

# java成神之路读书笔记

借鉴地址Gitee Pages 完整阅读:<http://hollischuang.gitee.io/tobetopjavaer>

作者: Hollis , 阿里巴巴技术专家, 51CTO 专栏作家, CSDN 博客专家, 掘金优秀作者, 《程序员的三门课》联合作者, 《Java 工程师成神之路》系列文章作者;热衷于分享计算机编程相关技术, 博文全网阅读量数千万。

## 面向对象

java是一种面向对象的编程语言

### 面向过程

什么是面向过程?

面向过程(Procedure Oriented)是一种以过程为中心的编程思想, 是一种自顶而下的编程模式。简单来说, 面向过程的开发范式中, 程序员需要把问题分解成一个一个步骤, 每个步骤用函数实现, 依次调用即可。

面向过程的编程语言以诸多流程控制语句来实现一个功能, 整体表现为流程化。

#### 优缺点

优点

流程化, 执行效率高

缺点

维护困难, 复用性差

## 面向对象

面向对象 (Object Oriented) , java是一种面向对象的编程语言。在面向对象的开发过程中, 回将某件事情进行抽象, 将一件事物的方法属性封装到一个类中, 通过多个类之间的组合调用来实现某种功能。

### 面向对象三大基本特征

封装、继承、多态

#### 封装

如果一个类希望其他类访问其内部属性存在不同限制, 那么我们可以将其方法和属性设置不同的访问权限, 这就是封装。

访问级别有以下几种

- public 所有类都可以访问
- protected 受保护的, 默认访问级别, 同级别包下的类可以访问

- private 私有的，任何其他类都不可以访问，只供其内部访问

一般来说如果不是清楚的知道一个类的属性或方法需要被其他类访问，我们会将其设置为私有属性，不对外暴露。

## 继承

继承是java为我们提供的可以实现代码复用的一种能力。可以拥有现有类的所有属性和功能（包括私有属性和私有方法），并且可以在此基础上进行扩展。

## 多态

java中的多态指的是同一种操作，作用于不同的实例可以有不同的结果。是一种运行时状态，只有在运行期间才会直到调用的具体方法是什么。

具体表现形式为父类或接口的引用指向子类或实现类的实例。调用父类或接口中定义或声明的方法，会根据传入的不同的子类或实现类来表现不同的逻辑。

多态机制使具有不同内部结构的对象可以共享相同的外部接口。

### 编译期&运行期

编译期指的是，将源代码编译成另一个中间语言，在此期间会做一些代码规范检查，以及编译期间代码优化。

运行期，指的是程序运行在内存中，进行交互。

### 编译期间多态

在编译期间已经明确知道，具体类型，知道调用什么方法。

比如说方法重载、可以通过参数列表的不同确定调用的具体方法。

### 运行期多态

指的是在运行期间才会确认具体类型，才会知道调用的方法，需要`extends``implement`关键字一层一层去找。

比如说使用父类或接口的引用，指向子类或实现类的实例。

```
public class Demo {
    public static void main(String[] args) throws ClassNotFoundException,
        InstantiationException, IllegalAccessException {
        Class<?> sonClass =
        Class.forName("com.roily.booknode.javatogod._01faceobj.extendsiscompile.Son");
        Person son = (Person) sonClass.newInstance();
        son.method1();
        Class<?> daughterClass =
        Class.forName("com.roily.booknode.javatogod._01faceobj.extendsiscompile.Daughter");
        Person daughter = (Person) daughterClass.newInstance();
    }
}
```

```

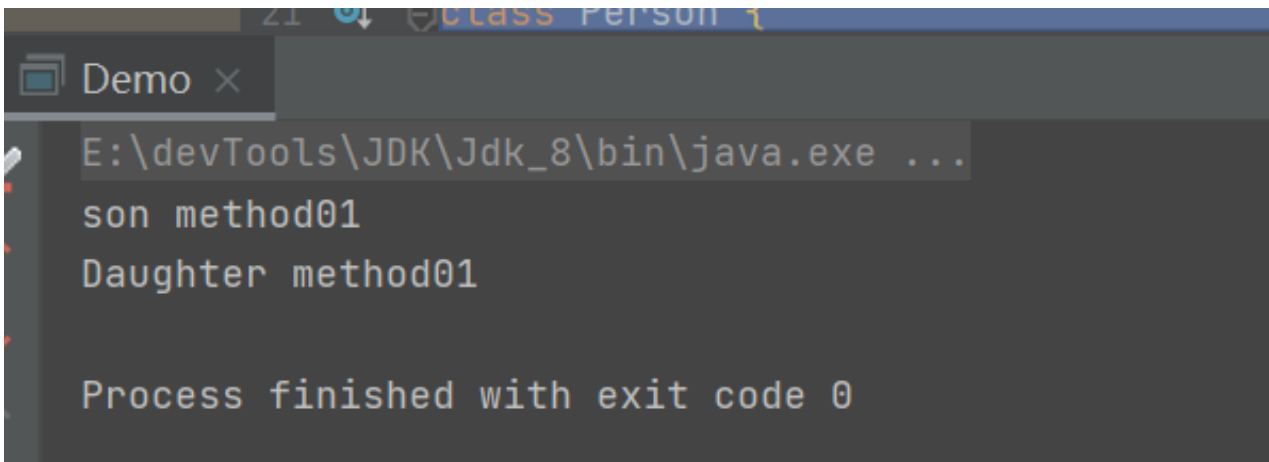
        daughter.method1();
    }
}

class Person {
    void method1() {
        System.out.println("method1");
    }
}

class Son extends Person {
    @Override
    void method1() {
        System.out.println("son method01");
    }
}

class Daughter extends Person {
    @Override
    void method1() {
        System.out.println("Daughter method01");
    }
}

```



```

E:\devTools\JDK\Jdk_8\bin\java.exe ...
son method01
Daughter method01

Process finished with exit code 0

```

## 重写和重载

重写（Overriding）和重载（Overloading）是两个比较重要的概念。

### 重载

指的是在同一个类中，多个方法的方法名称相同而方法签名不同的现象称为重载，这些方法互称为重载方法。

方法签名：方法名+参数列表。（也就是方法名相同，参数列表不同才会构成重载）

返回类型不同不会构成重载。

- 方法名相同，参数列表不同

- 可以改变返回类型
- 可以修改访问修饰符
- 可以声明新的检查异常
- 重载可以发生在一个类中，或在子类和父类中

## 重写

严格意义上指的是子类中定义了和父类相同方法签名，且符合重写要求的方法，那么称子类重写了父类的方法。

接口声明抽象方法，其实现类实现抽象方法，对应方法上可以加上@Override注解，也可以称为重写，更多的称为实现。

```
public class OverWriting {
    public static void main(String[] args) {
        final Animal dog = new Dog();
        dog.bark();
    }
}
class Animal {
    void bark() {
        System.out.println("动物叫");
    }
}
class Dog extends Animal {
    @Override
    void bark() {
        System.out.println("狗叫");
    }
}
```

输出：狗叫

这里子类实例指向父类引用，是多态的表现形式，编译期间会去检查父类中是否存在对应调用方法。而运行期间具体需要调用哪个方法，需要根据具体指向的实例来决定

方法重写的条件需要具备以下条件和要求：

两同两小一大

- 两同（方法签名相同）
  - 方法名相同
  - 参数列表相同
- 两小
  - 返回类型的范围需要相等或更小（比如父类返回ArrayList子类就不能返回list）
  - 抛出的检查异常范围要比父类被重写方法要小
- 一大

- 访问级别限制，比被重写方法访问范围要大(即父类是protected的那么子类重写的方法不能申明为private)

#### 其他

- 不能重写被final标识的方法
- 重写的前提是继承

```
class Person {
    void method1(int a, int b) {
        System.out.println("XX");
    }
    ArrayList<Integer> method2() {
        return null;
    }
}
class Student extends Person {
    /**
     * 两同
     * - 方法名和参数列表相同
     */
    @Override
    void method1(int a, int b) {
        System.out.println("XX");
    }
    /**
     * 两小
     * - 返回参数要比被重写方法要小（范围）
     */
    // @Override
    // List<Integer> method2() {
    //     //通过不了编译
    //     return null;
    // }
}
```

子类的返回范围比父类的大，通过不了编译，反过来就行

## 继承&实现

继承的关键字 `extends`，实现关键字 `implements`。

## 继承

通过继承可以拥有父类的所有属性和方法，实现代码的重用。继承可以发生在类与类之间，这个类可以是具体的也可以是抽象的，同时继承也可以发生在接口与接口之间。

如果说可以从某个类中抽出来可以供于公共使用的功能，那么就可以抽出一个父类出来，其他类去继承这个父类，以继承的方式来实现对代码的重用。但前提是这个抽出来的这个父类得保持稳定，也就是少量修改，且这个父类得对其他类都得适用。

一般来说不会使用继承来实现重用，特别是继承至具体的类，如果说非得继承可以继承至抽象类。

## 实现

实现发生在类与接口或抽象类之间，如果说一组业务的处理方式是一样的那么就可以制定抽象（制定标准），具体业务去实现定义的抽象

```
/**
 * 可以实现一个接口
 */
interface IPerson{
    /**
     * 抽象方法
     */
    void method();
}

class Teacher implements IPerson{

    @Override
    public void method() {

    }
}

/**
 * 可以是类实现抽象类的抽象方法
 */
abstract class AbstractPerson{

    abstract void method();

}

class StudentImpl extends AbstractPerson{

    @Override
    void method() {

    }
}
```

```
/**
 * 可以是抽象类实现接口
 */
abstract class AbstractPersonX implements IPerson{
    @Override
    public void method() {

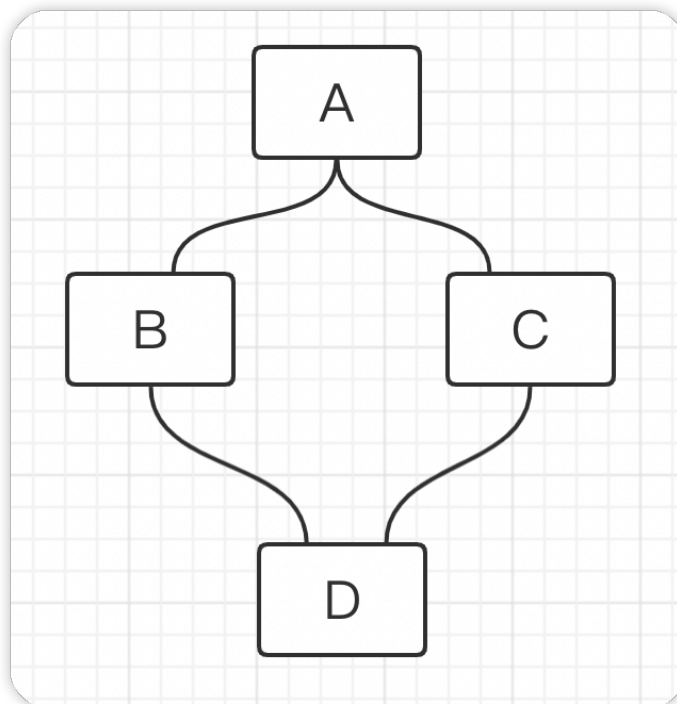
    }
}
```

## java单继承

java 通过 `extends` 关键字实现继承，且不支持多继承。

### 为什么

菱形问题：假设B和C都继承自A，B和C都继承了父类A的所有属性和方法，如果java支持多继承的话，此刻有一个D继承自B和C，那么类D就同时拥有类B和类C的所有属性和方法，并且类D继承了两份来自于A的属性和方法，拥有同名属性和相同方法签名的方法是通过不了编译的，且如果通过编译，在调用的时候也会产生歧义。



### java可以多实现

java不支持多继承但是支持多实现

如下例子，我们在InterfaceA和InterfaceB中定义了两个同名的方法，然后使用ClassC实现它们，发现实现类对于相同方法只实现了一次。

```

/**
 * 但是java可以多实现, 且java8之后接口中可以定义default方法
 */
interface InterfaceA {
    void method1();
}
interface InterfaceB {
    void method1();

    void method2();
}
class ClassC implements InterfaceA, InterfaceB {
    @Override
    public void method1() {
        System.out.println("method1");
    }
    @Override
    public void method2() {
        System.out.println("method2");
    }
}

```

对于接口而言它只是一个标准、抽象, 实现类按照约定实现标准。方然也可以指定标准, 使用某个接口的引用指向实现类的实例。

```

@Test
public void test1() {
    InterfaceB classC1 = new ClassC();
    InterfaceA classC2 = new ClassC();
}

```

接口中可以定义default方法, 且我们可以使用Implement从多个接口中继承得到多个默认方法, 特别的如果说两个接口存在相同方法签名的方法, 实现类会被要求强制重写同名方法签名的方法来解决菱形问题。

```

interface InterfaceC {
    default void method1() {
        System.out.println("InterfaceC default1方法");
    }
}
interface InterfaceD {
    default void method1() {
        System.out.println("InterfaceD default1方法");
    }
}
/**
 * 可以使用implement从多个接口中得到多个default方法,
 * 如果存在菱形问题, 会强制要求实现类重写同名方法
 */

```



```
class ClassD implements InterfaceC, InterfaceD {  
    @Override  
    public void method1() {  
        InterfaceC.super.method1();  
    }  
}
```

## 五大基本原则

面向对象五大基本原则，指导程序员编码，符合五大基本原则的程序，健壮性、可维护性和可扩展性都大大提高。

五大基本原则，都旨在：

- 高内聚、低耦合
- 面向抽象、接口，而不是面向具体、实现

### 单一职责

适用于方法、接口和类。一个类的职责尽量单一，只有一个引起它的变化。

对于方法而言，我们一般都遵守其单一职责原则

对于接口而言，抽象方法尽量要求要少，如果方法太多可以进行接口拆分

对于类而言，一般来说都不会严格遵守单一职责，比如说有一个类UserService，进行堆用户的增删改查，那么这个类想要严格遵守单一职责，完全可以拆分为四个类。

所以说，单一职责尽量遵守，类、接口、方法不要过于臃肿。在业务要求基础之上，合理遵守。

### 开闭原则

对扩展开放、对修改关闭。

- 1、对扩展开放，意味着有新的需求时，可以在现有代码上进行扩展，以适应新的变化。
- 2、对修改关闭，当软件或系统一旦设计完成，可以独立完成工作，而不要其进行任何修改的尝试

开闭原则的重点在于，面向接口、抽象编程而不是面向实现、具体编程。

因为抽象也就是接口相对稳定，接口定义了一套标准，如果说接口添加新的抽象方法，那么就必须修改其实现类，所以说对于修改是关闭的。

而如果说现在有了一个新的需求，可以通过实现现有接口定义一个新的类，配合多态可以对当前系统功能进行扩展，所以说对扩展开放。

而不能面向具体，一般来说会以继承或组合的方式实现具体类的复用。具体并没有一套标准，也就是说父类修改对应逻辑，并不会要求子类修改，也就是对修改开放，违背开闭原则。

## 里氏替换原则

里氏替换原则知道我们如何使用继承，是一种编程思想。

要求软件实体：子类必须能够替换其基类，并且不改变业务逻辑。

### 里氏替换原则和继承的关系

- 继承

继承是java提供了一种可以实现代码复用的语法，但是继承是侵入式的，如果在继承的过程中子类重写了父类的方法，那么说明父类的方法并不通用。

- 里氏替换原则

里氏替换原则是一种编程思想，用于在指导我们合理使用继承。只有遵守了里氏替换原则，才可以实现继承复用。

## 依赖倒置原则

面向接口编程，依赖于抽象。

具体定义：高层模块不依赖于底层模块，二者都依赖于抽象；抽象不依赖于具体，具体依赖于抽象。

## 接口隔离原则

将臃肿的大接口拆分为一个个小接口，可以使用接口继承或实现多个接口的方式来实现多个接口定义的抽象方法。

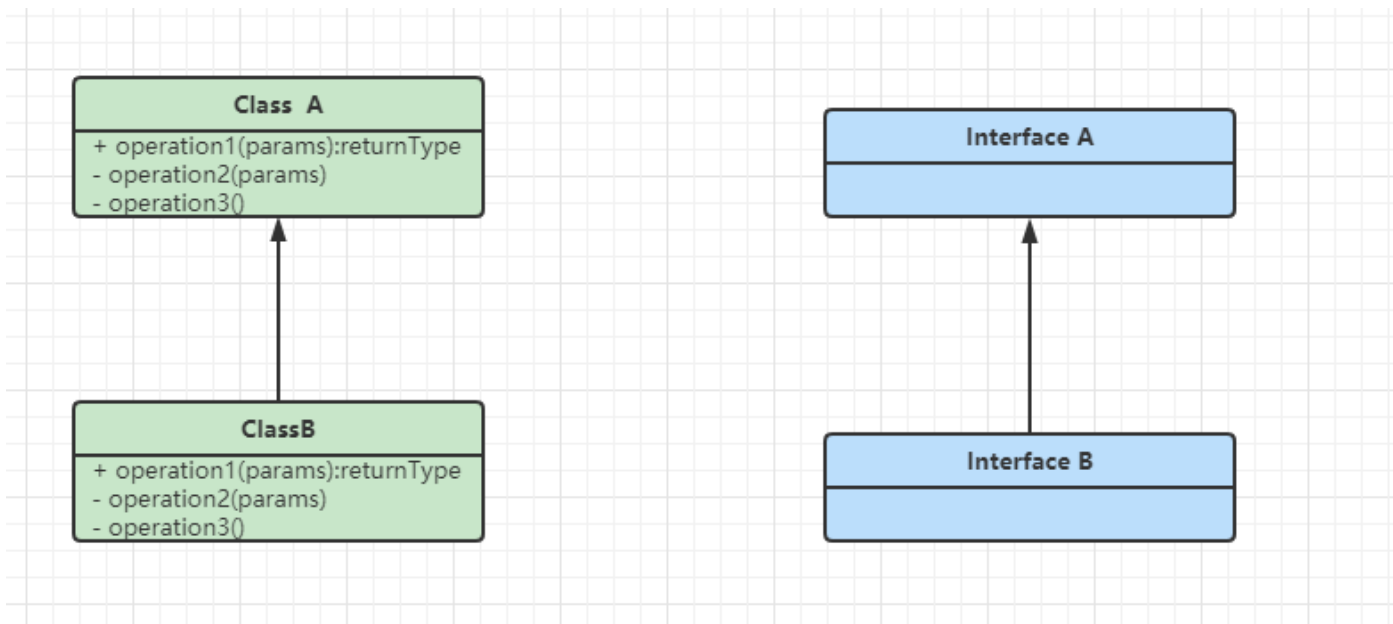
如果一个接口过于庞大，或存在一些不必要实现的方法时，这是一种接口污染。

## 继承和组合

继承和组合都是java用于实现代码复用的技术之二。优先考虑组合，尽量避免使用继承。

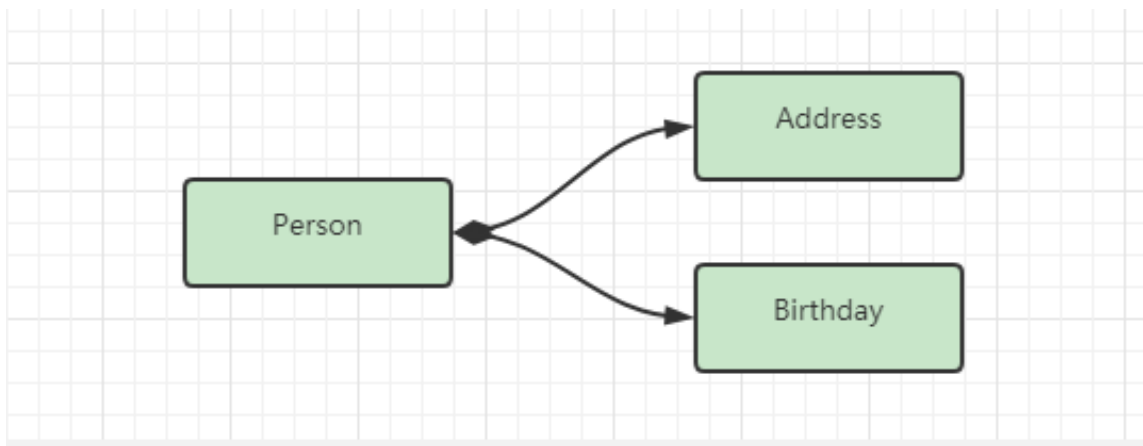
### 继承

继承(Inheritance)是一种联结类与类的层次模型。指的是一个类(子类、子接口)继承另外一个类(父类、父接口)的功能。并且可以增加自己新的功能的能力。继承是一种 **AS-a** 的关系。



## 组合

组合(Composition)体现的是整体与部分、拥有的关系，即 `has-a` 关系。



## 组合&继承关系

继承是一种侵入式的代码结构，在继承关系中，父类的内部细节对于子类是可见的（继承是一种白盒式代码复用），如果父类的代码逻辑发生改变，那么如果子类调用了父类的类方法，子类的逻辑也会随之修改，甚至出错。（继承是一种编译期概念，在编译器就确认了类与类的关系）

组合是将现有对象进行拼装组合来实现较复杂的业务逻辑，将对象作为内部的一个属性，组合进来的对象其内部实现细节是不可见的（黑盒式代码复用）。（如果说组合进来的是一个接口或抽象类型，那么在编译期无法确认其具体类型，只有在运行期间才会确认，所以一定程度上复用性更高）

## 对比

组合	继承
不会破坏封装，整体类与局部类之间松耦合	破坏封装，子类依赖于父类，表现为父类与父类之间前耦合
组合进来的可以是一个抽象，因此具有一定扩展性	可通过定义新方法
整体类对组合进来的类进行包装，并可以做一些拓展，就是装饰器模式和代理模式	
创建整体对象的时候，必须创建局部对象	创建子类对象，会自动创建父类对象

如何选择

- 多用组合、少用继承。
- 在同等情况下优先选择组合，利于扩展
  - 继承也有用处，如果当前类必须要向基类进行向上转型，则可以考虑使用继承

构造函数和默认构造函数

- 构造函数配合 `new` 关键字，用于创建实例对象和给成员变量赋值。
- 构造函数长什么样子？
- 构造函数和普通方法很相似，①构造函数名为类名②构造函数不声明返回类型，返回当前类对象
- 构造函数的重载
- 构造函数根据参数列表的不同可以实现重载。并且可以为特殊属性给与默认值。

```
class Person{
    int age;
    String name;
    String address;
    Boolean sex;

    private Person(int age, String name, String address, Boolean sex){
        this.age = age;
        this.name = name;
        this.address = address;
        this.sex = sex;
    }

    public Person(int age, String name, String address){
        return Person(age,name,address,false);
    }
}
```

如果当前类没有构造函数，编译器会自动生成一个无参构造函数。其成员变量会被赋予默认值。

## 如果没有构造函数

会生成默认构造函数

```
public class ConstructorTest {  
    int i;  
    String str;  
}
```

反编译后：

```
public class ConstructorTest{  
    public ConstructorTest(){  
    }  
    int i;  
    String str;  
}
```

## 如果存在构造函数

会使用定义的构造函数，此刻空参构造函数不可用。

```
class ConstructorTest2 {  
    int i;  
    String str;  
    public ConstructorTest2(int i) {  
        this.i = i;  
    }  
}
```

反编译后：

```
class ConstructorTest2{  
    public ConstructorTest2(int i){  
        this.i = i;  
    }  
    int i;  
    String str;  
}
```

## 类变量、成员变量和局部变量

java 中如果从，生命周期，作用域和内存角度看，java 的变量分为，类变量、成员变量和局部变量。

### 类变量

类变量被 `static` 修饰，属于类，生命周期等同于类的生命周期，当一个类被类加载器成功加载到方法区，其就已经存在与方法区。当类被卸载的时候也跟着消失。

### 成员变量

成员变量属于实例，生命周期等同于实例，当一个实例被new出来(或反射等其他方式)，会为其赋值，跟随实例存在于堆内存中。当实例对象被回收时，他也跟着消失。

### 局部变量

局部变量存在于栈内存，一般存在于方法参数，循环体或方法中。

```
public class Demo2 {  
    //类变量  
    final static String str1 = "abc";  
    static String str2 = "abc";  
  
    //成员变量  
    String str3 = "abc";  
  
    //局部变量  
    void method01(String str1) {  
        String str2 = "abc";  
        for (StringBuilder str = new StringBuilder("abc");  
str.toString().equals("abc"); ) {  
            }  
        }  
    }  
}
```

## 访问修饰符

- public

公开的，被public修饰的成员变量和方法对所有类都是可见的，所有类和对象都可以直接访问

- private

私有的，被private修饰的成员变量和方法是私有的，只有当前类有访问权限。即便是子类也不可以访问

- protected

受保护的，被protected修饰的成员变量和方法是受保护的，只有当前类和与其处于同一包下的类有访问权限。除非是子类

- default

默认的，被default修饰的成员变量和方法是受保护的，只有当前类和与其处于同一包下的类有访问权限。即便是子类

```
package com.roily.booknode.javatogod._01faceobj.extendsiscompile;

public class Demo3 {
    public String str1;
    String str2;
    private String str3;
    protected String str4;
}
```

子类不在同一个包下：

```
package com.roily.booknode.javatogod._01faceobj;

import com.roily.booknode.javatogod._01faceobj.extendsiscompile.Demo3;
public class TestDemo extends Demo3{
    void method1(){
        System.out.println(str1);
        System.out.println(str4);
    }
}
```

可见如果不指定属性和方法的访问级别的话，默认为default。

## java的值传递

java 关于参数的传递只有值传递，在传递参数的时候会将参数进行拷贝，在方法体中操作的都是拷贝的参数。

### 形参、实参

形参：在定义方法的时候使用的参数，[参数类型+形参名称]，目的是为了接收参数

实参：在调用方法的时候，被调方法会被传入一个参数 [参数名]，这个参数就叫实参

```
/**
 * @param str 形参
 */
void method(String str) {
}

void method2() {
    /**
     * 123 实参
     * str 实参
     */
    method("123");
}
```

```
String str = "123";
method(str);
}
```

## 为什么说java只有值传递

对于基本数据类型来说，它只有值的概念，所以对于基本数据类型的值传递没有任何异议。

对于引用数据类型来说，在对引用类型的参数进行传递的时候，会将参数进行拷贝，在方法体内实际操作的是拷贝的副本，如果我们没有改变引用关系而直接操作属性，是会对原对象有影响的，应为两个引用指向的是同一个对象。

例：

```
/**
 * 基本数据类型，只有值的概念
 *
 * @param i
 */
void simpType(int i) {
    i = 999;
}

void referenceType1(StringBuilder sb) {
    sb.append("追加");
}

void referenceType2(StringBuilder sb) {
    sb = new StringBuilder();
    sb.append("追加");
}

@Test
public void test1() {
    System.out.println("基本数据类型");
    int i = 1;
    System.out.println("原值: " + i);
    simpType(i);
    System.out.println("修改后: " + i);

    System.out.println("引用数据类型，未修改引用");
    final StringBuilder sb1 = new StringBuilder("123");
    System.out.println("原值: " + sb1.toString());
    referenceType1(sb1);
    System.out.println("修改后: " + sb1.toString());

    System.out.println("引用数据类型，修改引用");
    final StringBuilder sb2 = new StringBuilder("123");
```



```
System.out.println("原值: " + sb2.toString());
referenceType2(sb2);
System.out.println("修改后: " + sb2.toString());
}
```

✓ Tests passed: 1 of 1 test – 1 ms

/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home/bin

基本数据类型

原值: 1

修改后: 1

引用数据类型, 未修改引用

原值: 123

修改后: 123追加

引用数据类型, 修改引用

原值: 123|

修改后: 123

Process finished with exit code 0

- 对于基本数据类型来说, 值传递没有异议

原始参数通过值传递给方法。这意味着对参数值的任何更改都只存在于方法的范围内。当方法返回时, 参数将消失, 对它们的任何更改都将丢失

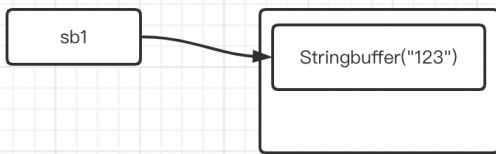
- 对于引用数据类型来说

也就是说, 引用数据类型参数(如对象)也按值传递给方法。这意味着, 当方法返回时, 传入的引用仍然引用与以前相同的对象。但是, 如果对象字段具有适当的访问级别, 则可以在方法中更改这些字段的值

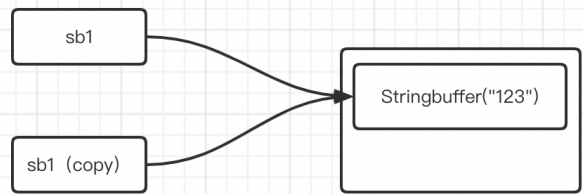
引用类型传递的时候发生了什么?

void referenceType1(StringBuilder sb)方法

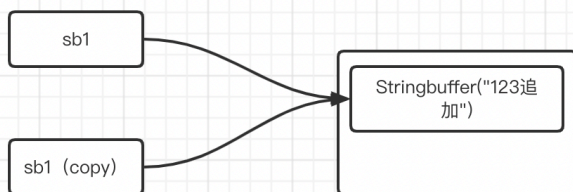
```
final StringBuilder sb1 = new StringBuilder("123");
```



```
referenceType1(sb1);
```

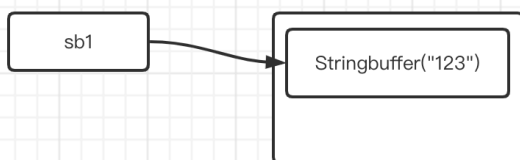


```
sb.append("追加");
```

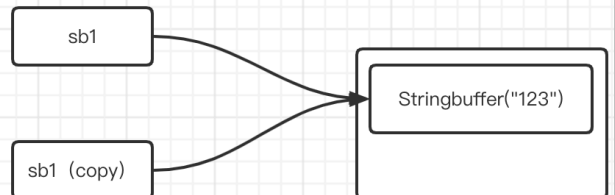


void referenceType2(StringBuilder sb)方法

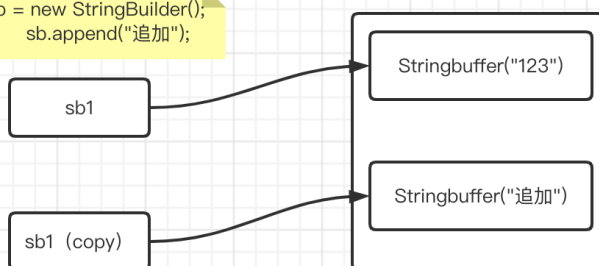
```
final StringBuilder sb1 = new StringBuilder("123");
```



```
referenceType1(sb1);
```



```
sb = new StringBuilder();  
sb.append("追加");
```



# java基础

## 8大基本数据类型

Java中有8种基本数据类型分为三大类。

- 字符型

char

- 布尔型

boolean

- 数值型

1.整型: byte、short、int、long

2.浮点型: float、double

### 取值范围

在java中整数类型属于有符号类型，第一位用来表示符号0代表整数1代表负数。

比如说byte类型，占1字节8位，那么他的表示范围为：

最大值：0111 1111 ( $2^7-1$ )

最小值：1000 0000 ( $-2^7$ )

这里会有一个疑问？ 1000 0000 为什么 表示 $-2^7$ 呢？不是 -0么？

第一点：在计算机中，数据的运算是以补码进行的[源码反码补码](#)

第二点：为了防止 +0 和 -0的出现，约定了 补码 1000 0000代表 -128 且移除 -0概念

以一个找规律的方式解释：

原码	反码	补码	值（10进制）
0111 1111	0111 1111	0111 1111	127
0000 0000	0000 0000	0000 0000	0
.....	.....	.....	.....
1000 0001	1111 1110	1111 1111	-1
1000 0010	1111 1101	1111 1110	-2
1000 0011	1111 1100	1111 1101	-3
		补码不断减一	
1111 1111	1000 0000	1000 0001	-127
无法表示	无法表示	1000 0000	-128

整型

取值范围

数据类型	字节数、位数	范围
byte	1字节、8位	【-128, 127】
short	2字节、16位	【-2 <sup>15</sup> , 2 <sup>15</sup> -1】
Int	4字节、32位	【-2 <sup>31</sup> , 2 <sup>31</sup> -1】
long	8字节、64位	【-2 <sup>63</sup> , 2 <sup>63</sup> -1】

溢出问题

由于整型的存储空间是有限的，那么就会存在溢出问题

这是因为int只占32位

0111 1111

0111 1111 +

1111 1110（补）=》 1111 1101(反)=》 1000 0010(原) = -2

```

/**
 * 溢出问题
 */
@Test
public void test2() {
    final int value = Integer.MAX_VALUE + Integer.MAX_VALUE;
    System.out.println(value);
}

```

```
>> ✔ Tests passed: 1 of 1 test - 1 ms
```

```
ms /Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home
```

```
ms -2
```

```
Process finished with exit code 0
```

## 浮点数

### [定点数&浮点数](#)

java为我们提供了float和double两个浮点数数据类型，分别占4字节32位和8字节64位。

相较于float(单精度),double(双精度)其表示的范围更大，且精度更高。

#### 存储结构

float: 1位符号位，8位指数位，23位尾数位

double: 1位符号位，11位指数位，52位尾数位

浮点数存在精度问题，对于金额有严格精度要求的业务，不可使用浮点数来表示金额。

## 自动装箱与拆箱

八大基本数据类型自动装箱与自动拆箱。八大基本数据类型都有对应的对象类型，自动装箱拆箱的意思就是在需要基本数据类型需要转化为对应的包装类型的时候不需要程序员主动的去操作，而是编译器会自动帮我们去做。

除了 `int` 对应的包装类型为 `Integer`，`char` 对应包装类型为 `Character` 外其他基本数据类型对应的包装类型都为对应基本数据类型首字母大写。

`Java` 是一种面向对象的编程语言，一切皆对象，为何需要基本数据类型？

基本数据类型，相较于对象类型运算简单。

包装类型存在的意义？

基本数据类型的包装类型，不仅仅只有值的概念，其扩展了额外的方法(比如equals)。且对于集合框架来说，需要的是对象类型，我们无法将基本数据类型放进去。

## 装箱&拆箱

### 装箱

```
int i = 10;
Integer i2 = new Integer(i);
或
Integer i2 = Integer.valueOf(i);
```

### 拆箱

```
Integer i = new Integer(10);
int i2 = i.intValue();
```

## 自动装箱拆箱

基本数据类型在需要转化为对应包装类型的时候，无需程序员手动操作

```
Integer i = 10;
int i2 = i;
```

对其进行反编译可以发现确实自动帮我们转化了：

```
@Test
public void testBoxing2() {
    Integer i = Integer.valueOf(10);
    int i2 = i.intValue();
}
```

还有就是集合的泛型是一个对象类型，但是我们在编码的时候可以直接将基本数据类型放入，因为编译器会帮我们自动装箱。

```
List<Integer> ints = new ArrayList<>();
ints.add(10);
```

反编译看：

```
@Test
public void testBoxing3() {
    List<Integer> ints = new ArrayList<>();
    ints.add(Integer.valueOf(10));
}
```

## 问题

自动装箱与拆箱虽然给我们编码带来了方便，但也会有一些问题。

- 对于基本数据类型来说，我们只关心其数值，在自动装箱过后，超过缓存范围的包装类型，必须使用equals判等。不可使用 ==
- 将包装类型拆箱的过程中，可能出现空指针异常(NPE)

```
Integer methodRe(){
    return null;
}
@Test
public void testRe(){
    int i = methodRe();
}
```

## 基本数据类型的池化技术

基本数据类型（除了double、float）都有缓存技术，会缓存一定范围内的对象，原因就是jvm认为在此范围内的对象很常用，在需要使用的时候直接去池中拿取，而无需重新创建。

## 缓存范围

除了Character没有负数概念，其缓存范围为：【0,127】，Boolean缓存范围 {true,false}

其他都是：【-128,127】

需要注意的是Integer的缓存范围是可配置的，其他的是固定的。

```
@Test
public void testCache() {

    System.out.println("=====char=====");
    Character c1 = 127;
    Character c2 = 127;
    Character c3 = 128;
```

```

Character c4 = 128;
System.out.println(c1 == c2);
System.out.println(c3 == c4);

System.out.println("=====byte=====");
Byte b1 = 127;
Byte b2 = 127;
Byte b3 = -128;
Byte b4 = -128;
System.out.println(b1 == b2);
System.out.println(b3 == b4);

System.out.println("=====short=====");
Short s1 = 127;
Short s2 = 127;
Short s3 = -129;
Short s4 = -129;
System.out.println(s1 == s2);
System.out.println(s3 == s4);

System.out.println("=====int=====");
Integer i1 = 127;
Integer i2 = 127;
Integer i3 = -129;
Integer i4 = -129;
System.out.println(i1 == i2);
System.out.println(i3 == i4);

System.out.println("=====long=====");
Long l1 = 127L;
Long l2 = 127L;
Long l3 = -129L;
Long l4 = -129L;
System.out.println(l1 == l2);
System.out.println(l3 == l4);

System.out.println("=====Boolean=====");
Boolean bb1 = false;
Boolean bb2 = false;
System.out.println(bb1 == bb2);
}

```



✓ Tests passed: 1 of 1 test – 2 ms

/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home/bin/java ...

=====char=====

true

false

=====byte=====

true

true

=====short=====

true

false

=====int=====

true

false

=====long=====

true

false

=====Boolean=====

true

Process finished with exit code 0

## new关键字

特别的，如果使用 `new` 关键字创建包装类型，其不会存放于缓存池中，而是存放于堆内存中

```
@Test
public void testCache2() {
    Integer i1 = new Integer(128);
    Integer i2 = 128;
    System.out.println(i1 == i2);
    System.out.println("equals方法: " + i1.equals(i2));
}
```

>> ✓ Tests passed: 1 of 1 test – 1 ms

1 ms /Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home/bin/java .

1 ms false

equals方法: true

Process finished with exit code 0

## 问题

池化技术可有效的节省内存空间，但是也会给我们带来一些问题。对于基本数据类型我们一般来说只关心其数值的大小，并不会去关心其对象具体。所以说对于基本数据类型的判等一般采用equals方法，这样即便数据超过缓存范围也可以准确判断。

## 谁负责缓存

java中会有专门的类负责缓存

有ByteCache用于缓存Byte对象

有ShortCache用于缓存Short对象

有LongCache用于缓存Long对象

有CharacterCache用于缓存Character对象

有IntegerCache用于缓存Integer对象

## 对于boolean属性如何命名及返回值如何定义

Boolean 作为实体类的属性的时候如何命名？ success Or isSuccess?， Boolean 作为方法返回参数的时候使用基本类型还是包装类型？

## boolean作为属性

我们测试 Boolean 作为属性？ 其生成的 getter 和 setter 方法是什么样子的，对RPC框架有什么影响。

存在四种情况：

```
Boolean success;  
Boolean isSuccess;  
boolean success;  
boolean isSuccess;
```

分别举例：

使用Lombok自动生成getter和setter方法，编译查看对应代码

```
@Data  
class BooleanType1{  
    boolean success;  
}  
@Data  
class BooleanType2{  
    boolean isSuccess;  
}
```

```

@Data
class BooleanType3{
    Boolean success;
}
@Data
class BooleanType4{
    Boolean isSuccess;
}

```

编译后查看：

```

class BooleanType1 {
    boolean success;
    public boolean isSuccess() {
        return this.success;
    }
    public void setSuccess(boolean success) {
        this.success = success;
    }
}
class BooleanType2 {
    boolean isSuccess;
    public boolean isSuccess() {
        return this.isSuccess;
    }
    public void setSuccess(boolean isSuccess) {
        this.isSuccess = isSuccess;
    }
}
class BooleanType3 {
    Boolean success;
    public Boolean getSuccess() {
        return this.success;
    }
    public void setSuccess(Boolean success) {
        this.success = success;
    }
}
class BooleanType4 {
    Boolean isSuccess;
    public Boolean getIsSuccess() {
        return this.isSuccess;
    }
    public void setIsSuccess(Boolean isSuccess) {
        this.isSuccess = isSuccess;
    }
}

```

这里可以发现如果属性是基本数据类型的 `boolean` 生成的getter和setter方法是:isXXXX()和setXXX();

如果是包装类型生成的getter和setter方法是getXXX()和setXXX()

这里可以发现，如果是基本数据类型 `boolean` 作为属性的话，属性名success和isSuccess其对应的getter和setter方法是相同的。那么如果我们的属性名是isSuccess的话，在部分RPC框架中，得到的getter方法是isSuccess()，会误认为对应的属性名称是success，会导致获取不到属性，从而报出异常。

所以说对于实体类如果存在Boolean数据类型的属性，使用包装类型。

## boolean对序列化的影响

使用Fastjson JACKSON、GSON这几个常见JSON序列话对比区别。

同样的对于基本数据类型的boolean，对不同的序列话工具会有不同的结果。而对于包装类型则没有影响。

fastjson和jackson是通过反射得到所有的getter方法（getXXX或isXXXX），然后认为 XXXX就是字段名称，并得到对应的值，直接序列化成对应JSON字符串。

Gson则是通过反射，遍历对象对应类的属性，再序列话成json字符串。

```
@Test
public void test() throws JsonProcessingException {

    Gson gson = new Gson();
    ObjectMapper om = new ObjectMapper();

    BooleanType1 booleanType1 = new BooleanType1(true);
    System.out.println("booleanType1");
    System.out.println("FastJson:boolean success: => " + JSON.toJSONString(booleanType1));
    System.out.println("Gson:boolean success: => " + gson.toJson(booleanType1));
    System.out.println("Jackson:boolean success: => " +
om.writeValueAsString(booleanType1));

    BooleanType2 booleanType2 = new BooleanType2(true);
    System.out.println("booleanType2");
    System.out.println("FastJson:boolean isSuccess: => " + JSON.toJSONString(booleanType2));
    System.out.println("Gson:boolean isSuccess: => " + gson.toJson(booleanType2));
    System.out.println("Jackson:boolean isSuccess: => " +
om.writeValueAsString(booleanType2));

    BooleanType3 booleanType3 = new BooleanType3(true);
    System.out.println("booleanType3");
    System.out.println("FastJson:Boolean success: => " + JSON.toJSONString(booleanType3));
    System.out.println("Gson:Boolean success: => " + gson.toJson(booleanType3));
    System.out.println("Jackson:Boolean success: => " +
om.writeValueAsString(booleanType3));

    BooleanType4 booleanType4 = new BooleanType4(true);
    System.out.println("booleanType4");
    System.out.println("FastJson:Boolean isSuccess: => " + JSON.toJSONString(booleanType4));
    System.out.println("Gson:Boolean isSuccess: => " + gson.toJson(booleanType4));
```

```

        System.out.println("Jackson:Boolean isSuccess: => " +
om.writeValueAsString(booleanType4));
    }

```

```

Tests passed: 1 of 1 test - 333 ms
E:\devTools\JDK\Jdk_8\bin\java.exe ...
booleanType1
FastJson:boolean success: => {"success":true}
Gson:boolean success: => {"success":true}
Jackson:boolean success: => {"success":true}
booleanType2
FastJson:boolean isSuccess: => {"success":true}
Gson:boolean isSuccess: => {"isSuccess":true}
Jackson:boolean isSuccess: => {"success":true}
booleanType3
FastJson:Boolean success: => {"success":true}
Gson:Boolean success: => {"success":true}
Jackson:Boolean success: => {"success":true}
booleanType4
FastJson:Boolean isSuccess: => {"isSuccess":true}
Gson:Boolean isSuccess: => {"isSuccess":true}
Jackson:Boolean isSuccess: => {"isSuccess":true}

Process finished with exit code 0

```

对于boolean isSuccess 不同的JSON序列化工具，生成的JSON字符串并不是一样的。那么如果对于同一对象使用不同序列化工具序列化和反序列化会产生什么结果？

```

@Test
public void testSer() throws IOException {

    BooleanType2 booleanType2 = new BooleanType2(true);
    //使用fastjson序列化
    String jsonStr = JSON.toJSONString(booleanType2);
    System.out.println("json字符串: =》" + jsonStr);
    //分别使用 fastJson 、 GSON 、 Jackson反序列化
    BooleanType2 t1 = JSON.toJavaObject(JSON.parseObject(jsonStr), BooleanType2.class);
    System.out.println("FastJson反序列化后=》" + t1);

    ObjectMapper om = new ObjectMapper();
    BooleanType2 t2 = om.readValue(jsonStr, BooleanType2.class);
    System.out.println("Jackson反序列化后=》" + t2);

    Gson gson = new Gson();
    BooleanType2 t3 = gson.fromJson(jsonStr, BooleanType2.class);
    System.out.println("Gson反序列化后=》" + t3);
}

```

✓ Tests passed: 1 of 1 test – 425 ms

```
E:\devTools\JDK\Jdk_8\bin\java.exe ...
```

```
json字符串: =》{"success":true}
```

```
FastJson反序列化后=》BooleanType2(isSuccess=true)
```

```
Jackson反序列化后=》BooleanType2(isSuccess=true)
```

```
Gson反序列化后=》BooleanType2(isSuccess=false)
```

还是Gson出现的问题，对于同一个类，使用不同的序列化工具进行，序列化反序列化，对象会产生前后不一致问题。

同样的对于fastjson和jackson来说，会根据success通过反射来找对应得setter方法，将属性set进去。而Gson会通过反射去找success属性，发现没找到，那么就只能赋予默认值false。

又一次证明了只能使用success而不能使用isSuccess

## Boolean or boolean

编码得时候使用Boolean 还是 boolean

- 对于实体类的属性，一律使用包装类型
- 对于远程调用的接口来说，必须使用包装类型。避免默认值的出现
- 对于局部变量来说使用基本数据类型

## 小结

对于布尔值如何命名和使用success还是isSuccess。

第一：布尔值命名必须去掉 is

第二：除了局部变量，其他地方一律使用包装类型

## String

String 在 java 中很常用，看似简单，也有很多知识点。

### 不可变性

对象的不可变性指的是什么？

对象的不可变性指的是在对象创建完成，我们不可以调用方法来修改其属性。

### 现象

在编程中我们常常通过等号和加号来"修改"字符串的值，为什么还是不可变得呢？

比如：

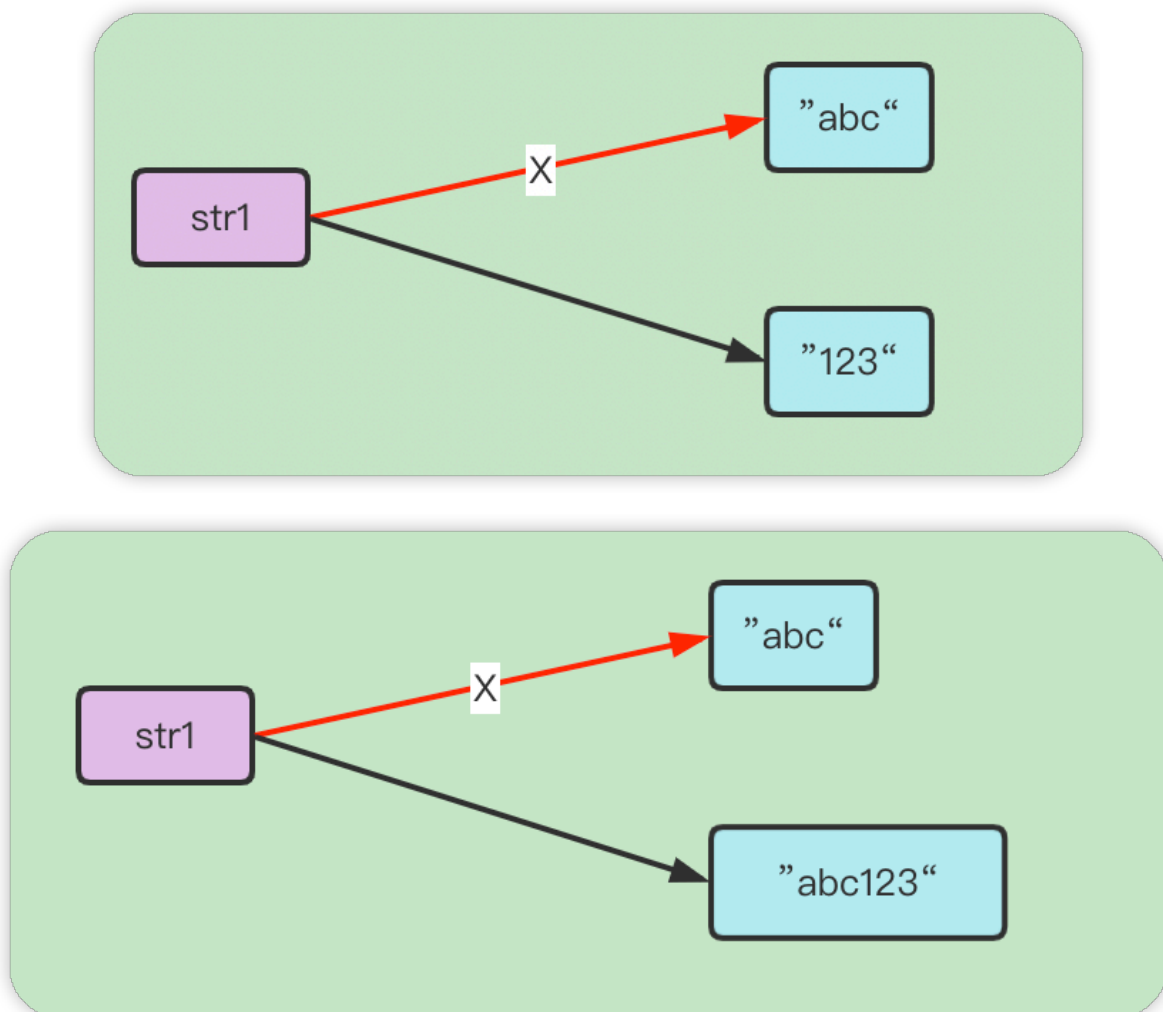
```
String str1 = "abc";
str1 = "123";
//
String str1 = "abc";
str1 += "123";
```

这不是修改了么？

这里两种方式好像都修改了str1的值，其实是修改了str1的引用，将str1指向了一个新的字符串对象。

对于字符串相加，回收先得到相加后的结果，创建对应字符串，然后赋予引用。

图示：



## String为什么是不可变的

简单理解一下为什么String不可变

打开 `String` 类的源码，可以发现 `String` 底层是一个字符数组，且该属性被 `final` 修饰：

```
private final char value[];
```

那么此刻自然就想到被 `final` 修饰的对象不可变。

其实被 `final` 修饰的对象不可变指的是，不可以修改其引用，如果说我可以通过调用对象提供的修改方法那么完全是可以修改的。

那么String类不可变的真正原因是什么呢？

- 首先String类被final修饰，也就是不可以被继承，我们知道继承及侵入式的，不可以继承那么就没有子类可以破坏其不可变性。（不仅仅是String八大基本数据类型的包装类型都是final的）
- String类底层是一个字符数组，其作为String的属性，被private修饰，也就是不提供外部访问
- String类底层是一个字符数组，其作为String的属性，被final修饰，不可修改字符数组引用
- 最后一点也是最重要的，String类中未提供任何修改其字符数组的方法（无论是私有的还是公开的），其内部方法返回的都是一个String

## String真的不可变吗

我们直到 `Java` 提供了一个很强大的机制，就是反射，那么我们是否可以通过反射来破坏String底层数组的 `private` 和 `final` 呢？

写一个测试案例：可以通过反射来修改被final修饰的属性

```
@Test
public void testFinal() throws NoSuchFieldException, IllegalAccessException {
    TestFinal testFinal = new TestFinal();
    System.out.println(testFinal.sb + ":" + VM.current().addressOf(testFinal.sb));
    final Field sb = testFinal.getClass().getDeclaredField("sb");
    final StringBuilder abc = new StringBuilder("abc");
    //反射破坏不可修改
    sb.setAccessible(true);
    sb.set(testFinal, abc);
    System.out.println(testFinal.sb + ":" + VM.current().addressOf(testFinal.sb));
}
class TestFinal {
    final StringBuilder sb = new StringBuilder("123");
}
```

结果很意外，被final修饰的属性其值可以被改变，且其内存地址也发生了改变，也就是sb的引用也修改了

```
/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents
# WARNING: Unable to attach Serviceability Agent. Una
123:31865269960
abc:31879702440
```



同理我们尝试修改String的字符数组：

```
/**
 * 我们都知道String其内部是字符数组，且是私有的，那么我们是否可以通过反射修改其私有属性
 */
@Test
public void test3() throws NoSuchFieldException, IllegalAccessException {
    String str = "123";
    System.out.println(str + ":" + VM.current().addressOf(str));

    final Field value = str.getClass().getDeclaredField("value");
    value.setAccessible(true);
    value.set(str, "abc".toCharArray());
    System.out.println(str + ":" + VM.current().addressOf(str));

    String str2 = "123";
    System.out.println(str2 + ":" + VM.current().addressOf(str2));

    String str3 = "abc";
    System.out.println(str3 + ":" + VM.current().addressOf(str3));
}
```

结果有令人很惊讶

- 可以修改String的属性字符数组的值，且不会修改其引用
- String str2 = "123";为何值为"abc"，这里我只能猜测，String的缓存池中记录着这么一个"123"字符串，但是其内部的字符数组指向的是['a','b','c']

```
# WARNING: Unable to attach Serviceability Agent. Unable to attach
123:31865252808
abc:31865252808
abc:31865252808
abc:31879669728
```

String为什么设计成不可变的

- 缓存池

String是很常用的，为了避免频繁创建相同的字符串，JVM特地在堆内存中开辟了一块空间叫字符串缓存池，专门用于缓存已创建的字符串。

如果说需要的字符串在缓存池中存在，那么直接去缓存池中取即可，不用再去创建。

- 安全

字符串用于存储的内容还是很广泛的，密码、url、账号信息等等，如果说String可以很容易的被改变，那么整个系统就没有可信度可言了。

- 线程安全

线程安全性问题，只出现在可修改的数据上，String不可变那么自动保证线程安全。

- 拷贝安全

我们知道在深拷贝的时候，需要考虑对象属性的拷贝，以不影响原型对象，而String不可变在拷贝的时候无需考虑他的拷贝。

- hash缓存

当字符串作为哈希实现的key值的时候。在对这些散列实现进行操作时，经常调用hashCode()方法。

不可变性保证了字符串的值不会改变，其哈希值也不会变，只有在首次哈希的时候会计算哈希值，之后会直接去取已计算的哈希值。

```
/** Cache the hash code for the string */
private int hash; // Default to 0
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

## 小结

以上提到的String的缓存池技术，hashCode()缓存技术，都旨在与提高性能，因为String是非常常用的数据类型，对于它的性能即便是小小的提升，映射到整个java生态中，也是庞大的提升。

## substring

介绍subString方法的原理，以及在jdk6和jdk6之后版本中的不同之处，jdk6中的substring方法的问题

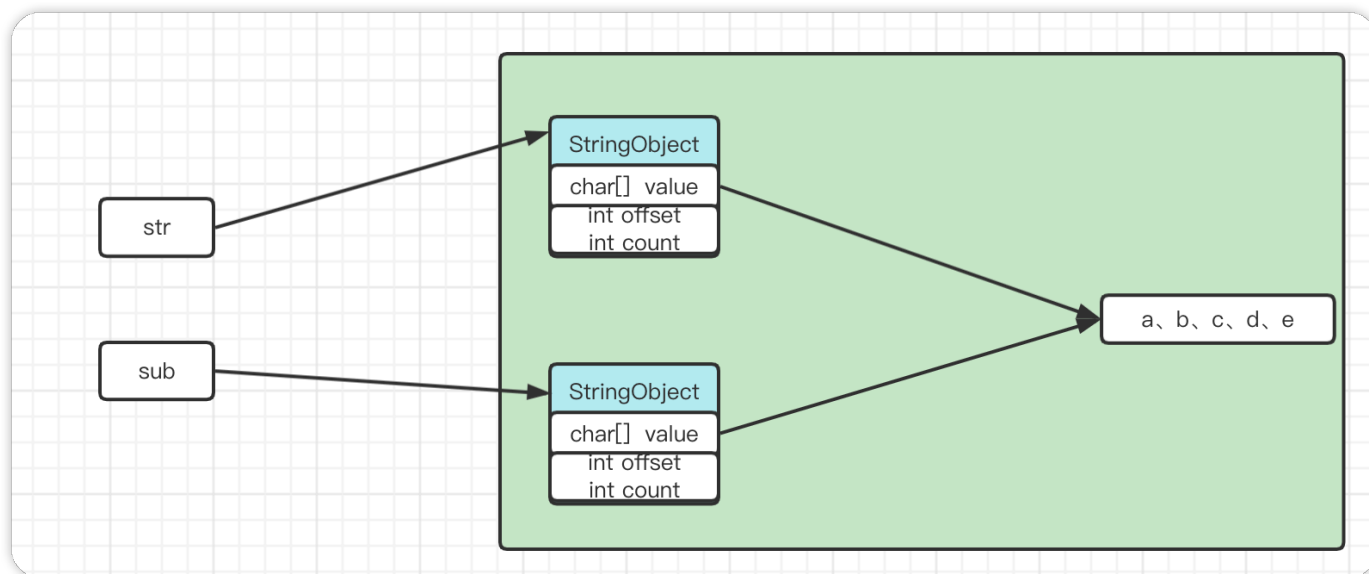
## JDK6中substring的实现原理

String底层是一个字符数组，在jdk6中String有三个成员变量：`char value[]` `int offset` `int count`。分别表示字符数组、起始下标、字符个数。

```
String(int offset, int count, char value[]) {  
    this.value = value;  
    this.offset = offset;  
    this.count = count;  
}  
  
public String substring(int beginIndex, int endIndex) {  
    //check boundary  
    return new String(offset + beginIndex, endIndex - beginIndex, value);  
}
```

在调用substring方法的时候会返回一个新的string对象，但其内部的字符数组引用任然指向原堆中的字符数组，只不过其实下标和字符数量不同。

图示：



## JDK6中substring存在的问题

由于截取的字符串和原字符串引用的是同一个字符数组，如果原字符串很大，但是截取的部分很小，那么就会导致，原来很长的字符串所指向的字符数组即便不会使用也一直被引用，就会无法回收，导致内存泄漏。还有就是效率问题，我只需要截取一小段，却引用了整个字符数组。

解决方式是截取后的字符串重新创建一个字符串。

```
x = x.substring(x, y) + ""
```

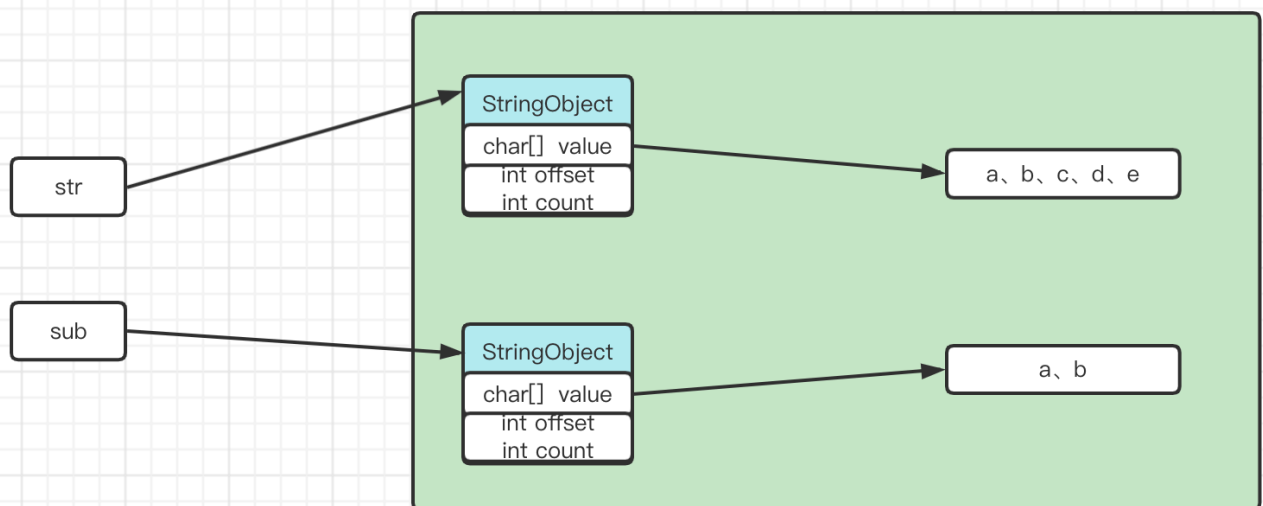
## jdk7对于sbusttring的优化

优化方式是调用sbusttring方法生成的字符串其内部字符数组的引用，指向一个新创建的字符数组。

```
//JDK 7
public String(char value[], int offset, int count) {
    //check boundary
    this.value = Arrays.copyOfRange(value, offset, offset + count);
}

public String substring(int beginIndex, int endIndex) {
    //check boundary
    int subLen = endIndex - beginIndex;
    return new String(value, beginIndex, subLen);
}
```

○ sub = str.substring(0,2) ○



replcae

```

//将所有的replacement字符替换为target字符
public String replace(CharSequence target字符, CharSequence replacement) {
    return Pattern.compile(target.toString(), Pattern.LITERAL).matcher(
        this).replaceAll(Matcher.quoteReplacement(replacement.toString()));
}
//将所有的replacement字符串替换为target字符串
public String replaceAll(String regex, String replacement) {
    return Pattern.compile(regex).matcher(this).replaceAll(replacement);
}
//将首个replacement字符串替换为target字符串
public String replaceFirst(String regex, String replacement) {
    return Pattern.compile(regex).matcher(this).replaceFirst(replacement);
}

```

## 字符串拼接

### 通过+拼接

字符串通过+拼接的原理

```

public void test1() {
    String str1 = "abc" + "123";
    System.out.println(str1);
}

void method(String str1, String str2) {
    final String strX = str1 + "," + str2;
}

```

反编译后:

```

public void test1() {
    String str1 = "abc123";
    System.out.println(str1);
}

void method(String str1, String str2) {
    (new StringBuilder()).append(str1).append(",").append(str2).toString();
}

```

- 对于编译时期就知道字面量的字符串，进行常量折叠
- 对于编译器不确定的变量，会使用StringBuilder.append拼接

## 通过concat拼接

会重新生成一个字符串对象，其内部的字符数组也是通过ArrayCopy新拷贝出来的。

```
public String concat(String str) {
    int otherLen = str.length();
    if (otherLen == 0) {
        return this;
    }
    int len = value.length;
    char buf[] = Arrays.copyOf(value, len + otherLen);
    str.getChars(buf, len);
    return new String(buf, true);
}
```

## StringBuffer&StringBuilder

可以使用在这两类对字符串进行拼接，最后toString返回即可

## 第三方工具类

StringUtils，可以使用Spring提供的也可以是apache提供的，都是一个用法，将一个String数组或集合，以某个字符分割拼接

```
@Test
public void testAppendByUtil(){
    String[] value = {"hello", "你好", "hello"};
    String result = StringUtils.join(value, ",");
    System.out.println(result);
}
```

hello,你好,hello

## String的join方法

```
@Test
public void testAppendByStr(){
    String join = String.join(",", "hello", "你好", "hello", "4");
    System.out.println(join);
}
```

hello,你好,hello,4

## 性能对比

```
@Test
public void testAppend() {
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.start("使用+拼接字符串");//任务说明
    String str1 = "";
    for (int i = 0; i < 50000; i++) {
        //str1 += "a"; 拼接代码
    }
    stopWatch.stop();
    System.out.println(stopWatch.getLastTaskName() + "消耗时
长: "+stopWatch.getTotalTimeNanos());
}
```

使用+拼接字符串消耗时长: 1942387300  
使用StringBuilder拼接字符串消耗时长: 2846001  
使用StringBuffer拼接字符串消耗时长: 4217800  
使用concat拼接字符串消耗时长: 5055200  
使用StringUtils join拼接字符串消耗时长: 39924299

结果是: StringBuilder > StringBuffer > concat > StringUtils > +

StringBuffer append方法基于StringBuilder实现, 同时也是同步的, 性能差一点点。

concat每次循环, 都会进行数组拷贝, 创建新字符串, 性能差点。但也是为了保证字符串的不可变性

StringUtils底层使用的StringBuilder实现, 拼接过程存在很多其他操作, 回去判断对象是否为空等, 性能也差点

+号是我们很常用的, 性能却最差, 这是为什么呢?

查看使用+拼接字符串的反编译后的代码:

发现每次循环都会new一个StringBuilder出来, 再进行append, 性能自然不会很高了。频繁的创建对象, 也是对内存资源的浪费。

```
void append1() {
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.start("使用+拼接字符串");
    String str1 = "";
    for (int i = 0; i < 50000; i++) {
        str1 += "a";
    }
    stopWatch.stop();
    System.out.println(stopWatch.getLastTaskName() + "消耗时长: " +
stopWatch.getTotalTimeNanos());
}
```

```

void append1()
{
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.start("\u4F7F\u7528\u62FC\u63A5\u5B57\u7B26\u4E32");
    String str1 = "";
    for(int i = 0; i < 50000; i++)
        str1 = (new StringBuilder()).append(str1).append("a").toString();

    stopWatch.stop();
    System.out.println((new
StringBuilder()).append(stopWatch.getLastTaskName()).append("\u6D88\u8017\u65F6\u957F\u
FF1A").append(stopWatch.getTotalTimeNanos()).toString());
}

```

## 小结

对于循环体内字符串的拼接禁止使用`+`，采用`StringBuilder`的`append`的方式进行字符串拼接。有并发需求时，使用`StringBuffer`代替`StringBuilder`。

## StringBuffer & StringBuilder

`String`是不可变的，java还为我们提供了两个可变的用于操作字符串的类，`StringBuffer` & `StringBuilder`

`StringBuffer`和`StringBuilder`都是`AbstractStringBuilder`的子类。底层也是字符数组，使用一个成员变量`count`来表示字符数组已使用的字符数。

```

char[] value;

int count;

```

`StringBuilder`是非线程安全的，`StringBuffer`是线程安全的（使用`Synchronized`保证）。

## String.valueOf & Integer.toString

将一个`Integer`转化为`String`有几种方式？

```

@Test
public void test(){
    int i = 10;
    String str1 = i + "";
    String str2 = Integer.valueOf(i).toString();
    String str3 = String.valueOf(10);
}

```



第一种方式使用 `StringBuilder`

```
String str1 = (new StringBuilder()).append(i).append(" ").toString();
```

第二种和第三种都是使用 `Integer.toString()`

## switch支持String

jdk7之后 `switch` 添加了对 `String` 的支持。

`switch` 目前支持的类型有 `Character`, `Byte`, `Short`, `Integer`, `String`, or an enum, `switch` 真正意义上只支持整型, 对于 `Character` 会转化成ASCII码, ASCII是一个 `int` 类型的数据。 `String` 会优先通过 `hashCode` 判断, 然后再通过 `equals` 进行安全检查, `hashCode` 也是 `int` 类型的

## int&short&byte

代码:

```
@Test
public void testInt() {
    int i = 10;
    switch (i) {
        case 1:
            System.out.println(1);
            break;
        case 2:
            System.out.println(2);
            break;
        case 3:
            System.out.println(3);
            break;
        default:
            System.out.println(i);
    }
}
```

反编译查看:

没什么特别的, `switch`对`int`支持很好。对`Short`和`byte`也是一样的, 不支持`long`

```
public void testInt()
{
    int i = 10;
    switch(i)
    {
        case 1: // '\001'
            System.out.println(1);
            break;
```

```

    case 2: // '\002'
        System.out.println(2);
        break;

    case 3: // '\003'
        System.out.println(3);
        break;

    default:
        System.out.println(i);
        break;
}
}

```

## char

代码:

```

@Test
public void testChar() {
    char c = 'a';
    switch (c) {
        case 'a':
            System.out.println('a');
            break;
        case 'b':
            System.out.println('b');
            break;
        case 'c':
            System.out.println('c');
            break;
        default:
            System.out.println(c);
    }
}

```

反编译查看:

会将char转化成对应的ascii码值，再通过整型switch

```

public void testChar()
{
    char c = 'a';
    switch(c)
    {
        case 97: // 'a'

```

```

        System.out.println('a');
        break;
    case 98: // 'b'
        System.out.println('b');
        break;
    case 99: // 'c'
        System.out.println('c');
        break;
    default:
        System.out.println(c);
        break;
    }
}

```

## string

代码:

```

@Test
public void testString() {
    String str = "abc";
    switch (str) {
        case "abc":
            System.out.println("a");
            break;
        case "bac":
            System.out.println("b");
            break;
        case "cab":
            System.out.println("c");
            break;
        default:
            System.out.println(str);
    }
}

```

反编译查看:

发现首先获取哈希值，哈希值是整型，然后进行switch，最后使用equals进行安全判断。

```

public void testString()
{
    String str = "abc";
    String s = str;
    byte byte0 = -1;
    switch(s.hashCode())
    {

```

```

case 96354:
    if(s.equals("abc"))
        byte0 = 0;
    break;

case 97284:
    if(s.equals("bac"))
        byte0 = 1;
    break;

case 98244:
    if(s.equals("cab"))
        byte0 = 2;
    break;
}
switch(byte0)
{
case 0: // '\0'
    System.out.println("a");
    break;

case 1: // '\001'
    System.out.println("b");
    break;

case 2: // '\002'
    System.out.println("c");
    break;

default:
    System.out.println(str);
    break;
}
}

```

## 字符串缓存池

创建字符串的方式有以下两种方式:

```

@Test
public void testStrCache(){
    String str1 = "abc";
    String str2 = new String("abc");
}

```

- 第一种方式通过"字面量"的形式赋值，字符串如果在缓存池中不存在，则会创建并放入缓存池
- 第二种方式会将字符串对象当作一个普通的对象类型，放在堆内存中

当我们使用字面量创建字符串的时候，jvm会对此字符串进行检查，如果该字符串在缓存池中不存在，则会创建该字符串，并将其放入字符串缓存池；如果该字符串存在，那么直接将缓存池中的字符串对象的引用返回。

```
@Test
public void testStrCache2(){
    String str1 = "abc";
    String str2 = String.valueOf("abc");//String.valueOf也是字面量，调用toString方法直接返回
    String str3 = "abc";
    System.out.println(str1 == str2);//true
    System.out.println(str2 == str3);//true
    System.out.println(str1 == str3);//true
}
```

字符串缓存池在内存中的哪个位置

jdk7之前，字符串缓存池在永久代中。

jdk7中，由于后续版本计划通过元空间代替永久代，所以先将字符串缓存池从永久代移出，暂时放入堆内存。

jdk8中，彻底废除了永久代，使用元空间代替永久代，字符串常量池从堆内存，移动到永久代。

### String长度限制？

String 存不存在长度限制呢？

- 在编译期间不可以超过  $2^{16}-1 = 65535$   
也就是我们在使用字面量对字符串赋值的时候如果字符串长度大于等于65535，就通过不了编译
- 运行期间限制：不能超过int的范围

## java中的各种关键字

### transient

短暂瞬时的意思，java提供的关键字，用于修饰成员变量。如果一个变量被 `transient` 修饰，当对象需要序列化传输、或存储时，会忽略该变量。

当我们不希望对象的某个变量需要被序列化的时候，比如我们定义一个变量，该变量我们只希望它在当前系统中使用，而不希望他在上下游系统传输，可以使用 `transient` 修饰。

被transient修饰的引用类型也就是对象类型，在被反序列化的时候初始化为null，基本数据类型为默认值int就是0。

创建一个对象，注意需要实现序列化接口支持序列化操作。如果存在特殊需求可以重写writeObject方法和readObject方法。

```

@Data
class TransientTestClass implements Serializable {
    private static final long serialVersionUID = 9167810647635375505L;

    private String str;
    private Integer value;
    private transient String name;
    private transient int age;
}

```

将对象序列化持久化到本地

```

String filePath = "/Users/rolyfish/Desktop/MyFoot/myfoot/foot/testfile";

@Test
public void test1() throws IOException {
    final TransientTestClass transientTestClass = new TransientTestClass();
    transientTestClass.setName("element");
    transientTestClass.setStr("element");
    transientTestClass.setValue(123);
    //序列化到文件
    final ObjectOutputStream objectOutputStream = new ObjectOutputStream(
        new FileOutputStream(new File(filePath,
transientTestClass.getClass().getName())));
    objectOutputStream.writeObject(transientTestClass);
    objectOutputStream.flush();
    objectOutputStream.close();
}

```

再通过反序列化将文件中的对象读出来，查看其属性值

```

/**
 * 读出来，使用对象接收看看
 */
@Test
public void test2() throws IOException, ClassNotFoundException {
    final ObjectInputStream objectInputStream = new ObjectInputStream(
        new FileInputStream(new File(filePath,
TransientTestClass.class.getName())));
    TransientTestClass transientTestClass = (TransientTestClass)
objectInputStream.readObject();
    System.out.println(transientTestClass);
}

```

结果也如我们说的一样

✓ Tests passed: 1 of 1 test – 42 ms

```
/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home/bin/java ...  
TransientTestClass(str=element, value=123, name=null, age=0)
```

Process finished with exit code 0

## instanceof

java关键字，类似于一个二元操作符，用于判断 `instanceOf` 左右两边对象类型是否一致。

```
@Test  
public void test() {  
    System.out.println(InstanceofTest.class instanceof Object);  
    System.out.println("InstanceofTest.class" instanceof String);  
    System.out.println(Integer.valueOf(10) instanceof Integer);  
  
    Object o = Integer.valueOf(10);  
    System.out.println(o instanceof String);  
}
```

✓ Tests passed: 1 of 1 test – 1 ms

```
/Library/Java/JavaVirtualMachines/zulu-8.jdk/Content  
true  
true  
true  
false
```

## synchronized

[synchronized](#)

后续重点看

## volatile

### [volatile](#)

后续重点看

## final

### [final](#)

`final` 是 `java` 提供的关键字，表示该部分不可修改，可修饰类、方法、变量。

#### final修饰类

`final` 修饰类表示该类不可以被继承。一般是类的自我保护，不希望子类对父类造成破坏。

比如说 `String` 和八大基本数据类型的包装类型

#### final修饰方法

表示该方法不可以被子类重写，但是可以在本类中重载。

#### final修饰变量

被`final`修饰的变量如果是基本数据类型则其不可变，如果是引用数据类型则其引用地址不可变。

作为局部变量

不管是引用类型还是基本数据类型，都不可以使用等号赋值。但是如果引用数据类型存在修改方法的时候是可以修改对象的引用的。

```
@Test
public void test01() {
    final StringBuilder sb = new StringBuilder("abc");
    final int i = 10;
    sb.append("123");
    System.out.println(sb.toString());
}
```

作为成员变量



```

class MemberField {
    /**
     * 被static final修饰，属于类不可变。必须1、在声明的时候赋值 2、或static代码块中赋值
     */
    static final StringBuilder sb1 = new StringBuilder();
    /**
     * 被final修饰，属于实例，不可变。必须1、在声明的时候赋值 2、非static代码块中赋值 3、构造方法赋值
     */
    final StringBuilder sb2 ;
    {
        sb2 = new StringBuilder();
    }
}

```

## static

用于修饰成员变量、方法或代码块，被static修饰的成员变量称为静态成员变量或类变量属于类，被static修饰的代码块称为静态代码块。

### 静态成员变量

也称为类变量，它不属于类的某个实例，它被所有该类的实例共享，因此存在线程安全问题。

如果类变量没有被private修饰，可以使用“类名.变量名”的方式访问。

### 静态方法

和静态变量一样，静态方法也属于类，以 `类名.方法名` 调用，在此期间不必创建类的实例，因此会方便许多。

比如说集合工具类返回空集合 `Collections.emptyList()`;

java8支持在接口中定义静态方法

```

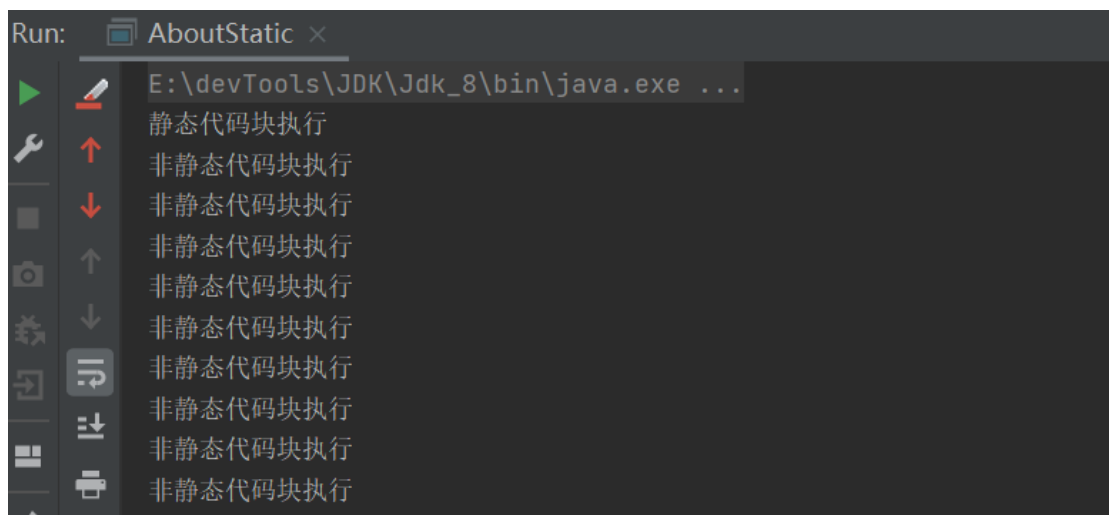
interface IStaticMethod{
    static void method1() {
        Collections.emptyList();
    }
    default void method2(){
    }
}

```

## 静态代码块

静态代码块会在类初始化的时候，将所有静态代码组合成一个 `cinit<>` 方法，由类加载器执行。一个类在其一个生命周期内只加载一次，所以说对于静态代码块，在类生命周期中只执行一次。

```
public class AboutStatic {
    static int a;
    static {
        a = 1;
        System.out.println("静态代码块执行");
    }
    {
        System.out.println("非静态代码块执行");
    }
    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            new AboutStatic();
        }
    }
}
```



## 静态类

静态内部类定义于普通类内部，可以和普通类一样使用。

下面列出类：内部类、静态内部类和匿名内部类写法

```
public class AboutStaticClass {
    static class StaticClass {
        int a;
        static int b;
    }
    class InnerClass {
        int a;
        //不可以定义静态变量
        //static int b;
    }
}
```

```

    }
    void method() {
        AboutStaticClass.InnerClass innerClass = new AboutStaticClass.InnerClass();
    }
    //匿名内部类写法
    IInterface ia = new IInterface() {
        @Override
        public void method() {
        }
    };
}
class TestClass {
    public static void main(String[] args) {
        AboutStaticClass.StaticClass staticClass = new AboutStaticClass.StaticClass();
        System.out.println(AboutStaticClass.StaticClass.b);
    }
}
@FunctionalInterface
interface IInterface{
    void method();
}

```

## const

const和final相似，用于后期扩展。

## 枚举

枚举类型是java5引入的，由一组固定常量组成的合法类型。

### 在枚举引入之前如何定义一组常量

java在枚举引入之前，我们一般会用一组int常量值，来表示一组固定的数据。比如使用1、2、3、4来表示春、夏、秋、冬。

```

/**
 * 枚举类型一般会被系统共享，所以其访问修饰符一般为public
 */
class Season {
    public static final int SPRING = 1;
    public static final int SUMMER = 2;
    public static final int AUTUMN = 3;
    public static final int WINTER = 4;
}

```

可以根据传入的int值来判断对应季节

```
@Test
```

```

public void test1() {
    final int spring = Season.SPRING;
    season(spring);
}

public void season(int value) {
    switch (value) {
        case 1:
            System.out.println("春天");
            break;
        case 2:
            System.out.println("夏天");
            break;
        case 3:
            System.out.println("秋天");
            break;
        case 4:
            System.out.println("冬天");
            break;
        default:
            System.out.println("输入不合法");
            break;
    }
}

```

这种方法称作int枚举模式。存在一些安全问题，就如上面判断季节的方法，default分支是我们不愿意看到的场景，如果说我们不加校验可能会产生问题。并且Season这个类打印出来的也只是一个int值1、2、3、4，表面并不能看出任何的意思。所以说int枚举模式他的安全性和可读性是可观的。

当然了我们也可以使用字符串作为枚举值，但是字符串的比较算法相对来说比较浪费性能，也是不可取的。

## 定义枚举

由于int枚举和字符串枚举存在着缺陷，java5引入了枚举类型 `enum type`，接下来我们看如何定义一个枚举。

使用enum声明一个枚举，在枚举类中列举枚举值，使用逗号隔开，尾部使用分号结尾。

```

enum Season2 {
    SPRING, SUMMER, AUTUMN, WINTER;
}

```

并且我们还可以为枚举定义属性：

```

@AllArgsConstructor
enum Season3 {
    SPRING(1, "春天"),
    SUMMER(1, "春天"),
    AUTUMN(1, "春天"),
    WINTER(1, "春天");
    int code;
    String msg;
}

```

## 特点

- 简约
- 和普通class类一样，枚举类可以单独存在，也可以存在于其他java类中
- 枚举类可以实现接口
- 也可以定义新的属性和方法

## switch对于枚举的支持

使用枚举改造上面代码

```

public void seasonUseEnum(Season2 season) {
    System.out.println(Season2.SPRING);
    switch (season) {
        case SPRING:
            System.out.println("春天");
            break;
        case SUMMER:
            System.out.println("夏天");
            break;
        case AUTUMN:
            System.out.println("秋天");
            break;
        case WINTER:
            System.out.println("冬天");
            break;
        default:
            System.out.println("输入不合法");
            break;
    }
}

@Test
public void test2() {
    seasonUseEnum(Season2.SPRING);
}

```

如此判断季节的方法对于传入参数存在类型限制，不会再有不合法参数的出现。一般来说我们会对枚举添加表示域的属性 and 对应的描述，方便统一管理。

```
public void seasonUseEnum(Season3 season) {
    System.out.println(Season2.SPRING);
    final StringBuilder sb = new StringBuilder();
    switch (season) {
        case SPRING:
        case WINTER:
        case AUTUMN:
        case SUMMER:
            sb.append(season.msg);
            break;
        default:
            System.out.println("输入不合法");
            break;
    }
    System.out.println(sb.toString());
}

@Test
public void test3() {
    seasonUseEnum(Season3.SPRING);
}
```

## jad查看原理

可以使用jad 反编译一下，查看一下底层原理

可以得出如下结论：

- 枚举类经过编译器编译后会被当作普通类处理，继承自 `java.lang.Enum`
- 每一个枚举项是一个 `final static` 的成员变量。天生是一个单例

```
final class Season3 extends Enum
{
    private Season3(String s, int i, int code, String msg)
    {
        super(s, i);
        this.code = code;
        this.msg = msg;
    }
    public static final Season3 SPRING;
    public static final Season3 SUMMER;
    public static final Season3 AUTUMN;
    public static final Season3 WINTER;
    int code;
```

```
String msg;
private static final Season3 $VALUES[];
static
{
    SPRING = new Season3("SPRING", 0, 1, "\u6625\u5929");
    SUMMER = new Season3("SUMMER", 1, 1, "\u6625\u5929");
    AUTUMN = new Season3("AUTUMN", 2, 1, "\u6625\u5929");
    WINTER = new Season3("WINTER", 3, 1, "\u6625\u5929");
    $VALUES = (new Season3[] {
        SPRING, SUMMER, AUTUMN, WINTER
    });
}
}
```

但是要知道switch对枚举的支持的原理，其实就在构造函数内，会调用super(s,i)。s是String类型为枚举项的字段名称，i为自动生成的编号。

我们使用jad对switch相关代码反编译一下：

- 首先枚举类中的每一个枚举都是一个单例对象，在使用new 关键字创建实例的时候会为各个实例添加一个编号 ordinal
- 在引用了枚举类的类中，会在static代码块中初始化一个int类型的数组，用于描述各个枚举值对应的编号
- switch还是对int做操作

```
{
    static final int
$SwitchMap$com$roily$booknode$javatogod$_01faceobj$jvakeywords$aboutenum$Season3[];
    static
    {

$SwitchMap$com$roily$booknode$javatogod$_01faceobj$jvakeywords$aboutenum$Season3 = new
int[Season3.values().length];

$SwitchMap$com$roily$booknode$javatogod$_01faceobj$jvakeywords$aboutenum$Season3[Season3.SPRING.ordinal()] = 1;

$SwitchMap$com$roily$booknode$javatogod$_01faceobj$jvakeywords$aboutenum$Season3[Season3.WINTER.ordinal()] = 2;

$SwitchMap$com$roily$booknode$javatogod$_01faceobj$jvakeywords$aboutenum$Season3[Season3.AUTUMN.ordinal()] = 3;

$SwitchMap$com$roily$booknode$javatogod$_01faceobj$jvakeywords$aboutenum$Season3[Season3.SUMMER.ordinal()] = 4;
    }
}
public void seasonUseEnum(Season3 season)
{
    System.out.println(Season2.SPRING);
}
```

```
StringBuilder sb = new StringBuilder();

switch(_cls1..SwitchMap.com.roily.booknode.javatogod._01faceobj.javakeywords.aboutenum
.Season3[season.ordinal()])
{
    case 1: // '\001'
    case 2: // '\002'
    case 3: // '\003'
    case 4: // '\004'
        sb.append(season.msg);
        break;
    default:
        System.out.println("\u8F93\u5165\u4E0D\u5408\u6CD5");
        break;
}
}
```

## 异常处理

`Throwable` 类下有两个重要的子类：`Error` 和 `Exception`，并且这两个子类下面也存在着大量的子类。

`Error` 表示系统或硬件级别的错误，由java虚拟机抛出异常，程序员无法处理。

`Exception` 表示程序级别的错误，是由于程序设计不完善而出现的问题，程序员必须手动处理

## 异常类型

主要分两大类：

- 受检异常(`checked exception`)
- 非受检异常(`unchecked exception`)

## 受检异常

受检异常声明：在对应方法上通过 `throws` 关键字，声明一个异常。然后此方法在被调用的时候，调用方一定要对其做处理(要么捕获、要么向上抛出)，否则是无法通过编译的。

所以当我们希望调用者，必须处理一些特殊情况的时候，就可以声明受检异常。

受检异常在io操作中使用的非常频繁，比如说 `FileNotFoundException` 异常以及 `IOException` 及其子类。

比如：



```

public void test1() throws IOException {
    IOUtils.readLines(new FileInputStream("filename"), StandardCharsets.UTF_8);
}
public void test2() {
    try {
        IOUtils.readLines(new FileInputStream("filename"), StandardCharsets.UTF_8);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

查看 IOUtils.readLines()方法：发现此方法声明了受检异常：

```

public static List<String> readLines(final InputStream input, final Charset charset) throws IOException {
    final InputStreamReader reader = new InputStreamReader(input, Charsets.toCharset(charset));
    return readLines(reader);
}

```

## 非受检异常

非受检异常，在编码的时候不用强制捕获，但是如果不捕获，在出现异常的时候就会中断程序的运行。

一般来说都是运行时异常，为 `RuntimeException` 及其子类。

比如说空指针异常(NPE)、数组下标越界异常(IOE)以及一些我们自定义的运行期间异常。对于非受检异常来说，如果代码编写的合理，这些异常都是可以避免的。

## 关键字

- throws 方法声明异常
- throw 后跟异常实例显示抛出异常
- try 用来包裹一块可能出现异常的代码块
- catch 跟在try代码后，指定异常类型，并对异常进行处理
- finally 一些代码无论是否出现异常都会执行，可以定义在finally代码块李

## 异常处理

要么自己try catch处理

要么向上抛出，交给调用者处理

## 自定义异常

一般通过继承 `RuntimeException` 定义一个自定义异常，用于抛出一些错误的业务。

```

public class MyException extends RuntimeException{

    private final String DEFAULT_ERROR_CODE = "5000";
    private final String DEFAULT_ERROR_MSG = "运行时异常";
}

```

```

String code;
String msg;
//someMethod
public MyException(Throwable cause, String code, String msg) {
    super(cause);
    this.code = code;
    this.msg = msg;
}
}

```

## 异常链

是指java在运行期捕获了一个异常，处理的时候，抛出了一个新的异常，所抛出的新的异常包含前一个异常的信息，如此形成一个异常链。

如果抛出的异常不包含前一个异常信息的话，我们就不会清楚的知道这个异常具体出现的原因：

```

public void test1() {
    try {
        String str = null;
        byte[] bytes = str.getBytes(StandardCharsets.UTF_8);
    } catch (NullPointerException npe) {
        throw new MyException("5000", "空指针异常");
    }
}

```

```

E:\devTools\JDK\Jdk_8\bin\java.exe ...

com.roily.booknode.javatogod._01faceobj.abouthrowable.MyException Create breakpoint
at com.roily.booknode.javatogod._01faceobj.abouthrowable.MyExceptionTest.test1(MyExceptionTest.java:19) <25 internal lines>

Process finished with exit code -1

```

如果我们包含前一个异常信息，在异常抛出的时候可以，追溯到根本原因：

```

public void test1() {
    . . .
    throw new MyException(npe, "5000", "空指针异常");
    . . .
}

```

```

E:\devTools\JDK\Jdk_8\bin\java.exe ...

com.roily.booknode.javatogod._01faceobj.abouthrowable.MyException: java.lang.NullPointerException
at com.roily.booknode.javatogod._01faceobj.abouthrowable.MyExceptionTest.test1(MyExceptionTest.java:19) <25 internal lines>
Caused by: java.lang.NullPointerException Create breakpoint
at com.roily.booknode.javatogod._01faceobj.abouthrowable.MyExceptionTest.test1(MyExceptionTest.java:17)
... 25 more

Process finished with exit code -1

```

## try-with-resources

java对于资源的操作，比如说io流、数据库连接，这些资源在非常昂贵，必须在使用结束后显示的关闭资源。

即在finally代码块内调用对应资源的close()方法。

```
public void test2() {
    BufferedReader bi = null;
    try {
        bi = new BufferedReader(new FileReader("filename"));
        String line;
        while ((line = bi.readLine()) != null){
            System.out.println(line);
        }
    } catch (IOException e) {
        //dosomething
    } finally {
        try {
            IOUtils.close(bi);
        } catch (IOException e) {
            //dosomething
        }
    }
}
```

java7 开始提供了一个跟好的处理资源的方式：try-with-resources 语句。这是一个类似于语法糖的语法，方便程序员编码，但是经过编译器编译后，都会转化成jvm认识的。

将资源定义在try括号内，便无需我们手动去关闭资源了：

```
@Test
public void test4() {
    try( BufferedReader bi = new BufferedReader(new FileReader("filename"))) {
        String line;
        while ((line = bi.readLine()) != null){
            System.out.println(line);
        }
    } catch (IOException e) {
        //dosomething
    }
}
```

可以使用jad反编译查看一下：

发现编译器帮我们做了：

```

        if(bi != null)
            if(throwable != null)
                try
                {
                    bi.close();
                }
                catch(Throwable throwable1)
                {
                    throwable.addSuppressed(throwable1);
                }
            else
                bi.close();
        break MISSING_BLOCK_LABEL_113;

```

## finally & return

- finally代码块一定会执行么？
- return的结果是否被finally影响？
- return和finally代码执行顺序，孰先孰后？

### finally代码块不一定执行

finally代码块不一定会执行

- 当我们的代码在进入try代码块之前就已经return了，那么整个方法就结束了，finally代码块就不会执行
- 当虚拟机强制停止的时候 exit(0),finally代码块就不会执行

例：

以下两种方式finally代码块都不会执行

```

public StringBuilder method1(Boolean flag) {
    StringBuilder sb = new StringBuilder();
    if (flag) {
        sb.append("方法在try代码块之前return\n");
        return sb;
    }
    try {

    } catch (Exception e) {
        System.out.println("进入try代码块\n");
    } finally {
        System.out.println("finally代码块执行\n");
    }
    return sb;
}

public StringBuilder method2(Boolean flag) {
    StringBuilder sb = new StringBuilder();
    if (flag) {
        System.exit(0);
    }
}

```

```

    }
    try {

    } catch (Exception e) {
        System.out.println("进入try代码块\n");
    } finally {
        System.out.println("finally代码块执行\n");
    }
    return sb;
}

@Test
public void test1() {
    method1(true);
    method2(true);
}

```

## finally对return结果的影响

finally代码可能会对return的结果产生影响。

对于基本数据类型和一些不可变的引用类型return的结果不受finally的影响

对于可变的提供修改方法的引用类型，return的结果会受到finally的影响

- 对于基本数据类型 和 一些不可变的比如说String

finally代码块执行但是不影响return的结果

```

public int method3() {
    int i = 0;
    try {
        return i;
    } finally {
        System.out.println("finally代码块执行");
        i += 1;
    }
}

public String method4() {
    String str = "123";
    try {
        return str;
    } finally {
        System.out.println("finally代码块执行");
        str += "abc";
    }
}

@Test
public void test2() {
    int i = method3();
    System.out.println("method3返回结果: " + i);
}

```

```
String str = method4();
System.out.println("method4返回结果: " + str);
}
```

✓ Tests passed: 1 of 1 test – 4 ms

E:\devTools\JDK\Jdk\_8\bin\java.exe ...

finally代码块执行

method3返回结果: 0

finally代码块执行

method4返回结果: 123

Process finished with exit code 0

- 对于可修改的引用类型(比如说StringBuilder)

finally代码会执行且影响了返回的结果

```
public StringBuilder method5() {
    StringBuilder sb = new StringBuilder("");
    try {
        return sb.append("123");
    } finally {
        System.out.println("finally代码块执行");
        sb.append("abc");
    }
}

@Test
public void test3() {
    StringBuilder sb = method5();
    System.out.println("method5返回结果: " + sb.toString());
}
```

✓ Tests passed: 1 of 1 test – 16 ms

E:\devTools\JDK\Jdk\_8\bin\java.exe ...

finally代码块执行

method5返回结果: 123abc

Process finished with exit code 0

所以说我们可以得出一个结论：

return会记住需要返回结果的字面量信息，对于基本数据类型来说就是值，对于引用类型来说就是引用地址的值。对于基本数据类型和不可变引用类型需要通过`=`等号赋值，那就直接修改了引用，而return所记住的引用指向的对象并没有被修改。那么对于可变引用类型来说，return所记住的引用指向的对象可以在finally中被修改。

### return和finally代码执行顺序

其实在上一个例子中已经有结果了，我们可以发现返回的sb为 123abc。

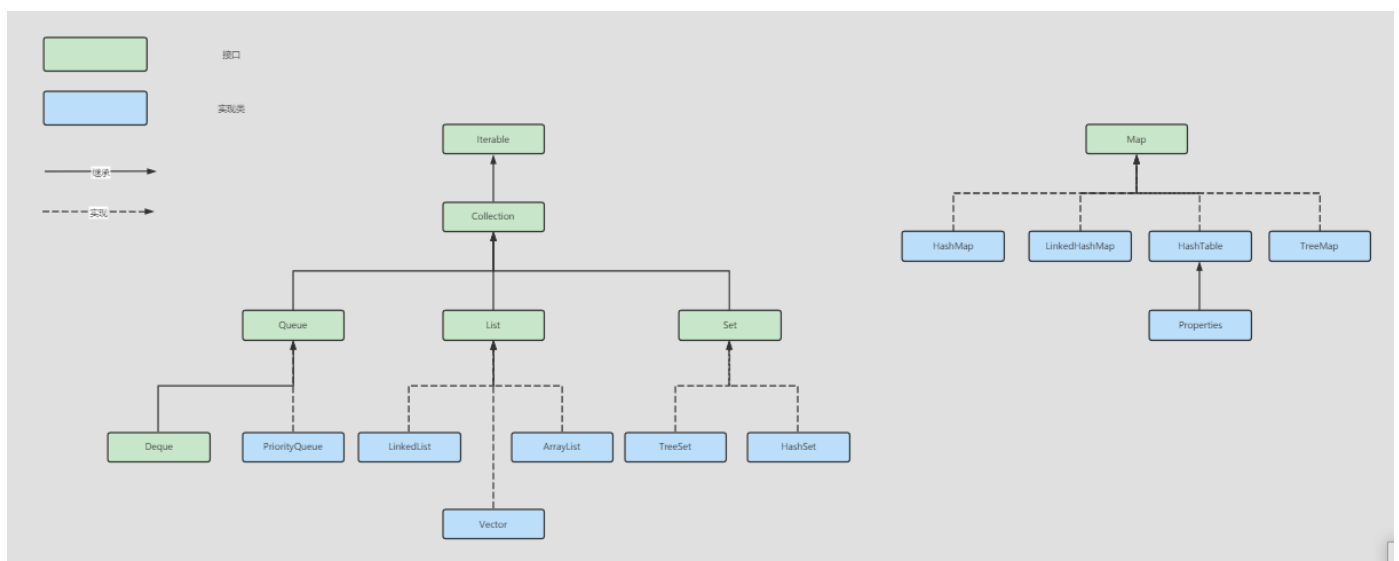
所以说可以得出的结论是：

return 的代码执行在finally代码块之前

finally代码执行在return代码之后，在完全return之前

## 集合

集合相关简单关系如下图，没有列出所有的集合类



## Iterable

- `Iterable` 接口提供了一个获取迭代器的抽象方法，各种集合类去实现它，返回各自需要的迭代器 `Iterator`。这些迭代器一般作为各种集合的内部类。
- 一个遍历方法 `foreach(Consumer action)`。各个集合实现
- 一个获取分离迭代器的方法。①可以split集合，用于可能并行操作的场景。②遍历

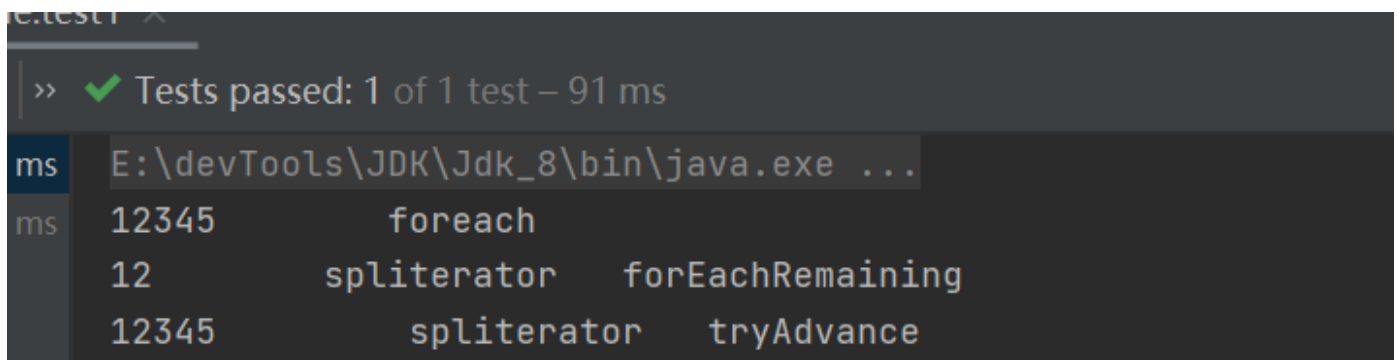
以ArrayList为例：

```
@Test
public void test1() {
    List<String> list = new ArrayList<>(Arrays.asList("1", "2", "3", "4", "5"));

    list.forEach(System.out::print);
    System.out.println("        foreach");

    Spliterator<String> spliterator = list.spliterator().trySplit();
    spliterator.forEachRemaining(System.out::print);
    System.out.println("        spliterator    forEachRemaining");

    Spliterator<String> spliterator1 = list.spliterator();
    while (spliterator1.tryAdvance(System.out::print)) ;
    System.out.println("        spliterator    tryAdvance");
}
```

A screenshot of a Java IDE's output window. At the top, it shows a green checkmark and the text "Tests passed: 1 of 1 test - 91 ms". Below this, the command prompt shows the execution of a Java program. The output consists of three lines of numbers: "12345", "12", and "12345". Each line is followed by a label: "foreach", "spliterator forEachRemaining", and "spliterator tryAdvance" respectively. The labels are aligned to the right of the numbers.

```
>> ✓ Tests passed: 1 of 1 test - 91 ms
ms E:\devTools\JDK\Jdk_8\bin\java.exe ...
ms 12345        foreach
   12        spliterator    forEachRemaining
   12345        spliterator    tryAdvance
```

使用trySplit方法将集合分割为多个小集合：每一次trySplit都会跟新 Spliterator的index属性

```
public void test3() {
    List<String> list = new ArrayList<>(Arrays.asList("1", "2", "3", "4", "5", "6",
"7", "8", "9", "10"));
    Spliterator<String> spliterator = list.spliterator();

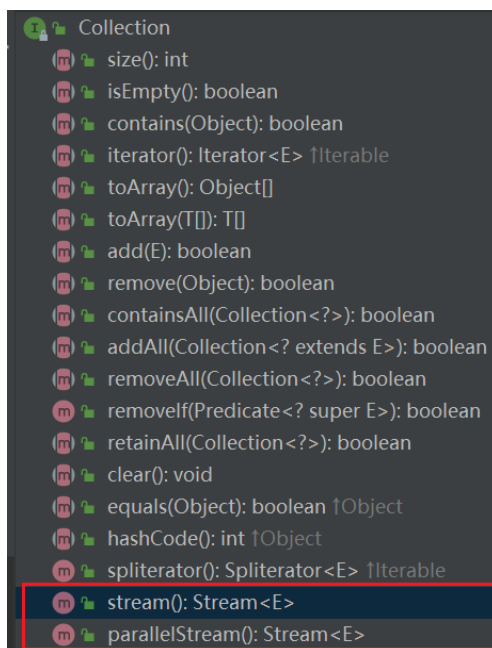
    Spliterator<String> spliteratorTemp = null;
    while (null != (spliteratorTemp = spliterator.trySplit())) {
        spliteratorTemp.forEachRemaining(System.out::print);
        System.out.println();
    }
    spliterator.forEachRemaining(System.out::print);
}
```



```
> ✓ Tests passed: 1 of 1 test – 78 ms
E:\devTools\JDK\Jdk_8\bin\java.exe ...
12345
67
8
9
10
Process finished with exit code 0
```

## collection

Collection接口中除了一些关于集合状态的方法合一些对集合操作的方法外，还有两个获取流的方法



### 使用stream来对集合进行操作

```
@Test
public void test1() {
    System.out.println("=====流 转集合=====");
    List<String> list = new ArrayList<>(Arrays.asList("1", "1", "2", "3", "4", "5", "6", "7", "8", "9"));
    List<String> collect1 = list.stream().collect(Collectors.toList());
    collect1.stream().forEach(System.out::print);
    System.out.println();

    System.out.println("=====遍历=====");
    list.stream().forEach(System.out::print);
    System.out.println();
}
```

```

        System.out.println("=====过滤=====");
        List<String> collect2 = list.stream().filter((ele) ->
ele.equals("2")).collect(Collectors.toList());
        collect2.stream().forEach(System.out::print);
        System.out.println();

        System.out.println("=====映射=====");
        List<Integer> collect3 =
list.stream().map(Integer::valueOf).collect(Collectors.toList());
        collect3.stream().forEach(System.out::print);
        System.out.println();

        System.out.println("=====求和 求平均值=====");
        int sum = list.stream().mapToInt(Integer::valueOf).sum();
        System.out.println(sum);

        System.out.println("=====去重=====");
        List<String> collect4 = list.stream().distinct().collect(Collectors.toList());
        collect4.stream().forEach(System.out::print);
        System.out.println();

        System.out.println("=====判断=====");
        final boolean b1 = list.stream().allMatch(ele -> "2".equals(ele));
        final boolean b2 = list.stream().anyMatch(ele -> "2".equals(ele));
        System.out.println(b1 + " " + b2);
        System.out.println();

        System.out.println("=====获取option=====");
        String s1 = list.stream().findAny().get();
        String s2 = list.stream().findFirst().get();
        System.out.println(s1 + " " + s2);
        System.out.println();

        System.out.println("=====对每一个元素进行操作=====");
        List<String> collect5 = list.stream().peek(ele -> {
            if (ele.equals("2")){
                System.out.println("xxxx");
            }
        }).collect(Collectors.toList());
        collect5.stream().forEach(System.out::print);
        System.out.println();
    }
}

```

## Collectors

`Collectors` 构造器私有化且未提供获取实例的方法，那么此类无法实例化。这是一个工具类，可以加快我们处理集合的效率。

我们经常需要将一个处理过的Stream转化为集合类，需要调用`collect()`方法，此方法需要一个参数：`Collector`，实现`Collector`接口还是很麻烦的，所以`Collectors`提供了许多静态方法，给我们构建需要的`Collector`。

+++以下例子基于着两个集合：

```
final List<String> list1 = Arrays.asList("a", "ab", "abc", "abcd", "abcd");
final List<String> list2 = Arrays.asList("1", "12", "123", "1234", "1234");
```

### toList

`Collector.toList` 方法，查看源码发现，默认转化为`ArrayList`。

```
List<String> collect1 =
list1.stream().filter("a"::equals).collect(Collectors.toList());
```

### toSet

`Collector.toSet` 方法，查看源码发现，默认转化为`HashSet`。 转化为元素不重复集合

```
final Set<String> collect2 = list1.stream().collect(Collectors.toSet());
```

### toCollection

以上的 `toList`、`toSet` 方法转化的是特定的集合，那么如果有特殊需求需要转化为自定义集合的话就需要使用 `toCollection` 方法。

查看源码发现就是自定义集合类型：

```
final LinkedList<String> collect3 = list1.stream().collect(Collectors.toCollection(() -
> new LinkedList<>()));
//lambda表达式写法
final LinkedList<String> collect4 =
list1.stream().collect(Collectors.toCollection(LinkedList::new));
```

### toMap

将集合元素转化为map，默认`HashMap`。`Collectors.toMap()` 方法需要两个参数：`keyMapper`和`valueMapper`，两个参数都是 `Function` 接口的实现类，参数是集合元素，返回结果是对应key value。

```
final Map<String, Integer> map1 =
list1.stream().collect(Collectors.toMap(String::toString, String::length));
```

如果转化后的map的key存在重复元素，会报 `java.lang.IllegalStateException` 异常。需要我们主动合并。也就是 `Collectors.toMap` 的几个重载

这个合并的大致思路就是，会将存在重复记录的map节点提出来，然后重复记录的key对应的了两个value作为BinaryOperator接口的参数，返回结果类型和两个参数类型都一样。

比如：上面的集合转化成map{ab=2, a=1, abc=3, abcd=4,abcd=4},这个map是存在key重复记录的，是不允许的，那么需要将这个map分为两个map1{ab=2, a=1, abc=3, abcd=4},map2{abcd=4}。然后将两个map对应key重复的记录的value提出来作为BinaryOperator接口apply(T t,T u)的参数，我们这里做一个相加，即apply(4,4) return 4 + 4;。那么最终的结果为 map{ab=2, a=1, abc=3, abcd=8}。

```
toMap(Function<? super T, ? extends K> keyMapper,  
        Function<? super T, ? extends U> valueMapper,  
        BinaryOperator<U> mergeFunction)
```

```
final Map<String, Integer> map2 =  
list1.stream().collect(Collectors.toMap(String::toString, String::length,  
Integer::sum));  
System.out.println(map2);
```

还有一个重载，可以自定义map

```
final Map<String, Integer> map3 =  
list1.stream().collect(Collectors.toMap(String::toString, String::length, Integer::sum,  
LinkedHashMap::new));
```

## collectingAndThen()

此方法允许我们对转化后的集合再做一次操作

这里的第二个参数，是一个函数式接口实现类，需要注意 第一个泛型 R是第一步流转集合的结果，也是函数式接口Function的apply(T t)方法的参数，第二个参数RR为apply(T t)方法返回结果，也是最终需要返回的结果，可以是任意的。这里返回集合

```
Function<R,RR> finisher
```

```
final List<String> collect5 =  
list1.stream().collect(Collectors.collectingAndThen(Collectors.toList(),  
    (list -> list.stream().filter("abc"::contains).collect(Collectors.toList()))));  
collect5.forEach(System.out::println);
```

## joining

将集合内元素拼接成字符串

参数说明：

第一个参数：分割符号

第二个参数：返回结果字符串前缀

第三个参数：返回结果字符串后缀

```
final String joinResult = list1.stream().collect(Collectors.joining(", ", "<", ">"));
System.out.println(joinResult);
```

## counting

统计个数

```
final Long size = list1.stream().collect(Collectors.counting());
System.out.println(size);
```

## summarizingDouble/Long/Int()

做统计

这里对集合内字符串长度做统计，得出合、最大、最小值

```
final IntSummaryStatistics intSummaryStatistics =
list1.stream().collect(Collectors.summarizingInt(String::length));
System.out.println(intSummaryStatistics.getSum());
System.out.println(intSummaryStatistics.getMax());
System.out.println(intSummaryStatistics.getMin());
```

## groupBy

以一定条件分组，这里以字符串长度分组

```
//分组
final Map<Integer, List<String>> map =
list1.stream().collect(Collectors.groupingBy(String::length, Collectors.toList()));
System.out.println(map);
```

## partitioningBy

特殊的分组，将集合分为两组，key值为boolean

```
//特殊分组，以boolean作为map的key
final Map<Boolean, List<String>> map1 =
list1.stream().collect(Collectors.partitioningBy(ele -> ele.length() > 2
));
System.out.println(map1);
```

## Set & List

`Set` 和 `List` 接口都是 `Collection` 接口的子接口，用于存储同一类型的元素。

`List`：元素按顺序插入，可重复

`Set`：元素插入无序，不可重复。`Set`的实现由`HashSet`、`TreeSet`，虽然`set`插入无序但是`TreeSet`底层原理是红黑树，元素整体上大小有序。

## ArrayList & LinkedList & Vector

这三个类都是 `List` 的实现类。

### ArrayList

`ArrayList` 底层是一个可边长数组，数据连续，当容量补不足的时候会进行扩容，扩1.5倍，使用 `System.arraycopy()`进行浅拷贝。

`ArrayList`实现了 `RandomAccess` 接口，表明支持随机访问，搜索效率高。

`elementData` 使用 `transient` 修饰，优化序列化传输和存储

如果说在使用`ArrayList`之前知道需要存入集合的元素大致个数，可以一次性将集合扩容足

`ensureCapacity(int minCapacity)`，避免频繁扩容导致降低集合效率。

重要属性：

- `elementData` 存放元素的数组
- `size` 集合大小（元素个数）

```
transient Object[] elementData;
private int size;
private static final int DEFAULT_CAPACITY = 10;
//private static final Object[] EMPTY_ELEMENTDATA = {};
//private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
```

### 优缺点

优点：

- 搜索效率高
- 优化了序列化传输和序列化存储。（重写了`WriteObject`和`ReadObject`方法，不对`null`元素序列化传输）

缺点：

- 对插入不友好

插入默认尾插法：如果插入时容量足够，直接在对应位置赋值即可，但是容量如果不足的化，首先需要扩容，扩容时就必须涉及数组的拷贝，效率自然受影响。

如果说插入位置是程序员指定的，那么需要将该位置及其之后的元素都后移一位，然后再赋值，效率也会受影响。

## LinkedList

`LinkedList` 除了实现了 `List` 接口，还实现了 `Deque` 接口，实现了 `offer\peek\poll` 等方法，对集合的操作更加灵活。

LinkedList 底层是一个双向链表，数据不连续，没有容量限制。

对插入友好，对访问不友好。 `LinkedList` 不支持随机访问。

链表的访问效率地下，特别的：如果我们每次访问的元素在链表尾部的时候，那么每次遍历都几乎需要循环整个链表。

重要属性:

- size 集合大小
- first 头节点
- last 尾节点

```
transient int size = 0;
transient Node<E> first;
transient Node<E> last;
```

### 优缺点

优点：

- 没有容量限制，添加元素无需考虑扩容，且添加元素只需要修改引用，效率较高

缺点：

- 访问效率低下

## Vector

Vector 的实现和 ArrayList 基本相同，主要存在如下不同处：

- Vector 属于强同步类，而ArrayList非同步类
- Vector默认每次扩容两倍，ArrayList扩容1.5倍
- Arraylist对序列话传输和存储做了优化，而Vector没有

## Vector关于扩容

容量增长步数 `capacityIncrement` 如果不设置，默认每次扩两倍，如果设置，每次扩容`capacityIncrement`。

[illegible]

## Collections

`Collections` 的构造函数私有化被 `private` 修饰，不可实例化，除了构造方法外，存在很多被 `static` 修饰的静态方法，目的在于对集合进行操作。

- 排序

调用的就是List的sort方法

```
//必须是Comparable的子类
void sort(List<T> list);
//可自定义比较器
void sort(List<T> list, Comparator<? super T> c);
```

- 搜索

二分查找。如果支持随机访问且集合不是很大，调用`indexedBinarySearch`方法，否则调用`iteratorBinarySearch`方法。

```
binarySearch(List<? extends Comparable<? super T>> list, T key);
binarySearch(List<? extends T> list, T key, Comparator<? super T> c);
```

返回指定集合，指定元素的出现次数

```
static int frequency(Collection<?> c, Object o);
```

返回目标集合在源集合中首次出现的下标。

返回目标集合在源集合中首次出现的下标。

没有则返回-1

```
public static int indexOfSubList(List<?> source, List<?> target) ;
public static int lastIndexOfSubList(List<?> source, List<?> target);
```

- 复制集合

将源集合复制到目标集合中

```
public static <T> void copy(List<? super T> dest, List<? extends T> src);
```

- 反转

将集合元素反转

```
void reverse(List<?> list);
```



- 洗牌

打乱现有元素顺序，达到 洗牌 效果

```
void shuffle(List<?> list, Random rnd);
void shuffle(List<?> list)
```

- 交换

这个方法在reverse中也可能会使用到

```
swap(List<?> list, int i, int j);
```

- 填充

填充集合。以某个对象替换集合中的所有元素

```
void fill(List<? super T> list, T obj);
```

- 拷贝

将源集合中的元素拷贝到目标集合。src 是源集合，dest 是目标集合。

目标集合的 size 需要大于等于源集合，否则会报出 `IndexOutOfBoundsException` 异常。

拷贝过后目标集合和源集合共享集合内的元素

```
void copy(List<? super T> dest, List<? extends T> src)
```

```
@Test
public void test1() {
    final StringBuffer sb1 = new StringBuffer("a");
    final StringBuffer sb2 = new StringBuffer("b");
    final StringBuffer sb3 = new StringBuffer("c");
    final StringBuffer sb4 = new StringBuffer("d");
    final StringBuffer sb5 = new StringBuffer("e");
    final StringBuffer sb6 = new StringBuffer("f");
    final List<StringBuffer> sbSource = Arrays.asList(sb1, sb2, sb3, sb4, sb5);
    final List<StringBuffer> sbTarget = Arrays.asList(sb6, sb6, sb6, sb6, sb6,
sb6);
    Collections.copy(sbTarget, sbSource);
    sb1.append(" | update |");

    sbTarget.forEach(System.out::print);
}
```

- 最大值最小值

获取一个集合的极值，如果元素未实现Comparable接口，需要自定义Comparator。

```
<T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll);  
<T> T min(Collection<? extends T> coll, Comparator<? super T> comp);  
<T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll);  
<T> T max(Collection<? extends T> coll, Comparator<? super T> comp);
```

- 集合旋转特定距离

什么意思？相当于集合右移  $\text{distance} \% \text{size}()$ 。右边被挤出来的元素添加到集合头部。

如果distance为正数，整体右移，负数整体左移。

比如：

```
@Test  
public void test1(){  
    List<Integer> integers = Arrays.asList(0,1, 2, 3, 4, 5, 6, 7, 8, 9);  
    Collections.rotate(integers,5);  
    integers.stream().forEach(System.out::print);  
}
```

✓ Tests passed: 1 of 1 test – 51 ms

E:\devTools\JDK\Jdk\_8\bin\java.exe ...

5678901234

Process finished with exit code 0

```
public static void rotate(List<?> list, int distance);
```

- 转化集合

注意转化为不可变集合后，源集合任然可以进行修改操作并且可以直接影响到不可变集合的不可变性。因为Collections转化不可变集合的操作是将源集合作为转换后不可变集合的属性。

```
//将目标集合转化成不可变集合，如果调用修改Api则会报出UnsupportedOperationException异常  
public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);  
public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m);  
//转化为受检查的集合 在添加时会Class.isInstance(o)判断  
<E> Collection<E> checkedCollection(Collection<E> c,Class<E> type);  
//转化为同步集合  
synchronizedCollection(Collection<T> c);
```

- 创建集合

返回的集合类型没有实现add等添加方法，如果做添加元素操作，会报出

`UnsupportedOperationException` 异常

//创建只有一个元素的集合

```
public static <T> Set<T> singleton(T o);
```

返回空集合,返回的集合类型没有实现add等添加方法，如果做添加元素操作，会报出

`UnsupportedOperationException` 异常

```
<T> Set<T> emptySet();
```

生成只有一个元素的集合，该集合不可变

//set

```
<T> Set<T> singleton(T o);
<T> List<T> singletonList(T o);
<K,V> Map<K,V> singletonMap(K key, V value);
```

生成一个由指定对象的 n 个副本组成的不可变列表

//n拷贝个数      o 集合元素

```
<T> List<T> nCopies(int n, T o);
```

生成一个线程安全的集合

`Collections.SynchronizedCollection` 和 `Vector` 的区别：

- ①两者实现同步的关键就在于使用 `Synchronized` 关键字实现，而 `Vector` 大部分代码使用的是同步方法，也就是锁的 `this`。而 `Collections.SynchronizedCollection` 可以指定锁的对象 `mutex`，如果不传默认锁的还是 `this`。
- ② `Vector` 的底层是一个对象数组，在其构造函数的重载中，可以直接将一个 `Collection` 转化为 `Vector`，但是如果被转化的集合是一个 `LinkedList` 的时候，需要改变其底层数据结构，也就是需要调用 `toArray()` 方法，将链表转化为数组。而 `Collections.SynchronizedCollection` 是不需要改变集合底层结构的，同样的被转化的集合作为 `Collections.SynchronizedCollection` 的内部属性。

```
<T> Collection<T> synchronizedCollection(Collection<T> c);
//mutex 作为对象监视器。如果主动设置，则锁的是mutex。否则默认锁的this(这也是和Vector的区别)
<T> Collection<T> synchronizedCollection(Collection<T> c, Object mutex);
```

## Set如何保证元素不重复

Set 的实现主要有两个，一个是 `HashSet`，一个是 `TreeSet`。特点是元素不重复

- `HashSet`

`HashSet` 基于 `HashMap` 实现，`HashMap` 的 key 值不重复，`HashSet` 的元素就是 `HashMap` 的key值。只能存在一个null元素（`HashMap` 中，null的hash值为0）。

其判重方法是：首先使用 `hash` 值(散列值)判断，如果散列值不相等那么直接就不相等，如果散列值相等再使用 `equals()` 方法进行安全校验。原因在于：哈希值的比较效率高于对象的 `equals()` 方法。

- `TreeSet`

`TreeSet` 基于 `TreeMap` 实现，`TreeMap` 底层是一颗红黑树(红黑树是对平衡二叉查找树的优化)。其内不可存储null元素(会报NPE异常)。

其判重方式是：①如果元素实现了 `Comparable` 接口，直接使用元素的 `compareTo()` 方法。②如果元素没有实现 `Comparable` 接口必须指定 `Comparator`，调用 `Comparator.compare()` 方法。

都是子节点与父节点进行比较，比较结果小于0作为左孩子，大于0作为右孩子，等于0替换父节点value值。

## hashMap & hashTable

`HashTable` 是一个较为古老的类，是一个线程安全的key - value键值对数据类型，其有一个 `Property` 子类，一般作为配置文件的工具类。

`HashMap` 可以认为是单线程环境下 `HashTable` 的替代品，其在避免哈希冲突、查找效率上都比 `HashTable` 要强。

一般情况下，`HashTable` 已被弃用，单线程环境下使用 `HashMap`，多线程环境需要保证线程安全的情况下使用 `ConcurrentHashMap`。

### 区别

- 线程安全

`HashMap` 非同步，多线程环境下需要同步，则使用 `ConcurrentHashMap`。

`HashTable` 同步，使用 `Synchronized` 保证，同步方法，锁的是 `this`。

- 继承关系

`HashMap` 是 `AbstractMap` 的子类，并实现了 `Map` 接口。

`HashTable` 是 `Dictionary` 的子类(JDK1.0提出的)，并实现了 `Map` 接口。

- 是否允许null值

`HashMap` 键和值都可以添加null值，null值作为键只能出现一次(`HashMap`的 `hash()` 方法，null对应的值是0)，null值作为value可以出现多次(即多个键值对应value都是null)。

`HashTable` 键和值都可以不可以为null值，在 `put` 的时候，会对值进行空校验，会调用键的 `hashCode()` 方法。

- 容量Capacity & 扩容机制

`HashMap` 容量默认  $1 < 4$  (16), 每次扩容2倍, 为  $2^n$ 。

`HashTable` 容量默认11, 每次扩容  $2n+1$ 。

- hash值

`HashMap` 对 `key` 值的 `hashCode` 进行了哈希扰动, 有效的降低了 `HashMap` 的哈希冲突。

`HashTable` 直接使用的键的 `hashCode`

- 遍历方式

`HashMap` 的遍历可以通过获取 `EntryIterator`、`ValueIterator`、`KeyIterator` 这些迭代器来遍历。

`HashTable` 的遍历: 对于 `key` 和 `value` 来说, 可以通过 `keys()` 和 `elements()` 方法获取

`Enumeration` 进行遍历, 对于 `Map.Entry` 可以获取 `EntrySet` 再进行遍历。

`Iterator` 支持 fast-fail, 而 `Enumeration` 不支持。

## hashTable相关算法

先了解 `hashTable` 的相关算法, 相比于 `HashMap` 容易理解, 同时后续和 `HashMap` 比较着理解, 可以感受 `HashMap` 设计的巧妙。

底层是数组+单向链表

- 如何确定元素散列下标

`hashTable` 使用的元素的哈希值, 通过取余的方式获取散列下标。关键代码如下

哈希值和 `0x7FFFFFFF` 按位与, 是为了防止负数的出现。和 `tab.length` 除取余得到的是  $(0 - \text{tab.length})$ , 随机散列到数组中。

```
index = (hash & 0x7FFFFFFF) % tab.length;
```

- 扩容 (reHash)

首先了解两个属性 `loadFactor` 和 `threshold` 分别为加载因子和临界值。加载因子默认为0.75, 算上哈希冲突, 所以说 `hashmap` 的存储效率一般不会超过百分之五十。

存在如下关系:

$$threshold = table.length * loadFactor$$

扩容的目的是为了避免频繁哈希冲突, 扩容的时机是当集合元素个数 `count` 大于临界值 `loadfactor` 时, 进行扩容, 扩为2倍加一, `loadfactor` 也随之修改。

扩容的方法是, 数组容量扩充, 新建一个扩充后长度的数组, 将旧数组元素放入新创建的数组。

- 序列化

序列化传输的时候会剔除空节点, 同时这也是必须的, 因为 `hashTable` 不支持 `key` 或 `value` 中任意的 `null` 值。

## hashMap相关算法

hashMap 底层是数组 + 单向链表 + 红黑树

hashMap 是对单线程下 hashTable 的优化。

主要通过 ①位运算 ②截断链表 ③转换红黑树 进行优化

- hashMap的capacity

HashMap 的容量为  $2^n$ 。

通过一系列的移位运算和或运算找到任意数离其最近大于它的 $2^n$ 值，比如 5 ---> 8 、 11 --> 16、33 ---> 64。

基本思想就是：

5 的二进制表示为 0101，将高位第一位不为0的及所有低位置为1，即 0111，转换后加一，即 1000 => 8

11的二进制表示为 1011，将高位第一位不为0的及所有低位置为1，即 1111，转换后加一，即 0001 0000 => 16

- 如何确定散列下标

前提是：HashMap 的容量为 $2^n$ 。散列下标为哈希值和table.length - 1按位与

hashTable 是通过除取余的方式，同样的 hashMap 也是，也可以通过除取余的方式。

但是当数组的长度为 $2^n$ 时候，除取余 = hash() & table.length - 1。

- hashMap的扰动函数

hashTable 对key值的哈希值没做任何处理，会出现一个问题，就是如果key的哈希算法很糟糕的话，会很频繁的出现哈希冲突，通过拉链法解决冲突的话，链表将会拉的很长，且 hashTable 没有截断链表和转化红黑树的操作，如此查询效率将会降低。

所以针对如上 hashMap 做了优化，可以理解为 hashMap 不信任我们写的哈希算法，它自己会做一层处理，基本思想是将高16位和低16位进行按位异或(哈希值是int类型32位)，如此低16位既代表了整个哈希值的特征，在使用扰动后的哈希值来确定散列下标，可有效降低哈希冲突。

- 扩容

hashMap的扩容方法是 `resize()`，扩容为2倍。

此方法不仅仅只做了扩容操作，它还会将链表缩短(缩短一倍)，比如在原数组(数组长度为len)下标j处有一个链表长度为5，经resize方法后，会将此链表拆为两份，分别为长度为2和长度为3的链表，分别放在 新数组  $(2 + len)j$ 处和  $len + j$ 处。

- 转化红黑树

当哈希表的链表过长会影响到查询的效率，所以需要转化为红黑树，利用红黑树的有序性质，可以使得查询效率逼近二分查找。

转化红黑树的条件是：当链表长度等于8且table节点数组长度大于等于64

如果不满足table节点数组长度大于64的话，会进行扩容处理，因为扩容存在缩短链表的操作。

## loadfactory

加载因子为何默认为0.75，在hashMap中容量为 $2^n$ ，乘3/4刚好没有小数位

## 尽量设置初始容量

在创建hashMap的时候建议一次性申请足够多的容量，避免频繁扩容，因为每次扩容都需要重建hash表。

那么初始容量设置多少合适呢？

需要考虑装载因子，hashMap的有效容量为实际容量的0.75，所以设置初始化容量的时候申请大小需要大于实际需要大小。

在hashMap的putAll()方法中就有类型实现：

```
//s 为需要容量大小， ft为实际申请容量大小
float ft = ((float)s / loadFactor) + 1.0F;
```

在guava包下也有类型实现：

```
public static <K extends @Nullable Object, V extends @Nullable Object>
    HashMap<K, V> newHashMapWithExpectedSize(int expectedSize) {
    return new HashMap<>(capacity(expectedSize));
}
static int capacity(int expectedSize) {
    if (expectedSize < 3) {
        checkNonnegative(expectedSize, "expectedSize");
        return expectedSize + 1;
    }
    if (expectedSize < Ints.MAX_POWER_OF_TWO) {
        // This is the calculation used in JDK8 to resize when a putAll
        // happens; it seems to be the most conservative calculation we
        // can make. 0.75 is the default load factor.
        return (int) ((float) expectedSize / 0.75F + 1.0F);
    }
    return Integer.MAX_VALUE; // any large value
}
```

## Stream

可以使用Stream来处理集合，结合Lambda表达式和函数式编程可以编写出简洁、高效的代码

### 特点

- 无存储

Stream不是一种数据结构，它并不存储数据，它只是数据源的一个视图(操作集合的说明书)，数据源可以是一个集合或数组。

- 简洁

`Stream`的特性就是为函数式编程而生，结合函数式编程可以编写出简洁高效的代码。

- 惰式执行

`Stream`上的操作不会立刻执行，而是在被消费的时候才会真正执行

- 可消费性

`Stream`可被消费，且只能被消费一次。一旦遍历过就会失效。想要在此操作必须重写生成流。

如下test2方法stringStream已被遍历过，再次对其操作会报出

```
java.lang.IllegalStateException: stream has already been operated upon or closed
```

错误。

必须如test3生成新的流。

```
@Test
public void test2() {
    final Stream<String> stringStream = Stream.of("1", "2", "3");
    stringStream.filter("2"::equals);
    stringStream.forEach(System.out::println);
}

@Test
public void test3() {
    final Stream<String> stringStream = Stream.of("1", "2", "3");
    final Stream<String> stringStreamFilter = stringStream.filter("2"::equals);
    stringStreamFilter.forEach(System.out::println);
}
```

## Stream操作

对于 `Stream` 流的处理主要有三种关键性操作：创建流、中间操作、最终操作

### 创建流

- 通过集合类的stream方法创建流
- 使用Stream.of(T t)创建流
- 使用Arrays.stream(T[] t)创建流。Stream.of(T ...t)底层就是使用此方式

```
//通过集合类创建流
final Collection<String> strings = Arrays.asList("1", "2", "3");
final Stream<String> stream = strings.stream();

//Stream创建流
final Stream<String> stringStream = Stream.of("1", "2", "3");
final Stream<String> a = Stream.of("a");

//Arrays.stream(T[] t)
final Stream<String> stream1 = Arrays.stream(new String[]{"1", "2", "3"});
```



中间操作

对 `Stream` 做处理，包括过滤、映射、排序等

操作(Stream opration)	说明	参数
filter	过滤	Predicate<? super T> predicate
map	映射	Function<? super T, ? extends R> mapper
limit、skip	限制	long maxSize
sorted	自然排序或指定比较器	Comparator<? super T> comparator
distinct	使用元素的equals去除重复元素	

最终操作

`Stream` 是集合或容器的视图，是对集合或容器的操作描述，但是如果我们要得到结果的话，就需要使用最终操作来将流转化为我们想要的结果。遍历、统计(个数)、转化集合等。

操作	说明	参数
foreach	遍历	Consumer<? super T> action
count	计数	
Collect	转化集合	Collector<? super T, A, R> collector

Stream转化

`Stream`转化为`IntStream`、`LongStream`。。。。

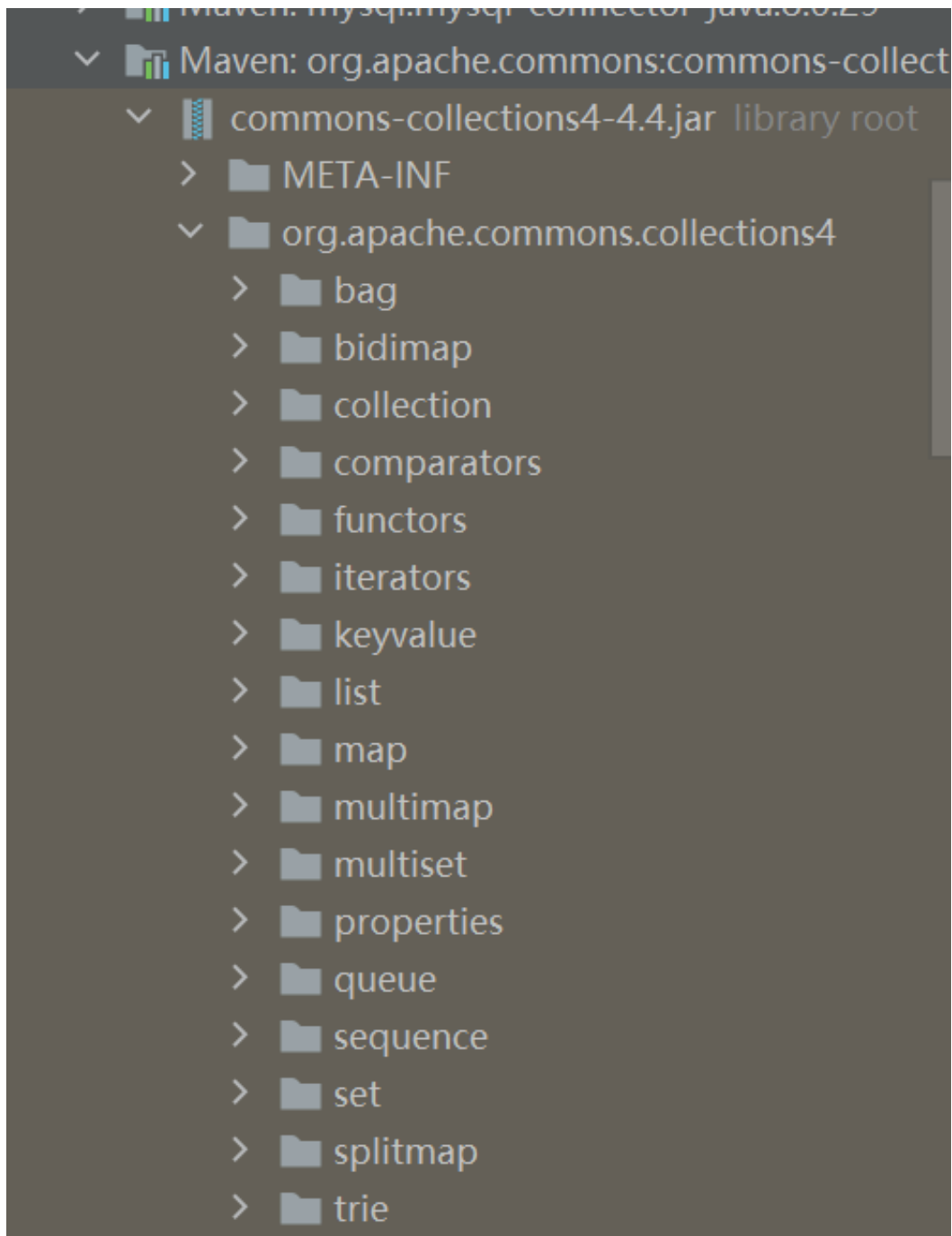
```
final IntStream intStream = stream.mapToInt(StringBuffer::length);
final DoubleStream doubleStream = stream.mapToDouble(StringBuffer::length);
final LongStream longStream = stream.mapToLong(StringBuffer::length);
```

## 集合工具类

许多开源机构为我们提供了操作集合的工具类。

### apache

Apache.commons下的commons-collectionsX包对java集合框架(java collection framework)做扩展。



- Bag - 简化了一个对象在集合中存在多个副本的操作
- BidiMap - 提供双向映射，可通过键查找值，也可以通过值查找键
- Iterators - 方便迭代
- Transforming Decorators 在添加元素时，修改集合元素
- CompositeCollection 需要统一处理多个集合时可用

## Bag

Bag 简化了一个对象存在多个副本的操作。

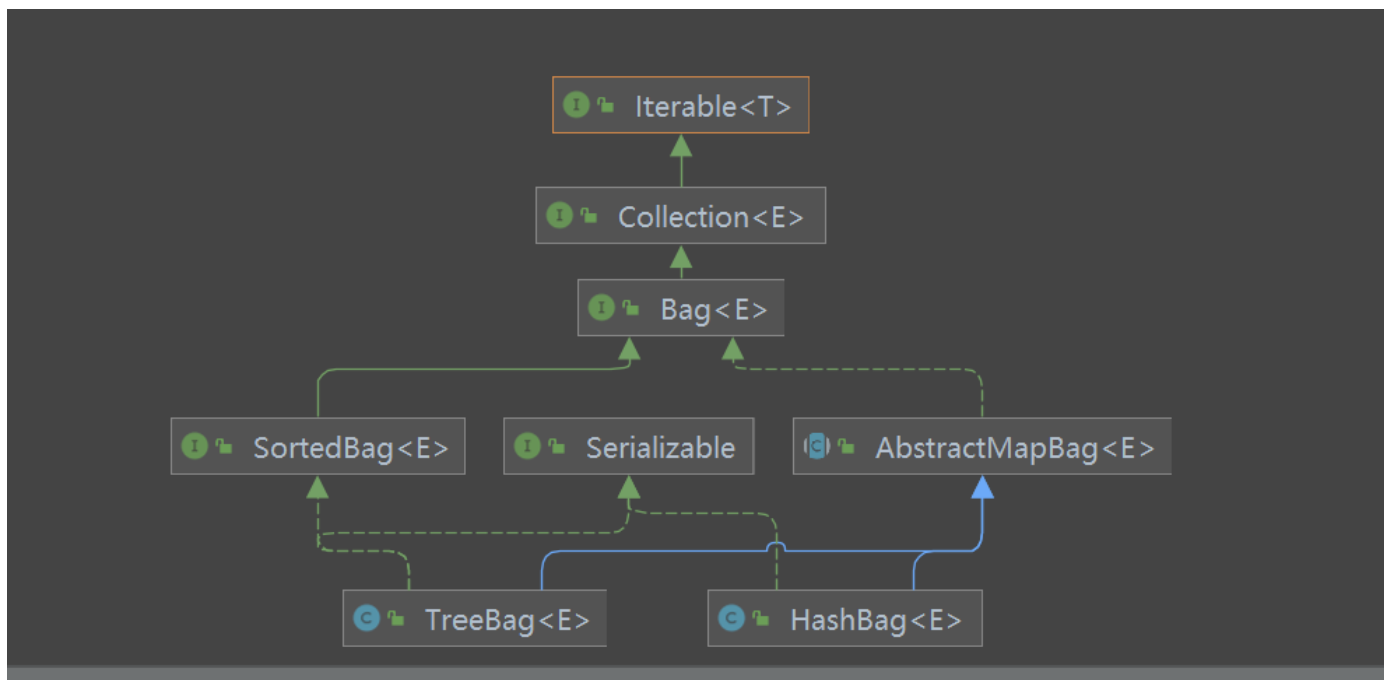
### HashBag & TreeBag

hashBag & TreeBag的继承关系如下图所示：

TreeBag 相较于HashBag多实现一个接口： SortedBag即表现为一个有序的bag。

HashBag其内封装了一个HashMap，key是元素，value是元素个数。

TreeMap其内封装了一个TreeMap，key是元素，value是元素个数。



hashBag&TreeBag的创建。

```
public HashBag() {
    super(new HashMap<E, MutableInteger>());
}

public HashBag(final Collection<? extends E> coll) {
    this();
    addAll(coll);
}

private transient Map<E, MutableInteger> map;
protected AbstractMapBag(final Map<E, MutableInteger> map) {
    super();
    this.map = map;
}
```

hashBag&TreeBag基本使用

```

/**
 * hashBag底层是一个HashMap 元素是key值，添加个数是value
 * 可以通过add(object,nCopies)方法为每个元素添加n个副本
 * 如果add两次则会跟新value值（加一）
 */
StringBuilder sb1 = new StringBuilder("a");
StringBuilder sb2 = new StringBuilder("b");
StringBuilder sb3 = new StringBuilder("c");
Bag<Object> hashBag = new HashBag<>();

hashBag.add(sb1);
hashBag.add(sb1);
hashBag.add(sb2, 3);
hashBag.add(sb3, 3);
System.out.println("Bag+元素个数：" + hashBag.size());
System.out.println("Bag中sb1个数：" + hashBag.getCount(sb1));
String result1 = hashBag.stream().map(Object::toString).collect(Collectors.joining(", ", "[", "]"));
System.out.println("HashBag 内容：" + result1);
//可以使用Collection作为构造方法参数
Bag<Object> hashBag2 = new HashBag<>(Arrays.asList("1", "2", "3"));
String result3 =
hashBag2.stream().map(Object::toString).collect(Collectors.joining(", ", "[", "]"));
System.out.println("hashBag2 内容：" + result3);

/**
 * TreeBag：底层封装了一个TreeMap。
 * 其实可依发现HashBag中的元素是无序的，那么TreeBag就是一个有序的Bag
 */
TreeBag<String> treeBag = new TreeBag<>(Arrays.asList("99", "2", "3", "7"));
String result2 = treeBag.stream().map(Object::toString).collect(Collectors.joining(", ", "[", "]"));
System.out.println("TreeBag 内容：" + result2);

```

## CollectionBag

CollectionBag 没有无参构造，必须依赖一个 Bag 类型的参数。CollectionBag 只是对 Bag 的封装，任何操作实际上操作的是封装的Bag。

```

/**
 * CollectionBag的创建依赖于现有Bag，不可使用Collection作为构造方法参数
 *
 * 其内方法是对Bag的一层封装，实际调用的还是Bag的方法
 */
Bag<Object> collectionBag = CollectionBag.collectionBag(hashBag);
String result4 =
collectionBag.stream().map(Object::toString).collect(Collectors.joining(", ", "[",
"]"));
System.out.println("CollectionBag的内容: " + result4);

Bag<Object> collectionBag2 = CollectionBag.collectionBag(treeBag);
String result5 =
collectionBag2.stream().map(Object::toString).collect(Collectors.joining(", ", "[",
"]"));
System.out.println("CollectionBag2的内容: " + result5);

```

## PredicatedBag & PredicatedSortedBag

`PredicatedBag` 的构造器依赖一个现有Bag和一个 `Predicate`，可以对加入进来的元素进行限制，比如不允许添加空元素。

```

/**
 * 创建PredicatedBag也依赖一个现有Bag，以及一个‘判断器’
 */
PredicatedBag<Object> predicatedBag = PredicatedBag.predicatedBag(collectionBag,
Objects::nonNull);
String result6 =
predicatedBag.stream().map(Object::toString).collect(Collectors.joining(", ", "[",
"]"));
System.out.println("predicatedBag的内容:" + result6);

```

## SynchronizedBag & SynchronizedSortedBag

```

/**
 * SynchronizedBag同步的bag，使用Synchronized同步代码块实现同步。
 * 可以指定锁对象，如果不指定则锁this
 */
SynchronizedBag<Object> synchronizedBag =
SynchronizedBag.synchronizedBag(collectionBag);
SynchronizedSortedBag<Object> synchronizedSortedBag =
SynchronizedSortedBag.synchronizedSortedBag(treeBag);

```

## TransformedBag

对原集合进行转化，一般不会用这个，java8提供的 `Stream API` 有一个Map方法，可以将结果映射。

- `transformingBag` 方法，只会对后面add进来的元素进行转换，而对之前的初始化的不会转化
- `transformedBag`方法，会对后加的以及一开始初始化的都进行转化

```
/**
 * TransformedBag
 */
Bag<Object> bag1 = TransformedBag.transformingBag(hashBag, Object::hashCode);
bag1.add("XX", 3);
String result8 = bag1.stream().map(Objects::toString).collect(Collectors.joining(", ", "[", "]"));
System.out.println("TransformedBag.transformingBag " + result8);

Bag<Object> bag2 = TransformedBag.transformedBag(hashBag, Object::hashCode);
bag2.add("xx", 3);
String result9 = bag2.stream().map(Objects::toString).collect(Collectors.joining(", ", "[", "]"));
System.out.println("TransformedBag.transformingBag " + result9);
```

## UnmodifiableBag

不可修改的Bag。

## BagUtils

协助生成bag，其实调用的就是 `XXXBag.xxxBag()` 方法，每一个Bag类都会有一个静态方法，用于生成Bag。

```
/**
 * BagUtils  Bag工具类 协助生成Bag
 */
final Bag<Object> bag = BagUtils.collectionBag(new HashBag<>());
```

## BidiMap

BidiMap - 提供双向映射，可通过键查找值，也可以通过值查找键

## DualHashBidiMap

双重hashMap。其内封装了两个hashMap，bidimap的put()方法中key-value有任意重复的此条记录会被覆盖

```
/**
 * bidimap
 * - bidimap的put()方法中key-value有任意重复的此条记录会被覆盖
```

```

*/
@Test
public void testBidiMap() {

    final DualHashBidiMap<String, Integer> dualHashBidiMap = new DualHashBidiMap<>();
    dualHashBidiMap.put("a", 1);
    dualHashBidiMap.put("b", 2);
    dualHashBidiMap.put("c", 3);
    dualHashBidiMap.put("d", 3);
    dualHashBidiMap.put("e", 12);
    dualHashBidiMap.put("e", 123);

    System.out.println(dualHashBidiMap.get("a"));
    System.out.println(dualHashBidiMap.getKey(1));
    System.out.println(dualHashBidiMap.getKey(3));
    System.out.println(dualHashBidiMap.values());
    System.out.println(dualHashBidiMap.keySet());
}

```

```

1
a
d
[1, 2, 3, 123]
[a, b, d, e]

```

## iterators

`iterators` 提供了许多迭代包装类使我们很容易迭代集合，并且支持逆向迭代

### ArrayIterator

数组迭代器，接收一个数组、迭代起始下标、迭代终止下标。

只接受迭代数组，因为`next()`方法会调用，本地静态方法 `Array.get(array, index)`

```

public static native Object get(Object array, int index)
    throws IllegalArgumentException, ArrayIndexOutOfBoundsException;

```

简单使用

```

/**
 * ArrayIterator 数组迭代器，接受一个数组、起始下标、终止下标
 */
Iterator<Object> arrayIterator1 = new ArrayIterator<>(Arrays.asList("1", "2",
"3").toArray(), 0, 2);
while (arrayIterator1.hasNext()) {
    System.out.println(arrayIterator1.next());
}

```

## ArrayListIterator

接收一个数组，起始下标，终止下标。

对 `ArrayIterator` 的拓展，支持正向迭代、也支持反向迭代。

```

/**
 * ArrayListIterator
 * 支持正向迭代、逆向迭代
 */
ArrayListIterator<Object> arrayListIterator2 = new ArrayListIterator<>
(Arrays.asList("1", "2", "3", "4", "5", "6").toArray(), 2, 6);
while (arrayListIterator2.hasNext()) {
    System.out.println(arrayListIterator2.next());
}
while (arrayListIterator2.hasPrevious()) {
    System.out.println(arrayListIterator2.previous());
}

```

## BoundedIterator

有边界的迭代器，接收三个参数 `Iterator` 迭代器、`offset` 偏移量、`max` 迭代数量。

如下表示从下标2开始、迭代2个元素，结果是 3、4

```

List<String> list = Arrays.asList("1", "2", "3", "4", "5");
BoundedIterator<String> boundedIterator = new BoundedIterator<>(list.iterator(), 2, 2);
while (boundedIterator.hasNext()) {
    System.out.println(boundedIterator.next());
}

```

## CollectionIterator



```
//比较器，会影响迭代结果
private Comparator<? super E> comparator = null;
//迭代器
private List<Iterator<? extends E>> iterators = null;
//待比较的元素
private List<E> values = null;
//迭代器是否还有值
private BitSet valueSet = null;
private int lastReturned = -1;
```

```
CollatingIterator(final Comparator<? super E> comp, final Iterator<? extends E>[]
iterators)
CollatingIterator(final Comparator<? super E> comp, final Collection<Iterator<? extends
E>> iterators)
```

接收一个或多个迭代器和一个比较器，迭代结果会按一定顺序输出(原集合元素顺序不变)。

```
System.out.println("CollatingIterator");
List<String> list1 = Arrays.asList("5", "4", "1", "2", "3");
List<String> list2 = Arrays.asList("2", "1", "c", "d", "e");
CollatingIterator<String> collatingIterator = new CollatingIterator<>
(String::compareTo, list1.iterator(), list2.iterator());
while (collatingIterator.hasNext()) {
    System.out.print(collatingIterator.next());
}
```

输出:

```
[21]54123[cde]
```

## MapIterator

util包下的map的迭代，如果想迭代key或value，需要借助entry。

```
System.out.println("MapIterator");
final HashMap<Object, Object> map = new HashMap<>(8);
map.put("1", "a");
map.put("2", "b");
map.put("3", "c");
map.put("4", "d");
map.put("5", "e");
map.put("6", "f");
map.put("7", "g");
map.put("8", "h");
//entry迭代器
final Iterator<Map.Entry<Object, Object>> iterator =
    map.entrySet().iterator();
while (iterator.hasNext()) {
```

```

    final Map.Entry<Object, Object> next = iterator.next();
    System.out.println("key值:" + next.getKey() + "value值:" + next.getValue());
}
//key迭代器
map.keySet().iterator();
//value迭代器
map.values().iterator();
final HashMap<Object, Object> hashMap = new HashMap<>(map);
final MapIterator<Object, Object> hashMapIterator = hashMap.mapIterator();
while (hashMapIterator.hasNext()) {
    System.out.println(hashMapIterator.next());
    System.out.println("key值:" + hashMapIterator.getKey() + "value值:" +
hashMapIterator.getValue());
}

```

## CollectionUtils

Apache 的集合工具类，提供很多有用的方法，此工具类在java8之前很有用，但java8的Stream api提供了类似功能，因此许多方法都可以用stream代替。

ignor null

添加元素时忽略null值

```

final List<String> listX = new ArrayList<>(Arrays.asList("1", "2", "3", "4", "5"));
final List<String> listY = new ArrayList<>(Arrays.asList("1", "2", "c", "b", "5"));
System.out.println(CollectionUtils.addIgnoreNull(listX, null));
System.out.println(CollectionUtils.addIgnoreNull(listX, "6"));

```

merge & sort

合并排序，如果不指定比较器，则会自然排序，如果指定比较器则按指定比较器来排序

```

System.out.println("merge and sort");
final List<String> collate1 = CollectionUtils.collate(listX, listY);
System.out.println(collate1);
final List<String> collate2 = CollectionUtils.collate(listX, listY, String::compareTo);
System.out.println(collate2);

```

安全空检查

很多时候对集合遍历的时候，需要对集合进行安全空检查，CollectionUtils为我们提供了一套方式

```

System.out.println("安全空检查");
System.out.println(CollectionUtils.isEmpty(listX));
System.out.println(CollectionUtils.isNotEmpty(listX));

```

交集 并集 外集

```
System.out.println("交集" + CollectionUtils.intersection(listX, listY));
System.out.println("并集" + CollectionUtils.union(listX, listY));
System.out.println("外集" + CollectionUtils.subtract(listX, listY));
```

## 小结

java8的Stream可以适用于大部分的集合操作。

## 交集 外集 并集

```
final List<String> collect1 =
listX.stream().filter(listY::contains).collect(Collectors.toList());
System.out.println("交集" + collect1);
final List<String> collect2 = listX.stream().filter(ele ->
!listY.contains(ele)).collect(Collectors.toList());
System.out.println("外集" + collect2);

System.out.println("Stream 合并集合");
final ArrayList<List<String>> lists = new ArrayList<>();
lists.add(listX);
lists.add(listY);
final List<Object> collect =
lists.stream().flatMap(Collection::stream).collect(Collectors.toList());
System.out.println("并集"+collect);
```

其他的诸如过滤、排序、转换(映射)等Stream都可以

## Arrays.asList(T ...t)

使用此方式创建集合需要注意什么？

- 此方式创建的集合是Arrays的一个子类ArrayList，并未实现增删方法，不可对其进行增删操作。  
会报出 `java.lang.UnsupportedOperationException`
- 可进行修改操作
- 可将其作为参数，构造真正的ArrayList

## 集合中的fail-fast

fail-fast 快速失败，一种一旦检测出系统异常就会立刻上报的机制，此种机制可使系统避免在有安全隐患的情况下继续运行，常用的比如说参数校验。

看一下集合中的fail-fast机制：

常常出现在迭代中，防止下标越界或迭代不完全，以ArrayList为例

expectedModCount此属性在获取迭代器的时候会赋值为modCount，如果在迭代时我们通过修改集合改变modeCount则会报出此异常

```
final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```

对集合进行增删操作会触发fail-fast机制

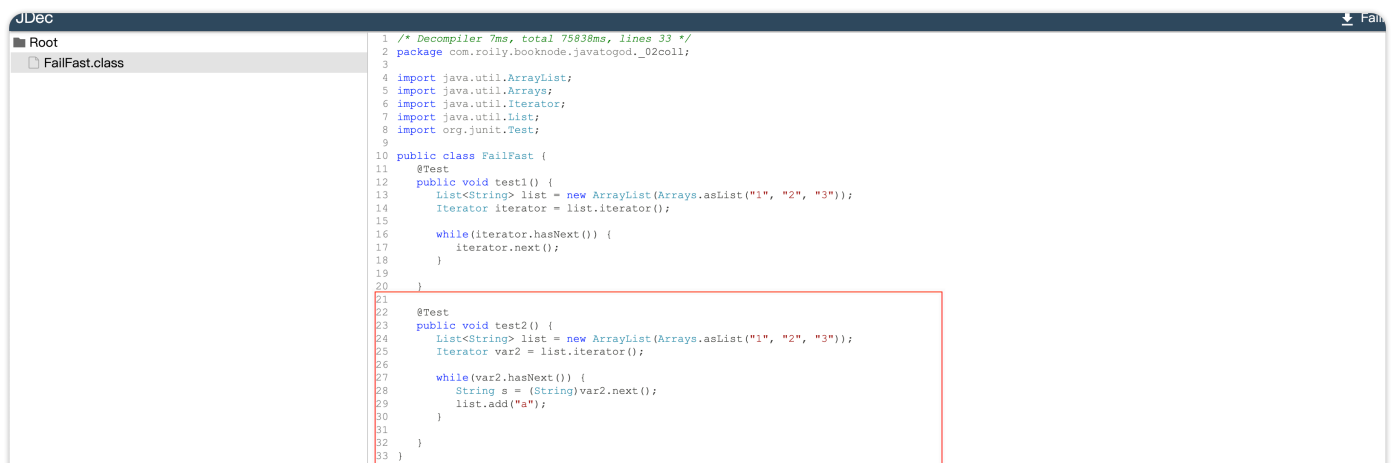
```
final List<String> list = new ArrayList<>(Arrays.asList("1", "2", "3"));
final Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    //list.remove("1");
    //list.add("1");
    iterator.next();
}
```

foreach也会触发fail-fast机制，因为其底层就是使用迭代器迭代的

如果安装了阿里代码规约插件的话，那么已经帮你提示出来了

```
final List<String> list = new ArrayList<>(Arrays.asList("1", "2", "3"));
for (String s : list) {
    list.add("a");
}
```

反编译看一下字节码



## 集合中的fail - safe

为了避免fail-fast机制，可以使用采用fail-safe机制的集合类，这样的类在对集合进行操作的时候不会直接操作集合内容，而是通过拷贝一份，在拷贝的内容上操作，最后需要同步更改，则同步更改。

### CopyOnWriteArrayList

这就是一个fail-safe集合类。

其内部的add、remove、set等操作都使用ReentrantLock保证同步，任何操作都是在拷贝对象数组上进行，最后再替换原集合对象数组。

COW集合的迭代器:其内部包含了一个源集合对象数组的快照、副本、拷贝(snapshot)，任何迭代都是在此对象数组上完成的。并且由于COW在修改对象数组的时候，COW都会拷贝一份，所以并不会影响迭代器中的拷贝所指向的对象数组

使用COW代替ArrayList就不会发生CME异常

```
CopyOnWriteArrayList<String> cowList = new CopyOnWriteArrayList<>
(Arrays.asList("1", "2", "3"));
for (String s : cowList) {
    if ("1".equals(s)) {
        cowList.remove(s);
    }
}
```

但是也造成了一个问题：我们对集合的修改，修改对象数组，是对迭代器不可见的，因为在集合修改的时候，会使用System.arraycopy()方法生成一个新的对象数组。

```
CopyOnWriteArrayList<String> cowList = new CopyOnWriteArrayList<>
(Arrays.asList("1", "2", "3"));
Iterator<String> iterator = cowList.iterator();
//fail-safe 集合修改
for (String s : cowList) {
    if ("1".equals(s)) {
        cowList.remove(s);
    }
}
//已经修改
System.out.println(cowList);
//但对迭代器不可见
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

打印结果：

```
[2, 3]
1
2
3
```

## 特点

- copy-on-write, 写时复制, 所有修改操作在快照上操作
- 同步, 使用ReentrantLock
- 写时加锁, 避免拷贝出多个副本, 导致并发写
- 读时不加锁, 读写分离。会导致弱一致性问题: 读取的不是最新数据

## 循环中remove

fail-fast机制出现于迭代器中修改集合操作, 触发fail-fast机制, 导致并发异常。

- for 普通for循环
- iterator 的remove方法 (主要思想就是在修改modCount的同时修改expectedModCount)
- fail-safe集合 (fail-safe集合的迭代器一般不支持remove方法, 直接使用集合的remove方法)
- 增强for: fail-fast机制在next方法中调用, 在修改集合后避免使用next方法即可

```
List<String> list = new ArrayList<>(Arrays.asList("1", "2", "3"));
for (String s : list) {
    if ("1".equals(s)){
        list.remove(s);
    }
    break;
}
System.out.println(list);
```

# IO

文件操作是编程一部分, 学习一下IO流。

## 字符流&字节流

从名称来看区别在于流的传输方式: 字节 or 字符。

有了字节流为何还需要字符流?

字符流可以认为是字节流 + 编码方式。编码方式指导字节流如何处理字节, 将其组合成字符。原因就在于方便操作, 对于中文可能不同的编码方式得到的字节数据是不同的, 那么使用字节流读取可能出现乱码的情况, 那么有了字符流可以指定编码, 指导字节流读几个字节作为一个汉字。

## 位、字节、字符

- Bit 最小二进制单位，0或1.
- Byte 字节，1 Byte = 8 Bit，取值 [-128,127]
- Char 字符，1Char = 16Bit，人能直观认识的最小单位，取值 [0,2<sup>16</sup>-1]

## 字节流

操作字节，用于读取单个字节或字节数组，直接对文件进行操作，无需缓冲区（读出来的数据直接就可以用）。

主要操作类是：InputStream和OutputStream的子类

InputStream常用子类：

- FileInputStream：文件输入流，用于读取文件信息到内存中。
- ByteArrayInputStream：字节数组输入流
- ObjectInputStream: 对象输入流

OutputStream常用子类：

- FileOutputStream:文件输出流，用于将数据输出到文件中
- ByteArrayOutputStream:字节数组输出流
- ObjectOutputStream:对象输出流

## FileInputStream & FileOutputStream

用于读取文件信息到内存和将内存中的数据输出到文件

查看一下FileInputStream api

```
//文件描述实例，由java创建，不用我们创建
private final FileDescriptor fd;
//文件路径
private final String path;
//用于读取、写入、映射和操作文件的通道。与FileInputStream唯一关联。底层，不用管
private FileChannel channel = null;
//对象锁，阻塞io，使用Synchronized关键字阻塞
private final Object closeLock = new Object();
//资源是否关闭
private volatile boolean closed = false;

//使用文件名创建一个文件输入流会调用FileInputStream(File file)方法
public FileInputStream(String name) throws FileNotFoundException;
public FileInputStream(File file) throws FileNotFoundException;
//打开流，本地方法由c\c++编写，无需主动调用，构造方法已经调用，且是私有方法
private native void open0(String name) throws FileNotFoundException;
private void open(String name) throws FileNotFoundException {open0(name);}
//从流中读取一个字节，声明式异常，需要主动处理
public int read() throws IOException;
private native int read0() throws IOException
```

```

//从流中读取off开始读取len个字符到字符数组中指定下标处,并返回读取长度。一般来说从0开始
private native int readBytes(byte b[], int off, int len) throws IOException;
public int read(byte b[]) throws IOException {
    return readBytes(b, 0, b.length);
}
//跳过指定长度字符, 返回跳过长度。移动指针
public long skip(long n) throws IOException {
    return skip0(n);
}
private native long skip0(long n) throws IOException;
//剩余字节数
public int available() throws IOException {
    return available0();
}
private native int available0() throws IOException;
//关闭流, 必须关闭资源
public void close() throws IOException;

```

查看一下FileOutputStream api

```

//是否追加文件内容。默认false, 即覆盖
private final boolean append;
public FileOutputStream(String name) throws FileNotFoundException;
public FileOutputStream(String name, boolean append) throws FileNotFoundException;
public FileOutputStream(File file) throws FileNotFoundException;
public FileOutputStream(File file, boolean append) throws FileNotFoundException;
//open open0
//写入文件, 字节byte可直接转化为int, 安全不会溢出, 没有负数
private native void write(int b, boolean append) throws IOException;
private native void writeBytes(byte b[], int off, int len, boolean append) throws
IOException;

```

常用操作: 这里使用try with resource语法趟, 编译器会自动帮我们关闭资源, 可以反编译查看

```

@Test
public void inputStreamTest1() {
    String filePath = "/Users/rolyfish/Desktop/MyFoot/myfoot/foot/testfile/test.txt";
    try (FileInputStream fin = new FileInputStream(filePath)) {
        //跳过指定长度字符
        final long skip = fin.skip(3L);
        System.out.println(skip);
        final byte[] bytes = new byte[5];
        //从1开始读取4个字符, 放入字符数组指定下标处
        final int read = fin.read(bytes, 1, 4);
        System.out.println(read);
        for (byte aByte : bytes) {
            System.out.println(Character.valueOf((char) aByte));
        }
    }
}

```



```

    }
} catch (IOException e) {
    e.printStackTrace();
}
}
@Test
public void outputStreamTest1() {
    String filePath = "/Users/rolyfish/Desktop/MyFoot/myfoot/foot/testfile/test.txt";
    try (final FileOutputStream fop = new FileOutputStream(filePath, true)) {
        fop.write("可爱".getBytes());
        fop.write("abc".getBytes());
        //刷新流，将此之前的所有数据给操作系统，让操作系统写入硬件设备
        fop.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

## ByteArrayInputStream & ByteArrayOutputStream

字节数组输入输出流。内部组合一个字节数组，用于缓冲数据。

```

@Test
public void byteArrayInputStreamTest1() {
    String filePath = "/Users/rolyfish/Desktop/MyFoot/myfoot/foot/testfile/test.txt";
    try (final FileOutputStream fop = new FileOutputStream(filePath, true);
        final ByteArrayInputStream byteArrayInputStream = new
ByteArrayInputStream("abcd".getBytes(StandardCharsets.UTF_8))) {
        System.out.println((char)byteArrayInputStream.read());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
@Test
public void byteArrayOutputStreamTest1() {
    String filePath = "/Users/rolyfish/Desktop/MyFoot/myfoot/foot/testfile/test.txt";
    try (final FileOutputStream fop = new FileOutputStream(filePath, true);
        final ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream(1024)) {
        //写入到字节数组
        byteArrayOutputStream.write("可可爱爱".getBytes());
        //一次性写入到，另一个输出流
        byteArrayOutputStream.writeTo(fop);
        //刷新流，将此之前的所有数据给操作系统，让操作系统写入硬件设备
        fop.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

## ObjectInputStream & ObjectOutputStream

对象输入输出流，一般用于序列化操作，又称为序列化流和反序列化流。ObjectOutputStream用于将对象序列化输出到文件中，持久化保存，ObjectInputStream用于将对象从文件中读出来。

一般用于数据传输，或当某个对象实例生命周期已经结束，但是需要保存其状态，以便下次直接反序列化恢复的情况。

首先作为字节流，它拥有字节流的所有相关操作

```
@Test
public void objectOutputStreamTest1() {
    String filePath =
"/Users/rolyfish/Desktop/MyFoot/myfoot/foot/testfile/objectTest.txt";
    try (final ObjectOutputStream objectOutputStream = new ObjectOutputStream(new
FileOutputStream(filePath, false))) {
        //写入到字节数组
        objectOutputStream.write("xxx".getBytes());
        //刷新流，将此之前的所有数据给操作系统，让操作系统写入硬件设备
        objectOutputStream.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

作为对象流，设计目的就是操作对象：

- class类必须实现Serializable接口，否则会报出 `java.io.NotSerializableException` 异常
- 序列号serialVersionUID对应唯一类，不可随意修改，在反序列化时会去匹配。如果修改则报出 `java.io.InvalidClassException` 异常
- 被transient修饰的属性在序列化时会被忽略

例子：

```
@Data
@Accessors(chain = true)
class Person implements Serializable {
    private static final long serialVersionUID = -8861126921891657698L;

    String str1;
    transient String str2;
    final String str3 = "123";
    static String str4;
    int age;
    Date birthday;
}

@Test
public void objectInputStreamTest1() {
    String filePath =
"/Users/rolyfish/Desktop/MyFoot/myfoot/foot/testfile/objectTest.txt";
}
```

```

        try (final ObjectInputStream objectInputStream = new ObjectInputStream(new
FileInputStream(filePath))) {
            final Person o = (Person) objectInputStream.readObject();
            System.out.println(o);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

@Test
public void objectOutputStreamTest2() {
    String filePath =
"/Users/rolyfish/Desktop/MyFoot/myfoot/foot/testfile/objectTest.txt";
    try (final ObjectOutputStream objectOutputStream = new ObjectOutputStream(new
FileOutputStream(filePath, false))) {
        final Person person = new Person();
        person.setStr1("str1")
            .setStr2("str2")
            .setAge(1)
            .setBirthday(Calendar.getInstance().getTime());
        objectOutputStream.writeObject(person);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

## 字符流

操作字符，需要缓冲区，操作Reader、Writer的子类

## 字节流与字符流转化

java io包下除了字节流与字符流以外，还包含一组字节流-字符流转化流。

InputStreamReader，是Reader的子类属于字符流，可以将输入的字节流转化为输入的字符流。

OutputStreamWriter，是Writer的子类属于字符流，可以将输出的字符流转换为输出的字节流。

## 输入输出流

输入输出流是相对于参考系来说的，此参考系为存储数据的介质，往介质中存数据则为输入流，从介质中读出数据则为输出流。

比如：

将文件中数读出来，存到内存中，则为输入流，使用FileInputStream、FileReader

将内存中的数据输出到文件中，则为输出流，使用FileOutputStream、FileWriter

# 反射

反射式java为程序员提供的强大机制，赋予程序可以在运行期间，知道任意类的所有属性和方法，调用或修改任意对象的属性和方法的能力。

## Class类

Class类用于封装加载到jvm中的类(包括接口和类)。当一个类被装载进jvm就会生成一个与之唯一对应的Class对象，通过这个Class对象我们就知道此类的所有信息。