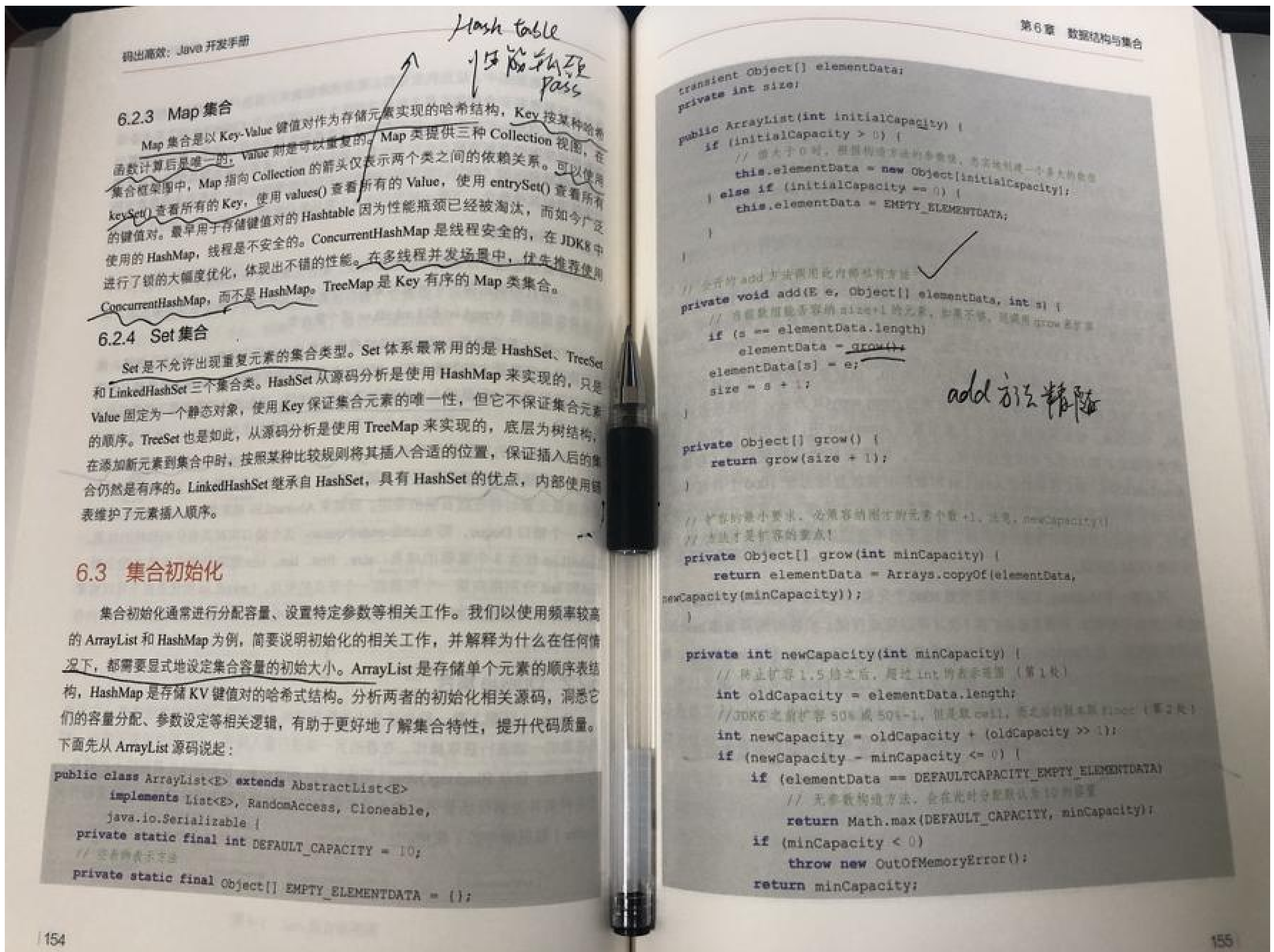
链滴

《码出高效》系列笔记（一）：面向对象中的其他知识点

作者：matthewhan
原文链接：https://ld246.com/article/1579142810371
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)



6.2.3 Map 集合

Map 集合是以 Key-Value 键值对作为存储元素实现的哈希结构，Key 按某种哈希函数计算后是唯一的，Value 则是可以重复的。Map 类提供三种 Collection 视图，在集合框架图中，Map 指向 Collection 的箭头仅表示两个类之间的依赖关系。可以使用 keySet() 查看所有的 Key，使用 values() 查看所有的 Value，使用 entrySet() 查看所有的键值对。最早用于存储键值对的 Hashtable 因为性能瓶颈已经被淘汰，而如今广泛使用的 HashMap，线程是不安全的。ConcurrentHashMap 是线程安全的，在 JDK8 中进行了锁的大幅度优化，体现出不错的性能。在多线程并发场景中，优先推荐使用 ConcurrentHashMap，而不是 HashMap。TreeMap 是 Key 有序的 Map 类集合。

6.2.4 Set 集合

Set 是不允许出现重复元素的集合类型。Set 体系最常用的是 HashSet、TreeSet 和 LinkedHashSet 三个集合类。HashSet 从源码分析是使用 HashMap 来实现的，只是 Value 固定为一个静态对象，使用 Key 保证集合元素的唯一性，但它不保证集合元素的顺序。TreeSet 也是如此，从源码分析是使用 TreeMap 来实现的，底层为树结构，在添加新元素到集合中时，按照某种比较规则将其插入合适的位置，保证插入后的集合仍然是有序的。LinkedHashSet 继承自 HashSet，具有 HashSet 的优点，内部使用链表维护了元素插入顺序。

6.3 集合初始化

集合初始化通常进行分配容量、设置特定参数等相关工作。我们以使用频率较高的 ArrayList 和 HashMap 为例，简要说明初始化的相关工作，并解释为什么在任何情况下，都需要显式地设定集合容量的初始大小。ArrayList 是存储单个元素的顺序表结构，HashMap 是存储 KV 键值对的哈希式结构。分析两者的初始化相关源码，洞悉它们的容量分配、参数设定等相关逻辑，有助于更好地了解集合特性，提升代码质量。下面先从 ArrayList 源码说起：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable,
        java.io.Serializable {
    private static final int DEFAULT_CAPACITY = 10;

    private static final Object[] EMPTY_ELEMENTDATA = {};
```



```
transient Object[] elementData;
private int size;

public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    }
}

private void add(E e, Object[] elementData, int s) {
    if (s == elementData.length)
        elementData = grow();
    elementData[s] = e;
    size = s + 1;
}

private Object[] grow() {
    return grow(size + 1);
}

private Object[] grow(int minCapacity) {
    return elementData = Arrays.copyOf(elementData,
newCapacity(minCapacity));
}

private int newCapacity(int minCapacity) {
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity <= 0) {
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
            return Math.max(DEFAULT_CAPACITY, minCapacity);
        if (minCapacity < 0)
            throw new OutOfMemoryError();
        return minCapacity;
```


良好的编码风格和完善统一的规约是最高效的方式。

前言

本篇汲取了本书中较为精华的知识要点和实践经验加上读者整理，作为本系列里的第一篇章第三节：向对象的其他知识点篇。

本系列目录：

- 《码出高效》系列笔记（一）：面向对象中的类
- 《码出高效》系列笔记（一）：面向对象中的方法
- 《码出高效》系列笔记（一）：面向对象中的其他知识点
- 《码出高效》系列笔记（二）：代码风格
- 《码出高效》系列笔记（三）：异常与日志
- 《码出高效》系列笔记（四）：数据结构与集合的框架
- 《码出高效》系列笔记（四）：数据结构与集合的数组和泛型
- 《码出高效》系列笔记（四）：元素的比较

同步与异步

同步调用是刚性调用，是阻塞式操作，必须等待调用方法体执行结束。而异步调用是柔性调用，是非塞式操作。

举个与我们息息相关的例子，Git 代码提交托管时，提交代码的操作是同步调用，需要实时返回给用结果，但是当前库代码相关活动就不是时间敏感的，在提交代码时，发送一个消息到后台的缓存队列，后台服务器定时消费这些消息即可。

覆写

覆写是多态的一中表现，大部分人可能把它称为重写，我也遵循了本书的命名，重写可能暗示了推倒来的过程，而覆写则更多的表达出重写一部分而覆盖。出现覆写的情况非常的多，POJO 类 toString() 方法的覆写，工厂模式的抽象工厂类调用具体工厂子类的方法。通常这也被称作向上转型：

```
Father father = new Son();
// son 覆写了此方法
father.doSomething();
```

向上转型时，通过父类应用执行子类方法时需要注意以下两点：

- 无法调用到子类中存在而父类本身不存在的方法。
- 可以调用到子类中覆写了父类的方法，这是一种多态实现。

成功覆写父类方法，满足以下 4 个条件：

- 访问权限不能变小。具体表现为父类中 public 的方法，在子类继承覆写该方法时变成了 private，破坏了封装，编译也不会通过，所以不允许将访问权限缩小：

```
class Father {
    public void say() {
        System.out.println("我要逃离这里。");
    }
}
class Son extends Father {
    // 编译报错
    @Override
    private void say() {
        System.out.println("用笔尖微颤歪歪扭扭的线条");
    }
}
```

- 返回类型能够向上转型称为父类的返回类型。
- 异常也要能向上转型成为父类的异常。unchecked 异常（空指针异常这些）不需要显式的向上抛，但是 checked 异常只能抛出异常或者此异常的子类。
- 方法名和参数类型以及个数必须严格一致。所以建议添加一个 @Override 注解，编译器会自动查覆写方法签名是否一致。防止出现明明是要覆写该方法变成新的方法。@Override 还可以避免控权限修饰符可见范围引发的问题，比如 Father 类中的 A 方法，是无权限修饰符，Son 类继承 Father 类但是不在一个包下，Son 类直接覆写 A 方法，若是没有加上 @Override 注解，可能会被编译器认是一个新的方法。

总结成口诀：「一大两小两同」。

- 一大：之类的方法访问权限控制符只能相同或者变大。

● 两小：抛出的异常和返回值只能变小，可以转型成父类对象。子类的返回值、抛出异常类型必须与类的返回值、抛出异常类型存在继承关系。

● 两同：方法名和参数必须相同。

另外注意：子类和父类不要相互调用彼此的方法，不然会变成循环调用，最后直至 JVM 崩溃，产生 SackOverflowError 异常。

重载

在同一个类中，如果多个方法有相同的方法名、不同的参数，即称为重载，比如一个 POJO 类中多个造方法。以及 String 类中的 valueOf，它有 9 个方法，可以将输入的基本数据类型、数组、Object 转化成为字符串。

我们再回顾下方法签名的概念：方法名称 + 参数类型 + 参数个数，组成一个<font color=red>**唯一键**/font>，成为方法签名，这个唯一键是 ⑧能重复的。JVM 就是通过这个唯一键决定调用那种重载的法。

以下代码就是错误的重载方式，他们都有共同的特征，就是方法签名重复冲突了。

```
public class EasyCoding {

    public void methodForOverload() {}

    // 编译出错，返回值并不是方法签名的一部分。
    public final int methodForOverload() {}

    // 编译出错，访问权限控制符也不是方法签名的一部分。
    private void methodForOverload() {}

    // 编译出错，静态标识符而不是方法签名的一部分。
    public static void methodForOverload() {}

    // 编译出错，final标识符一样也⑧是方法签名的一部分。
    public final void methodForOverload() {}
}
```

那么下面这几种情况，编译器是如何判断正确调用的呢，比如下面几种重载方法，第一处和第二处的果是什么呢：

```
public class EasyCoding {

    public void methodForOverload(int arg) {
        System.out.println("int");
    }

    public void methodForOverload(Integer arg) {
        System.out.println("Integer");
    }

    public void methodForOverload(Integer... args) {
        System.out.println("Integer...");
    }

    public void methodForOverload(Object arg) {
```

```
        System.out.println("Object");
    }

    public static void main(String[] args){
        EasyCoding ec = new EasyCoding();
        // 第一处
        ec.methodForOverload(996);
        // 第二处
        ec.methodForOverload();
    }
}
```

第一处答案是int，第二处的答案是Integer...
那么为什么 ec.methodForOverload(996) 方法编译器会匹配到 methodForOverload(int arg) 呢，
 ec.methodForOverload() 无参方法缺席的情况下却匹配到了可变参数 methodForOverload(Integer
.. args)?

可变参数的个数其实是从 0 个到多个，所以首先它也会和其他方法抢夺匹配 ec.methodForOverload
996)，然而他的优先级是最低的，弟中弟嗷。但是在无参方法缺席的情况，只有他符合这一条件，所
 ec.methodForOverload() 无参方法自然匹配上。而 int 和 Integer 还有 Object 的较量中胜出的原
是不需要自动装箱，假如把 int 类型改成 long 类型，编译器一定也是 Match long 类型，而优于装
的 Interger。

但是如果是 methodForOverload(null) 这种情况则会调用参数为 Integer 的方法，Null 可以匹配任
类对象，从最底层一次向上查找，会找到 Integer 和 Integer... 这两个参数方法，但是会报错，因为
们同时 Match 上了。

JVM 在重载方法的顺序如下：

1. 精确匹配。
2. 如果是基本数据类型，自动转换成更大表示范围的基本类型。
3. 通过自动拆箱和装箱。
4. 通过子类向上转型继承路线依次匹配。
5. 通过可变参数匹配。

我们程序猿心情不好的时候也会把气撒在编译器上，有时候，我们会不断挑战编译器的下限，比如这
：

```
public class EasyCoding {

    public void methodForOverload(int arg, Integer arg2) {
        System.out.println("int arg, Integer arg2");
    }

    public void methodForOverload(Integer arg, int arg2) {
        System.out.println("Integer arg, int arg2");
    }

    public void methodForOverload(int... args) {
        System.out.println("int...");
    }

    public void methodForOverload(Integer... args) {
        System.out.println("Integer...");
```

```
    }

    public static void main(String[] args){
        EasyCoding ec = new EasyCoding();
        // 第一处
        ec.methodForOverload(965,996);
    }
}
```

此种方式就是完全在挑战编译器的底线了，虽然编译会通过，但是调用的时候一定会报错。

泛型

泛型也许是很多人相对陌生的领域，但是它⑧是多么神秘，本质是类型参数化，解决不确定具体对象型的问题。Java 在引入泛型之前，表示可变类型往往存在类型安全的风险，举个例子，微波炉最主要功能是加热食物，而食物也有几十几百种可能，所以一般会像下面这样写业务设计：

```
public class EasyCoding {

    public static Object heat(Object food) {
        System.out.println(food + "isOK");
        return food;
    }
    public static void main(String[] args) {
        Meat m = new Meat();
        m = (Meat) EasyCoding.heat(m);
        Soup s = new Soup();
        s = (Soup) EasyCoding.heat(s);
    }
}
```

这里的 heat 方法就是为了避免给每个类型的食物定义一个加热方法，但是只能采用“向上转型”的式才能具备加热任意类型食物的能力。但是会让客户端困惑，因为对加热的内容不能正确区分，在取时进行强制类型转换就会存在类型转换风险。而泛型就是为了解决这个而生。

泛型使用

泛型可以定义在类、接口、方法中，编译器通过识别尖括号和尖括号内的字母来解析泛型。现在一般定俗成的符号有：

- E：代表 Element，用于集合中的元素。
- T：代表 the type of object，表示某个类。
- K：代表 Key。

● V：代表 Value，K 和 V 用于键值对元素。

下面这段代码可以很好地说明泛型定义的概念：

```
public class EasyCoding<T> {

    static <String, T, Object> String get(String arg1, Object arg2) {
        System.out.println(arg1.getClass());
        System.out.println(arg2.getClass());
        return arg1;
    }

    public static void main(String[] args) {
        Integer arg1 = 996;
        Long arg2 = 965L;
        Integer result1 = get(arg1,arg2);

        byte[] b1 = new byte[666];
        byte[] b2 = new byte[666];
        byte[] result2 = get(b1,b2);

    }
}
```

首先这段代码是完全可以通过编译的，可能没用过泛型的小伙伴会疑问为什么 get() 方法可以传入 Integer 和 Long 甚至是 byte[] 类型？而且返回的结果不应该是 String 类型吗？其实关键就在于 <String, T, Object> 这个泛型标识，String 是我们常见的所熟知的包装类了，Object 是所有类的父类，但是泛型标识里，它就不是 String 和 Object 了，而是可以成为任意类型，属于完全未知的类型，入参的一个参数如果是 Integer 类型，那么在方法体内的所有 arg1 就不是我们认知里的 java.lang.String，这个 String 就是相当于之前说明的 T，Object 也是一种 T，仅仅只是一个代号。

当然我们平时编码不会也不要这样去定义泛型，确实会容易引发歧义和造成其他问题。所以我应该注以下几点：

1. 尖括号里的每个元素都指代一种位置类型。 <String> 这里的 String 就不是我们认知上的 java.lang.String 了，仅仅只是个代号。包括类名后的 <T> 和 get 方法前的 <T> 是两个指代，互不影响。

2. 尖括号的位置非常讲究，必须在类名之后或方法返回值之前。

3. 泛型在定义处只具备执行 Object 方法的能力。所以 arg1 和 arg2 只能调用 Object 类中的方法比如 toString()。

4. 对于编码之后的字节码指令，其实没有这些花头花脑的方法签名，充分说明了泛型只是一种编码时语法检查。

所以之前微波炉加热食物的例子可以用泛型这样改写：

```
public class EasyCoding {

    public static <T> T heat(T food) {
        System.out.println(food.getClass());
        System.out.println(food + "isOK");
        return food;
    }

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

```
        heat(new Meat());
        heat(new Soup());
    }
}
```

避免使用 Object 作为输入和输出可以控制强制转换带来的风险。因为依据墨菲定律，只要这种风险在，就一定会发生 ClassCastException 异常。

数据类型

基本数据类型

基本数据类型 9 种：int、short、long、float、double、char、boolean、byte、refvar，其中 refvr 是句柄，是面向对象中的引用变量，默认值为 Null。详细如下表格：

类型名称	默认值	大小	最小值	最大值	装类	缓存区域
boolean	false	1B	0(flase)	1(true)	oolean	无
byte	(byte)0	1B	-128	127	By e	-128~127
char	'\u0000'	2B	'\u0000'	'\uFFFF'	haracter	(char)0~(char)127
short	(short)0	2B	-2^15	2^15-1(32767)	hort	-128~127
int	0	4B	-2^31	2^31-1	Integ r	-128~127
long	0L	8B	-2^63	2^63-1	L ng	-128~127
float	0.0f	4B	1.4e-45	3.4e+38	loat	无
double	0.0d	8B	4.9e-324	1.798e+308	ouble	无

注意：其中缓存区间是个有意思的东西，估计很多人没有深究过，大厂面试很有能就来这么一道题。

对象分为三块存储区域：

- 对象头。对象头占用 12B，其中包括：哈希码、GC 标记、GC 次数、同步锁标记、偏向锁持有者反正 older_man⑧是很懂）。
- 实例数据。存储本类对象的实例成员变量和所有可见的父类成员变量。
- 对齐填充。

包装类型

包装类的存在是为了解决了基本数据类型无法做到的事情：泛型类型参数、序列化、类型转换、高频

间数据缓存。尤其是最后一个，因为除了 Float 和 Double 之外，其他包装类型都会缓存。拿 Integer 举例，缓存区间在-127~128，所以在这个区间的赋值，Integer 对象会由 IntegerCache.cache 产生就不会复用已有对象。**因此，推荐所有包装类对象之间的比较，全都使用 equals()方法。**

源码如下：

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

各个包装类的缓存区间：

- Boolean：使用静态 final 变量定义，valueOf()就是返回这两个静态值。
- Byte：表示范围是-128~127，全部缓存。
- Short：表示范围是-3276832~~767，缓存范围是-128~~127。
- Charater：表示范围是 065~~535，缓存范围是 0~~127。
- Long：表示范围是[-2^63,2^63-1]，缓存范围是-128~127。
- Integer：表示范围 hi[-2^31,2^31-1]。

接下来我们看看如果不使用 equals()方法去进行包装类的比较会出现什么情况

```
public static void main(String[] args) {
    Integer i1 = 127;
    Integer i2 = 127;
    Integer i3 = 128;
    Integer i4 = 128;

    Long l1 = 127L;
    Long l2 = 127L;
    Long l3 = 128L;
    Long l4 = 128L;

    // 第一处
    System.out.println(i1 == i2);
    System.out.println(i3 == i4);
    // 第二处
    System.out.println(i1.equals(i2));
    System.out.println(i3.equals(i4));

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

    // 第三处
    System.out.println(l1 == l2);
    System.out.println(l3 == l4);
    // 第四处
    System.out.println(l1.equals(l2));
    System.out.println(l3.equals(l4));
  }
```

以上代码四处打印的结果是不是让人觉得都是 true?

但是答案是第一处打印的结果是 true 和 flase，第二处全是 true，第三处也是 true 和 flase，第四处是 true。

那么为什么 System.out.println(i1 == i2); 的结果是 true，而 System.out.println(i3 == i4); 是 flase 呢?

该例很好地说明了 Integer 和 Long 只是缓存了-128~127 之间的值，而大于或者小于区间的值没有缓存，i3 和 i4 是 128，刚好超出了这个区间，下面的 l3 和 l4 同理。

当然我们也可以修改包装类的缓存范围，在 VM options 加入参数 -XX:AutoBoxCacheMax=7777 即可设置最大缓存值为 7777，那么以上代码的打印结果全为 true。

在选择使用包装类和基本类型的时候，也不能完全按照心情，我们可以从以下几点来看：

1. 所有的 POJO 类属性必须使用包装数据类型。
2. RPC 方法的返回值和参数必须使用包装数据类型。
3. 所有的局部变量推荐使用基本数据类型。

字符串

字符串是从堆上分配而来，算是基本数据类型的小弟。主要是三种：String、StringBuilder、StringBffer。

1. String 是只读字符串，典型的 immutable 对象，对它的任何改动，其实都是创建一个新对象，再引用指向该对象。String 对象赋值操作后，会在常量池中进行缓存，下次申请创建对象时，缓存中已存在，则直接返回相应引用给调用者。
2. StringBuffer 可以在原对象上进行修改，是线程安全的。
3. StringBuilder 是非线程安全的，把多线程的锁的处理交给工程师（也就是宁 lder_man）来处理，所以操作效率比较高。

线程安全的对象的产生一般是因为计算机的发展总是从单线程到多线程，从单机到分布式。

字符串的连接方式在循环体内非常不推荐使用 String 类型相加，而是应该使用 StringBuilder 的 append 方法。