5. 方法的使用

【本节目标】

1. 掌握方法的定义以及使用
2. 掌握方法传参
3. 掌握方法重载
4. 掌握递归

1. 一个例子

每年在秋招，比特都会做一件重要的事情：校招跟进，帮助大家解决校招中遇到的各种问题，以及鼓励大家积极坚持找下去，斩获自己满意的offer。但是大家普遍都会遇到一些类似的问题：

学生A：老师最近笔试总过不了？怎么办啊?
老师：你大概投递了多少简历呢?
学生A：投递了有20多家了
老师：你投递太少了，现在大家基本都投到100多家了
学生A：好的老师，下来我多投下
老师：投递的时候，不要挑，大小厂全投了
老师：简历投越多，收到笔试的机会越多
老师：笔试的时候，可以优先选择招人比较多的公司，招人多，笔试通过的概率更高
老师：好好笔试，有问题随时和我联系着
老师：你一定没问题的，加油

学生B：老师好难受啊，最近笔试总过不了
老师：你大概投递了多少简历呢?
学生B：投递了10来家左右
老师：投递太少了，现在大家基本都投到100+
学生B：那估计是的吧，我下来再多投投
老师：投递的时候，不要挑，大小厂全投了
老师：简历投越多，收到笔试的机会越多
老师：笔试的时候，可以优先选择招人比较多的公司，招人多，笔试通过的概率更高
老师：好好笔试，有问题随时和我联系着
老师：你一定没问题的，加油

过了几天之后，**又有很多同学问类似的问题，老师不得不将之前重复做的事情再做一遍**，后来一想，估计还有好多同学遇到，可能不好意思问，**于是老师写了一篇帖子，发到了比特论坛**：

笔试过不了这里集合：
最近有学生反馈笔试总过不了，建议：
1. 多投递简历，简历投递越多越好
2. 投递简历时不要挑，大小厂全投
3. 笔试的时候优先选择招人多的公司，招人多，笔试通过率更高
4. 大家有问题随时和我联系
加油，你一定没问题。

然后将帖子的链接发到群中，说：最近笔试不顺利的同学看这里，https://www.xxx.com **学生点击链接就可以看到帖子的内容。**

从这个过程我们可以看到：

1. 老师将学生普遍的问题整理成帖子，减少了重复性工作，然后就有时间解决同学更多的问题
2. 同学点击链接，就可以进入帖子进行阅读，问题得到解决。
3. 学生随机可以点击链接阅读，而不需要一遍一遍和老师做重复的事情。

在编程中也是一样，某段功能的代码可能频繁使用到，如果在每个位置都重新实现一遍，会：

1. 使程序变得繁琐
2. 开发效率低下，做了大量重复性的工作
3. 不利于维护，需要改动时，所有用到该段代码的位置都需要修改
4. 不利于复用

因此，在编程中，我们也可以将频繁使用的代码封装成"帖子"(方法)，需要时直接拿来链接(即方法名--方法的入口地址)使用即可，避免了一遍一遍的累赘。

2. 方法概念及使用

2.1 什么是方法(method)

方法就是一个代码片段. 类似于 C 语言中的 "函数"。方法存在的意义(不要背, 重在体会):

1. 是能够模块化的组织代码(当代码规模比较复杂的时候).
2. 做到代码被重复使用, 一份代码可以在多个位置使用.
3. 让代码更好理解更简单.
4. 直接调用现有方法开发, 不必重复造轮子.

比如：现在要开发一款日历，在日历中经常要判断一个年份是否为闰年，则有如下代码：

```
int  year = 1900;
if((0 == year % 4 && 0 != year % 100) || 0 == year % 400){
   System.out.println(year+"年是闰年");
}else{
   System.out.println(year+"年不是闰年");
}
```

那方法该如何来定义呢?

1.2 方法定义

方法语法格式

```
// 方法定义
修饰符 返回值类型 方法名称([参数类型 形参 ...]){
   方法体代码;
   [return 返回值];
}
```

示例一：实现一个函数，检测一个年份是否为闰年

```
public class Method{
    // 方法定义
    public static boolean isLeapYear(int year){
        if((0 == year % 4 && 0 != year % 100) || 0 == year % 400){
            return true;
        }else{
            return false;
        }
    }
}
```

示例二: 实现一个两个整数相加的方法

```
public class Method{
    // 方法的定义
    public static int add(int x, int y) {
        return x + y;
    }
}
```

【注意事项】

1. 修饰符：现阶段直接使用public static 固定搭配
2. 返回值类型：如果方法有返回值，返回值类型必须要与返回的实体类型一致，如果没有返回值，必须写成void
3. 方法名字：采用小驼峰命名
4. 参数列表：如果方法没有参数，()中什么都不写，如果有参数，需指定参数类型，多个参数之间使用逗号隔开
5. 方法体：方法内部要执行的语句
6. 在java当中，方法必须写在类当中
7. 在java当中，方法不能嵌套定义
8. 在java当中，没有方法声明一说

1.3 方法调用的执行过程

【方法调用过程】

调用方法--->传递参数--->找到方法地址--->执行被调方法的方法体--->被调方法结束返回--->回到主调方法继续往下执行

【注意事项】

- 定义方法的时候, 不会执行方法的代码. 只有调用的时候才会执行.
- 一个方法可以被多次调用.

代码示例1 计算两个整数相加

```
public class Method {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        System.out.println("第一次调用方法之前");
        int ret = add(a, b);
        System.out.println("第一次调用方法之后");
        System.out.println("ret = " + ret);

        System.out.println("第二次调用方法之前");
        ret = add(30, 50);
        System.out.println("第二次调用方法之后");
        System.out.println("ret = " + ret);
    }

    public static int add(int x, int y) {
        System.out.println("调用方法中 x = " + x + " y = " + y);
        return x + y;
    }
}

// 执行结果
一次调用方法之前
调用方法中 x = 10 y = 20
第一次调用方法之后
ret = 30
第二次调用方法之前
调用方法中 x = 30 y = 50
第二次调用方法之后
ret = 80
```

代码示例: 计算 1! + 2! + 3! + 4! + 5!

```
public class TestMethod {
    public static void main(String[] args) {
        int sum = 0;
        for (int i = 1; i <= 5; i++) {
            sum += fac(i);
        }
        System.out.println("sum = " + sum);
    }

    public static int fac(int n) {
        System.out.println("计算 n 的阶乘中n! = " + n);
        int result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}

// 执行结果
计算 n 的阶乘中 n! = 1
计算 n 的阶乘中 n! = 2
计算 n 的阶乘中 n! = 3
计算 n 的阶乘中 n! = 4
计算 n 的阶乘中 n! = 5
sum = 153
```

使用方法, 避免使用二重循环, 让代码更简单清晰.

1.4 实参和形参的关系(重要)

方法的形参相当于数学函数中的自变量，比如：1 + 2 + 3 + ... + n的公式为**sum(n) =** $\frac{(1+n)*n}{2}$

Java中方法的形参就相当于sum函数中的自变量n，用来接收sum函数在调用时传递的值的。形参的名字可以随意取，对方法都没有任何影响，**形参只是方法在定义时需要借助的一个变量，用来保存方法在调用时传递过来的值。**

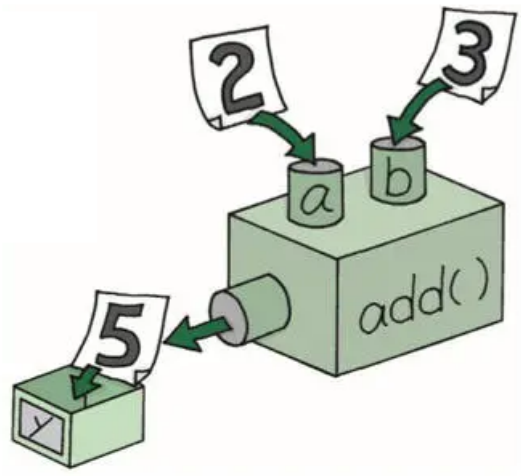
```
public static int getSum(int N){    // N是形参
    return (1+N)*N / 2;
}

getSum(10);     // 10是实参,在方法调用时，形参N用来保存10
getSum(100);    // 100是实参，在方法调用时，形参N用来保存100
```

再比如：

```
public static int add(int a, int b){
    return a + b;
}

add(2, 3);  // 2和3是实参，在调用时传给形参a和b
```



注意：**在Java中，实参的值永远都是拷贝到形参中，形参和实参本质是两个实体**

代码示例: 交换两个整型变量

```
public class TestMethod {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        swap(a, b);
        System.out.println("main: a = " + a + " b = " + b);
    }

    public static void swap(int x, int y) {
        int tmp = x;
        x = y;
        y = tmp;
        System.out.println("swap: x = " + x + " y = " + y);
    }
}

// 运行结果
swap: x = 20 y = 10
main: a = 10 b = 20
```

可以看到，在swap函数交换之后，形参x和y的值发生了改变，但是main方法中a和b还是交换之前的值，即没有交换成功。

【原因分析】

实参a和b是main方法中的两个变量，其空间在main方法的栈(一块特殊的内存空间)中，而形参x和y是swap方法中的两个变量，x和y的空间在swap方法运行时的栈中，因此：实参a和b 与 形参x和y是两个没有任何关联性的变量，**在swap方法调用时，只是将实参a和b中的值拷贝了一份传递给了形参x和y**，因此对形参x和y操作不会对实参a和b产生任何影响。

注意：对于**基础类型**来说, 形参相当于实参的拷贝. 即 **传值调用**

```
int a = 10;
int b = 20;

int x = a;
int y = b;

int tmp = x;
x = y;
y = tmp;
```

可以看到, 对 x 和 y 的修改, 不影响 a 和 b.

【解决办法】: 传引用类型参数 (例如数组来解决这个问题)

> 这个代码的运行过程, 后面学习数组的时候再详细解释.

```
public class TestMethod {
    public static void main(String[] args) {
        int[] arr = {10, 20};
        swap(arr);
        System.out.println("arr[0] = " + arr[0] + " arr[1] = " + arr[1]);
    }

    public static void swap(int[] arr) {
        int tmp = arr[0];
        arr[0] = arr[1];
        arr[1] = tmp;
    }
}

// 运行结果
arr[0] = 20 arr[1] = 10
```

1.5 没有返回值的方法

方法的返回值是可选的. 有些时候可以没有的，没有时返回值类型必须写成void

代码示例

```
class Test {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        print(a, b);
    }

    public static void print(int x, int y) {
        System.out.println("x = " + x + " y = " + y);
    }
}
```

另外, 如刚才的交换两个整数的方法, 就是没有返回值的.

2. 方法重载

2.1 为什么需要方法重载

```
public class TestMethod {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int ret = add(a, b);
        System.out.println("ret = " + ret);

        double a2 = 10.5;
        double b2 = 20.5;
        double ret2 = add(a2, b2);
        System.out.println("ret2 = " + ret2);
    }

    public static int add(int x, int y) {
        return x + y;
    }
}

// 编译出错
Test.java:13: 错误: 不兼容的类型: 从double转换到int可能会有损失
            double ret2 = add(a2, b2);
                          ^
```

由于参数类型不匹配, 所以不能直接使用现有的 add 方法.

一种比较简单粗暴的解决方法如下:

```
public class TestMethod {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;

        int ret = addInt(a, b);
```

```
        System.out.println("ret = " + ret);

        double a2 = 10.5;
        double b2 = 20.5;
        double ret2 = addDouble(a2, b2);
        System.out.println("ret2 = " + ret2);
    }

    public static int addInt(int x, int y) {
        return x + y;
    }

    public static double addDouble(double x, double y) {
        return x + y;
    }
}
```

上述代码确实可以解决问题，但不友好的地方是：需要提供许多不同的方法名，而取名字本来就是让人头疼的事情。那能否将所有的名字都给成`add`呢?

2.2 方法重载概念

在自然语言中，经常会出现“一词多义”的现象，比如：“好人”。

在自然语言中，一个词语如果有多重含义，那么就说该词语被重载了，具体代表什么含义需要结合具体的场景。

在Java中方法也是可以重载的。

在Java中，如果多个方法的名字相同，参数列表不同，则称该几种方法被重载了。

```
public class TestMethod {
    public static void main(String[] args) {
        add(1, 2);             // 调用add(int, int)
        add(1.5, 2.5);         // 调用add(double, double)
        add(1.5, 2.5, 3.5);    // 调用add(double, double, double)
    }

    public static int add(int x, int y) {
        return x + y;
```

```
    }

    public static double add(double x, double y) {
        return x + y;
    }

    public static double add(double x, double y, double z) {
        return x + y + z;
    }
}
```

注意：

1. 方法名必须相同
2. 参数列表必须不同(参数的个数不同、参数的类型不同、类型的次序必须不同)
3. 与返回值类型是否相同无关

```
// 注意：两个方法如果仅仅只是因为返回值类型不同，是不能构成重载的
public class TestMethod {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;
        int ret = add(a, b);
        System.out.println("ret = " + ret);
    }

    public static int add(int x, int y) {
        return x + y;
    }

    public static double add(int x, int y) {
        return x + y;
    }
}

// 编译出错
Test.java:13: 错误: 已在类 Test中定义了方法 add(int,int)
        public static double add(int x, int y) {
                            ^
1 个错误
```

4. 编译器在编译代码时，会对实参类型进行推演，根据推演的结果来确定调用哪个方法

2.3 方法签名

在同一个作用域中不能定义两个相同名称的标识符。比如：方法中不能定义两个名字一样的变量，那**为什么类中就可以定义方法名相同的方法呢?**

方法签名即：经过编译器编译修改过之后方法最终的名字。具体方式：**方法全路径名+参数列表+返回值类型，构成方法完整的名字。**

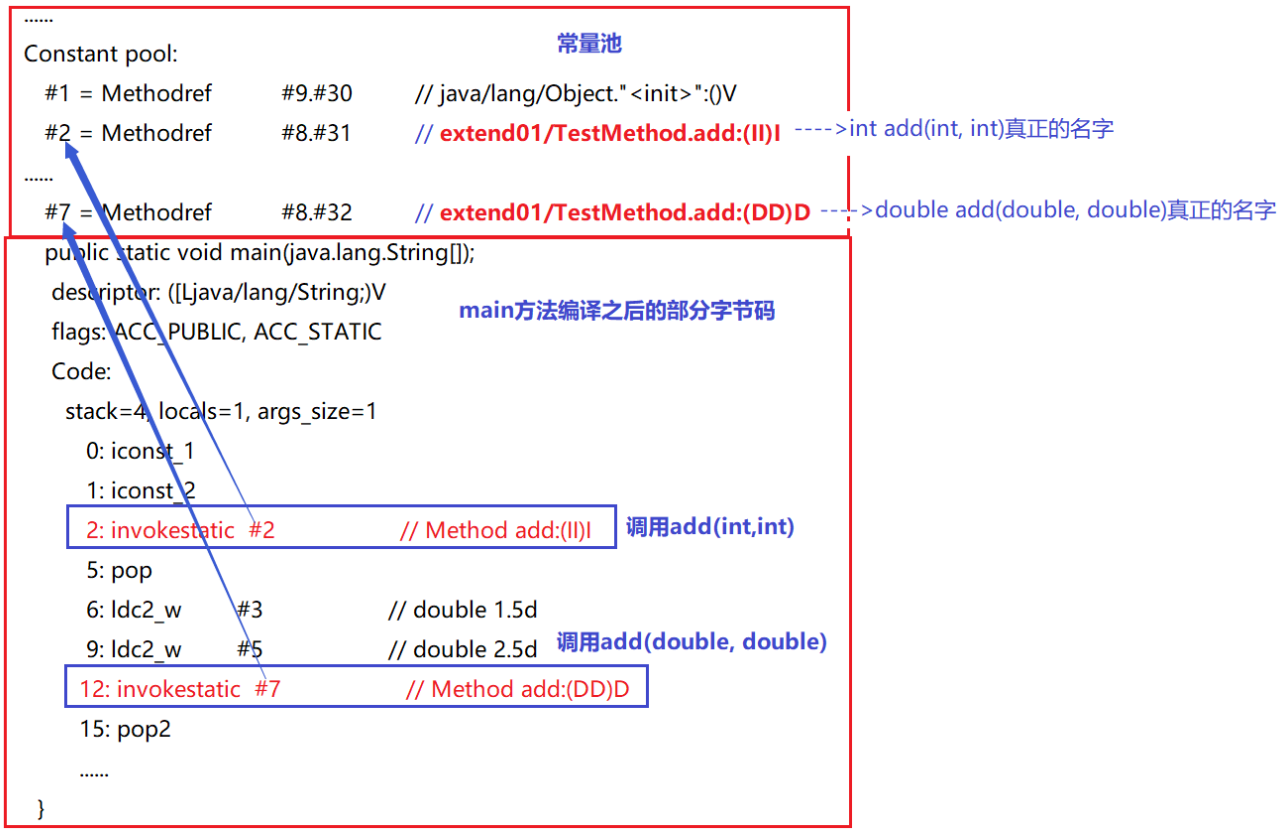
```java
public class TestMethod {
    public static int add(int x, int y){
        return x + y;
    }

    public static double add(double x, double y){
        return x + y;
    }

    public static void main(String[] args) {
        add(1,2);
        add(1.5, 2.5);
    }
}
```

上述代码经过编译之后，然后使用JDK自带的javap反汇编工具查看，具体操作：

1. 先对工程进行编译生成.class字节码文件
2. 在控制台中进入到要查看的.class所在的目录
3. 输入：javap -v 字节码文件名字即可



方法签名中的一些特殊符号说明：

特殊字符	数据类型
V	void
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
[	数组(以[开头，配合其他的特殊字符，表述对应数据类型的数组，几个[表述几维数组)
L	引用类型，以L开头，以;结尾，中间是引用类型的全类名

3. 递归

3.1 生活中的故事

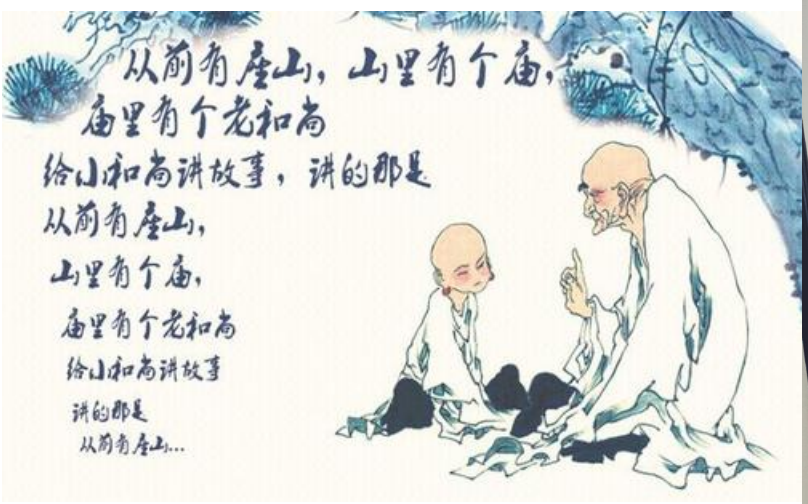
从前有坐山，山上有座庙，庙里有个老和尚给小和尚将故事，讲的就是：

"从前有座山，山上有座庙，庙里有个老和尚给小和尚讲故事，讲的就是：

"从前有座山，山上有座庙..."

"从前有座山....."

"



上面的两个故事有个共同的特征：**自身中又包含了自己**，该种思想在数学和编程中非常有用，因为有些时候，我们**遇到的问题直接并不好解决，但是发现将原问题拆分成其子问题之后，子问题与原问题有相同的解法，等子问题解决之后，原问题就迎刃而解了。**

3.2 递归的概念

一个方法在执行过程中调用自身, 就称为 "递归".

递归相当于数学上的 "数学归纳法", 有一个起始条件, 然后有一个递推公式.

> 例如, 我们求 N!
>
> 起始条件: N = 1 的时候, N! 为 1. 这个起始条件相当于递归的结束条件.
>
> 递归公式: 求 N! , 直接不好求, 可以把问题转换成 N! => N * (N-1)!

递归的必要条件：

1. 将原问题划分成其子问题，注意：子问题必须要与原问题的解法相同
2. 递归出口

代码示例: 递归求 N 的阶乘

```
public static void main(String[] args) {
    int n = 5;
    int ret = factor(n);
    System.out.println("ret = " + ret);
}

public static int factor(int n) {
    if (n == 1) {
        return 1;
    }
    return n * factor(n - 1); // factor 调用函数自身
}

// 执行结果
ret = 120
```

3.2 递归执行过程分析

递归的程序的执行过程不太容易理解, 要想理解清楚递归, 必须先理解清楚 "方法的执行过程", 尤其是 "方法执行结束之后, 回到调用位置继续往下执行".

代码示例: 递归求 N 的阶乘

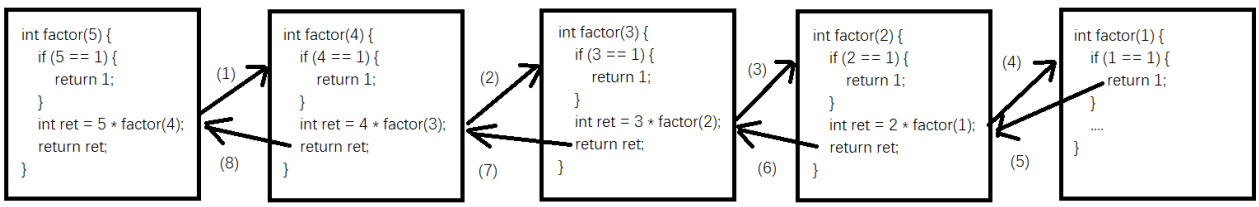
```
public static void main(String[] args) {
    int n = 5;
    int ret = factor(n);
    System.out.println("ret = " + ret);
}


public static int factor(int n) {
```

```
    System.out.println("函数开始, n = " + n);
    if (n == 1) {
        System.out.println("函数结束, n = 1 ret = 1");
        return 1;
    }
    int ret = n * factor(n - 1);
    System.out.println("函数结束, n = " + n + " ret = " + ret);
    return ret;
}

// 执行结果
函数开始, n = 5
函数开始, n = 4
函数开始, n = 3
函数开始, n = 2
函数开始, n = 1
函数结束, n = 1 ret = 1
函数结束, n = 2 ret = 2
函数结束, n = 3 ret = 6
函数结束, n = 4 ret = 24
函数结束, n = 5 ret = 120
ret = 120
```

执行过程图



程序按照序号中标识的 (1) -> (8) 的顺序执行.

> 关于 "调用栈"
>
> 方法调用的时候, 会有一个 "栈" 这样的内存空间描述当前的调用关系. 称为调用栈.
>
> 每一次的方法调用就称为一个 "栈帧", 每个栈帧中包含了这次调用的参数是哪些, 返回到哪里继续执行等信息.
>
> 后面我们借助 IDEA 很容易看到调用栈的内容.

3.3 递归练习

代码示例1 按顺序打印一个数字的每一位(例如 1234 打印出 1 2 3 4)

```
public static void print(int num) {
    if (num > 9) {
        print(num / 10);
    }
    System.out.println(num % 10);
}
```

代码示例2 递归求 1 + 2 + 3 + ... + 10

```
public static int sum(int num) {
    if (num == 1) {
        return 1;
    }
    return num + sum(num - 1);
}
```

代码示例3 写一个递归方法，输入一个非负整数，返回组成它的数字之和. 例如，输入 1729, 则应该返回 1+7+2+9，它的和是19

```
public static int sum(int num) {
    if (num < 10) {
        return num;
    }
    return num % 10 + sum(num / 10);
}
```

代码示例4 求斐波那契数列的第 N 项

斐波那契数列介绍: https://baike.sogou.com/v267267.htm?fromTitle=%E6%96%90%E6%B3%A2%E9%82%A3%E5%A5%91%E6%95%B0%E5%88%97

```
public static int fib(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}
```

当我们求 fib(40) 的时候发现, 程序执行速度极慢. 原因是进行了大量的重复运算.

```
class Test {
    public static int count = 0;   // 这个是类的成员变量. 后面会详细介绍到.
```

```
    public static void main(String[] args) {
        System.out.println(fib(40));
        System.out.println(count);
    }

    public static int fib(int n) {
        if (n == 1 || n == 2) {
            return 1;
        }
        if (n == 3) {
            count++;
        }
        return fib(n - 1) + fib(n - 2);
    }
}

// 执行结果
102334155
39088169    // fib(3) 重复执行了 3 千万次.
```

可以使用循环的方式来求斐波那契数列问题, 避免出现冗余运算.

```
public static int fib(int n) {
    int last2 = 1;
    int last1 = 1;
    int cur = 0;
    for (int i = 3; i <= n; i++) {
        cur = last1 + last2;
        last2 = last1;
        last1 = cur;
    }
    return cur;
}
```

此时程序的执行效率大大提高了.