[java技术 2](#_Toc25972)

[java学习： 3](#_Toc11683)

[java简介 3](#_Toc12944)

[程序基础 4](#_Toc23502)

[数组操作 7](#_Toc1719)

[面向对象编程 8](#_Toc26112)

[面向对象的基本概念 9](#_Toc27722)

[类 9](#_Toc24681)

[实例 9](#_Toc21701)

[方法 9](#_Toc5244)

[构造方法 11](#_Toc4644)

[方法重载(Overload) 11](#_Toc26929)

[面向对象的实现方式 11](#_Toc8367)

[继承 11](#_Toc32307)

[多态(Polymorphic) 13](#_Toc27612)

[Java语言本身提供的机制 19](#_Toc6581)

[Package 19](#_Toc20315)

[作用域 20](#_Toc17863)

[内部类 20](#_Toc23960)

[Classpth和jar 22](#_Toc8535)

[模块（moudle） 23](#_Toc32563)

[Java标准库提供的核心类 24](#_Toc19018)

[字符串 24](#_Toc23349)

[StringBuilder 25](#_Toc12624)

[包装类型 25](#_Toc13850)

[JavaBean 25](#_Toc27998)

[枚举 25](#_Toc9692)

[常用工具类 25](#_Toc5108)

[小技巧： 25](#_Toc11172)

# java技术

* Spring
* SpringMVC
* SpringBoot
* SpringCloud
* MyBatis
* 熟练使用Redis/Mysl/oracle数据库，sql优化水平
* 熟悉git、maven、Linux环境

# 

# java学习：

java简介

学习进阶路线：

1.Java SE，掌握Java语言本身、Java核心开发技术以及Java标准库的使用；

2.Java EE，那么Spring框架、数据库开发、分布式架构就是需要学习的；

-----------------------------------------------------------------

3.大数据开发，那么Hadoop、Spark、Flink这些大数据平台就是需要学习的， 他们都基于Java或Scala开发；

4.移动开发，那么就深入Android平台，掌握Android App开发。

jdk：开发者工具包组成：

JRE就是运行Java字节码的虚拟机；

只有Java源码，要编译成Java字节码，就需要JDK，因为JDK除了包含JRE， 还提供了编译器、调试器等开发工具。



https://jcp.org/en/jsr/detail?id=914

JSR规范：Java Specification Request

保证Java语言的规范性，

SUN公司搞了一个JSR规范，

凡是想给Java平台加一个功能

，比如说访问数据库的功能，

大家要先创建一个JSR规范，

定义好接口，这样，各个数据

库厂商都按照规范写出Java驱

动程序，开发者就不用担心自

己写的数据库代码在MySQL上

能跑，却不能跑在PostgreSQL上。

所以JSR是一系列的规范，从

JVM的内存模型到Web程序接口，

全部都标准化了。

JCP组织：Java Community Process

而负责审核

JSR的组织就是JCP。

一个JSR规范发布时，为了让大

家有个参考，还要同时发布一

个“参考实现”，以及一个“兼容

性测试套件”：

RI：Reference Implementation

TCK：Technology Compatibility Kit

搞一个基于Java开发的消息服务

器，这个提议很好啊，但是光有

提议还不行，得贴出真正能跑的

代码，这就是RI。如果有其他人

也想开发这样一个消息服务器，

如何保证这些消息服务器对开发

者来说接口、功能都是相同的？

所以还得提供TCK。

## 程序基础

java基础：

基本类型的变量和引用类型的变量

基本类型包括整型，浮点型，布尔型，字符型。

变量可重新赋值，等号是赋值语句，不是数学意义的等号。

常量在初始化后不可重新赋值，使用常量便于理解程序意图。

java 10开始支持关键字var，声明变量

整数运算

整数运算的结果永远是精确的；

运算结果会自动提升；

可以强制转型，但超出范围的强制转型会得到错误的结果；

应该选择合适范围的整型（int或long），没有必要为了节省内存而使用byte 和 short进行整数运算。

浮点数运算

浮点数常常无法精确表示，并且浮点数的运算结果可能有误差；

比较两个浮点数通常比较它们的差的绝对值是否小于一个特定值；

整型和浮点型运算时，整型会自动提升为浮点型；

可以将浮点型强制转为整型，但超出范围后将始终返回整型的最大值

布尔运算

与运算和或运算是短路运算；

三元运算b ? x : y后面的类型必须相同，三元运算也是“短路运算”，只计算x 或y。

字符和字符串

Java的字符类型char是基本类型，字符串类型String是引用类型；

基本类型的变量是“持有”某个数值，引用类型的变量是“指向”某个对象；

引用类型的变量可以是空值null；

要区分空值null和空字符串""。

java13开始支持使用多行字符串

数组类型

数组是同一数据类型的集合，数组一旦创建后，大小就不可变；（任意数据类 型都会初始化）

可以通过索引访问数组元素，但索引超出范围将报错；

数组元素可以是值类型（如int）或引用类型（如String），但数组本身是引用 类型；

流程控制

在Java程序中，

JVM默认总是顺序执行以分号;结束的语句。

但是，在实际的代码中，程序经常需要做

条件判断、循环，因此，需要有多种流程

控制语句，来实现程序的跳转和循环等功能

输入和输出

Java提供的输出包括：System.out.println() / print() / printf()，其中printf()可以 格式化输出；

Java提供Scanner对象来方便输入，读取对应的类型可以使用：scanner.nextLine() / nextInt() / nextDouble() / ...

if

if语句从上到下执行时，正确的方式是按照判断范围从大到小依次判断，或者改写成从小到大依次判断：

判断浮点数相等用==判断不靠谱：

正确的方法是利用差值小于某个临界值来判断：

equals()，使用&&注意避免NullPointerException，把不是null的对象放前面

switch多重选择

switch语句可以做多重选择，然后执行匹配的case语句后续代码；

switch的计算结果必须是整型、字符串或枚举类型；

注意千万不要漏写break，建议打开fall-through警告；

总是写上default，建议打开missing default警告；

从Java 7开始，case允许使用字符串。

从Java 12开始，switch语句升级为更简洁的表达式语法，使用类似模式匹配 （Pattern Matching）的方法，保证只有一种路径会被执行，并且不需要break 语句

从Java 14开始，switch语句正式升级为表达式，不再需要break，并且允许使 用yield返回值。

while循环

while循环先判断循环条件是否满足，再执行循环语句；

while循环可能一次都不执行；

编写循环时要注意循环条件，并避免死循环。

do while循环

do while循环先执行循环，再判断条件；

do while循环会至少执行一次。

for循环

for循环通过计数器可以实现复杂循环；

for each循环可以直接遍历数组的每个元素；

最佳实践：计数器变量定义在for循环内部，循环体内部不修改计数器；

break和continue

可设置label不建议使用，当嵌套太多，优化代码

break语句可以跳出当前循环；

break语句通常配合if，在满足条件时提前结束整个循环；

break语句总是跳出最近的一层循环；

continue语句可以提前结束本次循环；

continue语句通常配合if，在满足条件时提前结束本次循环。

### 数组操作

遍历

遍历数组可以使用for循环，for循环可以访问数组索引，for each循环直接迭 代每个数组元素，但无法获取索引；

使用Arrays.toString()可以快速获取数组内容。

排序。

常用的排序算法有冒泡排序、插入排序和快速排序等；

冒泡排序使用两层for循环实现排序；

交换两个变量的值需要借助一个临时变量。

可以直接使用Java标准库提供的Arrays.sort()进行排序；

对数组排序会直接修改数组本身。

多维数组

二维数组就是数组的数组，三维数组就是二维数组的数组；

多维数组的每个数组元素长度都不要求相同；

打印多维数组可以使用Arrays.deepToString()；

最常见的多维数组是二维数组，访问二维数组的一个元素使用array[row][col]。

命令行参数

命令行参数类型是String[]数组；

命令行参数由JVM接收用户输入并传给main方法；

如何解析命令行参数需要由程序自己实现。

由于引入了，package com.wxz.learn;

Main.class的完全限定名是“ com.wxz.learn.Main”。

在src下执行java命令+空格+Main的完全限定名

## 面向对象编程

java是一种面向对象的编程语言。面向对象编程，英文是Object-Oriented Programming，简称OOP。

那什么是面向对象编程？

和面向对象编程不同的，是面向过程编程。面向过程编程，是把模型分解成一步一 步的过程。比如，老板告诉你，要编写一个TODO任务，必须按照以下步骤一步 一步来：

1. 读取文件；
2. 编写TODO；
3. 保存文件。

面向对象编程，是一种通过对象的方式，把现实世界映射到计算机模型的一种编程方法。

### 面向对象的基本概念

#### 类

定义class就是定义了一种数据类型，对应的instance是这种数据类型的实例；

class定义的field，在每个instance都会拥有各自的field，且互不干扰；

#### 实例

通过new操作符创建新的instance，然后用变量指向它，即可通过变量来引用这个instance；

访问实例字段的方法是变量名.字段名；

指向instance的变量都是引用变量。

 一个Java源文件可以包含多个类的定义，但只能定义一个public类，且public类名必须与文件名一致。如果要定义多个public类，必须拆到多个Java源文件中。

#### 方法

一个类通过定义方法，就可以给外部代码暴露一些操作的接口，同时，内部自己保证逻辑一致性。

调用方法的语法是实例变量.方法名(参数);。一个方法调用就是一个语句，所以不要忘了在末尾加;。例如：ming.setName("Xiao Ming");。

定义方法

从上面的代码可以看出，定义方法的语法是：

修饰符 方法返回类型 方法名(方法参数列表) {

若干方法语句;

return 方法返回值;

}

方法返回值通过return语句实现，如果没有返回值，返回类型设置为void，可以省略return。

this变量

一个隐含的变量this，它始终指向当前实例。因此，通过this.field就可以访问当前实例的字段

有局部变量和字段重名，那么局部变量优先级更高，就必须加上this：

方法参数

方法可以包含0个或任意个参数。方法参数用于接收传递给方法的变量值。调用方法时，必须严格按照参数的定义一一传递。

可变参数

可变参数用类型...定义，可变参数相当于数组类型：

lass Group {

private String[] names;

public void setNames(String... names) {

this.names = names;

}

}

上面的setNames()就定义了一个可变参数。调用时，可以这么写：

Group g = new Group();

g.setNames("Xiao Ming", "Xiao Hong", "Xiao Jun"); // 传入3个String

g.setNames("Xiao Ming", "Xiao Hong"); // 传入2个String

g.setNames("Xiao Ming"); // 传入1个String

g.setNames(); // 传入0个String

完全可以把可变参数改写为String[]类型：

class Group {

private String[] names;

public void setNames(String[] names) {

this.names = names;

}

}

但是，调用方需要自己先构造String[]，比较麻烦。例如：

Group g = new Group();

g.setNames(new String[] {"Xiao Ming", "Xiao Hong", "Xiao Jun"}); // 传入1个String[]

另一个问题是，调用方可以传入null：

Group g = new Group();

g.setNames(null);

而可变参数可以保证无法传入null，因为传入0个参数时，接收到的实际值是一个空数组而不是null。

参数绑定

调用方把参数传递给实例方法时，调用时传递的值会按参数位置一一绑定。

* 基本类型参数的传递，是调用方值的复制。双方各自的后续修改，互不影响.
* 引用类型参数的传递，调用方的变量，和接收方的参数变量，指向的是同一个对象。双方任意一方对这个对象的修改，都会影响对方（因为指向同一个对象)

#### 构造方法

* 实例在创建时通过new操作符会调用其对应的构造方法，构造方法用于初始化实例；
* 没有定义构造方法时，编译器会自动创建一个默认的无参数构造方法；
* 可以定义多个构造方法，编译器根据参数自动判断；
* 可以在一个构造方法内部调用另一个构造方法，便于代码复用。

#### 方法重载(Overload)

* 方法重载是指多个方法的方法名相同，但各自的参数不同；
* 重载方法应该完成类似的功能，参考String的indexOf()；
* 重载方法返回值类型应该相同。

### 面向对象的实现方式

#### 继承

* 继承是面向对象编程的一种强大的代码复用方式；
* 继承树

Java只允许单继承，所有类最终的根类是Object；

* Protected

子类无法访问父类的private字段或者private方法

protected允许子类访问父类的字段和方法；

* Super

子类的构造方法可以通过super()调用父类的构造方法；

如果父类没有默认的构造方法，子类就必须显式调用super()并给出参数以便让编译器定位到父类的一个合适的构造方法。

子类不会继承任何父类的构造方法。子类默认的构造方法是编译器自动生成的，不是继承的。

* 阻止继承final关键字

可以安全地向上转型为更抽象的类型；

class没有final修饰符，那么任何类都可以从该class继承

Java 15开始，允许使用sealed修饰class，并通过permits明确写出能够从该class继承的子类名称。

public sealed class Shape permits Rect, Circle, Triangle {

...

}

这种sealed类主要用于一些框架，防止继承被滥用。

sealed类在Java 15中目前是预览状态，要启用它，必须使用参数--enable-preview和--source 15

* 向上转型

Student s = new Student();

Person p = new Person();

Person p = new Student();

Student继承自Person

一个子类类型安全地变为父类类型的赋值，被称为向上转型（upcasting）

* 向下转型

一个父类类型强制转型为子类类型，就是向下转型（downcasting）

Person p1 = new Student(); // upcasting, ok

Person p2 = new Person();

Student s1 = (Student) p1; // ok

Student s2 = (Student) p2; // runtime error! ClassCastException!

可以强制向下转型，最好借助instanceof判断；

Person p = new Student();

if (p instanceof Student) {

// 只有判断成功才会向下转型:

Student s = (Student) p; // 一定会成功

}

* 区分继承和组合

子类和父类的关系是is-a，has关系不能用继承。

Student是Person的一种，它们是is关系，而Student并不是Book。实际上Student和Book的关系是has关系。

#### 多态(Polymorphic)

* 方法重写Override

子类可以覆写父类的方法（Override），覆写在子类中改变了父类方法的行为；

Override和Overload不同的是，如果方法签名不同，就是Overload，Overload方法是一个新方法；如果方法签名相同，并且返回值也相同

加上@Override可以让编译器帮助检查是否进行了正确的覆写。希望进行覆写，但是不小心写错了方法签名，编译器会报错。

* 多态与动态调用

Java的实例方法调用是基于运行时的实际类型的动态调用，而非变量的声明类型。

Java的方法调用总是作用于运行期对象的实际类型，这种行为称为多态；

例如：

Person p = new Student();

p.run(); // 无法确定运行时究竟调用哪个run()方法

public void runTwice(Person p) {

p.run();

p.run();

}

它传入的参数类型是Person，我们是无法知道传入的参数实际类型究竟是Person，还是Student，还是Person的其他子类，因此，也无法确定调用的是不是Person类定义的run()方法。

多态具有一个非常强大的功能，就是允许添加更多类型的子类实现功能扩展，却不需要修改基于父类的代码。

* 覆写Object方法

所有的class最终都继承自Object，而Object定义了几个重要的方法：

toString()：把instance输出为String；

equals()：判断两个instance是否逻辑相等；

hashCode()：计算一个instance的哈希值。

在必要的情况下，我们可以覆写Object的这几个方法。例如：

class Person {

...

// 显示更有意义的字符串:

@Override

public String toString() {

return "Person:name=" + name;

}

// 比较是否相等:

@Override

public boolean equals(Object o) {

// 当且仅当o为Person类型:

if (o instanceof Person) {

Person p = (Person) o;

// 并且name字段相同时，返回true:

return this.name.equals(p.name);

}

return false;

}

// 计算hash:

@Override

public int hashCode() {

return this.name.hashCode();

}

}

* 调用super

在子类的覆写方法中，如果要调用父类的被覆写的方法，可以通过 super来调用。例如：

class Person {

protected String name;

public String hello() {

return "Hello, " + name;

}

}

Student extends Person {

@Override

public String hello() {

// 调用父类的hello()方法:

return super.hello() + "!";

}

}

* final修饰符有多种作用：

final修饰的方法可以阻止被覆写；

class Person {

protected String name;

public final String hello() {

return "Hello, " + name;

}

}

Student extends Person {

// compile error: 不允许覆写

@Override

public String hello() {

}

}

final修饰的class可以阻止被继承；

final class Person {

protected String name;

}

// compile error: 不允许继承自Person

Student extends Person {

}

final修饰的field必须在创建对象时初始化，随后不可修改。

class Person {

public final String name = "Unamed";

}

对final字段重新赋值会报错：

Person p = new Person();

p.name = "New Name"; // compile error!

可以在构造方法中初始化final字段：

class Person {

public final String name;

public Person(String name) {

this.name = name;

}

}

这种方法更为常用，因为可以保证实例一旦创建，其final字段就不 可修改

* 抽象类

一个class定义了方法，但没有具体执行代码，这个方法就是抽象方法，抽象方法用abstract修饰。

因为无法执行抽象方法，因此这个类也必须申明为抽象类（abstract class）。

使用abstract修饰的类就是抽象类。我们无法实例化一个抽象类：

Person p = new Person(); // 编译错误

无法实例化的抽象类

面向抽象编程的本质就是：

上层代码只定义规范（例如：abstract class Person）；

抽象类本身被设计成只能用于被继承，因此，抽象类可以强迫子类实现其定义的抽象方法，否则编译会报错。因此，抽象方法实际上相当于定义了“规范”。

不需要子类就可以实现业务逻辑（正常编译）；

通过抽象类Person类型去引用具体的子类的实例：

Person s = new Student();

Person t = new Teacher();

这种引用抽象类的好处在于，我们对其进行方法调用，并不关心 Person类型变量的具体子类型：

// 不关心Person变量的具体子类型:

s.run();

t.run();

具体的业务逻辑由不同的子类实现，调用者并不关心。

* 接口

Java的接口（interface）定义了纯抽象规范，一个类可以实现多个接 口；

interface，就是比抽象类还要抽象的纯抽象接口，因为它连字段 都不能有。因为接口定义的所有方法默认都是public abstract的， 所以这两个修饰符不需要写出来

interface，就是比抽象类还要抽象的纯抽象接口，因为它连字段 都不能有。因为接口定义的所有方法默认都是public abstract的， 所以这两个修饰符不需要写出来，java提供了接口来实现类似多 继承的功能

* 术语

注意区分术语：

Java的接口特指interface的定义，表示一个接口类型和一组方法 签名，而编程接口泛指接口规范，如方法签名，数据格式，网络 协议等。

接口也是数据类型，适用于向上转型和向下转型；

接口的所有方法都是抽象方法，接口不能定义实例字段；

接口可以定义default方法（JDK>=1.8）。

抽象类和接口的对比如下：

|  | **abstract class** | **interface** |
| --- | --- | --- |
| 继承 | 只能extends一个class | 可以implements多个interface |
| 字段 | 可以定义实例字段 | 不能定义实例字段 |
| 抽象方法 | 可以定义抽象方法 | 可以定义抽象方法 |
| 非抽象方法 | 可以定义非抽象方法 | 可以定义default方法 |

接口继承

接口之间可以继承。interface继承自 interface使用extends，它 相当于扩展了接口的方法。例如：

interface Hello {

void hello();

}

interface Person extends Hello {

void run();

String getName();

}

此时，Person接口继承自Hello接口，因此，Person接口现在实 际上有3个抽象方法签名，其中一个来自继承的Hello接口。

继承关系

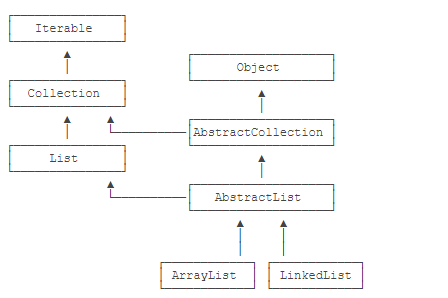
合理设计interface和abstract class的继承关系，可以充分复用代 码。

一般来说，公共逻辑适合放在abstract class中，

具体逻辑放到各个子类，

而接口层次代表抽象程度。

可以参考Java的集合类定义的一组接口、抽象类以及具体子类 的继承关系：



实例化的对象永远只能是某个具体的子类，但总是通过接口去引 用它，因为接口比抽象类更抽象：

List list = new ArrayList(); // 用List接口引用具体子类的实例

Collection coll = list; // 向上转型为Collection接口

Iterable it = coll; // 向上转型为Iterable接口

* default方法

在接口中，可以定义default方法。例如，把Person接口的run()方法改为default方法：

* 静态字段和静态方法

静态字段属于所有实例“共享”的字段，实际上是属于class的 字段；

调用静态方法不需要实例，无法访问this，但可以访问静态字段 和其他静态方法；

静态方法常用于工具类和辅助方法。

interface的字段只能是public static final类型，所以我们可以把 这些修饰符都去掉，上述代码可以简写为：

public interface Person {

// 编译器会自动加上public statc final:

int MALE = 1;

int FEMALE = 2;

}

编译器会自动把该字段变为public static final类型。

### Java语言本身提供的机制

#### Package

Java内建的package机制是为了避免class命名冲突；

在Java虚拟机执行的时候，JVM只看完整类名，因此，只要包名不同， 类就不同。

包可以是多层结构，用.隔开

假设以package\_sample作为根目录，src作为源码目录，那么所有文件结构就是：

package\_sample

└─ src

├─ hong

│ └─ Person.java

│ ming

│ └─ Person.java

└─ mr

└─ jun

└─ Arrays.java

即所有Java文件对应的目录层次要和包的层次一致。

编译后的.class文件也需要按照包结构存放。如果使用IDE，把编译后的.class文件放到bin目录下，那么，编译的文件结构就是：

package\_sample

└─ bin

├─ hong

│ └─ Person.class

│ ming

│ └─ Person.class

└─ mr

└─ jun

└─ Arrays.class

编译的命令相对比较复杂，我们需要在src目录下执行javac命令：

javac -d ../bin ming/Person.java hong/Person.java mr/jun/Arrays.java

JDK的核心类使用java.lang包，编译器会自动导入；

包作用域

位于同一个包的类，可以访问包作用域的字段和方法。不用public、 protected、private修饰的字段和方法就是包作用域。

JDK的其它常用类定义在java.util.\*，java.math.\*，java.text.\*，……；

包名推荐使用倒置的域名，例如org.apache

#### 作用域

Java内建的访问权限包括public、protected、private和package权限；

Java在方法内部定义的变量是局部变量，局部变量的作用域从变量声明开始，到一个块结束；

final修饰符不是访问权限，它可以修饰class、field和method；

用final修饰method可以阻止被子类覆写：

用final修饰method可以阻止被子类覆写：

用final修饰field可以阻止被重新赋值：

用final修饰局部变量可以阻止被重新赋值：

package abc;

public class Hello {

protected void hi(final int t) {

t = 1; // error!

}

}

一个.java文件只能包含一个public类，但可以包含多个非public类。

#### 内部类

Java的内部类可分为Inner Class、Anonymous Class和Static Nested Class 三种：

Inner Class

如果一个类定义在另一个类的内部，这个类就是Inner Class：

class Outer {

class Inner {

// 定义了一个Inner Class

}

}

Outer.Inner inner = outer.new Inner();

Java编译器编译后的.class文件可以发现，Outer类被编译为Outer.class， 而Inner类被编译为Outer$Inner.class

Inner Class的作用域在Outer Class内部，所以能访问Outer Class的private字段和方法。

Anonymous Class

还有一种定义Inner Class的方法，它不需要在Outer Class中明确地定义这个Class，而是在方法内部，通过匿名类（Anonymous Class）来定义。示例代码如红色

public class Main {

public static void main(String[] args) {

Outer outer = new Outer("Nested");

outer.asyncHello();

}

}

class Outer {

private String name;

Outer(String name) {

this.name = name;

}

void asyncHello() {

**Runnable r = new Runnable() {**

**@Override**

**public void run() {**

**System.out.println("Hello, " + Outer.this.name);**

**}**

**};**

**new Thread(r).start();**

}

}

//实例化方法

Outer outer = new Outer("Nested");

outer.asyncHello();

Java编译器编译后的.class文件可以发现，Outer类被编译为Outer.class， 而匿名类被编译为Outer$1.class。如果有多个匿名类，Java编译器会将每 个匿名类依次命名为Outer$1、Outer$2、Outer$3…

Inner Class和Anonymous Class本质上是相同的，都必须依附于Outer Class 的实例，即隐含地持有Outer.this实例，并拥有Outer Class的private访问 权限；

静态内部类（Static Nested Class）

Static Nested Class是独立类，但拥有Outer Class的private访问权限。

public class Main {

public static void main(String[] args) {

Outer.StaticNested sn = new Outer.StaticNested();

sn.hello();

}

}

class Outer {

private static String NAME = "OUTER";

private String name;

Outer(String name) {

this.name = name;

}

static class StaticNested {

void hello() {

System.out.println("Hello, " + Outer.NAME);

}

}

}

#### Classpth和jar

JVM通过环境变量classpath决定搜索class的路径和顺序；

不推荐设置系统环境变量classpath，始终建议通过-cp命令传入；

jar包相当于目录，可以包含很多.class文件，方便下载和使用；

MANIFEST.MF文件可以提供jar包的信息，如Main-Class，这样可以直接运行jar包。

Java是编译型语言，源码文件是.java，而编译后的.class文件才是真正可以被JVM执行的字节码。因此，JVM需要知道，如果要加载一个abc.xyz.Hello的类，应该去哪搜索对应的Hello.class文件。

classpath就是一组目录的集合，它设置的搜索路径与操作系统相关。例如，在Windows系统上，用;分隔，带空格的目录用""括起来，可能长这样：

C:\work\project1\bin;C:\shared;"D:\My Documents\project1\bin"

在Linux系统上，用:分隔，可能长这样：

/usr/shared:/usr/local/bin:/home/liaoxuefeng/bin

现在我们假设classpath是.;C:\work\project1\bin;C:\shared，当JVM在加载abc.xyz.Hello这个类时，会依次查找：

<当前目录>\abc\xyz\Hello.class

C:\work\project1\bin\abc\xyz\Hello.class

C:\shared\abc\xyz\Hello.class

注意到.代表当前目录。如果JVM在某个路径下找到了对应的class文件，就不再往后继续搜索。如果所有路径下都没有找到，就报错。

classpath的设定方法有两种：

在系统环境变量中设置classpath环境变量，不推荐；

在启动JVM时设置classpath变量，推荐。

我们强烈不推荐在系统环境变量中设置classpath，那样会污染整个系统环境。在启动JVM时设置classpath才是推荐的做法。实际上就是给java命令传入-classpath或-cp参数：

java -classpath .;C:\work\project1\bin;C:\shared abc.xyz.Hello

或者使用-cp的简写：

java -cp .;C:\work\project1\bin;C:\shared abc.xyz.Hello

没有设置系统环境变量，也没有传入-cp参数，那么JVM默认的classpath为.，即当前目录：

java abc.xyz.Hello

上述命令告诉JVM只在当前目录搜索Hello.class。

在IDE中运行Java程序，IDE自动传入的-cp参数是当前工程的bin目录和引入的jar包。

#### 模块（moudle）

Java 9引入的模块目的是为了管理依赖；

rt.jar分拆成了几十个模块，这些模块以.jmod扩展名标识，可以在$JAVA\_HOME/jmods目录下找到它们：

这些.jmod文件每一个都是一个模块，模块名就是文件名

使用模块可以按需打包JRE；

1. 项目路径下，编译所有java源文件

javac -d bin src/module-info.java src/com/itranswarp/sample/\*.java

2.打包的时候传入--main-class参数指定主方法

jar --create --file hello.jar --main-class com.itranswarp.sample.Main -C bin .

3.此时得到了一个普通的jar文件，利用jmod命令把一个jar包转换成模块

jmod create --class-path hello.jar hello.jmod

获得一个同名的hello.jmod文件，这就是模块文件

4.再将该模块，使用jlink命令，进行裁剪jmod。将程序中无用的jre删掉。只带上 用到的模块，打包jre

Jlink --module-path hello.jmod --add-modules java.base,java.xml,hello.world --output jre/

5.运行jre

java --module hello.world

使用模块对类的访问权限有了进一步限制。

编写的模块hello.world用到了模块java.xml的一个类javax.xml.XMLConstants，之所以能直接使用这个类，是因为模块java.xml的module-info.java中声明了若干导出：

module java.xml {

exports java.xml;

exports javax.xml.catalog;

exports javax.xml.datatype;

...

}

只有它声明的导出的包，外部代码才被允许访问。换句话说，如果外部代码想要访问我们的hello.world模块中的com.itranswarp.sample.Greeting类，我们必须将其导出：

module hello.world {

exports com.itranswarp.sample;

requires java.base;

requires java.xml;

}

### Java标准库提供的核心类

#### 字符串

Java字符串String是不可变对象；

字符串操作不改变原字符串内容，而是返回新字符串；

常用的字符串操作：提取子串、查找、替换、大小写转换等；

Java使用Unicode编码表示String和char；

转换编码就是将String和byte[]转换，需要指定编码；

转换为byte[]时，始终优先考虑UTF-8编码。

#### StringBuilder

StringBuilder是可变对象，用来高效拼接字符串；

StringBuilder可以支持链式操作，实现链式操作的关键是返回实例本身；

StringBuffer是StringBuilder的线程安全版本，现在很少使用。

#### StringJoiner

**public** **final** **class** StringJoiner {

**private** **final** String prefix;

**private** **final** String delimiter;

**private** **final** String suffix;

**private** String[] elts;

**private** **int** size;

**private** **int** len;

**private** String emptyValue;

}

用指定分隔符拼接字符串数组时，使用StringJoiner或者String.join()更方便；

用StringJoiner拼接字符串时，还可以额外附加一个“开头”和“结尾”。

#### 包装类型

Java核心库提供的包装类型可以把基本类型包装为class；

自动装箱和自动拆箱都是在编译期完成的（JDK>=1.5）；

装箱和拆箱会影响执行效率，且拆箱时可能发生NullPointerException；

包装类型的比较必须使用equals()；

整数和浮点数的包装类型都继承自Number；

包装类型提供了大量实用方法。

#### JavaBean

在Java中，有很多class的定义都符合这样的规范：

若干private实例字段；

通过public方法来读写实例字段。

这种类就可以称之为javaBean

**public** **class Person {**

**private** String name;

**private** **int** age;

**public** String getName() { **return** **this**.name; }

**public** **void** setName(String name) { **this**.name = name; }

**public** **int** getAge() { **return** **this**.age; }

**public** **void** setAge(**int** age) { **this**.age = age; }

}

通常把一组对应的读方法（getter）和写方法（setter）称为属性（property）。例如，name属性：

* 对应的读方法是String getName()
* 对应的写方法是setName(String)

JavaBean的作用

JavaBean主要用来传递数据，即把一组数据组合成一个JavaBean便于传输。此外，JavaBean可以方便地被IDE工具分析，生成读写属性的代码，主要用在图形界面的可视化设计中。

枚举JavaBean属性

要枚举一个JavaBean的所有属性，可以直接使用Java核心库提供的Introspector

#### 枚举

Java使用enum定义枚举类型，它被编译器编译为final class Xxx extends Enum { … }；

通过name()获取常量定义的字符串，注意不要使用toString()；

通过ordinal()返回常量定义的顺序（无实质意义）；

可以为enum编写构造方法、字段和方法

enum的构造方法要声明为private，字段强烈建议声明为final；

enum适合用在switch语句中。

在Java中，我们可以通过static final来定义常量。例如，我们希望定义周一到周日这7个常量，可以用7个不同的int表示：

**public** **class Weekday {**

**public** **static** **final** **int** SUN = 0;

**public** **static** **final** **int** MON = 1;

**public** **static** **final** **int** TUE = 2;

**public** **static** **final** **int** WED = 3;

**public** **static** **final** **int** THU = 4;

**public** **static** **final** **int** FRI = 5;

**public** **static** **final** **int** SAT = 6;

}

使用常量的时候，可以这么引用：

**if** (day == Weekday.SAT || day == Weekday.SUN) {

// TODO: work at home

}

也可以把常量定义为字符串类型，例如，定义3种颜色的常量：

**public** **class Color {**

**public** **static** **final** String RED = "r";

**public** **static** **final** String GREEN = "g";

**public** **static** **final** String BLUE = "b";

}

使用常量的时候，可以这么引用：

String color = ...**if** (Color.RED.equals(color)) {

// TODO:

}

无论是int常量还是String常量，使用这些常量来表示一组枚举值的时候，有一个严重的问题就是，编译器无法检查每个值的合理性。例如：

**if** (weekday == 6 || weekday == 7) {

**if** (tasks == Weekday.MON) {

// TODO:

}

}

上述代码编译和运行均不会报错，但存在两个问题：

* 注意到Weekday定义的常量范围是0~6，并不包含7，编译器无法检查不在枚举中的int值；
* 定义的常量仍可与其他变量比较，但其用途并非是枚举星期值。

### enum

为了让编译器能自动检查某个值在枚举的集合内，并且，不同用途的枚举需要不同的类型来标记，不能混用，我们可以使用enum来定义枚举类：

窗体顶端

// enum

 Run

窗体底端

注意到定义枚举类是通过关键字enum实现的，我们只需依次列出枚举的常量名。

和int定义的常量相比，使用enum定义枚举有如下好处：

首先，enum常量本身带有类型信息，即Weekday.SUN类型是Weekday，编译器会自动检查出类型错误。例如，下面的语句不可能编译通过：

int day = 1;**if** (day == Weekday.SUN) { // Compile error: bad operand types **for** binary operator '=='

}

其次，不可能引用到非枚举的值，因为无法通过编译。

最后，不同类型的枚举不能互相比较或者赋值，因为类型不符。例如，不能给一个Weekday枚举类型的变量赋值为Color枚举类型的值：

Weekday x = Weekday.SUN; // ok!Weekday y = Color.RED; // Compile error: incompatible types

这就使得编译器可以在编译期自动检查出所有可能的潜在错误。

### enum的比较

使用enum定义的枚举类是一种引用类型。前面我们讲到，引用类型比较，要使用equals()方法，如果使用==比较，它比较的是两个引用类型的变量是否是同一个对象。因此，引用类型比较，要始终使用equals()方法，但enum类型可以例外。

这是因为enum类型的每个常量在JVM中只有一个唯一实例，所以可以直接用==比较：

**if** (day == Weekday.FRI) { // ok!

}**if** (day.equals(Weekday.SUN)) { // ok, but more code!

}

### enum类型

通过enum定义的枚举类，和其他的class有什么区别？

答案是没有任何区别。enum定义的类型就是class，只不过它有以下几个特点：

* 定义的enum类型总是继承自java.lang.Enum，且无法被继承；
* 只能定义出enum的实例，而无法通过new操作符创建enum的实例；
* 定义的每个实例都是引用类型的唯一实例；
* 可以将enum类型用于switch语句。

例如，我们定义的Color枚举类：

**public** **enum** Color {

RED, GREEN, BLUE;

}

编译器编译出的class大概就像这样：

**public** **final** **class Color extends Enum {** *// 继承自Enum，标记为final class*

*// 每个实例均为全局唯一:*

**public** **static** **final** Color RED = **new** Color();

**public** **static** **final** Color GREEN = **new** Color();

**public** **static** **final** Color BLUE = **new** Color();

*// private构造方法，确保外部无法调用new操作符:*

**private** Color() {}

}

所以，编译后的enum类和普通class并没有任何区别。但是我们自己无法按定义普通class那样来定义enum，必须使用enum关键字，这是Java语法规定的。

因为enum是一个class，每个枚举的值都是class实例，因此，这些实例有一些方法：

#### name()

返回常量名，例如：

String s = Weekday.SUN.name(); // "SUN"

#### ordinal()

返回定义的常量的顺序，从0开始计数，例如：

**int** n = Weekday.MON.ordinal(); *// 1*

改变枚举常量定义的顺序就会导致ordinal()返回值发生变化。例如：

**public** **enum** Weekday {

SUN, MON, TUE, WED, THU, FRI, SAT;

}

和

**public** **enum** Weekday {

MON, TUE, WED, THU, FRI, SAT, SUN;

}

的ordinal就是不同的。如果在代码中编写了类似if(x.ordinal()==1)这样的语句，就要保证enum的枚举顺序不能变。新增的常量必须放在最后。

有些童鞋会想，Weekday的枚举常量如果要和int转换，使用ordinal()不是非常方便？比如这样写：

String task = Weekday.MON.ordinal() + "/ppt";

saveToFile(task);

但是，如果不小心修改了枚举的顺序，编译器是无法检查出这种逻辑错误的。要编写健壮的代码，就不要依靠ordinal()的返回值。因为enum本身是class，所以我们可以定义private的构造方法，并且，给每个枚举常量添加字段：

窗体顶端

// enum

 Run

Work at home!

窗体底端

这样就无需担心顺序的变化，新增枚举常量时，也需要指定一个int值。

 注意：枚举类的字段也可以是非final类型，即可以在运行期修改，但是不推荐这样做！

默认情况下，对枚举常量调用toString()会返回和name()一样的字符串。但是，toString()可以被覆写，而name()则不行。我们可以给Weekday添加toString()方法：

窗体顶端

// enum

 Run

Today is 星期日. Work at home!

窗体底端

覆写toString()的目的是在输出时更有可读性。

 注意：判断枚举常量的名字，要始终使用name()方法，绝不能调用toString()！

### switch

最后，枚举类可以应用在switch语句中。因为枚举类天生具有类型信息和有限个枚举常量，所以比int、String类型更适合用在switch语句中：

窗体顶端

// switch

 Run

Today is SUN. Work at home!

窗体底端

加上default语句，可以在漏写某个枚举常量时自动报错，从而及时发现错误。

### 小结

Java使用enum定义枚举类型，它被编译器编译为final class Xxx extends Enum { … }；

通过name()获取常量定义的字符串，注意不要使用toString()；

通过ordinal()返回常量定义的顺序（无实质意义）；

可以为enum编写构造方法、字段和方法

enum的构造方法要声明为private，字段强烈建议声明为final；

enum适合用在switch语句中。

### [记录类](https://www.liaoxuefeng.com/wiki/1252599548343744/1331429187256353)

从Java 14开始，提供新的record关键字，可以非常方便地定义Data Class：

**public** **record** Point2(**int** x, **int** y) {}

使用record定义的是不变类；

可以编写Compact Constructor对参数进行验证；

**public** Point2 {

**if** (x < 0 || y < 0) {

**throw** **new** IllegalArgumentException();

}

}

可以定义静态方法of来创建一个默认对象。

**public** **static** Point2 of() {

**return** **new** Point2(0, 0);

}

### [BigInteger](https://www.liaoxuefeng.com/wiki/1252599548343744/1279767986831393)

BigInteger用于表示任意大小的整数；

BigInteger big = **new** BigInteger("1234567890");

BigInteger是不变类，并且继承自Number；

**public** **class** BigInteger **extends** Number **implements** Comparable<BigInteger> { **final** **int** signum;**final** **int**[] mag;...}

将BigInteger转换成基本类型时可使用longValueExact()等方法保证结果准确。BigInteger i1 = **new** BigInteger("1234567890");

BigInteger i2 = **new** BigInteger("12345678901234567890");

BigInteger sum = i1.add(i2);

BigInteger i = sum.multiply(i).longValueExact();//超long出范围抛出ArithmeticException算术异常

### [BigDecimal](https://www.liaoxuefeng.com/wiki/1252599548343744/1279768011997217)

BigDecimal用于表示精确的小数，常用于财务计算；

BigDecimal 是不变类，并且继承自Number，intVal表示整数，scale表示小数位数；

**public** **class** BigDecimal **extends** Number **implements** Comparable<BigDecimal> { **private** **final** BigInteger intVal; **private** **final** **int** scale;...}

比较BigDecimal的值是否相等，必须使用compareTo()而不能使用equals()。

### [常用工具类](https://www.liaoxuefeng.com/wiki/1252599548343744/1260473555087392)

#### Math:数学计算public final class Math {...}

提供了大量静态数学运算方法和数学常量e,pi

#### Random:生成伪随机数

**public** **class** Random **implements** java.io.Serializable {...}

Random用来创建伪随机数。所谓伪随机数，是指只要给定一个初始的种子，产生的随机数序列是完全一样的

生成一个随机数，可以使用nextInt()、nextLong()、nextFloat()、nextDouble()：

该构造指定系统时间作为种子

**public** Random() {

**this**(*seedUniquifier*() ^ System.*nanoTime*());

}

#### SecureRandom:生成安全的随机数

**public** **class** SecureRandom **extends** java.util.Random {

// Seed Generator

**private** **static** **volatile** SecureRandom *seedGenerator*;

...}

继承了Radom工具类，扩展

SecureRandom无法指定种子，它使用RNG（random number generator）算法。JDK的SecureRandom实际上有多种不同的底层实现，有的使用安全随机种子加上伪随机数算法来产生安全的随机数，有的使用真正的随机数生成器。实际使用的时候，可以优先获取高强度的安全随机数生成器，如果没有提供，再使用普通等级的安全随机数生成器：

SecureRandom的安全性是通过操作系统提供的安全的随机种子来生成随机数。这个种子是通过CPU的热噪声、读写磁盘的字节、网络流量等各种随机事件产生的“熵”。

在密码学中，安全的随机数非常重要。如果使用不安全的伪随机数，所有加密体系都将被攻破。因此，时刻牢记必须使用SecureRandom来产生安全的随机数。

## [异常处理](https://www.liaoxuefeng.com/wiki/1252599548343744/1255943543190176)

程序运行的时候，经常会发生各种错误。

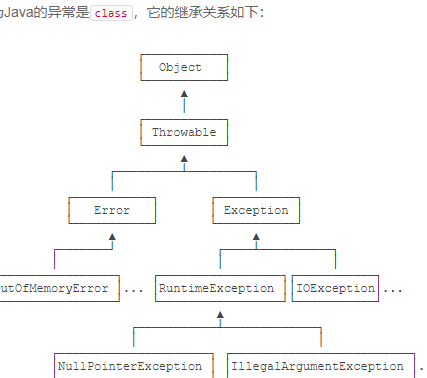
比如，使用Excel的时候，它有时候会报错：



本章我们讨论如何在Java程序中处理各种异常情况。

### [Java的异常](https://www.liaoxuefeng.com/wiki/1252599548343744/1264734349295520)

Java的异常是class，并且从Throwable继承；



**public** **class** Throwable **implements** Serializable {

@java.io.Serial

**private** **static** **final** **long** ***serialVersionUID*** = -3042686055658047285L;

...}

↓

**public** **class** Exception **extends** Throwable {

@java.io.Serial

**static** **final** **long** ***serialVersionUID*** = -3387516993124229948L;

...}

↓

**public** **class** RuntimeException **extends** Exception {

@java.io.Serial

**static** **final** **long** ***serialVersionUID*** = -7034897190745766939L;

...}

↓

**public** **class** SecurityException **extends** RuntimeException {

@java.io.Serial

**private** **static** **final** **long** ***serialVersionUID*** = 6878364983674394167L;

...}

**public** **class** IOException **extends** Exception {

@java.io.Serial

**static** **final** **long** ***serialVersionUID*** = 7818375828146090155L;

...}

↓

**public** **class** FileNotFoundException **extends** IOException {

@java.io.Serial

**private** **static** **final** **long** ***serialVersionUID*** = -897856973823710492L;

...}

Java规定：

必须捕获的异常，包括Exception及其子类，但不包括RuntimeException及其子类，这种类型的异常称为Checked Exception。

Java使用异常来表示错误，并通过try ... catch捕获异常；

RuntimeException无需强制捕获，非RuntimeException（Checked Exception）需强制捕获，或者用throws声明；

不推荐捕获了异常但不进行任何处理。

不需要捕获的异常，包括Error及其子类，RuntimeException及其子类。

Error是无需捕获的严重错误，Exception是应该捕获的可处理的错误；

OutOfMemoryError：内存耗尽

NoClassDefFoundError：无法加载某个Class

StackOverflowError：栈溢出

### [捕获异常](https://www.liaoxuefeng.com/wiki/1252599548343744/1264738442933472)

使用try ... catch ... finally时：

多个catch语句的匹配顺序非常重要，子类必须放在前面；

多个catch语句只有一个能被执行

finally语句保证了有无异常都会执行，它是可选的；

直接使用finally语句来,消除重复代码,finally语句块保证有无错误都会执行

一个catch语句也可以匹配多个非继承关系的异常。

### [抛出异常](https://www.liaoxuefeng.com/wiki/1252599548343744/1264738764506656)

异常的传播

调用printStackTrace()可以打印异常的传播栈，对于调试非常有用；

抛出异常

捕获异常并再次抛出新的异常时，应该持有原始异常信息；

(NullPointerException e) {

// 所以为了能定位,案发现场,将e传入抛出的异常中

**throw** **new** IllegalArgumentException(e);

获取原始异常可以使用Throwable.getCause()方法

catch中抛出异常，不会影响finally的执行。JVM会先执行finally，然后抛出异常

异常屏蔽

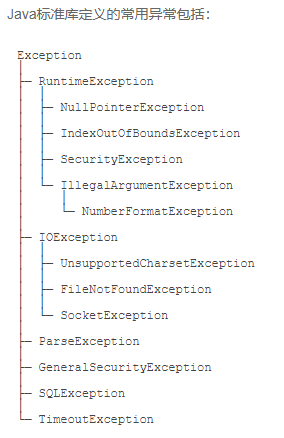
finall抛出异常,catch执行,catch中异常不会抛出,被屏蔽

因为只能抛出一个异常

绝大部分情况下不要在finally中抛出异常。如果在finally中抛出异常， 应该原始异常加入到原有异常中。调用方可通过 Throwable.getSuppressed()获取 所有添加的Suppressed Exception。

如果用final要抛出所有异常，务必将所有异常添加到原始异常，最后打 印异常方法栈

### [自定义异常](https://www.liaoxuefeng.com/wiki/1252599548343744/1264737765214592)



抛出异常时，尽量复用JDK已定义的异常类型；

自定义异常体系时，推荐从RuntimeException派生“根异常”，再派生出业务异常；

自定义异常时，应该提供多种构造方法。

### [NullPointerException](https://www.liaoxuefeng.com/wiki/1252599548343744/1337645544243233)

NullPointerException是Java代码常见的逻辑错误，应当早暴露，早修复；

使用空字符串""而不是默认的null可避免很多NullPointerException，编写业务逻辑时，用空字符串""表示未填写比null安全得多。

返回空字符串""、空数组而不是null：

避免空指针,最好的方式还是做好非空校验

可以启用Java 14的增强异常信息来查看NullPointerException的详细错误信息。

### [使用断言](https://www.liaoxuefeng.com/wiki/1252599548343744/1264740093521088)

断言（Assertion）是一种调试程序的方式。在Java中，使用assert关键字来实现断言

**public** **static** **void** main(String[] args) {

**double** x = Math.abs(-123.45);

**assert** x >= 0;

System.out.println(x);

}语句assert x >= 0;即为断言，断言条件x >= 0预期为true。如果计算结果为false，则断言失败，抛出AssertionError。

Java断言的特点是：断言失败时会抛出AssertionError，导致程序结束退出。因此，断言不能用于可恢复的程序错误，只应该用于开发和测试阶段。

JVM默认关闭断言指令，

要执行assert语句，必须给Java虚拟机传递-enableassertions（可简写为-ea）参数启用断言。

java -ea Main.javaException **in** thread "main" java.lang.AssertionError

at Main.main(Main.java:5)

**断言是一种调试方式，断言失败会抛出AssertionError，只能在开发和测试阶段启用断言；**

**对可恢复的错误不能使用断言，而应该抛出异常；**

**断言很少被使用，更好的方法是编写单元测试。**

### [使用JDK Logging](https://www.liaoxuefeng.com/wiki/1252599548343744/1264738568571776)

输出日志，而不是用System.out.println()，有以下几个好处：

1. 可以设置输出样式，避免自己每次都写"ERROR: " + var；
2. 可以设置输出级别，禁止某些级别输出。例如，只输出错误日志；
3. 可以被重定向到文件，这样可以在程序运行结束后查看日志；
4. 可以按包名控制日志级别，只输出某些包打的日志；
5. 可以…

Java标准库内置了日志包java.util.logging

日志的输出可以设定级别。JDK的Logging定义了7个日志级别，从严重到普通：

SEVERE、WARNING、INFO、CONFIG、FINE、FINER、FINEST

默认是INFO级别，低于INFO不会打印

Logging系统在JVM启动时读取配置文件并完成初始化，一旦开始运行main()方法，就无法修改配置；

配置不太方便，需要在JVM启动时传递参数-Djava.util.logging.config.file=<config-file-name>。

因此，Java标准库内置的Logging使用并不是非常广泛。更方便的日志系统我们稍后介绍。

**日志是为了替代System.out.println()，可以定义格式，重定向到文件等；**

**日志可以存档，便于追踪问题；**

**日志记录可以按级别分类，便于打开或关闭某些级别；**

**可以根据配置文件调整日志，无需修改代码；**

**Java标准库提供了java.util.logging来实现日志功能**

### [使用Commons Logging](https://www.liaoxuefeng.com/wiki/1252599548343744/1264738932870688)

“日志接口”

Commons Logging是使用最广泛的日志模块；

org.apache.commons.logging.LogFactory;

Commons Logging是一个第三方日志库，它是由Apache创建的日志模块。

可以挂接不同的日志系统，并通过配置文件指定挂接的日志系统。

默认情况下，Commons Loggin自动搜索并使用Log4j（Log4j是另一个流行的日志系统），如果没有找到Log4j，再使用JDK Logging。

Commons Logging的API非常简单；

Commons Logging可以自动检测并使用其他日志模块。

Commons Logging定义了6个日志级别：

FATAL/ERROR/WARNING/INFO/DEBUG/TRACE/默认级别是INFO。

### [使用Log4j](https://www.liaoxuefeng.com/wiki/1252599548343744/1264739436350112)

真正的“日志实现”可以使用Log4j

通过Commons Logging实现日志，不需要修改代码即可使用Log4j；

使用Log4j只需要把log4j2.xml和相关jar放入classpath；

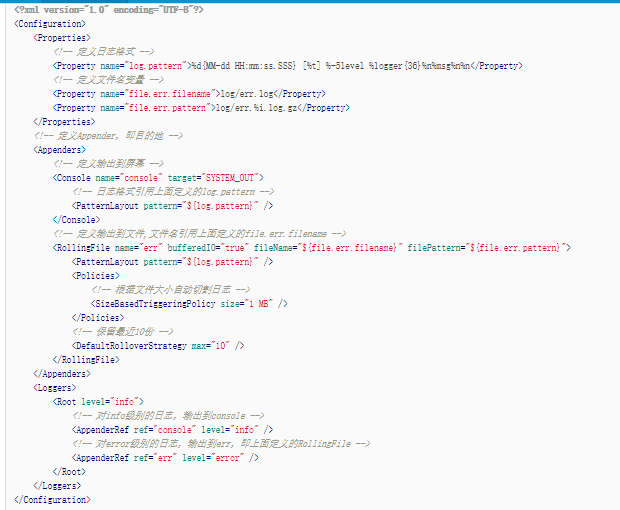
如果要更换Log4j，只需要移除log4j2.xml和相关jar；

只有扩展Log4j时，才需要引用Log4j的接口（例如，将日志加密写入数据库的功能，需要自己开发）。



在输出日志的过程中，通过Filter来过滤哪些log需要被输出，哪些log不需要被输出。例如，仅输出ERROR级别的日志。

最后，通过Layout来格式化日志信息，例如，自动添加日期、时间、方法名称等信息

以XML配置为例，使用Log4j的时候，我们把一个log4j2.xml的文件放到classpath下就可以让Log4j读取配置文件并按照我们的配置来输出日志

### [使用SLF4J和Logback](https://www.liaoxuefeng.com/wiki/1252599548343744/1264739155914176)

SLF4J和Logback可以取代Commons Logging和Log4j；

始终使用SLF4J的接口写入日志，使用Logback只需要配置，不需要修改代码。

和Log4j类似，我们仍然需要一个Logback的配置文件，把logback.xml放到classpath下，配置如下：

**<?xml version="1.0" encoding="UTF-8"?>**<configuration>

<appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

<charset>utf-8</charset>

</encoder>

<file>log/output.log</file>

<rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">

<fileNamePattern>log/output.log.%i</fileNamePattern>

</rollingPolicy>

<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">

<MaxFileSize>1MB</MaxFileSize>

</triggeringPolicy>

</appender>

<root level="INFO">

<appender-ref ref="CONSOLE" />

<appender-ref ref="FILE" />

</root></configuration>

运行即可获得类似如下的输出：

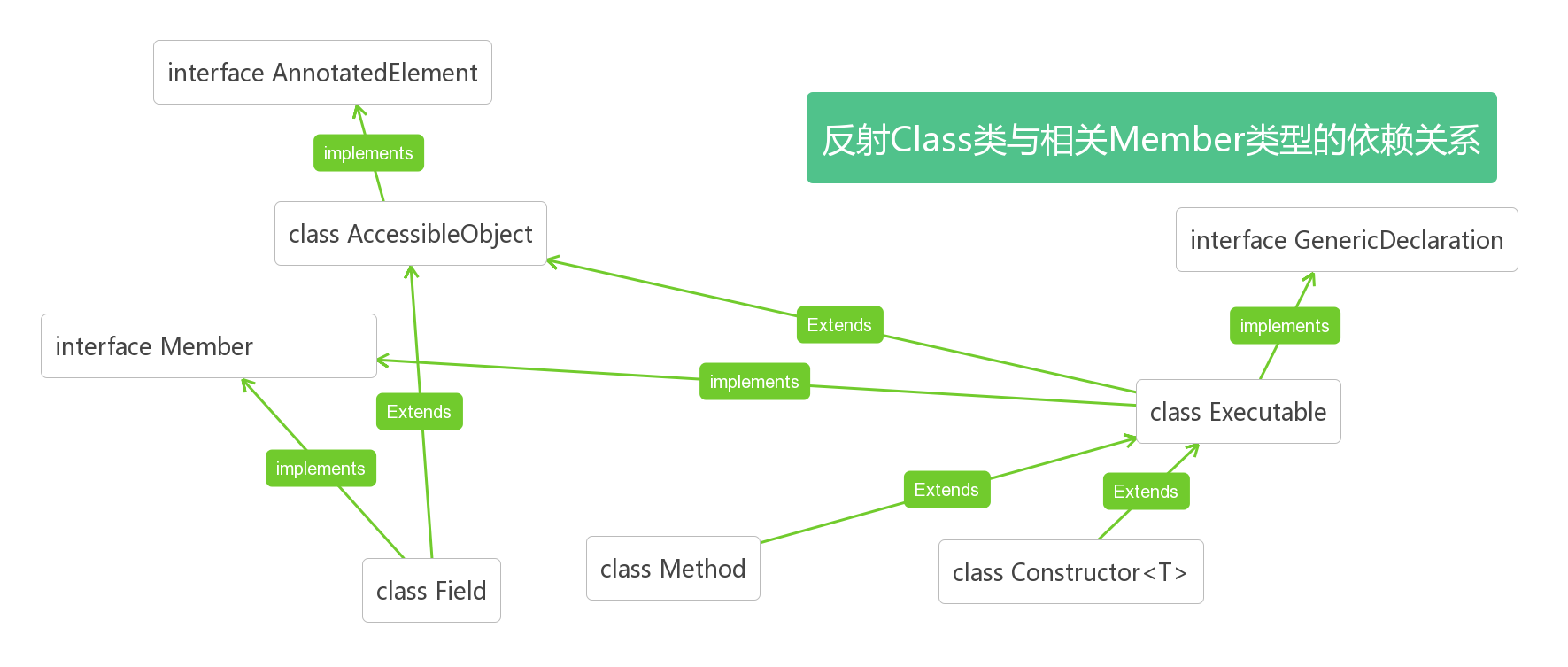
13:15:25**.328** [main] INFO com**.itranswarp.learnjava.Main** - Start process...

## [反射](https://www.liaoxuefeng.com/wiki/1252599548343744/1255945147512512)

什么是反射？

反射就是Reflection**，Java的反射是指程序在运行期**可以**拿到一个一无所知对象的所有信息**。

调用一个对象的方法，或者访问一个对象的字段，通常会传入对象实例



### [Class类](https://www.liaoxuefeng.com/wiki/1252599548343744/1264799402020448)

JVM为每个加载的class及interface创建了对应的Class实例来保存class及interface的所有信息；

获取一个class对应的Class实例后，就可以获取该class的所有信息；

通过Class实例获取class信息的方法称为反射（Reflection）；

三种获取class实例的方法。

类名.class通过一个class的静态变量class

某个实例.getClass() **final** **native**;

Class.forName(完全限定名);

**JVM总是动态加载class，**可以**在运行期根据条件来控制加载class。**

不是一次性把所有用到的class全部加载到内存，而是第一次需要用到class时才加载。

### [访问字段](https://www.liaoxuefeng.com/wiki/1252599548343744/1264803033837024)Field

* java的反射API提供的Field类封装了字段的所有信息：

Class类提供了以下几个方法来获取字段：

Field getField(name)：根据字段名获取某个public的field（包括父类）

Field getDeclaredField(name)：根据字段名获取当前类的某个field（不包括父类）

Field[] getFields()：获取所有public的field（包括父类）

Field[] getDeclaredFields()：获取当前类的所有field（不包括父类）

* 通过Class实例的方法可以获取Field实例：getField()，getFields()，getDeclaredField()，getDeclaredFields()；
* 通过Field实例可以获取字段信息：getName()，getType()，getModifiers()；
* 通过Field实例可以读取或设置某个对象的字段，如果存在访问限制，要首先调用setAccessible(true)来访问非public字段。
* 通过反射读写字段是一种非常规方法，它会破坏对象的封装。

### [调用方法](https://www.liaoxuefeng.com/wiki/1252599548343744/1264803678201760)**Method**

* Java的反射API提供的Method对象封装了方法的所有信息：

**Method getMethod(name, Class...)：获取某个public的Method（包括父类）**

**Method getDeclaredMethod(name, Class...)：获取当前类的某个Method（不包括父类）**

**Method[] getMethods()：获取所有public的Method（包括父类）**

**Method[] getDeclaredMethods()：获取当前类的所有Method（不包括父类）**

* 通过Class实例的方法可以获取Method实例：getMethod()，getMethods()，getDeclaredMethod()，getDeclaredMethods()；
* 通过Method实例可以获取方法信息：getName()，getReturnType()，getParameterTypes()，getModifiers()；
* 通过Method实例可以调用某个对象的方法：Object invoke(Object instance, Object... parameters)；
* 通过设置setAccessible(true)来访问非public方法；
* 某个SecurityManager可能不允许对java和javax开头的package的类调用setAccessible(true)，这样可以保证JVM核心库的安全
* 通过反射调用方法时，仍然遵循多态原则。
* 使用反射调用方法时，仍然遵循多态原则：即总是调用实际类型的覆写方法（如果存在）

### [调用构造方法](https://www.liaoxuefeng.com/wiki/1252599548343744/1264802263123712)Constructor

Constructor对象封装了构造方法的所有信息；

通过Class实例的方法可以获取Constructor实例：getConstructor()，getConstructors()，getDeclaredConstructor()，getDeclaredConstructors()；

通过Constructor实例可以创建一个实例对象：newInstance(Object... parameters)； 通过设置setAccessible(true)来访问非public构造方法。

### [获取继承关系](https://www.liaoxuefeng.com/wiki/1252599548343744/1264804244564000)Class getSuperclass()

通过Class对象可以获取继承关系：

Class getSuperclass()：获取父类类型；

Class[] getInterfaces()：获取当前类实现的所有接口。

通过Class对象的isAssignableFrom()方法可以判断一个向上转型是否可以实现。

### [动态代理](https://www.liaoxuefeng.com/wiki/1252599548343744/1264804593397984)Proxy

**public** **class** Proxy **implements** java.io.Serializable {...}

Java标准库提供了动态代理功能，允许在运行期动态创建一个接口的实例；

在运行期动态创建一个interface实例的方法如下：

定义一个InvocationHandler实例，它负责实现接口的方法调用；

通过Proxy.newProxyInstance()创建interface实例，它需要3个参数：

使用的ClassLoader，通常就是接口类的ClassLoader；

需要实现的接口数组，至少需要传入一个接口进去；

用来处理接口方法调用的InvocationHandler实例。

将返回的Object强制转型为接口。

动态代理是通过Proxy创建代理对象，然后将接口方法“代理”给InvocationHandler完成的。

## [注解](https://www.liaoxuefeng.com/wiki/1252599548343744/1255945389098144)

介绍Java程序的一种特殊“注释”——注解（Annotation）

### [使用注解](https://www.liaoxuefeng.com/wiki/1252599548343744/1265102413966176)

注解（Annotation）是Java语言用于工具处理的标注：

注解可以配置参数，没有指定配置的参数使用默认值；

如果参数名称是value，且只有一个参数，那么可以省略参数名称。

### [定义注解](https://www.liaoxuefeng.com/wiki/1252599548343744/1265102803921888)

Java使用@interface定义注解：

可定义多个参数和默认值，核心参数使用value名称；

必须设置@Target来指定Annotation可以应用的范围；

应当设置@Retention(RetentionPolicy.RUNTIME)便于运行期读取该Annotation。

### [处理注解](https://www.liaoxuefeng.com/wiki/1252599548343744/1265102026065728)

在运行期通过反射读取RUNTIME类型的注解，注意千万不要漏写@Retention(RetentionPolicy.RUNTIME)，否则运行期无法读取到该注解。

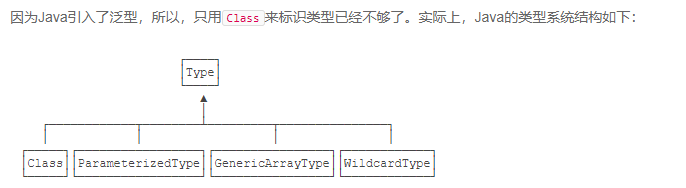
可以通过程序处理注解来实现相应的功能：

对JavaBean的属性值按规则进行检查；

JUnit会自动运行@Test标记的测试方法。

## [泛型](https://www.liaoxuefeng.com/wiki/1252599548343744/1255945193293888)

泛型是一种“代码模板”，可以用一套代码套用各种类型。



### [什么是泛型](https://www.liaoxuefeng.com/wiki/1252599548343744/1265102638843296)

用于解决可以创建任何类型的Arraylist，<E> E可以是任意类型

**public** **class** ArrayList<E> **extends** AbstractList<E>

**implements** List<E>, RandomAccess, Cloneable, java.io.Serializable

{

/\*\*

\* Shared empty array instance used for empty instances.

\*/

际上ArrayList内部就是一个Object[]数组，配合存储一个当前分配的长度，就可以充当“可变数组”

**private** **static** **final** Object[] ***EMPTY\_ELEMENTDATA*** = {};

/\*\*

\* Default initial capacity.

\*/

**private** **static** **final** **int** ***DEFAULT\_CAPACITY*** = 10;

**private** **static** **final** Object[] ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*** = {};

private int size;

public void add(Object e) {...}

public void remove(int index) {...}

public Object get(int index) {...}

}

泛型就是编写模板代码来适应任意类型；

泛型的好处是使用时不必对类型进行强制转换，**它通过编译器对类型进行检查；**

注意泛型的继承关系：可以把ArrayList<Integer>向上转型为List<Integer>（T不能变！），但不能把ArrayList<Integer>向上转型为ArrayList<Number>（T不能变成父类）。

### [使用泛型](https://www.liaoxuefeng.com/wiki/1252599548343744/1265103567584000)

使用泛型时，把泛型参数<T>替换为需要的class类型，例如：ArrayList<String>，ArrayList<Number>等；

可以省略编译器能自动推断出的类型，例如：List<String> list = new ArrayList<>();；

不指定泛型参数类型时，编译器会给出警告，且只能将<T>视为Object类型；

可以在接口中定义泛型类型，实现此接口的类必须实现正确的泛型类型。

### [编写泛型](https://www.liaoxuefeng.com/wiki/1252599548343744/1265105853480864)

编写泛型时，需要定义泛型类型<T>；

静态方法不能引用泛型类型<T>，必须定义其他类型（例如<K>）来实现静态泛型方法；

泛型可以同时定义多种类型，例如Map<K, V>。

### [擦拭法](https://www.liaoxuefeng.com/wiki/1252599548343744/1265104600263968)

<T>

Java的泛型是采用擦拭法实现的；

擦拭法决定了泛型<T>：

不能是基本类型，例如：int；

不能获取带泛型类型的Class，例如：Pair<String>.class；

不能判断带泛型类型的类型，例如：x instanceof Pair<String>；

不能实例化T类型，例如：new T()。

泛型方法要防止重复定义方法，例如：public boolean equals(T obj)；

子类可以获取父类的泛型类型<T>。

### [extends通配符](https://www.liaoxuefeng.com/wiki/1252599548343744/1265105899616928)

使用类似<? extends Number>通配符作为方法参数时表示：

方法内部可以调用获取Number引用的方法，例如：Number n = obj.getFirst();；

方法内部无法调用传入Number引用的方法（null除外），例如：obj.setFirst(Number n);。

即一句话总结：使用extends通配符表示可以读，不能写。

使用类似<T extends Number>定义泛型类时表示：

泛型类型限定为Number以及Number的子类。

### [super通配符](https://www.liaoxuefeng.com/wiki/1252599548343744/1265105920586976)

使用类似<? super Integer>通配符作为方法参数时表示：

方法内部可以调用传入Integer引用的方法，例如：obj.setFirst(Integer n);；

方法内部无法调用获取Integer引用的方法（Object除外），例如：Integer n = obj.getFirst();。

即使用super通配符表示只能写不能读。

使用extends和super通配符要遵循PECS原则。

无限定通配符<?>很少使用，可以用<T>替换，同时它是所有<T>类型的超类。

### [泛型和反射](https://www.liaoxuefeng.com/wiki/1252599548343744/1265105940850016)

部分反射API是泛型，例如：Class<T>，Constructor<T>；

可以声明带泛型的数组，但不能直接创建带泛型的数组，必须强制转型；

可以通过Array.newInstance(Class<T>, int)创建T[]数组，需要强制转型；

同时使用泛型和可变参数时需要特别小心。

## [集合](https://www.liaoxuefeng.com/wiki/1252599548343744/1255943629175808)

### [Java集合简介](https://www.liaoxuefeng.com/wiki/1252599548343744/1265109905179456)

Java的集合类定义在java.util包中，支持泛型，主要提供了3种集合类，包括 List一种有序列表的集合，例如，按索引排列的Student的List，

Set一种保证没有重复元素的集合，例如，所有无重复名称的Student的Set,

Map一种通过键值（key-value）查找的映射表集合，例如，根据Student的name查找对应Student的Map

。Java集合使用统一的Iterator遍历，尽量不要使用遗留接口。

数组初始化后大小不可变；

数组只能按索引顺序存取。

--

采用Collection

可变大小的顺序链表；

保证无重复元素的集合；

Java集合的设计有几个特点：一是实现了接口和实现类相分离

### [使用List](https://www.liaoxuefeng.com/wiki/1252599548343744/1265112034799552)

List是按索引顺序访问的长度可变的有序表，优先使用ArrayList而不是LinkedList；

可以直接使用for each遍历List；

List可以和Array相互转换。

ArrayList把添加和删除的操作封装起来，让我们操作List类似于操作数组，却不用关心内部元素如何移动。

LinkedList通过“链表”也实现了List接口。在LinkedList中，它的内部每个元素都指向下一个元素：

Arraylist，内存占用小，删除是内部元素移动，添加从最后元素开始，速度快

Linkedlist，内存占用大，删除添加，不需要移动元素，查找元素从头开始

LIST的特点：

关注List的接口规范：List允许添加重复元素，允许添加null。

List的创建：Arrlist，linkedList，List接口of方法，指定元素不接受null

便利List

for循环便利，list集合：不同list遍历耗时不同，不建议使用

使用迭代器Iterator来遍历list元素：Iterator对象，被List对象实例调用iterator（）方法来创建。Iterator对象有两个方法,boolean hasNext()判断是否有下一个元素， E next()返回下一个元素

迭代器是最高效的方式

List和Array转换

List接口提供了toArray（）方法直接返回一个Object数组；

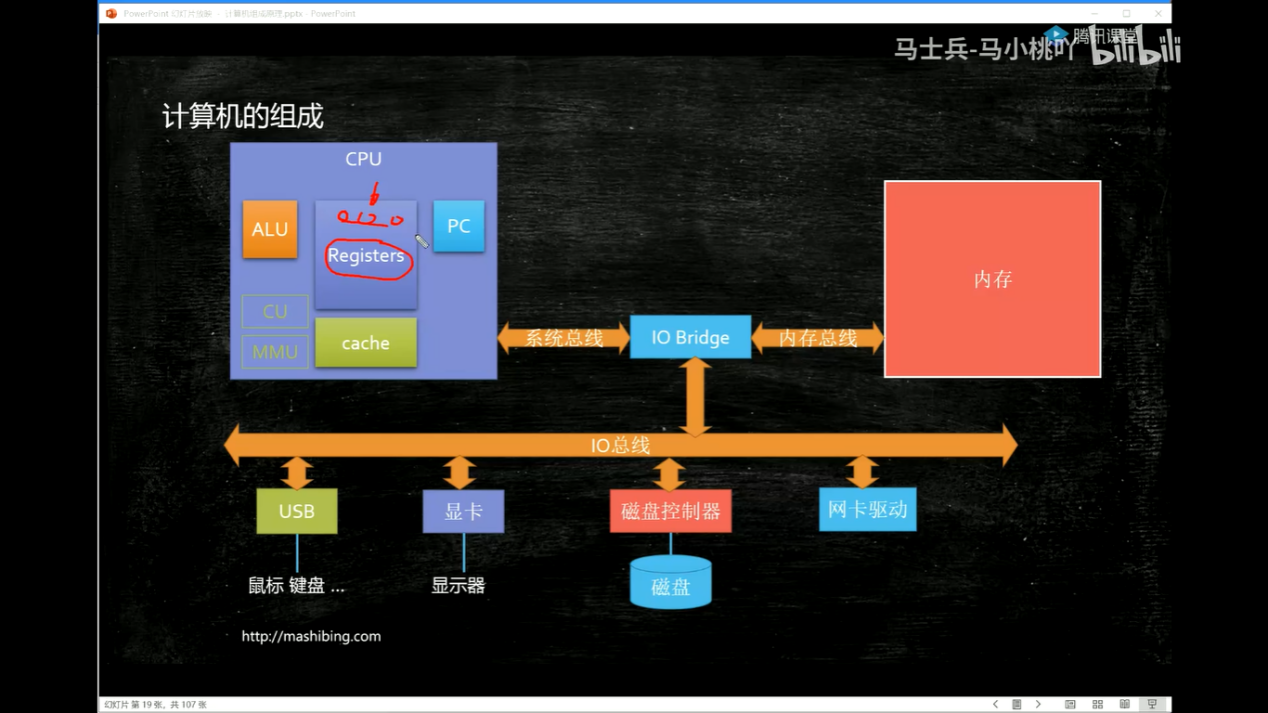
会丢失类型信息

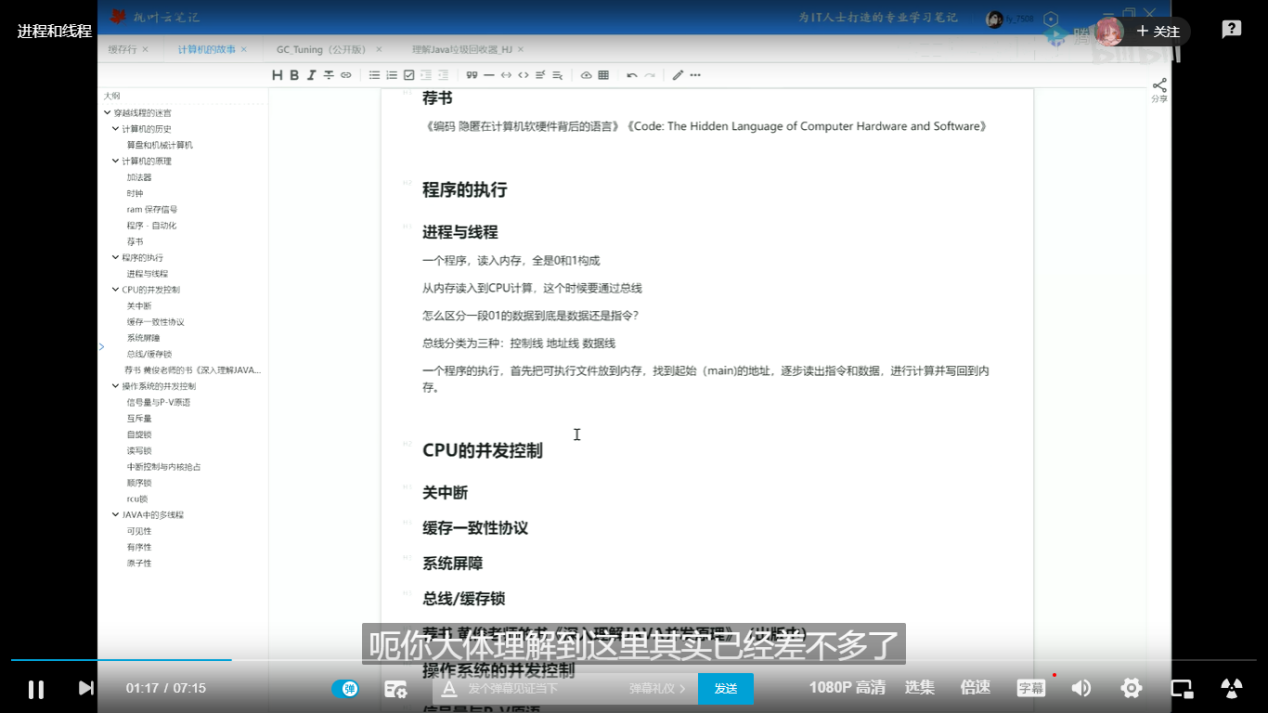
toArray(T[])传入一个类型相同的Array，List内部自动把元素复制到传入的Array中：

T不是泛型<>中指定的类型，只要是相同类型即可。传入数组类型不匹配，会报ArrayStoreException异常；数组copy的过程中类型不匹配

List接口定义的T[] toArray(IntFunction<T[]> generator)方法：

Integer[] **array** = **list**.toArray(Integer[]::**new**);





把Array变为List就简单多了，通过List.of(T...)方法最简单：

JDK 11之前的版本，可以使用Arrays.asList(T...)方法把数组转换成List。

Integer[] **array** = { 1, 2, 3 };**List**<Integer> **list** = **List**.of(**array**);

### [编写equals方法](https://www.liaoxuefeng.com/wiki/1252599548343744/1265116446975264)

在List中查找元素时，List的实现类通过元素的equals()方法比较两个元素是否相等，因此，放入的元素必须正确覆写equals()方法，Java标准库提供的String、Integer等已经覆写了equals()方法；

编写equals()方法可借助Objects.equals()判断。

如果不在List中查找元素，就不必覆写equals()方法。

### [使用Map](https://www.liaoxuefeng.com/wiki/1252599548343744/1265118019954528)

Map是一种映射表，可以通过key快速查找value。

Map也是一个接口，最常用的实现类是HashMap

可以通过for each遍历keySet()，也可以通过for each遍历entrySet()，直接获取key-value。

最常用的一种Map实现是HashMap。

### [编写equals和hashCode](https://www.liaoxuefeng.com/wiki/1252599548343744/1265117217944672)

要正确使用HashMap，作为key的类必须正确覆写equals()和hashCode()方法；

当value存储的是引用类，且该类如果覆写了equals()，就必须覆写hashCode()，并且覆写规则是：

如果equals()返回true，则hashCode()返回值必须相等；

如果equals()返回false，则hashCode()返回值尽量不要相等。

实现hashCode()方法可以通过Objects.hashCode()辅助方法实现。

### [使用EnumMap](https://www.liaoxuefeng.com/wiki/1252599548343744/1300277678309410)

如果Map的key是enum类型，推荐使用EnumMap，既保证速度，也不浪费空间。

使用EnumMap的时候，根据面向抽象编程的原则，应持有Map接口。

### [使用TreeMap](https://www.liaoxuefeng.com/wiki/1252599548343744/1265117109276544)

SortedMap在遍历时严格按照Key的顺序遍历，最常用的实现类是TreeMap；

作为SortedMap的Key必须实现Comparable接口，或者传入Comparator；

要严格按照compare()规范实现比较逻辑，否则，TreeMap将不能正常工作。

### [使用Properties](https://www.liaoxuefeng.com/wiki/1252599548343744/1265119084411136)

Java集合库提供的Properties用于读写配置文件.properties。.properties文件可以使用UTF-8编码。

可以从文件系统、classpath或其他任何地方读取.properties文件。

**读取配置文件：**

用Properties读取配置文件，一共有三步：

1. 创建Properties实例；
2. 调用load()读取文件；
3. 调用getProperty()获取配置。

**写入配置文件：**

Properties props = **new** Properties();

props.setProperty("url", "http://www.liaoxuefeng.com");

props.setProperty("language", "Java");

props.store(**new** FileOutputStream("C:\\conf\\setting.properties"), "这是写入的properties注释");

**编码：**

早期版本的Java规定.properties文件编码是ASCII编码（ISO8859-1），如果涉及到中文就必须用name=\u4e2d\u6587来表示，非常别扭。从JDK9开始，Java的.properties文件可以使用UTF-8编码了。

Properties props = **new** Properties();

props.load(**new** FileReader("settings.properties", StandardCharsets.UTF\_8));

读写Properties时，注意仅使用getProperty()和setProperty()方法，不要调用继承而来的get()和put()等方法。

### [使用Set](https://www.liaoxuefeng.com/wiki/1252599548343744/1265121225603904)

Set用于存储不重复的元素集合：

Set接口常用集合类hashSet

而hashSet仅仅是对hashMap的一个简单封装

Set接口并不保证有序，而SortedSet接口则保证元素是有序的

SortedSet接口是Set接口的实现

* 放入HashSet的元素与作为HashMap（实现了Set接口）的key要求相同；
* 放入TreeSet的元素与作为TreeMap(实现了SortedSet接口)的Key要求相同；

利用Set可以去除重复元素；

遍历SortedSet按照元素的排序顺序遍历，也可以自定义排序算法。

### [使用Queue](https://www.liaoxuefeng.com/wiki/1252599548343744/1265121791832960)

队列Queue实现了一个先进先出（FIFO）的数据结构：

是接口，

是一类常用集合。First in First out的有序表。区别于List的是，只有两个操作，把元素添加到队列末尾，从队列头部取出元素。

* 通过add()/offer()方法将元素添加到队尾；
* 通过remove()/poll()从队首获取元素并删除；
* 通过element()/peek()从队首获取元素但不删除。

要避免把null添加到队列。

### [使用PriorityQueue](https://www.liaoxuefeng.com/wiki/1252599548343744/1265120632401152)

PriorityQueue实现了一个优先队列：从队首获取元素时，总是获取优先级最高的元素。

PriorityQueue默认按元素比较的顺序排序（必须实现Comparable接口），也可以通过Comparator自定义排序算法（元素就不必实现Comparable接口）。

### [使用Deque](https://www.liaoxuefeng.com/wiki/1252599548343744/1265122668445536)

Deque实现了一个双端队列（Double Ended Queue），它可以：



* 将元素添加到队尾或队首：addLast()/offerLast()/addFirst()/offerFirst()；
* 从队首／队尾获取元素并删除：removeFirst()/pollFirst()/removeLast()/pollLast()；
* 从队首／队尾获取元素但不删除：getFirst()/peekFirst()/getLast()/peekLast()；
* 总是调用xxxFirst()/xxxLast()以便与Queue的方法区分开；
* 避免把null添加到队列。

### [使用Stack](https://www.liaoxuefeng.com/wiki/1252599548343744/1265121668997888)

栈（Stack）是一种后进先出（LIFO）的数据结构，操作栈的元素的方法有：

* 把元素压栈：push(E)；
* 把栈顶的元素“弹出”：pop(E)；
* 取栈顶元素但不弹出：peek(E)。

在Java中，我们用Deque可以实现Stack的功能，注意只调用push()/pop()/peek()方法，避免调用Deque的其他方法。

最后，不要使用遗留类Stack。

Stack的作用

1.Stack在计算机中使用非常广泛，JVM在处理Java方法调用的时候就会通过栈这种数据结构维护方法调用的层次。例如：

**static** **void** main(String[] args) {

foo(123);

}

**static** String foo(int x) {

**return** "F-" + bar(x + 1);

}

**static** **int** bar(**int** x) {

**return** x << 2;

}

当调用方法超出stack数据结构限制，则会报栈溢出，StackOverFlowErro（无限递归复现该问题）

1. 计算进制转换问题：获取一个空栈，将十进制除以对应进制获取余数和商。每次取的余数压栈，商继续取余压栈，当商是0的时候。依次弹栈，从栈尾依次取出的字符串即为十进制的对应转换进制对应的表示字符串。
2. 计算中缀表达式：运算符在中间即为中缀表达式。Eq,1+2\*5/6.计算机首先通过编译器编译表达式，替换为后缀表达式。Eq,1 2 5 6 \* / +

而计算机在编译的时候会用到栈，编译比较复杂，只看栈计算表达式的过程:

后缀表达式不考虑优先级，从左到右以此计算。

首先准备空栈，然后依次扫描后缀表达式1 2 5 6 \* / +，遇到数字直接压栈。

遇到算术符号的时候，弹栈，依次弹出两个栈顶的元素，并计算，并将结果压栈。

再遇到算术符号，依次弹出两个栈顶元素，运算，将结果压栈。

...

扫描结束后，弹出栈的唯一一个元素。

### [使用Iterator](https://www.liaoxuefeng.com/wiki/1252599548343744/1265124784468736)

Iterator是一种抽象的数据访问模型。使用Iterator模式进行迭代的好处有：

* 对任何集合都采用同一种访问模型；
* 调用者对集合内部结构一无所知；
* 集合类返回的Iterator对象知道如何迭代。

Java提供了标准的迭代器模型，即集合类实现java.util.Iterable接口，返回java.util.Iterator实例。

### [使用Collections](https://www.liaoxuefeng.com/wiki/1252599548343744/1299919855943714)

Collections类提供了一组工具方法来方便使用集合类：

* 创建空集合；
* 创建单元素集合；
* 创建不可变集合；
* 排序／洗牌等操作。

## [IO](https://www.liaoxuefeng.com/wiki/1252599548343744/1255945227202752)

标准IO， 基于字节流和字符流进行操作，阻塞IO。

IO流是一种流式的数据输入/输出模型：

二进制数据以byte为最小单位在InputStream/OutputStream中单向流动；

字符数据以char为最小单位在Reader/Writer中单向流动。

Java标准库的java.io包提供了同步IO功能：

字节流接口：InputStream/OutputStream；

字符流接口：Reader/Writer。

### [File对象](https://www.liaoxuefeng.com/wiki/1252599548343744/1298069154955297)

Java标准库的java.io.File对象表示一个文件或者目录：

创建File对象本身不涉及IO操作；

可以获取路径／

绝对路径／

规范路径：getPath()/getAbsolutePath()/getCanonicalPath()；

可以获取目录的文件和子目录：list()/listFiles()；

可以创建或删除文件和目录。

### [InputStream](https://www.liaoxuefeng.com/wiki/1252599548343744/1298069163343905)

Java标准库的java.io.InputStream定义了所有输入流的超类：

**public abstract int read() throws IOException;**

FileInputStream实现了文件流输入；

ByteArrayInputStream在内存中模拟一个字节流输入。

总是使用try(resource)来保证InputStream正确关闭。

### [OutputStream](https://www.liaoxuefeng.com/wiki/1252599548343744/1298069169635361)

Java标准库的java.io.OutputStream定义了所有输出流的超类：

FileOutputStream实现了文件流输出；

ByteArrayOutputStream在内存中模拟一个字节流输出。

某些情况下需要手动调用OutputStream的flush()方法来强制输出缓冲区。

总是使用try(resource)来保证OutputStream正确关闭。

### [Filter模式](https://www.liaoxuefeng.com/wiki/1252599548343744/1298364142452770)

Java的IO标准库使用Filter模式为InputStream和OutputStream增加功能：

可以把一个InputStream和任意个FilterInputStream组合；

可以把一个OutputStream和任意个FilterOutputStream组合。

Filter模式可以在运行期动态增加功能（又称Decorator模式）

### [操作Zip](https://www.liaoxuefeng.com/wiki/1252599548343744/1298366336073762)

### [读取classpath资源](https://www.liaoxuefeng.com/wiki/1252599548343744/1298366384308257)

[序列化](https://www.liaoxuefeng.com/wiki/1252599548343744/1298366845681698)

[Reader](https://www.liaoxuefeng.com/wiki/1252599548343744/1298366902304801)

[Writer](https://www.liaoxuefeng.com/wiki/1252599548343744/1298366912790561)

[PrintStream和PrintWriter](https://www.liaoxuefeng.com/wiki/1252599548343744/1302299230076961)

[使用Files](https://www.liaoxuefeng.com/wiki/1252599548343744/1375100746072098)

[日期与时间](https://www.liaoxuefeng.com/wiki/1252599548343744/1255943660631584)

[基本概念](https://www.liaoxuefeng.com/wiki/1252599548343744/1298613246361634)

[Date和Calendar](https://www.liaoxuefeng.com/wiki/1252599548343744/1303791989162017)

[LocalDateTime](https://www.liaoxuefeng.com/wiki/1252599548343744/1303871087444002)

[ZonedDateTime](https://www.liaoxuefeng.com/wiki/1252599548343744/1303904694304801)

[DateTimeFormatter](https://www.liaoxuefeng.com/wiki/1252599548343744/1303985694703650)

[Instant](https://www.liaoxuefeng.com/wiki/1252599548343744/1303905346519074)

[最佳实践](https://www.liaoxuefeng.com/wiki/1252599548343744/1303978948165666)

[单元测试](https://www.liaoxuefeng.com/wiki/1252599548343744/1255945269146912)

[编写JUnit测试](https://www.liaoxuefeng.com/wiki/1252599548343744/1304048154181666)

[使用Fixture](https://www.liaoxuefeng.com/wiki/1252599548343744/1304049490067490)

[异常测试](https://www.liaoxuefeng.com/wiki/1252599548343744/1304064312737826)

[条件测试](https://www.liaoxuefeng.com/wiki/1252599548343744/1304073489874978)

[参数化测试](https://www.liaoxuefeng.com/wiki/1252599548343744/1304065789132833)

[正则表达式](https://www.liaoxuefeng.com/wiki/1252599548343744/1255945288020320)

[正则表达式简介](https://www.liaoxuefeng.com/wiki/1252599548343744/1304066130968610)

[匹配规则](https://www.liaoxuefeng.com/wiki/1252599548343744/1304066080636961)

[复杂匹配规则](https://www.liaoxuefeng.com/wiki/1252599548343744/1306046675025953)

[分组匹配](https://www.liaoxuefeng.com/wiki/1252599548343744/1306046706483233)

[非贪婪匹配](https://www.liaoxuefeng.com/wiki/1252599548343744/1306046731649057)

[搜索和替换](https://www.liaoxuefeng.com/wiki/1252599548343744/1306046817632290)

[加密与安全](https://www.liaoxuefeng.com/wiki/1252599548343744/1255943717668160)

[编码算法](https://www.liaoxuefeng.com/wiki/1252599548343744/1304227703947297)

[哈希算法](https://www.liaoxuefeng.com/wiki/1252599548343744/1304227729113121)

[BouncyCastle](https://www.liaoxuefeng.com/wiki/1252599548343744/1305362418368545)

[Hmac算法](https://www.liaoxuefeng.com/wiki/1252599548343744/1305366354722849)

[对称加密算法](https://www.liaoxuefeng.com/wiki/1252599548343744/1304227762667553)

[口令加密算法](https://www.liaoxuefeng.com/wiki/1252599548343744/1304227859136546)

[密钥交换算法](https://www.liaoxuefeng.com/wiki/1252599548343744/1304227905273889)

[非对称加密算法](https://www.liaoxuefeng.com/wiki/1252599548343744/1304227873816610)

[签名算法](https://www.liaoxuefeng.com/wiki/1252599548343744/1304227943022626)

[数字证书](https://www.liaoxuefeng.com/wiki/1252599548343744/1304227968188450)

[多线程](https://www.liaoxuefeng.com/wiki/1252599548343744/1255943750561472)

[多线程基础](https://www.liaoxuefeng.com/wiki/1252599548343744/1304521607217185)

[创建新线程](https://www.liaoxuefeng.com/wiki/1252599548343744/1306580710588449)

[线程的状态](https://www.liaoxuefeng.com/wiki/1252599548343744/1306580742045730)

[中断线程](https://www.liaoxuefeng.com/wiki/1252599548343744/1306580767211554)

[守护线程](https://www.liaoxuefeng.com/wiki/1252599548343744/1306580788183074)

[线程同步](https://www.liaoxuefeng.com/wiki/1252599548343744/1306580844806178)

[同步方法](https://www.liaoxuefeng.com/wiki/1252599548343744/1306580867874849)

[死锁](https://www.liaoxuefeng.com/wiki/1252599548343744/1306580888846370)

[使用wait和notify](https://www.liaoxuefeng.com/wiki/1252599548343744/1306580911915042)

[使用ReentrantLock](https://www.liaoxuefeng.com/wiki/1252599548343744/1306580960149538)

[使用Condition](https://www.liaoxuefeng.com/wiki/1252599548343744/1306581033549858)

[使用ReadWriteLock](https://www.liaoxuefeng.com/wiki/1252599548343744/1306581002092578)

[使用StampedLock](https://www.liaoxuefeng.com/wiki/1252599548343744/1309138673991714)

[使用Concurrent集合](https://www.liaoxuefeng.com/wiki/1252599548343744/1306581060812834)

[使用Atomic](https://www.liaoxuefeng.com/wiki/1252599548343744/1306581083881506)

[使用线程池](https://www.liaoxuefeng.com/wiki/1252599548343744/1306581130018849)

[使用Future](https://www.liaoxuefeng.com/wiki/1252599548343744/1306581155184674)

[使用CompletableFuture](https://www.liaoxuefeng.com/wiki/1252599548343744/1306581182447650)

[使用ForkJoin](https://www.liaoxuefeng.com/wiki/1252599548343744/1306581226487842)

[使用ThreadLocal](https://www.liaoxuefeng.com/wiki/1252599548343744/1306581251653666)

[Maven基础](https://www.liaoxuefeng.com/wiki/1252599548343744/1255945359327200)

[Maven介绍](https://www.liaoxuefeng.com/wiki/1252599548343744/1309301146648610)

[依赖管理](https://www.liaoxuefeng.com/wiki/1252599548343744/1309301178105890)

[构建流程](https://www.liaoxuefeng.com/wiki/1252599548343744/1309301196980257)

[使用插件](https://www.liaoxuefeng.com/wiki/1252599548343744/1309301217951777)

[模块管理](https://www.liaoxuefeng.com/wiki/1252599548343744/1309301243117601)

[使用mvnw](https://www.liaoxuefeng.com/wiki/1252599548343744/1305148057976866)

[发布Artifact](https://www.liaoxuefeng.com/wiki/1252599548343744/1347981037010977)

[网络编程](https://www.liaoxuefeng.com/wiki/1252599548343744/1255945371526048)

[网络编程基础](https://www.liaoxuefeng.com/wiki/1252599548343744/1305163149082658)

[TCP编程](https://www.liaoxuefeng.com/wiki/1252599548343744/1305207629676577)

[UDP编程](https://www.liaoxuefeng.com/wiki/1252599548343744/1319099802058785)

[发送Email](https://www.liaoxuefeng.com/wiki/1252599548343744/1319099923693601)

[接收Email](https://www.liaoxuefeng.com/wiki/1252599548343744/1319099948859426)

[HTTP编程](https://www.liaoxuefeng.com/wiki/1252599548343744/1319099982413858)

[RMI远程调用](https://www.liaoxuefeng.com/wiki/1252599548343744/1323711850348577)

[XML与JSON](https://www.liaoxuefeng.com/wiki/1252599548343744/1255945389334784)

[XML简介](https://www.liaoxuefeng.com/wiki/1252599548343744/1305161429418018)

[使用DOM](https://www.liaoxuefeng.com/wiki/1252599548343744/1320414976409634)

[使用SAX](https://www.liaoxuefeng.com/wiki/1252599548343744/1320418577219618)

[使用Jackson](https://www.liaoxuefeng.com/wiki/1252599548343744/1320418596093986)

[使用JSON](https://www.liaoxuefeng.com/wiki/1252599548343744/1320418650619938)

[JDBC编程](https://www.liaoxuefeng.com/wiki/1252599548343744/1255943820274272)

[JDBC简介](https://www.liaoxuefeng.com/wiki/1252599548343744/1305152088703009)

[JDBC查询](https://www.liaoxuefeng.com/wiki/1252599548343744/1321748435828770)

[JDBC更新](https://www.liaoxuefeng.com/wiki/1252599548343744/1321748475674658)

[JDBC事务](https://www.liaoxuefeng.com/wiki/1252599548343744/1321748500840481)

[JDBC Batch](https://www.liaoxuefeng.com/wiki/1252599548343744/1322290857902113)

[JDBC连接池](https://www.liaoxuefeng.com/wiki/1252599548343744/1321748528103458)

[函数式编程](https://www.liaoxuefeng.com/wiki/1252599548343744/1255943847278976)

[Lambda基础](https://www.liaoxuefeng.com/wiki/1252599548343744/1305158055100449)

[方法引用](https://www.liaoxuefeng.com/wiki/1252599548343744/1305207799545890)

[使用Stream](https://www.liaoxuefeng.com/wiki/1252599548343744/1322402873081889)

[创建Stream](https://www.liaoxuefeng.com/wiki/1252599548343744/1322655160467490)

[使用map](https://www.liaoxuefeng.com/wiki/1252599548343744/1322402942287906)

[使用filter](https://www.liaoxuefeng.com/wiki/1252599548343744/1322402956967969)

[使用reduce](https://www.liaoxuefeng.com/wiki/1252599548343744/1322402971648033)

[输出集合](https://www.liaoxuefeng.com/wiki/1252599548343744/1322656099991586)

[其他操作](https://www.liaoxuefeng.com/wiki/1252599548343744/1322403061825570)

[设计模式](https://www.liaoxuefeng.com/wiki/1252599548343744/1264742167474528)

[创建型模式](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319090782242)

[工厂方法](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319170474017)

[抽象工厂](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319134822433)

[生成器](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319155793953)

[原型](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319195639841)

[单例](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319214514210)

[结构型模式](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319233388578)

[适配器](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319245971489)

[桥接](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319266943009)

[组合](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319283720226)

[装饰器](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319302594594)

[外观](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319346634785)

[享元](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319417937953)

[代理](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319432618017)

[行为型模式](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319453589538)

[责任链](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319474561057)

[命令](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319491338273)

[解释器](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319508115489)

[迭代器](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319524892705)

[中介](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319541669922)

[备忘录](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319562641441)

[观察者](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319577321505)

[状态](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319592001569)

[策略](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319606681634)

[模板方法](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319636041762)

[访问者](https://www.liaoxuefeng.com/wiki/1252599548343744/1281319659110433)

[Web开发](https://www.liaoxuefeng.com/wiki/1252599548343744/1255945497738400)

[Web基础](https://www.liaoxuefeng.com/wiki/1252599548343744/1304265903570978)

[Servlet入门](https://www.liaoxuefeng.com/wiki/1252599548343744/1304265949708322)

[Servlet开发](https://www.liaoxuefeng.com/wiki/1252599548343744/1266264743830016)

[Servlet进阶](https://www.liaoxuefeng.com/wiki/1252599548343744/1328705066500130)

[重定向与转发](https://www.liaoxuefeng.com/wiki/1252599548343744/1328761739935778)

[使用Session和Cookie](https://www.liaoxuefeng.com/wiki/1252599548343744/1328768897515553)

[JSP开发](https://www.liaoxuefeng.com/wiki/1252599548343744/1266262958498784)

[MVC开发](https://www.liaoxuefeng.com/wiki/1252599548343744/1266264917931808)

[MVC高级开发](https://www.liaoxuefeng.com/wiki/1252599548343744/1337408645759009)

[使用Filter](https://www.liaoxuefeng.com/wiki/1252599548343744/1266264823560128)

[修改请求](https://www.liaoxuefeng.com/wiki/1252599548343744/1328976435871777)

[修改响应](https://www.liaoxuefeng.com/wiki/1252599548343744/1328976456843298)

[使用Listener](https://www.liaoxuefeng.com/wiki/1252599548343744/1304266123771937)

[部署](https://www.liaoxuefeng.com/wiki/1252599548343744/1304266260086817)

[Spring开发](https://www.liaoxuefeng.com/wiki/1252599548343744/1266263217140032)

[IoC容器](https://www.liaoxuefeng.com/wiki/1252599548343744/1266265100383840)

[IoC原理](https://www.liaoxuefeng.com/wiki/1252599548343744/1282381977747489)

[装配Bean](https://www.liaoxuefeng.com/wiki/1252599548343744/1282382145519649)

[使用Annotation配置](https://www.liaoxuefeng.com/wiki/1252599548343744/1282382596407330)

[定制Bean](https://www.liaoxuefeng.com/wiki/1252599548343744/1308043627200545)

[使用Resource](https://www.liaoxuefeng.com/wiki/1252599548343744/1282383017934882)

[注入配置](https://www.liaoxuefeng.com/wiki/1252599548343744/1282383225552930)

[使用条件装配](https://www.liaoxuefeng.com/wiki/1252599548343744/1308043874664482)

[使用AOP](https://www.liaoxuefeng.com/wiki/1252599548343744/1266265125480448)

[装配AOP](https://www.liaoxuefeng.com/wiki/1252599548343744/1310052352786466)

[使用注解装配AOP](https://www.liaoxuefeng.com/wiki/1252599548343744/1310052317134882)

[AOP避坑指南](https://www.liaoxuefeng.com/wiki/1252599548343744/1339039378571298)

[访问数据库](https://www.liaoxuefeng.com/wiki/1252599548343744/1282383540125729)

[使用JDBC](https://www.liaoxuefeng.com/wiki/1252599548343744/1282383699509281)

[使用声明式事务](https://www.liaoxuefeng.com/wiki/1252599548343744/1282383642886177)

[使用DAO](https://www.liaoxuefeng.com/wiki/1252599548343744/1282383605137441)

[集成Hibernate](https://www.liaoxuefeng.com/wiki/1252599548343744/1266263275862720)

[集成JPA](https://www.liaoxuefeng.com/wiki/1252599548343744/1282383789686817)

[集成MyBatis](https://www.liaoxuefeng.com/wiki/1252599548343744/1331313418174498)

[设计ORM](https://www.liaoxuefeng.com/wiki/1252599548343744/1282383340896289)

[开发Web应用](https://www.liaoxuefeng.com/wiki/1252599548343744/1282383877767201)

[使用Spring MVC](https://www.liaoxuefeng.com/wiki/1252599548343744/1282383921807393)

[使用REST](https://www.liaoxuefeng.com/wiki/1252599548343744/1282384941023266)

[集成Filter](https://www.liaoxuefeng.com/wiki/1252599548343744/1282384114745378)

[使用Interceptor](https://www.liaoxuefeng.com/wiki/1252599548343744/1347180610715681)

[处理CORS](https://www.liaoxuefeng.com/wiki/1252599548343744/1282384360112162)

[国际化](https://www.liaoxuefeng.com/wiki/1252599548343744/1282384236380194)

[异步处理](https://www.liaoxuefeng.com/wiki/1252599548343744/1282384506912802)

[使用WebSocket](https://www.liaoxuefeng.com/wiki/1252599548343744/1282384966189089)

[集成第三方组件](https://www.liaoxuefeng.com/wiki/1252599548343744/1282385643569186)

[集成JavaMail](https://www.liaoxuefeng.com/wiki/1252599548343744/1282385704386594)

[集成JMS](https://www.liaoxuefeng.com/wiki/1252599548343744/1304266721460258)

[使用Scheduler](https://www.liaoxuefeng.com/wiki/1252599548343744/1282385878450210)

[集成JMX](https://www.liaoxuefeng.com/wiki/1252599548343744/1282385687609378)

[Spring Boot开发](https://www.liaoxuefeng.com/wiki/1252599548343744/1266265175882464)

[第一个Spring Boot应用](https://www.liaoxuefeng.com/wiki/1252599548343744/1282386201411617)

[使用开发者工具](https://www.liaoxuefeng.com/wiki/1252599548343744/1282386532761633)

[打包Spring Boot应用](https://www.liaoxuefeng.com/wiki/1252599548343744/1282386595676193)

[瘦身Spring Boot应用](https://www.liaoxuefeng.com/wiki/1252599548343744/1304267002478625)

[使用Actuator](https://www.liaoxuefeng.com/wiki/1252599548343744/1282386381766689)

[使用Profiles](https://www.liaoxuefeng.com/wiki/1252599548343744/1282388483112993)

[使用Conditional](https://www.liaoxuefeng.com/wiki/1252599548343744/1282386318852129)

[加载配置文件](https://www.liaoxuefeng.com/wiki/1252599548343744/1304267426103329)

[禁用自动配置](https://www.liaoxuefeng.com/wiki/1252599548343744/1282389045149729)

[添加Filter](https://www.liaoxuefeng.com/wiki/1252599548343744/1282389221310497)

[集成第三方组件](https://www.liaoxuefeng.com/wiki/1252599548343744/1282389200338977)

[集成Open API](https://www.liaoxuefeng.com/wiki/1252599548343744/1283318525984802)

[访问Redis](https://www.liaoxuefeng.com/wiki/1252599548343744/1282386499207201)

[集成Artemis](https://www.liaoxuefeng.com/wiki/1252599548343744/1282388602650658)

[集成RabbitMQ](https://www.liaoxuefeng.com/wiki/1252599548343744/1282385960239138)

[集成Kafka](https://www.liaoxuefeng.com/wiki/1252599548343744/1282388443267106)

[Spring Cloud开发](https://www.liaoxuefeng.com/wiki/1252599548343744/1266263401691296)

# 小技巧：

1. 表达式，公式的作用：使用更多的数学公式，让一些运算问题代码编写更简单。
2. 溢出,是常见问题
3. 对于多个数值==判断的情况，字符串.equals判断的情况，使用switch结构更加清晰。
4. 检查是否漏写了break语句和default语句，穿透效应

1. 遍历数组，注意数组长度不要越界

1. 基本类型数据使用快速排序法，对象数组使用堆排序
2. 严格来说面向对象，时先确定数据结构，数据类型；最后确定实用算法；面向过程先确定，基于算法，在抽象数据结构。

Java中，class，abstract class，interface等定义出来的都是数据类型

1. String类提供了多个重载方法indexOf()，可以查找子串：

int indexOf(int ch)：根据字符的Unicode码查找；

int indexOf(String str)：根据字符串查找；

int indexOf(int ch, int fromIndex)：根据字符查找，但指定起始位置；

int indexOf(String str, int fromIndex)根据字符串查找，但指定起始位置。

1. 子类自动获得了父类的所有字段，严禁定义与父类重名的字段！
2. 在OOP的术语中，我们把Person称为超类（super class），父类（parent class），基类（base class），把Student称为子类（subclass），扩展类（extended class）
3. 任何类，除了Object，都会继承自某个类
4. 向下转型很可能会失败。失败的时候，Java虚拟机会报ClassCastException
5. 从Java 14开始，判断instanceof后，可以直接转型为指定变量，避免再次强制转型
6. 构造方法中初始化final字段,保证实例一旦创建，其final字段就不可修改
7. 通过abstract定义的方法是抽象方法，它只有定义，没有实现。抽象方法定义了子类必须实现的接口规范；
8. 定义了抽象方法的class必须被定义为抽象类，从抽象类继承的子类必须实现抽象方法；
9. 如果不实现抽象方法，则该子类仍是一个抽象类；
10. 面向抽象编程使得调用者只关心抽象方法的定义，不关心子类的具体实现。
11. 当我们需要给接口新增一个方法时，会涉及到修改全部子类,这个时候使用default方法，来增加接口的新方法；子类就不必全部修改，只需要在需要覆写的地方去覆写新增方法。
12. 比如jdbc接口升级了，老驱动就会挂，有default方法就可以给老版本续命
13. 任何类型的.java的存放源代码的文件，都能被编成.class文件,与对应定义的数据类型无关;
14. class关键字定义类 ，interface关键字定义接口，所以接口不是类。
15. 在java 5之后，接口没有父类，允许使用重写覆盖@override。实现接口中的方法，是重写机制。
16. IDE集成开发工具编译的命令相对比较复杂，我们需要在src目录下执行javac命令：

javac -d ../bin ming/Person.java hong/Person.java mr/jun/Arrays.java

1. 不使用修饰符的方法就是包作用域
2. 主要在方法内部采用匿名内部类的方式来实例化对象，相比outer inner class少了更多代码
3. 除了接口，匿名类可以完全继承自普通类，

Runnable r = new Runnable() { // 实现必要的抽象方法...};

HashMap<String, String> map2 = new HashMap<>() {}

1. classpath是JVM用到的一个环境变量，它用来指示JVM如何搜索class。
2. 不要把任何Java核心库添加到classpath中！JVM根本不依赖classpath加载核心库！
3. 更好的做法是，不要设置classpath！默认的当前目录.对于绝大多数情况都够用了
4. 执行一个jar包的class，就可以把jar包放到classpath中：java -cp ./hello.jar abc.xyz.Hello
5. 创建jar包的方式：手动使用zip压缩工具，打包修改后缀为.jar/通过命令行参数打包类似unix tar命令jar cvfe CoreClasses.jar inheritance.ManagerTest(指定main启动类的package-path到清单文件中) ../inheritance/\*.class（打包所有class文件）
6. jar包还可以包含其它jar包，这个时候，就需要在MANIFEST.MF文件里配置classpath了。
7. 在大型项目中，不可能手动编写MANIFEST.MF文件，再手动创建zip包。Java社区提供了大量的开源构建工具，例如Maven，可以非常方便地创建jar包。
8. 模块是java9引进的，针对大型java程序，包含多个jar文件的处理方式，解决class的依赖。
9. Jar是存放class文件的容器。 但实际上开发过程中会产生java核心库rt.jar，自己的app.Jar，还有一堆第三方jar包。运行的时候需要将所有的jar写到classpath中，
10. jar包是class的封装，一个容器，jmod是模块，解决应用的依赖关系，jre是最终的java程序，可以非常方便的分发部署。
11. Java的String和char在内存中总是以Unicode编码表示
12. StringJoiner是jdk8新增的一类class，分隔符拼接数组的需求很常见，所以Java标准库还提供了一个StringJoiner，用指定分隔符拼接字符串数组时，使用StringJoiner可指定分隔符,开头,结尾字符，或者String.join()更方便
13. 常量是指在程序的整个运行过程中值保持不变的量,java中通常final修饰
14. 普通的字符串+操作ava编译器在编译时就自动把多个连续的+操作编码为StringConcatFactory的操作。在运行期，StringConcatFactory会自动把字符串连接操作优化为数组复制或者StringBuilder操作。
15. 通过new操作符创建Integer实例(不推荐使用,会有编译警告)
16. 编译器帮助我们自动拆箱装，少写代码
17. 装箱和拆箱会影响代码的执行效率，因为编译后的class代码是严格区分基本类型和引用类型的。并且，自动拆箱执行时可能会报NullPointerException：

所有的包装类型都是不变类。我们查看Integer的源码可知 **private** **final** **xxx** value;

1. Integer是引用类型，必须使用equals()比较
2. Java中没有无符号unsigned类型的无符号整型，都是有signed byte、short、int、long无符号整型和有符号整型的转换在Java中就需要借助包装类型的静态方法完成。
3. switch

最后，枚举类可以应用在switch语句中。因为枚举类天生具有类型信息和有限个枚举 常量，所以比int、String类型更适合用在switch语句中

1. 在Java中，由CPU原生提供的整型最大范围是64位long型整数。使用long型整数可以直接通过CPU指令进行计算，速度非常快。
2. 总是使用compareTo()比较两个BigDecimal的值，不要使用equals()，返回值负数，0正数，小于等于大于
3. java标准库还提供了一个StrictMath，它提供了和Math几乎一模一样的方法。这两个类的区别在于，由于浮点数计算存在误差，不同的平台（例如x86和ARM）计算的结果可能不一致（指误差不同），因此，StrictMath保证所有平台计算结果都是完全相同的，而Math会尽量针对平台优化计算速度，所以，绝大多数情况下，使用Math就足够了。
4. 伪随机数Radom，不指定种子根据系统时间随机。指定种子，生成一串固定的伪随机数。Math.radom就是调用了Radom的无参构造
5. 要的是一个不可预测的安全的随机数，SecureRandom来获取，无法指定种子
6. SecureRandom.getInstanceStrong(); // 获取高强度安全随机数生成器
7. new SecureRandom(); // 获取普通的安全随机数生成器
8. 需要使用安全随机数的时候，必须使用SecureRandom，绝不能使用Random！
9. 编译器对RuntimeException及其子类不做强制捕获要求，不是指应用程序本身不应该捕获并处理RuntimeException。是否需要捕获，具体问题具体分析。
10. 方法声明的Checked Exception，不在调用层捕获，也必须在更高的调用层捕获。
11. main方法是最后捕获异常的地方
12. 测试代码，可以直接把main()方法定义为throws Exception：
13. 所有异常都可以调用printStackTrace()方法打印异常栈，这是一个简单有用的快速打印异常的方法。
14. 方法声明了可能抛出的异常，所以可以不写catch。

抛出异常创建某个Exception的实例；

用throw语句抛出。

NullPointerException e = **new** NullPointerException();

**throw** e;

throw new NullPointerException();

1. 捕获到异常并再次抛出时，一定要留住原始异常Exception，否则很难定位第一案发现场
2. 自定义异常，建议BaseException 继承 runtimeexcepton，设计尽可能多的构造方法。别的自定义异常扩展该异常即可
3. 最佳实践
4. 在开发阶段，始终使用Commons Logging接口来写入日志，并且开发阶段无需引入Log4j。
5. **instanceof判断类型的时候可以判断父类类型，子类类型，**

**而getClass()通过反射获取当前数据类型 使用==，equals可以精确判断数据 类型**

1. 反射的目的是为了获得某个实例对应的Class信息，也就是获取该对象的Class实例，Class实例在JVM中有且只有一个。同时可以获取该Class对应一个实例化对象
2. cls.newInstance()就可以创建一个当前class的对象，前提是被调用对象包含公共的无参构造。jdk9之后Class类弃用了该方法，当外部调用对象实例且该对象构造方法私有化会报IllegalAccessException参数不合法异常
3. Java语句顺序执行，.class文件顺序加载,先将主函数所在类加载到jvm中，而后执行到创建实例对象的方法时，才会将另一个.class文件加载到虚拟机中。

这就叫动态加载

1. 动态代理（Dynamic Proxy） 和静态创建一个inteface文件是相对应的
2. 而动态则是通过jdk提供java.lang.reflect.Proxy类去实现,通过这种方法我们并不用编写某个接口的实现类
3. 而是直接通过JDK提供的一个Proxy.newProxyInstance()创建了一个接口对象。
4. 没有实现类但是在运行期动态创建了一个接口对象的方式，我们称为动态代码。
5. JDK提供的动态创建接口对象的方式，就叫动态代理。
6. 必须设置@Target和@Retention，@Retention一般设置为RUNTIME，因为我们自定义的注解通常要求在运行期读取。一般情况下，不必写@Inherited和@Repeatable。
7. 接口也可以使用泛型interface Comparable<T>
8. 使用泛型接口，来进行自定义对象排序，可用于String类型，根据对象自身属性去进行合理排序
9. 由于擦拭法的T，在编译后实际为Object类型，所以擦拭法中定义方法不能复写Object中的方法，编译器会阻止
10. 如果在方法内部创建了泛型数组，最好不要将它返回给外部使用。
11. 只要实现了Iterable接口的集合类都可以直接用for each循环来遍历，Java编译器本身并不知道如何遍历集合对象，但它会自动把for each循环变成Iterator的调用，原因就在于Iterable接口定义了一个Iterator<E> iterator()方法，强迫集合类必须返回一个Iterator实例。
12. 正确使用List的contains()、indexOf()方法，存入对象必须复写equals（0方法，这样才能正确校验是否包含该实例
13. java.util.Objects一般使用Util工具类下的Objects类的equals方法可以直接对比引用类型，且不用直接比较null类型
14. Map是一种键值映射关系，因此在便利的时候不会保证顺序
15. 一个类如果覆写了equals()，就必须覆写hashCode()
16. 由于properties类实际上是hashtable，这个历史遗留问题，不用管。往往只需要读取，写入properties文件即可，而不需要调用get（），put（）等从hashtable继承的方法。
17. 使用List和HashMap实现一个简单的缓存机制；使用Set的去重功能，根据sequence去除网络故障产生的重复消息；使用TreeMap进行排序
18. 注意：不要把null添加到队列Queue中，否则poll()方法返回null时，很难确定是取到了null元素还是队列为空。
19. LinkedList即实现了List接口，又实现了Queue接口
20. PriorityQueue默认按照元素比较顺序排序（必须实现Comparable接口）， \* 也可以通过Comparator自定义排序算法（元素不必实现Comparable接口）比如Vip客户优先级高于普通用户。可通过Comparator接口自定义排序算法。排票系统
21. Deque是一个接口，它的实现类有ArrayDeque和LinkedList。
22. 发现LinkedList真是一个全能选手，它即是List，又是Queue，还是Deque。但是我们在使用的时候，总是用特定的接口来引用它，这是因为持有接口说明代码的抽象层次更高，而且接口本身定义的方法代表了特定的用途。
23. 可见面向抽象编程的一个原则就是：尽量持有接口，而不是具体的实现类。
24. 一个Stack的用途：对整数进行进制的转换就可以利用栈。jvm调用方法就用到栈类型数据结构。计算中缀表达式的过程，编译器在编译，且计算后缀表达式用到了栈类型数据结构
25. 迭代器Iterator将迭代遍历集合方式统一，这样不必关心具体数据类型，及不必关心内部存储结构。
26. 对于自定义集合类，若使用foreach循环。满足组：1.集合类实现Iterator接口，并返回Iterator对象。2.用Iterator对象迭代内部数据
27. 从Java 5开始，引入了更高效的并发集合类,则Collections提供的线程安全方法基本不用了
28. IO，input是把外部数据读取到内存中的方向，如将磁盘数据/网络读取到内存；Output是把内存往外写的方向，将内存数据写到文件，把数据从内存输出到网络
29. Java.io.File对象表示一个文件或者目录
30. java.nio.file.Path 对象可以用于对目录进行复杂的拼接、遍历等操作