**Java**

目录

[1.0 final 1](#_Toc17797743)

[1.1 java虚拟机 3](#_Toc17797744)

[1.2 clone 克隆 4](#_Toc17797745)

[1.3 类加载器 7](#_Toc17797746)

[1.3.1 类加载器的种类 7](#_Toc17797747)

[1.3.2双亲委派模型（jdk1.2推出） 7](#_Toc17797748)

[1.3.2 类加载的过程 8](#_Toc17797749)

[1、 通过子类引用父类中的静态字段，这时对子类的引用为被动引用，因此不会初始化子类，只会初始化父类 9](#_Toc17797750)

[2、 使用new实例化对象时，会先把父类初始化，然后再初始化此类本身 10](#_Toc17797751)

[3、 通过数组定义来引用类，不会触发此类的初始化 10](#_Toc17797752)

[4、 常量在编译阶段会存入调用它的类的常量池中，本质上没有直接引用到定义该常量的类，因此不会触发定义常量的类的初始化 10](#_Toc17797753)

[1.4垃圾回收 10](#_Toc17797754)

[1.4.1 jvm的内存模型 10](#_Toc17797755)

[1.4.2如何确定对象为垃圾 12](#_Toc17797756)

[1.4.3常用的垃圾回收算法 13](#_Toc17797757)

[1.4.4常用的垃圾回收器 1. Serial/Serial Old 14](#_Toc17797758)

[2. ParNew 14](#_Toc17797759)

[3. Parallel Scavenge 14](#_Toc17797760)

[4. Parallel Old 15](#_Toc17797761)

[5. CMS 15](#_Toc17797762)

[6. G1 15](#_Toc17797763)

[垃圾回收器的分类 15](#_Toc17797764)

[1.4.5 GC 17](#_Toc17797765)

[1.5 代理 18](#_Toc17797766)

[1.5.1 代理的概念 18](#_Toc17797767)

[1.5.2 代理的使用场景 18](#_Toc17797768)

[1.5.3 动态代理与静态代理 18](#_Toc17797769)

[1、静态代理 18](#_Toc17797770)

[2、动态代理 18](#_Toc17797771)

[1.6 finally 20](#_Toc17797772)

[1.6.1、finally的使用逻辑 20](#_Toc17797773)

[1.6.2、finally的原理及实例 21](#_Toc17797774)

# 1.0 final

Final关键字用于修饰变量、类、方法。

被final修饰的类不能被继承

final修饰的方法的意义：

1. 把方法锁定，以防止任何继承类修改它的含义；
2. 提高效率，final方法是基于静态绑定的，在编译时就确定好是哪个类的方法，所以final比非final方法快

Final修饰的变量

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

String a="xiaoming2";

**final** String b="xiaoming";

String d="xiaoming";

String c=b+2;

String e=d+2;

System.***out***.println(a==c);

System.***out***.println(a==e);

}

结果为True，False

变量a指的是字符串常量池中的 xiaomeng2；

变量 b 是 final 修饰的，变量 b 的值在编译时候就已经确定了它的确定值，换句话说就是提前知道了变量 b 的内容到底是个啥，相当于一个编译期常量；

变量 c 是 b + 2得到的，由于 b 是一个常量，所以在使用 b 的时候直接相当于使用 b 的原始值（xiaomeng）来进行计算，所以 c 生成的也是一个常量，a 是常量，c 也是常量，都是 xiaomeng2 而 Java 中常量池中只生成唯一的一个 xiaomeng2 字符串，所以 a 和 c 是相等的！

d 是指向常量池中 xiaomeng，但由于 d 不是 final 修饰，也就是说在使用 d 的时候不会提前知道 d 的值是什么，所以在计算 e 的时候就不一样了，e的话由于使用的是 d 的引用计算，变量d的访问却需要在运行时通过链接来进行，所以这种计算会在堆上生成 xiaomeng2 ,所以最终 e 指向的是堆上的 xiaomeng2 ， 所以 a 和 e 不相等。

Final修饰常量跟引用常量的区别

Final修饰的常量为普通变量则不可改变，为引用变量引用不可改变，引用对象的内容可以改变

# java虚拟机

Java虚拟机是一个可以执行字节码文件的虚拟进程，JVM虚拟机知道硬件平台的指令长度与其他特性使java可以运行在任意平台

JVM 之所以被称为虚拟的是因为它提供了一个不依赖于底层操作系统和机器硬件的接口。

Java 开发工具包 (JDK)

Java 开发工具包是 Java 环境的核心组件，并提供编译、调试和运行一个 Java 程序所需的所有工具，可执行文件和二进制文件。JDK 是一个平台特定的软件，有针对 Windows，Mac 和 Unix 系统的不同的安装包。可以说 JDK 是 JRE 的超集，它包含了 JRE 的 Java 编译器，调试器和核心类。目前 JDK 的版本号是 1.7，也被称为 Java 7。

Java 虚拟机(JVM)

JVM 是 Java 编程语言的核心。当我们运行一个程序时，JVM 负责将字节码转换为特定机器代码。JVM 也是平台特定的，并提供核心的 Java 方法，例如内存管理、垃圾回收和安全机制等。JVM 是可定制化的，我们可以通过 Java 选项(java options)定制它，比如配置 JVM 内存的上下界。JVM 之所以被称为虚拟的是因为它提供了一个不依赖于底层操作系统和机器硬件的接口。这种独立于硬件和操作系统的特性正是 Java 程序可以一次编写多处执行的原因。

Java 运行时环境(JRE)

JRE 是 JVM 的实施实现，它提供了运行 Java 程序的平台。JRE 包含了 JVM、Java 二进制文件和其它成功执行程序的类文件。JRE 不包含任何像 Java 编译器、调试器之类的开发工具。如果你只是想要执行 Java 程序，你只需安装 JRE 即可，没有安装 JDK 的必要。

JDK, JRE 和 JVM 的区别

JDK 是用于开发的而 JRE 是用于运行 Java 程序的。

JDK 和 JRE 都包含了 JVM，从而使得我们可以运行 Java 程序。

JVM 是 Java 编程语言的核心并且具有平台独立性。

# clone 克隆

克隆分为浅克隆(ShallowClone)和深克隆(DeepClone)。区别很简单深克隆能够操作引用类型(类,接口,数组)，而浅克隆只能操作八大基本数据类型。

1. 浅克隆

在浅克隆中，如果原型对象的成员变量是值类型，将复制一份给克隆对象；如果原型对象的成员变量是引用类型，则将引用对象的地址复制一份给克隆对象，也就是说原型对象和克隆对象的成员变量指向相同的内存地址。

简单来说，在浅克隆中，当对象被复制时只复制它本身和其中包含的值类型的成员变量，而引用类型的成员对象并没有复制。

1）被复制的类需要实现Clonenable接口（不实现的话在调用clone方法会抛出CloneNotSupportedException异常)， 该接口为标记接口(不含任何方法)

2）覆盖clone()方法，访问修饰符设为public。方法中调用super.clone()方法得到需要的复制对象。（native为本地方法)

class Student implements Cloneable{

private int number;

public int getNumber() {

return number;

}

public void setNumber(int number) {

this.number = number;

}

@Override

public Object clone() {

Student stu = null;

try{

stu = (Student)super.clone();

}catch(CloneNotSupportedException e) {

e.printStackTrace();

}

return stu;

}

}

1. 深度克隆

在深克隆中，无论原型对象的成员变量是值类型还是引用类型，都将复制一份给克隆对象，深克隆将原型对象的所有引用对象也复制一份给克隆对象。

简单来说，在深克隆中，除了对象本身被复制外，对象所包含的所有成员变量也将复制。

深度克隆有两种实现方式

实现cloneable接口，对象与对象的引用变量均实现

class Address implements Cloneable {    
   private String add;    
   public String getAdd() {    
       return add;    
   }    
   public void setAdd(String add) {    
       this.add = add;    
   }     
   @Override    
   public Object clone() {    
       Address addr = null;    
       try{    
           addr = (Address)super.clone();    
       }catch(CloneNotSupportedException e) {    
           e.printStackTrace();    
       }    
       return addr;    
   }    
}    
class Student implements Cloneable{    
   private int number;    
   private Address addr;    
   public Address getAddr() {    
       return addr;    
   }    
   public void setAddr(Address addr) {    
       this.addr = addr;    
   }    
   public int getNumber() {    
       return number;    
   }    
   public void setNumber(int number) {    
       this.number = number;    
   }    
   @Override    
   public Object clone() {    
       Student stu = null;    
       try{    
           stu = (Student)super.clone();   //浅复制    
       }catch(CloneNotSupportedException e) {    
           e.printStackTrace();    
       }    
       stu.addr = (Address)addr.clone();   //深度复制    
       return stu;    
   }    
}

序列化实现深度克隆

public class Outer implements Serializable{

2 private static final long serialVersionUID = 369285298572941L; //最好是显式声明ID

3 public Inner inner;

4 　//Discription:[深度复制方法,需要对象及对象所有的对象属性都实现序列化]

5 public Outer myclone() {

6 Outer outer = null;

7 try { // 将该对象序列化成流,因为写在流里的是对象的一个拷贝，而原对象仍然存在于JVM里面。所以利用这个特性可以实现对象的深拷贝

8 ByteArrayOutputStream baos = new ByteArrayOutputStream();

9 ObjectOutputStream oos = new ObjectOutputStream(baos);

10 oos.writeObject(this);

11 　　　　　　// 将流序列化成对象

12 ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());

13 ObjectInputStream ois = new ObjectInputStream(bais);

14 outer = (Outer) ois.readObject();

15 } catch (IOException e) {

16 e.printStackTrace();

17 } catch (ClassNotFoundException e) {

18 e.printStackTrace();

19 }

20 return outer;

# 类加载器

判断两个类是否相等，前提是两个类的类加载器相同

## 类加载器的种类

1. 启动类加载器BootStrap：主要加载JVM自身需要的类，C++语言实现，是虚拟机自身的一部分加载JAVA-HOME/lib下的核心类库、-XbootClassPath参数指定的路径下jar包加载到内存中 输出System.getProperty(“sun.boot.Class.Path”),显示所有加载器加载的文件
2. 扩展类加载器Extension：Sun.Misc.Laucher $ExtClassLoader类，Java语言实现，是Launcher静态内部类，负责加载<JAVA\_HOME>\lib\ext目录下扩展包或-DJava.ext.dir指定路径类库

输出System.getProperty(“Java.ext.dirs”),显示所有加载器加载的文件

1. 应用程序类加载器（系统类加载器）Application/System：加载用户路径(classpath)上指定的类库，通过ClassLoader.getSystemClassLoader()获得

输出System.getProperty(“java.class.path”),显示所有加载器加载的文件

## 1.3.2双亲委派模型（jdk1.2推出）

加载器的继承关系：BootStrap（顶级）——》Extension——》Application——》自定义加载器

**双亲委派模型的概念：**一个类加载器收到类加载的请求，他不会自己先去加载，而是把请求委托给父类加载器去加载，如果父类加载器还存在更高级别的父类加载器，则继续向上依次委托，最终达到顶级启动类加载器，如果启动类加载器不能加载则向下依次传递，如果父类加载器均无法完成加载任务，则由子加载器自己完成加载

**优点：**Java类伴随其类加载器具备了带有优先级的层次关系，确保了在各种加载环境的加载顺序。防止类的重复加载，保证了运行的安全性，防止不可信类扮演可信任的类。

## 类加载的过程

加载——连接（验证、准备、解析）——初始化

类名.class方式 JVM将使用类装载器, 将类装入内存(前提是:类还没有装入内存),不做类的初始化工作.返回Class的对象

Class.forName(“类名字符串”) （注：类名字符串是包名+类名） 说明：装入类,并做类的静态初始化，返回Class的对象

实例对象.getClass() 说明：对类进行静态初始化、非静态初始化；返回引用运行时真正所指的对象(因为:子对象的引用可能会赋给父对象的引用变量中)所属的类的Class的对象

**加载：**将class文件读入内存，并创建一个Java.lang.class对象

1、通过类的全限定名类获取定义此类的二进制字节流

2、将这个字节流所代表的静态存储结构转换成方法区运行时数据结构

3、在内存中生成一个代表这个类的java.lang.Class对象，作为数据访问的入口

**连接：**当类加载后，系统会生成一个对应的class对象，接着进入到连接阶段，，他负责把类的二进制数据合并到JRE中

**A验证：**用于检测被加载的类是否有正确的内部结构，并与其他的类协调一致

1. 文件验证

魔术因子（类型）、主次版本号、MD5指纹、常量池中的常量类型

1. 元数据验证（语义的，JVM规范）
2. 该类是否存在父类或接口
3. 检查是否继承了被final修饰的类
4. 检查是否为抽象类
5. 检查方法重载的合法性
6. 字节码验证
7. 指令跳转的合法性（循环、if）；
8. 类型转换的合法性（装箱）；
9. 虚拟机栈操作的合法性
10. 符号的验证
11. 符号引用方法类
12. 通过符号引用能否找到正确的类
13. 验证类、方法、字段的可见性

**B准备：**

给静态变量分配空间并赋值默认值

**C解析**：从常量池中寻找类、接口、字段和方法的符号引用，并将符号引用替换为直接引用

符号引用：在编译过程中用符号来代表类中信息，与虚拟机的内存布局无关，未加入到内存

直接引用：在装载阶段给常量分配实际的内存空间，与虚拟机内存布局相关，加载到内存，class对象、类变量、类方法使用直接指向目标的指针，实例方法实例变量使用的相对偏移量，也可以是一个间接指向目标的句柄

1. 类、接口的解析
2. 如果该类型不是数组类型，在加载过程中需要先完成对该类的加载
3. 如果该类型是数组，虚拟机不需要完成对该类的加载，只需要在虚拟机生成代表该类型的数组对象，并且在堆内存中开辟一片连续的地址空间（延迟加载）
4. 类方法的解析：类方法可以直接使用该类进行调用
5. 如果方法表中发现该类是一个接口，报错
6. 在该类中查找是否存在与方法描述一致的的方法，如果有直接返回这个方法的引用，如果没有按照继承关系向上查找，如果没有、报错
7. 如果找到方法的所属类是一个抽象类，报错
8. 接口方法的解析

接口方法必须有相应的实现类才能继续调用，规范

**初始化：**

被static修饰又被final修饰的，已在编译期把结果放入类的常量池

在类new对象过程中，实例变量实例方法被初始化

**主动使用：**

1、遇到new、getstatic、putstatic、invokestatic这四条字节码指令时，如果类还没有进行过初始化，则需要先触发其初始化。生成这四条指令最常见的Java代码场景是：使用new关键字实例化对象时、读取或设置一个类的静态字段（static）时（被static修饰又被final修饰的，已在编译期把结果放入常量池的静态字段除外）、以及调用一个类的静态方法时。

第二种：使用Java.lang.refect包的方法对类进行反射调用时，如果类还没有进行过初始化，则需要先触发其初始化。

第三种：当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。

第四种：当虚拟机启动时，用户需要指定一个要执行的主类，虚拟机会先执行该主类（main方法）。

第五种：当使用JDK1.5支持时，如果一个java.langl.incoke.MethodHandle实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

**被动使用：**

### 通过子类引用父类中的静态字段，这时对子类的引用为被动引用，因此不会初始化子类，只会初始化父类

### 使用new实例化对象时，会先把父类初始化，然后再初始化此类本身

### 通过数组定义来引用类，不会触发此类的初始化

### 常量在编译阶段会存入调用它的类的常量池中，本质上没有直接引用到定义该常量的类，因此不会触发定义常量的类的初始化

Clinit方法：存在于class文件中记录静态变量赋值的操作，如果类中没有静态变量或静态代码块，则没有clinit方法

1. 虚拟机会保证在子类的clinit方法执行前父类的clinit方法先执行，所以虚拟机第一个执行object的clinit方法
2. Clinit方法是由编译器自动收集类中所有类变量的赋值动作和静态代码块中的语句合并产生的，编译器收集的的顺序是根据其在源文件中的顺序决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在他之后的变量，在前面的静态语句块可以赋值但不能访问
3. 某个类或接口中如果不包含静态变量和静态语句块，不会生成clinit方法
4. 虚拟机会保证在多线程环境下该类的clinit方法会正确的synchronized加锁、同步，也就是只有一个类执行该类的clinit方法，其他线程会被阻塞，这也是单例设计模式中后来的设计方式的原理

# 1.4垃圾回收

## 1.4.1 jvm的内存模型



1. **栈区（线程私有）**
2. **程序计数器**

指向当前线程正在执行的字节码指令地址/行号，线程恢复是依赖于程序计数器，从1开始

**此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域**

1. **虚拟机栈**

每一个线程都有一个虚拟机栈，每一个虚拟机栈又包含多个栈帧，栈帧又包含局部变量表、操作数栈、动态链接、返回地址

1. 局部变量表

用于存放方法的参数跟方法内的局部变量，即8大基本类型，对象引用，returnAddress

1. 操作数栈

操作数栈并非采用访问索引的方式来进行数据访问的，而是通过标准的入栈和出栈操作来完成一次数据访问，操作数栈就是JVM执行引擎的一个工作区，当一个方法被调用的时候，一个新的栈帧也会随之被创建出来，但这个时候栈帧中的操作数栈却是空的，只有方法在执行的过程中，才会有各种各样的字节码指令往操作数栈中执行入栈和出栈操作

1. 动态链接

在运行期间符号引用才转换为直接引用，机运行时才确定具体操作

1. 返回地址

调用方法前所在位置

**C、本地方法栈**

**2、方法区**

a、方法区是被所有线程共享区域，用于存放已被虚拟机加载的类信息，常量，静态变量等数据。被Java虚拟机描述为堆的一个逻辑部分。习惯是也叫它永久代（permanment generation）  
    永久代也会垃圾回收，主要针对常量池回收，类型卸载（比如反射生成大量的临时使用的Class等信息）。  
    常量池用于存放编译期生成的各种字节码和符号引用，常量池具有一定的动态性，里面可以存放编译期生成的常量；运行期间的常量也可以添加进入常量池中，比如string（jdk1.7开始string对象移入堆区）的intern()方法。  
    当方法区满时，无法在分配空间，就会抛出内存溢出的异常（OutOfMemoneyError）。  
java8中已经没有方法区了，取而代之的是元空间（Metaspace）

B、元空间

移除永久代的工作从JDK1.7就开始了。JDK1.7中，存储在永久代的部分数据就已经转移到了Java Heap或者是 Native Heap。但永久代仍存在于JDK1.7中，并没完全移除，譬如符号引用(Symbols)转移到了native heap；字面量(interned strings)转移到了java heap；类的静态变量(class statics)转移到了java heap

元空间是方法区的在HotSpot jvm 中的实现，方法区主要用于存储类的信息、常量池、方法数据、方法代码等。方法区逻辑上属于堆的一部分，但是为了与堆进行区分，通常又叫“非堆”。

元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。，理论上取决于32位/64位系统可虚拟的内存大小。可见也不是无限制的，需要配置参数，初始化的Metaspace大小，控制元空间发生GC的阈值。GC后，动态增加或降低MetaspaceSize。在默认情况下，这个值大小根据不同的平台在12M到20M浮动。使用[Java](http://lib.csdn.net/base/javase) -XX:+PrintFlagsInitial命令查看本机的初始化参数

3、**堆区**

1. 年轻代
2. Eden，用于存放新生成的对象
3. Survivor 两个大小相等的空间，用于实现复制回收算法，Eden中进行minorGC后对象进入from

8:1:1

1. 老年代

默认对象经过15次新生代的垃圾回收处理后还存在的对象进入老年代

堆内存的分配，jvm初始分配的内存由-Xms指定，默认是物理内存的1/64,jvm最大分配的内存由-Xmx指定，默认是物理内存的1/4，默认空余堆内存小于40%时，jvm就会增大堆直到-Xmx的最大限制，可以由-XX：Min HeapFreeRatio指定，默认空余堆内存大于70%，jvm会减少堆直到-Xms最小限制，可以由-XX：Man HeapFreeRatio指定

非堆的内存分配，jvm使用-XX：PermSize设置非堆内存初始值，默认是初始内存的1/64,由XX：MaxPermSize设置的最大非堆内存大小为物理内存的1/4

## 1.4.2如何确定对象为垃圾

1. **可达性算法GCRoots（根搜索算法）**

该算法的基本思路就是通过一些被称为引用链（GC Roots）的对象作为起点，从这些节点开始向下搜索，搜索走过的路径被称为（Reference Chain)，当一个对象到GC Roots没有任何引用链相连时（即从GC Roots节点到该节点不可达），则证明该对象是不可用的。

**作用的对象**：

1. 虚拟机栈（栈帧的本地变量表）中引用的对象
2. 方法区：类静态属性引用的对象
3. 方法区：常量引用的对象
4. 本地方法栈中JNI本地方法的引用对象

**如何判断无用的类：**

(1).该类的所有实例都已经被回收，即java堆中不存在该类的实例对象。

(2).加载该类的类加载器已经被回收。

(3).该类所对应的java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射机制访问该类的方法。

**2、引用计数法**

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器都为0的对象就是不再被使用的，垃圾收集器将回收该对象使用的内存。

引用计数算法实现简单，效率很高，微软的COM技术、ActionScript、Python等都使用了引用计数算法进行内存管理，但是引用计数算法对于对象之间相互循环引用问题难以解决，因此java并没有使用引用计数算法。

## 1.4.3常用的垃圾回收算法

**目前jvm联合使用分代复制回收算法、标记-清除回收算法、标记-整理回收算法**

**1、复制回收算法（新生代垃圾回收算法）**

该算法的核心是将可用内存按容量分为大小相等的两块，每次只用其中的一块，当其中一块的内存用完，就将存活的对象复制到另一块上面，然后把自己的内存空间一次清理掉

**优点：**不必考虑内存碎片的问题；效率高

**缺点**：可用容量为原来的一半，浪费空间

**最优设置**：90%的对象都是朝生夕死，所以使用10%的空间作为交换区，因为交换区必须有两个，所以采用交换算法的新生代中的三个区采用8:1:1的默认分配比例

**新生代的对象分配**，基本上新的对象优先分配在Eden区，当Eden区没有足够的空间时，会发起一次minorGC（第一次GC）

**大对象直接进入老年代**，超过Eden的剩余空间或超过一个参数值（-XX：PretenureSizeThreshold=xx，无默认值）

**提前晋升（组团）**：动态年龄判断，如果在survivor空间中相同年龄所有对象的大小的总和大于survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代

**年龄阀值：**每个对象定义了年龄计数器，经过一次minorGC后（在交换区）年龄加1，对象年龄达到15（可以通过参数-XX:MaxTenuringThreshold设置，默认15）之后，将会晋升到老年代，老年代空间不够时进行fullGC

1. **标记-清除算法Mark-Sweep**

最基础的垃圾收集算法，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成之后统一回收掉所有被标记的对象。

标记-清除算法的缺点有两个：首先，效率问题，标记和清除效率都不高。其次，标记清除之后会产生大量的不连续的内存碎片，空间碎片太多会导致当程序需要为较大对象分配内存时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

1. **标记-整理算法Mark-Compact**

标记-整理算法在标记-清除算法基础上做了改进，标记阶段是相同的标记出所有需要回收的对象，在标记完成之后不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，在移动过程中清理掉可回收的对象，这个过程叫做整理。

标记-整理算法相比标记-清除算法的优点是内存被整理以后不会产生大量不连续内存碎片问题。

复制算法在对象存活率高的情况下就要执行较多的复制操作，效率将会变低，而在对象存活率高的情况下使用标记-整理算法效率会大大提高。

FullGC时程序停掉，只进行垃圾回收操作，minorGC避免不了，fullGC必须尽量避免

处理方式：调整内存设置控制垃圾回收频率，选择合适的垃圾回收器

Jconsole 能效FullGC MAT java分析dump文件

## 1.4.4常用的垃圾回收器 1. Serial/Serial Old

最古老的收集器，是一个单线程收集器，用它进行垃圾回收时，必须暂停所有用户线程。Serial是针对新生代的收集器，采用Copying算法；而Serial Old是针对老生代的收集器，采用Mark-Compact算法。优点是简单高效，缺点是需要暂停用户线程。

## 2. ParNew

Seral/Serial Old的多线程版本，使用多个线程进行垃圾收集。

它是运行在server模式下的首选新生代收集器，除了Serial收集器外，目前只有它能与CMS收集器配合工作。CMS收集器是一个被认为具有划时代意义的并发收集器，因此如果有一个垃圾收集器能和它一起搭配使用让其更加完美，那这个收集器必然也是一个不可或缺的部分了

## 3. Parallel Scavenge

新生代的并行收集器，回收期间不需要暂停其他线程，采用Copying算法。该收集器与前两个收集器不同，主要为了达到一个可控的吞吐量。

## 4. Parallel Old

Parallel Scavenge的老生代版本，采用Mark-Compact（标记整理）算法和多线程。

## 5. CMS

Current Mark Sweep收集器是一种以最小回收时间停顿为目标的并发回收器，因而采用Mark-Sweep算法。

## 6. G1

G1(Garbage First)收集器技术的前沿成果，是面向服务端的收集器，能充分利用CPU和多核环境。是一款并行与并发收集器，它能够建立可预测的停顿时间模型。

## 垃圾回收器的分类

**吞吐量：**CPU用于用户代码的时间/CPU总消耗时间的比值，即=运行用户代码的时间/(运行用户代码时间+垃圾收集时间)。比如，虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

1. **串行垃圾回收器（Serial Garbage Collector）**

串行垃圾回收器通过持有应用程序所有的线程进行工作。它为单线程环境设计，只使用一个单独的线程进行垃圾回收，通过冻结所有应用程序线程进行工作，所以可能不适合服务器环境。它最适合的是简单的命令行程序。

1. **并行垃圾回收器（Parallel Garbage Collector）**

并行垃圾回收器也叫做 throughput collector 。它是JVM的默认垃圾回收器。与串行垃圾回收器不同，它使用多线程进行垃圾回收。相似的是，它也会冻结所有的应用程序线程当执行垃圾回收的时候

1. **并发标记扫描垃圾回收器（CMS Garbage Collector老年代）**

CMS收集器（Concurrent Mark Sweep：并发标记清除）是一种以获取最短回收停顿时间为目标的收集器。适合应用在互联网站或者B/S系统的服务器上，这类应用尤其重视服务器的响应速度，希望系统停顿时间最短。   
CMS收集器运行过程：（着重实现了标记的过程）   
（1）初始标记   
　　根可以直接关联到的对象 速度快   
（2）并发标记（和用户线程一起）   
　　主要标记过程，标记全部对象   
（3）重新标记   
　　由于并发标记时，用户线程依然运行，因此在正式清理前，再做修正   
（4）并发清除（和用户线程一起）   
　　基于标记结果，直接清理对象   
整个过程如下图所示：



上图中，初始标记和重新标记时，需要stop the world。整个过程中耗时最长的是并发标记和并发清除，这两个过程都可以和用户线程一起工作。   
优点：   
　　并发收集，低停顿   
缺点：   
（1）导致用户的执行速度降低。   
（2）无法处理浮动垃圾。因为它采用的是标记-清除算法。有可能有些垃圾在标记之后，需要等到下一次GC才会被回收。如果CMS运行期间无法满足程序需要，那么就会临时启用Serial Old收集器来重新进行老年代的收集。   
（3）由于采用的是标记-清除算法，那么就会产生大量的碎片。往往会出现老年代还有很大的空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次full GC。   
疑问：既然标记-清除算法会造成内存空间的碎片化，CMS收集器为什么使用标记清除算法而不是使用标记整理算法：   
答案：   
CMS收集器更加关注停顿，它在做GC的时候是和用户线程一起工作的（并发执行），如果使用标记整理算法的话，那么在清理的时候就会去移动可用对象的内存空间，那么应用程序的线程就很有可能找不到应用对象在哪里。

 并发标记垃圾回收器只会在下面两种情况持有应用程序所有线程。

1. 当标记的引用对象在tenured区域；
2. 在进行垃圾回收的时候，堆内存的数据被并发的改变。

相比并行垃圾回收器，并发标记扫描垃圾回收器使用更多的CPU来确保程序的吞吐量。如果我们可以为了更好的程序性能分配更多的CPU，那么并发标记上扫描垃圾回收器是更好的选择相比并发垃圾回收器。

1. **G1垃圾回收器（G1 Garbage Collector）**

G1垃圾回收器适用于堆内存很大的情况，他将堆内存分割成不同的区域，并且并发的对其进行垃圾回收。G1也可以在回收内存之后对剩余的堆内存空间进行压缩。并发扫描标记垃圾回收器在STW情况下压缩内存。G1垃圾回收会优先选择第一块垃圾最多的区域

**它最大的优点是结合了空间整合，不会产生大量的碎片，也降低了进行gc的频率**。二是可以让使用者明确指定停顿时间。（可以指定一个最小时间，超过这个时间，就不会进行回收了）它有了这么高效率的原因之一就是：对垃圾回收进行了划分优先级的操作，这种有优先级的区域回收方式保证了它的高效率。   
如果你的应用追求停顿，那G1现在已经可以作为一个可尝试的选择；如果你的应用追求吞吐量，那G1并不会为你带来什么特别的好处。

**垃圾回收器的选择**



## 1.4.5 GC

Minor GC触发条件：当新生代无法为新生对象分配内存空间的时候，会触发Minor GC，比如Eden区满了会触发一次  
Major GC触发条件：回收老年代，通常至少经历过一次Minor GC，Minor GC后导致老年代空间不足  
Full GC触发条件：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法区空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

# 1.5 代理

## 1.5.1 代理的概念

代理就是设计模式，目的是希望能做到代码重用，具体来讲，代理设计模式是通过不直接访问被代理的对象的方式，而访问被代理对象的方法

## 1.5.2 代理的使用场景

**a.** 设计模式中有一个设计原则是开闭原则，是说的对修改关闭对扩展开放 不清楚代码逻辑的情形下使用代理对类进行增强

**b.** RPC框架 框架不明确的知道要调用哪些接口的哪些方法，使用动态代理方式来建立一个中间人给客服端使用，也方便框架进行搭建逻辑

**c.** spring的AOP机制就是采用动态代理的机制来实现切面编程

## 1.5.3 动态代理与静态代理

### 1、静态代理

对于静态代理方式代理类也要去实现和被代理类相同的一个接口

### 2、动态代理

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

InvocationHandler 方法委托类，我们通过代理调用被代理类的方法时，就可以将方法名和方法参数都委托给该类

ClassLoader loader 指定一个动态加载代理类的类加载器

Class<?>[] interfacesClasses 指明被代理类实现的接口，之后我们通过拼接字码生成的类才能知道调用哪些方法

$prox类 继承Proxy 实现了service接口

2. 执行了代理对象的方法 --》 InvocationHandler invoke方法

传入的参数分别 当前的代理对象，当前执行的方法 和参数

InvocationHandler提供invoke方法，proxy顶层接口维护InvocationHandler的实例，通过newProxyInstance给InvocationHandler的实例赋值，InvocationHandler通过自身的构造参数获取被代理类对象

**小结： 动态代理局限性**

通过反射类Proxy和InvocationHandler回调接口实现的JDK动态代理，必须要求委托类实现一个接口，但是事实上 不是所有类都有接口，如果有些没有接口的类，他们是无法使用动态代理的

**总结**：

jdk代理让我们在不直接访问某些对象的情况下，可以通过代理机制间接访问被代理对象的方法

AOP RPC

cglib进行任意代理

**字节码修改技术**

class文件格式

字节码修改技术 ASM JAVASSIST

cglib基于封装 的ASM Spring就是使用cglib代理库

ASM 是一个字节码的操控框架 它能够以二进制形式修改已有类或者动态生成类，ASM可以直接产生二进制文件，也可以被类加载家载入Java虚拟机之前动态改变类行为

操作的级别是底层JVM的汇编指令级别

JAVASSIST --》 生成Java字节码文件的开源框架

是一个开源的分析，编辑和创建Java字节码的类库

不需要了解虚拟机的指令，就能动态改变类的结构或者动态生成类





# 1.6 finally

## 1.6.1、finally的使用逻辑

1、除了以下2种情况外，不管有木有出现异常，finally块中代码都会执行；

①程序未进入try{}块的执行，如在try之前出现运行时异常，程序终止。

②程序进入到try{}和catch{}块的执行，但是在try{}或者catch{}块碰到了System.exit(0)语句，jvm直接退出。 finally{}块不会执行

2、当try和catch中有return时，finally仍然会执行；

3、finally是在return后面的表达式运算后执行的（此时并没有返回运算后的值，而是先把要返回的值的引用地址保存起来，而不管finally中的代码怎么样，最后返回的都是这个引用地址（或者说这个引用地址指向的对象），而这个返回值在finally中会被不会被改变要分以下2种情况）。

①若这个返回值是基本数据类型（int,double）或者不可变类对象（如String,Integer），

②则不管finally中的代码怎么样，返回的值都不会改变，仍然是之前保存的值，若这个值是可变类对象），所以函数返回值是在finally执行前确定的；

1. finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值，而是finally中的return值。

## 1.6.2、finally的原理及实例

**1.finally不执行的情况**

|  |
| --- |
| public class Test {  public static void main(String[] args) {  System.out.println("return value of test(): " + test());  }    public static int test() {  int i = 1;    //  if(i == 1)  //   return 0;  System.out.println("the previous statement of try block");  i = i / 0;    try {   System.out.println("try block");    return i;    }finally {    System.out.println("finally block");   }  }  } |

清单 1 的执行结果如下：

|  |  |
| --- | --- |
| 1  2 | the previous statement of try block  Exception in thread "main" java.lang.ArithmeticException: / by zero  at com.bj.charlie.Test.test(Test.java:15)  at com.bj.charlie.Test.main(Test.java:6) |

另外，如果去掉上例中被注释的两条语句前的注释符，执行结果则是：

|  |  |
| --- | --- |
| 1 | return value of test(): 0 |

在以上两种情况下，finally 语句块都没有执行，说明什么问题呢？只有与 finally 相对应的 try 语句块得到执行的情况下，finally 语句块才会执行。以上两种情况，都是在 try 语句块之前返回（return）或者抛出异常，所以 try 对应的 finally 语句块没有执行。  
那好，即使与 finally 相对应的 try 语句块得到执行的情况下，finally 语句块一定会执行吗？不好意思，这次可能又让大家失望了，答案仍然是否定的。请看下面这个例子（清单 2）。  
**清单 2.**

|  |  |
| --- | --- |
| 5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Test {  public static void main(String[] args) {  System.out.println("return value of test(): " + test());  }    public static int test() {  int i = 1;    try {  System.out.println("try block");  System.exit(0);  return i;  }finally {  System.out.println("finally block");   }  }  } |

清单 2 的执行结果如下：

|  |  |
| --- | --- |
| 1 | try block |

finally 语句块还是没有执行，为什么呢？因为我们在 try 语句块中执行了 System.exit (0) 语句，终止了 Java 虚拟机的运行。那有人说了，在一般的 Java 应用中基本上是不会调用这个 System.exit(0) 方法的。OK ！没有问题，我们不调用 System.exit(0) 这个方法，那么 finally 语句块就一定会执行吗？  
再一次让大家失望了，答案还是否定的。当一个线程在执行 try 语句块或者 catch 语句块时被打断（interrupted）或者被终止（killed），与其相对应的 finally 语句块可能不会执行。还有更极端的情况，就是在线程运行 try 语句块或者 catch 语句块时，突然死机或者断电，finally 语句块肯定不会执行了。可能有人认为死机、断电这些理由有些强词夺理，没有关系，我们只是为了说明这个问题。

**finally 语句剖析**  
说了这么多，还是让我们拿出些有说服力的证据吧！还有什么证据比官方的文档更具说服力呢？让我们来看看官方网站上的《The Java Tutorials》中是怎样来描述 finally 语句块的吧！  
以下位于 \*\*\*\* 之间的内容原封不动的摘自于《 The Java Tutorials 》文档。  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
The finally Block  
The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return,continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.  
Note: If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
请仔细阅读并认真体会一下以上两段英文，当你真正的理解了这两段英文的确切含义，你就可以非常自信的来回答“finally 语句块是否一定会执行？”这样的问题。看来，大多时候，并不是 Java 语言本身有多么高深，而是我们忽略了对基础知识的深入理解。  
接下来，我们看一下 finally 语句块是怎样执行的。在排除了以上 finally 语句块不执行的情况后，finally 语句块就得保证要执行，既然 finally 语句块一定要执行，那么它和 try 语句块与 catch 语句块的执行顺序又是怎样的呢？还有，如果 try 语句块中有 return 语句，那么 finally 语句块是在 return 之前执行，还是在 return 之后执行呢？带着这样一些问题，我们还是以具体的案例来讲解。  
关于 try、catch、finally 的执行顺序问题，我们还是来看看权威的论述吧！以下 \*\*\*\* 之间的内容摘自 Java 语言规范第四版（《The Java™ Programming Language, Fourth Edition》）中对于 try，catch，和 finally 的描述。  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
12.4. Try, catch, and finally

|  |  |
| --- | --- |
| 3  4  5  6  7  8  9  10  11 | You catch exceptions by enclosing code in Try blocks. The basic syntax for a Try block is:  try {  statements  } catch (exception\_type1 identifier1) {  statements  } catch (exception\_type2 identifier2) {  statements  ...  } finally {  statements  } |

where either at least one catch clause, or the finally clause, must be present. The body of the try statement is executed until either an exception is thrown or the body finishes successfully. If an exception is thrown, each catch clause is examined in turn, from first to last, to see whether the type of the exception object is assignable to the type declared in the catch. When an assignable catch clause is found, its block is executed with its identifier set to reference the exception object. No other catch clause will be executed. Any number of catch clauses, including zero, can be associated with a particular TRy as long as each clause catches a different type of exception. If no appropriate catch is found, the exception percolates out of the try statement into any outer try that might have a catch clause to handle it.  
If a finally clause is present with a try, its code is executed after all other processing in the try is complete. This happens no matter how completion was achieved, whether normally, through an exception, or through a control flow statement such as return or break.  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
上面这段文字的大体意思是说，不管 try 语句块正常结束还是异常结束，finally 语句块是保证要执行的。如果 try 语句块正常结束，那么在 try 语句块中的语句都执行完之后，再执行 finally 语句块。如果 try 中有控制转移语句（return、break、continue）呢？那 finally 语句块是在控制转移语句之前执行，还是之后执行呢？似乎从上面的描述中我们还看不出任何端倪，不要着急，后面的讲解中我们会分析这个问题。如果 try 语句块异常结束，应该先去相应的 catch 块做异常处理，然后执行 finally 语句块。同样的问题，如果 catch 语句块中包含控制转移语句呢？ finally 语句块是在这些控制转移语句之前，还是之后执行呢？我们也会在后续讨论中提到。  
其实，关于 try，catch，finally 的执行流程远非这么简单，有兴趣的读者可以参考 Java 语言规范第三版（《The Java™ Language Specification, Third Edition》）中对于 Execution of try-catch-finally 的描述，非常复杂的一个流程。限于篇幅的原因，本文不做摘录，请感兴趣的读者自行阅读。

**finally 语句示例说明**  
下面，我们先来看一个简单的例子（清单 3）。  
清单 3.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Test {  public static void main(String[] args) {  try {  System.out.println("try block");    return ;  } finally {  System.out.println("finally block");   }  }  } |

清单 3 的执行结果为：

|  |  |
| --- | --- |
| 1  2 | try block  finally block |

清单 3 说明 finally 语句块在 try 语句块中的 return 语句之前执行。我们再来看另一个例子（清单 4）。  
**清单 4.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class Test {  public static void main(String[] args) {  System.out.println("reture value of test() : " + test());  }    public static int test(){  int i = 1;    try {  System.out.println("try block");   i = 1 / 0;  return 1;  }catch (Exception e){  System.out.println("exception block");  return 2;  }finally {  System.out.println("finally block");   }  }  } |

清单 4 的执行结果为：

|  |  |
| --- | --- |
| 1  2  3  4 | try block  exception block  finally block  reture value of test() : 2 |

清单 4 说明了 finally 语句块在 catch 语句块中的 return 语句之前执行。  
从上面的清单 3 和清单 4，我们可以看出，其实 finally 语句块是在 try 或者 catch 中的 return 语句之前执行的。更加一般的说法是，finally 语句块应该是在控制转移语句之前执行，控制转移语句除了 return 外，还有 break 和 continue。另外，throw 语句也属于控制转移语句。虽然 return、throw、break 和 continue 都是控制转移语句，但是它们之间是有区别的。其中 return 和 throw 把程序控制权转交给它们的调用者（invoker），而 break 和 continue 的控制权是在当前方法内转移。请大家先记住它们的区别，在后续的分析中我们还会谈到。  
还是得来点有说服力的证据，下面这段摘自 Java 语言规范第四版（《The Java™ Programming Language, Fourth Edition》），请读者自己体会一下其含义。  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Afinallyclause can also be used to clean up forbreak,continue, andreturn, which is one reason you will sometimes see atryclause with nocatchclauses. When any control transfer statement is executed, all relevantfinallyclauses are executed. There is no way to leave atryblock without executing itsfinallyclause.  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
好了，看到这里，是不是有人认为自己已经掌握了 finally 的用法了？先别忙着下结论，我们再来看两个例子 – 清单 5 和清单 6。  
**清单 5.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class Test {  public static void main(String[] args) {     System.out.println("return value of getValue(): " + getValue());  }    public static int getValue() {     try {          return 0;     } finally {          return 1;      }  }  } |

清单 5 的执行结果：

|  |  |
| --- | --- |
| 1 | return value of getValue(): 1 |

**清单 6.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class Test {  public static void main(String[] args) {     System.out.println("return value of getValue(): " + getValue());  }    public static int getValue() {     int i = 1;     try {          return i;     } finally {          i++;     }  }  } |

清单 6 的执行结果：

|  |  |
| --- | --- |
| 1 | return value of getValue(): 1 |

利用我们上面分析得出的结论：finally 语句块是在 try 或者 catch 中的 return 语句之前执行的。 由此，可以轻松的理解清单 5 的执行结果是 1。因为 finally 中的 return 1；语句要在 try 中的 return 0；语句之前执行，那么 finally 中的 return 1；语句执行后，把程序的控制权转交给了它的调用者 main（）函数，并且返回值为 1。那为什么清单 6 的返回值不是 2，而是 1 呢？按照清单 5 的分析逻辑，finally 中的 i++；语句应该在 try 中的 return i；之前执行啊？ i 的初始值为 1，那么执行 i++；之后为 2，再执行 return i；那不就应该是 2 吗？怎么变成 1 了呢？  
关于 Java 虚拟机是如何编译 finally 语句块的问题，有兴趣的读者可以参考《 The JavaTM Virtual Machine Specification, Second Edition 》中 7.13 节 Compiling finally。那里详细介绍了 Java 虚拟机是如何编译 finally 语句块。实际上，Java 虚拟机会把 finally 语句块作为 subroutine（对于这个 subroutine 不知该如何翻译为好，干脆就不翻译了，免得产生歧义和误解。）直接插入到 try 语句块或者 catch 语句块的控制转移语句之前。但是，还有另外一个不可忽视的因素，那就是在执行 subroutine（也就是 finally 语句块）之前，try 或者 catch 语句块会保留其返回值到本地变量表（Local Variable Table）中。待 subroutine 执行完毕之后，再恢复保留的返回值到操作数栈中，然后通过 return 或者 throw 语句将其返回给该方法的调用者（invoker）。请注意，前文中我们曾经提到过 return、throw 和 break、continue 的区别，对于这条规则（保留返回值），只适用于 return 和 throw 语句，不适用于 break 和 continue 语句，因为它们根本就没有返回值。  
是不是不太好理解，那我们就用具体的例子来做形象的说明吧！  
为了能够解释清单 6 的执行结果，我们来分析一下清单 6 的字节码（byte-code）： Compiled from "Test.java"  
 public class Test extends java.lang.Object{   
 public Test();   
  Code:   
   0:   aload\_0   
   1:invokespecial#1; //Method java/lang/Object."<init>":()V   
   4:   return

  LineNumberTable:   
   line 1: 0

 public static void main(java.lang.String[]);   
  Code:   
   0:   getstatic   #2; //Field java/lang/System.out:Ljava/io/PrintStream;   
   3:   new   #3; //class java/lang/StringBuilder   
   6:   dup   
   7:   invokespecial   #4; //Method java/lang/StringBuilder."<init>":()V   
   10:   ldc   #5; //String return value of getValue():   
   12:   invokevirtual     
   #6; //Method java/lang/StringBuilder.append:(  
       Ljava/lang/String;)Ljava/lang/StringBuilder;   
   15:   invokestatic   #7; //Method getValue:()I   
   18:   invokevirtual     
   #8; //Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;   
   21:   invokevirtual     
   #9; //Method java/lang/StringBuilder.toString:()Ljava/lang/String;   
   24:   invokevirtual   #10; //Method java/io/PrintStream.println:(Ljava/lang/String;)V   
   27:   return

 public static int getValue();   
  Code:   
   0:   iconst\_1   
   1:   istore\_0   
   2:   iload\_0   
   3:   istore\_1   
   4:   iinc   0, 1   
   7:   iload\_1   
   8:   ireturn   
   9:   astore\_2   
   10:   iinc   0, 1   
   13:   aload\_2   
   14:   athrow   
  Exception table:   
   from   to  target type   
     2     4     9   any   
     9    10     9   any   
 }

对于 Test（）构造方法与 main（）方法，在这里，我们不做过多解释。让我们来分析一下 getValue（）方法的执行。在这之前，先让我把 getValue（）中用到的虚拟机指令解释一下，以便读者能够正确的理解该函数的执行。  
1. iconst\_   
Description: Push the int constant  (-1, 0, 1, 2, 3, 4 or 5) onto the operand stack.  
Forms: iconst\_m1 = 2 (0x2)  iconst\_0 = 3 (0x3)  iconst\_1 = 4 (0x4)    
iconst\_2 = 5 (0x5) iconst\_3 = 6 (0x6)  iconst\_4 = 7 (0x7)  iconst\_5 = 8 (0x8)

2. istore\_   
Description: Store int into local variable. The  must be an index into the   
local variable array of the current frame.   
Forms: istore\_0 = 59 (0x3b)  istore\_1 = 60 (0x3c)  istore\_2 = 61 (0x3d)    
istore\_3 = 62 (0x3e)

3. iload\_   
Description: Load int from local variable. The  must be an index into the   
local variable array of the current frame.   
Forms: iload\_0 = 26 (0x1a)  iload\_1 = 27 (0x1b)  iload\_2 = 28 (0x1c)  iload\_3 = 29 (0x1d)

4. iinc index, const   
Description: Increment local variable by constant. The index is an unsigned byte that   
must be an index into the local variable array of the current frame. The const is an   
immediate signed byte. The local variable at index must contain an int. The value   
const is first sign-extended to an int, and then the local variable at index is   
incremented by that amount.  
Forms:  iinc = 132 (0x84)

Format:  
iinc    
index    
const

5. ireturn   
Description: Return int from method.  
Forms:  ireturn = 172 (0xac)

6. astore\_   
Description: Store reference into local variable. The  must be an index into the   
local variable array of the current frame.  
Forms: astore\_0 = 75 (0x4b) astore\_1 = 76 (0x4c) astore\_2 =77 (0x4d) astore\_3 =78 (0x4e)

7. aload\_   
Description: Load reference from local variable. The  must be an index into the   
local variable array of the current frame.  
Forms: aload\_0 = 42 (0x2a) aload\_1 = 43 (0x2b) aload\_2 = 44 (0x2c) aload\_3 = 45 (0x2d)

8. athrow   
Description: Throw exception or error.  
Forms: athrow = 191 (0xbf)  
有了以上的 Java 虚拟机指令，我们来分析一下其执行顺序：分为正常执行（没有 exception）和异常执行（有 exception）两种情况。我们先来看一下正常执行的情况，如图 1 所示：  
图 1. getValue（）函数正常执行的情况



由上图，我们可以清晰的看出，在 finally 语句块（iinc 0, 1）执行之前，getValue（）方法保存了其返回值（1）到本地表量表中 1 的位置，完成这个任务的指令是 istore\_1；然后执行 finally 语句块（iinc 0, 1），finally 语句块把位于 0 这个位置的本地变量表中的值加 1，变成 2；待 finally 语句块执行完毕之后，把本地表量表中 1 的位置上值恢复到操作数栈（iload\_1），最后执行 ireturn 指令把当前操作数栈中的值（1）返回给其调用者（main）。这就是为什么清单 6 的执行结果是 1，而不是 2 的原因。  
再让我们来看看异常执行的情况。是不是有人会问，你的清单 6 中都没有 catch 语句，哪来的异常处理呢？我觉得这是一个好问题，其实，即使没有 catch 语句，Java 编译器编译出的字节码中还是有默认的异常处理的，别忘了，除了需要捕获的异常，还可能有不需捕获的异常（如：RunTimeException 和 Error）。  
从 getValue（）方法的字节码中，我们可以看到它的异常处理表（exception table）， 如下：

[?](https://www.jb51.net/article/74771.htm)

|  |  |
| --- | --- |
| 1  2  3 | Exception table:  from to target type  2 4 9 any |

它的意思是说：如果从 2 到 4 这段指令出现异常，则由从 9 开始的指令来处理。  
图 2. getValue（）函数异常执行的情况



先说明一点，上图中的 exception 其实应该是 exception 对象的引用，为了方便说明，我直接把它写成 exception 了。  
由上图（图 2）可知，当从 2 到 4 这段指令出现异常时，将会产生一个 exception 对象，并且把它压入当前操作数栈的栈顶。接下来是 astore\_2 这条指令，它负责把 exception 对象保存到本地变量表中 2 的位置，然后执行 finally 语句块，待 finally 语句块执行完毕后，再由 aload\_2 这条指令把预先存储的 exception 对象恢复到操作数栈中，最后由 athrow 指令将其返回给该方法的调用者（main）。  
通过以上的分析，大家应该已经清楚 try-catch-finally 语句块的执行流程了吧！  
为了更具说服力，我们还是来引经据典吧！大家可以不相信我，难道还不相信“高司令”（Gosling）吗？下面这段仍然摘自 Java 语言规范第四版 《The Java™ Programming Language, Fourth Edition》，请读者自己体会吧！  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
a finally clause is always entered with a reason. That reason may be that the try code finished normally, that it executed a control flow statement such as return, or that an exception was thrown in code executed in the Try block. The reason is remembered when the finally clause exits by falling out the bottom. However, if the finally block creates its own reason to leave by executing a control flow statement (such as break or return) or by throwing an exception, that reason supersedes the original one, and the original reason is forgotten. For example, consider the following code:  
try {  
// … do something …   
return 1;  
} finally {  
return 2;  
}  
When the Try block executes its return, the finally block is entered with the “reason” of returning the value 1. However, inside the finally block the value 2 is returned, so the initial intention is forgotten. In fact, if any of the other code in the try block had thrown an exception, the result would still be to return 2. If the finally block did not return a value but simply fell out the bottom, the “return the value 1 ″ reason would be remembered and carried out.  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
好了，有了以上的知识，让我们再来看以下 3 个例子。  
**清单 7.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class Test {  public static void main(String[] args) {     System.out.println("return value of getValue(): " + getValue());  }    @SuppressWarnings("finally")  public static int getValue() {     int i = 1;     try {          i = 4;     } finally {          i++;          return i;     }  }  } |

清单 7 的执行结果：

[?](https://www.jb51.net/article/74771.htm)

|  |  |
| --- | --- |
| 1 | return value of getValue(): 5 |

**清单 8.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class Test {  public static void main(String[] args) {     System.out.println("return value of getValue(): " + getValue());  }    public static int getValue() {     int i = 1;     try {          i = 4;     } finally {          i++;     }       return i;  }  } |

清单 8 的执行结果：

[?](https://www.jb51.net/article/74771.htm)

|  |  |
| --- | --- |
| 1 | return value of getValue(): 5 |

清单 7 和清单 8 应该还比较简单吧！利用我们上面讲解的知识，很容易分析出其结果。让我们再来看一个稍微复杂一点的例子 – 清单 9。我建议大家最好先不要看执行结果，运用学过的知识来分析一下，看是否能推断出正确的结果。  
**清单 9.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class Test {  public static void main(String[] args) {  System.out.println(test());  }    public static String test() {  try {  System.out.println("try block");  return test1();  } finally {  System.out.println("finally block");   }  }  public static String test1() {  System.out.println("return statement");  return "after return";  }  } |

清单 9 的结果：

[?](https://www.jb51.net/article/74771.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | try block  return statement  finally block  after return |

你分析对了吗？其实这个案例也不算很难，return test1();这条语句等同于 :

[?](https://www.jb51.net/article/74771.htm)

|  |  |
| --- | --- |
| 1  2 | String tmp = test1();  return tmp; |

这样，就应该清楚为什么是上面所示的执行结果了吧！  
好了，就写到这吧！希望大家看完这篇文章能够有所收获！

**总结**  
没想到吧！一个小小的、看似简单的 finally 语句块背后居然隐藏了这么多玄机。看来，我们平时还是应该认真的阅读 Java 相关的基础文档，比如：Java 语言规范、Java 虚拟机规范等，很多棘手的问题都可以从中得到答案。只有真正的吃透了基础知识，才能达到运用自如的境界！