**Java**基础核心总结

Java基础核心总结

Java概述

什么是Java?

Java的特点

Java开发环境

JDK

JRE

Java开发环境配置

Java基本语法

数据类型

基础语法

运算符

Java执行控制流程

条件语句

if条件语句

if...else条件语句

if...else if多分支语句

switch多分支语句

循环语句

while循环语句

do...while 循环

for循环语句

跳转语句

break语句

continue 语句

面向对象

类也是一种对象

对象的创建

属性和方法

构造方法

方法重载

方法的重写

初始化

类的初始化

成员初始化

构造器初始化

初始化顺序

数组初始化

对象的销毁

对象作用域

this 和 super

访问控制权限

继承

多态

组合

代理

向上转型

static

final

接口和抽象类

接口

抽象类

异常

认识 Exception 什么是Throwable 常见的Exception

与Exception有关的Java关键字 throws 不口 throw try、finally、catch

什么是Error

内部类

创建内部类

隹厶

Iterable 接口

顶层接口

ArrayList

Vector

LinkedList 类

Stack

HashSet

TreeSet

LinkedHashSet 类

PriorityQueue

HashMap

TreeMap 类 LinkedHashMap 类 Hashtable 类 IdentityHashMap 类 WeakHashMap 类

Collections 类

集合实现类特征图

泛形

泛型的使用

用泛型表示类

用泛型表示接口

泛型方法 泛型通配符

反射

Class 类

Field 类

Method 类

ClassLoader 类

枚举

枚举特性

枚举和普通类一样

枚举神秘之处 枚举类

I/O

File 类

基础IO类和相关方法

Inputstream

Outputstream

Reader 类

Writer 类

InputStream及其子类 OutputStream及其子类 Reader及其子类 Writer及其子类

注解

关于null的几种处理方式

大小写敏感

null是任何引用类型的初始值 null只是一种特殊的值 使用Null-Safe方法 null判断 关于思维导图

Java.IO

Java.lang

Java.math

Java.net

先来看一下本篇文章的思维导图吧，我会围绕下面这些内容进行讲解。内容很干，小伙伴们看完还希望

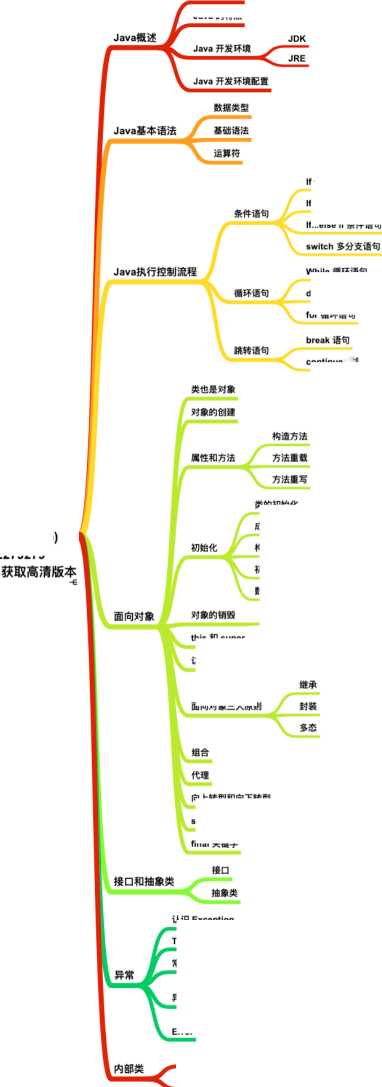
不吝转发。（高清思维导图版本关注作者公众号

Java建设者回复

Java666

获取，其他思维导图获取

方式在文末）。



iterable 接口

顶层接口

Vector

LlnkedList

Stack

TreeSet

LlnkedHashSet

PriorityQueue

HashMap

集合

TreeMap

LinkedHashMap

HashTable

IdentityHashMap

WeakHashMap

Collections

Inputstream 类

Outputstream 类

Reader 类

Writer 类

集合实现类特征图

Constructor 类

基础I/O类和相关方法

Inputstream及其子类

Writer及其子类

1、大小写敏感

2、null是任何引用类型的初始值

4、null不能赋值给基本数据类型

5、将nulHB给包装类，自动拆箱会报NPE

6、帯有nul堕jl用类型变量，instanceof会报false

7、静态変量为null调用静态方法不会拋岀NPE

8,使用nulKI安全的方法

9、使用==或!=判断null

3、null既不是对象也不是类型，它是一种特殊的值 你可以将它赋值给任何引用类型

用泛型表示接口

Method 类

ClassLoader 类

枚举就是一种普通类

枚举的神秘之处

Outputstream及其子类

**Java**基础核心总结 欢迎关注公众号

**（Java**建设者&程序员**cxuan）** 作者微信：**1x252279279** 公众号回复**Java666**

什么是Java

If条件语句

lf...else条件语句

lf...else if条件语句

do...while循环语句

成员初始化

构造器初始化

初始化顺序

敬组初始化

对象作用域

访问控制权限

static关键字

Throwable

常见的Exception

throws 和 throw

内部类的创建

创建内部类的几种方式

异常关键字/

try、finally、catch

Java的特点

this 和 super

**While**循环语句

**for**循环语句

continue 语句

类的初始化

面向对象三大原则

向上转型和向下转型

final关键字

认识 Exception

Error

下面开始我们的文章。

**Java**概述

什么是**Java?**

Java是Sun Microsystems于1995年首次发布的一种编程语言和计算平台。编程语言还比较好理 解，那么什么是计算平台呢？

计算平台是在电脑中运行应用程序(软件)的环境，包括硬件环境和软件环境。一般系统平台 包括一台电脑的硬件体系结构、操作系统、运行时库。

Java是快速，安全和可靠的。从笔记本电脑到数据中心，从游戏机到科学超级计算机，从手机到互联 网，Java无处不在！ Java主要分为三个版本

* JavaSE(J2SE)(Java2 Platform Standard Edition, java平台标准版)
* JavaEE(J2EE)(Java 2 Platform,Enterprise Edition, java平台企业版)
* JavaME(J2ME)(Java 2 Platform Micro Edition, java平台微型版)。

**Java**的特点

* Java是一门面向对象的编程语言

什么是面向对象？面向对象(Object Oriented)是一种软件开发思想。它是对现实世界的一种抽象, 面向对象会把相关的数据和方法组织为一个整体来看待。

相对的另外一种开发思想就是面向过程的开发思想，什么面向过程？面向过程(Procedure

Oriented)是一种以过程为中心的编程思想。举个例子：比如你是个学生，你每天去上学需要做几件 事情？

起床、穿衣服、洗脸刷牙，吃饭，去学校。一般是顺序性的完成一系列动作。

1. class student {

void student\_wakeUp(){...}

void student\_cloth(){...}

void student\_wash(){...}

void student\_eating(){...}

void student\_gotoSchool(){...}

而面向对象可以把学生进行抽象，所以这个例子就会变为

class student(){

1

2

3

4

5

6

7

void wakeUp(){...}

void cloth(){...}

void wash(){...}

void eating(){...}

void gotoSchool(){...}

可以不用严格按照顺序来执行每个动作。这是特点一。

-Java摒弃了 C++中难以理解的多继承、指针、内存管理等概念；不用手动管理对象的生命周期， 这是特征二。

-Java语言具有功能强大和简单易用两个特征，现在企业级开发，快速敏捷开发，尤其是各种框架 的出现，使Java成为越来越火的一门语言。这是特点三。

・Java是一门静态语言，静态语言指的就是在编译期间就能够知道数据类型的语言，在运行前就能 够检查类型的正确性，一旦类型确定后就不能再更改，比如下面这个例子。

1. public void foo() {
2. int x = 5;
3. boolean b = x;
4. }

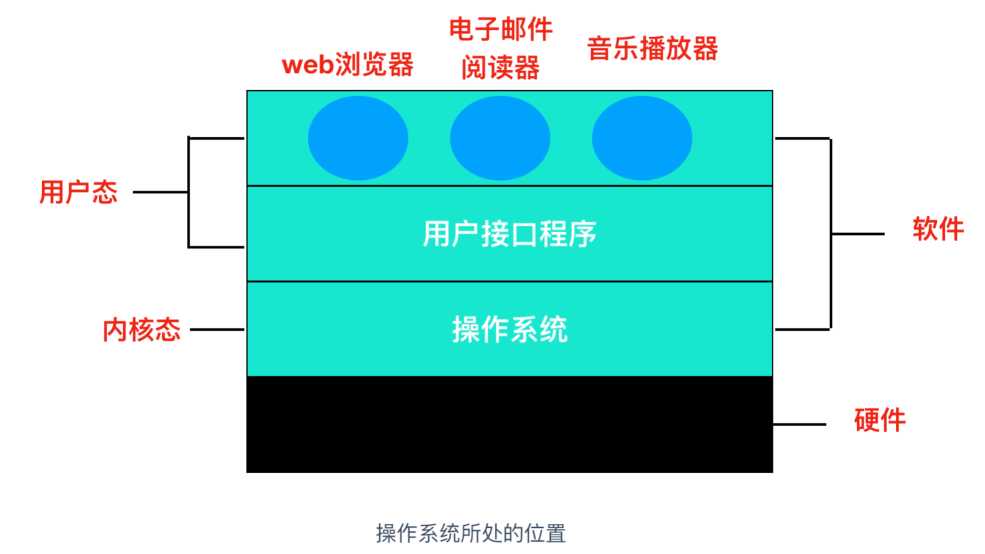
静态语言主要有 Pascal, Perl, C/C++, JAVA, C#, Scala 等。

相对应的，动态语言没有任何特定的情况需要指定变量的类型，在运行时确定的数据类型。比如有**Lisp, Perl, Python**、**Ruby**、**JavaScript** 等。

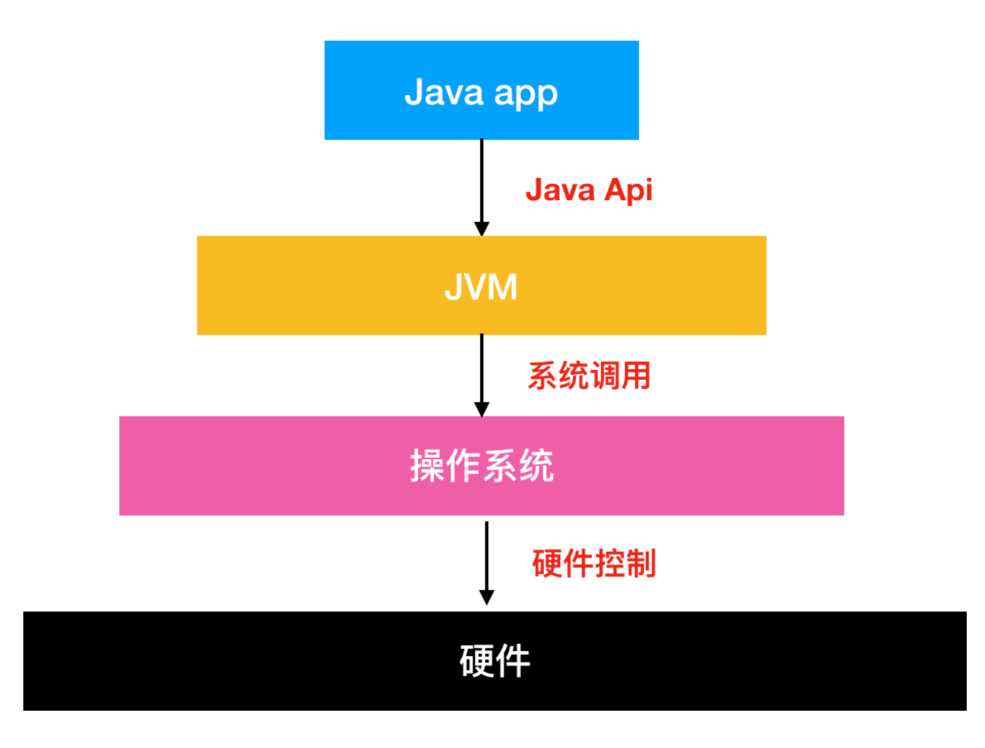
从设计的角度上来说，所有的语言都是设计用来把人类可读的代码转换为机器指令。动态语言是为了能 够让程序员提高编码效率，因此你可以使用更少的代码来实现功能。静态语言设计是用来让硬件执行的 更高效，因此需要程序员编写准确无误的代码，以此来让你的代码尽快的执行。从这个角度来说，静态 语言的执行效率要比动态语言高，速度更快。这是特点四。

-Java具有平台独立性和可移植性

Java有一句非常著名的口号：Write once, run anywhere，也就是一次编写、到处运行。为什么 Java能够吹出这种牛批的口号来？核心就是JVM。我们知道，计算机应用程序和硬件之间会屏蔽很多 细节，它们之间依靠操作系统完成调度和协调，大致的体系结构如下



那么加上Java应用、JVM的体系结构会变为如下



Java是跨平台的，已编译的Java程序可以在任何带有JVM的平台上运行。你可以在Windows平台 下编写代码，然后拿到Linux平台下运行，该如何实现呢？

首先你需要在应用中编写Java代码；

用Eclipse或者javac把Java代码编译为.class文件；

然后把你的.class文件打成.jar文件；

然后你的.jar文件就能够在Windows*、* Mac OS *X、*Linux系统下运行了。不同的操作系统有不同的 JVM实现，切换平台时，不需要再次编译你的Java代码了。这是特点五。

-Java能够容易实现多线程

Java是一门高级语言，高级语言会对用户屏蔽很多底层实现细节。比如Java是如何实现多线程的。从 操作系统的角度来说，实现多线程的方式主要有下面这几种

**在用户空间中实现多线程**

**在内核空间中实现多线程**

**在用户和内核空间中混合实现线程**

而我认为Java应该是在用户空间实现的多线程，内核是感知不到Java存在多线程机制的。这是特 点/ \。

• Java具有高性能我们编写的代码，经过javac编译器编译称为 字节码(bytecode)，经过JVM内嵌的解释器将字节码 转换为机器代码，这是解释执行，这种转换过程效率较低。但是部分JVM的实现比如Hotspot JVM 都提供了 JIT(Just-In-Time)编译器，也就是通常所说的动态编译器，JIT能够在运行时将热点代 码编译机器码，这种方式运行效率比较高，这是编译执行。所以Java不仅仅只是一种解释执行的语 言。这是特点七。

* Java语言具有健壮性

Java的强类型机制、异常处理、垃圾的自动收集等是Java程序健壮性的重要保证。这也是Java与C 语言的重要区别。这是特点八。

* Java很容易开发分布式项目

Java语言支持Internet应用的开发，Java中有net api，它提供了用于网络应用编程的类库，包括 URL、URLConnection、Socket、ServerSocket等。Java的RMI (远程方法激活)机制也是开发分布 式应用的重要手段。这是特点九。

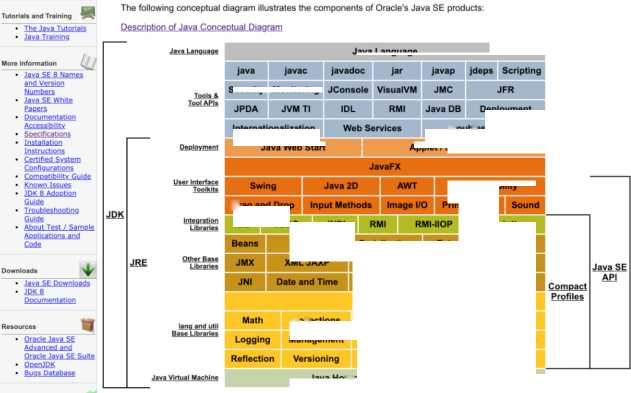
**Java**开发环境

**JDK**

JDK (Java Development Kit)称为Java开发包或Java开发工具，是一个编写Java的Applet 小程序和应用程序的程序开发环境。JDK是整个Java的核心，包括了 Java运行环境(Java Runtime Environment),—些 Java 工具和 Java 的核心类库(Java API)

Java Platform Standard Edition 8 Documentation

Oracle has two products that implement Java Platform Standard Edition (Java SE) 8: Java SE Development Kit (JDK) 8 and Java SE Runtime Environment (JRE) 8.

JDK 8 is a superset of JRE 8, and contains everything that is in JRE 8. plus tools such as the compilers and debuggers necessary for developing applets and applications. JRE 8 provides the libraries, the Java Virtual Machine (JVM), and other components to run applets and applications written in the Java programming language. Note that the JRE includes components not required by the Java SE specification, including both standard and non-standard Java components.

Security Monitoring

Scripting

Extension Mechanism

Override Mechanism

XML JAXP

Networking

Input/Output

Internationalization

Zip

Java HotSpot Client and Server VM

lang and util

Regular Expressions

Preferences API JAR

□rag and Drop

IDL JDBC JNDI

Internationalization

Java Web Start

Java Language

Deployment

Troubleshooting

Applet / Java Plug-in

Accessibility

Print Service

Security Serialization

Collections Ref Objects

Management Instrumentation Concurrency Utilities

我们可以认真研究一下这张图，它几乎包括了 Java中所有的概念，我使用的是jdk1.8，可以点进去 Description of Java Conceptual Diagram ,可以发现这里面包括了所有关于Java的描述

由Xnip截图

Documentation Conter

Java

Description of Java Conceptual Diagram

The Java conceptual diagram categorizes Java component technologies in levels. The following lists these levels, from top to bottom, and the technologies that are part of each one. Technologies that are part of a lower level serve as a foundation for those that are part of a higher level.

* Java Language
* Tools & Tool APIs

o java

o javac

o javadoc

o jar

o javao

o JPDA

o JConsole

o Java VisualVM

o Java DB

o Security

o inti

o RM1

o iDL

o Deploy

o Monitoring

o Troubleshoot

o Scripting

o JVM Tl

o Web Services

* Deployment

o Java Web Start

o Applet / Java Plug—in

* User Interface Toolkits

o JavaFX

o AWT

o Swing

o Java 2D

o Accessibility

o Drag and Drop

o Input Methods

o Image I/O

o Print Service

o Sound

* Integration Libraries

O iDL

o JDBC

o JND1

o RM1

o RMI-IIOP

o Scripting

* Other Base Libraries

o Beans

o Int'l Support

o Inout/Qutput

o JMX

o JN1

o Math

o Networking

o Override Mechanism

o Security

o Serialization

o Extension Mechanism

o XML JAXP

* lang and util Base Libraries

o lang and util

o Collections

o Concurrencv Utilities

o JAR

o Logging

o Management

o Preferences API

o Ref Objects

o Reflection

o Regular Expressions

o Versioning

° Zip

o Instrumentation

* Java Virtual Machine

o Java Hotspot Client and Server VM

The JDK consists of all levels.

The JRE consists of the following levels:

* Deployment
* User Interface Toolkits
* Integration Libraries
* Other Base Libraries
* lang and util Base Libraries
* Java Virtual Machine

The Java SE API consists of the following levels:

* User Interface Toolkits except JavaFX
* Integration Libraries
* Other Base Libraries
* lang and util Base Libraries

Compact Profiles consists of the following levels:

* Integration Libraries
* Other Base Libraries
* lang and util Base Libraries

Oracle提供了两种Java平台的实现，一种是我们上面说的JDK, Java开发标准工具包，一种是 JRE,叫做Java Runtime Environment, Java运行时环境。JDK的功能要比JRE全很多。

**JRE**

JRE是个运行环境，JDK是个开发环境。因此写Java程序的时候需要JDK，而运行Java程序的时候 就需要JRE。而JDK里面已经包含了JRE，因此只要安装了JDK,就可以编辑Java程序，也可以正常 运行Java程序。但由于JDK包含了许多与运行无关的内容，占用的空间较大，因此运行普通的Java 程序无须安装JDK，而只需要安装JRE即可。

**Java**开发环境配置

这个地方不再多说了，网上有很多教程配置的资料可供参考。

**Java**基本语法

在配置完Java开发环境，并下载Java开发工具（Eclipse、IDEA等）后，就可以写Java代码了，因 为本篇文章是从头梳理Java体系，所以有必要从基础的概念开始谈起。

数据类型

在Java中，数据类型只有四类八种

•整数型：byte、short、int、long

byte也就是字节，1 byte = 8 bits, byte的默认值是0 ；

short占用两个字节，也就是16位，1 short = 16 bits，它的默认值也是0 ；

int占用四个字节，也就是32位，1 int = 32 bits，默认值是0 ；

long占用八个字节，也就是64位，1 long = 64 bits,默认值是0L；

所以整数型的占用字节大小空间为long > int > short > byte

-浮点型

浮点型有两种数据类型：float和double

float是单精度浮点型，占用4位，1 float = 32 bits,默认值是0.0f；

double是双精度浮点型，占用8位，1 double = 64 bits，默认值是0.0d；

・字符型

字符型就是char, char类型是一个单一的16位Unicode字符，最小值是\u0000 （也就是0 ） 最大值是\uffff （即为65535） , char数据类型可以存储任何字符，例如char a = 'A'。

・布尔型

布尔型指的就是boolean, boolean只有两种值，true或者是false,只表示1位，默认值是false。

以上x位都指的是在内存中的占用。

|  |  |  |
| --- | --- | --- |
| **数据类型** | **内存** | **默认值** |
| byte | 8位 | 0 |
| short | 16位 | 0 |
| int | 32位 | 0 |
| long | 64位 | 0L |
| float | 32位 | O.Of |
| double | 64位 | O.Od |
| char | 16位 | \u0000 |
| boolean | 8位 | fals |

基础语法

•大小写敏感：Java是对大小写敏感的语言，例如Hello与hello是不同的，这其实就是Java的字 符串表示方式

•类名：对于所有的类来说，首字母应该大写，例如MyFirstClass

•包名：包名应该尽量保证小写，例如my.first.package

•方法名：方法名首字母需要小写，后面每个单词字母都需要大写，例如myFirstMethod()

运算符

运算符不只Java中有，其他语言也有运算符，运算符是一些特殊的符号，主要用于数学函数、一些类 型的赋值语句和逻辑比较方面，我们就以Java为例，来看一下运算符。

-赋值运算符

赋值运算符使用操作符=来表示，它的意思是把=号右边的值复制给左边，右边的值可以是任何常 数、变量或者表达式，但左边的值必须是一个明确的，已经定义的变量。比如int a = 4

但是对于对象来说，复制的不是对象的值，而是对象的引用，所以如果说将一个对象复制给另一个对 **象**，实际上是将**一个对象的引用赋值给另一个对象**。

-算数运算符

算数运算符就和数学中的数值计算差不多，主要有

|  |  |
| --- | --- |
| **运算符** | **说明** |
| + | 加 |
|  | 减 |
| \* | 乘 |
| *1* | 除 |
| *%* | 取余 |

算数运算符需要注意的就是优先级问题，当一个表达式中存在多个操作符时，操作符的优先级顺序就 决定了计算顺序，最简单的规则就是先乘除后加减，（）的优先级最高，没必要记住所有的优先级顺 序，不确定的直接用0就可以了。

•自增、自减运算符

这个就不文字解释了，解释不如直接看例子明白

1. int a = 5;
2. b = ++a;
3. c = a++;

•比较运算符

比较运算符用于程序中的变量之间，变量和自变量之间以及其他类型的信息之间的比较。

比较运算符的运算结果是boolean型。当运算符对应的关系成立时，运算的结果为true,否则为 false。比较运算符共有6个，通常作为判断的依据用于条件语句中。



-逻辑运算符

逻辑运算符主要有三种，与、或、非



下面是逻辑运算符对应的true/false符号表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | **b** | **a & b** | **a&&b** | **a | b** | **a||b** | **!a** | **a A b** |
| **true** | **true** | true | true | true | true | false | false |
| **true** | **false** | false | false | true | true | false | true |
| **false** | **true** | false | false | true | true | true | true |
| **false** | **false** | false | false | false | false | true | false |

-按位运算符 按位运算符用来操作整数基本类型中的每个比特位，也就是二进制位。按位操作符会对两个参数中对 应的位执行布尔代数运算，并最终生成一个结果。



如果进行比较的双方是数字的话，那么进行比较就会变为按位运算。

按位与：按位进行与运算（AND），两个操作数中位都为1,结果才为1,否则结果为0。需要首先把比 较双方转换成二进制再按每个位进行比较

按位或：按位进行或运算（OR）,两个位只要有一个为1,那么结果就是1,否则就为0。

按位非：按位进行异或运算（XOR）,如果位为0,结果是1,如果位为1,结果是0。

按位异或：按位进行取反运算（NOT），两个操作数的位中，相同则结果为0,不同则结果为1。

-移位运算符

移位运算符用来将操作数向某个方向（向左或者右）移动指定的二进制位数。

|  |  |
| --- | --- |
| **运算符** | **含义** |
| » | 右移运算符eg: 8 » 1 = 4 |
| « | 左移运算符eg: 9 « 2 = 36 |

-三元运算符

三元运算符是类似if...else...这种的操作符，语法为：**条件表达式？表达式1：表达式2**。问号 前面的位置是判断的条件，判断结果为布尔型，为true时调用表达式1,为false时调用表达式2。

**Java**执行控制流程

Java中的控制流程其实和C —样，在Java中，流程控制会涉及到包括**if-else**、**while**、**do-while**、 **for**、**return**、**break**以及选择语句switch。下面以此进行分析

条件语句

条件语句可根据不同的条件执行不同的语句。包括if条件语句与switch多分支语句。

**if**条件语句

if语句可以单独判断表达式的结果，表示表达的执行结果，例如

1. int a = 10;
2. if(a > 10){
3. return true;
4. }
5. return false;

**if...else**条件语句

if语句还可以与else连用，通常表现为**如果满足某种条件，就进行某种处理，否则就进行另一种处 理**。

1. int a = 10;
2. int b = 11;
3. if(a >= b){
4. System.out.println("a >= b");
5. }else{
6. System.out.println("a < b");
7. }

if后的0内的表达式必须是boolean型的。如果为true,则执行if后的复合语句；如果为false,则执 行else后的复合语句。

**if...else if**多分支语句

上面中的if...else是单分支和两个分支的判断，如果有多个判断条件，就需要使用**if...else if**

1. int x = 40;
2. if(x > 60) {
3. System .out. println("x 的值大于 60");
4. } else if (x > 30) {
5. System.out.println("x 的值大于 30 但小于 60");
6. } else if (x > 0) {
7. System.out.println("x 的值大于 0 但小于 30");
8. } else {
9. System.out.println("x 的值小于等于 0”);
10. }

**switch**多分支语句 一种比**if...else if**语句更优雅的方式是使用switch多分支语句，它的示例如下

1. switch (week) {
2. case 1:
3. System.out.println("Monday");
4. break;
5. case 2:
6. System.out.println("Tuesday");
7. break;
8. case 3:
9. System.out.println("Wednesday");
10. break;
11. case 4:
12. System.out.println("Thursday");
13. break;
14. case 5:
15. System.out.println("Friday");
16. break;
17. case 6:
18. System.out.println("Saturday");
19. break;
20. case 7:
21. System.out.println("Sunday");
22. break;
23. default:
24. System.out.println("No Else");
25. break;
26. }

循环语句

循环语句就是在满足一定的条件下反复执行某一表达式的操作，直到满足循环语句的要求。使用的循环

语句主要有 **for**、**do...while()**、**while ,**

**while**循环语句

while循环语句的循环方式为利用一个条件来控制是否要继续反复执行这个语句。while循环语句的格式 如下

1. while(布尔值){
2. 表达式

3}

它的含义是，当(布尔值)为true的时候，执行下面的表达式，布尔值为false的时候，结束循环，布尔 值其实也是一个表达式，比如

1. int a = 10;
2. while(a > 5){
3. a--;

4}

**do...while** 循环

while与do...while循环的唯一区别是do...while语句至少执行一次，即使第一次的表达式为false。而 在while循环中，如果第一次条件为false,那么其中的语句根本不会执行。在实际应用中，while要比 do...while应用的更广。它的一般形式如下

1. int b = 10;
2. // do・・・while循环语句
3. do {
4. System.out.println("b == " + b);
5. b\_\_;
6. } while(b == 1);

**for**循环语句

for循环是我们经常使用的循环方式，这种形式会在第一次迭代前进行初始化。它的形式如下

1 for(初始化；布尔表达式；步进){}

每次迭代前会测试布尔表达式。如果获得的结果是false，就会执行for语句后面的代码；每次循环结 束，会按照步进的值执行下一次循环。

逗号操作符

这里不可忽略的一个就是逗号操作符，Java里唯一用到逗号操作符的就是for循环控制语句。在表达式 的初始化部分，可以使用一系列的逗号分隔的语句；通过逗号操作符，可以在f。r语句内定义多个变 量，但它们必须具有相同的类型

1. for(int i = 1;j = i + 10;i < 5;i++, j = j \* 2){}

for-each 语句

在Java JDK 1.5中还引入了一种更加简洁的、方便对数组和集合进行遍历的方法，即for-each语 句，例子如下

1. int array[] = {7, 8, 9};

2

1. for (int arr : array) {
2. System.out.println(arr);

5}

跳转语句

Java语言中，有三种跳转语句：**break**、**continue和return**

**break**语句

break语句我们在switch中已经见到了，它是用于终止循环的操作，实际上break语句在for、while、 do-whil e循环语句中，用于强行退出当前循环，例如

1. for(int i = 0;i < 10;i++){
2. if(i == 5){
3. break;
4. }
5. }

**continue** 语句

continue也可以放在循环语句中，它与break语句具有相反的效果，它的作用是用于执行下一次循环， 而不是退出当前循环，还以上面的例子为主

1. for(int i = 0;i < 10;i++){

2

1. System.out.printl(" i = " + i );
2. if(i == 5){
3. System.out.printl("continue ...");
4. continue;
5. }
6. }

return语句

return语句可以从一个方法返回，并把控制权交给调用它的语句。

1. public void getName() {
2. return name;
3. }

面向对象

下面我们来探讨面向对象的思想，面向对象的思想已经逐步取代了过程化的思想---面向过程，Java是 面向对象的高级编程语言，面向对象语言具有如下特征

-面向对象是一种常见的思想，比较符合人们的思考习惯；

・面向对象可以将复杂的业务逻辑简单化，增强代码复用性；

・面向对象具有抽象、封装、继承、多态等特性。

面向对象的编程语言主要有：C++、Java、C#等。

所以必须熟悉面向对象的思想才能编写出Java程序。

类也是一种对象

现在我们来认识一个面向对象的新的概念---类，什么是类，它就相当于是一系列对象的抽象，就比如 书籍一样，类相当于是书的封面，大多数面向对象的语言都使用class来定义类，它告诉你它里面定 义的对象都是什么样的，我们一般使用下面来定义类

1. class ClassName {
2. // body;
3. }

代码段中涉及一个新的概念//，这个我们后面会说。上面，你声明了一个class类，现在，你就可 以使用new来创建这个对象

1. ClassName classname = new ClassName();

一般，类的命名遵循驼峰原则，它的定义如下

骆驼式命名法(Camel-Case)又称驼峰式命名法，是电脑程式编写时的一套命名规则(惯例)。 正如它的名称CamelCase所表示的那样，是指混合使用大小写字母来构成变量和函数的名字。 程序员们为了自己的代码能更容易的在同行之间交流，所以多采取统一的可读性比较好的命名方 式。

对象的创建

在Java中，万事万物都是对象。这句话相信你一定不陌生，尽管一切都看作是对象，但是你操纵的却 是一个对象的引用(reference)。在这里有一个很形象的比喻：你可以把车钥匙和车看作是一组对象 引用和对象的组合。当你想要开车的时候，你首先需要拿出车钥匙点击开锁的选项，停车时，你需要点 击加锁来锁车。车钥匙相当于就是引用，车就是对象，由车钥匙来驱动车的加锁和开锁。并且，即使没 有车的存在，车钥匙也是一个独立存在的实体，也就是说，你有一个对象引用，但你不一定需要一个对 象与之关联，也就是

1. Car carKey;

这里创建的只是引用，而并非对象，但是如果你想要使用S这个引用时，会返回一个异常，告诉你需要 一个对象来和这个引用进行关联。一种安全的做法是，在创建对象引用时同时把一个对象赋给它。

1 Car carKey = new Car();

在Java中，一旦创建了一个引用，就希望它能与一个新的对象进行关联，通常使用new操作符来实 现这一目的。new的意思是，给我一个新对象，如果你不想相亲，自己new 一个对象就好了。祝你 下辈子幸福。

属性和方法

类一个最基本的要素就是有属性和方法。

属性也被称为字段，它是类的重要组成部分，属性可以是任意类型的对象，也可以是基本数据类型。例 如下

1. class A{
2. int a;
3. Apple apple;
4. }

类中还应该包括方法，方法表示的是做某些事情的方式。方法其实就是函数，只不过Java习惯把函数 称为方法。这种叫法也体现了面向对象的概念。

方法的基本组成包括方法名称、参数、返回值和方法体，下面是它的示例

1. public int getResult(){
2. // ...
3. return 1;
4. }

其中，getResult就是方法名称、()里面表示方法接收的参数、return表示方法的返回值，注 意：方法的返回值必须和方法的参数类型保持一致。有一种特殊的参数类型void表示方法无返 回值。{}包含的代码段被称为方法体。

构造方法

在Java中，有一种特殊的方法被称为 构造方法，也被称为构造函数、构造器等。在Java中，通过 提供这个构造器，来确保每个对象都被初始化。构造方法只能在对象的创建时期调用一次，保证了对象 初始化的进行。构造方法比较特殊，它没有参数类型和返回值，它的名称要和类名保持一致，并且构造 方法可以有多个，下面是一个构造方法的示例

1. class Apple {

2

1. int sum;
2. String color;

5

1. public Apple(){}
2. public Apple(int sum){}
3. public Apple(String color){}
4. public Apple(int sum,String color){}

10

1. }

上面定义了一个Apple类，你会发现这个Apple类没有参数类型和返回值，并且有多个以Apple同名 的方法，而且各个Apple的参数列表都不一样，这其实是一种多态的体现，我们后面会说。在定义完成 构造方法后，我们就能够创建Apple对象了。

class createApple {

1

2

3

4

5

6

7

8

9

10

public static void main(String[] args) {

Apple apple1 = new Apple();

Apple apple2 = new Apple(1);

Apple apple3 = new Apple("red");

Apple apple4 = new Apple(2,"color");

}

}

如上面所示，我们定义了四个Apple对象，并调用了 Apple的四种不同的构造方法，其中，不加任何 参数的构造方法被称为默认的构造方法，也就是

1 Apple apple1 = new Apple();

如果类中没有定义任何构造方法，那么JVM会为你自动生成一个构造方法，如下

1. class Apple {

2

1. int sum;
2. String color;

5

1. }

7

1. class createApple {

9

1. public static void main(String[] args) {
2. Apple apple1 = new Apple();

12

1. }
2. }

上面代码不会发生编译错误，因为Apple对象包含了一个默认的构造方法。

默认的构造方法也被称为默认构造器或者无参构造器。

这里需要注意一点的是，即使JVM会为你默认添加一个无参的构造器，但是如果你手动定义了任何一 个构造方法，**JVM就不再为你提供默认的构造器，你必须手动指定，否则会出现编译错误。**

**public class** Apple {

**int sum;**

String **color;**

**public** Apple(**int** sum){}

**class** createApple {

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

Apple applel = **new** Appleby；  
 A

**Apple (int)** in **Apple** cannot be applied to 0

显示的错误是，必须提供Apple带有int参数的构造函数，而默认的无参构造函数没有被允许使用。

方法重载

在Java中一个很重要的概念是方法的重载，它是类名的不同表现形式。我们上面说到了构造函数，其 实构造函数也是重载的一种。另外一种就是方法的重载

1. public class Apple {

2

1. int sum;
2. String color;

5

1. public Apple(){}
2. public Apple(int sum){}

8

1. public int getApple(int num){
2. return 1;
3. }

12

1. public String getApple(String color){
2. return "color";
3. }

16

1. }

如上面所示，就有两种重载的方式，一种是Apple构造函数的重载，一种是getApple方法的重载。

但是这样就涉及到一个问题，要是有几个相同的名字，Java如何知道你调用的是哪个方法呢？这里记 住一点即可，**每个重载的方法都有独一无二的参数列表**。其中包括参数的类型、顺序、参数数量等，满 足一种一个因素就构成了重载的必要条件。

请记住下面重载的条件

•方法名称必须相同。

-参数列表必须不同(个数不同、或类型不同、参数类型排列顺序不同等)。

-方法的返回类型可以相同也可以不相同。

-仅仅返回类型不同不足以成为方法的重载。

-重载是发生在编译时的，因为编译器可以根据参数的类型来选择使用哪个方法。

方法的重写

方法的重写与重载虽然名字很相似，但却完全是不同的东西。方法重写的描述是对子类和父类之间 的。而重载指的是同一类中的。例如如下代码

1. class Fruit {

2

1. public void eat(){
2. System.out.printl('eat fruit');
3. }
4. }

7

1. class Apple extends Fruit{

9

1. @Override
2. public void eat(){
3. System.out.printl('eat apple');
4. }
5. }

上面这段代码描述的就是重写的代码，你可以看到，子类Apple中的方法和父类Fruit中的方法同名， 所以，我们能够推断出重写的原则

-重写的方法必须要和父类保持一致，包括返回**值类型，方法名,参数列**表也都一样。

•重写的方法可以使用@Override注解来标识

-子类中重写方法的访问权限不能低于父类中方法的访问权限。

初始化

类的初始化

上面我们创建出来了一个Car这个对象，其实在使用new关键字创建一个对象的时候，其实是调用了 这个对象无参数的构造方法进行的初始化，也就是如下这段代码

1. class Car{
2. public Car(){}
3. }

这个无参数的构造函数可以隐藏，由JVM自动添加。也就是说，构造函数能够确保类的初始化。

成员初始化

Java会尽量保证每个变量在使用前都会获得初始化，初始化涉及两种初始化。

-一种是编译器默认指定的字段初始化，基本数据类型的初始化

|  |  |
| --- | --- |
| **类型** | **初始值** |
| boolean | false |
| char | /uOOOO |
| byte | (byte)O |
| short | (short)O |
| int | 0 |
| long | OL |
| float | O.Of |
| double | O.Od |

一种是其他对象类型的初始化，String也是一种对象，对象的初始值都为null，其中也包括基 本类型的包装类。

-一种是指定数值的初始化，例如

1 int a = 11

也就是说，指定a的初始化值不是0，而是11。其他基本类型和对象类型也是一样的。

构造器初始化

可以利用构造器来对某些方法和某些动作进行初始化，确定初始值，例如

1. public class Counter{
2. int i;
3. public Counter(){
4. i = 11;
5. }
6. }

利用构造函数，能够把i的值初始化为11。

初始化顺序

首先先来看一下有哪些需要探讨的初始化顺序

•静态属性：static开头定义的属性

•静态方法块：static (｝包起来的代码块

•普通属性：非static定义的属性

•普通方法块：(｝包起来的代码块

•构造函数：类名相同的方法

•方法：普通方法

public class LifeCycle {

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

//静态属性

private static String staticField = getStaticField();

//静态方法块

static {

System.out.println(staticField);

System .out. println(”静态方法块初始化");

}

//普通属性

private String field = getField();

//普通方法块

(

System.out.println(field);

}

//构造函数

public LifeCycle() {

System .out. println("构造函数初始化");

}

public static String getStaticField() {

String statiFiled = "Static Field Initial"; return statiFiled;

}

public static String getField() {

String filed = "Field Initial";

return filed;

}

//主函数

public static void main(String[] argc) { new LifeCycle();

}

}

这段代码的执行结果就反应了它的初始化顺序

静态属性初始化静态方法块初始化普通属性初始化普通方法块初始化构造函数初始化

数组初始化 数组是相同类型的、用一个标识符名称封装到一起的一个对象序列或基本类型数据序列。数组是通过方 括号下标操作符 口 来定义使用。

一般数组是这么定义的

1. int[] al;

2

1. //或者

4

1. int a1[];

两种格式的含义是一样的。

•直接给每个元素赋值:int array[4] = {1,2,3,4};

•给一部分赋值，后面的都为0 : int array[4] = {1,2};

•由赋值参数个数决定数组的个数：int array[] = {1,2};

可变参数列表

Java中一种数组冷门的用法就是可变参数，可变参数的定义如下

1. public int add(int... numbers){
2. int sum = 0;
3. for(int num : numbers){
4. sum += num;
5. }
6. return sum;
7. }

然后，你可以使用下面这几种方式进行可变参数的调用

1. add(); //不传参数
2. add(1); //传递一个参数
3. add(2,1); //传递多个参数
4. add(new Integer]] {1, 3, 2}); // 传递数组

对象的销毁

虽然Java语言是基于C++的，但是它和C/C++ 一个重要的特征就是不需要手动管理对象的销毁工 作。在著名的一书《深入理解Java虚拟机》中提到一个观点

**I**Java与C++之间有一堵由内存动态分配和垃圾收集技术所围成的“高墙”，墙外面的人 想进去，墙里面的人却想出来。

在Java中，我们不再需要手动管理对象的销毁，它是由Java虚拟机进行管理和销毁的。虽然我们 不需要手动管理对象，但是你需要知道对象作用域这个概念。

对象作用域

J多数语言都有作用域(scope)这个概念。作用域决定了其内部定义的变量名的可见性和生命周期。在 C、C++和Java中，作用域通常由{}的位置来决定，例如

1. (
2. int *a =* 11;
3. {
4. int b = 12;
5. }
6. }

a变量会在两个{}作用域内有效，而b变量的值只能在它自己的{}内有效。

虽然存在作用域，但是不允许这样写

1. (
2. int x = 11;
3. {
4. int x = 12;
5. }

6}

这种写法在C/C++中是可以的，但是在Java中不允许这样写，因为Java设计者认为这样写会导致程 序混舌2

**this** 和 **super**

this和super都是Java中的关键字

this表示的当前对象，this可以调用方法、调用属性和指向对象本身。this在Java中的使用一般有三 种：指向当前对象

1. public class Apple {

2

1. int i = 0;

4

1. Apple eatApple(){
2. i++;
3. return this;
4. }

9

1. public static void main(String[] args) {
2. Apple apple = new Apple();
3. apple.eatApple().eatApple();
4. }
5. }

这段代码比较精妙，精妙在哪呢，我一个eatApple。方法竟然可以调用多次，你在后面还可以继续调 用，这就很神奇了，为啥呢？其实就是this在作祟了，我在eatApple方法中加了一个return this的返回值，也就是说哪个对象调用eatApple方法都能返回对象的自身。

this还可以修饰属性，最常见的就是在构造方法中使用this，如下所示

1. public class Apple {

2

1. private int num;

4

1. public Apple(int num){
2. this.num = num;
3. }

8

1. public static void main(String[] args) {
2. new Apple(10);
3. }
4. }

main方法中传递了一个int值为10的参数，它表示的就是苹果的数量，并把这个数量赋给了 num全 局变量。所以num的值现在就是10。

this还可以和构造函数一起使用，充当一个全局关键字的效果

1. public class Apple {

2

1. private int num;
2. private String color;

5

1. public Apple(int num){
2. this(num,”红色");
3. }

9

1. public Apple(String color){
2. this(1,color);
3. }

13

1. public Apple(int num, String color) {
2. this.num = num;
3. this.color = color;
4. }

18

1. }

你会发现上面这段代码使用的不是this,而是this(参数)。它相当于调用了其他构造方法，然后传递 参数进去。这里注意一点：this。必须放在构造方法的第一行，否则编译不通过

public class Apple {

private int num;

private String color;

public Apple(int num){ this (numf color:**"红色”);**

}

public Apple(String color)(  
this.color = color;  
this ( num: 1. color):

*}*

Call to 'this(< must be first statement in constructor body

public A

this

this

com. sendmessage ・ api. Apple *^Contract(pure = true)* public **Apple(**int num,

String color)

annotations: *@org. jet bra ins. annotations. Contractfpur e = true)*

如果你把this理解为指向自身的一个引用，那么super就是指向父类的一个引用。super关键字和this —样，

你可以使用super.对象 来引用父类的成员，如下

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

public class Fruit {

int num; String color;

public void eat(){

System.out.println("eat Fruit");

}

public class Apple extends Fruit{

@Override public void eat() {

super.num = 10;

System.out.println("eat " + num + " Apple");

你也可以使用

super（参数）来调用父类的构造函数，这里不再举例子了。

下面为你汇总了 this关键字和super关键字的比较。

|  |  |  |
| --- | --- | --- |
| **关键字** | **this** | **super** |
| 调用方式 | 调用本类中的属性、构造函数、方法 | 调用父类中属性、构造函数、方法 |
| 调用位置 | 构造函数第一行，其他可自行指定 | 构造函数第一行，其他可自行指定 |
| 调用次数 | —个构造函数只能调用一次 | 一个构造函数只能调用一次 |

访问控制权限

访问控制权限又称为封装，它是面向对象三大特性中的一种，我之前在学习过程中经常会忽略封装， 心想这不就是一个访问修饰符么，怎么就是三大特性的必要条件了？后来我才知道，**如果你信任的下属 对你隐瞒bug，你是根本不知道的**。

访问控制权限其实最核心就是一点：只对需要的类可见。

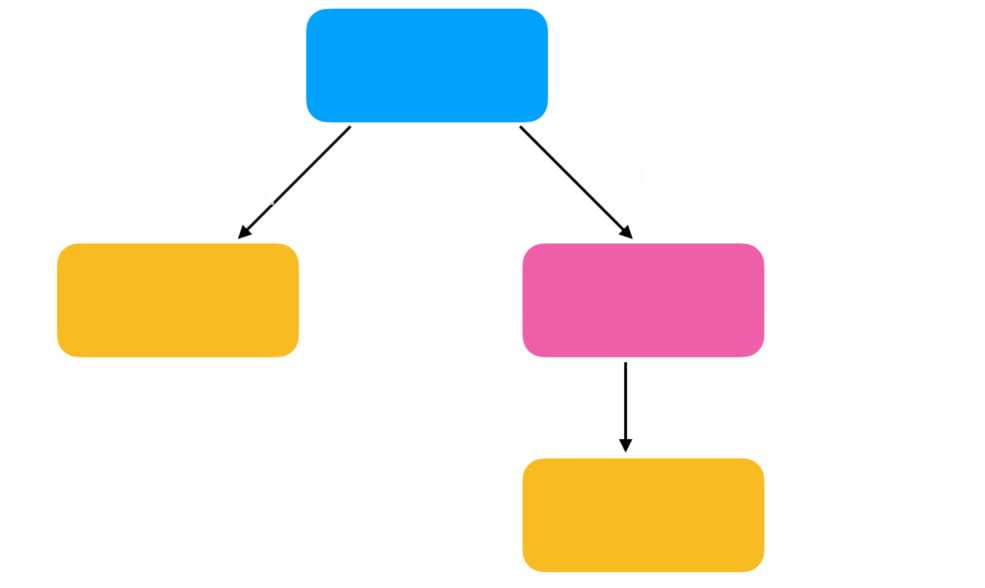
Java中成员的访问权限共有四种，分别是**public**、**protecteds default**、**private**，它们的可见性如下

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **private** | **default** | **protected** | **public** |
| 同一类 | **8** | **Q** | **8** | **a** |
| 同一包中的类 |  | **S** | **S** | **s** |
| 子类 |  |  | **8** | **a** |
| 其他包中的类 |  |  |  | **a** |

**Q**表示的是可以进行访问。

继承

继承是所有OOP(Object Oriented Programming)语言和Java语言都不可或缺的一部分。只要我 们创建了一个类，就隐式的继承自Object父类，只不过没有指定。如果你显示指定了父类，那么你 继承于父类，而你的父类继承于Object类。



隐式指定

隐式指定

extends显示指定

Object

Class Father

Class Son

Class Son

继承的关键字是extends，如上图所示，如果使用了 extends显示指定了继承，那么我们可以说 Father是父类，而Son是子类，用代码表示如下

1. class Father{}

2

1. class Son extends Father{}

继承双方拥有某种共性的特征

1. class Father{

2

1. public void feature（）{
2. System.out.println（” 父亲的特征"）;
3. }
4. }

7

1. class Son extends Father {
2. }

如果Son没有实现自己的方法的话，那么默认就是用的是父类的feature方法。如果子类实现了自 己的feature方法，那么就相当于是重写了父类的feature方法，这也是我们上面提到的重写了。

歹心'

多态指的是同一个行为具有多个不同表现形式。是指一个类实例（对象）的相同方法在不同情形下具有 不同表现形式。封装和继承是多态的基础，也就是说，多态只是一种表现形式而已。

如何实现多态？多态的实现具有三种充要条件

・继承

-重写父类方法

・父类引用指向子类对象

比如下面这段代码

public void eat(){

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

public class Fruit {

int num;

System.out.println("eat Fruit");

} public class Apple extends Fruit{

@Override

public void eat() { super.num = 10;

System.out.println("eat " + num + " Apple"); }

public static void main(String[] args) { Fruit fruit = new Apple(); fruit.eat();

}

}

你可以发现main方法中有一个很神奇的地方，Fruit fruit = new Apple。，Fruit类型的对象 竟然指向了 Apple对象的引用，这其实就是多态-> 父类引用指向子类对象，因为Apple继承于 Fruit,并且重写了 eat方法，所以能够表现出来多种状态的形式。

组合

组合其实不难理解，就是将对象引用置于新类中即可。组合也是一种提高类的复用性的一种方式。如果 你想让类具有更多的扩展功能，你需要记住一句话多**用组合，少用继承**。

1. public class SoccerPlayer {

2

1. private String name;
2. private Soccer soccer;

5

1. }

7

1. public class Soccer {

9

1. private String soccerName;
2. }

代码中SoccerPlayer引用了 Soccer类，通过引用Soccer类，来达到调用soccer中的属性和方法。

组合和继承是有区别的，它们的主要区别如下。

|  |  |  |
| --- | --- | --- |
| **特征** | **组合** | **继承** |
| 关系 | 组合是一种has- a的关系，可以理解为有一 个 | 继承是一种is - a的关系，可以理解为是一 个 |
| 耦合性 | 组合的双方是一种松耦合的关系 | 继承双方紧耦合 |
| 是否具有多 态 | 组合不具备多态和向上转型 | 继承是多态的基础，可以实现向上转型 |
| 时期 | 组合是运行期绑定 | 继承是编译期绑定 |

关于继承和组合孰优孰劣的争论没有结果，只要发挥各自的长处和优点即可，一般情况下，组合和继承 也是一对可以连用的好兄弟。

代理

除了继承和组合外，另外一种值得探讨的关系模型称为代理。代理的大致描述是，A想要调用B类 的方法，A不直接调用，A会在自己的类中创建一个B对象的代理，再由代理调用B的方法。例如如 下代码

1. public class Destination {

2

1. public void todo(){
2. System.out.println("control...");
3. }
4. }

7

8 public class Device {

private String name;

10

11

12

13

14

15

16

17

private Destination destination;

private DeviceController deviceController;

public void control(Destination destination){ destination.todo();

}

18 }

19

1. public class DeviceController {

21

1. private Device name;
2. private Destination destination;

24

1. public void control(Destination destination){
2. destination.todo();
3. }
4. }

向上转型 向上转型代表了父类与子类之间的关系，其实父类和子类之间不仅仅有向上转型，还有向下转型，它们 的转型后的范围不一样

* 向上转型：通过子类对象（小范围）转化为父类对象（大范围），这种转换是自动完成的，不用强制。
* 向下转型：通过父类对象（大范围）实例化子类对象（小范围），这种转换不是自动完成的，需要强制 指定。

**static**

static是Java中的关键字，它的意思是 静态的，static可以用来修饰成员变量和方法，static用在没 有创建对象的情况下调用方法/变量。

・用static声明的成员变量为静态成员变量，也成为类变量。类变量的生命周期和类相同，在整个应 用程序执行期间都有效。

1 static String name = "cxuan";

-使用static修饰的方法称为静态方法，静态方法能够直接使用**类名.方法名**进行调用。由于静态方 法不依赖于任何对象就可以直接访问，因此对于静态方法来说，是没有this关键字的，实例变量 都会有this关键字。在静态方法中不能访问类的非静态成员变量和非静态方法，

1. static void printMessage（）{
2. System.out.println（"cxuan is writing the article"）;
3. }

static除了修饰属性和方法外，还有静态代码块的功能，可用于类的初始化操作。进而提升程序的性 能。

1. public class StaicBlock {
2. static{
3. System.out.println（"I'm A static code block"）;
4. }
5. }

由于静态代码块随着类的加载而执行，因此，很多时候会将只需要进行一次的初始化操作放在static代 码块中进行。

**final**

final的意思是最后的、最终的，它可以修饰类、属性和方法。

・final修饰类时，表明这个类不能被继承。final类中的成员变量可以根据需要设为final，但是要注 意final类中的所有成员方法都会被隐式地指定为final方法。

・final修饰方法时，表明这个方法不能被任何子类重写，因此，如果只有在想明确禁止该方法在子 类中被覆盖的情况下才将方法设置为final。

・final修饰变量分为两种情况，一种是修饰基本数据类型，表示数据类型的值不能被修改；一种是 修饰引用类型，表示对其初始化之后便不能再让其指向另一个对象。

接口和抽象类

接口

接口相当于就是对外的一种约定和标准，这里拿操作系统举例子，为什么会有操作系统？就会为了屏蔽 软件的复杂性和硬件的简单性之间的差异，为软件提供统一的标准。

在Java语言中，接口是由interface关键字来表示的，比如我们可以向下面这样定义一个接口

1. public interface CxuanGoodJob {}

比如我们定义了一个CxuanGoodJob的接口，然后你就可以在其内部定义cxuan做的好的那些事情， 比如cxuan写的文章不错。

1. public interface CxuanGoodJob {

2

1. void writeWell();
2. }

这里隐含了一些接口的特征：

・interface接口是一个完全抽象的类，他不会提供任何方法的实现，只是会进行方法的定义。

•接口中只能使用两种访问修饰符，一种是public，它对整个项目可见；一种是default缺省 值，它只具有包访问权限。

-接口只提供方法的定义，接口没有实现，但是接口可以被其他类实现。也就是说，实现接口的类需 要提供方法的实现，实现接口使用implements关键字来表示，一个接口可以有多个实现。

1. class CXuanWriteWell implements CxuanGoodJob{

2

1. @Override
2. public void writeWell() {
3. System.out.println("Cxuan write Java is vary well");
4. }
5. }

-接口不能被实例化，所以接口中不能有任何构造方法，你定义构造方法编译会出错。

-接口的实现比如实现接口的全部方法，否则必须定义为抽象类，这就是我们下面要说的内容

抽象类

抽象类是一种抽象能力弱于接口的类，在Java中，抽象类使用abstract关键字来表示。如果把接 口形容为狗这个物种，那么抽象类可以说是毛发是白色、小体的品种，而实现类可以是具体的类，比如 说是博美、泰迪等。你可以像下面这样定义抽象类

1. public interface Dog {

2

1. void FurColor();

4

1. }

6

1. abstract class WhiteDog implements Dog{

8

1. public void FurColor(){
2. System.out.println("Fur is white");
3. }

12

1. abstract void SmallBody();
2. }

在抽象类中，具有如下特征

-如果一个类中有抽象方法，那么这个类一定是抽象类，也就是说，使用关键字abstract修饰的 方法一定是抽象方法，具有抽象方法的类一定是抽象类。实现类方法中只有方法具体的实现。

-抽象类中不一定只有抽象方法，抽象类中也可以有具体的方法，你可以自己去选择是否实现这些方 法。

-抽象类中的约束不像接口那么严格，你可以在抽象类中**定义构造方法、抽象方法、普通属性、方 法、静态属性和静态方法**

-抽象类和接口一样不能被实例化，实例化只能实例化具体的类

异常

异常是程序经常会出现的，发现错误的最佳时机是在编译阶段，也就是你试图在运行程序之前。但是， 在编译期间并不能找到所有的错误，有一些NullPointerException和

ClassNotFoundException异常在编译期找不到，这些异常是RuntimeException运行时异常，这些 异常往往在运行时才能被发现。

我们写Java程序经常会出现两种问题，一种是java.Iang.Exception，—种是java.Iang.Error，都用来 表示出现了异常情况，下面就针对这两种概念进行理解。

认识 **Exception**

Exception位于java.lang包下，它是一种顶级接口，继承于Throwable类，Exception类及 其子类都是Throwable的组成条件，是程序出现的合理情况。

在认识Exception之前，有必要先了解一下什么是Throwable

什么是**Throwable**

Throwable类是Java语言中所有错误(errors)和异常(exceptions)的父类。只有继承于 Throwable的类或者其子类才能够被抛出，还有一种方式是带有Java中的@throw注解的类也可以 抛出。

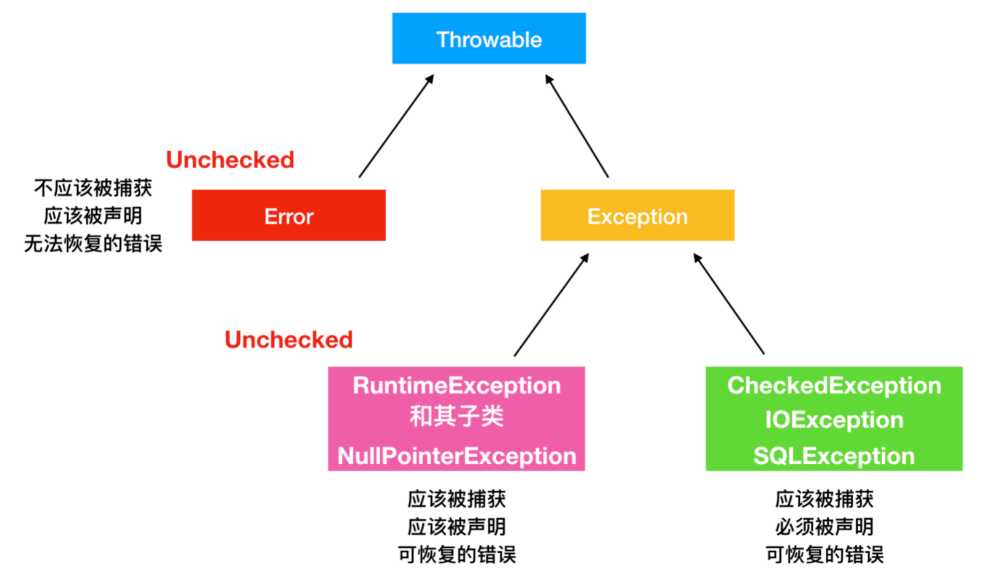
在**[Java](https://docs.oracle.com/javase/specs/jls/se9/html/jls-11.html%23jls-11.1.1)**[规范](https://docs.oracle.com/javase/specs/jls/se9/html/jls-11.html%23jls-11.1.1)中，对非受查异常和受查异常的定义是这样的：

The *unchecked exception classes* are the run-time exception classes and the error classes.

The *checked exception classes* are all exception classes other than the unchecked exception classes. That is, the checked exception classes are Throwable and all its subclasses other than RuntimeException and its subclasses and Error and its subclasses.

也就是说，除了 RuntimeException和其子类，以及error和其子类，其它的所有异常都是 checkedException

那么，按照这种逻辑关系，我们可以对Throwable及其子类进行归类分析



可以看到，Throwable位于异常和错误的最顶层，我们查看Throwable类中发现它的方法和属性有很 多，我们只讨论其中几个比较常用的

1. //返回抛出异常的详细信息
2. public string getMessage();
3. public string getLocalizedMessage();

4

1. //返回异常发生时的简要描述
2. public public String toString();

7

1. //打印异常信息到标准输出流上
2. public void printStackTrace();
3. public void printStackTrace(PrintStream s);
4. public void printStackTrace(PrintWriter s)

12

1. //记录栈帧的的当前状态
2. public synchronized Throwable fillInStackTrace();

此外，因为Throwable的父类也是Object，所以常用的方法还有继承其父类的getClass。和 getName()方法。

常见的**Exception**

下面我们回到Exception的探讨上来，现在你知道了 Exception的父类是Throwable,并且Exception 有两种异常，一种是RuntimeException ；—种是CheckedException，这两种异常都应该去捕 获。

下面列出了一些Java中常见的异常及其分类，这块面试官也可能让你举出几个常见的异常情况并将其 分类

RuntimeException

|  |  |  |
| --- | --- | --- |
| 序号 | 异常名称 | 异常描述 |
| 1 | ArraylndexOutOfBoundsException | 数组越界异常 |
| 2 | NullPointerException | 空指针异常 |
| 3 | HlegalArgumentException | 非法参数异常 |
| 4 | NegativeArraySizeException | 数组长度为负异常 |
| 5 | HlegalStateException | 非法状态异常 |
| 6 | ClassCastException | 类型转换异常 |

UncheckedException

|  |  |  |
| --- | --- | --- |
| 序号 | 异常名称 | 异常描述 |
| 1 | NoSuchFieldException | 表示该类没有指定名称抛出来的异常 |
| 2 | NoSuchMethod Exception | 表示该类没有指定方法抛出来的异常 |
| 3 | HlegalAccessException | 不允许访问某个类的异常 |
| 4 | ClassNotFoundException | 类没有找到抛出异常 |

与**Exception**有关的**Java**关键字

那么Java中是如何处理这些异常的呢？在Java中有这几个关键字**throws**、**throw**、**try**、**finally**、 **catch**下面我们分别来探讨一下

**throws** 和 **throw**

在Java中，异常也就是一个对象，它能够被程序员自定义抛出或者应用程序抛出，必须借助于 throws和throw语句来定义抛出异常。

throws和throw通常是成对出现的，例如

1. static void cacheException() throws Exception{

2

1. throw new Exception。；

4

1. }

throw语句用在方法体内，表示抛出异常，由方法体内的语句处理。throws语句用在方法声明后面， 表示再抛出异常，由该方法的调用者来处理。

throws主要是声明这个方法会抛出这种类型的异常，使它的调用者知道要捕获这个异常。throw是具 体向外抛异常的动作，所以它是抛出一个异常实例。

**try**、**finally**、**catch**

这三个关键字主要有下面几种组合方式try...catch、try...finally、try...catch...finally。

try...catch表示对某一段代码可能抛出异常进行的捕获，如下

1. static void cacheException() throws Exception{

2

1. try {
2. System.out.println("1");
3. }catch (Exception e){
4. e.printStackTrace();
5. }

8

1. }

try...finally表示对一段代码不管执行情况如何，都会走finally中的代码

1. static void cacheException() throws Exception{
2. for (int i = 0; i < 5; i++) {
3. System.out.println("enter: i=" + i);
4. try {
5. System.out.println("execute: i=" + i);
6. continue;
7. } finally {
8. System.out.println("leave: i=" + i);
9. }
10. }
11. }

try...catch...finally也是一样的，表示对异常捕获后，再走finally中的代码逻辑。

什么是**Error**

Error是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作 无关，而表示代码运行时JVM (Java虚拟机)出现的问题。这些错误是不可检查的，因为它们在应用 程序的控制和处理能力之夕卜，而且绝大多数是程序运行时不允许出现的状况，比如

OutOfMemoryError和StackOverflowError异常的出现会有几种情况，这里需要先介绍一下 Java内存模型JDK1.7。

运行时数据区域

方法区

**Method Area**

虚拟机栈

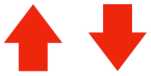
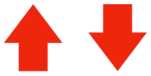
**VM Stack**

**本地方法栈  
Native Method  
Stack**

堆  
**Heap**

程序计数器

**Program Counter Register**



执行引擎

本地库接口

♦本地方法库

所有线程共享的数据区

线程隔离的数据区

其中包括两部分，**由所有线程共享的数据区和线程隔离的数据区**组成，在上面的Java内存模型中，只 **有程序计数**器是不会发生OutOfMemoryError情况的区域，程序计数器控制着计算机指令的分支、 循环、跳转、异常处理和线程恢复，并且程序计数器是每个线程私有的。

|什么是线程私有：表示的就是各条线程之间互不影响，独立存储的内存区域。

如果应用程序执行的是Java方法，那么这个计数器记录的就是虚拟机字节码指令的地址；如果正在执 行的是Native方法，这个计数器值则为空(Undefined)。

除了程序计数器外，其他区域：方法区(Method Area)*、*虚拟机栈(VM Stack)*、*本地方法栈 (Native Method Stack)和堆(Heap)都是可能发生 OutOfMemoryError 的区域。

-虚拟机栈：如果线程请求的栈深度大于虚拟机栈所允许的深度，将会出现 StackOverflowError异常；如果虚拟机动态扩展无法申请到足够的内存，将出现 OutOfMemoryError 。

-本地方法栈和虚拟机栈一样

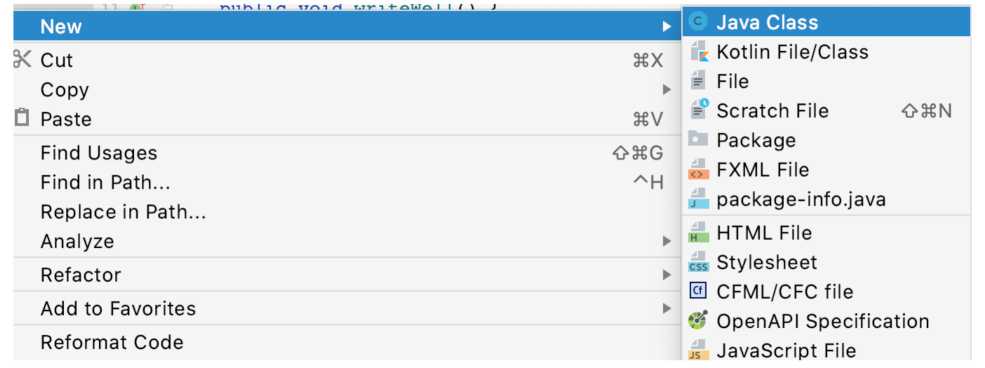
・堆：Java堆可以处于物理上不连续，逻辑上连续，就像我们的磁盘空间一样，如果堆中没有内存 完成实例分配，并且堆无法扩展时，将会抛出OutOfMemoryError。

•方法区：方法区无法满足内存分配需求时，将抛出。utOfMemoryError异常。

在Java中，你可以把异常理解为是一种能够提高你程序健壮性的机制，它能够让你在编写代码中注意 这些问题，也可以说，如果你写代码不会注意这些异常情况，你是无法成为一位硬核程序员的。

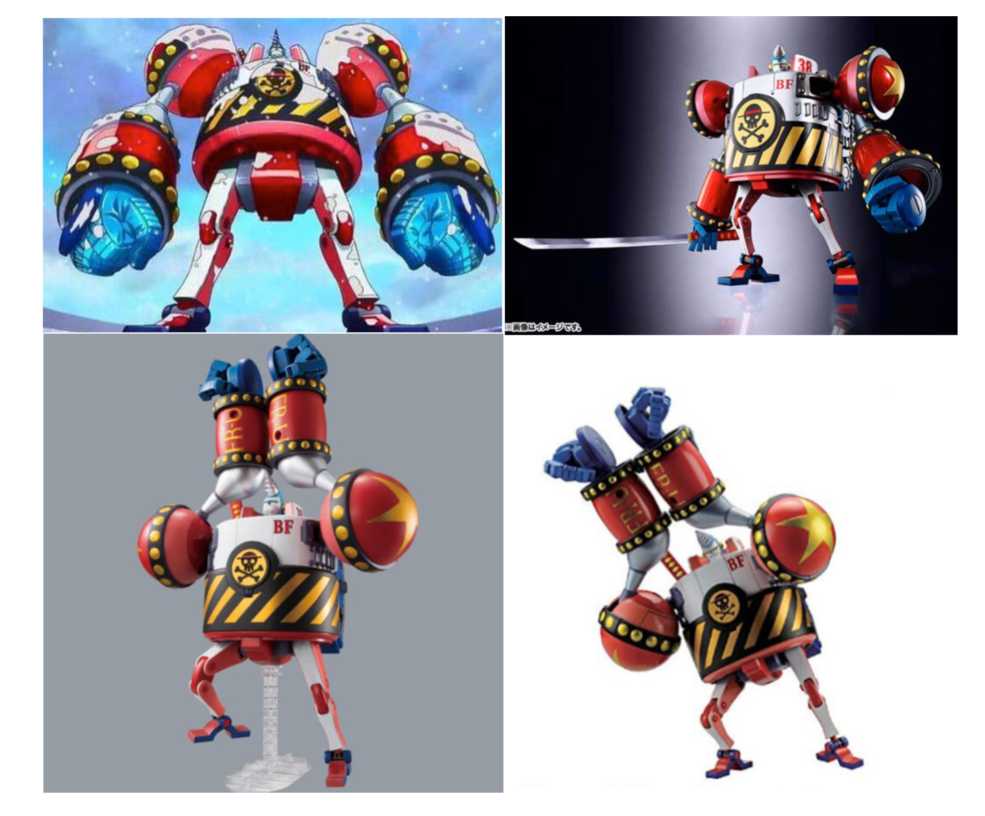
内部类

距今为止，我们了解的都是普通类的定义，那就是直接在IDEA中直接新建一个class。



新建完成后，你就会拥有一个class文件的定义，这种操作太简单了，时间长了就会枯燥，我们年轻人 多需要更新潮和骚气的写法，好吧，既然你提到了那就使用内部类吧，这是一种有用而且骚气的定义 类的方式，内部类的定义非常简单：**可以将一个类的定义放在另一个类的内部，这就是内部类**。

内部类是一种非常有用的特性，定义在类内部的类，持有外部类的引用，但却对其他外部类不可见，看 起来就像是一种隐藏代码的机制，就和弗兰奇将军似的，弗兰奇可以和弗兰奇将军进行通讯，但是外 面的敌人却无法直接攻击到弗兰奇本体。



下面我们就来聊一聊创建内部类的方式。

创建内部类

定义内部类非常简单，就是直接将一个类定义在外围类的里面，如下代码所示

1

2

3

4

5

6

7

8

9

10

11

public class OuterClass { private String name ; private int age;

class InnerClass{

public InnerClass(){ name = "cxuan"; age = 25;

}

在这段代码中，InnerClass就是。uterClass的一个内部类。也就是说，每个内部类都能独立地继承一 个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响。这 也是隐藏了内部实现细节。**内部类拥有外部类的访问权。**

内部类不仅仅能够定义在类的内部，还可以定义在方法和作用域内部，这种被称为局部内部类，除此 之外，还有匿名内部类、内部类可以实现Java中的 多重继承。下面是定义内部类的方式

-一个在方法中定义的类（局部内部类）

-一个定义在作用域内的类，这个作用域在方法的内部（成员内部类）

・一个实现了接口的匿名类（匿名内部类）

-一个匿名类，它扩展了非默认构造器的类

-一个匿名类，执行字段初始化操作

-一个匿名类，它通过实例初始化实现构造

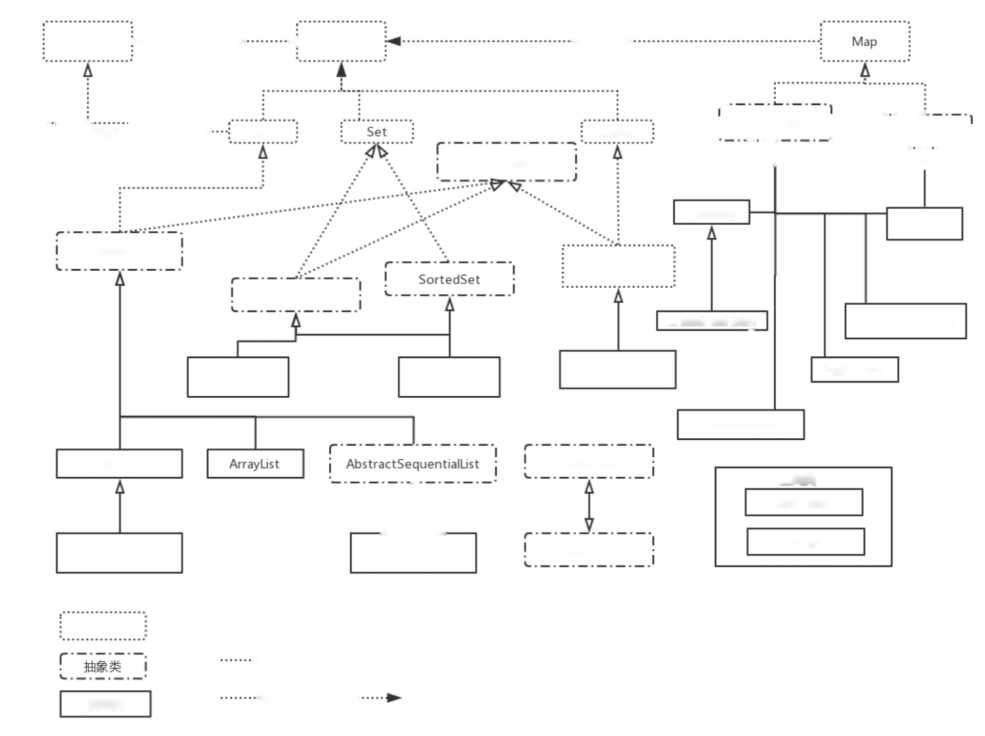
由于每个类都会产生一个-class文件，其中包含了如何创建该类型的对象的全部信息，那么，如何 表示内部类的信息呢？可以使用$来表示，比如**OuterClass$lnnerClass.class**。

集合

集合在我们的日常开发中所使用的次数简直太多了，你已经把它们都用的熟透了，但是作为一名合格的 程序员，你不仅要了解它的基本用法，你还要了解它的源码；存在即合理，你还要了解它是如何设计和 实现的，你还要了解它的衍生过程。

这篇博客就来详细介绍一下Collection这个庞大集合框架的家族体系和成员，让你了解它的设计与实 现。

**是时候祭出这张神图了**



Queue

Abstractcollection

HashTable

:.風施滩?L.X'Produces List

i" A^ractMap j p ^dMap j

'> —J- —

接口

类与抽象类的关系一A

抽象类^口接口的关系••••》

Iterable

Comparable

Comparator

V produces,

v Collection

,produces

AbstractList

AbstractSet

LinkedHashMap

AbstractQueue

工具类

Collections

Arrays

X

UnkedList

Stack

PriorityQueue

IdentityHashMap

HashSet

TreeSet

WeakHashMap

Vector

HashMap

TreeMap

理类

賴口和接口的关系

首先来介绍的就是列表爷爷辈儿的接口 - Iterator

**Iterable** 接口

实现此接口允许对象成为for-each循环的目标，也就是增强for循环，它是Java中的一种语法糖

1. List<Object> list = new ArrayList();
2. for (Object obj: list){}

除了实现此接口的对象外，数组也可以用for-each循环遍历，如下:

1. Object]] list = new Object[10];
2. for (Object obj: list){}

其他遍历方式

jdk 1.8之前Iterator只有iterator 一个方法，就是

1 Iterator<T> iterator。；

实现次接口的方法能够创建一个轻量级的迭代器，用于安全的遍历元素，移除元素，添加元素。这里面 涉及到一个fail-fast机制。

总之一点就是能创建迭代器进行元素的添加和删除的话，就尽量使用迭代器进行添加和删除。

也可以使用迭代器的方式进行遍历

for(Iterator it = coll.iterator(); it.hasNext(); ){

System. out. pri ntl n(it. next。)；

1. }

顶层接口

Collection是一个顶层接口，它主要用来定义集合的约定

List接口也是一个顶层接口，它继承了 C ollection接口，同时也是Array List, LinkedList等集合元素 的父类

Set接口位于与List接口同级的层次上，它同时也继承了 Collection接口。Set接口提供了额夕卜的规 定。它对add、equals, hashCode方法提供了额外的标准。

Queue是和List、Set接口并列的Collection的三大接口之一。Queue的设计用来在处理之前保持元 素的访问次序。除了 Collection基础的操作之外，队列提供了额外的插入，读取，检查操作。

SortedSet接口直接继承于Set接口，使用Comparable对元素进行自然排序或者使用Comparator在 创建时对元素提供定制的排序规则。set的迭代器将按升序元素顺序遍历集合。

Map是一个支持key-value存储的对象，Map不能包含重复的key,每个键最多映射一个值。这个接 口代替了 Dictionary类，Dictionary是一个抽象类而不是接口。

**ArrayList**

ArrayList是实现了 List接口的可扩容数组(动态数组)，它的内部是基于数组实现的。它的具体定义如 下：

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, java.io.Serializable {...}

* ArrayList可以实现所有可选择的列表操作，允许所有的元素，包括空值。ArrayList还提供了内部 存储list的方法，它能够完全替代Vector,只有一点例外，ArrayList不是线程安全的容器。
* ArrayList有一个容量的概念，这个数组的容量就是List用来存储元素的容量。

・ArrayList不是线程安全的容器，如果多个线程中至少有两个线程修改了 ArrayList的结构的话就会 导致线程安全问题，作为替代条件可以使用线程安全的List,应使用

Collections.synchronizedLi st o

List list = Collections.synchronizedList(new ArrayList(...

* ArrayList具有fail-fast快速失败机制，能够对ArrayList作出失败检测。当在迭代集合的过程中该 集合在结构上发生改变的时候，就有可能会发生fail-fast,即抛出

ConcurrentModificationException 异常。

**Vector**

Vector同ArrayList —样，都是基于数组实现的，只不过Vector是一个线程安全的容器，它对内部的每 个方法都简单粗暴的上锁，避免多线程引起的安全性问题，但是通常这种同步方式需要的开销比较大， 因此，访问元素的效率要远远低于ArrayListo

还有一点在于扩容上，ArrayList扩容后的数组长度会增加50%,而Vector的扩容长度后数组会增加一 倍。

**LinkedList** 类

LinkedList是一个双向链表，允许存储任何元素(包括null )o它的主要特性如下：

-LinkedList所有的操作都可以表现为双向性的，索引到链表的操作将遍历从头到尾，视哪个距离近 为遍历顺序。

-注意这个实现也不是线程安全的，如果多个线程并发访问链表，并且至少其中的一个线程修改了链 表的结构，那么这个链表必须进行外部加锁。或者使用

List list = Collections.synchronizedList(new LinkedList(・・・))

**Stack**

堆栈是我们常说的后入先出(吃了吐)的容器。它继承了 Vector类，提供了通常用的push和pop操 作，以及在栈顶的peek方法，测试stack是否为空的empty方法，和一个寻找与栈顶距离的search 方法。

第一次创建栈，不包含任何元素。一个更完善，可靠性更强的LIFO栈操作由Deque接口和他的实现 提供，应该优先使用这个类

Deque<Integer> stack = new ArrayDeque<Integer>()

**HashSet**

HashSet是Set接口的实现类，由哈希表支持(实际上HashSet是HashMap的一个实例)。它不能保证 集合的迭代顺序。这个类允许null元素。

-注意这个实现不是线程安全的。如果多线程并发访问HashSet,并且至少一个线程修改了set,必 须进行外部加锁。或者使用Collections.synchronizedSet()方法重写。

-这个实现支持fail-fast机制。

**TreeSet**

TreeSet是一个基于TreeMap的NavigableSet实现。这些元素使用他们的自然排序或者在创建时提供 的Comparator进行排序，具体取决于使用的构造函数。

•此实现为基本操作add,remove和contains提供了 log(n)的时间成本。

-注意这个实现不是线程安全的。如果多线程并发访问TreeSet,并且至少一个线程修改了 set,必 须进行外部加锁。或者使用

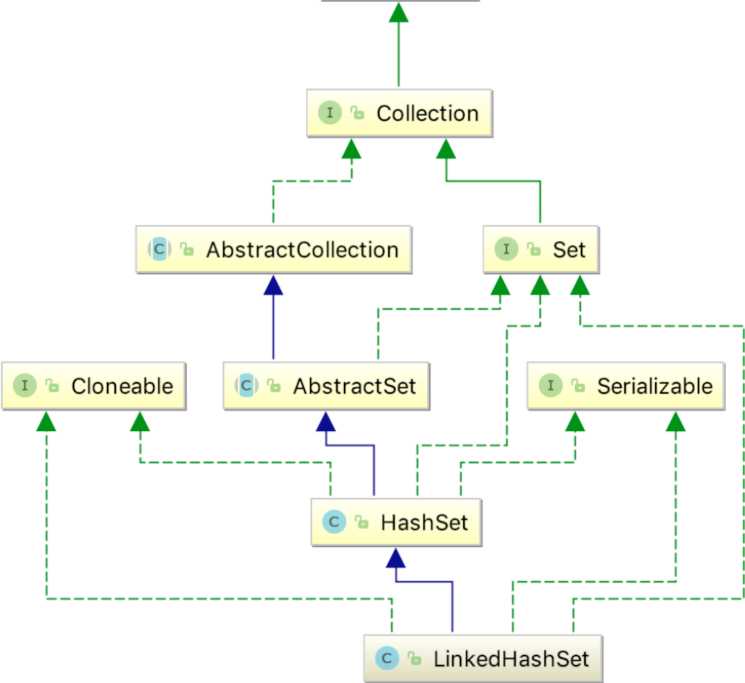
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...

-这个实现持有fail-fast机制。

**LinkedHashSet** 类

LinkedHashSet继承于Set,先来看一下LinkedHashSet的继承体系:

Iterable



LinkedHashSet是Set接口的Hash表和LinkedList的实现。这个实现不同于HashSet的是它维护着 一个贯穿所有条目的双向链表。此链表定义了元素插入集合的顺序。注意：如果元素重新插入，则插入 顺序不会受到影响。

• LinkedHashSet有两个影响其构成的参数：初始容量和加载因子。它们的定义与HashSet完全相 同。但请注意：对于LinkedHashSet,选择过高的初始容量值的开销要比HashSet小，因为 LinkedHashSet的迭代次数不受容量影响。

•注意LinkedHashSet也不是线程安全的，如果多线程同时访问LinkedHashSet,必须加锁，或者 通过使用

Collections.synchronizedSet

•该类也支持fail-fast机制

**PriorityQueue**

PriorityQueue是AbstractQueue的实现类，优先级队列的元素根据自然排序或者通过在构造函数时期 提供Comparator来排序，具体根据构造器判断。PriorityQueue不允许null元素。

•队列的头在某种意义上是指定顺序的最后一个元素。队列查找操作poll,remove,peek和element 访问队列头部元素。

-优先级队列是无限制的，但具有内部capacity,用于控制用于在队列中存储元素的数组大小。

-该类以及迭代器实现了 Collection, Iterator接口的所有可选方法。这个迭代器提供了 iterator()方法不能保证以任何特定顺序遍历优先级队列的元素。如果你需要有序遍历，考虑 使用 Arrays. sort(pq. toArray())。

-注意这个实现不是线程安全的，多线程不应该并发访问PriorityQueue实例如果有某个线程修改了

队列的话，使用线程安全的类PriorityBlockingQueue。

**HashMap**

HashMap是一个利用哈希表原理来存储元素的集合，并且允许空的key-value键值对。HashMap是非 线程安全的，也就是说在多线程的环境下，可能会存在问题，而Hashtable是线程安全的容器。

HashMap也支持fail-fast机制。HashMap的实例有两个参数影响其性能：初始容量和加载因子。可 以使用 Collections.synchronizedM叩(new HashM叩(...))来构造一个线程安全的 HashMap。

**TreeMap** 类

一个基于NavigableMap实现的红黑树。这个map根据key自然排序存储，或者通过Comparator进 行定制排序。

* TreeMap 为 containsKey,get,put 和remove方法提供了 log(n)的时间开销。

・注意这个实现不是线程安全的。如果多线程并发访问TreeMap,并且至少一个线程修改了 map, 必须进行外部加锁。这通常通过在自然封装集合的某个对象上进行同步来实现，或者使用

SortedMap m = Collections.synchronizedSortedMap(new T「eeMap(...))。

・这个实现持有fail-fast机制。

**LinkedHashMap** 类

LinkedHashMap是Map接口的哈希表和链表的实现。这个实现与HashMap不同之处在于它维护了一 个贯穿其所有条目的双向链表。这个链表定义了遍历顺序，通常是插入map中的顺序。

•它提供一个特殊的LinkedHashMap(int,float,boolean)构造器来创建LinkedHashMap,其遍历顺 序是其最后一次访问的顺序。

•可以重写removeEldestEntry(Map.Entry)方法，以便在将新映射添加到map时强制删除过期映射 的策略。

-这个类提供了所有可选择的map操作，并且允许null元素。由于维护链表的额外开销，性能可能 会低于HashMap,有一条除外：遍历 LinkedHashMap 中的 collection-views 需要与 map.size 成 正比，无论其容量如何。HashMap的迭代看起来开销更大，因为还要求时间与其容量成正比。

* LinkedHashMap有两个因素影响了它的构成：初始容量和加载因子。

-注意这个实现不是线程安全的。如果多线程并发访问LinkedHashMap,并且至少一个线程修改了 map,必须进行外部加锁。这通常通过在自然封装集合的某个对象上进行同步来实现Map m = Collections.synchronizedMap(new LinkedHashMap。..))。

•这个实现持有fail-fast机制。

**Hashtable** 类

Hashtable类实现了一个哈希表，能够将键映射到值。任何非空对象都可以用作键或值。

-此实现类支持fail-fast机制

-与新的集合实现不同，Hashtable是线程安全的。如果不需要线程安全的容器，推荐使用 HashMap,如果需要多线程高并发，推荐使用ConcurrentHashMap。

**IdentityHashMap** 类

IdentityHashMap是比较小众的Map实现了。

・这个类不是一个通用的Map实现！虽然这个类实现了 Map接口，但它故意违反了 Map的约定， 该约定要求在比较对象时使用equals方法，此类仅适用于需要引用相等语义的极少数情况。

•同HashMap, IdentityHashMap也是无序的，并且该类不是线程安全的，如果要使之线程安全， 可以调用 Collections.synchronizedMap（new IdentityHashM叩（...））方法来实现。

•支持fail-fast机制

**WeakHashMap** 类

WeakHashMap类基于哈希表的Map基础实现，带有弱键。WeakHashMap中的entry当不再使用时 还会自动移除。更准确的说，给定key的映射的存在将不会阻止key被垃圾收集器丢弃。

•基于map接口，是一种弱键相连，WeakHashMap里面的键会自动回收

・支持null值和null键。

* fast-fail 机制

-不允许重复

* WeakHashMap经常用作缓存

**Collections** 类

Collections不属于Java框架继承树上的内容，它属于单独的分支，Collections是一个包装类，它的作 用就是为集合框架提供某些功能实现，此类只包括静态方法操作或者返回collections。

同步包装

同步包装器将自动同步（线程安全性）添加到任意集合。六个核心集合接口 （Collection, Set, List, Map, SortedSet和SortedMap）中的每一个都有一个静态工厂方法。

Collection synchronizedCollection(Collection c);

public static

2 public static

public static

Set synchronizedSet(Set s);

List synchronizedList(List list);

static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);

public

public

SortedSet synchronizedSortedSet(SortedSet s);

public static

static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);

不可修改的包装

不可修改的包装器通过拦截修改集合的操作并抛出UnsupportedOperationException ,主要用在下 面两个情景：

-构建集合后使其不可变。在这种情况下，最好不要去获取返回collection的引用，这样有利于保证 不变性

-允许某些客户端以只读方式访问你的数据结构。你保留对返回的collection的引用，但分发对包 装器的引用。通过这种方式，客户可以查看但不能修改，同时保持完全访问权限。

这些方法是:

1

2

3

4

5

6

7

public static public static public static

public

Collection unmodifiableCollection(Collection<? extends T> c);

Set unmodifiableSet(Set<? extends T> s);

List unmodifiableList(List<? extends T> list);

static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m);

public static

public

V> m);

SortedSet unmodifiableSortedSet(SortedSet<? extends T> s);

static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends

线程安全的Collections

Java1.5并发包(java.util.concurrent)提供了线程安全的collections允许遍历的时候进行修 改，通过设计 iterator 为 fail-fast 并抛出 ConcurrentModificationException。一些实现类 是 CopyOnWriteArrayList , ConcurrentHashMap , CopyOnWriteArraySet

Collections 算法

此类包含用于集合框架算法的方法，例如二进制搜索，排序，重排，反向等。

集合实现类特征图

下图汇总了部分集合框架的主要实现类的特征图，让你能有清晰明了看出每个实现类之间的差异性

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **集合** | **排序** | **随机访问** | **key・value存储** | **重复元素** | **空元素** | **线程安全** |
| ArrayList | Y | Y | N | Y | Y | N |
| LinkedList | Y | N | N | Y | Y | N |
| HashSet | N | N | N | N | Y | N |
| TreeSet | Y | N | N | N | N | N |
| HashMap | N | Y | Y | N | Y | N |
| TreeMap | Y | Y | Y | N | N | N |
| Vector | Y | Y | N | Y | Y | Y |
| Hashtable | N | Y | Y | N | N | Y |
| ConcurrentHashMap | N | Y | Y | N | N | Y |
| Stack | Y | N | N | Y | Y | Y |
| CopyOnWriteArrayList | Y | Y | N | Y | Y | Y |

还有一种类型是关于强引用、弱引用、虚引用的文章，请参考

[https://mp.weixin.qq.eom/s/Zf IBpn2TBzTNv\_-G-zZxNg](https://mp.weixin.qq.com/s/ZflBpn2TBzTNv_-G-zZxNg)

泛形

在Jdk1.5中，提出了一种新的概念，那就是泛型，那么什么是泛型呢? 泛型其实就是一种参数化的集合，它限制了你添加进集合的类型。泛型的本质就是一种参数化类型。多 态也可以看作是泛型的机制。一个类继承了父类，那么就能通过它的父类找到对应的子类，但是不能通 过其他类来找到具体要找的这个类。泛型的设计之处就是希望对象或方法具有最广泛的表达能力。

下面来看一个例子说明没有泛型的用法

1. List arrayList = new ArrayList();
2. arrayList.add("cxuan");
3. arrayList.add(100);

4

1. for(int i = 0; i< arrayList.size();i++){
2. String item = (String)arrayList.get(i);
3. System.out.println("test === ", item);
4. }

这段程序不能正常运行，原因是Integer类型不能直接强制转换为String类型

1. java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

如果我们用泛型进行改写后，示例代码如下

1. List<String> arrayList = new ArrayList<String>();

2

1. arrayList.add(100);

这段代码在编译期间就会报错，编译器会在编译阶段就能够帮我们发现类似这样的问题。

泛型的使用

泛型的使用有多种方式，下面我们就来一起探讨一下。

用泛型表示类

泛型可以加到类上面，来表示这个类的类型

1. //此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型
2. public class GenericDemo<T>{
3. //value这个成员变量的类型为T,T的类型由外部指定
4. private T value;

5

1. public GenericDemo(T value) {
2. this.value = value;
3. }

9

1. public T getValue(){ //泛型方法getKey的返回值类型为T, T的类型由外部指定
2. return value;
3. }

13

1. public void setValue(T value){
2. this.value = value
3. }
4. }

用泛型表示接口

泛型接口与泛型类的定义及使用基本相同。

1. //定义一个泛型接口
2. public interface Generator<T> {
3. public T next();
4. }

一般泛型接口常用于生成器(generator)中，生成器相当于对象工厂，是一种专门用来创建对象的 类。

泛型方法

可以使用泛型来表示方法

1. public class GenericMethods {
2. public <T> void f(T x){
3. System.out.println(x.getClass().getName());
4. }
5. }

泛型通配符

List是泛型类，为了表示各种泛型List的父类，可以使用类型通配符，类型通配符使用问号(？)表 示，它的元素类型可以匹配任何类型。例如

1. public static void main(String[] args) {
2. List<String> name = new ArrayList<String>();
3. List<Integer> age = new ArrayList<Integer>();
4. List<Number> number = new ArrayList<Number>();
5. name.add("cxuan");
6. age.add(18);
7. number.add(314);
8. generic(name);
9. generic(age);
10. generic(number);
11. }

12

1. public static void generic(List<?> data) {
2. System.out.println("Test cxuan :" + data.get(0));
3. }

**上界通配符**：<? extends ClassType>该通配符为ClassType的所有子类型。它表示的是任何类型都是 Classfype类型的子类。

**下界通配符：**<? super ClassType>该通配符为ClassType的所有超类型。它表示的是任何类型的父 类都是 ClassType。

反射 反射是Java中一个非常重要同时也是一个高级特性，基本上Spring等一系列框架都是基于反射的思想 写成的。我们首先来认识一下什么反射。

**Java反射机制是在程序的运行过程中，对于任何一个类，都能够知道它的所有属性和方法；对于任意 一个对象，都能够知道调用它的任意属性和方法，这种动态获取信息以及动态调用对象方法的功能称为 java语言的反射机制。**（来源于百度百科）

Java反射机制主要提供了以下这几个功能

-在运行时判断任意一个对象所属的类

-在运行时构造任意一个类的对象

・在运行时判断任意一个类所有的成员变量和方法

-在运行时调用任意一个对象的方法

这么一看，反射就像是一个掌控全局的角色，不管你程序怎么运行，我都能够知道你这个类有哪些属性 和方法，你这个对象是由谁调用的，嗯，很屋。

在Java中，使用Java. lang, reflect包实现了反射机制。Java.lang.reflect所设计的类如下

c AccessibleObject

1. An notated ArrayType

* AnnotatedElement
* Annotated ParameterizedType
* AnnotatedType

1. AnnotatedTypeVariable

* AnnotatedWildcardType

J Array

*七* Constructor

c Executable

缶 Field

* GenericArrayType
* GenericDeclaration

c GenericSig nature Format Error

* InvocationHandler

c InvocationTargetException

c MalformedParameterizedTypeException

c MalformedParametersException

* Member

c Method

c Modifier

j package-info.java

J Parameter

* ParameterizedType

c Proxy

c ReflectAccess

c ReflectPermission

* Type
* TypeVariable

c UndeclaredThrowableException

c WeakCache

* WildcardType

下面是一个简单的反射类

1

2

3

4

5

6

7

8

9

10

11

12

13

public class Person {

public String name;// 姓名

public int age;// 年龄

public Person() {

super();

public Person(String name, int age) { super();

this.name = name;

this.age = age;

public String showInfo() { return "name=" + name + ", age=" + age;

}

}

public class Student extends Person implements Study {

public String className;// 班级

private String address;// 住址

public Student。{

super。；

}

public Student(String name, int age, String className, String address) { super(name, age);

this.className = className; this.address = address;

}

public Student(String className) { this.className = className;

}

public String toString() {

return ”姓名："+ name + "，年龄："+ age + "，班级："+ className + "，住址: + address;

}

public String getAddress() {

return address;

}

public void setAddress(String address) { this.address = address;

}

}

public class TestRelect {

public static void main(String[] args) { Class student = null;

try {

student = Class.forName("com.cxuan.reflection.Student");

} catch (ClassNotFoundException e) { e.printStackTrace();

}

//获取对象的所有公有属性。

Field[] fields = student.getFields();

for (Field f : fields) {

System.out.println(f);

}

System. out. println("

//获取对象所有属性，但不包含继承的。

Field[] declaredFields = student.getDeclaredFields();

for (Field df : declaredFields) {

System.out.println(df);

}

//获取对象的所有公共方法

Method]] methods = student.getMethods();

for (Method m : methods) {

System.out.println(m);

}

System. out. println(" ");

//获取对象所有方法，但不包含继承的

Method]] declaredMethods = student.getDeclaredMethods();

for (Method dm : declaredMethods) {

System.out.println(dm);

}

//获取对象所有的公共构造方法

Constructor]] constructors = student.getConstructors();

for (Constructor c : constructors) {

System.out.println(c);

}

System. out. println(" ");

//获取对象所有的构造方法

Constructor]] declaredConstructors = student.getDeclaredConstructors(); for (Constructor dc : declaredConstructors) {

System.out.println(dc);

}

Class c = Class.forName("com.cxuan.reflection.Student");

Student stul = (Student) c.newInstance();

//第一种方法，实例化默认构造方法，调用set赋值

stul. setAddress("河北石家庄");

System.out.println(stul);

//第二种方法 取得全部的构造函数 使用构造函数赋值

Constructor<Student> constructor = c.getConstructor(String.class, int.class,

String.class, String.class);

Student student2 = (Student) constructor.newInstance("cxuan", 24,”六 班”，"石家庄");

System.out.println(student2);

/\*\*

\*獲取方法并执行方法

\*/

Method show = c. getMethod( "showinfo"); //获取 showinfo ()方法

Object object = show.invoke(stu2);//调用 s howinfo ()方法

116

1. }
2. }

有一些是比较常用的，有一些是我至今都没见过怎么用的，下面进行一个归类。

与Java反射有关的类主要有

**Class** 类

在Java中，你每定义一个java class实体都会产生一个Class对象。也就是说，当我们编写一个类， 编译完成后，在生成的-class文件中，就会产生一个Class对象，这个Class对象用于表示这个类 的类型信息。Class中没有公共的构造器，也就是说Class对象不能被实例化。下面来简单看一下 Class类都包括了哪些方法

toStringO

public String toStringO {

return (islnterface() ? "interface " : (isPrimitive() ? "" : "class "))

+ getName();

1. }

toStringO方法能够将对象转换为字符串，toStringO首先会判断Class类型是否是接口类型，也就是 说，普通类和接口都能够用Class对象来表示，然后再判断是否是基本数据类型，这里判断的都是基本 数据类型和包装类，还有void类型。

所有的类型如下

* java.lang.Boolean :代表boolean数据类型的包装类
* java.lang.Character:代表char数据类型的包装类
* java.lang.Byte:代表byte数据类型的包装类
* java.Iang.Short:代表short数据类型的包装类
* java.lang.Integer:代表int数据类型的包装类
* java.lang.Long:代表long数据类型的包装类
* java.lang.Float:代表float数据类型的包装类
* java.lang.Double:代表double数据类型的包装类
* java.lang.Void:代表void数据类型的包装类

然后是getNameC)方法，这个方法返回类的全限定名称。

•如果是引用类型，比如 String.class.getNameQ -> java. lang. String

•如果是基本数据类型，byte.class.getNameO -> byte

•如果是数组类型，new Object[3]).getClassQ.getNameQ -> [L java. lang.Object

toGenericStringO

这个方法会返回类的全限定名称，而且包括类的修饰符和类型参数信息。

forNameQ

根据类名获得一个Class对象的引用，这个方法会使类对象进行初始化。

例如 Class t = Class. forName(" java. lang.Thread")就能够初始化一个 Thread 线程对象

在Java中，一共有三种获取类实例的方式

* Class.forName(java.lang.Thread)
* Thread.class
* thread.getClassQ

newInstanceO

创建一个类的实例，代表着这个类的对象。上面forNameQ方法对类进行初始化，newlnstance方法对 类进行实例化。

getClassLoaderO

获取类加载器对象。

getTypeParametersO

按照声明的顺序获取对象的参数类型信息。

getPackageO

返回类的包

getlnterfacesO

获得当前类实现的类或是接口，可能是有多个，所以返回的是Class数组。

Cast

把对象转换成代表类或是接口的对象

asSubclass(Class clazz)

把传递的类的对象转换成代表其子类的对象

getClassesO

返回一个数组，数组中包含该类中所有公共类和接口类的对象

getDeclaredClassesO

返回一个数组，数组中包含该类中所有类和接口类的对象

getSimpleNameO

获得类的名字

getFieldsO

获得所有公有的属性对象

getField(String name)

获得某个公有的属性对象

getDeclaredField(String name)

获得某个属性对象 **getDeclaredFieldsQ** 获得所有属性对象

**getAnnotation(Class annotationClass)**

返回该类中与参数类型匹配的公有注解对象

**getAnnotationsO**

返回该类所有的公有注解对象

**getDeclaredAnnotation(Class annotationClass)**

返回该类中与参数类型匹配的所有注解对象

**getDeclaredAnnotations()**

返回该类所有的注解对象

**getConstructor(Class...<?> parameterTypes)**

获得该类中与参数类型匹配的公有构造方法

**getConstructorsO**

获得该类的所有公有构造方法

**getDeclaredConstructor(Class...<?> parameterTypes)**

获得该类中与参数类型匹配的构造方法

**getDeclaredConstructors()**

获得该类所有构造方法

**getMethod(String name, Class...<?> parameterTypes)**

获得该类某个公有的方法

**getMethods()**

获得该类所有公有的方法

**getDeclaredMethod(String name, Class...<?> parameterTypes)**

获得该类某个方法

**getDeclaredMethods()**

获得该类所有方法

**Field** 类

Field类提供类或接口中单独字段的信息，以及对单独字段的动态访问。 这里就不再对具体的方法进行介绍了，读者有兴趣可以参考官方API 这里只介绍几个常用的方法

**equals(Object obj)**

属性与ob j相等则返回true

get(Object obj)

获得obj中对应的属性值

set(Object obj, Object value)

设置obj中对应属性值

**Method** 类

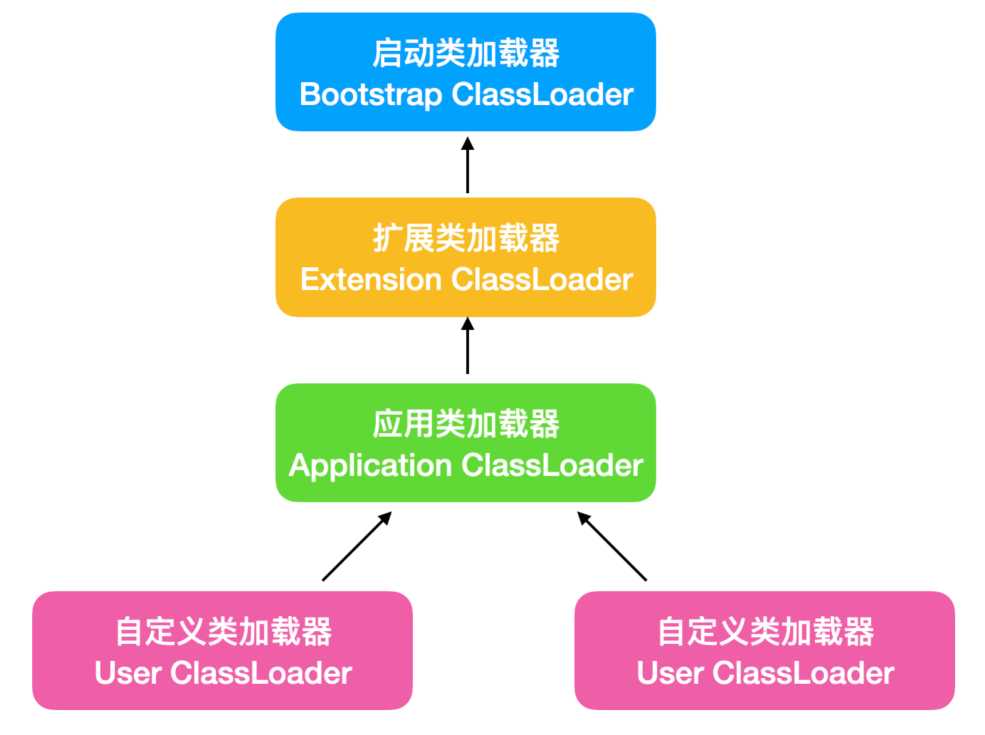
invoke(Object obj, Object... args)

传递object对象及参数调用该对象对应的方法

**ClassLoader** 类

反射中，还有一个非常重要的类就是ClassLoader类，类装载器是用来把类(class)装载进JVM 的。ClassLoader使用的是双亲委托模型来搜索加载类的，这个模型也就是双亲委派模型。

ClassLoader的类继承图如下



枚举 枚举可能是我们使用次数比较少的特性，在Java中，枚举使用enum关键字来表示，枚举其实是一 项非常有用的特性，你可以把它理解为具有特定性质的类。enum不仅仅Java有，C和C++也有枚举 的概念。下面是一个枚举的例子。

1. public enum Family {

2

1. FATHER,
2. MOTHER,
3. SON,
4. Daughter;

7

1. }

上面我们创建了一个Family的枚举类，它具有4个值，由于枚举类型都是常量，所以都用大写字母 来表示。那么enum创建出来了，该如何引用呢？

1. public class EnumUse {

2

1. public static void main(String[] args) {
2. Family s = Family.FATHER;
3. }

6}

枚举特性

enum枚举这个类比较有意思，当你创建完enum后，编译器会自动为你的enum添加toString() 方法，能够让你方便的显示enum实例的具体名字是什么。除了 toString。方法外，编译器还会添加 ordinal。方法，这个方法用来表示enum常量的声明顺序，以及values。方法显示顺序的值。

1. public static void main(String[] args) {

2

1. for(Family family : Family.values()){
2. System.out.println(family + ", ordinal" + family.ordinal());
3. }

6}

enum可以进行静态导入包，静态导入包可以做到不用输入枚举类名.常量，可以直接使用常量，神奇 吗？使用ennum和static关键字可以做到静态导入包

import static com, sendmessage ・ api ・ Family ・ \*;

public class EnumUse (

public static void main(String[] args) {

Family family = *MOTHER;\*

*}*

}

**Enum**

m t Enum(String, int)

m clone(): Object 个Object

m)% compareTo(E): int

m) % equals(Object): boolean ^Object m finalize(): void 个Object

getDeclaringClass(): Class<E>

m) % hashCode(): int 个Object

m}也 name(): String

m) ■ ordinal(): int

readObject(Objectinputstream): void readObjectNoData(): void

m任 toStringO： String TObject

m) % valueOf(Class<T>, String): T

f fl name: String

f A ordinal: int

上面代码导入的是Family中所有的常量，也可以单独指定常量。

枚举和普通类一样

枚举就和普通类一样，除了枚举中能够方便快捷的定义常量，我们日常开发使用的public static final xxx其实都可以用枚举来定义。在枚举中也能够定义属性和方法，千万不要把它看作是异类， 它和万千的类一样。

1. public enum OrdinalEnum {

2

1. WEST("live in west"),

EAST("live in east"),

SOUTH("live in south"),

NORTH("live in north");

String description;

OrdinalEnum(String description){ this.description = description;

}

public String getDescription() {

return description;

}

public void setDescription(String description) { this.description = description;

}

public static void main(String[] args) { for(OrdinalEnum ordinalEnum : OrdinalEnum.values()){ System.out.println(ordinalEnum.getDescription());

}

}

}

switch可以和enum —起连用，来构造一个小型的状态转换机。

enum Signal {

GREEN, YELLOW, RED

}

public class TrafficLight {

Signal color = Signal.RED;

public void change() {

switch (color) {

case RED:

color = Signal.GREEN;

break;

case YELLOW:

color = Signal.RED;

break;

case GREEN:

color = Signal.YELLOW;

break;

}

}

是不是代码顿时觉得优雅整洁了些许呢?

枚举神秘之处 在Java中，万事万物都是对象，enum虽然是个关键字，但是它却隐式的继承于Enum类。我们来看 一下Enum类，此类位于java.lang包下，可以自动引用。

**▼Enum**

m Enum(String, int)

m clone(): Object 个Object

mE compareTo(E): int

m} % equals(Object): boolean 个0bject m finalize(): void 个Object getDeclaringClass(): Class<E>

m) % hashCode(): int 个Object

m) % name(): String

m) ordinalf): int

readObject(ObjectlnputStream): void readObjectNoData(): void mE toStringO： String 个Object

valueOf(Class<T>, String): T

f A name: String

f ordinal: int

此类的属性和方法都比较少。你会发现这个类中没有我们的values方法。前面刚说到，values。方 法是你使用枚举时被编译器添加进来的static方法。**可以使用反射来验证一下**。

除此之外，enum还和Class类有交集，在Class类中有三个关于Enum的方法

m ■ getEnumConstants(): T[]

m o getEnumConstantsShared(): T[]

m ■ isEnum(): boolean

前面两个方法用于获取enum常量，isEnum用于判断是否是枚举类型的。

枚举类

除了 Enum夕卜，还需要知道两个关于枚举的工具类，一个是EnumSet，一个是EnumMap

**EnumSet 和 EnumMap**

EnumSet是JDK1.5引入的，EnumSet的设计充分考虑到了速度因素，使用EnumSet可以作为Enum 的替代者，因为它的效率比较高。

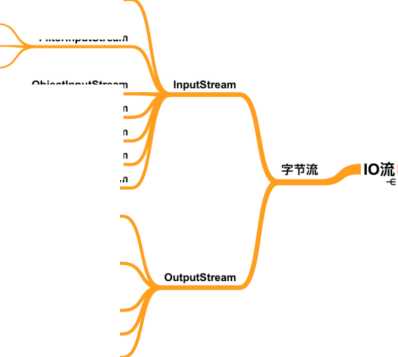
EnumMap是一种特殊的Map,它要求其中的key键值是来自一个enum。因为EnumMap速度也很 快，我们可以使用EnumMap作为key的快速查找。

总的来说，枚举的使用不是很复杂，它也是Java中很小的一块功能，但有时却能够因为这一个小技 巧，能够让你的代码变得优雅和整洁。

**I/O**

创建一个良好的I/O程序是非常复杂的。JDK开发人员编写了大量的类只为了能够创建一个良好的工具 包，想必编写I/O工具包很费劲吧。

IO类设计出来，肯定是为了解决IO相关操作的，最常见的I/O读写就是网络、磁盘等。在Java中, 对文件的操作是一个典型的I/O操作。下面我们就对I/O进行一个分类。



Fileinputstream

BufferedlnputStream

DatalnputStream

Filterinputstream

PushbaklnputStream

Objectinputstream

PipedlnputStream

Sequenceinputstream

StringBufferlnputStream

ByteArraylnputStream

FileOutputStream

BufferedOutputStream

FilterOutputStream

ObjectOutputStream

PipedOutputStream

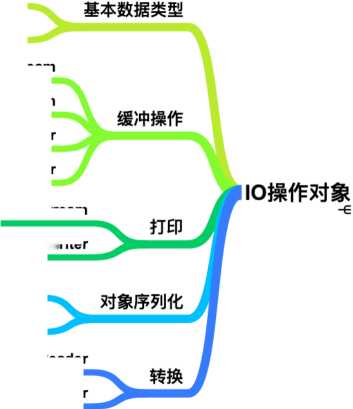
ByteArrayOutputStream

DataOutputStream

Printstream

公号回复10获取思维导图

I/O还可以根据操作对象来进行区分：主要分为



DatalnputStream

BufferedlnputStream

DataOutputStream

BufferedOutputStream

BufferedReader

BufferedWriter

Objectinputstream

ObjectOutputStream

InputStreamReader

OutputStreamWriter

Printstream

PrintWriter

Fileinputstream

FileOutputStream

FileReader

FileWriter

字节数组

字符数组

ByteArrayOutputStream

CharArray Writer

ByteArraylnputStream

CharArrayReader

PipedlnputStream

PipedReader

PipedWriter

PipedOutputStream

除此之外，I/O中还有其他比较重要的类

**File** 类

File类是对文件系统中文件以及文件夹进行操作的类，可以通过面向对象的思想操作文件和文件夹，是 不是很神奇？

文件创建操作如下，主要涉及**文件创建**、删**除文件、获取文件描述符等**

1. class FileDemo{
2. public static void main(String[] args) {
3. File file = new File("D:\\file.txt");
4. try{
5. f.createNewFile(); // 创建一个文件

6

1. // File类的两个常量
2. //路径分隔符(与系统有关的)<windows里面是；linux里面是：>
3. System.out.println(File.pathSeparator); // ;
4. //与系统有关的路径名称分隔符<windows里面是\ linux里面是/ >
5. System.out.println(File.separator); // \

12

1. //删除文件
2. /\*
3. File file = new File(fileName);
4. if(f.exists()){
5. f.delete();
6. }else{
7. System.out.println(” 文件不存在”)；
8. }
9. \*/

22

23

1. }catch (Exception e) {
2. e.printStackTrace();
3. }
4. }
5. }

也可以对文件夹进行操作

1. class FileDemo{
2. public static void main(String[] args) {
3. String fileName = "D:"+ File.separator + "filepackage";
4. File file = new File(fileName);
5. f.mkdir();

6

1. //列出所有文件
2. /\*
3. String]] str = file.list();
4. for (int i = 0; i < str.length; i++) {
5. System.out.println(str[i]);
6. }
7. \*/
8. //使用file.listFiles();列出所有文件，包括隐藏文件

16

1. //使用file.isDirectory()判断指定路径是否是目录
2. }
3. }

上面只是举出来了两个简单的示例，实际上，还有一些其他对文件的操作没有使用。比如创建文件，就 可以使用三种方式来创建

1. File(String directoryPath);
2. File(String directoryPath, String filename);
3. File(File dirObj, String filename);

directoryPath是文件的路径名，filename是文件名，dirObj是一个File对象。例如

1. File file = new File("D:\\java\\file1.txt"); //双\\是转义
2. System.out.println(file);
3. File file2 = new File("D:\\java" ,"file2.txt");//父路径、子路径一可以适用于多个文件的!
4. System.out.println(file2);
5. File parent = new File("D:\\java");
6. File file3 = new File(parent,"file3.txt");//File 类的父路径、子路径
7. System.out.println(file3);

现在对File类进行总结

|  |  |
| --- | --- |
| **方法** | **方法描述** |
| **public String getNameO** | 返回此路径名表示的文件或目录的名称。 |
| **public String getParentO** | 返回此路径名的父路径名的字符串。 |
| **public String getPath()** | 将此路径名转换为一个路径名字符串。 |
| **public boolean isFileO** | 测试此路径名表示的文件是否是一个标准文件 |
| **public boolean equals(Object obj)** | 测试此路径名与给定对象是否相等。 |
| **public StringQ listQ** | 返回由此路径名所表示的目录中文件和目录的名称所组成字符串数 组。 |
| **public boolean mkdirQ** | 创建此路径名指定的目录。 |
| **public String getAbsolutePathO** | 返回路径名的绝对路径名字符串。 |
| **public boolean existsQ** | 测试此路径名表示的文件或目录是否存在。 |

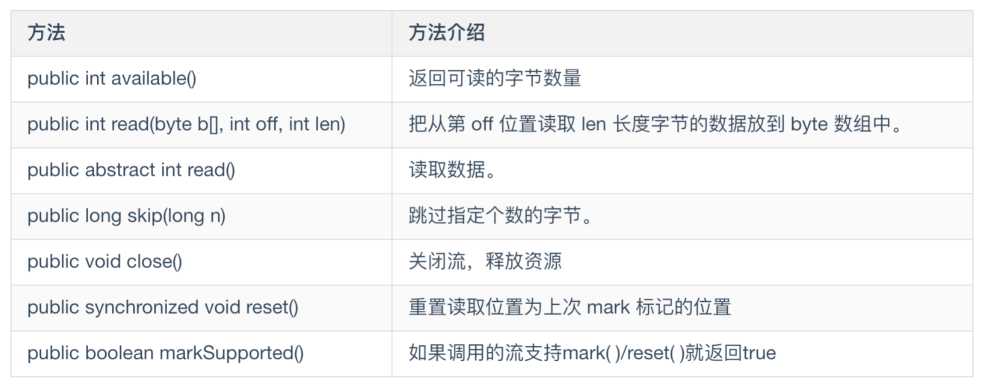
基础**IO**类和相关方法

虽然.10类有很多，但是最基本的是四个抽象类，**Inputstreams Outputstream s Reader**、**Writer**。

最基本的方法也就是read。和write。方法，其他流都是上面这四类流的子类，方法也是通过这 两类方法衍生而成的。而且大部分的10源码都是native标志的，也就是说源码都是C/C++写的。 这里我们先来认识一下这些流类及其方法

**Inputstream**

Inputstream是一个定义了 Java流式字节输入模式的抽象类。该类的所有方法在出错条件下引发一个 lOException异常。它的主要方法定义如下



**Outputstream**

Outputstream是定义了流式字节输出模式的抽象类。该类的所有方法返回一个v oid值并且在出错情况 下引发一个lOException异常。它的主要方法定义如下



**Reader** 类

Reader是Java定义的流式字符输入模式的抽象类。类中的方法在出错时引发 ^Exception异常。

|  |  |
| --- | --- |
| **方法** | **描述** |
| int read() | 如果调用的输入流的下一个字符可读则返回一个整型。遇到文件尾时返 回-1。 |
| int read(char buffer[]) | 从缓冲区中读取字符，返回实际成功读取的字符数。遇到文件尾返回-1 |
| abstract int read (char buffer[],int | 试图读取buffer中从buffer[offset]开始的numChars个字符，返回实 |
| offset, int numChars) | 际成功读取的字符数。遇到文件尾返回-1 |
| boolean ready() | 如果下一个输入请求不等待则返回true,否则返回false |
| long skip(long numChars) | 跳过numChars个输入字符，返回跳过的字符数 |
| boolean markSupportedO | 判断当前流是否支持标记流 |
| void resetQ | 重置读取位置为上次mark标记的位置 |
| void mark(int numChars) | 在输入流的当前位置设立一个标志。该输入流在numChars个字符被读 取之前有效 |
| abstract void close() | 关闭输入源。进一步的读取将会产生lOException异常 |

**Writer** 类

Writer是定义流式字符输出的抽象类。所有该类的方法都返回一个void值并在出错条件下引发 lOException 异常

|  |  |
| --- | --- |
| **方法** | **描述** |
| void write(char buffer[]) | 向一个输出流写一个完整的字符数组 |
| abstract void write(char buffer[ ], int offset,int numChars) | 向调用的输出流写入数组buffer以buffer[offset]为起点的 numChars个字符区域的内容 |
| abstract void close() | 关闭输出流。关闭后的写操作会产生lOException异常 |
| abstract void flush() | 刷新缓冲区 |
| Writer append(CharSequence esq) | 追加一个字符序列 |
| Writer append(CharSequence esq, int start, int end) | 追加写入一个字符序列的一部分，从start位置幵始，end位置 结束 |
| Writer append(char c) | 追加写入一个16位的字符 |

**Inputstream**及其子类

**FileInputStream文件输入流：**FilelnputStream类创建一个能从文件读取字节的Inputstream类

**ByteArrayInputStream字节数组输入流**：把内存中的一个缓冲区作为InputStream使用

**PipedInputStream管道输入流**：实现了 pipe管道的概念，主要在线程中使用

**Sequenceinputstream 顺序输入流：**把多个 InputStream 合并为一个 InputStream

**FilterOutputStream过滤输入流：**其他输入流的包装。

**ObjectInputStream反序列化输入流**：将之前使用ObjectOutputStream序列化的原始数据恢复为对 象，以流的方式读取对象

**DataInputStream** :数据输入流允许应用程序以与机器无关方式从底层输入流中读取基本Java数据类 型。

**PushbackInputStream推回输入流**：缓冲的一个新颖的用法是实现推回(pushback)

Pushback用于输入流允许字节被读取然后返回到流。

**Outputstream**及其子类

**FileOutputStream文件输出流**：该类实现了一个输出流，其数据写入文件。

**ByteArrayOutputStream字节数组输出流**：该类实现了一个输出流，其数据被写入由byte数组充当 的缓冲区，缓冲区会随着数据的不断写入而自动增长。

**PipedOutputStream管道输出流：**管道的输出流，是管道的发送端。

**ObjectOutputStream基本类型输出流**：该类将实现了序列化的对象序列化后写入指定地方。

FilterOutputStream过滤输出流：其他输出流的包装。

**PrintStream打印流**通过Printstream可以将文字打印到文件或者网络中去。

**DataOutputStream** :数据输出流允许应用程序以与机器无关方式向底层输出流中写入基本Java数据

类型。

**Reader**及其子类

**FileReader文件字符输入流**：把文件转换为字符流读入

**CharArrayReader字符数组输入流**：是一个把字符数组作为源的输入流的实现

**BufferedReader缓冲区输入流：**BufferedReader类从字符输入流中读取文本并缓冲字符，以便有效 地读取字符，数组和行

**PushbackReader** : PushbackReader类允许一个或多个字符被送回输入流。

**PipedReader管道输入流**：主要用途也是在线程间通讯，不过这个可以用来传输字符

**Writer**及其子类

**FileWriter字符输出流：**FileWriter创建一个可以写文件的Writer类。

**CharArrayWriter字符数组输出流：**CharArrayWriter实现了以数组作为目标的输出流。

**BufferedWriter 缓冲区输出流：**BufferedWriter是一个增加了 flush()方法的Writer。flush()方法 可以用来确保数据缓冲器确实被写到实际的输出流。

**PrintWriter :** PrintWriter本质上是PrintStream的字符形式的版本。

**PipedWriter管道输出流**：主要用途也是在线程间通讯，不过这个可以用来传输字符

Java的输入输出的流式接口为复杂而繁重的任务提供了一个简洁的抽象。过滤流类的组合允许你动态 建立客户端流式接口来配合数据传输要求。继承高级流类InputStream、InputStreamReader、Reader 和Writer类的Java程序在将来(即使创建了新的和改进的具体类)也能得到合理运用。

注解

Java注解（Annotation）又称为元数据，它为我们在代码中添加信息提供了一种形式化的方法。 它是JDK1.5引入的，Java定义了一套注解，共有7个，3个在java, lang中，剩下4个在 java.lang.annotation 中。

作用在代码中的注解有三个，它们分别是

* Override :重写标记，一般用在子类继承父类后，标注在重写过后的子类方法上。如果发现 其父类，或者是引用的接口中并没有该方法时，会报编译错误。

・©Deprecated :用此注解注释的代码已经过时，不再推荐使用

* @SuppressWarnings :这个注解起到忽略编译器的警告作用

元注解有四个，元注解就是用来标志注解的注解。它们分别是

* ©Retention :标识如何存储，是只在代码中，还是编入class文件中，或者是在运行时可以通过 反射访问。

RetentionPolicy.SOURCE:注解只保留在源文件，当Java文件编译成class文件的时候，注解被遗 弃；

RetentionPolicy.CLASS:注解被保留到class文件，但jvm加载class文件时候被遗弃，这是默认的 生命周期；

RetentionPolicy.RUNTIME：注解不仅被保存到class文件中，jvm加载class文件之后，仍然存在；

* ©Documented :标记这些注解是否包含在JavaDoc中。
* @Target :标记这个注解说明了 Annotation所修饰的对象范围，Annotation可被用于 packages, types （类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变 量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。取值如下

public enum ElementType {

1. TYPE,

FIELD,

METHOD,

PARAMETER,

CONSTRUCTOR,

1. LOCAL-VARIABLE,

ANNOTATION\_TYPE,

PACKAGE,

TYPE\_PARAMETER,

TYPE.USE

* inherited :标记这个注解是继承于哪个注解类的。

从JDK1.7开始，又添加了三个额外的注解，它们分别是

* @SafeVarargs :在声明可变参数的构造函数或方法时，Java编译器会报unchecked警告。使 用@SafeVarargs可以忽略这些警告
* ©FunctionalInterface :表明这个方法是一个函数式接口
* @Repeatable :标识某注解可以在同一个声明上使用多次。

注意：注解是不支持继承的。

关于**null**的几种处理方式

对于Java程序员来说，空指针一直是恼人的问题，我们在开发中经常会受到NullPointerException的 蹂踽和壁咚。Java的发明者也承认这是一个巨大的设计错误。

那么关于null，你应该知道下面这几件事情来有效的了解null，从而避免很多由null引起的错误。

**1、大小写敏感**

**2、null是任何引用类型的初始值**

**3、null既不是对象也不是类型，它是一种特殊的值  
你可以将它赋值给任何引用类型**

**4、null不能赋值给基本数据类型**

**5、将null赋给包装类，自动拆箱会报NPE**

**6、帯有null的引用类型变量，instanceof会报false**

**7、静态变量为null调用静态方法不会抛出NPE**

**8、使用null值安全的方法**

**9、使用==或!=判断null**

大小写敏感

首先，null是Java中的关键字，像是**public**、**static**、**final**。它是大小写敏感的，你不能将null写 成Null或NULL，编辑器将不能识别它们然后报错。

public class AccessModifierNull {

Object object = Null;

Object object1 = null;

这个问题已经几乎不会出现，因为eclipse和Idea编译器已经给出了编译器提示，所以你不用考虑这 个问题。



**null**是任何引用类型的初始值

null是所有引用类型的默认值，Java中的任何引用变量都将nulI作为默认值，也就是说所有Object类 下的引用类型默认值都是null。这对所有的引用变量都适用。就像是基本类型的默认值一样，例如int 的默认值是0, boolean的默认值是false。

下面是基本数据类型的初始值

|  |  |
| --- | --- |
| **类型** | **初始值** |
| boolean | false |
| char | /uOOOO |
| byte | (byte)O |
| short | (short)O |
| int | 0 |
| long | 0L |
| float | O.Of |
| double | O.Od |

**null**只是一种特殊的值

null既不是对象也不是一种类型，它仅是一种特殊的值，你可以将它赋予任何类型，你可以将null转换 为任何类型

1. public static void main(String[] args) {
2. String str = null;
3. Integer itr = null;
4. Double dou = null;

5

1. Integer integer = (Integer) null;
2. String string = (String)null;

8

1. System.out.println("integer = " + integer);
2. System.out.println("string = " + string);
3. }

你可以看到在编译期和运行期内，将nul l转换成任何的引用类型都是可行的，并且不会抛出空指针异 常。

**null只能赋值给引用变量，不能赋值给基本类型变量**。

**持有null的包装类在进行自动拆箱的时候，不能完成转换，会抛出空指针异常，并且null也不能和基 本数据类型进行对比**

1. public static void main(String[] args) {
2. int i = 0;
3. Integer itr = null;
4. System.out.println(itr == i);
5. }

使用了帯有null值的引用类型变量，instanceof操作会返回false

1. public static void main(String[] args) {
2. Integer isNull = null;
3. // instanceof = isInstance 方法
4. if(isNull instanceof Integer){
5. System.out.println("isNull is instanceof Integer");
6. }else{
7. System.out.println("isNull is not instanceof Integer");
8. }
9. }

这是instanceof操作符一个很重要的特性，使得对类型强制转换检查很有用

**静态变量为null调用静态方法不会抛出NullPointerException。因为静态方法使用了静态绑定。**

使用**Null-Safe**方法

你应该使用null-safe安全的方法，java类库中有很多工具类都提供了静态方法，例如基本数据类型的 包装类,Integer , Double等。例如

1. public class NullSafeMethod {

2

1. private static String number;

4

1. public static void main(String[] args) {
2. String s = String.valueOf(number);
3. String string = number.toString();
4. System.out.println("s = " + s);
5. System.out.println("string = " + string);
6. }
7. }

number没有赋值，所以默认为null,使用String.value(number)静态方法没有抛出空指针异常， 但是使用toString。却抛出了空指针异常。所以尽量使用对象的静态方法。

**null**判断

你可以使用==或者!=操作来比较null值，但是不能使用其他算法或者逻辑操作，例如小于或者 大于。跟SQL不一样，在Java中null == null将返回true,如下所示：

public class CompareNull {

1

2

3

4

5

6

7

8

9

10

private static String str1; private static String str2;

public static void main(String[] args) { System.out.println("str1 == str2 ? " + str1 == str2); System.out.println(null == null);

}

}

关于思维导图

我把一些常用的Java工具包的思维导图做了汇总，方便读者查阅。

**Java.IO**

**Fileinputstream**

**BufferedlnputStream**

DatalnputStream

Filterinputstream

PushbaklnputStream

Objectinputstream

PipedlnputStream

Sequenceinputstream

StringBufferlnputStream

ByteArraylnputStream

FileOutputStream

BufferedOutputStream

DataOutputStream

FilterOutputStream

ObjectOutputStream

PipedOutputStream

ByteArrayOutputStream

Printstream

InputStreamReader

StringReader

PipedReader

CharArrayReader

FilterReader

BufferedWriter

OutputStreamWriter

PrinterWriter

StringWriter

PipedWriter

CharArrayWriter

FilterWriter

**BufferedReader**

**FileReader**

**PushbackReader**

**FileWriter**

**DatalnputStream**

基本数据类型

**FileinputStream**

DataOutputStream

FileOutputStream

FileReader

F eWr ter

字符数组

字节数组 -—

ByteArrayOutputStream

**ByteArraylnputStream**

**CharArrayReader**

**CharArrayWriter**

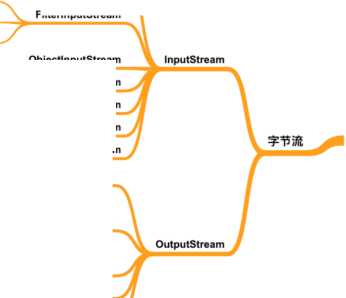
**PipedlnputStream**

管道

**PipedOutputStream**

**PipedReader**

**PipedWriter**



**Java.lang**

Process

AutoCloseable

ProcessBuilder

Annotation

ProcessEnvironment

CharSequence

Processlmpl

Thread

ThreadGroup

ThreadDeath

Readable

ThreadLocal

Runnab e

Runnable

Runtime

Invocation Handler

RuntimePerssion

GenericDeclaration

System

AnnotatedElement

Member

NoClassDefFoundError

Boolean

NoSuchFieldError

Character

OutOfMemoryError

ArraylndexOutOfBoundsException

ClassCastException

ClassNotFoundException

HlegalArgumentException

IndexOutOfBoundsException

NoSuchFieldException

NoSuchMethodException

NullPointerException

Package

NumberFormatException

RuntimeException

InterruptedExce ption

Deprecated

StringBuffer

Functionallnterface

StringBuilder

Override

SafeVarargs

SuppressWamings

Documented

Inherited

Native

Repeatable

Executable

Retention

AccessibleObject

Target

**Cloneable**

**Comparable**

StackTraceElement

**NoSuchMethodError**

**Java.math**

MathContext

MutableBiglnteger

RoundingMode

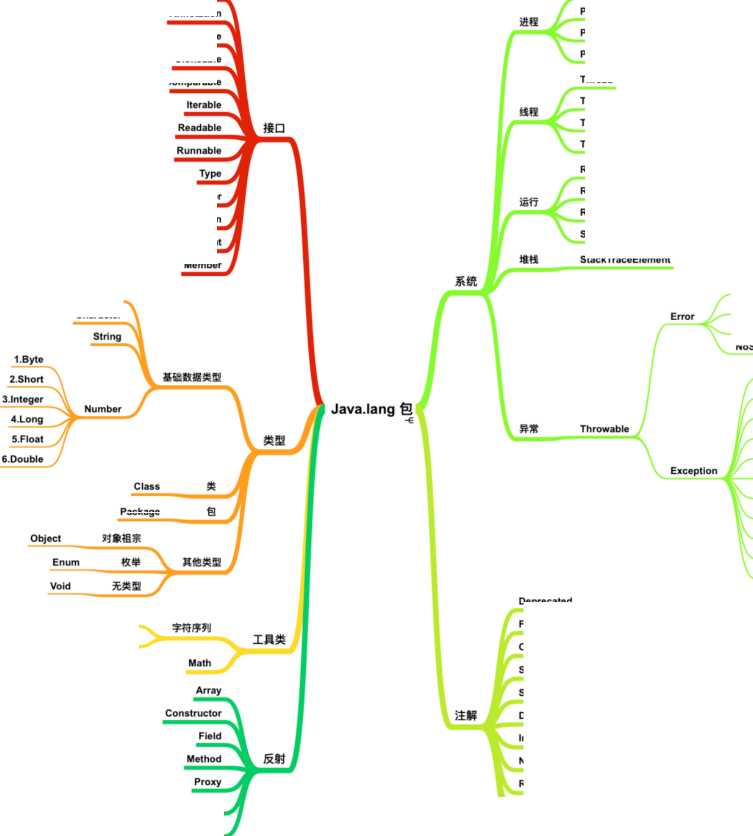
SignedMutableBiglnteger

BigDecimal

Biginteger

Java.Math

BitSieve



**Java.net**

CookieManager DatagramPacket DatagramSocket DefaultDatagramSocketlmpIFactory Defaultinterface HostPortrange HttpConnectSocketlmpI HttpCookie IDN lnet4Address lnet4Addresslmpl lnet6Address lnet6Addresslmpl InetAddress InetAddressContainer InetSocketAddress InMemoryCookieStore InterfaceAddress MulticastSocket NetPermission Networkinterface 类

PasswordAuthentication PlainDatagramSocketlmpI PlainSocketlmpI Proxy SdpSocketlmpI ServerSocket Socket Socketinputstream SocketOutputStream SocketPermission SocketSecrets SocksSocketlmpI StandardProtocolFamily StandardSocketOptions

URI

ContentHandlerFactory

CookiePolicy

CookieStore

DatagramSocketlmpIFactory

FileNameMap

InetAddressImpI

ProtocolFamily

SocketlmpIFactory

SocketOption

SocketOptions

Authenticator

CookieHandler

DatagramSocketlmpI

HttpURLConnection

JarURLConnection

ProxySelector

ResponseCache

Socketlmpl

URLConnection

URLStreamHandler

BindException

ConnectException

MalformedURLException

URISyntaxException

UnknownServiceException

UnknownHostException

SocketTimeoutException

SocketException

URLStreamHandlerFactory

AbstractPlamSocketlmpI

CacheRequest

ContentHandler

SecureCacheResponse

URL

URLCIassLoader

URLDecoder

URLEncoder

URLPermission

ProtocolException

PortUnreachableException

NoRouteToHostException

HttpRetryException

