**目录**

[JDK、JRE、JVM 1](#_Toc3090)

[对象占多少存储空间 2](#_Toc21892)

[String 3](#_Toc8396)

[初始化&垃圾回收 4](#_Toc29772)

[finalize() 4](#_Toc5427)

[System.gc() 4](#_Toc28153)

[初始化顺序 5](#_Toc18475)

[域、方法、类的访问权限 6](#_Toc1846)

[枚举 6](#_Toc12194)

[内部类 7](#_Toc23143)

[HashMap 8](#_Toc12272)

[迭代 10](#_Toc10234)

[Comparable & Comparator 10](#_Toc10076)

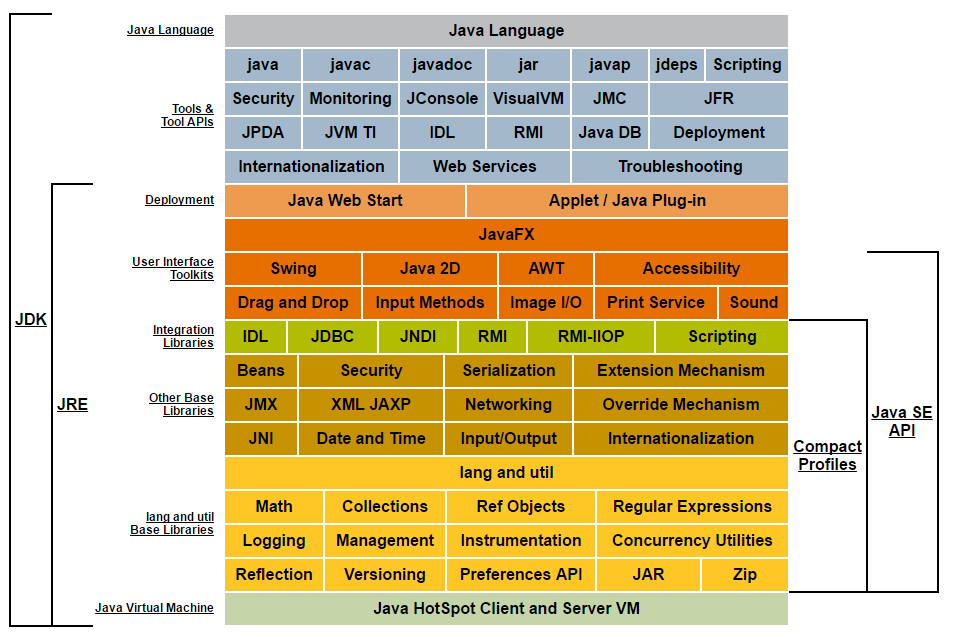
[接口与抽象类的异同点 11](#_Toc15976)

[Object 11](#_Toc5790)

[操蛋的时间 12](#_Toc24354)

# **JDK、JRE、JVM**

从左往右，这是一个包含关系，JDK包含JRE，JRE包含JVM。我觉得没有什么比这张图来的更实在了。



这个包含关系真是不言而喻啊，JDK由javadoc,javap,jdb等组件及JRE组成；JRE由核心类库及JVM组成；JVM是一个虚构出来的计算机，它将.class字节码文件编译成对应的机器语言。

Java 运行已编译完的.class文件

Javac 编译.java文件并生成.class文件

Javadoc 将注释变成文档

Jar 平台无关的文件格式，可以将多个Java程序，连带着.class在内合并为 一个jar文件。

**///////////////////////////////////////////////////////////////////////////////////**

# 对象占多少存储空间

http接口返回一堆数据，从数据库内查询出一堆数据，程序接收，并封装成对象。那么问题来了，这些对象在内存内占多少空间？有时候，看一个http接口返回500kb的数据我就感觉害怕，总有一种服务器要爆了的感觉。当然这是开玩笑了，服务器怎么会爆，充其量宕机。

之前一直不知道怎么计算对象在内存上的大小。其实是思维太死，一个java对象，说到底就是由field与method组成的。method我先不管，field，这个就很好理解了，基础类型，包装类型，或者类类型。不过真的仔细追踪下去，其实不管什么类类型，到头来都是由基础类型组成的，君不见数据库也就存存数字、字符串……。因此，以model类为例，这玩意就像一个套娃，不停的把基础类型包来包去。

而基础类型有多大，这是有规定的。理论上通过递归加反射，是可以计算出一个model类的对象至少占多少字节的。

boolean 占1 bit

char 占16 bit（2字节）

short 占16 bit（2字节）

int 占32bit（4字节）

long 占64bit（8字节）

float 占32bit（4字节）

double 占64bit（8字节）

至于程序本身是否在内存里占有一定空间，这他妈也是一门学问，奈何知识水平有限啊。

**///////////////////////////////////////////////////////////////////////////////////**

# String

* 常用方法

// 根类Object的方法就不算了。

String是不可变对象，也就是说，无论你做什么操作该字符串对象都是不会变的。看到着你一定觉得扯淡，因为如果对象不可变，那我截取或是拼接后得到的字符串又是哪来的？

全是new出来的，你对字符串A做的任何更改都会被new成字符串B返回给你，感觉没变化是因为引用没变。

String str = “shabi”;

str = str.subString(0，3);

看上去像是从字符串“shabi”中截取字符串“sha”,其实是new了一个新字符串然后把“sha”赋进去。此刻引用str指向新字符串“sha”，丢失了引用的“shabi”只能静静等待垃圾回收。

subString源码：

**return** (beginIndex == 0) ? **this** : **new** String(value, beginIndex, subLen);

**///////////////////////////////////////////////////////////////////////////////////**

# 初始化&垃圾回收

## finalize()

不能妖魔化这个方法。

我们先看源码

**protected** **void** finalize() **throws** Throwable { }

一共只有1行，压根就是个空方法。

当某对象被垃圾回收时，此对象的finalize()方法会被调用，但因为默认的finalize()方法是空的，所以如果不重载它，那么什么都不会发生。

什么情况下需要重载呢。

由于垃圾回收仅能释放关键字new创建的对象。因此如果一个对象中的部分存储空间是由C的malloc()方法赋予的，那么只能调用相应的free()方法才能释放，垃圾回收机制起不了作用。解决方案是覆盖Object的finalize()方法，在里面用本地方法调用free()或其他的释放函数。

至于为什么会出现C语言的malloc()函数，是因为Java存在native方法。

还有，垃圾回收不是一定会发生的，因为垃圾回收本身有开销，所以不到程序内存不够用的时候，不会去浪费性能做垃圾回收。

*垃圾回收的工作原理“假定”是这样的：一旦垃圾回收器准备好释放对象占用的存储空间，将首先调用其finalize()方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。所以要是你打算用finalize()，就能在垃圾回收时刻做一些重要的清理工作。*

————thinking in Java

因此finalize()还有另一种用法，终结条件，就是在finalize()里加if-else，并输出相关信息，老实讲，我觉得这更像log。

## System.gc()

与Runtime.getRuntime().gc()等价

强制进行回收动作

不过看文档，这个方法也只是建议JVM此刻进行垃圾回收，然后JVM会尽可能的去释放它认为可以被回收的内存空间。这个方法不是回收全部对象，也就是说方法调用结束后，某些你希望被回收的对象可能依旧存在。

/\*\*

\* Runs the garbage collector.

\* <p>

\* Calling the <code>gc</code> method suggests that the Java Virtual

\* Machine expend effort toward recycling unused objects in order to

\* make the memory they currently occupy available for quick reuse.

\* When control returns from the method call, the Java Virtual

\* Machine has made a best effort to reclaim space from all discarded

\* objects.

\* <p>

\* The call <code>System.gc()</code> is effectively equivalent to the

\* call:

\* <blockquote><pre>

\* Runtime.getRuntime().gc()

\* </pre></blockquote>

\*

\* **@see** java.lang.Runtime#gc()

\*/

## 初始化顺序

结论第一：

先初始化静态变量静态代码块，然后初始化普通变量普通代码块，最后才是构造函数。如果有父类，那就先父类再子类。另外，静态变量及代码块只会初始化一次。

初始化代表一个动作，这个动作概括起来就是，当new一个类的对象时，为该类成员变量赋默认值。

class A{

static int j;

int i;

String str;

}

当 A a = new A(); 时，变量j第一个有默认值0，然后变量i的默认值为0，变量str的默认值为null。

粗暴来看，好像不new对象，这一切都不会发生，可事实上却存在一种特殊情况。由于static变量可以直接通过类来使用（像下面）

A.j;

因此，只要有地方这样用了静态变量，那么即便没有new对象，静态成员们还是会被

初始化。注意“们”这个字，静态变量以及静态代码块都会被初始化，很团结、很社会主义的。然后又由于静态成员只会被初始化一次，所以下次开始new对象时，静态成员们就不在参与初始化过程了。

栗子：

**public** **class** Work {

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

System.***out***.println(A.*j*);

**new** A();

}

}

**class** A {

**static** **int** *j*;

**static** {

System.***out***.println("第一个静态块：j被初始化了");

}

**static** {

System.***out***.println("第二个静态块：");

}

}

**///////////////////////////////////////////////////////////////////////////////////**

# 域、方法、类的访问权限

这个知识点很简单，对于域与方法。

private是类内访问权限。

啥都不写是包内访问权限。

protected除包内访问权限外，继承后的子类也可以访问。

public是，只要import了这个类，可以尽情访问。

有意思的是这个：

public class A{}

类的前面也有“public”关键字。其实这个关键字相当重要，因为类名前如果没有“public”，那么这个类是不能被import的。

类名前不加public也不会报错，这样就代表包内访问权限，粗暴点，只有在这个包内，才能创建这个类的对象。

类名前加private和protected就要报错了。搞不懂创始团队脑子里怎么想的，其实这个特性，基本来讲用不到，就默认类名前都要加public关键字就好了。

**///////////////////////////////////////////////////////////////////////////////////**

# 继承

new子类对象时，编译器会自动去new父类对象，然后这个父类对象就一直隐藏在子类对象里。

针对这条规则，这里头有个生僻的知识点。

new这个动作必然通过构造方法，如果父类仅有一个构造方法且无参，那么编译器会自动调用。如果父类仅有一个构造方法且带参，那么子类就必须显示调用了，不然就要报错。

**public** **class** A {

**private** **int** i;

**public** A(**int** i) {

**this**.i = i;

}

}

**public** **class** Work **extends** A {

**public** Work() {

**super**(1);

}

}

// 感觉没什么卵子用，现在的企业级项目，因为分工明确，已经不依赖构造函数这种东西了，基本都是get&set方法。

## 代理

这是一种设计模式。

public class A{

void doSomething(){

// syso ……  
}

}

public class AProxy{

A a = new A();

void doSomething(){

// syso ……

a.doSomething();

}

}

真正使用这种模式的时候不可能真的像上面那样把A对象new出来，可以通过各种配置参数来获取想要的实例。

真正使用这种模式的时候也不可能向上面那样，代理的方法只是很单纯的调用一下A对象的方法，总要在干点别点什么。

## @Override

表示子类覆盖父类的方法，可写可不写但最好还是写上。因为：（底下我百度来的）

1．可以当注释用，方便阅读．

2．告诉阅读你代码的人，这是方法的复写．

3．编译器可以给你验证@Override下面的方法名是否是你父类中所有的，如果没有则报错．

**///////////////////////////////////////////////////////////////////////////////////**

# 枚举

具名值

ordinal()

values()

**///////////////////////////////////////////////////////////////////////////////////**

# 内部类

把类的定义放在外部类的里面，就是内部类。

外部类的非static方法可以随意创建内部类对象。Static方法除了要显示指明，还要通过外部类对象：

OuterClass.InnerClass in = out.new InnerClass ;

用 .new 创建毕竟还是麻烦，所以向下面那样额外写一个方法用来创建内部类对象会方便点。

**public** **class** Outer {

**class** Inner {

// TODO....

}

**public** Inner createInner() {

**return** **new** Inner();

}

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

Outer out = **new** Outer();

Outer.Inner in = out.createInner();

System.***out***.println(in);

}

}

内部类可以随意访问外部类的所有元素。那这是为什么呢？当创建内部类对象时，此内部类对象必然会获取外围类对象的引用，内部类对象访问外围类对象成员就是通过这个引用。当然这是比较底层的知识了，一般程序员接触不到。

不扣底层知识的话还有另一种理解方式，private、default、protected、public是Java的四种访问权限。外部类成员变量private即类内访问权限，内部类在外部类内，因此可以随意访问变量及方法。

// 一点感想，访问对象，访问静态类，不存在单独访问变量这种说法，都是要通过媒介的

当将内部类向上转型为其基类，尤其是转型为一个接口时，内部类就有了用武之地。这是因为此内部类——某个接口的实现——能够完全不可见，并且不可用（？）。所得的只是指向基类或接口的引用，所以能够很方便的隐藏实现细节。

***匿名内部类***

目光看向匿名内部类，情况顿时有点复杂。（好吧夸张了）匿名内部类的出现，主要是有时候，你只需要调用某接口或某类的方法，犯不着特意去写个子类来实现或继承，完了还要创建对象。因此最常见的使用方式是 return new A(){...};

// 自己总结的不知道对不对

**interface** A {

**public** **void** say();

}

**public** **class** Test {

**public** A create(**final** **int** i){

**return** **new** A(){

**private** **int** b = i;

**public** **void** say(){

System.***out***.println(b);

}

};

}

}

烦的是上面那种情况，有时候匿名内部类会需要使用外部变量，这时候传进方法内的引用必须是final类型。还有一种情况是，匿名内部类继承的那个类只有一个带参构造函数，那这时候新建匿名内部类对象的伪构造函数也要带参。

return new A(i){...};

匿名内部类只是一种比较吊诡的继承。。。。。（不知道这么理解好不好）

***嵌套类***

也叫静态内部类，被static修饰的内部类。

创建静态内部类（嵌套类）对象，不需要其外围类的对象。

静态内部类（嵌套类）不能访问非静态的外围类成员。

普通内部类不能包含静态成员，静态内部类（嵌套类）可以包含一切

**///////////////////////////////////////////////////////////////////////////////////**

# HashMap

Map，是一个以键值对方式存储数据的容器。

取数据时会调用key的hashcode()方法，若在map中找到了相应的key，在调用equals()方法进行比较。相同，则返回数据；不同，返回null。

存数据时也是这个道理，调用hashcode()以及equals()。存在该key就替换为新值，不存在直接存入map。散列还涉及到碰撞等问题，虽说是底层，但这个知识点很重要。

***静态常量***

**static** **final** **int** ***DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4;

**static** **final** **int** ***MAXIMUM\_CAPACITY*** = 1 << 30;

**static** **final** **float** ***DEFAULT\_LOAD\_FACTOR*** = 0.75f;

**static** **final** **int** ***TREEIFY\_THRESHOLD*** = 8;

**static** **final** **int** ***UNTREEIFY\_THRESHOLD*** = 6;

**static** **final** **int** ***MIN\_TREEIFY\_CAPACITY*** = 64;

// 这种东西都是二话不说直接上源码。

默认初始容量，16；最大容量，2^30；负载因子，0.75

顺带一提ArrayList的默认初始容量为10，当元素个数超过10时会扩容，扩容规则是乘1.5倍再加1。

负载因子决定了HashMap在自动扩容前的装填程度，0.75意味着填入12个键值对后，HashMap就要进行扩容。负载因子越大，装填程度越高，但相应的查找效率低，反之亦然。

当一个桶上的节点个数大于8时链表转树，小于6时树转链表。64是树的最低容量。

***遍历HashMap***

1. 内部类Map.Entry<K, V>

HashMap有个内部类，会保存键值对，只是这个造型总让我误以为这内部类也是个map，但人家只是个泛型啊。（我泛型学的真差。。。。）HashMap还有个方法：

public Set<Map.Entry<K, V>> entrySet()

那么很自然而然的：

foreach(Map.Entry<> entry：map.entrySet()){

System.out.println(entry.getKey()+””+entry.getValue());

}

1. keySet() & values()

上面那个方法是获取完整的键值对，这里就是只获取键或者只获取值。

public Set<K> keySet()

public Collection<V> values()

// 我觉得接下来的这两个方法都很无聊

3. Iterator + entrySet()

4. Iterator + keySet()

**///////////////////////////////////////////////////////////////////////////////////**

# 迭代

java.util.Enumeration<E>

是interface，而且只有两个方法。hasMoreElements() 与 nextElement()。JDK 1.0就存在的老东西，已被Iterator接口替代。

// 老子就好奇了，为什么ServletConfig取参数还要用着老接口？

java.lang.Iterable<T>

也是接口，我看得懂的方法就1个 Iterator<T> Iterator()，这货是为生产Iterator而生的。

java.util.Iterator<E>

还是接口，主要方法3个hasNext()，next()，remove()，多了个删除功能。

java.util.ListIterator<E>

还有一个接口，这个接口只能用来遍历List，有9个方法，把增删查改都包含在内，前后向遍历并获取索引位置。

***快速失败(fail-fast)与安全失败(fail-safe)***

// 高级知识啊。。。牛客带我看世界。

它们与迭代器有关，牵扯到java.util包与java.util.concurrent包。首先除了这俩技术名词，再背一个异常ConcurrentModificationException。简单点，快速失败会抛这个异常，而安全失败不会抛。快速失败主要来源于“并发修改”，即一个ArrayList，线程A在迭代它时，线程B对它进行了结构上的修改。也因此快速失败不是肯定会发生的。

**///////////////////////////////////////////////////////////////////////////////////**

# Comparable & Comparator

俩接口，作用都是排序java.lang.Comparable与java.util.Comparator，而且均为泛型。

Comparable接口只有一个方法 compareTo(T t) ，返回值为int。实现了comparable接口的类可以直接使用Collections.sort()与Arrays.sort()方法。

Comparator接口主要有两个方法，compare（T o1，T o2）与equals()。equals()方法可以不重载，它的作用就是保证两个comparator相等。将比较器作为Collections.sort()及Arrays.sort()的参数并调用。

直接调用内部方法与间接调用外部方法的区别。

comparator的升序与降序。

o1 < o2

如果return 1；那么这是升序。

如果return -1；那么这是降序。

**///////////////////////////////////////////////////////////////////////////////////**

# 接口与抽象类的异同点

相同点：

可以被继承，但都不能被实例化。

不同点：

从变量上讲，接口中一般不包含成员变量，如果有必须是static final 类型的；抽象类没有这种限制。

从方法上讲，接口中的方法自动是public abstract类型的；抽象类的可以存在普通方法访问类型也可以自己设置。

从继承上讲，接口可以实现多个，抽象类只能继承一个。

**///////////////////////////////////////////////////////////////////////////////////**

# Object

protected Object clone()

// protected我不是很懂，因为从访问权限上讲，任何类都自动继承Object，因此该方法也是可以随意访问的。

使用前有一点要注意，调用该方法的对象的类必须实现Cloneable接口，不然会抛出CloneNotSupportedException。

这个方法让我想起了C++的浅拷贝与深拷贝。类内可能还包含其他类的引用，直接调用clone()方法只会复制引用，而引用背后的对象是不会复制，这种浅拷贝存在很大的不稳定因素，比如修改。

要适当重载，虽说这个方法我根本就没用过。。。。

public boolean equals(Object obj)

我在这个方法上栽过跟头，不止一次。比较字符串之间是否相等就要用equals()不能用==，这个操作符只比较引用。另外默认的equals()方法只比较内存地址，String类的equals()方法是重载过的，它会完整比较两个字符串的字符。

protected void finalize()

并非释放内存，详情见垃圾回收。

////////////////////////////////////////////////////////////////////////////////////////////

***///////////////////////////////////////////////////////////////////////////////////***

# 操蛋的时间

作者：马小明  
链接：https://www.zhihu.com/question/23730083/answer/25475256  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

yyyy-MM-dd HH:mm:ss   
年-月-日 时:分:秒  
大写是为了区分“**月**”与“**分**”

顺便说下HH为什么大写，是为了区分**12小时制**与**24小时制**。  
小写的h是12小时制，大写的H是24小时制。

书写格式和语言规定有关，上述写法是Windows系统中的我们常见的写法，包括日期设置于办公软件在内。在其他语言中有类似的但使用符号或格式不同的写法。

有的时候我们会看到这样的格式：yyyy-M-d H:m:s  
mm与m等，它们的区别为**是否有前导零**：H,m,s表示**非零开始**，HH,mm,ss表示**从零开始**。  
比如凌晨1点2分，HH:mm显示为**01:02**，H:m显示为**1:2**。

以2014年1月1日凌晨1点1分1秒（当天是星期三）为例子介绍一下其他的：  
**yyyy/yyy/yy/y** 显示为 **2014/2014/14/4**  
（3个y与4个y是一样的，为了便于理解多写成4个y）

**MMMM/MMM/MM/M** 显示为 **一月/一月/01/1**  
（4个M显示全称，3个M显示缩写，不过中文显示是一样的，英文就是January和Jan）

**dddd/ddd/dd/d** 显示为 **星期三/周三**(有的语言显示为“三”)**/01/1**  
（在英文中同M一样，4个d是全称，3个是简称；  
dddd/ddd表示星期几，dd/d表示几号）

**HH/H/hh/h** 显示为 **01/1/01 AM/1 AM**

剩下的**mm/m/ss/s**只是前导零的问题了。

yyyy/M/d/dddd H:mm:ss 就是 2014年1月1日星期三 1:01:01