**IDEA开发工具和Lambda表达式**

# 回顾

## JUnit的使用

### JUnit编写的测试方法有什么要求？

1. 访问修饰符：public
2. 返回值：void
3. 参数列表：没有参数列表
4. 方法命名规范：以testXxx()方法名

### 如何运行测试方法

|  |  |
| --- | --- |
| **在eclipse中运行测试方法** | **操作方式** |
| **运行一个测试方法** | 在方法名上点右键运行 |
| **运行一个类中所有的测试方法** | 在类名上点右键运行 |
| **运行整个工程中的所有类中的测试方法** | 在项目名上点右键运行 |

### JUnit中常用注解

|  |  |
| --- | --- |
| **注解名** | **作用** |
| **@BeforeClass** | 只运行1次，放在静态方法上，所有的测试方法之前运行 |
| **@Before** | 每个方法运行前都会运行 |
| **@Test** | 测试方法 |
| **@After** | 每个测试方法运行后都会运行 |
| **@AfterClass** | 只运行1次，放在静态方法上，所有的测试方法之后运行 |

## Properties属性类常用方法

|  |  |
| --- | --- |
| **常用方法** | **作用** |
| **String setProperty(String key,String value)** | 向属性集中添加一个键和值 |
| **String getProperty(String key)** | 通过键取一个值 |
| **Set<String> stringPropertyName()** | 得到一个属性集中所有的键集合 |
| **load(InputStream/Reader in)** | 从输入流中读取数据加载到属性集中 |

## 类加载器

### 类加载器的分类

|  |  |
| --- | --- |
| **类加载器** | **作用** |
| **引导类** | 加载JDK中类 |
| **扩展类** | 加载lib/ext包中类 |
| **应用类** | 加载自己的类 |

### 类加载器的常用方法

|  |  |
| --- | --- |
| **类加载器常用方法** | **作用** |
| **ClassLoader getClassLoader()** | 得到类加载器对象 |
| **InputStream getResourceAsStream(String path)** | 加载类路径下的文件，得到一个输入流对象 |

# 学习目标

1. 掌握idea开发工具的安装
2. 能够使用idea创建Java工程
3. 能够使用idea创建包和类
4. 掌握idea常用开发快捷键
5. 掌握IDEA的Debug调试
6. 掌握IDEA导入第三方Jar包
7. 能够掌握Lambda表达式的标准格式
8. 能够掌握Lambda表达式的省略格式与规则
9. 能够使用Lambda表达式使用Runnable与Comparator接口
10. 能够通过Lambda表达式使用自定义的接口

# IntelliJ IDEA的安装

## 开发工具概述

IDEA是一个专门针对Java的集成开发工具(IDE)，由Java语言编写。所以需要有JRE运行环境并配置好环境变量。它可以极大地提升我们的开发效率。可以自动编译，检查错误。越来越多公司使用IDEA开发工具。

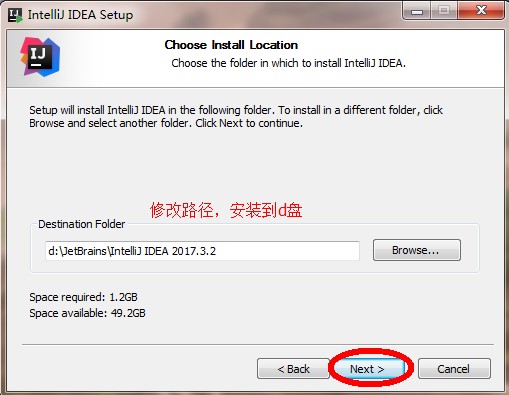
## IDEA软件安装

此软件集成了32位和64位，双击ideaIU-2017.3.2.exe 进入安装。

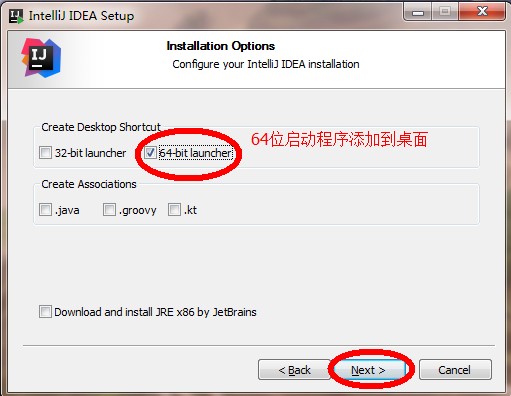
1. 欢迎界面



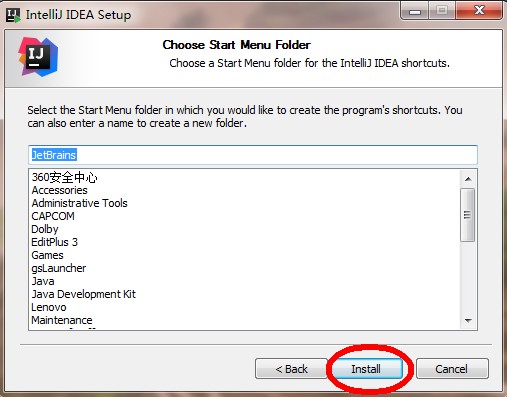
1. 选择安装路径



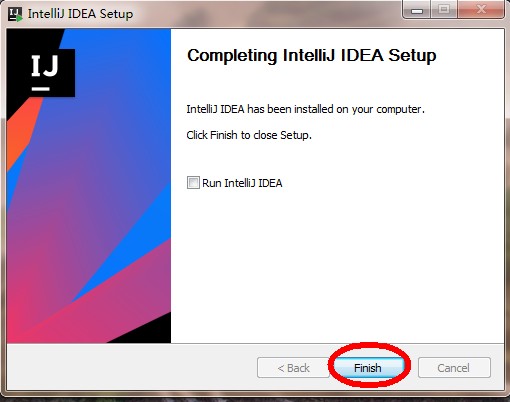
1. 配置安装选项



1. 开始菜单



1. 安装完毕



IDEA开发工具安装完成

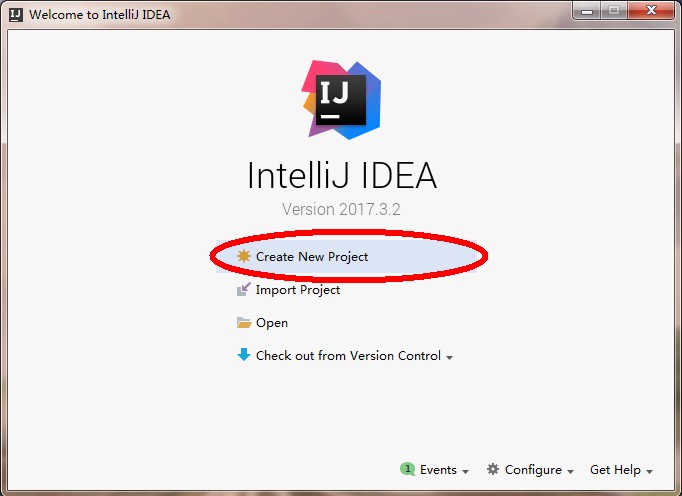
# IDEA基本使用

## IDEA创建项目

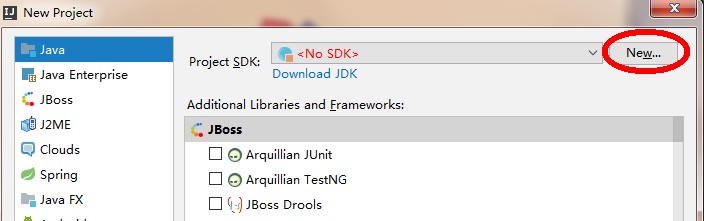
1. 选择不导入任何设置，点击OK



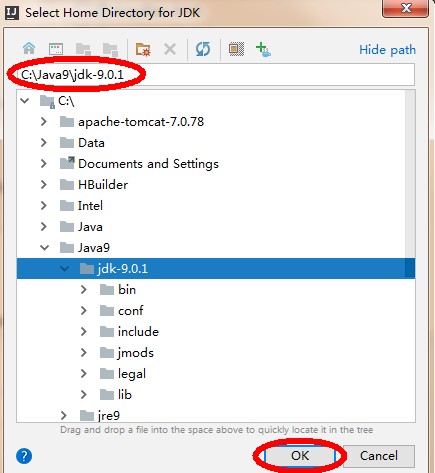
1. 选择 Create New Project

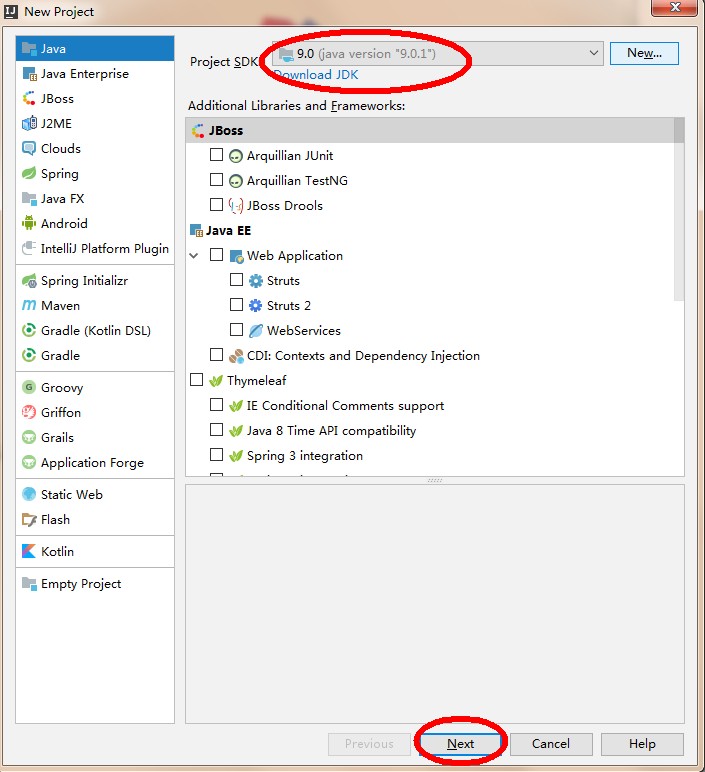


1. 点击new 按钮，配置安装的JDK9 版本

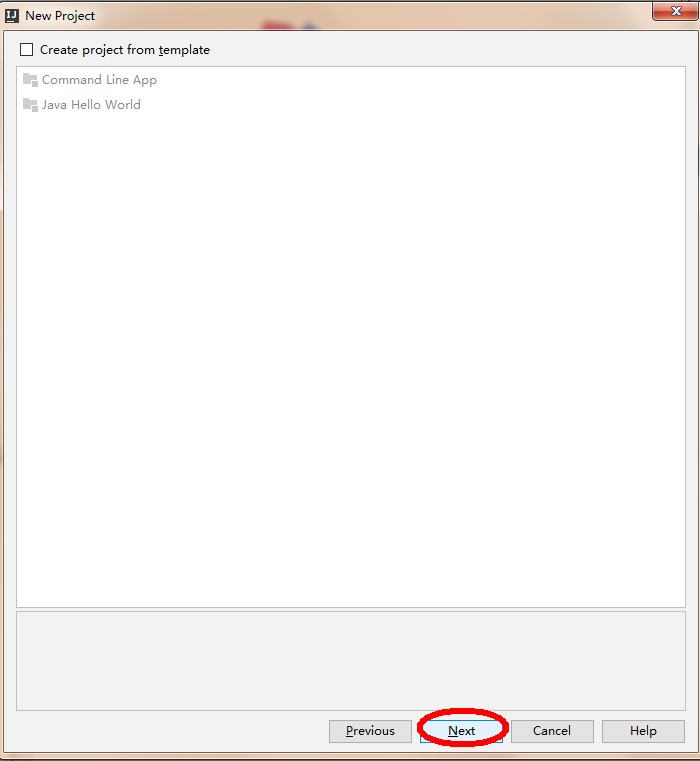


选择JDK9 目录，点击确定

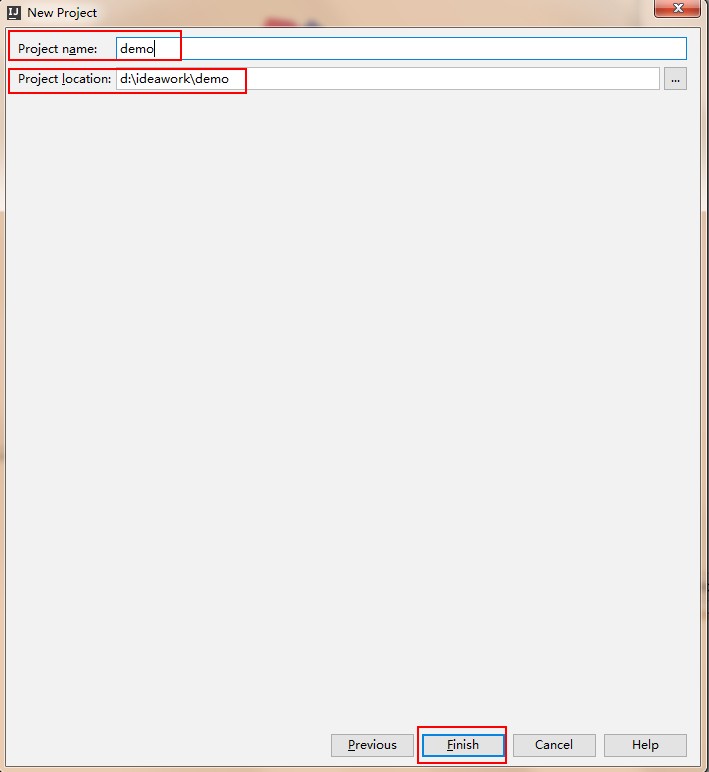




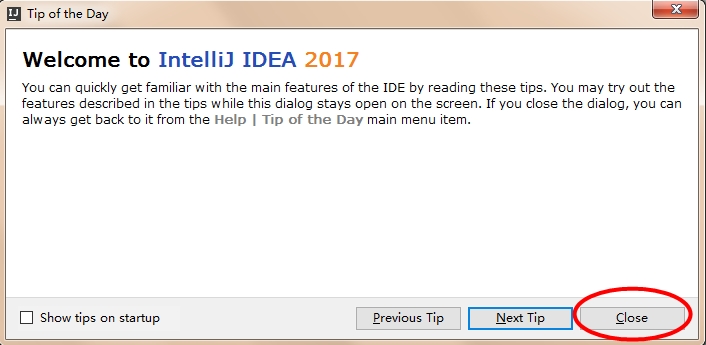
1. 不使用模板



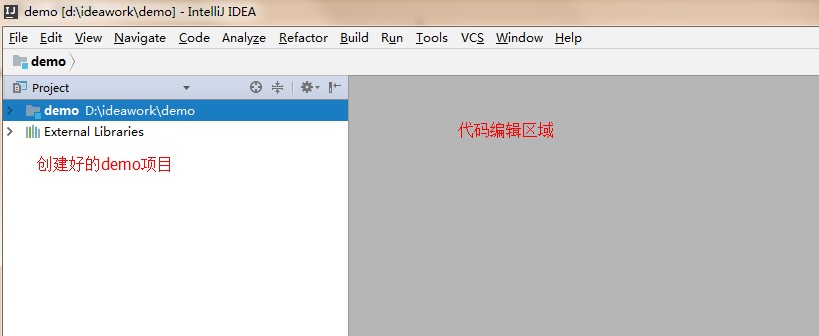
1. 为工程起名字 demo ，并存储到d:\ideawork\demo 目录下，如果d盘没有这个目录，会自动创建。



1. 打开一个每日一帖对话框，勾掉每次启动显示，点击close

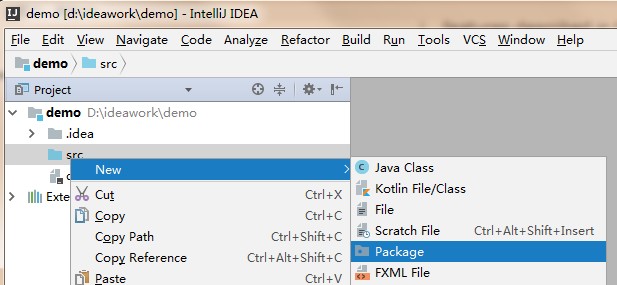


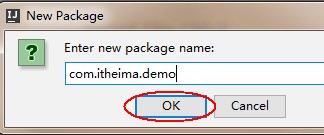
1. IDEA的工作界面，我们的项目已经创建好了



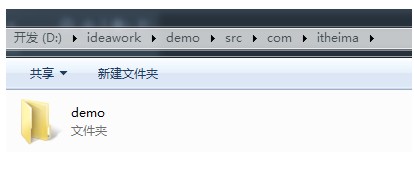
## 创建包和类

1. 展开创建的工程，在源代码目录src 上，鼠标右键，选择new->package ，键入包名com.itheima.demo ，点击确定。





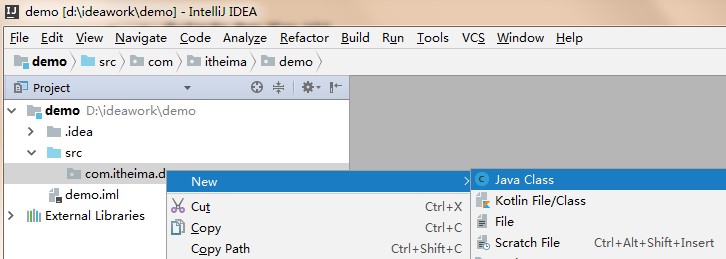
右键点击com.itheima.demo ，选择 Show in Explorer ，可以在windows下的窗体中打开创建的文件夹。

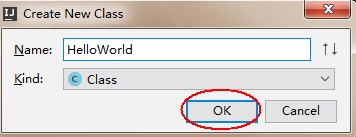


所以com.itheima.demo ，表示创建了多级的文件夹。

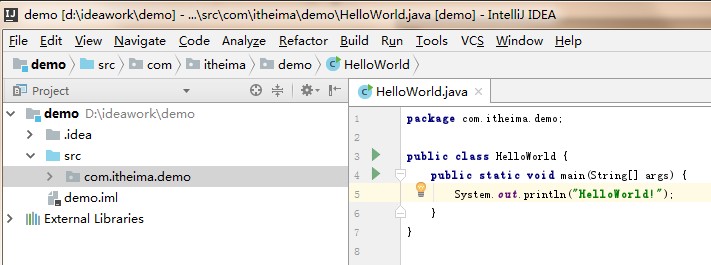
* + 小贴士：所谓包，就是文件夹，用来对类文件进行管理。

1. 在创建好的包上，鼠标右键，选择 new->class 创建类，键入类名Demo01HelloWorld。

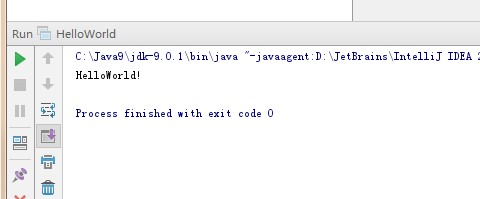




1. 在代码编辑区，键入主方法，并输出HelloWorld 。



1. 运行程序，在代码编辑区鼠标右键，选择Run HelloWorld 即可，或在菜单中选择Run->Run HelloWorld 。



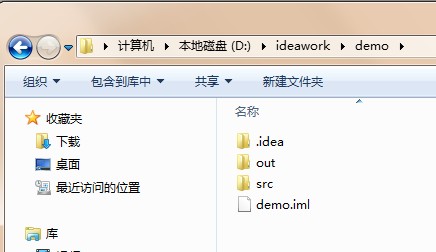
## 运行Java代码常见的方式

1. 在源代码上右键，在菜单中选择Run->Run HelloWorld
2. 在左边的行号上点绿色的三角形运行  
   
3. 点右上角的工具栏运行  
   
4. 点上面的Run菜单选择运行  
   

## IDEA的项目结构

假设我们创建的项目，在d:\ideawork目录的helloworld下

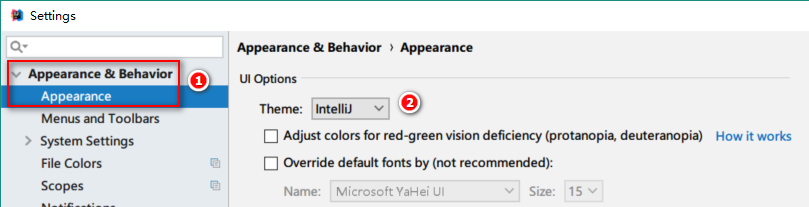
|  |  |
| --- | --- |
| **目录结构** | **说明** |
| **.idea目录和helloworld.iml文件** | 和我们开发无关，是IDEA工具自己使用的，这其实是一个XML文件 |
| **out** | 目录是存储编译后的.class文件，类似于eclipse中的bin文件夹 |
| **src** | 目录是存储我们编写的.java源文件 |



# IDEA的设置

## 主题设置

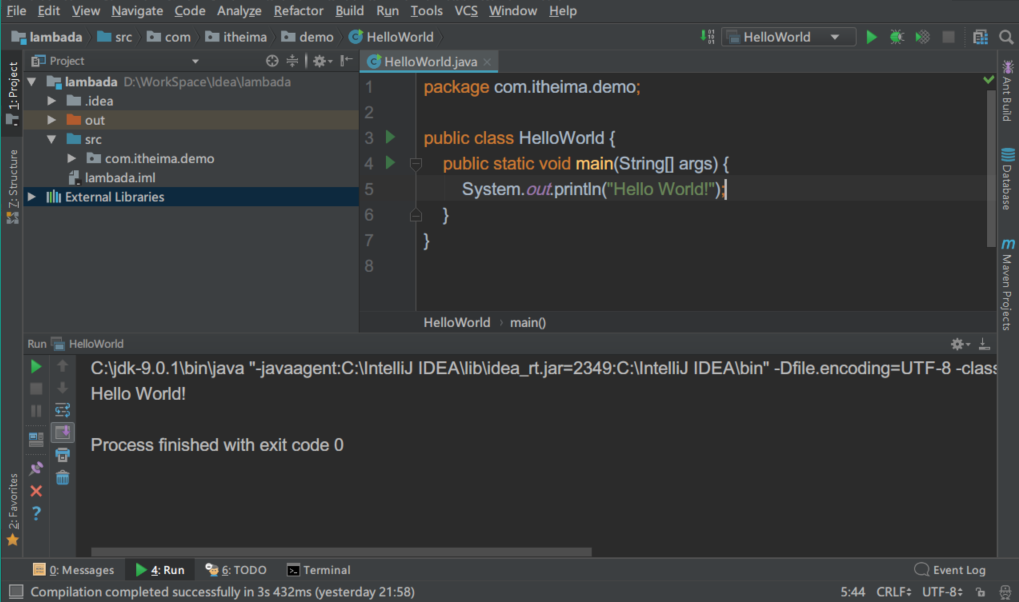
如果你喜欢不同的主题可以选择File🡪 Settings选择如下选项。



默认有如下三项可供选择：



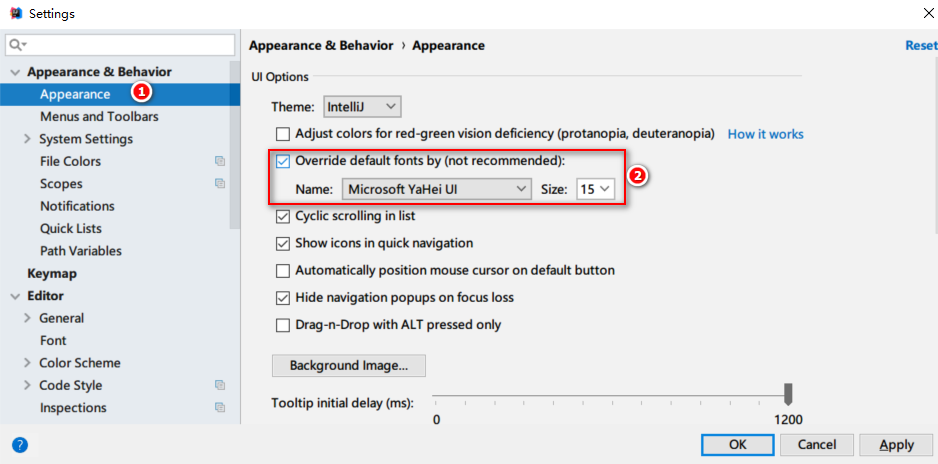
比如选择黑色主题，界面将变成这个样子：



## 字体设置

### 界面字体

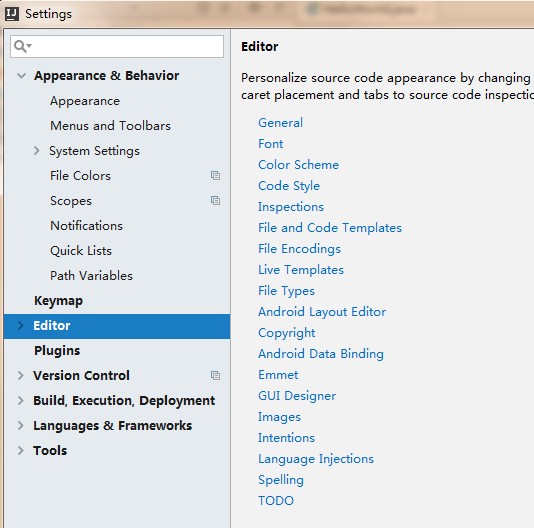
File🡪Settings🡪 Appearance & Behavior 🡪 Appearance



### 程序字体

IDEA工具的默认字体非常小，代码编辑器和控制台的输出字体都需要进行调整。

1. 点击菜单栏上的File->Settings->Editor->Font修改代码区域字体。



1. 选择自己喜欢的字体   
   Font选择字体，后面的勾表示只显示等宽字体  
   Size表示大小  
   Line spacing表示几倍行距

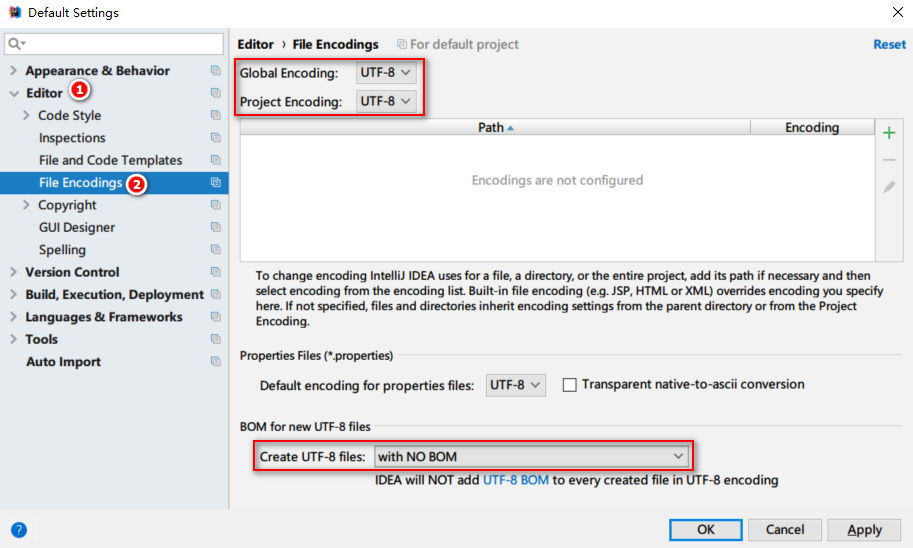


## 编码的设置

### 工程编码设置

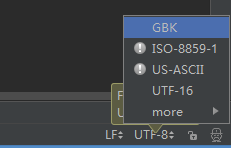
File -> Other Settings -> Default Settings -> Editor -> File Encodings

这里可以进行三项编码的设置：全局编码、工程编码、创建文件的编码，都建议设置成UTF-8



### 文件编码设置

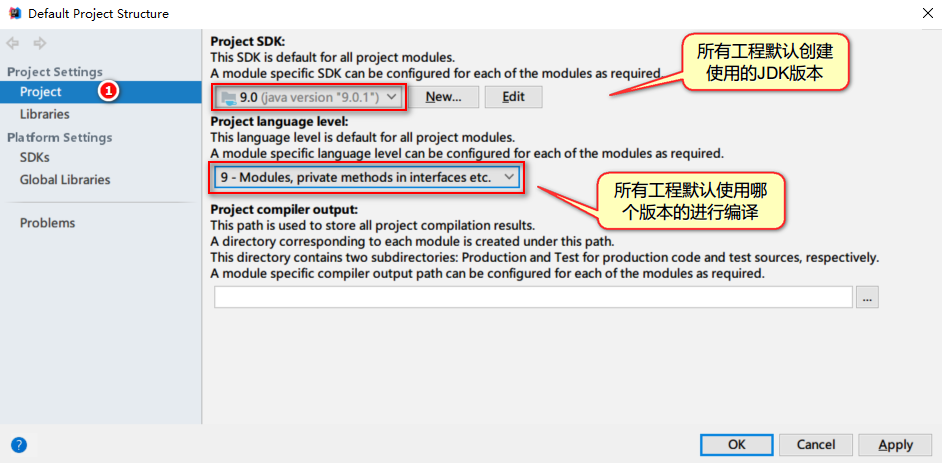
打开需要设置编码的文件，在右下角进行设置



任何开发统一编码都是第一步，为了解决中文编码，这里统一设为UTF-8

## JDK设置

File -> Other Settings -> Default Project Structure



## IDEA常用快捷键

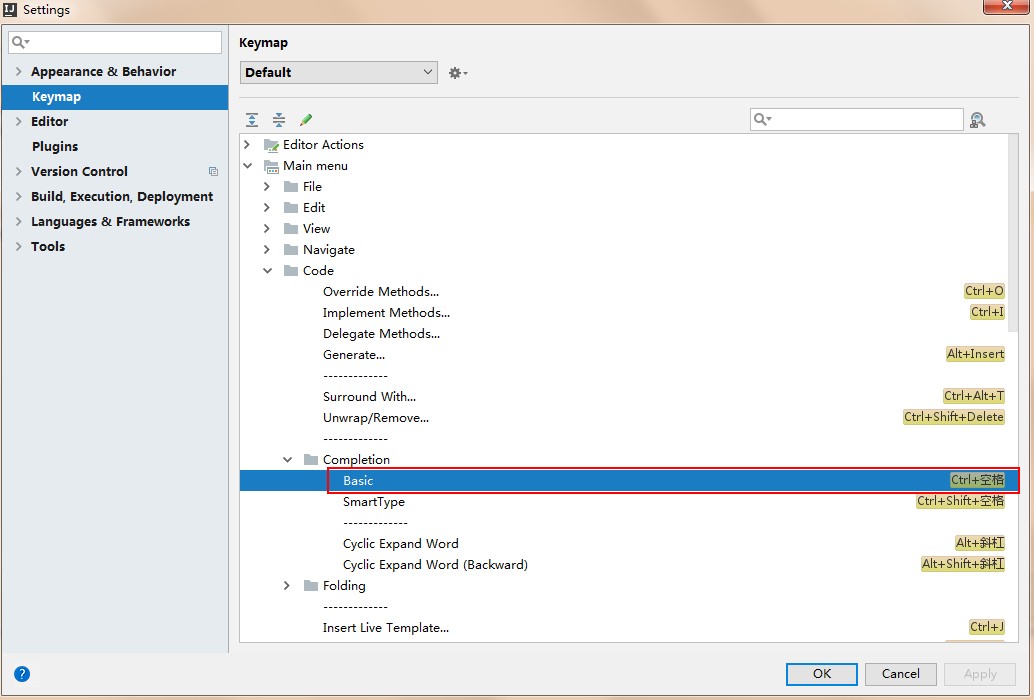
|  |  |
| --- | --- |
| **快捷键** | **功能** |
| **Ctrl+Y** | 删除光标所在行 |
| **Ctrl+D** | 复制光标所在行并把复制的内容插入光标位置下面，类似于Eclipse中ctrl+alt+下 |
| **Ctrl+Alt+L** | 格式化代码 |
| **Alt+Enter** | 导入包，异常处理等，自动修正代码。类似于Eclipse中的Ctrl+1 |
| **Shift+Enter** | 在当前行的下面插入一个空行 |
| **Ctrl+Shift+Enter** | 在当前行的末尾插入一个分号 |
| **Ctrl+/** | 注释/反注释 |
| **按两下Shift** | 出现搜索对话框，可以搜索任何东西 |
| **Ctrl+Shift+/** | 选中代码注释，多行注释，再按取消注释 |
| **Ctrl+Space** | 自动补全代码，与汉字的输入法冲突，修改成alt+/ |
| **Alt+Ins** | 自动生成代码，toString()，get()，set()等方法 |
| **Ctrl + J** | 显示代码模板 |
| **Ctrl + P** | 提示方法参数 |
| **Ctrl + Alt + T** | 先选中代码块使用代码包围：if/else、try/catch等。 |
| **Shift + F6** | 方法、变量的重命名 |
| **Ctrl + F12** | 在一个类中快速定位到指定的方法 |
| **Ctrl +ALT+O** | 优化导入的类和包 |
| **Ctrl +Shift+U** | 选中文字，大小写切换 |
| **Ctrl +Shift+V** | 从剪贴板中选择要粘入的内容 |

## 修改快捷键

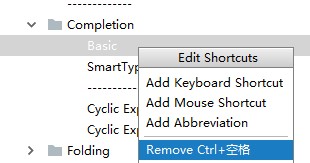
### 修改方法

在IDEA工具中，Ctrl+空格的快捷键，可以帮助我们补全代码，但是这个快捷键和Windows中的输入法切换快捷键冲突，需要修改IDEA中的快捷键。

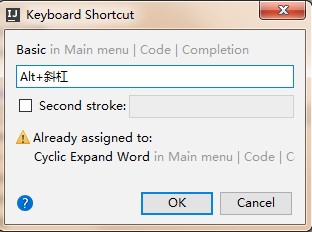
File->Settings->keymap->Main menu->code->Completion->Basic



双击Basic->remove->Ctrl+空格



再次双击Basic->Add Keyboard->键入 Alt+/-> 点击OK



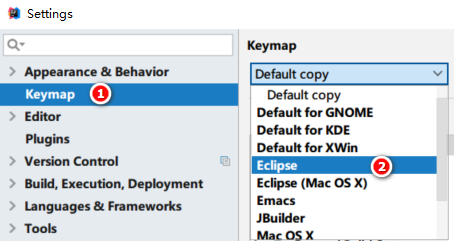
### 按自己的操作习惯可以有选择的修改

|  |  |
| --- | --- |
| **修改或添加的快捷键** | **按键** |
| **创建包或文件夹** | Ctrl+Shift+P |
| **创建类** | Ctrl+Shift+C |
| **创建HTML文件** | Ctrl+Shift+H |
| **Ctrl+Space** | ALT+/ |

### 整个修改所有的键盘主题

如果习惯了其它开发工具，比如eclipse，可以将IDEA的快捷键大部分设置成与eclipse一样。

选择File🡪Settings🡪Keymap



## Live Templates代码模板

### 常用的自带模板

|  |  |
| --- | --- |
| **常用的Java模板** | **说明** |
| **psvm** | main函数 |
| **sout** | System.out.println(); |
| **psf** | public static final |
| **fori** | for循环 |
| **ifn** | 插入if null语句 |
| **itar** | 循环数组 |
| **iter** | 对集合生成forEach的循环代码 |

### 创建自己的代码模板

如我们经常要写日期格式化代码：

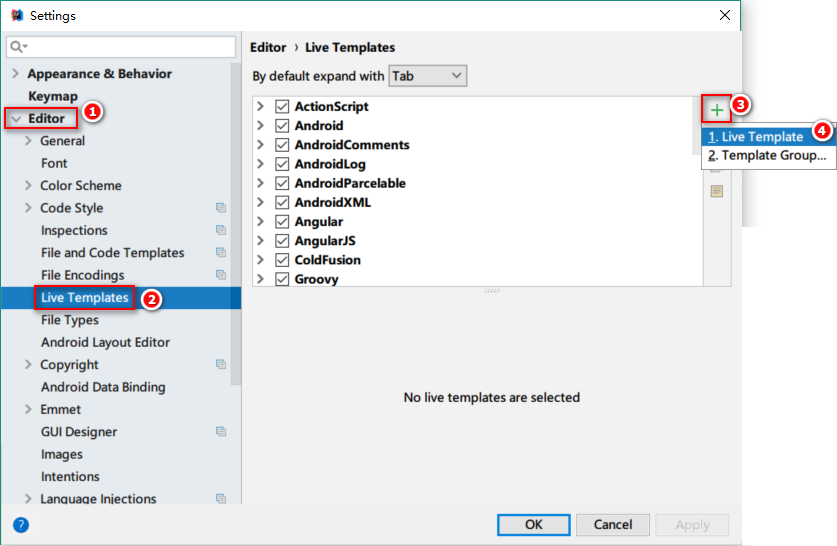
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String sdate = dateFormat.format(new Date());

System.out.println(sdate);

如果每次都手敲或复制是不是有点繁琐，这里就可以借用Live Templates实现快速代码编写。

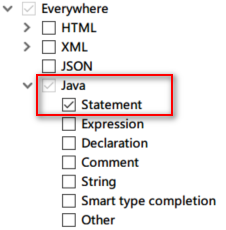
1. 使用菜单：File 🡪 Settings🡪Editor🡪Live Templates 点右边的绿色加号。



1. 指定代码模板的各项参数



1. 指定呼出的上下文位置，如：在Java的代码中



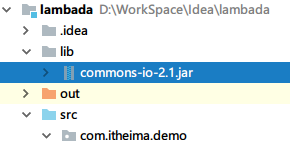
1. 测试使用，在Java中输入sdf，并且按Tab键就可以出现自定义的代码段。

## 导入第三方jar包

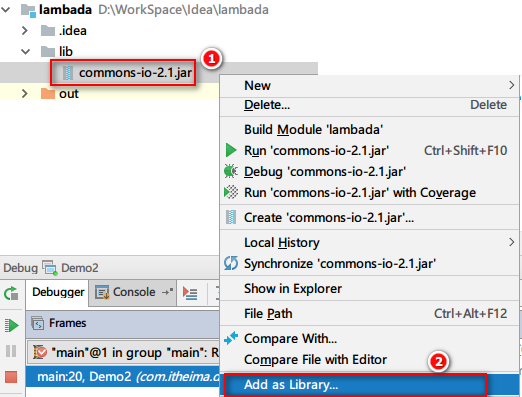
一个项目中可以包含多个模块

如果我们要使用第三方的jar包，如：commons-io包实现文件的复制，则

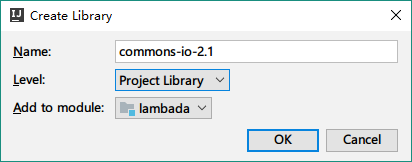
1. 在项目根目录下创建lib文件夹，并将jar包复制到该文件夹下，如下图：



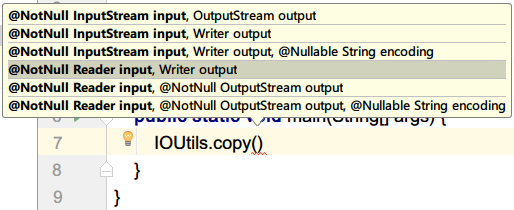
1. 在jar包上点击右键，在弹出的菜单中选择Add as Library，将jar添加成一个库。



1. 在弹出的框中输入库的名字，选择库的级别：全局库>项目库>模块库三个当中的选一个。



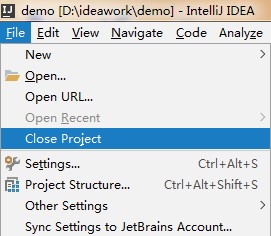
1. 之后就可以在源代码中使用第三方的jar包，与方式一的效果相同。



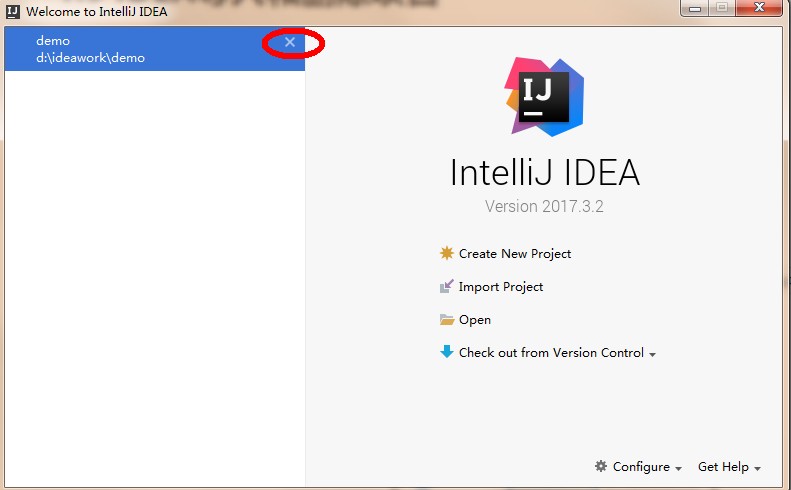
## IDEA导入和关闭项目

### 项目的关闭

* + 注：IDEA每次只打开一个项目，如果要关闭已经在编辑的项目，使用File->Close Project

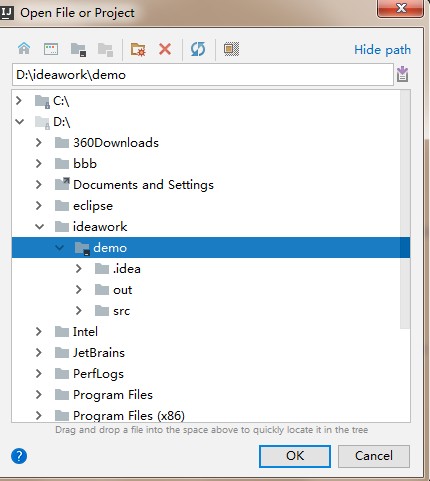


File->Close Project这时IDEA回到了刚启动界面，让我们再创建或选择一个新的项目。如果点图中红色的X，将清除历史记录，IDEA中就没有这个项目了，但硬盘上这个项目还是存在的。



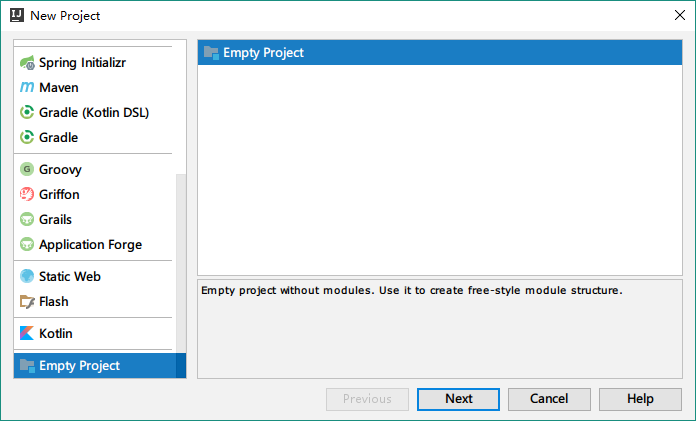
### 项目的打开

在IDEA的启动界面上，点击OPEN ，选择项目目录即可

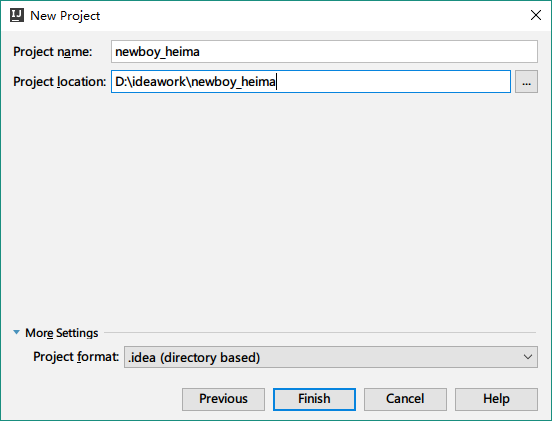


### 创建模块

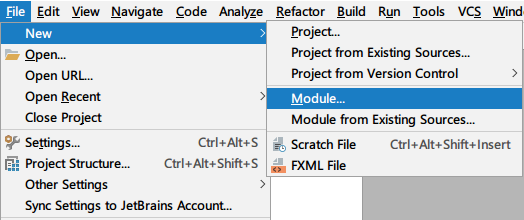
1. 创建一个空的项目，每天的内容创建成一个模块，选最下面的Empty Project创建一个空项目。



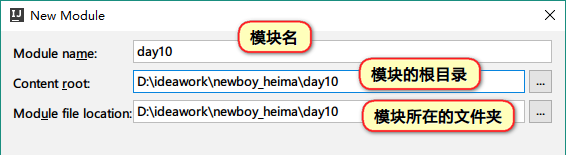
1. 给项目起名，指定项目创建的文件夹



1. 选择菜单创建一个模块



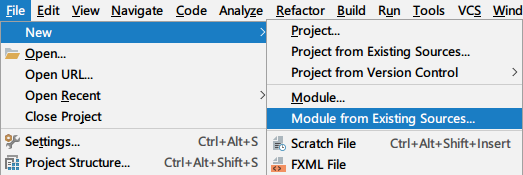
1. 指定创建模块的参数



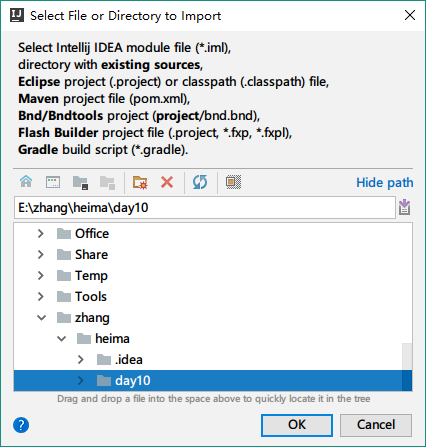
1. 创建好模块后就可以编写代码了

### 导入模块

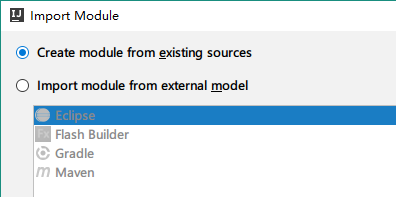
1. 创建一个空的项目或者打开一个现有的项目
2. 选择菜单创建一个模块，从已经存在的源代码来创建



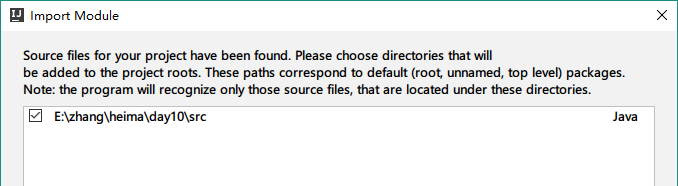
1. 选择要导入的文件或目录



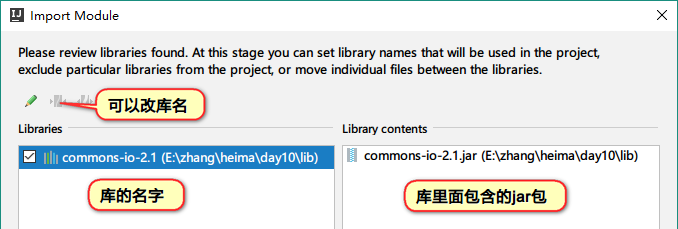
1. 选择从已经存在的源代码中创建模块



1. 选择要导入的源代码目录，并且将这个目录做为当前模块的源代码根目录，即默认包的位置。



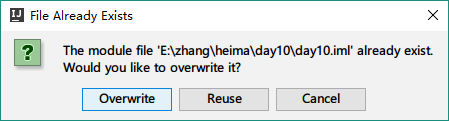
1. 如果原来的模块中有jar包，也会在这里找到。



1. 确认导入的模块名和模块依赖的包



1. 以前的模块已经存在模块描述文件，可以选择现在的文件覆盖原来的，也可以选择使用以前的。



1. 如果以前的模块中有用到其它的开发框架，这里会选择导入框架



# IDEA的Debug调试

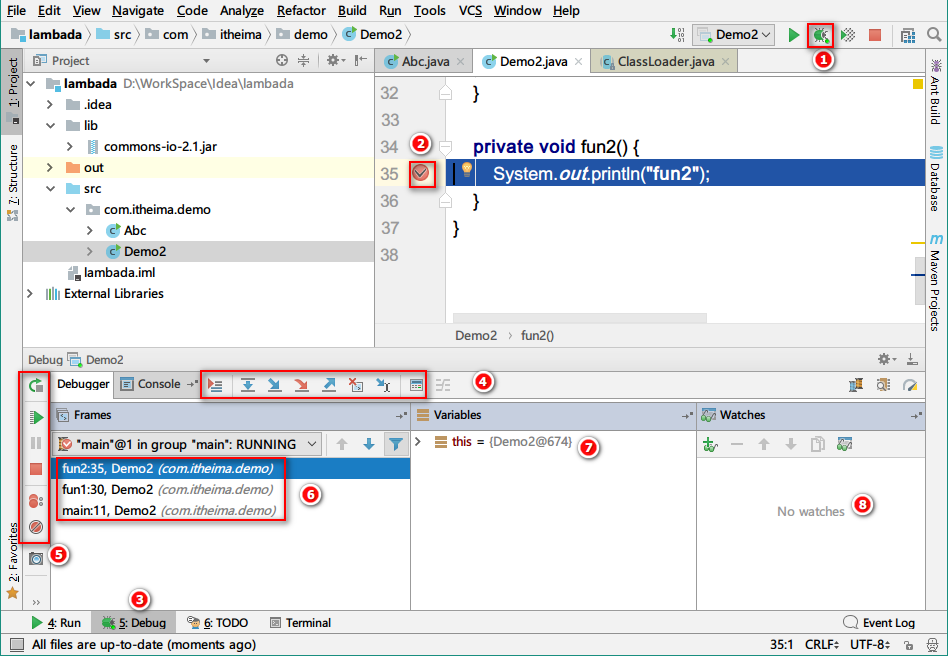
Debug用来追踪代码的运行流程，通常在程序运行过程中出现异常，启用Debug模式可以分析定位异常发生的位置，以及在运行过程中参数的变化。如下是在IDEA中启动Debug模式，进入断点后的界面。

## Debug窗口概述

输入如下代码用于调试：

1. 创建主类
2. 创建一个私有方法没有返回值fun2()，输出一句话。
3. 创建一个私有方法，返回一个字符串，传入一个字符串的参数，调用fun2()
4. 创建主函数，实例化当前主类
5. 调用fun1()，打印fun1()返回的值
6. 创建日期格式化类，对日期进行格式化，打印现在的时间
7. 创建一个1到10的循环，输出每个i的值
8. 最后创建while(true)的循环，一开始注释掉。

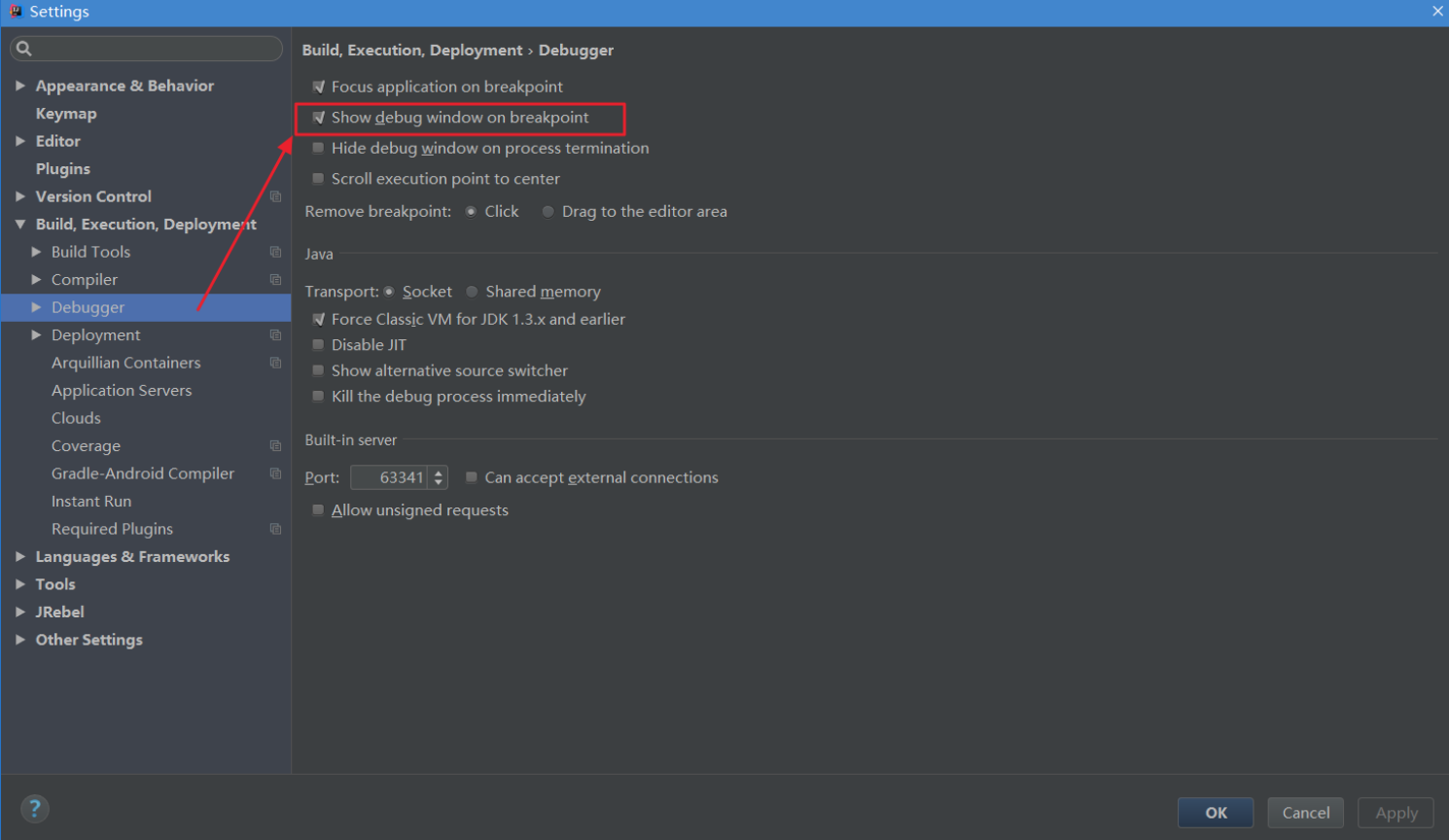
**import** java.text.SimpleDateFormat;  
**import** java.util.Date;  
  
**public class** Demo02Debug {  
 **public static void** main(String[] args) {  
 Demo02Debug demo2 = **new** Demo02Debug();  
  
 String str = demo2.fun1(**"hello"**);  
 System.***out***.println(str);  
  
 SimpleDateFormat dateFormat = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  
  
 String sdate = dateFormat.format(**new** Date());  
 System.***out***.println(sdate);  
 *//断点条件* **for** (**int** i = 0; i < 10 ; i++) {  
 System.***out***.println(**"i="** + i);  
 }  
  
 *//暂停按钮  
 /\*while(true) {  
  
 }\*/* }  
  
 **private** String fun1(String str) {  
 fun2();  
 **return** str;  
 }  
  
 **private void** fun2() {  
 System.***out***.println(**"fun2"**);  
 }  
}



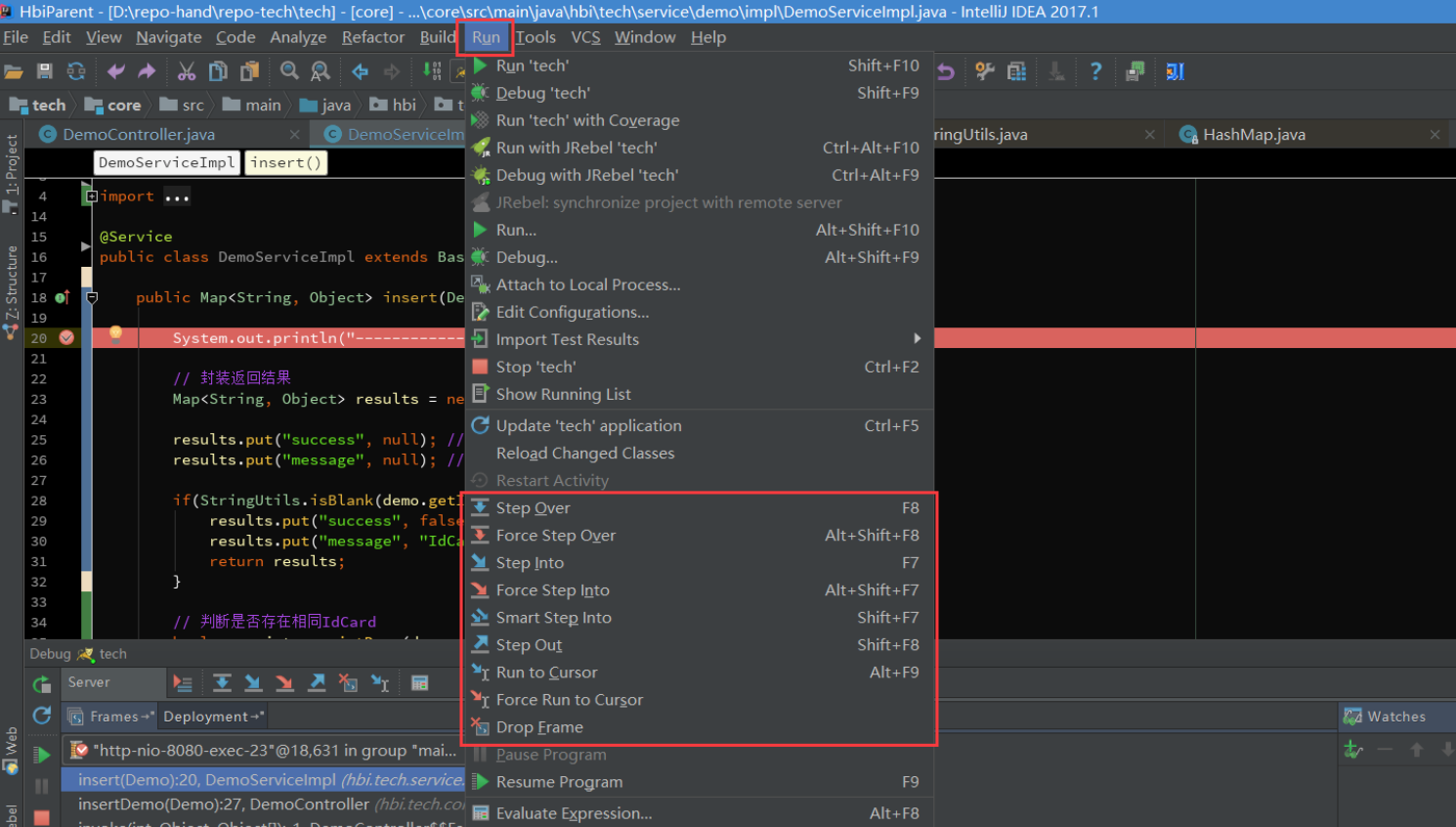
① 以Debug模式运行程序，左边的一个按钮则是以Run模式启动。在开发中，一般会直接启动Debug模式，方便随时调试代码。

② 断点：在左边行号栏单击左键，或者快捷键Ctrl+F8 打上/取消断点。

③ Debug窗口：程序执行到第一个断点后，会自动激活Debug窗口。如果没有自动激活，可以去设置里设置，如下图



④ 调试按钮：一共有8个按钮，调试的主要功能就对应着这几个按钮，鼠标悬停在按钮上可以查看对应的快捷键。在菜单栏Run里可以找到同样的对应的功能，如下图



⑤ 竖排按钮：可以在这里关闭/启动服务，设置断点等。

⑥ 方法调用栈：这里显示了该线程调试所经过的所有方法，勾选右上角的[Show All Frames] 按钮，就不会显示其它第三方类库的方法了，否则这里会有一大堆的方法。

⑦ Variables：在变量区可以查看当前断点之前的当前方法内的变量。

⑧ Watches：查看变量，可以将Variables区中的变量拖到Watches中查看。

如果看不到上面的某个窗口，可以点左边竖排按钮，将Debug窗口的布局还原到初始状态。



## 横排调试按钮



从左到右依次说明如下：

1. Show Execution Point (Alt + F10)：显示正在执行的点，如果你的光标在其它行或其它页面，点击这个按钮可跳转到代码执行到的行。
2. Step Over (F8)：步过，一行一行地往下走，如果这一行上有方法则不会进入方法，而是直接向后运行。
3. Step Into (F7)：步入，如果当前行有方法，可以进入方法内部，一般用于进入自定义方法内，不会进入官方类库的方法。
4. Force Step Into (Alt + Shift + F7)：强制步入，能进入任何方法，查看底层源码的时候可以用这个进入官方类库的方法。
5. Step Out (Shift + F8)：步出，从步入的方法内退出到方法调用处，此时方法已执行完毕，只是还没有完成赋值。
6. Drop Frame ：运行到某个方法内部时，放弃当前的函数堆栈，让你重新运行整个函数。
7. Run to Cursor (Alt + F9)：在断点已经停下来的状态下，运行到指定光标处，你可以将光标定位到你需要查看的那一行，然后使用这个功能，代码会运行至光标行，而不需要再打断点。
8. Evaluate Expression (Alt + F8)：临时计算一个表达式。

## 竖排调试按钮



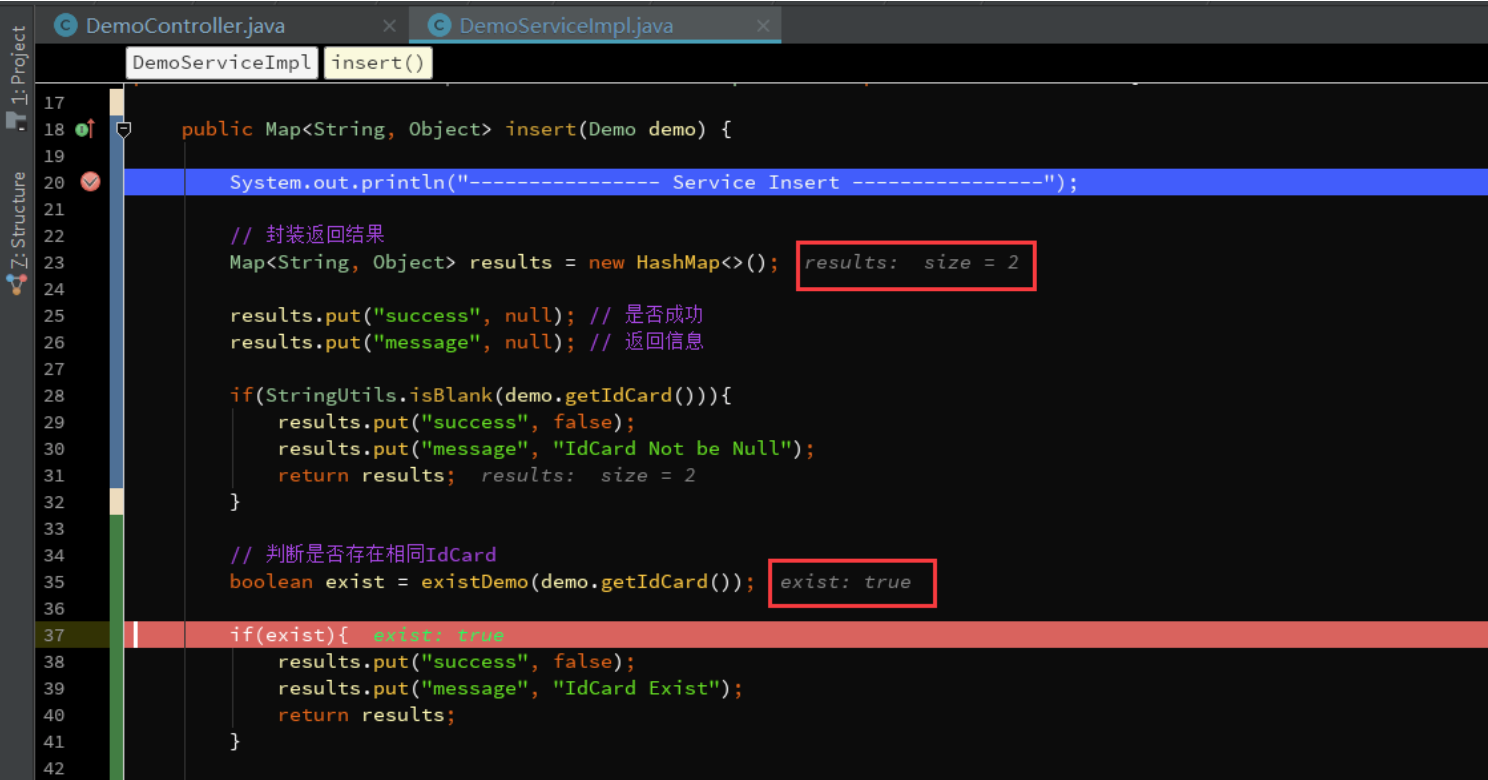
从上到下依次说明如下：

1. Rerun 'xxxx'：重新运行程序，会关闭服务后重新启动程序。
2. Resume Program (F9)：继续程序，比如，你在第20行和25行有两个断点，当前运行至第20行，按F9，则运行到下一个断点(即第25行)，再按F9，则运行完整个流程，因为后面已经没有断点了。
3. Pause Program：暂停程序，如果代码在一个长时间的循环或死循环中，可以使用这个按钮暂停程序。
4. Stop 'xxx' (Ctrl + F2)：停止/关闭程序。有时候你会发现关闭服务再启动时，报端口被占用，这是因为没完全关闭服务的原因，你就需要查杀所有JVM进程了。
5. View Breakpoints (Ctrl + Shift + F8)：查看所有断点。
6. Mute Breakpoints：哑的断点，选择这个后，所有断点变为灰色，断点失效，按F9则可以直接运行完程序。再次点击，断点变为红色，有效。

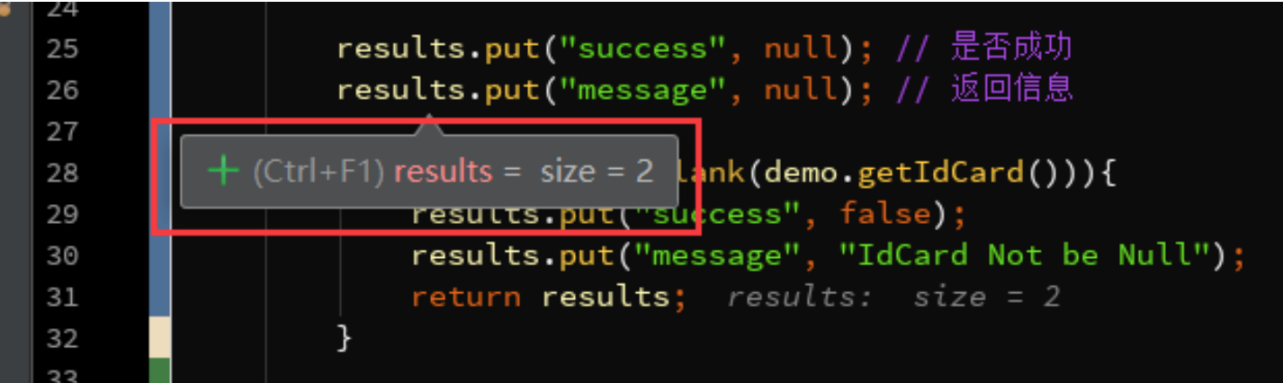
## 查看变量值的几种方式

在Debug过程中，跟踪查看变量的变化是非常必要的，下面是IDEA中可以查看变量的几个地方

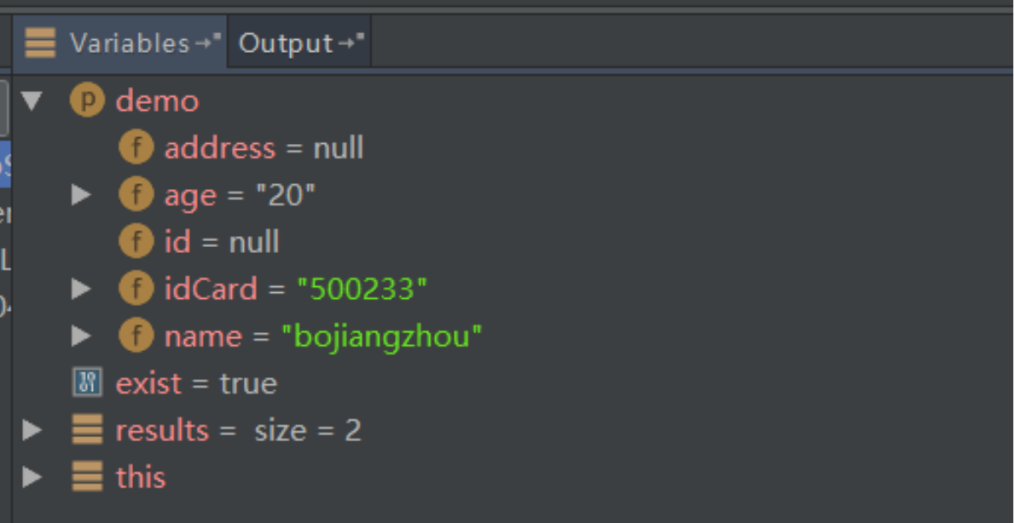
1. 如下，在IDEA中，参数所在行后面会显示当前变量的值。



1. 光标悬停到参数上，显示当前变量信息



1. 在Variables里查看，这里显示当前方法里的所有变量

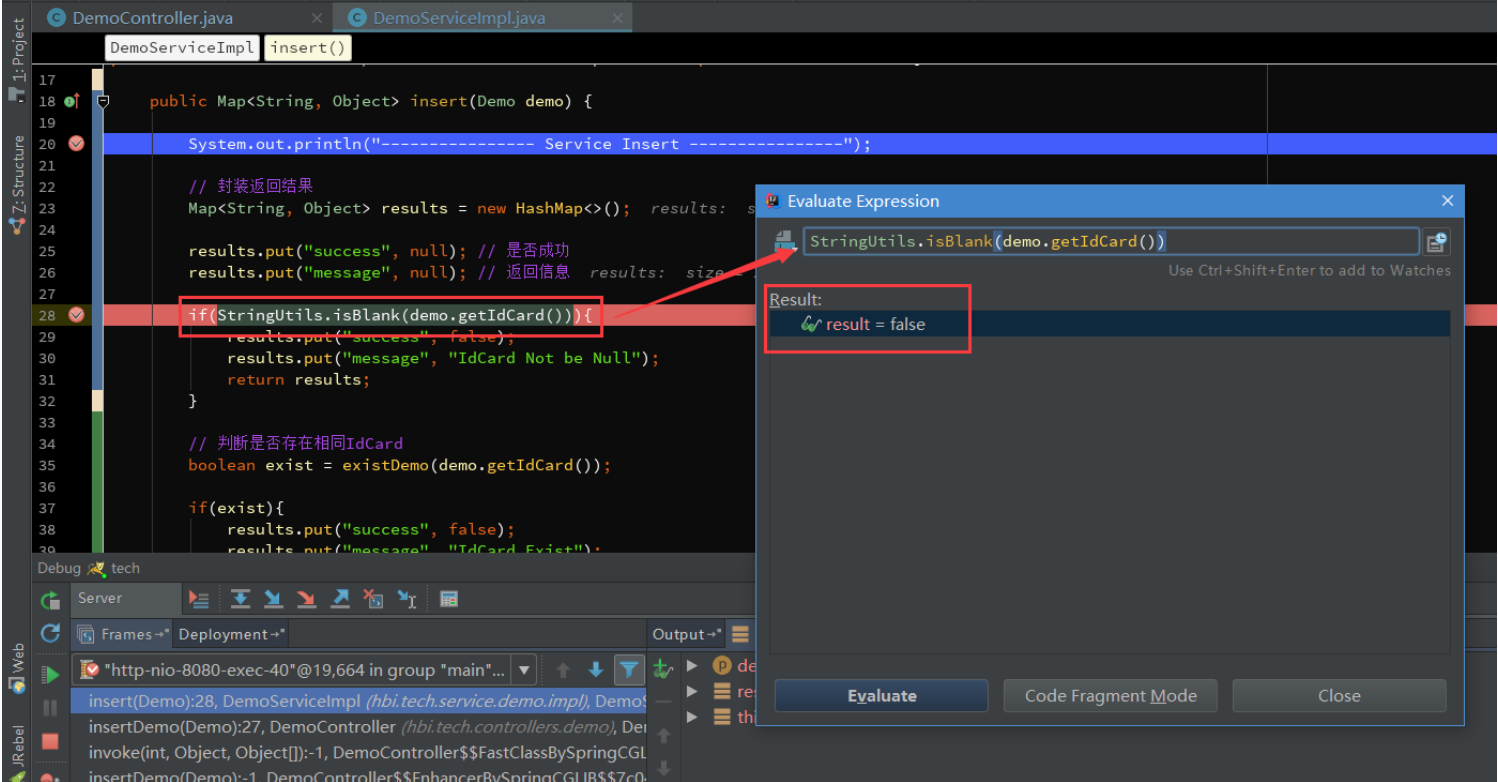


## 计算表达式窗口

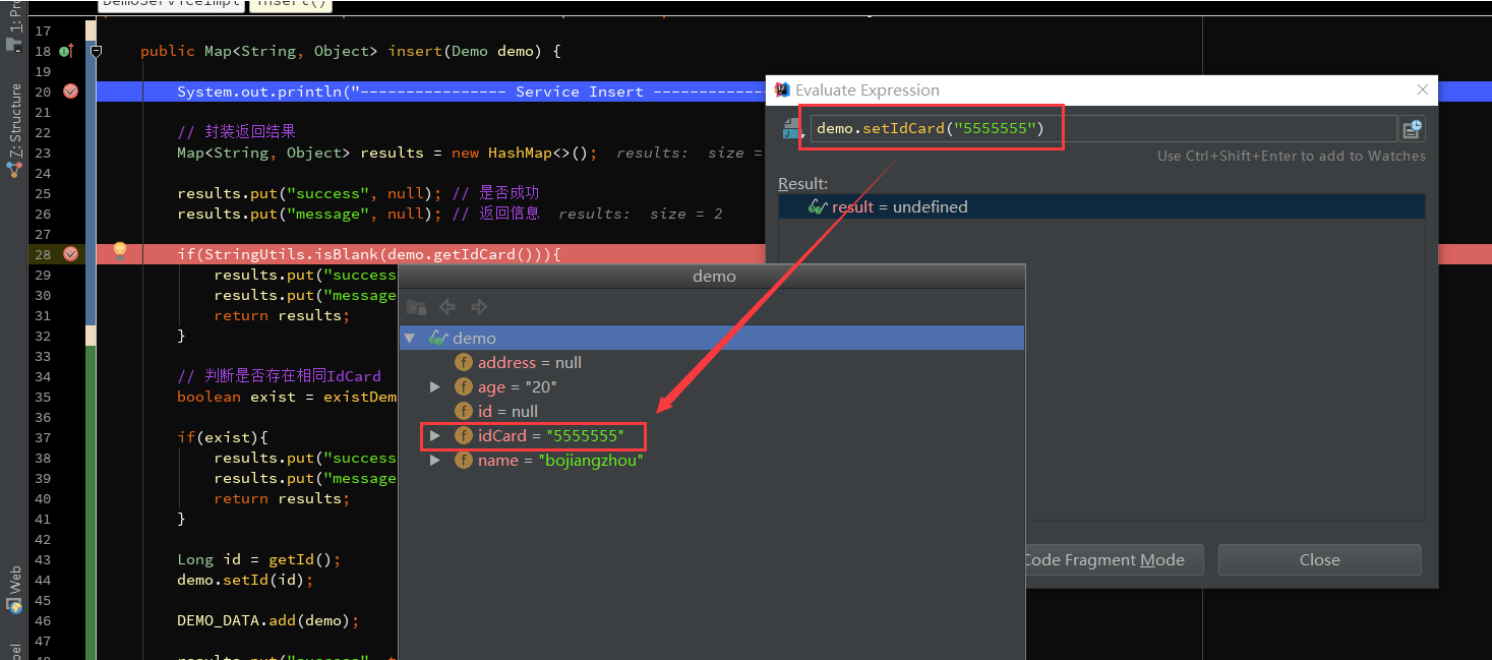
在第一组调试按钮提到的计算表达式按钮，如下图，Evaluate Expression (Alt + F8) 。可以使用这个操作在调试过程中计算某个表达式的值，而不用再去打印信息。



1. 在调试状态下，按Alt + F8或按钮，或者你可以选中某个表达式再Alt + F8，弹出计算表达式的窗口。回车或点击Evaluate计算表达式的值。这个表达式不仅可以是一般变量或参数，也可以是方法，当你的一行代码中调用了几个方法时，就可以通过这种方式查看某个方法的返回值。



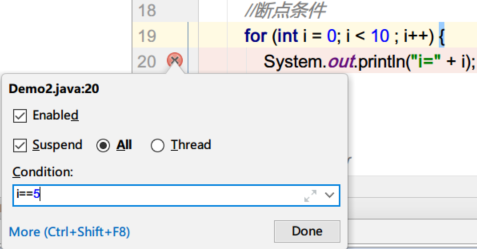
1. 设置变量，在计算表达式的框里，可以改变方法参数的值，就能很方便我们去调试各种值的情况了。



## 条件断点

通过设置断点条件，在满足条件时，才停在断点处，否则直接运行。通常，当我们在遍历一个比较大的集合或数组时，在循环内设置了一个断点，难道我们要一个一个去看变量的值？那肯定很累，说不定你还错过这个值得重新来一次。

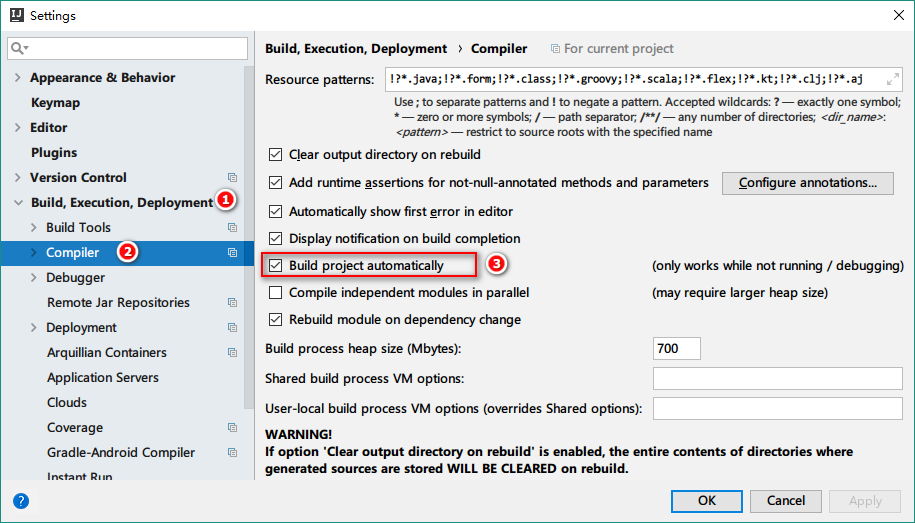
在断点上右键直接设置当前断点的条件，当i==5时断点才生效。在断点上点鼠标右键进行设置。



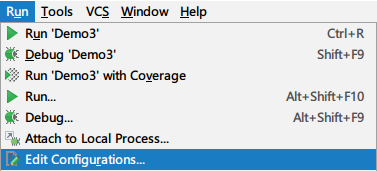
## 提升代码启动的速度

IDEA启动Java的速度很慢，为什么呢？因为每次执行前都需要build，即编译。就是这个编译时间让运行速度显得特别慢。我们可以进行两步设置：先让代码运行前自动编译，然后取消掉每个程序启动前的编译，达到提升程序启动速度的目的。

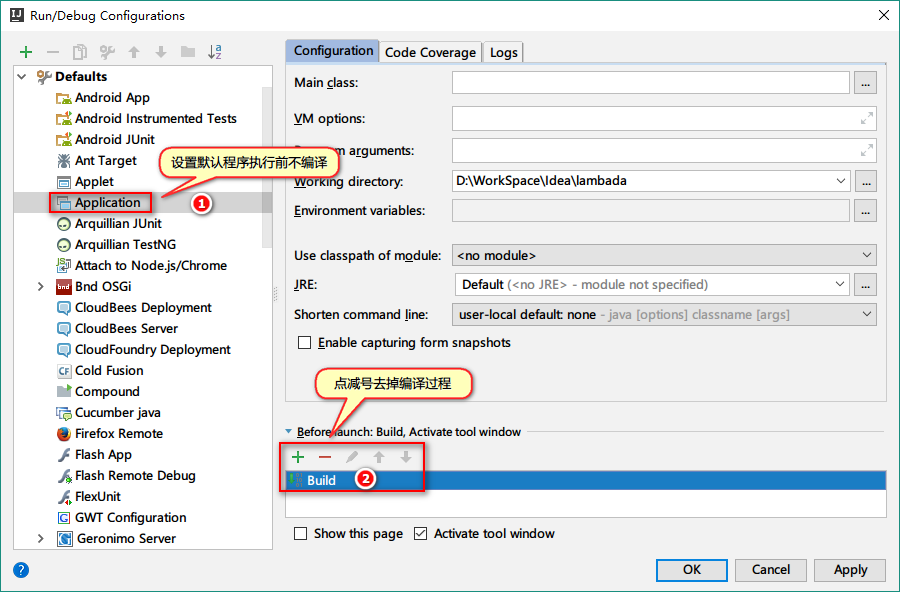
1. 选择File 🡪 Settings 🡪 Build,Exeution,Deployment🡪 Compiler设置为自动编译。



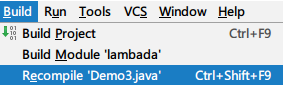
1. 选择运行菜单下的编辑配置



1. 设置程序在启动前不进行编译



1. 通过以上设置后会发现程序启动运行速度快多了。不过有时也会出现找不到字节码文件的副作用，因为程序可能没有提前编译，如果出现这样的问题，可以自己手动编译一次。



# Java的比较器

## java.lang.Comparable接口

### 什么是Comparable接口

对实现它的每个类的对象进行整体排序。此排序被称为该类的自然排序，接口中的compareTo() 方法被称为它的比较方法。实现此接口的对象集合（或数组）可以通过 Collections.sort （或Arrays.sort ）进行排序。实现此接口的对象可以用作有序映射表TreeMap中的键或有序集合List中的元素，无需指定比较器。

### Comparable接口中的方法

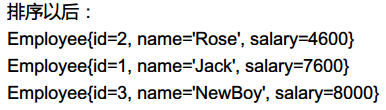
|  |  |
| --- | --- |
| **接口中的方法** | **说明** |
| **int compareTo(T o)** | 比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数。  这是Java的一种约定，当排序的sort()方法在对集合中的元素两两进行比较的时候会调用我们实现的compareTo(T o)方法。如果compareTo()返回正整数，则表示当前元素大于和它比较的元素；如果返回负整数则表示当前元素小于和它比较的元素；如果返回0，则表示两者相等。 |
| 参数：o - 要比较的对象。  返回值：  this < obj 返回负  this == obj 返回 0  this > obj 返回正  抛出：ClassCastException - 如果指定对象的类型不允许它与此对象进行比较。 |

### 案例演示

* 案例需求：

在数组中有三个员工对象，员工包含三个属性：编号、姓名、工资。我们使用Arrays.sort()方法对数组中的员工按工资从小到大排序。

* 运行效果：



* 案例分析：

1. 创建员工实体类，包含属性int id，String name和int salary。
2. 通过员工的构造方法传入三个属性，生成toString()方法，生成无参的构造方法。
3. 员工实现Comparable接口，并且实现compareTo()方法：用当前员工工资减去另一个员工的工资
4. 创建类Demo03Comparable在main函数中创建三个元素的数组，并且创建三个员工，添加到数组中。
5. 调用Arrays.sort()方法对员工进行排序
6. 思考：如果要按工资的降序排序如何实现?如果要用id排序呢?

* 案例代码：

Employee.java

public class Employee implements Comparable<Employee> {

private int id;

private String name;

private int salary;

@Override

public String toString() {

return "Employee{" +

"id=" + id +

", name='" + name + '\'' +

", salary=" + salary +

'}';

}

public Employee() {

}

public Employee(int id, String name, int salary) {

this.id = id;

this.name = name;

this.salary = salary;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getSalary() {

return salary;

}

public void setSalary(int salary) {

this.salary = salary;

}

@Override

public int compareTo(Employee o) {

return o.salary - this.salary;

}

}

Demo03Comparable.java

import java.util.Arrays;

public class Demo3Comparable {

public static void main(String[] args) {

//创建三个员工

Employee[] employees = {

new Employee(1,"孙悟空",3000),

new Employee(2,"孙悟天",8000),

new Employee(3,"孙悟饭",5000)

};

//输出每个员工

System.out.println("排序前：");

for (int i = 0; i < employees.length; i++) {

Employee employee = employees[i];

System.out.println(employee);

}

//排序

Arrays.sort(employees);

System.out.println("排序后：");

for (int i = 0; i < employees.length; i++) {

Employee employee = employees[i];

System.out.println(employee);

}

}

}

## java.util.Comparator接口

### 什么是Comparator接口

除了上面使用java.lang.Comparable接口实现元素的比较之外，还可以使用另一个接口java.util.Comparator接口也可以实现同样的功能。

### Comparator接口中的方法

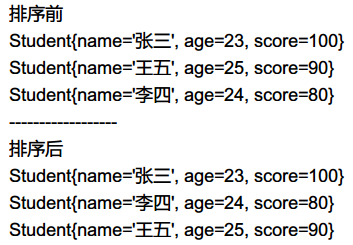
|  |  |
| --- | --- |
| **接口中的方法** | **说明** |
| **int compare(T o1, T o2)** | 作用:如果o1小于o2,返回一个负数;如果o1大于o2，返回一个正数;如果他们相等，则返回0; |
|  | 参数：  o1 - 要比较的第一个对象。  o2 - 要比较的第二个对象。  返回：根据第一个参数小于、等于或大于第二个参数分别返回负整数、零或正整数。  抛出：ClassCastException - 如果参数的类型不允许此 Comparator 对它们进行比较。 |

### 案例演示

* 需求说明

1. 定义一个学生类，成员变量有：姓名，年龄
2. 创建多个学生对象添加到集合中
3. 创建类Demo04Comparator按年龄从小到大对集合的学生对象进行排序

* 运行效果



* 代码实现

Student.java

public class Student {

private String name;

private int age;

private int score;

@Override

public String toString() {

return "Student{" +

"name='" + name + '\'' +

", age=" + age +

", score=" + score +

'}';

}

public Student() {

}

public Student(String name, int age, int score) {

this.name = name;

this.age = age;

this.score = score;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public int getScore() {

return score;

}

public void setScore(int score) {

this.score = score;

}

}

Demo04Comparator.java

import java.util.\*;

public class Demo4Comparator {

public static void main(String[] args) {

List<Student> students = new ArrayList<>();

students.add(new Student("张三",20,98));

students.add(new Student("李四",18,68));

students.add(new Student("王五",30,80));

System.out.println("排序前：");

for (Student student : students) {

System.out.println(student);

}

//集合的排序

// Collections.sort(students, new AgeComparator());

Collections.sort(students, new ScoreComparator());

System.out.println("排序后：");

for (Student student : students) {

System.out.println(student);

}

}

}

//创建一个年龄 比较器的类

class AgeComparator implements Comparator<Student> {

@Override

public int compare(Student o1, Student o2) {

return o1.getAge() - o2.getAge();

}

}

//按分数的降序

class ScoreComparator implements Comparator<Student> {

@Override

public int compare(Student o1, Student o2) {

return o2.getScore() - o1.getScore();

}

}

## Comparable和Comparator的区别

Comparable & Comparator 都是用来实现集合中的排序的，只是 Comparable 是在对象内部定义的方法实现的排序，Comparator 是在集合外部实现的排序。

Comparable 是一个对象本身就已经支持自比较，需要实现的接口（如 String、Integer 自己就可以完成比较大小操作）; 而 Comparator 是一个专用的比较器，当这个对象不支持自比较或者自比较函数不能满足你的要求时，你可以写一个比较器来完成两个对象之间大小的比较。 可以说一个是类自已完成比较，一个是外部程序实现比较的差别而已。

## Comparable和Comparator的小结：

1. Comparator位于包java.util下，而Comparable位于包java.lang下，Comparable接口将比较代码嵌入自身类中，而Comparator在一个独立的类中实现比较。
2. 如果类的设计师没有考虑到Compare的问题而没有实现Comparable接口，可以通过Comparator 来实现比较算法进行排序。
3. Comparator为了使用不同的排序规则做准备。比如：升序、降序或按不同的属性进行排序。

## 字符串比较大小

### 字符串比较的规则

两个字符串比较大小，先比较第1个字符的ASCII码值。如果第1个字符相等。则比较第2个字符，依此类推。直至比较的字符串有一方全比较完，这时就比较字符串的长度。



* 所有大写字母的ASCII码值都小于小写字母的值
* 所有的数字都小于字母

|  |  |  |
| --- | --- | --- |
| **字符串1** | **结果** | **字符串2** |
| **"abcde"** | < | **"ac"** |
| **"abc"** | > | **"ab"** |
| **"ab"** | > | **"AB"** |
| **"1000"** | < | **"2"** |

另外String类已经实现了Comparable接口

### 字符串排序案例

* 案例需求：

有一个字符串数组，包含5个字符串，请按从小到大的顺序输出字符串。

* 案例效果：



* 案例代码：

import java.util.Arrays;

public class Demo5StringComp {

public static void main(String[] args) {

String[] arr = {"Jack","Rose","make","tom","1000","9"};

System.out.println("排序前：" + Arrays.toString(arr));

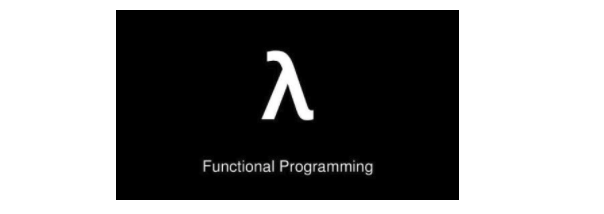
Arrays.sort(arr);

System.out.println("排序后：" + Arrays.toString(arr));

}

}

# Lambda表达式引入



## 什么是λ表达式

作为比较老牌的面向对象的编程语言Java，在对函数式编程的支持上一直不温不火。

认为面向对象式编程就应该纯粹的面向对象，于是经常看到这样的写法：如果你想写一个方法，那么就必须把它放到一个类里面，然后new出来对象，对象调用这个方法，这种方式在函数式编程语言看来太死板。Java8也开始加入这种函数式编程特性，Java码农们终于在之前老土的方法之外有了一种更为简便的选择。

Lambda表达式即函数式编程，在数学中，函数就是有输入量、输出量的一套计算方案，也就是“拿什么东西做什么事情”。相对而言，面向对象编程过分强调“必须通过对象的形式来做事情”，而函数式思想则尽量忽略面向对象的复杂语法——强调做什么，而不是以什么形式做。

## 冗余的Runnable代码

### 传统写法

当需要启动一个线程去完成任务时，通常会通过java.lang.Runnable接口来定义任务内容，并使用java.lang.Thread类来启动该线程，在任务中输出一句话。代码如下：

import org.junit.Test;

public class Demo6Run {

public static void main(String[] args) {

//创建一个Runable接口的实现类

Runnable runnable = new Runnable() {

@Override

public void run() {

System.out.println("后台线程执行");

}

};

//开启线程

new Thread(runnable).start();

System.out.println("主线程");

}

}

本着“一切皆对象”的思想，这种做法是无可厚非的：首先创建一个Runnable接口的匿名内部类对象来指定任务内容，再将其交给一个线程来启动。

### 代码分析

对于Runnable的匿名内部类用法，可以分析出几点内容：

* Thread类需要Runnable接口作为参数，其中的抽象run方法是用来指定线程任务内容的核心；
* 为了指定run的方法体，**不得不**需要Runnable接口的实现类；
* 为了省去定义一个RunnableImpl实现类的麻烦，**不得不**使用匿名内部类；
* 必须重写抽象run方法，所以方法名称、方法参数、方法返回值**不得不**再写一遍，且不能写错；
* 而实际上，似乎**只有方法体才是关键所在**。

## 编程思想转换

### 做什么，而不是怎么做

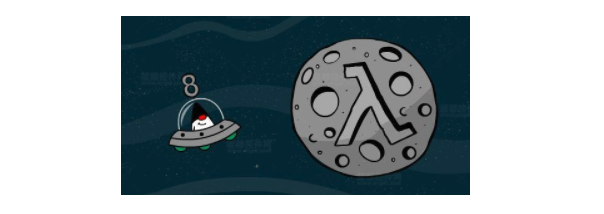
我们真的希望创建一个匿名内部类对象吗？不。我们只是为了做这件事情而**不得不**创建一个对象。我们真正希望做的事情是：将run方法体内的代码传递给Thread类知晓。

**传递一段代码**——这才是我们真正的目的。而创建对象只是受限于面向对象语法而不得不采取的一种手段方式。那，有没有更加简单的办法？如果我们将关注点从“怎么做”回归到“做什么”的本质上，就会发现只要能够更好地达到目的，过程与形式其实并不重要。

### 生活举例



当我们需要从北京到上海时，可以选择高铁、汽车、骑行或是徒步。我们的真正目的是到达上海，而如何才能到达上海的形式并不重要，所以我们一直在探索有没有比高铁更好的方式——搭乘飞机。



而现在这种飞机（甚至是飞船）已经诞生：2014年3月Oracle所发布的Java 8（JDK 1.8）中，加入了**Lambda表达式**的重量级新特性，为我们打开了新世界的大门。

# Lambda的格式

## Lambda格式

|  |
| --- |
| **Lambda表达式的标准格式为** |
| **(参数列数) -> {方法体}** |

|  |
| --- |
| **Lambda表达式的三个组成部分：** |
| **() 方法的参数** |
| **-> 固定写法** |
| **{} 写方法体** |

## 体验Lambda的更优写法

借助Java 8的全新语法，上述Runnable接口的匿名内部类写法可以通过更简单的Lambda表达式达到等效：

Runnable runnable = ()-> {System.out.println("我是Lambda");};

new Thread(runnable).start();

System.out.println("主线程");

这段代码和刚才的执行效果是完全一样的，可以在1.8或更高的编译级别下通过。从代码的语义中可以看出：我们启动了一个线程，而线程任务的内容以一种更加简洁的形式被指定。

不再有“**不得不创建接口对象**”的束缚，不再有**“抽象方法覆盖重写**”的负担，就是这么简单！

## Lambda的使用前提

Lambda的语法非常简洁，完全没有面向对象复杂的束缚。但是使用时有几个问题需要注意：

1. 使用Lambda必须具有接口，且要求**接口中有且仅有一个抽象方法**。 无论是JDK内置的Runnable、Comparator接口还是自定义的接口，只有当接口中的抽象方法存在且唯一时，才可以使用Lambda。
2. 使用Lambda必须具有**上下文推断**。也就是方法的参数类型必须为Lambda对应的接口类型(如：上面的Runnable接口)，才能使用Lambda作为该接口的实例。

## Lambda省略格式

在Lambda标准格式的基础上，使用省略写法的规则为：

1. 小括号内参数的类型可以省略；
2. 如果小括号内有且仅有一个参数，则小括号()可以省略；
3. 如果大括号内有且仅有一条语句，可以省略大括号 {}、语句结尾的分号和返回值用到的 return。如 (x,y) -> x + y 只有一条求和语句。

## Lambda格式小结：

λ表达式本质上是一个匿名方法。让我们来看下面这个例子，两个整数相加返回一个整数：

int add(int m, int n)

转成λ表达式后是完整的写法：

(int m, int n) -> {return m+n;}

参数类型可以省略，Java编译器会根据上下文推断出来，返回值：

(m,n)->{return m+n;}

因为方法体只有一条语句，可以省略return、大括号和分号

(m,n) -> m+n

如果只有一个参数，则小括号也可以省略

int sum(int m)

m -> {return m;}

# Lambda基本使用

## 无参无返回值

* 需求说明

1. 给定一个厨子Cook接口，内含唯一的方法makeFood()，且无参数、无返回值。
2. 创建Demo05InvokeCook类的main函数
3. 直接实例化Cook接口，使用匿名内部类，实现makeFood()方法，输出：做饭啦!
4. 使用Lambda的实例化Cook对象，调用makeFood()方法，输出：“做菜啦!”

* 运行效果



* 案例代码：

import org.junit.Test;

//厨师

interface Cook {

void makeFood();

}

public class Demo6Lambda {

@Test

public void testCook() {

//匿名内部类

Cook cook = new Cook() {

@Override

public void makeFood() {

//实现方法

System.out.println("做饭啦!");

}

};

//调用方法

cook.makeFood();

//lambda

Cook cook1 = () -> System.out.println("做菜啦！");

cook1.makeFood();

}

}

## 有参有返回值

* 案例需求

1. 给定一个计算器Calculator接口，内含抽象方法int calc(int a, int b) 将两个int数字相加得到和并且返回。
2. 创建Demo06InvokeCalc类和main函数
3. 使用匿名内部类实例化对象c1，并且调用calc()方法输出结果。
4. 使用Lambda实例化对象c2，并且调用calc()方法输出结果。

* 案例效果



* 案例代码

Calculator.java

interface Calculator {

int add(int m, int n);

}

public class Demo07Lambda {

public static void main(String[] args) {

//匿名内部类

Calculator c1 = new Calculator() {

public int add(int m, int n) {

return m + n;

}

};

System.out.println(c1.add(5,4));

//lambda

Calculator c2 = (m, n) -> m+ n;

System.out.println(c2.add(3,5));

}

}

## 基本使用小结

Lambda其实是对一段方法中的行为进行传递，即将方法体做为参数传递。它对匿名内部类的编写方式进行了优化，以前使用匿名内部类写的代码既难看，又冗长。Lambda让这种代码变得更加简洁。

# Lambda使用场景

## 作为方法的参数

* 需求说明

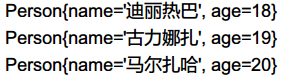
假设有一个Person类，含有String name和int age两个成员变量：

创建对象数组，存储多个Person对象，然后对该数组的Person进行排序：按照年龄从小到大

数据：

new Person("古力娜扎", 19),new Person("迪丽热巴", 18),new Person("马尔扎哈", 20)

* 案例效果：



## 传统写法

如果使用传统的代码对Person[]数组进行排序，写法如下：

1. 创建对象数组
2. 使用匿名内部类创建比较器，并重写方法，按年龄从小到大排序
3. 调用数组类的sort()方法排序
4. 输出排序后的结果

这种做法在面向对象的思想中，似乎也是“理所当然”的。其中Comparator接口的实例（使用了匿名内部类）代表了“按照年龄从小到大”的排序规则。

* 代码分析

下面我们来搞清楚上述代码真正要做什么事情。

* 为了排序，Arrays.sort方法需要排序规则，即Comparator接口的实例，抽象方法compare是关键；
* 为了指定compare的方法体，不得不需要Comparator接口的实现类；
* 为了省去定义一个ComparatorImpl实现类的麻烦，不得不使用匿名内部类；
* 必须覆盖重写抽象compare方法，所以方法名称、方法参数、方法返回值不得不再写一遍，且不能写错；
* 实际上，只有参数和方法体才是关键。

## Lambda写法

import java.util.Arrays;

import java.util.Comparator;

class Person{

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "Person{" +

"name='" + name + '\'' +

", age=" + age +

'}';

}

}

public class Demo08 {

public static void main(String[] args) {

Person[] persons = {

new Person("古力娜扎", 19),

new Person("迪丽热巴", 18),

new Person("马尔扎哈", 20)

};

//匿名类

/\*Arrays.sort(persons, new Comparator<Person>() {

@Override

public int compare(Person o1, Person o2) {

return o1.getAge() - o2.getAge();

}

});\*/

//使用lambda

Arrays.sort(persons,(o1, o2) -> o1.getAge() - o2.getAge());

System.out.println(Arrays.toString(persons));

}

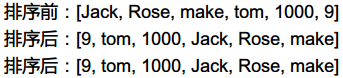
}

## 作为方法的返回值

如果一个方法的返回类型是一个接口，并且接口中只有一个方法，那么就可以使用Lambda表达式实现方法的主体，并且返回这个接口的对象。

如：当使用方法Arrays.sort(数组, 比较器)对数组进行排序的时候，这个方法需要传递一个java.util.Comparator类型的对象作为排序器。我们可以写一个方法返回比较器接口对象，而这个比较器对象可以在方法中使用Lambda表达式得到。

* 案例需求
  1. 创建类Demo10StringLengthComparator ，有如下字符串数组："Jack","Rose","make","tom","1000","9"
  2. 创建方法 private static Comparator<String> stringLengthComparator()，用于返回一个比较器对象。
  3. 使用内部匿名类实现这个比较器，比较规则如下：先比较长度，长度大的比较大。如果长度相同，则按以前的规则比较大小，然后返回比较器对象。
  4. 输出排序前的数组
  5. 调用Arrays.sort()排序方法，调用上面的方法得到一个比较器。
  6. 输出排序后的结果
  7. 使用Lambda表达式实现这个比较器，返回比较器对象。
* 案例效果



* 案例代码

import java.util.Arrays;

import java.util.Comparator;

public class Demo09 {

public static void main(String[] args) {

//修改排序规则，先按长度排序，如果长度相同，再按以前的规则排序

String[] arr = {"Jack","Rose","make","tom","1000","9"};

Arrays.sort(arr, getLengthComparator());

System.out.println("排序后：");

System.out.println(Arrays.toString(arr));

}

//按长度进行比较

public static Comparator<String> getLengthComparator() {

//返回匿名内部类

/\* return new Comparator<String>() {

@Override

public int compare(String o1, String o2) {

return o1.length() - o2.length() == 0? o1.compareTo(o2): o1.length() - o2.length();

}

};\*/

//使用lambda

return (o1, o2) -> o1.length() - o2.length() == 0? o1.compareTo(o2): o1.length() - o2.length();

}

}

## 对集合进行遍历

在Java中使用集合最常见的操作就是进行遍历。现在使用Lambda可以让遍历的代码更加简洁。

注：Consumer接口下次会详细讲

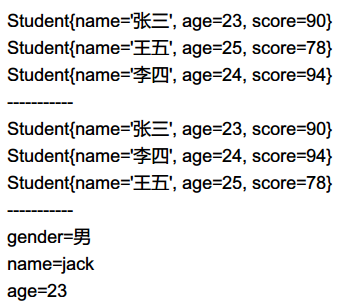
### 遍历的语法

|  |  |
| --- | --- |
| **java.lang.Iterable接口中的方法** | **说明** |
| **void forEach(Consumer action)** | 在JDK8的java.lang.Iterable这个接口中新定义了迭代的方法，用于对单列集合的遍历。Iterable的子接口有List和Set，也就是所有的List和Set中都有这个方法。 |
| 参数：Consumer是操作一个变量的接口，我们可以使用Lambda实现接口中的方法，方法没有返回值。  **public interface Consumer<T> {**  **//对给定的变量进行指定的操作**  **void accept(T t);**  **}** |

|  |  |
| --- | --- |
| **java.util.Map接口中的方法** | **说明** |
| **void forEach(BiConsumer action)** | 在JDK8的java.util.Map接口新增的迭代元素的方法，用于对双列集合进行遍历。 |
| 参数：BiConsumer是操作两个变量的接口，我们可以使用Lambda实现接口中的方法，方法没有返回值。  **public interface BiConsumer<T, U> {**  **//对给定的2个变量进行指定的操作**  **void accept(T t, U u);**  **}** |

### 演示案例

* 案例需求：
  1. 创建List集合，Set集合，Map集合，各添加三个学生对象
  2. 使用Consumer匿名内部类和BiConsumer匿名内部类遍历输出每种集合
  3. 使用Lambda遍历输出每种集合
* 案例效果：



* 案例代码：

import java.util.\*;

public class Demo10 {

public static void main(String[] args) {

List<Integer> nums = Arrays.asList(1,4,26,3,51,7);

/\*for (Integer num : nums) {

System.out.println(num);

}\*/

//使用lambda

nums.forEach(n-> System.out.println(n));

Set<String> sets = new HashSet<>();

sets.add("aaa");

sets.add("cccc");

sets.add("xxx");

sets.add("eeee");

sets.forEach(s-> System.out.println(s));

Map<String, String> map = new HashMap<>();

map.put("a","aaa");

map.put("b","aaa");

map.put("c","aaa");

map.put("d","aaa");

map.forEach((k,v) -> System.out.println(k + "=" + v));

}

}