名词解释

JDK

Java Develement Kit，提供给Java开发人员使用，其中包含了Java的开发工具，也包括了JRE。

JRE

Java Runtime Environment，Java运行环境，包含Java虚拟机和Java程序所需的核心类库等，如果想要运行一个开发好的Java程序，计算机中只需安装JRE即可。

JDK、JRE、JVM 三者关系

- JDK = JRE + 开发工具集（例如Javac编译工具等）
- JRE = JVM + Java SE标准类库

垃圾回收

Java 消除了程序员自动回收的责任；

但还是会内存泄漏和内存溢出；

变量

基本数据类型

整型

整数类型：byte、short、int、long

- Java各整数类型有固定的表数范围和字段长度，不受具体OS的影响，以保证java程序的可移植性。
- java的整型常量默认为 int 型，声明long型常量须后加‘l’或‘L’
- java程序中变量通常声明为int型，除非不足以表示较大的数，才使用long

类　型	占用存储空间	表数范围
byte	1字节=8bit位	-128 ~ 127
short	2字节	-2^{15} ~2^{15}-1
int	4字节	-2^{31} ~ 2^{31}-1 (约21亿)
long	8字节	-2^{63} ~ 2^{63}-1

500MB　1MB = 1024KB　1KB= 1024B　B= byte ? bit?
bit: 计算机中的最小存储单位。byte:计算机中基本存储单元。

浮点类型

浮点类型：float、double

- 与整数类型类似，Java 浮点类型也有固定的表数范围和字段长度，不受具体操作系统的影响。
- 浮点型常量有两种表示形式：
  - 十进制数形式：如：5.12　　512.0f　　.512　(必须有小数点)
  - 科学计数法形式:如：5.12e2　　512E2　　100E-2
- float:单精度，尾数可以精确到7位有效数字。很多情况下，精度很难满足需求。
  double:双精度，精度是float的两倍。通常采用此类型。
- **Java 的浮点型常量默认为double型，声明float型常量，须后加‘f’或‘F’。**

类　型	占用存储空间	表数范围
单精度float	4字节	-3.403E38 ~ 3.403E38
双精度double	8字节	-1.798E308 ~ 1.798E308

> 数值范围可具体参考操作系统的浮点数存储方式。

字符类型

字符类型：char

- char 型数据用来表示通常意义上“字符”(2字节)
- Java中的所有字符都使用Unicode编码，故一个字符可以存储一个字母，一个汉字，或其他书面语的一个字符。
- 字符型变量的三种表现形式：
  - 字符常量是用单引号(‘ ’)括起来的单个字符。例如：char c1 = 'a'; char c2 = '中'; char c3 = '9';
  - Java中还允许使用转义字符‘\’来将其后的字符转变为特殊字符型常量。例如：char c3 = ‘\n’; // '\n'表示换行符
  - 直接使用 Unicode 值来表示字符型常量：‘\uXXXX’。其中，XXXX代表一个十六进制整数。如：\u000a 表示 \n。
- char类型是可以进行运算的。因为它都对应有Unicode码。

> 单引号里面只能有一个字符
>
> 在JAVA中，对char类型字符运行时，直接当做ASCII表对应的整数来对待。

布尔类型

- boolean类型数据只允许取值true和false，无null。
  - 不可以使用0或非 0 的整数替代false和true，这点和C语言不同。
  - Java虚拟机中没有任何供boolean值专用的字节码指令，Java语言表达所操作的boolean值，在编译之后都使用java虚拟机中的int数据类型来代替：true用1表示，false用0表示。———《java虚拟机规范 8版》

注意事项

- 高精度强转低精度会报编译错误：不兼容类型

```
long a = 123; // 整型默认为int，定义long需要加L或l，123L
float b = 12.2; // 浮点型默认为 double，定义float需要加F或f，12.2F
```

引用数据类型

字符串类型

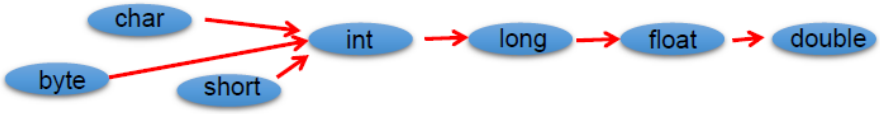
- String不是基本数据类型，属于引用数据类型
- 使用方式与基本数据类型一致。例如：String str = “abcd”;
- 一个字符串可以串接另一个字符串，也可以直接串接其他类型的数据。例如：

```
str = str + “xyz” ;
int n = 100;
str = str + n;
```

基本数据类型转换

小转大

- 自动类型转换：容量小的类型自动转换为容量大的数据类型。数据类型按容量大小排序为：



- 有多种类型的数据混合运算时，系统首先自动将所有数据转换成容量最大的那种数据类型，然后再进行计算。
- byte,short,char之间不会相互转换，他们三者在计算时首先转换为int类型。
- boolean类型不能与其它数据类型运算。
- 当把任何基本数据类型的值和字符串(String)进行连接运算时(+)，基本数据类型的值将自动转化为字符串(String)类型。


```
byte a = 1;
short b = a + a; // 编译不通过，只要做运算哪怕是相同类型相加，也会转换为int，
short c = a; // 赋值不报错
```

```
System.out.println('a' + 'b'); // 195，char类型会先转为int类型做加法运算
System.out.println('a' + "b"); // ab，有字符串时，加号为连接
System.out.println("a" + "b"); // ab
System.out.println('a' + 'b' + "c"); // 195c，先加法再连接
```

强制类型转换

- 自动类型转换的逆过程，将容量大的数据类型转换为容量小的数据类型。使用时要加上强制转换符：()，但可能造成精度降低或溢出,格外要注意。
- 通常，字符串不能直接转换为基本类型，但通过基本类型对应的包装类则可以实现把字符串转换成基本类型。

➢如： String a = “43”; int i = Integer.parseInt(a);

➢boolean类型不可以转换为其它的数据类型。

```
double a = 1.1;
int b = (int)a;
```

```
int a = 128;
byte b = (byte)a; // b输出为-128，由于强转后精度损失，剩下10000000，而且byte类型的第一位是符号位，所以输出-128
```

内存解析

堆

此内存区域的唯一目的就是存放对象实例（new 出来的结构）

栈（虚拟机栈）

存储局部变量

> int[] nums = new int[5];
>
> 其中nums为局部变量存储在栈中，new int[5]会在堆中开辟5个int存储空间。

方法区

用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

类

三大特性

封装性

能不能调用的问题

继承性

能不能获取的问题

功能

- 减少代码冗余，提高了代码的复用性
- 便于功能的扩展
- 为之后多态性的使用提供了前提

特征

子类A继承了父类B之后，子类A中就获取了父类B中声明的所有属性和方法。

> - 父类中声明为private的属性和方法，由于封装性，子类不能直接调用，可以通过其他方式调用
> - 如 private int age; 子类中不能直接调用age，但是可以调用getAge()、setAge()
> - 子类是可以获取（不是调用）父类的private属性和方法
> - 父类的静态方法和属性也会被子类继承，但是不能重写，只能覆盖，也就是子类父类共享静态方法和属性

多态性

- 对象的多态性：父类的引用指向子类的对象（或子类的对象赋给父类的引用）

```
Person p = new Man(); // Man为Person的子类
```

- 多态的使用：当调用子类父类同名同参数的方法时，实际执行的是子类重写父类的方法---虚拟方法调用

```
p.eat();// 实际执行的是Man类里重写的eat()方法
p.earnMoney(); // earnMoney为Man类特有的类，会编译错误，因为p只能调用p定义的方法
// 总之，p只能调用Person（父类）声明的方法，如果调用了子类重写的方法，则实际执行的是子类的方法；
// 编译看左边（父类定义的方法）；运行看右边（运行子类重写的方法，子类一般会重写父类的方法，父类的方法，子类都会继承到）
```

- 多态的前提：类的继承关系；方法的重写（一般会重写，不重写，即子父类方法的功能一样）
- 例子

```
public class Animal {
    public void func(Animal animal) { // 参数相当于 Animal animal = New Dog()
        animal.eat();
    }
    public void eat(){System.out.println("动物吃");}
}
public class Dog extends Animal {
    public void eat(){System.out.println("狗吃");}
}
public class Cat extends Animal {
    public void eat(){System.out.println("猫吃");}
}

Animal animal = new Animal();
animal.fund(animal); // 动物吃
animal.fund(new Dog()); // 狗吃
animal.fund(new Cat()); // 猫吃
// 减少代码的冗余，若没有多态性，则在Animal类中，需要对每个子类都分别声明一个方法
```

- 多态性不适用于属性。谁调的属性就是谁的属性

- 多态性是运行时行为，不是编译时行为。只有在运行时，才知道父类的引用指向哪个子类的对象。
- 虚拟方法：对象在调用与子类同名同参数的方法时，编译时认为是调用父类的方法，而在运行时真正执行的是子类的方法。

权限修饰符

Java权限修饰符public、protected、(缺省)、private置于类的成员定义前，用来限定对象对该类成员的访问权限。

修饰符	类内部	同一个包	不同包的子类	同一个工程
private	Yes			
(缺省)	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

对于class的权限修饰只可以用public和default(缺省)。
- public类可以在任意地方被访问。
- default类只可以被同一个包内部的类访问。

构造器

也称为构造方法、constructor

- 用于创建对象、给对象进行初始化。
- 一个类中可以有多个构造器，重载。
- 若没有定义构造器，则系统提供默认的空参构造器；反之，系统不提供。

static

static修饰属性

- 静态变量随着类的加载而加载。可以通过"类.静态变量"的方式进行调用
- 静态变量的加载要早于对象的创建。
- 由于类只会加载一次，则静态变量在内存中也只会存在一份：存在**方法区的静态域**中。

static修饰方法：静态方法

- 随着类的加载而加载，可以通过"类.静态方法"的方式进行调用
- 静态方法中，只能调用静态的方法或属性

  > 静态方法先于对象加载，而非静态方法或属性只有等创建对象后才会加载出来，所以没创建对象时非静态方法和属性还不存在与内存中，因而不能调用非静态方法和属性。

- 非静态方法中，既可以调用非静态的方法或属性，也可以调用静态的方法或属性

static注意点

- 在静态的方法内，不能使用this关键字、super关键字(对象没加载，自然也就没有this)
- 关于静态属性和静态方法的使用，从生命周期的角度去理解。

> 开发中，如何确定一个属性是否要声明为static的？
>
> - 属性是可以被多个对象所共享的，不会随着对象的不同而不同的。
> - 类中的常量也常常声明为static
>
> 开发中，如何确定一个方法是否要声明为static的？
>
> - 操作静态属性的方法，通常设置为static的
> - 工具类中的方法，习惯上声明为static的。 比如：Math、Arrays、Collections

final

修饰类

此类不能被其他类所继承，比如：String类、System类、StringBuffer类

修饰方法

表明此方法不可以被重写，比如：Object类中getClass()

修饰变量

此时的"变量"就称为是一个常量

- 修饰属性：可以考虑赋值的位置有显式初始化、代码块中初始化、构造器中初始化
- 修饰局部变量：尤其是使用final修饰形参时，表明此形参是一个常量。当我们调用此方法时，给常量形参赋一个实参。一旦赋值以后，就只能在方法体内使用此形参，但不能进行重新赋值。
- static final 用来修饰属性：全局常量
- final若修饰的是引用数据类型变量，则变量不能重新赋值，但是变量的非final属性可以修改

```
public class FinalTest {
    final int WIDTH = 0; // 显示初始化
    final int LEFT;
    final int RIGHT;
    {
        LEFT=1;// 代码块中初始化
    }
    public FinalTest(){
        RIGHT=1;// 构造器中初始化
    }
    public FinalTest(int n){
        RIGHT=n;// 构造器中初始化，多个构造器的话就每个构造器中都要初始化，因为构造器只会执行一个
    }
    // 对象加载出来后，final非静态属性必须得初始化（不支持默认初始化），而不能修改否则编译错误
}
```

this关键字

- 在方法内部使用，表示这个方法所属对象的引用
- 在构造器内部使用，表示该构造器正在初始化的对象

```java
public class Triangle {
    private double base;
    private double height;

    public Triangle() {}
    public Triangle(double base, double height) {
        this(); // 调用无参的构造器，必须放在首行
        this.base = base;
        this.height = height;
    }
}
```

- 使用this访问属性和方法时，如果在本类中找不到，会从父类中查找；

super关键字

- 用于访问父类中定义的属性，成员方法
- 用于在子类构造器中调用父类的构造器
  - 子类中所有的构造器默认都会访问父类中空参数的构造器
  - 当父类中没有空参数的构造器时，子类的构造器必须通过this(参数列表)或super(参数列表)语句指定调用本类或者父类中相应的构造器。同时，只能“二选一”，且必须**放在构造器的首行**。
  - 如果子类构造器中既未显示调用父类或本类的构造器，且父类中没有无参的构造器，则编译出错。

package关键字

- 包帮助管理大型软件系统：将功能相近的类划分到同一个包中 。 比如：MVC 的设计模式
- 包可以包含类和子包，划分项目层次，便于管理
- 解决类命名冲突的问题
- 控制访问权限

重写

- 子类重写的方法**必须**和父类被重写的方法具有相同的名称、参数列表；
- 子类重写的方法的返回值类型只能是父类被重写的方法的返回值类型或子类；
  - 如果父类被重写方法的返回值类型是基本数据类型，则子类重写方法的返回值类型只能一样
- 子类重写的方法使用的访问权限**不能小于**父类被重写的方法的访问权限；
  - 子类不能重写父类中声明为private权限的方法
- 子类方法抛出的异常不能大于父类被重写方法的异常

> 子类与父类中同名同参数的方法必须**同时声明**为**非static**的（重写），或者**同时声明**为**static**的（不是重写，两个声明）。因为static方法是属于类的，子类无法覆盖父类的方法。
>
> 如果方法在父类中是静态方法，那么子类不能声明非静态的同名同参数方法。

代码块

- 用来初始化类、对象
- 代码块如果有修饰的话，只能使用static.

静态代码块

- 内部可以有输出语句
- 随着类的加载而执行,而且只执行一次
- 作用：初始化类的信息
- 如果一个类中定义了多个静态代码块，则按照声明的先后顺序执行
- 静态代码块内只能调用静态的属性、静态的方法，不能调用非静态的结构

非静态代码块

- 内部可以有输出语句
- **每创建一个对象，就执行一次非静态代码块。且先于构造器执行。**
- 作用：可以在创建对象时，对对象的属性等进行初始化
- 如果一个类中定义了多个非静态代码块，则按照声明的先后顺序执行
- 非静态代码块内可以调用静态的属性、静态的方法，或非静态的属性、非静态的方法

注意事项

- 成员变量有默认值；局部变量没有默认值，需要显式地赋值初始化；
- 非static成员变量加载到堆中，局部变量加载到栈中；
- 子类对象实例化过程
    - 当我们通过子类的构造器创建子类对象时，我们一定会直接或间接的调用其父类的构造器，进而调用父类的父类的构造器，直到调用了java.lang.Object类中空参的构造器为止。正因为加载过所有的父类的结构，所以才可以看到内存中有父类中的结构，子类对象才可以考虑进行调用。
    - 虽然创建子类对象时，调用了父类的构造器，但是自始至终就创建过一个对象，即为new的子类对象
- 类加载顺序
    1. 首先在main方法中,调用了B类的构造方法.
    2. 由于B类有父类,因此先加载A类.
    3. 加载A类的静态代码块
    4. 加载B类的静态变量
    5. 加载B类的静态代码块
    6. 加载A类的普通变量
    7. 加载A类的构造方法
    8. 加载B类的普通变量
    9. 加载B类的构造方法
- 对属性可以赋值的位置：
    1. 默认初始化
    2. 显式初始化/在代码块中赋值 （同级别，看代码中写的位置，自上而下）
    3. 构造器中初始化
    4. 有了对象以后，可以通过"对象.属性"或"对象.方法"的方式，进行赋值

abstract

不能用 abstract 修饰变量、代码块、构造器；
不能用 abstract 修饰 私有方法、 静态方法、final的方法、final的类。

abstract修饰类

- 此类不能实例化
- 抽象类中一定有构造器，便于子类实例化时调用（涉及：子类对象实例化的全过程）
- 开发中，都会提供抽象类的子类，让子类对象实例化，完成相关的操作

abstract修饰方法：抽象方法

- 抽象方法只有方法的声明，没有方法体
- **包含抽象方法的类，一定是一个抽象类。反之，抽象类中可以没有抽象方法的。**
- 若子类不是抽象类，则子类必须重写父类中的所有的抽象方法，子类方可实例化
- 若子类没有重写父类中的所有的抽象方法，则此子类也是一个抽象类，需要使用abstract修饰

```
public abstract class AbstractTest {
    public AbstractTest() {}
    public abstract void func ();
}
```

接口

成员

- JDK7及以前：只能定义全局常量和抽象方法
  - 全局常量：**public static final**的
  - 抽象方法：**public abstract**的

  > 书写时，可以省略不写，默认有

- JDK8：除了定义全局常量和抽象方法之外，还可以定义静态方法、默认方法（略）
  - 接口中定义的静态方法，只能通过接口来调用。

    ```
    public static void func() {方法体} // 定义
    ```

  - 通过实现类的对象，可以调用接口中的默认方法。

    ```
    public default void func() {方法体} // 定义
    ```

    > 如果实现类重写了接口中的默认方法，调用时，仍然调用的是重写以后的方法。

  - 如果子类(或实现类)继承的父类和实现的接口中声明了同名同参数的默认方法，那么子类在没有重写此方法的情况下，默认调用的是父类中的同名同参数的方法。-->类优先原则

    > 属性没有类优先原则，会编译错误，需要super.属性 和 接口名.属性 来区分调用。

  - 如果实现类实现了多个接口，而这多个接口中定义了同名同参数的默认方法，那么在实现类没有重写此方法的情况下，报错。-->接口冲突。
    这就需要我们必须在实现类中重写此方法

注意事项

- 接口中不能定义构造器，意味着接口不可以实例化
- 接口与接口之间可以继承，而且可以多继承
- Java类可以实现多个接口 --->弥补了Java单继承性的局限性
  - 格式：class AA extends BB implements CC,DD,EE
- Java开发中，接口通过让类去实现(implements)的方式来使用.
  - 如果实现类覆盖了接口中的所有抽象方法，则此实现类就可以实例化
  - 如果实现类没有覆盖接口中所有的抽象方法，则此实现类仍为一个抽象类
- 接口的具体使用，体现多态性

内部类

Java中允许将一个类A声明在另一个类B中，则类A就是内部类，类B称为外部类

实际开发中用的不多，能看懂源码即可

成员内部类

（静态、非静态）

在局部内部类的方法中如果调用局部内部类所声明的方法中的局部变量的话，要求此局部变量声明为final的。

> jdk 7及之前版本：要求此局部变量显式的声明为final的
>
> jdk 8及之后的版本：可以省略final的声明

作为外部类的成员

- 调用外部类的结构

```
Person.this.func(); // Person为外部类，成员内部类中调用外部类的方法的完整形式，也可以直接调用func()
```

- 可以被static修饰，不可以调用非静态属性和方法
- 可以被4种不同的权限修饰

作为一个类

- 类内可以定义属性、方法、构造器等
- 可以被final修饰，表示此类不能被继承。言外之意，不使用final，就可以被继承
- 可以被abstract修饰

创建实例

```
// Person 为外部类
Person.Dog dog = new Person.Dog(); // 静态成员内部类的实例化

Person p = new Person();
Person.Bird bird = p.new Bird(); // 非静态成员内部类的实例化
```

局部内部类

(方法内、代码块内、构造器内)

Object类

Java中所有类的直接或间接父类

equals()

可以查看源码。Java中提供的类大部分都重写了equals方法。

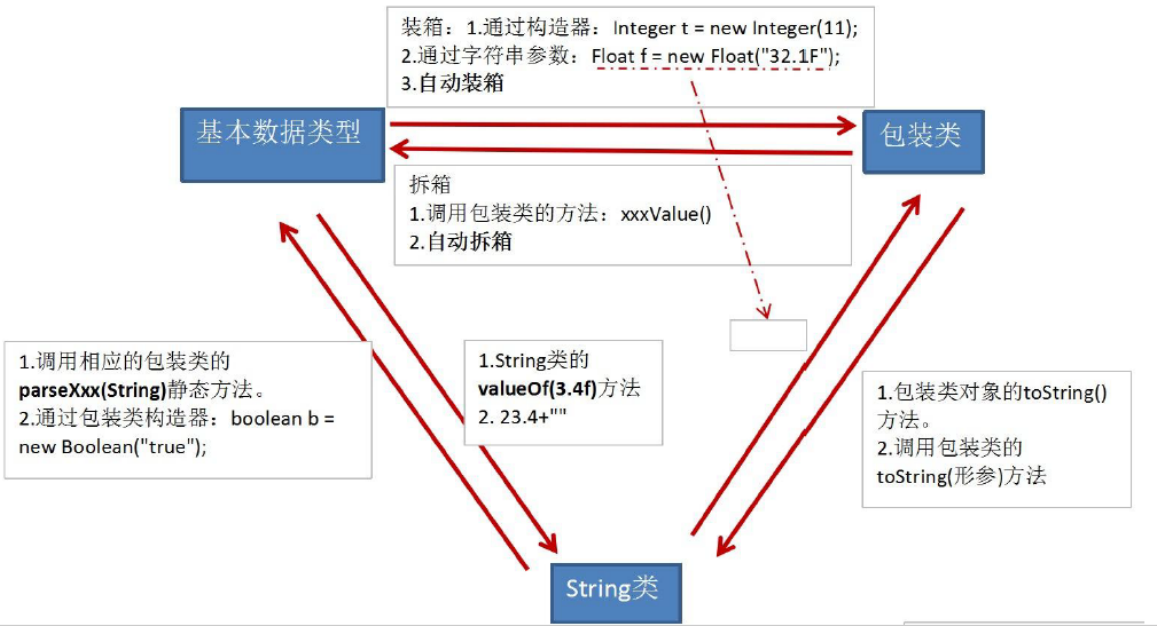
x.equals(null) 永远是false

equals与==的区别：

- ==如果是基本数据类型之间的比较则是比较数值，如果是引用数据类型之间的比较则是比较地址
- equals是Object类里定义的方法，内部实现也是用==运算符。通常我们需要重写equals方法来满足自己的需求。

Wrapper 包装类

基本数据类型、包装类、String三者之间的转换



自动装箱

```
Integer i = 10; // Integer i = new Integer(10);
```

自动拆箱

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

> 自动装箱与自动拆箱是JDK5.0的新特性

面试题

```
Object o1 = true ? new Integer(1) : new Double(2.0);
System.out.println(o1); // 1.0   三元运算符JDK5之前要求两个表达式类型需要一致，5以后有了自动拆箱与自动装箱，就可以比较了。其中int 会自动提升为double，所以输出1.0
```

```
Integer i = new Integer(1);
Integer j = new Integer(1);
System.out.println(i == j);//false

//Integer内部定义了IntegerCache结构，IntegerCache中定义了Integer[],
//保存了从-128~127范围的整数。如果我们使用自动装箱的方式，给Integer赋值的范围在
//-128~127范围内时，可以直接使用数组中的元素，不用再去new了。目的：提高效率

Integer m = 1;
Integer n = 1;
System.out.println(m == n);//true

Integer x = 128;//相当于new了一个Integer对象
Integer y = 128;//相当于new了一个Integer对象
System.out.println(x == y);//false
```

异常

Error

- 栈溢出：java.lang.StackOverflowError 递归
- 堆溢出：java.lang.OutOfMemoryError

```
Integer[] arr = new Integer[1024*1024*1024]; // OOM
```

Exception

常见异常

- java.lang.Exception:可以进行异常的处理
  - 编译时异常(checked)
    - IOException
    - FileNotFoundException
    - ClassNotFoundException
  - 运行时异常(unchecked,RuntimeException)
    - NullPointerException
    - ArrayIndexOutOfBoundsException
    - ClassCastException
    - NumberFormatException
    - InputMismatchException：输入不匹配，如Scanner要求是int是，输入字符串
    - ArithmeticException：算数异常，如除0

线程

程序：为完成特定任务的一组指令的集合，即一段代码。

进程：一个运行中的程序。是一个动态的过程，有生命周期。

线程：是进程中的一个执行流程，一个进程中可以运行多个线程。每个线程都各自加载**虚拟机栈**和**程序计数器**，所有线程共用一个**方法区**和**堆**。

> 一个Java应用程序java.exe，其实至少有三个线程main()主线程，gc()垃圾回收线程，异常处理线程。当然如果发生异常，会影响主线程 。
>
> 多线程执行时，线程之间争夺cpu的执行权

创建线程方式

继承Thread类

常用方法

- start()：
  - 启动当前线程
  - 调用当前线程的run()
  - 同个对象的start()只能调用一次

```
// start()的源码
public synchronized void start() {
    if (threadStatus != 0) // 判断线程是否启动过，若已经启动过，则抛出异常
        throw new IllegalThreadStateException();

    group.add(this);

    boolean started = false;
    try {
        start0();   // 启动线程，native方法
        started = true; // 判断线程是否启动成功
    } finally {
        try {
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
        }
    }
}
```

- run()：通常需要重写Thread类中的此方法，将创建的线程要执行的操作声明在此方法中
- currentThread()：静态方法，返回执行当前代码的线程
- getName()：获取当前线程的名字
- setName()：设置当前线程的名字
- yield()：释放当前cpu的执行权（但当前线程还可能会获取下次的cpu执行权）

- join()：在线程a中调用线程b的join(),此时线程a就进入阻塞状态，直到线程b完全执行完以后，线程a才结束阻塞状态。
- stop()：已过时。当执行此方法时，强制结束当前线程。
- sleep(long millitime)：静态方法，让当前线程"睡眠"指定的millitime毫秒。在指定的millitime毫秒时间内，当前线程是阻塞状态。
- isAlive()：判断当前线程是否存活

线程的优先级

MAX_PRIORITY：10

MIN _PRIORITY：1

NORM_PRIORITY：5 -->默认优先级

getPriority():获取线程的优先级

setPriority(int p):设置线程的优先级

> 说明：高优先级的线程要抢占低优先级线程cpu的执行权。但是只是从概率上讲，高优先级的线程高概率的情况下被执行。并不意味着只有当高优先级的线程执行完以后，低优先级的线程才执行。

实现Runnable接口

1. 创建一个实现了Runnable接口的类
2. 实现类去实现Runnable中的抽象方法：run()
3. 创建实现类的对象
4. 将此对象作为参数传递到Thread类的构造器中，创建Thread类的对象
5. 通过Thread类的对象调用start()

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {}
    }; // 匿名子类
Thread thread = new Thread(runnable);
thread.start();
```

实现Callable接口

JDK 5.0新增

```
//1.创建一个实现Callable的实现类
class NumThread implements Callable{
    //2.实现call方法，将此线程需要执行的操作声明在call()中
    @Override
    public Object call() throws Exception {}
}

public class ThreadNew {
    public static void main(String[] args) {
        //3.创建Callable接口实现类的对象
        NumThread numThread = new NumThread();
        //4.将此Callable接口实现类的对象作为传递到FutureTask构造器中，创建FutureTask的对象
        FutureTask futureTask = new FutureTask(numThread);
```

```
        //5.将FutureTask的对象作为参数传递到Thread类的构造器中，创建Thread对象，并调用start()
        new Thread(futureTask).start();

        try {
            //6.获取Callable中call方法的返回值
            //get()返回值即为FutureTask构造器参数Callable实现类重写的call()的返回值。
            Object sum = futureTask.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

如何理解实现Callable接口的方式创建多线程比实现Runnable接口创建多线程方式强大？

1. call()可以有返回值的。
2. call()可以抛出异常，被外面的操作捕获，获取异常的信息
3. Callable是支持泛型的

使用线程池

JDK 5.0新增

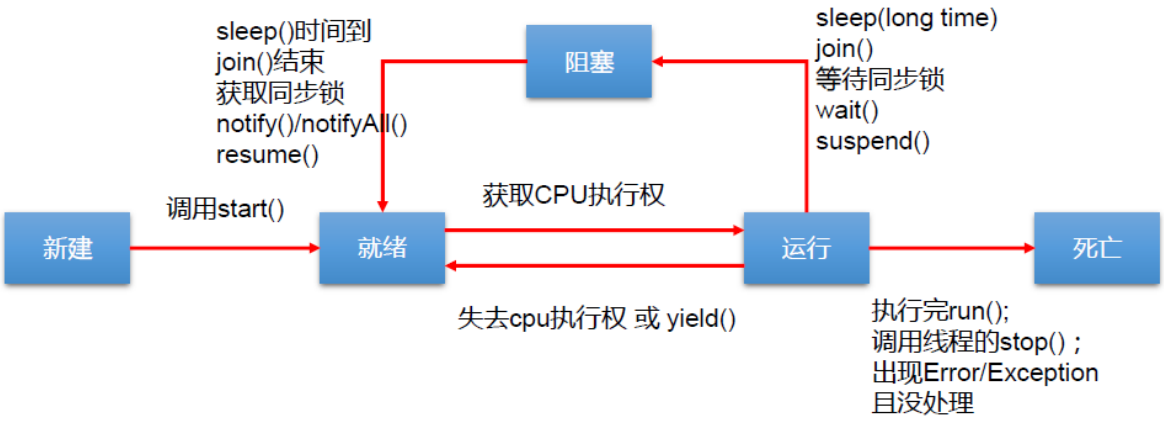
```
public class NumberThread implements Runnable {}
public class ThreadPool {
    public static void main(String[] args) {
        //1. 提供指定线程数量的线程池
        ExecutorService service = Executors.newFixedThreadPool(10);
        ThreadPoolExecutor service1 = (ThreadPoolExecutor) service;
        //设置线程池的属性
//        System.out.println(service.getClass());
//        service1.setCorePoolSize(15);
//        service1.setKeepAliveTime();

        //2.执行指定的线程的操作。需要提供实现Runnable接口或Callable接口实现类的对象
        service.execute(new NumberThread());//适合适用于Runnable
        service.execute(new NumberThread1());//适合适用于Runnable

//        service.submit(Callable callable);//适合使用于Callable
        //3.关闭连接池
        service.shutdown();
    }

}
```

思路

提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。

优点

生命周期



线程同步

解决了线程的安全问题。---好处

操作同步代码时，只能有一个线程参与，其他线程等待。相当于是一个单线程的过程，效率低。 ---局限性

同步代码块

```
synchronized(同步监视器){
      //需要被同步的代码
}
```

1.操作共享数据的代码，即为需要被同步的代码。 -->不能包含代码多了，也不能包含代码少了。

2.共享数据：多个线程共同操作的变量。比如：ticket就是共享数据。

3.同步监视器，俗称：锁。任何一个类的对象，都可以充当锁。

要求：多个线程必须要共用同一把锁。

> 在实现Runnable接口创建多线程的方式中，我们可以考虑使用this充当同步监视器。
>
> 在继承Thread类创建多线程的方式中，可以使用 "类名.class"来充当同步监视器

同步方法

如果操作共享数据的代码完整的声明在一个方法中，我们不妨将此方法声明同步的。

- 同步方法仍然涉及到同步监视器，只是不需要我们显式的声明。

默认同步监视器

- 非静态的同步方法，同步监视器是：this
- 静态的同步方法，同步监视器是：当前类本身

```
// 正确的方式
public synchronized void func() {} // 实现Runnable接口的推荐方式，同步监视器（锁）是 this
public static synchronized void func() {} // 继承Thread类的推荐方式，同步监视器是"类名.class"
```

Lock

JDK5.0新增

```
private ReentrantLock lock = new ReentrantLock(); // 定义lock
public void func() {
    lock.lock(); // 上锁
    // 同步代码
    lock.unlock(); // 解锁
}
```

> 注意：如果同步代码有异常，要将unlock() 写入 finally 语句块

优先使用顺序

Lock > 同步代码块（已经进入了方法体，分配了相应资源） > 同步方法（在方法体之外）

线程通信

线程通信的例子：使用两个线程打印 1-100。线程1, 线程2 交替打印

涉及到的三个方法：

- wait():一旦执行此方法，当前线程就进入阻塞状态，并释放同步监视器。
- notify():一旦执行此方法，就会唤醒被wait的一个线程。如果有多个线程被wait，就唤醒优先级高的那个。
- notifyAll():一旦执行此方法，就会唤醒所有被wait的线程。

说明：

- 1.wait()，notify()，notifyAll()三个方法**必须使用在同步代码块或同步方法**中。
- 2.wait()，notify()，notifyAll()三个方法的**调用者必须是同步代码块或同步方法中的同步监视器。**否则，会出现IllegalMonitorStateException异常
- 3.wait()，notify()，notifyAll()三个方法是**定义在java.lang.Object类**中。

面试题

比较创建线程的两种方式

开发中：优先选择实现Runnable接口的方式

原因：

1. **实现的方式没有类的单继承性的局限性**
2. 实现的方式更适合来处理多个线程有共享数据的情况。

联系：public class Thread implements Runnable （Thread类也实现了Runnable接口）

相同点：两种方式都需要重写run(),将线程要执行的逻辑声明在run()中。

synchronized 与 Lock的异同

相同：

- 二者都可以解决线程安全问题

不同：

- synchronized机制在执行完相应的同步代码以后，自动的释放同步监视器
- Lock需要手动的启动同步（lock()），同时结束同步也需要手动的实现（unlock()）

sleep() 和 wait()的异同？

1.相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态。

2.不同点：

1）两个方法声明的位置不同：Thread类中声明sleep()，Object类中声明wait()

2）调用的要求不同：sleep()可以在任何需要的场景下调用。 wait()必须使用在同步代码块或同步方法中

3）关于是否释放同步监视器：如果两个方法都使用在同步代码块或同步方法中，**sleep()不会释放锁，wait()会释放锁。**

常用类

String类

字符串常量池在jdk6、jdk7、jdk8都有所不同

特性

- String声明为final的，不可被继承
- String实现了Serializable接口：表示字符串是支持序列化的；实现了Comparable接口：表示String可以比较大小
- String内部定义了final char[] value用于存储字符串数据
- 不可变性
    1. 当对字符串重新赋值时，需要重写指定内存区域赋值，不能使用原有的value进行赋值。

    ```
    2. 当对现有的字符串进行连接操作时，也需要重新指定内存区域赋值，不能使用原有的value进行赋值。
    3. 当调用String的replace()方法修改指定字符或字符串时，也需要重新指定内存区域赋值，不能使用原有的value进行赋值。
    ```

- 通过**字面量的方式（区别于new）**给一个字符串赋值，此时的字符串值声明在字符串常量池中。
- 字符串常量池中是不会存储相同内容的字符串的。

```
String s1 = "javaEE";
String s2 = "hadoop";

String s3 = "javaEEhadoop";
```

```
String s4 = "javaEE" + "hadoop"; // 常量折叠，在.class字节码处显示的就是"javaEEhadoop"
String s5 = s1 + "hadoop"; // new StringBuffer(s1).append("hadoop").toString(); 新对象
String s6 = "javaEE" + s2;

System.out.println(s3 == s4);//true
System.out.println(s3 == s5);//false
System.out.println(s3 == s6);//false
System.out.println(s5 == s6);//false

String s8 = s6.intern();//返回值得到的s8使用的常量值中已经存在的“javaEEhadoop”
System.out.println(s3 == s8);//true
```

结论：

- 常量与常量的拼接结果在常量池。且常量池中不会存在相同内容的常量。
- 只要其中有一个是变量，结果就在堆中。
- 如果拼接的结果调用intern()方法，返回值就在常量池中

重点理解

```
String temp = "hello";
String str = new String(temp); // 在堆上
```

当一个.java文件被编译成.class文件时，和所有其他常量一样，每个字符串字面量都通过一种特殊的方式被记录下来。

当一个.class文件被加载时（**注意加载发生在初始化之前**），JVM在.class文件中寻找**字符串字面量**(上面的第一行)。当找到一个时，JVM会检查是否有相等的字符串在常量池中存放了堆中引用。**如果找不到，就会在堆中创建一个对象，然后将它的引用存放在池中的一个常量表中。**

一旦一个字符串对象的引用在常量池中被创建，这个字符串在程序中的所有字面量引用都会被常量池中已经存在的那个引用代替。

> 即是说，常量池存放的实际是引用（即地址），指向了堆中创建的对象，这个对象可以通过intern()访问。

这段代码就创建了2个`String`对象，`temp`指向在常量池中的，`str`指向堆上的，而`str`内部的`char value[]`则指向常量池中指向的String对象的`char value[]`

```
Class<String> clazz = String.class;
Constructor<String> cons = clazz.getDeclaredConstructor(String.class);
String s1 = cons.newInstance("abc");
String s2 = cons.newInstance("abc"); // 相当于new了两个String对象
System.out.println(s1 == s2);
System.out.println("*****************"); // false

Field value = clazz.getDeclaredField("value"); // 获取底层的char[] value
value.setAccessible(true);
System.out.println(value.get(s1) == value.get(s2)); // true，证明了就算是两个不同的String对象，底层的char[] value是共用的（共用常量池中所指向的String对象的char[] value属性）。
```

面试题

- String s = new String("abc");方式创建对象，在内存中创建了几个对象？

  两个:一个是堆空间中new结构，另一个是char[]对应的常量池中的数据："abc"（如果常量池已经存在"abc"，就不用创建了）

- String、StringBuffer、StringBuilder三者的异同？

  String:不可变的字符序列；底层使用char[]存储

  StringBuffer:可变的字符序列；线程安全的，效率低；底层使用char[]存储

  StringBuilder:可变的字符序列；jdk5.0新增的，线程不安全的，效率高；底层使用char[]存储

- 对比String、StringBuffer、StringBuilder三者的效率：
  从高到低排列：StringBuilder > StringBuffer > String

StringBuffer/StringBuilder

```
1 StringBuffer sb1 = new StringBuffer();//char[] value = new char[16];底层创建了一个长度是16的数组
```

问题1. System.out.println(sb2.length());//3，输出count，实际元素数量

问题2. 扩容问题:如果要添加的数据底层数组盛不下了，那就需要扩容底层的数组。

默认情况下，扩容为原来容量的`2倍 + 2`，同时将原有数组中的元素复制到新的数组中。

指导意义：开发中建议大家使用：StringBuffer(int capacity) 或 StringBuilder(int capacity)

> String类型在修改操作中会不断新建副本，占用内存资源

比较器

Comparable接口

自然排序

- 像String、包装类等实现了Comparable接口，重写了compareTo(obj)方法，给出了比较两个对象大小的方式。
- 像String、包装类重写compareTo()方法以后，进行了从小到大的排列
- 重写compareTo(obj)的规则：
  - 如果当前对象this大于形参对象obj，则返回正整数，
  - 如果当前对象this小于形参对象obj，则返回负整数，
  - 如果当前对象this等于形参对象obj，则返回零。
- 对于自定义类来说，如果需要排序，我们可以让自定义类实现Comparable接口，重写compareTo(obj)方法。
  在compareTo(obj)方法中指明如何排序

Comparator接口

定制排序，临时性的，用于参数传递，如：`Arrays.sort(arr, new Comparator(){});`

重写规则同上。

枚举类

自定义枚举类

```java
class Season{
    //1.声明Season对象的属性:private final修饰
    private final String seasonName;
    private final String seasonDesc;

    //2.私有化类的构造器,并给对象属性赋值
    private Season(String seasonName,String seasonDesc){
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }

    //3.提供当前枚举类的多个对象：public static final的
    public static final Season SPRING = new Season("春天","春暖花开");
    public static final Season SUMMER = new Season("夏天","夏日炎炎");
    public static final Season AUTUMN = new Season("秋天","秋高气爽");
    public static final Season WINTER = new Season("冬天","冰天雪地");

    //4.其他诉求1：获取枚举类对象的属性
    public String getSeasonName() {
        return seasonName;
    }
    public String getSeasonDesc() {
        return seasonDesc;
    }
    //4.其他诉求1：提供toString()
    @Override
    public String toString() {
        return "Season{" +
                "seasonName='" + seasonName + '\'' +
                ", seasonDesc='" + seasonDesc + '\'' +
                '}';
    }
}
```

使用enum关键字定义枚举类

定义的枚举类默认继承于java.lang.Enum类

```java
enum SeasonEnum {
    //1.提供当前枚举类的对象，多个对象之间用","隔开，末尾对象";"结束
    SPRING("春天", "春风又绿江南岸"),
    SUMMER("春天", "春风又绿江南岸"),
    AUTUMN("春天", "春风又绿江南岸"),
    WINTER("春天", "春风又绿江南岸");
    //2.声明Season对象的属性:private final修饰
    private final String seasonName;
    private final String seasonDesc;
    //3.私有化类的构造器,并给对象属性赋值
    private SeasonEnum(String seasonName, String seasonDesc) {
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }
    //4.其他诉求1：获取枚举类对象的属性
```

```
    public String getSeasonName() {
        return seasonName;
    }
    public String getSeasonDesc() {
        return seasonDesc;
    }
    // Enum类已经重写toString()，输出变量名字
}

```

Enum类常用方法

- values()：返回枚举类型的对象数组。该方法可以很方便地遍历所有的枚举值。
- valueOf(String str)：可以把一个字符串转为对应的枚举类对象。要求字符串必须是枚举类对象的“名字”。
- toString()：返回当前枚举类对象常量的名称。

实现接口的枚举类

```
interface Info {
    void show();
}
enum Season1 implements Info{
    //1.在每个对象后重写show()，这样保证了每个对象重写方法的独特性
    SPRING("春天","春暖花开"){
        @Override
        public void show() {
            System.out.println("春天在哪里？");
        }
    }
}
```

注解

jdk 5.0 新增的功能

Annotation 其实就是代码里的特殊标记，这些标记可以在编译、类加载、运行时被读取、并执行相应的处理。通过使用 Annotation，程序员可以在不改变原有逻辑的情况下, 在源文件中嵌入一些补充信息。

jdk 提供的4种元注解

元注解：对现有的注解进行解释说明的注解 // 修饰注解

- Retention：指定所修饰的 Annotation 的生命周期：SOURCE\CLASS（默认行为）\RUNTIME
  只有声明为RUNTIME生命周期的注解，才能通过反射获取。
- Target：用于指定被修饰的 Annotation 能用于修饰哪些程序元素
  出现的频率较低
- Documented：表示所修饰的注解在被javadoc解析时，保留下来。
- Inherited：被它修饰的 Annotation 将具有继承性。

数组

特点

- 一旦初始化以后，其长度就确定了。
- 数组一旦定义好，其元素的类型也就确定了。我们也就只能操作指定类型的数据了。

  比如：String[] arr;int[] arr1;Object[] arr2;

缺点

- 一旦初始化以后，其长度就不可修改。
- 数组中提供的方法非常有限，对于添加、删除、插入数据等操作，非常不便，同时效率不高。
- 获取数组中实际元素的个数的需求，数组没有现成的属性或方法可用
- 数组存储数据的特点：有序、可重复。对于无序、不可重复的需求，不能满足。

集合

Collection接口

> 单列集合，用来存储一个一个的对象
>
> 向Collection接口的实现类的对象中添加数据obj时，要求obj所在类要重写equals()。因为一些方法会用到equals()，如contains()、remove()、
>
> List接口中用于排序。
>
> Set接口中需要根据equals()来实现不可重复性

- boolean contains(Object obj)：通过元素的`equals`方法来判断是否是同一个对象
- boolean remove(Object obj)：通过元素的`equals`方法判断是否是要删除的那个元素。只会删除找到的第一个元素
- boolean removeAll(Collection coll)：从当前集合中移除coll1中所有的元素。取当前集合的差集
- boolean retainAll(Collection c)：把交集的结果存在当前集合中，不影响c
- Object[] toArray()：转成对象数组
- Iterator iterator()：返回`Iterator`迭代器对象，用于集合遍历；集合对象每次调用iterator()方法都得到一个全新的迭代器对象，即重新开始遍历。
  - foreach：JDK5.0新增，遍历集合的底层调用`Iterator`完成操作。

```
String[] arr = new String[]{"MM","MM","MM"};

//方式一：普通for赋值
for(int i = 0;i < arr.length;i++){
    arr[i] = "GG";
} // 改变arr指向的值，本身做修改

//方式二：增强for循环
for(String s : arr){
    s = "GG";
} // 不改变arr指向的值，因为foreach是 新建个变量，并赋值给变量，String变量修改值，不影响原来的值，可以看反编译的结果，底层仍然是迭代器

for(int i = 0;i < arr.length;i++){
```

```
    System.out.println(arr[i]);
}
```

List接口

存储有序的、可重复的数据。 -->“动态”数组

ArrayList

作为List接口的主要实现类；线程不安全的，效率高；底层使用Object[] elementData存储

```
new ArrayList().remove(int index);
new ArrayList().remove(Object obj);
remove(2); // 删除的是索引为2的对象
remove(new Integer(2)); // 删除的是Integer对象
```

new ArrayList().remove()

jdk 7情况下

- ArrayList list = new ArrayList();//底层创建了长度为10的Object[]数组elementData
- list.add(123);//elementData[0] = new Integer(123);
- list.add(11);//如果此次的添加导致底层elementData数组容量不够，则扩容。
- 默认情况下，扩容为原来的容量的1.5倍，同时需要将原有数组中的数据复制到新的数组中。

  `int newCapacity = oldCapacity + (oldCapacity >> 1);`
- 结论：建议开发中使用带参的构造器：ArrayList list = new ArrayList(int capacity)

jdk 8中ArrayList的变化：

- ArrayList list = new ArrayList();//底层Object[] elementData初始化为{}。并没有创建长度为10的数组
- list.add(123);//第一次调用add()时，底层才创建了长度10的数组，并将数据123添加到elementData[0]
- 后续的添加和扩容操作与jdk 7 无异。

LinkedList

对于频繁的插入、删除操作，使用此类效率比ArrayList高；底层使用双向链表存储

> 创建LinkedList对象时，内部初始化了 Node类型的 first、last，分别存储首节点和尾节点

Vector

> 作为List接口的古老实现类；线程安全的，效率低；底层使用Object[] elementData存储

jdk7和jdk8中通过Vector()构造器创建对象时，底层都创建了长度为10的数组。

在扩容方面，默认扩容为原来的数组长度的2倍。

List接口常用方法

- void add(int index, Object ele):在index位置插入ele元素
- boolean addAll(int index, Collection eles):从index位置开始将eles中的所有元素添加进来
- Object get(int index):获取指定index位置的元素
- int indexOf(Object obj):返回obj在集合中首次出现的位置
- int lastIndexOf(Object obj):返回obj在当前集合中末次出现的位置
- Object remove(int index):移除指定index位置的元素，并返回此元素
- Object set(int index, Object ele):设置指定index位置的元素为ele
- List subList(int fromIndex, int toIndex):返回从fromIndex到toIndex位置的子集合

Set接口

Set接口中没有额外定义新的方法，使用的都是Collection中声明过的方法。

存储无序的、不可重复的数据

1. 无序性：不等于随机性。存储的数据在底层数组中并非按照数组索引的顺序添加，而是根据数据的哈希值决定的。
2. 不可重复性：保证添加的元素按照（hashcode()判断先）equals()判断时，不能返回true.即：相同的元素只能添加一个。

添加数据的过程，以HashSet为例：

我们向HashSet中添加元素a,首先调用元素a所在类的hashCode()方法，计算元素a的哈希值，此哈希值接着通过某种算法计算(底层是&运算)出在HashSet底层数组中的存放位置（即为：索引位置），判断数组此位置上是否已经有元素：

- 如果此位置上没有其他元素，则元素a添加成功。 --->情况1
- 如果此位置上有其他元素b(或以链表形式存在的多个元素），则比较元素a与元素b的hash值：
    - 如果hash值不相同，则元素a添加成功。--->情况2
    - 如果hash值相同，进而需要调用元素a所在类的equals()方法：
        - equals()返回true,元素a添加失败
        - equals()返回false,则元素a添加成功。--->情况3

对于添加成功的情况2和情况3而言：元素a 与已经存在指定索引位置上数据以链表的方式存储。

jdk 7 :新来的元素放到数组中，指向原来的元素。

jdk 8 :原来的元素在数组中，指向新来的元素

总结：七上八下

HashSet底层：数组+链表的结构。

> 要求：向Set(主要指：HashSet、LinkedHashSet)中添加的数据，其所在的类一定要重写hashCode()和equals()。
>
> 要求：重写的hashCode()和equals()尽可能保持一致性：相等的对象必须具有相等的散列码（哈希值）。
>
> 重写两个方法的小技巧：对象中用作 equals() 方法比较的 Field，都应该用来计算 hashCode 值。

HashSet

底层用 `HashMap` 实现

作为Set接口的主要实现类；线程不安全的；可以存储null值

LinkedHashSet

底层用`LinkedHashMap`实现，类似LinkedList

作为HashSet的子类；遍历其内部数据时，**可以按照添加的顺序遍历**。对于频繁的遍历操作，LinkedHashSet效率高于HashSet。

LinkedHashSet（实际是LinkedHashMap）作为HashSet的子类，在添加数据的同时，每个数据还维护了两个引用，记录此数据前一个数据和后一个数据。

优点：对于频繁的遍历操作，LinkedHashSet效率高于HashSet

TreeSet

底层用红黑树实现，查询速度快

可以按照添加对象的指定属性，进行排序。

- 向TreeSet中添加的数据，要求是相同类的对象；
- 两种排序方式：自然排序（实现Comparable接口）和 定制排序（Comparator）；

```
// 定制排序
// 定义Comparator接口实现类对象
Comparator com = new Comparator() {
    @Override
    public int compare(Object o1, Object o2) {}
};
// 作为参数传入TreeSet对象
TreeSet set = new TreeSet(com);
```

- 自然排序中，比较两个对象是否相同的标准为：compareTo()返回0，不再是equals()；
- 定制排序中，比较两个对象是否相同的标准为：compare()返回0，不再是equals()。

面试题

```
HashSet set = new HashSet();
Person p1 = new Person(1001,"AA");
Person p2 = new Person(1002,"BB");
set.add(p1);
set.add(p2);
p1.name = "CC";
set.remove(p1); // hash值计算后没有相同值
System.out.println(set);
// [Person{id=1002, name='BB'}, Person{id=1001, name='CC'}]
set.add(new Person(1001,"CC")); // 虽然内容相同，但是由于p1计算的哈希值是基础"AA"的，所以哈希值仍然不同，添加成功
System.out.println(set);
// [Person{id=1002, name='BB'}, Person{id=1001, name='CC'}, Person{id=1001, name='CC'}]
set.add(new Person(1001,"AA")); // 计算的哈希值与之前的p1是相同的，但是后面用equals()比较后返回false，所以添加成功
System.out.println(set);
// [Person{id=1002, name='BB'}, Person{id=1001, name='CC'}, Person{id=1001, name='CC'}, Person{id=1001, name='AA'}]
```

Map接口

双列集合，用来存储键值对的数据

常用方法

- 添加、删除、修改操作：
  - Object put(Object key,Object value)：将指定key-value添加到(或修改)当前map对象中
  - void putAll(Map m):将m中的所有key-value对存放到当前map中
  - Object remove(Object key)：移除指定key的key-value对，并返回value
  - void clear()：清空当前map中的所有数据
- 元素查询的操作：
  - Object get(Object key)：获取指定key对应的value
  - boolean containsKey(Object key)：是否包含指定的key
  - boolean containsValue(Object value)：是否包含指定的value
  - int size()：返回map中key-value对的个数
  - boolean isEmpty()：判断当前map是否为空
  - boolean equals(Object obj)：判断当前map和参数对象obj是否相等
- 元视图操作的方法：
  - Set keySet()：返回所有key构成的Set集合
  - Collection values()：返回所有value构成的Collection集合
  - Set entrySet()：返回所有key-value对构成的Set集合

HashMap

作为Map的主要实现类；线程不安全的，效率高；存储null的key和value

HashMap的底层：

数组+链表 （jdk7及之前）

HashMap map = new HashMap():

在实例化以后，底层创建了长度是16的一维数组Entry[] table。

...可能已经执行过多次put...

map.put(key1,value1):

首先，调用key1所在类的hashCode()计算key1哈希值，此哈希值经过某种算法计算以后，得到在Entry数组中的存放位置。

- 如果此位置上的数据为空，此时的key1-value1添加成功。 ----情况1
- 如果此位置上的数据不为空，(意味着此位置上存在一个或多个数据(以链表形式存在)),比较key1和已经存在的一个或多个数据的哈希值：
  - 如果key1的哈希值与已经存在的数据的哈希值都不相同，此时key1-value1添加成功。----情况2
  - 如果key1的哈希值和已经存在的某一个数据(key2-value2)的哈希值相同，继续比较：调用key1所在类的equals(key2)方法，比较：
    - 如果equals()返回false:此时key1-value1添加成功。----情况3
    - 如果equals()返回true:使用value1替换value2。

> 关于情况2和情况3：此时key1-value1和原来的数据以链表的方式存储。
>
> 在不断的添加过程中，会涉及到扩容问题，当超出临界值(且要存放的位置非空)时，扩容。默认的扩容方式：扩容为原来容量的2倍，并将原有的数据复制过来。

数组+链表+红黑树 （ jdk 8 ）

- new HashMap():底层没有创建一个长度为16的数组
- jdk 8底层的数组是：Node[],而非Entry[]
- 首次调用put()方法时，底层创建长度为16的数组
- jdk7底层结构只有：数组+链表。jdk8中底层结构：数组+链表+红黑树。
    - 形成链表时，七上八下（ jdk7:新的元素指向旧的元素。jdk8：旧的元素指向新的元素 ）
    - 当数组的**某一个索引位置上的元素以链表形式存在的数据个数 > 8 且当前数组的长度 > 64**时，此时此索引位置上的所数据改为使用红黑树存储。
    - 当数组的**某一个索引位置上的元素以链表形式存在的数据个数 > 8 且当前数组的长度 不大于64**时，此时Node[]数组只会扩容。

jdk 8 HashMap 常量

- DEFAULT_INITIAL_CAPACITY : HashMap的默认容量，16
- DEFAULT_LOAD_FACTOR：HashMap的默认加载因子：0.75
- threshold：扩容的临界值，=容量*填充因子：16 * 0.75 => 12
- TREEIFY_THRESHOLD：Bucket中链表长度大于该默认值，转化为红黑树:8
- MIN_TREEIFY_CAPACITY：桶中的Node被树化时最小的hash表容量:64

> 在jdk7中如果键值是null，在put的时候会调用putNullKey方法
>
> 在jdk8的put代码中对null的处理为

```
static final int hash(Object key) {
     int h;
     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
 }
// key为null，hash方法返回0，
```

LinkedHashMap

保证在遍历map元素时，可以按照添加的顺序实现遍历。

原因：在原有的HashMap底层结构基础上，添加了一对指针，指向前一个和后一个元素。

对于频繁的遍历操作，此类执行效率高于HashMap。

源码中：

```
static class Entry<K,V> extends HashMap.Node<K,V> {
     Entry<K,V> before, after;//能够记录添加的元素的先后顺序，比HashMap多出的一对指针
     Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
     }
 }
```

TreeMap

保证按照添加的key-value对进行排序，实现排序遍历。此时考虑key的自然排序或定制排序底层使用红黑树

按照key排序，要求key必须是由同一个类创建的对象

Hashtable

作为古老的实现类；线程安全的，效率低；不能存储null的key和value

Properties

常用来处理配置文件。key和value都是String类型

ConcurrentHashMap

泛型

- 集合接口或集合类在jdk5.0时都修改为带泛型的结构。
- 在实例化集合类时，可以指明具体的泛型类型
- 指明完以后，在集合类或接口中凡是定义类或接口时，内部结构（比如：方法、构造器、属性等）使用到类的泛型的位置，都指定为实例化的泛型类型。

  比如：add(E e) --->实例化以后：add(Integer e)
- 注意点：泛型的类型必须是类，**不能是基本数据类型。**需要用到基本数据类型的位置，拿包装类替换
- 如果实例化时，没指明泛型的类型。默认类型为java.lang.Object类型。

泛型在继承方面的体现

虽然类A是类B的父类，但是G 和G**二者不具备子父类关系，二者是并列关系。**

```
// 编译不通过
List<Object> list1 = new ArrayList<Object>();
List<String> list2 = new ArrayList<String>();
list1 = list1;
```

补充：类A是类B的父类，A 是 B 的父类

```
// 编译通过
List<String> list1 = null;
ArrayList<String> list2 = null;
list1 = list2;
```

通配符的使用

通配符：?

类A是类B的父类，G和G是没关系的，二者共同的父类是：G<?>

```
// print(new List<String>());
// print(new List<Integer>());
// 此时List<?>相当于List<String> 和 List<Integer>的父类，即可用多态
public void print(List<?> list){
    Iterator<?> iterator = list.iterator();
    while(iterator.hasNext()){
        Object obj = iterator.next(); // 用Object接收
        System.out.println(obj);
    }
}
```

> 添加(写入)：对于List<?>就不能向其内部添加数据。除了添加null之外。
>
> 获取(读取)：允许读取数据，读取的数据类型为Object。

限制条件的通配符的使用

? extends A:

G<? extends A> 可以作为G和G的父类，其中B是A的子类，可以将 类A 或者 类A的子类 对象赋值给 类G<? extends A> 的对象。

? super A:

G<? super A> 可以作为G和G的父类，其中B是A的父类，可以将 类A 或者 类A的父类 对象赋值给 类G<? super A> 的对象。

File类

- File类的一个对象，代表一个文件或一个文件目录(俗称：文件夹)
- File类声明在java.io包下
- File类中涉及到关于文件或文件目录的创建、删除、重命名、修改时间、文件大小等方法，并未涉及到写入或读取文件内容的操作。如果需要读取或写入文件内容，必须使用IO流来完成。
- 后续File类的对象常会作为参数传递到流的构造器中，指明读取或写入的"终点"。

> IDEA中：
> 如果大家开发使用JUnit中的单元测试方法测试，相对路径即为当前Module下。
> 如果大家使用main()测试，相对路径即为当前的Project下。
> Eclipse中：
> 不管使用单元测试方法还是使用main()测试，相对路径都是当前的Project下。

构造方法

- File(String filePath)
- File(String parentPath,String childPath)
- File(File parentFile,String childPath)

常用方法

获取方法

- **public String getAbsolutePath()：获取绝对路径**
- **public String getPath() ：获取路径**
- **public String getName() ：获取名称**
- **public String getParent()：获取上层文件目录路径。若无，返回null**
- **public long length() ：获取文件长度（即：字节数）。不能获取目录的长度。**
- **public long lastModified() ：获取最后一次的修改时间，毫秒值**

如下的两个方法适用于文件目录：

- **public String[] list() ：获取指定目录下的所有文件或者文件目录的名称数组**
- **public File[] listFiles() ：获取指定目录下的所有文件或者文件目录的File数组**

判断方法

- **public boolean isDirectory()：判断是否是文件目录**
- **public boolean isFile() ：判断是否是文件**
- **public boolean exists() ：判断是否存在**
- **public boolean canRead() ：判断是否可读**
- **public boolean canWrite() ：判断是否可写**
- **public boolean isHidden() ：判断是否隐藏**

重命名方法

- **public boolean renameTo(File dest)：把文件重命名为指定的文件路径**

  比如：file1.renameTo(file2)为例：（移动文件file1到file2的路径下）

  要想保证返回true,需要file1在硬盘中是存在的，且file2不能在硬盘中存在。

删除方法

- **public boolean delete()：删除文件或者文件夹**

> **删除注意事项：Java中的删除不走回收站。**

创建方法

- **public boolean createNewFile() ：创建文件。若文件存在，则不创建，返回false**
- **public boolean mkdir() ：创建文件目录。如果此文件目录存在，就不创建。如果此文件目录的上层目录不存在，也不创建。**
- **public boolean mkdirs() ：递归创建文件目录。如果此文件目录存在，就不创建了。如果上层文件目录不存在，一并创建**

IO流

分类	字节输入流	字节输出流	字符输入流	字符输出流
抽象基类	InputStream	OutputStream	Reader	Writer
访问文件	FileInputStream	FileOutputStream	FileReader	FileWriter
访问数组	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
访问管道	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
访问字符串			StringReader	StringWriter
缓冲流	BufferedInputStream	BufferedOuputStream	BufferedReader	BufferedWriter
转换流			InputStreamReader	OutputStreamWriter
对象流	ObjectInputStream	ObjectOutputStream		
	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
打印流		PrintStream		PrintWriter
推回输入流	PushbackInputStream		PushbackReader	
特殊流	DataInputStream	DataOutputStream		

流的分类

- **操作数据单位：字节流、字符流**
- **数据的流向：输入流、输出流**
- **流的角色：节点流、处理流**

> **字符流用 `char []`，处理文本文件(.txt,.java,.c,.cpp)**
>
> **字节流用 `byte []`，处理非文本文件(.jpg,.mp3,.mp4,.avi,.doc,.ppt,...)**

输入、输出的标准化过程

输入过程

1. **创建File类的对象，指明读取的数据的来源。（要求此文件一定要存在）**
2. **创建相应的输入流，将File类的对象作为参数，传入流的构造器中**
3. **具体的读入过程：**

   创建相应的byte[] 或 char[]。
4. **关闭流资源**

   说明：程序中出现的异常需要使用try-catch-finally处理。

```
public void testFileReader1()  {
    FileReader fr = null;
    try {
        //1.File类的实例化
        File file = new File("hello.txt");
        //2.FileReader流的实例化
        fr = new FileReader(file);
        //3.读入的操作
        //read(char[] cbuf):返回每次读入cbuf数组中的字符的个数。如果达到文件末尾，
返回-1
```

```
        char[] cbuf = new char[5];
        int len;
        while((len = fr.read(cbuf)) != -1){
            //方式一：
            //错误的写法
            //                for(int i = 0;i < cbuf.length;i++){
            //                    System.out.print(cbuf[i]);
            //                }
            //正确的写法
            //                for(int i = 0;i < len;i++){
            //                    System.out.print(cbuf[i]);
            //                }
            //方式二：
            //错误的写法,对应着方式一的错误的写法
            //                String str = new String(cbuf);
            //                System.out.print(str);
            //正确的写法
            String str = new String(cbuf,0,len);
            System.out.print(str);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if(fr != null){
            //4.资源的关闭
            try {
                fr.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```

输出过程

1. **创建File类的对象，指明写出的数据的位置。（不要求此文件一定要存在）**
2. **创建相应的输出流，将File类的对象作为参数，传入流的构造器中**
3. **具体的写出过程：**

   write(char[]/byte[] buffer,0,len)
4. **关闭流资源**

   说明：程序中出现的异常需要使用try-catch-finally处理。

```
public void testFileWriter() {
    FileWriter fw = null;
    try {
        //1.提供File类的对象，指明写出到的文件
        File file = new File("hello1.txt");
        //2.提供FileWriter的对象，用于数据的写出
        fw = new FileWriter(file,false);
        //3.写出的操作
        fw.write("I have a dream!\n");
```

```
        fw.write("you need to have a dream!");
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        //4.流资源的关闭
        if(fw != null){
            try {
                fw.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

缓冲流

处理流

作用

提供流的读取、写入的速度

提高读写速度的原因：内部提供了一个缓冲区，大小为8192(字节流的单位是字节，字符流的单位是字符)

缓冲流的文件复制

```
public void testBufferedReaderBufferedWriter(){
    BufferedReader br = null;
    BufferedWriter bw = null;
    try {
        //创建文件和相应的流
        br = new BufferedReader(new FileReader(new File("dbcp.txt")));
        bw = new BufferedWriter(new FileWriter(new File("dbcp1.txt")));

        //读写操作
        //若是字节流则用bute []数组
        //方式一：使用char[]数组
        //            char[] cbuf = new char[1024];
        //            int len;
        //            while((len = br.read(cbuf)) != -1){
        //                bw.write(cbuf,0,len);
        //    //            bw.flush();
        //            }

        //方式二：使用String
        String data;
        while((data = br.readLine()) != null){
            //方法一：
            //                bw.write(data + "\n");//data中不包含换行符
            //方法二：
            bw.write(data);//data中不包含换行符
            bw.newLine();//提供换行的操作
        }
    } catch (IOException e) {
```

```
            e.printStackTrace();
        } finally {
            //关闭资源
            if(bw != null){
                try {
                    bw.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(br != null){
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
}
```

转换流

属于字符流

- **InputStreamReader：将一个字节的输入流转换为字符的输入流**
- **OutputStreamWriter：将一个字符的输出流转换为字节的输出流**

```
public void test2() throws Exception {
    //1.造文件、造流
    File file1 = new File("dbcp.txt");
    File file2 = new File("dbcp_gbk.txt");

    FileInputStream fis = new FileInputStream(file1);
    FileOutputStream fos = new FileOutputStream(file2);

    InputStreamReader isr = new InputStreamReader(fis,"utf-8");
    OutputStreamWriter osw = new OutputStreamWriter(fos,"gbk");

    //2.读写过程
    char[] cbuf = new char[20];
    int len;
    while((len = isr.read(cbuf)) != -1){
        osw.write(cbuf,0,len);
    }

    //3.关闭资源
    isr.close();
    osw.close();
}
```

对象流

序列化与反序列化

- **序列化**

将Java对象转换成二进制流，进而将二进制流保存在磁盘中或者通过网络传输到其它的网络节点。

- 反序列化

  程序够将二进制流转成Java对象

> 这种对象必须是可序列化的

```
// 存数据
oos = new ObjectOutputStream(new FileOutputStream("object.dat"));
oos.writeObject(new String("我爱北京天安门"));
oos.flush();//刷新操作

// 读数据
ois = new ObjectInputStream(new FileInputStream("object.dat"));
Object obj = ois.readObject();
String str = (String) obj;
```

条件

1. 需要实现接口：Serializable
2. 当前类提供一个全局常量：serialVersionUID（用来唯一标识类，即使类修改了，也能得到对象）

   如果类没有显示定义这个静态常量，它的值是Java运行时环境根据类的内部细节自动生成的。若类做了修改serialVersionUID可能发生变化。故建议，显式声明。
3. 除了当前Person类需要实现Serializable接口之外，还必须保证其内部所属性也必须是可序列化的。（默认情况下，基本数据类型可序列化）

> ObjectOutputStream和ObjectInputStream不能序列化static和transient修饰的成员变量

反射

程序中一般的对象的类型都是在编译期就确定下来的，而 Java 反射机制可以动态地创建对象并调用其属性，这样的对象的类型在编译期是未知的。所以我们可以通过反射机制直接创建对象，即使这个对象的类型在编译期是未知的。

反射的核心是 JVM 在运行时才动态加载类或调用方法/访问属性，它不需要事先（写代码的时候或编译期）知道运行对象是谁。

功能

- 在运行时判断任意一个对象所属的类；
- 在运行时构造任意一个类的对象；
- 在运行时判断任意一个类所具有的成员变量和方法（通过反射甚至可以调用private方法）；
- 在运行时调用任意一个对象的方法

有class对象的类型

- class：外部类 成员（成员内部类 静态内部类） 局部内部类 匿名内部类
- interface：接口
- []：数组

> 只要数组的类型和维度一样，则class对象是同一个。如`int[10]`和`int[100]`的class对象相同，
>
> `int[][]`则不同

- enum ：枚举
- annotation ：注解 @interface
- primitive type ：基本数据类型
- void

获取Class实例的方式

```
public void test3() throws ClassNotFoundException {
    //方式一：调用运行时类的属性：.class
    Class clazz1 = Person.class;
    System.out.println(clazz1);
    //方式二：通过运行时类的对象,调用getClass()
    Person p1 = new Person();
    Class clazz2 = p1.getClass();
    System.out.println(clazz2);

    //方式三：调用Class的静态方法：forName(String classPath) (常用)
    Class clazz3 = Class.forName("com.atguigu.java.Person");
    //        clazz3 = Class.forName("java.lang.String");
    System.out.println(clazz3);

    System.out.println(clazz1 == clazz2);
    System.out.println(clazz1 == clazz3);

    //方式四：使用类的加载器：ClassLoader  (了解)
    ClassLoader classLoader = ReflectionTest.class.getClassLoader();
    Class clazz4 = classLoader.loadClass("com.atguigu.java.Person");
    System.out.println(clazz4);

    System.out.println(clazz1 == clazz4);
}
```

获取当前运行时类的结构的方法

```
Class clazz = Class.forName(classPath);
//clazz.getConstructors():获取当前运行时类中声明为public的构造器
//clazz.getDeclaredConstructors():获取当前运行时类中声明的所有的构造器

//clazz.getFields():获取当前运行时类及其父类中声明为public访问权限的属性
//clazz.getDeclaredFields():获取当前运行时类中声明的所有属性。（不包含父类中声明的属性）

//clazz.getMethods():获取当前运行时类及其所有父类中声明为public权限的方法
//clazz.getDeclaredMethods():获取当前运行时类中声明的所有方法。（不包含父类中声明的方法）

//其它的结构可自行查询
```

调用运行时类中的指定结构

```
Class clazz = Class.forName(classPath);
//获取指定的属性：要求运行时类中属性声明为public，通常不采用此方法，下同
//Field field = clazz.getField(fieldName);
//1.getDeclaredField(String fieldName):获取运行时类中指定变量名的属性，通常用这个，不限权限
Field name = clazz.getDeclaredField("name");
//2.保证当前属性是可访问的
name.setAccessible(true);
//3.获取、设置指定对象的此属性值
name.set(p,"Tom");

//获取指定的某个方法
//1.getDeclaredMethod():参数1 ：指明获取的方法的名称  参数2：指明获取的方法的形参列表
Method show = clazz.getDeclaredMethod("show", String.class);
//2.保证当前方法是可访问的
show.setAccessible(true);
//3.调用方法的invoke():参数1：方法的调用者  参数2：给方法形参赋值的实参
//invoke()的返回值即为对应类中调用的方法的返回值。
//如果调用的运行时类中的方法没有返回值，则此invoke()返回null
Object returnValue = show.invoke(p,"CHN"); //String nation = p.show("CHN");

//获取指定的构造器
//1.getDeclaredConstructor():参数：指明构造器的参数列表
Constructor constructor = clazz.getDeclaredConstructor(String.class);
//2.保证此构造器是可访问的
constructor.setAccessible(true);
//3.调用此构造器创建运行时类的对象
Person per = (Person) constructor.newInstance("Tom");
```

Lambda

格式

`() -> {}`

- **-> :lambda操作符 或 箭头操作符**
- **左边：lambda形参列表（其实就是接口中的抽象方法的形参列表）**
- **右边：lambda体（其实就是重写的抽象方法的方法体）**

函数式接口

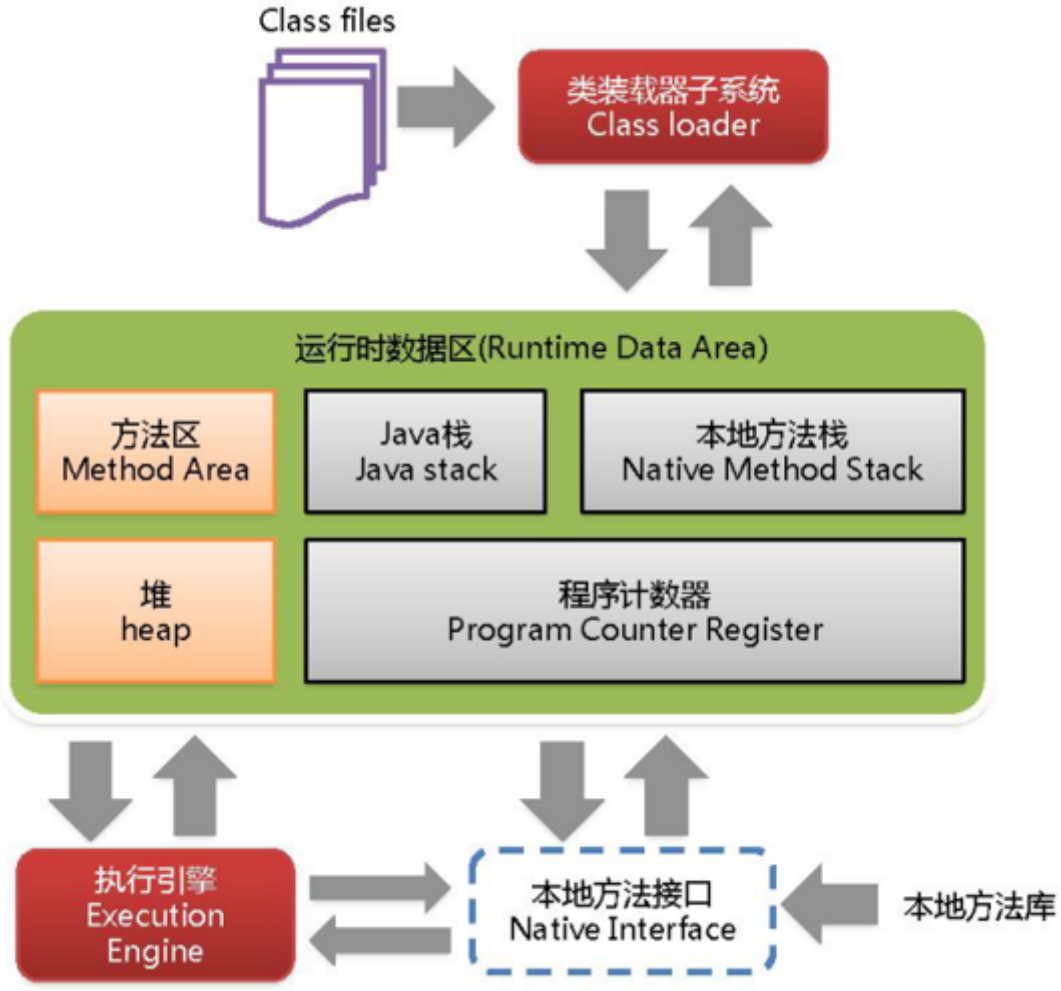
如果一个接口中，只声明了一个抽象方法，则此接口就称为函数式接口。我们可以在一个接口上使用 @FunctionalInterface 注解，这样做可以检查它是否是一个函数式接口。

> Lambda表达式的本质：作为函数式接口的实例

java内置的4大核心函数式接口

- 消费型接口 Consumer　void accept(T t)
- 供给型接口 Supplier　T get()
- 函数型接口 Function<T,R>　R apply(T t)
- 断定型接口 Predicate　boolean test(T t)

JVM



类加载器

负责加载class文件，class文件在文件开头有特定的文件标示，将class文件字节码内容加载到内存中，并将这些内容转换成方法区中的运行时数据结构并且ClassLoader只负责class文件的加载，至于它是否可以运行，则由Execution Engine决定

类别

- 启动类加载器(Bootstrap)

  加载JDK最原始的类的，如 Object、String、等等(java包下的)

- 扩展类加载器(Extension)

  加载后面JDK升级时，扩展的类(javax包下的、等等)，或者人为编写的类封装成jar包

- 应用程序类加载器（系统加载器）(AppClassLoader)

  加载自己编写的类

- 自定义类加载器

双亲委派

当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次类加载器都是如此，**因此所有的加载请求都应该传送到启动类加载器中**，只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的Class），子类加载器才会尝试自己去加载。

沙箱安全机制

采用双亲委派的一个好处是比如加载位于 rt.jar 包中的类 java.lang.Object，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就**保证了使用不同的类加载器最终得到的都是同样一个 Object对象**。

```
package java.lang;
public class String {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
// 执行时报错：在类 java.lang.String 中找不到 main 方法
// 因为加载类时先从顶层的类加载器去加载，看是否有这个类，有就加载顶层类加载器加载的类，没有就逐层往下，此时启动类加载器能够加载java.lang.String类，所以就调用启动类加载器加载的java.lang.String类，而不用自己编写的java.lang.String类。
```

执行引擎

Execution Engine执行引擎负责解释命令，提交操作系统执行。

Native Interface本地接口

（基本不用，仅了解）

本地接口的作用是融合不同的编程语言为 Java 所用，它的初衷是融合 C/C++程序，Java 诞生的时候是 C/C++横行的时候，要想立足，必须有调用 C/C++程序，于是就在内存中专门开辟了一块区域处理标记为native的代码，它的具体做法是 Native Method Stack中登记 native方法，在Execution Engine 执行时加载native libraies。

Native Method Stack

登记native方法，在Execution Engine 执行时加载本地方法库。

PC寄存器（Program Counter Register）

每个线程都有一个程序计数器，是线程私有的,就是一个**指针**，指向方法区中的**方法字节码**（用来存储指向下一条指令的地址,也即将要执行的指令代码），由执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不记。

这块内存区域很小，它是当前线程所执行的字节码的行号指示器，字节码解释器通过改变这个计数器的值来选取下一条需要执行的字节码指令。

如果执行的是一个Native方法，那这个计数器是空的。

用以完成分支、循环、跳转、异常处理、线程恢复等基础功能。不会发生内存溢出(OutOfMemory=OOM)错误

方法区

供各线程共享的运行时内存区域。它存储了每一个类的结构信息，例如运行时常量池（Runtime Constant Pool）、字段和方法数据、构造函数和普通方法的字节码内容。

> 方法区存储的信息有待查询！！！！

上面讲的是规范，在不同虚拟机里头实现是不一样的，最典型的就是永久代(PermGen space)和元空间(Metaspace)。

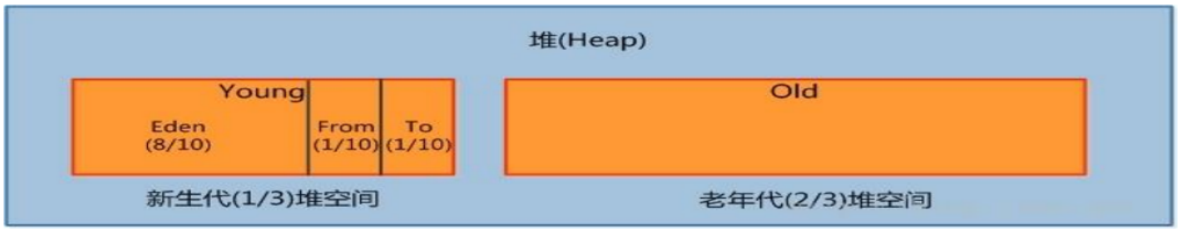
> 实例变量存在堆内存中,和方法区无关

堆

> JDK 7之前的堆内存
>
> 堆内存物理上只有新生区跟养老区（永久代/元空间只是逻辑上的存在）
>
> 虽然JVM规范将方法区描述为堆的一个逻辑部分，但它却还有一个别名叫做Non-Heap(非堆)，目的就是要和堆分开。本质上永久代是用来实现方法区，永久代是方法区(相当于是一个接口interface)的一个实现。JDK 8方法区的实现方式为元空间。
>
> 永久代使用的JVM的堆内存，但是java8以后的元空间并不在虚拟机中而是**使用本机物理内存**。运行时常量池也从JDK 8起，存储在了元空间。
>
> 永久存储区（永久代）是一个常驻内存区域，用于存放JDK自身所携带的 Class，Interface 的元数据，也就是说它存储的是运行环境必须的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭 JVM 才会释放此区域所占用的内存。

新生区是类的诞生、成长、消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集，结束生命。新生区又分为两部分： 伊甸区（Eden space）和幸存者区（Survivor pace），所有的类都是在伊甸区被new出来的。幸存区有两个： 0区（Survivor 0 space）和1区（Survivor 1 space）。当伊甸园的空间用完时，程序又需要创建对象，JVM的垃圾回收器将对伊甸园区进行垃圾回收(Minor GC)。

Java 堆从 GC 的角度还可以细分为: **新生代**(*Eden 区*、 *From Survivor 区*和 *To Survivor 区*)和**老年代。**



堆空间中，内存比例新生区：养老区=1：2，其中新生区再细分 三者比例是 8：1：1（可以通过虚拟机参数调节，一般不建议调）

MinorGc（下面的复制算法）过程

1. **eden、SurvivorFrom 复制到 SurvivorTo，年龄+1**
   首先，当Eden区满的时候会触发第一次GC，把还活着的对象拷贝到SurvivorFrom区，当Eden区再次触发GC的时候会扫描Eden区和From区域，对这两个区域进行垃圾回收，经过这次回收后还存活的对象,则直接复制到To区域（如果有对象的年龄已经达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1

2. **清空 eden、SurvivorFrom**
   然后，清空Eden和SurvivorFrom中的对象，也即复制之后有交换，谁空谁是to
3. **SurvivorTo和 SurvivorFrom 互换**
   最后，SurvivorTo和SurvivorFrom互换，原SurvivorTo成为下一次GC时的SurvivorFrom区。部分对象会在From和To区域中复制来复制去,如此交换15次（由JVM参数MaxTenuringThreshold决定，这个参数默认是15），最终如果还是存活，就存入到老年代

> **SurvivorTo跟SurvivorFrom区并不是确定的，而是动态变化的**

> **MinorGC总结：**
>
> **初始时，SurvivorFrom区为Survivor 0区，SurvivorTo区为Survivor 1区**
>
> **复制（分两种情况）：**
>
> - **第一次触发MinorGC时，只会将Eden区中没被垃圾回收的对象复制到SurvivorFrom区；**
> - **第二次触发MinorGC时，则将Eden区和SurvivorFrom区中没被垃圾回收的对象复制到SurvivorTo区；**
>
> **清空：**
>
> **复制完成后，则清空Eden区和SurvivorFrom区中的对象**
>
> **互换：**
>
> **此时原来的SurvivorFrom区就被清空了，变成下次GC的SurvivorTo区；原来的SurvivorTo区则被复制进了对象，变成了下次GC的SurvivorFrom区**

当养老区也满了，那么这个时候将产生MajorGC（FullGC），进行养老区的内存清理。若养老区执行了Full GC之后发现依然无法进行对象的保存，就会产生OOM异常“OutOfMemoryError”。

参数调节

主要参数：

-Xms	设置初始分配大小，默认为物理内存的“1 / 64”
-Xmx	最大分配内存，默认为物理内存的 “1 / 4”
-XX:+PrintGCDetails	输出详细的GC处理日志

> **生产环境中，通常需要将初始大小与最大分配内存大小调成一致**

```
MinorGC                    新生代                          堆
[GC (Allocation Failure) [PSYoungGen: 1731K->488K(2560K)] 7875K->6856K(9728K), 0.0012884 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]

FullGC                  新生代                          老年代
[Full GC (Ergonomics) [PSYoungGen: 488K->0K(2560K)] [ParOldGen: 6368K->663K(7168K)]
堆（新生代+老年代）       元空间（堆外内存）
6856K->663K(9728K), [Metaspace: 3242K->3242K(1056768K)], 0.0076479 secs]
[Times: user=0.00 sys=0.00, real=0.01 secs]
// 日志信息
// [对应区: GC前的内存占用->GC后的内存内存(对应区的内存分配)]
```

GC 垃圾回收

Minor GC和Full GC的区别

普通GC（minor GC）：只针对新生代区域的GC,指发生在新生代的垃圾收集动作，因为大多数Java对象存活率都不高，所以Minor GC非常频繁，一般回收速度也比较快。

全局GC（major GC or Full GC）：指发生在老年代的垃圾收集动作，出现了Major GC，经常会伴随至少一次的Minor GC（但并不是绝对的）。Major GC的速度一般要比Minor GC慢上10倍以上（FullGC对堆的全空间垃圾回收，MinorGC只对新生代垃圾回收）

四大算法

> - 是什么
> - 怎么用
> - 各自的优缺点

引用计数法

```
// 缺点
public class RefCountGC
{
  private byte[] bigSize = new byte[2 * 1024 * 1024];//这个成员属性唯一的作用就是占用一点内存
  Object instance = null;

  public static void main(String[] args)
  {
    RefCountGC objectA = new RefCountGC();
    RefCountGC objectB = new RefCountGC();
    // 互相引用着，若引用计数法，则不会被回收
    objectA.instance = objectB;
    objectB.instance = objectA;
    objectA = null;
    objectB = null;
    System.gc();
  }
}
```

> 引用计数法也可用于判断对象死亡，但由于缺点，因而使用可达性分析算法来判断对象死亡

复制算法

标记清除

标记压缩

> 分代收集算法：即按照不同的代（新生代、老年代）的特性而使用不同的算法
>
> 新生代：复制。新生代特点是区域相对老年代较小，对像存活率低。
>
> 老年代：清除、压缩的混合。老年代特点是区域较大，对像存活率高。

比较

内存效率：复制算法>标记清除算法>标记整理算法（此处的效率只是简单的对比时间复杂度，实际情况不一定如此）。

内存整齐度：复制算法=标记整理算法>标记清除算法。

内存利用率：标记整理算法=标记清除算法>复制算法。