f96c7b71-2e05-41fc-8f6d-4f5a05aa4e13\_Java

# 第 1 页

## Java基础

### Java概述

### 何为编程

编程就是让计算机为解决某个问题⽽使⽤某种程序设计语⾔编写程序代码，并最终得到结果的过程。

为了使计算机能够理解⼈的意图，⼈类就必须要将需解决的问题的思路、⽅法、和⼿段通过计算机能

够

理解的形式告诉计算机，使得计算机能够根据⼈的指令⼀步⼀步去⼯作，完成某种特定的任务。这种

⼈

和计算机之间交流的过程就是编程。

### 什么是Java

Java是⼀⻔⾯向对象编程语⾔，不仅吸收了C++语⾔的各种优点，还摒弃了C++⾥难以理解的多继

承、

指针等概念，因此Java语⾔具有功能强⼤和简单易⽤两个特征。Java语⾔作为静态⾯向对象编程语⾔

的

代表，极好地实现了⾯向对象理论，允许程序员以优雅的思维⽅式进⾏复杂的编程。

### jdk1.5之后的三⼤版本

JavaSE（J2SE，Java2PlatformStandardEdition，标准版）

JavaSE以前称为J2SE。它允许开发和部署在桌⾯、服务器、嵌⼊式环境和实时环境中使⽤的Java

应⽤程序。JavaSE包含了⽀持JavaWeb服务开发的类，并为JavaEE和JavaME提供基础。

JavaEE（J2EE，Java2PlatformEnterpriseEdition，企业版）

JavaEE以前称为J2EE。企业版本帮助开发和部署可移植、健壮、可伸缩且安全的服务器端Java应

⽤程序。JavaEE是在JavaSE的基础上构建的，它提供Web服务、组件模型、管理和通信API，

可以⽤来实现企业级的⾯向服务体系结构（service-orientedarchitecture，SOA）和Web2.0应

⽤程序。2018年2⽉，Eclipse宣布正式将JavaEE更名为JakartaEE

JavaME（J2ME，Java2PlatformMicroEdition，微型版）

JavaME以前称为J2ME。JavaME为在移动设备和嵌⼊式设备（⽐如⼿机、PDA、电视机顶盒和

# 第 2 页

打印机）上运⾏的应⽤程序提供⼀个健壮且灵活的环境。JavaME包括灵活的⽤⼾界⾯、健壮的安

全模型、许多内置的⽹络协议以及对可以动态下载的连⽹和离线应⽤程序的丰富⽀持。基于Java

ME规范的应⽤程序只需编写⼀次，就可以⽤于许多设备，⽽且可以利⽤每个设备的本机功能。

## JVM、JRE和JDK的关系

JVM

JavaVirtualMachine是Java虚拟机，Java程序需要运⾏在虚拟机上，不同的平台有⾃⼰的虚拟

机，因此Java语⾔可以实现跨平台。

JRE

JavaRuntimeEnvironment包括Java虚拟机和Java程序所需的核⼼类库等。核⼼类库主要是

java.lang包：包含了运⾏Java程序必不可少的系统类，如基本数据类型、基本数学函数、字符串处

理、线程、异常处理类等，系统缺省加载这个包

如果想要运⾏⼀个开发好的Java程序，计算机中只需要安装JRE即可。

JDK

JavaDevelopmentKit是提供给Java开发⼈员使⽤的，其中包含了Java的开发⼯具，也包括了

JRE。所以安装了JDK，就⽆需再单独安装JRE了。其中的开发⼯具：编译⼯具(javac.exe)，打包⼯

具(jar.exe)等

JVM&JRE&JDK关系图

## 什么是跨平台性？原理是什么

所谓跨平台性，是指java语⾔编写的程序，⼀次编译后，可以在多个系统平台上运⾏。

实现原理：Java程序是通过java虚拟机在系统平台上运⾏的，只要该系统可以安装相应的java虚拟

机，

该系统就可以运⾏java程序。

## Java语⾔有哪些特点

简单易学（Java语⾔的语法与C语⾔和C++语⾔很接近）

⾯向对象（封装，继承，多态）

平台⽆关性（Java虚拟机实现平台⽆关性）

⽀持⽹络编程并且很⽅便（Java语⾔诞⽣本⾝就是为简化⽹络编程设计的）

⽀持多线程（多线程机制使应⽤程序在同⼀时间并⾏执⾏多项任）

# 第 3 页

健壮性（Java语⾔的强类型机制、异常处理、垃圾的⾃动收集等）

安全性

## 什么是字节码？采⽤字节码的最⼤好处是什么

字节码：Java源代码经过虚拟机编译器编译后产⽣的⽂件（即扩展为.class的⽂件），它不⾯向任何

特

定的处理器，只⾯向虚拟机。

采⽤字节码的好处：

Java语⾔通过字节码的⽅式，在⼀定程度上解决了传统解释型语⾔执⾏效率低的问题，同时⼜保留了

解

释型语⾔可移植的特点。所以Java程序运⾏时⽐较⾼效，⽽且，由于字节码并不专对⼀种特定的机

器，

因此，Java程序⽆须重新编译便可在多种不同的计算机上运⾏。

先看下java中的编译器和解释器：

Java中引⼊了虚拟机的概念，即在机器和编译程序之间加⼊了⼀层抽象的虚拟机器。这台虚拟的机器

在

任何平台上都提供给编译程序⼀个的共同的接⼝。编译程序只需要⾯向虚拟机，⽣成虚拟机能够理解

的

代码，然后由解释器来将虚拟机代码转换为特定系统的机器码执⾏。在Java中，这种供虚拟机理解的

代

码叫做字节码（即扩展为.class的⽂件），它不⾯向任何特定的处理器，只⾯向虚拟机。每⼀种平台的

解释器是不同的，但是实现的虚拟机是相同的。Java源程序经过编译器编译后变成字节码，字节码由

虚

拟机解释执⾏，虚拟机将每⼀条要执⾏的字节码送给解释器，解释器将其翻译成特定机器上的机器

码，

然后在特定的机器上运⾏，这就是上⾯提到的Java的特点的编译与解释并存的解释。

## 什么是Java程序的主类？应⽤程序和⼩程序的主类有何不同？

⼀个程序中可以有多个类，但只能有⼀个类是主类。在Java应⽤程序中，这个主类是指包含main()⽅

法

的类。⽽在Java⼩程序中，这个主类是⼀个继承⾃系统类JApplet或Applet的⼦类。应⽤程序的主类不

⼀

定要求是public类，但⼩程序的主类要求必须是public类。主类是Java程序执⾏的⼊⼝点。

# 第 4 页

## Java应⽤程序与⼩程序之间有那些差别？

简单说应⽤程序是从主线程启动(也就是main()⽅法)。applet⼩程序没有main⽅法，主要是嵌在浏览

器

⻚⾯上运⾏(调⽤init()线程或者run()来启动)，嵌⼊浏览器这点跟flash的⼩游戏类似。

## Java和C++的区别

代码块

Java源代码---->编译器---->jvm可执⾏的Java字节码(即虚拟指令)---->jvm---->jvm中解释器--

--

->机器可执⾏的⼆进制机器码---->程序运⾏。

我知道很多⼈没学过C++，但是⾯试官就是没事喜欢拿咱们Java和C++⽐呀！没办法！！！就算没学

过

C++，也要记下来！

都是⾯向对象的语⾔，都⽀持封装、继承和多态

Java不提供指针来直接访问内存，程序内存更加安全

Java的类是单继承的，C++⽀持多重继承；虽然Java的类不可以多继承，但是接⼝可以多继承。

Java有⾃动内存管理机制，不需要程序员⼿动释放⽆⽤内存

## OracleJDK和OpenJDK的对⽐

1.OracleJDK版本将每三年发布⼀次，⽽OpenJDK版本每三个⽉发布⼀次；

2.OpenJDK是⼀个参考模型并且是完全开源的，⽽OracleJDK是OpenJDK的⼀个实现，并不是完全

开源的；

3.OracleJDK⽐OpenJDK更稳定。OpenJDK和OracleJDK的代码⼏乎相同，但OracleJDK有更多

的

类和⼀些错误修复。因此，如果您想开发企业/商业软件，我建议您选择OracleJDK，因为它经过

了彻底的测试和稳定。某些情况下，有些⼈提到在使⽤OpenJDK可能会遇到了许多应⽤程序崩溃

的问题，但是，只需切换到OracleJDK就可以解决问题；

4.在响应性和JVM性能⽅⾯，OracleJDK与OpenJDK相⽐提供了更好的性能；

5.OracleJDK不会为即将发布的版本提供⻓期⽀持，⽤⼾每次都必须通过更新到最新版本获得⽀持来

获取最新版本；

6.OracleJDK根据⼆进制代码许可协议获得许可，⽽OpenJDK根据GPLv2许可获得许可。

1

2

# 第 5 页

## ⾯向对象

### 什么是⾯向对象？

⾯向对象程序设计是以建⽴模型体现出来的抽象思维过程和⾯向对象的⽅法。我们可以将某个事物抽

象

出来，赋予它⾃⼰的特征，并且可以针对这个事物进⾏相应的操作，以及规定与其他对象之间的关

系。

可以降低代码的耦合度，使程序更加灵活。

### 多态的好处

允许不同类对象对同⼀消息做出响应，即同⼀消息可以根据发送对象的不同⽽采⽤多种不同的⾏为⽅

式

(发送消息就是函数调⽤)。

即⽗类型的引⽤指向⼦类型的对象。主要有以下优点：

可替换性：多态对已存在代码具有可替换性

可扩充性：增加新的⼦类不影响已经存在的类结构

更加灵活

### ⾯向对象和⾯向过程的区别？

⾯向过程：

优点：性能⽐⾯向对象⾼，因为类调⽤时需要实例化，开销⽐较⼤，⽐较消耗资源;⽐如单⽚机、嵌⼊

式

开发、Linux/Unix等⼀般采⽤⾯向过程开发，性能是最重要的因素。

缺点：没有⾯向对象易维护、易复⽤、易扩展

⾯向对象：

优点：易维护、易复⽤、易扩展，由于⾯向对象有封装、继承、多态性的特性，可以设计出低耦合的

系

统，使系统更加灵活、更加易于维护

缺点：性能⽐⾯向过程低

⾯向过程是具体化的，流程化的，解决⼀个问题，你需要⼀步⼀步的分析，⼀步⼀步的实现。

⾯向对象是模型化的，你只需抽象出⼀个类，这是⼀个封闭的盒⼦，在这⾥你拥有数据也拥有解决问

题

# 第 6 页

的⽅法。需要什么功能直接使⽤就可以了，不必去⼀步⼀步的实现，⾄于这个功能是如何实现的，管

我

们什么事？我们会⽤就可以了。

⾯向对象的底层其实还是⾯向过程，把⾯向过程抽象成类，然后封装，⽅便我们使⽤的就是⾯向对象

了。

抽象：抽象是将⼀类对象的共同特征总结出来构造类的过程，包括数据抽象和⾏为抽象两⽅⾯。抽象

只

关注对象有哪些属性和⾏为，并不关注这些⾏为的细节是什么。

## ⾯向对象三⼤特性？

其中Java⾯向对象编程三⼤特性：封装继承多态

封装

封装把⼀个对象的属性私有化，同时提供⼀些可以被外界访问的属性的⽅法，如果属性不想被外界访

问，我们⼤可不必提供⽅法给外界访问。但是如果⼀个类没有提供给外界访问的⽅法，那么这个类也

没

有什么意义了。

封装隐藏对象的属性和实现细节，仅对外提供公共访问⽅式，将变化隔离，便于使⽤，提⾼复⽤性和

安

全性。

继承

继承是使⽤已存在的类的定义作为基础建⽴新类的技术，新类的定义可以增加新的数据或新的功能，

也

可以⽤⽗类的功能，但不能选择性地继承⽗类。通过使⽤继承我们能够⾮常⽅便地复⽤以前的代码。

关于继承如下3点请记住：

7.⼦类拥有⽗类⾮private的属性和⽅法。

8.⼦类可以拥有⾃⼰属性和⽅法，即⼦类可以对⽗类进⾏扩展。

9.⼦类可以⽤⾃⼰的⽅式实现⽗类的⽅法。（以后介绍）。

多态

所谓多态就是指程序中定义的引⽤变量所指向的具体类型和通过该引⽤变量发出的⽅法调⽤在编程时

并

不确定，⽽是在程序运⾏期间才确定，即⼀个引⽤变量到底会指向哪个类的实例对象，该引⽤变量发

出

的⽅法调⽤到底是哪个类中实现的⽅法，必须在由程序运⾏期间才能决定。

# 第 7 页

在Java中有两种形式可以实现多态：继承（多个⼦类对同⼀⽅法的重写）和接⼝（实现接⼝并覆盖接

⼝

中同⼀⽅法）。

⽅法重载（overload）实现的是编译时的多态性（也称为前绑定），⽽⽅法重写（

override）实现的是运⾏时的多态性（也称为后绑定）。

⼀个引⽤变量到底会指向哪个类的实例对象，该引⽤变量发出的⽅法调⽤到底是哪个类中实现的⽅

法，

必须在由程序运⾏期间才能决定。

运⾏时的多态是⾯向对象最精髓的东西，要实现多态需要做两件事：

⽅法重写（⼦类继承⽗类并重写⽗类中已有的或抽象的⽅法）；

对象造型（⽤⽗类型引⽤⼦类型对象，这样同样的引⽤调⽤同样的⽅法就会根据⼦类对象的不同⽽

表现出不同的⾏为）。

## 什么是多态机制？Java语⾔是如何实现多态的？

所谓多态就是指程序中定义的引⽤变量所指向的具体类型和通过该引⽤变量发出的⽅法调⽤在编程时

并

不确定，⽽是在程序运⾏期间才确定，即⼀个引⽤变量倒底会指向哪个类的实例对象，该引⽤变量发

出

的⽅法调⽤到底是哪个类中实现的⽅法，必须在由程序运⾏期间才能决定。

因为在程序运⾏时才确定具体的类，这样，不⽤修改源程序代码，就可以让引⽤变量绑定到各种不同

的类实现上，从⽽导致该引⽤调⽤的具体⽅法随之改变，即不修改程序代码就可以改变程序运⾏时所

绑定的具体代码，让程序可以选择多个运⾏状态，这就是多态性。

多态分为编译时多态和运⾏时多态。其中编辑时多态是静态的，主要是指⽅法的重载，它是根据参数

列

表的不同来区分不同的函数，通过编辑之后会变成两个不同的函数，在运⾏时谈不上多态。⽽运⾏时

多

态是动态的，它是通过动态绑定来实现的，也就是我们所说的多态性。

## 多态的实现

Java实现多态有三个必要条件：继承、重写、向上转型。

继承：在多态中必须存在有继承关系的⼦类和⽗类。

重写：⼦类对⽗类中某些⽅法进⾏重新定义，在调⽤这些⽅法时就会调⽤⼦类的⽅法。

向上转型：在多态中需要将⼦类的引⽤赋给⽗类对象，只有这样该引⽤才能够具备技能调⽤⽗类的⽅

法

# 第 8 页

和⼦类的⽅法。

只有满⾜了上述三个条件，我们才能够在同⼀个继承结构中使⽤统⼀的逻辑实现代码处理不同的对

象，

从⽽达到执⾏不同的⾏为。

对于Java⽽⾔，它多态的实现机制遵循⼀个原则：当超类对象引⽤变量引⽤⼦类对象时，被引⽤对象

的

类型⽽不是引⽤变量的类型决定了调⽤谁的成员⽅法，但是这个被调⽤的⽅法必须是在超类中定义过

的，也就是说被⼦类覆盖的⽅法。

## ⾯向对象五⼤基本原则是什么?

单⼀职责原则SRP(SingleResponsibilityPrinciple)

类的功能要单⼀，不能包罗万象，跟杂货铺似的。

开放封闭原则OCP(Open−ClosePrinciple)

⼀个模块对于拓展是开放的，对于修改是封闭的，想要增加功能热烈欢迎，想要修改，哼，⼀万个

不乐意。

⾥式替换原则LSP(theLiskovSubstitutionPrincipleLSP)

⼦类可以替换⽗类出现在⽗类能够出现的任何地⽅。⽐如你能代表你爸去你姥姥家⼲活。哈哈~~

依赖倒置原则DIP(theDependencyInversionPrincipleDIP)

⾼层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。抽象不应该依赖于具体实现，

具体实现应该依赖于抽象。就是你出国要说你是中国⼈，⽽不能说你是哪个村⼦的。⽐如说中国⼈

是抽象的，下⾯有具体的xx省，xx市，xx县。你要依赖的抽象是中国⼈，⽽不是你是xx村的。

接⼝分离原则ISP(theInterfaceSegregationPrincipleISP)

设计时采⽤多个与特定客⼾类有关的接⼝⽐采⽤⼀个通⽤的接⼝要好。就⽐如⼀个⼿机拥有打电

话，看视频，玩游戏等功能，把这⼏个功能拆分成不同的接⼝，⽐在⼀个接⼝⾥要好的多。

## 什么是值传递和引⽤传递？

值传递，是对基本型变量⽽⾔的，传递的是该变量的⼀个副本，改变副本不影响原变量。

引⽤传递，⼀般是对于对象型变量⽽⾔的，传递的是该对象地址的⼀个副本，并不是原对象本⾝。

⼀般认为，Java内的传递都是值传递，Java中实例对象的传递是引⽤传递。

## 代码中如何实现多态

实现多态主要有以下三种⽅式：

# 第 9 页

10.接⼝实现

11.继承⽗类重写⽅法

12.同⼀类中进⾏⽅法重载

## 接⼝的意义

规范，扩展，回调。

## 抽象类的意义

为其他⼦类提供⼀个公共的类型

封装⼦类中重复定义的内容

定义抽象⽅法,⼦类虽然有不同的实现，但是定义时⼀致的

## 接⼝和抽象类的区别

## ⽗类的静态⽅法能否被⼦类重写



*图片 1 (562x576)*

# 第 10 页

不能。重写只适⽤于实例⽅法,不能⽤于静态⽅法，⽽⼦类当中含有和⽗类相同签名的静态⽅法，我们

⼀

般称之为隐藏。

### 什么是不可变对象

不可变对象指对象⼀旦被创建，状态就不能再改变。任何修改都会创建⼀个新的对象，如String、

Integer及其它包装类。

### 静态变量和实例变量的区别?

静态变量存储在⽅法区，属于类所有。实例变量存储在堆当中，其引⽤存在当前线程栈。

### 能否创建⼀个包含可变对象的不可变对象?

当然可以创建⼀个包含可变对象的不可变对象的，你只需要谨慎⼀点，不要共享可变对象的引⽤就可

以

了，如果需要变化时，就返回原对象的⼀个拷⻉。最常⻅的例⼦就是对象中包含⼀个⽇期对象的引

⽤。

### Overload和Override的区别。Overloaded的⽅法是否可以改变返回值的

### 类型

答：⽅法的重写Overriding和重载Overloading是Java多态性的不同表现。重写Overriding是⽗类与

⼦

类之间多态性的⼀种表现，重载Overloading是⼀个类中多态性的⼀种表现。如果在⼦类中定义某⽅

法

与其⽗类有相同的名称和参数，我们说该⽅法被重写(Overriding)。如果在⼀个类中定义了多个同名

的

⽅法，它们或有不同的参数个数或有不同的参数类型，则称为⽅法的重载(Overloading)。

Overloaded

的⽅法是可以改变返回值的类型

## 基础语法

### 数据类型

### Java有哪些数据类型

定义：Java语⾔是强类型语⾔，对于每⼀种数据都定义了明确的具体的数据类型，在内存中分配了不

同⼤⼩的内存空间。

# 第 11 页

分类

基本数据类型

## •

数值型

## ◦

整数类型(byte,short,int,long)

## ◦

浮点类型(float,double)

## •

字符型(char)

## •

布尔型(boolean)

## •

引⽤数据类型

## ◦

类(class)

## ◦

接⼝(interface)

## ◦

数组([])

### Java基本数据类型图

### switch是否能作⽤在byte上，是否能作⽤在long上，是否能作⽤在String上

在Java5以前，switch(expr)中，expr只能是byte、short、char、int。从Java5开始，Java中引

⼊

了枚举类型，expr也可以是enum类型，从Java7开始，expr还可以是字符串（String），但是⻓

整

型（long）在⽬前所有的版本中都是不可以的。

### ⽤最有效率的⽅法计算2乘以8

2<<3（左移3位相当于乘以2的3次⽅，右移3位相当于除以2的3次⽅）。

Math.round(11.5)等于多少？Math.round(-11.5)等于多少

Math.round(11.5)的返回值是12，Math.round(-11.5)的返回值是-11。四舍五⼊的原理是在参数上加

0.5然后进⾏下取整。



*图片 1 (558x186)*

# 第 12 页

### floatf=3.4;是否正确

不正确。3.4是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-

casting，

也称为窄化）会造成精度损失，因此需要强制类型转换floatf=float)3.4;或者写成floatf=3.4F;。

### shorts1=1;s1=s1+1;有错吗?shorts1=1;s1+=1;有错吗

对于shorts1=1;s1=s1+1;由于1是int类型，因此s1+1运算结果也是int型，需要强制转换类型

才能赋值给short型。

⽽shorts1=1;s1+=1;可以正确编译，因为s1+=1;相当于s1=(short(s1+1);其中有隐含的强制类型

转换。

## 编码

### Java语⾔采⽤何种编码⽅案？有何特点？

Java语⾔采⽤Unicode编码标准，Unicode（标准码），它为每个字符制订了⼀个唯⼀的数值，因此

在任何的语⾔，平台，程序都可以放⼼的使⽤。

## 注释

### 什么Java注释

定义：⽤于解释说明程序的⽂字

分类

单⾏注释

格式：//注释⽂字

多⾏注释

格式：/\*注释⽂字\*/

⽂档注释

格式：/\*\*注释⽂字\*/

作⽤

在程序中，尤其是复杂的程序中，适当地加⼊注释可以增加程序的可读性，有利于程序的修改、调试

和交流。注释的内容在程序编译的时候会被忽视，不会产⽣⽬标代码，注释的部分不会对程序的执⾏

结果产⽣任何影响。

注意事项：多⾏和⽂档注释都不能嵌套使⽤。

# 第 13 页

## 访问修饰符

### 访问修饰符public,private,protected,以及不写（默认）时的区别

定义：Java中，可以使⽤访问修饰符来保护对类、变量、⽅法和构造⽅法的访问。Java⽀持4种不同

的访问权限。

分类

private:在同⼀类内可⻅。使⽤对象：变量、⽅法。注意：不能修饰类（外部类）

default(即缺省，什么也不写，不使⽤任何关键字）:在同⼀包内可⻅，不使⽤任何修饰符。使⽤对

象：类、接⼝、变量、⽅法。

protected:对同⼀包内的类和所有⼦类可⻅。使⽤对象：变量、⽅法。注意：不能修饰类（外部

类）。

public:对所有类可⻅。使⽤对象：类、接⼝、变量、⽅法

### 访问修饰符图

## 运算符

### &和&&的区别

&运算符有两种⽤法：(1)按位与；(2)逻辑与。

&&运算符是短路与运算。逻辑与跟短路与的差别是⾮常巨⼤的，虽然⼆者都要求运算符左右两端的布

尔值都是true整个表达式的值才是true。&&之所以称为短路运算，是因为如果&&左边的表达式的值

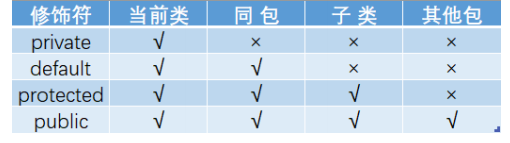
是false，右边的表达式会被直接短路掉，不会进⾏运算。

注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

### 3\*0.1==0.3返回值是什么

false，因为有些浮点数不能完全精确的表⽰出来。

### a=a+b与a+=b有什么区别吗?



*图片 1 (510x143)*

# 第 14 页

+=操作符会进⾏隐式⾃动类型转换，此处a+=b隐式的将加操作的结果类型强制转换为持有结果的类

型，⽽a=a+b则不会⾃动进⾏类型转换。如：

bytea=127;

byteb=127;

b=a+b;//error:cannotconvertfrominttobyte

b+=a;//ok

（译者注：这个地⽅应该表述的有误，其实⽆论a+b的值为多少，编译器都会报错，因为a+b操作会

将a、b提升为int类型，所以将int类型赋值给byte就会编译出错）

## shorts1=1;s1=s1+1;该段代码是否有错,有的话怎么改？

在Java中，short类型在进⾏运算时会⾃动提升为int类型。因此，s1 + 1的运算结果是

int类型，⽽不能直接赋值给short类型的变量s1，这会导致编译错误。

修改⽅式：

代码块

shorts1 =1;

s1 = (short) (s1 +1);

## int和Integer有什么区别

答：Java提供两种不同的类型：引⽤类型和原始类型（或内置类型）。Int是java的原始数据类型，

Integer是java为int提供的封装类。Java为每个原始类型提供了封装类。引⽤类型和原始类型具有不同

的

特征和⽤法，它们包括：⼤⼩和速度问题，这种类型以哪种类型的数据结构存储，当引⽤类型和原始

类型⽤作某个类的实例数据时所指定的缺省值。对象引⽤实例变量的缺省值为null，⽽原始类型实例

变量的缺省值与它们的类型有关

## 如何将byte转为String

可以使⽤String接收byte[]参数的构造器来进⾏转换，需要注意的点是要使⽤的正确的编码，否则会

使⽤平台默认编码，这个编码可能跟原来的编码相同，也可能不同。

## 可以将int强转为byte类型么?会产⽣什么问题?

我们可以做强制转换，但是Java中int是32位的⽽byte是8位的，所以,如果强制转化int类型的⾼24位

将

会被丢弃，byte类型的范围是从-128到128

1

2

# 第 15 页

## 关键字

### Java有没有goto

goto是Java中的保留字，在⽬前版本的Java中没有使⽤。

### final有什么⽤？

⽤于修饰类、属性和⽅法；

被final修饰的类不可以被继承

被final修饰的⽅法不可以被重写

被final修饰的变量不可以被改变，被final修饰不可变的是变量的引⽤，⽽不是引⽤指向的内容，引⽤

指向的内容是可以改变的

### final有哪些⽤法

final也是很多⾯试喜欢问的地⽅，能回答下以下三点就不错了：

1.被final修饰的类不可以被继承

2.被final修饰的⽅法不可以被重写

3.被final修饰的变量不可以被改变。如果修饰引⽤，那么表⽰引⽤不可变，引⽤指向的内容可变。

4.被final修饰的⽅法，JVM会尝试将其内联，以提⾼运⾏效率

5.被final修饰的常量，在编译阶段会存⼊常量池中。

回答出编译器对final域要遵守的两个重排序规则更好：

1.在构造函数内对⼀个final域的写⼊，与随后把这个被构造对象的引⽤赋值给⼀个引⽤变量,这两个操

作之间不能重排序。

2.初次读⼀个包含final域的对象的引⽤，与随后初次读这个final域,这两个操作之间不能重排序。

### finalfinallyfinalize区别

final可以修饰类、变量、⽅法，修饰类表⽰该类不能被继承、修饰⽅法表⽰该⽅法不能被重写、修饰

变量表⽰该变量是⼀个常量不能被重新赋值。

finally⼀般作⽤在try-catch代码块中，在处理异常的时候，通常我们将⼀定要执⾏的代码⽅法

finally代码块中，表⽰不管是否出现异常，该代码块都会执⾏，⼀般⽤来存放⼀些关闭资源的代码。

finalize是⼀个⽅法，属于Object类的⼀个⽅法，⽽Object类是所有类的⽗类，该⽅法⼀般由垃圾回收

器来调

⽤，当我们调⽤System.gc()⽅法的时候，由垃圾回收器调⽤finalize()，回收垃圾，⼀个对象是否可回

收的最后判断。

# 第 16 页

## this关键字的⽤法

### this是⾃⾝的⼀个对象，代表对象本⾝，可以理解为：指向对象本⾝的⼀个指针。

### this的⽤法在java中⼤体可以分为3种：

### 1.普通的直接引⽤，this相当于是指向当前对象本⾝。

### 2.形参与成员名字重名，⽤this来区分：

代码块

publicPerson(String name,intage) {

this.name = name;

this.age = age;

}

### 3.引⽤本类的构造函数

代码块

classPerson{

privateString name;

privateintage;

publicPerson() {

}

publicPerson(String name) {

this.name = name;

}

publicPerson(String name,intage) {

this(name);

this.age = age;

}

}

## super关键字的⽤法

### super可以理解为是指向⾃⼰超（⽗）类对象的⼀个指针，⽽这个超类指的是离⾃⼰最近的⼀个⽗类。

### super也有三种⽤法：

### 1.普通的直接引⽤

### 与this类似，super相当于是指向当前对象的⽗类的引⽤，这样就可以⽤super.xxx来引⽤⽗类的成

### 员。

### 2.⼦类中的成员变量或⽅法与⽗类中的成员变量或⽅法同名时，⽤super进⾏区分

1

2

3

4

1

2

3

4

5

6

7

8

9

10

11

12

13

14

# 第 17 页

代码块classPerson{

protectedString name;

publicPerson(String name) {

this.name = name;

}

}

classStudentextendsPerson{

privateString name;

publicStudent(String name, String name1) {

super(name);

this.name = name1;

}

publicvoidgetInfo(){

System.out.println(this.name);//Child

System.out.println(super.name);//Father

}

}

publicclassTest{

publicstaticvoidmain(String[] args) {

Students1=newStudent("Father","Child");

s1.getInfo();

}

3.引⽤⽗类构造函数

super（参数）：调⽤⽗类中的某⼀个构造函数（应该为构造函数中的第⼀条语句）。

this（参数）：调⽤本类中另⼀种形式的构造函数（应该为构造函数中的第⼀条语句）。

## this与super的区别

## •

super:它引⽤当前对象的直接⽗类中的成员（⽤来访问直接⽗类中被隐藏的⽗类中成员数据或函

数，基类与派⽣类中有相同成员定义时如：super.变量名super.成员函数据名（实参）

## •

this：它代表当前对象名（在程序中易产⽣⼆义性之处，应使⽤this来指明当前对象；如果函数的

形参与类中的成员数据同名，这时需⽤this来指明成员变量名）

## •

super()和this()类似,区别是，super()在⼦类中调⽤⽗类的构造⽅法，this()在本类内调⽤本类的其

它构造⽅法。

## •

super()和this()均需放在构造⽅法内第⼀⾏。

## •

尽管可以⽤this调⽤⼀个构造器，但却不能调⽤两个。

## •

this和super不能同时出现在⼀个构造函数⾥⾯，因为this必然会调⽤其它的构造函数，其它的构造

函数必然也会有super语句的存在，所以在同⼀个构造函数⾥⾯有相同的语句，就失去了语句的意

义，编译器也不会通过。

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

# 第 18 页

### •

this()和super()都指的是对象，所以，均不可以在static环境中使⽤。包括：static变量,static⽅

法，static语句块。

### •

this可⽤于synchronized：因为this表⽰当前对象，所以this可⽤于synchronized(this){....}加

锁，⽽super则不能实现此功能。

### •

从本质上讲，this是⼀个指向本对象的指针,然⽽super是⼀个Java关键字。

### static存在的主要意义

static的主要意义是在于创建独⽴于具体对象的域变量或者⽅法。以致于即使没有创建对象，也能使⽤

属性和调⽤⽅法！

static关键字还有⼀个⽐较关键的作⽤就是⽤来形成静态代码块以优化程序性能。static块可以置于类

中的任何地⽅，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执⾏每个

static块，并且只会执⾏⼀次。

为什么说static块可以⽤来优化程序性能，是因为它的特性:只会在类加载的时候执⾏⼀次。因此，很多

时候会将⼀些只需要进⾏⼀次的初始化操作都放在static代码块中进⾏。

### static的独特之处

1、被static修饰的变量或者⽅法是独⽴于该类的任何对象，也就是说，这些变量和⽅法不属于任何⼀

个实例对象，⽽是被类的实例对象所共享。

2、在该类被第⼀次加载的时候，就会去加载被static修饰的部分，⽽且只在类第⼀次使⽤时加载并进

⾏初始化，注意这是第⼀次⽤就要初始化，后⾯根据需要是可以再次赋值的。

3、static变量值在类加载的时候分配空间，以后创建类对象的时候不会重新分配。赋值的话，是可以

任意赋值的！

4、被static修饰的变量或者⽅法是优先于对象存在的，也就是说当⼀个类加载完毕之后，即便没有创

建对象，也可以去访问。

### static应⽤场景

因为static是被类的实例对象所共享，因此如果某个成员变量是被所有对象所共享的，那么这个成员变

量就应该定义为静态变量。

因此⽐较常⻅的static应⽤场景有：

1、修饰成员变量2、修饰成员⽅法3、静态代码块4、修饰类【只能修饰内部类也就是静态内部类】

5、静态导包

15static注意事项

1、静态只能访问静态。2、⾮静态既可以访问⾮静态的，也可以访问静态的。

## 流程控制语句

# 第 19 页

### break,continue,return的区别及作⽤

break跳出总上⼀层循环，不再执⾏循环(结束当前的循环体)

continue跳出本次循环，继续执⾏下次循环(结束正在执⾏的循环进⼊下⼀个循环条件)

return程序返回，不再执⾏下⾯的代码(结束当前的⽅法直接返回)

### 在Java中，如何跳出当前的多重嵌套循环

在Java中，要想跳出多重循环，可以在外⾯的循环语句前定义⼀个标号，然后在⾥层循环体的代码中

使

⽤带有标号的break语句，即可跳出外层循环。例如：

代码块

publicstaticvoidmain(String[] args) {

ok:

for(inti=0; i <10; i++) {

for(intj=0; j <10; j++) {

System.out.println("i="+ i +",j="+ j);

if(j ==5) {

breakok;

}

}

}

}

## 类

### 类与接⼝

### 抽象类和接⼝的对⽐

抽象类是⽤来捕捉⼦类的通⽤特性的。接⼝是抽象⽅法的集合。

从设计层⾯来说，抽象类是对类的抽象，是⼀种模板设计，接⼝是⾏为的抽象，是⼀种⾏为的规范。

相同点

### •

接⼝和抽象类都不能实例化

### •

都位于继承的顶端，⽤于被其他实现或继承

### •

都包含抽象⽅法，其⼦类都必须覆写这些抽象⽅法

不同点

1

2

3

4

5

6

7

8

9

10

11

# 第 20 页

### 备注：Java8中接⼝中引⼊默认⽅法和静态⽅法，以此来减少抽象类和接⼝之间的差异。

### 现在，我们可以为接⼝提供默认实现的⽅法了，并且不⽤强制⼦类来实现它。

代码块

//定义接⼝Animal

publicinterfaceAnimal{

//抽象⽅法，所有实现类必须实现

voidmakeSound();

//默认⽅法，提供默认实现

defaultvoidsleep() {

System.out.println("This animal is sleeping.");

}

}

//实现类Dog实现了Animal接⼝，并覆盖了默认⽅法sleep()

publicclassDogimplementsAnimal{

@Override

publicvoidmakeSound() {

System.out.println("Bark");

}

@Override

publicvoidsleep() {

System.out.println("Dog is sleeping.");

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18



*图片 1 (587x512)*

# 第 21 页

}

}

//实现类Cat实现了Animal接⼝，但没有覆盖默认⽅法sleep()

publicclassCatimplementsAnimal{

@Override

publicvoidmakeSound() {

System.out.println("Meow");

}

}

### 接⼝和抽象类各有优缺点，在接⼝和抽象类的选择上，必须遵守这样⼀个原则：

### ⾏为模型应该总是通过接⼝⽽不是抽象类定义，所以通常是优先选⽤接⼝，尽量少⽤抽象类。

### 选择抽象类的时候通常是如下情况：需要定义⼦类的⾏为，⼜要为⼦类提供通⽤的功能。

## 接⼝和抽象类有什么区别？

### Java是⼀⻔⾯向对象的编程语⾔，⾯向对象的编程语⾔有四⼤特征：抽象、封装、继承和多态。⽽本

### ⽂介绍的接⼝和抽象类就是⾯向对象编程中“抽象”的具体实现，也就是说接⼝和抽象类都是⽤来定

### 义实体类的公共⾏为的，它们是对实体类（对象）更⾼层次的抽象。

### 接⼝

### 接⼝是Java语⾔中的⼀个抽象类型，⽤于定义对象的公共⾏为。它的创建关键字是interface，在接

### ⼝的实现中可以定义⽅法和常量，其普通⽅法是不能有具体的代码实现的，⽽在JDK8之后，接⼝中

### 可以创建static和default⽅法了，并且这两种⽅法可以有默认的⽅法实现，如下代码所⽰：

代码块

publicinterfaceInterface\_1{

intcount=1;

voidsayHi();

// default⽅法

defaultvoidprint() {

System.out.println("Do print method.");

}

// static⽅法

staticvoidsmethod() {

System.out.println("Do static method.");

}

}

### 接下来，创建⼀个类来实现以上接⼝：

代码块

19

20

21

22

23

24

25

26

27

1

2

3

4

5

6

7

8

9

10

11

12

# 第 22 页

publicclassInterfaceImpl\_1implementsInterface\_1{

@Override

publicvoidsayHi() {

System.out.println("Hi,I am InterfaceImpl 1.");

}

publicstaticvoidmain(String[] args) {

InterfaceImpl\_1inter=newInterfaceImpl\_1();

inter.sayHi();

//调⽤接⼝中static⽅法

InterfaceExample.smethod();

//调⽤接⼝中的常量count

System.out.println(InterfaceExample.count);

}

}

### 以上程序的执⾏结果如下：

### 通过上述代码和执⾏结果我们可以得出以下结论：

### 13.

### JDK8中接⼝可以定义static和default⽅法，并且这两种⽅法可以包含具体的代码实现。

## •

### 2.实现接⼝要使⽤implements关键字。

## •

### 3.接⼝不能直接实例化。

### 14.

### 接⼝中定义的变量默认为publicstaticfinal类型。

### 15.

### ⼦类可以不重写接⼝中的static和default⽅法，不重写的情况下，默认调⽤的是接⼝的⽅法实

### 现。

### 抽象类

### 抽象类和接⼝类似，它也是⽤来定义对象的公共⾏为的，并且它也不能直接实例化，抽象类的实现关

### 键字为abstractclass，⼦类⽤extends关键字继承⽗类。抽象类的使⽤如下：

代码块

publicabstractclassAbstractExample{

//定义普通变量

intcount=2;

//定义私有变量

privatestaticinttotal=10;

//定义抽象⽅法

publicabstractvoidmethodA();

//定义普通⽅法

publicvoidmethodB() {

1

2

3

4

5

6

7

8

9

10

11

12

13

14

1

2

3

4

5

6

7

8

9



*图片 1 (190x53)*

# 第 23 页

System.out.println("Hi,methodB.");

}

}

接下来使⽤⼀个普通类继承上⾯的抽象类：

代码块

publicclassAbstractSonextendsAbstractExample{

@Override

publicvoidmethodA() {

System.out.println("Hi,method A.");

}

publicstaticvoidmain(String[] args) {

AbstractSonabs=newAbstractSon();

//抽象类中的变量重新赋值

abs.count =666;

System.out.println(abs.count);

//抽象类中的抽象⽅法

abs.methodA();

//抽象类中的普通⽅法

abs.methodB();

}

}

以上程序的执⾏结果如下：

通过上述代码和执⾏结果可以得出以下结论：

16.抽象类使⽤abstract关键字声明。

## •

2.抽象类中可以包含普通⽅法和抽象⽅法，抽象⽅法不能有具体的代码实现。

## •

3.抽象类需要使⽤extends关键字实现继承。

17.抽象类不能直接实例化。

18.抽象类中属性控制符⽆限制，可以定义private类型的属性。

### 接⼝和抽象类的区别主要体现在以下7个⽅⾯。

区别1：定义关键字不同

接⼝使⽤关键字interface来定义。抽象类使⽤关键字abstract来定义。

区别2：继承或实现的关键字不同

接⼝使⽤implements关键字定义其具体实现。抽象类使⽤extends关键字实现继承。

区别3：⼦类扩展的数量不同

10

11

12

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

# 第 24 页

接⼝的实现类可以有多个，如下图所⽰：

⽽抽象类的⼦类，只能继承⼀个抽象类，如下图所⽰，继承多个抽象类就会报错：

在Java语⾔中，⼀个类只能继承⼀个⽗类（单继承），但可以实现多个接⼝。

区别4：属性访问控制符不同

接⼝中属性的访问控制符只能是public，

接⼝中的属性默认是publicstaticfinal修饰的。

抽象类中的属性访问控制符⽆限制，可为任意控制符

区别5：⽅法控制符不同

接⼝中⽅法的默认控制符是public，并且不能定义为其他控制符

抽象类中的⽅法控制符⽆限制，其中抽象⽅法不能使⽤private修饰

区别6：⽅法实现不同

接⼝中普通⽅法不能有具体的⽅法实现，在JDK8之后static和default⽅法必须有⽅法实现

从上述结果可以看出：static或default⽅法如果没有⽅法实现就会报错，⽽普通⽅法如果有⽅法实现

就会报错。

抽象类中普通⽅法可以有⽅法实现，抽象⽅法不能有⽅法实现

从上述结果可以看出：抽象类中的普通⽅法如果没有⽅法实现就会报错，⽽抽象⽅法如果有⽅法实现

则会报错。

区别7：静态代码块使⽤不同

接⼝中不能使⽤静态代码块

抽象类中可以使⽤静态代码块

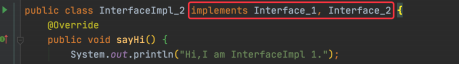
总结

接⼝和抽象类都是⽤来定义对象的公共⾏为的，但⼆者有以下7点不同：

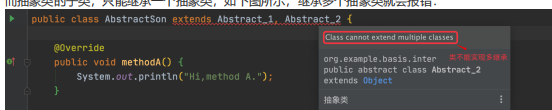
19.定义的关键字不同。

20.⼦类继承或实现关键字不同。

21.类型扩展不同：抽象类是单继承，⽽接⼝是多继承。



*图片 1 (459x64)*



*图片 2 (552x110)*

# 第 25 页

22.⽅法访问控制符：抽象类⽆限制，只是抽象类中的抽象⽅法不能被private修饰；⽽接⼝有限制，

接⼝默认的是public控制符。

23.属性⽅法控制符：抽象类⽆限制，⽽接⼝有限制，接⼝默认的是public控制符。

24.⽅法实现不同：抽象类中的普通⽅法必须有实现，抽象⽅法必须没有实现；⽽接⼝中普通⽅法不

能

有实现，但在JDK8中的static和defualt⽅法必须有实现。

25.静态代码块的使⽤不同：抽象类可以有静态代码块，⽽接⼝不能有。

### 普通类和抽象类有哪些区别？

普通类不能包含抽象⽅法，抽象类可以包含抽象⽅法。

抽象类不能直接实例化，普通类可以直接实例化。

### 抽象类能使⽤final修饰吗？

不能，定义抽象类就是让其他类继承的，如果定义为final该类就不能被继承，这样彼此就会产⽣⽭

盾，所以final不能修饰抽象类

### 创建⼀个对象⽤什么关键字？对象实例与对象引⽤有何不同？

new关键字，new创建对象实例（对象实例在堆内存中），对象引⽤指向对象实例（对象引⽤存放在

栈内存中）。

⼀个对象引⽤可以指向0个或1个对象（⼀根绳⼦可以不系⽓球，也可以系⼀个⽓球）;

⼀个对象可以有n个引⽤指向它（可以⽤n条绳⼦系住⼀个⽓球）

### Java中，什么是构造⽅法？什么是构造⽅法重载？什么是复制构造⽅法？

当新对象被创建的时候，构造⽅法会被调⽤。每⼀个类都有构造⽅法。在程序员没有给类提供构造⽅

法的情况下，Java编译器会为这个类创建⼀个默认的构造⽅法。

Java中构造⽅法重载和⽅法重载很相似。可以为⼀个类创建多个构造⽅法。每⼀个构造⽅法必须有它

⾃⼰唯⼀的参数列表。

Java不⽀持像C++中那样的复制构造⽅法，这个不同点是因为如果你不⾃⼰写构造⽅法的情况下，

Java不会创建默认的复制构造⽅法。

### Java⽀持多继承么？

Java中类不⽀持多继承，只⽀持单继承（即⼀个类只有⼀个⽗类）。但是java中的接⼝⽀持多继承，

即⼀个⼦接⼝可以有多个⽗接⼝。（接⼝的作⽤是⽤来扩展对象的功能，⼀个⼦接⼝继承多个⽗接

⼝，说明⼦接⼝扩展了多个功能，当类实现接⼝时，类就扩展了相应的功能）。

## 变量与⽅法

# 第 26 页

### 成员变量与局部变量的区别有哪些

变量：在程序执⾏的过程中，在某个范围内其值可以发⽣改变的量。从本质上讲，变量其实是内存中

的⼀⼩块区域

成员变量：⽅法外部，类内部定义的变量

局部变量：类的⽅法中的变量。

成员变量和局部变量的区别

作⽤域

## •

成员变量：针对整个类有效。

## •

局部变量：只在某个范围内有效。(⼀般指的就是⽅法,语句体内)

存储位置

## •

成员变量：随着对象的创建⽽存在，随着对象的消失⽽消失，存储在堆内存中。

## •

局部变量：在⽅法被调⽤，或者语句被执⾏的时候存在，存储在栈内存中。当⽅法调⽤完，或者语

句结束后，就⾃动释放。

⽣命周期

## •

成员变量：随着对象的创建⽽存在，随着对象的消失⽽消失

## •

局部变量：当⽅法调⽤完，或者语句结束后，就⾃动释放。

初始值

## •

成员变量：有默认初始值。

## •

局部变量：没有默认初始值，使⽤前必须赋值。

使⽤原则

在使⽤变量时需要遵循的原则为：就近原则

⾸先在局部范围找，有就使⽤；接着在成员位置找。

### 在Java中定义⼀个不做事且没有参数的构造⽅法的作⽤

Java程序在执⾏⼦类的构造⽅法之前，如果没有⽤super()来调⽤⽗类特定的构造⽅法，则会调⽤⽗类

中“没有参数的构造⽅法”。因此，如果⽗类中只定义了有参数的构造⽅法，⽽在⼦类的构造⽅法中

⼜没有⽤super()来调⽤⽗类中特定的构造⽅法，则编译时将发⽣错误，因为Java程序在⽗类中找不到

没有参数的构造⽅法可供执⾏。

## •

失败案例：

代码块

publicclassParent{

privateintvalue;

publicParent(intvalue) {

1

2

3

# 第 27 页

this.value = value;

System.out.println("Parent constructor called with value: "+ value);

}

}

publicclassChildextendsParent{

publicChild() {

//这⾥没有显式调⽤⽗类的构造⽅法

System.out.println("Child constructor called");

}

publicstaticvoidmain(String[] args) {

Childchild=newChild();

}

}

### 报错：

代码块

Child.java:3: error: constructor Parent inclassParentcannot be applied to

given types;

publicChild() {

^

required:int

found: no arguments

reason: actual and formal argument lists differ in length

1error

### 解决办法是在⽗类⾥加上⼀个不做事且没有参数的构造⽅法。

代码块

publicclassParent{

privateintvalue;

publicParent() {

this.value =0;//或者其他默认值

System.out.println("Parent default constructor called");

}

publicParent(intvalue) {

this.value = value;

System.out.println("Parent constructor called with value: "+ value);

}

}

4

5

6

7

8

9

10

11

12

13

14

15

16

1

2

3

4

5

6

7

1

2

3

4

5

6

7

8

9

10

11

# 第 28 页

### 在调⽤⼦类构造⽅法之前会先调⽤⽗类没有参数的构造⽅法，其⽬的是？

帮助⼦类做初始化⼯作。

### ⼀个类的构造⽅法的作⽤是什么？若⼀个类没有声明构造⽅法，改程序能正确执

### ⾏吗？为什么？

主要作⽤是完成对类对象的初始化⼯作。可以执⾏。因为⼀个类即使没有声明构造⽅法也会有默认的

不带参数的构造⽅法。

### 构造⽅法有哪些特性？

## •

名字与类名相同；

## •

没有返回值，但不能⽤void声明构造函数；

## •

⽣成类的对象时⾃动执⾏，⽆需调⽤。

### 静态变量和实例变量区别

## •

静态变量：静态变量由于不属于任何实例对象，属于类的，所以在内存中只会有⼀份，在类的加载

过程中，JVM只为静态变量分配⼀次内存空间。

## •

实例变量：每次创建对象，都会为每个对象分配成员变量内存空间，实例变量是属于实例对象的，

在内存中，创建⼏次对象，就有⼏份成员变量。

### 静态变量与普通变量区别

static变量也称作静态变量，静态变量和⾮静态变量的区别是：静态变量被所有的对象所共享，在内存

中只有⼀个副本，它当且仅当在类初次加载时会被初始化。⽽⾮静态变量是对象所拥有的，在创建对

象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。

还有⼀点就是static成员变量的初始化顺序按照定义的顺序进⾏初始化。

### 静态⽅法和实例⽅法有何不同？

静态⽅法和实例⽅法的区别主要体现在两个⽅⾯：

26.在外部调⽤静态⽅法时，可以使⽤"类名.⽅法名"的⽅式，也可以使⽤"对象名.⽅法名"的⽅式。⽽

实例⽅法只有后⾯这种⽅式。也就是说，调⽤静态⽅法可以⽆需创建对象。

27.静态⽅法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态⽅法），⽽不允许

访问实例成员变量和实例⽅法；实例⽅法则⽆此限制

在⼀个静态⽅法内调⽤⼀个⾮静态成员为什么是⾮法的？

由于静态⽅法可以不通过对象进⾏调⽤，因此在静态⽅法⾥，不能调⽤其他⾮静态变量，也不可以访

问⾮静态变量成员。

### 什么是⽅法的返回值？返回值的作⽤是什么？

# 第 29 页

⽅法的返回值是指我们获取到的某个⽅法体中的代码执⾏后产⽣的结果！（前提是该⽅法可能产⽣结

果）。返回值的作⽤:接收出结果，使得它可以⽤于其他的操作！

## Object根类

### equals()和hashcode()的联系

hashCode()是Object类的⼀个⽅法，返回⼀个哈希值。如果两个对象根据equal()⽅法⽐较相等，那么

调⽤这两个对象中任意⼀个对象的hashCode()⽅法必须产⽣相同的哈希值。

如果两个对象根据eqaul()⽅法⽐较不相等，那么产⽣的哈希值不⼀定相等(碰撞的情况下还是会相等

的。)

### hashCode()有什么⽤?

将对象放⼊到集合中时，⾸先判断要放⼊对象的hashcode是否已经在集合中存在，不存在则直接放⼊

集合。

如果hashcode相等，然后通过equal()⽅法判断要放⼊对象与集合中的任意对象是否相等：如果

equal()判断不相等，直接将该元素放⼊集合中，否则不放⼊。

### 有没有可能两个不相等的对象有相同的hashcode

有可能，两个不相等的对象可能会有相同的hashcode值，这就是为什么在hashmap中会有冲突。

如果两个对象相等，必须有相同的hashcode值，反之不成⽴。

### 可以在hashcode中使⽤随机数字吗?

不⾏，因为同⼀对象的hashcode值必须是相同的

### a==b与a.equals(b)有什么区别

如果a和b都是对象，则a==b是⽐较两个对象的引⽤，只有当a和b指向的是堆中的同⼀个对象才会

返回true，

⽽a.equals(b)是进⾏逻辑⽐较，所以通常需要重写该⽅法来提供逻辑⼀致性的⽐较。例如，String类

重写equals()⽅法，所以可以⽤于两个不同对象，但是包含的字⺟相同的⽐较。

基本类型⽐较⽤==，⽐较的是他们的值。默认下，对象⽤==⽐较时，⽐较的是内存地址，如果需要⽐

较对象内容，需要重写equal⽅法。

### 为什么要类型转换？

如果你在Java5下进⾏过编程的话，你⼀定不会陌⽣这⼀点，你不能直接地向集合(Collection)中放

⼊原始类型值，因为集合只接收对象。

通常这种情况下你的做法是，将这些原始类型的值转换成对象，然后将这些转换的对象放⼊集合中。

# 第 30 页

使⽤Integer、Double、Boolean等这些类，我们可以将原始类型值转换成对应的对象，但是从某些

程度可能使得代码不是那么简洁精炼。

为了让代码简练，Java5引⼊了具有在原始类型和对象类型⾃动转换的装箱和拆箱机制。

但是⾃动装箱和拆箱并⾮完美，在使⽤时需要有⼀些注意事项，如果没有搞明⽩⾃动装箱和拆箱，可

能会引起难以察觉的Bug。

## 内部类

### 什么是内部类？

在Java中，可以将⼀个类的定义放在另外⼀个类的定义内部，这就是内部类。内部类本⾝就是类的⼀

个属性，与其他属性定义⽅式⼀致。

### 内部类的分类有哪些

20内部类可以分为四种：成员内部类、局部内部类、匿名内部类和静态内部类。

### 静态内部类

定义在类内部的静态类，就是静态内部类。

代码块

publicclassOuter{

privatestaticintradius=1;

staticclassStaticInner{

publicvoidvisit() {

System.out.println("visit outer static variable:"+ radius);

}

}

}

静态内部类可以访问外部类所有的静态变量，⽽不可访问外部类的⾮静态变量；静态内部类的创建⽅

式，new外部类.静态内部类()，如下：

代码块

Outer.StaticInnerinner=newOuter.StaticInner();

inner.visit();

### 成员内部类

定义在类内部，成员位置上的⾮静态类，就是成员内部类。

1

2

3

4

5

6

7

8

1

2

# 第 31 页

代码块publicclassOuter{

privatestaticintradius=1;

privateintcount=2;

classInner{

publicvoidvisit() {

System.out.println("visit outer static variable:"+ radius);

System.out.println("visit outer variable:"+ count);

}

}

}

### 成员内部类可以访问外部类所有的变量和⽅法，包括静态和⾮静态，私有和公有。成员内部类依赖于

### 外部类的实例，它的创建⽅式外部类实例.new内部类()，如下：

代码块

Outerouter=newOuter();

Outer.Innerinner= outer.newInner();

inner.visit();

## 局部内部类

### 定义在⽅法中的内部类，就是局部内部类。

代码块

publicclassOuter{

privateintout\_a=1;

privatestaticintSTATIC\_b=2;

publicvoidtestFunctionClass(){

intinner\_c=3;

classInner{

privatevoidfun(){

System.out.println(out\_a);

System.out.println(STATIC\_b);

System.out.println(inner\_c);

}

}

Innerinner=newInner();

inner.fun();

}

publicstaticvoidtestStaticFunctionClass(){

intd=3;

classInner{

privatevoidfun(){

1

2

3

4

5

6

7

8

9

10

1

2

3

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

# 第 32 页

// System.out.println(out\_a);编译错误，定义在静态⽅法中的局部类不可

以

访问外部类的实例变量

System.out.println(STATIC\_b);

System.out.println(d);

}

}

Innerinner=newInner();

inner.fun();

}

}

### 定义在实例⽅法中的局部类可以访问外部类的所有变量和⽅法，定义在静态⽅法中的局部类只能访问

### 外

### 部类的静态变量和⽅法。局部内部类的创建⽅式，在对应⽅法内，new内部类()，如下：

代码块

publicstaticvoidtestStaticFunctionClass(){

classInner{

}

Innerinner=newInner();

}

## 匿名内部类

### 匿名内部类就是没有名字的内部类，⽇常开发中使⽤的⽐较多。

代码块

publicclassOuter{

privatevoidtest(finalinti) {

newService() {

publicvoidmethod() {

for(intj=0; j < i; j++) {

System.out.println("匿名内部类");

}

}

}.method();

}

}

//匿名内部类必须继承或实现⼀个已有的接⼝

interfaceService{

voidmethod();

}

20

21

22

23

24

25

26

27

28

29

1

2

3

4

5

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

# 第 33 页

除了没有名字，匿名内部类还有以下特点：

匿名内部类必须继承⼀个抽象类或者实现⼀个接⼝。

匿名内部类不能定义任何静态成员和静态⽅法。

当所在的⽅法的形参需要被匿名内部类使⽤时，必须声明为final。

匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接⼝的所有抽象⽅法。

匿名内部类创建⽅式：

代码块

new类/接⼝{

//匿名内部类实现部分

}

### 内部类的优点

我们为什么要使⽤内部类呢？因为它有以下优点：

## •

⼀个内部类对象可以访问创建它的外部类对象的内容，包括私有数据！

## •

内部类不为同⼀包的其他类所⻅，具有很好的封装性；

## •

内部类有效实现了“多重继承”，优化java单继承的缺陷。

## •

匿名内部类可以很⽅便的定义回调。

### 内部类有哪些应⽤场景

28.⼀些多算法场合

29.解决⼀些⾮⾯向对象的语句块。

30.适当使⽤内部类，使得代码更加灵活和富有扩展性。

31.当某个类除了它的外部类，不再被其他的类使⽤时。

### 局部内部类和匿名内部类访问局部变量的时候，为什么变量必须要加上final？

局部内部类和匿名内部类访问局部变量的时候，为什么变量必须要加上final呢？它内部原理是什么

呢？

先看这段代码：

代码块

publicclassOuter{

voidoutMethod(){

finalinta=10;

1

2

3

1

2

3

# 第 34 页

classInner{

voidinnerMethod(){

System.out.println(a);

}

}

}

}

### 以上例⼦，为什么要加final呢？是因为⽣命周期不⼀致，局部变量直接存储在栈中，当⽅法执⾏结束

### 后，⾮final的局部变量就被销毁。⽽局部内部类对局部变量的引⽤依然存在，如果局部内部类要调⽤

### 局部变量时，就会出错。加了final，可以确保局部内部类使⽤的变量与外层的局部变量区分开，解决

### 了这个问题。

### 内部类相关，看程序说出运⾏结果

代码块

publicclassOuter{

privateintage=12;

classInner{

privateintage=13;

publicvoidprint() {

intage=14;

System.out.println("局部变量："+ age);

System.out.println("内部类变量："+this.age);

System.out.println("外部类变量："+ Outer.this.age);

}

}

publicstaticvoidmain(String[] args) {

Outer.Innerin=newOuter().newInner();

in.print();

}

}

### 运⾏结果：

代码块

局部变量：14

内部类变量：13

外部类变量：12

## 重写与重载

4

5

6

7

8

9

10

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

1

2

3

# 第 35 页

构造器（constructor）是否可被重写（override）

构造器不能被继承，因此不能被重写，但可以被重载。

### 重载（Overload）和重写（Override）的区别。重载的⽅法能否根据返回类型进

### ⾏区分？

⽅法的重载和重写都是实现多态的⽅式，区别在于前者实现的是编译时的多态性，⽽后者实现的是运

⾏时的多态性。

### •

重载：发⽣在同⼀个类中，⽅法名相同参数列表不同（参数类型不同、个数不同、顺序不同），与

⽅法返回值和访问修饰符⽆关，即重载的⽅法不能根据返回类型进⾏区分

### •

重写：发⽣在⽗⼦类中，⽅法名、参数列表必须相同，返回值⼩于等于⽗类，抛出的异常⼩于等于

⽗类，访问修饰符⼤于等于⽗类（⾥⽒代换原则）；如果⽗类⽅法访问修饰符为private则⼦类中就

不是重写。

## 值传递

### java是值传递还是引⽤传递

当⼀个对象被当作参数传递到⼀个⽅法后，此⽅法可改变这个对象的属性，并可返回变化后的结果，

那么这⾥到底是值传递还是引⽤传递

是值传递。

Java语⾔的⽅法调⽤只⽀持参数的值传递。

当⼀个对象实例作为⼀个参数被传递到⽅法中时，参数的值就是对该对象的引⽤。

对象的属性可以在被调⽤过程中被改变，但对对象引⽤的改变是不会影响到调⽤者的

### 为什么Java中只有值传递

⾸先回顾⼀下在程序设计语⾔中有关将参数传递给⽅法（或函数）的⼀些专业术语。

按值调⽤(callbyvalue)表⽰⽅法接收的是调⽤者提供的值，⽽按引⽤调⽤（callbyreference)表⽰

⽅法接收的是调⽤者提供的变量地址。

⼀个⽅法可以修改传递引⽤所对应的变量值，⽽不能修改传递值调⽤所对应的变量值。

它⽤来描述各种程序设计语⾔（不只是Java)中⽅法参数传递⽅式。

# 第 36 页

### Java程序设计语⾔总是采⽤按值调⽤。也就是说，⽅法得到的是所有参数值的⼀个拷⻉，也就是说，

### ⽅法不能修改传递给它的任何参数变量的内容。

### 下⾯通过3个例⼦来给⼤家说明

### example1

代码块

publicstaticvoidmain(String[] args) {

intnum1=10;

intnum2=20;

swap(num1, num2);

System.out.println("num1 = "+ num1);

System.out.println("num2 = "+ num2);

}

publicstaticvoidswap(inta,intb) {

inttemp= a;

a = b;

b = temp;

System.out.println("a = "+ a);

System.out.println("b = "+ b);

}

### 结果：

代码块

a =20

b =10

num1 =10

num2 =20

### 解析：

### 在swap⽅法中，a、b的值进⾏交换，并不会影响到num1、num2。因为，a、b中的值，只是从

### num1、num2的复制过来的。也就是说，a、b相当于num1、num2的副本，副本的内容⽆论怎么修

### 改，都不会影响到原件本⾝。

### 通过上⾯例⼦，我们已经知道了⼀个⽅法不能修改⼀个基本数据类型的参数，⽽对象引⽤作为参数就

### 不⼀样，请看example2.

### example2

代码块

publicstaticvoidmain(String[] args) {

1

2

3

4

5

6

7

8

9

10

11

12

13

14

1

2

3

4

1

# 第 37 页

int[] arr = {1,2,3,4,5};

System.out.println(arr[0]);

change(arr);

System.out.println(arr[0]);

}

publicstaticvoidchange(int[] array) {

//将数组的第⼀个元素变为0

array[0] =0;

}

### 结果：

代码块

1

0

### 解析：

### array被初始化arr的拷⻉也就是⼀个对象的引⽤，也就是说array和arr指向的时同⼀个数组对象。

### 因此，外部对引⽤对象的改变会反映到所对应的对象上。

### 通过example2我们已经看到，实现⼀个改变对象参数状态的⽅法并不是⼀件难事。理由很简单，⽅

### 法得到的是对象引⽤的拷⻉，对象引⽤及其他的拷⻉同时引⽤同⼀个对象。

### 很多程序设计语⾔（特别是，C++和Pascal)提供了两种参数传递的⽅式：值调⽤和引⽤调⽤。有些程

### 序员（甚⾄本书的作者）认为Java程序设计语⾔对对象采⽤的是引⽤调⽤，实际上，这种理解是不对

### 的。

### 由于这种误解具有⼀定的普遍性，所以下⾯给出⼀个反例来详细地阐述⼀下这个问题。

### example3

代码块

publicclassTest{

publicstaticvoidmain(String[] args) {

// TODO Auto-generated method stub

Students1=newStudent("⼩张");

Students2=newStudent("⼩李");

Test.swap(s1, s2);

System.out.println("s1:"+ s1.getName());

39System.out.println("s2:"+ s2.getName());

}

publicstaticvoidswap(Student x, Student y) {

Studenttemp= x;

x = y;

2

3

4

5

6

7

8

9

10

1

2

1

2

3

4

5

6

7

8

9

10

11

12

# 第 38 页

y = temp;

System.out.println("x:"+ x.getName());

System.out.println("y:"+ y.getName());

}

}

结果：

代码块

x:⼩李

y:⼩张

s1:⼩张

s2:⼩李

⽅法并没有改变存储在变量s1和s2中的对象引⽤。swap⽅法的

参数x和y被初始化为两个对象引⽤的拷⻉，这个⽅法交换的是这两个拷⻉

总结

Java程序设计语⾔对对象采⽤的不是引⽤调⽤，实际上，对象引⽤是按值传递的。

下⾯再总结⼀下Java中⽅法参数的使⽤情况：

⼀个⽅法不能修改⼀个基本数据类型的参数（即数值型或布尔型》

⼀个⽅法可以改变⼀个对象参数的状态。

⼀个⽅法不能让对象参数引⽤⼀个新的对象。

### 值传递和引⽤传递有什么区别

值传递：指的是在⽅法调⽤时，传递的参数是按值的拷⻉传递，传递的是值的拷⻉，也就是说传递后

就互不相关了。

引⽤传递：指的是在⽅法调⽤时，传递的参数是按引⽤进⾏传递，其实传递的引⽤的地址，也就是变

量所对应的内存空间的地址。传递的是值的引⽤，也就是说传递前和传递后都指向同⼀个引⽤（也就

是同⼀个内存空间）。

## Java包

### JDK中常⽤的包有哪些

java.lang：这个是系统的基础类；

java.io：这⾥⾯是所有输⼊输出有关的类，⽐如⽂件操作等；

13

14

15

16

17

1

2

3

4

# 第 39 页

java.nio：为了完善io包中的功能，提⾼io包中性能⽽写的⼀个新包；

java.net：这⾥⾯是与⽹络有关的类；

java.util：这个是系统辅助类，特别是集合类；

java.sql：这个是数据库操作的类。

## importjava和javax有什么区别

刚开始的时候JavaAPI所必需的包是java开头的包，javax当时只是扩展API包来说使⽤。

然⽽随着时间的推移，javax逐渐的扩展成为JavaAPI的组成部分。但是，将扩展从javax包移动到

java包将是太⿇烦了，最终会破坏⼀堆现有的代码。因此，最终决定javax包将成为标准API的⼀部

分。

所以，实际上java和javax没有区别。这都是⼀个名字。

## java中IO流分为⼏种?

按照流的流向分，可以分为输⼊流和输出流；

按照操作单元划分，可以划分为字节流和字符流；

按照流的⻆⾊划分为节点流和处理流。

JavaIo流共涉及40多个类，这些类看上去很杂乱，但实际上很有规则，⽽且彼此之间存在⾮常紧密的

联系，JavaI0流的40多个类都是从如下4个抽象类基类中派⽣出来的。

InputStream/Reader:所有的输⼊流的基类，前者是字节输⼊流，后者是字符输⼊流。

OutputStream/Writer:所有输出流的基类，前者是字节输出流，后者是字符输出流。

按操作⽅式分类结构图：

# 第 40 页

42按操作对象分类结构图：

## BIO,NIO,AIO有什么区别?

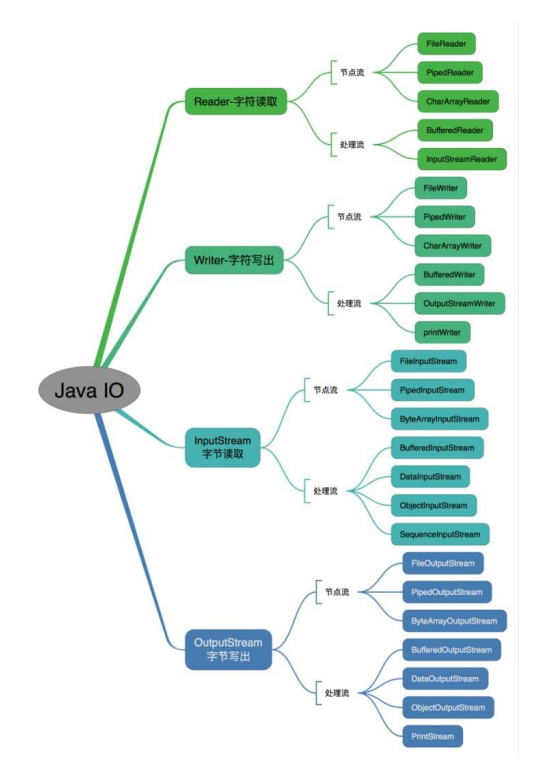
简答

BIO：BlockIO同步阻塞式IO，就是我们平常使⽤的传统IO，它的特点是模式简单使⽤⽅便，并发处

理能⼒低。

NIO：NonIO同步⾮阻塞IO，是传统IO的升级，客⼾端和服务器端通过Channel（通道）通讯，实

现了多路复⽤。



*图片 1 (533x765)*

# 第 41 页

AIO：AsynchronousIO是NIO的升级，也叫NIO2，实现了异步⾮堵塞IO，异步IO的操作基于事件

和回调机制。

详细回答

BIO(BlockingI/O):同步阻塞I/O模式，数据的读取写⼊必须阻塞在⼀个线程内等待其完成。在活动连

接数不是特别⾼（⼩于单机1000）的情况下，这种模型是⽐较不错的，可以让每⼀个连接专注于⾃⼰

的I/O并且编程模型简单，也不⽤过多考虑系统的过载、限流等问题。

线程池本⾝就是⼀个天然的漏⽃，可以缓冲⼀些系统处理不了的连接或请求。但是，当⾯对⼗万甚⾄

百万级连接的时候，传统的BIO模型是⽆能为⼒的。因此，我们需要⼀种更⾼效的I/O处理模型来应

对更⾼的并发量。

NIO(NewI/O):

NIO是⼀种同步⾮阻塞的I/O模型，在Java1.4中引⼊了NIO框架，对应java.nio

包，提供了Channel,Selector，Buffer等抽象。NIO中的N可以理解为Non-blocking，不单纯是

New。它⽀持⾯向缓冲的，基于通道的I/O操作⽅法。

NIO提供了与传统BIO模型中的Socket和ServerSocket相对应的SocketChannel和

ServerSocketChannel两种不同的套接字通道实

现,两种通道都⽀持阻塞和⾮阻塞两种模式。阻塞模式使⽤就像传统中的⽀持⼀样，⽐较简单，但是性

能和可靠性都不好；⾮阻塞模式正好与之相反。对于低负载、低并发的应⽤程序，可以使⽤同步阻塞

I/O来提升开发速率和更好的维护性；对于⾼负载、⾼并发的（⽹络）应⽤，应使⽤NIO的⾮阻塞模式

来开发

AIO(AsynchronousI/O):

AIO也就是NIO2。在Java7中引⼊了NIO的改进版NIO2,它是异步⾮阻塞的IO模型。异步IO是基于

事件和回调机制实现的，也就是应⽤操作之后会直接返回，不会堵塞在那⾥，当后台处理完成，操作

系统会通知相应的线程进⾏后续的操作。

AIO是异步IO的缩写，虽然NIO在⽹络操作中，提供了⾮阻塞的⽅法，但是NIO的IO⾏为还是同步

的。

对于NIO来说，我们的业务线程是在IO操作准备好时，得到通知，接着就由这个线程⾃⾏进⾏IO操

作，

IO操作本⾝是同步的。查阅⽹上相关资料，我发现就⽬前来说AIO的应⽤还不是很⼴泛，Netty之前也

尝试使⽤过AIO，不过⼜放弃了。

## Files的常⽤⽅法都有哪些？

Files.exists()：检测⽂件路径是否存在。

Files.createFile()：创建⽂件。

Files.createDirectory()：创建⽂件夹。

Files.delete()：删除⼀个⽂件或⽬录。

# 第 42 页

Files.copy()：复制⽂件。

Files.move()：移动⽂件。

Files.size()：查看⽂件个数。

Files.read()：读取⽂件。

Files.write()：写⼊⽂件。

## 反射与动态代理

### 反射

### 什么是反射机制？

简单说，反射机制值得是程序在运⾏时能够获取⾃⾝的信息。在java中，只要给定类的名字，那么就

可以通过反射机制来获得类的所有信息。

JAVA反射机制是在运⾏状态中，对于任意⼀个类，都能够知道这个类的所有属性和⽅法；

对于任意⼀个对象，都能够调⽤它的任意⼀个⽅法和属性；这种动态获取的信息以及动态调⽤对象的

⽅法的功能称为java语⾔的反射机制。

静态编译和动态编译

静态编译：在编译时确定类型，绑定对象

动态编译：运⾏时确定类型，绑定对象

### 反射机制优缺点

优点：运⾏期类型的判断，动态加载类，提⾼代码灵活度。

缺点：性能瓶颈：反射相当于⼀系列解释操作，通知JVM要做的事情，性能⽐直接的java代码要慢很

多。

### 反射机制的应⽤场景有哪些？

反射是框架设计的灵魂。

在我们平时的项⽬开发过程中，基本上很少会直接使⽤到反射机制，但这不能说明反射机制没有⽤，

实际上有很多设计、开发都与反射机制有关，例如模块化的开发，通过反射去调⽤对应的字节码；动

态代理设计模式也采⽤了反射机制，还有我们⽇常使⽤的Spring∕Hibernate等框架也⼤量使⽤到了

反射机制。

举例：

①我们在使⽤JDBC连接数据库时使⽤Class.forName()通过反射加载数据库的驱动程序；

②Spring框架也⽤到很多反射机制，最经典的就是xml的配置模式。Spring通过XML配置模式装载

# 第 43 页

Bean的过程：1)将程序内所有XML或Properties配置⽂件加载⼊内存中;2)Java类⾥⾯解析xml或

properties⾥⾯的内容，得到对应实体类的字节码字符串以及相关的属性信息;

3)使⽤反射机制，根据这个字符串获得某个类的Class实例;

4)动态配置实例的属性

## Java获取反射的三种⽅法

1.通过new对象实现反射机制2.通过路径实现反射机制3.通过类名实现反射机制

## 哪⾥⽤到反射机制？

jdbc中有⼀⾏代码：

代码块

Class.forName('com.MySQL.jdbc.Driver.class').newInstance();

那个时候只知道⽣成驱动对象实例，后来才知道，这就是反射，现在

很多框架都⽤到反射机制，hibernate，struts都是⽤反射机制实现的。

## 反射机制的优缺点？

静态编译：在编译时确定类型，绑定对象

动态编译：运⾏时确定类型，绑定对象。动态编译最⼤限度的发挥了java的灵活性，体现了多态的应

⽤，有利于降低类之间的耦合性。

⼀句话，反射机制的优点就是可以实现动态创建对象和编译，体现出很⼤的灵活性，特别是在J2EE的

开发中

它的灵活性就表现的⼗分明显。⽐如，⼀个⼤型的软件，不可能⼀次就把把它设计的很完美，当这个

程序编

译后，发布了，当发现需要更新某些功能时，我们不可能要⽤⼾把以前的卸载，再重新安装新的版

本，假如

这样的话，这个软件肯定是没有多少⼈⽤的。采⽤静态的话，需要把整个程序重新编译⼀次才可以实

现功能的更新，⽽采⽤反射机制的话，它就可以不⽤卸载，只需要在运⾏时才动态的创建和编译，就

可以实现该功能。

它的缺点是对性能有影响。使⽤反射基本上是⼀种解释操作，我们可以告诉JVM，我们希望做什么并

且它满⾜我们的要求。这类操作总是慢于只直接执⾏相同的操作。

## 反射的功能：

在运⾏时构造⼀个类的对象。

1

# 第 44 页

判断⼀个类所具有的成员变量和⽅法。

调⽤⼀个对象的⽅法。

⽣成动态代理。

反射的应⽤很多，很多框架都有⽤到：

### 反射的⽤途：

Spring框架的IoC基于反射创建对象和设置依赖属性。

SpringMVC的请求调⽤对应⽅法，也是通过反射。

JDBC的Class#forName(StringclassName)⽅法，也是使⽤反射。

### 反射中，Class.forName和ClassLoader区别

java中class.forName()和classLoader都可⽤来对类进⾏加载。

class.forName()前者除了将类的.class⽂件加载到jvm中之外，还会对类进⾏解释，执⾏类中的static

块。

⽽classLoader只⼲⼀件事情，就是将.class⽂件加载到jvm中，不会执⾏static中的内容,只有在

newInstance才会去执⾏static块。

## 动态代理

### 什么是动态代理

代理类在程序运⾏时创建的代理⽅式被成为动态代理。也就是说，这种情况下，代理类并不是在Java

代码中定义的，⽽是在运⾏时根据我们在Java代码中的“指⽰”动态⽣成的。

相⽐于静态代理，动态代理的优势在于可以很⽅便的对代理类的函数进⾏统⼀的处理，⽽不⽤修改每

个代理类的函数。

### Java动态代理的两种实现⽅法

jdk动态代理是由java内部的反射机制来实现的，cglib动态代理底层则是借助asm来实现的。总的来

说，反射机制在⽣成类的过程中⽐较⾼效，⽽asm在⽣成类之后的相关执⾏过程中⽐较⾼效（可以通

过将asm⽣成的类进⾏缓存，这样解决asm⽣成类过程低效问题）。

ASM（AbstractSyntaxModel）是⼀个⽤于Java字节码操作和分析的框架。它允许开发者直接操作

Java字节码，⽣成、转换和分析Java类⽂件

还有⼀点必须注意：jdk动态代理的应⽤前提，必须是⽬标类基于统⼀的接⼝。如果没有上述前提，

jdk动态代理不能应⽤。由此可以看出，jdk动态代理有⼀定的局限性，cglib这种第三⽅类库实现的动

态代理应⽤更加⼴泛，且在效率上更有优势。

# 第 45 页

### jdk动态代理是jdk原⽣就⽀持的⼀种代理⽅式，它的实现原理，就是通过让target类和代理类实现同⼀

### 接⼝，代理类持有target对象，来达到⽅法拦截的作⽤，这样通过接⼝的⽅式有两个弊端，⼀个是必须

### 保证target类有接⼝，第⼆个是如果想要对target类的⽅法进⾏代理拦截，那么就要保证这些⽅法都要

### 在接⼝中声明，实现上略微有点限制。

代码块

importjava.lang.reflect.Method;

publicinterfaceMyService{

voidserve();

}

publicclassMyServiceImplimplementsMyService{

@Override

publicvoidserve() {

System.out.println("Serving...");

}

}

importjava.lang.reflect.Method;

publicclassMyInvocationHandlerimplements

java.lang.reflect.InvocationHandler {

privateObject target;

publicMyInvocationHandler(Object target) {

this.target = target;

}

@Override

publicObjectinvoke(Object proxy, Method method, Object[] args)throws

Throwable {

System.out.println("Before method call...");

Objectresult= method.invoke(target, args);

System.out.println("After method call...");

returnresult;

}

}

importjava.lang.reflect.Proxy;

publicclassProxyDemo{

publicstaticvoidmain(String[] args) {

//⽬标对象

MyServicetarget=newMyServiceImpl();

//创建代理对象

MyServiceproxy= (MyService) Proxy.newProxyInstance(

target.getClass().getClassLoader(),//类加载器

target.getClass().getInterfaces(),//接⼝数组

newMyInvocationHandler(target)//代理处理器

);

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

34

35

36

37

# 第 46 页

//通过代理对象调⽤⽅法

proxy.serve();

}

}

### Cglib是⼀个优秀的动态代理框架，它的底层使⽤ASM在内存中动态的⽣成被代理类的⼦类，使⽤

### CGLIB即使代理类没有实现任何接⼝也可以实现动态代理功能。CGLIB具有简单易⽤，它的运⾏速度要

### 远远快于JDK的Proxy动态代理：

代码块

<dependencies>

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>3.3.0</version>

</dependency>

<dependency>

<groupId>asm</groupId>

<artifactId>asm</artifactId>

<version>3.3.1</version>

</dependency>

</dependencies>

packagecom.chz.myproxy.dynamicproxy;

publicclassBookFacade{

publicvoidaddBook() {

System.out.println("添加⼀本书...");

}

publicvoidremoveBook() {

System.out.println("删除⼀本书...");

}

}

代码块

packagecom.chz.myproxy.dynamicproxy;

importnet.sf.cglib.proxy.MethodInterceptor;

importnet.sf.cglib.proxy.MethodProxy;

importnet.sf.cglib.proxy.Enhancer;

importnet.sf.cglib.proxy no.1.0;

importjava.lang.reflect.Method;

publicclassBookFacadeCglibProxyimplementsMethodInterceptor{

@Override

38

39

40

41

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

1

2

3

4

5

6

7

8

# 第 47 页

publicObjectintercept(Object obj, Method method, Object[] args,

MethodProxy proxy)throwsThrowable {

System.out.println("代理⽅法执⾏前："+ method.getName());

Objectresult= proxy.invokeSuper(obj, args);//调⽤原⽅法

System.out.println("代理⽅法执⾏后："+ method.getName());

returnresult;

}

publicstaticObjectgetProxyInstance(Object target) {

Enhancerenhancer=newEnhancer();

enhancer.setSuperclass(target.getClass());//设置被代理类

enhancer.setCallback(this);//设置回调函数

returnenhancer.create();//创建代理对象

}

}

### cglib的原理是什么

CGLIB动态代理的核⼼思想是通过继承⽬标类来⽣成代理类。具体来说，CGLIB会动态⽣成⼀个⽬标类

的⼦类，并覆盖⼦类中⽗类⽅法的实现，从⽽实现对⽅法调⽤的拦截和增强

### 为什么要⽤动态代理

他可以在不修改别代理对象代码的基础上，通过扩展代理类，进⾏⼀些功能的附加与增强

### 静态代理与动态代理的区别

动态代理使我们免于去重写接⼝中的⽅法，⽽着重于去扩展相应的功能或是⽅法的增强，与静态代理

相⽐简单了不少，减少了项⽬中的业务量

### jdk动态代理机制

Proxy这个类的作⽤就是⽤来动态创建⼀个代理对象的类。每⼀个动态代理类都必须要实现

InvocationHandler这个接⼝，并且每个代理类的实例都关联到了⼀个handler，当我们通过代理对象

调⽤⼀个⽅法的时候，这个⽅法的调⽤就会被转发为由InvocationHandler这个接⼝的invoke⽅法来

进⾏调⽤。

## 序列化

### 什么是Java序列化？

序列化就是⼀种⽤来处理对象流的机制，所谓对象流也就是将对象的内容进⾏流化。

可以对流化后的对象进⾏读写操作，也可将流化后的对象传输于⽹络之间。

9

10

11

12

13

14

15

16

17

18

19

20

21

# 第 48 页

序列化是为了解决在对对象流进⾏读写操作时所引发的问题。

反序列化的过程，则是和序列化相反的过程。

另外，我们不能将序列化局限在Java对象转换成⼆进制数组，例如说，我们将⼀个Java对象，转换

成JSON字符串，或者XML字符串，这也可以理解为是序列化。

### 如何实现Java序列化？

如下的⽅式，就是Java内置的序列化⽅案，实际场景下，我们可以⾃定义序列化的⽅案，例如说

GoogleProtobuf。

将需要被序列化的类，实现Serializable接⼝，该接⼝没有需要实现的⽅法，implements

Serializable

只是为了标注该对象是可被序列化的。

序列化

然后，使⽤⼀个输出流(如：FileOutputStream)来构造⼀个ObjectOutputStream(对象流)对象

接着，使⽤ObjectOutputStream对象的#writeObject(Objectobj)⽅法，就可以将参数为obj的对

象写出(即保存其状态)。

反序列化

要恢复的话则⽤输⼊流。

### Java序列话中，如果有些字段不想进⾏序列化怎么办？

对于不想进⾏序列化的变量，使⽤transient关键字修饰。

当对象被序列化时，阻⽌实例中那些⽤此关键字修饰的的变量序列化。

当对象被反序列化时，被transient修饰的变量值不会被持久化和恢复。

transient只能修饰变量，不能修饰类和⽅法。

## 常⽤API

### String相关

### 字符型常量和字符串常量的区别

32.形式上:字符常量是单引号引起的⼀个字符字符串常量是双引号引起的若⼲个字符

33.含义上:字符常量相当于⼀个整形值(ASCII值),可以参加表达式运算字符串常量代表⼀个地址值(该

字符串在内存中存放位置)

34.占内存⼤⼩字符常量只占⼀个字节字符串常量占若⼲个字节(⾄少⼀个字符结束标志)

### 什么是字符串常量池？

# 第 49 页

字符串常量池位于堆内存中，专⻔⽤来存储字符串常量，可以提⾼内存的使⽤率，避免开辟多块空间

存储相同的字符串，在创建字符串时JVM会⾸先检查字符串常量池，如果该字符串已经存在池中，则

返回它的引⽤，如果不存在，则实例化⼀个字符串放到池中，并返回其引⽤。

## String是最基本的数据类型吗

不是。Java中的基本数据类型只有8个：byte、short、int、long、float、double、char、

boolean；除了基本类型（primitivetype），剩下的都是引⽤类型（referencetype），Java5以后

引⼊的枚举类型也算是⼀种⽐较特殊的引⽤类型。

这是很基础的东西，但是很多初学者却容易忽视，Java的8种基本数据类型中不包括String，基本数

据

类型中⽤来描述⽂本数据的是char，但是它只能表⽰单个字符，⽐如‘a’,‘好’之类的，如果要描

述⼀段⽂本，就需要⽤多个char类型的变量，也就是⼀个char类型数组，⽐如“你好”就是⻓度为2

的数组

char[]chars={‘你’,‘好’};

但是使⽤数组过于⿇烦，所以就有了String，String底层就是⼀个char类型的数组，只是使⽤的时候

开发者不需要直接操作底层数组，⽤更加简便的⽅式即可完成对字符串的使⽤。

## String有哪些特性

不变性：String是只读字符串，是⼀个典型的immutable对象，对它进⾏任何操作，其实都是创建⼀

个新的对象，再把引⽤指向该对象。

不变模式的主要作⽤在于当⼀个对象需要被多线程共享并频繁访问时，可以保证数据的⼀致性。

常量池优化：String对象创建之后，会在字符串常量池中进⾏缓存，如果下次创建同样的对象时，会

直接返回缓存的引⽤。

final：使⽤final来定义String类，表⽰String类不能被继承，提⾼了系统的安全性。

## String为什么是不可变的吗？

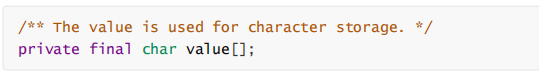
简单来说就是String类利⽤了final修饰的char类型数组存储字符，源码如下图所以：

## String真的是不可变的吗？

我觉得如果别⼈问这个问题的话，回答不可变就可以了。下⾯只是给⼤家看两个有代表性的例⼦：

35.String不可变但不代表引⽤不可以变

代码块



*图片 1 (539x71)*

# 第 50 页

/\*\* The value is used for character storage. \*/

privatefinalcharvalue[];

Stringstr="Hello";

str = str +" World";

System.out.println("str="+ str);

str=Hello World

### 结果：

代码块

str=HelloWorld

### 解析：

### 实际上，原来String的内容是不变的，只是str由原来指向"Hello"的内存地址转为指向"Hello

### World"的

### 内存地址⽽已，也就是说多开辟了⼀块内存区域给"HelloWorld"字符串。

### 2)通过反射是可以修改所谓的“不可变”对象

代码块

//创建字符串"HelloWorld"，并赋给引⽤s

Strings="HelloWorld";

System.out.println("s="+s);//HelloWorld

//获取String类中的value字段

FieldvalueFieldOfString=String.class.getDeclaredField("value");//改变value属性的

访问权限

valueFieldOfString.setAccessible(true);

//获取s对象上的value属性的值

char[]value=(char[])valueFieldOfString.get(s);

//改变value所引⽤的数组中的第5个字符

value[5]='\_';

System.out.println("s="+s);//Hello\_World

### 结果：

代码块

s=HelloWorld s=Hello\_World

### 解析：

1

2

3

4

5

6

1

1

2

3

4

5

6

7

8

9

10

11

1

# 第 51 页

### ⽤反射可以访问私有成员，然后反射出String对象中的value属性，进⽽改变通过获得的value引⽤改

### 变数组的结构。但是⼀般我们不会这么做，这⾥只是简单提⼀下有这个东西。

## 是否可以继承String类

### String类是final类，不可以被继承。

## Stringstr="i"与Stringstr=newString(“i”)⼀样吗？

### 不⼀样，因为内存的分配⽅式不⼀样。Stringstr="i"的⽅式，java虚拟机会将其分配到常量池中；⽽

### Stringstr=newString(“i”)则会被分到堆内存中。

## Strings=newString(“xyz”);创建了⼏个字符串对象

### 两个对象，⼀个是静态区的"xyz"，⼀个是⽤new创建在堆上的对象。

代码块

Stringstr1="hello";//str1指向静态区

Stringstr2=newString("hello");//str2指向堆上的对象Stringstr3="hello";

Stringstr4=newString("hello");

System.out.println(str1.equals(str2));//true

System.out.println(str2.equals(str4));//true

System.out.println(str1==str3);//true

System.out.println(str1==str2);//false

System.out.println(str2==str4);//false

System.out.println(str2=="hello");//false

str2=str1;

System.out.println(str2=="hello");//true

## 如何将字符串反转？

### 使⽤StringBuilder或者stringBuffer的reverse()⽅法。

### ⽰例代码：

代码块

//StringBufferreverse

StringBuffer stringBuffer=newStringBuffer();

stringBuffer.append("abcdefg");

System.out.println(stringBuffer.reverse());//gfedcba

//StringBuilderreverse

StringBuilder stringBuilder=newStringBuilder();

stringBuilder.append("abcdefg");

System.out.println(stringBuilder.reverse());//gfedcba

1

2

3

4

5

6

7

8

9

10

11

1

2

3

4

5

6

7

8

# 第 52 页

## 数组有没有length()⽅法？String有没有length()⽅法

数组没有length()⽅法，有length的属性。String有length()⽅法。JavaScript中，获得字符串的⻓

度

是通过length属性得到的，这⼀点容易和Java混淆。

## String类的常⽤⽅法都有那些？

indexOf()：返回指定字符的索引。

charAt()：返回指定索引处的字符。

replace()：字符串替换。

trim()：去除字符串两端空⽩。

split()：分割字符串，返回⼀个分割后的字符串数组。

getBytes()：返回字符串的byte类型数组。

length()：返回字符串⻓度。

toLowerCase()：将字符串转成⼩写字⺟。

toUpperCase()：将字符串转成⼤写字符。

substring()：截取字符串。

equals()：字符串⽐较。

## 在使⽤HashMap的时候，⽤String做key有什么好处？

HashMap内部实现是通过key的hashcode来确定value的存储位置，因为字符串是不可变的，所以

当创建字符串时，它的hashcode被缓存下来，不需要再次计算，所以相⽐于其他对象更快。

## String,StringBuffer和StringBuilder区别

String是字符串常量，final修饰：StringBuffer字符串变量(线程安全)；

StringBuilder字符串变量(线程不安全)。

String和StringBuffer

String和StringBuffer主要区别是性能：String是不可变对象，每次对String类型进⾏操作都等同于产

⽣了⼀个新的String对象，然后指向新的String对象。所以尽量不在对String进⾏⼤量的拼接操作，否

则会产⽣很多临时对象，导致GC开始⼯作，影响系统性能。

StringBuffer是对对象本⾝操作，⽽不是产⽣新的对象，因此在有⼤量拼接的情况下，我们建议使⽤

StringBuffer。

但是需要注意现在JVM会对String拼接做⼀定的优化：

Strings=“Thisisonly”+”simple”+”test”会被虚拟机直接优化成Strings=“Thisisonly

simpletest”，此时就不存在拼接过程。

# 第 53 页

StringBuffer和StringBuilder

StringBuffer是线程安全的可变字符串，其内部实现是可变数组。StringBuilder是jdk1.5新增的，其

功

能和StringBuffer类似，但是⾮线程安全。因此，在没有多线程问题的前提下，使⽤StringBuilder会取

得更好的性能。

### String和StringBuffer、StringBuilder的区别是什么？String为什么是不可变的

可变性

String类中使⽤字符数组保存字符串，privatefinalcharvalue[]，所以string对象是不可变的。

StringBuilder与StringBuffer都继承⾃AbstractStringBuilder类，在AbstractStringBuilder中也是使

⽤字符数组保存字符串，char[]value，这两种对象都是可变的。

线程安全性

String中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder是

StringBuilder

与StringBuffer的公共⽗类，定义了⼀些字符串的基本操作，如expandCapacity、append、insert、

indexOf等公共⽅法。StringBuffer对⽅法加了同步锁或者对调⽤的⽅法加了同步锁，所以是线程安全

的。StringBuilder并没有对⽅法进⾏加同步锁，所以是⾮线程安全的。

性能

每次对String类型进⾏改变的时候，都会⽣成⼀个新的String对象，然后将指针指向新的String对

象。

StringBuffer每次都会对StringBuffer对象本⾝进⾏操作，⽽不是⽣成新的对象并改变对象引⽤。相同

情况下使⽤StirngBuilder相⽐使⽤StringBuffer仅能获得10%~15%左右的性能提升，但却要冒多线

程不安全的⻛险。

对于三者使⽤的总结

如果要操作少量的数据⽤=String

单线程操作字符串缓冲区下操作⼤量数据=StringBuilder

多线程操作字符串缓冲区下操作⼤量数据=StringBuffer

## 包装类相关

### ⾃动装箱与拆箱

装箱：将基本类型⽤它们对应的引⽤类型包装起来；

拆箱：将包装类型转换为基本数据类型；

# 第 54 页

### int和Integer有什么区别

Java是⼀个近乎纯洁的⾯向对象编程语⾔，但是为了编程的⽅便还是引⼊了基本数据类型，但是为了

能

够将这些基本数据类型当成对象操作，Java为每⼀个基本数据类型都引⼊了对应的包装类型

（wrapperclass），int的包装类就是Integer，从Java5开始引⼊了⾃动装箱/拆箱机制，使得⼆者

可以相互转换。

Java为每个原始类型提供了包装类型：

原始类型:boolean，char，byte，short，int，long，float，double

包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

### Integera=127与Integerb=127相等吗

对于对象引⽤类型：==⽐较的是对象的内存地址。

对于基本数据类型：==⽐较的是值。

如果整型字⾯量的值在-128到127之间，那么⾃动装箱时不会new新的Integer对象，⽽是直接引⽤常

量池中的Integer对象，超过范围a1==b1的结果是false

代码块

publicstaticvoidmain(String[] args) {

Integera=newInteger(3);

Integerb=3;//将3⾃动装箱成Integer类型

intc=3;

System.out.println(a == b);// false两个引⽤没有引⽤同⼀对象

System.out.println(a == c);// true a⾃动拆箱成int类型再和c⽐较

System.out.println(b == c);// true

Integera1=128;

Integerb1=128;

System.out.println(a1 == b1);// false

Integera2=127;

Integerb2=127;

System.out.println(a2 == b2);// true

}

## 异常处理

### error和exception有什么区别

error表⽰系统级的错误，是java运⾏环境内部错误或者硬件问题，不能指望程序来处理这样的问题，

除了退出运⾏外别⽆选择，它是Java虚拟机抛出的。

1

2

3

4

5

6

7

8

9

10

11

12

13

14

# 第 55 页

exception表⽰程序需要捕捉、需要处理的异常，是由与程序设计的不完善⽽出现的问题，程序必须

处理的问题

## 运⾏时异常和⼀般异常有何不同

Java提供了两类主要的异常：runtimeException和checkedException

⼀般异常（checkedException）主要是指IO异常、SQL异常等。对于这种异常，JVM要求我们必须对

其进⾏catch处理，所以，⾯对这种异常，不管我们是否愿意，都是要写⼀⼤堆的catch块去处理可能

出现的异常。

运⾏时异常（runtimeException）我们⼀般不处理，当出现这类异常的时候程序会由虚拟机接管。⽐

如，我们从来没有去处理过NullPointerException，⽽且这个异常还是最常⻅的异常之⼀。

出现运⾏时异常的时候，程序会将异常⼀直向上抛，⼀直抛到遇到处理代码，如果没有catch块进⾏处

理，到了最上层，如果是多线程就有Thread.run()抛出，如果不是多线程那么就由main.run()抛出。

抛

出之后，如果是线程，那么该线程也就终⽌了，如果是主程序，那么该程序也就终⽌了。

其实运⾏时异常的也是继承⾃Exception，也可以⽤catch块对其处理，只是我们⼀般不处理罢了，也

就是说，如果不对运⾏时异常进⾏catch处理，那么结果不是线程退出就是主程序终⽌。

如果不想终⽌，那么我们就必须捕获所有可能出现的运⾏时异常。如果程序中出现了异常数据，但是

它

不影响下⾯的程序执⾏，那么我们就该在catch块⾥⾯将异常数据舍弃，然后记录⽇志。如果，它影响

到了下⾯的程序运⾏，那么还是程序退出⽐较好些。

## Java中异常处理机制的原理

Java通过⾯向对象的⽅式对异常进⾏处理，Java把异常按照不同的类型进⾏分类，并提供了良好的接

⼝。当⼀个⽅法出现异常后就会抛出⼀个异常对象，该对象中包含有异常信息，调⽤这个对象的⽅法

可

以捕获到这个异常并对异常进⾏处理。Java的异常处理是通过5个关键词来实现的：trycatchthrow

throwsfinally。

⼀般情况下是⽤try来执⾏⼀段程序，如果出现异常，系统会抛出（throws），我们可以通过它的类型

来捕捉它，或最后由缺省处理器来处理它（finally）。

try：⽤来指定⼀块预防所有异常的程序

catch：紧跟在try后⾯，⽤来捕获异常

throw：⽤来明确的抛出⼀个异常

throws：⽤来标明⼀个成员函数可能抛出的各种异常

finally：确保⼀段代码⽆论发⽣什么异常都会被执⾏的⼀段代码。

# 第 56 页

## 你平时在项⽬中是怎样对异常进⾏处理的。

（1）尽量避免出现runtimeException。例如对于可能出现空指针的代码，带使⽤对象之前⼀定要判

断⼀下该对象是否为空，必要的时候对runtimeException

也进⾏trycatch处理。

（2）进⾏trycatch处理的时候要在catch代码块中对异常信息进⾏记录，通过调⽤异常类的相关⽅法

获取到异常的相关信息，返回到web端，不仅要给⽤⼾良好

的⽤⼾体验，也要能帮助程序员良好的定位异常出现的位置及原因。例如，以前做的⼀个项⽬，程序

遇到异常⻚⾯会显⽰⼀个图⽚告诉⽤⼾哪些操作导致程序出现了什么异常，同时图⽚上有⼀个按钮⽤

来点击展⽰异常的详细信息给程序员看的。

## final、finally、finalize的区别

（1）、final⽤于声明变量、⽅法和类的，分别表⽰变量值不可变，⽅法不可覆盖，类不可以继承

（2）、finally是异常处理中的⼀个关键字，表⽰finally{}⾥⾯的代码⼀定要执⾏

（3）、finalize是Object类的⼀个⽅法，在垃圾回收的时候会调⽤被回收对象的此⽅法。

try()⾥⾯有⼀个return语句，那么后⾯的finally{}⾥⾯的code会不会被执⾏，什么时候执⾏，是在

return前还是return后？你曾经⾃定义实现过异常吗？怎么写的?

## throw和throws有什么区别？

throw关键字⽤来在程序中明确的抛出异常，相反，throws语句⽤来表明⽅法不能处理的异常。每⼀

个⽅法都必须要指定哪些异常不能处理，所以⽅法的调⽤者才能够确保处理可能发⽣的异常，多个异

常是⽤逗号分隔的。

## 异常处理的时候，finally代码块的重要性是什么？

⽆论是否抛出异常，finally代码块总是会被执⾏。就算是没有catch语句同时⼜抛出异常的情况下，

finally代码块仍然会被执⾏。最后要说的是，finally代码块主要⽤来释放资源，⽐如：I/O缓冲区，数

据库连接。

## 异常的使⽤的注意地⽅？

不要将异常处理⽤于正常的控制流（设计良好的API不应该强迫它的调⽤者为了正常的控制流⽽使⽤

异常）。

对可以恢复的情况使⽤受检异常，对编程错误使⽤运⾏时异常。

避免不必要的使⽤受检异常（可以通过⼀些状态检测⼿段来避免异常的发⽣）。

优先使⽤标准的异常。

每个⽅法抛出的异常都要有⽂档。

# 第 57 页

保持异常的原⼦性

不要在catch中忽略掉捕获到的异常。

### 请列出5个运⾏时异常？

NullPointerException

IndexOutOfBoundsException

ClassCastException

ArrayStoreException当你试图将错误类型的对象存储到⼀个对象数组时抛出的异常

BufferOverflowException写⼊的⻓度超出了允许的⻓度

## 元素排序Comparable和Comparator什么区别？

在Java语⾔中，Comparable和Comparator都是⽤来进⾏元素排序的，但⼆者有着本质的区别。它

们两也是常⻅的⾯试题，所以今天我们⼀起来盘它。

1.字⾯含义不同

我们先从⼆者的字⾯含义来理解它，Comparable翻译为中⽂是“⽐较”的意思，⽽Comparator

是“⽐较

器”的意思。Comparable是以-able结尾的，表⽰它⾃⾝具备着某种能⼒，⽽Comparator是以-or

结

尾，表⽰⾃⾝是⽐较的参与者，这是从字⾯含义先来理解⼆者的不同。

2.⽤法不同

⼆者都是顶级的接⼝，但拥有的⽅法和⽤法是不同的，下⾯我们分别来看。

### 2.1Comparable

Comparable接⼝只有⼀个⽅法compareTo，实现Comparable接⼝并重写compareTo⽅法就可以

实现某个类的排序了，它⽀持Collections.sort和Arrays.sort的排序。

在我们没有使⽤Comparable时，程序的执⾏是这样的：

代码块

publicclassComparableExample{

publicstaticvoidmain(String[] args) {

//创建对象

Personp1=newPerson(1,18,"Java");

Personp2=newPerson(2,22,"MySQL");

Personp3=newPerson(3,6,"Redis");

//添加到集合

1

2

3

4

5

6

7

# 第 58 页

List<Person> list =newArrayList<>();

list.add(p1);

list.add(p2);

list.add(p3);

//打印集合信息

list.forEach(p -> System.out.println(p.getName() +

"："+ p.getAge()));

}

}

//以下set/get/toString都使⽤的是lombok提供的注解

@Getter

@Setter

@ToString

classPerson{

privateintid;

privateintage;

privateString name;

publicPerson(intid,intage, String name) {

this.id = id;

this.age = age;

this.name = name;

}

}

### 程序执⾏结果如下：

### 从上图可以看出，当⾃定义类Person没有实现Comparable时，List集合是没有排序的，只能以元素

### 的插⼊顺序作为输出的顺序。

### 然⽽这个时候，⽼板有⼀个需求：需要根据Person对象的年龄age属性进⾏倒序，也就是根据age

### 属性从⼤到⼩进⾏排序，这个时候就可以请出，我们本⽂的主⻆：Comparable出场了。

### Comparable的使⽤是在⾃定义对象的类中实现Comparable接⼝，并重写compareTo⽅法来实现

### ⾃

### 定义排序规则的，具体实现代码如下：

代码块

publicclassComparableExample{

publicstaticvoidmain(String[] args) {

Personp1=newPerson(1,18,"Java");

Personp2=newPerson(2,22,"MySQL");

Personp3=newPerson(3,6,"Redis");

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

1

2

3

4

5



*图片 1 (95x60)*

# 第 59 页

List<Person> list =newArrayList<>();

list.add(p1);

list.add(p2);

list.add(p3);

//进⾏排序操作(根据Person类中compareTo中定义的排序规则)

Collections.sort(list);

list.forEach(p -> System.out.println(p.getName() +

"："+ p.getAge()));

}

}

//以下set/get/toString都使⽤的是lombok提供的注解实现的

@Getter

@Setter

@ToString

staticclassPersonimplementsComparable<Person> {

privateintid;

privateintage;

privateString name;

publicPerson(intid,intage, String name) {

this.id = id;

this.age = age;

this.name = name;

}

@Override

publicintcompareTo(Person p) {

returnp.getAge() -this.getAge();

}

}

### 程序的执⾏结果如下图所⽰：

### compareTo排序⽅法说明

### compareTo⽅法接收的参数p是要对⽐的对象，排序规则是⽤当前对象和要对⽐的对象进⾏⽐较，然

### 后返回⼀个int类型的值。正序从⼩到⼤的排序规则是：使⽤当前的对象值减去要对⽐对象的值；⽽倒

### 序从⼤到⼩的排序规则刚好相反：是⽤对⽐对象的值减去当前对象的值。

### 注意事项：如果⾃定义对象没有实现Comparable接⼝，那么它是不能使⽤Collections.sort⽅法进

### ⾏

### 排序的，编译器会提⽰如下错误：

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

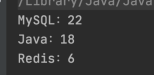
29

30

31

32

33



*图片 1 (154x75)*

# 第 60 页

## 2.2Comparator

### Comparator和Comparable的排序⽅法是不同的，Comparable排序的⽅法是compareTo，⽽

### Comparator排序的⽅法是compare，具体实现代码如下：

代码块

publicclassComparatorExample{

publicstaticvoidmain(String[] args) {

Personp1=newPerson(1,18,"Java");

Personp2=newPerson(2,22,"MySQL");

Personp3=newPerson(3,6,"Redis");

List<Person> list =newArrayList<>();

list.add(p1);

list.add(p2);

list.add(p3);

//进⾏排序操作(根据PersonComparator中定义的排序规则)

Collections.sort(list,newPersonComparator());

list.forEach(p -> System.out.println(p.getName() +

"："+ p.getAge()));

}

}

classPersonComparatorimplementsComparator<Person> {

@Override

publicintcompare(Person p1, Person p2) {

returnp2.getAge() - p1.getAge();

}

}

@Getter

@Setter

classPerson{

privateintid;

privateintage;

privateString name;

publicPerson(intid,intage, String name) {

this.id = id;

this.age = age;

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



*图片 1 (760x178)*

# 第 61 页

}

}

### 程序的执⾏结果如下图所⽰：

## 扩展：Comparator匿名类

### Comparator除了可以通过创建⾃定义⽐较器外，还可以通过匿名类的⽅式，更快速、便捷的完成⾃定

### 义⽐较器的功能，具体的代码实现如下：

代码块

publicclassComparatorExample{

publicstaticvoidmain(String[] args) {

//构建并添加数据

List<Person> list =newArrayList<>();

list.add(newPerson(1,18,"Java"));

list.add(newPerson(2,20,"MySQL"));

list.add(newPerson(3,6,"Redis"));

//使⽤Comparator匿名类的⽅式进⾏排序

list.sort(newComparator<Person>() {

@Override

publicintcompare(Person p1, Person p2) {

returnp2.getAge() - p1.getAge();

}

});

//打印集合数据

list.forEach(p -> System.out.println(p.getName() +

"："+ p.getAge()));

}

}

@Getter

@Setter

staticclassPerson{

privateintid;

privateintage;

privateString name;

publicPerson(intid,intage, String name) {

this.id = id;

this.age = age;

this.name = name;

}

}

### 程序的执⾏结果如下图所⽰：

32

33

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

# 第 62 页

### 3.使⽤的场景不同

通过上⾯⽰例的实现代码我们可以看出，使⽤Comparable必须要修改原有的类，也就是你要排序那

个类，就要在那个中实现Comparable接⼝并重写compareTo⽅法，所以Comparable更像是“对

内”进⾏排序的接⼝。

⽽Comparator的使⽤则不相同，Comparator⽆需修改原有类。也就是在最极端情况下，即使

Person类是第三⽅提供的，我们依然可以通过创建新的⾃定义⽐较器Comparator，来实现对第三⽅

类Person的排序功能。也就是说通过Comparator接⼝可以实现和原有类的解耦，在不修改原有类

的

情况下实现排序功能，所以Comparator可以看作是“对外”提供排序的接⼝。

## 总结

Comparable和Comparator都是⽤来实现元素排序的，它们⼆者的区别如下：

Comparable是“⽐较”的意思，⽽Comparator是“⽐较器”的意思；

Comparable是通过重写compareTo⽅法实现排序的，⽽Comparator是通过重写compare⽅

法实现排序的；

Comparable必须由⾃定义类内部实现排序⽅法，⽽Comparator是外部定义并实现排序的。

所以⽤⼀句话总结⼆者的区别：Comparable可以看作是“对内”进⾏排序接⼝，⽽Comparator

是“对外”进⾏排序的接⼝。

## 如何实现List集合去重？

List去重指的是将List中的重复元素删除掉的过程，此题⽬考察的是对List迭代器、Set集合和JDK

8中新特性的理解与灵活运⽤的能⼒。

List去重有以下3种实现思路：

36.⾃定义⽅法去重，通过循环判断当前的元素是否存在多个，如果存在多个，则删除此重复项，循

环整个集合最终得到的就是⼀个没有重复元素的List；

### •

2.使⽤Set集合去重，利⽤Set集合⾃⾝⾃带去重功能的特性，实现List的去重；

37.使⽤JDK8中Stream流的去重功能。

### 1.⾃定义去重

⾃定义去重的实现⽅法有两种，⾸先我们可以创建⼀个新集合，通过循环原集合判断循环的元素，是

否已存在于新集合，如果不存在则插⼊，否则就忽略，这样循环完，最终得到的新集合就是⼀个没有

重复

元素的集合，具体实现代码如下：

# 第 63 页

代码块importlombok.Data;

importjava.util.ArrayList;

importjava.util.List;

publicclassDistinctExample{

publicstaticvoidmain(String[] args) {

//创建并给List赋值

List<Person> list =newArrayList<>();

list.add(newPerson("李四","123456",20));

list.add(newPerson("张三","123456",18));

list.add(newPerson("王五","123456",22));

list.add(newPerson("张三","123456",18));

//去重操作

List<Person> newList =newArrayList<>(list.size());

list.forEach(i -> {

if(!newList.contains(i)) {//如果新集合中不存在则插⼊

newList.add(i);

}

});

//打印集合

newList.forEach(p -> System.out.println(p));

}

}

@Data

classPerson{

privateString name;

privateString password;

privateintage;

publicPerson(String name, String password,intage) {

this.name = name;

this.password = password;

this.age = age;

}

}

### 以上程序执⾏的结果如下图所⽰：

### ⾃定义去重功能实现⽅法⼆，使⽤迭代器循环并判断当前元素⾸次出现的位置（indexOf）是否等于最

### 后出现的位置（lastIndexOf），如果不等于则说明此元素为重复元素，删除当前元素即可，这样循环

### 完就能得到⼀个没有重复元素的集合，实现代码如下：

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

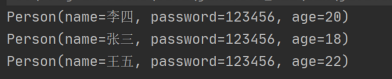
29

30

31

32

33



*图片 1 (392x79)*

# 第 64 页

代码块importlombok.Data;

importjava.util.ArrayList;

importjava.util.Iterator;

importjava.util.List;

publicclassDistinctExample{

publicstaticvoidmain(String[] args) {

//创建并给List赋值

List<Person> list =newArrayList<>();

list.add(newPerson("李四","123456",20));

list.add(newPerson("张三","123456",18));

list.add(newPerson("王五","123456",22));

list.add(newPerson("张三","123456",18));

//去重操作

Iterator<Person> iterator = list.iterator();

while(iterator.hasNext()) {

//获取循环的值

Personitem= iterator.next();

//如果存在两个相同的值

if(list.indexOf(item) != list.lastIndexOf(item)) {

//移除相同的值

iterator.remove();

}

}

//打印集合信息

list.forEach(p -> System.out.println(p));

}

}

@Data

classPerson{

privateString name;

privateString password;

privateintage;

publicPerson(String name, String password,intage) {

this.name = name;

this.password = password;

this.age = age;

}

}

## 2.利⽤Set集合去重

### Set集合天⽣具备去重特性，在创建Set集合时可以传递⼀个List集合，这样就能实现数据转移和去

### 重

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

34

35

36

37

38

# 第 65 页

### 的功能了，具体实现代码如下：

代码块

importlombok.Data;

importjava.util.ArrayList;

importjava.util.HashSet;

importjava.util.List;

publicclassDistinctExample{

publicstaticvoidmain(String[] args) {

//创建并给List赋值

List<Person> list =newArrayList<>();

list.add(newPerson("李四","123456",20));

list.add(newPerson("张三","123456",18));

list.add(newPerson("王五","123456",22));

list.add(newPerson("张三","123456",18));

//去重操作

HashSet<Person> set =newHashSet<>(list);

//打印集合信息

set.forEach(p -> System.out.println(p));

}

}

@Data

classPerson{

privateString name;

privateString password;

privateintage;

publicPerson(String name, String password,intage) {

this.name = name;

this.password = password;

this.age = age;

}

}

### 通过上述结果，我们发现了⼀个问题，在使⽤了HashSet去重之后，元素的先后顺序竟然也发⽣了变

### 化。为了能解决这个问题，我们可以使⽤LinkedHashSet来实现去重功能，具体实现代码如下：

代码块

importlombok.Data;

importjava.util.ArrayList;

importjava.util.LinkedHashSet;

importjava.util.List;

publicclassDistinctExample{

publicstaticvoidmain(String[] args) {

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

1

2

3

4

5

6

# 第 66 页

//创建并给List赋值

List<Person> list =newArrayList<>();

list.add(newPerson("李四","123456",20));

list.add(newPerson("张三","123456",18));

list.add(newPerson("王五","123456",22));

list.add(newPerson("张三","123456",18));

//去重操作

LinkedHashSet<Person> set =newLinkedHashSet<>(list);

//打印集合信息

set.forEach(p -> System.out.println(p));

}

}

@Data

classPerson{

privateString name;

privateString password;

privateintage;

publicPerson(String name, String password,intage) {

this.name = name;

this.password = password;

this.age = age;

}

}

## 3.使⽤Stream去重

### 最后⼀种也是最简单的⼀种去重⽅式，我们可以使⽤JDK8中提供的Stream进⾏去重，Stream中包

### 含了⼀个去重⽅法：distinct，可以直接实现集合的去重功能，具体实现代码如下：

### importlombok.Data;

### importjava.util.ArrayList;

### importjava.util.List;

代码块

importjava.util.stream.Collectors;

publicclassDistinctExample{

publicstaticvoidmain(String[] args) {

//创建并给List赋值

List<Person> list =newArrayList<>();

list.add(newPerson("李四","123456",20));

list.add(newPerson("张三","123456",18));

list.add(newPerson("王五","123456",22));

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

1

2

3

4

5

6

7

8

# 第 67 页

list.add(newPerson("张三","123456",18));

//去重操作

list = list.stream().distinct().collect(Collectors.toList());

//打印集合信息

list.forEach(p -> System.out.println(p));

}

}

@Data

classPerson{

privateString name;

privateString password;

privateintage;

publicPerson(String name, String password,intage) {

this.name = name;

this.password = password;

this.age = age;

}

}

## 总结

本⽂介绍了List集合去重的3种实现思路，其中⾃定义去重功能实现起来相对繁琐，⽽Set集合依靠

其

⾃带的去重特性，可以很⽅便的实现去重功能，并且可以使⽤LinkedHashSet在去重的同时⼜保证了

元素所在位置不被更改。⽽最后⼀种去重的⽅法，是JDK8中新增的，使⽤Stream中的distinct⽅

法

实现去重，它的优点是不但写法简单，⽽且⽆需创建新的集合，是实现去重功能的⾸选⽅法。

## Hashset

### HashSet如何保证元素不重复？

HashSet实现了Set接⼝，由哈希表（实际是HashMap）提供⽀持。HashSet不保证集合的迭代顺

序，但允许插⼊null值。也就是说HashSet不能保证元素插⼊顺序和迭代顺序相同。HashSet具备

去重的特性，也就是说它可以将集合中的重复元素⾃动过滤掉，保存存储在HashSet中的元素都是唯

⼀的。

### 1.HashSet基本⽤法

HashSet基本操作⽅法有：add（添加）、remove（删除）、contains（判断某个元素是否存在）和

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

# 第 68 页

### size（集合数量）。这些⽅法的性能都是固定操作时间，如果哈希函数是将元素分散在桶中的正确位

### 置。HashSet基本使⽤如下：

代码块

//创建HashSet集合

HashSet<String> strSet =newHashSet<>();

//给HashSet添加数据

strSet.add("Java");

strSet.add("MySQL");

strSet.add("Redis");

//循环打印HashSet中的所有元素

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

## 2.HashSet⽆序性

### HashSet不能保证插⼊元素的顺序和循环输出元素的顺序⼀定相同，也就是说HashSet其实是⽆序的

### 集合，具体代码⽰例如下：

代码块

HashSet<String> mapSet =newHashSet<>();

mapSet.add("深圳");

mapSet.add("北京");

mapSet.add("西安");

//循环打印HashSet中的所有元素

mapSet.forEach(m -> System.out.println(m));

### 输出结果：

### 从上述代码和执⾏结果可以看出，HashSet插⼊的顺序是：深圳->北京->西安，⽽循环打印的顺序

### 却

### 是：西安->深圳->北京，所以HashSet是⽆序的，不能保证插⼊和迭代的顺序⼀致。

### PS：如果要保证插⼊顺序和迭代顺序⼀致，可使⽤LinkedHashSet来替换HashSet。

## 3.HashSet错误⽤法

### 有⼈说HashSet只能保证基础数据类型不重复，却不能保证⾃定义对象不重复？这样说对吗？我们通

### 过以下⽰例来说明此问题。

1

2

3

4

5

6

7

8

1

2

3

4

5

6



*图片 1 (80x61)*

# 第 69 页

### 3.1HashSet与基本数据类型

### 使⽤HashSet存储基本数据类型，实现代码如下：

代码块

HashSet<Long> longSet =newHashSet<>();

longSet.add(666l);

longSet.add(777l);

longSet.add(999l);

longSet.add(666l);

//循环打印HashSet中的所有元素

longSet.forEach(l -> System.out.println(l));

### 以上程序的执⾏结果如下：

### 从上述结果可以看出，使⽤HashSet可以保证基础数据类型不重复。

### 3.2HashSet与⾃定义对象类型

### 接下来，将⾃定义对象存储到HashSet中，实现代码如下：

代码块

publicclassHashSetExample{

publicstaticvoidmain(String[] args) {

HashSet<Person> personSet =newHashSet<>();

personSet.add(newPerson("曹操","123"));

personSet.add(newPerson("孙权","123"));

personSet.add(newPerson("曹操","123"));

//循环打印HashSet中的所有元素

personSet.forEach(p -> System.out.println(p));

}

}

@Getter

@Setter

@ToString

classPerson{

privateString name;

privateString password;

publicPerson(String name, String password) {

this.name = name;

this.password = password;

}

1

2

3

4

5

6

7

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

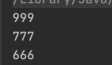
16

17

18

19

20



*图片 1 (112x65)*

# 第 70 页

}

### 以上程序的执⾏结果如下：

### 从上述结果可以看出，⾃定义对象类型确实没有被去重，那也就是说HashSet不能实现⾃定义对象类

### 型的去重咯？其实并不是，HashSet去重功能是依赖元素的hashCode和equals⽅法判断的，通过

### 这两个⽅法返回的都是true那就是相同对象，否则就是不同对象。⽽前⾯的Long类型元素之所以能

### 实现

### 去重，正是因为Long类型中已经重写了hashCode和equals⽅法，具体实现源码如下：

代码块

@Override

publicinthashCode() {

returnLong.hashCode(value);

}

publicbooleanequals(Object obj) {

if(objinstanceofLong) {

returnvalue == ((Long)obj).longValue();

}

returnfalse;

}

### 那么，想让HashSet⽀持⾃定义对象去重，只需要在⾃定义对象中重写hashCode和equals⽅法即

### 可，具体实现代码如下：

代码块

@Setter

@Getter

@ToString

classPerson{

privateString name;

privateString password;

publicPerson(String name, String password) {

this.name = name;

this.password = password;

}

21

1

2

3

4

5

6

7

8

9

10

1

2

3

4

5

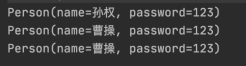
6

7

8

9

10



*图片 1 (246x66)*

# 第 71 页

@Override

publicbooleanequals(Object o) {

if(this== o)returntrue;//引⽤相等返回true

//如果等于null，或者对象类型不同返回false

if(o ==null|| getClass() != o.getClass())returnfalse;

//强转为⾃定义Person类型

Personpersion= (Person) o;

//如果name和password都相等，就返回true

returnObjects.equals(name, persion.name) &&

Objects.equals(password, persion.password);

}

@Override

publicinthashCode() {

//对⽐name和password是否相等

returnObjects.hash(name, password);

}

}

### 重新运⾏以上代码，执⾏结果如下图所⽰：

### 从上述结果可以看出，之前的重复项“曹操”已经被去重了。

## 4.HashSet如何保证元素不重复？

### 我们只要了解了HashSet执⾏添加元素的流程，就能知道为什么HashSet能保证元素不重复了？

### HashSet添加元素的执⾏流程是：当把对象加⼊HashSet时，HashSet会先计算对象的hashcode值

### 来判断对象加⼊的位置，同时也会与其他加⼊的对象的hashcode值作⽐较，如果没有相符的

### hashcode，HashSet会假设对象没有重复出现，会将对象插⼊到相应的位置中。但是如果发现有相同

### hashcode值的对象，这时会调⽤对象的equals()⽅法来检查对象是否真的相同，如果相同，则

### HashSet就不会让重复的对象加⼊到HashSet中，这样就保证了元素的不重复。

### 为了更清楚的了解HashSet的添加流程，我们可以尝试阅读HashSet的具体实现源码，HashSet添

### 加⽅法的实现源码如下（以下源码基于JDK8）：

代码块

// hashmap中put()返回null时，表⽰操作成功

publicbooleanadd(E e) {

returnmap.put(e, PRESENT)==null;

}

### 从上述源码可以看出HashSet中的add⽅法，实际调⽤的是HashMap中的put，那么我们继续看

### HashMap中的put实现：

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

1

2

3

4

# 第 72 页

代码块//返回值：如果插⼊位置没有元素则返回null，否则返回上⼀个元素

publicVput(K key, V value) {

returnputVal(hash(key), key, value,false,true);

}

从上述源码可以看出，HashMap中的put()⽅法⼜调⽤了putVal()⽅法，putVal()的源码如下：

## •

finalVputVal(inthash,Kkey,Vvalue,booleanonlyIfAbsent,

booleanevict){

Node<K,V>[]tab;

Node<K,V>p;

intn,i;

//如果哈希表为空，调⽤resize()创建⼀个哈希表，并⽤变量n记录哈希表⻓度

if((tab=table)==null||(n=tab.length)==0)

n=(tab=resize()).length;

/\*\*

\*如果指定参数hash在表中没有对应的桶，即为没有碰撞

\*Hash函数，(n-1)&hash计算key将被放置的槽位

\*(n-1)&hash本质上是hash%n位运算更快

/

if((p=tab[i=(n-1)&hash])==null)

//直接将键值对插⼊到map中即可

tab[i]=newNode(hash,key,value,null);

else{//桶中已经存在元素

Node<K,V>e;

Kk;

//⽐较桶中第⼀个元素(数组中的结点)的hash值相等，key相等

if(p.hash==hash&&

((k=p.key)==key||(key!=null&&key.equals(k))))

//将第⼀个元素赋值给e，⽤e来记录

e=p;

//当前桶中⽆该键值对，且桶是红⿊树结构，按照红⿊树结构插⼊

elseif(pinstanceofTreeNode)

e=((TreeNode<K,V>)p).putTreeVal(this,tab,hash,key,

value);

//当前桶中⽆该键值对，且桶是链表结构，按照链表结构插⼊到尾部

else{

for(intbinCount=0;;++binCount){

//遍历到链表尾部

1

2

3

4

# 第 73 页

if((e=p.next)==null){

p.next=newNode(hash,key,value,null);

//检查链表⻓度是否达到阈值，达到将该槽位节点组织形式转为红⿊树

if(binCount>=TREEIFY\_THRESHOLD-1)//-1for1st

treeifyBin(tab,hash);

break;

}

//链表节点的<key,value>与put操作<key,value>

//相同时，不做重复操作，跳出循环

if(e.hash==hash&&

((k=e.key)==key||(key!=null&&

key.equals(k))))

break;

p=e;

}

}

//找到或新建⼀个key和hashCode与插⼊元素相等的键值对，进⾏put操作

if(e!=null){//existingmappingforkey

//记录e的value

VoldValue=e.value;

\*/\*

\*

onlyIfAbsent为false或旧值为null时，允许替换旧值

否则⽆需替换

\*/

if(!onlyIfAbsent||oldValue==null)

e.value=value;

//访问后回调

afterNodeAccess(e);

//返回旧值

returnoldValue;

}

}

//更新结构化修改信息

++modCount;

//键值对数⽬超过阈值时，进⾏rehash

if(++size>threshold)

resize();

//插⼊后回调

afterNodeInsertion(evict);

# 第 74 页

returnnull;

}从上述源码可以看出，当将⼀个键值对放⼊HashMap时，⾸先根据key的hashCode()返回值决定

该Entry的存储位置。如果有两个key的hash值相同，则会判断这两个元素key的equals()是否相

同，

如果相同就返回true，说明是重复键值对，那么HashSet中add()⽅法的返回值会是false，表⽰

HashSet添加元素失败。因此，如果向HashSet中添加⼀个已经存在的元素，新添加的集合元素不会

覆盖已有元素，从⽽保证了元素的不重复。如果不是重复元素，put⽅法最终会返回null，传递到

HashSet的add⽅法就是添加成功。

## 总结

HashSet底层是由HashMap实现的，它可以实现重复元素的去重功能，如果存储的是⾃定义对象必

须

重写hashCode和equals⽅法。HashSet保证元素不重复是利⽤HashMap的put⽅法实现的，在存

储之前先根据key的hashCode和equals判断是否已存在，如果存在就不在重复插⼊了，这样就保

证

了元素的不重复。

## HashMap

### HashMap有⼏种遍历⽅法？推荐使⽤哪种？

HashMap的遍历⽅法有很多种，不同的JDK版本有不同的写法，其中JDK8就提供了3种HashMap

的遍历⽅法，并且⼀举打破了之前遍历⽅法“很臃肿”的尴尬。

### 1.JDK8之前的遍历

JDK8之前主要使⽤EntrySet和KeySet进⾏遍历，具体实现代码如下。

### 1.1EntrySet遍历

EntrySet是早期HashMap遍历的主要⽅法，其实现代码如下：

代码块

publicstaticvoidmain(String[] args) {

//创建并赋值hashmap

HashMap<String, String> map =newHashMap() {{

put("Java"," Java Value.");

put("MySQL"," MySQL Value.");

put("Redis"," Redis Value.");

}};

1

2

3

4

5

6

7

# 第 75 页

//循环遍历

for(Map.Entry<String, String> entry : map.entrySet()) {

System.out.println(entry.getKey() +":"+ entry.getValue());

}

}

### 以上程序的执⾏结果，如下图所⽰：

## 1.2KeySet遍历

### KeySet的遍历⽅式是循环Key内容，再通过map.get(key)获取Value的值，具体实现如下：

代码块

publicstaticvoidmain(String[] args) {

//创建并赋值hashmap

HashMap<String, String> map =newHashMap() {{

put("Java"," Java Value.");

put("MySQL"," MySQL Value.");

put("Redis"," Redis Value.");

}};

//循环遍历

for(String key : map.keySet()) {

System.out.println(key +":"+ map.get(key));

}

}

### 以上程序的执⾏结果，如下图所⽰：

### KeySet性能问题

### 通过以上代码，我们可以看出使⽤KeySet遍历，其性能是不如EntrySet的，因为KeySet其实循环

### 了两遍集合，第⼀遍循环是循环Key，⽽获取Value有需要使⽤map.get(key)，相当于有循环了⼀遍

### 集合，所以KeySet循环不能建议使⽤，因为循环了两次，效率⽐较低。

## 1.3EntrySet迭代器遍历

代码块

publicstaticvoidmain(String[] args) {

//创建并赋值hashmap

HashMap<String, String> map =newHashMap() {{

put("Java"," Java Value.");

8

9

10

11

12

1

2

3

4

5

6

7

8

9

10

11

12

1

2

3

4



*图片 1 (137x56)*

# 第 76 页

put("MySQL"," MySQL Value.");

put("Redis"," Redis Value.");

}};

//循环遍历

Iterator<Map.Entry<String, String>> iterator = map.entrySet().iterator();

while(iterator.hasNext()) {

Map.Entry<String, String> entry = iterator.next();

System.out.println(entry.getKey() +":"+ entry.getValue());

}

}

### EntrySet和KeySet除了以上直接循环外，我们还可以使⽤它们的迭代器进⾏循环，如EntrySet的迭

### 代

### 器实现代码如下：

### 以上程序的执⾏结果，如下图所⽰：

## 1.4KeySet迭代器遍历

### KeySet也可以使⽤迭代器的⽅式进⾏遍历，实现代码如下：

代码块

publicstaticvoidmain(String[] args) {

//创建并赋值hashmap

HashMap<String, String> map =newHashMap() {{

put("Java"," Java Value.");

put("MySQL"," MySQL Value.");

put("Redis"," Redis Value.");

}};

//循环遍历

Iterator<String> iterator = map.keySet().iterator();

while(iterator.hasNext()) {

Stringkey= iterator.next();

System.out.println(key +":"+ map.get(key));

}

}

### 以上程序的执⾏结果，如下图所⽰：

5

6

7

8

9

10

11

12

13

14

1

2

3

4

5

6

7

8

9

10

11

12

13

14



*图片 1 (147x57)*

# 第 77 页

### 虽然KeySet循环⽅式不推荐使⽤，但还是有必要了解⼀下的。

## 1.5迭代器的作⽤

### 既然能直接遍历，那为什么还要⽤迭代器呢？通过以下例⼦我们就知道了。

### 不使⽤迭代器删除

### 如果不使⽤迭代器，假如我们在遍历EntrySet时，在遍历代码中删除元素，代码的实现如下：

代码块

publicstaticvoidmain(String[] args) {

//创建并赋值hashmap

HashMap<String, String> map =newHashMap() {{

put("Java"," Java Value.");

put("MySQL"," MySQL Value.");

put("Redis"," Redis Value.");

}};

//循环遍历

for(Map.Entry<String, String> entry : map.entrySet()) {

if("Java".equals(entry.getKey())) {

//删除此项

map.remove(entry.getKey());

continue;

}

System.out.println(entry.getKey() +":"+ entry.getValue());

}

}

### 以上程序的执⾏结果，如下图所⽰：

### 可以看到，如果在遍历的代码中动态删除元素，⾮迭代器的⽅式就会报错。

### 使⽤迭代器删除

### 接下来，我们使⽤迭代器循环EntrySet，并且在循环中动态删除元素，实现代码如下：

代码块

publicstaticvoidmain(String[] args) {

//创建并赋值hashmap

HashMap<String, String> map =newHashMap() {{

put("Java"," Java Value.");

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

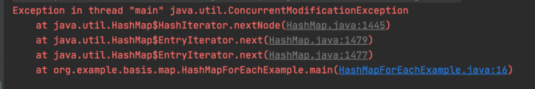
17

1

2

3

4



*图片 1 (535x89)*

# 第 78 页

put("MySQL"," MySQL Value.");

put("Redis"," Redis Value.");

}};

//循环遍历

Iterator<Map.Entry<String, String>> iterator = map.entrySet().iterator();

while(iterator.hasNext()) {

Map.Entry<String, String> entry = iterator.next();

if("Java".equals(entry.getKey())) {

//删除此项

iterator.remove();

continue;

}

System.out.println(entry.getKey() +":"+ entry.getValue());

}

}

以上程序的执⾏结果，如下图所⽰：

从上述结果可以看出，使⽤迭代器的优点是可以在循环的时候，动态的删除集合中的元素。⽽上⾯⾮

迭

代器的⽅式则不能在循环的过程中删除元素（程序会报错）。

### 2.JDK8之后的遍历

在JDK8之后HashMap的遍历就变得⽅便很多了，JDK8中包含了以下3种遍历⽅法：

## •

使⽤Lambda遍历

## •

使⽤Stream单线程遍历

## •

使⽤Stream多线程遍历

我们分别来看。

### 2.1Lambda遍历

使⽤Lambda表达式的遍历⽅法实现代码如下：

以上程序的执⾏结果，如下图所⽰：

代码块

publicstaticvoidmain(String[] args) {

//创建并赋值hashmap

HashMap<String, String> map =newHashMap() {{

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

1

2

3



*图片 1 (158x40)*

# 第 79 页

put("Java"," Java Value.");

put("MySQL"," MySQL Value.");

put("Redis"," Redis Value.");

}};

//循环遍历

map.forEach((key, value) -> {

System.out.println(key +":"+ value);

});

}

## 2.2Stream单线程遍历

### Stream遍历是先得到map集合的EntrySet，然后再执⾏forEach循环，实现代码如下：

代码块

publicstaticvoidmain(String[] args) {

//创建并赋值hashmap

HashMap<String, String> map =newHashMap() {{

put("Java"," Java Value.");

put("MySQL"," MySQL Value.");

put("Redis"," Redis Value.");

}};

//循环遍历

map.entrySet().stream().forEach((entry) -> {

System.out.println(entry.getKey() +":"+ entry.getValue());

});

}

### 2.3Stream多线程遍历

### Stream多线程的遍历⽅式和上⼀种遍历⽅式类似，只是多执⾏了⼀个parallel并发执⾏的⽅法，此⽅

### 法会根据当前的硬件配置⽣成对应的线程数，然后再进⾏遍历操作，实现代码如下：

代码块

publicstaticvoidmain(String[] args) {

//创建并赋值hashmap

HashMap<String, String> map =newHashMap() {{

put("Java"," Java Value.");

put("MySQL"," MySQL Value.");

put("Redis"," Redis Value.");

}};

//循环遍历

map.entrySet().stream().parallel().forEach((entry) -> {

4

5

6

7

8

9

10

11

12

1

2

3

4

5

6

7

8

9

10

11

12

1

2

3

4

5

6

7

8

9

# 第 80 页

System.out.println(entry.getKey() +":"+ entry.getValue());

});

}

因为程序是并发执⾏的，所以没有办法保证元素的执⾏顺序和打印顺序，这就是并发编程的特点。

### 推荐使⽤哪种遍历⽅式？

不同的场景推荐使⽤的遍历⽅式是不同的，例如，如果是JDK8之后的开发环境，推荐使⽤Stream

的遍历⽅式，因为它⾜够简洁；⽽如果在遍历的过程中需要动态的删除元素，那么推荐使⽤迭代器的

遍历⽅式；如果在遍历的时候，⽐较在意程序的执⾏效率，那么推荐使⽤Stream多线程遍历的⽅式，

因为

它⾜够快。所以这个问题的答案是不固定的，我们需要知道每种遍历⽅法的优缺点，再根据不同的场

景灵活变通。

## 总结

本⽂介绍了7种HashMap的遍历⽅式，其中JDK8之前主要使⽤EntrySet和KeySet的遍历⽅式，

⽽

KeySet的遍历⽅式性能⽐较低，⼀般不推荐使⽤。然⽽在JDK8之后遍历⽅式就有了新的选择，可以

使⽤⽐较简洁的Lambda遍历，也可以使⽤性能⽐较⾼的Stream多线程遍历。

### 说⼀下HashMap底层实现？以及插⼊流程？

HashMap是使⽤频率最⾼的数据类型之⼀，同时也是⾯试必问的问题之⼀，尤其是它的底层实现原

理，既是常⻅的⾯试题⼜是理解HashMap的基⽯，所以重要程度不⾔⽽喻。

### HashMap底层实现

HashMap在JDK1.7和JDK1.8的底层实现是不⼀样的，在JDK1.7中，HashMap使⽤的是数组+

链表实现的，⽽JDK1.8中使⽤的是数组+链表或红⿊树实现的。HashMap在JDK1.7中的实现如

下图

10

11

12

# 第 81 页

所⽰：

HashMap在JDK1.8中的实现如下图所⽰：

我们本⽂重点来学习主流版本JDK1.8中的HashMap。HashMap中每个元素称之为⼀个哈希桶

（bucket），哈希桶包含的内容有4个：

## •

hash值

## •

key

## •

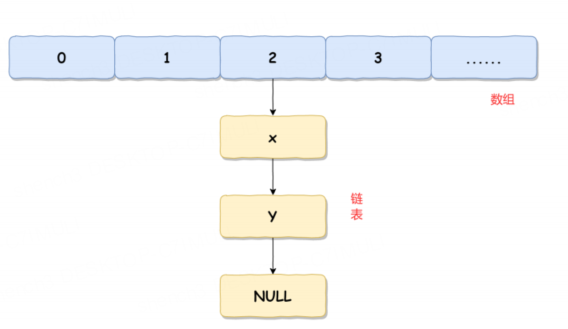
value

## •

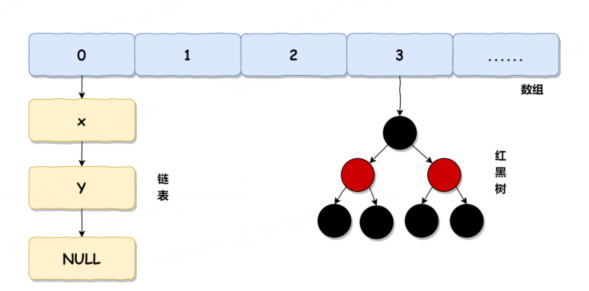
next（下⼀个节点）

### HashMap插⼊流程

HashMap元素新增的实现源码如下（下⽂源码都是基于主流版本JDK1.8）：



*图片 1 (568x329)*



*图片 2 (589x291)*

# 第 82 页

代码块publicVput(K key, V value) {

//对key进⾏哈希操作

returnputVal(hash(key), key, value,false,true);

}

finalVputVal(inthash, K key, V value,booleanonlyIfAbsent,

booleanevict) {

Node<K,V>[] tab; Node<K,V> p;intn, i;

//哈希表为空则创建表

if((tab = table) ==null|| (n = tab.length) ==0)

n = (tab = resize()).length;

//根据key的哈希值计算出要插⼊的数组索引i

if((p = tab[i = (n -1) & hash]) ==null)

//如果table[i]等于null，则直接插⼊

tab[i] = newNode(hash, key, value,null);

else{

Node<K,V> e; K k;

//如果key已经存在了，直接覆盖value

if(p.hash == hash &&

((k = p.key) == key || (key !=null&& key.equals(k))))

e = p;

//如果key不存在，判断是否为红⿊树

elseif(pinstanceofTreeNode)

//红⿊树直接插⼊键值对

e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

else{

//为链表结构，循环准备插⼊

for(intbinCount=0; ; ++binCount) {

//下⼀个元素为空时

79if((e = p.next) ==null) {

p.next = newNode(hash, key, value,null);

//转换为红⿊树进⾏处理

if(binCount >= TREEIFY\_THRESHOLD -1)// -1 for 1st

treeifyBin(tab, hash);

break;

}

// key已经存在直接覆盖value

if(e.hash == hash &&

((k = e.key) == key || (key !=null&& key.equals(k))))

break;

p = e;

}

}

if(e !=null) {// existing mapping for key

VoldValue= e.value;

if(!onlyIfAbsent || oldValue ==null)

e.value = value;

afterNodeAccess(e);

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

34

35

36

37

38

39

40

41

42

43

44

45

46

47

# 第 83 页

returnoldValue;

}

}

++modCount;

//超过最⼤容量，扩容

if(++size > threshold)

resize();

afterNodeInsertion(evict);

returnnull;

}

上述的源码都添加了相应的代码注释，简单来说HashMap的元素添加流程是，先将key值进⾏hash

得到哈希值，根据哈希值得到元素位置，判断元素位置是否为空，如果为空直接插⼊，不为空判断是

否为红⿊树，如果是红⿊树则直接插⼊，否则判断链表是否⼤于8，且数组⻓度⼤于64，如果满⾜这

两个

条件则把链表转成红⿊树，然后插⼊元素，如果不满⾜这两个条件中的任意⼀个，则遍历链表进⾏插

⼊，它的执⾏流程如下图所⽰：

### 为什么要将链表转红⿊树？

JDK1.8中引⼊了新的数据结构红⿊树来实现HashMap，主要是出于性能的考量。因为链表超过⼀定

⻓度之后查询效率就会很低，它的时间复杂度是O(n)，⽽红⿊树的时间复杂度是O(logn)，因此引⼊

红⿊树可以加快HashMap在数据量⽐较⼤情况下的查询效率。

### 哈希算法实现

HashMap的哈希算法实现源码如下：

代码块

staticfinalinthash(Object key) {

inth;

return(key ==null) ?0: (h = key.hashCode()) ^ (h >>>16);

}

其中，key.hashCode()是Java中⾃带的hashCode()⽅法，返回⼀个int类型的散列值，后⾯

hashCode再右移16位，正好是32bit的⼀半，与⾃⼰本⾝做异或操作（相同为0，不同为1），主要

是为了混合哈希值的⾼位和地位，增加低位的随机性，这样就实现了HashMap的哈希算法。

## 总结

HashMap在JDK1.7时，使⽤的是数组+链表实现的，⽽在JDK1.8时，使⽤的是数组+链表或红⿊

48

49

50

51

52

53

54

55

56

57

1

2

3

4

# 第 84 页

树的⽅式来实现的，JDK1.8之所以引⼊红⿊树主要是出于性能⽅⾯的考虑。HashMap在插⼊时，会

判断当前链表的⻓度是否⼤于8且数组的⻓度⼤于64，如果满⾜这两个条件就会把链表转成红⿊树再

进⾏插⼊，否则就是遍历链表插⼊。

### 为什么HashMap会产⽣死循环？

HashMap死循环是⼀个⽐较常⻅、⽐较经典的问题，在⽇常的⾯试中出现的频率⽐较⾼，所以接下来

咱们通过图解的⽅式，带⼤家彻底理解死循环的原因。

## 前置知识

死循环问题发⽣在JDK1.7版本中，造成这个问题主要是由于HashMap⾃⾝的运⾏机制，加上并发

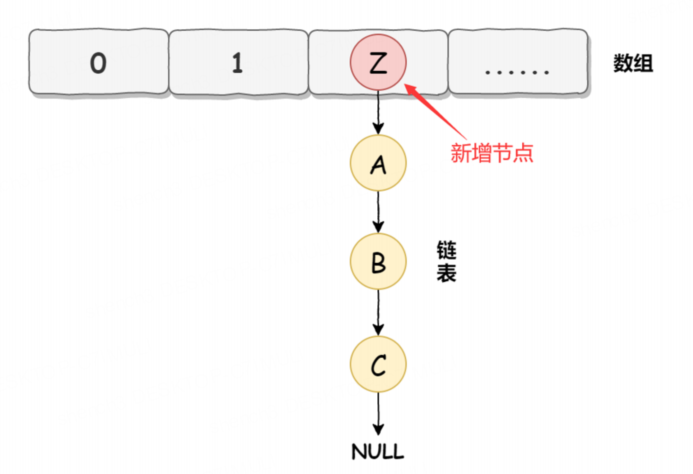
操

作，从⽽导致了死循环。在JDK1.7中HashMap的底层数据实现是数组+链表的⽅式，如下图所

⽰：

⽽HashMap在数据添加时使⽤的是头插⼊，如下图所⽰：

HashMap正常情况下的扩容实现如下图所⽰：



*图片 1 (693x474)*

# 第 85 页

旧HashMap的节点会依次转移到新HashMap中，旧HashMap转移的顺序是A、B、C，⽽新

HashMap使⽤的是头插法，所以最终在新HashMap中的顺序是C、B、A，也就是上图展⽰的那样。

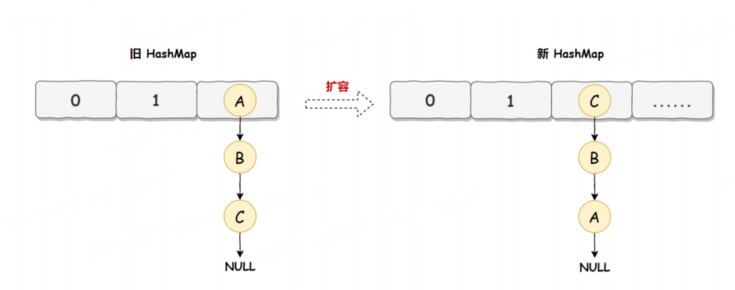
有了这些前置知识之后，咱们来看死循环是如何诞⽣的？

## 死循环执⾏步骤1

死循环是因为并发HashMap扩容导致的，并发扩容的第⼀步，线程T1和线程T2要对HashMap进⾏

扩容操作，此时T1和T2指向的是链表的头结点元素A，⽽T1和T2的下⼀个节点，也就是T1.next

和T2.next指向的是B节点，如下图所⽰：



*图片 1 (735x290)*

# 第 86 页

## 死循环执⾏步骤2

死循环的第⼆步操作是，线程T2时间⽚⽤完进⼊休眠状态，⽽线程T1开始执⾏扩容操作，⼀直到线

程T1扩容完成后，线程T2才被唤醒，扩容之后的场景如下图所⽰：

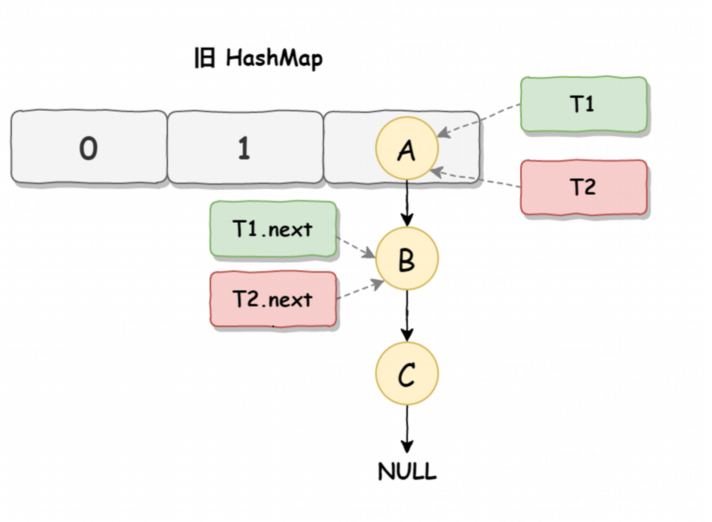
从上图可知线程T1执⾏之后，因为是头插法，所以HashMap的顺序已经发⽣了改变，但线程T2对

于发⽣的⼀切是不可知的，所以它的指向元素依然没变，如上图展⽰的那样，T2指向的是A元素，

T2.next指向的节点是B元素。

## 死循环执⾏步骤3

当线程T1执⾏完，⽽线程T2恢复执⾏时，死循环就建⽴了，如下图所⽰：



*图片 1 (712x522)*

# 第 87 页

因为T1执⾏完扩容之后B节点的下⼀个节点是A，⽽T2线程指向的⾸节点是A，第⼆个节点是B，这

个顺序刚好和T1扩完容完之后的节点顺序是相反的。T1执⾏完之后的顺序是B到A，⽽T2的顺序

是

A到B，这样A节点和B节点就形成死循环了，这就是HashMap死循环导致的原因。

## 解决⽅案

HashMap死循环的常⽤解决⽅案有以下3个：

使⽤线程安全容器ConcurrentHashMap替代（推荐使⽤此⽅案）。

使⽤线程安全容器Hashtable替代（性能低，不建议使⽤）。

使⽤synchronized或Lock加锁HashMap之后，再进⾏操作，相当于多线程排队执⾏（⽐较⿇烦，

也不建议使⽤）。

## 总结

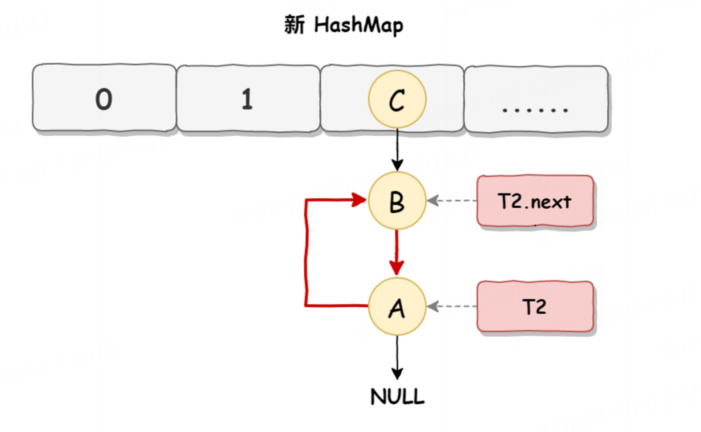
HashMap死循环发⽣在JDK1.7版本中，形成死循环的原因是HashMap在JDK1.7使⽤的是头插

法，

头插法+链表+多线程并发+HashMap扩容，这⼏个点加在⼀起就形成了HashMap的死循环，解决

死锁可以采⽤线程安全容器ConcurrentHashMap替代。

### HashMap数据覆盖问题



*图片 1 (701x432)*

# 第 88 页

数据覆盖问题发⽣在并发添加元素的场景下，它不⽌出现在JDK1.7版本中，其他版本中也存在此问

题，数据覆盖产⽣的流程如下：

38.线程T1进⾏添加时，判断某个位置可以插⼊元素，但还没有真正的进⾏插⼊操作，⾃⼰时间⽚就

⽤完了。

39.线程T2也执⾏添加操作，并且T2产⽣的哈希值和T1相同，也就是T2即将要存储的位置和T1相

同，因为此位置尚未插⼊值（T1线程执⾏了⼀半），于是T2就把⾃⼰的值存⼊到当前位置了。

40.T1恢复执⾏之后，因为⾮空判断已经执⾏完了，它感知不到此位置已经有值了，于是就把⾃⼰的

值也插⼊到了此位置，那么T2的值就被覆盖了。

具体执⾏流程如下图所⽰。

2.1数据覆盖执⾏流程⼀

线程T1准备将数据k1:v1插⼊到Null处，但还没有真正的执⾏，⾃⼰的时间⽚就⽤完了，进⼊休眠状

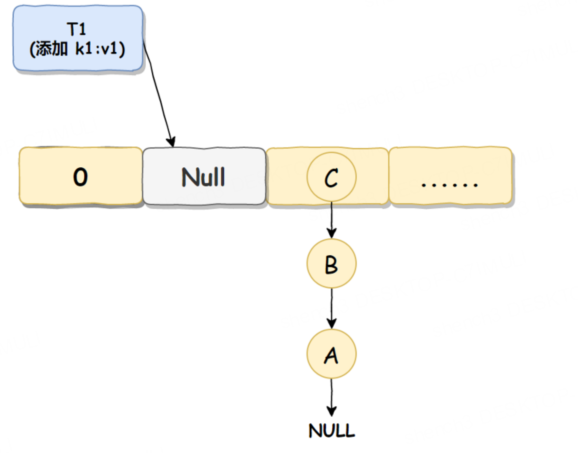
态了，如下图所⽰：

2.2数据覆盖执⾏流程⼆

线程T2准备将数据k2:v2插⼊到Null处，因为此处现在并未有值，如果此处有值的话，它会使⽤链式

法将数据插⼊到下⼀个没值的位置上，但判断之后发现此处并未有值，那么就直接进⾏数据插⼊了，

如下图所⽰：



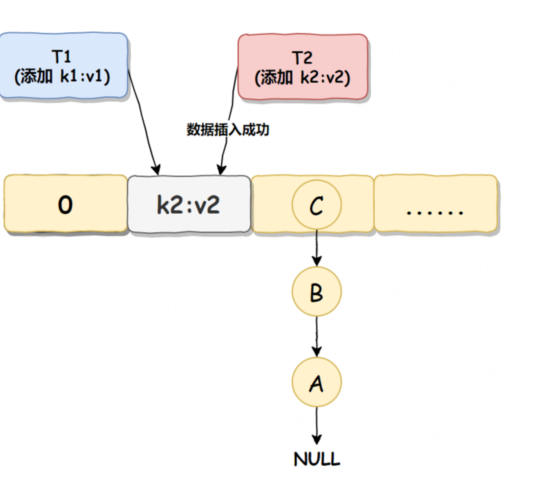
*图片 1 (578x453)*

# 第 89 页

2.3数据覆盖执⾏流程三

1线程T2执⾏完成之后，线程T1恢复执⾏，因为线程T1之前已经判断过此位置没值了，所以会直接

插⼊，此时线程T2插⼊的值就被覆盖了，如下图所⽰：



*图片 1 (559x487)*

# 第 90 页

### 2.4解决⽅案

### 解决⽅案和第⼀个解决⽅案相同，使⽤ConcurrentHashMap来替代HashMap就可以解决此问题

### 了。

## 什么是hashMap⽆序性问题？

### 这⾥的⽆序性问题指的是HashMap添加和查询的顺序不⼀致，导致程序执⾏的结果和程序员预期的

### 结果不相符，如以下代码所⽰：

代码块

HashMap<String, String> map =newHashMap<>();

//添加元素

for(inti=1; i <=5; i++) {

map.put("2022-10-"+ i,"Hello，Java："+ i);

}

//查询元素

map.forEach((k, v) -> {

System.out.println(k +"："+ v);

});

### 我们添加的顺序：

1

2

3

4

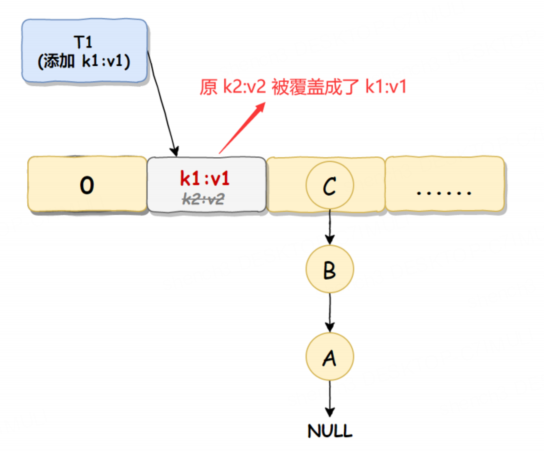
5

6

7

8

9



*图片 1 (544x451)*

# 第 91 页

### 我们期望查询的顺序和添加的顺序是⼀致的，然⽽以上代码输出的结果却是：

### 执⾏结果和我们预期结果不相符，这就是HashMap的⽆序性问题。我们期望输出的结果是Hello，

### Java1、2、3、4、5，⽽得到的顺序却是2、1、4、3、5。

### 解决⽅案

### 想要解决HashMap⽆序问题，我们只需要将HashMap替换成LinkedHashMap就可以了，如下代码

### 所⽰：

代码块

LinkedHashMap<String, String> map =newLinkedHashMap<>();

//添加元素

for(inti=1; i <=5; i++) {

map.put("2022-10-"+ i,"Hello，Java："+ i);

}

//查询元素

map.forEach((k, v) -> {

System.out.println(k +"："+ v);

});

1

2

3

4

5

6

7

8

9