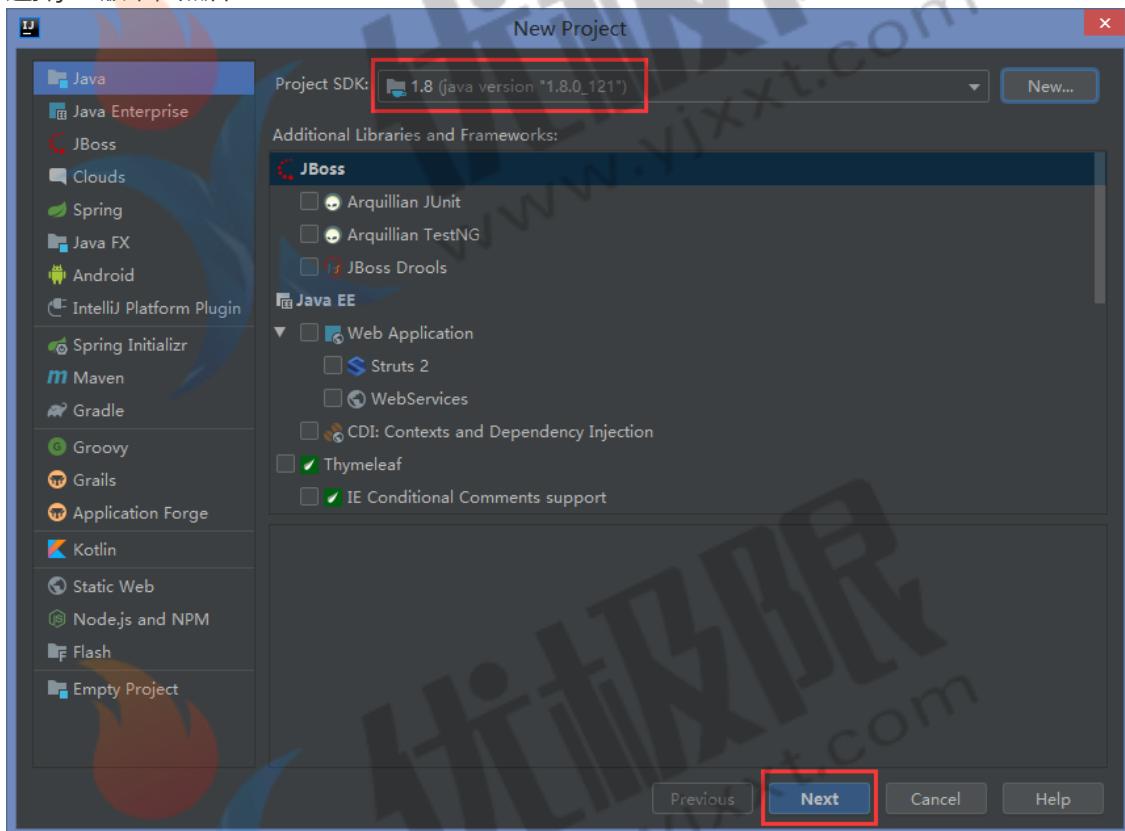
1. 点击 "Create New Project"



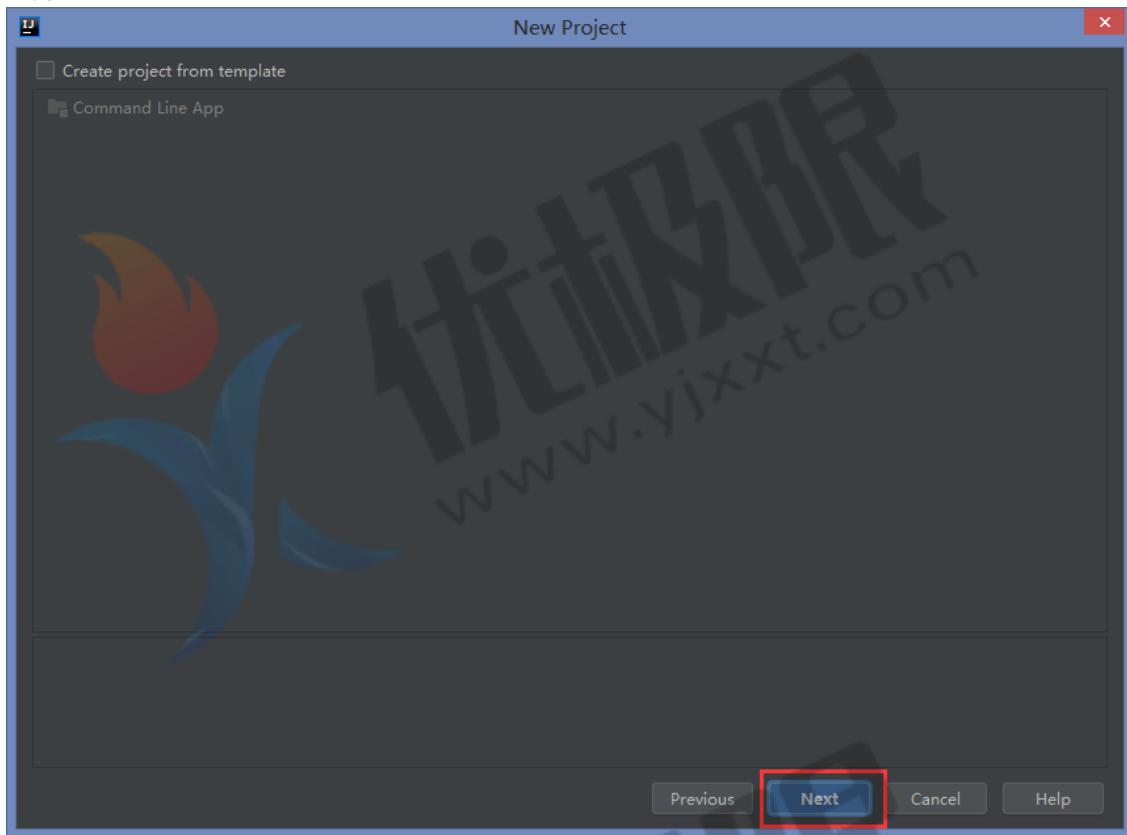
## 2. 添加新的JDK版本 (idea默认使用自带的版本)



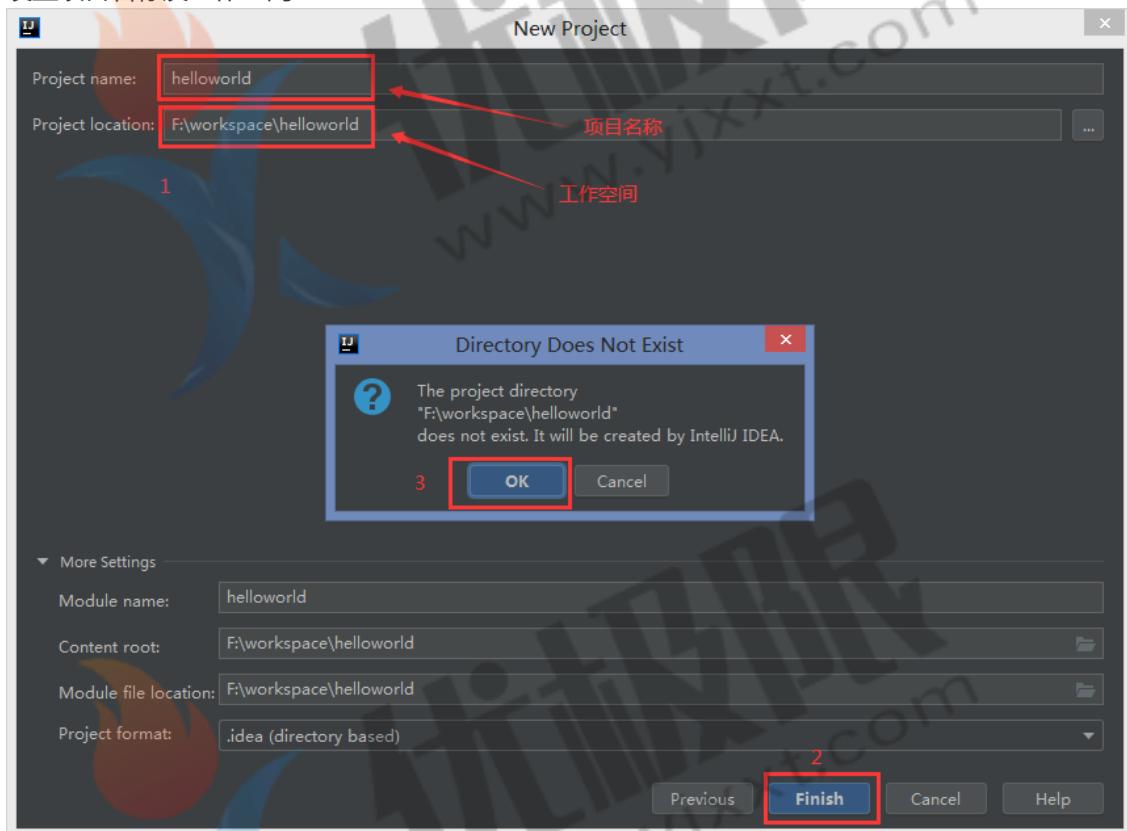
## 3. 选择JDK版本, 然后 "Next"



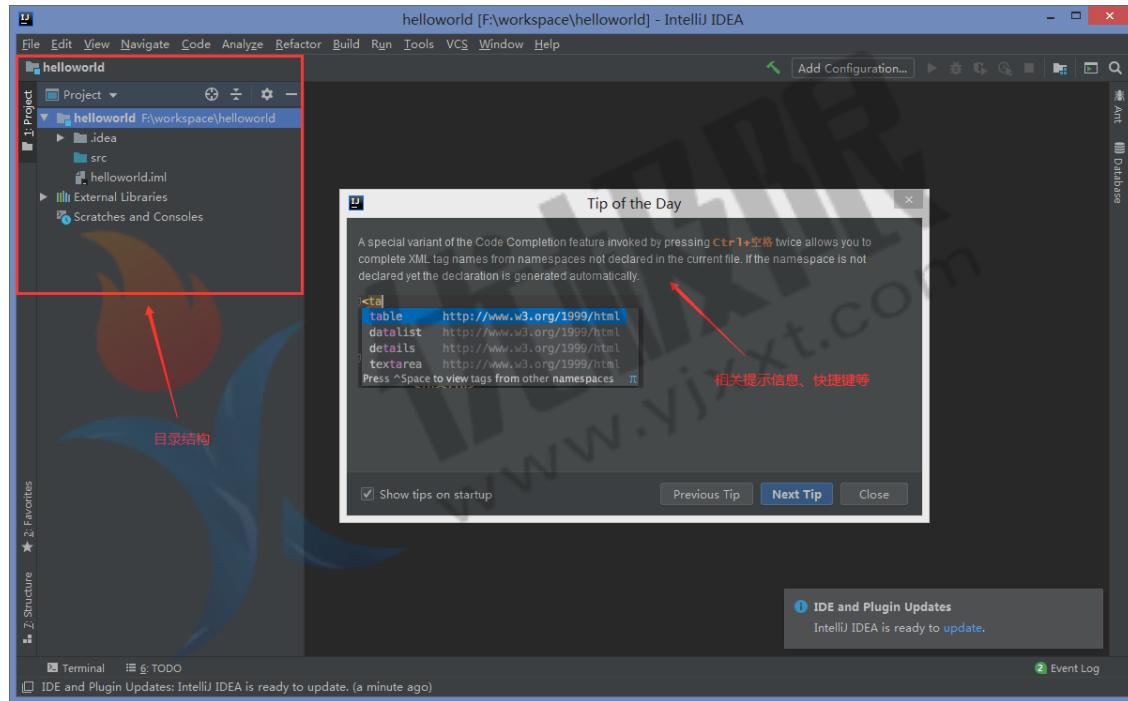
4. 选择 "Next"



5. 设置项目名称及工作空间

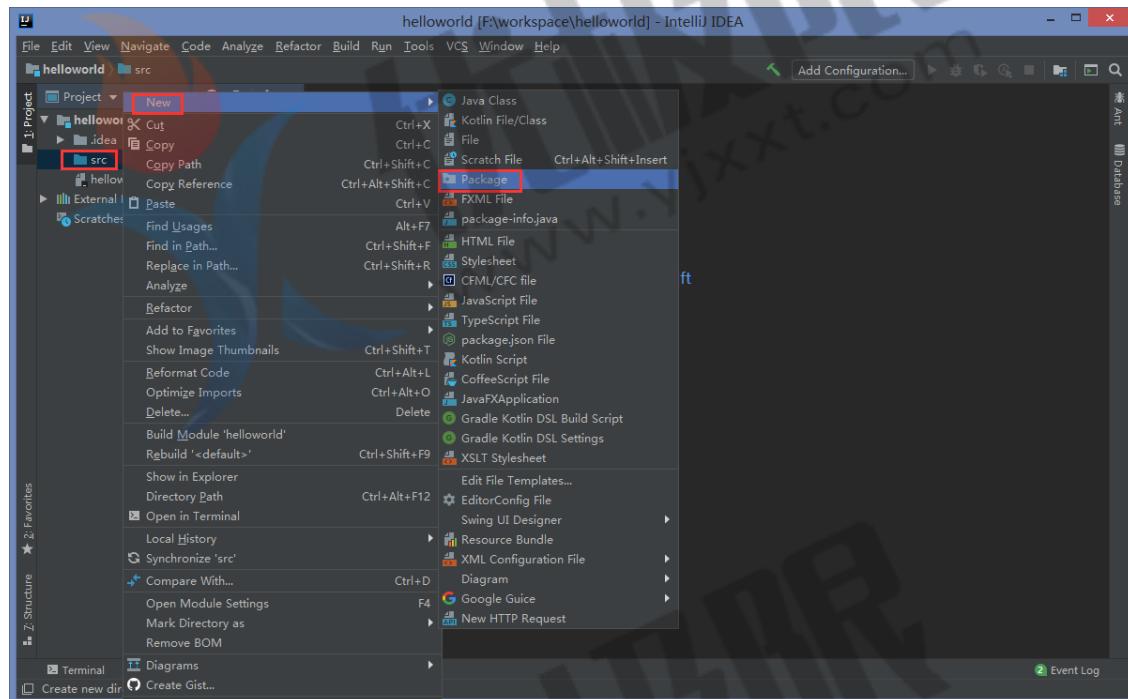


## 6. 项目目录结构及提示信息 (提示信息可选择"Close")

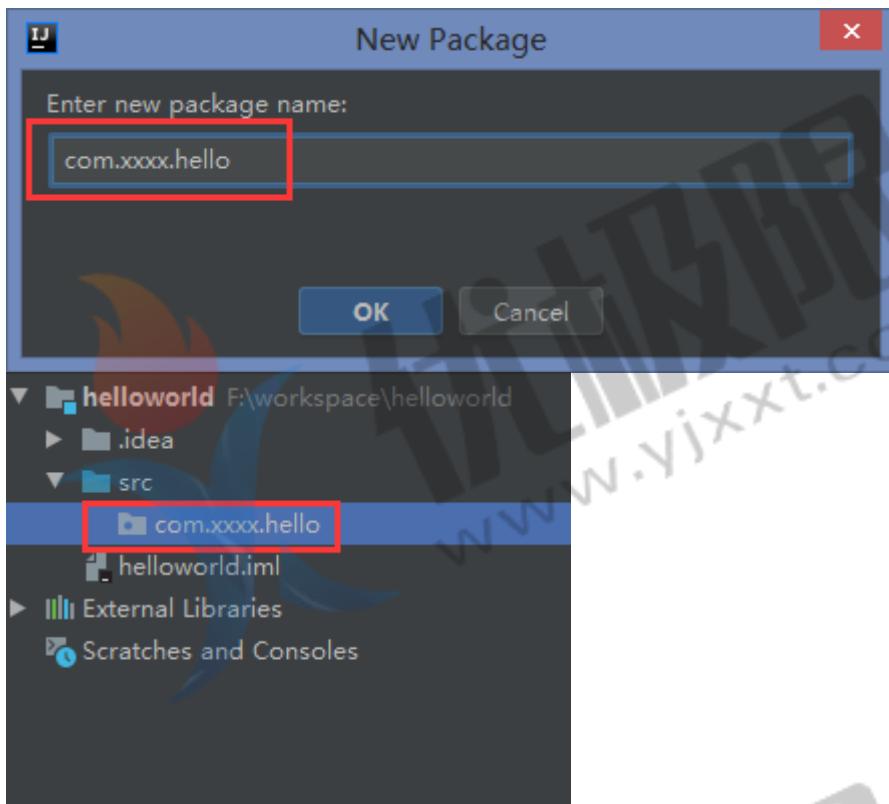


## 创建Java类

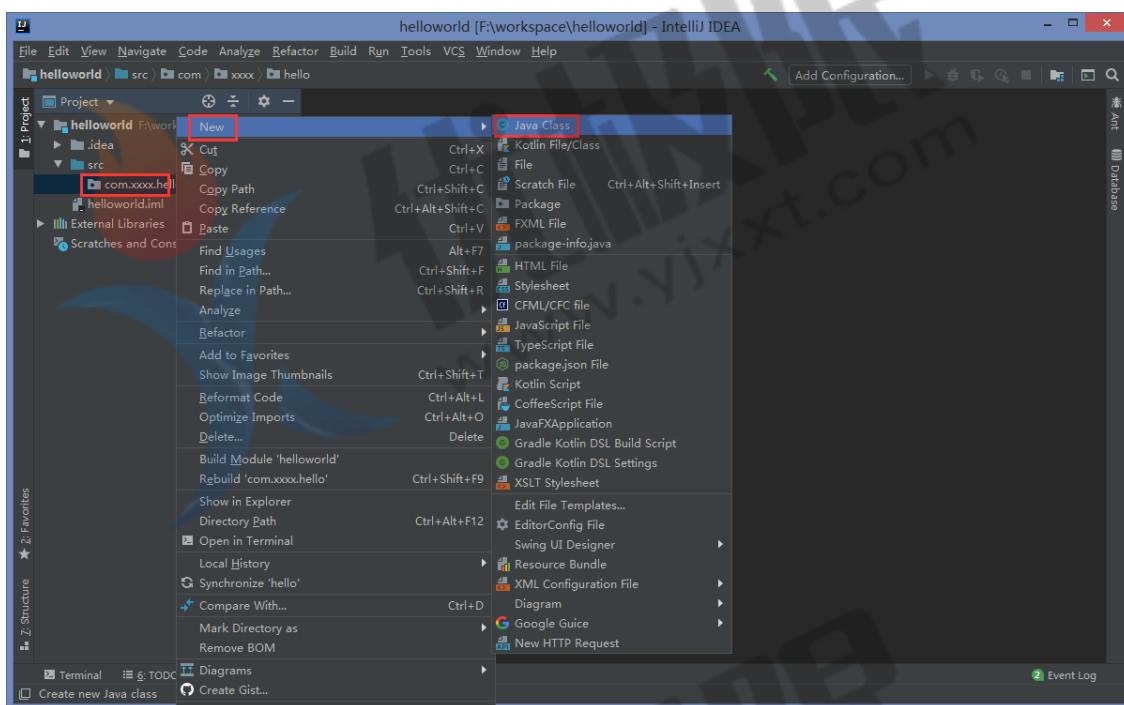
1. 点击 "src" —> "new" —> "package"， 创建一个文件包



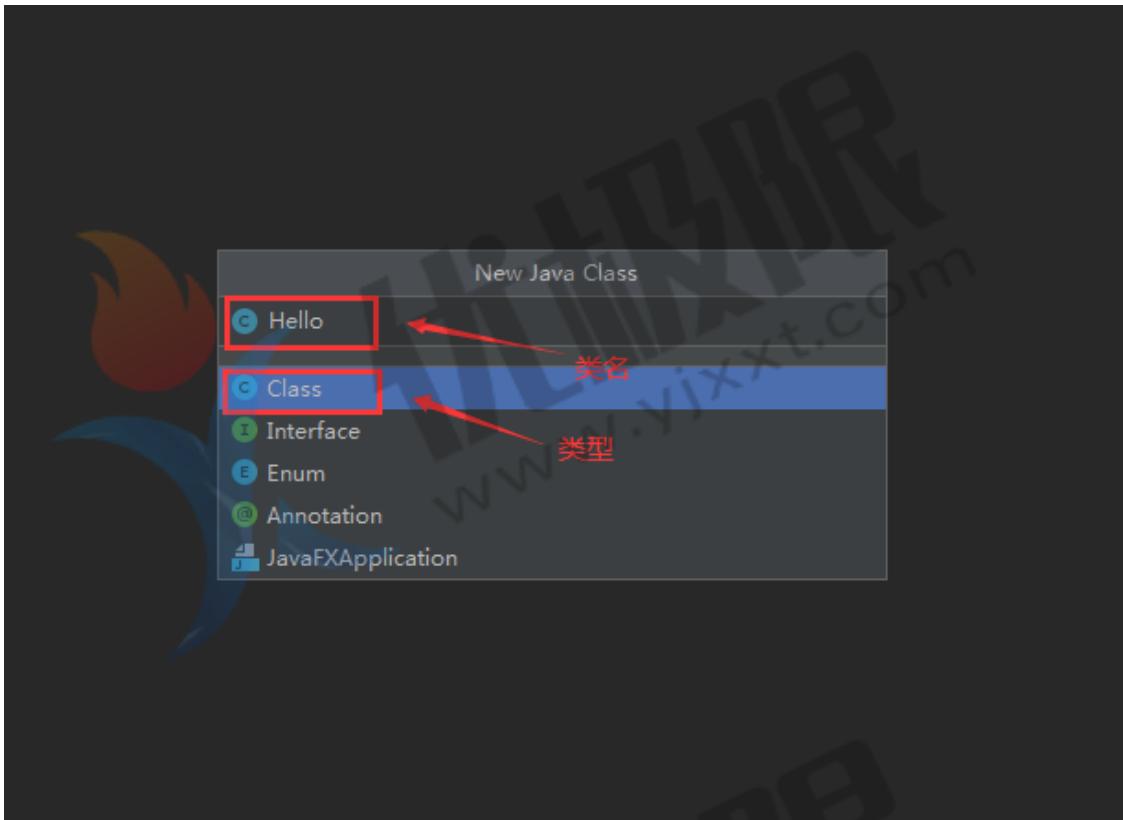
2. 设置包名，与Eclipse的包类似



3. 在包下面创建 Java 类文件，点击包名 —> "New" —> "Java Class"



4. 选择类型，并设置类的名称



5. 在类中写一个main方法

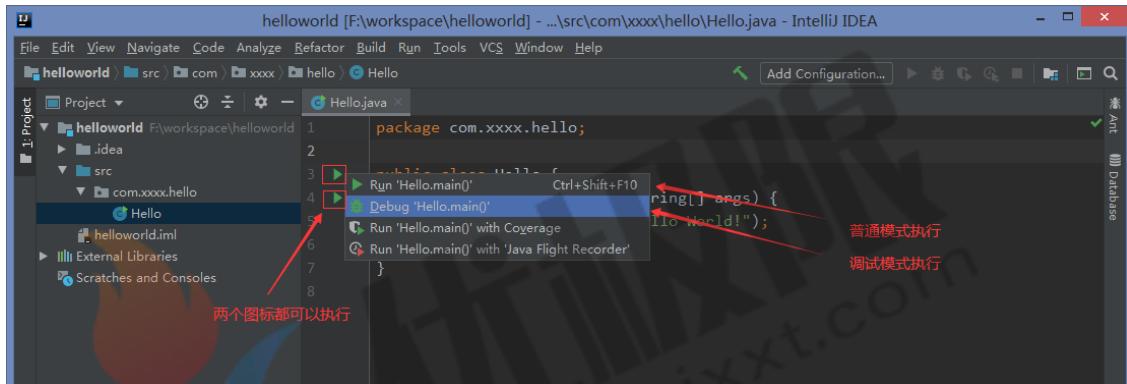
```
helloworld [F:\workspace\helloworld] - ...\\src\\com\\xxxx\\hello\\Hello.java - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
helloworld > src > com > xxxx > hello > Hello.java
Project helloworld F:\workspace\helloworld 1 package com.xxxx.hello;
2
3 public class Hello {
4     public static void main(String[] args) {
5         System.out.println("Hello World!");
6     }
7 }
8

Hello > main()

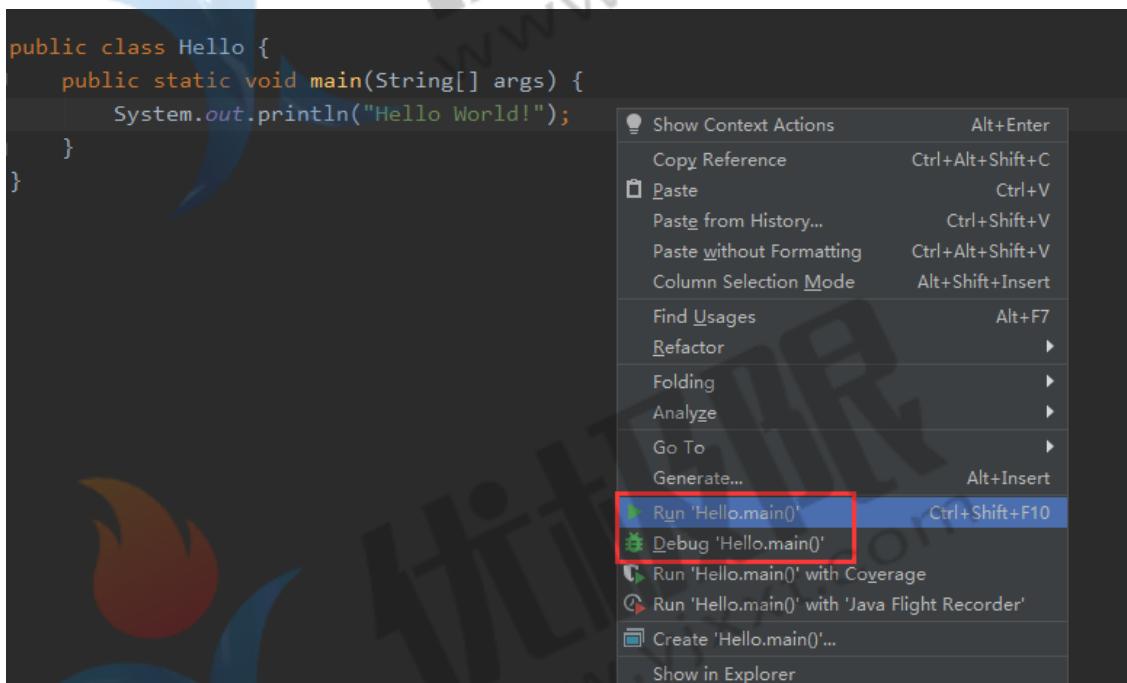
Event Log
5:44 CRLF UTF-8 4 spaces
IDE and Plugin Updates: IntelliJ IDEA is ready to update. (21 minutes ago)
```

执行main方法

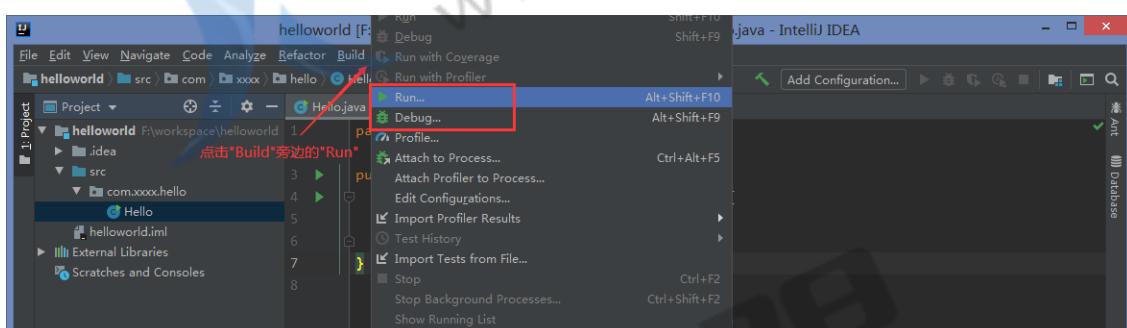
## 1. 方式一：代码左侧的绿色三角符号



## 2. 方式二：在类中右键，选择Run或Debug



## 3. 方式三：点击最上方菜单栏的Run



出现以下弹框，点击要运行的文件名，这里是 Hello



## 4. 运行结果

The screenshot shows the IntelliJ IDEA interface with a Java project named "helloworld". The code editor displays a single file, `Hello.java`, containing the classic "Hello World!" program. The run configuration "Hello" is selected in the "Run" tool window, and the output shows the application has started and printed "Hello World!". The status bar at the bottom indicates that all files are up-to-date.

```
package com.xxxx.hello;
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Run: Hello > main()

Process finished with exit code 0

All files are up-to-date (2 minutes ago)

## IDEA的基本设置

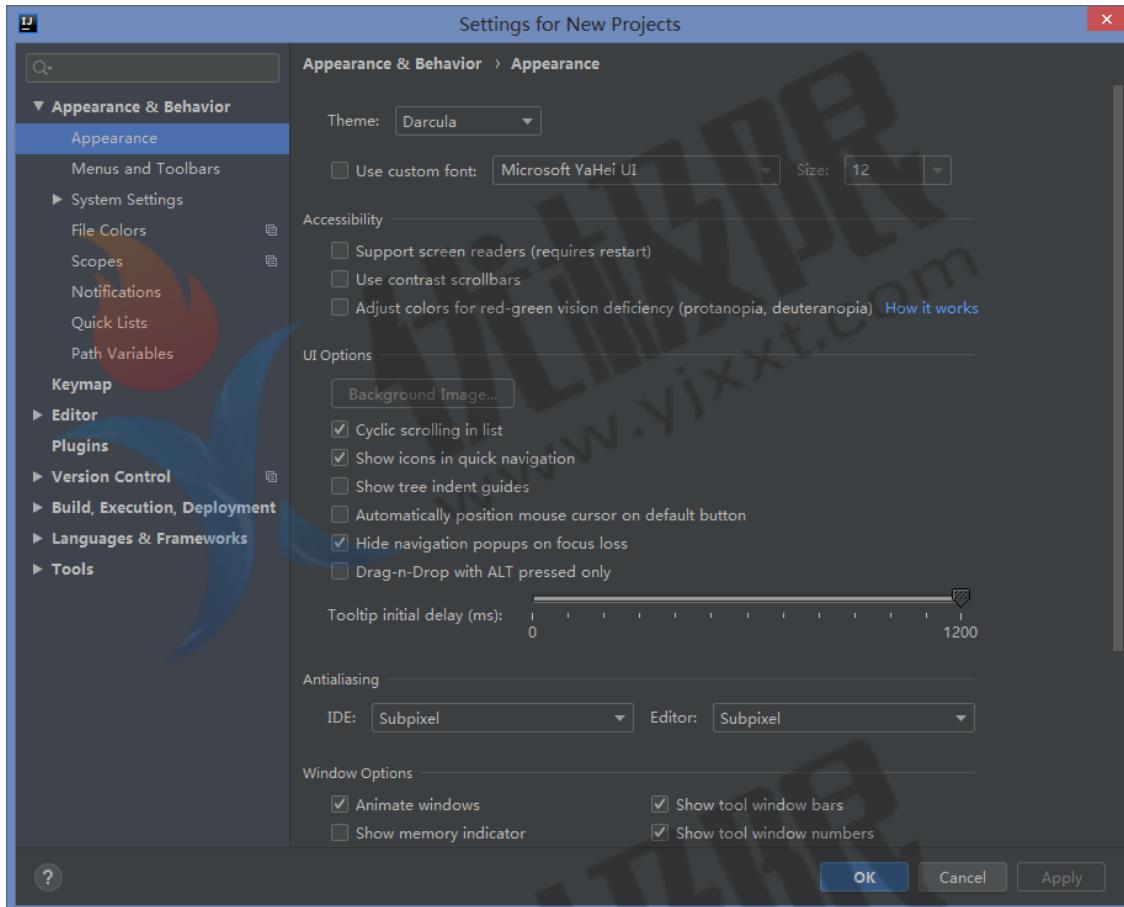
使用IDEA时，可以对它进行一些简单的设置，通过设置用户的偏好设置，可提高使用者的体验感。

## 进入设置页面

1. 选择右下角的 "Configure", 选择"Settings" (或在IDEA中, 选择左上角的 "File", 选择"Settings")

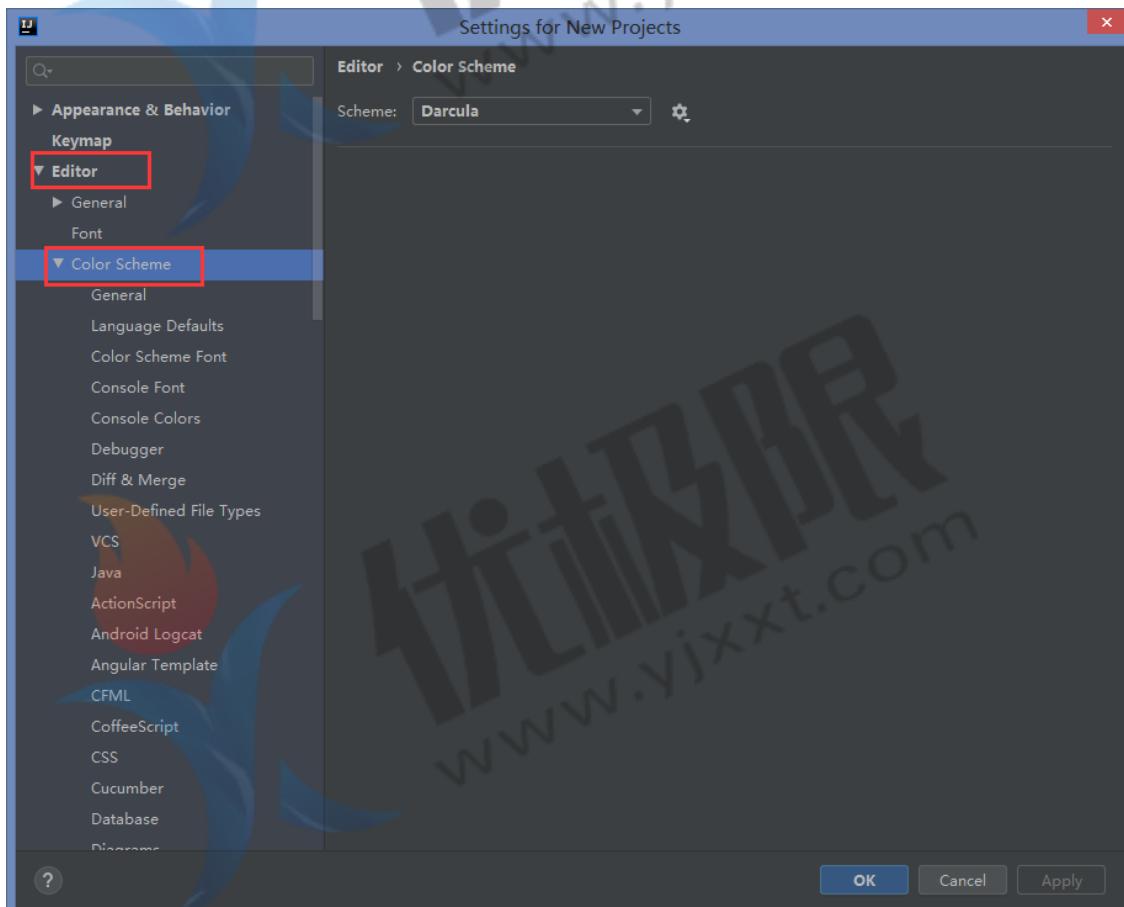


## 2. 进入设置页面

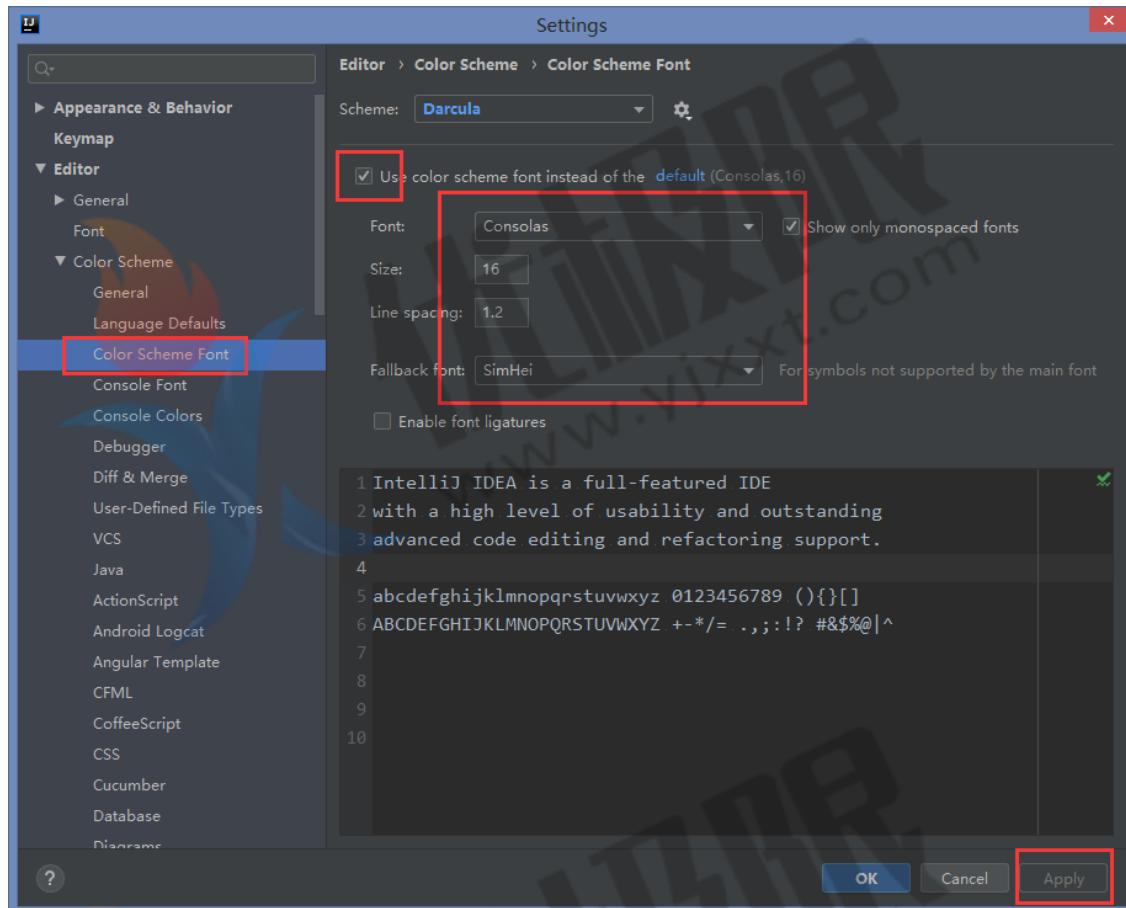


## 设置字体

1. 在Settings窗口中，点击 "Editor" —> "Color Scheme"



2. 选择 "Color Scheme Font", 设置字体风格和字体大小, 设置完之后选择 "Apply" 应用



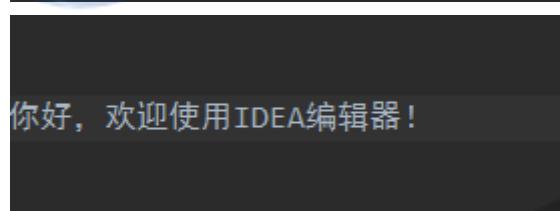
3. 设置中文字体

Idea更新2019.2后

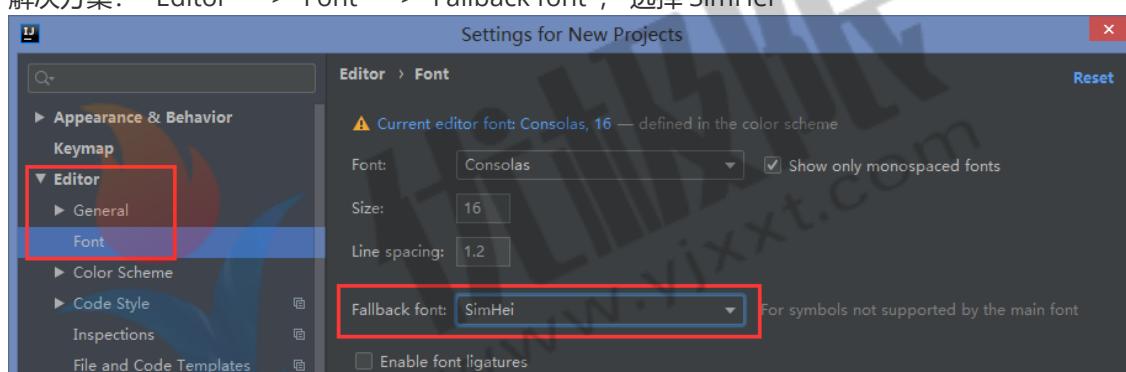
中文字体的默认效果:



修改字体之后的效果:

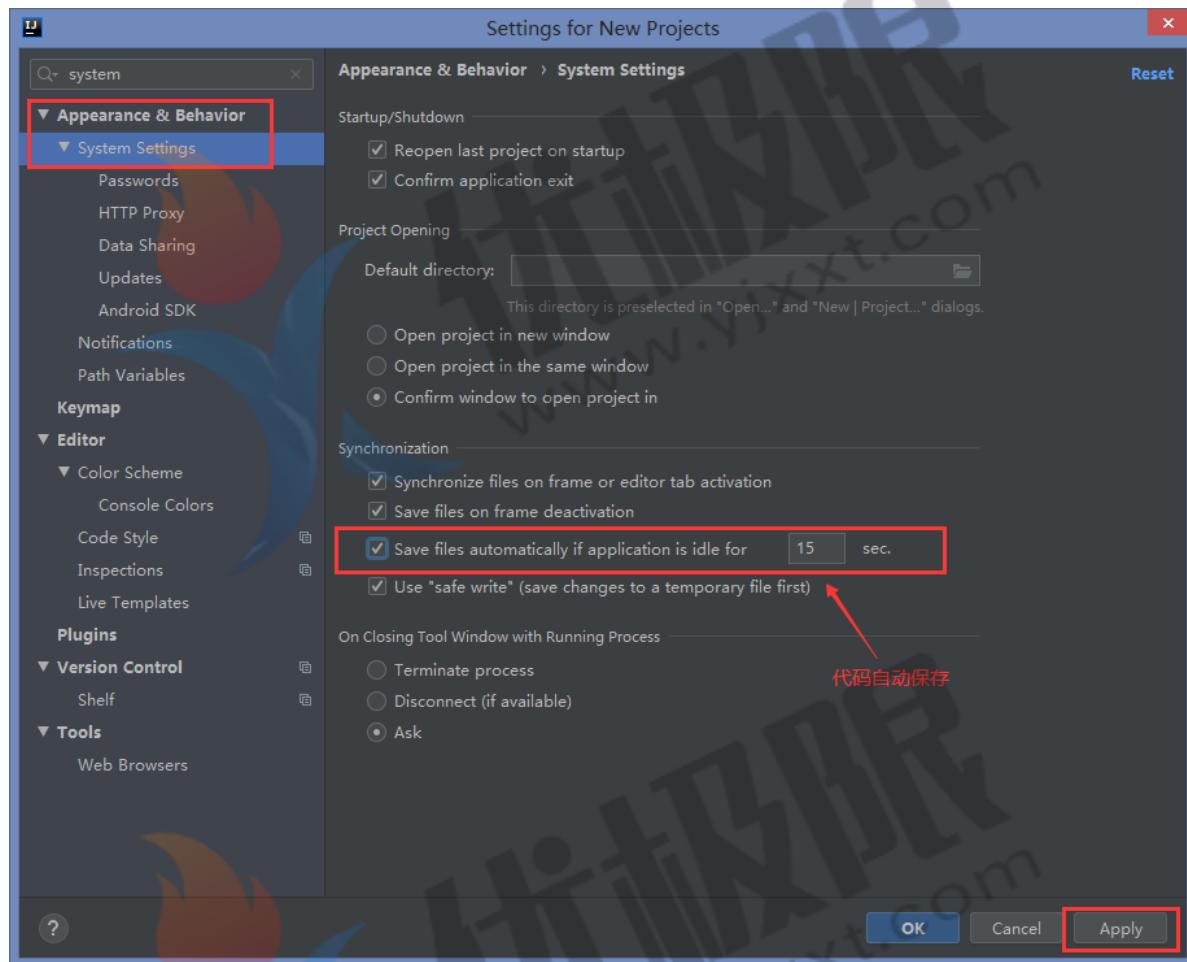


解决方案: "Editor" —> "Font" —> "Fallback font", 选择 SimHei



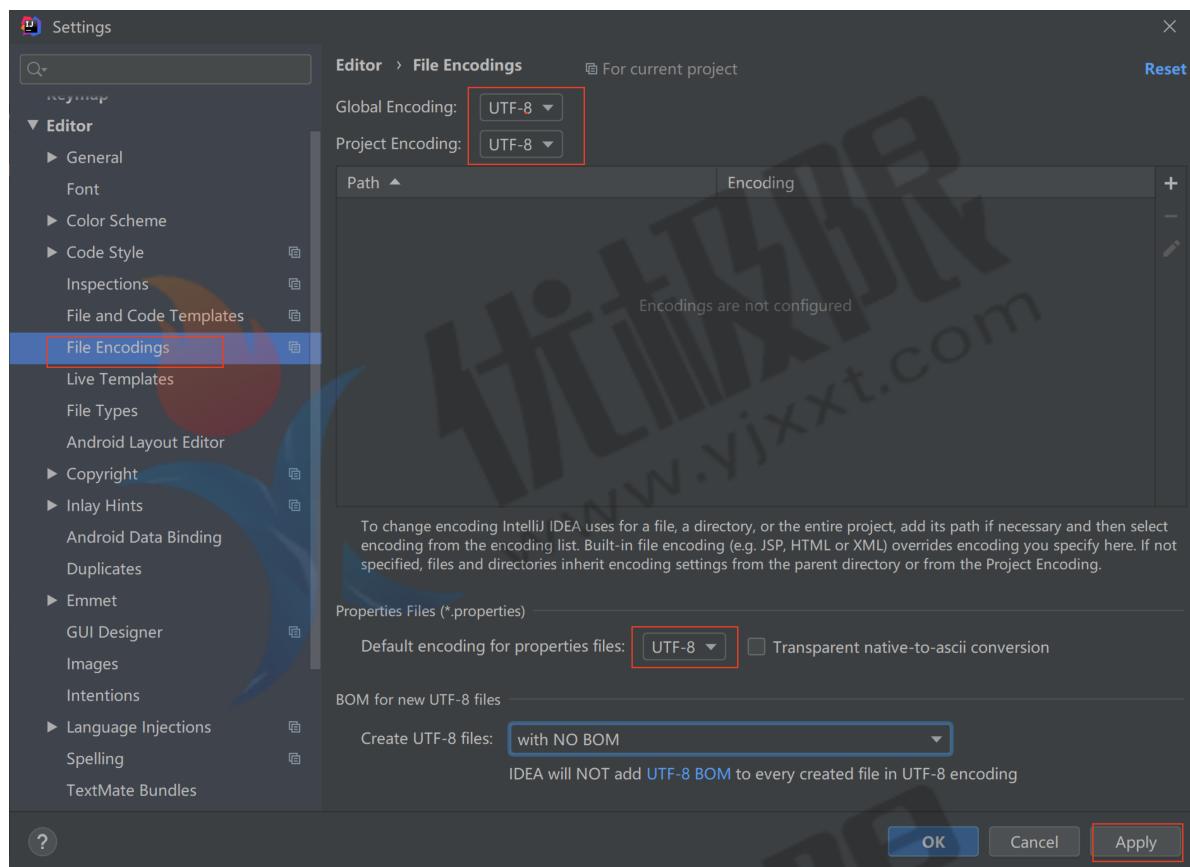
## 设置自动保存

选择 "Appearance & Behavior", 选择 "System Settings"



## 设置字体编码

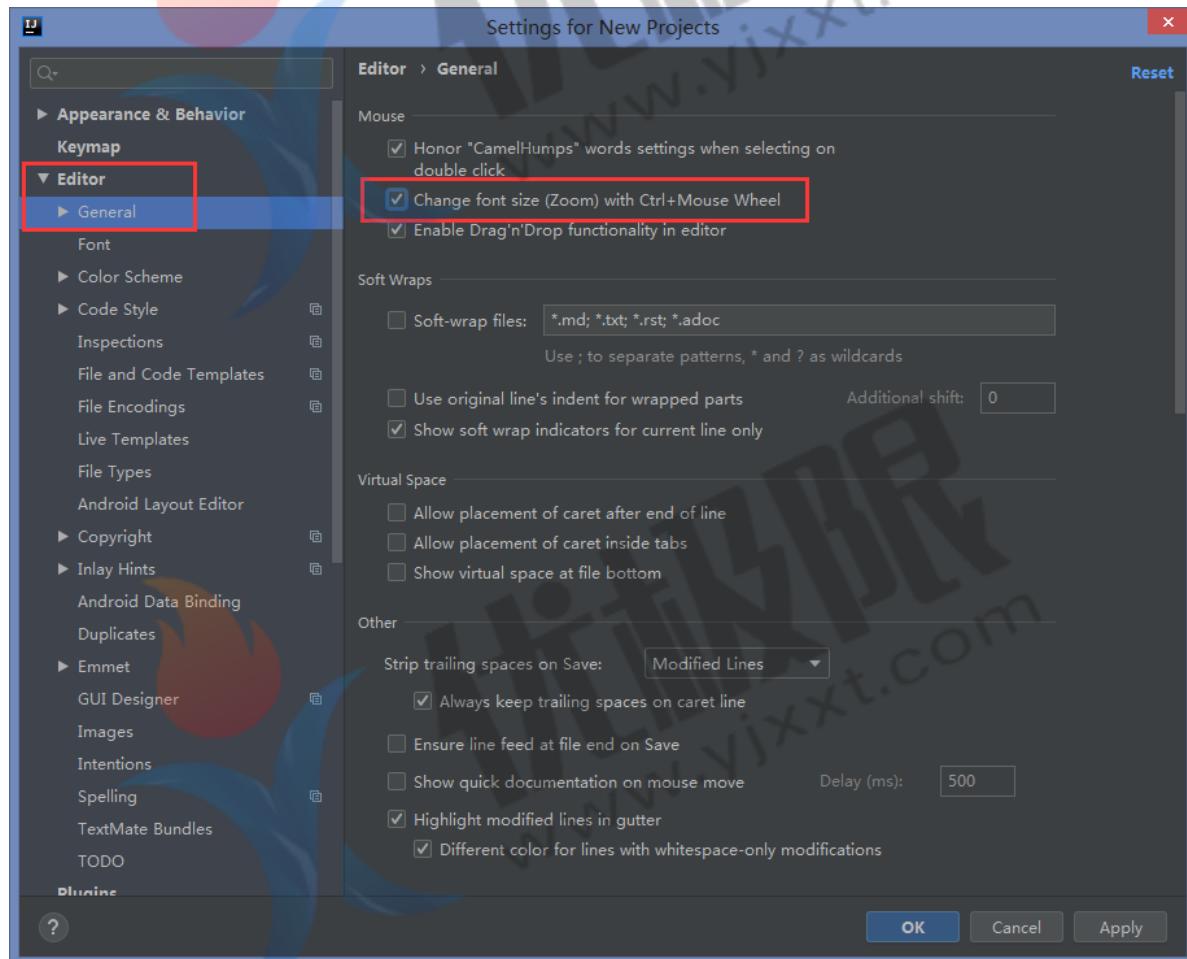
选择 "Editor", 选择 "File Encoding", 设置编码为 "UTF-8"



## 设置字体大小改变

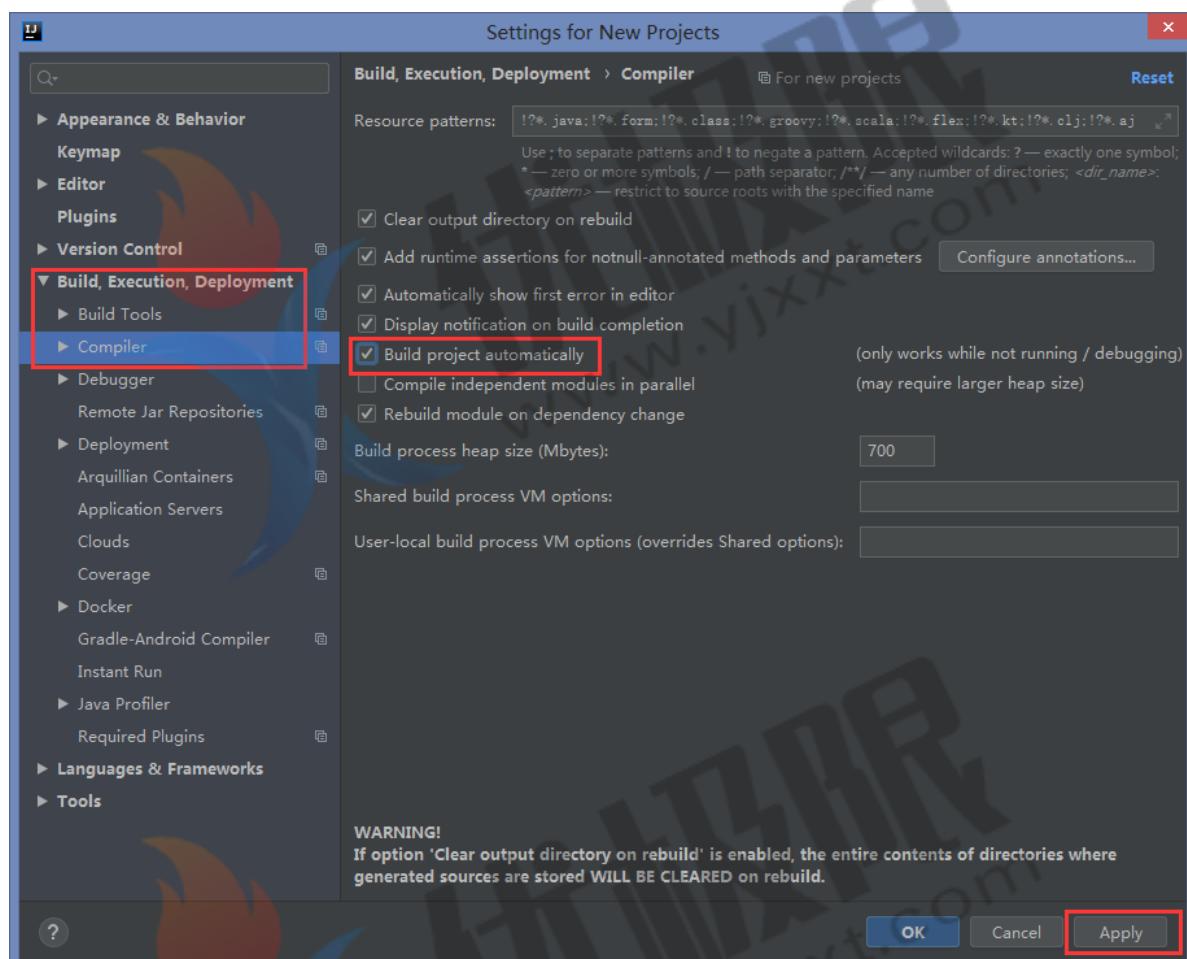
可以通过按住 "Ctrl"，滚动鼠标滚轮改变字体大小。

选择 "Editor"，选择 "General"



## 设置自动编译

选择 "Build,Execution,Deployment", 选择 "Compiler"



IDEA常用快捷键

快捷键	作用
Alt+Insert	生成代码 (如get, set方法, 构造函数等)
Alt+↑/ ↓	在方法间快速定位
Alt+【F3】	查找相同文本，并高亮显示
Ctrl+B	快速打开光标处的类或方法
Ctrl+J	自动代码(main方法)
Ctrl+N	查找类
Ctrl+Y	删除行
Ctrl+D	复制行
Ctrl+O	重写方法
Ctrl+E	最近打开的文件
Ctrl+F	查找文本
Ctrl+R	替换文本
Ctrl+P	方法参数提示
Ctrl+/	单行注释//
Ctrl+Shift+/	多行注释/* */
Ctrl+Shift+N	查找文件
Ctrl+Alt+L	格式化代码
Ctrl+Shift+↑/ ↓	代码向上/向下移动
Shift+F6	重构-重命名

## HTTP协议

HTTP 协议 (Hypertext Transfer Protocol, 超文本传输协议) , 是一个客户端请求和响应的标准协议, 这个协议详细规定了浏览器和万维网服务器之间互相通信的规则。用户输入地址和端口号之后就可以从服务器上取得所需要的网页信息。

通信规则规定了客户端发送给服务器的内容格式, 也规定了服务器发送给客户端的内容格式。客户端发送给服务器的格式叫“**请求协议**”; 服务器发送给客户端的格式叫“**响应协议**”。

在浏览器中 F12 可查看

The screenshot shows the Firefox developer tools Network tab with the Headers section selected. It displays two sections: Response Headers (16) and Request Headers (10). Red arrows point from the text '请求头' (Request Headers) to the 'Request Headers (10)' section and from the text '响应头' (Response Headers) to the 'Response Headers (16)' section.

## 浏览器中的书写格式

服务器端资源需要通过浏览器进行，此时由浏览器将我们给出的请求解析为满足 HTTP 协议的格式并发出。我们发出的请求格式需要按照浏览器规定的格式来书写，在浏览器中书写格式如下：



当浏览器获取到信息以后，按照特定格式解析并发送即可。接收到服务器端给出的响应时，也按照 HTTP 协议进行解析获取到各个数据，最后按照特定格式展示给用户。

## HTTP协议的特点

1. 支持客户/服务器模式。
2. 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、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。
3. 灵活：HTTP 允许传输任意类型的数据对象。传输的类型由Content-Type加以标记。
4. 无连接：无连接是表示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。

HTTP1.1 版本后支持可持续连接。通过这种连接，就有可能在建立一个 TCP 连接后，发送请求并得到回应，然后发送更多的请求并得到更多的回应。通过把建立和释放 TCP 连接的开销分摊到多个请求上，则对于每个请求而言，由于 TCP 而造成的相对开销被大大地降低了。而且，还可以发送流水线请求，也就是说在发送请求 1 之后的回应到来之前就可以发送请求 2。也可以认为，一次连接发送多个请求，由客户机确认是否关闭连接，而服务器会认为这些请求分别来自不同的客户端。

5. 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（超文本传输协议）是一个基于请求与响应模式的、应用层的协议，常基于 TCP 的连接方式，绝大多数的 Web 开发，都是构建在 HTTP 协议之上的 Web 应用。

HTTP URL (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)的格式如下：

`http://host[:port]/[abc_path]`

`http://IP(主机名/域名):端口/访问的资源路径`

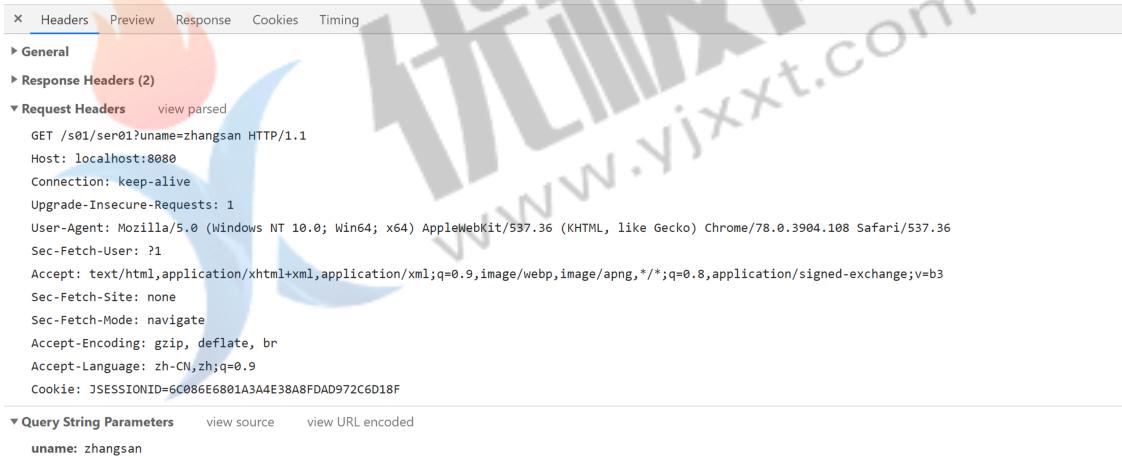
- http 表示要通过 HTTP 协议来定位网络资源；
- host 表示合法的 Internet 主机域名或者 IP 地址；
- port 指定一个端口号，为空则使用缺省端口 80；
- abs\_path 指定请求资源的 URI；如果 URL 中没有给出 abs\_path，那么当它作为请求 URI 时，必须以“/”的形式给出，通常这个工作浏览器自动帮我们完成。

## HTTP 请求

HTTP 请求由三部分组成，分别是：请求行、请求头、请求正文。

通过chrome浏览器， F12 —> Network查看。

### 1. Get 请求 (没有请求体)



The screenshot shows the Network tab in the Chrome DevTools developer tools. A single network request is listed:

- Request Headers**:
  - GET /s01/ser01?uname=zhangsan HTTP/1.1
  - Host: localhost:8080
  - Connection: keep-alive
  - Upgrade-Insecure-Requests: 1
  - User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36
  - Sec-Fetch-User: ?1
  - Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3
  - Sec-Fetch-Site: none
  - Sec-Fetch-Mode: navigate
  - Accept-Encoding: gzip, deflate, br
  - Accept-Language: zh-CN,zh;q=0.9
- Query String Parameters**:
  - uname: zhangsan

### 2. Post 请求

X Headers Preview Response Cookies Timing

General

Response Headers (2)

Request Headers view parsed

```
POST /s01/ser01 HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Content-Length: 14
Cache-Control: max-age=0
Origin: http://localhost:8080
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36
Sec-Fetch-User: ?1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Referer: http://localhost:8080/s01/index.jsp
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9
Cookie: JSESSIONID=C8F1ADD271D601C8435C541139F546F7
```

Form Data view source view URL encoded 请求正文

uname: zhangsan

## 格式

请求行  
请求头1  
请求头2  
...  
请求空行  
请求体

请求行以一个方法符号开头，以空格分开，后面跟着请求的 URI 和协议的版本。

格式如下：Method Request-URI HTTP-Version CRLF

Method 表示请求方法；

Request-URI 是一个统一资源标识符；

HTTP-Version 表示请求的 HTTP 协议版本；

CRLF 表示回车和换行；

## HTTP 响应

在接收和解释请求消息后，服务器返回一个 HTTP 响应消息。HTTP 响应也是由三个部分组成，分别是：状态行、消息报头、响应正文。

Response Headers view parsed

```
HTTP/1.1 200 OK
Bdpagetype: 2
Bdqid: 0xb366b9ab0066b9a1
Cache-Control: private
Connection: keep-alive
Content-Encoding: gzip
Content-Type: text/html;charset=utf-8
```

## 格式

状态行  
响应头1  
响应头2  
...  
响应空行  
响应体

## 消息头

HTTP 消息由客户端到服务器的请求和服务器到客户端的响应组成。请求消息和响应消息都是由开始行（对于请求消息，开始行就是请求行，对于响应消息，开始行就是状态行），消息报头（可选），空行（只有 CRLF 的行），消息正文（可选）组成。

每一个报头域都是由 **名字":"+空格+值** 组成，消息报头域的名字是大小写无关的。

### 请求头

请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息。

- **Referer**: 该请求头指明请求从哪里来。

如果是地址栏中输入地址访问的都没有该请求头 地址栏输入地址，通过请求可以看到，此时多了一个 Referer 的请求头，并且后面的值为该请求从哪里发出。比如：百度竞价，只能从百度来的才有效果，否则不算；通常用来做统计工作、防盗链。

### 响应头

响应报头允许服务器传递不能放在状态行中的附加响应信息，以及关于服务器的信息和对 Request-URI 所标识的资源进行下一步访问的信息。

- **Location**: Location 响应报头域用于重定向接受者到一个新的位置。

Location 响应报头域，常用在更换域名的时候。

```
response.sendRedirect("http://www.baidu.com");
```

- **Refresh**: 自动跳转（单位是秒），可以在页面通过meta标签实现，也可在后台实现。

```
<meta http-equiv="refresh" content="3;url=http://www.baidu.com">
```

## Tomcat 服务器

### 什么是 Tomcat

Tomcat 是一个符合 JavaEE WEB 标准的最小的 WEB 容器，所有的 JSP 程序一定要有 WEB 容器的支持才能运行，而且在给定的 WEB 容器里面都会支持事务处理操作。

Tomcat 是由 Apache 提供的 ([www.apache.org](http://www.apache.org)) 提供的可以用安装版和解压版，安装版可以在服务中出现一个 Tomcat 的服务，免安装没有，开发中使用免安装版。Tomcat 简单的说就是一个运行 Java 的网络服务器，**底层是 Socket 的一个程序**，它也是 JSP 和 Servlet 的一个容器。Tomcat 是 Apache 软件基金会 (Apache Software Foundation) 的 Jakarta 项目中的一个核心项目，由 Apache、Sun 和其他一些公司及个人共同开发而成。

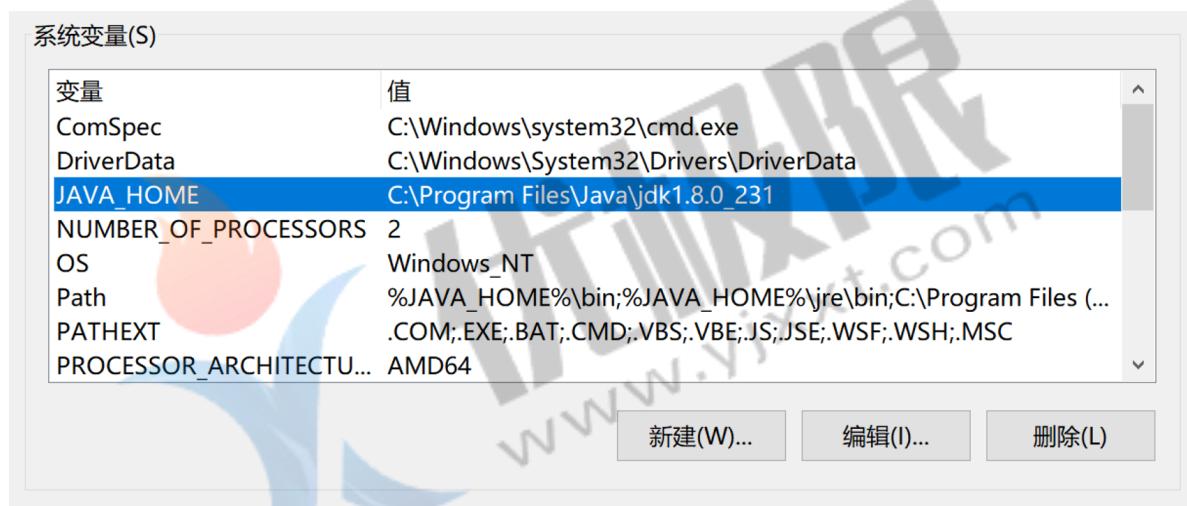
由于有了 Sun 的参与和支持，最新的 Servlet 和 JSP 规范总是能在 Tomcat 中得到体现。因为 Tomcat 技术先进、性能稳定，而且免费，因而深受 Java 爱好者的喜爱并得到了部分软件开发商的认可，成为目前比较流行的 **Web 应用服务器**。

Tomcat 服务器是一个免费的开放源代码的 Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的情况下被普遍使用，是开发和调试 JSP 程序的首选。对于一个初学者来说，可以这样认为，当在一台机器上配置好 Apache 服务器，可利用它响应 HTML（标准通用标记语言下的一个应用）页面的访问请求。实际上 Tomcat 部分是 Apache 服务器的扩展，但它是独立运行的，所以当你运行 tomcat 时，它实际上作为一个与 Apache 独立的进程单独运行的。

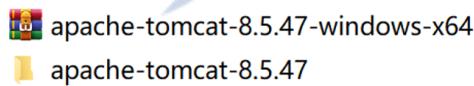
当配置正确时，Apache 为 HTML 页面服务，而 Tomcat 实际上是在运行 JSP 页面和 Servlet。另外，Tomcat 和 IIS 等 Web 服务器一样，具有处理 HTML 页面的功能，另外它还是一个 Servlet 和 JSP 容器，独立的 Servlet 容器是 Tomcat 的默认模式。不过，Tomcat 处理静态 HTML 的能力不如 Apache 服务器。目前 Tomcat 最新版本为 9.0。

## 安装Tomcat

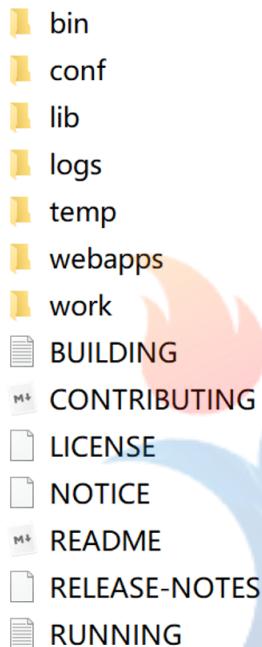
运行 Tomcat 需要 JDK 的支持【Tomcat 会通过 **JAVA\_HOME** 找到所需要的 JDK】。安装就是解压缩过程。启动 Tomcat，能访问则算安装好了



1. 解压Tomcat8的压缩包



2. 解压后目录结构



3. 启动 Tomcat (在 tomcat 的安装目录下的 bin 目录 使用命令行启动 tomcat)

方式一：双击脚本文件启动



Windows 批处理文件

方式二：使用脚本命令启动

```
C:\software\Tomcat\apache-tomcat-8.5.47\bin>startup.bat
Using CATALINA_BASE:   "C:\software\Tomcat\apache-tomcat-8.5.47"
Using CATALINA_HOME:   "C:\software\Tomcat\apache-tomcat-8.5.47"
Using CATALINA_TMPDIR: "C:\software\Tomcat\apache-tomcat-8.5.47\temp"
Using JRE_HOME:        "C:\Program Files\Java\jdk1.8.0_231"
Using CLASSPATH:       "C:\software\Tomcat\apache-tomcat-8.5.47\bin\bootstrap.jar;C:\software\Tomcat\apache-tomcat-8.5.47\bin\tomcat-juli.jar"
C:\software\Tomcat\apache-tomcat-8.5.47\bin>
```

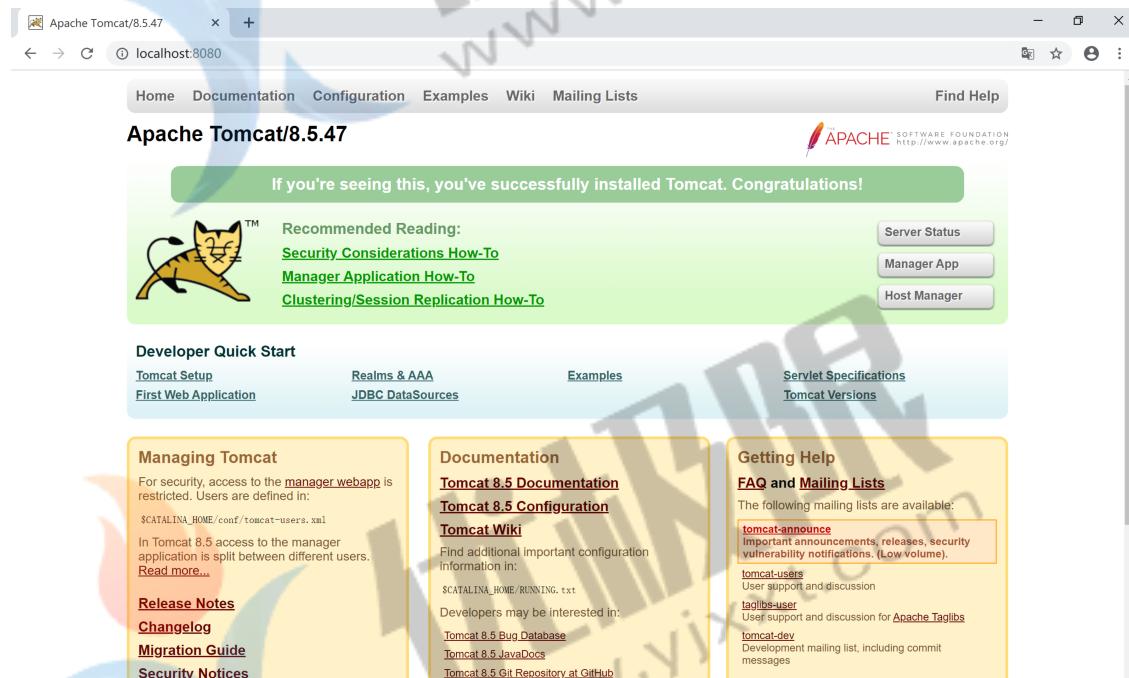
4. 服务器启动成功

```
Tomcat
27-Feb-2020 04:53:21.355 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [C:\software\Tomcat\apache-tomcat-8.5.47\webapps\examples] has finished in [356] ms
27-Feb-2020 04:53:21.357 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 把web 应用程序部署到目录 [C:\software\Tomcat\apache-tomcat-8.5.47\webapps\host-manager]
27-Feb-2020 04:53:21.399 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [C:\software\Tomcat\apache-tomcat-8.5.47\webapps\host-manager] has finished in [42] ms
27-Feb-2020 04:53:21.403 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 把web 应用程序部署到目录 [C:\software\Tomcat\apache-tomcat-8.5.47\webapps\manager]
27-Feb-2020 04:53:21.439 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [C:\software\Tomcat\apache-tomcat-8.5.47\webapps\manager] has finished in [36] ms
27-Feb-2020 04:53:21.442 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory 把web 应用程序部署到目录 [C:\software\Tomcat\apache-tomcat-8.5.47\webapps\ROOT]
27-Feb-2020 04:53:21.465 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [C:\software\Tomcat\apache-tomcat-8.5.47\webapps\ROOT] has finished in [23] ms
27-Feb-2020 04:53:21.470 信息 [main] org.apache.coyote.AbstractProtocol.start
开始协议处理句柄["http-nio-8080"]
27-Feb-2020 04:53:21.484 信息 [main] org.apache.coyote.AbstractProtocol.start
开始协议处理句柄["ajp-nio-8009"]
27-Feb-2020 04:53:21.488 信息 [main] org.apache.catalina.startup.Catalina.start
Server startup in 778 ms
```

注:

1. Tomcat默认占用端口8080。（注意端口冲突问题）
2. 如果需要使用服务器，启动成功后，该启动窗口不要关闭。

## 5. 打开浏览器，输入<http://localhost:8080> 访问



## 6. 调用 shutdown 命令关闭Tomcat服务器

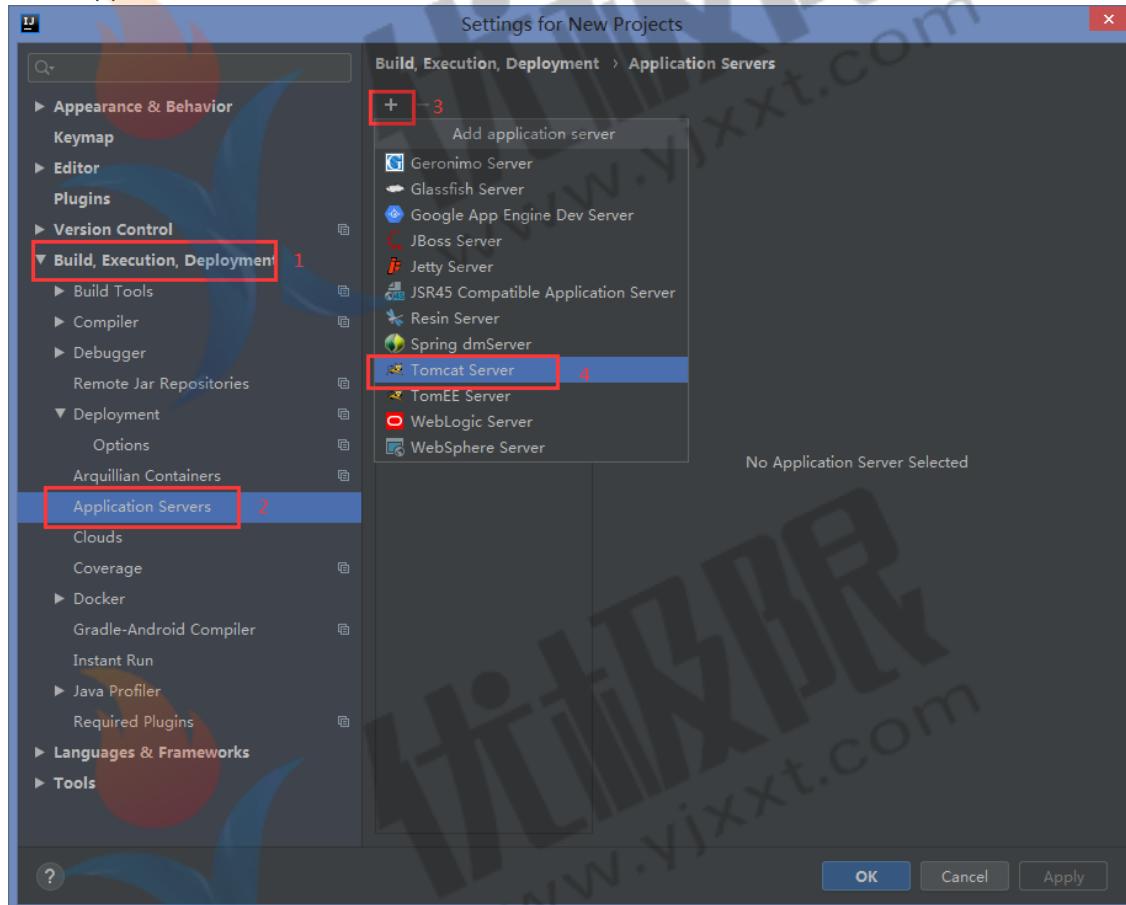
## Tomcat目录结构



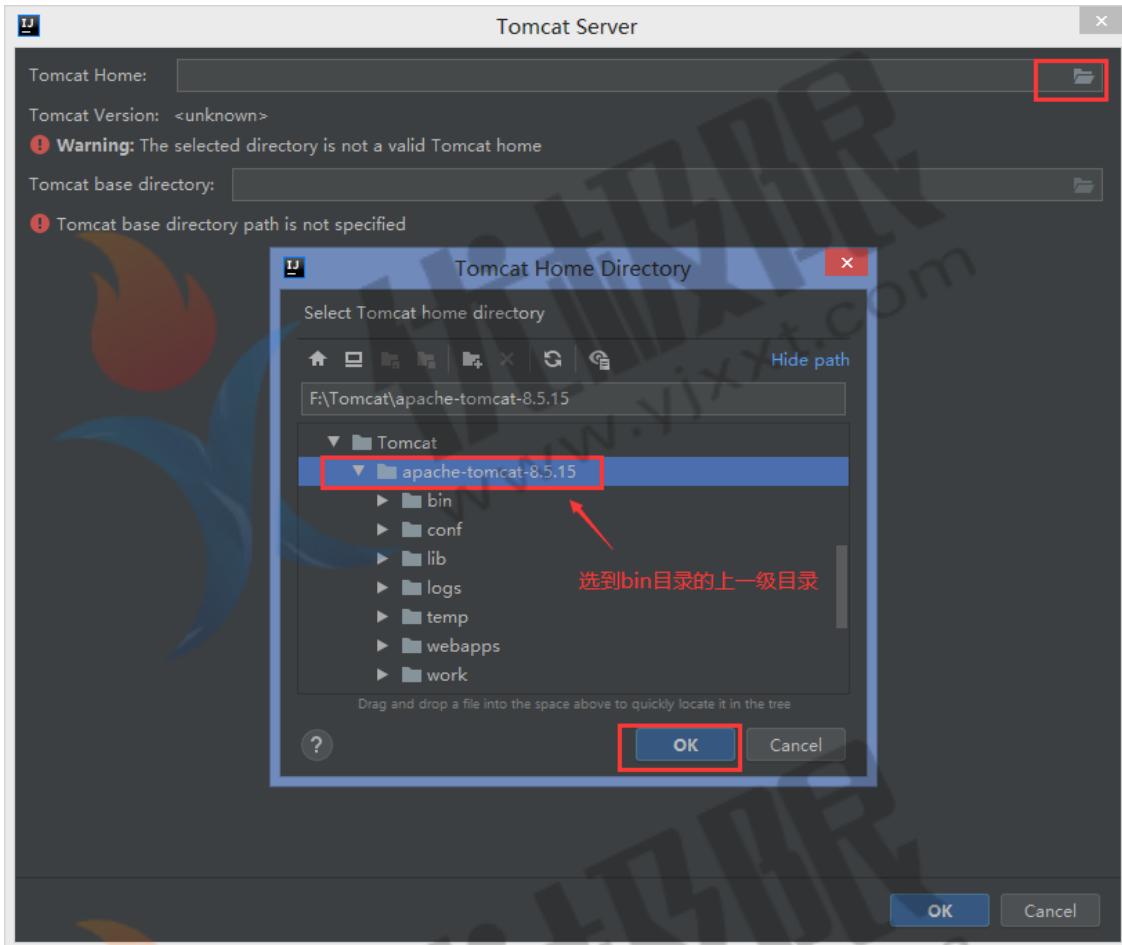
1. bin: 启动和关闭 tomcat 的 bat 文件
2. conf: 配置文件server.xml 该文件用于配置 server 相关的信息，比如 tomcat 启动的端口号，配置主机(Host)； web.xml 文件配置与 web 应用 (web 应用相当于一个 web站点)； tomcat-user.xml 配置用户名密码和相关权限
3. lib: 该目录放置运行 tomcat 需要的 jar 包
4. logs: 存放日志，当我们需要查看日志的时候，可以查询信息
5. webapps: 放置我们的 web 应用
6. work 工作目录：该目录用于存放 jsp 被访问后生成对应的 server 文件和.class 文件

## IDEA配置Tomcat

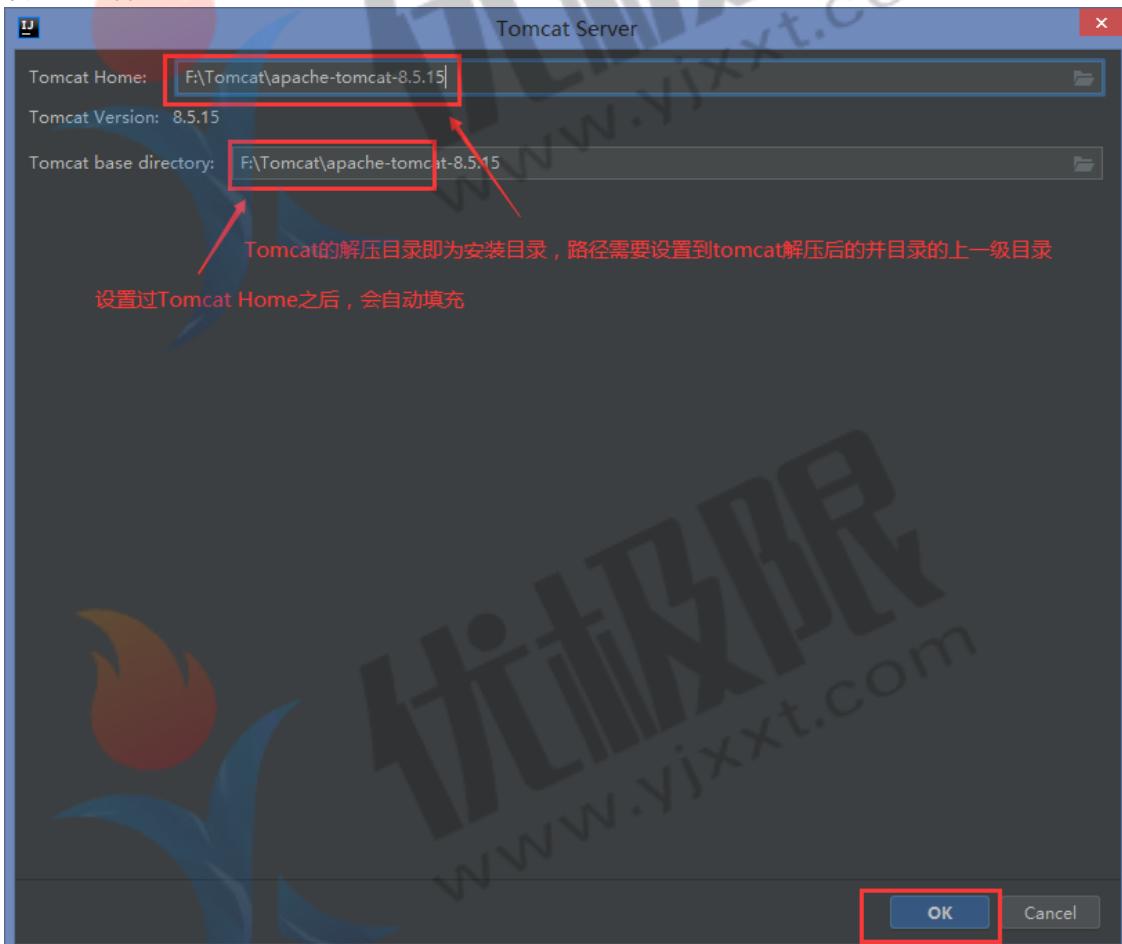
1. 选择 "Application Servers"，点击右侧的 "+" 号，选择 "Tomcat Server"



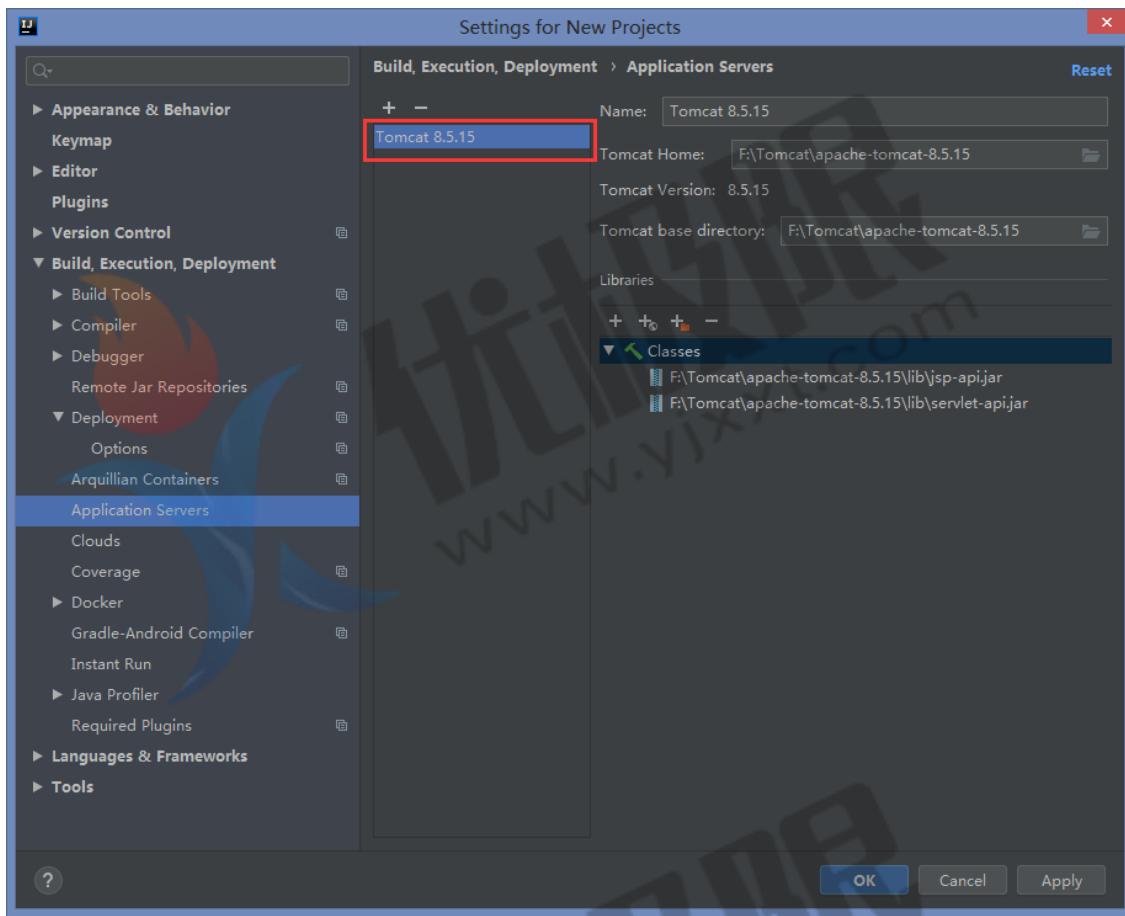
## 2. 设置 Tomcat 的安装目录



设置好之后



3. 配置Tomcat服务器完成



## Servlet的实现

Servlet 是 Server 与 Applet 的缩写，是服务端小程序的意思。使用 Java 语言编写的服务器端程序，可以像生成动态的 WEB 页，Servlet 主要运行在服务器端，并由服务器调用执行，是一种按照 Servlet 标准来开发的类。是 SUN 公司提供的一门用于开发动态 Web 资源的技术。（言外之意：要实现 web 开发，需要实现 Servlet 标准）

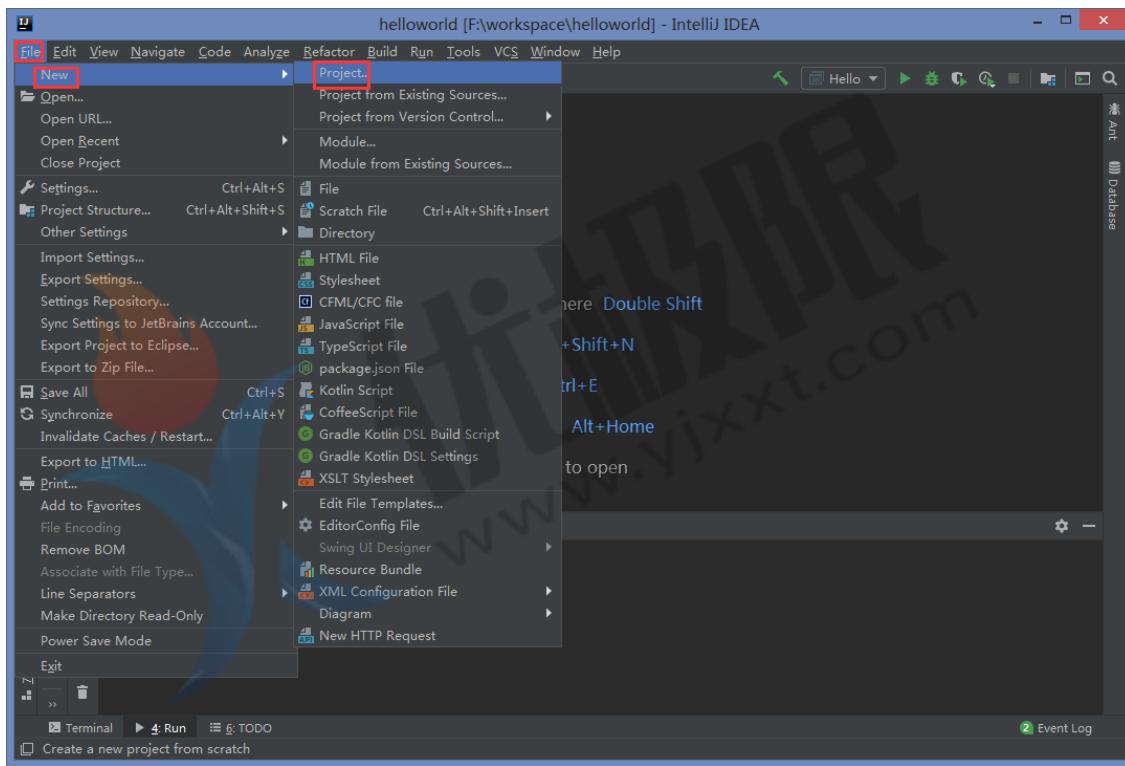
Servlet 本质上也是 Java 类，但要遵循 Servlet 规范进行编写，没有 main()方法，它的创建、使用、销毁都由 Servlet 容器进行管理(如 Tomcat)。（言外之意：写自己的类，不用写 main 方法，别人自动调用）

Servlet 是和 HTTP 协议是紧密联系的，其可以处理 HTTP 协议相关的所有内容。这也是 Servlet 应用广泛的原因之一。

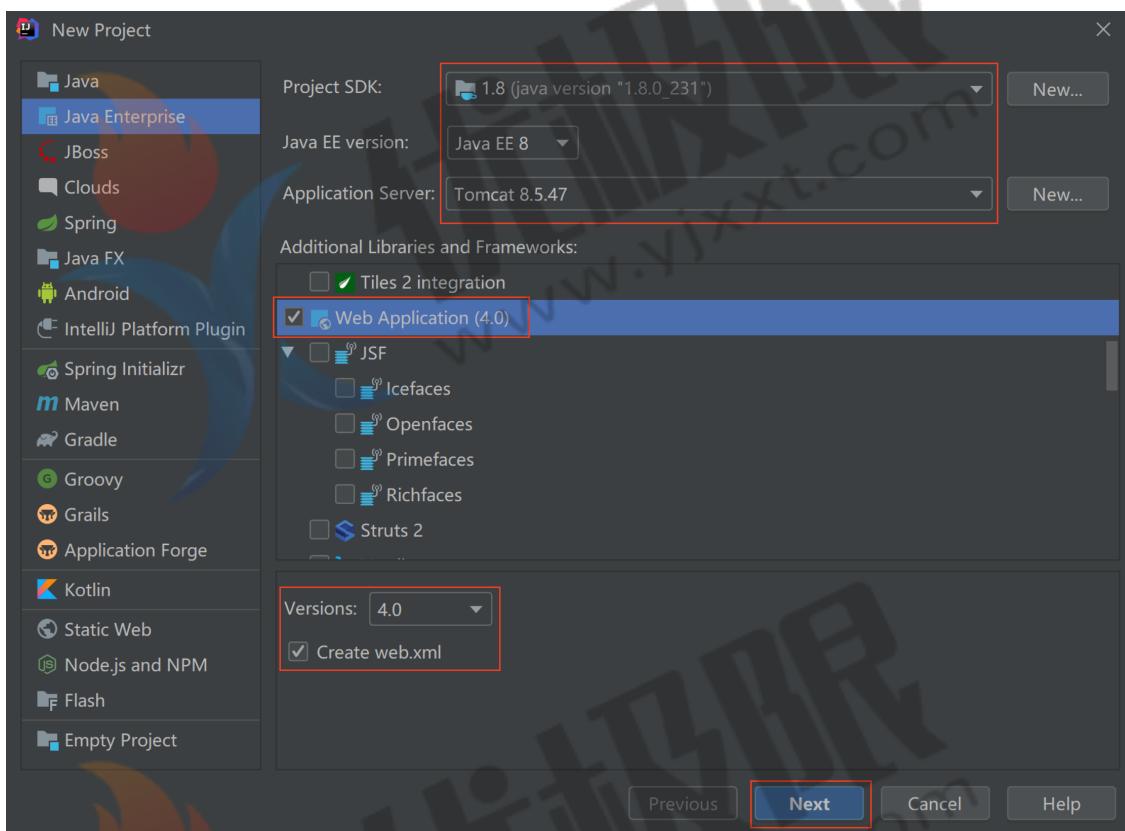
提供了 Servlet 功能的服务器，叫做 Servlet 容器，其常见容器有很多，如 Tomcat, Jetty, WebLogic Server, WebSphere, JBoss 等等。

## 创建Web项目

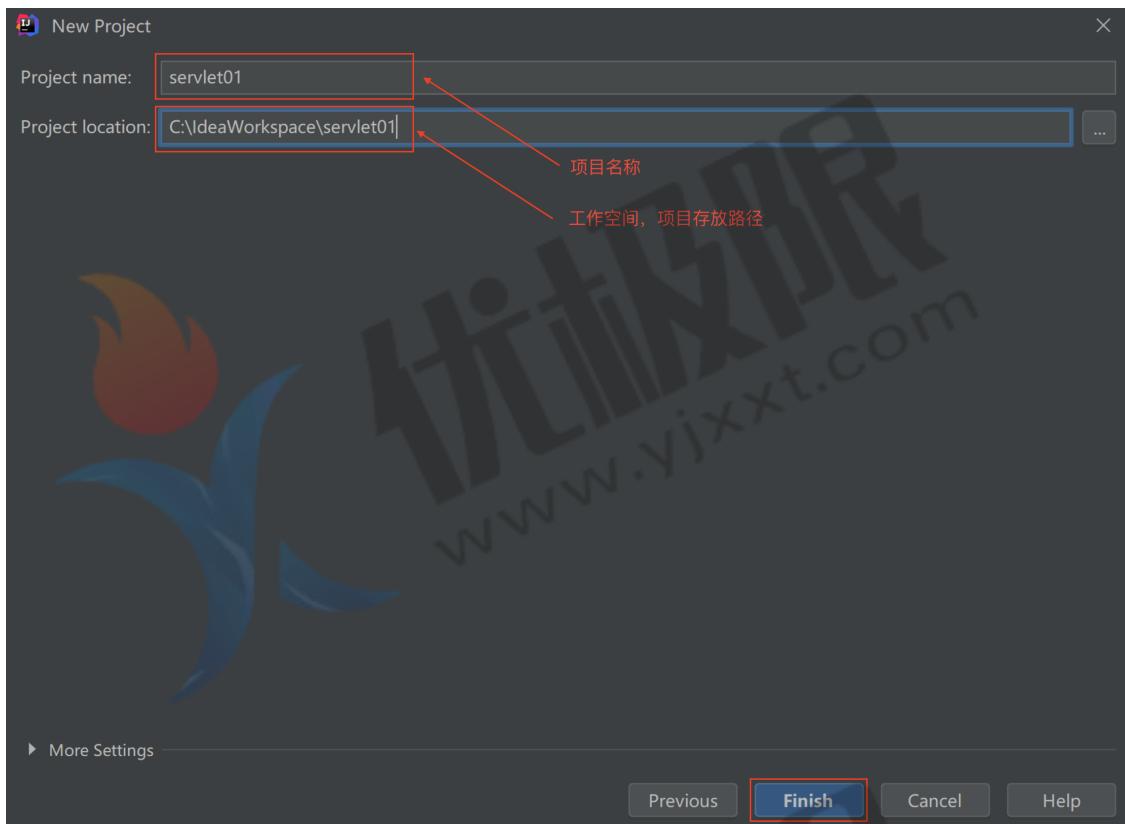
1. 选择 "File" —> "New" —> "Project"



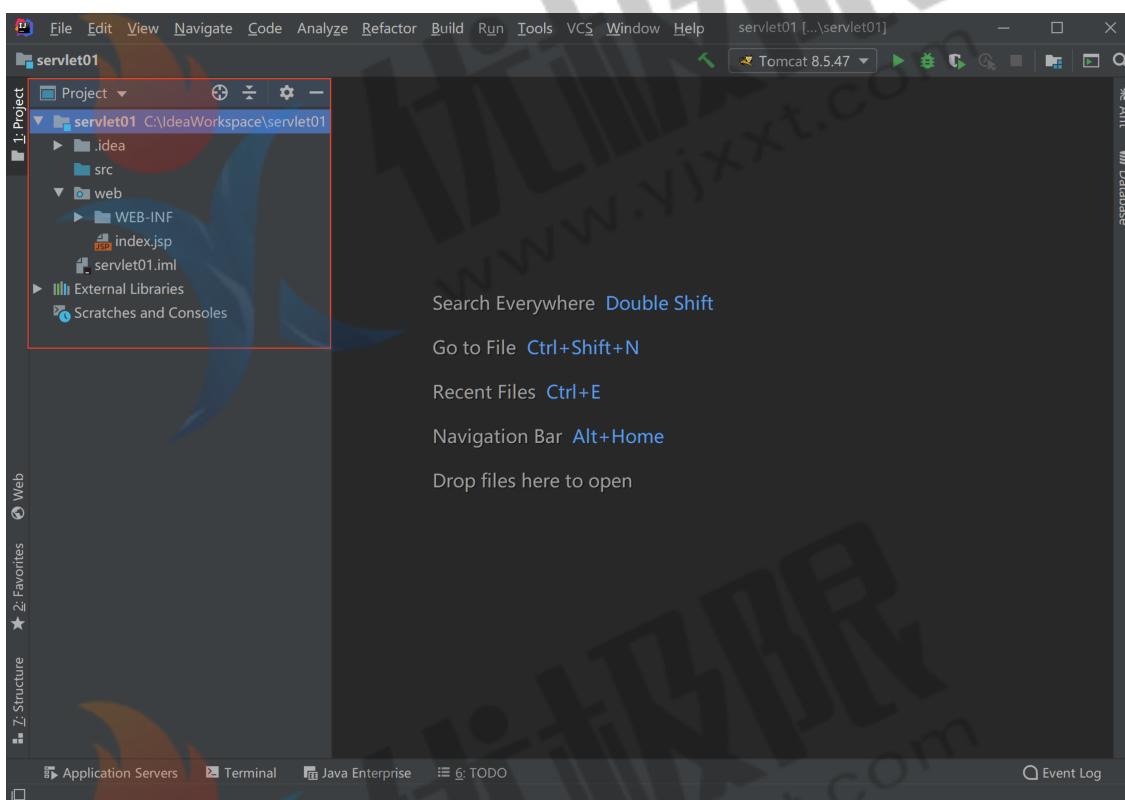
## 2. 设置项目的相关信息，选择 "Next"



## 3. 设置项目名称及工作空间



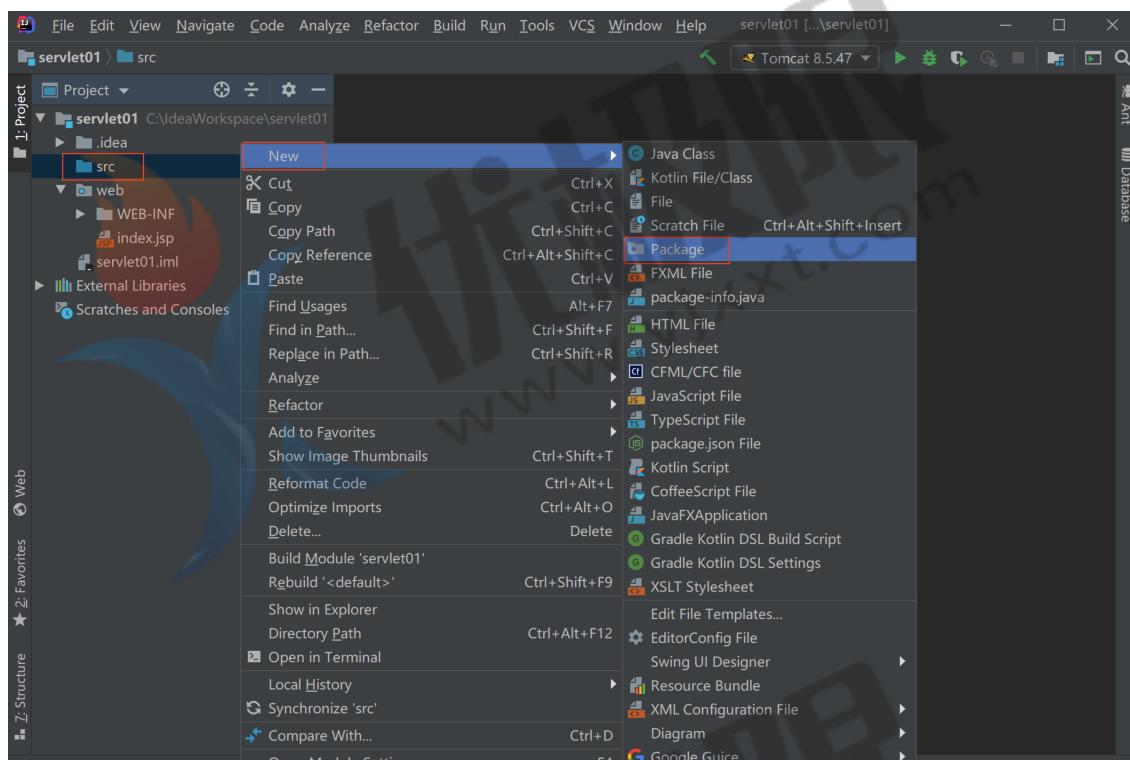
#### 4. web项目目录结构如下



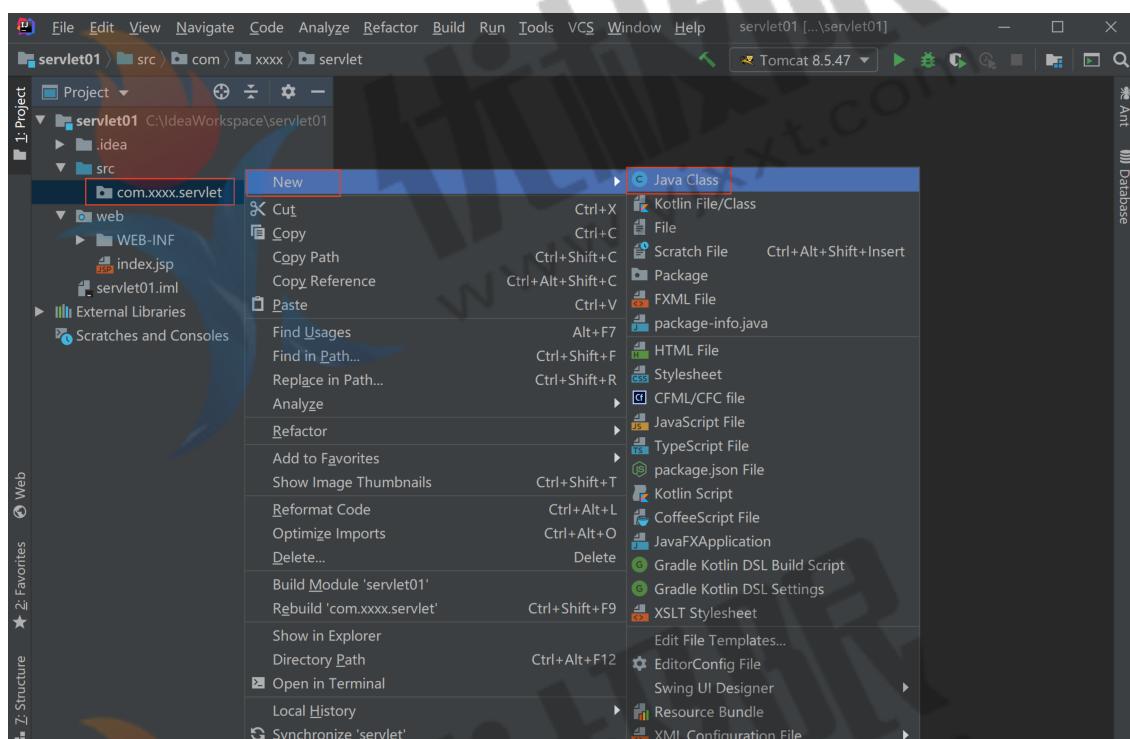
Servlet的实现

## 新建类

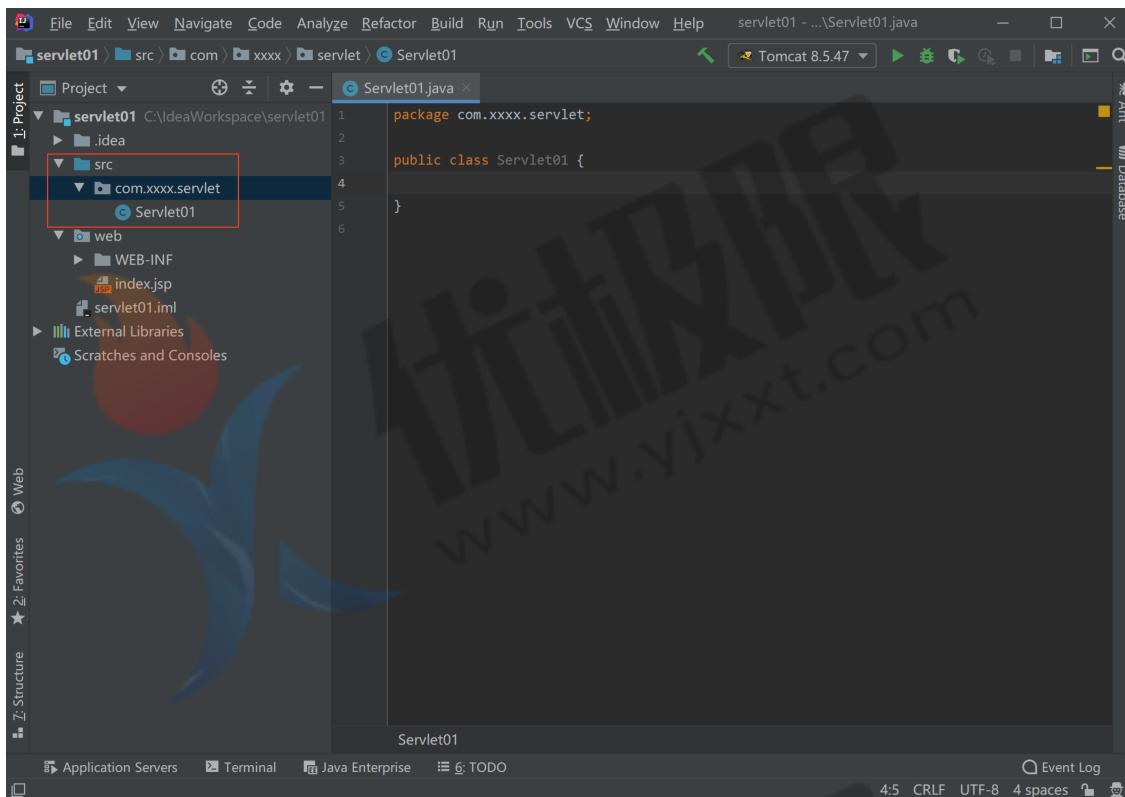
1. 点击 "src" —> "new" —> "package", 创建一个文件包



2. 在包下面创建 Java 类文件, 点击包名 —> "New" —> "Java Class"



3. 创建一个普通的 Java 类



4. 如下

```
package com.xxxx.servlet;

public class Servlet01 {
```

## 实现 Servlet 规范

实现 Servlet 规范，即继承 HttpServlet 类，并重写响应的包，该类中已经完成了通信的规则，我们只需要进行业务的实现即可。

```
package com.xxxx.servlet;

import javax.servlet.http.HttpServlet;

public class Servlet01 extends HttpServlet {
```

## 重写 service 方法

满足 Servlet 规范只是让我们的类能够满足接收请求的要求，接收到请求后需要对请求进行分析，以及进行业务逻辑处理，计算出结果，则需要添加代码，在规范中有一个叫做 service 的方法，专门用来做请求处理的操作，业务代码则可以写在该方法中。

```
package com.xxxx.servlet;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import java.io.IOException;

public class Servlet01 extends HttpServlet {

    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        System.out.println("Hello Servlet!");
        resp.getWriter().write("Hello world");
    }
}
```

## 设置注解

在完成好了一切代码的编写后，还需要向服务器说明，特定请求对应特定资源。

开发servlet项目，使用@WebServlet将一个继承于javax.servlet.http.HttpServlet 的类定义为Servlet组件。在Servlet3.0中，可以使用@WebServlet注解将一个继承于javax.servlet.http.HttpServlet的类标注为可以处理用户请求的 Servlet。

```
package com.xxxx.servlet;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/ser01")
public class Servlet01 extends HttpServlet {

    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        System.out.println("Hello Servlet!");
        resp.getWriter().write("Hello world");
    }
}
```

## 用注解配置 Servlet

```
@webServlet(name="Servlet01",value="/ser01")
```

```
@webServlet(name="Servlet01",urlPatterns = "/ser01")
```

也可以配置多个访问路径

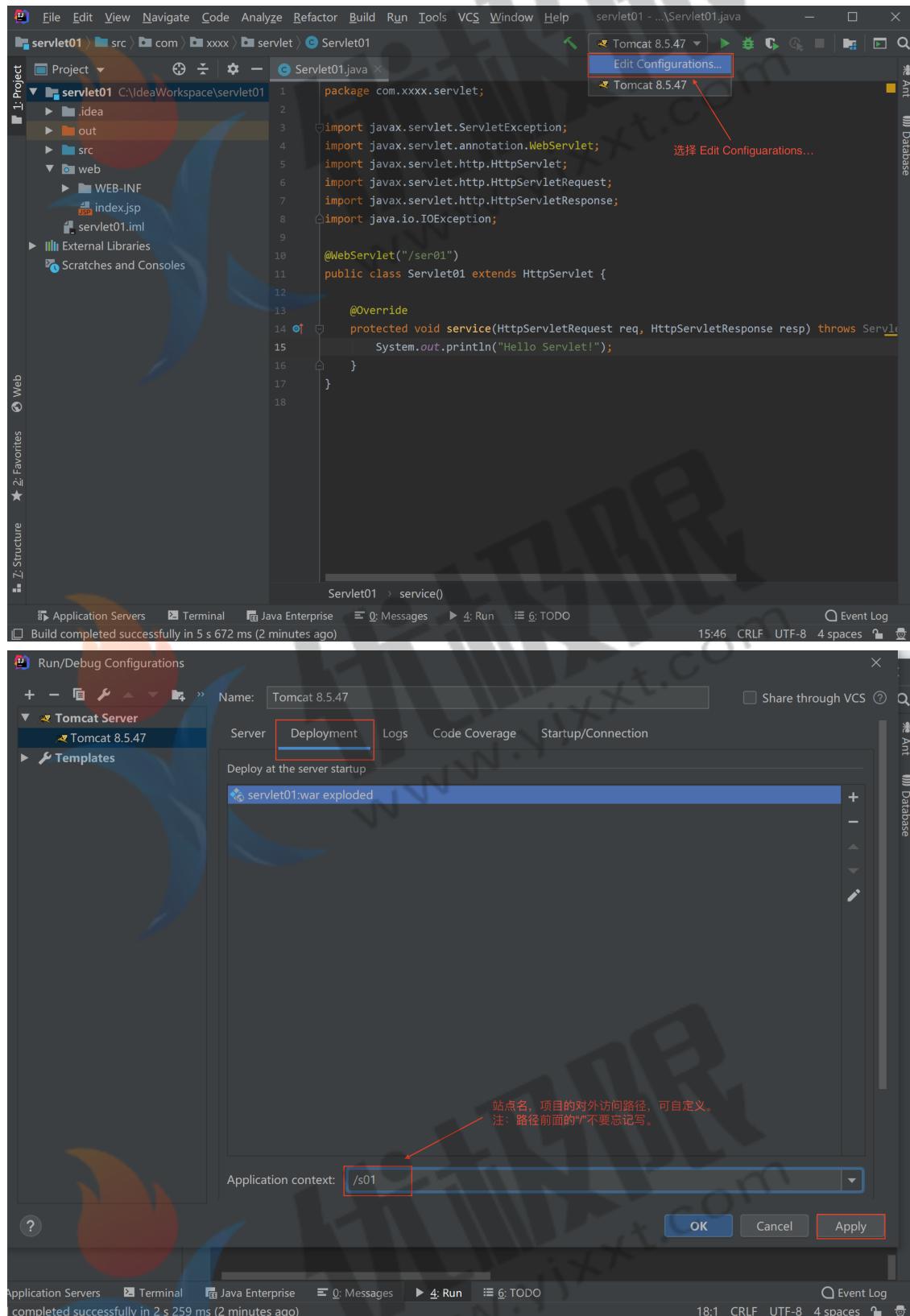
```
@webServlet(name="Servlet01",value={"/ser01','/ser001'})
```

```
@webServlet(name="Servlet01",urlPatterns={"/ser01','/ser001"})
```

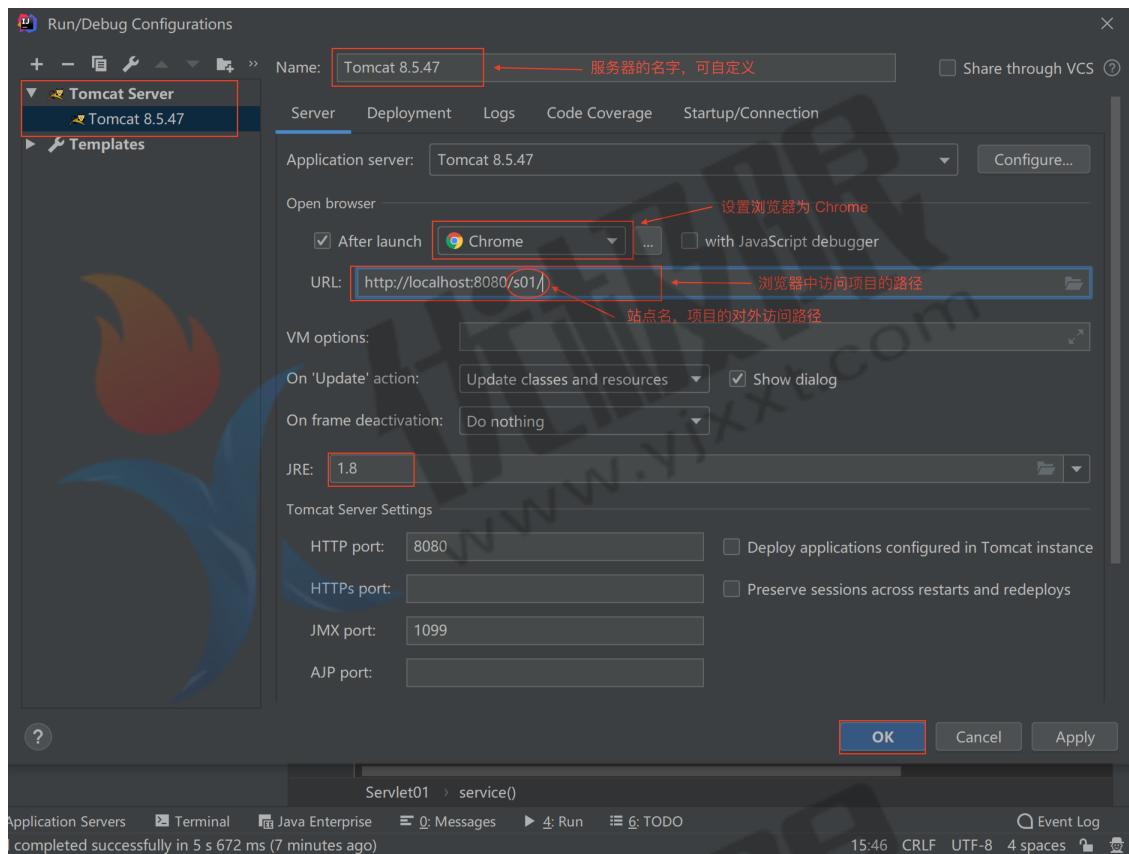
## 发布项目并启动服务

到此，需要编写和配置的地方已经完成，项目已经完整了，但是如果需要外界能够访问，还需要将项目发布到服务器上并运行服务器。

### 1. 设置项目的站点名（项目对外访问路径）



### 2. 设置项目的Tomcat配置



### 3. 启动服务器

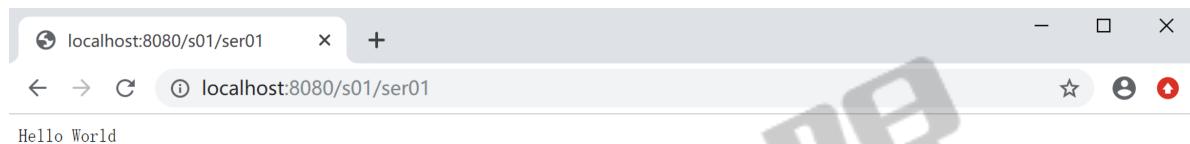


## 访问并查看结果

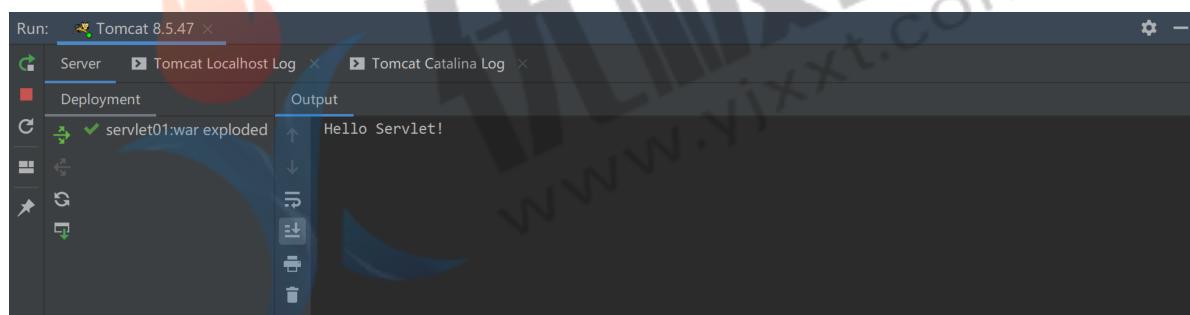
在项目正确发布到服务器上之后，用户即可通过浏览器访问该项目中的资源。注意 url 的格式正确，tomcat 的端口为 8080。

浏览器访问地址：<http://localhost:8080/s01/ser01>

### 页面效果



### 后台结果



到这里我们的第一个 Servlet 就实现了！

## Servlet的工作流程

1. 通过请求头获知浏览器访问的是哪个主机
2. 再通过请求行获取访问的是哪个一个web应用
3. 再通过请求行中的请求路径获知访问的是哪个资源
4. 通过获取的资源路径在配置中匹配到真实的路径,
5. 服务器会创建servlet对象, (如果是第一次访问时, 创建servlet实例, 并调用init方法进行初始化操作)
6. 调用service (request, response) 方法来处理请求和响应的操作
7. 调用service完毕后返回服务器 由服务器讲response缓冲区的数据取出, 以http响应的格式发送给浏览器

## Servlet的生命周期

Servlet没有 main()方法, 不能独立运行, 它的运行完全由 Servlet 引擎来控制和调度。 所谓生命周期, 指的是 servlet 容器何时创建 servlet 实例、何时调用其方法进行请求的处理、何时并销毁其实例的整个过程。

- 实例和初始化时机

当请求到达容器时, 容器查找该 servlet 对象是否存在, 如果不存在, 则会创建实例并进行初始化。

- 就绪/调用/服务阶段

有请求到达容器, 容器调用 servlet 对象的 service()方法, 处理请求的方法在整个生命周期中可以被多次调用; HttpServlet 的 service()方法, 会依据请求方式来调用 doGet()或者 doPost()方法。但是, 这两个 do 方法默认情况下, 会抛出异常, 需要子类去 override。

- 销毁时机

当容器关闭时 (应用程序停止时), 会将程序中的 Servlet 实例进行销毁。

上述的生命周期可以通过 Servlet 中的生命周期方法来观察。在 Servlet 中有三个生命周期方法, 不由用户手动调用, 而是在特定的时机由容器自动调用, 观察这三个生命周期方法 即可观察到 Servlet 的生命周期。

**init** 方法, 在 Servlet 实例创建之后执行 (证明该 Servlet 有实例创建了)

```
public void init(ServletConfig config) throws ServletException {  
    System.out.println("实例创建了...");  
}
```

**service** 方法, 每次有请求到达某个 Servlet 方法时执行, 用来处理请求 (证明该Servlet 进行服务了)

```
protected void service(HttpServletRequest req, HttpServletResponse resp)  
    throws ServletException, IOException {  
    System.out.println("服务调用了...");  
}
```

**destroy** 方法, Servlet 实例销毁时执行 (证明该 Servlet 的实例被销毁了)

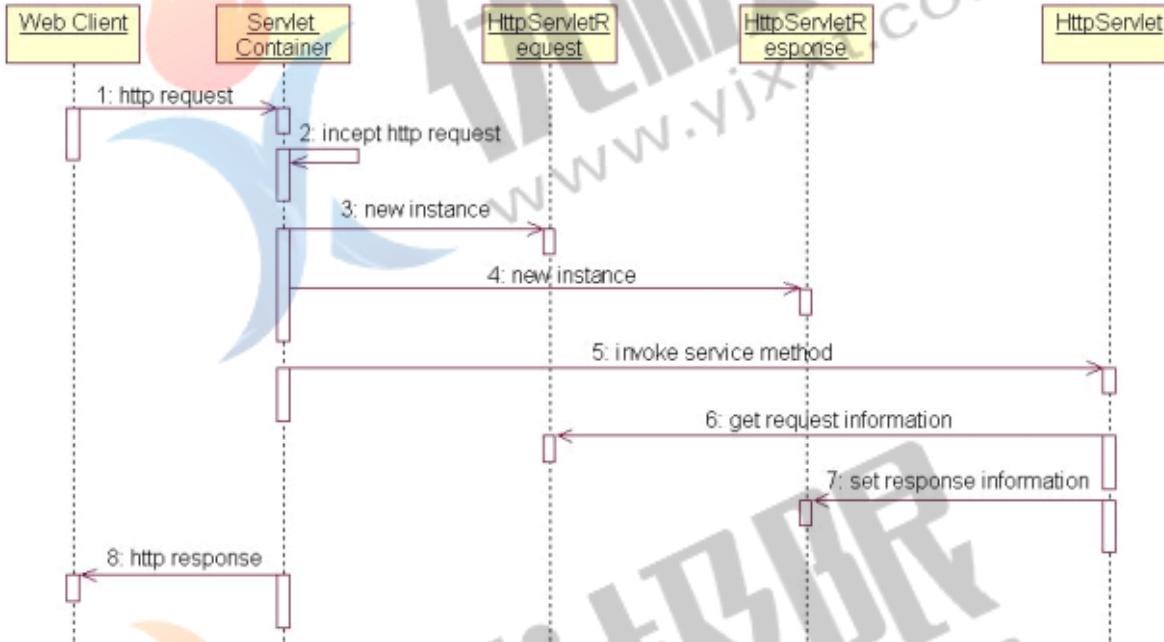
```

public void destroy() {
    System.out.println("实例销毁了....");
}

```

Servlet 的生命周期，简单的概括这就分为四步：servlet 类加载-->实例化-->服务-->销毁。

下面我们描述一下 Tomcat 与 Servlet 是如何工作的,看看下面的时序图：



1. Web Client 向 Servlet 容器 (Tomcat) 发出 Http 请求
2. Servlet 容器接收 Web Client 的请求
3. Servlet 容器创建一个 HttpServletRequest 对象，将 Web Client 请求的信息封装到这个对象 中
4. Servlet 容器创建一个 HttpServletResponse 对象
5. Servlet 容器调用 HttpServlet 对象的 service 方法，把 Request 与 Response 作为参数，传给 HttpServlet
6. HttpServlet 调用 HttpServletRequest 对象的有关方法，获取 Http 请求信息
7. HttpServlet 调用 HttpServletResponse 对象的有关方法，生成响应数据
8. Servlet 容器把 HttpServletResponse 的响应结果传给 Web Client

## HttpServletRequest对象

HttpServletRequest 对象：主要作用是用来接收客户端发送过来的请求信息，例如：请求的参数，发送的头信息等都属于客户端发来的信息， service()方法中形参接收的是 HttpServletRequest 接口的实例化对象，表示该对象主要应用在 HTTP 协议上，该对象是由 Tomcat 封装好传递过来。

HttpServletRequest 是 ServletRequest 的子接口，ServletRequest 只有一个子接口，就是 HttpServletRequest。既然只有一个子接口为什么不将两个接口合并为一个？

从长远上讲：现在主要用的协议是 HTTP 协议，但以后可能出现更多新的协议。若以后想要支持这种新协议，只需要直接继承 ServletRequest 接口就行了。

在 HttpServletRequest 接口中，定义的方法很多，但都是围绕接收客户端参数的。但是怎么拿到该对象呢？不需要，直接在 Service 方法中由容器传入过来，而我们需要做的就是取出对象中的数据，进行分析、处理。

## 接收请求

### 常用方法

#### 1. 方法

getRequestURL()	获取客户端发出请求时的完整 URL
getRequestURI()	获取请求行中的资源名称部分（项目名称开始）
getQueryString()	获取请求行中的参数部分
getMethod()	获取客户端请求方式
getProtocol()	获取 HTTP 版本号
getContextPath()	获取 webapp 名字

#### 2. 示例

```
// 获取客户端请求的完整URL（从http开始，到?前面结束）
String url = request.getRequestURL().toString();
System.out.println("获取客户端请求的完整URL: " + url);
// 获取客户端请求的部分URL（从站点名开始，到?前面结束）
String uri = request.getRequestURI();
System.out.println("获取客户端请求的部分URL: " + uri);
// 获取请求行中的参数部分
String queryString = request.getQueryString();
System.out.println("获取请求行中的参数部分: " + queryString);
// 获取客户端的请求方式
String method = request.getMethod();
System.out.println("获取客户端的请求方式: " + method);
// 获取HTTP版本号
String protocol = request.getProtocol();
System.out.println("获取HTTP版本号: " + protocol);
// 获取webapp名字（站点名）
String webapp = request.getContextPath();
System.out.println("获取webapp名字: " + webapp);
```

## 获取请求参数

### 1. 方法

getParameter(name)	获取指定名称的参数
getParameterValues(String name)	获取指定名称参数的所有值

#### 2. 示例

```
// 获取指定名称的参数，返回字符串
String uname = request.getParameter("uname");
System.out.println("uname的参数值: " + uname);
// 获取指定名称参数的所有参数值，返回数组
String[] hobbys = request.getParameterValues("hobby");
System.out.println("获取指定名称参数的所有参数值: " + Arrays.toString(hobbys));
```

## 请求乱码问题

由于现在的 request 属于接收客户端的参数，所以必然有其默认的语言编码，主要是由于在解析过程中默认使用的编码方式为 ISO-8859-1(此编码不支持中文)，所以解析时一定会出现乱码。要想解决这种乱码问题，需要设置 request 中的编码方式，告诉服务器以何种方式来解析数据。或者在接收到乱码数据以后，再通过相应的编码格式还原。

### 方式一：

```
request.setCharacterEncoding("UTF-8");
```

这种方式只针对 POST 有效 (必须在接收所有的数据之前设定)

### 方式二：

```
new String(request.getParameter(name).getBytes("ISO-8859-1"), "UTF-8");
```

借助了String 对象的方法，该种方式对任何请求有效，是通用的。

Tomcat8起，以后的GET方式请求是不会出现乱码的。

## 请求转发

请求转发，是一种**服务器的行为**，当客户端请求到达后，服务器进行转发，此时会将请求对象进行保存，地址栏中的 **URL 地址不会改变**，得到响应后，服务器端再将响应发送给客户端，**从始至终只有一个请求发出**。

实现方式如下，达到多个资源协同响应的效果。

```
request.getRequestDispatcher(url).forward(request, response);
```

## request作用域

通过该对象可以在一个请求中传递数据，作用范围：在一次请求中有效，即服务器跳转有效。

```
// 设置域对象内容  
request.setAttribute(String name, String value);  
// 获取域对象内容  
request.getAttribute(String name);  
// 删除域对象内容  
request.removeAttribute(String name);
```

request 域对象中的数据在一次请求中有效，则经过请求转发，request 域中的数据依然存在，则在请求转发的过程中可以通过 request 来传输/共享数据。

## HttpServletResponse对象

Web服务器收到客户端的http请求，会针对每一次请求，分别创建一个用于**代表请求**的 request 对象和**代表响应**的 response 对象。

request 和 response 对象代表请求和响应：获取客户端数据，需要通过 request 对象；**向客户端输出数据，需要通过 response 对象**。

HttpServletResponse 的主要功能用于服务器对客户端的请求进行响应，将 Web 服务器处理后的结果返回给客户端。service()方法中形参接收的是 HttpServletResponse 接口的实例化对象，这个对象中封装了向客户端发送数据、发送响应头，发送响应状态码的方法。

## 响应数据

接收到客户端请求后，可以通过 HttpServletResponse 对象直接进行响应，响应时需要获取输出流。

有两种形式：

`getWriter()` 获取字符流(只能响应回字符串)

`getOutputStream()` 获取字节流(能响应一切数据)

响应回的数据到客户端被浏览器解析。

**注意：两者不能同时使用。**

```
// 字符输出流  
Printwriter writer = response.getWriter();  
writer.write("Hello");  
writer.write("<h2>Hello</h2>");
```

```
// 字节输出流  
ServletOutputStream out = response.getOutputStream();  
out.write("Hello".getBytes());  
out.write("<h2>Hello</h2>".getBytes());
```

设置响应类型，默认是字符串

```
// 设置响应MIME类型  
response.setHeader("content-type", "text/html"); // html
```

## 响应乱码问题

在响应中，如果我们响应的内容中含有中文，则有可能出现乱码。这是因为服务器响应的数据也会经过网络传输，服务器端有一种编码方式，在客户端也存在一种编码方式，当两端使用的编码方式不同时则出现乱码。

### getWriter()的字符乱码

对于 getWriter()获取到的字符流，响应中文必定出乱码，由于服务器端在进行编码时默认会使用 ISO-8859-1 格式的编码，该编码方式并不支持中文。

要解决该种乱码只能在服务器端**告知服务器**使用一种能够支持中文的编码格式，比如我们通常用的"UTF-8"。

```
response.setCharacterEncoding("UTF-8");
```

此时还只完成了一半的工作，要保证数据正确显示，还需要**指定客户端的解码方式**。

```
response.setHeader("content-type", "text/html;charset=UTF-8");
```

两端指定编码后，乱码就解决了。一句话：**保证发送端和接收端的编码一致**

```
// 设置服务端的编码  
response.setCharacterEncoding("UTF-8");  
// 设置客户端的响应类型及编码  
response.setHeader("content-type", "text/html;charset=UTF-8");  
// 得到字符输出流  
PrintWriter writer = response.getWriter();  
writer.write("<h2>你好</h2>");
```

以上两端编码的指定也可以使用一句替代，同时指定服务器和客户端

```
response.setContentType("text/html;charset=UTF-8");
```

### getOutputStream()字节乱码

对于 getOutputStream() 方式获取到的字节流，响应中文时，由于本身就是传输的字节，所以此时可能出现乱码，也可能正确显示。当服务器端给的字节恰好和客户端使用的编码方式一致时则文本正确显示，否则出现乱码。无论如何我们都应该准确掌握服务器和客户端使用的是那种编码格式，以确保数据正确显示。

**指定客户端和服务器使用的编码方式一致。**

```
response.setHeader("content-type", "text/html;charset=UTF-8");
```

```
// 设置客户端的编码及响应类型  
ServletOutputStream out = response.getOutputStream();  
response.setHeader("content-type", "text/html;charset=UTF-8");  
out.write("<h2>你好</h2>".getBytes("UTF-8"));
```

同样也可以使用一句替代

```
// 设置客户端与服务端的编码  
response.setContentType("text/html;charset=UTF-8");
```

**总结：要想解决响应的乱码，只需要保证使用支持中文的编码格式。并且保证服务器端 和客户端使用相同的编码方式即可。**

## 重定向

重定向是一种服务器指导，客户端的行为。客户端发出第一个请求，被服务器接收处理后，服务器会进行响应，在响应的同时，服务器会给客户端一个新的地址（下次请求的地址 response.sendRedirect(url);），当客户端接收到响应后，会立刻、马上、自动根据服务器给的新地址发起第二个请求，服务器接收请求并作出响应，重定向完成。

从描述中可以看出重定向当中有两个请求存在，并且属于客户端行为。

```
// 重定向跳转到index.jsp  
response.sendRedirect("index.jsp");
```

通过观察浏览器我们发现第一次请求获得的响应码为 302，并且含有一个 location 头信息。并且地址栏最终看到的地址是和第一次请求地址不同的，地址栏已经发生了变化。

▼ Response Headers view parsed  
HTTP/1.1 302  
Location: index.jsp  
Content-Length: 0

## 请求转发与重定向的区别

请求转发和重定向比较：

请求转发（req.getRequestDispatcher().forward()）	重定向（resp.sendRedirect()）
一次请求，数据在 request 域中共享	两次请求，request 域中数据不共享
服务器端行为	客户端行为
地址栏不发生变化	地址栏发生变化
绝对地址定位到站点后	绝对地址可写到 http://

两者都可进行跳转，根据实际需求选取即可。

## Cookie对象

Cookie是浏览器提供的一种技术，通过服务器的程序能将一些只须保存在客户端，或者在客户端进行处理的数据，放在本地的计算机上，不需要通过网络传输，因而提高网页处理的效率，并且能够减少服务器的负载，但是由于 Cookie 是服务器端保存在客户端的信息，所以其安全性也是很差的。例如常见的记住密码则可以通过 Cookie 来实现。

有一个专门操作Cookie的类 **javax.servlet.http.Cookie**。随着服务器端的响应发送给客户端，保存在浏览器。当下次再访问服务器时把Cookie再带回服务器。

Cookie 的格式：键值对用“=”链接，多个键值对间通过“；”隔开。

## Cookie的创建和发送

通过 new Cookie("key","value");来创建一个 Cookie 对象，要想将 Cookie 随响应发送到客户端，需要先添加到 response 对象中，response.addCookie(cookie);此时该 cookie 对象则随着响应发送至了客户端。在浏览器上可以看见。

```
// 创建Cookie对象  
Cookie cookie = new Cookie("uname", "zhangsan");  
// 发送Cookie对象  
response.addCookie(cookie);
```

▼ Response Headers [view parsed](#)  
HTTP/1.1 200  
Set-Cookie: uname=zhangsan  
Content-Length: 0

## Cookie的获取

在服务器端只提供了一个 getCookies() 的方法用来获取客户端回传的所有 cookie 组成的一个数组，如果需要获取单个 cookie 则需要通过遍历，getName() 获取 Cookie 的名称，getValue() 获取 Cookie 的值。

```
// 获取cookie数组
Cookie[] cookies = request.getCookies();
// 判断数组是否为空
if (cookies != null && cookies.length > 0) {
    // 遍历Cookie数组
    for (Cookie cookie : cookies){
        System.out.println(cookie.getName());
        System.out.println(cookie.getValue());
    }
}
```

## Cookie设置到期时间

除了 Cookie 的名称和内容外，我们还需要关心一个信息，到期时间，到期时间用来指定该 cookie 何时失效。默认为当前浏览器关闭即失效。我们可以手动设定 cookie 的有效时间（通过到期时间计算），通过 setMaxAge(int time); 方法设定 cookie 的最大有效时间，以秒为单位。

### 到期时间的取值

- 负整数

若为负数，表示不存储该 cookie。

cookie 的 maxAge 属性的默认值就是 -1，表示只在浏览器内存中存活，一旦关闭浏览器窗口，那么 cookie 就会消失。

- 正整数

若大于 0 的整数，表示存储的秒数。

表示 cookie 对象可存活指定的秒数。当生命大于 0 时，浏览器会把 Cookie 保存到硬盘上，就算关闭浏览器，就算重启客户端电脑，cookie 也会存活相应的时间。

- 零

若为 0，表示删除该 cookie。

cookie 生命等于 0 是一个特殊的值，它表示 cookie 被作废！也就是说，如果原来浏览器已经保存了这个 Cookie，那么可以通过 Cookie 的 setMaxAge(0) 来删除这个 Cookie。无论是在浏览器内存中，还是在客户端硬盘上都会删除这个 Cookie。

### 设置Cookie对象指定时间后失效

```
// 创建Cookie对象  
Cookie cookie = new Cookie("uname", "zhangsan");  
// 设置Cookie 3天后失效  
cookie.setMaxAge(3 * 24 * 60 * 60);  
// 发送Cookie对象  
response.addCookie(cookie);
```

## Cookie的注意点

### 1. Cookie保存在当前浏览器中。

在一般的站点中常常有记住用户名这样一个操作，该操作只是将信息保存在本机上，换电脑以后这些信息就无效了。而且 cookie 还不能跨浏览器。

### 2. Cookie存中文问题

Cookie 中不能出现中文，如果有中文则通过 URLEncoder.encode()来进行编码，获取时通过 URLDecoder.decode()来进行解码。

```
String name = "姓名";  
String value = "张三";  
// 通过 URLEncoder.encode()来进行编码  
name = URLEncoder.encode(name);  
value = URLEncoder.encode(value);  
// 创建Cookie对象  
Cookie cookie = new Cookie(name,value);  
// 发送Cookie对象  
response.addCookie(cookie);
```

```
// 获取时通过 URLDecoder.decode()来进行解码  
URLDecoder.decode(cookie.getName());  
URLDecoder.decode(cookie.getValue());
```

### 3. 同名Cookie问题

如果服务器端发送重复的Cookie那么会覆盖原有的Cookie。

### 4. 浏览器存放Cookie的数量

不同的浏览器对Cookie也有限定，Cookie的存储有上限的。Cookie是存储在客户端（浏览器）的，而且一般是由服务器端创建和设定。后期结合Session来实现回话跟踪。

## Cookie的路径

Cookie的setPath设置cookie的路径，这个路径直接决定服务器的请求是否会从浏览器中加载某些cookie。

**情景一：**当前服务器下任何项目的任意资源都可获取Cookie对象

```
/* 当前项目路径为: s01 */  
Cookie cookie = new Cookie("xxx", "XXX");  
// 设置路径为"/"，表示在当前服务器下任何项目都可访问到Cookie对象  
cookie.setPath("/");  
response.addCookie(cookie);
```

**情景二：**当前项目下的资源可获取Cookie对象（默认不设置Cookie的path）

```
/* 当前项目路径为: s01 */
Cookie cookie = new Cookie("xxx", "xxx");
// 设置路径为"/s01", 表示在当前项目下任何项目都可访问到Cookie对象
cookie.setPath("/s01"); // 默认情况, 可不设置path的值
response.addCookie(cookie);
```

**情景三：**指定项目下的资源可获取Cookie对象

```
/* 当前项目路径为: s01 */
Cookie cookie = new Cookie("xxx", "xxx");
// 设置路径为"/s02", 表示在s02项目下才可访问到Cookie对象
cookie.setPath("/s02"); // 只能在s02项目下获取Cookie, 就算cookie是s01产生的, s01也不能
获取它
response.addCookie(cookie);
```

**情景四：**指定目录下的资源可获取Cookie对象

```
/* 当前项目路径为: s01 */
Cookie cookie = new Cookie("xxx", "xxx");
// 设置路径为"/s01/cook", 表示在s02/cook目录下才可访问到Cookie对象
cookie.setPath("/s01/cook");
response.addCookie(cookie);
```

如果我们设置path，如果当前访问的路径包含了cookie的路径（当前访问路径在cookie路径基础上要比cookie的范围小）cookie就会加载到request对象之中。

cookie的路径指的是可以访问该cookie的顶层目录，该路径的子路径也可以访问该cookie。

**总结：**当访问的路径包含了cookie的路径时，则该请求将带上该cookie；如果访问路径不包含cookie路径，则该请求不会携带该cookie。

## HttpSession对象

**HttpSession**对象是 javax.servlet.http.HttpSession 的实例，该接口并不像 HttpServletRequest 或 HttpServletResponse 还存在一个父接口，该接口只是一个纯粹的接口。这因为 session 本身就属于 HTTP 协议的范畴。

对于服务器而言，每一个连接到它的客户端都是一个 session，servlet 容器使用此接口创建 HTTP 客户端和 HTTP 服务器之间的会话。会话将保留指定的时间段，跨多个连接或来自用户的页面请求。一个会话通常对应于一个用户，该用户可能多次访问一个站点。可以通过此接口查看和操作有关某个会话的信息，比如会话标识符、创建时间和最后一次访问时间。在整个 session 中，最重要的就是属性的操作。



session 无论客户端还是服务器端都可以感知到，若重新打开一个新的浏览器，则无法取得之前设置的 session，因为每一个 session 只保存在当前的浏览器当中，并在相关的页面取得。

Session 的作用就是为了标识一次会话，或者说确认一个用户；并且在一次会话（一个用户的多次请求）期间共享数据。我们可以通过 `request.getSession()` 方法，来获取当前会话的 session 对象。

```
// 如果session对象存在，则获取；如果session对象不存在，则创建  
HttpSession session = request.getSession();
```

## 标识符 JSESSIONID

Session 既然是为了标识一次会话，那么此次会话就应该有一个唯一的标志，这个标志就是 `sessionId`。

每当一次请求到达服务器，如果开启了会话（访问了 session），服务器第一步会查看是否从客户端回传一个名为 JSESSIONID 的 cookie，如果没有则认为这是一次新的会话，会创建一个新的 session 对象，并用唯一的 `sessionId` 为此次会话做一个标志。如果有 JSESSIONID 这个 cookie 回传，服务器则会根据 JSESSIONID 这个值去查看是否含有 id 为 JSESSION 值的 session 对象，如果没有则认为是一个新的会话，重新创建一个新的 session 对象，并标志此次会话；如果找到了相应的 session 对象，则认为是之前标志过的一次会话，返回该 session 对象，数据达到共享。

这里提到一个叫做 JSESSIONID 的 cookie，这是一个比较特殊的 cookie，当用户请求服务器时，如果访问了 session，则服务器会创建一个名为 JSESSIONID，值为获取到的 session（无论是获取到的还是新创建的）的 `sessionId` 的 cookie 对象，并添加到 response 对象中，响应给客户端，有效时间为关闭浏览器。

所以 Session 的底层依赖 Cookie 来实现。

## session域对象

Session 用来表示一次会话，在一次会话中数据是可以共享的，这时 session 作为域对象存在，可以通过 `setAttribute(name,value)` 方法向域对象中添加数据，通过 `getAttribute(name)` 从域对象中获取数据，通过 `removeAttribute(name)` 从域对象中移除数据。

```
// 获取session对象
HttpSession session = request.getSession();
// 设置session域对象
session.setAttribute("uname", "admin");
// 获取指定名称的session域对象
String uname = (String) request.getAttribute("uname");
// 移除指定名称的session域对象
session.removeAttribute("uname");
```

数据存储在 session 域对象中，当 session 对象不存在了，或者是两个不同的 session 对象时，数据也就不能共享了。这就不得不谈到 session 的生命周期。

## session对象的销毁

### 默认时间到期

当客户端第一次请求 servlet 并且操作 session 时，session 对象生成，Tomcat 中 session 默认的存活时间为 30min，即你不操作界面的时间，一旦有操作，session 会重新计时。

那么 session 的默认时间可以改么？答案是肯定的。

可以在 Tomcat 中的 conf 目录下的 web.xml 文件中进行修改。

```
<!-- session 默认的最大不活动时间。单位：分钟。 -->
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

### 自己设定到期时间

当然除了以上的修改方式外，我们也可以在程序中自己设定 session 的生命周期，通过 session.setMaxInactiveInterval(int) 来设定 session 的最大不活动时间，单位为秒。

```
// 获取session对象
HttpSession session = request.getSession();
// 设置session的最大不活动时间
session.setMaxInactiveInterval(15); // 15秒
```

当然我们也可以通过 getMaxInactiveInterval() 方法来查看当前 Session 对象的最大不活动时间。

```
// 获取session的最大不活动时间
int time = session.getMaxInactiveInterval();
```

### 立刻失效

或者我们也可以通过 session.invalidate() 方法让 session 立刻失效

```
// 销毁session对象
session.invalidate();
```

## 关闭浏览器

从前面的 JSESSIONID 可知道，session 的底层依赖 cookie 实现，并且该 cookie 的有效时间为关闭浏览器，从而 session 在浏览器关闭时也相当于失效了（因为没有 JSESSION 再与之对应）。

## 关闭服务器

当关闭服务器时，session 销毁。

Session 失效则意味着此次会话结束，数据共享结束。

## ServletContext对象

每一个 web 应用都有且仅有一个ServletContext 对象，又称 Application 对象，从名称中可知，该对象是与应用程序相关的。在 WEB 容器启动的时候，会为每一个 WEB 应用程序创建一个对应的 ServletContext 对象。

该对象有两大作用，第一、作为域对象用来共享数据，此时数据在整个应用程序中共享；第二、该对象中保存了当前应用程序相关信息。例如可以通过 getServerInfo() 方法获取当前服务器信息，getRealPath(String path) 获取资源的真实路径等。

## ServletContext对象的获取

获取 ServletContext 对象的途径有很多。比如：

1. 通过 request 对象获取

```
ServletContext servletContext = request.getServletContext();
```

2. 通过 session 对象获取

```
ServletContext servletContext = request.getSession().getServletContext();
```

3. 通过 servletConfig 对象获取，在 Servlet 标准中提供了 ServletConfig 方法

```
ServletConfig servletConfig = getServletConfig();
ServletContext servletContext = servletConfig.getServletContext();
```

4. 直接获取，Servlet 类中提供了直接获取 ServletContext 对象的方法

```
ServletContext servletContext = getServletContext();
```

## 常用方法

```
// 获取项目存放的真实路径
String realPath = request.getServletContext().getRealPath("/");
// 获取当前服务器的版本信息
String serverInfo = request.getServletContext().getServerInfo();
```

## ServletContext域对象

ServletContext 也可当做域对象来使用，通过向 ServletContext 中存取数据，可以使得整个应用程序共享某些数据。当然不建议存放过多数据，因为 ServletContext 中的数据一旦存储进去没有手动移除将会一直保存。

```
// 获取ServletContext对象
ServletContext servletContext = request.getServletContext();
// 设置域对象
servletContext.setAttribute("name", "zhangsan");
// 获取域对象
String name = (String) servletContext.getAttribute("name");
// 移除域对象
servletContext.removeAttribute("name");
```

## Servlet的三大域对象

### 1. request域对象

在一次请求中有效。请求转发有效，重定向失效。

### 2. session域对象

在一次会话中有效。请求转发和重定向都有效，session销毁后失效。

### 3. servletContext域对象

在整个应用程序中有效。服务器关闭后失效。

## 文件上传和下载

在上网的时候我们常常遇到文件上传的情况，例如上传头像、上传资料等；当然除了上传，遇见下载的情况也很多，接下来看看我们 servlet 中怎么实现文件的上传和下载。

## 文件上传

文件上传涉及到前台页面的编写和后台服务器端代码的编写，前台发送文件，后台接收并保存文件，这才是一个完整的文件上传。

### 前台页面

在做文件上传的时候，会有一个上传文件的界面，首先我们需要一个表单，并且表单的请求方式为 **POST**；其次我们的 form 表单的 enctype 必须设为"multipart/form-data"，即 **enctype="multipart/form-data"**，意思是设置表单的类型为文件上传表单。默认情况下这个表单类型是 "application/x-www-form-urlencoded"，不能用于文件上传。只有使用了 multipart/form-data 才能完整地传递文件数据。

```

<!--
文件上传表单
1. 表单提交类型 method="post"
2. 表单类型 enctype="multipart/form-data"
3. 表单元素类型 文件域设置name属性值
-->
<form method="post" action="uploadServlet" enctype="multipart/form-data">
    姓名: <input type="text" name="uname" > <br>
    文件: <input type="file" name="myfile" > <br>
    <button type="submit">提交</button>
</form>

```

## 后台实现

使用注解 `@MultipartConfig` 将一个 Servlet 标识为支持文件上传。Servlet 将 multipart/form-data 的 POST 请求封装成 Part，通过 Part 对上传的文件进行操作。

```

package com.xxxx.servlet;

import javax.servlet.ServletException;
import javax.servlet.annotation.MultipartConfig;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.Part;
import java.io.IOException;

@WebServlet("/uploadServlet")
@MultipartConfig // 如果是文件上传表单，一定要加这个注解
public class UploadServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // 设置请求的编码格式
        request.setCharacterEncoding("UTF-8");
        // 获取普通表单项（文本框）
        String uname = request.getParameter("uname"); // "uname"代表的是文本框的
        name属性值
        // 通过 getPart(name) 方法获取Part对象（name代表的是页面中file文件域的name属性
        值）
        Part part = request.getPart("myfile");
        // 通过Part对象，获取上传的文件名
        String fileName = part.getSubmittedFileName();
        // 获取上传文件需要存放的路径（得到项目存放的真实路径）
        String realPath = request.getServletContext().getRealPath("/");
        // 将文件上传到指定位置
        part.write(realPath + fileName);
    }
}

```

# 文件下载

文件下载，即将服务器上的资源下载（拷贝）到本地，我们可以通过两种方式下载。第一种是通过超链接本身的特性来下载；第二种是通过代码下载。

## 超链接下载

当我们在 HTML 或 JSP 页面中使用a标签时，原意是希望能够进行跳转，但当超链接遇到浏览器不识别的资源时会自动下载；当遇见浏览器能够直接显示的资源，浏览器就会默认显示出来，比如 txt、png、jpg 等。当然我们也可以通过 **download 属性** 规定浏览器进行下载。但有些浏览器并不支持。

### 默认下载

```
<!-- 当超链接遇到浏览器不识别的资源时，会自动下载 -->
<a href="test.zip">超链接下载</a>
```

### 指定 download 属性下载

```
<!-- 当超链接遇到浏览器识别的资源时，默认不会下载。通过download属性可进行下载 -->
<a href="test.txt" download>超链接下载</a>
```

download 属性可以不写任何信息，会自动使用默认文件名。如果设置了download属性的值，则使用设置的值做为文件名。当用户打开浏览器点击链接的时候就会直接下载文件。

## 后台实现下载

### 实现步骤

1. 需要通过 response.setContentType 方法设置 Content-type 头字段的值，为浏览器无法使用某种方式或激活某个程序来处理的 MIME 类型，例如 "application/octet-stream" 或 "application/x-msdownload" 等。
2. 需要通过 response.setHeader 方法设置 Content-Disposition 头的值为 "attachment;filename=文件名"
3. 读取下载文件，调用 response.getOutputStream 方法向客户端写入附件内容。

```
package com.xxxx.servlet;

import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class DownloadServlet extends HttpServlet {
    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // 设置请求的编码
        request.setCharacterEncoding("UTF-8");
        // 获取文件下载路径
        String path = getServletContext().getRealPath("/");
        // 将文件输出流设置为响应体
        response.setContentType("application/octet-stream");
        response.setHeader("Content-Disposition", "attachment;filename=" + new File(path + "test.txt").getName());
        // 将文件输入流写入响应体
        try (InputStream in = new FileInputStream(new File(path + "test.txt"));
             ServletOutputStream out = response.getOutputStream()) {
            byte[] buffer = new byte[1024];
            int len;
            while ((len = in.read(buffer)) != -1) {
                out.write(buffer, 0, len);
            }
        }
    }
}
```

```
// 获取要下载的文件名
String name = request.getParameter("fileName");
// 通过路径得到file对象
File file = new File(path + name);
// 判断file对象是否存在，且是否是一个标准文件
if (file.exists() && file.isFile()) {
    // 设置响应类型（浏览器无法使用某种方式或激活某个程序来处理的类型）
    response.setContentType("application/x-msdownload");
    // 设置头信息
    response.setHeader("Content-Disposition", "attachment;filename=" +
name);
    // 得到输入流
    InputStream is = new FileInputStream(file);
    // 得到输出流
    ServletOutputStream os = response.getOutputStream();
    // 定义byte数组
    byte[] car = new byte[1024];
    // 定义长度
    int len = 0;
    // 循环 输出
    while ((len = is.read(car)) != -1) {
        os.write(car, 0, len);
    }
    // 关闭流 释放资源
    os.close();
    is.close();
} else {
    System.out.println("文件不存在，下载失败！");
}
}
```