Java

Java基础

输入

```
Scanner input = new Scanner(System.in);
String s = input.next();
System.out.println(s);
```

引用赋值

```
int []arr1 = {1,2,3};
int []arr2 = arr1;
arr2[0] = 1;
arr1[0] // 1
```

独立空间复制

```
int[] arr1 = {1,2,3};
int[] arr2 = new int[arr1.length];
```

可变参数

> 解决了部分方法的重载问题

```
public static void main(String[] args) {
	num(6,8,6,10);
}

public static void num(int a, int...num){
	System.out.println(a);
	for(int i:num){
		System.out.print(i+"\t");
	}
}
```

- 可变参数的实现相当于传入了一个数组
- 可变参数必须要在固定参数之后

注解

> 注解（annotation）是一种引用类型，编译后生成xxx.class

[修饰符列表] @interface 注解类型名{

}

注解怎么使用，用在什么地方？

语法格式：@注解类型名

注解可以出现在方法、属性、宏、接口上

```
@MyAnnotation
public class AnnotationTest {

	@MyAnnotation
	private String name;

	@MyAnnotation
	public void say(@MyAnnotation int s,@MyAnnotation String k){

    }
}

@MyAnnotation
interface ook{

}
```

JDK内置的注解：

`@Deprecated`

`@override` 只能注解方法，编译器会进行检查，如果不是重写父类的方法就会报错。

元注解：

注解上的注解被称为元注解。比如

`@Target` 用于标注注解出现在哪个位置。 `@Target(ElementType.METHOD)`

`@Retention` 用于被标注的注解最终保存在哪个位置。 `@Retention(RetentionPolicy.SOURCE)` 保留在java源文件中 `@Retention(RetentionPolicy.CLASS)` 保存在class文件中 `@Retention(RetentionPolicy.RUNTIME)` 保存在class文件中，并且可以被反射机制读取。

equals和==

```
public static void main(String[] args) {
	String a = "123";
	String b = "123";
	System.out.println(a==b);    //true
	System.out.println(a.equals(b));      //true

	a = new String("123");
	b = new String("123");
	System.out.println(a==b);    //false
	System.out.println(a.equals(b));      //true
```

```
        System.out.println(ao==a);      //false
        System.out.println(ao.equals(a));       //true

        ArrayList arr = new ArrayList();
        arr.add("e");arr.add("123");arr.add(123);
        System.out.println("ENd:" + (arr.contains(a) && arr.contains(ao)));
}
```

自定义对象equals

```
public class ContainsDemo {
      public static void main(String[] args) {
            User u1 = new User("A");
            ArrayList a = new ArrayList();
            a.add(u1);
            User u2 = new User("A");
            a.contains(u2);       // false 因为没有重写equals方法
      }
}

class User{
      private String name;
      public User(String name) {this.name = name;}
}
```

改正为：

```
public class ContainsDemo {
      public static void main(String[] args) {
            User u1 = new User("A");
            ArrayList a = new ArrayList();
            a.add(u1);
            User u2 = new User("A");
            System.out.println(a.contains(u2));       // false 因为没有重写equals方法
      }
}

class User{
      private String name;
      public User(String name) {this.name = name;}

      @Override
      public boolean equals(Object obj) {
            if(obj instanceof User){
                  User o = (User)obj;
                  return this.name == o.name;
            }return false;
      }
}
```

String类

> 属于不可变的类型，直接存储在“方法区”的字符串常量池中。

```
String a = "xyz";
String b = "xyz";
System.out.println(a == b); //true
String a1 = new String("xyz");
String a2 = new String("xyz");
System.out.println(a1 == a2);   //false
System.out.println(a1.equals(b));   //true

System.out.println("xyz".equals(a));    //一般使用这一种方式比较防止空指针异常
```

创建字符串的常用方法：

`String(byte [], @pre offset, @pre length)` 通过byte数组构造字符串

`String(char [], @pre offset, @pre count)`

`contains()` `endsWith()` `equalsIgnoreCase(String s)` `getBytes(byte)`

`indexOf(String s)` `matches(String regex)` `replace(String oldChar, String newChar)`

`replaceAll(String regex, String replacement)` `split(String regex)`

`substring(int begin, int end)` `[begin, end)`

`valueOf()` 将非字符串类型的变量转化为字符串。`println()` 的底层实现就是valueOf

StringBuffer和StringBuilder

> 如果业务中要使用到频繁的拼接字符串，就会占用大量的方法区，造成内存空间的浪费。

底层实现是使用的 `byte[]`

- StringBuilder兼容StringBuffer，但是StringBuilder不是线程安全的，缓冲区需要保证线程安全。
- 为了减小底层的多次扩容，可以指定初始容量。

Arrays工具类

```
public static void main(String[] args) {
      int []arr = {9,2,5,6,4,2};

      System.out.println(Arrays.toString(arr));    //tostring
      Arrays.sort(arr);    //sort  [2, 2, 4, 5, 6, 9]
      System.out.println(Arrays.toString(arr));    //tostring
      System.out.println(Arrays.binarySearch(arr, 5));

      //数组复制
      int[] newArr = Arrays.copyOf(arr, 3);
      System.out.println(Arrays.toString(newArr));      //[2, 2, 4]
      int[] newArr2 = Arrays.copyOfRange(arr, 1, 5);   // 区间复制 [1, 5)
      System.out.println(Arrays.toString(newArr2));    //[2, 4, 5, 6]
      System.arraycopy(src,srcPos , des,desPos ,length ); //也可以使用system的这个复制

      //数组比较
      int[] arr3 = {1,2,3};
      int[] arr4 = {1,2,3};
      System.out.println(Arrays.equals(arr3, arr4));   //true
      System.out.println(arr3 == arr4);          //false 相当于arr3.equals(arr4) 比较的是地址的值
```

```
        //数组填充
        Arrays.fill(arr3, 10);
        System.out.println(Arrays.toString(arr3));
}
```

集合(Collection, Iterator, List, Set, Map, 泛型)

> 都在java.uil.*下

所有以单个元素存储的集合的超级父接口是 `java.util.Collection`，继承了 `Iterable` 类表示都是可迭代可遍历的，使用迭代器 `Iterator` 来进行处理，其方法有 `hasNext()` `next()` `remove()`

所有以键值对存储的超级父接口都是 `java.util.Map`。

Collection：List，Set，Queue，Deque，SortedSet等

Map：HashMap，HashTable（线程安全）

Collection

List

> 父类Collection，存储元素有序可重复有下标。

ArrayList：底层采用数组结构（非线程安全）

`size()` 获取数组的长度，在创建ArrayList的时候预先估计一下给定的容量，减小多次分配。

```
public class Collection01 {
      public static void main(String[] args) {
            ArrayList a = new ArrayList();
            a.add(1);
            a.add(3.2);
            a.add(true);
            System.out.println(a.size());
            a.clear();
            a.add("e");
            a.add("god");
            a.set(1, 6);
            a.add("GOod");
            a.remove("e");
            System.out.println(a.isEmpty());
            Object[] array = a.toArray();    // 转化为数组

            // 迭代

            Iterator it = a.iterator(); //适用于所有的Set，List，!!!集合只要改变，迭代器必须重写!!!
            while(it.hasNext()){
                  // 删除时候只能使用 it.remove()，来删除当前元素
                  System.out.print(it.next()+",");
            }
      }
}
```

LinkedList：采用双向链表

Vector：底层采用数组结构（线程安全 所有方法含synchronized，但是效率较低使用少）

Set

> 父类Collection，无序不重复没有下标

HashSet：底层实际上是HashMap，实际存储到HashMap。等同于放在HashMap的Key部分了。

TreeSet：底层是TreeMap

`contains(Object o)`：底层是有`equals`方法实现的，因此对于自己定义的类需要实现重写equals方法。

`remove(Object o)`：底层同样调用了`equals`方法。

TreeSet Demo:

```
public class GenericDemo {
	public static void main(String[] args) {
		Set<Integer> s = new TreeSet<>();

		s.add(10);
		s.add(21);
		s.add(24);
		s.add(13);
		s.add(14);
		s.add(20);

		for (int i : s) {
			System.out.print(i + ",");
		}
	}
}

// 10,13,14,20,21,24
```

Map

HashMap：底层哈希表，非线程安全。

TreeMap：底层是二叉树，key可以按照大小顺序排序

```
public static void main(String[] args) {    //使用迭代器迭代map
	Map<Integer, String> map = new HashMap<>();
	map.put(2, "Good");
	map.put(1, "God");
	map.put(3, "Bike");

	Set<Integer> keys = map.keySet();
	Iterator<Integer> it = keys.iterator();
	while (it.hasNext()) {
		int k = it.next();
		String v = map.get(k);
		System.out.println(v);
	}
}
```

```
// foreach类型
public static void main(String[] args) {
      Map<Integer, String> map = new HashMap<>();
      map.put(2, "Good");
      map.put(1, "God");
      map.put(3, "Bike");
      for (int i : map.keySet()) {
            String s = map.get(i);
            System.out.println(s);
     }
}
```

泛型

> Generic，存储类型统一零 ，不需要向下转型了。

`ArrayList<User> uList = new ArrayList<User>();`

钻石表达式（JDK8）：`ArrayList<User> uList = new ArrayList<>();` 后面的“User”可以不用写。

反射

> java.lang.reflect.* 。可以通过java的反射机制操作（读和修改）字节码文件

相关类：

- java.lang.Class，代表字节码文件
- java.lang.reflect.Method，方法字节码
- java.lang.reflect.Constructor，构造器字节码
- java.lang.reflect.Field，属性字节码

获取Class的三种方式：

- `Class.forName()`

  ```
  Class c1 = Class.forName("java.lang.String");
  ```

- `obj.getClass()`

  ```
  public static void main(String[] args) throws ClassNotFoundException {
        Class c1 = Class.forName("java.lang.String");
        //c1 就代表了字节文件或者 String 类
        String s = "";
        Class c2 = s.getClass();
        System.out.println(c1 == c2);   //true
  }
  ```

- `${class}.class`

  `String.class`， `Integer.class`， `Date.class`， `double.class`

实例化反射Class

- 为什么使用反射实例化class文件更加灵活?

  在编写代码的时候可以跟去读取的流文件来得到具体的类名。在想创建其他对象的时候只需要修改文件（比如properties文件）的内容即可，不需要再修改代码重新编译。更加灵活。符合开闭原则。

- 如果只是像让代码的静态代码执行，只需要 `class.forName()` 即可。更不需要 `newInstance()`

方式一：

```
public static void main(String[] args) throws ClassNotFoundException {
      Class c = Class.forName("com.yz.reflect.User");
      try {
            Object o = c.newInstance();
            System.out.println(o);
     } catch (InstantiationException e) {e.printStackTrace();}
      catch (IllegalAccessException e) {e.printStackTrace();}
}
```

这种方法需要使用 `obj.newInstance()` 来实例化，必须要保证无参数构造方法存在才可以。

文件IO流

流的分类：输入流输出流，字符流字节流。再java.io中。

四大家族：字节流： `java.io.InputStream` ， `java.io.OutputStream` ，字符流： `java.io.Reader` ， `java.io.Writer`

所有的流都实现了 `java.io.Closeable` ，所有的输出流还实现了 `java.io.Flushable`

需要掌握的流：

- 文件专属： `java.io.FileInputStream` `java.io.FileOutputStream` `java.io.FileReader` `java.io.FileWriter`
- 转换流（字节 $\Longleftrightarrow$ 字符流） `java.io.InputStreamReader` `java.io.InputStreamWriter`
- 缓冲流： `java.io.BufferdReader` `java.io.BufferdWriter` `java.io.BufferdInputStream` `java.io.BufferdOutputStream`
- 数据流专属： `java.io.DataInputStream` `java.io.DataOutputStream`
- 标准输出流： `java.io.PrintWriter` `java.io.PrintStream`
- 对象专属流： `java.io.ObjectInputStream` `java.io.ObjectOutputStream`

文件流

读取文件字节流：

```
public class FileInputStream01 {
      public static void main(String[] args) {
            FileInputStream fis = null;

            try {
                  fis = new FileInputStream("C:/prac/Java/JavaLearnNotes/projects/Threads/src/a.txt");
                  // 一次性读
            int cnt = 0;
                  byte []data = new byte[4];
                  while((cnt = fis.read(data)) != -1){
                        System.out.println(new String(data, 0, cnt));   //转成字符串必须要指定长度，否则会出现覆盖的情况
```

```
                }

                // 一次读取一个字节
                int data;
                while((data = fis.read()) != -1)
                      System.out.println(data);
          } catch (IOException e) {e.printStackTrace();}
           finally{
                try {fis.close();}
                catch (IOException e) {e.printStackTrace();}
          }
     }
}
```

其他函数：`available()`：剩下没有读的字节数量 和 `skip(int n)`：跳过n个字节不读。

写出文件流：

`new FileOutputStream(String filename, Boolean append)` append 可以决定是否为追加模式。

```
public class FileOutput {
      public static void main(String[] args) {
            FileOutputStream fo = null;

            try {
                  fo = new FileOutputStream("myfile");
                  byte []data = {97,99,101,103};
                  fo.write(data);
            } catch (IOException e) {e.printStackTrace();}
            finally{
                  try {fo.close();}
                  catch (IOException e) {e.printStackTrace();}
            }
     }
}
```

查看文件路径

> src目录就是工程的根目录

```
public static void main(String[] args) {
      //getResource()  是获取类目录（src目录）下的文件
      String path = Thread.currentThread().getContextClassLoader().getResource("a.txt").getPath();
      System.out.println(path);
}
```

或者资源绑定器(绑定xxx.properties)：比如获取src目录下的user.properties `ResourceBundle rb = ResourceBundle.getBundle("user")`

面向对象

基本创建对象

```
public class Person {
	public static void main(String[] args) {
		Army army = new Army();
		System.out.println(army);
	}
}

class Army {
	private String name;
	private int age, weight;
}
```

- 只有第一次创建对象的时候创建类 `loadClass()`，第二次创建对象的时候就不会再创建类。

成员变量和局部变量

```
class Army {
	private String name;	// global var
	private int age, weight;
	public int say(){
		int num;	// loval var
		{
			int k;	// loval var
		}
		return age + weight;
	}
}
```

成员变量有初始值，局部变量没有初始值。

构造器

重载构造器之后，系统不会默认分配构造器。

```
class Person{
	String name;
	int age, height;
	public Person(){}
	public Person(String name){
		this.name = name;
	}
	public Person(String name, int age){
		this(name);
		this.age = age;
	}
	public Person(String name, int age, int height){
		this(name, age);
		this.height = height;
	}
}
```

- 同一个类中的构造器可以相互this调用，但是this构造器必须放在第一行。

static关键字

修饰属性

```
public class Demo {
      static int cnt = 0;
      int age;

      public Demo(){}

      public Demo(int age) {
            this.age = age;
     }
      public static void main(String[] args) {
            Demo d1 = new Demo();
            Demo d2 = new Demo();
            d1.age = 1;
            d2.age = 2;
            d1.cnt = 20;
            d2.cnt = 30;
            System.out.println(d1.age);
            System.out.println(d2.cnt); //直接通过类名访问  Demo.cnt
     }
}
```

- 在加载类的同时会将静态的内容（静态域）也加载到方法区。被所有对象共享

修饰方法:

```
public class Demo {
      static int cnt = 0;
      int age;

      public Demo(){}

      public Demo(int age) {
            this.age = age;
     }

      public static String say(){
            // this.Demo  不可以
            // this.age 不可以
            // Demo.cnt 可以
            return "school" ;
     }
}
```

- 非静态可以访问静态
- 静态方法不可以访问静态属性和方法

代码块

> 分为：普通块，构造块，静态块，同步块（多线程）

```
public class CodeBlock {
      int age;
      static int height;
```

```
    {//构造块（很少用）
        System.out.println("构造块");
    }

    static {//静态块
        System.out.println("静态块");
    }

    public CodeBlock(int age) {
        this.age = age;
        {// 普通块
            System.out.println("普通块");
        }
    }

    public static void main(String[] args) {
        CodeBlock codeBlock = new CodeBlock(20);
    }
}

//静态块
//构造块
//普通块
```

执行顺序：静态块 =》 构造块 =》 普通块

静态块：只在类加载的时候执行一次。一般用于创建工厂、数据库的初始化信息。

包和import

包（package）可以解决重名问题和权限问题

- 全部小写，中间用“.”隔开，公司域名倒着写com.tencent，具体模块com.tencent.login
- 但是不能使用系统中的关键字nul,con,com1-com9
- 必须在非注释的第一行，否则之间包名调用 `com.yz.Demo demo = com.yz.Demo();`
- java.lang下的包直接用
- 包之间没有包含与被包含的责任

```
tencent
|Person
|一login
    |一Administrator

import com.tencent.*;
import com.tencent.login.*;
```

- 静态导入：

```
import static java.lang.Math.*; //导入java.lang下Math类中所有的静态方法

public class CodeBlock {
    public static void main(String[] args) {
        System.out.println(round(5.6)); //直接可以用round
    }
}
```

面向对象三大特性之一——封装

> 高内聚：内部细节自己完成。低耦合：仅对外暴露少量的方法。“private”。封装：encapsulation

```
public class Girl {
	private int age;

	public int getAge() {
		return age;
	}
}
```

面向对象三大特性之一——继承

> Inheritance. 继承 is - a 关系。

```
public class Person {
	private int age, height;
	private String name;

	public int getAge() {return age;}

	public void setAge(int age) {this.age = age;}
}

class Girl extends Person{
	private int brain;

	public int getBrain() {return brain;}

	public void setBrain(int brain) {this.brain = brain;}

	public static void main(String[] args) {Girl girl = new Girl();}
}
```

- 子类无法访问private属性和方法，但是仍可以使用暴露的方法对属性进行修改。
- 便于代码扩展，是多态的前提
- 不允许多继承，但继承具有传递性

权限修饰符：

	同一个类	同一个包	子类	所有类
private	*			
default	*	*		
protected	*	*	*	
public	*	*	*	*

重载和重写：

重写：发生在子类和父类之间的方法。参数的个数，类型必须相同。但是加入返回值是引用类型，父类的返回值是子类的返回值的父类，则可以。父类的权限修饰符要低于子类。

重载：发生在同一个类中，方法名相同，形参列表不同。

super关键字

```
public class Girl extends Person{
      private int brain;

      public int getBrain() {
            return brain;
      }

      public void setBrain(int brain) {
            this.brain = brain;
            super.getAge();      // 默认情况下super默认不写。当子类中后者局部变量也含相同变量名
      }

      public static void main(String[] args) {
            Girl girl = new Girl();
      }
}
```

一般执行流程：子类构造器`Girl()`，父类构造器`Person()`

错误❌：

```
public class Girl extends Person{
      private int brain;

      public Girl(int id, String name, int age){
            super(id, name);    //❌
            this(age);      //❌ 因为构造器只能放在第一行，两者冲突，因此只能存在一个构造器
      }

      public static void main(String[] args) {
            Girl girl = new Girl();
      }
}
```

Object类

`toString()`

```
public class Girl extends Person{
      private int brain;

      public static void main(String[] args) {
            Girl girl = new Girl();
            System.out.println(girl);
      }
}
//com.yz3.Girl@1540e19d
```

`equals()`

底层依旧使用的是`==`，需要重写equals方法

```
public class Girl extends Person{
	private int brain;

	public static void main(String[] args) {
		Girl girl = new Girl();
		System.out.println(girl);
	}

	@Override
	public boolean equals(Object obj) { // 初级版 需要instanceof改进
		Girl girl = (Girl) obj;
		return this.brain == girl.brain && this.getAge() == girl.getAge();
	}
}
```

`instanceof`

```
	@Override
	public boolean equals(Object o) {
		if (this == o) return true;
		if (o == null || getClass() != o.getClass()) return false;
		Girl girl = (Girl) o;
		return brain == girl.brain;
	}
```

面向对象三大特性之一——多态

> 多态和属性无关，多态指的是方法的多态，而不是属性的多态。

```
public class Animal {
	public void shout(){System.out.println("小动物叫");}
}

class Cat extends Animal{
	@Override
	public void shout() {System.out.println("猫猫叫");}
}

class Dog extends Animal{
	@Override
	public void shout() {System.out.println("狗狗叫");}
}

class Girl{
	public void play(Animal a){a.shout();}
}

class Test{
	public static void main(String[] args) {
		Girl girl = new Girl();
		Animal dog = new Dog();
		girl.play(dog);

		Animal cat = new Cat();
		girl.play(cat);
	}
```

```
}
```

现在的代码扩展性好，当小女孩想和不同动物玩的时候就不需要再改代码了。符合开闭原则：扩展是开放的，修改是关闭的。

多态的要素：继承，重写，父类引用指向子类对象 `父类 v = new 子类()` 。

向上转型和向下转型：

```
Animal a = new Dog();   // 多态
Dog d = (Dog)a; //向下转型
```

简单工厂设计模式

传参即可返回子类，方法必须是static

final关键字

修饰变量：

```
final Dog d = new Dog("10");
//d = new Dog("20");  //error
d.name = "30";  //okay
```

`final` 只管 `d` 的地址不被改变。作为函数参数的时候不可以进行改变。相当于C++中的 `const`

修饰方法：final修饰的方法，子类不可以继承

修饰类：final修饰的类，该类不可以被继承，里面的方法也可以不用final修饰。比如 `Math` 类。

其中Math类中有一个细节 `private Math(){}` 。其构造方法为private类型，就是不让任何人去随意实例化。

抽象类和抽象方法

```
public abstract class Person {
      public abstract void say();

      public void look(){
            System.out.println("Looking for u");
     }
}
```

- 含抽象方法的类必须是抽象类
- 子类必须重写抽象父类中的抽象方法，如果不重写，则子类必须是抽象类
- 抽象类不可以创建对象

问题：

- 抽象类中是否有构造器？有。构造器的作用给子类创建对象的时候先用super调用父类构造器。
- 抽象类不能被final修饰，因为抽象类必须要被继承才有作用。

接口

- 接口（interface）没有构造器。类和接口的关系是实现关系（has-a关系）。
- Java只有单继承，但有多实现。
- 也可以多态

```
public interface Person {
      int NUM = 10;
      void say();
      int getHeight();
}

class Student implements Person{
      @Override
      public void say() {System.out.println("ni hao");}

      @Override
      public int getHeight() {return 0;}
}
```

JDK1.8之后新增内容：

- default修饰符新增的内容，default可以修饰非抽象方法。
- 可以写静态方法，但是静态方法不能重写

内部类

> 成员内部类和局部内部类（方法内、块内、构造器内）

成员内部类：

```
public class Person {
      int age;
      String name;
      static double height = 10.8;
      public class InnerClass{
            int age;
            public void say(){
                  System.out.println(age);      // InnerClass.this.age
                  System.out.println(Person.this.age);
                  System.out.println(height);
            }
      }

      static class Ic2{
            public void go(){
                  System.out.println(height);
//                 System.out.println(age);        error
            }
      }

      public void stand(){
            InnerClass innerClass = new InnerClass();
            innerClass.say();
      }
}
```

```
class Test{
      public static void main(String[] args) {
            Person person = new Person();
            Person.InnerClass innerClass = person.new InnerClass(); // 创建非静态内部类

            Person.Ic2 ic2 = new Person.Ic2();      // 创建静态内部类
      }
}
```

局部内部类：

- 在局部内部类中访问的变量必须是final修饰的
- 如果类只使用一次的话，可以使用内部类（安卓常用）
- 匿名内部类

```
public class Person {
      int age;
      String name;
      static double height = 10.8;

      public Comparable say(){
            return new Comparable() {   // 返回匿名内部类对象
                  @Override
                  public int compareTo(Object o) {
                        return 0;
                  }
            };
      }
}
```

多线程

多线程创建

线程创建的三种方式：Thread类、Runnable接口、Callable接口

创建方式一：继承Thread类

- 继承Thread类并且重写 `run()` 方法，创建对象调用 `start()` ，由线程执行 `run()` 方法。(不直接使用 `run()` 方法，否则相当于程序调用，为一个线程)

```
public class Thread1 extends Thread{
      @Override
      public void run() {
            for (int i = 0; i < 200; i++) {
                  System.out.println("线程"+ i);
            }
      }

      public static void main(String[] args) {
            Thread1 thread1 = new Thread1();
            thread1.start();
            for (int i = 0; i < 200; i++) {
                  System.out.println("Main Thread" + i);
            }
```

```
    }
}
```

创建方式二：实现Runnable接口

```
public class RunnableDemo implements Runnable{
      @Override
      public void run() {
            for (int i = 0; i < 30; i++) {
                  System.out.println("Runnable== " + i);
            }
      }

      public static void main(String[] args) {
            // 创建实现类的对象
            RunnableDemo runnableDemo = new RunnableDemo();
            // 创建线程对象，通过线程对象开启“代理”
            Thread thread = new Thread(runnableDemo);
            thread.start();

            for (int i = 0; i < 30; i++) {
                  System.out.println("Main Thread" + i);
            }
      }
}
```

避免单继承的局限性，灵活方便。

创建方式三：实现Callable接口

流程：

1. 创建执行服务 `ExecutorService ser = Executors.newFixedThreadPool(3);`
2. 提交执行 `Future<Boolean> r1 = ser.submit(t1);`
3. 获取结果 `Boolean rs1 = r1.get();`
4. 关闭服务 `ser.shutdownNow();`

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableDemo implements Callable<Boolean>{

      private String name;
      @Override
      public Boolean call() {
            for (int i = 0; i < 10; i++) {
                  System.out.println("Callable:  " + name + " ->  " + i);
            }
            return true;
      }
      public static void main(String[] args) throws InterruptedException, ExecutionException {
            CallableDemo t1 = new CallableDemo("小红");
            CallableDemo t2 = new CallableDemo("小黄");
            CallableDemo t3 = new CallableDemo("小韩");
```

```
            //创建服务
            ExecutorService ser = Executors.newFixedThreadPool(3);   //3个线程池

            //提交执行
            Future<Boolean> r1 = ser.submit(t1);
            Future<Boolean> r2 = ser.submit(t2);
            Future<Boolean> r3 = ser.submit(t3);

            //获取结果
            Boolean rs1 = r1.get();
            Boolean rs2 = r2.get();
            Boolean rs3 = r3.get();

            //关闭服务
            ser.shutdownNow();

            System.out.println(rs1 + " = " + rs2   + " = " + rs3);

      }

       public CallableDemo(String name) {
            this.name = name;
      }

}
```

Callable的好处：

- 可以定义返回值
- 可以抛出异常

线程不安全情况

> 类型：多个线程拿同一个资源

以买火车票为案例。

```
public class Tickets implements Runnable{
      private int ticketsNum = 10;
      @Override
      public void run() {
            while (true){
                  if (ticketsNum<=0)break;
                  try {
                        Thread.sleep(100);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
                  System.out.println(Thread.currentThread().getName() + "==> 抢到了第" + ticketsNum-- +
"张票。");
            }
      }

      public static void main(String[] args) {
            Tickets tickets = new Tickets();

            new Thread(tickets, "小红").start();
```

```
            new Thread(tickets, "小明").start();
            new Thread(tickets, "小黄").start();

      }
}
```

输出：

```
小黄==> 抢到了第8张票。
小明==> 抢到了第10张票。
小红==> 抢到了第9张票。
小明==> 抢到了第7张票。
小黄==> 抢到了第6张票。
小红==> 抢到了第5张票。
小红==> 抢到了第4张票。
小黄==> 抢到了第4张票。
小明==> 抢到了第4张票。
小红==> 抢到了第1张票。
小黄==> 抢到了第3张票。
小明==> 抢到了第2张票。
```

出现了线程不安全的情况，数据紊乱。

龟兔赛跑模型

> 两个对象共用一个资源

```
public class Race implements Runnable{
      private String winner;

      @Override
      public void run() {
            for (int i = 0; i <= 100; i++) {
                  if (Thread.currentThread().getName().equals("rabbit") && i % 10 == 0){
                        try {
                              Thread.sleep(2);
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                  }

                  if (isGameOver(i))break;
                  System.out.println(Thread.currentThread().getName() + "跑了" + i + "步");
            }

      }

      public boolean isGameOver(int steps){
            if (winner != null){
                  return true;
            }else{
                  if (steps>=100){
                        winner = Thread.currentThread().getName();
                        System.out.println("胜利者是" + winner);
                        return true;
                  }
            }
```

```
            return false;
        }

        public static void main(String[] args) {
            Race race = new Race();

            new Thread(race, "rabbit").start();
            new Thread(race, "tortoise").start();
        }
}
```

静态代理

> static proxy

```
public class StaticProxy {
        public static void main(String[] args) {
            WeddingCompant w = new WeddingCompant(new You());
            w.happyMarry();

            // 相当于这个
            new Thread(()->System.out.println("Good")).start(); // 两个方法开启的方式类似
            new WeddingCompant(new You()).happyMarry();         //
        }
}


interface Marry{void happyMarry();}

class You implements Marry{
        @Override
        public void happyMarry() {System.out.println("Happy marry");}
}

class WeddingCompant implements Marry{
        private Marry target;

        @Override
        public void happyMarry() {
            before();   //做原对象不能做的行为，进行细节补充
            target.happyMarry();
            after();
        }

        public void before(){System.out.println("Before wedding preparation");}

        public void after(){System.out.println("After wedding charge.");}

        public WeddingCompant(Marry target) {this.target = target;}
}
```

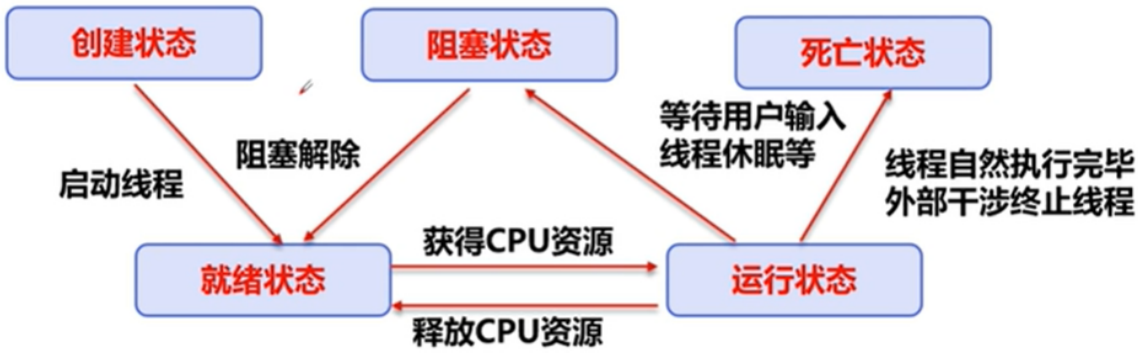
lambda表达式

> 避免匿名内部类过多，

函数式接口：对于一个类，如果只含有一个抽象方法，则其是一个函数式接口。可以直接继承。

```
public class TestLambda {

      public static void main(String[] args) {
            Bribe b = new Bribe(){
                  @Override
                  public void marry(String name) {
                        System.out.println("匿名内部类" + name);
                  }
            };

            b.marry("Karry");

            b = (name) -> System.out.println("Lambda 表达式" + name);  // 可以简化类型
            b.marry("Mate");
      }
}

interface Bribe{
      void marry(String name);
}
```

线程状态



状态：

- 创建状态：`new`创建线程
- 就绪状态：调用`start()`方法，线程会立即进入就绪状态，但不意味着立即执行
- 运行状态：线程获得了CPU处理机资源
- 阻塞状态：线程调用`sleep()` `wait()`或者`同步锁定`的时候线程就会进入阻塞状态。阻塞解除之后继续进入就绪状态。
- 死亡状态：线程运行完毕或者终止。

线程方法：

- 停止线程

> 不推荐使用`destroy()` `resume()` `stop()`等方法；推荐使用标志位进行中止让自己停下来

```
public class ThreadStop implements Runnable{
      private Boolean flag = true;

      @Override
      public void run() {
            int i = 0;
            while(flag)System.out.println("Thread:  " + i++);
      }

      public void stop(){flag = false;}

      public static void main(String[] args) {
            ThreadStop ts = new ThreadStop();
            new Thread(ts).start();

            for (int i = 0; i < 1000; i++) {
                  System.out.println("Main Thread " + i);
            }
            ts.stop();
      }
}
```

- 线程休眠

> `sleep()` 函数，模拟网络延时（放大问题的发生率，可能引起线程不安全），模拟倒计时

每个对象都有一把锁，但是`sleep()` 不会释放锁。

- 线程礼让（这个需要深入）

> `yield()` 函数。礼让线程，让当前执行的线程暂停但不阻塞转为就绪状态。**但礼让不一定会成功**

```
public class ThreadYield{

      public static void main(String[] args) {
            my ty = new my();

            new Thread(ty, "A").start();
            new Thread(ty, "B").start();
      }
}


class my implements Runnable{
      @Override
      public void run() {
            System.out.println(Thread.currentThread().getName() + "线程开始");
            Thread.yield();
            System.out.println(Thread.currentThread().getName() + "线程结束");
      }
}
```

- 线程插队JOIN

> `join()` 合并线程，待此线程执行完毕之后再执行其他线程，其他线程阻塞。十分霸道

```
public class ThreadJoin implements Runnable{
      @Override
      public void run() {
            for (int i = 0; i < 10; i++) {
                  System.out.println("线程VIP JOIN" + i);
            }
      }

      public static void main(String[] args) throws InterruptedException {
            ThreadJoin tj = new ThreadJoin();
            Thread t = new Thread(tj);
            t.start();

            for (int i = 0; i < 20; i++) {
                  if(i == 15)t.join();
                  System.out.println("主线程 " + i);
            }
      }
}
```

输出如下：

```
主线程 0
线程VIP JOIN0
主线程 1
线程VIP JOIN1
主线程 2
线程VIP JOIN2
主线程 3
主线程 4
主线程 5
主线程 6
主线程 7
主线程 8
主线程 9
主线程 10
主线程 11
主线程 12
主线程 13
主线程 14
线程VIP JOIN3     到15的时候插入，直到VIP线程死亡为止
线程VIP JOIN4
线程VIP JOIN5
线程VIP JOIN6
线程VIP JOIN7
线程VIP JOIN8
线程VIP JOIN9
主线程 15
主线程 16
主线程 17
主线程 18
主线程 19
```

观测线程状态

NEW，RUNNABLE，BLOCKED，WAITING，TIMED_WAITING，TERMINATED

```
import java.lang.Thread.State;

public class ThreadStatus {
      public static void main(String[] args) throws InterruptedException {
            Thread t = new Thread(() -> {
                  for (int i = 0; i < 1; i++) {
                        try {
                              Thread.sleep(200);
                        } catch (InterruptedException e) {
                              // TODO Auto-generated catch block
                              e.printStackTrace();
                        }
                  }
                  System.out.println("Over");
            });

            State state = t.getState();
            System.out.println(state); // NEW

            t.start();
            state = t.getState();
            System.out.println(state);   // RUNNABLE

            while(state != State.TERMINATED){
                  Thread.sleep(20);
                  state = t.getState();
                  System.out.println(state);
            }

      }
}
```

输出：

```
NEW
RUNNABLE
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
Over
TERMINATED
```

线程优先级

设置方法：`setPriority(int)` `getPriority()`

优先级数字：用1~10表示。常量表示`Thread.MIN_PRIORITY` `Thread.NORM_PRIORITY` `Thread.MAX_PRIORITY`

```
public class ThreadPriority {
	public static void main(String[] args) {
		Pr pobj = new Pr();

		Thread t1 = new Thread(pobj);
		Thread t2 = new Thread(pobj);
		Thread t3 = new Thread(pobj);
		Thread t4 = new Thread(pobj);

		t1.start();

		t2.setPriority(3);
		t2.start();

		t3.setPriority(Thread.MAX_PRIORITY);
		t3.start();

		t4.setPriority(Thread.MIN_PRIORITY);
		t4.start();
	}
}

class Pr implements Runnable{
	@Override
	public void run() {
		System.out.println(Thread.currentThread().getName() +" Priority -> " +
Thread.currentThread().getPriority());
	}
}
```

输出：

```
Thread-0 Priority -> 5
Thread-3 Priority -> 1
Thread-1 Priority -> 3
Thread-2 Priority -> 10
```

守护线程

> Daemon

线程分为用户线程和守护线程。虚拟机必须确保用户线程执行完毕，不需要等待守护线程执行完毕。

```
public class ThreadDaemon {
	public static void main(String[] args) {
		DaemonProcess daemon = new DaemonProcess();
		Thread daemont = new Thread(daemon);
		daemont.setDaemon(true);
		daemont.start();
```

```
            new Thread(new AccProcess()).start();
      }
}

class DaemonProcess implements Runnable{
      @Override
      public void run() {
            while(true){
                  System.out.println("Daemon running");
            }
      }
}

class AccProcess implements Runnable{
      @Override
      public void run() {
            for (int i = 0; i < 10; i++) {
                  System.out.println("Person good.");
            }

            System.out.println("Good Bye...");
      }
}
```

线程同步

> 并发：多个线程操作同一个资源。

多个线程需要同时访问一个资源，这就需要使用排队的思想来解决这个问题。**线程同步**其实是一个等待机制，多个线程访问此对象的线程需要进入这个对象的**等待池**。而且还需要一种锁机制(synchronized)

线程不安全案例

多人买票问题：

```
public class unsafeBuyTickets {
      public static void main(String[] args) {
            BuyTickets station = new BuyTickets();

            new Thread(station, "A").start();
            new Thread(station, "B").start();
            new Thread(station, "C").start();
      }
}

class BuyTickets implements Runnable{
      private int ticketsCnt = 10;
      private Boolean flag = true;

      @Override
      public void run() {
            while(true){
                  if(!flag)break;
                  buyTicket();
            }
      }
```

```
        public void buyTicket(){
            if(ticketsCnt<=0){
                flag = false;
                return;
            }
            try{Thread.sleep(100);}
            catch (InterruptedException e){e.printStackTrace();}
            System.out.println(Thread.currentThread().getName() + "抢到了第" + ticketsCnt-- + "张票");
        }
}
```

输出：

```
B抢到了第8张票
A抢到了第10张票
C抢到了第9张票
B抢到了第7张票
C抢到了第6张票
A抢到了第5张票
C抢到了第4张票
A抢到了第3张票
B抢到了第3张票
C抢到了第2张票
A抢到了第1张票
B抢到了第0张票
C抢到了第-1张票
```

会出现多人买到同一张票，或者出现负数的问题。多人同一个时间段内进入判断条件的时候变量`ticketCnt`还大于0，由于延时的因素，前面先运行的程序先取走了票，后面的程序未再进行检测，因此导致负数。

不安全取钱问题：

```
package com.yz.unsafeDemo;

public class unsafeBank {
        public static void main(String[] args) {
            Account acc = new Account(100);
            Bank a = new Bank(50, acc);
            Bank b = new Bank(60, acc);

            new Thread(a, "A").start();
            new Thread(b, "B").start();
        }
}

class Account{
        private int money;
        public Account(int money) {this.money = money;}
        public int getMoney() {return money;}
        public void setMoney(int money) {this.money = money;}
}

class Bank implements Runnable{
        private int getMoney;
        Account acc;
        @Override
```

```
        public void run() {
            if(acc.getMoney() - getMoney <= 0)System.out.println("你的钱余额不足！");
            else{
                try {Thread.sleep(1000);}   // 延时会放大错误的发生率
                catch (InterruptedException e) {e.printStackTrace();}
                acc.setMoney(acc.getMoney() - getMoney);
                System.out.println(Thread.currentThread().getName() + "已经取走钱" + getMoney + "元。剩余："+ acc.getMoney());
            }
        }

        public Bank(int getMoney, Account acc) {
            this.getMoney = getMoney;
            this.acc = acc;
        }
}
//A已经取走钱50元。剩余：-10
//B已经取走钱60元。剩余：-10
```

不安全的集合

```
public class unsafeList {
        public static void main(String[] args) {
            List<String> l = new ArrayList<String>();
            for (int i = 0; i < 10000; i++) {
                new Thread(() -> {
                    l.add("a");
                }).start();
            }
            System.out.println(l.size());
        }
}
//7686  延时后会变多
```

※同步方法和同步块

使用 `synchronized` 的关键字对于将要进行增删改查操作对象进行修饰。

修复多人取票问题：

```
public synchronized void buyTicket(){
        if(ticketsCnt<=0){
            flag = false;
            return;
        }
        try{Thread.sleep(500);}
        catch (InterruptedException e){e.printStackTrace();}
        System.out.println(Thread.currentThread().getName() + "抢到了第" + ticketsCnt-- + "张票");
}
```

修复多人取钱问题：

对于进行增删改查的acc对象进行synchronized限制。

```
@Override
        public void run() {
            synchronized(acc){
                if(acc.getMoney() - getMoney <= 0)System.out.println("你的钱余额不足！");
```

```
                else{
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    acc.setMoney(acc.getMoney() - getMoney);
                    System.out.println(Thread.currentThread().getName() + "已经取走钱" + getMoney +
"元。剩余："+ acc.getMoney());
                }
            }
    }
```

修复不安全集合问题：

> 未修复

```
```

死锁

死锁样例，两个对象都占有并且不想放弃对方想要的资源，就造成了死锁。

```
public class DeadLock{
    public static void main(String[] args) {
        MakeUp g1 = new MakeUp("小莉", 0);
        MakeUp g2 = new MakeUp("小萌", 1);

        new Thread(g1).start();
        new Thread(g2).start();
    }
}

class Mirror {}
class LipStick {}

class MakeUp implements Runnable{
    static final Mirror mirror = new Mirror();
    static final LipStick lipStick = new LipStick();

    private String name;
    private int choice;

    public MakeUp(String name, int choice) {
        this.name = name;
        this.choice = choice;
    }

    @Override
    public void run() {
        try {mu();}
        catch (InterruptedException e) {e.printStackTrace();}
    }

    private void mu() throws InterruptedException {
        if (choice == 1){
```

```
                synchronized (mirror){
                    System.out.println(name + "拿到了镜子");
                    Thread.sleep(1000);
                    synchronized (lipStick){
                        System.out.println(name + "拿到了口红。=化妆完毕");
                    }
                }
            }else{
                synchronized (lipStick){
                    System.out.println(name + "拿到了口红");
                    Thread.sleep(2000);
                    synchronized (mirror){
                        System.out.println(name + "拿到了镜子。=化妆完毕");
                    }
                }

            }
        }
}
```

线程通信协作（生产者消费者模型）

Maven

安装和配置环境变量

https://maven.apache.org/download

配置环境变量 MAVEN_HOME

maven的目录结构：

目录	目的
${basedir}	存放 `pom.xml` 和其他子目录
${basedir}/src/main/java	项目Java源代码
${basedir}/src/main/resourses	项目的资源，property文件
${basedir}/src/test/java	项目的测试类，JUnit代码
${basedir}/src/test/resourses	测试使用的资源

```
C:.
|    pom.xml
|
\---src
    +---main
    |    +---java
    |    |    \---com
    |    |         \---yz
    |    |                   App.java
```

```
|   |
|   \---resources
\---test
    +---java
    |   \---com
    |       \---yz
    |                   AppTest.java
    |
    \---resources
```

更改仓库文件依赖文件存放地址：

- 找到maven目录，找到conf -> settings.xml
- 找到 `<localRepository>/path/to/local/repo</localRepository>`，并且更改里面的地址e.g. `<localRepository>C:/Apps/maven/repo</localRepository>` (改为反斜杠)

修改maven仓库镜像地址：

- 同样在 `settings.xml` 文件内，在 `<mirrors></mirrors>` 内添加一下配置

```
<mirror>
  <id>aliyunmaven</id>
  <mirrorOf>*</mirrorOf>
  <name>aliyun</name>
  <url>https://maven.aliyun.com/repository/public</url>
</mirror>
```

编译运行

- 在项目目录下输入指令 `mvn compile`，第一次等下文件下载完毕。
- 运行java文件：`mvn exec:java -Dexec.mainClass="com.yz.App"` (不带“.java”后缀名)

maven命令

- maven -version
- maven clean
- maven compile 编译src/main/目录下的java文件
- -D 指定属性 -P 指定profile，设置运行环境

Spring

> Spring框架分为四部分

BV1wy4y1D7JT

IoC控制反转

> Inverse of Control是一种思想。对象的创建、赋值、管理都是由容器实现的。

java中创建对象的的方法：构造方法、反射、序列化、克隆、IoC、动态代理

IoC的技术实现：DI（Dependency Injection）依赖注入。底层使用的是反射机制。

Maven项目

Maven-archetype-quickstart

创建Spring项目：

1. 创建maven项目
2. 加入maven依赖，spring依赖(version 5.2.5)，juint依赖

```
<properties>
	<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
	<maven.compiler.source>1.8</maven.compiler.source>
	<maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
	<!--	juint单元测试-->
	<dependency>
		<groupId>junit</groupId>
		<artifactId>junit</artifactId>
		<version>4.11</version>
		<scope>test</scope>
	</dependency>
	<!--	spring 依赖-->
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-context</artifactId>
		<version>5.3.5</version>
	</dependency>
</dependencies>
```

3. 创建类，创建spring的配置文件

   在 `resources` 目录下创建 `beans.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans.xsd">
	<!-- xmlns:p="http://www.springframework.org/schema/p" -->

	<!-- 使用spring来创建对象
		声明bean，告诉spring来创建某个类的对象
		id 是对象的自定义名称，spring通过名称来找到对应的对象
		class 是对象的全限定名称，不能是接口（反射）
		Spring是使用map来进行存储对象的 SpringMap.put("SomeService", new SomeServiceImpl())
	-->

	<bean id="SomeService" class="com.yz.spring01.services.impl.SomeServiceImpl" />

</beans>
<!--
    beans是根标签，spring把java对象转为bean
    xsd文件是约束文件
-->
```

1. 在Test中测试spring创建对象

```
@Test
public void test02(){
      // 指定spring的配置文件
      String config = "beans.xml";
      //创建spring容器的对象，ApplicationContext表示spring容器，就可以获取对象了
      ApplicationContext ac = new ClassPathXmlApplicationContext(config); // 执行所有bean的构造方法
      //从类路径下加载对象文件
      SomeService service = (SomeService)ac.getBean("SomeService");
      service.doSome();
}
```

初始化项目后操作：

1. 进入pom.xml文件
2. 删除 `<name>` 和 `<url>` 标签
3. 将maven编译源和目标版本改为1.8

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

4. 删除 `<build>` 标签内容

Junit单元测试

1. 需要在 `pom.xml` 加入junit依赖：

```
<dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
</dependency>
```

2. 在src/test/java中进行方法定义：

```
//public 没有参数、返回值，使用 @Test 进行标注
@Test
public void t5() {
      String conf = "ba01/applicationContext.xml";
      ApplicationContext ac = new ClassPathXmlApplicationContext(conf);
      Student s = (Student)ac.getBean("StudentBean");
      System.out.println(s);
}
```

3. Run test

获取Spring中对象的信息

```
@Test
public void test3(){
      String conf = "beans.xml";
      ApplicationContext ac = new ClassPathXmlApplicationContext(conf);

      int nums = ac.getBeanDefinitionCount();
      System.out.println("Beans中定义的对象数量：  " + nums);

      System.out.println("容器中对象的名字为：");
      String[] names = ac.getBeanDefinitionNames();
      for (String name : names) {
            System.out.println(name);
     }
}
```

Spring创建非自定义的对象

比如说java自带的Date类

```
@Test
public void Test04() {
      String conf = "beans.xml";
      ApplicationContext ac = new ClassPathXmlApplicationContext(conf);
      Date my = (Date) ac.getBean("MyDate");
      System.out.println(my);
}
```

DI(基于XML的依赖注入)

> DI：表示创建对象，给对象属性赋值

1. 基于XML的DI：在spring的配置文件中使用标签和属性进行完成
2. 基于注解的DI：在spring中注解完成属性赋值

DI的分类：

- set设值注入：调用spring中的set方法
- 构造注入：创建对象在构造方法中实现注入。

基于XML

Set 注入

简单类型的xml注入：

首先定义类

```
package com.yz.spring01.ba01;

public class Student {
      private String name;
      private int age;
      public void setName(String name) {
            this.name = name;
     }
      public void setAge(int age) {
```

```
            this.age = age;
        }
        @Override
        public String toString() {
            return "Student [age=" + age + ", name=" + name + "]";
        }
}
```

然后再beans.xml文件中进行赋值：

原理：是调用对象实现的setter方法进行赋值。只要有set方法就能调用

```
<bean id="StudentBean" class="com.yz.spring01.ba01.Student">
    <property name="name" value="小萌"></property>
    <property name="age" value="18"></property>
</bean>
```

在Test中进行创建容器创建对象，输出即可。

引用类型的赋值

```
<bean id="StudentBean" class="com.yz.spring01.ba01.Student">
    <property name="name" value="小萌"></property>
    <property name="age" value="18"></property>
    <!-- 引用类型，ref设置为对应的bean id -->
    <property name="school" ref="MySchool"></property>
</bean>

<bean id="MySchool" class="com.yz.spring01.ba01.School">
    <property name="name" value="SZU"></property>
</bean>
```

构造注入：

需要对应的类要实现构造方法，使用`<constructor-arg>`标签

```
<bean id="Student2" class="com.yz.spring01.ba01.Student">
    <constructor-arg index="0" value="萌萌"></constructor-arg>
    <constructor-arg index="1" value="18"></constructor-arg>
    <constructor-arg index="2" ref="MySchool"></constructor-arg>
</bean>
<!-- 也可以省略index属性 -->
```

自动注入：

1. 引用类型自动注入：

2. 自动注入-ByType（按类型注入）：

   保证同源（同类，父子类，接口实现类）关系

```
<bean id="MyStu" class="com.yz.spring01.ba03.Student" autowire="byType">
      <property name="age" value="19"/>
      <property name="name" value="萌萌子"/>
</bean>

<!-- byType 类型无需对id进行一致-->

<bean id="myschool" class="com.yz.spring01.ba03.School">
      <property name="name" value="SZU[JNU] == bytype"/>
</bean>
```

多配置文件

为了防止配置文件过大，读取时间过长。

一般分为主配置文件和子配置文件。主配置文件是用来包含其他配置文件，不定义对象，用于导入其他配置文件。

```
配置文件目录
ba04
    spring-total.xml
    spring-student.xml
    spring-school.xml
```

student和schoold的xml文件只保存自己的bean，类似上述的配置。

在total主配置文件中进行配置：

```
<beans xmlns="http://www.springframework.org/schema/beans"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
            <!--    classpath需要在target目录下的classes进行寻找 -->

      <import resource="classpath:ba04/spring-student.xml"/>
      <import resource="classpath:ba04/spring-school.xml"/>

      <!-- 可以进行包含关系进行通配符匹配 -->
      <import resource="classpath:ba04/spring-*.xml"/>
      <!-- 但是主配置文件不能与这个文件名匹配  -->
</beans>
```

DI(基于注解的依赖注入)

资料

- 资料百度云地址 **https://pan.baidu.com/s/1uQV8miKR5LM3u2ZfD5Q51Q**，提取码tlnb
- 思维导图： **https://pan.baidu.com/s/1HMWH8wDbmrBoGcPdvArfcw** 提取码：6666