第01天 java基础加强

**【学习目标】理解、了解、应用、记忆**

1. **【理解】阐述Junit单元测试的使用及反射概述**
2. 【应用】独立编写代码使用JUnit运行
3. **【应用】通过反射操作构造方法、成员方法、成员属性**
4. 【记忆】阐述类加载的三个阶段是分别做了什么
5. 【应用】编码通过反射获取Class字节码文件对象
6. 【应用】通过反射获取构造方法的对象,并完成对象的创建
7. 【应用】通过反射获取方法的对象，完成方法的调用
8. 【应用】通过反射获取属性的对象，能够给对象的属性设置值
9. **【理解】应用Properties的基本操作**
10. 【应用】编写代码使用Properties的常用方法
11. 【应用】编写代码使用Properties完成配置文件的读取和加载配置文件
12. **【应用】独立完成相关的综合案例**
13. 【应用】能够阐述综合的需求
14. 【应用】能够编写代码完成教程中的案例（数据读取、数据加载、数据封装、反射获取对象，反射运行方法）

# Junit单元测试

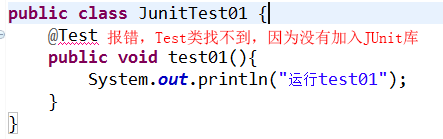
## Junit单元测试

### Junit介绍

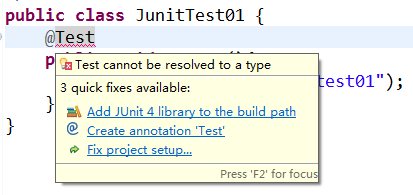
Junit是Java语言编写单元测试框架，最直观的理解，就是取代java类中的main方法。Junit属于第三方工具，一般情况下需要导入jar包，而多数Java开发环境都集成了Junit。

### 使用方式

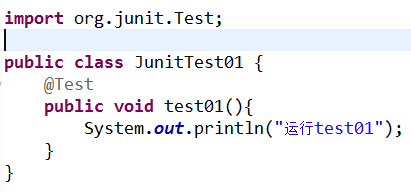
1. 编写测试类，简单理解Junit可以用于取代java的main方法
2. 在测试类方法上添加注解 @Test
3. @Test修饰的方法要求：public void 方法名() {…} ，方法名自定义建议test开头，没有参数。



1. 添加Eclipse中集成的Junit库，鼠标点击“@Test”，使用快捷键“ctrl + 1”，点击“Add Junit …”



添加成功之后查看导入的包名：



1. 使用：选中方法右键，执行当前方法；选中类名右键，执行类中所有方法（方法必须标记@Test）



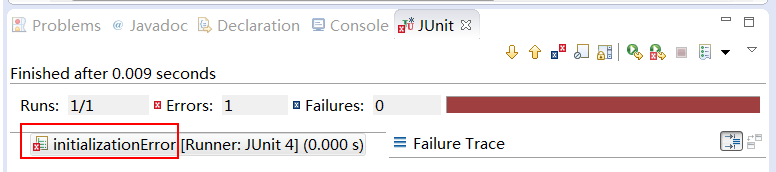
1. 常用注解

@Test，用于修饰需要执行的方法

@Before，测试方法前执行的方法

@After，测试方法后执行的方法

1. 常见使用错误，如果没有添加“@Test”，使用“Junit Test”进行运行，将抛异常



### 案例代码一

**public** **class** JunitTest01 {

/\*要求

\* 1、导入JUnit jar包

\* 2、import org.junit.Test;

\* 3、方法上面加注解@Test

\* 4、方法不能有形参

\*/

@Test

**public** **void** test01(){

System.***out***.println("运行test01");

}

@Test

**public** **void** test02(){

System.***out***.println("运行test02");

}

@Before

**public** **void** before(){

System.***out***.println("Junit运行之前做的准备工作。比如打开资源等");

}

@After

**public** **void** after(){

System.***out***.println("Junit运行之后做的善后工作。比如关闭资源等");

}

@Test

**public** **void** test03(){

System.***out***.println("运行test03");

}

}

# 反射

## 类的加载

### 类的加载

当程序要使用某个类时，如果该类还未被加载到内存中，则系统会通过加载，连接，初始化三步来实现对这个类进行初始化。

* 加载

就是指将class文件读入内存，并为之创建一个Class对象。

任何类被使用时系统都会建立一个Class对象

* 连接

验证 是否有正确的内部结构，并和其他类协调一致

准备 负责为类的静态成员分配内存，并设置默认初始化值

解析 将类的二进制数据中的符号引用替换为直接引用

* 初始化

就是我们以前讲过的初始化步骤

### 类的初始化

1. 创建类的实例

2. 类的静态变量，或者为静态变量赋值

3. 类的静态方法

4. 使用反射方式来强制创建某个类或接口对应的java.lang.Class对象

5. 初始化某个类的子类

6. 直接使用java.exe命令来运行某个主类

## 反射概述

Java反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

要想解剖一个类，必须先要获取到该类的字节码文件对象。而解剖这个类，使用的就是Class类中的方法，所以先要获取到每一个字节码文件所对应的Class类型对象。

反射需要用到的成员及方法：

* + Class类
  + Constructor 构造
  + Method 方法
  + Field 字段
  + instance 实例
  + invoke 执行

## 反射数据准备

提供JavaBean，用于之后反射操作 JavaBean规范

1.提供私有字段，例如：private String id;

2.必须提供getter或setter方法 （可以使用快捷键生产，alt+shift+s）

3.提供无参构造方法

### 案例代码二

**public** **class** Student {

//字段 field

**public** **int** id;

**private** String name;

//构造器 Constructor

**public** Student(**int** id, String name) {

**super**();

**this**.id = id;

**this**.name = name;

}

**public** Student() {

**super**();

// **TODO** Auto-generated constructor stub

}

//方法 Method

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**private** **void** setName(String name) {

**this**.name = name;

}

**public** **static** **int** getCount(**int** c){

**return** c\*100;

}

@Override

**public** String toString() {

**return** "Student [id=" + id + ", name=" + name + "]";

}

}

## 反射Class获取

### 三种获取方式及应用场景

**1 通过类型获得**

语法：类名.class

应用场景：确定类型 等

Class clazz1 = Bean.class;

**2 通过实例对象获得**

语法：对象.getClass()

应用场景：在方法内部通过参数获得类型 等

Bean bean = new Bean();

Class clazz2 = bean.getClass();

**3 通过字符串获得**

语法：Class.forName("全限定类名")

应用场景：通过配置获得字符串 等

Class clazz3 = Class.forName("com.igeek\_00\_Bean.Bean");

### 案例代码三

**public** **class** ReflectTest01 {

/\*

\* 3种新建Class对象的方法（Student.class->Class对象 ）

\*/

@Test

**public** **void** test01(){

//1、已知明确的类名，获取Class对象

Class clazz=Student.**class**;

System.***out***.println(clazz);

}

@Test

**public** **void** test02(){

//2、已知对象，获取Class对象

Student s1=**new** Student();

Class clazz=s1.getClass();

System.***out***.println(clazz);

}

@Test

**public** **void** test03() **throws** ClassNotFoundException{

//3、全限定类名（包名.类名）,获取Class对象 \*\*\*\*\*常用

String className="com.igeek\_02.Student";

Class clazz=Class.*forName*(className);

System.***out***.println(clazz);

}

@Test

**public** **void** test04() **throws** ClassNotFoundException{

Class clazz1=Student.**class**;

Student s1=**new** Student();

Class clazz2=s1.getClass();

String className="com.igeek\_02.Student";

Class clazz3=Class.*forName*(className);

System.***out***.println(clazz1==clazz2);//true clazz1和clazz2是同一个对象

System.***out***.println(clazz2==clazz3);//true clazz2和clazz3是同一个对象

}

## 反射操作构造方法

### 通过newInstance（）方法创建对象

如果类中有public的无参构造器，可以使用Class类中的newInstance()方法创建对象

步骤:

1. 获得Class对象
2. 通过Class类中的newInstance()方法创建对象

### 通过获取public的构造创建对象

步骤:

1.获得Class对象

2获得构造

3.通过构造对象获得实例化对象

**public** **class** ReflectTest01 {

@Test

**public** **void** test01() {

// 1、创建实例

Student s1 = **new** Student();

System.***out***.println(s1);

// 2、利用实例去调用成员方法

s1.setId(20);

System.***out***.println(s1);

}

@Test

**public** **void** test02() **throws** Exception {

// 1、创建该类的Class对象

String className = "com.igeek\_02.Student";

Class clazz = Class.*forName*(className);

// 2、获取构造器，利用构造器的newInstance方法创建实例

Object s1 = clazz.newInstance();

System.***out***.println(s1);

// 3、获取成员方法

Method m\_setId = clazz.getMethod("setId", **int**.**class**);

// 4、使用成员方法中的invoke方法去调用该方法

m\_setId.invoke(s1, 20);

System.***out***.println(s1);

}

// 获取方法的返回值问题

@Test

**public** **void** test03() **throws** Exception {

// 1、创建该类的Class对象

String className = "com.igeek\_02.Student";

Class clazz = Class.*forName*(className);

// 2、获取构造器，利用构造器的newInstance方法创建实例

Object s1 = clazz.newInstance();

System.***out***.println(s1);

// 3、获取成员方法

Method m\_getId = clazz.getMethod("getId");

// 4、使用成员方法中的invoke方法去调用该方法

Object returnValue = m\_getId.invoke(s1);

System.***out***.println("returnValue:" + returnValue);

}

// 如何调用私有成员方法

@Test

**public** **void** test04() **throws** Exception {

// 1、创建该类的Class对象

String className = "com.igeek\_02.Student";

Class clazz = Class.*forName*(className);

// 2、获取构造器，利用构造器的newInstance方法创建实例

Object s1 = clazz.newInstance();

System.***out***.println(s1);

//3、获取私有成员方法setName

//利用getDeclaredMethod(String name, Class<?>... parameterTypes)

Method m\_setName = clazz.getDeclaredMethod("setName",String.**class**);

System.***out***.println("m\_setName:"+m\_setName);

//4、打破私有的访问界限

m\_setName.setAccessible(**true**);

m\_setName.invoke(s1, "tom");

System.***out***.println(s1);

}

}

## 反射操作成员方法

### 反射public方法

**通过反射运行public方法流程**

1 获得实例

2 通过 setId 方法，设置数据

2.1获得方法 ，需要明确形参列表，setId

2.2 执行方法 ，bean.setId(...)

3.确定实例对象，并执行确定方法，具体实际参数

### 案例代码四

**public** **class** MethodDemo\_01 {

@Test

**public** **void** demo01(){

Bean bean = **new** Bean();

bean.setId("b001");

String id = bean.getId();

System.***out***.println(id);

}

@Test

**public** **void** demo02() **throws** Exception{

//1 获得实例

Class clazz = Class.*forName*("com.igeek\_01\_bean.Bean");

Object obj = clazz.newInstance();

//2 通过 setId 方法，设置数据

//2.1 获得方法 ，需要明确形参列表，setId

Method method = clazz.getMethod("setId", String.**class**);

//2.2 执行方法 ，bean.setId(...)

// \* 确定实例对象，并执行确定方法，具体实际参数

method.invoke(obj, "b002");

System.***out***.println(obj);

}

@Test

**public** **void** demo03() **throws** Exception{

//1 获得实例

Class clazz = Class.*forName*("com.igeek\_01\_bean.Bean");

Object obj = clazz.newInstance();

//2 通过 setId 方法，设置数据

//2.1 获得方法 ，需要明确形参列表，setId

Method method = clazz.getMethod("setId", String.**class**);

//2.2 执行方法 ，bean.setId(...)

// \* 确定实例对象，并执行确定方法，具体实际参数

method.invoke(obj, "b002");

//3 通过 getId 方法，获得数据

Method getMethod = clazz.getMethod("getId");

String str = (String) getMethod.invoke(obj);

System.***out***.println(str);

}

}

### 反射private方法

**通过反射运行private方法流程**

1 获得实例

2 执行方法

2.1 获得方法

2.2 强制设置运行访问私有访问（暴力）

2.3 执行方法

### 案例代码五

**public** **class** MethodDemo\_02 {

@Test

**public** **void** demo01() **throws** Exception{

// 执行 bean 私有方法

//1 获得实例

Class clazz = Class.*forName*("com.igeek\_01\_bean.Bean");

Object obj = clazz.newInstance();

//2 执行方法

// 2.1 获得方法

// \* clazz.getMethod(name,... ) 只能获得public类型方法

// \* clazz.getDeclaredMethod(name,... ) 获得已声明的方法

Method method = clazz.getDeclaredMethod("show", **int**.**class**);

// 2.2 强制设置运行访问私有访问（暴力）

method.setAccessible(**true**);

// 2.3 执行方法

Object r = method.invoke(obj, 18);

System.***out***.println(r);

}

@Test

**public** **void** demo02() **throws** Exception{

// 执行 静态方法 main

//1 获得实例，不需要

//2 执行方法，必须需要Class对象

//1 获得Class对象

Class clazz = Class.*forName*("com.igeek\_01\_bean.Bean");

//2获得main方法

Method method = clazz.getMethod("main", String[].**class**);

//3 执行main方法

// \* 参数1：为实例对象(变量名)，static方法不需要

// \* 参数2：main方法执行的实际参数

// \*\* 可变参数在执行时，JVM内部将传递实参数组打散，变成多个参数。

// \*\*\*\* method.invoke(null, args);

// 等效 method.invoke(null, "abc","123","xxx");

String[] args = {"abc","123","xxx"};

//method.invoke(null, args); //错误写法

// \* 数组参数需要特殊处理

// \*\* 方式1：将 String[]转换成Object,可变参数将不处理数组

method.invoke(**null**, (Object)args);

// \*\* 方法2：将提供二维数组，args将作为二维数组的第一个数。

method.invoke(**null**, **new** Object[] {args});

}

}

### 反射静态方法

**通过反射运行静态方法流程**

1 获得实例

2 执行方法

2.1 获得方法

2.2 执行方法

### 案例代码六

**public** **class** ReflectTest02 {

@Test

**public** **void** test01(){

//静态方法如何调用：类名.方法

**int** count=Student.*getCount*(2);

System.***out***.println(count);

}

@Test

**public** **void** test02() **throws** Exception{

// 1、创建该类的Class对象

String className = "com.igeek\_02.Student";

Class clazz = Class.*forName*(className);

//2、找到静态方法

Method m\_getCount=clazz.getMethod("getCount", **int**.**class**);

//3、调用该静态方法 因为不需要实例，所以传入null

Object returnValue=m\_getCount.invoke(**null**, 2);

System.***out***.println(returnValue);

}

}

## 反射操作字段

### 反射public字段

**反射public**字段**执行流程**

1 获得实例对象

2 给字段赋值

3 获得字段值

### 案例代码六

**public** **class** FieldTest01 {

@Test

**public** **void** test01(){

//给类中的public的字段赋值

Student s1=**new** Student();

s1.id=20;

System.***out***.println(s1);

}

//类中公共字段

@Test

**public** **void** test02() **throws** Exception{

//1、获取该类的Clazz类型对象

String className = "com.igeek\_02.Student";

Class clazz = Class.*forName*(className);

//2、创建该类的实例

Object s1=clazz.newInstance();

System.***out***.println(s1);

//3、获取该类的字段

Field f\_id=clazz.getField("id");

System.***out***.println("f\_id:"+f\_id);

//4、给这个字段赋值

//void set(Object obj, Object value)

f\_id.set(s1, 20);

System.***out***.println(s1);

//5、获取字段的值

Object id\_value=f\_id.get(s1);

System.***out***.println("id\_value:"+id\_value);

}

}

### 反射private字段

**反射private**字段**流程**

1 获得实例

2 获得声明的字段(私有)

3 设置运行访问私有（暴力）

4 设置数据

5 获得数据

### 案例代码七

**public** **class** FieldTest02 {

//如何获取类中私有的字段

@Test

**public** **void** test01() **throws** Exception{

//1、获取该类的Clazz类型对象

String className = "com.igeek\_02.Student";

Class clazz = Class.*forName*(className);

//2、创建该类的实例

Object s1=clazz.newInstance();

System.***out***.println(s1);

//3、获取该类的私有字段

Field f\_name=clazz.getDeclaredField("name");

System.***out***.println("f\_name:"+f\_name);

//4、打破私有的访问限定

f\_name.setAccessible(**true**);

//5、赋值

f\_name.set(s1, "tom");

System.***out***.println(s1);

//6、拿值

Object s1\_name=f\_name.get(s1);

System.***out***.println("s1\_name:"+s1\_name);

}

}

# Properties的基本操作

## Properties的概述及基本使用

### Properties的概述

Properties 类表示了一个持久的属性集。Properties 可保存在流中或从流中加载。属性列表中每个键及其对应值都是一个字符串。

**特点：**

1、Map接口的子类，map中的方法都可以用。

2、该集合没有泛型。键值都是字符串。

3、它是一个可以持久化的属性集。键值可以存储到集合中，也可以存储到持久化的设备(硬盘、U盘、光盘)上。键值的来源也可以是持久化的设备。

4、有和流技术相结合的方法。

## 常用方法

* public Object setProperty(String key, String value)调用 Map 的方法 put。
* public Set<String> stringPropertyNames()返回此属性列表中的键集，
* public String getProperty(String key)用指定的键在此属性列表中搜索属性

### 案例代码八

**public** **class** PropsDemo\_01 {

@Test

**public** **void** demo01(){

//Properties对象 特殊Map<String,String>

//1 创建对象

Properties props = **new** Properties();

//2 设置数据

//props.put(key, value)

props.setProperty("k01", "v01");

props.setProperty("k02", "v02");

props.setProperty("k03", "v03");

//3 获得指定名称的数据

String prop = props.getProperty("k02");

System.***out***.println(prop);

//4获得所有名称，并遍历获得所有的数据

System.***out***.println("--------------");

Set<String> names = props.stringPropertyNames();

**for**(String name : names){

String value = props.getProperty(name);

System.***out***.println(name + " : " + value);

}

}

}

## Properties加载文件写入文件

### 将集合中内容存储到文件

* store(OutputStream,comments)
* store(Writer,comments);

把集合中的数据，保存到指定的流所对应的文件中，参数comments代表对描述信息

### 案例代码九

@Test

**public** **void** demo01() **throws** Exception{

//将内容写入到文件

// \* Properties 对应一种特殊文件：\*.properties

// \* 一行表示一个键值对，格式：k=v

//1 创建对象

Properties props = **new** Properties();

//2 设置数据

//props.put(key, value)

props.setProperty("k01", "v01极客营 ");

props.setProperty("k02", "v02程序员");

props.setProperty("k03", "v03");

//3 将props写入到硬盘

Writer writer = **new** OutputStreamWriter(**new** FileOutputStream("1.properties"),"UTF-8");

props.store(writer, "描述");

writer.close();

}

### 读取文件中的数据，并保存到集合

* load(InputStream)
* load(Reader)

把指定流所对应的文件中的数据，读取出来，保存到Propertie集合中

### 案例代码十

@Test

**public** **void** demo02() **throws** Exception{

//从properties文件中获得数据

//1 获得资源流

Reader reader = **new** InputStreamReader(**new** FileInputStream("1.properties"),"UTF-8");

//2 通过Properties对象加载流

Properties props = **new** Properties();

props.load(reader);

//3 遍历所有

**for**(String name : props.stringPropertyNames()){

String value = props.getProperty(name);

System.***out***.println(name + " @ " + value);

}

}

# 综合案例

## 需求分析

将javaBean的数据和类名以配置文件的方式写进配置文件中，在测试类中去读取配置文件中保存的数据，并以反射的方式根据读取的数据创建对象，给对象的属性设置值。  
 反射的好处：我们不用将数据写死到java代码中，可以使用配置文件将数据写入配置文件中。提高了代码的灵活性。

## 代码案例十一

### 首先编写JavaBean

**Student类**

**public** **class** Student {

**private** **int** id;

**private** String name;

**public** Student(**int** id, String name) {

**super**();

**this**.id = id;

**this**.name = name;

}

@Override

**public** String toString() {

**return** "Student [id=" + id + ", name=" + name + "]";

}

}

**Teacher类**

**public** **class** Teacher {

**private** **int** id;

**private** String name;

**public** Teacher(**int** id, String name) {

**super**();

**this**.id = id;

**this**.name = name;

}

@Override

**public** String toString() {

**return** "Teacher [id=" + id + ", name=" + name + "]";

}

}

### 创建配置文件

**className.properties**

classname=com.igeek\_01.Teacher

**data.properties**

id=2018

name=Eric

### 编写测试类

**目的: 封装2个Properties文件中的内容**

**public** **class** ConfigTest01 {

@Test

**public** **void** test01(){

// //创建Student对象

// Student s1=new Student(20, "tom");

// System.out.println(s1);

// //创建Teacher对象

// Teacher t1=new Teacher(20, "tom");

// System.out.println(t1);

}

//从配置文件className.properties中读取应该要创建的是哪一个类

@Test

**public** **void** test02() **throws** Exception{

//1、从className.properties获取类名

Properties prop=**new** Properties();

prop.load(**new** FileReader("className.properties"));

String className=prop.getProperty("classname");

System.***out***.println("className:"+className);

//2、创建该类的Class实例

Class clazz=Class.*forName*(className);

//3、找到该类的构造器

Constructor ctor=clazz.getConstructor(**int**.**class**,String.**class**);

//4、利用构造器填入数据

Object obj=ctor.newInstance(20,"tom");

System.***out***.println(obj);

}

//从配置文件data.properties中读取数据并填入实例

@Test

**public** **void** test03() **throws** Exception{

//1、从className.properties获取类名

Properties prop=**new** Properties();

prop.load(**new** FileReader("className.properties"));

String className=prop.getProperty("classname");

System.***out***.println("className:"+className);

//2、创建该类的Class实例

Class clazz=Class.*forName*(className);

//3、找到该类的构造器

Constructor ctor=clazz.getConstructor(**int**.**class**,String.**class**);

//4、利用构造器填入数据

//4.1从配置文件data.properties获取数据

Properties prop\_Data=**new** Properties();

prop\_Data.load(**new** FileReader("data.properties"));

String v\_id=prop\_Data.getProperty("id");

String v\_name=prop\_Data.getProperty("name");

Object obj=ctor.newInstance(Integer.*parseInt*(v\_id),v\_name);

System.***out***.println(obj);

}

}