

## 目录

1. java 中==和 equals 和 hashCode 的区别.....	2
2. int、char、long 各占多少字节数.....	5
3. int 与 integer 的区别.....	5
4. String、StringBuffer、StringBuilder 区别.....	6
5. Java 中 String 的了解（源码） .....	8
6. String 为什么要设计成不可变的？（源码） .....	10
7. string 转换成 integer 的方式及原理.....	11
8. utf-8 编码中的中文占几个字节；int 型几个字节？（源码） ..	19
9. Object 类的 equals 和 hashCode 方法重写，为什么？（源码） .	19
10. 静态代理和动态代理的区别，什么场景使用？（源码） .....	21

# 1. java 中==和 equals 和 hashCode 的区别

在 java 中：

- 1) ==是运算符，用于比较两个变量是否相等。
- 2) equals，是 Object 类的方法，用于比较两个对象是否相等，默认 Object 类的 equals 方法是比较两个对象的地址，跟==的结果一样。Object 的 equals 方法如下：

```
public boolean equals(Object obj) {  
  
    return (this == obj);  
  
}
```

- 3) hashCode 也是 Object 类的一个方法。返回一个离散的 int 型整数。在集合类操作中使用，为了提高查询速度。(HashMap, HashSet 等)

java 中的数据类型，可分为两类：

1. 基本数据类型，也称原始数据类型。  
byte, short, char, int, long, float, double, boolean 他们之间的比较，应用双等号（==），比较的是他们的值。

## 2. 复合数据类型(类)

当他们用（==）进行比较的时候，比较的是他们在内存中的存放

地址，所以，除非是同一个 new 出来的对象，他们的比较后的结果为 true，否则比较后结果为 false。 JAVA 当中所有的类都是继承于 Object 这个基类的，在 Object 中的基类中定义了一个 equals 的方法，这个方法的行为是比较对象的内存地址，但在一些类库当中这个方法被覆盖掉了，如 String, Integer, Date 在这些类当中 equals 有其自身的实现，而不再是比较类在堆内存中的存放地址了。

对于复合数据类型之间进行 equals 比较，在没有覆写 equals 方法的情况下，他们之间的比较还是基于他们在内存中的存放位置的地址值的，因为 Object 的 equals 方法也是用双等号 (==) 进行比较的，所以比较后的结果跟双等号 (==) 的结果相同。

1) 如果两个对象根据 equals() 方法比较是相等的，那么调用这两个对象中任意一个对象的 hashCode 方法都必须产生同样的整数结果。

2) 如果两个对象根据 equals() 方法比较是不相等的，那么调用这两个对象中任意一个对象的 hashCode 方法，则不一定要产生相同的整数结果

**从而在集合操作的时候有如下规则：**

将对象放入到集合中时，首先判断要放入对象的 hashCode 值与集合中的任意一个元素的 hashCode 值是否相等，

如果不相等直接将该对象放入集合中。

如果 hashCode 值相等，然后再通过 equals 方法判断要放入对象与集合中的任意一个对象是否相等，如果 equals 判断不相等，直接将该元素放入到集合中，否则不放入。

回过来说 get 的时候，HashMap 也先调 key.hashCode() 算出数组下标，然后看 equals 如果是 true 就是找到了，所以就涉及了 equals。

### 哈希表的存储原理：

当我们向哈希表插入一个 object 时，首先调用 hashCode() 方法获得该对象的哈希码，通过该哈希码直接定位 object 在哈希表中的位置。如果该位置没有对象，将 object 插入该位置，如果该位置有对象（可能有多个，通过链表实现），则调用 equals() 方法将这些对象与 object 比较，如果相等，则不需要保存 object，否则，将该对象插入到该链表中。

所以 equals() 相等，则 hashCode() 必须相等。

重写 equals() 方法之后，必须重写 hashCode() 方法。

### 重写 hashCode 注意：

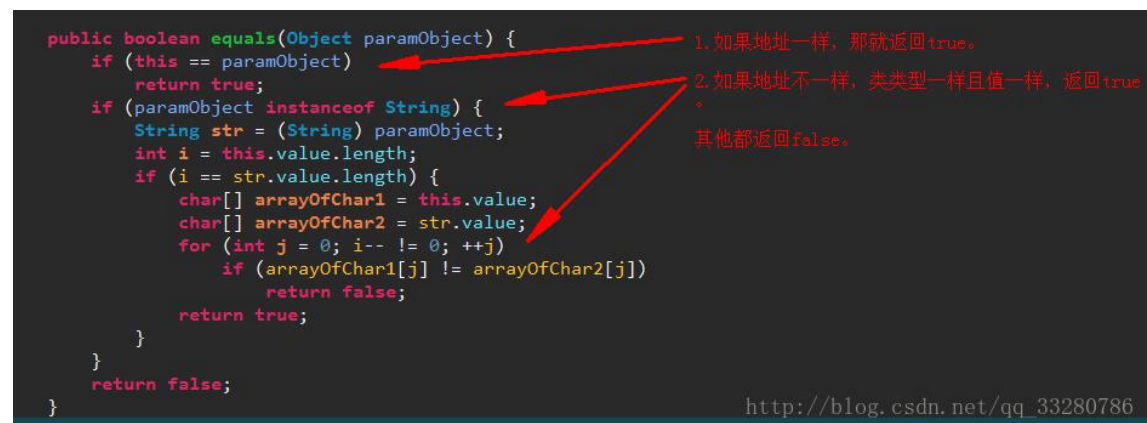
- 1) 如果重写 equals()，两个对象 equals() 方法判断相等，则对应的 hashCode 也是相等的 如果不等，则需要重写 hashCode 方法。
- 2) hashCode 不能简单到容易造成 Hash 冲突
- 3) hashCode 不能太复杂，以至于影响性能。

## String.equals()

而String已经重写了equals方法。提供了地址不相同的情况下，判断两个String对象的值是否一致的方法。

1. 判断类类型是否为String。如果是，则进入第二步，否则返回false。

2. 两个char数组同位置（下标）的值逐一对比，如果都相等，返回true，否则返回false。



```
public boolean equals(Object paramObject) {  
    if (this == paramObject) {  
        return true;  
    }  
    if (paramObject instanceof String) {  
        String str = (String) paramObject;  
        int i = this.value.length;  
        if (i == str.value.length) {  
            char[] arrayOfChar1 = this.value;  
            char[] arrayOfChar2 = str.value;  
            for (int j = 0; i-- != 0; ++j)  
                if (arrayOfChar1[j] != arrayOfChar2[j])  
                    return false;  
            return true;  
        }  
    }  
    return false;  
}
```

1. 如果地址一样，那就返回true。  
2. 如果地址不一样，类类型一样且值一样，返回true。  
其他都返回false。

[http://blog.csdn.net/qq\\_33280786](http://blog.csdn.net/qq_33280786)

## 2. int、char、long 各占多少字节数

4 字节、2 字节、8 字节

## 3. int 与 integer 的区别

1) 所占内存不同:

Integer 对象会占用更多的内存。Integer 是一个对象，需要存储对象的元数据。但是 int 是一个原始类型的数据，所以占用的空间更少。

## 2) 类型及初始值:

int 是基本类型，直接存数值，在类进行初始化时 int 类的变量初始为 0;而 integer 是对象(Integer 是 int 的封装类)，用一个引用指向这个对象，Integer 的变量则初始化为 null

int 和 Integer 都可以表示某一个数值，但 int 和 Integer 不能够互用，因为他们两种不同的数据类型;

# 4. String、StringBuffer、StringBuilder 区别

- 1) 都是 final 类，都不允许被继承;
- 2) String 类长度是不可变的，StringBuffer 和 StringBuilder 类长度是可以改变的;
- 3) StringBuffer 类是线程安全的，StringBuilder 不是线程安全的;

## String 类型和 StringBuffer 类型的主要性能区别:

- 1) String 是不可变的对象，因此每次在对 String 类进行改变的时候都会生成一个新的 string 对象，然后将指针指向新的

string 对象，所以经常要改变字符串长度的话不要使用 string，因为每次生成对象都会对系统性能产生影响，特别是当内存中引用的对象多了以后，JVM 的 GC 就会开始工作，性能就会降低；

2) 使用 StringBuffer 类时，每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用，所以多数情况下推荐使用 StringBuffer，特别是字符串对象经常要改变的情况；

在某些情况下，String 对象的字符串拼接其实是被 Java Compiler 编译成了 StringBuffer 对象的拼接，所以这些时候 String 对象的速度并不会比 StringBuffer 对象慢，

## StringBuilder

用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）

### 使用策略

1) 基本原则：

- ① 如果要操作少量的数据，用 String ；
- ② 单线程操作大量数据，用 StringBuilder ；
- ③ 多线程操作大量数据，用 StringBuffer。

2) 不要使用 String 类的”+”来进行频繁的拼接，因为那样的性能极差的，应该使用 StringBuffer 或 StringBuilder 类

3) StringBuilder 一般使用在方法内部来完成类似”+”功能，因为是线程不安全的，所以用完以后可以丢弃。StringBuffer 主要用在全局变量中。

相同情况下使用 `StringBuilder` 相比使用 `StringBuffer` 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

而在现实的模块化编程中，负责某一模块的程序员不一定能清晰地判断该模块是否会放入多线程的环境中运行，因此：除非确定系统的瓶颈是在 `StringBuffer` 上，并且确定你的模块不会运行在多线程模式下，才可以采用 `StringBuilder`；否则还是用 `StringBuffer`

## 5. Java 中 String 的了解（源码）

看完源码后总结

<https://www.jianshu.com/p/799c4459b808>

可能要手写出来

比如说 `String` 类被设计成 `final` 类型是出于哪些考虑？

在 Java 中，被 `final` 类型修饰的类不允许被其他类继承，被 `final` 修饰的变量赋值后不允许被修改

对于 `String` 类，官方有如下注释说明：



String 字符串是常量，其值在实例创建后就不能被修改，但字符串缓冲区支持可变的字符串，因为缓冲区里面的不可变字符串对象们可以被共享。（其实就是使对象的引用发生了改变）

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

...

}
```

可以看出 String 是 final 类型的，表示该类不能被其他类继承，同时该类实现了三个接口：`java.io.Serializable`、`Comparable<String>`、`CharSequence`

这是一个字符数组，并且是 final 类型，用于存储字符串内容。从 final 关键字可以看出，String 的内容一旦被初始化后，其不能被修改的。（

看到这里也许会有人疑惑，String 初始化以后好像可以被修改啊。比如找一个常见的例子：

```
String str = "hello" ; str = "hi"
```

其实这里的赋值并不是对 str 内容的修改，而是将 str 指向了新的字符串。

）

另外可以明确的一点:String 其实是基于字符数组 char[] 实现的。

```
/** 用于存放字符串的数组 */  
  
private final char value[];
```

下面再来看 String 其他属性:

- 1) 比如缓存字符串的 hash Code, 其默认值为 0:
- 2) 关于序列化 serialVersionUID:

因为 String 实现了 Serializable 接口, 所以支持序列化和反序列化支持。Java 的序列化机制是通过在运行时判断类的 serialVersionUID 来验证版本一致性的。

在进行反序列化时, JVM 会把传来的字节流中的 serialVersionUID 与本地相应实体(类)的 serialVersionUID 进行比较, 如果相同就认为是一致的, 可以进行反序列化, 否则就会出现序列化版本不一致的异常 (InvalidCastException)。

## 6. String 为什么要设计成不可变的? (源码)

- 1) 字符串常量池的需要

字符串常量池(String pool, String intern pool, String 保

留池) 是 Java 堆内存中一个特殊的存储区域, 当创建一个 String 对象时, 假如此字符串值已经存在于常量池中, 则不会创建一个新的对象, 而是引用已经存在的对象。

## 2) 允许 String 对象缓存 hashCode

Java 中 String 对象的哈希码被频繁地使用, 比如在 hashMap 等容器中。

字符串不变性保证了 hash 码的唯一性, 因此可以放心地进行缓存. 这也是一种性能优化手段, 意味着不必每次都去计算新的哈希码

## 3) 安全性

String 被许多的 Java 类(库)用来当做参数, 例如 网络连接地址 URL, 文件路径 path, 还有反射机制所需要的 String 参数等, 假若 String 不是固定不变的, 将会引起各种安全隐患

# 7. string 转换成 integer 的方式及原理

## 1) Integer.parseInt (String str) 方法

```
public static int parseInt(String s) throws NumberFormatException {  
  
    //内部默认调用 parseInt(String s, int radix)基数设置为 10  
  
    return parseInt(s,10);  
  
}
```

## 2) Integer.parseInt (String s, int radix) 方法

```
public static int parseInt(String s, int radix)

    throws NumberFormatException

{

    /*

    * WARNING: This method may be invoked early during VM initialization

    * before IntegerCache is initialized. Care must be taken to not use

    * the valueOf method.

    */

    //判断字符是否为 null

    if (s == null) {

        throw new NumberFormatException("s == null");

    }

    //基数是否小于最小基数

    if (radix < Character.MIN_RADIX) {

        throw new NumberFormatException("radix " + radix +

                                           " less than Character.MIN_RADIX");

    }

    //基数是否大于最大基数

    if (radix > Character.MAX_RADIX) {

        throw new NumberFormatException("radix " + radix +

                                           " greater than Character.MAX_RADIX");

    }

}
```

```
}

int result = 0;

//是否时负数

boolean negative = false;

//char 字符数组下标和长度

int i = 0, len = s.length();

//限制

int limit = -Integer.MAX_VALUE;

int multmin;

int digit;

//判断字符长度是否大于 0，否则抛出异常

if (len > 0) {

    //第一个字符是否是符号

    char firstChar = s.charAt(0);

    //根据 ascii 码表看出加号(43)和负号(45)对应的

    //十进制数小于'0'(48)的

    if (firstChar < '0') { // Possible leading "+" or "-"

        //是负号

        if (firstChar == '-') {

            //负号属性设置为 true

            negative = true;

        }

    }

}
```

```
        limit = Integer.MIN_VALUE;

    }

    //不是负号也不是加号则抛出异常

    else if (firstChar != '+')

        throw NumberFormatException.forInputString(s);

    //如果有符号（加号或者减号）且字符串长度为 1，则抛出异常

    if (len == 1) // Cannot have lone "+" or "-"

        throw NumberFormatException.forInputString(s);

    i++;

}

multmin = limit / radix;

while (i < len) {

    // Accumulating negatively avoids surprises near MAX_VALUE

    //返回指定基数中字符表示的数值。（此处是十进制数值）

    digit = Character.digit(s.charAt(i++),radix);

    //小于 0，则为非 radix 进制数

    if (digit < 0) {

        throw NumberFormatException.forInputString(s);

    }

    //这里是为了保证下面计算不会超出最大值

    if (result < multmin) {

        throw NumberFormatException.forInputString(s);

    }

}
```

```

    }

    result *= radix;

    if (result < limit + digit) {

        throw NumberFormatException.forInputString(s);

    }

    result -= digit;

}

} else {

    throw NumberFormatException.forInputString(s);

}

//根据上面得到的是否负数，返回相应的值

return negative ? result : -result;

}

```

### 3) Character.digit(char ch, int radix)方法

返回指定基数中字符表示的数值。

```

public static int digit(int codePoint, int radix) {

    //基数必须再最大和最小基数之间

    if (radix < MIN_RADIX || radix > MAX_RADIX) {

        return -1;

    }

}

```

```
if (codePoint < 128) {

    // Optimized for ASCII

    int result = -1;

    //字符在 0-9 字符之间

    if ('0' <= codePoint && codePoint <= '9') {

        result = codePoint - '0';

    }

    //字符在 a-z 之间

    else if ('a' <= codePoint && codePoint <= 'z') {

        result = 10 + (codePoint - 'a');

    }

    //字符在 A-Z 之间

    else if ('A' <= codePoint && codePoint <= 'Z') {

        result = 10 + (codePoint - 'A');

    }

    //通过判断 result 和基数大小，输出对应值

    //通过我们 parseInt 对应的基数值为 10，

    //所以，只能在第一个判断（字符在 0-9 字符之间）

    //中得到 result 值 否则后续程序会抛出异常

    return result < radix ? result : -1;

}

return digitImpl(codePoint, radix);
```



```
}
```

## 总结

- 1) `parseInt(String s)`--内部调用 `parseInt(s, 10)` (默认为 10 进制)
- 2) 正常判断 `null`, 进制范围, `length` 等
- 3) 判断第一个字符是否是符号位
- 4) 循环遍历确定每个字符的十进制值
- 5) 通过 `*=` 和 `-=` 进行计算拼接
- 6) 判断是否为负值 返回结果。

Integer 转化为 string

<https://www.jianshu.com/p/9b3cedfdb0d>

```
Integer a = 2;

private void test() {

    String s1 = a.toString(); //方式一

    String s2 = Integer.toString(a); //方式二
```

```
String s3 = String.valueOf(a); //方式三

}
```

方式一源码:

```
public String toString() {
    return toString(value);
}

public static String toString(int i) {
    if (i == Integer.MIN_VALUE)
        return "-2147483648";
    int size = (i < 0) ? stringSize(-i) + 1 : stringSize(i);
    char[] buf = new char[size];
    getChars(i, size, buf);
    return new String(buf, true);
}
```

可以看出 方式一最终调用的是方式二。

通过 `toString()` 方法, 可以把整数 (包括 0) 转化为字符串, 但是 `Integer` 如果是 `null` 的话, 就会报空指针异常。

方式三源码:

```
public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}

public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

可以看出 当 `Integer` 是 `null` 的时候, 返回的 `String` 是 字符串 "null" 而不是 `null`

如果想当 Integer 为 null 的时候也返回 null 重写一下 valueOf 方法就行了。

```
public static String valueOf(Object obj) {  
    return (obj == null) ? null : obj.toString();  
}
```

## 8. utf-8 编码中的中文占几个字节； int 型几个字节？（源码）

utf-8 的编码规则：

如果一个字节，最高位为 0，表示这是一个 ASCII 字符（00~7F）

如果一个字节，以 11 开头，连续的 1 的个数暗示这个字符的字节数

一个 utf8 数字占 1 个字节

一个 utf8 英文字母占 1 个字节

少数是汉字每个占用 3 个字节，多数占用 4 个字节。

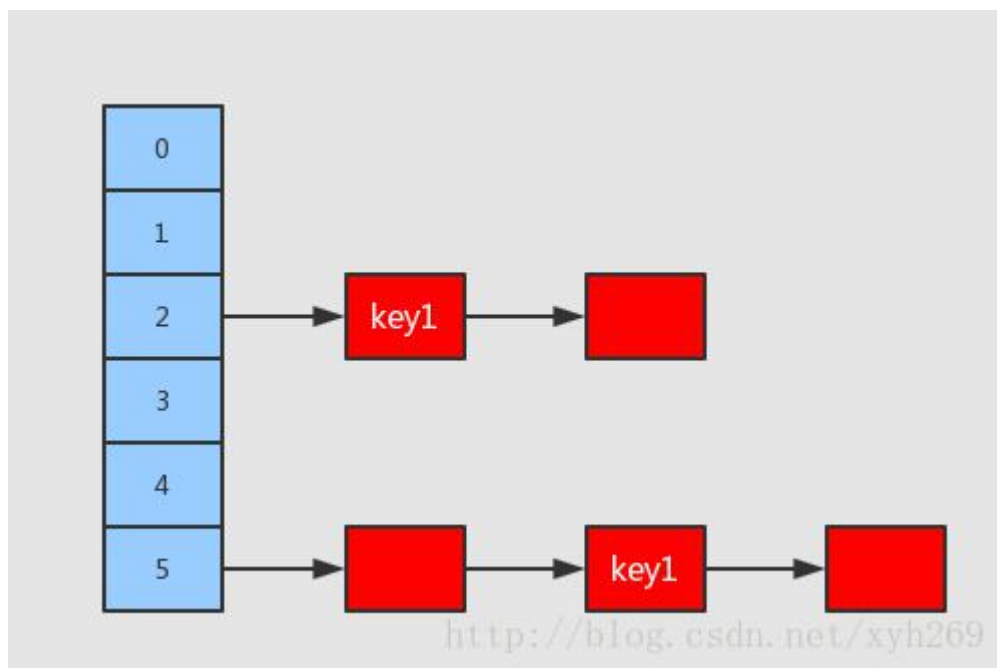
## 9. Object 类的 equal 和 hashCode 方法 重写，为什么？（源码）

往 HashMap 添加元素的时候，需要先定位到在数组的位置（hashCode 方法）。

如果只重写了 equals 方法，两个对象 equals 返回了 true，集合是不允许出现重复元素的，只能插入一个。

此时如果没有重写 hashCode 方法，那么就无法定位到同一个位置，集合还是会插入元素。这样集合中就出现了重复元素了。那么重写的 equals 方法就没有意义了。

如下图：



如果重写了 `hashCode` 方法，确保两个对象都能够定位到相同的位置，那么就可以遍历这条单向链表，使用 `equals` 方法判断两个对象是否相同，如果相同，那么就不插入了（`HashMap` 的实现仍然插入，但是覆盖掉旧的 `value`）。如果不相同，就插入到链表的头节点处。

## 10. 静态代理和动态代理的区别，什么场景使用？（源码）

静态代理类优缺点

优点：

代理使客户端不需要知道实现类是什么，怎么做的，而客户端只需知道代理即可（解耦合），对于如上的客户端代码，`newUserManagerImpl()`可以应用工厂将它隐藏，如上只是举个例子而已。

缺点：

1) 代理类和委托类实现了相同的接口，代理类通过委托类实现了相同的方法。这样就出现了大量的代码重复。如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。

2) 代理对象只服务于一种类型的对象，如果要服务多类型的对象。势必要为每一种对象都进行代理，静态代理在程序规模稍大时就无法胜任了。如上的代码是只为 `UserManager` 类的访问提供了代理，但是如果还要为其他类如 `Department` 类提供代理的话，就需要我们再次添加代理 `Department` 的代理类。

举例说明：代理可以对实现类进行统一的管理，如在调用具体实现类之前，需要打印日志等信息，这样我们只需要添加一个代理类，在代理类中添加打印日志的功能，然后调用实现类，这样就避免了修改具体实现类。满足我们所说的开闭原则。但是如果想让每个实现类都添加打印日志的功能的话，就需要添加多个代理类，以及代理类中各个方法都需要添加打印日志功能（如上的代理方法中删除，修改，以及查询都需要添加上打印日志的功能）

即静态代理类只能为特定的接口(Service)服务。如想要为多个接口服务则需要建立很多个代理类。

动态代理优点：

动态代理与静态代理相比较，最大的好处是接口中声明的所有方法都被转移到调用处理器一个集中的方法中处理（`InvocationHandler.invoke`）。这样，在接口方法数量比较多时，我们可以进行灵活处理，而不需要像静态代理那样每一个方法进行中转。而且动态代理的应用使我们的类职责更加单一，复用性更强

## 总结：

其实所谓代理，就是一个人或者一个机构代表另一个人或者另一个机构采取行动。在一些情况下，一个客户不想或者不能够直接引用一个对象，而代理对象可以在客户端和目标对象之前起到中介的作用。

代理对象就是把被代理对象包装一层，在其内部做一些额外的工作，比如用户需要上 facebook,而普通网络无法直接访问，网络代理帮助用户先 FQ，然后再访问 facebook。这就是代理的作用了。

纵观静态代理与动态代理，它们都能实现相同的功能，而我们看从静态代理到动态代理这个过程，我们会发现其实动态代理只是对类做了进一步抽象和封装，使其复用性和易用性得到进一步提升而这不仅仅符合了面向对象的设计理念，其中还有 AOP 的身影，这也提供给我们对类抽象的一种参考。关于动态代理与 AOP 的关系，个人觉得 AOP 是一种思想，而动态代理是一种 AOP 思想的实现！

**AOP（AspectOrientedProgramming）**：将日志记录，性能统计，安全控制，事务处理，异常处理等代码从业务逻辑代码中划分出来，通过对这些行为的分离，我们希望能将它们独立到非指导业务逻辑的方法中，进而改变这些行为的时候不影响业务逻辑的代码---解耦。

针对如上的示例解释：

我们来看上面的 `UserManagerImplProxy` 类，它的两个方法

`System.out.println("start-->addUser()")`和

`System.out.println("success-->addUser()")`，这是做核心动作之前和之后的两个截取段，正是这两个截取段，却是我们 AOP 的基础，在 OOP 里，

`System.out.println("start-->addUser()")`、核心动作、

`System.out.println("success-->addUser()")`这个三个动作在多个类里始终在一起，但他们所要完成的逻辑却是不同的，如 `System.out.println("start-->addUser()")`里做的可能是权限的判断，在所有类中它都是做权限判断，而在每个类里核心动作

却各不相同，`System.out.println("success-->addUser()")`可能做的是日志，在所有类里它都做日志。正是在所有的类里，核心代码之前的操作和核心代码之后的操作都做的是同样的逻辑，因此我们需要将它们提取出来，单独分析，设计和编码，这就是我们的 AOP 思想。一句话，AOP 只是在对 OOP 的基础上进行进一步抽象，使我们的类的职责更加单一。