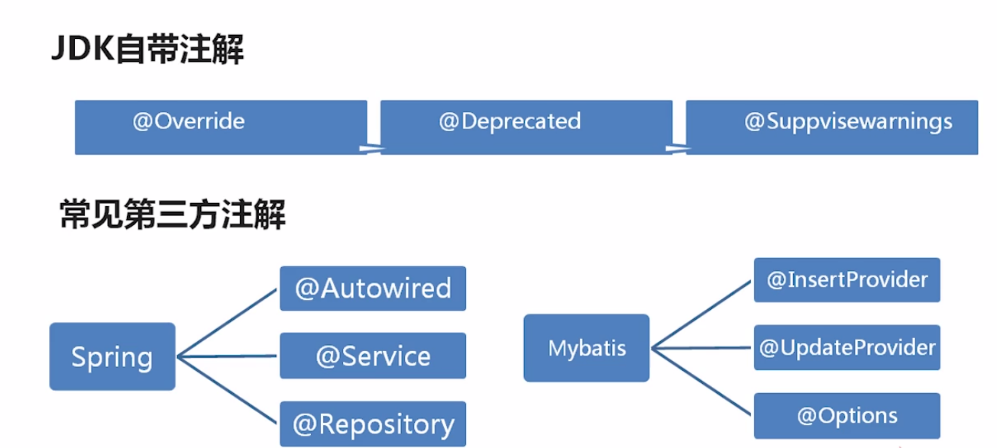
# Java注解

## 分类



/\*\*

\* Override覆写注解

\*/

@Override

/\*\*

\* Deprecated方法过时注解

\*/

@Deprecated

忽略警告注解

@SuppressWarnings("deprecation")

## 按运行机制分：

**源码注解**

注解只在源码中存在，编译成class文件后就不存在了

**编译注解**

注解在源文件和.class文件中都存在。如Override等

**运行注解**

运行时还存在或能起作用，影响程序运行逻辑。

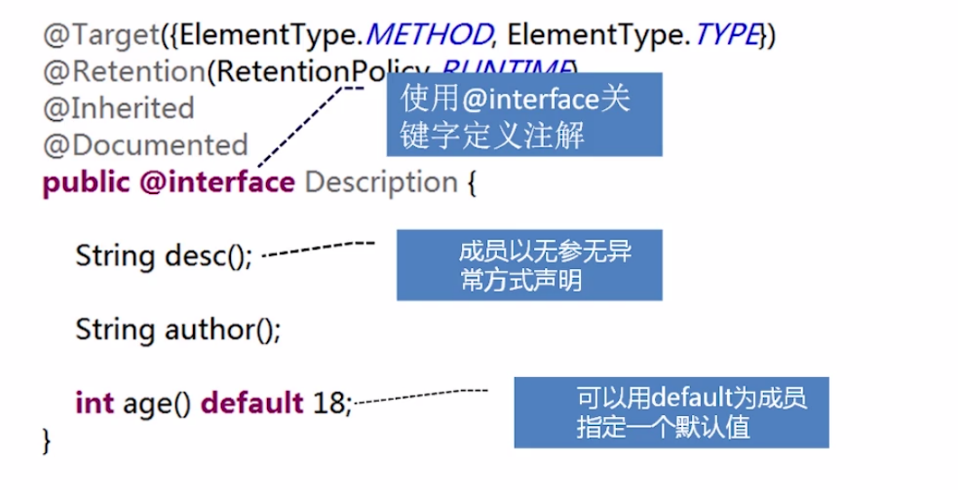
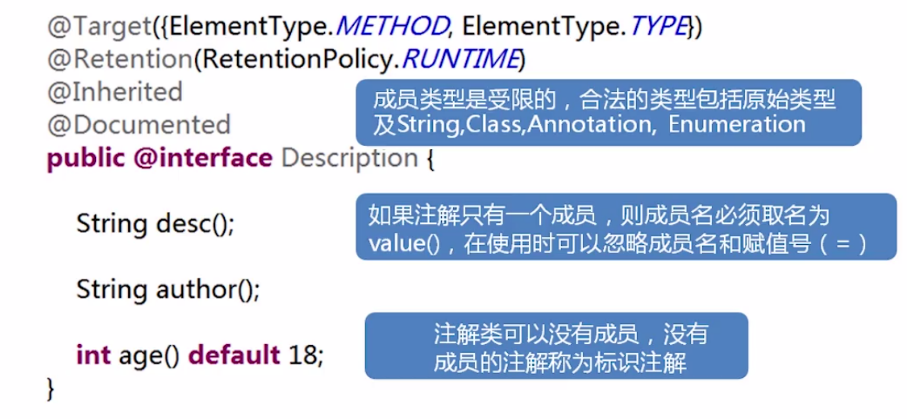
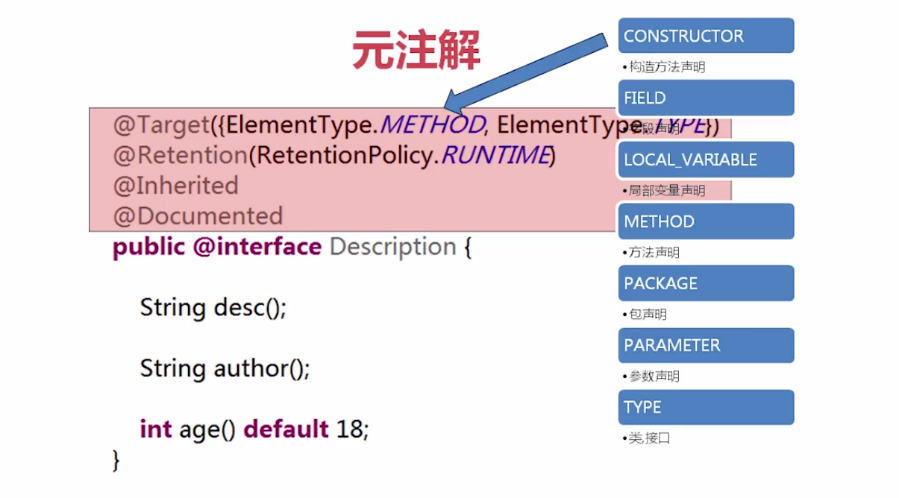
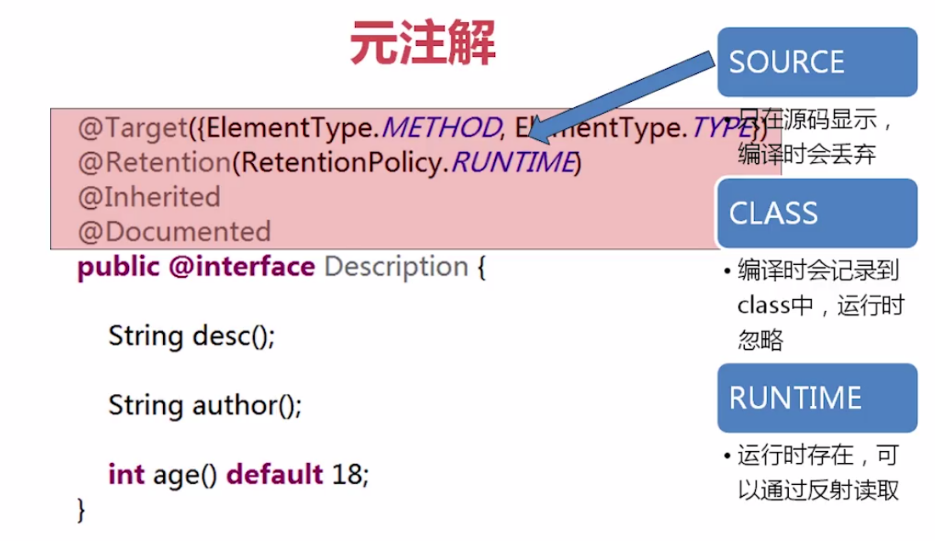
## 按注解来源分：

来自JDK

来自第三方

自己定义的

元注解（注解的注解）





## 自定义注解

@Target({ElementType.***METHOD***,ElementType.***TYPE***})

@Retention(RetentionPolicy.***RUNTIME***)

@Inherited

@Documented

**public** **@interface** Description {

String desc();

String author();

**int** age() **default** 18;

}

## 自定义注解的使用

@Description(author = "duan", desc = "any",age = 24)

**public** **void** getAge()

{

## 解析注解

通过反射获取类、函数或成员上的运行时注解信息，从而实现动态控制程序运行的逻辑。

//使用类加载器加载类

try {

Class c = Class.forName("test.Child");

//找到类上面的注解

boolean isExist = c.isAnnotationPresent(Description.class);

if (isExist)

{

//拿到注解实例

Description des = (Description)c.getAnnotation(Description.class);

System.out.println(des.value());

}

//找到方法上的注解

Method[] me = c.getMethods();

for(Method m: me)

{

if (m.isAnnotationPresent(Description.class))

{

Description des = (Description)m.getAnnotation(Description.class);

System.out.println(des.value());

}

}

//另一种解析方法上注解的方法

for (Method m : me)

{

Annotation[] an = m.getAnnotations();

for (Annotation a : an)

System.out.println(a.toString());

}

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

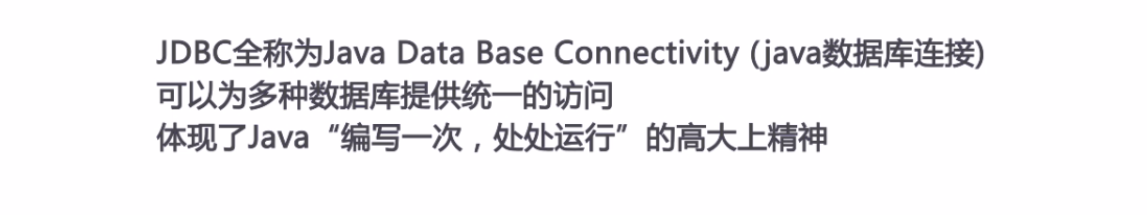
e.printStackTrace();

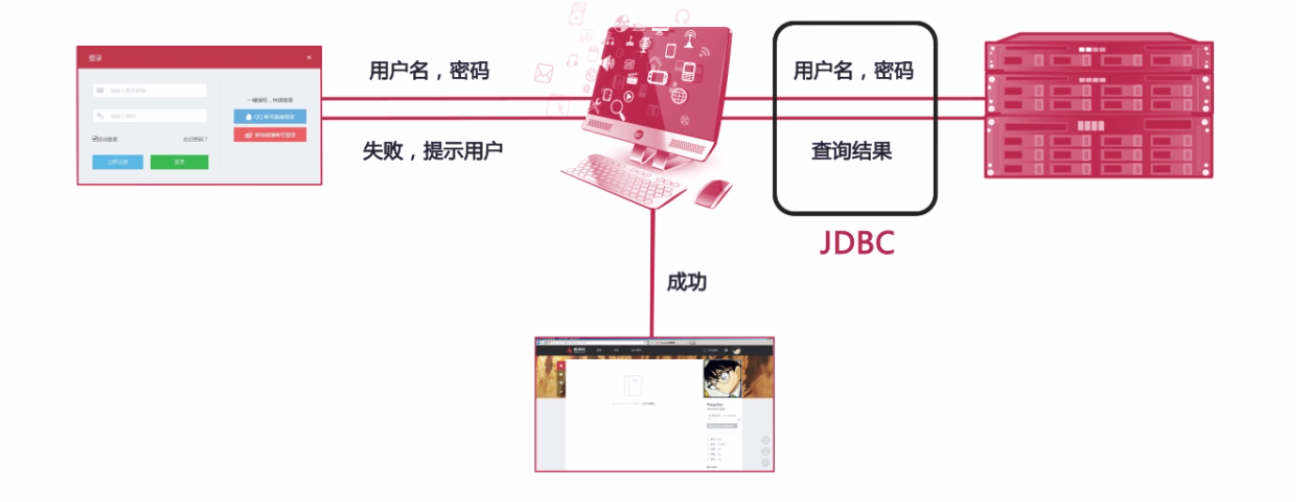
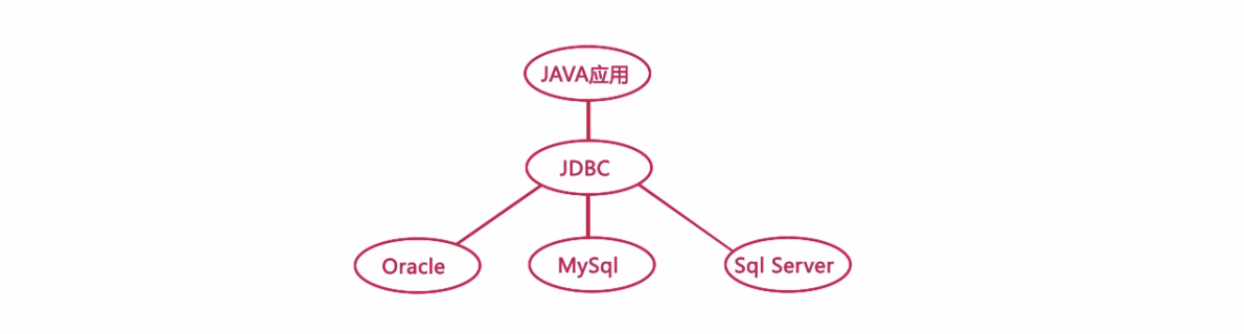
}

}

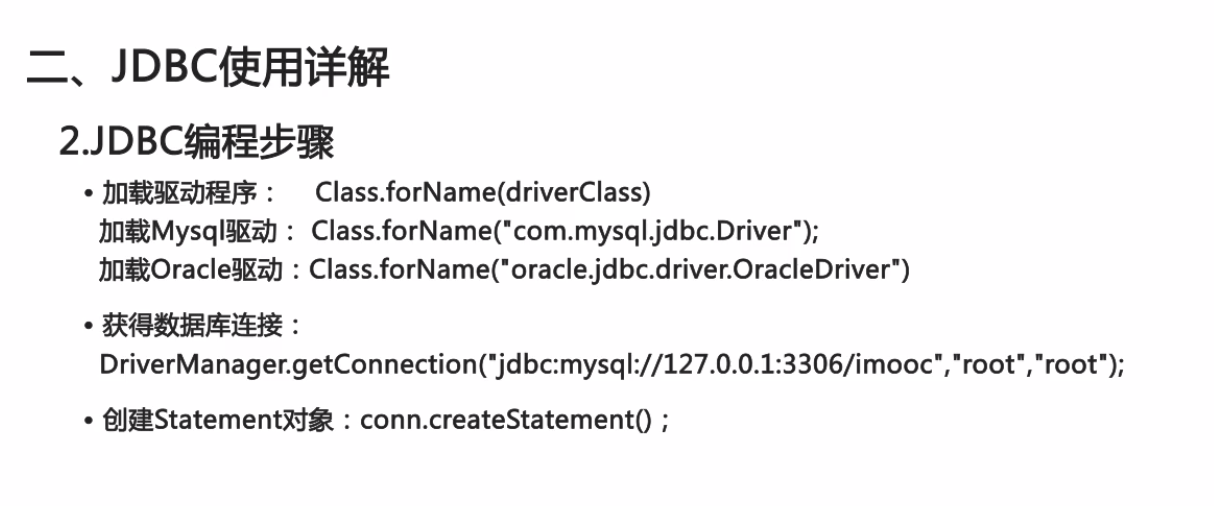
}

# JDBC简介

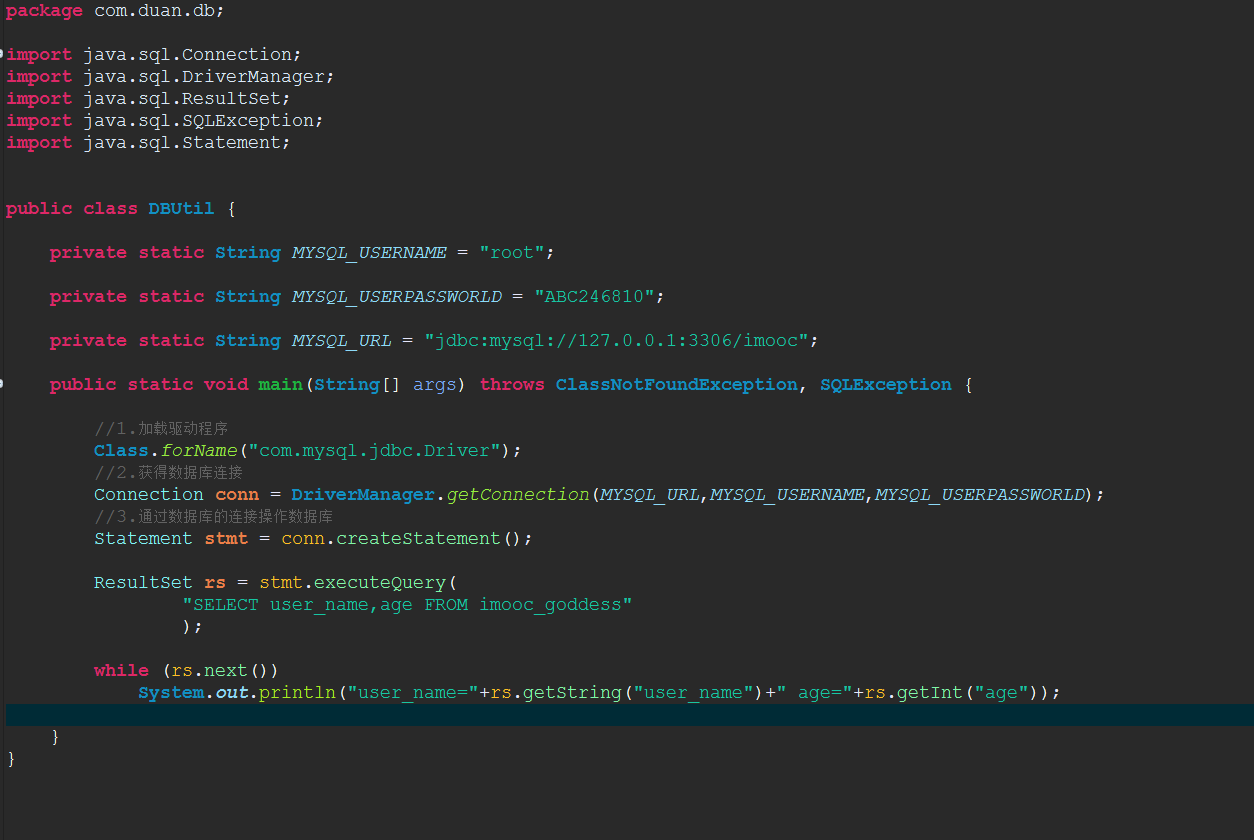




## JDBC使用步骤



## 从数据库获取数据

****

# java反射

反射主要是指程序可以访问、检测和修改它本身状态或行为的一种能力。在计算机科学领域，反射是一类应用，它们能够自描述和自控制。这类应用通过某种机制来实现对自己行为的描述和检测，并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。

在Java中的反射机制，被称为Reflection（大家看到这个单词，第一个想法应该就是去开发文档中搜一下了）。它允许运行中的Java程序对自身进行检查，并能直接操作程序的内部属性或方法。

Reflection机制允许程序在正在执行的过程中，利用Reflection APIs取得任何已知名称的类的内部信息，

包括：

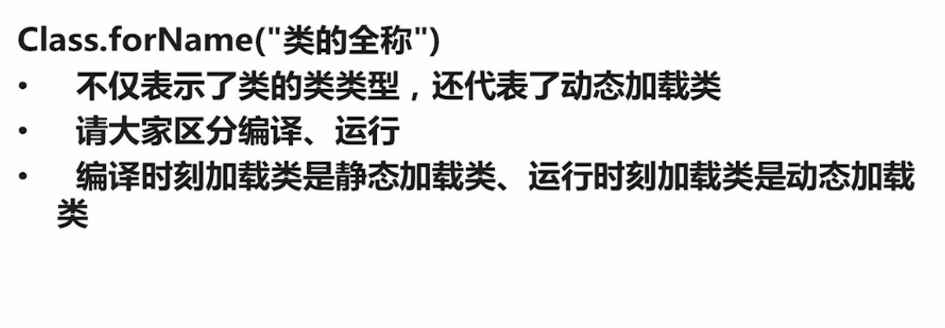
package、 type parameters、 superclass、 implemented interfaces、 inner classes、 outer classes、 fields、 constructors、 methods、 modifiers等，并可以在执行的过程中，

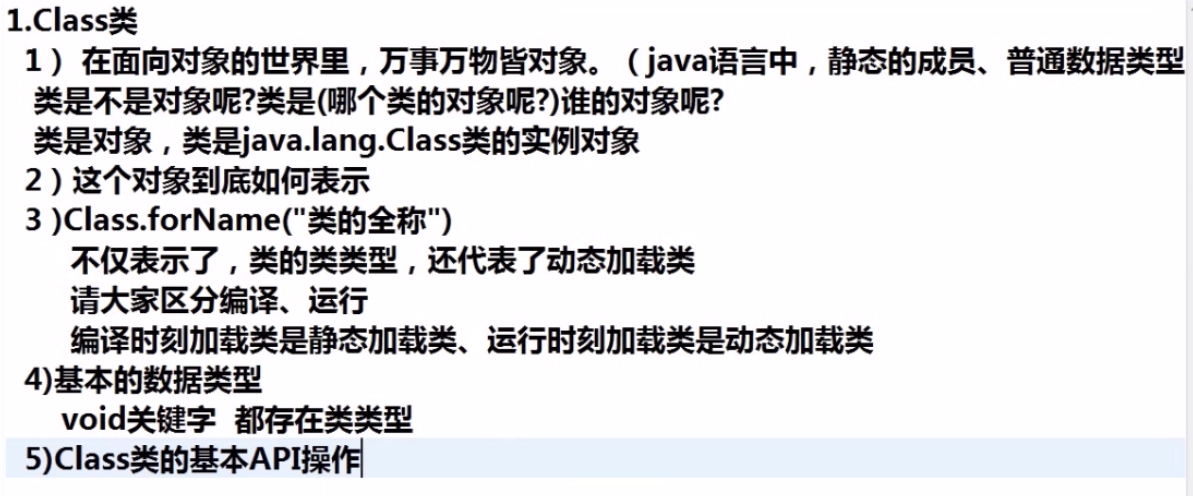
1. 动态生成Instances、
2. 变更fields（成员变量）内容
3. 唤起methods。

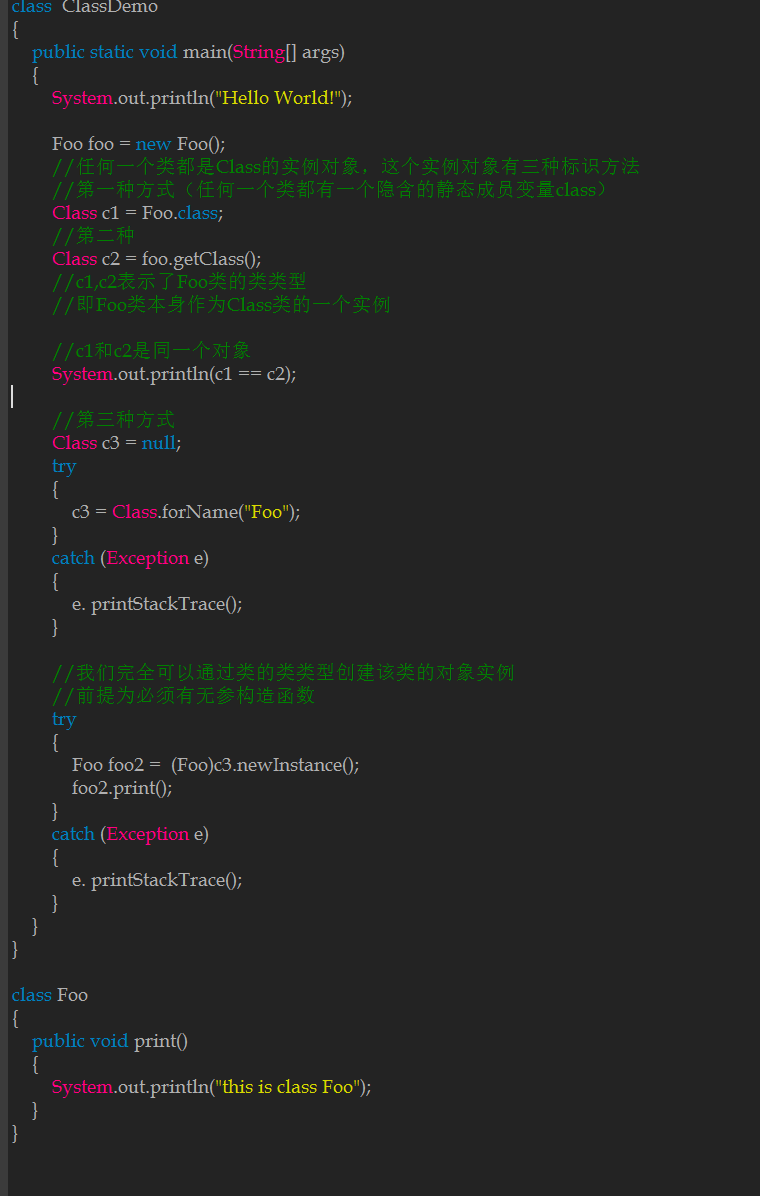
## Class类的使用

java里万事万物皆为对象，类也是对象，是Class类，这个对象我们称为该类的类类型。但静态成员（静态成员属于类不属于对象）、普通的数据类型不是对象

类是java.lang.Class类的实例对象







### 静态加载类（编译时刻加载类）：

new创建对象是静态加载类，在编译时刻就需要加载可能用到的类。即：编译时刻类中有错误就会导致程序编译失败无法运行，而出错的地方可能在程序实际运行时并不会用到，通过动态加载类可以解决该问题

### 动态加载类（运行时刻加载类）：

Class c = **Class.forName(args[0]);**

Officeable officeable = (Officeable)c.newInstance();

officeable.start();

## 方法的反射

### Java.lang.reflect.Method类封装了对类的方法的操作

//首先获取类类类型，传递的是哪个子类的对象，c就是该子类的类类型

Class c = obj.getClass();

//获取类的名称

String className = c.getName();

System.out.println("Class "+className);

//获得类的方法，一个成员方法就是一个Method对象

//getMethod方法获得所有的public方法包括继承于父类的

//getDeclaredMethods方法获得所有该类自己的方法，与权限无关

Method[] ms = c.**getMethods**();

Method[] mds = c.**getDeclaredMethods**();

for (int i=0;i < ms.length ;i++ )

{

//得到方法的返回值类型的类类型

Class returnType = ms[i].**getReturnType**();

//得到方法的名称

String name = ms[i].**getName**();

System.out.print(name+" (");

//获得参数类型\_\_的到参数列表的类型的类类型

Class[] paramTypes = ms[i].**getParameterTypes**();

for (Class class1 : paramTypes)

{

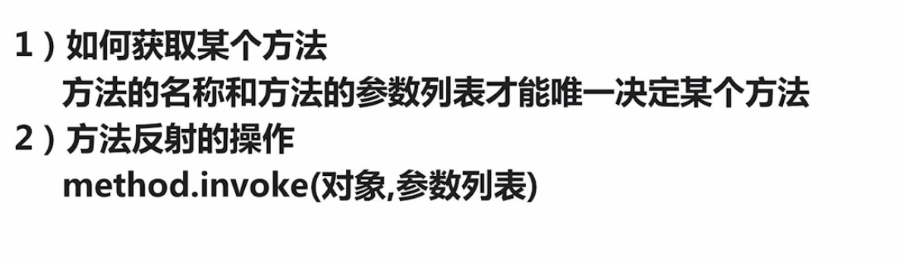
System.out.print(class1.getName()+" ");

}

System.out.print(")");

System.out.println();

}



### 获得方法对象并通过反射调用方法

//System.out.println(new Test().getNum(3,"asdf"));

Test test = new Test();

Class clt = test.getClass();

//获取方法 名称和参数列表决定（方法可能重载）

//getMethod()

//**getDeclaredMethod**()

Method m = clt.**getMethod**("getNum",int.class,String.class);

Method m2 = clt.getMethod("getNum");

//方法的反射操作

//用Method的实例调用方法与一般的方法调用效果完全相同

int intRe = (int)m.**invoke**(test,789,"sadf");

System.out.println(intRe+"");

System.out.println((int)m2.invoke(test));

## 通过反射了解泛型的本质

反射的操作都是编译之后的操作，编译之后集合的泛型是去泛型化的

Java中的泛型是为了防止错误输入（赋值），只在编译阶段有效，绕过编译就无效了。

验证：通过方法的反射绕过编译

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

{

List list = new ArrayList<String>();

list.add("hello");

Class li = list.getClass();

Method m = li.getMethod("add",Object.class);

//绕过了编译就绕过了泛型限制

Integer integer = 100;

m.invoke(list,integer);

System.out.println(list);

}

## 成员变量的反射

Java.lang.reflect.Field类封装了对java类的成员变量的操作

//首先获取类类类型，传递的是哪个子类的对象，c就是该子类的类类型

Class c = obj.getClass();

//获取类的名称

String className = c.getName();

System.out.println("Class "+className);

Field[] f = c.**getDeclaredFields**();

for (**Field** field:f )

…..

## 构造函数的反射

Java.lang.reflect.Constructor类封装了对java类的构造函数的操作

//首先获取类类类型，传递的是哪个子类的对象，c就是该子类的类类型

Class c = obj.getClass();

//获的自己声明的构造函数

Constructor[] con = c.**getDeclaredConstructors**();

for (Constructor cons : con)

{

Class[] paramTypes = cons.getParameterTypes();

System.out.print(cons.getName()+"(");

for (Class cl : paramTypes)

{

System.out.print(cl.getName()+",");

}

System.out.println(")");

}

<?xml version="1.0" encoding="UTF-8"?>

<bookstore>

<book id="001" number="2045">

<name>song fo ice and fire</name>

<author>judgematy</author>

<year>2014</year>

<price>89</price>

</book>

<book id="002">

<name>fail story</name>

<year>2004</year>

<price>77</price>

<language>English</language>

</book>

</bookstore>

# Java解析XML

## DOM解析方式

import java.io.\*;

import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

import org.w3c.dom.NodeList;

import org.w3c.dom.Node;

import org.w3c.dom.NamedNodeMap;

import javax.xml.parsers.DocumentBuilder;

import org.xml.sax.SAXException;

import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom .Element;

public class main

{

public static void main(String[] args)

{

DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

try

{

//创建DocumentBuilder

DocumentBuilder db = dbf.newDocumentBuilder();

//加载xml文件

Document document = db.parse("F:\\javaStuff\\books.xml");

//获取所有book结点

NodeList bookList = document.getElementsByTagName("book");

System.out.println("sum of books is "+bookList.getLength());

//NodeList.length()获取长度

for (int i = 0;i<bookList.getLength() ;i++ )

{

//获取每一个book

Node book = bookList.item(i);

//获取book节点的所有属性集合

NamedNodeMap attrs = book.getAttributes();

System.out.println("book "+(i+1)+" have attrs "+attrs.getLength());

for (int j =0;j<attrs.getLength() ; j++)

{

System.out.println(attrs.item(j).getNodeName()+"："+attrs.item(j).getNodeValue());

}

NodeList childNodes = book.getChildNodes();

//System.out.println("\n"+childNodes.getLength());

for (int k=0;k<childNodes.getLength() ;k++ )

{

//区分出text类型的node以及element类型的node

if (childNodes.item(k).getNodeType() == Node.ELEMENT\_NODE)

{

System.out.println(childNodes.item(k).getNodeName()+"："+childNodes.item(k).getFirstChild().getNodeValue());

}

}

//前提：知道book节点有且只有一个id属性

/\*Element book = (Element)bookList.item(i);

String attr = book.getAttribute("id");

\*/

}

}

catch (ParserConfigurationException e)

{

e.printStackTrace();

}

catch (SAXException e)

{

e.printStackTrace();

}

catch (IOException e)

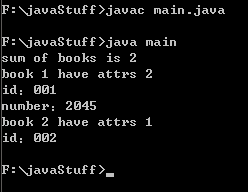
{

e.printStackTrace();

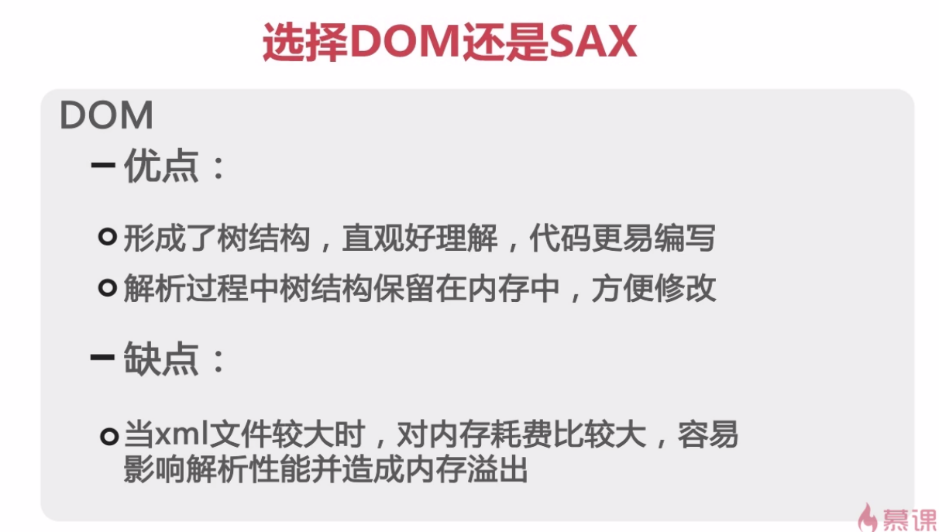
}

}

}



## DOM4J解析方式（建议使用）





import org.dom4j.Document;

import org.dom4j.io.SAXReader;

import org.dom4j.Element;

import org.dom4j.DocumentException;

import org.dom4j.Attribute;

import java.io.File;

import java.util.Iterator;

import java.util.List;

class xmlDemo

{

public static void main(String[] args)

{

//解析book.xml

//创建SAXReader以加载.xml文件

SAXReader reader = new SAXReader();

try

{

//获取document对象

Document document = reader.read(new File("books.xml"));

//获得根节点

Element bookstore = document.getRootElement();

//获得迭代器

Iterator it = bookstore.elementIterator();

//遍历迭代器

while (it.hasNext())

{

Element book = (Element)it.next();

//获取book的属性名和属性值

List<Attribute> bookAttrs = book.attributes();

for (Attribute arr : bookAttrs )

{

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

System.out.println("节点："+arr.getName()+" "+arr.getValue());

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

}

Iterator itt = book.elementIterator();

while (itt.hasNext())

{

Element bookChild = (Element)itt.next();

System.out.println(" "+bookChild.getName()+bookChild.getStringValue());

}

}

}

catch (DocumentException e)

{

e.printStackTrace();

}

}

}

