32 反射机制

✧ 32.1 初识反射机制

Java 最大的特征就是**反射机制**(Reflection)，反射机制是 Java 的精髓。
所有技术实现的目标就是为了重用性。

正向操作：1) 导包；2) 类实例化对象；3) 调用对象方法。
反向操作：根据对象获取类的信息。

Object 类的 getClass()方法可以获得 Class 类对象

```
public final Class<?> getClass()
```

```
    Person p1 = new Person();
    Person p2 = new Person("hikari", 25);
    Class<? extends Person> cls = p1.getClass();
    System.out.println(cls); // class hikari.Person
    System.out.println(cls == p2.getClass()); // true
    Object obj = new Person();
    // 向上转型,getClass()返回还是Person类
    System.out.println(cls == obj.getClass()); // true
```

也就是 getClass()可以获取对象的根源。

✧ 32.2 Class 类对象的三种实例化模式

java.lang.Class 类定义：

```
public final class Class<T> implements java.io.Serializable, GenericDeclaration,
Type, AnnotatedElement
```

Class 类是反射机制的核心。Class 类实例化，可以采用三种方法。

① Object 类支持

Object 类可以根据实例化对象的 getClass()方法获取 Class 对象

```
Person p = new Person();
Class<? extends Person> cls = p.getClass();
System.out.println(cls); // class hikari.Person
System.out.println(cls.getName()); // hikari.Person
```

这种方式有个不是缺点的缺点：如果只想得到 Class 类对象，需要先实例化指定类对象才行。

② JVM 直接支持：类.class 形式实例化

```
Class<? extends Person> cls = Person.class;
System.out.println(cls.getName()); // hikari.Person
```

此种方式必须导入包。

③ Class 类支持：Class 类的静态方法 forName()

```
public static Class<?> forName(String className)
            throws ClassNotFoundException
```

```
try {
    Class<?> cls = Class.forName("hikari.Person");
    System.out.println(cls.getName()); // hikari.Person
} catch (ClassNotFoundException e) {
}
```

此方式最大特点的直接接收字符串为参数，而且不需要 import。
如果类不存在，抛出 ClassNotFoundException 异常。

✧ 32.3 反射实例化对象

Class 类提供一个反射实例化方法(代替关键字 new)：

```
@Deprecated(since="9")
public T newInstance()
    throws InstantiationException, IllegalAccessException
```

```
Class<?> cls = Class.forName("hikari.Person");
@SuppressWarnings("deprecation")
Object obj = cls.newInstance(); // 已过时
System.out.println(obj); // name: 匿名, age: 0
```

调用类的无参构造，隐含了关键字 new，直接使用字符串代替。

JDK1.9 后不建议使用，因为其只能调用无参构造，建议不使用 `clazz.newInstance()` 而是改为：`clazz.getDeclaredConstructor().newInstance()`

```
Object obj = cls.getDeclaredConstructor(String.class,
int.class).newInstance("hikari", 25); // 指定具体参数类型的构造方法
System.out.println(obj); // name: hikari, age: 25
```

✧ 32.4 反射与工厂设计模式

工厂设计模式最大特点是：客户端的程序类不直接对象实例化，只和接口发生关联，通过工程类获取接口实例化对象。

之前的工厂模式属于静态工厂模式，如果接口新增了一个子类，需要修改工程类，添加一个分支判断实例化新子类对象。

之前的代码：

```
public class Factory {
    private Factory() {}
    public static IMessage getInstance(String className) {
        if ("netmessage".equalsIgnoreCase(className)) {
            return new NetMessage();
        }
        if ("cloudmessage".equalsIgnoreCase(className)) {
            return new CloudMessage();
```

```
        }
        // 如果接口的子类越来越多，就需要不断地添加if语句...
        return null;
    }
}

        IMessage msg=Factory.getInstance("cloudmessage");
        msg.send(); // 来自云端的消息...
```

工厂模式有效地解决了子类与客户端耦合的问题。但是其核心是提供一个工厂类作为过渡；随着项目进行，接口的子类越来越多，工程类需要不断地修改。

最好的解决方法就不使用关键字 new，因为 new 一个对象需要一个明确的类存在。而反射只需要一个类明确的名称字符串。

使用反射指定为某个接口服务的工厂类：

```
public class Factory {
    private Factory() {}
    public static IMessage getInstance(String className) {
        IMessage instance = null;
        try {
            instance = (IMessage)
Class.forName(className).getDeclaredConstructor().newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return instance;
    }
}

        IMessage msg=Factory.getInstance("hikari.NetMessage");
        msg.send(); // 来自网络的消息...
```

利用反射机制实现工厂模式最大优势是：接口子类的扩充不影响工厂类的定义。

实际项目开发可能存在大量接口，而且都需要通过工厂类进行实例化，所以一个工厂类不应该只为一个接口服务，而是为所有接口服务。
因为客户端一定知道要使用的接口，而工程类不关心具体的接口，此时可以使用泛型工厂类。

基础工厂模式终极版本：

```
public class Factory {
    private Factory() {}
    /**
     * 获取接口实例化对象
```

```
     * @param className 接口的子类，字符串形式
     * @param clazz 接口类型
     * @return 返回指定接口实例化对象
     */
    @SuppressWarnings("unchecked")
    public static <T> T getInstance(String className, Class<T> clazz) {
        T instance = null;
        try {
            // 类型转换会警告，此警告不重要,压制一下吧...
            instance = (T)
Class.forName(className).getDeclaredConstructor().newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return instance;
    }
}


        IMessage msg = Factory.getInstance("hikari.NetMessage", IMessage.class);
        msg.send(); // 来自网络的消息...
        IService ser = Factory.getInstance("hikari.HouseService",
IService.class);
        ser.service(); // 为您打扫房间，顺便拿走沙发底下的私房钱！
```

✧ 32.5 反射与单例设计模式

单例模式核心：构造方法私有化，内部产生实例，通过静态方法获取实例对象，调用实例对象的方法。饿汉式定义时就实例化，没有线程同步问题；懒汉式因为定义和实例化分开，可能会造成线程不安全。

懒汉式单例的问题：

```
public class Singleton {
    private static Singleton instance = null;
    private Singleton() {
        System.out.println("【" + Thread.currentThread().getName() + "】*****实例
化Singleton对象*****");
    }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    public void show() {
        System.out.println("hello, hiakri!");
```

```
    }
}


        for (int i = 0; i < 3; i++) {
            new Thread(() -> {
                Singleton.getInstance().show();
            }, "线程" + (i + 1)).start();
        }
```

结果：

```
【线程 2】*****实例化 Singleton 对象*****
hello, hiakri!
【线程 1】*****实例化 Singleton 对象*****
hello, hiakri!
【线程 3】*****实例化 Singleton 对象*****
hello, hiakri!
```

3 个线程，Singleton 对象实例化了 3 次，也就不是单例模式了。造成问题的原因是代码不同步。最先想到的应该是使用 **synchronized** 关键字。

将 getInstance()方法设为同步：

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

但是这样效率会下降，因为实际上只有 instance 实例化部分需要同步。

```
public class Singleton {
    // 对象实例化时应该立刻与主内存对象同步，使用volatile
    private static volatile Singleton instance = null;
    private Singleton() { // ... }
    public static Singleton getInstance() {
        // 如果instance为null，所有线程通过第一层判断，有一个线程先实例化，剩余线程需
要进行第2层判断，防止再次实例化
        if (instance == null) {
            // 同步锁使用Class对象
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

面试题：请编写单例设计模式

1) [100%] 直接编写饿汉式单例设计模式，构造方法私有化；

2) [120%] Java 中哪里使用到单例设计模式：Runtime、Pattern、Spring 框架等；

3) [200%] 懒汉式单例模式的问题

✧ 32.6 反射获取类结构信息

一个类的结构信息包括类所在包名、父类的定义、父接口的定义。

```
public Package getPackage() // 获取包
public Class<? super T> getSuperclass() // 获取直接父类 Class 对象
public Class<?>[] getInterfaces() // 获取直接实现的父接口 Class 对象数组
```

示例：Test 类继承于 AbsClass 抽象类，实现接口 A 和 B

```
Class<?> cls = Test.class;
Package pac = cls.getPackage();
System.out.println(pac.getName()); // hikari
Class<?> parent = cls.getSuperclass(); // 直接父类
System.out.println(parent.getName()); // hikari.AbsClass
Class<?>[] inters = cls.getInterfaces();
for (Class<?> i : inters) { // 直接父接口
    System.out.println(i.getName()); // hikari.A, hikari.B
}
```

当获取了一个类 Class 对象后，就能得到其所有继承结构信息。

接口、抽象类、类都属于 Class 对象。一个 Class 对象就是一个字节码文件(*.class)对象。

✧ 32.7 反射获取构造方法 (Constructor)

获取构造方法：

```
public Constructor<?>[] getDeclaredConstructors() throws SecurityException
public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)
    throws NoSuchMethodException, SecurityException
public Constructor<?>[] getConstructors() throws SecurityException
public Constructor<T> getConstructor(Class<?>... parameterTypes)
    throws NoSuchMethodException, SecurityException
```

没有 Declared 表示获取 public 的构造方法(包括继承来的)；

有 Declared 表示获取所有声明的构造方法(不包括继承来的)。

Constructor<T>类继承关系：

```
java.lang.Object
  java.lang.reflect.AccessibleObject - JDK 1.2
    java.lang.reflect.Executable - JDK 1.8
      java.lang.reflect.Constructor<T> - JDK 1.1
```

newInstance()是 Constructor 重要的方法，此方法可以使用 Constructor 对象对应构造方法创建并初始化新的实例对象。

```
public T newInstance(Object ... initargs)
    throws InstantiationException, IllegalAccessException,
           IllegalArgumentException, InvocationTargetException
```

示例：

Student.class：

```
public class Student extends Person {
    private static final long serialVersionUID = 1L;
    private String major;
    public Student() {
        this("砖院");
    }
    public Student(String major) {
        this.major = major;
    }
    public Student(String name, int age, String major) {
        super(name, age);
        this.major = major;
    }
    public String getMajor() {
        return major;
    }
    public void setMajor(String major) {
        this.major = major;
    }

    private void show() { // 私有方法 }
    void print() { // default方法 }

    @Override
    public String toString() {
        return super.toString() + ", major: " + this.major;
    }
}
```

测试：反射获取 Student 类指定构造方法，实例化对象

```
Class<?> cls = Student.class;
// 有参构造
Constructor<?> constructor = cls.getDeclaredConstructor(String.class, int.class,
String.class);
Object obj = constructor.newInstance("张三", 20, "计院");
System.out.println(obj); // name: 张三, age: 20, major: 计院
// 无参构造
Constructor<?> con1 = cls.getDeclaredConstructor();
```

```
Object obj1 = con1.newInstance();
System.out.println(obj1); // name: 匿名, age: 0, major: 砖院
```

虽然可以调用有参构造，但实际开发，所有使用反射的类最好提供无参构造，如此，实例化可以达到统一性。

20180616

✧ 32.8 反射获取普通方法 (Method)
如果想通过反射调用方法，类中要提高实例化对象。

获取方法：

```
public Method getDeclaredMethod(String name, Class<?>... parameterTypes)
    throws NoSuchMethodException, SecurityException
public Method[] getDeclaredMethods() throws SecurityException
public Method getMethod(String name, Class<?>... parameterTypes)
    throws NoSuchMethodException, SecurityException
public Method[] getMethods() throws SecurityException
```

Declared 表示类中所有声明的方法(包括 private，不包括继承来的)；
没有 Declared 表示所有 public 的方法(包括继承来的)。

Method 和 Constructor 都是 Executable 的直接子类

```
public final class Method extends Executable
```

```
Class<?> cls = Student.class;
Method[] methods = cls.getMethods();
for (Method m : methods) {
    System.out.println(m);
}
System.out.println("--------------------------------");
Method[] methodDeclared = cls.getDeclaredMethods();
for (Method m : methodDeclared) {
    System.out.println(m);
}
```

Method 类的一些非重要方法：

```
public String getName()
public Class<?> getReturnType() // 返回值类型
public Class<?>[] getParameterTypes() // 参数类型
public Class<?>[] getExceptionTypes() // 异常类型
public int getModifiers() // 修饰符,返回 int 类型
```

java.lang.reflect.Modifier 类中定义了很多修饰符的常量。
Modifier 类的静态方法 toString()可以根据 int 返回修饰符的字符串形式

```
public static String toString(int mod)
```

示例：获取 Student 类私有方法 show()

```
Method me = cls.getDeclaredMethod("show");
int mod = me.getModifiers(); // 修饰符int类型
// 根据int值返回修饰符字符串形式
System.out.println(Modifier.toString(mod)); // private
```

上面的一些方法可以根据反射获取 Method 对象的结构。

Method 类最重要的一个方法是 **invoke()**

```
public Object invoke(Object obj, Object... args)
    throws IllegalAccessException, IllegalArgumentException,
       InvocationTargetException
```

Method 对象对应一个类的某个方法，调用该方法(非 static)需要有实例化对象 obj，args 是该方法参数，invoke()相当于调用 obj 对象的该方法。

```
Class<?> cls = Class.forName("hikari.Student");
String value = "砖院";
// 要保存属性或调用方法都需要有实例化对象，使用反射实例化
Object obj = cls.getDeclaredConstructor().newInstance();
String methodName = "setMajor";
Method method = cls.getDeclaredMethod(methodName, String.class);
// 等价于: Student对象.setMajor(value);
method.invoke(obj, value);
System.out.println(obj); // name: 匿名, age: 0, major: 砖院
```

✧ 32.9 反射获取成员属性 (Field)

同样有 4 个方法：

```
public Field getDeclaredField(String name)
    throws NoSuchFieldException, SecurityException
public Field[] getDeclaredFields() throws SecurityException
public Field getField(String name)
    throws NoSuchFieldException, SecurityException
public Field[] getFields() throws SecurityException
```

示例：

Person 类定义常量：

```
public static final String COUNTRY="China";
```

测试：

```
Class<?> cls = Class.forName("hikari.Student");
Field[] decFields = cls.getDeclaredFields();
for (Field f : decFields) {
    System.out.println(f.getName()); // serialVersionUID, major
}
```

```
System.out.println("--------------------");
Field[] fields = cls.getFields();
for (Field f : fields) {
    System.out.println(f.getName()); // COUNTRY
}
```

Field 类定义：

```
public final class Field extends AccessibleObject implements Member
```

Filed 类重要的 3 个方法：

```
public void set(Object obj, Object value) // 设置属性值
       throws IllegalArgumentException, IllegalAccessException
public Object get(Object obj) // 获取属性值
       throws IllegalArgumentException, IllegalAccessException
public void setAccessible(boolean flag) // 解除封装 (设置可以访问)
```

Field 覆写了 AccessibleObject 类的 setAccessible()方法。
Constructor、Method、Field 都有 setAccessible()方法。

```
// 获取指定类的Class对象
Class<?> cls = Class.forName("hikari.Student");
// 获取Constructor对象, 实例化指定类对象
Object obj = cls.getDeclaredConstructor().newInstance();
// 获取成员对象,major为私有属性
Field majorField = cls.getDeclaredField("major");
majorField.setAccessible(true); // 设置属性可访问
majorField.set(obj, "码农");
System.out.println(majorField.get(obj)); // 码农
```

实际开发很少直接操作 Filed，一般还是通过 setter 和 getter 完成。不建议使用 setAccessible()打破封装机制。

Field 类在实际开发只有一个最常用方法 **getType**()：

```
public Class<?> getType() // 获取属性类型
```

```
// 获取完整类名称, 包.类名
System.out.println(majorField.getType().getName()); // java.lang.String
// 获取类名称
System.out.println(majorField.getType().getSimpleName()); // String
```

实际开发使用反射时，往往使用 Field 和 Method 实现 setter 方法调用。

✧ 32.10 Unsafe 工具类

sun.misc.Unsafe 类利用反射获取对象，直接使用底层 C++代替 JVM 执行，绕开 JVM 的对象管理机制，无法使用 JVM 内存管理和垃圾回收。

```
private Unsafe() {} // 构造方法私有化
private static final Unsafe theUnsafe = new Unsafe(); // 私有静态常量 Unsafe 实例
```

虽然 Unsafe 提供静态方法 getUnsafe()返回 Unsafe 实例，但是调用会抛出异常 SecurityException，需要通过反射机制获取。

示例：Unsafe 绕过 JVM 直接实例化对象

```
class Singleton {
    // 构造方法私有化，没有提供静态方法获取实例化对象
    private Singleton() {
        System.out.println("Singleton构造方法");
    }
    public void show() {
        System.out.println("hello, hikari!");
    }
}


        Field field = Unsafe.class.getDeclaredField("theUnsafe");
        field.setAccessible(true);
        Unsafe unsafe = (Unsafe) field.get(null); // 获取static属性不需要传入实例
        // 利用Unsafe绕过JVM管理机制，可以在没有实例化情况获取Singleton对象
        Singleton s = (Singleton) unsafe.allocateInstance(Singleton.class);
        s.show(); // hello, hikari! (没有调用构造方法)
```

Unsafe 不受 JVM 管理，如果不是必须的话不建议使用。

✧ 32.11 综合案例：反射与简单 Java 类

简单 Java 类主要由属性组成，并提供相应的 setter 和 getter 方法，最大特点是通过对象保存属性。

对于简单 Java 类实例化并设置属性过程中，设置数据部分(setter)是最麻烦的，如果一个类有 50 个属性，需要调用 50 次 setter 方法。如果此时使用构造方法，需要按照顺序填写 50 个属性值，也容易混淆。实际开发，简单 Java 类个数很多，这样的话，属性赋值的时候代码重复性会非常高。

此时需要使用反射机制。反射机制最大的特征是可以使用 Object 类直接操作，还可以操作属性和方法，这样可以实现相同功能类的重复操作。

① 单级属性设置

假设 Employee 类只有两个字符串类型属性 name 和 job：

```
public class Employee {
    private String name;
    private String job;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
```

```
    }
    public String getJob() {
        return job;
    }
    public void setJob(String job) {
        this.job = job;
    }
    @Override
    public String toString() {
        return "name: " + this.name + ", job: " + this.job;
    }
}
```

可以使用结构化字符串形式描述一个对象，如"属性:值|属性:值"的格式，此时需要有一个类(如 InstanceFactory 类)接收字符串和指定 Class 对象，经过处理后返回设置好属性的指定类型的实例化对象。

InstanceFactory.class：

```
public class InstanceFactory {
    private InstanceFactory() {}
    /**
     * 根据传入字符串结构"属性:值|属性:值"和Class对象实例化该类对象
     * @param cls  进行反射实例化的Class对象
     * @param value  属性和值的结构化字符串
     * @return 设置好属性的指定类对象
     */
    @SuppressWarnings("unchecked")
    public static <T> T create(Class<?> cls, String value) {
        try {
            // 实例化对象，需要有无参构造
            Object obj = cls.getDeclaredConstructor().newInstance();
            BeanUtil.setValue(obj, value); // 通过反射设置对象属性
            return (T) obj; // 类型转换
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

给 obj 对象通过字符串设置属性的操作交给 BeanUtil 工具类完成。

BeanUtil.class：

```
public class BeanUtil {
    private BeanUtil() {}
```

```
    public static void setValue(Object obj, String value) {
        String[] ret = value.split("\\|"); // 按照|进行每组属性的拆分
        for (String i : ret) {
            String[] tmp = i.split(":"); // 按照:进行属性名和值的拆分
            String attr = tmp[0];
            String val = tmp[1];
            // 拼接setter方法的名字
            String methodName = "set" + StringUtil.title(attr);
            try {
                // 根据属性名获取Field对象
                Field field = obj.getClass().getDeclaredField(attr);
                // 根据方法名和参数类型获取Method对象
                Method method = obj.getClass().getDeclaredMethod(methodName,
field.getType());
                // 调用setter方法设置内容
                method.invoke(obj, val);
            } catch (Exception e) {} // 没有对应属性,什么也不做
        }
    }
}
```

因为 setter 方法的格式是 set+属性名首字母大写，所以定义工具类 StringUtil 实现首字母大写的方法。

StringUtil.class：

```
public class StringUtil {
    private StringUtil() {}
    public static String title(String s) {
        // 字符串首字母大写，后面不管大小写，
        if (s == null || "".equals(s)) { // null或空字符串不处理直接返回
            return s;
        }
        if (s.length() == 1) {
            return s.toUpperCase();
        }
        return s.substring(0, 1).toUpperCase() + s.substring(1);
    }
}
```

测试：

```
        String value = "id:5|name:hikari|job:搬砖";
        Employee e = InstanceFactory.create(Employee.class, value);
        System.out.println(e); // name: hikari, job: 搬砖
```

此处字符串有 id 字段，但是 Employee 类没有此属性，直接被忽略了。

② 设置多种数据类型

上面实现了只有字符串类型的属性设置。实际开发简单 Java 类属性类型一般有：long、int、double、String、Date。

Employee 类添加 int id 和 Date hiredDate 字段：

```
public class Employee {
    // ...
    private int id;
    private Date hiredDate;
    // setter和getter略...
    private String parseHiredDate() { // Date对象转为字符串
        return new SimpleDateFormat("yyyy-MM-dd").format(this.hiredDate);
    }
    @Override
    public String toString() {
        return "id: " + this.id + ", name: " + this.name + ", job: " + this.job +
", hired date: " + this.parseHiredDate();
    }
}
```

因为传入的字符串经拆分后得到的每个键值对 attr 和 val 都是字符串，此时需要先通过 filed.getType()获取属性类型，再将属性值 val 转换成指定类型。

BeanUtil 类添加一个属性类型转换操作：

```
/**
 * 实现属性类型转换
 * @param type  属性类型，通过Field获取
 * @param value  属性值，传入字符串，需要转为指定类型
 * @return 转换后的数据
 */
private static Object convertAttrValue(String type, String value) {
    if ("int".equals(type) || "java.lang.int".equals(type)) {
        return Integer.parseInt(value);
    }
    if ("long".equals(type) || "java.lang.long".equals(type)) {
        return Long.parseLong(value);
    }
    if ("double".equals(type) || "java.lang.double".equals(type)) {
        return Double.parseDouble(value);
    }
    if ("java.util.Date".equals(type)) {
        // 日期判断非严格...
        String re = "(19\\d\\d|20\\d\\d)-(0?\\d|11|12)-(0?\\d|[12]\\d|30|31)";
        if (value.matches(re)) {
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
```

```
            try {
                return sdf.parse(value);
            } catch (ParseException e) {
                return new Date(); // 转换失败或不匹配都设置当前时间
            }
        } else {
            return new Date();
        }
    }
    // 字符串或其他类型什么也不做，直接返回
    return value;
}
```

修改 BeanUtil 类的 setValue()方法

```
public static void setValue(Object obj, String value) {
    String[] ret = value.split("\\|"); // 按照|进行每组属性的拆分
    for (String i : ret) {
        String[] tmp = i.split(":"); // 按照:进行属性名和值的拆分
        String attr = StringUtil.lowerCamel(tmp[0]); // 改为小驼峰式
        String val = tmp[1];
        // 拼接setter方法的名字
        String methodName = "set" + StringUtil.title(attr);
        try {
            // 根据属性名获取Field对象
            Field field = obj.getClass().getDeclaredField(attr);
            // 根据方法名和参数类型获取Method对象
            Method method = obj.getClass().getDeclaredMethod(methodName,
field.getType());
            // 将属性值转为指定类型
            Object conertVal = convertAttrValue(field.getType().getName(), val);
            // 调用setter方法设置内容
            method.invoke(obj, conertVal);
        } catch (Exception e) {} // 没有对应属性,什么也不做
    }
}
```

StringUtil.class 的 lowerCamel()方法

```
public class StringUtil {
    // ...
    public static String lowerCamel(String s) {
        // 将几个单词变为小驼峰式,如a secret base-->aSecretBase
        if (s == null || "".equals(s) || !s.contains(" ")) {
            return s;
        }
```

```
        String[] tmp = s.split(" ");
        StringBuffer sb = new StringBuffer(tmp[0]);
        for (int i = 1; i < tmp.length; i++) {
            sb.append(title(tmp[i]));
        }
        return sb.toString();
    }
}
```

测试：

```
String value = "name:hikari|job:搬砖|hired date:2018-03-15|id:5|salary:1234.56";
Employee e = InstanceFactory.create(Employee.class, value);
System.out.println(e); // id: 5, name: hikari, job: 搬砖, hired date: 2018-03-15
```

20180617

③ 级联对象实例化

比如有 Employee、Department、Company 三个类，一个员工属于一个部门，一个部分属于一个公司。

```
public class Company {
    private String name;
    private Date createDate;
}
public class Department {
    private String name;
    private Company company;
}
public class Employee {
    private int id;
    private String name;
    private String job;
    private Date hiredDate;
    private Department dept;
}
```

此时希望通过 Employee 操作，使用.作为级联关系操作：

如：dept.name: abc 相当于 Employee 对象.getDept().setName("abc")

dept.company:xyz 相当于 Employee 对象.getDept().getCompany().setName("xyz")

要实现此功能，需要通过级联的配置自动实现类中属性实例化。

BeanUtil.class 修改：

```
public class BeanUtil {
    private static final String KEYSEP = "\\|";
    private static final String KEYVALSEP = ":";
```

```
    private BeanUtil() {}

    public static void setValue(Object obj, String value) {
        String[] ret = value.split(KEYSEP); // 按照|进行每组属性的拆分
        for (String i : ret) {
            String[] tmp = i.split(KEYVALSEP); // 按照:进行属性名和值的拆分
            String attr = StringUtil.lowerCamel(tmp[0].trim());
            String val = tmp[1].trim(); // 去除空格
            try {
                if (attr.contains(".")) { // 多级配置
                    setCascade(obj, attr, val);
                } else {
                    setAttrVal(obj, attr, val);
                }
            } catch (Exception e) {} // 没有对应属性,什么也不做
        }
    }

    private static void setCascade(Object obj, String attr, String val) throws
Exception { // 级联对象实例化
        String[] tmp = attr.split("\\.");
        Object cur = obj;
        // 最后一个是属性名称, 不在实例化处理范围
        for (int x = 0; x < tmp.length - 1; x++) {
            // 获取getter方法
            Method getMethod = cur.getClass().getDeclaredMethod("get" +
StringUtil.title(tmp[x]));
            // 获取下一级对象
            Object tmpObj = getMethod.invoke(cur);
            if (tmpObj == null) { // 该对象没有实例化
                Field field = cur.getClass().getDeclaredField(tmp[x]);
                // 获取setter方法
                Method method = cur.getClass().getDeclaredMethod("set" +
StringUtil.title(tmp[x]), field.getType());
                // 下一级的实例化对象
                Object nextObj =
field.getType().getDeclaredConstructor().newInstance();
                // 设置属性为新实例化对象
                method.invoke(cur, nextObj);
                cur = nextObj; // 相当于链表工作指针后移
            } else {
                cur = tmpObj;
            }
        }
```

```
        // 最后一个是属性，设置方法与之前一样
        setAttrVal(cur, tmp[tmp.length - 1], val);
    }

    private static void setAttrVal(Object obj, String attr, String val) throws
Exception {
        // 根据属性名获取Field对象
        Field field = obj.getClass().getDeclaredField(attr);
        // 拼接setter方法的名字
        String methodName = "set" + StringUtil.title(attr);
        // 根据方法名和参数类型获取Method对象
        Method method = obj.getClass().getDeclaredMethod(methodName,
field.getType());
        // 将属性值转为指定类型
        Object conertVal = convertAttrValue(field.getType().getName(), val);
        // 调用setter方法设置内容
        method.invoke(obj, conertVal);
    }
```

测试：

```
String value = "name : hikari | job : 搬砖 | hired date : 2018-03-15 | id : 5 |
salary : 1234.56 | dept.name : 搬砖部 | dept.company.name : 皇家搬砖株式会社";
Employee e = InstanceFactory.create(Employee.class, value);
System.out.println(e); // id: 5, name: hikari, job: 搬砖, hired date: 2018-03-15
System.out.println(e.getDept().getName()); // 搬砖部
System.out.println(e.getDept().getCompany().getName()); //皇家搬砖株式会社
```

✧ 32.12 ClassLoader 类加载器

CLASSPATH 属性作用是定义 JVM 进程启动时类加载的路径。JVM 根据 ClassLoader 加载指定路径的类。

Class 类的 getClassLoader()方法可以获取 ClassLoader 对象

```
public ClassLoader getClassLoader()
```

抽象类 ClassLoader 的 getParent()方法可以获取父类的 ClassLoader 对象

```
public final ClassLoader getParent()
```

```
Class<Person> cls = Person.class;
System.out.println(cls.getClassLoader());
// jdk.internal.loader.ClassLoaders$AppClassLoader@6659c656
System.out.println(cls.getClassLoader().getParent());
// jdk.internal.loader.ClassLoaders$PlatformClassLoader@299a06ac
System.out.println(cls.getClassLoader().getParent().getParent()); // null
```

JDK 1.9 后提供 PlatformClassLoader 类加载器。JDK1.8 及之前提供的加载器为 ExtClassLoader，JDK 安装目录提供一个 ext 目录，可以将.jar 文件复制到此目录，就可以直接执行，但是这样不安全，JDK1.9 后废除了。为了与系统类加载器和

应用类加载器之间保持设计平衡，提供平台类加载器。

可以自定义类加载器，其加载顺序是在所有系统类加载器的最后。系统类加载器是根据 CLASSPATH 路径进行加载，自定义类加载器可以由开发者任意指派类的加载位置。利用类加载器可以实现类的反射加载处理。

示例：

1) 将 hikari.Person.java 在 D:/a/目录编译，并且不打包：

```
javac -encoding utf-8 Person.java
```

所以这个类无法通过 CLASSPATH 正常加载。

2) 自定义类加载器，继承于 ClassLoader

ClassLoader 类的 **defineClass**()方法可以根据读取的.class 文件的字节数组和类名称获取该字节码文件的 Class 对象。

```
protected final Class<?> defineClass(String name, byte[] b, int off, int len)
    throws ClassFormatError
```

```
public class MyClassLoader extends ClassLoader {
    private static final String CLASS_PATH = "D:/a/Person.class";
    /**
     * 进行制定类的加载
     * @param className 类的完整名称: 包.类名
     * @return 返回指定类的Class对象
     * @throws Exception 如果.class文件不存在, 无法加载
     */
    public Class<?> loadData(String className) throws Exception {
        // 通过文件进行类的加载
        InputStream in = new FileInputStream(CLASS_PATH);
        // 读取全部数据到字节数组(偷个懒, 此方法慎用)
        byte[] data = in.readAllBytes();
        try {
            return super.defineClass(className, data, 0, data.length);
        } finally {
            in.close();
        }
    }
}
```

3) 测试

```
        MyClassLoader clsloader = new MyClassLoader();
        Class<?> cls = clsloader.loadData("hikari.Person");
        Object obj = cls.getDeclaredConstructor().newInstance();
        Method method = cls.getDeclaredMethod("setName", String.class);
        method.invoke(obj, "张三");
```

```
        method = cls.getDeclaredMethod("setAge", int.class);
        method.invoke(obj, 30);
        System.out.println(obj); // name: 张三, age: 30
```

在网络程序开发时，可以通过一个远程服务器确定类的功能。
比如每当服务端的程序类更改，网络类加载器都可以将改变及时更新到客户端。

如果自定义类 java.lang.String，并使用自定义类加载器加载处理，这个类不会被加载。Java 对类加载器提供双亲加载机制，如果开发者定义的类与系统类名称相同，为了保证系统安全，不会被加载。

✧ 32.13 反射与代理设计模式

代理设计模式的核心是有真实业务类和代理业务类，且代理类完成操作更多。

① 传统代理模式
传统代理模式的弊端是必须基于接口设计，首先要定义核心接口。
之前的代理设计模式客户端与子类产生耦合问题，需要引入工厂模式。
而且这种代理属于静态代理设计，一个代理类只为一个接口服务。如果有 3000 个业务接口，需要有 3000 个代理类，而且结构相似，代码重复性高。

② 动态代理设计模式
java.lang.reflect.InvocationHandler 接口规定了代理方法的执行。
InvocationHandler 接口定义

```
public interface InvocationHandler {
    /**
     * 代理方法调用，代理主题类最终执行方法
     * @param proxy 要代理的对象
     * @param method 要执行的接口方法 Method 对象
     * @param args 方法参数
     * @return 方法的返回值
     * @throws Throwable 方法调用出现的错误继续向上抛出
     */
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

自定义代理类实现 InvocationHandler 接口，需要覆写 invoke()方法。

动态代理设计时的动态对象是由 JVM 底层创建，主要依靠 java.lang.reflect.Proxy 类，此类有一个核心方法 **newProxyInstance()**：

```
/**
 * @param loader 获取当前真实主题类的 ClassLoader
 * @param interfaces 真实主题类的接口，因为代理是围绕接口进行的
 * @param h 代理处理的方法
 * @return 代理实例
```

```
 */
public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,
InvocationHandler h)
```

该方法大量使用底层机制进行代理对象的动态创建。

IMessage.class

```
public interface IMessage {
    void send();
}
```

Message.class：真实业务

```
public class Message implements IMessage {
    @Override
    public void send() {
        System.out.println("hello, hikari!");
    }
}
```

MyProxy.class：自定义动态代理类

```
public class MyProxy implements InvocationHandler {
    private Object target; // 保存真实业务对象

    public Object bind(Object target) {
        this.target = target;
        // 获取类加载器和接口对象数组
        ClassLoader loader = target.getClass().getClassLoader();
        Class<?>[] inters = target.getClass().getInterfaces();
        // 代理类不代表具体接口，需要依赖于类加载器与接口进行代理对象的伪造
        return Proxy.newProxyInstance(loader, inters, this);
    }

    public boolean connect() {
        System.out.println("【正在连接】");
        return true;
    }

    public void close() {
        System.out.println("【已关闭】");
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable { // 代理不依赖具体接口
        Object obj = null;
        if (this.connect()) {
```

```
            obj = method.invoke(this.target, args); // 调用方法
            this.close();
        }
        return obj;
    }
}
```

MyProxyFactory.class：代理工厂类

```
public class MyProxyFactory {
    private MyProxyFactory() {}

    @SuppressWarnings("unchecked")
    public static <T> T create(Class<?> cls) {
        try {
            // 反射实例化对象
            Object obj = cls.getDeclaredConstructor().newInstance();
            // 返回代理对象
            return (T) new MyProxy().bind(obj);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

测试：

```
        IMessage msg = MyProxyFactory.create(Message.class);
        msg.send();
```

结果：

```
【正在连接】
hello, hikari!
【已关闭】
```

接口代理对象 msg 是 Proxy 的子类，实现了 IMessage 接口。调用 msg.func()相当于调用 MyProxy 对象的 invoke()方法，此时 invoke()参数 proxy 指向代理对象 msg，method 指向 func()方法 Method 对象，args 为 func()方法参数。在 invoke()方法中调用了真实业务对象 this.target.func()。

如果在 MyProxy 类的 invoke()方法试图调用 proxy 对象的方法，会出现栈溢出的错误，因为调用 proxy 对象的方法，还会继续调用 MyProxy 类的 invoke()方法，如此形成无限递归。

20180618

✧ 32.14 CGLIB 实现动态代理设计

Java 官方需要基于接口实现代理设计。但是利用第三方程序包 CGLIB 可以实现

基于类的代理设计模式。

1) Eclispe 安装第三方开发包
项目右击→Properties→Java Build Path→Libraries→Classpath→Add External JARs
找到 cglib-nodep-3.2.6.jar 并添加

2) 编写程序类，不实现任何接口

```
public class Message {
    public void send() {
        System.out.println("hello, hikari!");
    }
}
```

3) 使用 CGLIB 编写代理类，需要通过 CGLIB 生成代理对象

```
public class MyProxy implements MethodInterceptor { // 拦截器
    private Object target; // 保存真实业务对象
    public MyProxy(Object target) {
        this.target = target;
    }

    @Override
    public Object intercept(Object proxy, Method method, Object[] args,
MethodProxy mp) throws Throwable {
        Object obj = null;
        if (this.connect()) {
            obj = method.invoke(this.target, args);
            this.close();
        }
        return obj;
    }

    public boolean connect() {
        System.out.println("【正在连接】");
        return true;
    }

    public void close() {
        System.out.println("【已关闭】");
    }
}
```

4) 需要进行一系列 CGLIB 处理才能创建代理类对象

```
        Message msg = new Message(); // 真实主体对象
        Enhancer enhancer = new Enhancer(); // 负责代理操作的类，类似于Proxy类
```

```
        enhancer.setSuperclass(msg.getClass()); // 假定一个父类
        enhancer.setCallback(new MyProxy(msg));
        Message msgProxy = (Message) enhancer.create();// 创建代理对象
        msgProxy.send();
```

结果：警告非法反射访问操作

```
WARNING: An illegal reflective access operation has occurred...
【正在连接】
hello, hikari!
【已关闭】
```

从正常设计角度来说，还是基于接口的代理模式较为合理。

✧ 32.15 反射与 Annotation

① 获取 Annotation 信息

java.lang.reflect.AccessibleObject 类提供获取 Annotation 的方法，AccessibleObject 是 Constructor、Method、Field 三巨头的父类。

方法：

```
public <A extends Annotation> A getAnnotation(Class<A> annotationClass)
public Annotation[] getAnnotations()
```

```
@FunctionalInterface
@Deprecated(since = "1.8")
public interface ~~IMessage~~ {
    void send();
}

@SuppressWarnings("serial")
public class Message implements ~~IMessage~~, Serializable {
    @Override
    public void send() {
        System.out.println("hello, hikari!");
    }
}
```

测试：

```
        // 获取接口的Annotations
        printAnnotations(~~IMessage~~.class.getAnnotations());
        // 获取Message类的Annotations
        printAnnotations(Message.class.getAnnotations());
        // 获取Message类send()方法的Annotations
        Method method = Message.class.getDeclaredMethod("send");
        printAnnotations(method.getAnnotations());

    private static <T> void printAnnotations(Annotation[] annos) {
```

```
        for (Annotation a : annos) {
            System.out.println(a);
        }
        System.out.println("--------------------");
    }
```

结果：

```
@java.lang.FunctionalInterface()
@java.lang.Deprecated(forRemoval=false, since="1.8")
--------------------
--------------------
--------------------
```

@SuppressWarnings 和@Override 无法在程序执行时获取。不同的 Annotation 有各自存在范围。

比较 Override 和 FunctionalInterface：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {}

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

java.lang.annotation.RetentionPolicy 枚举类定义三个枚举常量用来表示 Annotation 保留策略。

1) **SOURCE**：源代码时有效，编译将被丢弃；
2) **CLASS**：注解被编译器记录到.class 文件，但运行时 JVM 不保留；
3) **RUNTIME**：注解被编译器记录到.class 文件，运行时 JVM 保留，可以通过反射读取。

② 自定义 Annotation

使用**@interface** 定义 Annotation。

1) 自定义 Annotation：

```
@Retention(RetentionPolicy.RUNTIME) // 定义Annotation保留策略
public @interface HikariAnnotation { // 自定义Annotation
    public String title();
    public String url() default "hikari.example.com"; // 默认值
}
```

2) 使用自定义的 Annotation：

```
public class Message {
    @HikariAnnotation(title = "hello")
    public void send(String s) {
        System.out.println("【消息发送】" + s);
```

```
    }
}
```

3) 获取 Annotation：

```
// 获取Message类的send()方法
Method method = Message.class.getDeclaredMethod("send", String.class);
// 获取自定义的Annotation对象
HikariAnnotation a = method.getAnnotation(HikariAnnotation.class);
String msg = a.title() + " (" + a.url() + ")";
method.invoke(Message.class.getDeclaredConstructor().newInstance(), msg);
// 【消息发送】hello (hikari.example.com)
```

使用 Annotation 最大特点是可以结合机制实现程序的处理。

③ 工厂模式与 Annotation 结合

拿之前的动态代理类个工厂类为例：

1) 自定义 Annotation

```
@Retention(RetentionPolicy.RUNTIME) // Annotation实现类的使用
public @interface UseMessage {
    public Class<?> cls();
}
```

2) 消息发送服务类，使用 Annotation 传入要实例化对象类型

```
@UseMessage(cls = Message.class) // 这样看起来更像Python的装饰器了...
public class MessageService {
    private IMessage msg;
    public MessageService() {
        // 通过Annotation获取类
        UseMessage use = MessageService.class.getAnnotation(UseMessage.class);
        this.msg = MyProxyFactory.create(use.cls());
    }
    public void send() {
        this.msg.send();
    }
}
```

3) 测试

```
        new MessageService().send();
```

面向接口的编程的配置可以使用 Annotation 的属性控制，使代码变得简洁。

33 Java 集合框架

✧ 33.1 Java 集合简介

传统数组最大的缺点是长度固定，最初只能依靠一些数据结构实现动态数组，如链表和树。但是数据结构代码实现困难，对于一般开发者无法使用；链表和树更新处理维护麻烦，且不能保证性能。

JDK 1.2 引入集合，主要对常见的数据结构进行完整实现包装。其中一系列接口与实现子类可以帮助用户减少数据结构带来的开发困难。

JDK 1.5 引入泛型后，集合可以使用泛型保存相同类型的数据。

Java 集合框架核心接口：Collection、List、Set、Map、Iterator、Enumeration、Queue、ListIterator。

✧ 33.2 Collection 接口

java.util.Collection 是单值集合最大的父接口，其中定义了所有单值数据操作。

Collection 接口常用方法：

```
boolean add(E e) // 向集合添加元素
boolean addAll(Collection<? extends E> c) // 向集合添加一组元素
void clear() // 清空集合元素
boolean contains(Object o) // 是否包含某个元素，需要覆写 equals()
boolean isEmpty() // 判断集合是否为空
E remove(int index) // 删除指定位置的元素
boolean remove(Object o) // 删除第一个指定元素，需要覆写 equals()
int size() // 获取集合元素个数
Object[] toArray() // 转为对象数组
Iterator<E> iterator() // 转为迭代器
```

Collection 继承于 Iterable 父接口；Collection 常用子接口是 List 和 Set

✧ 33.3 List 接口

List 接口最大特点是允许保存重复数据。

List 子接口对 Collection 接口进行了方法扩充：

```
E get(int index) // 根据索引获取元素
E set(int index, E element) // 指定索引设置数据
ListIterator<E> listIterator() // 转为 ListIterator 对象
```

List 接口常用子类：**ArrayList**、LinkedList、Vector。

JDK 1.9 开始 List 接口提供一系列重载的 static 的 of()方法，其返回一个不可修改的列表。

```
List<Integer> arr = List.of(54, 21, 64, 15, 34);
System.out.println(arr); //[54, 21, 64, 15, 34]
arr.add(12); //  不能修改，抛出 java.lang.UnsupportedOperationException
```

① ArrayList 类

ArrayList 类是 List 接口最常用的子类。

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

```
String[] arr = new String[] { "rin", "maki", "nozomi" };
List<String> al = new ArrayList<>();
```

```
for (String s : arr) {
    al.add(s);
}
System.out.println(al); // [rin, maki, nozomi]
```

以上直接使用提供的 toString()方法输出。JDK 1.8 后 Iterable 父接口提供了 forEach()方法：

```
default void forEach(Consumer<? super T> action)
```

```
al.forEach((s) -> {
    System.out.print(s + ", "); // rin, maki, nozomi,
});
```

但 forEach()方法不是开发中标准的输出。

ArrayList 部分源码：

```
    private static final int DEFAULT_CAPACITY = 10; // 默认容量 10
    private static final Object[] EMPTY_ELEMENTDATA = {};
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
    // ArrayList 内部维护的数组
    transient Object[] elementData; // non-private to simplify nested class access
    private int size; // ArrayList 元素个数
    // 构造方法，创建空数组，初始容量为指定值
    public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }
    // 无参构造，创建空数组，初始化容量为 10
    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }
```

ArrayList 里封装的就是一个对象数组。
JDK 1.9 之前，ArrayList 无参构造默认开辟大小为 10 的数组
JDK 1.9 之后，ArrayList 无参构造默认使用空数组，使用时才会开辟数组，默认开辟大小为 10。

add(E e)方法源码：

```
    public boolean add(E e) {
        modCount++; // 修改次数递增
```

```
        add(e, elementData, size);
        return true;
    }
    private void add(E e, Object[] elementData, int s) {
        if (s == elementData.length)
            elementData = grow();
        elementData[s] = e;
        size = s + 1;
    }
    private Object[] grow() {
        return grow(size + 1);
    }
    private Object[] grow(int minCapacity) { // 扩容保证当前元素能全部保存
        return elementData = Arrays.copyOf(elementData, newCapacity(minCapacity));
    }
    private int newCapacity(int minCapacity) { // 计算扩容后最少需要的容量
        // overflow-conscious code
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity + (oldCapacity >> 1); // 原来容量的 1.5 倍
        if (newCapacity - minCapacity <= 0) { // 新容量不大于最小需要容量(新容量不够)
            if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
               // 如果当前数组为空,返回默认容量 10 与最小需要容量的最大值
                return Math.max(DEFAULT_CAPACITY, minCapacity);
            if (minCapacity < 0) // 超过最大整数，溢出变为负数，报错
                throw new OutOfMemoryError();
            return minCapacity; // 返回最小需要容量
        }
        // 新容量满足最小需要容量，如果不超过 MAX_ARRAY_SIZE，直接返回新容量；如果超过通
过 hugeCapacity()获取
        return (newCapacity - MAX_ARRAY_SIZE <= 0)
            ? newCapacity
            : hugeCapacity(minCapacity);
    }
    private static int hugeCapacity(int minCapacity) {
        if (minCapacity < 0) // 超过最大整数，溢出变为负数，报错
            throw new OutOfMemoryError();
       // 超过 MAX_ARRAY_SIZE，则设为最大整数，否则设为 MAX_ARRAY_SIZE
        return (minCapacity > MAX_ARRAY_SIZE)
            ? Integer.MAX_VALUE
            : MAX_ARRAY_SIZE;
    }
    private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
```

如果添加元素时 ArrayList 长度不够，会创建新数组，采用 1.5 倍增长，并将原来

数组内容复制到新数组。

使用 ArrayList 时估算数据量如果大于 10，采用有参构造创建，必须垃圾数组空间产生。

② LinkedList 类
封装链表的实现。

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

Deque 是双端队列，是 Queue 的子接口。

```
String[] arr = new String[] { "rin", "maki", "nozomi" };
List<String> ll = new LinkedList<>();
for (String s : arr) {
    ll.add(s);
}
ll.forEach(System.out::println);
```

如果只看功能 LinkedList 和 ArrayList 几乎一样，但是其内部实现完全不同。

LinkedList 初始化不能指定初始容量，因为链表理论上可以无限添加。

add(E e)方法源码：

```
    public boolean add(E e) { // 所以数据都可以保存，null 也可以
        linkLast(e); // 在最后一个结点之后添加
        return true;
    }
    void linkLast(E e) {
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null) // 最后结点为空，链表也是空，设为头结点
            first = newNode;
        else // 否则直接添加到最后结点后面
            l.next = newNode;
        size++; // 大小递增
        modCount++; // 修改次数递增，为什么要有修改次数?
    }
```

面试题：ArrayList 和 LinkedList 的区别
1) ArrayList 基于数组实现；LinkedList 基于链表实现；
2) 使用 get()获取指定索引数据时，ArrayList 时间复杂度为 O(1)；LinkedList 时间复杂度为 O(*n*)；
3) ArrayList 默认初始化对象数组大小为 10，如果空间不足，采用 1.5 倍扩充，如

果保存大数据量可能造成垃圾产生和性能下降，此时可以使用 LinkedList。

③ Vector 类
JDK 1.0 提供的老古董。JDK 1.2 时许多开发者已经习惯了 Vector，并且许多系统类也是基于 Vector 实现，所以集合框架将其保留了下来。

```
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

继承结构与 ArrayList 相同。

Vector 构造方法：

```
    protected Object[] elementData;
    protected int elementCount;
    protected int capacityIncrement;
    public Vector(int initialCapacity, int capacityIncrement) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        this.elementData = new Object[initialCapacity];
        this.capacityIncrement = capacityIncrement;
    }

    public Vector(int initialCapacity) {
        this(initialCapacity, 0);
    }

    public Vector() {
        this(10);
    }
```

Vector 无参构造默认开辟一个长度为 10 的数组，后面实现与 ArrayList 相似。

Vector 类的 add(E e)方法：

```
    public synchronized boolean add(E e) {
        modCount++;
        add(e, elementData, elementCount);
        return true;
    }
```

Vector 类操作方法都是 synchronized 同步处理，而 ArrayList 并没有同步。所以 Vector 是线程安全的，但是性能不如 ArrayList。

✧ 33.4 Set 接口

Set 集合最大特点是不允许保存重复元素，其也是 Collection 子接口。
Set 集合没有 get()方法，不能通过索引获取元素。

JDK 1.9 后 Set 接口也提供了类型 List 接口的一系列 of()静态方法。

```
Set s = Set.of("a", "b", "a");
// java.lang.IllegalArgumentException: duplicate element: a
```

当设置了相同元素则直接抛出异常，这与 Set 不保存重复元素的特点相一致。
Set 接口常用子类是 HashSet 和 TreeSet。

① HashSet
HashSet 的 Set 接口最常用的子类，最大特点是保存数据是无序的。

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
```

继承关系与 ArrayList 十分相似

```
String[] arr = new String[] { "rin", "maki", "nozomi", "kotori", "rin", "umi",
"nico" };
HashSet<String> hs = new HashSet<>();
for (String s : arr) {
    hs.add(s);
}
System.out.println(hs); // [umi, maki, nico, rin, nozomi, kotori]
```

② TreeSet
TreeSet 和 HashSet 最大区别是 TreeSet 保存的数据是有序的。

```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
```

TreeSet 实现了 NavigableSet 接口，NavigableSet 继承于 SortedSet 接口，SortedSet 继承于 Set 接口。

```
String[] arr = new String[] { "rin", "maki", "nozomi", "kotori", "rin", "umi",
"nico" };
TreeSet<String> ts = new TreeSet<>();
for (String s : arr) {
    ts.add(s);
}
System.out.println(ts); // [kotori, maki, nico, nozomi, rin, umi]
```

TreeSet 保存数据按照升序进行自动排序。

TreeSet 存储自定义类时需要实现 Comparable 接口。TreeSet 本质利用 TreeMap 实现数据存储，TreeMap 根据 Comparable 确定大小关系。
TreeSet 根据 compare()方法是否为 0 确认数据是否重复。

TreeSet 存储自定义 Person 类：

```
public class Person implements Comparable<Person> {
    private String name;
```

```
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age > 0 ? age % 150 : 0;
    }
    // ...
    @Override
    public int compareTo(Person o) {
        int tmp = this.age - o.age;
        return tmp == 0 ? this.name.compareTo(o.name) : tmp;
    }
}

        TreeSet<Person> ts = new TreeSet<>();
        ts.add(new Person("rin", 15));
        ts.add(new Person("maki", 15));
        ts.add(new Person("nozomi", 17));
        ts.add(new Person("kotori", 16));
        System.out.println(ts); [<name: maki, age: 15>, <name: rin, age: 15>,
<name: kotori, age: 16>, <name: nozomi, age: 17>]
```