JavaSE:

- 小知识:
- 编程语言的特性:
  - 编译型和解释型:
- 动态类型语言和静态类型语言:
- 动态语言和静态语言:
- Java 第一个代码
- Java的变量:
  - 变量的概念:
  - 变量初始化:
  - 标识符必须遵守规则:
- Java的数据类型:
  - 基本数据类型:
    - 整型:
    - 浮点型:
    - 布尔型:
    - 字符型:
  - 引用数据类型:
  - 类型转换:
    - 隐式类型转换:
    - 强制类型转换:
  - 需要特别注意的问题总结:
- 运算符:
  - 算术运算符:
  - 关系(比较)运算符:
  - 逻辑运算符:
  - 位运算符:
  - 三目运算符:
- 运算符优先级:
  - 需要特别注意的点:
- 程序控制流程:
  - 顺序:
  - 选择结构:
    - if语句:
    - switch case 语句:
    - 循环结构:
      - for语句:
      - while语句:
      - do while语句:
  - 需要特别注意的点:
- 函数(function):
  - Java中如何定义函数:
  - 函数调用:
  - 函数分类:
    - 通过参数分类:
    - 通过返回分类:
    - 形参:

- 接口:
  - 接口的定义和实现:
  - 接口多实现多继承:
  - default关键字:
    - default冲突:
      - 一:
      - 二:
      - 三:
  - 接口与抽象类的区别
    - 相同点:
    - 区别:
    - 二者的选用:
  - 标记接口:
  - 接口注意事项:
  - 静态导包:
- 异常(类)(Exception):
  - 异常分类:
  - 异常类继承树:
  - 异常处理:
    - throw和throws
    - 抓捕异常:
    - 抛出异常:
  - finally关键字:
    - finally中写什么:
    - finally特殊用法:
  - 自定义异常类:
- 包装(封装)类:
  - 拆箱和装箱功能:
    - 拆箱:
    - 装箱:
    - 自动拆装箱:
  - Void包装类:
    - 用途:
- Object类:
  - native关键字:
  - 对象拷贝:
    - 浅拷贝:
    - 深拷贝:
      - 所有相关对象都实现浅拷贝:
      - 通过序列化对象实现深拷贝:
- String类:
  - 字符串拼接问题:
- StringBuilder和StringBuffer:
  - StringBuilder:
  - StringBuffer:
  - 两者区别:
- 时间对象:
  - 日期和时间的格式化编码
  - Date:

- 字节流:
  - InputStream(抽象类):
  - FileInputStream:
  - OutputStream(抽象类):
  - FileOutputStream:
  - FilterInputStream & FilterOutputStream:
    - BufferedInputStream & BufferedOutputStream:
    - DataOutputStream:
    - DataInputStream:
- 字符流:
  - Reader(输入流):
- Writer(输出流):
- BufferedReader & BufferedWriter:
  - PrintWriter(打印输出流):
- 转换流:
  - InputStreamReader:
  - OutputStreamWriter:
- 对象流:
  - ObjectInputStream:
  - ObjectOutputStream
  - Properties:

多线程编程:

- 多任务:
- 如何创建线程:
  - 继承Thread类, 重写run方法
  - 实现Runnable接口:
  - 实现Callable接口:
  - 线程池:
    - Executors:
      - 单例线程池(SingleThreadExecutor):
      - 固定大小线程池(FixedThreadPool):
      - 缓存线程池(CachedThreadPool):
      - 延迟任务线程池(ScheduledThreadPool):
      - 抢占线程池(WorkStealingPool):
      - 单例延迟线程池(SingleThreadScheduledExecutor):
      - 原始线程池(ThreadPoolExecutor):
    - 队列排队详解
    - 工作队列BlockingQueue详解
    - 线程池关闭
    - 线程池流程:
    - 线程池监控:
  - ThreadLocal对象:
  - Thread类方法:
  - 继承Thread实现线程和实现Runnable接口实现线程的区别:
- 结束线程的正确方式:
- 解决线程安全问题:
  - 通过加锁:
    - synchronized关键字:
    - Lock锁(接口):

多变量使用枚举:
数组变量:
设置注解的作用范围:
在自定义注解中的使用例子:
使用反射读取RUNTIME保留策略的Annotation信息的例子:
使用反射读取RUNTIME保留策略的Annotation信息的例子:
限定注解的使用:
注解的使用限定的例子:
在帮助文档中加入注解:
在注解中使用继承:
配置文件数据解析:
Properties:
JSON(JavaScript Simple Object Nation):
格式:
将对象转换为JSON:
gson:
XML(可扩展性标记语言):
格式:
对象和XML的转化:
dom4j:
爬虫(Spiders):
URL实现:
https实现:
使用第三方httpsClient:
Lambda表达式:
Lambda语法:
显式和隐式lambda表达式:
显式:
隐式:
省略参数类型:
单参数:
无参数:
final修饰符:
Lambda表达式主体:
Lambda类型推断:
Lambda行为参数化:
行为参数化模糊性:
函数式接口:
函数式接口定义:
@FunctionalInterface注释:
通用函数式接口:
Buildin函数式接口:
Function:
BiFunction:
Predicate:
BiPredicate:
Consumer:
BiConsumer
Supplier:

JavaSE:

JDK : Java开发工具包, 包含JRE, 编译器, 反编译器, 调试器等

JRE : Java运行环境, 运行JAVA程序所必须的环境的集合, 包含JVM标准实现及Java核心类库, 主要由Java虚拟机、Java平台核心类和若干支持文件组成. JRE不包含开发工具、编译器、调试器以及其他工具

JVM : Java虚拟机, JVM是执行代码并为该代码提供运行时环境的软件程序的规范. 运行Java代码

小知识:

- java内存管理中有一个小整型缓冲区(-128--127)
- 手动触发gc, System.gc
- 常量相加会先去常量区找对应结果, 找不到再新建
- 只要+拼接时, 存在已经定义的变量, 则会调用StringBuilder进行拼接操作, 所以在定义字符串时, 尽量不应使用变量

JDK : Java开发工具包, 包含JRE, 编译器, 反编译器, 调试器等

JRE : Java运行环境, 运行JAVA程序所必须的环境的集合, 包含JVM标准实现及Java核心类库, 主要由Java虚拟机、Java平台核心类和若干支持文件组成.
JRE不包含开发工具、编译器、调试器以及其他工具

JVM : Java虚拟机, JVM是执行代码并为该代码提供运行时环境的软件程序的规范. 运行Java代码

编程语言的特性:

Java是一门面向对象编程语言, 既是解释型语言也是编译型, 静态语言, 静态类型语言, 强数据类型语言

编译型和解释型:

类型	描述	语言	优点	缺点
编译型	需通过编译器(compiler)将源代码编译成机器码, 之后才能执行的语言. 一般需经过编译(compile)、链接(linker)这两个步骤. 编译是把源代码编译成机器码, 链接是把各个模块的机器码和依赖库串连起来生成可执行文件	C, C++, Pascal, Object-C, swift, java	编译器一般会有预编译的过程对代码进行优化. 因为编译只做一次, 运行时不需要编译, 所以编译型语言的程序执行效率高, 可以脱离语言环境独立运行	编译之后如果需要修改就需要整个模块重新编译. 编译的时候根据对应的运行环境生成机器码, 不同的操作系统之间移植就会有问题, 需要根据运行的操作系统环境编译不同的可执行文件
解释型	解释性语言的程序不需要编译, 相比编译型语言省了道工序, 解释性语言在运行程序的时候才逐行翻译	JavaScript, Python, Erlang, PHP, Perl, Ruby	有良好的平台兼容性, 在任何环境中都可以运行, 前提是安装了解释器(虚拟机). 灵活, 修改代码的时候直接修改就可以, 可以快速部署, 不用停机维护	每次运行的时候都要解释一遍, 性能上不如编译型语言
混合型	比如C#, C#在编译的时候不是直接编译成机器码而是中间码, .NET平台提供了中间语言运行库运行中间码, 中间语言运行库类似于Java虚拟机. .net在编译成IL代码后, 保存在dll中, 首次运行时由JIT在编译成机器码缓存在内存中, 下次直接执行	Java	-	-

动态类型语言和静态类型语言:

动态类型语言: 在运行期间才去做数据类型检查的语言, 说的是数据类型. 动态类型语言的数据类型不是在编译阶段决定的,
而是把类型绑定延后到了运行阶段, 例如Python、Ruby、Erlang、JavaScript、Swift、PHP、Perl

静态类型语言: 静态类型语言的数据类型是在编译期间(或运行之前)确定的, 编写代码的时候要明确确定变量的数据类型,
例如C、C++、C#、Java、Object-C

动态语言和静态语言:

这是完全不同于动/静态类型语言

动态语言: 在运行时可以改变其结构的语言. 例如新的函数、对象、甚至代码可以被引进, 已有的函数可以被删除或是其他结构上的变化.
通俗点说就是在运行时代码可以根据某些条件改变自身结构, 例如: Object-C、C#、JavaScript、PHP、Python、Erlang

静态语言: 与动态语言相对应的, 运行时结构不可变的语言就是静态语言, 例如: Java、C、C++

Java 第一个代码

```
public class Test {
    public static void main(String[] args) {
        // args 这个参数用于脚本传参
        System.out.println("Hello World!");
    }
}
```

// 代码先按照这个模板写, 上面的所有东西会慢慢解释
需要注意的几点:

1. 任何类名首字母都必须大写 (为了规范)
2. 如果class为public, 必须与文件名相同 (这点后面会讲到)
3. main函数必须是public, 否则会找不到执行的入口

Java的变量:

变量的概念:

变量表示内存中的一个存储区域, 该区域用来不断的存放同一类型的常量数据, 并可以重复使用这个区域. 并且这个区域有自己的名称(变量名)和类型(数据类型)

变量初始化:

变量使用注意 变量在使用时, 必须先给变量空间存放数据, 即初始化. 同时变量也有自己的生命周期

1. 数据类型 变量名称 = 变量值;
2. 数据类型 变量名称;
   变量名称 = 变量值;

例如:

```
class Test {
    void test() {
        // 第一种
        int a = 1;
        // 第二种
        int b;
        b = 2;
    }
}
```

标识符必须遵守规则:

1. **只能由 大小字母, 数字, 下划线, $, 这些有效符号组成**
   java是一门大小写敏感的编程语言
2. **数字不能开头**
3. 不能以关键字或保留字作为标识符名称
4. 变量尽量有意义
5. 如果标识符由多个单词组成时, 尽量使用驼峰法或者下划线法

   > 1.小驼峰法: 首字母小写, 其他单词的字母大写
   >
   > 2.大驼峰法: 首字母全部大写(Java中类名称)
   >
   > 3.下划线法: 将所有单词使用下划线连接到一起

   示例:

   ```
   class Test {
      String studentName;
      String StudentName;
      String student_name;
   }
   ```

6. 在定义类, 变量, 方法时, 尽量不要和系统已经内置好的名称冲突

所有代码都是被加载到内存中运行

Java的数据类型:

Java的数据类型分为两大类型: 基本数据类型, 引用数据类型

基本数据类型:

不用开发者手动创建, 定义, 而是直接可以使用的类型

函数体中的局部变量存储在栈中, 类中的方法外的变量随类示例化的对象一起存储在堆中

基本数据类型发生值传递, 相当于一种拷贝

整型:

> byte 字节 // 1个字节
>
> short 短整型 // 2个字节
>
> int 整型 // 4个字节
>
> long 长整型 // 8个字节

注意: 在Java中, 整数默认类型都是int, 小数默认类型是double
例如

```
class Test {
    void test() {
        short a = 10;
        a = a + 10; // 会报错, 因为计算时是int整形计算, 结果为一个int类型,
        //int类型不可赋值给short类型
// 正确:
        a = (short) (a + 10);
    }
}
```

注意: long a = 129l (或者long a = 129L, 大小写无所谓), 此时a为长整形; 如果 long a = 129则是由整形转换成长整型,
同样用float定义时应该float a = 1.0f

浮点型:

> float 单精度浮点数 // 4个字节
>
> double 双精度浮点数 // 8个字节

布尔型:

> boolean // 大小由环境决定
>
> 只有两个值
>
> true 真
>
> false 假

字符型:

> char // 2个字节
>
> // 使用单引号, 只能有一个字符

示例

```
class Test {
    char s = 'a';    // 正确定义
    char c = 'ab';  // 错误定义, char只能接受一个字符, 多个字符用String 定义
}
```

引用数据类型:

对象, 存储在堆中, 变量存储的是引用数据类型的地址哈希值

引用数据类型发生引用传递, 变量存储的地址哈希值修改为堆中另一个对象的地址哈希值, 相当于指针

> String 字符串
>
> Array 数组

> List 列表
>
> Set 集合
>
> Map 哈希表
>
> Object 对象
>
> 自定义类和对象

类型转换:

隐式类型转换:

当在存储整数数据时, Java中所有的整数默认都是int类型的. 所以在将-128~127之间的数据存储在 byte类型空间中时,
JVM会把这个int类型的数据自动转换成byte类型, 然后存放在byte空间中, short同理

例如:

```
class Test {
    byte b = 2; //会自动把int类型2转为byte类型存放在b空间 short s =234; //也会把int类型234转为short类型存放在s空间

    //当把一个int类型的数据存放在char类型空间中时,  也会发生隐式类型转换
    char ch = 65; //这里会用int类型65,  到编码表中查找对应的字符,  然后将字符存放在ch空间中 char ch2 = 'A';//直接将A字符存放在ch2空间中
}
```

1. 整型 和 浮点型 运算, 整型会自动向浮点型转换
2. 整型, 字符型, 浮点型的数据在混合运算中相互转换, 转换时遵循以下原则
   容量小的类型可自动转换为容量大的数据类型(类型自动提升)

   byte,short,char → int → long → float → double byte, short, char之间不会相互转换, 他们在计算时首先会转换为int类型整数默认转换类型是int
3. 浮点数默认转换类型是double
4. 注意: flat a = 3.14和long b = 200, 这样是错误写法, 正确写法为: float a = 3.14f, long b = 200L

   解释: java中小数默认是double类型, 整数默认是int类型

```
// 错误:
class Test {
    long a = 10;
    float b = 3.14;
    // 正确为:
    long a = 20L;
    float b = 3.14F;
}
```

强制类型转换:

容量大的数据类型转换为容量小的数据类型时, 要加上强制转换符, 但可能造成精度的降低或溢出, 使用时要格外注意

1. 在Java中, 低类型可以自动向高类型转换, 这就是上面的自动类型转换
2. 如果需要高类型向低类型转换, java不允许完成自动转换, 会报错
3. 如果确实需要这个操作, 则需要强转
   需要的类型 变量 = (需要的类型)变量或者值

示例:

```
class Test {
    long a = 123L;
    int b = (int) a;
    double c = 3.14;
    float d = (float) c;
}
```

需要特别注意的问题总结:

1. 在Java中, 整数默认类型都是int, 小数默认类型是double
2. flat a = 3.14和long b = 200, 这样是错误写法, 正确写法为: float a = 3.14f, long b = 200L

   解释: java中小数默认是double类型, 整数默认是int类型

```
class Test {
    // 错误:
    long a = 10;
    float b = 3.14;
    // 正确为:
    long a = 20L;
    float b = 3.14F;
}
```

```
class Test {
    void test() {
        short a = 10;
        a = a + 10; // 会报错, 因为计算时是int整形计算, 结果为一个int类型,
        //int类型不可赋值给short类型

        // 正确写法:
        short a = 10;
        a = (short) (a + 10);
    }
}

```

运算符:

算术运算符:

+
-
* : 乘
/ : 除
% : 取模

++ : 自增1
- - : 自减1
前++和前--优先级仅次于'()', 后--和后++优先级仅高于'='

```
class Test {
    public static void main(String[] args) {
        int a = 0;
        a = ++(a++); // 会报错, 因为括号内是一个数值, 自增自减运算符用于变量
    }
}
```

关系(比较)运算符:

A = 1;
B = 2;

运算符	描述	结果
==	检查两个操作数的值是否相等, 如果相等则条件为真	(A == B) 为假
!=	检查两个操作数的值是否相等, 如果不相等则条件为真	(A != B) 为真
>	检查左操作数的值是否大于右操作数的值, 如果是则条件为真	(A > B) 为假
<	检查左操作数的值是否小于右操作数的值, 如果是则条件为真	(A < B) 为真
>=	检查左操作数的值是否大于或等于右操作数的值, 如果是则条件为真	(A >= B) 为假
<=	检查左操作数的值是否小于或等于右操作数的值, 如果是则条件为真	(A <= B) 为真

返回boolean类型

逻辑运算符:

同数学的逻辑运算, 这里不做解释

|| : 逻辑或运算符
&& : 逻辑与运算符
& : 与运算符
| : 或运算符

// 以上两个也可以用在逻辑运算中, 效果等同上面两个
! : 逻辑非运算符

& 与 &&的区别:

> && 和 ||, 叫短路与和短路或
>
> 如果前面的条件已经可以得到结果, 则不会继续向后判断, 效率较高
>
> 而单与(或), 即便已经得到结果也会继续向后判断, 直到所有判断束才返回结果
>
> 编程一般用&&, ||

逻辑非 (!)　>　算术　>　逻辑与 (&&)、逻辑或 (||)　>　赋值 (=)

位运算符:

A = 60
B = 13

运算符	描述	运算规则
&	按位与操作, 按二进制位进行"与"运算	0&0=0; 0&1=0; 1&0=0; 1&1=1; (A & B) 将得到 12, 即为 0000 1100
\		按位或运算符, 按二进制位进行"或"运算
^	异或运算符, 按二进制位进行"异或"运算	0^0=0; 0^1=1; 1^0=1; 1^1=0; (A ^ B) 将得到 49, 即为 0011 0001
~	取反运算符, 按二进制位进行"取反"运算	~1=-2; ~0=-1; (~A) 将得到 -61, 即为 1100 0011, 一个有符号二进制数的补码形式
<<	二进制左移运算符. 将一个运算对象的各二进制位全部左移若干位(左边的二进制位丢弃, 右边补0	A << 2 将得到 240, 即为 1111 0000
>>	二进制右移运算符, 将一个数的各二进制位全部右移若干位, 正数左补0, 负数左补1, 右边丢弃	A >> 2 将得到 15, 即为 0000 1111
>>>	二进制有符号位移运算符(逻辑右移), 同上面的>>, 只是算上符号位	此运算符需注意, 只对32位和64位数据有意义

赋值运算符和二元运算符:

= : 赋值符
+= : 先加后赋值 , byte a = 0; a += 2相当于byte a =0; a = (byte)(a + 2), 下同
-= : 先减后赋值
/= : 先除后赋值
%= : 先取余后赋值
*=: 先乘后赋值

二元运算符隐含一个强制类型转换

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

        byte a = 10;
        byte b = 20;
        a += 10; // a = 20, 自动进行强制类型转换
        b = b + 10; // 会报错, b被提升为int类型, 运算结果为int类型, int不可赋值给byte
    }
}
```

三目运算符:

表达式 ? a : b
相当于一个if esle 语句
示例:

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

        int a = 1;
        int b = 2;
        int c = 0;
        c = a > b ? a : b;  // c的值为2
// 相当于:
        if (a > b) {
            c = a;
        } else {
            c = b;
        }
    }
}
```

运算符优先级:

优先级	运算符	结合性
1	()、[]、{}	从左向右
2	!、+、-、~、++、--	从右向左
3	*、/、%	从左向右
4	+、-	从左向右
5	<<、>>、>>>	从左向右
6	<、<=、>、>=、instanceof 从左向右	
7	==、!=	从左向右
8	&	从左向右
9	^	从左向右
10	|	
11	&&	从左向右
12	||	||
13	?:	从右向左
14	=、+=、-=、*=、/=、&=、	=、^=、~=、«=、»=、>>>=

需要特别注意的点:

1. & 和 | 可以用于逻辑运算

> & 与 &&的区别:
>
> && 和 ||, 叫短路与和短路或
>
> 如果前面的条件已经可以得到结果, 则不会继续向后判断, 效率较高
>
> 而单与(或), 即便已经得到结果也会继续向后判断, 直到所有判断束才返回结果
>
> 编程一般用&&, ||

3. 二元运算符自带一个强制类型转换
4. 前++和前--优先级仅次于'()', 后--和后++优先级仅高于'='
5. 自增自减用于变量, 不能用于数值和常量

```
class Test {
    public static void main(String[] args) {
        int a = 0;
        a = ++(a++); // 会报错, 因为括号内是一个数值, 自增自减运算符用于变量
    }
}
```

程序控制流程:

顺序:

从上到下, 从左到右执行

选择结构:

if语句:

```
class Test {
    public static void main(String[] args) {
        if (条件) {

        } else if (条件) {

        } else if (条件) {

        } else {

        }
    }
}
```

```
class Test {
    public static void main(String[] args) {
        int a = 2;
        if (a == 1) {
            System.out.println("a的值为1");
        } else if (a == 2) {
            System.out.println("a的值为2");
        }
// 结果为: a的值为2

        int b = -1;
        if (b > 0) {
            System.out.println("b大于0");
        } else if (b == 0) {
            System.out.println("b等于0");
        } else {
            System.out.println("b小于0");
        }
    }
}
// 结果为: b小于0
```

switch case 语句:

```
class Test {
    public static void main(String[] args) {
        String variable = "12";
        String constant_1 = "121";
        String constant_2 = "12";
        switch (variable) {
            case constant_1: // 如果variable 和 constant_1相等, 则执行case中的代码
                // 如果不想让它继续向后运行, 则使用break关键字中断
            case constant_2:
                int a;
                break;
            default:
        }
    }
}
```

匹配成功后执行case之后的所有代码, 可以不要break和default但必须看实际情况, default中的语句除非已经break跳出switch否则都会执行一次,
执行完所有满足条件的语句或遇到break时跳出

示例:

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

        int a = 1;
        // 不加break的情况
        switch (a) {
            case 1:
                System.out.println("a的值为1");
            case 2:
                System.out.println("a的值为2");
        }
        /* 结果为:
           a的值为1
           a的值为2
        */
        // 显然不是我们想要的结果

        // 加上break
        int b = 1;
        switch (b) {
            case 1:
                System.out.println("b的值为1");
                break;
            case 2:
                System.out.println("b的值为2");
```

```
                    break;
            }
            /* 结果为:
              b的值为1
            */
        }
}
```

注意: switch不可匹配long类型, 可匹配 char, byte, short, int, String

循环结构:

for, while, do while

for语句:

> for ([表达式1]; [表达式2(执行条件)]; [表达式3]){
> }

```
// 打印十个hello world
class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.println("hello world");
        }
    }
}
```

while语句:

> while (条件) {
> }

```
class Test {
    public static void main(String[] args) {
        // 1到100累加
        int i = 1;
        int sum = 0;
        while (i <= 100) {
            sum += i;
            i++;
        }
    }
}
```

do while语句:

> do {
> }while (条件); // 条件不成立也会执行一次// 主要用于人机交互模式的情况下

```
class Test {
    public static void main(String[] args) {
// 一个输出hello world的程序, 输入0退出
        Scanner input = new Scanner(System.in); /* 这是java的输入方式,
需要在代码最前面先写上import java.util.Scanner, 以后讲面向对象的时候会详细解释*/
        int a = 1;
        do {
            System.out.println("hello world");
            System.out.println("要继续打印吗, 输入任意键继续, 如果不, 输入0退出");
            a = input.nextInt();
        }
        while (a != 0);
    }
}
```

需要特别注意的点:

1. swtich case语句, 匹配成功后执行case之后的所有代码, 可以不要break和default但必须看实际情况, default中的语句除非已经break跳出switch否则都会执行一次, 执行完所有满足条件的语句或遇到break时跳出
2. switch不可匹配long类型, 可匹配 char, byte, short, int, String
3. 要用死循环, 只需要让循环条件一直满足就可以
   例如

```
class Test {
    public static void main(String[] args) {
        while (true) {
        }
        for (; true; ) {
        }
        do {
        } while (true);
    }
}
```

函数(function):

函数是什么? 函数就是具有名称的特定功能代码的集合.

为什么使用函数? 提高代码的复用度

Java中如何定义函数:

```
访问修饰符 [static] 返回值类型 函数名称([参数列表]) {
// 函数体由一行或多行组成
// 可不写访问修饰符, 不写也是一种访问修饰符
// [return 返回值]
}
```

Java中, 函数必须定义在类中

```
    public class Demo {
    public static void main(String[] args) {
        //
    }

    // 定义一个函数
    public static void hellofunc() {
        // public(访问修饰符) 意思是公共, 公开函数让其他包可以调用此函数
        // java的函数是一个对象, static修饰的不需要实例化就可以调用, 如果没有static关键字就需要先实例化才能调用
        System.out.println("hello function");
    }
}
```

函数调用:

函数定义完成后, 不会自己执行, 要在main函数调用.

```
函数名称([实参列表]);
```

函数分类:

通过参数分类:

有参数函数

无参数函数

通过返回分类:

有返回值

无返回值

return 之后函数结束

形参:

函数中的参数或者说自变量叫做形参, 初值来源于函数调用时的实参, 只是规定了参数形式, 在调用之前不会分配内存

注意的点:

如果形参是基本数据类型, 那么发生值传递, 是引用数据类型发生引用传递

局部变量:

定义在函数内部的变量, 叫做局部变量, 局部变量的作用域只在当前函数有效, 存储在栈中

注意: java中没有全局变量, main函数中的变量也是局部变量, 只是比较特殊, 作用域还是在main函数本身, 但是生命周期长

main函数之外的局部变量必须初始化

Java函数调用本质:

函数本质是一个对象, 储存在堆 (heap)中

函数定义后存储在堆 (heap)中, 调用时会在栈 (stack)中创建一个栈帧临时地压入栈中, 调用完成后会立刻弹出栈

函数重载:

两个同名函数的参数的类型和个数不同会行程重载

意义在于扩展功能

```
class Test {
    public static void add(int a, int b) {

    }

    public static void add(double a, int b) {

    }

    public static void add(int a) {

    }
// 以上三个函数都形成重载

    // 此函数没有与上面第一个函数形成重载
    public static int add(int a, int b) {

    }
}
```

函数递归:

自己调用自己, 反复压栈

注意栈溢出

递归一定要先写跳出条件

下面是两个例子:

```
    // 求n项斐波那契数列的和
class Test {
    public static int Fibonacci(int n) {
        if (n == 0 || n == 1 || n == 2) {         // 结束条件
            return n;
        }
        return Fibonacci(n - 1) + Fibonacci(n - 2);         // 注意一定要用return来调用否则会溢出
    }
}
```

```
    // 递归实现二分查找
class Test {
    public static int halfSearch(int[] arr, int left, int right, int target) {
        if (left >= right) {
            return -1;
        }
        mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (target < arr[mid]) {
            return halfSearch(arr, left, mid - 1, target);
        } else if (target > arr[mid]) {
            return halfSearch(arr, mid + 1, right, target);
        }
        return -1;
    }
}
```

需要特别注意的问题:

1. 如果形参是基本数据类型, 那么发生值传递, 是引用数据类型发生引用传递
2. 递归一定要先写跳出条件, 防止栈溢出
3. java中没有全局变量, main函数中的变量也是局部变量, 只是比较特殊, 作用域还是在main函数本身, 但是生命周期长
4. main函数之外的局部变量必须初始化

Java如何定义常量:

> final 关键字

```
class Test {
    // 如后是常量定义, 建议使用public static 一起修饰
    public static final int a = 10;

    public static void main(String[] args) {
        final int b = 2; // 常量, 不可修改
    }
}

```

> final 修饰类时, 被修饰的类不可被继承
>
> 为了安全性, 很多类都被final修饰

常用类:

Math 类:

> Math下的所有方法和属性都是静态的

```
// 原型:
public final class Math extends Object {

}
```

Math常用方法和属性:

修饰符和类型	属性	描述
static double	PI	圆周率
static double	E	自然常数

修饰符	方法	描述
static	abs(a)	返回a的绝对值
static	cbrt(a)	返回a的立方根
static	ceil(a)	返回a的向上取整结果
static	floor(a)	返回a的向下取整结果
static	round(a)	返回a的四舍五入结果
static	max(a, b)	返回a和b中的最大值
static	min(a, b)	返回a和b中的最小值
static	pow(a, b)	返回a的b次方
static	random()	返回一个[0, 1)的随机小数

Random类:

> 没有属性, 只有方法

```
//原型
public class Random implements Serializable {

}
```

创建Random对象:

```
// 创建Random对象

// 需要导入的包

import java.util.Random;

class Test {
    public static void main(String[] args) {
        // 第一种
        Random r = new Random();
        // 第二种
        Random rr = new Random();
    }
}
```

返回值类型	方法	描述
int	nextInt(a)	返回[0, a)中的一个的int型整数
float	nextFloat(a)	返回[0, a)中的一个float型整数
以此类推	以此类推	以此类推

Scanner类:

> 标准输入I/O流

```
// 原型
public final class Scanner extends Object {

}
```

创建Scanner对象并用于输入:

```
// 导入

import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        //1.创建Scanner对象
        Scanner input = new Scanner(System.in);
        //2.接收一个数字
        int num = input.nextInt();
    }
}
```

返回值类型	方法	描述
String	next()	查找并返回来自此扫描器的下一个完整标记, 以输入的空格键、Tab键或Enter键等视为分隔符或结束符
int	nextInt()	接受一个数字返回输入的数字, 空格为分隔符, \n结束
String	nextLine()	返回输入的值, 以\n作为结束符
boolean	nextBoolean()	扫描解释为一个布尔值的输入标记并返回该值
Scanner	reset()	重置扫描器
void	close()	关闭扫描器

注意:

> 在连续的两个及以上nextInt()之后用nextLine(), nextLine()会得到一个空
>
> 解释:
>
> > 输入nextInt()以\n进行分隔, 输入完成后\n留在缓冲区, 而nextLine()也以\n进行分隔, 此时nextLine()会读取到上一个留在缓冲区的\n从而直接结束
>
> 解决方法:
>
> > 在nextInt()后用nextLine()吸收\n

String类:

> Java中所有数据类型都是类, Java是一门完全面向对象的语言
>
> 字符串: 字符连在了一起, 形成了字符串
>
> 字符串是一个常量

```
// 原型
public final class String implements java.io.Serializable, Comparable<String>,
CharSequence {

}
```

> 在Java中, 一般用""括起来, 引号的内部内容就是字符串
>
> 可以理解为 Sting a = "123", a中只是存了一个地址, "123"才是创建的字符串对象
>
> 字符串是常量, 是不可变数据类型, java将字符串存储在堆中的常量池中, 一旦被创建就不能被修改也不能被销毁
>
> > 注意: 这里的不可变是说每一个字符串对象创建都会被存储在堆中, 常量池中已经存在的字符串对象就不能修改,
> > 如果定义了一个字符串: String s = "123"(s并不是一个对象, 是一个变量, 指向对象"123"的内存地址), 这时候"123"
> > 这个字符串对象会存储在常量池,
> > 不可被修改, 如果再写一句, s = "112", 这时变量s会指向字符串对象"112"(s中存储的地址修改为"112"的地址), "112"会存储进常量池,
> > 这时"123"并没有被删除或修改也在常量池中
>
> 为了安全, 设计为不可继承

创建一个String对象:

```java
class Test {
    public static void main(String[] args) {
        // 定义一个字符串
        //方法一：
        String str_1 = "hello";

        //方法二：
        String str_2 = new String("hello");
    }
}
```

常见的字符串方法:

修饰符和返回值类型	方法	描述
char	charAt(int index)	返回下标index对应的字符
int	length()	返回字符串长度
int	codePointAt(int index)	返回下标对应的字符的unicode值
boolean	contains(CharSequence str)	当且仅当此字符串包含指定的 char 值序列时, 返回 true
byte[]	getBytes([Charset charset])	字符串转换成字节
boolean	endsWith(String suffix)	以suffix结尾返回true否则返回false
boolean	startsWith(String suffix)	判断是否以suffix开头
int	indexOf(String str)	返回第一个匹配到的下标
int	lastIndexOf(String str)	返回最后一个匹配到的下标
String	trim()	清除两边空格
String	toUpperCase()	转换为大写
String	toLowerCase()	转换成小写
String	valueOf(args)	相当于拼接, 可用于将数字转换成字符串
String[]	split(String regex[, int limit])	根据给定正则表达式的匹配拆分此字符串, limit是切割份数
static String	join(CharSequence delimiter, CharSequence... elements)	将字符串数组按照符号拼接
static String	copyValueOf(char[] data)	将字符数组转换为字符串
boolean	equals(Sting str)	两个字符串相等返回true, 否则返回false

	equals(Object anObject)	将此字符串与指定的对象比较
boolean	equalsIgnoreCase(String anotherString)	忽略大小写的equals
int	compareTo(String Str)	比较两个字符串的unicode值, 返回0则两个字符串相等, 指定数小于参数返回-1大于则返回1
int	compareToIgnoreCase(String str)	忽略大小写的compareTo
String	concat(String str)	返回拼接后的字符串, 相当于两个字符串相加
boolean	isEmpty()	只当length()等于0(字符串为空)返回true
String	replace(char oldChar, char newChar) replace(CharSequence target, CharSequence replacement	替换每个匹配到的字符,
返回替换后的字符串 替换每个匹配到的子字符串, 返回替换后的字符串		
String	substring(int beginIndex) substring(int beginIndex, int endIndex)	返回一个新的字符串, 它是此字符串的一个子字符串

> 注意: 字符串不要使用等号来判断是否相等
>
> > 在java中 == 判断的是值, 用来判断字符串则是判断字符串的地址值, 不能用来判断字符串, 因为字符串创建的方式有多种,
> > 不同的创建方式地址会不一样

```
class Test {
    public static void main(String[] args) {
        String s = new String("abc");     // 并不只是创建一个对象, 而是一个或者两个
// 字符串是一个常量, 如果常量池中没有"abc", 则会在堆中创建一个对象, 在常量池中创建一个"abc"对象, 如果常量池中有"abc"则创建一个
        String b = "abc";     // 有"abc"则不创建, 没有就创建一个
    }
}
```

Character类:

Character 类在对象中包装一个基本类型 char 的值. Character 类型的对象包含类型为

char 的单个字段

修饰符和返回值类型	方法	描述
static boolean	isDigit(char ch)	确定指定字符是否为数字
static boolean	isLetter(char ch)	确定指定字符是否为字母
static boolean	isLetterOrDigit(char ch)	确定指定字符是否为字母或数字
static boolean	isLowerCase(char ch)	确定指定字符是否为小写字母
static boolean	isSpaceChar(char ch)	确定指定字符是否为 Unicode 空白字符
static boolean	isUpperCase(char ch)	确定指定字符是否为大写字母

二叉树的三种遍历:

前序: 根 左 右

中序: 左 根 右

后序: 左 右 根

容器(collection):

> 容器一般是一种数据结构, 是用来解决多个数据保存和计算的容器

1. 线性表:

   数组, 栈, 队列, 链表
2. 哈希表:

   树, 图

数组:

> 数组是一种连续内存, 大小固定的线性表, 是一种数据结构, 用来解决大量数据的存储
>
> 数组就是一片地址连续且空间大小一致的存储空间(但是每个空间存的也是其他数据的地址).

数组的特点:

1. 大小固定
2. 连续内存地址
3. 存储的数据类型固定
4. 数组是保存在堆内存中, 因此数组是一个对象, 堆中的数据都叫对象

> 数组变量存的就是数组在堆内存中首元素的地址

> 数据通过角标来访问元素的具体计算方式是: 所要访问数据的首地址 + 角标 * 数据类型大小
>
> 创建数组时必须明确规定大小或内容

Java中如何定义数组:

> 第一种
>
> 数据类型[] 变量名称 = new 数据类型[size];
>
> 第二种
>
> 数据类型[] 变量名称 = new 数据类型[] {elem_0, elem_1, elem_2};
>
> 第三种
>
> 数据类型[] 变量名称 = {elem_0, elem_1, elem_2}

示例

```
class Test {
    public static void main(String[] args) {
        // 第一种
        int[] arr_0 = new int[3];
        // 第二种
        int[] arr_1 = new int[]{1, 2, 3};
        // 第三种
        int[] arr_2 = {1, 2, 3};
    }
}
```

数组的长度(元素的个数):

> 数组对象.length // 属性

访问数组元素:

> 通过下标来访问, 注意在编程中, 99%情况下, 下标都是从0开始
>
> 数组对象[下标]　　// 通过下标来访问数组元素

修改数组元素:

> 数组对象[下标] = 新值

遍历数组:

1. for()
2. while()
3. do while()
4. foreach // 循环加强, 是一种迭代容器操作

   for (数据类型 临时变量: 可迭代对象)

示例:

```
class Test {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3};
        for (int item : arr) {
            System.out.println(item);
        }
    }
}
```

数组排序:

值传递:

> 仅仅只拷贝一次值
>
> 常量都是值传递

引用传递:

> 通过内存地址指向堆中的数据

值传递和引用传递的区别:

```
class Test {
    public static void main(String[] args) {
        int a = 10, b = 20;
        int[] arr = {1, 2};
        change(a, b);
        System.out.printf("%d, %d", a, b);
        change(arr);
        System.out.printf("%d, %d", arr[0], arr[1]);
    }

    public static void change(int a, int b) {
        a = a ^ b;
        b = a ^ b;
        a = a ^ b;
    }

    public static void change(int[] arr) {
        arr[0] = arr[0] ^ arr[1];
        arr[1] = arr[0] ^ arr[1];
        arr[0] = arr[0] ^ arr[1];
    }
}

// 输出为:
```

```
// 10, 20
// 2, 1

// 解释：基本数据类型发生的是值传递，引用数据类型(对象)是引用传递
```

如何让基本数据类型使用引用传递:

> Java被设计成一个安全, 可管理的环境, Java HotSpot有一个后门, 提供了对低级别的, 对直接内存和线程的操作.
> 这个后门是—-sun.misc.Unsafe. 这个类在JDK中有广泛的应用, 例如, java.nio和java.util.concurrent. 很难想象在日常开发中使用这些危险的,
> 不可移植和未经校验的API. 然而,Unsafe提供一种简单的方法来观察HotSpot JVM内部的一些技巧.
>
> 获取Unsafe
> sun.misc.Unsafe这个类的访问是受限的, 它的构造方法是私有的, 相应的方法要求必须被Bootloader载入才能使用, 也就是说,
> 只有JDK内部才能使用这个方法来构造Unsafe对象.
>
> 我们可以写一个反射方法, 来获取Unsafe实例：

```
class Test {
    public static Unsafe getUnsafe() {
        try {
            Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);
            return (Unsafe) f.get(null);
        } catch (Exception e) { /* ... */ }
    }
}
```

算法的特性:

1. 算法的时间复杂度
2. 算法的空间复杂度
3. 算法的稳定性

三种基本排序:

冒泡:

时间: O(n ^ 2)

arr = [1, 100, -10, 50, 3, -5, 96] // arr.length = 7

大数上浮法:

- origin: 1, 100, -10, 50, 3, -5, 96
- step 1: 1, -10, 50, 3, -5, 96, 100 // max = 100
- step 2: -10, 1, 3, -5, 50, 96, 100 // max = 96
- step 3: -10, 1, -5, 3, 50, 96, 100 // max = 3
- step 4: -10, -5, 1, 3, 50, 96, 100 // max = 1

示例:

```java
// int[] arr = {1, 100, -10, 50, 3, -5, 96};
class Test {
    public static void bubbleSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

选择:

时间: O(n^2)

arr = [1, 100, -10, 50, 3, -5, 96] // arr.length = 7

> 最小值:
>
> 假设第一个值为最小值, 如果找到一个比第一个值还小的就交换, 直到找到真正的最小值交换这两个数

- origin: 1, 100, -10, 50, 3, -5, 96 // assume_min = 1
- step 1: -10, 100, 1, 50, 3, -5, 96 // assume_min = 100
- step 2: -10, -5, 1, 50, 3, 100, 96 // assume_min = 1
- step 3: -10, -5, 1, 50, 3, 100, 96 // assume_min = 50
- step 4: -10, -5, 1, 3, 50, 100, 96 // assume_min = 100
- step 5: -10, -5, 1, 3, 50, 96, 100

示例:

```java
// int[] arr = {1, 100, -10, 50, 3, -5, 96};
class Test {
    public static void selectSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            int min = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[j] < arr[min]) {
                    // j对应的下标才是最下值
                    min = j;
                }
            }
            if (min != i) {
                int temp = arr[i];
                arr[i] = arr[min];
                arr[min] = temp;
```

```
            }
        }
    }

    // 改进的选择排序, 增加稳定性, 时间复杂度和空间复杂度不变
    public static void selectSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] > arr[j]) {
                    int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
        }
    }
}
```

插入:

以第一个数为有序数组之后的所有元素依次插入到这个有序数组中

arr = [1, 100, -10, 50, 3, -5, 96]

- origin: 1, 100, -10, 50, 3, -5, 96
- step 1: 100, -10, 1, 50, 3, -5, 96
- step 2: -10, 1, 100, 50, 3, -5, 96
- step 3: -10, 1, 50, 100, 3, -5, 96
- step 4: -10, 1, 3, 50, 100, -5, 96
- step 5: -10, -5, 1, 3, 50, 100, 5
- step 6: -10, -5, 1, 3, 550, 100,

示例:

```
public class Test {
    public static void insertSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = i + 1; j > 0; j--) {
                if (arr[j] < arr[j - 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j - 1];
                    arr[j - 1] = temp;
                }
            }
        }
    }
}
```

查找算法:

针对有序序列

二分查找:

针对有序的序列, 可以直接和中间值比较

如果小于中间值, 那么目标数肯定在左边, 如果大于则在右边

示例:

```
class Test {
    // int[] arr = new int[n];
    public static int halfSearch(int target, int... arr) {    // int... arr 是可变参数, 可以接受多个值
        insertSort(arr);
        int head = 0, rear = arr.length - 1, mid = 0;
        while (head <= rear) {
            mid = head + (rear - head) >> 1;
            if (arr[mid] > target) {
                rear = mid - 1;
            } else if (arr[mid] < target) {
                head = mid + 1;
            } else {
                return mid;
            }
        }
        return -1;
    }
}
```

二维数组:

二维数组的定义:

数据类型[ ][ ] = new 数据类型[行数][列数]

例如:

```java
class Test {
    public static void main(String[] args) {
// 定义一个三行四列的二维数组

        int[][] arr1 = new int[3][4];

        int[][] arr2 = {{1, 2, 3, 4},
                {2, 3, 4, 5},
                {3, 4, 5, 6}
        };
    }
}
```

注意: arr1.length是行数, arr[0].length是列数

Arrays工具类:

描述符和返回值类型	方法	描述
static int	sort(参数)	排序, 底层实现为大数据快速排序, 小数据插入排序
static <T> List<T>	asList(T a)	返回一个列表
static double[]	copyOf(double[] original, int newLength)	从original[0]开始拷贝newLength个新数组, 如果超出长度, 超出长度的那部分填零
static double[]	copyOf(double[] original, int from, int to)	从from开始到to结束, 拷贝范围内的字符串
static void	fill(Object[] a[, int fromIndex][, int toIndex], Object val)	用指定值填充对象
static boolean	equals(Object[] a, Object[] a2)	比较是否是同一地址的对象, 并不比较值比较地址的hashcode, 可以自己重写此方法来比较值
static String	toString(Object[] a)	将数组内容以字符串形式返回

可变参数:

函数定义的时候, 可以在参数中, 使用 数据类型... 形参名

实际上是一个数组

IDE(集成开发环境):

eclipse:

由IBM开发, 在2000年后捐给了开源社区(eclipse基金会)

包(package):

java为管理源码, 使用了文件夹隔离不同的java文件, 这些文件夹就可以被称为包

例如java.util.Scanner的路径为java/util/Scanner

包的目的:

1. 管理项目, 管理代码
2. 防止冲突

包的创建遵循的规则:

公司域名倒着写

面向对象编程基础:

面向对象专业术语:

OO (Oriented Object) : 面向对象

OOP (Oriented Object Programming) : 面向对象的编程

OOD (Oriented Object Design) : 面向对象的设计

OOA (Oriented Object Analysis) : 面向对象的分析

OOT(Oriented Object Test) : 面向对象的测试

什么是面向过程:

面向过程也是解决问题的一种思想, 当我们在解决问题时, 会按照预先设定的想法和步骤, 一步一步去实现, 而具体的每一步都需要我们去实现和操作.
这些步骤相互调用和协作, 完成我们的需求.

上述描述的每一个具体步骤我们都是参与者, 并且需要面对具体的每一个步骤和过程, 这就是面向过程最直接的体现.

通过上面简单的描述发现, 面向过程, 其实就是面向着具体的每一个步骤和过程, 就是面对具体的每一个功能函数. 这些功能函数相互调用, 完成需求.

提出问题, 分解问题, 一一解决

实质上是函数的调用

什么是面向对象:

OOP(Oriented Object Programming):

当不再面对具体的每一个方法时, 发现操作也变的简单了很多. 而封装具体功能的这类, 是我们需要面对的. 而基于这个封装了具体功能的类, 那怎么使用呢?

当面向封装了具体功能类, 若要使用这个类, 一般情况下, 在Java中需要通过创建这个类的实体来 使用. 这个实体称之为对象. 在开发中,
我们是在不断的找封装不同功能的类. 基于这些类, 创建其对 象, 使用这些对象完成相应的操作.

通过上面的讲解和分析得出:面向对象是基于面向过程, 对象是将功能进行了封装. 只要找到了具体的类, 创建出对象, 就可以调用其中的具体功能. 面向对象也是用来解决问题的一种思维模式.

在以后开发中, 先找对象, 调用对象中的具体功能. 如果真的没有能够完成需求的对象, 这时就自己创建对象, 并将所需的功能定义到对象中, 方便以后使用.

面向对象的思想也就是人认识世界所采用的方式: 分门别类的思想:

世界上的万物人都进行了分类, 按照相似性把数不清的事物分归类到有限的类别中, 比如不管是百灵鸟还是麻雀都是鸟类,
每个人有姓名都是不一样的个体但都叫人类

类(class): 类别

在java中, 使用class关键字定义类

> 属性:
>
> > 类的固有特征
>
> 行为和动作:
>
> > 函数 function(方法 method)

对象(object): 一个类别中的具体案例

如何定义类:

类的成员:

1. 属性
2. 方法
3. 构造方法
4. 静态属性
5. 静态方法 // 静态方法就是类方法
6. 构造代码块
7. 静态代码块

访问修饰符 类型 属性名称 = 属性值;

```
1  public class Test {
2      public static void main(String[] args) {
3
```

```
    }
}

// 定义了一个狗类
class Dog {
    // 固有属性, 类的固有特征
    // 访问修饰符 类型 属性名称 = 属性值;
    public String name;
    public int age = 3;

    // 狗会跑
    public void run() {

    }

    // 狗会吃
    public void eat() {

    }

    // 构造方法
    public Dog() {

    }

}
```

局部变量与成员变量的区别:

定义位置不同:

- 成员变量定义在类中
- 局部变量定义在方法中或者语句里面

在内存中的位置不同:

- 成员变量存储在堆内存的对象中
- 局部变量存储在栈内存的方法中

声明周期不同:

- 成员变量随着对象的出现而出现在堆中, 随着对象的消失而从堆中消失
- 局部变量随着方法的运行而出现在栈中, 随着方法的弹栈而消失

初始化不同:

- 成员变量因为在堆内存中, 所有都是默认的初始化值
- 局部变量没有默认的初始值, 必须手动给其赋值才可以使用

如何创建对象:

定义类, 在根据类来创建对象

可以通过类的构造方法创建对象, 使用new关键字

类型 对象名称 = new 构造函数();

Scanner scan = new Scanner(System.in);

Random rand = new Random();

```
public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Dog dog_2 = new Dog();
    }
}
```

如何访问对象:

对象.属性

```
class Test {

    public static void main(String[] args) {
        Dog dog = new Dog();
        System.out.println(dog.name);
        Dog dog_2 = new Dog();
        dog2.name = "screl";
        System.out.println(dog2.name);
    }
}
```

成员的初始化默认值:

int, long, short, byte: 0

double: 0

char: '\u0000' 0

对象的默认值是 null

this 指针:

在java中, 存在this关键字, 这个关键字默认会指向当前对象

每一个对象都带有一个this指针

非静态的属性和方法不能在静态方法中使用, this只能用在非静态方法中, 不能用在静态方法中(类方法)

非静态的方法可以使用静态属性和方法

构造方法(Constructor):

作用: 创建对象的时候初始化值, 对对象进行针对性初始化

- 初始化过程: 默认初始化 -> 显示初始化 -> 针对性初始化

与类名相同, 没有返回值, 访问修饰符为public, 非static

如果声明了返回值jvm会将其解释为普通方法

在构造对象的时候自动调用构造方法

调用其他构造函数的语句必须定义在构造函数的第一行, 原因是初始化动作要最先执行

构造函数之间禁止出现相互调用的情况

注意:

1. 在java类中如果没有自己定义构造函数, jvm会自动生成一个无参的构造函数, 构造出来的成员的值为默认值, 当定义了一个构造函数之后,
   jvm就不会生成
2. 定义构造函数时, 一定要定义一个无参构造
3. 构造方法如果是private不能从类外部实例化

构造方法的细节:

- 一个类中可以有多个构造函数, 多个构造函数是以重载的形式存在的
- 构造方法中也是有return语句的, 用于结束初始化动作
- 构造方法是可以被private修饰的, 作用:其他程序无法创建该类的对象

```java
class Dog {
    public String name;
    public String color;


    // 自定义构造方法
    public Dog(String name, String color) {
        this.name = name;
        this.color = color;
    }

    // 一定要定义一个无参的构造方法
    public Dog() {

    }
}
```

this调用构造方法

在之前学习方法之间调用时, 可以通过方法名进行调用. 可是针对构造方法, 无法通过构造方法名来相互调用.

构造方法之间的调用可以通过this关键字来完成

构造方法调用格式:

> this(参数列表);

调用其他构造方法的语句必须定义在构造函数的第一行, 原因是初始化动作要最先执行.

构造方法之间禁止出现相互调用的情况.

```
class Dog {
    String name;
    int age;
    String master;

    public Dog() {

    }

    public Dog(String name, int age, String master) {
        this(master);    // 调用另一个构造方法
        this.name = name;
        this.age = age;
    }

    public Dog(String master) {
        this.master = master;
    }
}
```

构造代码块:

在类中可以用{ } 表示构造代码块

```
class Dog {
    {
        System.out.ptintln("这是一个构造代码块");
    }
}
```

对象的创建流程及内存结构一:

1. 在堆内存中开辟一块对象的内存空间, 并为其分配物理内存地址
2. 对成员变量进行 **默认初始化** (零值)
3. 相应构造函数进栈, 先对成员变量进行 **显式初始化**
4. 其次执行构造函数的内容, 对成员变量进行 **针对性初始化**
5. 构造函数执行完成弹栈, 将对象的内存空间地址赋予相应的引用变量

static 关键字:

static 关键字可以修饰(属性, 方法, 类, 语法块, 导包)

static 还可以修饰内部类

被static修饰的只能执行一次

static 修饰的代码(方法或者属性), 会提前加载到内存中, 并且只有一份属于该类, 因此:

1. 静态方法中, 无法直接调用非静态方法(或者属性)
2. 被static 修饰的属性和方法, 不属于对象, 属于该类, 直接可以使用类名称.属性(或者方法)来调用
3. 非静态的方法可以使用静态属性和方法
4. 在静态方法中调用非静态方法和非静态属性, 需要先实例化

静态内部类:

```
class Test {

    // 静态内部类:
    static class Test_2 {

    }

}
```

静态方法:

```
class Test {
    public static void function() {

    }
}
```

静态属性:

```
class Test {
    static int num;
}
```

静态代码块:

static { } 表示静态代码块, 只会执行一次

```
class Dog {
    static {
        System.out.println("这是一个静态代码块");
    }
}
```

对象的创建流程及内存结构(二)

1. javac命令编译Java源代码生成字节码文件
2. java命令将字节码文件加载进虚拟机
3. 字节码文件具体加载进方法区(非静态区和静态区)
4. JVM从静态区中寻找主函数main, 并将其加载进栈开始执行程序
5.
    5. 后续和(一)一样

静态变量与成员变量的区别

1. **变量所属不同:**
    - 静态变量所属与类, 也称为类变量
    - 成员变量所属于对象, 也称为实例变量(实例变量)
2. **内存中的位置**
    - 静态变量存储于方法区中的静态区中
    - 成员变量存储于堆内存中
3. **在内存中出现的时间**
    - 静态变量随着类的加载而加载, 随着类的消失而消失
    - 成员变量随着对象的创建而在堆内存中出现, 随着对象的消失而消失
4. **调用方式**
    - 静态变量可以用类直接调用, 也可对象调用
    - 成员变量只能被对象调用

成员的加载顺序:

静态属性最先运行, 然后是非静态属性, 然后是静态代码块, 然后是构造代码块, 再然后是构造方法, 再是静态方法, 最后是普通方法

重写函数:

在打印对象的时候自动调用toString()方法

打印一个继承Object的类, 不重写则默认调用的是Object的toString()方法, 打印出来的是地址的哈希值

用@override 标记重写

```
class Dog {
    public String name;
    public String color;

    // 构造函数
    public Dog(String name, String color) {
        this.name = name;
        this.color = color;
    }

```

```
    // 重写
    @override
    public String toString() {
        return this.name + this.color;
    }
}
```

Java(权限)访问修饰符:

注意: 只有 缺省 和 public修饰符能修饰类

public: 任何地方都可以正常访问

protected: 只能在同包, 同类, 同子类中访问 // 主要用于修饰方法, 为继承设计

缺省(默认): 只能在同包和同类中访问 // 不能被继承, 子类不能访问

private: 只允许当前类中访问 // 不能被继承

public 修饰类时:

加public表示全局类, 该类可以import到任何类内. 不加public默认为保留类, 只能被同一个包内的其他类引用

私有化构造函数:

1. 工具类(全都是静态方法或者静态属性)
2. 单例模式

单例模式:

某类创建对象只能且最多只有一个对象

将构造函数用private修饰, 让外界无法创建对象

在特殊情况下, 不允许外界创建对象, 减少堆内存占用

步骤:

1. 私有化构造函数, 杜绝外界创建对象
2. 内部创建该类对象(静态)
3. 提供公共接口向外提供该类的对象

饿汉式:

提前提供一个对象

缺点: 即便没有使用对象, 也从始至终常驻内存, 浪费内存

```
public class Singleton {
    private static Singleton singleton = new Singleton();

    private Singleton() {

    }

    public static Singleton getInstance() {
        return singleton;
    }
}
```

饱汉式:

访问的才提供对象

缺点: 线程不安全

```
public class Singleton {
    private static Singleton singleton;

    private Singleton() {

    }

    public static Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

内部类:

普通内部类

内部类可以直接访问外部类中的成员, 但外部类不能直接访问内部类, 若要访问, 必须创建内部类对象才能访问

内部类本身就是静态的, 用static修饰后类似于一个外部类了

创建一个内部类和对象:

```
class Test {
    class Inner {

    }
}

class Main {
    Inner inner = new Test().new Inner();
}
```

静态内部类

如果用static修饰内部类, 则该内部类属于该外部类, 但不属于外部类的对象

静态内部类可包括静态成员也可包括非静态成员. 根据静态成员不能访问非静态成员的规定, 所以静态内部类不能访问外部类实例成员,
只能访问外部类的静态成员.

```
class Test2 {
    static class Inner2 {

    }
}

class Main2 {
    Test2.Inner2 inner = new Test2.Inner2();
}
```

局部内部类:

在方法中创建局部内部类, 创建内部接口或者内部枚举也是一样:

```
class Outer {
    public void test() {
        class Inner {
            int a = 1;
        }
    }
}
```

方法中的内部类不能有访问修饰符

匿名局部内部类:

```
class Outer {
    int s = 12;

    public void test() {
```

```
        class Inner {
            int a = 1;
        }

        new Inner() {
            {
                System.out.println(a);
                System.out.println(s);
                execute();
            }

            public void execute() {
                System.out.println(s + 10);
            }
        };

    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();

    }
}
```

非静态内部类细节

1. 注意非静态内部类中不能定义静态成员变量和静态成员函数. 但可以定义静态成员常量. 原因常量在生成字节码文件时直接就替换成对应的数字.
2. 当内部类在外部类成员位置上被私有修饰, 在外部类以外的其他地方是无法访问的. 只能在外部类中访问
3. 有一个指针指向外部类
4. 在外部创建内部类的实例时, 需要Outer.Inner inner = new Outer().new Inner

以下怎么拿到num:

```
public class Test {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        in.show();
    }
}

class Outer {
    int num = 5;// 外部类的成员变量

    class Inner {
        int num = 6;// 内部类的成员变量

```

```
        void show() {
            int num = 7; // 内部类局部变量
            System.out.println("内部类局部num=" + num);
            System.out.println("内部类成员num=" + this.num);
            System.out.println("外部类成员num=" + Outer.this.num);
        }
    }
}
```

注意如果外部类的成员为static, 则内部类在访问外部类成员和方法时用外部类名.来访问

内部类本质上是外部类的成员, 所以外部类的private修饰符对内部类无效

为什么内部类可以直接访问外部类的成员, 那时因为内部类持有外部类的引用(外部类.this). 对 于静态内部类不持有外部类.this
而是直接使用 外部类名.

面向对象三大特征:

1. 封装
2. 继承
3. 多态

有人认为抽象也是一种特征, 但并不被公认

一. 封装(Encapsulation):

为了安全性, 将成员私有化, 不让外界访问, 同时提供特定的访问方式, 相当于是一个外部类

```
class Cat {
    // 首先私有化属性
    private int id;
    private String name;
    private int age;

    // 提供对应的公开访问方法
    // get方法:
    public int getId() {
        return this.id;
    }

    // set方法
    public void setId(int id) {
        this.id = id;
    }
}
```

POJO对象(Plain Ordinary Java Object)

1. 根据封装来写

2. 私有化属性
3. 提供公开的setter 和 getter 方法
4. 至少两个或者以上的构造方法

Java Bean对象:

1. 所有的成员变量都要使用 private 关键字进行修饰
2. 为每一个成员变量编写 set、get 方法
3. 创建一个无参数的构造方法
4. 创建一个有参数的构造方法

二. 继承(Inheritance):

在面向对象中, 类与类之间可以存在继承关系

Java是一门典型的单继承编程语言

Java中如何实现继承:

关键字 extends

父类(超类, 基类), 子类

private 修饰的属性和方法都是无法被子类继承的

protected 修饰的方法就是用于子类继承的

```
class RichMan() {
    public int money = 100000000;
    private String wife = "wife";

    private void play() {
        System.out.println("炒股");
    }

    protected void controlCompany() {
        System.out.println("掌控公司");
    }
}

class Son extends RichMan {
    public static void main(String[] args) {
        Son son = new SOn();        // son中的成员只有money, 方法只有controlCompany
        System.out.println(son.money);
        son.contralCompany();
    }
}
```

继承的功能:

减少代码的重复, 提高代码的复用度

方法重写(覆盖):

重写(OverWrite)

覆盖(OverRide)

发生在继承中, 指的是, 子类继承了父类方法后不满足使用, 就重写该方法以满足子类使用

重写是访问修饰符权限可以放大但不能缩小, 除了访问修饰符, 返回值和参数还有函数名都保持一致

注解:

注解	作业
@Override	标记覆盖(重写)
@Deprecated	标记过时
@SuppressWarnings	压制警告
@SuppressWarnings("all")	忽略全部类型的警告
@SuppressWarnings("unchecked")	忽略未检查的转化, 例如集合没有指定类型的警告
@SuppressWarnings("unused")	忽略未使用的变量的警告
@SuppressWarnings("resource")	忽略与使用Closeable类型资源相关的警告
@SuppressWarnings("path")	忽略在类路径, 原文件路径中有不存在的路径的警告
@SuppressWarnings("deprecation")	忽略使用了某些不赞成使用的类和方法的警告
@SuppressWarnings("fallthrough")	忽略switch语句执行到底没有break关键字的警告
@SuppressWarnings("serial")	忽略某类实现Serializable, 但是没有定义serialVersionUID的警告
@SuppressWarnings("rawtypes")	忽略没有传递带有泛型的参数的警告

```
class RichMan {
    public int money = 100000000;
    private String wife = "wife";

    public void play() {
        System.out.println("炒股");
    }
}

class Son extends RichMan {

```

```
12      @Override
13      public void play() {
14          System.out.println("健身");
15      }
16
17      public static void main(String[] args) {
18          Son son = new SOn();
19          son.play();
20      }
21  }
```

super关键字:

super 在java中, 是一个指针, 类似于this关键字

this关键字指向创建的每一个对象

super会自动指向父类

super( ); // 调用父类的无参构造, 默认在构造函数的第一行, 早于this( ); 可以不写, 但是写就必须写在构造函数的第一行,
写this()就不能写this()

final关键字:

final : 最后

1. 被他修饰的变量就是常量

   static final

2. **用final修饰的方法不能被重写**
3. **final关键字修饰类则该类不能被继承**

为什么类用final修饰:

基于jvm的安全性考虑, 防止被继承之后底层代码被篡改从而引起各种安全问题

对象的创建流程及内存结构三:

1. javac命令将源码(.java)进行编译, 生成字节码文件(.class)
2. java命令执行字节码文件
3. 将字节码文件加载进虚拟机, 具体方法进入方法区(非静态区和静态区)
4. JVM从静态区中寻找main函数, 并将其加载进栈开始执行程序
5. main 函数开始执行, 创建对象代码, 如Son son = new Son();
6. 在堆内存中开辟对对象的内存空间, 并分配地址
7. 静态代码块执行
8. 创建成员变量开始初始化
9. 代码块执行
10. 子类构造函数从非静态方法区加载进栈开始执行
11. 第一句先执行父类的构造函数
12. 父类构造函数执行, 为子类继承到的成员变量进行初始化(对象内存空间里的父类空间)
13. 父类构造函数弹栈执行完成

14. 子类构造函数继续执行
15. 再执行子类构造函数的内容, 进行针对性初始化
16. 执行完成, 子类构造函数弹栈, 将对象的内存空间地址赋予相应的引用变量

instanceof 关键字 判断对象是否为对应类, 配合继承判断

三. 多态(Polymorphism):

在继承的基础上才有多态

父类应用指向子类示例

Animal cat = new Cat( );

访问不到子类自身的方法, 但可以重写父类方法来满足子类使用

多态的好处

提高了程序的扩展性.

多态的弊端

通过父类引用操作子类对象时, 只能使用父类中已有的方法, 不能操作子类特有的方法.

多态的前提

1. 必须有关系:继承, 实现
2. 通常都有重写操作

当父类的引用指向子类对象时, 就发生了向上转型, 即把子类类型对象转成了父类类型. 向上转型的好处是隐藏了子类类型, 提高了代码的扩展性.

```
public class Test {
    public static void main(String[] args) {
        Animal cat = new Cat("小黄", "猫");
        Animal dog = new Dog("小红", "狗");
        PetStore store = new PetStore();
        store.wash(cat);
        store.wash(dog);
    }
}

@Setter
@Getter
class Animal {
    String name;
    String type;

    public Animal() {

    }

    public Animal(String name, String type) {
```

```
        this.name = name;
        this.type = type;
    }

    public void beWash() {
        // 这段函数体没意义只用来被重写, 可以用抽象方法从而省去写这个函数
    }

}

class Dog extends Animal {

    public Dog() {

    }

    public Dog(String name, String type) {
        super(name, type);
    }

    @Override
    public void beWash() {
        System.out.printf("%s正在被洗, 是一只%s\n", this.name, this.type);
    }
}

class Cat extends Animal {

    public Cat() {

    }

    public Cat(String name, String type) {
        super(name, type);
    }

    @Override
    public void beWash() {
        System.out.printf("%s正在被洗, 是一只%s\n", this.name, this.type);
    }
}

class PetStore {

    public void wash(Animal pet) {
        pet.beWash();
    }
}
```

抽象:

抽象方法:

如果一个方法, 不需要实现体(函数内容), 就可以声明抽象方法

抽象方法: 没有方法体的方法, java中使用abstract关键字声明的方法

访问修饰符 abstract 返回值类型 方法名称

抽象方法必须写在抽象类中

抽象类:

被abstract关键字声明的类, 不能直接实力化

访问修饰符 abstract class 类名

抽象父类不写抽象方法, 则子类需要写, 但抽象子类可以不写, 留给子类以后的孙类解决

> 注意: 抽象类中不一定有抽象方法, 有抽象方法就肯定是抽象类
>
> 如果一个类继承了抽象类, 就必须实现抽象方法, 如果一定不实现, 则这个子类也声明为抽象类
>
> 抽象类不可以创建示例, 原因: 调用抽象方法没有意义

存在的意义是为了继承给子类, 通过多态调用子类对抽象方法的实现

抽象关键字abstract不可以和哪些关键字共存?

1. final: final修饰的类是无法被继承的, 而abstract修饰的类一定要有子类. final修饰的方法无法 被覆盖, 但是abstract修饰的方法必须要被子类去实现的
2. static: 静态修饰的方法属于类的, 它存在与静态区中, 和对象就没关系了. 而抽象方法没有 方法体, 使用类名调用它没有任何意义
3. private: 私有的方法子类是无法继承到的, 也不存在覆盖, 而abstract和private一起使用修饰 方法, abstract既要子类去实现这个方法,
   而private修饰子类根本无法得到父类这个方法. 互相矛盾

接口:

接口本质是特殊的类, 特殊抽象类

接口中所有方法都是没有实现的(抽象方法)

接口中的所有成员都是静态常量

在jdk8之前: 接口中的所有属性方法, 默认都是public的, 即使没有用public修饰

java中使用interface关键字来定义接口

接口无法示例化, 必须使用子类, 以多态的形式完成实现

java中, 接口可以多实现多继承

接口的定义和实现:

```
public interface TestInterface {
    public void say();

    void play();
}

public class MyInterface implements TestInterface {

    @Override
    public void say() {
        // 代码
    }

    @Override
    public void play() {
        // 代码
    }
}
```

在jdk8中接口可以定义普通方法, 但也只能通过多态的方式访问, 用default关键字修饰

```
public interface Test {
    void show();

    public default void func() {
        System.out.println("执行");
    }
}
```

也可定义静态方法, 相当于把接口做了工具类

```
public interface Test {
    void show();

    public static void func2() {
        System.out.println("执行");
    }
}
```

接口多实现多继承:

```
interface A {
    void show1();
}

```

```java
interface B {
    void show2();
}

// 接口多实现
class test implements A, B {
    public void show1() {
    }

    public void show2() {
    }
}

// 接口多继承
interface C extends A, B {
    void show3();
}
```

default关键字:

在接口中可以使用default关键字来声明一个默认方法, 例如:

```java
interface Test {
    int size();

    default isEmpty() {
        return size() == 0;
    }
}
```

实现了该接口的类都会继承这个默认方法, 就不需要再实现

default冲突:

一:

如果有两个接口定义了相同的default方法, 例如

```java
interface Test1 {
    default String getName() {
        return null;
    }
}

interface Test2 {
    default String getName() {
        return null;
    }
}
```

如果此时有一个类同时实现上面接口就会产生冲突, 需要复盖冲突的方法

```
public class Main implements Test1, Test2 {

    @Override
    public String getName() {
        return Test2.super.getName(); // 或者 return Test1.super.getName();
    }
}
```

二:

一个接口中的default方法和另一个接口的抽象方法相同, 那么当一个类同时实现这两个接口时会实现抽象方法而放弃default方法

三:

一个父类的方法和另一个接口的抽象方法相同, 那么当一个类继承这个父类并实现这个接口时, 接口的默认方法会被忽略而只考虑超类,
者也称作类优先原则

接口与抽象类的区别

相同点:

- 都位于继承的顶端,用于被其他实现或继承
- 都不能实例化
- 都包含抽象方法,其子类都必须覆写这些抽象方法

区别:

- 抽象类为部分方法提供实现,避免子类重复实现这些方法,提供代码重用性
- 接口只能包含抽象方 法; 一个类只能继承一个直接父类(可能是抽象类),却可以实现多个接口(接口弥补了Java的单继承)

二者的选用:

- 优先选用接口,尽量少用抽象类
- 需要定义子类的行为,又要为子类提供共性功能时才选用抽象类

标记接口:

只有接口的名字, 但是没有任何方法, 是标记给JVM用

接口注意事项:

- 不要让default方法与Object类中任何一个方法相同

静态导包:

针对于静态方法, 导入成功可以直接使用静态方法而不需要通过类名.的形式访问

例如:

```
import static java.util.Arrays.sort;

class Test {
    public static void main(String[] args) {
        int[] arr = {1, 2, 34, 1, 2, 5, 76, 4};
        sort(arr);
    }
}
```

异常(类)(Exception):

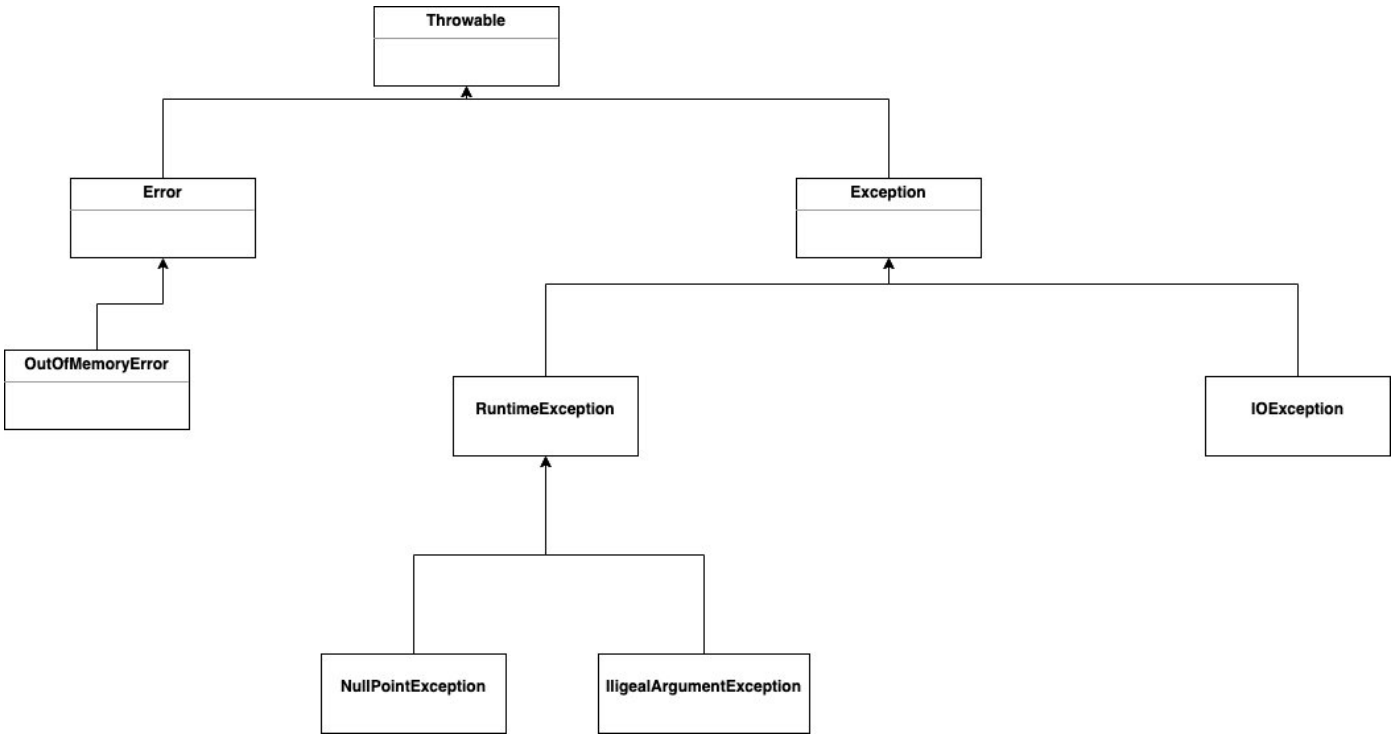
指程序运行过程中, 因为用户的误操作, 代码的bug等等一系列原因引起程序的崩溃

不是Error, 异常是可以挽救的错误, 而Error是不可挽救的.

异常分类:

- 编译型异常: 在源码编译阶段时抛出, 这种异常必须处理, 不处理就无法编译成功
- 运行时异常: 编译阶段看不出有错误, 在运行时有可能出现的异常

异常类继承树:



从继承关系可知: Throwable 是异常体系的根, 它继承自 Object.

Throwable 有两个体 系: Error 和, 表示严重的错误, 程序对此一般无能为力, 例如:

- OutOfMemoryError: 内存耗尽
- NoClassDefFoundError: 无法加载某个Class
- StackOverflowError: 栈溢出

而 Exception 则是运行时的错误, 它可以被捕获并处理.

某些异常是应用程序逻辑处理的一部分, 应该捕获并处理. 例如:

- NumberFormatException: 数值类型的格式错误
- FileNotFoundException: 未找到文件
- SocketException: 读取网络失败

还有一些异常是程序逻辑编写不对造成的, 应该修复程序本身. 例如:

- NullPointerException: 对某个 null 的对象调用方法或字段
- IndexOutOfBoundException: 数组索引越界

Exception 又分为两大类:

1. RuntimeException 以及它的子类;
2. 非RuntimeException (包括 IOException, ReflectiveOperationException 等等)

Java规定:

必须捕获的异常, 包括 Exception 及其子类, 但不包括 RuntimeException 及其子类, 这 种类型的异常为Checked Exception.

不需要捕获的异常, 包括 Error 及其子类, RuntimeException 及其子类.

异常处理:

解决掉异常的现象, 让程序继续运行

提高程序的容错能力, 也就提高了程序的稳定性

java有一个Exception的对象处理异常

尽量不要直接用Exception类来处理, 应该先用Exception的子类, 最后再写一个Exception

java进行异常处理, 有两种解决方案:

1. 抓捕异常(重点)
2. 抛出异常

throw和throws

throw 用在方法里, throws用在方法上(处理编译阶段的异常)

```
class Test {
    public static void test() {
        throw new MyException("异常");
    }

    public static void test2() throws Exception {

    }
}
```

抓捕异常:

针对于可能出现异常的代码, 进行抓捕

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

        } catch (Exception e) {

            // 如果出现异常, 代码会立刻进入catch中
            // 出现异常的语句后面的代码不会执行, 会立即进入catch
            // 在这里解决抓捕到的异常
            // 只有出现了异常catch中的代码才会执行
        } finally {
            // 必须执行的代码
        }
    }
}
```

如果使用抓捕异常, 通过这种处理, 程序即便是遇到了, 也不崩溃

```
class Test {
    public static void main(String[] args) {
        int result = 0;
        int num1;
        int num2;
        Scanner scan = new Scanner(System.in);
        num1 = scan.nextInt();
        num2 = scan.nextInt();
        try {
            System.out.println(1);
            result = num1 / num2;
            System.out.println(2);
        } catch (Exception e) {
            e.printStackTrace(); // 可以用此方法打印出异常
            System.out.println(3);
        }
```

```
        System.out.println(4);
        System.out.println(result);
    }
}
```

捕获异常使用try...catch语句, 把可能发生异常的代码放到try {...}中, 然后使用 catch 捕获对应的Exception及其子类:

```
public class Main {
    public static void main(String[] args) {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) {
        try {
            // 用指定编码转换String为byte[]:
            return s.getBytes("GBK");
        } catch (UnsupportedEncodingException e) {
            // 如果系统不支持GBK编码, 会捕获到UnsupportedEncodingException:
            System.out.println(e); // 打印异常信息
            return s.getBytes(); // 尝试使用用默认编码
        }
    }
}
```

如果我们不捕获 UnsupportedEncodingException , 会出现编译失败的问题.

编译器会报错, 错误信息类似: unreported exception UnsupportedEncodingException; must be caught or declared to be thrown,
并且准确地指出需要捕获的语句是 return s.getBytes("GBK"); .

意思是说, 像 UnsupportedEncodingException 这样的Checked Exception, 必须被捕获.

这是因为 String.getBytes(String) 方法定义是:

```
public final class String implements java.io.Serializable, Comparable<String>,
CharSequence, Constable, ConstantDesc {
    public byte[] getBytes(String charsetName) throws UnsupportedEncodingException {
        //...
    }
}
```

在方法定义的时候, 使用 throws Xxx 表示该方法可能抛出的异常类型. 调用方在调用的时候, 必须强制捕获这些异常, 否则编译器会报错.

在toGBK()方法中, 因为调用了 String.getBytes(String) 方法, 就必须捕获 .

我们也可以不捕获它, 而是在方法定义处用throws表示toGBK()方法可能会抛出UnsupportedEncodingException, 就可以让 toGBK()
方法通过编译器检查:

```
public class Main {
    public static void main(String[] args) {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) throws UnsupportedEncodingException {
        return s.getBytes("GBK");
    }
}
```

上述代码仍然会得到编译错误, 但这一次, 编译器提示的不是调用return s.getBytes("GBK") 的问题, 而是byte[] bs = toGBK("
中文").

因为在main()方法中调用toGBK() , 没有捕获它声明的可能抛出的 UnsupportedEncodingException.

修饰方法是在main()方法中捕获异常并处理:

```
public class Main {
    public static void main(String[] args) {
        try {
            byte[] bs = toGBK("中文");
            System.out.println(Arrays.toString(bs));
        } catch (UnsupportedEncodingException e) {
            System.out.println(e);
        }
    }

    static byte[] toGBK(String s) throws UnsupportedEncodingException {
        // 用指定编码转换String为byte[]:
        return s.getBytes("GBK");
    }
}
```

可见, 只要是方法声明的Checked Exception, 不在调用层捕获, 也必须在更高的调用层捕获. 所有 未捕获的异常, 最终也必须在main()
方法中捕获, 不会出现漏写 try 的情况. 这是由编译器保证 的. main() 方法也是最后捕获Exception的机会.

```
public class Main {
    public static void main(String[] args) throws Exception {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) throws UnsupportedEncodingException {
        // 用指定编码转换String为byte[]:
        return s.getBytes("GBK");
    }
}
```

因为 main() 方法声明了可能抛出 Exception , 也就声明了可能抛出所有的 Exception , 因此 在内部就无需捕获了.
代价就是一旦发生异常,
程序会立刻退出.

也可在内部消化

```
class Test {
    static byte[] toGBK(String s) {
        try {
            return s.getBytes("GBK");
        } catch (UnsupportedEncodingException e) {
            // 什么也不干
        }
        return null;
    }
}
```

这种捕获后不处理的方式是非常不好的, 即使真的什么也做不了, 也要先把异常记录下来:

```
class Test {
    static byte[] toGBK(String s) {
        try {
            return s.getBytes("GBK");
        } catch (UnsupportedEncodingException e) {
            // 先记下来再说:
            e.printStackTrace();
        }
        return null;
    }
}
```

所有异常都可以调用 printStackTrace() 方法打印异常栈, 这是一个简单有用的快速打印异常的方法.

抛出异常:

一种消极处理

方法后面用throws关键字加上异常类, 把异常抛给调用者, 让调用者处理, 如果抛到main函数不解决再继续抛, 最后会抛给JVM处理

子类不能抛出比父类更多的异常

```
class Test {
    public static void main() {
        try {
            createFile("Users/usus/a.txt");
        } catch (IoException e) {

            throw new IoExcption();
        } catch (Exception e) {

            throw new Exception();
        }
    }

    public static void createFile(String path) throws IoException, Exception {
        File file = new file(path);
        file.createNewFile();
    }
}
```

finally关键字:

finally 中的代码一定会执行, 即使finally之前有return也会执行finally中的代码, 因为finally是jvm级别而不是源码级别

```
class Test {
    public static void main(String[] args) {
        try {
            // 可能存在异常的代码
        } catch (Exception e) {
            // 处理的方式
        } finally {
            // 这里的代码可以不用写
            // 一旦这里写了代码, 不管try中有没有异常这里必须执行
        }
    }
}
```

finally中写什么:

- 回收垃圾
- 关闭IO流
- 关闭数据库链接
- .....
- 类似的核心代码

finally特殊用法:

在开发中有一些代码必须执行但不知道会发生什么异常, 为了保证执行, 可以这么写:

```
class Test {
    void test() {
        try {
            // 这里不写
        } catch (Exception e) {
            // 这里不写
        } finally {
            // 这里写重要代码
        }
    }
}
```

以下返回值是多少?

```
class Test {
    public static int test() {
        int a = 10;
        int b = 20;
        try {
            b += a++;
            return b;
        } catch (Exception e) {

        } finally {
            b += 10;
            a += 10;
        }
        return a;
    }
}
// 这个函数的返回值是30
```

自定义异常类:

继承异常类, 再自定义

自定义异常可以向调用者传递信息

```
class MyException extends RuntimeException {
    MyException(String s) {
        super(s);
    }

    MyException() {
        super();
    }
}
```

包装(封装)类:

扩展基本数据类型的功能

包装类是对象, 基本数据类型是常量

基本数据类型	包装类
byte	Byte
short	Short
int	Integer
long	Long
boolean	Boolean
float	Float
double	Double
char	Character

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

        Integer a = 1000;
        Integer b = Integer.valueOf(1000);
        Integer c = Integer.valueOf(1000).intValue();
        Integer d = Integer.parseInt("1000");
    }
}

```

```
/*
对于以上 a != b != c != d
如果值换成小于127的数就是==, 因为有小整形常量池(-128--127)
*/
```

拆箱和装箱功能:

拆箱:

将包装类的对象转换为基本数据类型

Integer a = 100; --> int b = a;

装箱:

将基本数据类型包装为包装类

int a = 100; --> Integer b = a;

int a = 10;

Integer b = integer.valueOf(a) // valueOf参数包装成Integer对象返回

自动拆装箱:

目前Java中提供了自动拆装箱功能

如上所示的将 int 附给 Integer 就是自动装箱, 将Integer附给int就是自动拆箱

```
class Test {
    public static void main(String[] args) {
        int a = 100;
        Integer b = a; // 自动装箱
        Integer c = 100; // 自动装箱
        int d = c; // 自动拆箱
    }
}
```

Void包装类:

Void是对void的引用, 不能被实例化. 如果方法返回值是Void, 那么该方法就只能返回null

用途:

- 让Callable的call方法只支持抛异常

```
public class Test {
    public static void main(String[] args) {
        ExecutorService pool = Executors.newSingleThreadExecutor();
        try {
            // 在Callable接口的call方法中必须有一个返回值, 但我们想让线程只支持抛出异常
```

```
            Future<Void> future = pool.submit(() -> {
                Thread.sleep(3000);
                return null;
            });
        } catch (Exception ignored) {
        } finally {
            pool.shutdown();
            pool.close();
        }
    }
}
```

- 使用泛型函数不需要返回值时

Object类:

Object类是java中所有类的父类

java.lang.Object

native关键字:

被它修饰的叫本地方法, 没有实现体, JVM调用底层的C/C++代码实现

修饰符和返回值	方法	描述
public final native Class<?>	getClass()	获取字节码文件
public native int	hashCode()	获取对象的hashcode
public boolean	equals(Object obj)	比较对象地址
protected native Object	Object clone() throws CloneNotSupportedException	克隆对象, 对象拷贝, 必须实现一个Cloneable接口和处理异常, 同时要重写
public String	toString()	打印
public final native void	notify()	用于唤醒随机一个在此对象监视器上等待的线程
public final native void	notifyAll()	唤醒在该对象上等待的所有线程
public final void	wait()	让当前线程进入等待状态. 直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法
protected void	finalize()	在gc触发前会自动触发, 用于做一些gc触发前的各种工作

对象拷贝:

引用拷贝只拷贝栈引用地址, 而对象拷贝则是在堆中拷贝出一个新对象

浅拷贝:

> 将对象的第一层完成拷贝, 对两个对象完成了基本的分离, 有可能还存在着藕断丝连

> 例如对象User中有一个Cat对象, 这时浅拷贝只会拷贝出一个新的User对象, 而两个User对象中的Cat对象还是指向堆中同一块内存

优点: 内存占用较少

缺点: 如果存在子对象, 则子对象没有拷贝, 还是指向同一个

```
class User implements Cloneable {

    public static void main(String[] args) {
        User user1 = new User();
        User user2 = null;
        user2 = user1;
        System.out.println(user2 == user1);
        user2 = user1.clone();
        System.out.println(user1 == user2);
    }

    @Override
    public User clone() {
        try {
            return (User) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

深拷贝:

也叫递归拷贝

> 将两个对象完全分离, 彼此之间无任何联系

java实现深拷贝有两种方法:

1. 所有相关对象都实现浅拷贝
2. 通过序列化对象实现深拷贝

所有相关对象都实现浅拷贝:

```
class User implements Cloneable {
    Cat cat = new Cat();
}

class Cat implements Cloneable {
    String name = "122";
}
```

通过序列化对象实现深拷贝:

对象 <-----> 字节数据

对象 <-----> 字符串数据 (如JSON)

再将字节数据转换成对象

String类:

String类是Java的底层实现的类, 为了安全用final修饰

底层是字节数组

字符串本质是一个常量

字符串缓冲池: 字符串常量池

面试题: 字符串存储在哪里?

> 在JDK7之前, 字符串常量池在方法区

> 在JDK7之后, 字符串常量池被设计到堆中

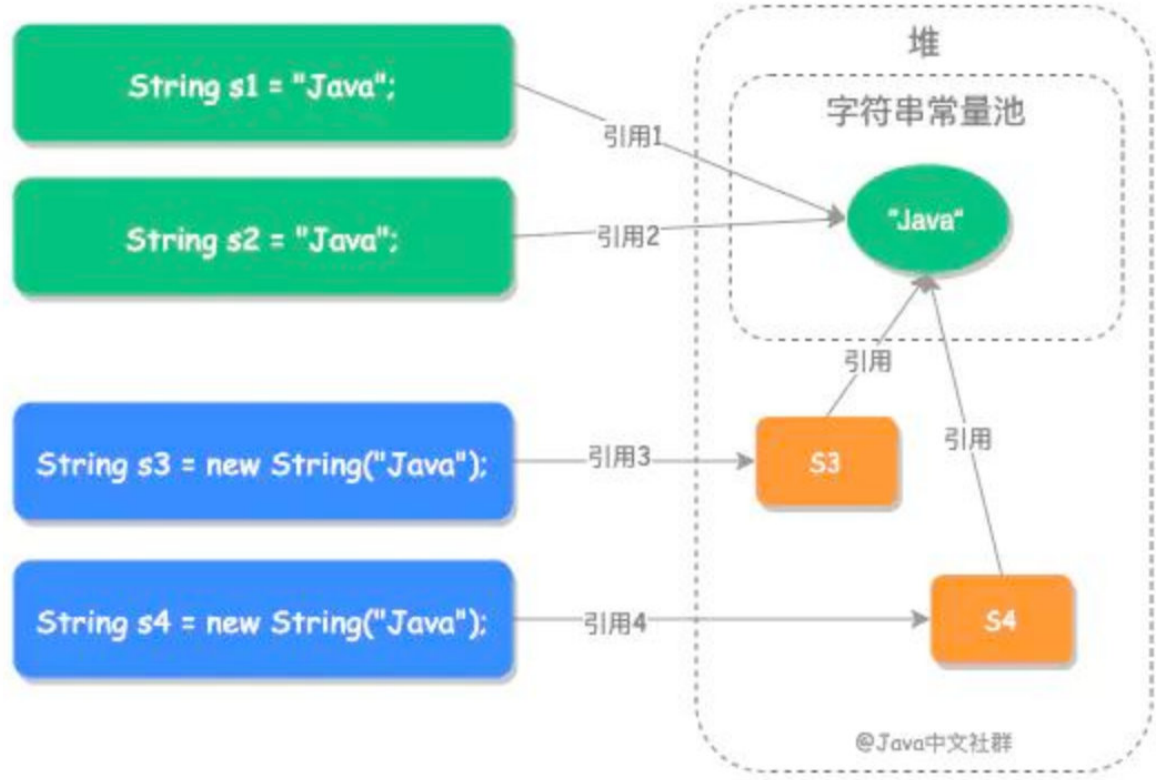
String s = "123";

> 这样创建一个字符串时会首先去字符串常量池找, 如果找到就引用, 找不到再创建字符串对象, s是常量池的引用
>
> 这样创建会创建0个或者1个

String s2 = new String("123");

> 这样如果字符串常量池中有这个字符串就先引用然后拷贝到堆中, 没有就在字符串常量池新建再拷贝到堆中, s2是堆中字符串的引用而不是常量池的引用,
> 在堆中

> 这样创建会创建1个或者2个



字符串拼接问题:

字符串拼接时, 如何判断拼接后结果是否相等(==):

- 常量+常量 = 常量
- 编译时结果确定则会相等, 得不到确定就会是false

```
class Test {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "he" + "llo";
        s1 == s2        // 为true, s2的字符串拼接发生在编译阶段, 编译器会将其优化成一个字符串
    }
}
```

```
class Test {
    public static void main(String[] args) {
        String s1 = "hello1";
        String s2 = "hello";
        final String s2 = "hello";
    }
}
```

字符串拼接时, 因为是常量不会回收, 所以会产生很多垃圾

StringBuilder和StringBuffer:

这两个对象都是用来拼接字符串的对象

底层是字节数组

不会产生垃圾, 不是String对象, 而是独立的两个对象

StringBuilder:

```
class Test {
    @Test
    void testStringBuilder() {
        StringBuilder s = new StringBuilder("123");
        s.append("4");
        s.insert("0");
    }
}
```

StringBuffer:

用法和StringBuilder一样

两者区别:

StringBuffer在所有操作方法上用synchronized修饰(同步锁), 所以是线程安全的

StringBuilder没有同步锁, 因此是非线程安全的, 但是效率高一点

时间对象:

日期和时间的格式化编码

时间模式字符串用来指定时间格式。在此模式中，所有的 ASCII 字母被保留为模式字母，定义如下：

字母	描述	示例
G	纪元标记	AD
y	四位年份	2001
M	月份	July or 07
d	一个月的日期	10
h	A.M./P.M. (1~12)格式小时	12
H	一天中的小时 (0~23)	22
m	分钟数	30
s	秒数	55
S	毫秒数	234
E	星期几	Tuesday
D	一年中的日子	360
F	一个月中第几周的周几	2 (second Wed. in July)
w	一年中第几周	40
W	一个月中第几周	1
a	A.M./P.M. 标记	PM
k	一天中的小时(1~24)	24
K	A.M./P.M. (0~11)格式小时	10
z	时区	Eastern Standard Time
'	文字定界符	Delimiter
"	单引号	`

Date:

多数方法已经过时, 不建议使用

注意: 获取的时间是当前系统时区的时间

构造函数:

- Date()　获取当前时间
- Date(long date)　　这个参数是时间戳

Date date = new Date(0L)　// 格林威治时间: 1970.1.1 00:00:00

时间戳就是从格林威治时间到现在的毫秒数

已过时的构造函数:

- Date(int year, int month, int date)　Date(100, 02, 03)　2000年的2月3日
- Date(int year, int month, int date, int hrs, int min)
- Date(int year, int month, int date, int hrs, int min, int sec)
- Date(String s)　Date date = new Date("1999/02/21")

这个类中重要方法:

返回值类型	方法	描述
long	getTime()	获取当前时间的时间戳
void	setTime(long time)	设置时间戳

DateFormat:

格式化时间对象的类

```
class Test {
    void Test01() {
        DateFormat dateFormat = new SimpleDateFormat("Gyyyy年MM月dd日 HH:mm:ss"); //小写h为十二小时制
        Date date = new Date();
        String strDate = dateFormat.format(date);
    }

    void test02() {
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date date = new Date();
        String strDate = dateFormat.format(date);
    }

    void test03() {
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String strDate = "2020-03-08 18:59:22";

        // 将特定时间字符串转换为时间对象
        Date date = null;
        try {
            date = dateFormat.parse(strDate); // 这里必须做异常处理, parse可能会出现不可转换的错误
        } catch (ParseException e) {
            e.printStackTrace();
        }
        System.out.println(date);
    }
}
```

LocalDate:

```
class Test {
    void test() {
        LocalDate now = LocalDate.now();
        LocalDate date = LocalDate.parse("2021-8-18");
    }
}
```

LocalTime:

```
class Test {
    void test() {
        LocalTime now = LocalTime.now();
    }
}
```

LocalDateTime:

```
class Test {
    void test() {
        LocalDateTime now = LocalDateTime.now();
    }
}
```

LocalTimeFormatter:

```
class Test {
    void test() {
        LocalDateTime now = LocalDateTime.now();
        LocalTimeFormatter pattern = LocalTimeFormatter.ofpattern("yyyy-MM-dd HH:mm:ss.SSS");
        String strDate = now.format(pattern);

        LocalDateTime
    }
}
```

Instant:

```
class Test {
    public static void main(String[] args) {
        Instant now = Instant.now(); // 格林威治时间
    }
}
```

数字格式转换:

float和double在运算过程中都是需要转换为整数的二进制数据

在数据较大时不建议用浮点型, 而是使用更加精确的decimal类型

NumberFormat:

```
class Test {
    void test() {
        System.out.println(0.1 + 0.2 == 0.3); // false
        BigDecimal num_1 = new BigDecimal(0.1);
        BigDecimal num_2 = new BigDecimal(0.2);

        // 如果不要以整数的形式操作, 可以使用字符串
        BigDecimal num_3 = new BigDecimal("0.1");
        BigDecimal num_4 = new BigDecimal("0.2");
        System.out.println(num3.add(num4) == 0.3); // true
    }
}
```

transient关键字:

如果一个类没有继承Serializable接口,那么它就不能被序列化,写入文件的时候会报异常.如果一个类继承了Serializable接口,那么这个类的所有属性和方法都可以自动被序列化,而现实中我们又希望继承Serializable的这个类的某些属性不被序列化保存该需要怎么做呢? 这时就需要Transient修饰符上场了.总之,java的transient关键字为我们提供了便利,你只需要实现Serializable接口,将不需要序列化的属性前添加关键字transient,序列化对象的时候,这个属性就不会序列化到指定的目的地中.

枚举:

使用enum关键字定义

```
class Test {
    enum Day {
        SATURDAY, SUNDAY
    }

    public static void main(String[] args) {
        Day day = Day.SATURDAY;
        System.out.println(day);
    }
}
```

泛型:

参考:

https://www.cnblogs.com/wuqinglong/p/9456193.html#1java%E6%B3%9B%E5%9E%8B%E7%9A%84%E5%AE%9E%E7%8E%B0%E6%96%B9%E6%B3%95%EF%BC%9A%E7%B1%BB%E5%9E%8B%E6%93%A6%E9%99%A4

https://www.runoob.com/java/java-generics.html

https://www.cnblogs.com/hoojjack/p/6817547.html

泛型, 即"参数化类型". 一提到参数, 最熟悉的就是定义方法时有形参, 然后调用此方法时 传递实参. 那么参数化类型怎么理解呢?顾名思义,
就是将类型由原来的具体的类型参数化, 类似于方法中的变量参数, 此时类型也定义成参数形式(可以称之为类型形参), 然后在使用/调用
时传入具体的类型(类型实参).

泛型的本质是为了参数化类型(在不创建新的类型的情况下, 通过泛型指定的不同类型来控
制形参具体限制的类型). 也就是说在泛型使用过程中, 操作的数据类型被指定为一个参数, 这
种参数类型可以用在类、接口和方法中, 分别被称为泛型类、泛型接口、泛型方法.

泛型保证了类型的统一

泛型只支持对象不支持基本数据类型

泛型标识符:

- **E** - Element (在集合中使用,因为集合中存放的是元素)
- **T** - Type(Java 类)
- **K** - Key(键)
- **V** - Value(值)
- **N** - Number(数值类型)
- **?** - 表示不确定的 java 类型

泛型方法:

```
class Test {
    // 第一个<T>为指定泛型, 第二个T为返回值类型
    public static <T> T getMiddle(T... a) {
        return a[a.length / 2];
    }

    // 调用方法时可以把具体类型卸载尖括号中
    public static void main(String[] args) {
        String middle = Test.<String>getMiddle("John", "Q", "Public"); // 必须用类名.的形式
        System.out.println(middle);
    }
}
```

泛型类:

```
class 类名称<泛型标识符> // (可以写任意标识号, 标识指定的泛型的类型)
{
    private 泛型标识符 /*(成员变量类型)*/ var;
}
```

```
// 在实例化泛型类时, 必须指定T的具体类型
public class Generic<T> {
    // key这个成员变量的类型为T, T由外部指定
    private T key;

    public Generic(T key) {
        this.key = key;
    }

}
```

泛型继承类和接口:

泛型后可以用extends关键字继承类, 且可以继承多个, 多个类之间用&链接, 类在前接口在后

```
class Test<T extends Comparable & Serializable> implements Serializable {

}
```

泛型的通配符:

?通配符:

代表未知类型

<? extends T>

上边界限定通配符 extends:

不能往里存，只能往外取

> 表示类型的上界，表示参数化类型的可能是T 或是 T的子类. `?` 这个对象包含的元素类型是 `T` 的子类型 (包含 `T` 本身) 的一种

`extends` 可用于返回类型限定，不能用于参数类型限定 (换句话说: `? extends T`
只能用于方法返回类型限定，jdk能够确定此类的最小继承边界为`T`, 只要是这个类的父类都能接收,
但是传入参数无法确定具体类型，只能接受null的传入).

```
class Fruit {

```

```
}

class Apple extends Fruit {

}

class Peach extends Fruit {

}

public class Test {
    public static void main(String[] args) {
        List<? extends Fruit> list = new ArrayList<>();
        list.add(new Apple()); // 报错，并不能添加
        Apple apple1 = list.get(0); // 可以取出
    }
}
```

下边界限定通配符 super:

<? super T>

不影响往里存，但往外取只能放在Object对象里

> 类型的下界, 我们并不知道`?`的类型, 但是可以知道? 必定是`T`的子类或者`T`本身是一个基类, 这是可以传入`T`的任何子类, `?`
> 类型也可以向上转型为`T`, 这个对象包含的元素就类型是T的超类型 (包含T本身) 的一种。

`super`可用于参数类型限定, 不能用于返回类型限定 (`? supper T` 只能用于方法传参, 因为jdk能够确定传入为`T`的子类,
返回只能用`Object`类接收).

```
class Fruit {

}

class Apple extends Fruit {

}

class Peach extends Fruit {

}

public class Test {
    public static void main(String[] args) {
        List<? super Fruit> list = new ArrayList<>();
        list.add(new Apple()); // 可以添加
        Apple apple1 = list.get(0); // 报错，此时只能返回Object
        Apple apple = (Apple) list.get(0); // 正确
```

```
    }
}
```

通配符规则:

满足PECS(“Producer Extends, Consumer Super”)---如果参数化类型表示一个生产者, 就使用`<? extends T>`; 如果它表示一个消费者,
就使用`<? super T>`

类型擦除:

Java的泛型基本上都是在编译器这个层次上实现的，在生成的字节码中是不包含泛型中的类型信息的，使用泛型的时候加上类型参数，在编译器编译的时候会去掉，这个过程成为类型擦除。

> 如在代码中定义`List<Object>`和`List<String>`等类型, 在编译后都会变成`List` , JVM看到的只是List, 而由泛型附加的类型信息对JVM是看不到的.
> Java编译器会在编译时尽可能的发现可能出错的地方, 但是仍然无法在运行时刻出现的类型转换异常的情况, 类型擦除也是Java的泛型与C++模板机制实现方式之间的重要区别.

例1.原始类型相等

```
public class Test {

    public static void main(String[] args) {

        ArrayList<String> list1 = new ArrayList<String>();
        list1.add("abc");

        ArrayList<Integer> list2 = new ArrayList<Integer>();
        list2.add(123);

        System.out.println(list1.getClass() == list2.getClass()); // true
    }

}
```

在这个例子中，我们定义了两个`ArrayList`数组，不过一个是`ArrayList<String>`
泛型类型的，只能存储字符串；一个是`ArrayList<Integer>`泛型类型的，只能存储整数，最后，我们通过`list1`对象和`list2`
对象的`getClass()`方法获取他们的类的信息，最后发现结果为`true`。说明泛型类型`String`和`Integer`都被擦除掉了，只剩下原始类型。

例2.通过反射添加其它类型元素

```
public class Test {

```

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

        ArrayList<Integer> list = new ArrayList<Integer>();

        list.add(1);  //这样调用 add 方法只能存储整形，因为泛型类型的实例为 Integer

        list.getClass().getMethod("add", Object.class).invoke(list, "asd");

        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
    }

}
```

在程序中定义了一个`ArrayList`泛型类型实例化为`Integer`对象，如果直接调用`add()`方法，那么只能存储整数数据，不过当我们利用反射调用`add()`方法的时候，却可以存储字符串，这说明了`Integer`泛型实例在编译之后被擦除掉了，只保留了原始类型。

类型擦除后保留的原始类型

在上面，两次提到了原始类型，什么是原始类型?

原始类型 就是擦除去了泛型信息，最后在字节码中的类型变量的真正类型，无论何时定义一个泛型，相应的原始类型都会被自动提供，类型变量擦除，并使用其限定类型（无限定的变量用Object）替换。

例3.原始类型`Object`

```
class Pair<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

`Pair`的原始类型为:

```
class Pair {
    private Object value;

    public Object getValue() {
        return value;
    }

    public void setValue(Object value) {
        this.value = value;
    }
}
```

因为在`Pair<T>`中，T 是一个无限定的类型变量，所以用`Object`
替换，其结果就是一个普通的类，如同泛型加入Java语言之前的已经实现的样子。在程序中可以包含不同类型的`Pair`，如`Pair<String>`
或`Pair<Integer>`，但是擦除类型后他们的就成为原始的`Pair`类型了，原始类型都是`Object`。

从上面的例2中，我们也可以明白`ArrayList<Integer>`被擦除类型后，原始类型也变为`Object`，所以通过反射我们就可以存储字符串了。

如果类型变量有限定，那么原始类型就用第一个边界的类型变量类替换。

比如: Pair这样声明的话

```
public class Pair<T extends Comparable> {}
```

那么原始类型就是`Comparable`。

要区分原始类型和泛型变量的类型。

在调用泛型方法时，可以指定泛型，也可以不指定泛型。

- 在不指定泛型的情况下，泛型变量的类型为该方法中的几种类型的同一父类的最小级，直到`Object`
- 在指定泛型的情况下，该方法的几种类型必须是该泛型的实例的类型或者其子类

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

        /**不指定泛型的时候*/
        int i = Test.add(1, 2); //这两个参数都是Integer，所以T为Integer类型
        Number f = Test.add(1, 1.2); //这两个参数，一个是Integer，一个是Float，所以取同一父类的最小级，为Number
        Object o = Test.add(1, "asd"); //这两个参数，一个是Integer，一个是String，所以取同一父类的最小级，为Object

        /**指定泛型的时候*/
        int a = Test.<Integer>add(1, 2); //指定了Integer，所以只能为Integer类型或者其子类
        int b = Test.<Integer>add(1, 2.2); //编译错误，指定了Integer，不能为Float
```

```
        Number c = Test.<Number>add(1, 2.2); //指定为Number，所以可以为Integer和Float

    }

    //这是一个简单的泛型方法
    public static <T> T add(T x, T y) {
        return y;
    }
}
```

其实在泛型类中，不指定泛型的时候，也差不多，只不过这个时候的泛型为`Object`，就比如`ArrayList`中，如果不指定泛型，那么这个`ArrayList`可以存储任意的对象。

例4.Object泛型

```
class Test {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        list.add(1);
        list.add("121");
        list.add(new Date());
    }
}
```

类型擦除引起的问题及解决方法

因为种种原因，Java不能实现真正的泛型，只能使用类型擦除来实现伪泛型，这样虽然不会有类型膨胀问题，但是也引起来许多新问题，所以，SUN对这些问题做出了种种限制，避免我们发生各种错误。

先检查，再编译以及编译的对象和引用传递问题

Q: 既然说类型变量会在编译的时候擦除掉，那为什么我们往`ArrayList`创建的对象中添加整数会报错呢？不是说泛型变量`String`
会在编译的时候变为`Object`类型吗？为什么不能存别的类型呢？既然类型擦除了，如何保证我们只能使用泛型变量限定的类型呢？

A: Java编译器是通过先检查代码中泛型的类型，然后在进行类型擦除，再进行编译。

例如：

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

        ArrayList<String> list = new ArrayList<String>();
        list.add("123");
        list.add(123);//编译错误
    }
}
```

在上面的程序中，使用`add`
方法添加一个整型，在IDE中，直接会报错，说明这就是在编译之前的检查，因为如果是在编译之后检查，类型擦除后，原始类型为`Object`
，是应该允许任意引用类型添加的。可实际上却不是这样的，这恰恰说明了关于泛型变量的使用，是会在编译之前检查的。

那么，这个类型检查是针对谁的呢？我们先看看参数化类型和原始类型的兼容。

以`ArrayList`举例子，以前的写法:

`ArrayList list=new ArrayList();`

现在的写法:

`ArrayList<String> list=new ArrayList<String>();`

如果是与以前的代码兼容，各种引用传值之间，必然会出现如下的情况：

`ArrayList<String> list1=new ArrayList();` 第一种 情况

`ArrayList list2=new ArrayList<String>();` 第二种 情况

这样是没有错误的，不过会有个编译时警告。

不过在第一种情况，可以实现与完全使用泛型参数一样的效果，第二种则没有效果。

因为类型检查就是编译时完成的，`new ArrayList()`
只是在内存中开辟了一个存储空间，可以存储任何类型对象，而真正设计类型检查的是它的引用，因为我们是使用它引用`list1`
来调用它的方法，比如说调用`add`方法，所以`list1`引用能完成泛型类型的检查。而引用`list2`没有使用泛型，所以不行。

举例子：

```
public class Test {

    public static void main(String[] args) {

        ArrayList<String> list1 = new ArrayList();
        list1.add("1"); //编译通过
        list1.add(1); //编译错误
        String str1 = list1.get(0); //返回类型就是String

        ArrayList list2 = new ArrayList<String>();
        list2.add("1"); //编译通过
        list2.add(1); //编译通过
        Object object = list2.get(0); //返回类型就是Object

        new ArrayList<String>().add("11"); //编译通过
        new ArrayList<String>().add(22); //编译错误

        String str2 = new ArrayList<String>().get(0); //返回类型就是String
    }
```

```
20
21 }
```

通过上面的例子，我们可以明白，类型检查就是针对引用的，谁是一个引用，用这个引用调用泛型方法，就会对这个引用调用的方法进行类型检测，而无关它真正引用的对象。

泛型中参数话类型为什么不考虑继承关系?

在Java中，像下面形式的引用传递是不允许的:

`ArrayList<String> list1=new ArrayList<Object>();` //编译错误

`ArrayList<Object> list2=new ArrayList<String>();` //编译错误

我们先看第一种情况，将第一种情况拓展成下面的形式：

```
1 class Test {
2     public static void main(String[] args) {
3         ArrayList<Object> list1 = new ArrayList<Object>();
4         list1.add(new Object());
5         list1.add(new Object());
6         ArrayList<String> list2 = list1; //编译错误
7     }
8 }
```

实际上，在第4行代码的时候，就会有编译错误。那么，我们先假设它编译没错。那么当我们使用`list2`引用用`get()`
方法取值的时候，返回的都是`String`
类型的对象（上面提到了，类型检测是根据引用来决定的），可是它里面实际上已经被我们存放了`Object`
类型的对象，这样就会有`ClassCastException`了。所以为了避免这种极易出现的错误，Java不允许进行这样的引用传递。（这也是泛型出现的原因，就是为了解决类型转换的问题，我们不能违背它的初衷）。

再看第二种情况，将第二种情况拓展成下面的形式：

```
1 ArrayList<String> list1 = new ArrayList<String>();
2 list1.add(new String());
3 list1.add(new String());
4
5 ArrayList<Object> list2 = list1; //编译错误
```

没错，这样的情况比第一种情况好的多，最起码，在我们用`list2`取值的时候不会出现`ClassCastException`，因为是从`String`
转换为`Object`
。可是，这样做有什么意义呢，泛型出现的原因，就是为了解决类型转换的问题。我们使用了泛型，到头来，还是要自己强转，违背了泛型设计的初衷。所以java不允许这么干。再说，你如果又用`list2`
往里面`add()`新的对象，那么到时候取得时候，我怎么知道我取出来的到底是`String`类型的，还是`Object`类型的呢?

所以，要格外注意，泛型中的引用传递的问题。

动类型转换

因为类型擦除的问题，所以所有的泛型类型变量最后都会被替换为原始类型。

既然都被替换为原始类型，那么为什么我们在获取的时候，不需要进行强制类型转换呢?

看下`ArrayList.get()`方法:

```
public class ArrayList {
    public E get(int index) {

        RangeCheck(index);

        return (E) elementData[index];

    }
}
```

可以看到，在`return`之前，会根据泛型变量进行强转。假设泛型类型变量为`Date`，虽然泛型信息会被擦除掉，但是会将`(E) elementData[index]`，编译为`(Date)elementData[index]`。所以我们不用自己进行强转。当存取一个泛型域时也会自动插入强制类型转换。假设`Pair`类的`value`域是`public`的，那么表达式:

`Date date=pair.value;`

也会自动地在结果字节码中插入强制类型转换。

类型擦除与多态的冲突和解决方法

现在有这样一个泛型类:

```
class Pair<T> {

    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

然后我们想要一个子类继承它。

```
class DateInter extends Pair<Date> {

    @Override
    public void setValue(Date value) {
        super.setValue(value);
    }

    @Override
    public Date getValue() {
        return super.getValue();
    }
}
```

在这个子类中，我们设定父类的泛型类型为`Pair<Date>`，在子类中，我们覆盖了父类的两个方法，我们的原意是这样的：将父类的泛型类型限定为`Date`，那么父类里面的两个方法的参数都为`Date`类型。

```
class Pair {
    public Date getValue() {
        return value;
    }

    public void setValue(Date value) {
        this.value = value;
    }
}
```

所以，我们在子类中重写这两个方法一点问题也没有，实际上，从他们的`@Override`标签中也可以看到，一点问题也没有，实际上是这样的吗?

分析：实际上，类型擦除后，父类的的泛型类型全部变为了原始类型`Object`，所以父类编译之后会变成下面的样子：

```
class Pair {
    private Object value;

    public Object getValue() {
        return value;
    }

    public void setValue(Object value) {
        this.value = value;
    }
}
```

再看子类的两个重写的方法的类型：

```
class DateInter extends Pair<Date> {
    @Override
    public void setValue(Date value) {
        super.setValue(value);
    }

    @Override
    public Date getValue() {
        return super.getValue();
    }
}
```

先来分析`setValue`方法，父类的类型是`Object`，而子类的类型是`Date`，参数类型不一样，这如果实在普通的继承关系中，根本就不会是重写，而是重载。

我们在一个main方法测试一下：

```
class Test {
    public static void main(String[] args) throws ClassNotFoundException {
        DateInter dateInter = new DateInter();
        dateInter.setValue(new Date());
        dateInter.setValue(new Object()); //编译错误
    }
}
```

如果是重载，那么子类中两个`setValue`方法，一个是参数`Object`类型，一个是`Date`类型，可是我们发现，根本就没有这样的一个子类继承自父类的Object类型参数的方法。所以说，却是是重写了，而不是重载了。

为什么会这样呢?

原因是这样的，我们传入父类的泛型类型是`Date, Pair<Date>`，我们的本意是将泛型类变为如下：

```
class Pair {
    private Date value;

    public Date getValue() {
        return value;
    }

    public void setValue(Date value) {
        this.value = value;
    }
}
```

然后再子类中重写参数类型为Date的那两个方法，实现继承中的多态。

可是由于种种原因，虚拟机并不能将泛型类型变为`Date`，只能将类型擦除掉，变为原始类型`Object`。这样，我们的本意是进行重写，实现多态。可是类型擦除后，只能变为了重载。这样，类型擦除就和多态有了冲突。JVM知道你的本意吗？知道！！！可是它能直接实现吗，不能！！！如果真的不能的话，那我们怎么去重写我们想要的`Date`
类型参数的方法啊。

于是JVM采用了一个特殊的方法，来完成这项功能，那就是桥方法。

首先，我们用`javap -c className`的方式反编译下`DateInter`子类的字节码，结果如下：

```
class com.tao.test.DateInter extends com.tao.test.Pair<java.util.Date>{
        com.tao.test.DateInter();
        Code:
        0:aload_0
        1:invokespecial #8                  // Method com/tao/test/Pair."<init>":
()V
        4:return

public void setValue(java.util.Date);  //我们重写的setValue方法
        Code:
        0:aload_0
        1:aload_1
        2:invokespecial #16                 // Method com/tao/test/Pair.setValue:
(Ljava/lang/Object;)V
        5:return

public java.util.Date getValue();    //我们重写的getValue方法
        Code:
        0:aload_0
        1:invokespecial #23                 // Method com/tao/test/Pair.getValue:
()Ljava/lang/Object;
        4:checkcast     #26                 // class java/util/Date
        7:areturn

public java.lang.Object getValue();     //编译时由编译器生成的桥方法
        Code:
        0:aload_0
        1:invokevirtual #28                 // Method getValue:()Ljava/util/Date 去
调用我们重写的getValue方法;
        4:areturn

public void setValue(java.lang.Object);   //编译时由编译器生成的桥方法
        Code:
        0:aload_0
        1:aload_1
        2:checkcast     #26                 // class java/util/Date
        5:invokevirtual #30                 // Method setValue:(Ljava/util/Date; 去
调用我们重写的setValue方法)V
        8:return
```

```
35          }
```

从编译的结果来看，我们本意重写`setValue`和`getValue`
方法的子类，竟然有4个方法，其实不用惊奇，最后的两个方法，就是编译器自己生成的桥方法。可以看到桥方法的参数类型都是`Object`
，也就是说，子类中真正覆盖父类两个方法的就是这两个我们看不到的桥方法。而打在我们自己定义的`setvalue`和`getValue`
方法上面的`@Override`只不过是假象。而桥方法的内部实现，就只是去调用我们自己重写的那两个方法。

所以，虚拟机巧妙的使用了桥方法，来解决了类型擦除和多态的冲突。

不过，要提到一点，这里面的`setValue`和`getValue`这两个桥方法的意义又有不同。

`setValue`方法是为了解决类型擦除与多态之间的冲突。

而`getValue`却有普遍的意义，怎么说呢，如果这是一个普通的继承关系：

那么父类的`setValue`方法如下：

```
public Object getValue(){
return super.getValue();
}
```

而子类重写的方法是：

```
public Date getValue(){
return super.getValue();
}
```

其实这在普通的类继承中也是普遍存在的重写，这就是协变。

并且，还有一点也许会有疑问，子类中的巧方法`Object getValue()`和`Date getValue()`是同
时存在的，可是如果是常规的两个方法，他们的方法签名是一样的，也就是说虚拟机根本不能分别这两个方法。如果是我们自己编写Java代码，这样的代码是无法通过编译器的检查的，但是虚拟机却是允许这样做的，因为虚拟机通过参数类型和返回类型来确定一个方法，所以编译器为了实现泛型的多态允许自己做这个看起来“不合法”的事情，然后交给虚拟器去区别。

泛型类型变量不能是基本数据类型

不能用类型参数替换基本类型。就比如，没有`ArrayList<double>`，只有`ArrayList<Double>`。因为当类型擦除后，`ArrayList`
的原始类型变为`Object`，但是`Object`类型不能存储`double`值，只能引用`Double`的值。

编译时集合的instanceof

```
ArrayList<String> arrayList=new ArrayList<String>();
```

因为类型擦除之后，`ArrayList<String>`只剩下原始类型，泛型信息`String`不存在了。

那么，编译时进行类型查询的时候使用下面的方法是错误的

```
if(arrayList instanceof ArrayList<String>)
```

泛型在静态方法和静态类中的问题

泛型类中的静态方法和静态变量不可以使用泛型类所声明的泛型类型参数

举例说明：

```
public class Test2<T> {
    public static T one;    //编译错误

    public static T show(T one) { //编译错误
        return null;
    }
}
```

因为泛型类中的泛型参数的实例化是在定义对象的时候指定的，而静态变量和静态方法不需要使用对象来调用。对象都没有创建，如何确定这个泛型参数是何种类型，所以当然是错误的。

但是要注意区分下面的一种情况：

```
public class Test2<T> {

    public static <T> T show(T one) { //这是正确的
        return null;
    }
}
```

因为这是一个泛型方法，在泛型方法中使用的T是自己在方法中定义的 `T`，而不是泛型类中的`T`。

容器(Collection):

`Collection`接口继承了一个接口: `Interable`(可迭代)

任何容器都可以转换成数组

`List` 接口(线性表)继承了容器`Collection`接口

任何线性表都应该实现List接口

无序容器无法通过下标访问, 必须实现`Interable`接口, 通过迭代访问

Collection:

- List
- Set
- Queue
- Map

注意: 容器中, 不能保存基本数据类型

List接口:

- ArrayList 非线程安全
- Vector 和ArrayList用法一模一样, 只是此类是线程安全的
- LinkedList
- .....

ArrayList:

修饰符和返回值	方法	描述
public boolean	add(E e)	增加一个元素
public void	add(int index, E element)	在对应下标插入一个元素
public E	set(int index, E element)	更新对应下标的元素
public E	get(int index)	获取对应下标的元素的值
public boolean	contains(Object o)	判断是否包含
public E	remove(int index)	删除对应下标的元素
public boolean	remove(Object o)	删除对应元素, 自处不能应该传包装类对象
public List	sumList(int fromIndex, int toIndex)	切割列表

源码解析:

1. Vector默认容量是10
2. ArrayList根据调用不同的构造函数, 初始化容量不同
    - 无参构造容量默认是0
        - 一旦添加元素就会发生一次扩容, 默认容量是10
    - int类型参数有参构造容量是传递的值
3. Vector是线程安全的容器, 操作方法都加了同步锁
4. ArrayList是非线程安全的容器
5. 如果新扩容后的长度大于整形的最大值, 就扩容Integer.MAX_VALUE - 8

forEach遍历数组, 使用forEach需要一个实现类

```
// 第一种：内部类
class Test1 {
    void test() {
        List<Integer> list = new ArrayList<>();
        list.forEach(new MyForEach());
    }

    class MyForEach implements Consumer<Integer> {
        @Override
```

```
        public void accept(Integer t) {
            System.out.println(t);
        }
    }
}

// 第二种：匿名内部类
class Test {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.forEach(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) {
                System.out.println(integer);
            }
        });
    }

}

// 第三种：lambda表达式
// 要使用此方法，接口必须只有一个未实现的方法
class Test {
    void test() {
        List list = new ArrayList();
        list.forEach(list);
        list.forEach(t -> {
            System.out.println(t);
        });
    }
}

class Test {
    void test() {
        List list = new ArrayList();
        list.forEach(list);
        list.forEach(t -> System.out.println(t));
    }
}

class Test {
    void test() {
        List list = new ArrayList();
        list.forEach(list);
        list.forEach(System.out::println);
    }
}
```

```
59
60
```

LinkedList不仅是双向链表, 也是一个双向队列

Set接口:

集合中存储的都是对象的地址, 不能存储基本数据类型, 需要包装类

无序列表迭代:

- foreach迭代器
- forEach对象

HashSet:

java标准的哈希表

非线程安全

保证元素唯一的方式依赖于: hashCode()和equals()方

底层用HashMap实现

HashSet和HashMap可以说是一个东西

add() 成功返回true, 失败返回false

什么是哈希表:

无序不重复的非线性结构

哈希表底层使用的也是数组机制, 数组中也存放对象, 而这些对象往数组中存放时的位置比较特
殊, 当需要把这些对象给数组中存放时, 那么会根据这些对象的特有数据结合相应的算法, 算法这个对
象在数组中的位置, 然后把这个对象存放在数组中. 而这样的数组就称为哈希数组, 即就是哈希表.

当给哈希表中存放元素时, 需要根据元素的特有数据结合相应的算法, 这个算法其实就是Object中 的hashCode方法. 由于任何对象都是Object类的子类, 所以任何对象有拥有这个方法. 即就是在给哈希 表中存放对象时, 会调用对象的hashCode方法,
算出对象在表中的存放位置, 这里需要注意, 如果两个 对象hashCode方法算出结果一样, 这样现象称为哈希冲突, 这时会调用对象的equals方法,
比较这两个 对象是不是同一个对象, 如果equals方法返回的是true, 那么就不会把第二个对象存放在哈希表中, 如 果返回的是false,
就会把这个值存放在哈希表中.

总结: 保证元素的唯一, 其实就是根据对象的hashCode和equals方法来决定的. 如果我们往集合中存放自定义的对象, 那么保证其唯一,
就必须复写hashCode和equals方法建立属于当前对象的比较方式.

TreeSet:

底层使用二叉树(红黑树)

这是一个有序的集合

TreeSet存储自定义类(实现Comparable):

需要指定一个排序规则

在java中如果对象需要比较, 则必须让该类实现Comparable接口, 然后重写compareTo()方法

```
class Person implements Comparable<Person> {
    int id;
    String name;

    @Override
    public int compareTo(Person o) {
        return this.id - o.getId();
        // 返回正数为升序, 负数为降序, 0 为不变
    }
}
```

通过CompareTo()方法就可以实现元素的排序

LinkedHashSet:

线性哈希表, 传入顺序是什么存储顺序就是什么

Map接口:

- HashMap
- Hashtable
- ConcurrentHashMap

Map接口是一种键值对式的结构, 底层使用hash表实现的

通过key找到值, 不能通过值找到key

key是唯一的, 不能重复, 值可以重复

HashMap:

线程不安全

是一个标准的哈希表

返回值	抽象方法	描述
V	put(K key, V value)	当key重复时不会创建多个会覆盖
V	get(Object key)	key 不存在返回空
V	remove(Object key)	移除键值对
V	remove(Object key, Object value)	键值对都匹配时才移除对应键值对
default V	getOrDefault(Object key, V defaultValue)	如果找不到key相匹配的键值对返回设置的默认值(value), 找到则返回key对应的值
static <K, V> Entry<K, V>	entrySet()	转换为Entry对象Set
Set	keySet()	将key转换成Set
default void	forEach(BiConsumer<? super K, ? super V> action)	迭代方法
boolean	containsValue(Object value)	判断是否包含值
boolean	containsKey(Object key)	是否包含键

HashMap中可以以null作为key

Entry对象是一个键值对

遍历:

```
class Test {
    Map<String, String> mps = new HashMap();

    // 用keySet迭代
    void test1() {
        Set<String> keySet = mps.keySet(); // 首先将map的key做成set再迭代数据
        Interator<String> keys = keySet.iterator();
        while (key.hashNext()) {
            String key = key.next();
            System.out.println(key + ":" + mps.get(key));
        }
    }

    // 用entrySet迭代
    void test2() {
        Set<Entry<String, String>> entrySet = mps.entrySet();
        // 使用lambda表达式简化
        entrySet.forEach(t -> System.out.println(t.getKey() + " : " +
t.getValue()));
```

```
    }

    // foreach简化迭代器
    void test3() {
        for (String key : mps.keySet) {
            System.out.println(key + " : " + mps.get(key));
        }

        for (Entry<String, String> entry : mps.entySet) {
            System.out.println(entry.getKey() + " : " + entry.value());
        }
    }


    // forEach方法
    void test4() {
        mps.forEach((k, v) -> System.out.println(k + " : " + v));
    }
}
```

统计单词数量:

```
class Test {
    void test() {
        Map<String, Integer> hashmap = new HashMap<>();
        String str = "this book is so good, I like this book";
        String s = str.replaceAll(",|,|\\.|\r|\n|:", " ");
        String[] str2 = s.split("\\s+");
        System.out.println(s);
        for (String item : str2) {
            Integer count = hashmap.getOrDefault(item, 0);
            hashmap.put(item, ++count);
        }

        Set<Map.Entry<String, Integer>> entrySet = hashmap.entrySet();
        entrySet.forEach(t -> System.out.println(t.getKey() + " : " +
t.getValue()));
    }
}
```

Hash冲突:

hash冲突是指, 两个不同的值计算出的hash值是一样的, 这时插入位置是一样的, 就产生了冲突, 发生了hash碰撞

四种解决方案:

- 再哈希法
  - 使用大量的hash算法, 一种冲突就换另一种, 效率比较低
- 开放地址法
  - 如果插入的位置已经有元素存在就换一个位置插入
- 链地址法(Java的HashMap采用此法)
  - 相同hash值的元素会形成一条链表
    - 链表(七上八下, 数组长度大于64链表长度大于8就会变成二叉树(红黑树))
    - 二叉树(红黑树, 数据量小于6时会转换成链表)
- 公共缓冲区(使用较少)

HashMap结构:

JDK8之前, HashMap 整体结构是一个 数组 + 链表 的结构

JDK8之后, HashMap 整体结构是一个 数组 + 链表 + "红黑树" 的结构, 目的是加快查询效率

HashMap源码解析:

1. 通过求模计算出插入位置将元素插进去
2. 如果要插入的已存在元素使用链地址法
3. 链表长度大于8且数组长度大于64时, 就会转换为二叉树(红黑树)
4. 红黑树的节点少于6时, 就会还原为链表
5. HashMap默认数组的容量是16, 每次扩容必须保证为2的幂次方倍
6. HashMap的加载因子是0.75, 因为此时**发生哈希碰撞的概率最小**, 由泊松分布得出. **表示Hash表中元素的填满程度**. *
   *加载因子越大,填满的元素越多,空间利用率越高,但发生冲突的机会变大了**
7. 在构造时如果指定容量没有传入2的幂次方, HashMap会将传入值变成最接近的2的幂次方倍的值
8. put()方法:
   1. 如果键是一个空值就插入第0个位置; 不为空则通过调用底层的hashCode()方法并且用返回的值做了一个高低位异或,
      这样做得目的是为了更加散列化, 让二进制高低位都参与进来, 降低碰撞概率.如果不做高低为异或, 数据会集中在低位,
      这样碰撞概率会变高
   2. 添加节点:

      如果数组没有初始化同时长度是0, 调用一次resize()方法发生一次扩容

      如果是第一次添加节点, 则容量是16, 负载因子是0.75, 触发扩容的阀值: 12
   3. 插入时, 求插入key的下标, 获取hashcode值, 求模运算(hash % length <===> (length - 1) & hash)

一个数模2的幂次方倍, 用右移运算, 移出去的是余数剩下的是商数, 要拿到移出去的数用与运算,

Hashtable:

Hashtable是线程安全的

默认容量是11, 加载因子是0.75

扩容是以前容量的二倍 + 1

ConcurrentHashMap:

线程安全, 采用分段加锁的方式, 效率至少是Hashtable的16倍, 高并发场景下建议使用ConcurrentHashMap

ConcurrentHashMap在JDK8做了一次优化, 优化了加锁方式, 引入了偏向锁(无锁)和自旋锁

偏向锁仅仅是一种偏向而不是加锁

Collections工具类:

Comparable: 在Java中所有需要排序比较的对象, 必须实现这个接口, 来实现比较规则

Comparator: 需要重定义规则的情况下使用, 临时指定

- public static <T extends Comparable<? super T>> void sort(List<T> list // 传入的List需要实现可比较接口(Comparable)
- public static <T> void sort(List<T> list, Comparator<? super T> c) // 需要临时改变排序规则时使用

```
class Person implements Comparable<Person> {
    int id;
    String name;

    @Override
    public int compareTo(Person o) {
        return this.id - o.getId();
        //返回正数为升序, 负数为降序, 0 为不变
    }

    public void test() {
        Person p = new Person();
        Collections.sort(p);
        Collections.sort(p, new Comparator<Perspm>() {
            @Override
            public int compare(Person o1, Person o2) {
                return 0;
            }
        });
        Collections.sort(p, (o1, o2) -> o1.getId - o2.getId);
    }
}
```

IO流:

Input Output Stream

狭义: 数据在内存中输入和输出, 本地进程间的数据流动

广义: 不同电脑之间的数据流动, 远程进程间的数据流动

流的流动方式(数据的传输方向) :

- 输入流
- 输出流

流的数据格式:

- 字节流
- 字符流

File对象:

Java封装的一个操作文件及文件夹的对象

返回值	方法	描述
	File(String pathname)	构造方法
	File(String parent, String child)	构造方法
	File(File parent, String child)	构造方法
	File(URI uri)	构造方法, 用于网络
boolean	createNewFile() throws IOException	创建文件, 已存在就返回false, 不存在就新建, 需要处理异常
boolean	exists()	是否存在
boolean	canExecute()	是否有执行权限
boolean	canWrite()	是否有写权限
boolean	canRead()	是否有读权限
boolean	delete()	删除文件
void	deleteOnExit()	在JVM退出时删除文件
File	getAbsoluteFile()	返回绝对路径对象
String	getAbsolutePath()	返回绝对路径
long	getFreeSpace()	空闲空间
long	getUsableSpace()	可用空间
long	getTotalSpace()	总空间
long	length()	返回文件大小

String	getName()	获取文件名
String	getParent()	获取父级路径, 必须使用绝对路径对象
String	getPath()	返回全路径
String	getParentFile()	
boolean	isHidden()	是否为隐藏文件
long	lastModified()	返回最后修改时间戳
boolean	isDirectory()	是否为一个文件夹
boolean	mkdir()	创建文件夹, 只能创建一个不能递归创建
boolean	mkdirs()	递归创建文件夹
boolean	renameTo(File dest)	剪切, 移动文件
String[]	list()	将文件夹下的文件返回成一个字符串数组
File[]	listFiles()	将文件夹下的文件返回成一个File对象数组
String[]	list(FilenameFilter filter)	过滤文件
File[]	listFiles(FilenameFilter filter)	过滤文件

IO流分类:

- InputStream 字节输入流
- OutputStream 字节输出流
- Reader 字符输入流
- Writer 字符输出流

字节流:

InputStream(抽象类):

直接子类:

- AudioStream
- ByteArrayInputStream
- FileStream
- FilterInputStream
- ObjectInputStream
- PipedInputStream
- SequenceInputStream
- StringBufferInputStream

write() 写文件

int read() 读文件

close() 关闭流

flush() 刷新缓冲区

FileInputStream:

FileInputStream(String name)

FileInputStream(File file)

read() 读取文件

```
class Test {
    void test() {
        InputStream file = null;
        try {
            file = new FileInputStream(new File("./a.txt")); // 创建一个输入流对象
            byte[] buf = new byte[1024]; // 用于缓冲, 大小用2的整数倍, 最好用4k(4k对齐)
            int len = 0;
            while ((len = file.read(buf)) != -1) {
                // System.out.write(buf); // 此方法可能导致读取内容超出
                System.out.write(buf, 0, len);
                String str = new String(buf, 0, len);
                System.out.println(str);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (file != null) {
                try {
                    file.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

OutputStream(抽象类):

- FileOutputStream

FileOutputStream:

```
class Test {
    void test() {
        String msg = "你好";
        FileOutputStream file = null;
```

```
        try {
            file_out = new FileOutputStream(new File("b.txt"));
            file_in = new FileInputStream(new File("a.txt"));
            byte[] buf = new byte[1024];
            int len = 0;
            while ((len = file_in.read(buf)) != -1) {
                file_out = write(buf, 0, len);
            }
            System.out.ptintln("写入数据成功");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (file_in != null) {
                try {
                    file_in.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (file_out != null) {
                try {
                    file_out.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

FilterInputStream & FilterOutputStream:

装饰流, 又称为过滤流

不能 直接使用, 主要的作用就是用来装饰节点流

本质是一种装饰者设计模式的体现. 对原有对象的功能进一步的装饰增强

GOF 23设计模式:

- 单例设计模式
- 装饰者设计模式

子类:

- BufferedInputStream
- BufferedOutputStream
- DataInputStream
- DataOutputStream

BufferedInputStream & BufferedOutputStream:

```
class Test {
    void test() {
        BufferedInputStream bis = null; // 缓冲池
        BufferedOutputStream bos = null;
        try {
            bis = new BufferedInputStream(new FileInputStream("a.txt")); // 建议把字节流写在里面, 这样不用单独关闭
            bos = new BufferedOutputStream(new FileOutputStream("b.txt"));
            byte[] buf = new byte[1024 * 8];
            int len = 0;
            while ((len = bis.read(buf)) != -1) {
                bos, write(buf, 0, len);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // 缓存流必须关闭, 因为JVM
            if (bis != null) {
                try {
                    bis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (bos != null) {
                try {
                    bos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

DataOutputStream:

可以不用转换成字节数组直接就进行写入

```
class Test {
    void test() {
        BufferedOutputStream bos = null;
        int msg = 1000000;
        try {
            dos = new DataOutputStream(new FileOutputStream("b.txt"));
```

```
            dos, writeInt(msg);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (dos != null) {
                try {
                    dos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

DataInputStream:

```
class Test {
    void test() {
        BufferedOutputStream bos = null;
        try {
            dis = new DataInputStream(new FileInputStream("a.txt"));
            System.out.println(dis.readInt());
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (dis != null) {
                try {
                    dis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

字符流:

计算机底层使用的二进制数据(字节数据)

字符流是为了加快流的操作, 而设计的专门用来操作字符串的一种流

注意: 字符串存在编码问题, 需要保证在读取和写入编码一致

建议使用装饰流按行读取

Reader(输入流):

```
class Test {
    void test() {
        Reader reader = null;
        try {
            reader = new FileReader(new File("/Users/erzbir/Desktop/IntelliJ Java/JavaTest/a.txt"));
            char[] buf = new char[1024];
            int len = 0;
            while ((len = reader.read(buf)) != -1) {
                writer.write(buf, 0, len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Writer(输出流):

```
class Test {
    void test() {
        Reader reader = null;
        Writer writer = null;
        try {
            writer = new FileWriter("/Users/erzbir/Desktop/IntelliJ Java/JavaTest/b.txt");
            reader = new FileReader(new File("/Users/erzbir/Desktop/IntelliJ Java/JavaTest/a.txt"));
            char[] buf = new char[1024];
            int len = 0;
            while ((len = reader.read(buf)) != -1) {
                writer.write(buf, 0, len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (reader != null) {
                try {
                    reader.close();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (writer != null) {
                try {
                    writer.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

BufferedReader & BufferedWriter:

```
class Test {
    void test() {
        BufferedReader reader = null;
        BufferedWriter writer = null;
        try {
            writer = new BufferedWriter(new
FileWriter("/Users/erzbir/Desktop/IntelliJ Java/JavaTest/b.txt"));
            reader = new BufferedReader(new FileReader(new
File("/Users/erzbir/Desktop/IntelliJ Java/JavaTest/a.txt")));
            char[] buf = new char[1024];
            int len = 0;
            while (reader.readLine(buf) != null) {
                writer.write(buf + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (writer != null) {
                try {
                    writer.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
```

```
    }
}
```

PrintWriter(打印输出流):

可以直接操作文件, 也可以操作对象

关闭时不会抛出异常

```
class Test {
    void test() {
        Reader reader = null;
        PrintWriter writer = null;
        try {
            PtintWriter = new PrintWriter("/Users/erzbir/Desktop/IntelliJ Java/JavaTest/b.txt");
            reader = new BufferedReader(new FileReader(new File("/Users/erzbir/Desktop/IntelliJ Java/JavaTest/a.txt")));
            char[] buf = new char[1024];
            while (reader.readLine(buf) != null) {
                writer.println(buf, 0, len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (writer != null) {
                try {
                    writer.close();
                } catch (IOExcetion e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

转换流:

将字节流转换为字符流操作

当操作的是文本数据的时候使用

InputStreamReader:

可以将字节输入流转换为字符输入流

```
class Test {
    void test() throws IOException {
        BufferedReader br = null;
        br = new BufferedReader(new InputStreamReader(System.in));
        String str = null;
        while ((str = br.readLine()) != null) {
            System.out.println(str);
        }
        br.close();
    }
}
```

OutputStreamWriter:

字节输出流转换成字符输入流

```
class Test {
    void test() throws IOException {
        BufferedReader br = null;
        PrintWriter out = null;
        br = new BufferedReader(new InputStreamReader(System.in));
        out = new PrintWriter(new OutputStreamWriter(new
FileOutputStream("b.txt")));
        String str = null;
        while ((str = br.readLine()) != null) {
            out.println(str);
        }
        br.close();
        out.close();
    }
}
```

对象流:

ObjectInputStream:

```
class Test {
    void test() {
        String msg;
        ObjectOutputStream ois = null;
        try {
            ois = new ObjectOutputStream(new FileOutputStream("a.txt"));
            msg = ois.readObject();
            System.out.ptintln(msg);
        } catch (IOException e) {
```

```
            e.printStackTrace();
        } finally {
            if (ois != null) {
                try {
                    ois.close();
                } catch (IOException e) {
                    e.ptintStackTrace();
                }
            }
        }
    }
}
```

ObjectOutputStream

```
class Test {
    void test() {
        String msg = "1asda";
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream("a.txt"));
            oos.writeObject(msg);
            System.out.ptintln("保存成功");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (oos != null) {
                try {
                    oos.close();
                } catch (IOException e) {
                    e.ptintStackTrace();
                }
            }
        }
    }
}
```

对象序列化:

Serialize: 将虚拟对象转换为一种可以直接传输或者保存的数据(字节, 字符)的过程

对象反序列化:

将序列化后的字节或者字符数据源重新转换为对象

对象持久化:

将数据永久保存(存到磁盘)

java官方提供的序列化, 是将java对象转换为字节数据

注意: java对象序列化必须实现可序列化接口(Serializable), 这是一个标记接口自动调用底层

transient关键字:

被这个关节字修饰的属性, 无法持久化

try-with-resources

jdk7的新特性, 如果使用这种结构, 打开的资源会自动完成关闭

```
class Test {
    void test() {
        try (InputStream is = new FileInputStream("a.txt")) {
            byte[] buf = new byte[1024];
            int len = 0;
            while ((len = is.read(buf)) != -1) {
                System.out.println(new String(buf, 0, len));
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

在开发过程中, 如果直接将一些确定的值写在代码中, 代码的设计可能有问题, 如果生产环境, 要再次修改值会非常麻烦, 需要重新编译.
这种叫做硬编码

开发环境: development environment

测试环境: test environment

生产环境: product environment

如果值永久不变:

- 做成常量
- 做成枚举

如果有可能变:

可以做成配置文件

Java配置文件:

- xml
- json
- yaml
- properties

Properties:

解析properties文件

底层是hashtable

如果配置文件在src或者其他类路径中, 如果类名称.class.getClassLoader().getResourceAsStream(name)拿到类的字节码文件再拿到加载器再拿到对应的资源文件

多线程编程:

多任务:

真正的多任务是在多核CPU之后, 在此之前所有多任务都是伪多任务

高并发: 多个任务抢占少量资源

高并发三大要素:

- 可见性
- 原子性
- 排序性
    - Java代码, 底层最后都会编译成汇编指令, 汇编指令做优化时代码执行顺序可能改变, 这样可能导致双重检查锁失效

时间片: 每个任务运行的时间, 时间片耗尽就必须退出进入等待队列

优先级别调度:

操作系统底层的多任务使用的就是时间片轮换机制, 配合优先级别调度完成

- 多进程编程
- 多线程编程
- 协程编程

进程: process

- 操作系统管理的基本单位, 一个进程可以看成一个软件, 一个端口, 直接申请独立的内存
- 一个软件可以由多个进程组成
- 进程包含线程
- 进程之间相互独立, 一个进程挂掉不会影响其他进程

线程: thread

- 线程是最小量级的进程, 依赖于进程
- 线程是独立的, 每个线程就是一个栈
- 线程占有的是每一个进程的内存空间
- 一个线程挂掉可能影响整个进程, 不稳定

协程: coroutine

-

Java提供了四种创建多线程的方案:

1. 基础Thread
2. 实现Runnable接口
3. 实现Callable接口
4. 线程池实现多线程

如何创建线程:

继承Thread类, 重写run方法

写在run方法里的就是线程方法, 用start()开启线程

使用继承 Thread 类的方式创建多线程时, 如果需要访问当前线程,则无需使用 Thread.currentThread() 方法,直接使用 this 即可获得当前线程

main函数是主线程

```
class Test extends Thread {
    public static void main(String[] args) {
        Test test = new Test();
        test.start();
        for (int i = 0; i < 100; i++) {
            System.out.println("主线程也在运行----" + i);
        }
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println("一个单独的线程运行了------" + i);
        }
    }
}
```

实现Runnable接口:

使用Runnable接口实现的多线程类没有start()方法, 启动线程必须借助Thread类的构造函数

Thread.currentThread().getName()获取进程名

```
class Test implements Runnable {

    public static void main(String[] args) {
        Test a1 = new Test();
        Test a2 = new Test();
        new Thread(a1).start();
        new Thread(a2).start();
    }

    @Override
    public void run() {
```

```
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() + "一个单独的线程运行了------" + i);
        }
    }
}
```

实现Callable接口:

可以有返回值

开启线程需要借助FutureTask, 再借助Thread

```
class Test implements Callable<Integer> {
    public static void main(String[] args) {
        Test test = new Test();
        FutureTask<? extends Integer> futureTask = new FutureTask<>(test);
        new Thread(futureTask).start();
        try {
            Integer a = futureTask.get(); // 使用Callable接口实现的多线程对象, 最后能够得到线程方法返回的结果
            System.out.println(a);
        } catch (Exception e) {
            e.printStackTrace();
        }
        for (int i = 0; i < 1000; i++) {
            System.out.println(Thread.currentThread().getName() + "主线程开始运行");
        }
    }

    @Override
    public Integer call() {
        int sum = 0;
        for (int i = 0; i < 1000; i++) {
            sum += 1;
            System.out.println("使用Callable接口创建的子线程" + i);
        }
        return sum;
    }
}
```

线程池:

池化模式: 将大量需要的对象提前创建好, 放在一个池中

对象提前创建完成, 也不需要销毁对象, 所以说使用效率比较好

优点: 使用效率比较好, 避免对象的重复创建和销毁

缺点: 内存占用高, 池的数量难以把控

池的数量把控问题是最关键的

Java线程池是1.5提供的, 底层是Callable和Future接口

根据场景创建不同的线程池

Executors:

参考: https://blog.csdn.net/qq_40093255/article/details/116990431

使用这个类根据需要创建线程池

ExecutorService继承了Executors, 一般用ExecutorService

单例线程池(SingleThreadExecutor):

> 所有任务都保存队列LinkedBlockingQueue中，核心线程数为1，线程空闲时间为0
> 等待唯一的单线程来执行任务，并保证所有任务按照指定顺序(FIFO或优先级)执行

```
class Test {
    void test() {
        // 创建一个单例的线程池
        // 单例线程池：只有一个线程，固定不会变化的一个线程
        // 使用场景：只有一个线程时，建议使用这种线程池
        ExecutorService executor = Executors.newSingleThreadExecutor();
        for (int i = 0; i < 10; i++) {
// 如果想要将外部的变量传递到匿名内部类中，可以做成常量
            final int finalI = i;
            executor.execute(() ->
System.out.println(Thread.currentThread().getName() + "--->" + finalI));
        }
    }
}
```

固定大小线程池(FixedThreadPool):

> 指定线程池的固定大小，对于超出的线程会在LinkedBlockingQueue队列中等待
>
> 核心线程数可以指定，线程空闲时间为0

```
class Test {
    void test() {
        // 创建一个固定大小的线程池
        // 适用场景：并发量不会发生变化(非常小)
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10; i++) {
// 如果想要将外部的变量传递到匿名内部类中，可以做成常量
            final int finalI = i;
            executor.submit(() ->
System.out.println(Thread.currentThread().getName() + "--->" + finalI));
        }
    }
}
```

缓存线程池(CachedThreadPool):

> 可缓存无界线程池测试
>
> 当线程池中的线程空闲时间超过60s则会自动回收该线程，核心线程数为0
>
> 当任务超过线程池的线程数则创建新线程。线程池的大小上限为Integer.MAX_VALUE

```
class Test {
    void test() {
        // 创建一个可变的线程池
        // 基于缓存，创建一个缓存的线程池，若线程数不够则新建线程
        // 并发量变化比较明显的建议使用这种线程池
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i = 0; i < 20; i++) {
// 如果想要将外部的变量传递到匿名内部类中，可以做成常量
            final int finalI = i;
            executor.submit(() ->
System.out.println(Thread.currentThread().getName() + "--->" + finalI));
        }
    }
}
```

延迟任务线程池(ScheduledThreadPool):

- schedule(Runnable command, long delay, TimeUnit unit)，延迟一定时间后执行Runnable任务；
- schedule(Callable callable, long delay, TimeUnit unit)，延迟一定时间后执行Callable任务；
- scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)，延迟一定时间后，以间隔period时间的频率周期性地执行任务；
- scheduleWithFixedDelay(Runnable command, long initialDelay, long delay,TimeUnit unit)，与scheduleAtFixedRate()方法很类似，
- 但是不同的是scheduleWithFixedDelay()
  方法的周期时间间隔是以上一个任务执行结束到下一个任务开始执行的间隔，而scheduleAtFixedRate()
  方法的周期时间间隔是以上一个任务开始执行到下一个任务开始执行的间隔，
- 也就是这一些任务系列的触发时间都是可预知的。

- ScheduledExecutorService功能强大，对于定时执行的任务，建议多采用该方法。

```
class Test {
    void test() throws InterruptedException {
        // 创建的线程具有延时执行的特点
        ScheduledExecutorService executor = Executors.newScheduledThreadPool(10);
        for (int i = 0; i < 20; i++) {
// 如果想要将外部的变量传递到匿名内部类中，可以做成常量
            final int finalI = i;
            // 如果用execute()方法则没有延迟
            // 第一个参数：需要执行的任务
            // 第二个参数：延迟时长
            // 第三个参数：时间单位
            executor.schedule(() ->
System.out.println(Thread.currentThread().getName() + "--->" + finalI), 5,
TimeUnit.SECONDS);
            TimeUnit.SECONDS.sleep(5);
        }
    }
}
```

```
class Test {
    void test() {
        // 创建指定核心线程数，但最大线程数是Integer.MAX_VALUE的可定时执行或周期执行任务的线程
池
        ScheduledExecutorService executorService =
Executors.newScheduledThreadPool(5);

        // 定时执行一次的任务，延迟1s后执行
        executorService.schedule(new Runnable() {
            @Override
            public void run() {
                print("scheduleThreadPool");
                System.out.println(Thread.currentThread().getName() + ", delay
1s");
            }
        }, 1, TimeUnit.SECONDS);

        // 周期性地执行任务，延迟2s后，每3s一次地周期性执行任务
        executorService.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + ", every
3s");
            }
        }, 2, 3, TimeUnit.SECONDS);


```

```
        executorService.scheduleWithFixedDelay(new Runnable() {
            @Override
            public void run() {
                long start = new Date().getTime();
                System.out.println("scheduleWithFixedDelay 开始执行时间:" +
                        DateFormat.getTimeInstance().format(new Date()));
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                long end = new Date().getTime();
                System.out.println("scheduleWithFixedDelay执行花费时间=" + (end -
start) / 1000 + "m");
                System.out.println("scheduleWithFixedDelay执行完成时间："
                        + DateFormat.getTimeInstance().format(new Date()));
                System.out.println("=====================================");
            }
        }, 1, 2, TimeUnit.SECONDS);

    }
}
```

抢占线程池(WorkStealingPool):

它会通过工作窃取的方式，使得多核的 CPU 不会闲置，总会有活着的线程让 CPU 去运行。

基于工作窃取算法，其中任务可以生成其他较小的任务，这些任务将添加到并行处理线程的队列中。如果一个线程完成了工作并且无事可做，则可以从另一线程的队列中“窃取”工作。

```
class Test {
    void test() throws InterruptedException {
        System.out.println("获得JAVA虚拟机可用的最大CPU处理器数量：" +
Runtime.getRuntime().availableProcessors());
        ExecutorService executorService = Executors.newWorkStealingPool();
        /**
         * call方法存在返回值futureTask的get方法可以获取这个返回值。
         * 使用此种方法实现线程的好处是当你创建的任务的结果不是立即就要时，
         * 你可以提交一个线程在后台执行，而你的程序仍可以正常运行下去，
         * 在需要执行结果时使用futureTask去获取即可。
         */
        List<Callable<String>> callableList = IntStream.range(0, 20).boxed().map(i
-> (Callable<String>) () -> {
            TimeUnit.SECONDS.sleep(3);
            System.out.println(String.format("当前【%s】线程正在执行>>>",
Thread.currentThread().getName()));
            return "callable type thread task: " + i;
        }).collect(Collectors.toList());

        // 执行给定的任务，返回持有他们的状态和结果的所有完成的期待列表。
```

```
        executorService.invokeAll(callableList).stream().map(futureTask -> {
            try {
                return futureTask.get();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
            return null;
        }).forEach(System.out::println);
    }
}
```

```
class Test {
    void test() {
        // 创建线程池
        ExecutorService threadPool = Executors.newWorkStealingPool();
        // 执行任务
        for (int i = 0; i < 10; i++) {
            final int index = i;
            threadPool.execute(() -> {
                System.out.println(index + " 被执行,线程名:" +
Thread.currentThread().getName());
            });
        }
        // 确保任务执行完成
        while (!threadPool.isTerminated()) {
        }
    }
}
```

单例延迟线程池(SingleThreadScheduledExecutor):

```
class Test {
    void test() {
        // 创建线程池
        ScheduledExecutorService threadPool =
Executors.newSingleThreadScheduledExecutor();
        // 添加定时执行任务(2s 后执行)
        System.out.println("添加任务,时间:" + new Date());
        threadPool.schedule(() -> {
            System.out.println("任务被执行,时间:" + new Date());
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
            }
        }, 2, TimeUnit.SECONDS);
    }
```

```
}
```

原始线程池(ThreadPoolExecutor):

ThreadPoolExecutor可以设置7个参数, 如下:

`public ThreadPoolExecutor (int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler)`

```

// 前5个参数为必填参数

// 1: 核心线程数, 线程中始终存活的线程数 >= 0
// 2: 最大线程数, 线程池中允许的最大线程数 >= 1
// 3: 最大线程数可以存活的时间, 当线程中没有任务执行时, 最大线程就会销毁一部分最终保持核心线程数量的线程 >= 0
// 4: 时间单位
// 5: 一个阻塞队列, 用来存储线程池等待执行的任务, 均为线程安全
// 6: 线程工厂, 主要用来创建线程, 默认为正常优先级\非守护线程
// 7: 线程饱和策略/拒绝策略, 拒绝处理任务时的策略
```

详解:

- corePoolSize（线程池基本大小）：

  > 当向线程池提交一个任务时，若线程池已创建的线程数小于corePoolSize，即便此时存在空闲线程，也会通过创建一个新线程来执行该任务，直到已创建的线程数大于或等于corePoolSize时，才会根据是否存在空闲线程，来决定是否需要创建新的线程。除了利用提交新任务来创建和启动线程（按需构造），也可以通过
  > prestartCoreThread() 或 prestartAllCoreThreads() 方法来提前启动线程池中的基本线程。

- maximumPoolSize（线程池最大大小）：

  > 线程池所允许的最大线程个数。当队列满了，且已创建的线程数小于maximumPoolSize，则线程池会创建新的线程来执行任务。另外，对于无界队列，可忽略该参数。

- keepAliveTime（线程存活保持时间）：

  > 默认情况下，当线程池的线程个数多于corePoolSize时，线程的空闲时间超过keepAliveTime则会终止。但只要keepAliveTime大于0，allowCoreThreadTimeOut(
  > boolean) 方法也可将此超时策略应用于核心线程。另外，也可以使用setKeepAliveTime()动态地更改参数。

- unit（存活时间的单位）：

  > 时间单位，分为7类，从细到粗顺序：NANOSECONDS（纳秒），MICROSECONDS（微秒），MILLISECONDS（毫秒），SECONDS（秒），MINUTES（分），HOURS（小时），DAYS（天

- workQueue（任务队列）：

> 用于传输和保存等待执行任务的阻塞队列。可以使用此队列与线程池进行交互：
>
> - 如果运行的线程数少于 corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队。
> - 如果运行的线程数等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。
> - 如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。

- threadFactory（线程工厂）：

> 用于创建新线程。由同一个threadFactory创建的线程，属于同一个ThreadGroup，创建的线程优先级都为Thread.NORM_PRIORITY，以及是非守护进程状态。threadFactory创建的线程也是采用new Thread()方式，threadFactory创建的线程名都具有统一的风格：pool-m-thread-n（m为线程池的编号，n为线程池内的线程编号）；

- handler（线程饱和策略）：

> 当线程池和队列都满了，则表明该线程池已达饱和状态。
>
> - ThreadPoolExecutor.AbortPolicy：处理程序遭到拒绝，则直接抛出运行时异常 RejectedExecutionException。(默认策略)
> - ThreadPoolExecutor.CallerRunsPolicy：调用者所在线程来运行该任务，此策略提供简单的反馈控制机制，能够减缓新任务的提交速度。
> - ThreadPoolExecutor.DiscardPolicy：无法执行的任务将被删除。
> - ThreadPoolExecutor.DiscardOldestPolicy：如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重新尝试执行任务（如果再次失败，则重复此过程）。

队列排队详解

- 直接提交

> 工作队列的默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

- 无界队列

> 使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize 的值也就无效了。）当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web 页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

- 有界队列

> 当使用有限的 maximumPoolSizes 时，有界队列（如
> ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能
> 需要相互折衷：使用大型队列和小型池可以最大限度地降低
> CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例
> 如，如果它们是 I/O
> 边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU 使
> 用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。

工作队列BlockingQueue详解

BlockingQueue的插入/移除/检查这些方法，对于不能立即满足但可能在将来某一时刻可以满足的操作，共有4种不同的处理方式：第一种是抛出一个异常，第二种是返回一个特殊值（null
或 false，具体取决于操作），第三种是在操作可以成功前，无限期地阻塞当前线程，第四种是在放弃前只在给定的最大时间限制内阻塞。如下表格：

操作	抛出异常	特殊值	阻塞	超时
插入	add(e)	offer(e)	put(e)	offer(e, time, unit)
移除	remove()	poll()	take()	poll(time, unit)
检查	element()	peek()	不可用	不可用

实现BlockingQueue接口的常见类如下：

- ArrayBlockingQueue：基于数组的有界阻塞队列。

  > 队列按FIFO原则对元素进行排序，队列头部是在队列中存活时间最长的元素，队尾则是存在时间最短的元素。新元素插入到队列的尾部，队列获取操作则是从队列头部开始获得元素。
  > 这是一个典型的“有界缓存区”，固定大小的数组在其中保持生产者插入的元素和使用者提取的元素。一旦创建了这样的缓存区，就不能再增加其容量。试图向已满队列中放入元素会导致操作受阻塞；试图从空队列中提取元素将导致类似阻塞。ArrayBlockingQueue构造方法可通过设置fairness参数来选择是否采用公平策略，公平性通常会降低吞吐量，但也减少了可变性和避免了“不平衡性”，可根据情况来决策。

- LinkedBlockingQueue：基于链表的无界阻塞队列。

  > 与ArrayBlockingQueue一样采用FIFO原则对元素进行排序。基于链表的队列吞吐量通常要高于基于数组的队列。

- SynchronousQueue：同步的阻塞队列。

  > 其中每个插入操作必须等待另一个线程的对应移除操作，等待过程一直处于阻塞状态，同理，每一个移除操作必须等到另一个线程的对应插入操作。SynchronousQueue没有任何容量。不能在同步队列上进行
  > peek，因为仅在试图要移除元素时，该元素才存在；除非另一个线程试图移除某个元素，否则也不能（使用任何方法）插入元素；也不能迭代队列，因为其中没有元素可用于迭代。
  > Executors.newCachedThreadPool使用了该队列。

- PriorityBlockingQueue：基于优先级的无界阻塞队列。

> 优先级队列的元素按照其自然顺序进行排序，或者根据构造队列时提供的 Comparator 进行排序，具体取决于所使用的构造方法。优先级队列不允许使用
> null 元素。依靠自然顺序的优先级队列还不允许插入不可比较的对象（这样做可能导致 ClassCastException）。虽然此队列逻辑上是无界的，但是资源被耗尽时试图执行 add 操作也将失败（导致 OutOfMemoryError）。

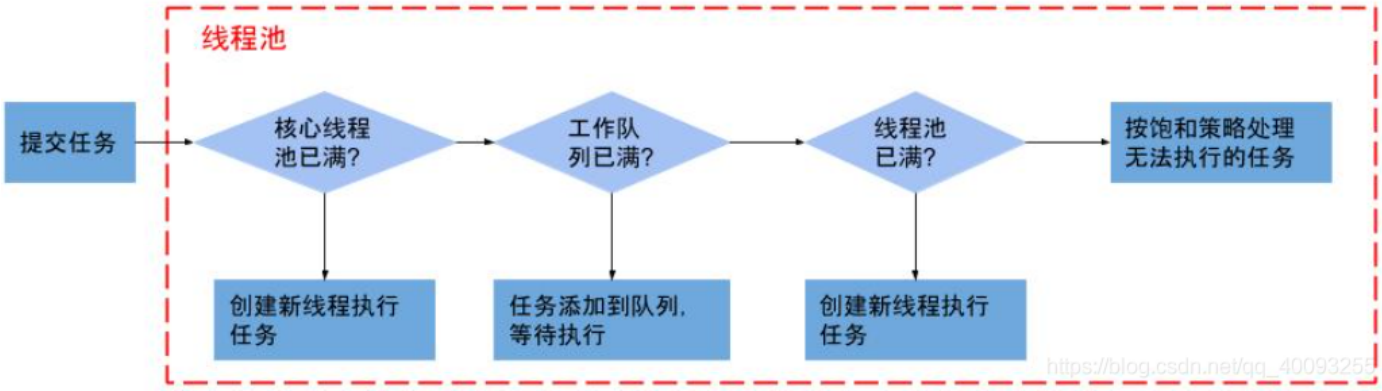
线程池关闭

调用线程池的shutdown()或shutdownNow()方法来关闭线程池

- shutdown原理：将线程池状态设置成SHUTDOWN状态，然后中断所有没有正在执行任务的线程。
- shutdownNow原理：将线程池的状态设置成STOP状态，然后中断所有任务(包括正在执行的)的线程，并返回等待执行任务的列表。

中断采用interrupt方法，所以无法响应中断的任务可能永远无法终止。但调用上述的两个关闭之一，isShutdown()方法返回值为true，当所有任务都已关闭，表示线程池关闭完成，则isTerminated()
方法返回值为true。当需要立刻中断所有的线程，不一定需要执行完任务，可直接调用shutdownNow()方法。

线程池流程:



1. 判断核心线程池是否已满，即已创建线程数是否小于corePoolSize？没满则创建一个新的工作线程来执行任务。已满则进入下个流程。
2. 判断工作队列是否已满？没满则将新提交的任务添加在工作队列，等待执行。已满则进入下个流程。
3. 判断整个线程池是否已满，即已创建线程数是否小于maximumPoolSize？没满则创建一个新的工作线程来执行任务，已满则交给饱和策略来处理这个任务。

合理配置线程池

- 需要针对具体情况而具体处理，不同的任务类别应采用不同规模的线程池，任务类别可划分为CPU密集型任务、IO密集型任务和混合型任务。
- 对于CPU密集型任务：线程池中线程个数应尽量少，不应大于CPU核心数；
- 对于IO密集型任务：由于IO操作速度远低于CPU速度，那么在运行这类任务时，CPU绝大多数时间处于空闲状态，那么线程池可以配置尽量多些的线程，以提高CPU利用率；
- 对于混合型任务：可以拆分为CPU密集型任务和IO密集型任务，当这两类任务执行时间相差无几时，通过拆分再执行的吞吐率高于串行执行的吞吐率，但若这两类任务执行时间有数据级的差距，那么没有拆分的意义。

线程池监控:

利用线程池提供的参数进行监控，参数如下：

- taskCount：线程池需要执行的任务数量。
- completedTaskCount：线程池在运行过程中已完成的任务数量，小于或等于taskCount。
- largestPoolSize：线程池曾经创建过的最大线程数量，通过这个数据可以知道线程池是否满过。如等于线程池的最大大小，则表示线程池曾经满了。
- getPoolSize：线程池的线程数量。如果线程池不销毁的话，池里的线程不会自动销毁，所以这个大小只增不减。
- getActiveCount：获取活动的线程数。

通过扩展线程池进行监控：继承线程池并重写线程池的beforeExecute()，afterExecute()和terminated()方法，可以在任务执行前、后和线程池关闭前自定义行为。如监控任务的平均执行时间，最大执行时间和最小执行时间等。

ThreadLocal对象:

线程对象提供了副本对象, 特点是, 每一个线程都独立拥有ThreadLocal对象

多线程情况下, 一般比较喜欢使用ThreadLocal, 多线程情况下, 将值保存到ThreadLocal中, 多线程之间都是各自拥有各自的值, 不会发生冲突

添加的值最后一定要移除, 否则容易出现内存溢出

set() 在自身线程中添加值

get() 获取自身线程中存储在ThreadLocal中的值

remove() 移除值

Thread类方法:

可以用Thread(String name)构造函数设置线程名

方法	描述
getName()	获取线程名
getId()	获取线程ID
getPriority()或者 Thread.currentThread().getPriority(Thread thread)	获取线程权重
getState()	获取线程状态
interrupt()	抛出一个终端信号, 线程不会立即中断
isAlive()	判断线程是否活着
stop()	进程终止, 已过时
isDaemon()	是否是守护线程
join()	其他线程不会和此线程进行资源抢占直至结束
setDaemon(Boolean flag)	flag为true则是设置守护进程
Thread.sleep()	休眠线程, sleep(0)可以用于重新分配时间片, 让线程进入阻塞状态
Thread.yield()	暗示当前线程放弃一次抢占, 当前线程不一定会放弃抢占

继承Thread实现线程和实现Runnable接口实现线程的区别:

- Thread实现的不会共享内存
- 如果使用不同的Thread对象启动同一个Runnable线程对象则会共享内存, 会引发线程安全问题

结束线程的正确方式:

Java中合理结束一个进程的执行（常用）

```
public class ThreadTest10 {
    public static void main(String[] args) {
        MyRunnable4 r = new MyRunnable4();
        Thread t = new Thread(r);
        t.setName("t");
        t.start();
        // 模拟5秒
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 终止线程
        // 想要什么时候终止t的执行, 把标记修改为false, 就结束了.
```

```
        r.run = false;
    }
}

class MyRunnable4 implements Runnable {
    // 打一个布尔标记
    boolean run = true;

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            if (run) {
                System.out.println(Thread.currentThread().getName() + "--->" + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } else {
                return;
            }
        }
    }
}
```

解决线程安全问题:

由于代码并不是一次就执行完, 例如自增转换成汇编需要三步, 在这三步中如果没有加锁, 那么其他的线程可能会干扰从而使值不合我们的预期

通过加锁:

synchronized关键字:

JDK1.0提供

非公平锁, 同步锁

不会产生死锁现象, JVM自动关闭

java提供了同步锁, synchronized, 该锁有三种使用方式:

- 同步块
  - 同步块加锁是用代码块将需要加锁的代码包含住, 需要传一个Object类型的key(钥匙)
- 方法加锁
  - 如果某个方法的所有代码都可能出现线程安全问题, 建议加方法锁, 该方法充当该锁的key
- 将静态方法加锁
  - 本质是类的字节码加锁, 类的字节码充当key

```
class Test {
    final Object key = new Object();

    void test() {
        synchronized (key) {
            //
        }
    }
}
```

加锁时范围应该尽量小

JDK5之前默认调用系统锁(重量级锁), 在之后JUC包提供了一种新锁---Lock锁

Lock锁(接口):

默认是非公平锁, 可以充当公平锁

此锁用完必须手动解开释放锁, 否则会出现死锁导致程序卡死, 解锁代码一定放在finally中

实现类:

- ReentrantLock(重入锁)

ReentrantLock:

`ReentrantLock()` 构造函数

`ReentrantLock(boolean fair)` 构造函数, fair为`true`可以改成公平锁

```
public class TestLock extends Thread {

    private int count;
    private Lock lock = new ReentrantLock();

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            try {
                // 加锁
                lock.lock();
                count++;

            } finally {
                // 一定要记得释放锁！！
                // 建议将释放锁的代码一定要放在finally中！！！
                lock.unlock();
            }
        }
        System.out.println(Thread.currentThread().getName() + ": count = " +
count);
    }
```

```
22
23      public static void main(String[] args) {
24          TestLock task = new TestLock();
25          new Thread(task).start();
26          new Thread(task).start();
27
28      }
29  }
30
```

synchronized锁升级:

JDK7之后, 三个阶段一个处理不了就到下一个阶段

偏向锁(无锁): 通过贴标签, 解决没有并发的场景 ---> 自旋锁(CSA): 通过循环等待, 无法解决ABA问题 ---> 重量级锁(系统锁)

乐观锁和悲观锁:

乐观锁: 默认不加锁, 添加一个version字段, 被修改之后版本号就往上加, 版本不匹配就不能修改, 以此保证线程安全, 效率高

悲观锁: 不管需不需要加锁都会直接加锁, 比如synchronized和Lock

单例模式:

饿汉式:

饱汉式:

饱汉式线程安全问题使用DCL(Double Check Lock)双重检查锁, 同时需要禁止指令重排序

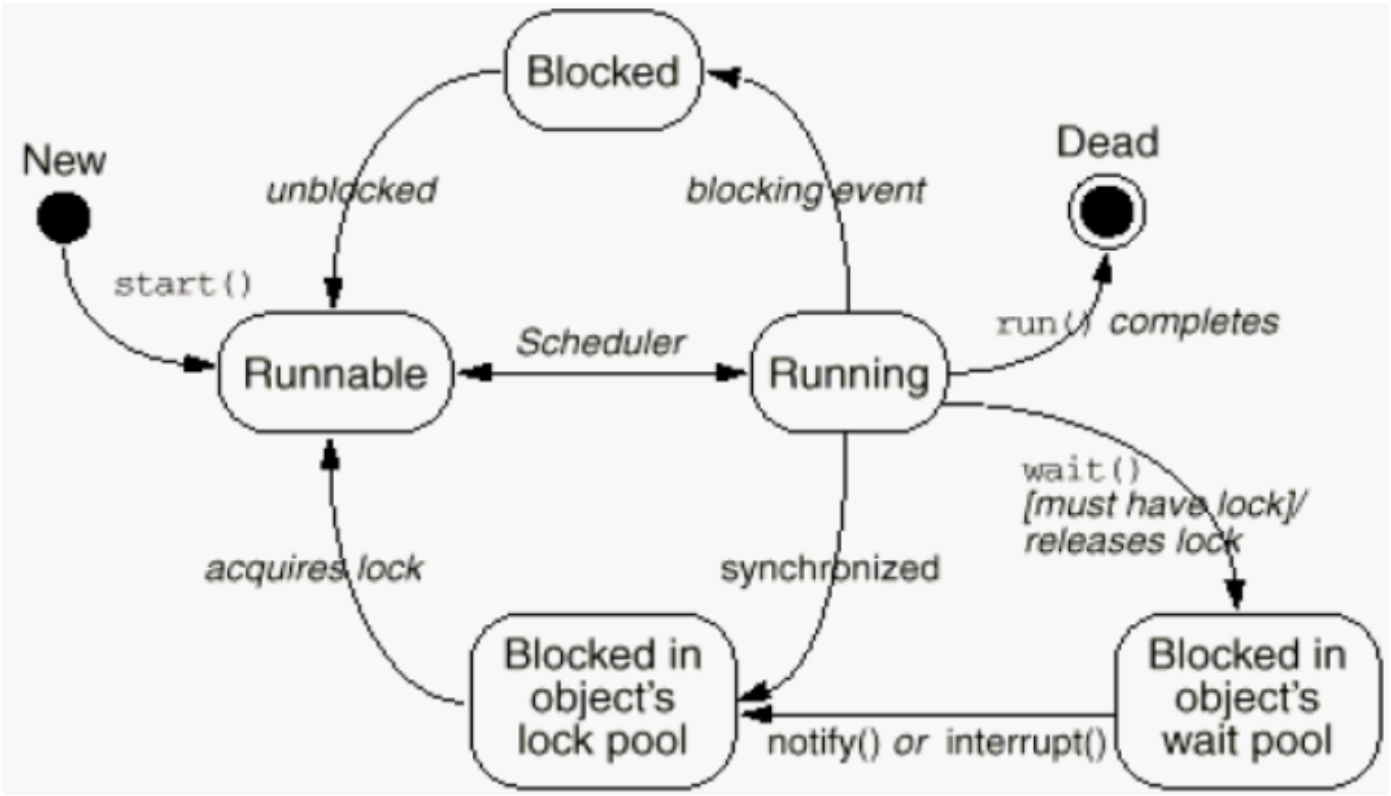
```
1   public class Singleton {
2
3       private volatile static Singleton singleton = null;
4
5       // 构造函数私有化
6       private Singleton() {
7       }
8
9       public static Sington newInstance() {
10          if (sington == null) {
11              synchronized (Sington.class) {
12                  if (sington == null) {
13                      sington = new Singleton();
14                  }
15              }
16          }
17          return singleton;
18      }
19
20  }
```

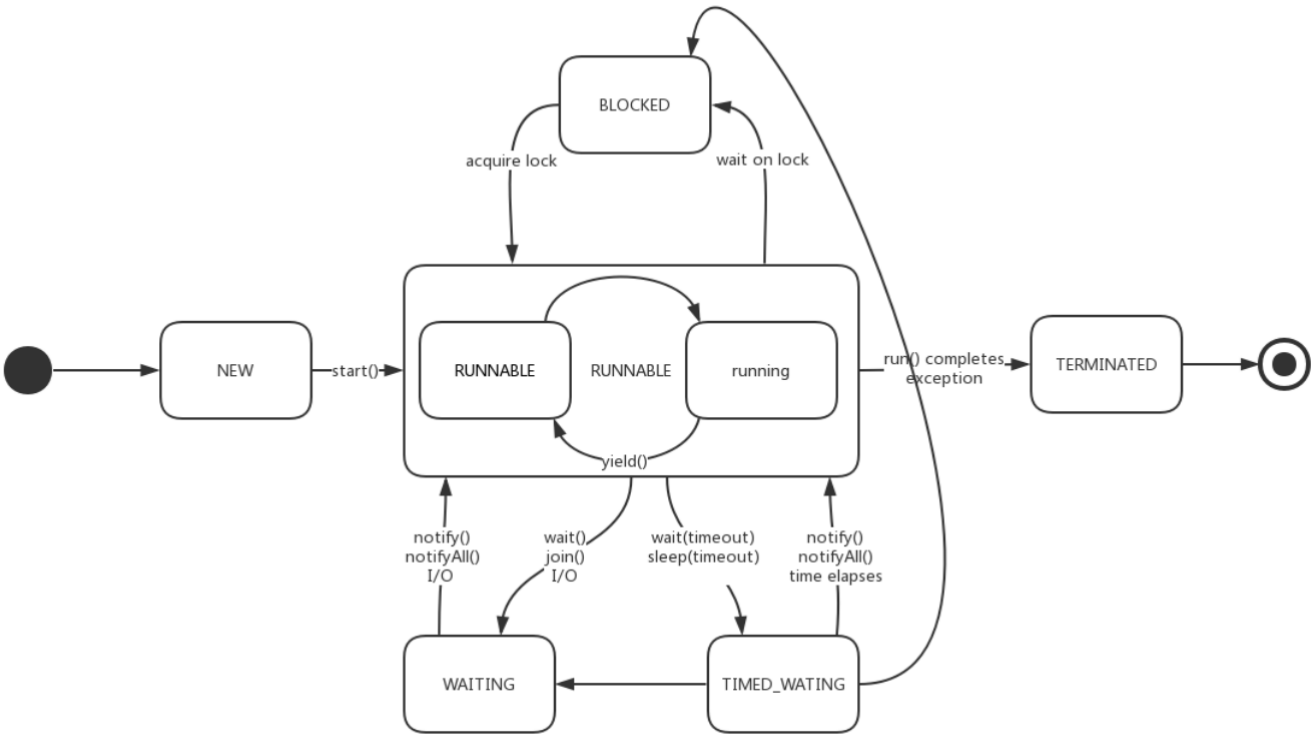
volatile关键字:

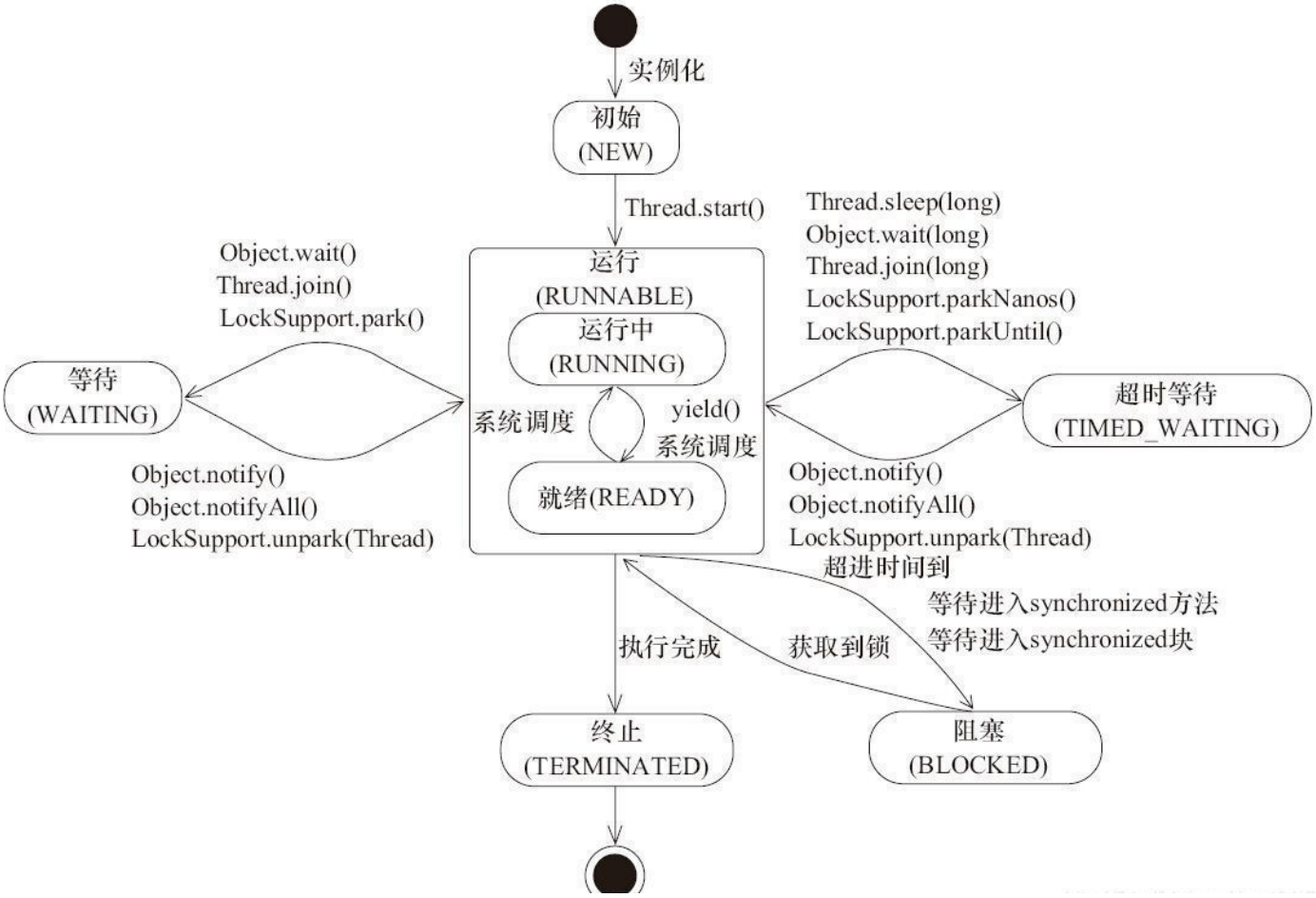
作用:

- 被volatile修饰的变量是线程可见的
- 禁止指令重排序

线程的生命周期:







- 新建状态:

  使用 **new** 关键字和 **Thread** 类或其子类建立一个线程对象后,该线程对象就处于新建状态.它保持这个状态直到程序 *
  *start()** 这个线程.

- 就绪状态:

  当线程对象调用了start()方法之后,该线程就进入就绪状态.就绪状态的线程处于就绪队列中,要等待JVM里线程调度器的调度.

- 运行状态:

  如果就绪状态的线程获取 CPU 资源,就可以执行 **run()**,此时线程便处于运行状态.处于运行状态的线程最为复杂,它可以变为阻塞状态、就绪状态和死亡状态.

- 阻塞状态:

  如果一个线程执行了sleep(睡眠)、suspend(挂起)等方法,失去所占用资源之后,该线程就从运行状态进入阻塞状态.在睡眠时间已到或获得设备资源后可以重新进入就绪状态.可以分为三种:

  - 等待阻塞:运行状态中的线程执行 wait() 方法,使线程进入到等待阻塞状态.
  - 同步阻塞:线程在获取 synchronized 同步锁失败(因为同步锁被其他线程占用).
  - 其他阻塞:通过调用线程的 sleep() 或 join() 发出了 I/O 请求时,线程就会进入到阻塞状态.当sleep() 状态超时,join()
    等待线程终止或超时,或者 I/O 处理完毕,线程重新转入就绪状态.

- 死亡状态:

  一个运行状态的线程完成任务或者其他终止条件发生时,该线程就切换到终止状态.

线程的优先级

每一个 Java 线程都有一个优先级,这样有助于操作系统确定线程的调度顺序.

Java 线程的优先级是一个整数,其取值范围是 1 (Thread.MIN_PRIORITY) - 10 (Thread.MAX_PRIORITY).

默认情况下,每一个线程都会分配一个优先级 NORM_PRIORITY(5).

具有较高优先级的线程对程序更重要,并且应该在低优先级的线程之前分配处理器资源.但是,线程优先级不能保证线程执行的顺序,而且非常依赖于平台.

死锁(deadLock):

两个线程之间, 进行资源竞争时, 彼此都拿着对方需要的资源, 但是因为线程的竞争性(请求保持), 没有释放锁就会形成相互等待释放资源的现象

此现象应该尽量避免, 但只要存在锁就不能完全避免

死锁的必要条件:

- 互斥
  - 都有相同请求
- 请求保持
- 环路等待
- 不可剥夺性

打破条件其一就可以解除死锁

线程的同步的问题:

协步同调. 使用Lock实现或者同步锁配合唤醒机制

唤醒机制:

notify()　唤醒已处于wait状态的对象

notifyAll()　唤醒所有处于wait状态的对象

wait()　进入等待状态

同步锁配合唤醒机制实现同步:

- 必须有一个公有对象
- 该共有对象就是同步锁的钥匙

生产者和消费者模式:

供需的平衡问题:

```
package demo;

import lombok.Getter;
import lombok.Setter;

```

```java
import java.util.Random;

@Getter
@Setter
class Customer extends Thread {
    private final Disk disk;

    public Customer(Disk disk) {
        this.disk = disk;
    }

    @Override
    public void run() {
        synchronized (this.disk) {
            while (true) {
                if (this.disk.isFull()) {
                    System.out.println(Thread.currentThread().getName() + "吃了" + this.disk.getFood());
                    this.disk.setFood(null);
                    this.disk.setFull(false);
                    // 唤醒生产者
                    this.disk.notify();

                    // 让生产者进入等待
                    try {
                        this.disk.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}

@Getter
@Setter
// 共有对象
class Disk {
    private boolean full;
    private String food;

    public boolean isFull() {
        return this.full;
    }
}

class Producer extends Thread {
    private final String[] foods;
```

```
    private final Disk disk;
    private final Random random;

    public Producer(Disk disk) {
        this.foods = new String[]{"apple", "eeg", "beef", "rice", "vegetables"};
        this.disk = disk;
        this.random = new Random();
    }

    @Override
    public void run() {
        makeFood();
    }

    private void makeFood() {
        synchronized (this.disk) {
            for (int i = 0; i < this.foods.length; i++) {
                if (!this.disk.isFull()) {
                    String food =
this.foods[this.random.nextInt(this.foods.length)];
                    System.out.println("创造了一个食物: " + food);
                    this.disk.setFood(food);
                    this.disk.setFull(true);

                    // 先唤醒消费者
                    this.disk.notify();

                    // 让生产者进入等待
                    try {
                        this.disk.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}

class Test {
    public static void main(String[] args) {
        Disk disk = new Disk();
        Producer producer = new Producer(disk);
        Customer customer = new Customer(disk);
        producer.start();
        customer.start();
    }
}
```

网络编程基础:

OSI七层模型(理论模型)

目前主流模型:

- TCP/IP
    - 将前三层统一为一层 --- 应用层, 由代码指定业务逻辑

同一个协议下一个进程占一个端口

netstat 查看使用中的端口

netstat -a 查看所有端口

IP地址

Socket编程:

UDP(User Data Protocol)网络协议下的socket对象:

- 无连接
- 不可靠
- 不安全

追求速度的网络协议

UDP:

- DatagramSocket类
- DatagramPacket类

DatagramSocket:

返回值类型	方法	描述
	DatagramSocket()	构造方法
	DatagramSocket(int port)	构造方法, 指定端口
void	send(DatagramPacket p) throws IOException	发送数据包
void	receive(DatagramPacket p) throws IOException	接受数据包

DatagramPacket:

返回值类型	方法	描述
	DatagramPacket(byte buf[], int length)	构造函数
	DatagramPacket(byte buf[], int offset, int length)	构造函数
	DatagramPacket(byte buf[], int offset, int length, InetAddress address, int port)	构造函数
	DatagramPacket(byte buf[], int offset, int length, SocketAddress address)	构造函数
	DatagramPacket(byte buf[], int length, InetAddress address, int port)	构造函数
	DatagramPacket(byte buf[], int length, SocketAddress address)	构造函数

数据需要转换成一个字节数组再进行发送

发送数据:

```
class Test {
    public static void main(String[] args) {
        try (DatagramSocket datagramSocket = new DatagramSocket(8080)) {
            String msg = "你好啊, socket对象";
            byte[] bytes = msg.getBytes("GBK");

            // 数据根对象
            DatagramPacket packet = new DatagramPacket(bytes, 0, bytes.length, new
InetSocketAddress("192.168.10.30", 10025));

            // 发送数据
            datagramSocket.send(packet);
            System.out.println("数据发送完成");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

接收数据:

```
class Test {
    public static void main(String[] args) {
        try (DatagramSocket socket = new DatagramSocket(8080)) {
            byte[] buf = new byte[1024];
```

```
            DatagramPacket packet = new DatagramPacket(buf, 0, buf.length, new InetSocketAddress(8080));

            System.out.println("正在等待客户端发送数据...");
            // 接受由其他UDP传输过来的数据
            socket.receive(packet);
            System.out.println("接收到数据了");
            String data = new String(packet.getData(), 0, packet.getLength(), "GBK");
            System.out.println(packet.getAddress());
            System.out.println(packet.getAddress().getHostAddress());
            System.out.println(packet.getAddress().getHostName());
            System.out.println(packet.getPort());
            System.out.println();
            System.out.println(data);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

多线程实现同时收发数据:

```
public class TestUDPSever02 {

    public static final String DES_IP = "127.0.0.1";
    public static final int DES_PORT = 9999;

    public static void main(String[] args) {
        DatagramSocket socket = null;
        Scanner sc = null;
        try {
            socket = new DatagramSocket(8888);

            // 启动一个接收到线程,来进行数据接受
            UDPReceiveMsg02 udpReceiveMsg = new UDPReceiveMsg02(socket);
            udpReceiveMsg.start();

            sc = new Scanner(System.in);
            while (true) {
                System.out.print(">");
                String msg = sc.nextLine();
                byte[] buf = msg.getBytes();
                InetSocketAddress address = new InetSocketAddress(DES_IP, DES_PORT);
                DatagramPacket pd = new DatagramPacket(buf, 0, buf.length, address);
                socket.send(pd);
            }
```

```
        } catch (SocketException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            socket.close();
        }

    }
}

class UDPReceiveMsg02 extends Thread {
    private DatagramSocket socket;

    public UDPReceiveMsg02(DatagramSocket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            receiveMsg();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void receiveMsg() throws IOException {
        while (true) {
            byte[] buf = new byte[1024];
            DatagramPacket pd = new DatagramPacket(buf, 0, buf.length);
            // 等待接受数据
            socket.receive(pd);
            // 将数据转换出来
            byte[] msg = pd.getData();
            String sendIP = pd.getAddress().getHostAddress();
            int sendPort = pd.getPort();
            System.out.println("接受到" + sendIP + ": " + sendPort + "发送过来的数据,数据是: " + new String(msg));
        }
    }
}

```

TCP:

- 安全
- 可靠
- 有连接的
- 点对点
- 长连接

连接需要经历三次握手, 断开需要四次挥手

服务器:

```
public class TestSockeServer {
    public static void main(String[] args) {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(8080);

            while (true) {
                System.out.println("服务器开始运行了,等待客户端连接上来");
                Socket socket = serverSocket.accept();
                System.out.println(socket.getInetAddress().getHostName());
                System.out.println(socket.getLocalAddress());
                System.out.println(socket.getLocalPort());

                // 推送一条消息过去
                String msg = "欢迎您进入我们的QQ";
                OutputStream os = socket.getOutputStream();
                PrintWriter out = new PrintWriter(os, true);
                out.println(msg);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                serverSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

客户端:

```
public class TestSocket {

    public static final String DES_IP = "127.0.0.1";

```

```
    public static final int DES_PORT = 8080;

    public static void main(String[] args) {
        Socket socket = new Socket();
        try {
            socket.connect(new InetSocketAddress(DES_IP, DES_PORT));
            System.out.println("成功连接到服务器");

            InputStream is = socket.getInputStream();
            String msg = null;
            BufferedReader br = new BufferedReader(new InputStreamReader(is));
            while ((msg = br.readLine()) != null) {
                System.out.println("接受到服务器端传递过来的数据,数据是: " + msg);
            }

            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            out.println("我是一个客户端,我上来了,服务器,你好着吗? ");
            out.println("第二条数据");

        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

正则表达式:

Regular Expression / regex / regexp / RE

Java中支持正在的正则的方法:

- split
- replaceAll
- replaceFirst
- matches
- .....

Java专门用于正则表达式的包:

> java.util.regex:
>
> > Pattern
> >
> > Matcher

需要使用一个 \ 将 \ 转义

元字符:

一次匹配一个位

正则	功能
.	匹配除换行符以外的所有符号
\d	匹配所有的数字
\w	匹配所有除$以外的有效符号
\s	匹配所有空白位(空格和\t)
[]	匹配中括号中的字符, 匹配一个位. 其中可以以[0-9]这种形式表示0到9十个数, 根据ascii码值
^...	以...开头
...$	以...结尾
^[]	以[]中的某一个字符开头
|	用或符号表示选择其中一个匹配

反义符:

正则	功能
\D	匹配非数字
\W	匹配特殊符号
\S	匹配非空白位
[^]	不以[]中的某一个字符开头

多位匹配:

正则	功能
*	匹配0或多个字符
+	匹配1或多个字符
?	匹配0或1个字符
{m}	匹配m个字符
{m,}	匹配至少m个字符
{m, n}	匹配m~n个字符

分组:

正则中的二次筛选

将匹配好的正则表达式, 进行括号的筛选,

正则	功能
()	分组, 用Pattern和Matcher将括号里的内容进行分组, 可以返回匹配到的内容中的括号中匹配中的内容

匹配邮箱:

```
"12584wdw@gmail.com"

\\w{4,20}@\\w*\\.com|cn|org
```

Pattern类和Matcher类:

Pattern:

方法	说明
Pattern Pattern.compile()	编译正则表达式, 返回一个Pattern对象
Matcher Pattern.matcher()	返回一个Matcher对象

Matcher 类的方法

索引方法

索引方法提供了有用的索引值，精确表明输入字符串中在哪能找到匹配：

方法	说明
int start()	返回以前匹配的初始索引。
int start(int group)	返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引
public int end()	返回最后匹配字符之后的偏移量。
public int end(int group)	返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量。

查找方法

查找方法用来检查输入字符串并返回一个布尔值，表示是否找到该模式：

方法	说明
boolean lookingAt()	尝试将从区域开头开始的输入序列与该模式匹配
boolean find()	尝试查找与该模式匹配的输入序列的下一个子序列。
boolean find(int start)	重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列。
boolean matches()	尝试将整个区域与模式匹配。

替换方法

替换方法是替换输入字符串里文本的方法：

方法	说明
Matcher appendReplacement(StringBuffer sb, String replacement)	实现非终端添加和替换步骤。
StringBuffer appendTail(StringBuffer sb)	实现终端添加和替换步骤。
String replaceAll(String replacement)	替换模式与给定替换字符串相匹配的输入序列的每个子序列。
String replaceFirst(String replacement)	替换模式与给定替换字符串匹配的输入序列的第一个子序列。
static String quoteReplacement(String s)	返回指定字符串的字面替换字符串。这个方法返回一个字符串，就像传递给Matcher类的appendReplacement 方法一个字面字符串一样工作。

贪婪模式和懒惰模式:

贪婪模式: 尽可能多的匹配能够匹配数据

懒惰模式(非贪婪模式): 会尽可能少的匹配数据

绝大多数编程默认都是采用贪婪模式, 包括Java, Python

将贪婪模式转换为懒惰模式只需要在多位匹配后面加上一个"?"

匹配网址:

```
class Test {
    void test() {
        String s = "<a class=\"img-spacing\" href='https://xueshu.baidu.com/' target='_blank' name='tj_xueshu'>";
        Pattern pattern = Pattern.compile("a.*href='(.*?)'.*");
        Matcher matcher = pattern.matcher(s);
        while (matcher.find()) {
            System.out.println(matcher.group(1));
        }
    }
}
```

匹配标签内容:

```
class Test {
    void test() {
        String s = """<ul>
                < li > 这个是列表1 </li >
<li > 这个是列表2 </li >
<li > 这个是列表3 </li >
<li > 这个是列表4 </li >
<li > 这个是列表5 </li >
<li > 这个是列表6 </li >
<li > 这个是列表7 </li >
<li > 这个是列表8 </li >
</ul > """;
        s.replaceAll("</?\\w+>", "");
    }

    void test2() {
        String s = "<ul>\n" +
                "    <li>这个是列表1</li>\n" +
                "    <li>这个是列表2</li>\n" +
                "    <li>这个是列表3</li>\n" +
                "    <li>这个是列表4</li>\n" +
                "    <li>这个是列表5</li>\n" +
                "    <li>这个是列表6</li>\n" +
                "    <li>这个是列表7</li>\n" +
                "    <li>这个是列表8</li>\n" +
                "</ul>";
        Pattern pattern = Pattern.compile("<\\w+>(.*?)</\\w+>");
        Matcher matcher = pattern.matcher(s);
        while (matcher.find()) {
            System.out.print(matcher.group(1));
        }
    }
}
```

正则表达式语法表:

字符	说明
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如， n匹配字符 n。\n 匹配换行符。序列 \\ 匹配 \ ，\(匹配 (。
^	匹配输入字符串开始的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，^ 还会与"\n"或"\r"之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，$ 还会与"\n"或"\r"之前的位置匹配。
*	零次或多次匹配前面的字符或子表达式。例如，zo* 匹配"z"和"zoo"。* 等效于 {0,}。
+	一次或多次匹配前面的字符或子表达式。例如，"zo+"与"zo"和"zoo"匹配，但与"z"不匹配。+ 等效于 {1,}。
?	零次或一次匹配前面的字符或子表达式。例如，"do(es)?"匹配"do"或"does"中的"do"。? 等效于 {0,1}。
{*n*}	*n* 是非负整数。正好匹配 *n* 次。例如，"o{2}"与"Bob"中的"o"不匹配，但与"food"中的两个"o"匹配。
{*n*,}	*n* 是非负整数。至少匹配 *n* 次。例如，"o{2,}"不匹配"Bob"中的"o"，而匹配"foooood"中的所有 o。"o{1,}"等效于"o+"。"o{0,}"等效于"o*"。
{*n*,*m*}	*m* 和 *n* 是非负整数，其中 *n* <= *m*。匹配至少 *n* 次，至多 *m* 次。例如，"o{1,3}"匹配"fooooood"中的头三个 o。'o{0,1}' 等效于 'o?'。注意：您不能将空格插入逗号和数字之间。
?	当此字符紧随任何其他限定符（、+、*?*、*{n}*、*{n,}*、*{n,m*}）之后时，匹配模式是"非贪心的"。"非贪心的"模式匹配搜索到的、尽可能短的字符串，而默认的"贪心的"模式匹配搜索到的、尽可能长的字符串。例如，在字符串"oooo"中，"o+?"只匹配单个"o"，而"o+"匹配所有"o"。
.	匹配除"\r\n"之外的任何单个字符。若要匹配包括"\r\n"在内的任意字符，请使用诸如"[\s\S]"之类的模式。
(*pattern*)	匹配 *pattern* 并捕获该匹配的子表达式。可以使用 **0 . . .9** 属性从结果"匹配"集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用"\(" "\)"。
(?:*pattern*)	匹配 *pattern* 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用"or"字符 (\
(?=*pattern)*	执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 *pattern* 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?=95\
(?!*pattern*)	执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 *pattern* 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95\
x\	*y*

[*xyz*]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。
[^*xyz*]	反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中"p"，"l"，"i"，"n"。
[*a-z*]	字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。
[^*a-z*]	反向范围字符。匹配不在指定的范围内的任何字符。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。
\b	匹配一个字边界，即字与空格间的位置。例如，"er\b"匹配"never"中的"er"，但不匹配"verb"中的"er"。
\B	非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。
\c*x*	匹配 *x* 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。*x* 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。
\d	数字字符匹配。等效于 [0-9]。
\D	非数字字符匹配。等效于 [^0-9]。
\f	换页符匹配。等效于 \x0c 和 \cL。
\n	换行符匹配。等效于 \x0a 和 \cJ。
\r	匹配一个回车符。等效于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等。与 [\f\n\r\t\v] 等效。
\S	匹配任何非空白字符。与 [^ \f\n\r\t\v] 等效。
\t	制表符匹配。与 \x09 和 \cI 等效。
\v	垂直制表符匹配。与 \x0b 和 \cK 等效。
\w	匹配任何字类字符，包括下划线。与"[A-Za-z0-9_]"等效。
\W	与任何非单词字符匹配。与"[^A-Za-z0-9_]"等效。
\x*n*	匹配 *n*，此处的 *n* 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，"\x41"匹配"A"。"\x041"与"\x04"&"1"等效。允许在正则表达式中使用 ASCII 代码。
num	匹配 *num*，此处的 *num* 是一个正整数。到捕获匹配的反向引用。例如，"(.)\1"匹配两个连续的相同字符。
n	标识一个八进制转义码或反向引用。如果 *n* 前面至少有 *n* 个捕获子表达式，那么 *n* 是反向引用。否则，如果 *n* 是八进制数 (0-7)，那么 *n* 是八进制转义码。
nm	标识一个八进制转义码或反向引用。如果 *nm* 前面至少有 *nm* 个捕获子表达式，那么 *nm* 是反向引用。如果 *nm* 前面至少有 *n* 个捕获，则 *n* 是反向引用，后面跟有字符 *m*。如果两种前面的情况都不存在，则 *nm* 匹配八进制值 *nm*，其中 *n* 和 *m* 是八进制数字 (0-7)。
\nml	当 *n* 是八进制数 (0-3)，*m* 和 *l* 是八进制数 (0-7) 时，匹配八进制转义码 *nml*。
	匹配 *n*，其中 *n* 是以四位十六进制数表示的 Unicode 字符。例如，\u00A9 匹配版权符号

\u*n*	(©)。

反射(reflect):

是一门Java提供的专门技术, 这门技术让Java成为一门准动态语言

存在性能问题, 会占用额外内存, 速度也会较慢

一个类在内存中只有一个Class对象

一个类被加载后, 类的整个结构都会封装在Class对象中

Class类:

class的类, 所有类都指向了Class类

包含了类的所有信息

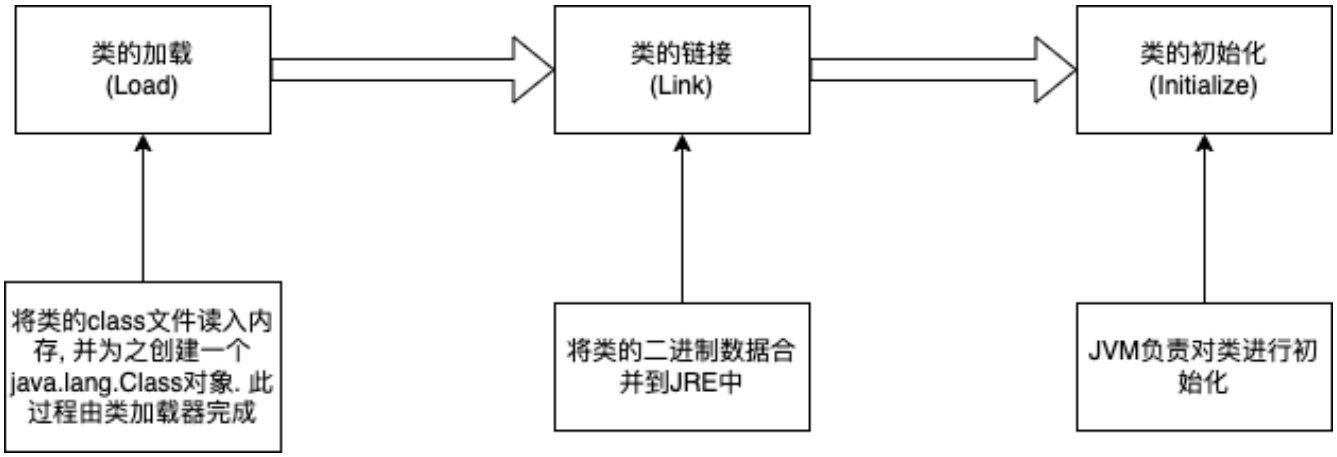
- Class只能由系统创建
- 一个加载的类在JVM中只会有一个Class实例
- 一个Class对象对应的是一个加载到JVM中的一个.class文件
- 每个类的实例都会记得自己是由哪个Class实例所生成
- 通过Class可以完整的得到一个类中的所有被加载的结构
- Class类是Reflection的根源, 针对任何你想动态加载 \ 运行的类, 唯有获得相应的Class对象

所有类都有Class对象, void/枚举 也有

只要类型和维度一样, 不管对象里面的值一不一样, Class对象都是同一个对象

内存分析:

类加载过程:



- 加载: 将class文件字节码内容加载在内存中, 并将这些静态数据转换为方法区的运行时数据结构, 然后生成一个代表这个类的java.lang.Class对象
- 链接: 将Java类的二进制代码合并到JVM的运行状态之中的过程
    - 验证: 确保加载的类信息符合JVM规范, 没有安全方面的问题
    - 准备: 正式为类变量(static)分配内存并设置类变量默认初始值的阶段, 这些内存都将在方法区中进行分配
    - 解析: 虚拟机常量池内的符合引用(常量名)替换为直接引用(地址)的过程

- 初始化:
  - 执行类构造器`<clinit>()`方法的过程. 类构造器`<clinit>()`
    方法是由编译器自动收集类中所有类变量的赋值动作和静态代码块中的语句产生的. (类构造器是构造类)

获取字节码对象:

通过类名称获取:

类名.class

```
class Test {
    void test() {
        try {
            Class<User> userClass = User.class;
            User user = userClass.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

通过对象获取:

对象.class

```
class Test {
    void test() {
        User user = new User();
        Class<User> userClass = (Class<User>) user.getClass();
    }
}
```

通过Class的方法:

Class.forName(类路径)

```
class Test {
    void test() throws ClassNotFoundException, NoSuchMethodException,
InvocationTargetException, InstantiationException, IllegalAccessException {
        Class<?> userClass = Class.forName("practise.User");
        Constructor<?> constructor = userClass.getDeclaredConstructor();
        User user = (User) constructor.newInstance();
    }
}
```

字节码对象的方法:

有Declared可以拿到所有不被权限限制

没有Declared只可以拿到可以访问的(包括父类)

获取构造器:

方法	描述
Constructor<?>[] getConstructors()	以列表的形式返回所有的可以访问的(包括父类)构造函数
Constructor<?>[] getDeclaredConstructors()	以列表的形式返回所有的自身声明的构造函数, 不受权限限制
Constructor<T> getConstructor(Class<?>... parameterTypes)	获取特定的可以访问的(包括父类)构造函数, 参数为对应构造函数的参数的字节码
Constructor<T> getDeclaredConstructor()	获得自身声明的构造器, 不受权限限制

获取方法:

方法	描述
Method getMethod(String name, Class<?>... parameterTypes)	获取指定方法
Method[] getMethods() throws SecurityException	获取所有可以访问的(包括父类)方法
Method getDeclaredMethod()	
Method[] getDeclaredMethods()	

获取属性:

方法	描述
Field getField(String name) throws NoSuchFieldException, SecurityException	获取属性, 形参为属性名
Field[] getFields() throws SecurityException	获取属性
Field getDeclaredField(String name) throws NoSuchFieldException, SecurityException	获取属性
Field[] getDeclaredFields() throws SecurityException	获取属性

Constructor方法:

方法	描述
String getName()	获取构造函数名
native int getModifiers()	获取构造函数访问修饰符. public-0x00000001, private-0x00000002, protected-0x00000004, static-0x00000008, final-0x00000010, synchronized-0x00000020, volatile-0x00000040, transient-0x00000080, native-0x00000100, interface-0x00000200, abstract-0x00000400
int getParameterCount()	获取构造函数参数个数
Class<?>[] getParameterTypes()	获取构造函数参数类型

Method方法:

方法	描述
void setAccessible(boolean flag)	true打破封装
Object invoke(Object obj, Object... args)	执行方法, obj是要执行此方法的类, args是参数

通过反射实现对象Clone:

```
class TestReflect {

    @Test
    void test01() {
        User user = new User("Erzbir, 18, 1");
        User user1 = new User();
        copyObject(user, user1);
        System.out.println(user1);
    }

    // 通过反射复制(拷贝)对象
    private void copyObject(User user, User user1) {
        try {
            Class<?> clazz = user.getClass();
            Field[] fields = clazz.getDeclaredFields();
            for (Field field : fields) {
                field.setAccessible(true);
                Object o = field.get(user);
                field.set(user1, o);
            }
        } catch (IllegalAccessException e) {
```

```
            e.printStackTrace();
        }
    }
}

```

注解(Annotation):

参考: https://www.itzhai.com/java-based-notebook-annotation-annotation-introduction-and-use-custom-annotations.html

元注解:

@Retention @Target @Document @Inherited

- @Retention: 定义注解的保留策略

  > `@Retention(RetentionPolicy.SOURCE)` //注解仅存在于源码中,在class字节码文件中不包含, 用于编译时, 典型为`@Gette`
  > r和`@Setter`和`@Override`
  >
  > `@Retention(RetentionPolicy.CLASS)` // 默认的保留策略,注解会在class字节码文件中存在,但运行时无法获得
  >
  > `@Retention(RetentionPolicy.RUNTIME)` // 注解会在class字节码文件中存在,在运行时可以通过反射获取到
  >
  > 生命周期大小排序为 SOURCE < CLASS < RUNTIME,前者能使用的地方后者一定也能使用.如果需要在运行时去动态获取注解信息,那只能用
  > RUNTIME 注解；如果要在编译时进行一些预处理操作,比如生成一些辅助代码(如 ButterKnife),就用 CLASS
  > 注解；如果只是做一些检查性的操作,比如 `@Override` 和 `@SuppressWarnings`,则可选用 SOURCE 注解.

- `@Target`: 定义注解的使用范围

  > `@Target` 注解用来指定一个注解的使用范围,即被 @Target 修饰的注解可以用在什么地方. `@Target` 注解有一个成员变量(
  > value)用来设置适用目标, value 是`java.lang.annotation.ElementType` 枚举类型的数组,下表为 `ElementType` 常用的枚举常量.

名称	说明
CONSTRUCTOR	用于构造方法
FIELD	用于成员变量(包括枚举常量)
LOCAL_VARIABLE	用于局部变量
METHOD	用于方法
PACKAGE	用于包
PARAMETER	用于类型参数(JDK 1.8新增)
TYPE	用于类、接口(包括注解类型)或 enum 声明

- @Document: 用于生成JavaDoc

> `@Documented` 是一个标记注解,没有成员变量.用 `@Documented` 注解修饰的注解类会被 JavaDoc 工具提取成文档.默认情况下,JavaDoc
> 是不包括注解的,但如果声明注解时指定了 `@Documented`, 就会被 JavaDoc 之类的工具处理,所以注解类型信息就会被包括在生成的帮助文档中.

- @Inherited: 指定注解可被继承

> `@Inherited` 是一个标记注解,用来指定该注解可以被继承.使用 `@Inherited` 注解的 Class 类,表示这个注解可以被用于该
> Class 类的子类.就是说如果某个类使用了被 `@Inherited` 修饰的注解,则其子类将自动具有该注解.

Annotation的工作原理:

JDK5.0中提供了注解的功能, 允许开发者定义和使用自己的注解类型. 该功能由一个定义注解类型的语法和描述一个注解声明的语法,
读取注解的API ,一个使用注解修饰的class文件和一个注解处理工具组成.

Annotation并不直接影响代码的语义, 但是他可以被看做是程序的工具或者类库. 它会反过来对正在运行的程序语义有所影响.

Annotation可以冲源文件、class文件或者在运行时通过反射机制多种方式被读取.

@Override注解:

> java.lang
> 注释类型 Override
> @Target(value=METHOD)
> @Retention(value=SOURCE)
> public @interface ***Override***

表示一个方法声明打算重写超类中的另一个方法声明.如果方法利用此注释类型进行注解但没有重写超类方法, 则编译器会生成一条错误消息.

`@Override`注解表示子类要重写父类的对应方法.

Override是一个Marker annotation,用于标识的Annotation, Annotation名称本身表示了要给工具程序的信息.

下面是一个使用@Override注解的例子:

```
class A {
    private String id;

    A(String id) {
        this.id = id;
    }

    @Override
    public String toString() {
        return id;
    }
}
```

@Deprecated注解:

> java.lang
> 注释类型 Deprecated
> @Documented
> @Retention(value=RUNTIME)
> public @interface ***Deprecated***

用 `@Deprecated` 注释的程序元素,不鼓励程序员使用这样的元素,通常是因为它很危险或存在更好的选择.在使用不被赞成的程序元素或在不被赞成的代码中执行重写时,编译器会发出警告.

`@Deprecated`注解表示方法是不被建议使用的.

Deprecated是一个Marker annotation.

下面是一个使用`@Deprecated`注解的例子:

```
class A {
    private String id;

    A(String id) {
        this.id = id;
    }

    @Deprecated
    public void execute() {
        System.out.println(id);
    }

    public static void main(String[] args) {
        A a = new A("a123");
        a.execute();
    }
```

```
}
```

@SuppressWarnings注解：

> java.lang
> 注释类型 SuppressWarnings
> @Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL_VARIABLE})
> @Retention(value=SOURCE)
> public @interface ***SuppressWarnings***

指示应该在注释元素(以及包含在该注释元素中的所有程序元素)
中取消显示指定的编译器警告.注意,在给定元素中取消显示的警告集是所有包含元素中取消显示的警告的超集.例如,如果注释一个类来取消显示某个警告,同时注释一个方法来取消显示另一个警告,那么将在此方法中同时取消显示这两个警告.
根据风格不同,程序员应该始终在最里层的嵌套元素上使用此注释,在那里使用才有效.如果要在特定的方法中取消显示某个警告,则应该注释该方法而不是注释它的类.

`@SuppressWarnings`注解表示抑制警告.

下面是一个使用`@SuppressWarnings`注解的例子：

```
class Test {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        List list = new ArrayList();
        list.add("abc");
    }
}
```

自定义注解:

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD) // ElementType.TYPE标明此注释可以标记的类型(类, 接口, 枚举, 方法), 可以以数组的方法添加多个
public @interface MyAnnotation {
    String value() default "Erzbir";

    int version();
}
```

使用自定义注解:

```
public class Test {

    @MyAnnotion(version = 1)
    public void test1() {

    }
}
```

注解中每一个成员都默认是public, 且格式必须是: 数据类型 成员名称() default 值

如果不指定默认值, 使用时必须给一个默认值

通过反射拿到注解内容:

```
class Test {
    @MyAnnotation
    void test() {
        Annotation[] annotations =
this.getClass().getAnnotation(MyAnnotation.class);
    }
}
```

使用@interface自定义注解时,自动继承了java.lang.annotation.Annotation接口,由编译程序自动完成其他细节.在定义注解时,不能继承其他的注解或接口.

多变量使用枚举:

```
public @interface MyAnnotation {

    String value1() default "abc";

    MyEnum value2() default MyEnum.Sunny;
}

enum MyEnum {
    Sunny, Rainy
}
```

数组变量:

```
public @interface MyAnnotation {

    String[] value1() default "abc";
}
```

设置注解的作用范围：

> @Documented
> @Retention(value=RetentionPolicy.CLASS)
> @Target(value=ElementType.ANNOTATION_TYPE)
> public @interface ***Retention***

指示注释类型的注释要保留多久.如果注释类型声明中不存在 `@Retention` 注释, 则保留策略默认为 `RetentionPolicy.CLASS`.
只有元注释类型直接用于注释时, `@Target` 元注释才有效. 如果元注释类型用作另一种注释类型的成员,则无效.

注解保留策略.此枚举类型的常量描述保留注释的不同策略.它们与 `@Retention`元注释类型一起使用,以指定保留多长的注释.可用的注释保留策略：

- ***CLASS***
  编译器将把注释记录在类文件中,但在运行时 VM 不需要保留注释.
- ***RUNTIME***
  编译器将把注释记录在类文件中,在运行时 VM 将保留注释,因此可以反射性地读取.
- ***SOURCE***
  编译器要丢弃的注释.

@Retention注解可以在定义注解时为编译程序提供注解的保留策略.

属于CLASS保留策略的注解有@SuppressWarnings,该注解信息不会存储于.class文件.

在自定义注解中的使用例子：

```
1
2  @Retention(RetentionPolicy.CLASS)
3  public @interface MyAnnotation {
4      String[] value1() default "abc";
5  }
```

使用反射读取RUNTIME保留策略的Annotation信息的例子：

> java.lang.reflect
> 接口 ***AnnotatedElement***
> 所有已知实现类：
> AccessibleObject, Class, Constructor, Field, Method, Package

表示目前正在此 VM
中运行的程序的一个已注释元素.该接口允许反射性地读取注释.由此接口中的方法返回的所有注释都是不可变并且可序列化的.调用者可以修改已赋值数组枚举成员的访问器返回的数组；这不会对其他调用者返回的数组产生任何影响.
如果此接口中的方法返回的注释(直接或间接地)包含一个已赋值的 `Class` 成员,该成员引用了一个在此 VM
中不可访问的类,则试图通过在返回的注释上调用相关的类返回的方法来读取该类,将导致一个 `TypeNotPresentException`.

> isAnnotationPresent
> boolean ***isAnnotationPresent***(Class<? extends Annotation> annotationClass)

如果指定类型的注释存在于此元素上,则返回 true,否则返回 false.此方法主要是为了便于访问标记注释而设计的.

参数:

- `annotationClass` - 对应于注释类型的 Class 对象

返回:

- 如果指定注释类型的注释存在于此对象上,则返回 `true`,否则返回 `false`

抛出:

- `NullPointerException` - 如果给定的注释类为 null

> getAnnotation
> T ***getAnnotation***(Class annotationClass)

如果存在该元素的指定类型的注释,则返回这些注释,否则返回 `null`.

参数:

- annotationClass - 对应于注释类型的 `Class` 对象

返回:

- 如果该元素的指定注释类型的注释存在于此对象上,则返回这些注释,否则返回 `null`

抛出:

- `NullPointerException` - 如果给定的注释类为 `null`

> getAnnotations
> Annotation[] ***getAnnotations***()

返回此元素上存在的所有注释.(如果此元素没有注释,则返回长度为零的数组.)该方法的调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响.
返回：此元素上存在的所有注释

> getDeclaredAnnotations
> Annotation[] ***getDeclaredAnnotations***()

返回直接存在于此元素上的所有注释.与此接口中的其他方法不同,该方法将忽略继承的注释.(
如果没有注释直接存在于此元素上,则返回长度为零的一个数组.)该方法的调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响.

返回:

- 直接存在于此元素上的所有注释

使用反射读取RUNTIME保留策略的Annotation信息的例子:

自定义注解:

```
1
2  @Retention(RetentionPolicy.RUNTIME)
3  public @interface MyAnnotation {
4
5      String[] value1() default "abc";
6  }
```

读取注解中的信息：

```
1  class Test {
2      public static void main(String[] args) throws SecurityException,
   NoSuchMethodException, IllegalArgumentException, IllegalAccessException,
   InvocationTargetException {
3          AnnotationTest2 annotationTest2 = new AnnotationTest2();
4          //获取AnnotationTest2的Class实例
5          Class<AnnotationTest2> c = AnnotationTest2.class;
6  //获取需要处理的方法Method实例
7          Method method = c.getMethod("execute", new Class[]{});
8          //判断该方法是否包含MyAnnotation注解
9          if (method.isAnnotationPresent(MyAnnotation.class)) {
10             //获取该方法的MyAnnotation注解实例
11             MyAnnotation myAnnotation = method.getAnnotation(MyAnnotation.class);
12             //执行该方法
13             method.invoke(annotationTest2, new Object[]{});
14             //获取myAnnotation
15             String[] value1 = myAnnotation.value1();
16             System.out.println(value1[0]);
17         }
18         //获取方法上的所有注解
19         Annotation[] annotations = method.getAnnotations();
20         for (Annotation annotation : annotations) {
21             System.out.println(annotation);
22         }
23     }
24 }
```

限定注解的使用：

限定注解使用

@Target

> @Documented
> @Retention(value=RUNTIME)
> @Target(value=ANNOTATION_TYPE)
> public @interface ***Target***

指示注释类型所适用的程序元素的种类.如果注释类型声明中不存在 Target 元注释,则声明的类型可以用在任一程序元素上.如果存在这样的元注释,则编译器强制实施指定的使用限制.
例如,此元注释指示该声明类型是其自身,即元注释类型.它只能用在注释类型声明上:

```
@Target(ElementType.ANNOTATION_TYPE)
public @interface MetaAnnotationType {
    //...
}
```

指定该注释可注释多种类型:

```
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface MemberType {
    //...
}
```

程序元素类型.此枚举类型的常量提供了 Java 程序中声明的元素的简单分类. 这些常量与 Target 元注释类型一起使用,以指定在什么情况下使用注释类型是合法的.

ANNOTATION_TYPE
注释类型声明
CONSTRUCTOR
构造方法声明
FIELD
字段声明(包括枚举常量)
LOCAL_VARIABLE
局部变量声明
METHOD
方法声明
PACKAGE
包声明
PARAMETER
参数声明
TYPE
类、接口(包括注释类型)或枚举声明

注解的使用限定的例子:

```
@Target(ElementType.METHOD)
public @interface MyAnnotation {
    String[] value1() default "abc";
}
```

在帮助文档中加入注解：

要想在制作JavaDoc文件的同时将注解信息加入到API文件中,可以使用`java.lang.annotation.Documented`.

在自定义注解中声明构建注解文档：

```
1
2  @Documented
3  public @interface MyAnnotation {
4
5      String[] value1() default "abc";
6  }
```

在注解中使用继承：

默认情况下注解并不会被继承到子类中,可以在自定义注解时加上`java.lang.annotation.Inherited`注解声明使用继承.

@Inherited

> @Documented
> @Retention(value=RUNTIME)
> @Target(value=ANNOTATION_TYPE)
> public @interface ***Inherited***

指示注释类型被自动继承.

如果在注释类型声明中存在 Inherited
元注释,并且用户在某一类声明中查询该注释类型,同时该类声明中没有此类型的注释,则将在该类的超类中自动查询该注释类型.此过程会重复进行,直到找到此类型的注释或到达了该类层次结构的顶层 (
Object) 为止.如果没有超类具有该类型的注释,则查询将指示当前类没有这样的注释.

注意,如果使用注释类型注释类以外的任何事物,此元注释类型都是无效的.还要注意,此元注释仅促成从超类继承注释；对已实现接口的注释无效.

配置文件数据解析:

- xml
- json
- yaml
- properties

Properties:

解析properties文件

底层是hashtable

如果配置文件在src或者其他类路径中, 类名称.class.getClassLoader().getResourceAsStream(name)拿到类的字节码文件再拿到加载器再拿到对应的资源文件

```
class TestPro {
    void test01() {
        Properties properties = new Properties();

        try {
            properties.load(new FileInputStream("common.properties"));
            System.out.println(properties.getProperty("age"));
            System.out.println(properties.getProperty("gender"));
            System.out.println(properties.getProperty("name"));

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    void test02() {
//    this.getClass().getClassLoader().getResourceAsStream(name)
        try {
            InputStream is =
TestPro.class.getClassLoader().getResourceAsStream("common.properties");
            Properties properties = new Properties();
            properties.load(is);
            System.out.println(properties.getProperty("age"));
            System.out.println(properties.getProperty("gender"));
            System.out.println(properties.getProperty("name"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

自定义工具类:

```
public class PropertiesUtils {

    /**
     *通过(类路径下的配置文件)配置文件的名称和配置的key或者需要的key对应的值
     * @param name  配置文件的名称
     * @param key key名称
     * @return key对应的值
     */
    public static String getValueByKey(String name, String key) {
        Properties prop = load(name);
        if (prop != null) {
            return prop.getProperty(key);
        }
        throw new RuntimeException("配置的路径或者名称有误");
```

```
    }

    private static Properties load(String name) {
        Properties properties = new Properties();

        try (InputStream is = PropertiesUtils.class.getClassLoader().getResourceAsStream(name)) {
            properties.load(is);
            return properties;
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    /**
     *  通过(类路径下的配置文件)配置文件(commons.properties) 和配置的key或者需要的key对应的值
     * @param key key名称
     * @return key对应的值
     */
    public static String getValueByKey(String key) {
        Properties prop = load("commons.properties");
        if (prop != null) {
            return prop.getProperty(key);
        }
        throw new RuntimeException("配置的路径或者名称有误");
    }
}

```

JSON(JavaScript Simple Object Nation):

将对象序列化为字符串, 一般需要有特定的格式

1. 方便传输
2. 方便描述数据

格式:

单个对象:

```
{
  "name": "Erzbir",
  "age": 18,
  "id": 1
}
```

多个对象:

```
{
  "users": [
    {},
    {},
    {}
  ],
  "admins": [
    {},
    {},
    {}
  ]
}
```

将对象转换为JSON:

手动:

```
class Test {
    void test() {
        User user = new User("Erzbir", 18, 1);
        StringBuilder sb = new StringBuilder();
        sb.append(" { \"id\": ");
        sb.append(user.getId());
        sb.append(", \"name\": ");
        sb.append("\"" + user.getName() + "\"");
        sb.append(", \"age\": ");
        sb.append(user.getAge() + "}");
    }
}
```

使用工具:

- jackson
- fastjson // 阿里巴巴提供, 不建议使用, 漏洞很多
- gson // 谷歌, 轻量级

gson:

```
class Test {
    void test() {
        User user = new User("Erzbir", 18, 1);
        Gson gson = new Gson();
        StringBuilder sb = new StringBuilder();
        sb.append("{\"id\":");
        sb.append(user.getId());
        sb.append(",\"name\":");
        sb.append("\"").append(user.getName()).append("\"");
```

```
        sb.append(",\"age\":");
        sb.append(user.getAge()).append("}");
        System.out.println(sb);
        String str = gson.toJson(user); //将对象转换为JSON
        System.out.println(str);
        User user2 = gson.fromJson(String.valueOf(sb), User.class); // 将JSON转换为对象
        System.out.println(user2.getAge());
    }
}
```

XML(可扩展性标记语言):

与之对应的就是HTML(超文本标记语言)

标签不固定, 可以自定义

需要一个头, 格式如下:

```
<?xml version="1.0" encoding="UTF-8"?>
```

格式:

```
<?xml version="1.0" encoding="UTF-8"?>
<user>
    <id>1</id>
    <name>Erzbir</name>
    <age>18</age>
</user>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
    <user1>
        <id>1</id>
        <name>Erzbir</name>
        <age>18</age>
    </user1>
    <user2>
    </user2>
</users>
```

可以在标签上添加属性

```
<user id="1">
    <id>1</id>
    <name>Erzbir</name>
    <age>18</age>
</user>
```

对象和XML的转化:

手动:

```
class Test {
    void test() {
        User user = new User("Erzbir", 18, 1);
        Gson gson = new Gson();
        StringBuilder sb = new StringBuilder();
        sb.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
        sb.append("<user>");
        sb.append("<id>").append(user.getId()).append("</id>");
        sb.append("<name>").append(user.getName()).append("</name>");
        sb.append("<age>").append(user.getAge()).append("</age>");
        sb.append("</user>");
        System.out.println(sb);
    }
}
```

使用工具:

- dom4j

dom4j:

```
class Test {
    void test() {
        User user = new User("Erzbir", 18, 1);
        Document document = DocumentHelper.createDocument();
        Element root = document.addElement("user") // 创建根标签
                .addAttribute("id", String.valueOf(user.getId())); // 这里将id作为属性
        Element author1 = root.addElement("name") // 创建标签
                .addText(user.getName());
        Element author2 = root.addElement("age")
                .addText(String.valueOf(user.getAge()));
        Element author2 = root.addElement("id")
                .addText(String.valueOf(user.getId()));
        System.out.println(document.asXML());
    }
}
```

爬虫(Spiders):

URL实现:

爬取网页信息:

```
class Test {
    void test() throws IOException {
        URL url = new URL("https://www.qq.com/");
        URLConnection connection = url.openConnection();
        BufferedReader br = new BufferedReader(new
InputStreamReader(connection.getInputStream(), "GBK"));
        String msg = null;
        StringBuilder sb = new StringBuilder();
        while ((msg = br.readLine()) != null) {
            sb.append(msg);
        }
        String content = sb.toString();


        // 数据提取
        Pattern pattern = Pattern.compile("<img.*?src=\"(.*?)\".*?>");
        Matcher matcher = pattern.matcher(content);
        sb = null;
        List<String> list = new ArrayList<>();
        while (matcher.find()) {
            String s = matcher.group(1);
            if (!s.startsWith("//")) {
                s = "https:" + s;
            }
            list.add(s);
            System.out.println(s);
        }

        // 数据下载
        for (int i = 0; i < list.size(); i++) {
            url = new URL(list.get(i));
            connection = url.openConnection();
            InputStream inputStream = connection.getInputStream();
            BufferedOutputStream outputStream = new BufferedOutputStream(new
FileOutputStream(String.valueOf(i) + ".png"));
            byte[] buf = new byte[1024];
            int len = 0;
            while ((len = inputStream.read(buf)) != -1) {
                outputStream.write(buf, 0, len);
            }
            outputStream.close();
        }
    }
```

```
}
```

https实现:

```
class Test {
    void test() throws IOException, InterruptedException {
        URL url = new URL("https://www.qq.com/");
        httpsURLConnection connection = (httpsURLConnection) url.openConnection();
        // 设置https的请求方式
        // connection.setRequestMethod("POST"); // 默认是GET
        String userAgent = "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.0.0 Safari/537.36";
        String cookie = "buvid3=CEAC3F76-9673-491E-10AC-6057A2CC7D7758840infoc; i-wanna-go-back=-1; _uuid=83211075F-2543-18410-475F-A106CBDD8DDA958965infoc; buvid4=E82B1EA6-E68F-AA7F-811F-F84D75D8C6CD47779-022041620-jZaAwkmHrIx0Axkn670sdA%3D%3D; buvid_fp_plain=undefined; buvid_fp=e7b0526ab0212daeb9027d32cbdfc40b; DedeUserID=178836086; DedeUserID__ckMd5=f9c5a4ea1f45eb94; b_ut=5; CURRENT_BLACKGAP=0; fingerprint3=b84859eebee1727dc4d76c080fddfed1; fingerprint=9a4906834ae3c364de742abe03ca7c5b; rpdid=|(ku|~k~|~0J'uYlmkl~~R~; LIVE_BUVID=AUTO6916592656596607; CURRENT_QUALITY=120; PVID=2; bp_video_offset_178836086=691552084947894300; CURRENT_FNVAL=80; innersign=0; b_lsid=C71E7D7F_182BA64917D; b_timer=%7B%22ffp%22%3A%7B%22333.1007.fp.risk_CEAC3F76%22%3A%22182BA6492BC%22%2C%22333.1193.fp.risk_CEAC3F76%22%3A%22182BA649781%22%7D%7D; SESSDATA=3d21f72c%2C1676536531%2C5990f%2A81; bili_jct=7ba0c772c6916636a27eb97326491d30; sid=8kzdy4il";
        connection.setRequestProperty(cookie, userAgent); // 伪造成浏览器
        BufferedReader br = new BufferedReader(new InputStreamReader(connection.getInputStream(), "GBK"));
        String msg = null;
        StringBuilder sb = new StringBuilder();
        while ((msg = br.readLine()) != null) {
            sb.append(msg);
        }
        String content = sb.toString();


        // 数据提取
        Pattern pattern = Pattern.compile("<img.*?src=\"(.*?)\".*?>");
        Matcher matcher = pattern.matcher(content);
        sb = null;
        List<String> list = new ArrayList<>();
        while (matcher.find()) {
            String s = matcher.group(1);
            if (s.startsWith("//")) {
                s = "https:" + s;
            }
```

```
            list.add(s);
            System.out.println(s);
        }

        // 数据下载
        for (int i = 0; i < list.size(); i++) {
            url = new URL(list.get(i));
            Thread.sleep(new Random().nextInt(10)); // 模拟人的行为
            connection = (httpsURLConnection) url.openConnection();
            InputStream inputStream = connection.getInputStream();
            BufferedOutputStream outputStream = new BufferedOutputStream(new
FileOutputStream(String.valueOf(i) + ".png"));
            byte[] buf = new byte[1024];
            int len = 0;
            while ((len = inputStream.read(buf)) != -1) {
                outputStream.write(buf, 0, len);
            }
            outputStream.close();
        }
    }
}
```

使用第三方httpsClient:

Lambda表达式:

此部分参考或搬运: https://www.erzbir.com/java/java-intersection-type.html

Lambda语法:

使用lambda表达式的一般语法是

`(Parameters) ->  { Body }`

`->`分隔参数和lambda表达式主体。

参数括在括号中，与方法相同，而lambda表达式主体是用大括号括起来的代码块。

lambda表达式主体可以有局部变量，语句。我们可以在lambda表达式主体中使用break，continue和return。我们甚至可以从lambda表达式主体中抛出异常。

lambda表达式没有名称，因为它表示匿名内部类。

lambda表达式的返回类型由编译器推断。

lambda表达式不能像方法一样有throws子句。

lambda表达式不能是泛型，而泛型在函数接口中定义。

显式和隐式lambda表达式:

未声明其参数类型的lambda表达式称为隐式lambda表达式。

显式lambda表达式是一个lambda表达式，它声明其参数的类型。

编译器将推断用于隐式lambda表达式的参数类型

显式:

以下代码使用单一方法创建接口，并将其用作lambda表达式类型。当创建lambda表达式时，我们声明参数`s1`的类型为`Integer`类型。

```
public class Main {

    public static void main(String[] args) {
        MyIntegerCalculator myIntegerCalculator = (Integer s1) -> s1 * 2;

        System.out.println("1- Result x2 : " + myIntegerCalculator.calcIt(5));

    }
}

interface MyIntegerCalculator {

    public Integer calcIt(Integer s1);

}
```

隐式:

这里是没有使用类型的演示。当忽略类型时，编译器必须计算出来。

```
public class Main {

    public static void main(String[] args) {
        MyIntegerCalculator myIntegerCalculator = (s1) -> s1 * 2;

        System.out.println("1- Result x2 : " + myIntegerCalculator.calcIt(5));

    }
}

interface MyIntegerCalculator {

    public Integer calcIt(Integer s1);

}
```

省略参数类型:

我们可以选择省略lambda表达式中的参数类型。

在lambda表达式 `(int x, int y) -> { return x + y; }` 声明的参数类型。

我们可以安全地重写lambda表达式，省略参数类型

`(x, y)  ->  { return x + y; }`

如果我们选择省略参数类型，我们必须省略所有参数的类型。

```
public class Main {
    public static void main(String[] argv) {
        Processor stringProcessor = (str) -> str.length();
        String name = "Java Lambda";
        int length = stringProcessor.getStringLength(name);
        System.out.println(length);

    }
}

@FunctionalInterface
interface Processor {
    int getStringLength(String str);
}
```

单参数:

对于单个参数lambda表达式，我们可以省略括号，因为我们省略了参数类型。

lambda表达式 `(String msg) -> {System.out.println(msg);}` 有一切。

然后我们可以省略参数类型

`(msg)->{System.out.println(msg);}`

我们可以进一步省略参数类型和括号，如下所示。

`msg ->  { System.out.println(msg); }`

```
public class Main {
    public static void main(String[] argv) {
        Processor stringProcessor = str -> str.length();
        String name = "Java Lambda";
        int length = stringProcessor.getStringLength(name);
        System.out.println(length);

    }
}

@FunctionalInterface
```

```
interface Processor {
    int getStringLength(String str);
}
```

无参数:

对于没有参数的lambda表达式，我们仍然需要括号。

`() -> { System.out.println("hi"); }`

以下示例显示如何使用`BooleanSupplier`。

```
import java.util.function.BooleanSupplier;

public class Main {
  public static void main(String[] args) {
    BooleanSupplier bs = () -> true;
    System.out.println(bs.getAsBoolean());

    int x = 0, y= 1;
    bs = () -> x > y;
    System.out.println(bs.getAsBoolean());
  }
}
```

final修饰符:

您可以在参数声明中为表达式lambda表达式使用`final`修饰符。

以下lambda表达式使用final修饰符。

`(final int x, final int y) -> { return x + y; }`

我们可以只使用一个修饰符如下。

```
public class Main {
    public static void main(String[] argv) {
        Processor stringProcessor = (final String str) -> str.length();
        String name = "Java Lambda";
        int length = stringProcessor.getStringLength(name);
        System.out.println(length);

    }
}

@FunctionalInterface
interface Processor {
    int getStringLength(String str);
}
```

Lambda表达式主体:

lambda表达式主体可以是块语句或单个表达式。

块语句用大括号括起来，而单个表达式可以没有大括号。

在块语句中，我们可以使用`return`语句返回值。

以下lambda表达式使用块语句并使用`return`语句返回总和。

`(int x, int y) -> { return x + y; }`

下面的lambda使用了一个表达式：

`(int x, int y) -> x + y`

表达式不需要大括号。

lambda不必返回值。以下两个lambda表达式只是将参数输出到标准输出，不返回任何内容。

`(String msg)->{System.out.println(msg);}` // a block statement

`(String msg)->System.out.println(msg)` //an expression

例:

```
1  public class Main {
2      public static void main(String[] argv) {
3          Processor stringProcessor = (String str) -> str.length();
4          String name = "Java Lambda";
5          int length = stringProcessor.getStringLength(name);
6          System.out.println(length);// www . j a  va 2  s. co m
7
8      }
9  }
10
11 @FunctionalInterface
12 interface Processor {
13     int getStringLength(String str);
14 }
```

Lambda类型推断:

lambda表达式表示函数接口的实例。

根据上下文，一个lambda表达式可以映射到不同的函数接口类型。

编译器推断lambda表达式的类型。

在下面的代码中有两个函数接口，`Processor`和`SecondProcessor`。

`Processor`有一个名为`getStringLength`的方法，它接受一个字符串作为参数，并返回`int`。

`SecondProcessor`有一个名为`noName`的方法，它接受一个字符串作为参数，并返回一个`int`。

从代码中，我们可以看到，我们可以为它们分配两个相同的lambda表达式。

```
public class Main {
    public static void main(String[] argv) {
        Processor stringProcessor = (String str) -> str.length();
        SecondProcessor secondProcessor = (String str) -> str.length();
        //stringProcessor = secondProcessor; //compile error
        String name = "Java Lambda";
        int length = stringProcessor.getStringLength(name);
        System.out.println(length);

    }
}

@FunctionalInterface
interface Processor {
    int getStringLength(String str);
}

@FunctionalInterface
interface SecondProcessor {
    int noName(String str);
}
```

`Processor`或`SecondProcessor`称为目标类型。

推断lambda表达式类型的过程称为目标类型。

编译器使用以下规则来确定lambda表达式是否可分配给其目标类型：

- 它必须是一个函数接口。
- lambda表达式的参数必须与函数接口中的抽象方法匹配。
- lambda表达式的返回类型与函数接口中抽象方法的返回类型兼容。
- 从lambda表达式抛出的检查异常必须与函数接口中抽象方法的已声明的throws子句兼容。

Lambda行为参数化:

我们可以将lambda表达式作为参数传递给方法。

以下代码创建了一个名为`Calculator`的函数接口。

在`Calculator`中有一个称为`calculate`的方法，它接受两个`int`参数并返回一个`int`值。

在`Main`类中有一个引擎方法，它接受函数接口`Calculator`作为参数。它从计算器调用计算方法并输出结果。``

在主方法中，我们用不同的lambda表达式调用引擎方法四次。

```
public class Main {
    public static void main(String[] argv) {
        engine((x, y) -> x + y);
        engine((x, y) -> x * y);
```

```
        engine((x, y) -> x / y);
        engine((x, y) -> x % y);
    }

    private static void engine(Calculator calculator) {
        int x = 2, y = 4;
        int result = calculator.calculate(x, y);
        System.out.println(result);
    }
}

@FunctionalInterface
interface Calculator {
    int calculate(int x, int y);
}
```

`engine`方法的结果取决于传递给它的lambda表达式。

引擎方法的行为被参数化。

通过其参数更改方法的行为称为行为参数化。

在行为参数化中，我们将在lambda表达式中封装的逻辑传递给数据的方法。

行为参数化模糊性:

编译器并不总是可以推断lambda表达式的类型。

一种情况是将lambda表达式传递给重载的方法。

在以下代码中有两个函数接口。 一个是`int`值计算，另一个用于`long`值。

在Main类中有称为`engine`的重载方法。 一个是期望`IntCalculator`，另一个是`LongCalculator`。

在main方法中，我们必须指定lambda表达式的参数，以指示我们要使用的重载函数的编译器。

```
public class Main {
    public static void main(String[] argv) {
        engine((int x, int y) -> x + y);
        engine((long x, long y) -> x * y);
        engine((int x, int y) -> x / y);
        engine((long x, long y) -> x % y);
    }

    private static void engine(IntCalculator calculator) {
        int x = 2, y = 4;
        int result = calculator.calculate(x, y);
        System.out.println(result);
    }

    private static void engine(LongCalculator calculator) {
        long x = 2, y = 4;
```

```
        long result = calculator.calculate(x, y);
        System.out.println(result);
    }
}

@FunctionalInterface
interface IntCalculator {
    int calculate(int x, int y);
}

@FunctionalInterface
interface LongCalculator {
    long calculate(long x, long y);
}
```

要解决歧义，我们可以通过指定参数的类型将隐式lambda表达式更改为explicit。这是为上面的代码做的。

或者我们可以使用cast如下。当第一次调用引擎时，我们将lambda表达式转换为`IntCalculator`。

```
public class Main {
    public static void main(String[] argv) {
        engine((IntCalculator) ((x, y) -> x + y));
        engine((long x, long y) -> x * y);
        engine((int x, int y) -> x / y);
        engine((long x, long y) -> x % y);
    }

    private static void engine(IntCalculator calculator) {
        int x = 2, y = 4;
        int result = calculator.calculate(x, y);
        System.out.println(result);
    }

    private static void engine(LongCalculator calculator) {
        long x = 2, y = 4;
        long result = calculator.calculate(x, y);
        System.out.println(result);
    }
}

@FunctionalInterface
interface IntCalculator {
    int calculate(int x, int y);
}

@FunctionalInterface
interface LongCalculator {
    long calculate(long x, long y);
}
```

或者我们可以避免直接使用lambda表达式作为参数。我们可以将lambda表达式分配给一个函数接口，然后将该变量传递给该方法。下面的代码显示了这种技术。

```
public class Main {
    public static void main(String[] argv) {
        IntCalculator iCal = (x, y) -> x + y;
        engine(iCal);
        engine((long x, long y) -> x * y);
        engine((int x, int y) -> x / y);
        engine((long x, long y) -> x % y);
    }

    private static void engine(IntCalculator calculator) {
        int x = 2, y = 4;
        int result = calculator.calculate(x, y);
        System.out.println(result);
    }

    private static void engine(LongCalculator calculator) {
        long x = 2, y = 4;
        long result = calculator.calculate(x, y);
        System.out.println(result);
    }
}

@FunctionalInterface
interface IntCalculator {
    int calculate(int x, int y);
}

@FunctionalInterface
interface LongCalculator {
    long calculate(long x, long y);
}
```

函数式接口:

函数式接口是具有一个方法的接口，用作lambda表达式的类型。

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent event);
}
```

`ActionListener`只有一个方法`actionPerformed`。它是一个函数式接口。无论调用什么单一方法，只要Java编译器具有兼容的方法签名，Java编译器就会将其匹配到您的lambda表达式。

lambda表达式表示函数式接口的实例。

lambda表达式的类型是一个函数式接口类型。

`(String str) -> str.length()` str.length() 获取一个String参数并返回其长度。

它的类型可以是任何具有抽象方法的函数接口类型，它使用String作为参数并返回int。

以下是这种函数式接口的示例:

```
1
2  @FunctionalInterface
3  interface Processor {
4      int getStringLength(String str);
5  }
```

我们可以为其函数式接口实例赋值lambda表达式。

`Processor stringProcessor = (String str) ->  str.length();`

在下面的代码中，我们为其函数接口赋值一个lambda表达式。然后我们通过调用函数接口中定义的方法来执行lambda表达式，并传入一个参数。

```
1  public class Main {
2      public static void main(String[] argv) {
3          Processor stringProcessor = (String str) -> str.length();
4          String name = "Java Lambda";
5          int length = stringProcessor.getStringLength(name);
6          System.out.println(length);
7
8      }
9  }
10
11 @FunctionalInterface
12 interface Processor {
13     int getStringLength(String str);
14 }
```

lambda表达式本身不能用作独立的表达式。

lambda表达式的类型由编译器推断。

函数式接口定义:

函数式接口是具有一个抽象方法的接口。

我们不能使用以下类型的方法来声明一个函数式接口：

- 默认方法
- 静态方法
- 从Object类继承的方法

一个函数式接口可以重新声明Object类中的方法。该方法不被视为抽象方法。因此，我们可以声明lambda表达式使用的另一种方法。

考虑`java.util`包中的Comparator类，如下所示:

```java
package java.util;

@FunctionalInterface
public interface Comparator<T> {
    // An  abstract method  declared in the functional interface
    int compare(T o1, T o2);

    // Re-declaration of the equals() method in the Object class
    boolean equals(Object obj);

    //...
}
```

`Comparator`接口有两个抽象方法：`compare()`和`equals()`。

`equals()`方法是Object类中的`equals()`方法的重新声明。

@FunctionalInterface注释:

`@FunctionalInterface`注释在java.lang包中定义。我们可以选择使用它来标记一个函数式接口。

如果注释`@FunctionalInterface`在非函数式接口或其他类型（如类）上注释，则会发生编译时错误。

具有一个抽象方法的接口仍然是一个功能接口，即使我们不用`@FunctionalInterface`注释。

```java
public class Main {
    public static void main(String[] argv) {
        Processor stringProcessor = (String str) -> str.length();
        String name = "Java Lambda";
        int length = stringProcessor.getStringLength(name);
        System.out.println(length);

    }
}

@FunctionalInterface
interface Processor {
    int getStringLength(String str);
}
```

通用函数式接口:

我们可以使用类型参数与函数式接口来创建通用函数式接口。

以下代码创建具有一个类型参数T的通用函数式参数函数接口。

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

以下代码使用抽象通用方法定义非通用函数式接口：

```
@FunctionalInterface
public interface Processor {
    <T> void process(T[] list);
}
```

Buildin函数式接口:

Java 8在包java.util.function中有函数式接口

Function:

表示接受类型`T`的参数并返回类型`R`的结果的函数。

```
public interface Function<T, R> {
    //...
    R apply(T t);
    //...
}
```

BiFunction:

表示一个函数，它接受类型T和U的两个参数，并返回类型R的结果。

```
public interface BiFunction<T, U, R> {
    //...
    R apply(T t, U u);
    //...
}
```

Predicate:

表示为指定参数返回`true`或`false`的布尔函数。

```
public interface Predicate<T> {
    //...
    boolean test(T t);
    //...
}
```

BiPredicate:

表示为两个指定的参数返回`true`或`false`的布尔函数。

```
public interface BiPredicate<T, U> {
    //...
    boolean test(T t, U u);
    //...
}
```

Consumer:

表示接受参数并且不返回结果的操作。

```
public interface Consumer<T> {
    //...
    void accept(T t);
    //...
}
```

BiConsumer

表示接受两个参数并且不返回结果的操作。

```
public interface BiConsumer<T, U> {
    //...
    void accept(T t, U u);
    //...
}
```

Supplier:

表示返回类型T的值的函数。

```
public interface Supplier<T> {
    //...
    T get();
    //...
}
```

UnaryOperator:

表示接受参数并返回相同类型的结果的函数。

```
public interface UnaryOperator<T> {
    //...
    T apply(T t);
    //...
}
```

BinaryOperator:

表示一个函数，它接受两个参数并返回相同类型的结果。

```
public interface BinaryOperator<T> {
    //...
    T apply(T t1, T t2);
    //...
}
```

上述通用buildin函数式接口都是更专用的函数式接口的通用版本。

例如，`IntConsumer`是`Consumer<T>`的专用版本。

交叉类型:

Java 8引入了一种称为交集类型的新类型。

交叉类型是多种类型的交叉。

交叉类型可能在投射中显示为目标类型。

在两种类型之间使用`Type1 & Type2`，以表示类型1，类型2的交集的新类型。

以下代码使用具有交集类型的转型，创建一个新的合成类型，它是所有类型的子类型。

它创建两个接口，Calculator是一个功能接口，可以与lambda表达式一起使用。另一个称为`NonFunction`，它不是函数接口。
为了创建一个lambda表达式并赋值给`NonFunction`，我们使用`&`来创建新的子类型。

交叉类型`NonFunction & Calculator`是一个函数接口。

```
public class Main {
    public static void main(String[] argv) {

        NonFunction nonFunction = (NonFunction & Calculator) (x, y) -> x + y;
    }
}

@FunctionalInterface
interface Calculator {
    long calculate(long x, long y);
}

interface NonFunction {
}
```

以下代码通过将`java.io.Serializable`标记接口与我们自己的功能接口相交来创建交叉类型。

```
public class Main {
    public static void main(String[] argv) {

        java.io.Serializable ser = (java.io.Serializable & Calculator) (x, y) -> x + y;
    }
}

@FunctionalInterface
interface Calculator {
    long calculate(long x, long y);
}
```

以这种方式，我们使一个lambda表达式可序列化。

函数接口:

`Function<T, R>` 接口有六个特殊化：

- `IntFunction<R>`
- `LongFunction<R>`
- `DoubleFunction<R>`
- `ToIntFunction<T>`
- `ToLongFunction<T>`
- `ToDoubleFunction<T>`

`IntFunction<R>, LongFunction<R>, 和 DoubleFunction<R>` 分别以int，long和double作为参数，它们的返回值在类型R中。

`ToIntFunction<T>, ToLongFunction<T>, 和 ToDoubleFunction<T>` 取T类型的参数，并分别返回int，long和double。

辅助方法:

`Function` 接口包含以下默认和静态方法:

```

@FunctionalInterface
public interface Function<T, R> {
    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after);

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before);

    static <T> Function<T, T> identity();
}
```

`andThen()` 创建一个 `Function`，调用当前函数和指定的函数后得到结果。

`compose()` 创建一个 `Function`，该函数调用指定的函数，然后调用当前函数并返回结果。

`identify()`创建一个返回其参数的函数。

谓词接口:

`谓词`接口包含以下默认和静态方法。

我们可以使用NOT，AND和OR方法来创建基于其他谓词的谓词。

```

@FunctionalInterface
public interface Predicate<T> {
    default Predicate<T> negate();

    default Predicate<T> and(Predicate<? super T> other);

    default Predicate<T> or(Predicate<? super T> other);

    static <T> Predicate<T> isEqual(Object targetRef);
}
```

`negate()`否定原始谓词的谓词。

`and()`组合两个具有短路逻辑AND的谓词。

`or()`组合了具有短路逻辑或的两个谓词。

`isEqual()`返回一个谓词，根据Objects.equals(Object，Object)测试两个参数是否相等。

我们可以链接上述方法来创建复杂谓词。

以下示例显示如何使用`Predicate`。

```
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        Predicate<String> i = (s) -> s.length() > 5;
        System.out.println(i.test("erzbir.com "));
    }
}
```

方法引用:

lambda表达式表示在函数接口中定义的匿名函数。

方法引用使用现有方法创建lambda表达式。

方法引用的一般语法是

`Qualifier::MethodName`

两个连续的冒号充当分隔符。

`MethodName`是方法的名称。

`限定符`告诉在哪里找到方法引用。

例如，我们可以使用`String::length`从`String`类引用length方法。这里`String`是限定符，`length`是方法名。

我们只需要指定方法名。

无需指定方法的参数类型和返回类型。

方法引用的目标类型是功能接口。它确定方法的签名并在必要时解析重载的方法。

方法引用类型:

有六种类型的方法引用。

- TypeName::staticMethod - 引用类的静态方法，接口或枚举
- objectRef::instanceMethod - 引用实例方法
- ClassName::instanceMethod - 从类中引用实例方法
- TypeName.super::instanceMethod - 从对象的父类型引用实例方法
- ClassName::new - 引用一个类的构造函数
- ArrayTypeName::new - 对指定数组类型的构造函数的引用

静态方法引用:

静态方法引用允许我们使用静态方法作为lambda表达式。

静态方法可以在类，接口或枚举中定义。

以下代码定义了两个lambda表达式。

第一个lambda表达式`func1`是通过定义输入参数x并提供lambda表达式主体来创建的。基本上，这是创建lambda表达式的正常方式。

第二个lambda表达式`func2`是通过从Integer类引用静态方法创建的。

```
import java.util.function.Function;

public class Main {
    public static void main(String[] argv) {
        // Using  a  lambda  expression
        Function<Integer, String> func1 = x -> Integer.toBinaryString(x);
        System.out.println(func1.apply(10));

        // Using  a  method  reference
        Function<Integer, String> func2 = Integer::toBinaryString;
        System.out.println(func2.apply(10));
    }
}
```

从Integer类的静态方法的签名如下。

`static String toBinaryString(int i)`

以下代码显示了如何使用`Integer.sum`作为lambda表达式。

```
import java.util.function.BiFunction;

public class Main {
    public static void main(String[] argv) {

        // Uses a lambda expression
        BiFunction<Integer, Integer, Integer> func1 = (x, y) -> Integer.sum(x, y);
        System.out.println(func1.apply(2, 3));

        // Uses a method reference
        BiFunction<Integer, Integer, Integer> func2 = Integer::sum;
        System.out.println(func2.apply(2, 3));

    }
}
```

重载中的静态方法引用:

我们可以在静态方法引用中使用重载的静态方法。

当重载方法时，我们必须更加注意方法签名和相应的函数接口。

在下面的列表中，我们有来自Integer类的三个版本的valueOf()。

`static Integer valueOf(int i)`
`static Integer  valueOf(String s)`
`static Integer  valueOf(String s, int radix)`

以下代码显示了如何使用不同的目标函数接口与重载的Integer.valueOf()静态方法。

```
import java.util.function.BiFunction;
import java.util.function.Function;

public class Main {
    public static void main(String[] argv) {
        // Uses  Integer.valueOf(int)
        Function<Integer, Integer> func1 = Integer::valueOf;

        // Uses  Integer.valueOf(String)
        Function<String, Integer> func2 = Integer::valueOf;

        // Uses  Integer.valueOf(String, int)
        BiFunction<String, Integer, Integer> func3 = Integer::valueOf;

        System.out.println(func1.apply(7));
        System.out.println(func2.apply("7"));
        System.out.println(func3.apply("101010101010", 2));
```

```
    }
}
```

实例方法引用:

我们可以通过两种方式获得一个实例方法引用，从对象实例或从类名。

基本上我们有以下两种形式。

- instance::MethodName
- ClassName::MethodName

这里`实例`表示任何对象实例。`ClassName`是的名称类，例如`String`，`Integer`。

`实例`和`ClassName`称为接收器。更具体地说，`instance`被称为有界接收器，而`ClassName`被称为无界接收器。

我们称为实例有界接收器，因为接收器被限制到实例。

`ClassName`是未经过排队的接收器，因为接收器以后有界。

绑定实例方法引用:

绑定接收器接收器具有以下形式：

`instance::MethodName`

在下面的代码中，我们使用buildin系统函数接口Supplier作为lambda表达式类型。

首先，我们以正常的方式定义一个lambda表达式。 lambda表达式不接受参数，并返回字符串 'erzbir.com' 的长度，

我们使用 'erzbir.com' 创建一个String实例，并使用它的length方法作为实例方法引用。

绑定意味着我们已经指定了实例。

以下示例显示如何使用没有参数的绑定接收器和方法来创建实例方法引用。

```
import java.util.function.Supplier;

public class Main {
    public static void main(String[] argv) {
        Supplier<Integer> supplier = () -> "erzbir.com".length();
        System.out.println(supplier.get());


        Supplier<Integer> supplier1 = "erzbir.com"::length;
        System.out.println(supplier1.get());
    }
}
```

以下示例显示如何使用绑定接收器和方法与参数创建实例方法引用。

```
import java.util.function.Consumer;

```

```
public class Main {
    public static void main(String[] argv) {
        Util util = new Util();

        Consumer<String> consumer = str -> util.print(str);
        consumer.accept("Hello");


        Consumer<String> consumer1 = util::print;
        consumer1.accept("erzbir.com");

        util.debug();
    }
}

class Util {
    private int count = 0;

    public void print(String s) {
        System.out.println(s);
        count++;
    }

    public void debug() {
        System.out.println("count:" + count);
    }
}
```

未绑定实例方法引用:

未绑定的接收器使用以下语法
`ClassName::instanceMethod`

它与我们用来引用静态方法的语法相同。

从以下代码，我们可以看到输入类型是ClassName的类型。

在下面的代码中，我们使用`String:length`，因为函数接口的输入类型为`String`。

lambda表达式在使用时获取输入。

以下代码使用String length方法作为unbind实例方法引用。

String length方法通常在字符串值实例上调用，并返回字符串实例的长度。因此，输入是String类型，输出是int类型，这是匹配buildin函数功能接口。

每次我们调用`str Length Func`，我们传入一个字符串值，并从传递的字符串值中调用length方法。

```
import java.util.function.Function;

public class Main {
    public static void main(String[] argv) {
```

```
        Function<String, Integer> strLengthFunc = String::length;
        String name = "erzbir.com";
        int len = strLengthFunc.apply(name);
        System.out.println("name  = " + name + ", length = " + len);

        name = "www.erzbir.com";
        len = strLengthFunc.apply(name);
        System.out.println("name  = " + name + ", length = " + len);

    }
}
```

下面的代码定义了一个具有称为append的静态方法的类Util。

`append`方法接受两个`String`类型参数，并返回一个`String`类型的结果。

然后使用`append`方法创建一个lambda表达式并赋值给Java buildin `BiFunction`函数接口。

`append`方法的签名与`BiFunction`函数接口中定义的抽象方法的签名相匹配。

```
import java.util.function.BiFunction;

public class Main {
    public static void main(String[] argv) {
        BiFunction<String, String, String> strFunc = Util::append;
        String name = "erzbir.com";
        String s = strFunc.apply(name, "hi");
        System.out.println(s);
    }
}

class Util {
    public static String append(String s1, String s2) {
        return s1 + s2;
    }
}
```

超类型实例方法引用:

关键字`super`仅在实例上下文中使用，引用覆盖的方法。

我们可以用下面的语法来创建是指在父类型的实例方法的方法引用。

`ClassName.super::instanceMethod`

下面的代码定义了一个称为`ParentUtil`的父类。在`ParentUtil`中有一个名为`append`的方法，它将两个String值附加在一起。

然后创建一个名为`Util`的子类并扩展`ParentUtil`。

在`Util`类中，覆盖`append`方法。

在Util的构造函数中，我们创建两个lambda表达式，一个是使用Util的append方法，另一个是使用ParentUtil类的append方法。

我们使用`this::append`引用当前类，同时使用`Util.super::append`从父类引用方法。

```
import java.util.function.BiFunction;

public class Main {
    public static void main(String[] argv) {
        new Util();
    }
}

class Util extends ParentUtil {

    public Util() {
        BiFunction<String, String, String> strFunc = this::append;
        String name = "erzbir.com";
        String s = strFunc.apply(name, " hi");
        System.out.println(s);

        strFunc = Util.super::append;
        name = "erzbir.com";
        s = strFunc.apply(name, " Java Lambda Tutorial");
        System.out.println(s);

    }

    @Override
    public String append(String s1, String s2) {
        System.out.println("child append");
        return s1 + s2;
    }
}

class ParentUtil {
    public String append(String s1, String s2) {
        System.out.println("parent append");
        return s1 + s2;
    }
}
```

构造函数引用:

我们可以使用构造函数创建一个lambda表达式。

使用构造函数引用的语法是:

`ClassName::new`

关键字new指的是类的构造函数。编译器根据上下文选择一个构造函数。

```
import java.util.function.Function;
import java.util.function.Supplier;

public class Main {
    public static void main(String[] argv) {
        Supplier<String> func1 = () -> new String();
        System.out.println("Empty String:" + func1.get());

        Function<String, String> func2 = str -> new String(str);

        System.out.println(func2.apply("erzbir.com"));

        Supplier<String> func3 = String::new;
        System.out.println("Empty String:" + func3.get());

        Function<String, String> func4 = String::new;
        System.out.println(func4.apply("erzbir.com"));
    }
}
```

数组构造函数引用:

我们可以使用数组构造函数创建一个数组如下。

`ArrayTypeName::new`

`int [] :: new`是调用`new int []`。`new int []`需要一个`int`类型值作为数组长度，因此`int [] :: new`需要一个`int`类型输入值。

以下代码使用数组构造函数引用创建一个int数组。

```
import java.util.Arrays;
import java.util.function.IntFunction;

public class Main{
  public static void main(String[] argv){
    IntFunction<int[]> arrayCreator1 = size ->  new int[size];
    // Creates an  int array of  five  elements
    int[] intArray1  = arrayCreator1.apply(5);
    System.out.println(Arrays.toString(intArray1));

    IntFunction<int[]> arrayCreator2 = int[]::new;
    int[] intArray2 = arrayCreator2.apply(5);
    System.out.println(Arrays.toString(intArray2));
  }
}
```

通过使用`Function< Integer, Array Type>`，我们可以在声明中指定数组类型。

```java
import java.util.Arrays;
import java.util.function.Function;

public class Main {
    public static void main(String[] argv) {
        Function<Integer, int[]> arrayCreator3 = int[]::new;
        int[] intArray = arrayCreator3.apply(5);
        System.out.println(Arrays.toString(intArray));
    }
}
```

我们可以在创建二维数组时指定第一维的长度。

```java
import java.util.Arrays;
import java.util.function.IntFunction;

public class Main {
    public static void main(String[] argv) {
        IntFunction<int[][]> TwoDimArrayCreator = int[][]::new;
        int[][] intArray = TwoDimArrayCreator.apply(5);
        // Creates an  int[5][]  array
        intArray[0] = new int[5];
        intArray[1] = new int[5];
        intArray[2] = new int[5];
        intArray[3] = new int[5];
        intArray[4] = new int[5];

        System.out.println(Arrays.deepToString(intArray));
    }
}
```

通用方法引用:

我们可以通过指定实际的类型参数来在方法引用中使用通用方法。

语法如下:

`ClassName::<TypeName>methodName`

通用构造函数引用的语法:

`ClassName<TypeName>::new`

以下代码使用通用的Arrays.asList方法创建lambda表达式，并将参数设置为String。

```java
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;

public class Main {
    public static void main(String[] argv) {
        Function<String[], List<String>> asList = Arrays::<String>asList;

        System.out.println(asList.apply(new String[]{"a", "b", "c"}));
    }
}
```

Lambda表达式作用域:

lambda表达式不定义自己的范围。

如果我们在lambda中使用关键字`this`和`super`表达式在方法中，它们的行为与我们在该方法中使用它们一样。

以下代码从lambda表达式输出this。这在lambda表达式中是指外部类不是lambda表达式本身。

```java
import java.util.function.Function;

public class Main {
    public Main() {
        Function<String, String> func1 = x -> {
            System.out.println(this);
            return x;
        };
        System.out.println(func1.apply(""));
    }

    public String toString() {
        return "Main";
    }

    public static void main(String[] argv) {
        new Main();
    }
}
```

Main方法中的第一行具有`x`的变量定义。

如果我们删除注释，我们会得到编译时错误，因为它与lambda表达式的变量定义冲突。

这是另一个演示，显示lambda表达式与其外部方法具有相同的范围。 lambda表达式不会创建自己的作用域。

```java
import java.util.function.Function;

public class Main {
```

```
    public Main() {
        //int x= 0;
        Function<String, String> func1 = x -> {
            System.out.println(this);
            return x;
        };
        System.out.println(func1.apply(""));
    }

    public String toString() {
        return "Main";
    }

    public static void main(String[] argv) {
        new Main();
    }
}
```

Lambda变量捕获:

lambda表达式可以访问最终局部变量或局部非最终初始化只有一次的变量。

下面的代码显示我们可以访问和使用最终的局部变量。

```
import java.util.function.Function;

public class Main {
    public static void main(String[] argv) {
        final String x = "Hello";
        Function<String, String> func1 = y -> {
            return y + " " + x;
        };
        System.out.println(func1.apply("erzbir.com"));

    }
}
```

下面的代码有一个变量x，它不是final，只能初始化一次。我们仍然可以在lambda表达式中使用它。

```java
import java.util.function.Function;

public class Main {
    public static void main(String[] argv) {
        String x = "Hello";

        Function<String, String> func1 = y -> {
            return y + " " + x;
        };
        System.out.println(func1.apply("erzbir.com"));

    }
}
```

下面的代码显示我们不能改变在lambda表达式之外定义的值。

```java
import java.util.function.Function;

public class Main {
    public static void main(String[] argv) {
        String x = "Hello";

        Function<String, String> func1 = y -> {/*x="a";*/
            return y + " " + x;
        };
        System.out.println(func1.apply("erzbir.com"));

    }
}
```

我们可以更改lambda表达式中的非局部变量。

```java
import java.util.function.Function;

public class Main {
    static String x = "Hello";

    public static void main(String[] argv) {


        Function<String, String> func1 = y -> {
            x = "a";
            return y + " " + x;
        };
        System.out.println(func1.apply("erzbir.com"));

    }
}
```

递归Lambda:

我们可以在创建递归lambda表达式时使用方法引用。

以下代码以正常方式创建递归函数，然后使用递归函数作为方法引用来创建lambda表达式。最后的lambda表达式成为递归。

```
import java.util.function.IntFunction;

public class Main {
    public static void main(String[] args) {
        IntFunction<Long> factorialCalc = Main::factorial;
        System.out.println(factorialCalc.apply(10));
    }

    public static long factorial(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }
}
```

值引用:

强引用:

在JDK1.2以前只有强引用

例如:

`Object o = new Object();`

这时o就是一个强引用, gc无法回收, 需要手动回收"o = null"

软引用:

内存空间足够时垃圾回收器不会回收它, 如果内存空间不足, 就会回收这些对象的内存

可用于实现内存敏感的高速缓存

软引用可以和一个引用队列(`ReferenceQueue`)联合使用

如果软引用所引用的对象被垃圾回收, Java虚拟机就会把这个软引用加入到与之关联的引用队列中

`SoftReference<byte[]>res=new SoftReference<>(new byte[1024]);` // res里面的byte数组是软引用. 只被软引用关联着的对象,
在系统将要发生内存溢出异常前. 会把这些对象列进回收范围之中进行第二次回收. 如果这次回收还没有足够的内存, 才会抛出内存溢出异常.

弱引用:

生命周期短. JVM进行垃圾回收, 一旦发现弱引用对象, 无论当前内存是否充足, 都会将其回收. 不过由于垃圾回收器是一个优先级较低的线程,
所以并不一定能迅速发现弱引用对象

```
WeakReference<Object> m=new WeakReference<>();
```

用途: 用于ThreadLocal中

虚引用:

如果一个对象仅持有虚引用, 那么它就和没有任何引用一样, 在任何时候都可能被垃圾回收, 虚引用主要用来跟踪对象被垃圾回收的活动,
虚引用必须和引用队列(ReferenceQueue)联合使用. 当垃圾回收器准备回收一个对象时, 如果发现它还有虚引用, 就会在回收对象之前,
把这个虚引用加入到与之关联引用队列中. 程序如果发现某个虚引用已经被加入到引用队列, 那么就可以在所引用的对象的内存被回收之前采取必要的行动

```
ReferenceQueue<Object> queue=new ReferenceQueue<>();
PhantomReference<Object> m=new PhantomReference(new Object().queue);
```

用途: 虚引用的唯一作用是管理堆外内存, 因为JVM无法直接清理堆外内存, 所以提供一个虚引用, 交给垃圾回收器的回收队列,
这个队列是用来标记哪些堆外内存需要回收, 再调用C++释放空间

JDK中直接内存的回收就用到虚引用

设计模式(此部分开始为Java高级):

回调(Callback):

在这种模式中, 可以指定某个特定事件发生时应该采取的动作. 比如鼠标点下某个按键时, 监听就是这个逻辑

回调四种写法:

- 反射
- 直接调用
- Lambda

```
1  /**
2   * 回调接口
3   */
4  interface CallBack {
5      void call();
6  }
7
8  /**
9   * 具体实现类
10  */
```

```java
class CallBackImpl implements CallBack {

    @Override
    public void call() {
        System.out.println("执行回调函数");
    }
}

/**
 * 反射调用
 */
class Listener1 {

    public void execute(Class<? extends CallBack> clazz, Method method) throws NoSuchMethodException, InstantiationException, InterruptedException {
        System.out.println("监听到事件");
        System.out.println("开始执行处理方法");
        try {
            method.invoke(clazz.getDeclaredConstructor().newInstance());
        } catch (IllegalAccessException | InvocationTargetException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws InterruptedException, NoSuchMethodException, InstantiationException {
        Listener1 listener1 = new Listener1();
        CallBackImpl callBack = new CallBackImpl();
        listener1.execute(callBack.getClass(), callBack.getClass().getMethod("call"));
    }
}

/**
 * 直接调用
 */
class Listener2 {

    public void execute(CallBack callback) {
        System.out.println("监听到事件");
        System.out.println("开始执行处理方法");
        callback.call();
    }

    public static void main(String[] args) {
        Listener2 listener2 = new Listener2();
        CallBackImpl callBack = new CallBackImpl();
        listener2.execute(callBack);
    }
```

```
}

/**
 * Lambda调用
 */

class Listener3 {
    public void execute(CallBack callBack) {
        System.out.println("监听到事件");
        System.out.println("开始执行处理方法");
        callBack.call();
    }

    public static void main(String[] args) {
        Listener3 listener3 = new Listener3();
        listener3.execute(() -> System.out.println("执行回调函数"));
    }
}

```

以下代码每隔两秒就会打印一次时间, 直到点击ok

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.time.Instant;

class TimePrinter implements ActionListener {

    // 回调函数
    @Override
    public void actionPerformed(ActionEvent event) {
        System.out.println("now time is " + Instant.ofEpochMilli(event.getWhen()));
    }
}

class Test {
    public static void main(String[] args) {
        var listener = new TimePrinter(); // 创建监听
        var timer = new Timer(2000, listener); // 定时器
        timer.start();
        JOptionPane.showMessageDialog(null, "是否退出"); // 弹出一个确认框
        System.exit(0);
    }
}
```

代理(Proxy):

工厂模式(Factory):

简单工厂模式(Simple Factory):

> 简单工厂模式又称静态工厂方法模式. 简单工厂模式中, 一个工厂类处于对产品类实例化调用的中心位置上, 它决定那一个产品类应当被实例化

组成:

- 工厂类角色: 这是本模式的核心, 含有一定的商业逻辑和判断逻辑. 在java中它往往由一个具体类实现.
- 抽象产品角色: 它一般是具体产品继承的父类或者实现的接口。在java中由接口或者抽象类来实现.
- 具体产品角色: 工厂类所创建的对象就是此角色的实例. 在java中由一个具体类实现.

适用场景:

工厂类负责创建的对象比较少，客户只知道传入了工厂类的参数，对于始何创建对象(逻辑)不关心。

优点:

- 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程。
- 灵活性增强，对于新产品的创建，只需多写一个相应的工厂类。
- 典型的解耦框架。高层模块只需要知道产品的抽象类，无须关心其他实现类，满足迪米特法则、依赖倒置原则和里氏替换原则。

缺点:

- 类的个数容易过多，增加复杂度
- 增加了系统的抽象性和理解难度
- 抽象产品只能生产一种产品，此弊端可使用抽象工厂模式解决。

代码:

```
// 抽象产品, 水果
interface Fruit {
    void eat();
}

// 具体产品, 苹果
class Apple implements Fruit {
    public Apple() {
        System.out.println("这是一个苹果");
    }

    @Override
    public void eat() {
        System.out.println("吃苹果");
    }
}

// 具体产品, 桃子
```

```
class Peach implements Fruit {
    public Peach() {
        System.out.println("这是一个桃子");
    }

    @Override
    public void eat() {
        System.out.println("吃桃子");
    }
}

class FruitFactory {
    public static Fruit createFruit(Class<? extends Fruit> type) throws
NoSuchMethodException, InvocationTargetException, InstantiationException,
IllegalAccessException {
        if (type != Apple.class && type != Peach.class) {
            throw new IllegalArgumentException("错误");
        }
        return type.getConstructor().newInstance();
    }

    public static Fruit createFruit(String name) {
        if (name.equals("苹果")) {
            return new Apple();
        } else if (name.equals("桃子")) {
            return new Peach();
        }
        throw new IllegalArgumentException("此工厂不能制造这个产品");
    }
}

public class Test {
    public static void main(String[] args) throws InvocationTargetException,
NoSuchMethodException, InstantiationException, IllegalAccessException {
        Apple apple = (Apple) FruitFactory.createFruit(Apple.class);
        Peach peach = (Peach) FruitFactory.createFruit("桃子");
        apple.eat();
        peach.eat();
    }
}
```

工厂方法模式(Method Factory):

> 工厂方法模式是简单工厂模式的进一步抽象化和推广, 工厂方法模式里不再只由一个工厂类决定那一个产品类应当被实例化,这个决定被交给抽象工厂的子类去做.

组成：

- 抽象工厂角色: 这是工厂方法模式的核心, 它与应用程序无关. 是具体工厂角色必须实现的接口或者必须继承的父类.
  在java中它由抽象类或者接口来实现.
- 具体工厂角色: 它含有和具体业务逻辑有关的代码. 由应用程序调用以创建对应的具体产品的对象.
- 抽象产品角色: 它是具体产品继承的父类或者是实现的接口. 在java中一般有抽象类或者接口来实现.
- 具体产品角色: 具体工厂角色所创建的对象就是此角色的实例. 在java中由具体的类来实现.

适用场景:

当一个类不知道它所必须创建对象的类或**一个类希望由子类来指定它所创建的对象时**，**当类将创建对象的职责委托给多个帮助子类中的某一个**
，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候，可以使用工厂方法，**支持多扩展少修改的OCP原则**

优点：

- 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程。
- 灵活性增强，对于新产品的创建，只需多写一个相应的工厂类。
- 典型的解耦框架。高层模块只需要知道产品的抽象类，无须关心其他实现类，满足迪米特法则、依赖倒置原则和里氏替换原则。

缺点：

- 类的个数容易过多，增加复杂度
- 增加了系统的抽象性和理解难度
- 抽象产品只能生产一种产品，此弊端可使用抽象工厂模式解决。

代码:

```
//抽象产品角色, 载具
public interface Moveable {
    void run();
}

//具体产品角色, 飞机
public class Plane implements Moveable {
    @Override
    public void run() {
        System.out.println("plane....");
    }
}

//具体产品角色, 车
public class Car implements Moveable {

    @Override
    public void run() {
        System.out.println("car.....");
```

```
    }
}

//抽象工厂, 载具工厂
public abstract class VehicleFactory {

    abstract Moveable create();
}

//具体工厂, 飞机工厂
public class PlaneFactory extends VehicleFactory {

    public Moveable create() {
        return new Plane();
    }
}

//具体工厂, 汽车工厂
public class CarFactory extends VehicleFactory {

    public Moveable create() {
        return new Broom();
    }
}

//测试类
public class Test {
    public static void main(String[] args) {
        VehicleFactory factory = new CarFactory();
        Moveable m = factory.create();
        m.run();
    }
}
```

抽象工厂模式(Abstract Factory):

> 抽象工厂模式是工厂方法模式的升级版本，他用来创建一组相关或者相互依赖的对象. 在抽象工厂模式中, 抽象产品 (
> AbstractProduct) 可能是一个或多个, 从而构成一个或多个产品族(Product Family). 在只有一个产品族的情况下,
> 抽象工厂模式实际上退化到工厂方法模式.

适用场景:

一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。这个系统有多于一个的产品族，而系统只消费其中某一产品族。同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现

优点：

- 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程。
- 灵活性增强，对于新产品的创建，只需多写一个相应的工厂类。
- 典型的解耦框架。高层模块只需要知道产品的抽象类，无须关心其他实现类，满足迪米特法则、依赖倒置原则和里氏替换原则。

缺点：

- 类的个数容易过多，增加复杂度
- 增加了系统的抽象性和理解难度
- 抽象产品只能生产一种产品，此弊端可使用抽象工厂模式解决。

代码:

```
//发动机以及型号
interface Engine {

}

class EngineA extends Engine {
    public EngineA() {
        System.out.println("制造-->EngineA");
    }
}

class EngineB extends Engine {
    public EngineB() {
        System.out.println("制造-->EngineB");
    }
}

//空调以及型号
interface AirCondition {

}

class AirConditionA extends AirCondition {
    public AirconditionA() {
        System.out.println("制造-->AirConditionA");
    }
}

class AirConditionB extends AirCondition {
    public AirconditionB() {
        System.out.println("制造-->AirConditionB");
    }
}

```

```java
//创建工厂的接口
interface AbstractFactory {
    //制造发动机
    Engine createEngine();

    //制造空调
    AirCondition createAirCondition();
}

//为宝马320系列生产配件
class FactoryBMW320 implements AbstractFactory {

    @Override
    public Engine createEngine() {
        return new EngineA();
    }

    @Override
    public Aircondition createAirCondition() {
        return new AirConditionA();
    }
}

//宝马523系列
class FactoryBMW523 implements AbstractFactory {

    @Override
    public Engine createEngine() {
        return new EngineB();
    }

    @Override
    public Aircondition createAirCondition() {
        return new AirConditionB();
    }
}

public class Test {
    public static void main(String[] args) {
        //生产宝马320系列配件
        FactoryBMW320 factoryBMW320 = new FactoryBMW320();
        factoryBMW320.createEngine();
        factoryBMW320.createAircondition();

        //生产宝马523系列配件
        FactoryBMW523 factoryBMW523 = new FactoryBMW523();
        factoryBMW320.createEngine();
        factoryBMW320.createAircondition();
    }
```

```
84 }
```

监听者模式(Listener):