# 注解Annotation

## 1、什么是注解

- Annotation 是从JDK5.0开始引入的新技术 .
- Annotation的作用
  - 不是程序本身 , 可以对程序作出解释.(这一点和注释(comment)没什么区别)
  - 可以被其他程序(比如:编译器等)读取.
- Annotation的格式
  - 注解是以"@注释名"在代码中存在的
  - 还可以添加一些参数值 , 例如:@SuppressWarnings(value="unchecked")
- Annotation在哪里使用?
  - 可以附加在package , class , method , field 等上面 , 相当于给他们添加了额外的辅助信息
  - 我们可以通过反射机制实现对这些元数据的访问

## 2、内置注解

- @Override
  - 定义在 java.lang.Override 中 , 此注释只适用于修辞方法 , 表示一个方法声明打算重写超类中的另一个方法声明.
- @Deprecated
  - 定义在java.lang.Deprecated中 , 此注释可以用于修辞方法 , 属性 , 类 ,
  - 表示不鼓励程序员使用这样的元素 , 通常是因为它很危险或者存在更好的选择 .
- @SuppressWarnings
  - 定义在java.lang.SuppressWarnings中,用来抑制编译时的警告信息.
  - 与前两个注释有所不同,你需要添加一个参数才能正确使用,这些参数都是已经定义好了的,我们选择性的使用就好了 .
    - @SuppressWarnings("all")
    - @SuppressWarnings("unchecked")
    - @SuppressWarnings(value={"unchecked","deprecation"})
    - 等等 .....

```
1  package com.annotation;
2
3  //测试内置注解
4  import java.util.ArrayList;
5  import java.util.List;
6  //所有类默认继承Object类
7  public class Test1 extends Object {
8
9      //@Override 表示方法重写
10     //--> 查看JDK帮助文档
11     //--> 测试名字不同产生的效果
12     @Override
13     public String toString() {
14         return super.toString();
```

```
    }

    //方法过时了，不建议使用 ，可能存在问题 ，并不是不能使用!
    //--> 查看JDK帮助文档
    @Deprecated
    public static void stop(){
        System.out.println("测试 @Deprecated");
    }

    //@SuppressWarnings  抑制警告 ，可以传参数
    //--> 查看JDK帮助文档
    //查看源码:发现 参数类型  和  参数名称 ，并不是方法!
    @SuppressWarnings("all")
    public void sw(){
        List list = new ArrayList();
    }

    public static void main(String[] args) {
        stop();
    }

}
```

## 3、元注解

- 元注解的作用就是负责注解其他注解 , Java定义了4个标准的meta-annotation类型,他们被用来提供对其他annotation类型作说明 .
- 这些类型和它们所支持的类在java.lang.annotation包中可以找到 .( @Target , @Retention , @Documented , @Inherited )
  - @Target : 用于描述注解的使用范围(即:被描述的注解可以用在什么地方)
  - @Retention : 表示需要在什么级别保存该注释信息 , 用于描述注解的生命周期
    - (SOURCE < CLASS < RUNTIME)
  - @Document：说明该注解将被包含在javadoc中
  - @Inherited：说明子类可以继承父类中的该注解

```
package com.annotation;

import java.lang.annotation.*;

//测试元注解
public class Test2 {
    @MyAnnotation
    public void test(){

    }
}

//定义一个注解
@Target(value = {ElementType.METHOD,ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
@Inherited
@Documented
@interface MyAnnotation{
```

```
    //测试作用域 , 了解@Retention的概念
}
```

## 4、自定义注解

- 使用 @interface自定义注解时 , 自动继承了java.lang.annotation.Annotation接口
- 分析 :
  - @ interface用来声明一个注解 , 格式 : public @ interface 注解名 { 定义内容 }
  - 其中的每一个方法实际上是声明了一个配置参数.
  - 方法的名称就是参数的名称.
  - 返回值类型就是参数的类型 ( 返回值只能是基本类型,Class , String , enum ).
  - 可以通过default来声明参数的默认值
  - 如果只有一个参数成员 , 一般参数名为value
  - 注解元素必须要有值 , 我们定义注解元素时 , 经常使用空字符串,0作为默认值 .

```
package com.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//测试自定义注解
public class Test3 {

    //显示定义值 / 不显示值就是默认值
    @MyAnnotation2(age = 18,name = "秦疆",id = 001,schools = {"西工大"})
    public void test() {

    }

    //只有一个参数, 默认名字一般是value.使用可省略不写
    @MyAnnotation3("aaa")
    public void test2(){
    }

}


@Target(value = {ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
@interface MyAnnotation2{
    //参数类型 , 参数名
    String name() default "";
    int age() default 0;
    int id() default -1;  //String indexOf("abc")  -1 , 不存在,找不到

    String[] schools() default {"西部开源","狂神说Java"};
}


@Target(value = {ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
@interface MyAnnotation3{
```

```
    // 参数类型     参数名称
    String value();
}
```

## 5、反射读取注解

```
package com.annotation;

import java.lang.annotation.*;
import java.lang.reflect.Field;

//测试ORM:对象关系映射

//使用反射读取注解信息三步:
// 1.定义注解 ,
// 2.在类中使用注解 ,
// 3. 使用反射获取注解 , 一般都是现成框架实现 , 我们手动实现
public class Test4 {
    public static void main(String[] args) {
        try {
            //反射 , Class可以获得类的全部信息 , 所有的东西
            Class clazz = Class.forName("com.annotation.Student");
            //获得这个类的注解
            Annotation[] annotations = clazz.getAnnotations();
            for (Annotation annotation:annotations){
                System.out.println(annotation);
            }

            //获得类的注解value的值
            TableKuang table = (TableKuang)
clazz.getAnnotation(TableKuang.class);
            System.out.println(table.value());

            //获得类指定注解的值
            Field name = clazz.getDeclaredField("name");
            FieldKuang fieldKuang = name.getAnnotation(FieldKuang.class);
            System.out.println(fieldKuang.columnName()+"--
>"+fieldKuang.type()
                +"-->"+fieldKuang.length());


            //我们可以根据得到的类的信息 , 通过JDBC生成相关的SQL语句,执行就可以动态生
成数据库表

        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
        }
    }
}

@TableKuang("db_student") //假设数据库表名为db_student .
class Student{

    @FieldKuang(columnName = "db_id",type="int",length = 10)
    private int id;
```

```java
    @FieldKuang(columnName = "db_name",type="varchar",length = 10)
    private String name;
    @FieldKuang(columnName = "db_age",type="int",length = 3)
    private int age;

    public Student() {
    }

    public Student(int id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}


//表名注解 , 只有一个参数 , 建议使用value命名
@Target(value = {ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
@interface TableKuang{
    String value();
}

//属性注解
@Target(value = {ElementType.FIELD})  //注意字段
```

```java
@Retention(value = RetentionPolicy.RUNTIME)
@interface FieldKuang{
    String columnName(); //列名
    String type(); //类型
    int length();//长度
}
```

# 反射机制Reflection

## 1、静态 VS 动态语言

- 动态语言
  - 是一类在运行时可以改变其结构的语言：例如新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化。通俗点说就是在运行时代码可以根据某些条件改变自身结构。
  - 主要动态语言：Object-C、C#、JavaScript、PHP、Python等。

```javascript
//体现动态语言的代码
function test() {
    var x = "var a=3;var b=5;alert(a+b)";
    eval(x);
}
```

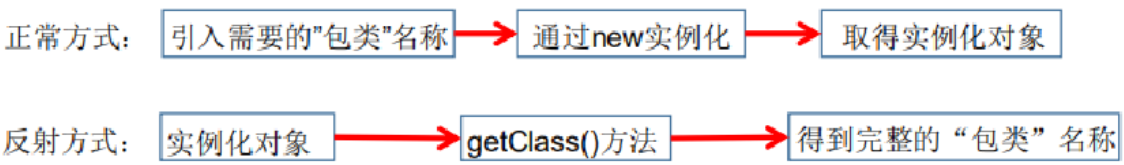
- 静态语言
  - 与动态语言相对应的，运行时结构不可变的语言就是静态语言。如Java、C、C++。
  - Java不是动态语言，但Java可以称之为“准动态语言”。即Java有一定的动态性，我们可以利用反射机制获得类似动态语言的特性。Java的动态性让编程的时候更加灵活！

## 2、Java Reflection

Reflection（反射）是Java被视为动态语言的关键，反射机制允许程序在执行期借助于Reflection API取得任何类的内部信息，并能直接操作任意对象的内部属性及方法。

```java
Class c = Class.forName("java.lang.String")
```

加载完类之后，在堆内存的方法区中就产生了一个Class类型的对象（一个类只有一个Class对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象看到类的结构。这个对象就像一面镜子，透过这个镜子看到类的结构，所以，我们形象的称之为：**反射**




```java
package com.reflection;

public class  Test2 {
    public static void main(String[] args) {
        try {
```

```
            //通过反射获取类的Class
            //--->查看JDK帮助文档
            Class<?> c1 = Class.forName("com.reflection.User");
            //一个类被加载后 , 类的整个结构信息会被放到对应的Class对象中
            System.out.println(c1);

            //一个类只对应一个Class对象
            Class<?> c2 = Class.forName("com.reflection.User");
            System.out.println(c1.hashCode());
            System.out.println(c2.hashCode());

        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

//1. 创建一个实体类
class User{
    private int id;
    private int age;
    private String name;

    public User() {
    }

    public User(int id, int age, String name) {
        this.id = id;
        this.age = age;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
```

```
        return "User{" +
                "id=" + id +
                ", age=" + age +
                ", name=" + name +
                '}';
    }
}
```

**Java反射机制提供的功能**

- 在运行时判断任意一个对象所属的类
- 在运行时构造任意一个类的对象
- 在运行时判断任意一个类所具有的成员变量和方法
- 在运行时获取泛型信息
- 在运行时调用任意一个对象的成员变量和方法
- 在运行时处理注解
- 生成动态代理
- ……

**Java反射优点和缺点**

优点：可以实现动态创建对象和编译，体现出很大的灵活性！

缺点：对性能有影响。使用反射基本上是一种解释操作，我们可以告诉JVM，我们希望做什么并且它满足我们的要求。这类操作总是慢于 直接执行相同的操作。

# 3、反射相关的主要API

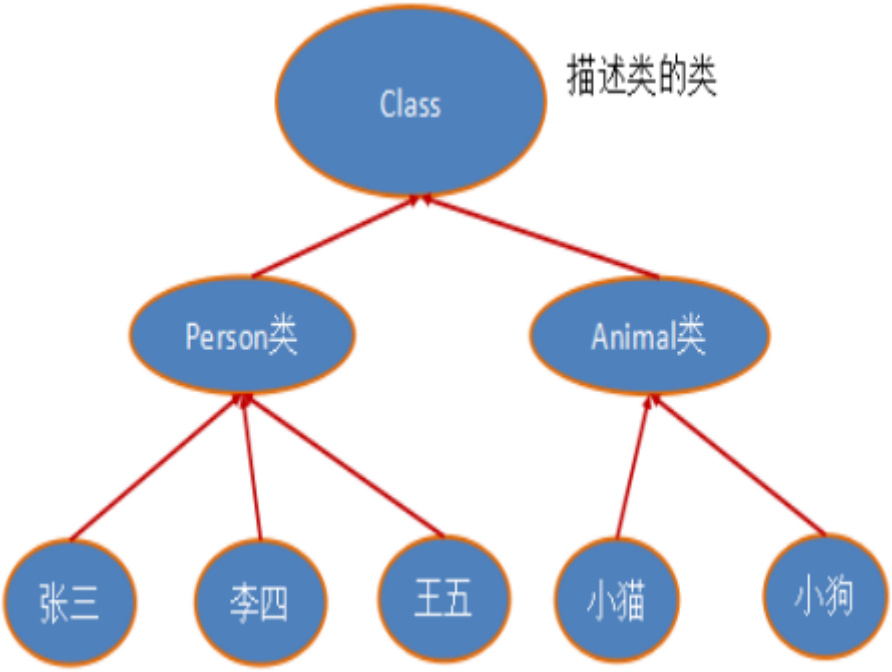
- java.lang.Class : 代表一个类
- java.lang.reflect.Method : 代表类的方法
- java.lang.reflect.Field : 代表类的成员变量
- java.lang.reflect.Constructor : 代表类的构造器
- .....

# 4、Class类

在Object类中定义了以下的方法，此方法将被所有子类继承

```
public final Class getClass();
```

以上的方法返回值的类型是一个Class类，此类是Java反射的源头，实际上所谓反射从程序的运行结果来看也很好理解，即：可以通过对象反射求出类的名称。




对象照镜子后可以得到的信息：某个类的属性、方法和构造器、某个类到底实现了哪些接口。对于每个类而言，JRE 都为其保留一个不变的 Class 类型的对象。一个 Class 对象包含了特定某个结构(class/interface/enum/annotation/primitive type/void/[])的有关信息。

- Class 本身也是一个类
- Class 对象只能由系统建立对象
- 一个加载的类在 JVM 中只会有一个Class实例
- 一个Class对象对应的是一个加载到JVM中的一个.class文件
- 每个类的实例都会记得自己是由哪个 Class 实例所生成
- 通过Class可以完整地得到一个类中的所有被加载的结构
- Class类是Reflection的根源，针对任何你想动态加载、运行的类，唯有先获得相应的Class对象

```java
package com.reflection;

//测试各种类型获得Class对象的方式
public class Test3 {
    public static void main(String[] args) throws ClassNotFoundException {
        Person person = new Student();

        System.out.println("这个人是:"+person.name);

        //获得class办法一:通过对象获得
        Class clazz1 = person.getClass();

        //获得class办法二:通过字符串获得(包名+类名)
        Class clazz2 = Class.forName("com.reflection.Student");

        //获得class办法三:通过类的静态成员class获得
        Class clazz3 = Person.class;

        //获得class办法四:只针对内置的基本数据类型
        Class clazz4 = Integer.TYPE;

        //获得父类类型
        Class clazz5 = clazz2.getSuperclass();
```

```
24
25          System.out.println(clazz1);
26          System.out.println(clazz2);
27          System.out.println(clazz3);
28          System.out.println(clazz4);
29          System.out.println(clazz5);
30
31      }
32  }
33
34  class Person {
35      public String name;
36
37      public Person() {
38      }
39
40      public Person(String name) {
41          this.name = name;
42      }
43
44      @Override
45      public String toString() {
46          return "Person{" +
47                  "name='" + name + '\'' +
48                  '}';
49      }
50  }
51
52  class Student extends Person{
53      public Student(){
54          this.name = "学生";
55      }
56  }
57
58  class Teacher extends Person{
59      public Teacher(){
60          this.name = "老师";
61      }
62  }
```

**Class类的常用方法**

| 方法名 | 功能说明 |
| --- | --- |
| static ClassforName(String name) | 返回指定类名name的Class对象 |
| Object newInstance() | 调用缺省构造函数，返回Class对象的一个实例 |
| getName() | 返回此Class对象所表示的实体（类，接口，数组类或void）的名称。 |
| Class getSuperClass() | 返回当前Class对象的父类的Class对象 |
| Class[] getinterfaces() | 获取当前Class对象的接口 |
| ClassLoader getClassLoader() | 返回该类的类加载器 |
| Constructor[] getConstructors() | 返回一个包含某些Constructor对象的数组 |
| Method getMothed(String name,Class.. T) | 返回一个Method对象，此对象的形参类型为paramType |
| Field[] getDeclaredFields() | 返回Field对象的一个数组 |

**获取Class类的实例**

a）若已知具体的类，通过类的class属性获取，该方法最为安全可靠，程序性能最高。

```
Class clazz = Person.class;
```

b）已知某个类的实例，调用该实例的getClass()方法获取Class对象

```
Class clazz = person.getClass();
```

c）已知一个类的全类名，且该类在类路径下，可通过Class类的静态方法forName()获取，可能抛出ClassNotFoundException

```
Class clazz = Class.forName("demo01.Student");
```

d）内置基本数据类型可以直接用类名.Type
e）还可以利用ClassLoader我们之后讲解

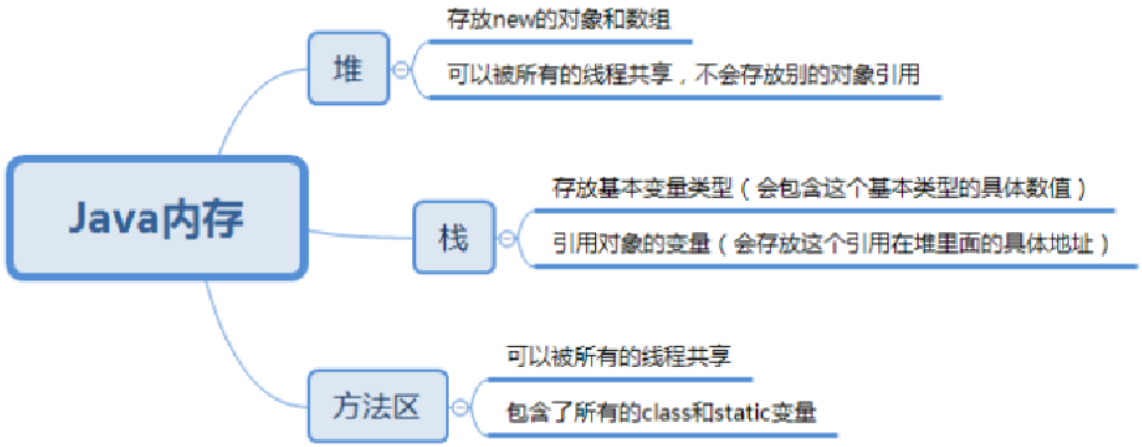
**哪些类型可以有Class对象?**

- class：外部类，成员(成员内部类，静态内部类)，局部内部类，匿名内部类。
- interface：接口
- []：数组
- enum：枚举
- annotation：注解@interface
- primitive type：基本数据类型
- void

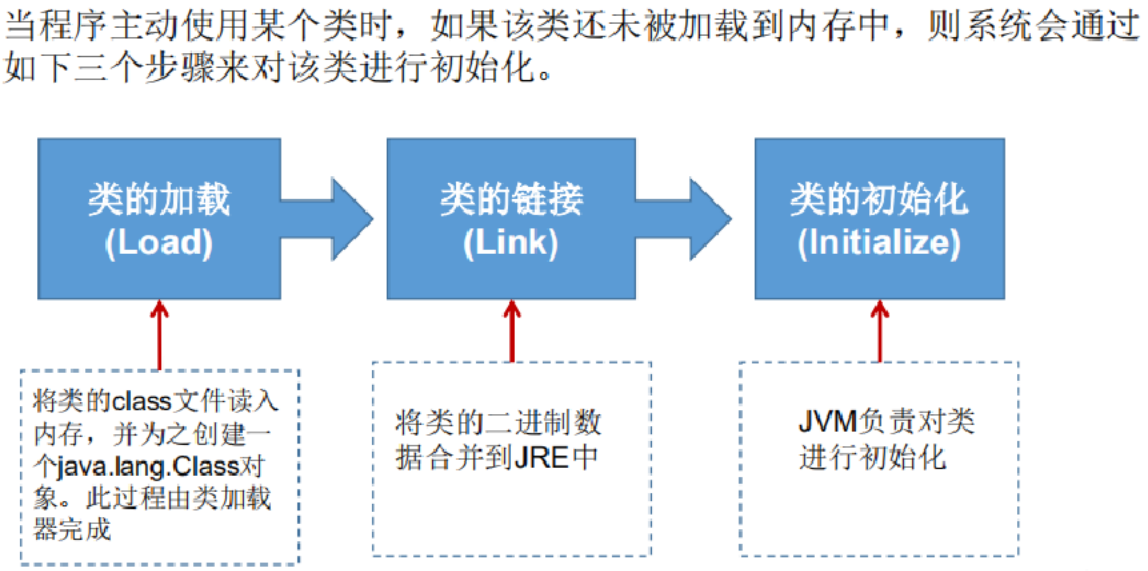
```
package com.reflection;

import java.lang.annotation.ElementType;

//演示 : 所有类型的class
```

```java
public class Test4 {
    public static void main(String[] args) {
        Class c1 = Object.class;
        Class c2 = Comparable.class;
        Class c3 = String[].class;
        Class c4 = int[][].class;
        Class c5 = ElementType.class;
        Class c6 = Override.class;
        Class c7 = Integer.class;
        Class c8 = void.class;
        Class c9 = Class.class;

        int[] a = new int[10];
        int[] b = new int[100];
        Class c10 = a.getClass();
        Class c11 = b.getClass();

        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
        System.out.println(c4);
        System.out.println(c5);
        System.out.println(c6);
        System.out.println(c7);
        System.out.println(c8);
        System.out.println(c9);
        System.out.println(c10);
        System.out.println(c11);
        //只要元素类型与维度一样,就是同一个Class
        System.out.println(c11==c10);


    }
}
```

## 5、Java内存分析



**类的加载过程**




**类的加载与ClassLoader的理解**

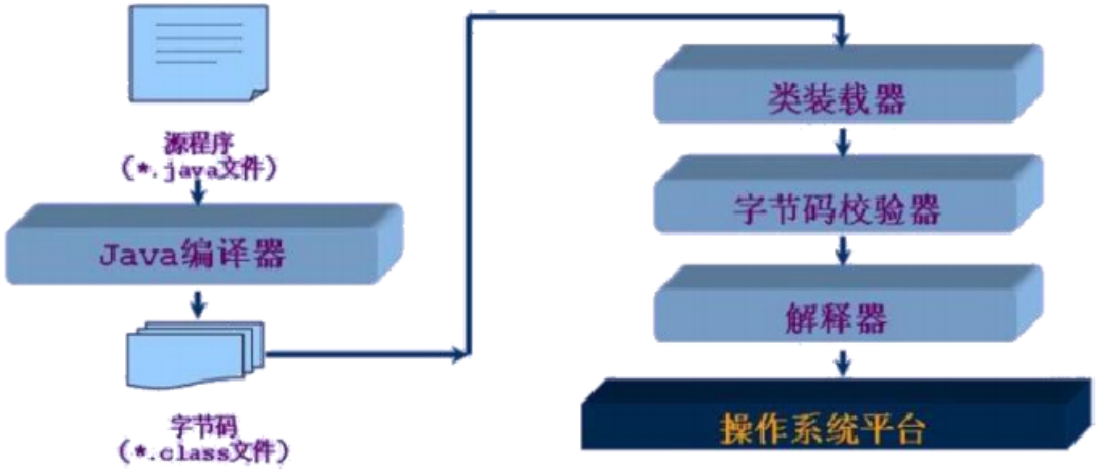
- 加载：
  - 将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区的运行时数据结构，然后生成一个代表这个类的java.lang.Class对象.
- 链接：将Java类的二进制代码合并到JVM的运行状态之中的过程。
  - 验证：确保加载的类信息符合JVM规范，没有安全方面的问题
  - 准备：正式为类变量（static）分配内存并设置类变量默认初始值的阶段，这些内存都将在方法区中进行分配。
  - 解析：虚拟机常量池内的符号引用（常量名）替换为直接引用（地址）的过程。
- 初始化：
  - 执行类构造器()方法的过程。类构造器()方法是由编译期自动收集类中所有类变量的赋值动作和静态代码块中的语句合并产生的。（类构造器是构造类信息的，不是构造该类对象的构造器）。
  - 当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。
  - 虚拟机会保证一个类的()方法在多线程环境中被正确加锁和同步。

**什么时候会发生类初始化?**

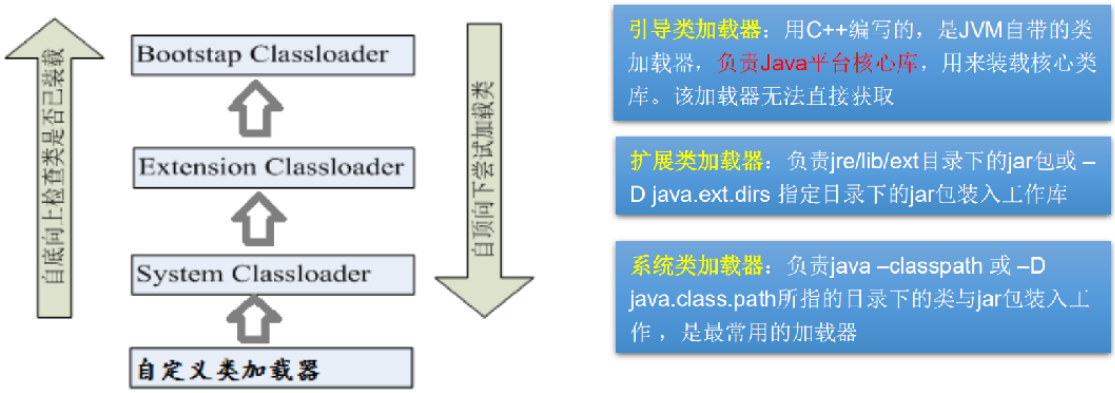
- 类的主动引用（一定会发生类的初始化）
  - 当虚拟机启动，先初始化main方法所在的类
  - new一个类的对象
  - 调用类的静态成员（除了final常量）和静态方法
  - 使用java.lang.reflect包的方法对类进行反射调用
  - 当初始化一个类，如果其父类没有被初始化，则先会初始化它的父类
- 类的被动引用（不会发生类的初始化）
  - 当访问一个静态域时，只有真正声明这个域的类才会被初始化。如：当通过子类引用父类的静态变量，不会导致子类初始化
  - 通过数组定义类引用，不会触发此类的初始化
  - 引用常量不会触发此类的初始化（常量在链接阶段就存入调用类的常量池中了）

**类加载器的作用**

- 类加载的作用：将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区的运行时数据结构，然后在堆中生成一个代表这个类的java.lang.Class对象，作为方法区中类数据的访问入口。
- 类缓存：标准的JavaSE类加载器可以按要求查找类，但一旦某个类被加载到类加载器中，它将维持加载（缓存）一段时间。不过JVM垃圾回收机制可以回收这些Class对象




- 类加载器作用是用来把类(class)装载进内存的。JVM 规范定义了如下类型的类的加载器




# 6、创建运行时类的对象

通过反射获取运行时类的完整结构
Field、Method、Constructor、Superclass、Interface、Annotation

- 实现的全部接口
- 所继承的父类
- 全部的构造器
- 全部的方法
- 全部的Field
- 注解
- 。。。

**小结**

- 在实际的操作中，取得类的信息的操作代码，并不会经常开发。
- 一定要熟悉java.lang.reflect包的作用，反射机制。

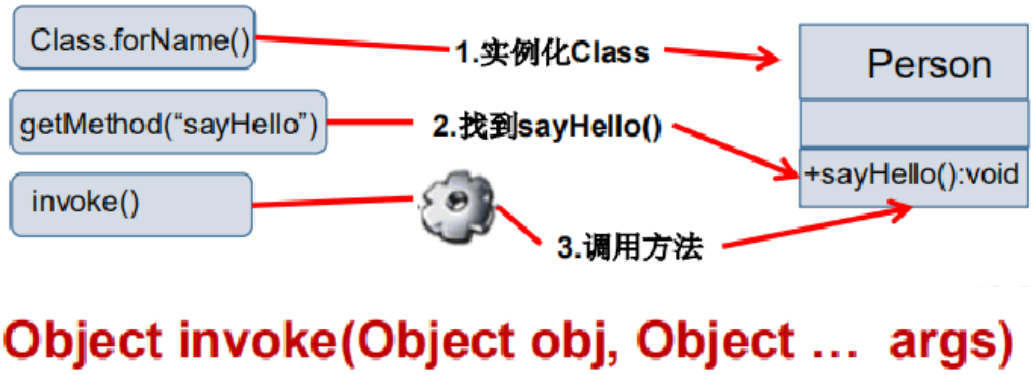
- 如何取得属性、方法、构造器的名称，修饰符等。

# 7、有了Class对象,能做什么?

- 创建类的对象：调用Class对象的newInstance()方法
  - 类必须有一个无参数的构造器。
  - 类的构造器的访问权限需要足够
- 思考？难道没有无参的构造器就不能创建对象了吗？只要在操作的时候明确的调用类中的构造器，并将参数传递进去之后，才可以实例化操作。
- 步骤如下：
  - 通过Class类的getDeclaredConstructor(Class ... parameterTypes)取得本类的指定形参类型的构造器
  - 向构造器的形参中传递一个对象数组进去，里面包含了构造器中所需的各个参数。
  - 通过Constructor实例化对象

**调用指定的方法**

- 通过反射，调用类中的方法，通过Method类完成。
  - 通过Class类的getMethod(String name,Class...parameterTypes)方法取得一个Method对象，并设置此方法操作时所需要的参数类型。
  - 之后使用Object invoke(Object obj, Object[] args)进行调用，并向方法中传递要设置的obj对象的参数信息。




  - Object 对应原方法的返回值，若原方法无返回值，此时返回null
  - 若原方法若为静态方法，此时形参Object obj可为null
  - 若原方法形参列表为空，则Object[] args为null
  - 若原方法声明为private,则需要在调用此invoke()方法前，显式调用方法对象的setAccessible(true)方法，将可访问private的方法。

# 8、setAccessible

- Method和Field、Constructor对象都有setAccessible()方法。
- setAccessible作用是启动和禁用访问安全检查的开关。
- 参数值为true则指示反射的对象在使用时应该取消Java语言访问检查。
- 提高反射的效率。如果代码中必须用反射，而该句代码需要频繁的被调用，那么请设置为true。

- 使得原本无法访问的私有成员也可以访问
- 参数值为false则指示反射的对象应该实施Java语言访问检查

## 9、反射操作泛型

- Java采用泛型擦除的机制来引入泛型 , Java中的泛型仅仅是给编译器javac使用的,确保数据的安全性和免去强制类型转换问题 , 但是 , 一旦编译完成 , 所有和泛型有关的类型全部擦除
- 为了通过反射操作这些类型 , Java新增了 ParameterizedType , GenericArrayType , TypeVariable 和 WildcardType 几种类型来代表不能被归一到Class类中的类型但是又和原始类型齐名的类型.
- ParameterizedType : 表示一种参数化类型,比如Collection
- GenericArrayType : 表示一种元素类型是参数化类型或者类型变量的数组类型
- TypeVariable : 是各种类型变量的公共父接口
- WildcardType : 代表一种通配符类型表达式

**反射操作注解**

- getAnnotations
- getAnnotation

## 10、练习:ORM

了解什么是ORM ?
Object relationship Mapping --> 对象关系映射

```
class Student{
    int id;
    String name;
    int age;
}
```

| id | name | age |
| --- | --- | --- |
| 001 | 秦疆 | 3 |
| 002 | 老沈 | 30 |

类和表结构对应
属性和字段对应
对象和记录对应

要求 : 利用注解和反射完成类和表结构的映射关系