

master

...

Java-Tutorial / docs / java / basic / 23、继承、封装、多态的实现原理.md



h2pl change config



1 contributor

Raw

Blame



544 lines (384 sloc) | 24.2 KB

目录

- [从JVM结构开始谈多态](#)
 - [JVM 的结构](#)
 - [Java 的方法调用方式](#)
 - [常量池 \(constant pool\)](#)
 - [图 2. 常量池各表的关系](#)
 - [方法表与方法调用](#)
 - [清单 1](#)
 - [接口调用](#)
 - [\[图 5.Dancer 的方法表 \(查看大图\)\]\(#图-5dancer-的方法表 \(\[查看大图\]httpswwwibmcomdeveloperworkscnjavaj-lo-polymorphimage011jpg\) \)](#)
- [继承的实现原理](#)
- [重载和重写的实现原理](#)
- [参考文章](#)
- [微信公众号](#)
 - [Java技术江湖](#)
 - [个人公众号：黄小斜](#)

title: 夯实Java基础系列23： 深入理解Java继承、封装、多态的底层实现原理 **date: 2019-9-23 15:56:26 # 文章生成时间，一般不改** **categories: - Java技术江湖 - Java基础** **tags: - 继承 - 封装 - 多态**

本系列文章将整理到我在GitHub上的《Java面试指南》仓库，更多精彩内容请到我的仓库里查看

<https://github.com/h2pl/Java-Tutorial>

喜欢的话麻烦点下Star哈

文章首发于我的个人博客：

www.how2playlife.com

本文是微信公众号【Java技术江湖】的《夯实Java基础系列博文》其中一篇，本文部分内容来源于网络，为了把本文主题讲得清晰透彻，也整合了很多我认为不错的技术博客内容，引用其中了一些比较好的博客文章，如有侵权，请联系作者。该系列博文会告诉你如何从入门到进阶，一步步地学习Java基础知识，并上手进行实战，接着了解每个Java知识点背后的实现原理，更完整地理解整个Java技术体系，形成自己的知识框架。为了更好地总结和检验你的学习成果，本系列文章也会提供每个知识点对应的面试题以及参考答案。

如果对本系列文章有什么建议，或者是有什么疑问的话，也可以关注公众号【Java技术江湖】联系作者，欢迎你参与本系列博文的创作和修订。

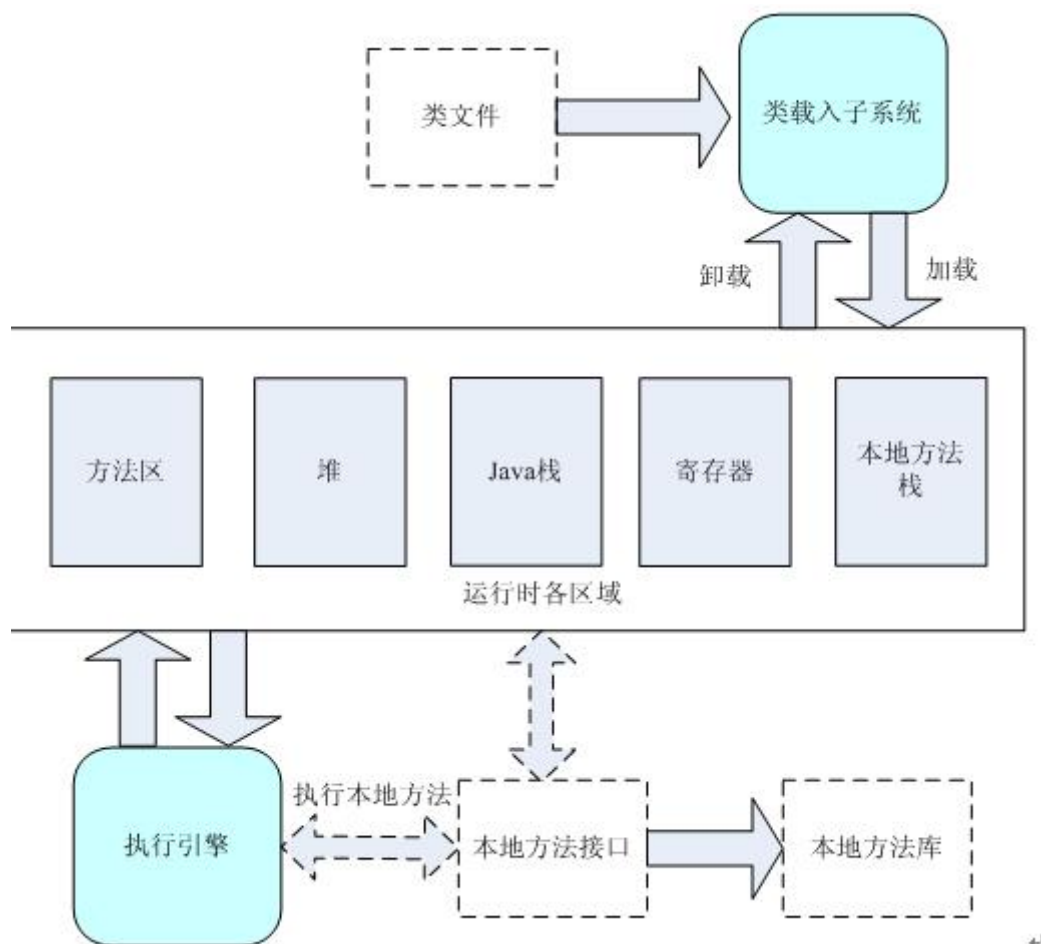
从JVM结构开始谈多态

Java 对于方法调用动态绑定的实现主要依赖于方法表，但通过类引用调用和接口引用调用的实现则有所不同。总体而言，当某个方法被调用时，JVM 首先要查找相应的常量池，得到方法的符号引用，并查找调用类的方法表以确定该方法的直接引用，最后才真正调用该方法。以下分别对该过程中涉及到的相关部分做详细介绍。

JVM 的结构

典型的 Java 虚拟机的运行时结构如下图所示

图 1.JVM 运行时结构



此结构中，我们只探讨和本文密切相关的方法区 (method area)。当程序运行需要某个类的定义时，载入子系统 (class loader subsystem) 装入所需的 class 文件，并在内部建立该类的类型信息，这个类型信息就存贮在方法区。类型信息一般包括该类的方法代码、类变量、成员变量的定义等等。可以说，类型信息就是类的 Java 文件在运行时的内部结构，包含了改类的所有在 Java 文件中定义的信息。

注意到，该类型信息和 class 对象是不同的。class 对象是 JVM 在载入某个类后于堆 (heap) 中创建的代表该类的对象，可以通过该 class 对象访问到该类型信息。比如最典型的应用，在 Java 反射中应用 class 对象访问到该类支持的所有方法，定义的成员变量等等。可以想象，JVM 在类型信息和 class 对象中维护着它们彼此的引用以便互相访问。两者的关系可以类比于进程对象与真正的进程之间的关系。

Java 的方法调用方式

Java 的方法调用有两类，动态方法调用与静态方法调用。静态方法调用是指对于类的静态方法的调用方式，是静态绑定的；而动态方法调用需要有方法调用所作用的对象，是动态绑定的。类调用 (invokestatic) 是在编译时刻就已经确定好具体调用方法的情况，而实例调用 (invokevirtual) 则是在调用的时候才确定具体的调用方法，这就是动态绑定，也是多态要解决的核心问题。

JVM 的方法调用指令有四个，分别是 invokestatic, invokespecial, invokesvirtual 和 invokeinterface。前两个是静态绑定，后两个是动态绑定的。本文也可以说是对于 JVM 后两种调用实现的考察。

常量池 (constant pool)

常量池中保存的是一个 Java 类引用的一些常量信息，包含一些字符串常量及对于类的符号引用信息等。Java 代码编译生成的类文件中的常量池是静态常量池，当类被载入到虚拟机内部的时候，在内存中产生类的常量池叫运行时常量池。

常量池在逻辑上可以分成多个表，每个表包含一类的常量信息，本文只探讨对于 Java 调用相关的常量池表。

CONSTANT_Utf8_info

字符串常量表，该表包含该类所使用的所有字符串常量，比如代码中的字符串引用、引用的类名、方法的名字、其他引用的类与方法的字符串描述等等。其余常量池表中所涉及到的任何常量字符串都被索引至该表。

CONSTANT_Class_info

类信息表，包含任何被引用的类或接口的符号引用，每一个条目主要包含一个索引，指向 CONSTANT_Utf8_info 表，表示该类或接口的全限定名。

CONSTANT_NameAndType_info

名字类型表，包含引用的任意方法或字段的名称和描述符信息在字符串常量表中的索引。

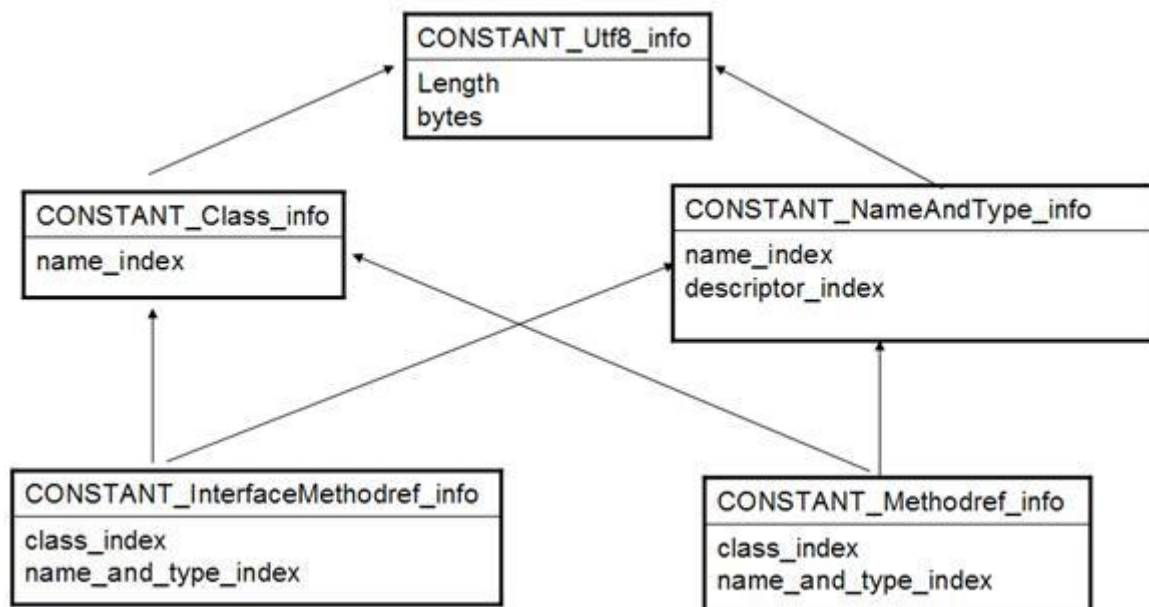
CONSTANT_InterfaceMethodref_info

接口方法引用表，包含引用的任何接口方法的描述信息，主要包括类信息索引和名字类型索引。

CONSTANT_Methodref_info

类方法引用表，包含引用的任何类型方法的描述信息，主要包括类信息索引和名字类型索引。

图 2. 常量池各表的关系



可以看到，给定任意一个方法的索引，在常量池中找到对应的条目后，可以得到该方法的类索引（class_index）和名字类型索引（name_and_type_index），进而得到该方法所属的类型信息和名称及描述符信息（参数，返回值等）。注意到所有的常量字符串都是存储在 CONSTANT_Utf8_info 中供其他表索引的。

方法表与方法调用

方法表是动态调用的核心，也是 Java 实现动态调用的主要方式。它被存储于方法区中的类型信息，包含有该类型所定义的所有方法及指向这些方法代码的指针，注意这些具体的方法代码可能是被覆写的方法，也可能是继承自基类的方法。

如有类定义 Person, Girl, Boy,

清单 1

```

class Person {
    public String toString(){
        return "I'm a person.";
    }
    public void eat(){}
    public void speak(){}
}

class Boy extends Person{
    public String toString(){
        return "I'm a boy";
    }
    public void speak(){}
    public void fight(){}
}

class Girl extends Person{
    public String toString(){

```

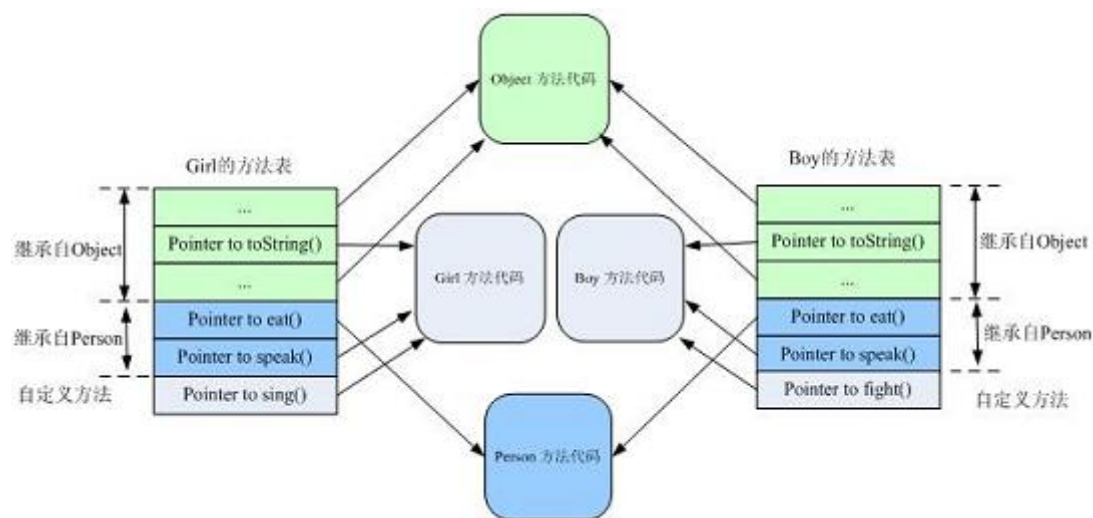
```

        return "I'm a girl";
    }
    public void speak(){}
    public void sing(){}
}

```

当这三个类被载入到 Java 虚拟机之后，方法区中就包含了各自的类的信息。Girl 和 Boy 在方法区中的方法表可表示如下：

图 3.Boy 和 Girl 的方法表



可以看到，Girl 和 Boy 的方法表包含继承自 Object 的方法，继承自直接父类 Person 的方法及各自新定义的方法。注意方法表条目指向的具体方法地址，如 Girl 的继承自 Object 的方法中，只有 toString() 指向自己的实现（Girl 的方法代码），其余皆指向 Object 的方法代码；其继承自于 Person 的方法 eat() 和 speak() 分别指向 Person 的方法实现和本身的实现。

Person 或 Object 的任意一个方法，在它们的方法表和其子类 Girl 和 Boy 的方法表中的位置 (index) 是一样的。这样 JVM 在调用实例方法其实只需要指定调用方法表中的第几个方法即可。

如调用如下：

清单 2

```

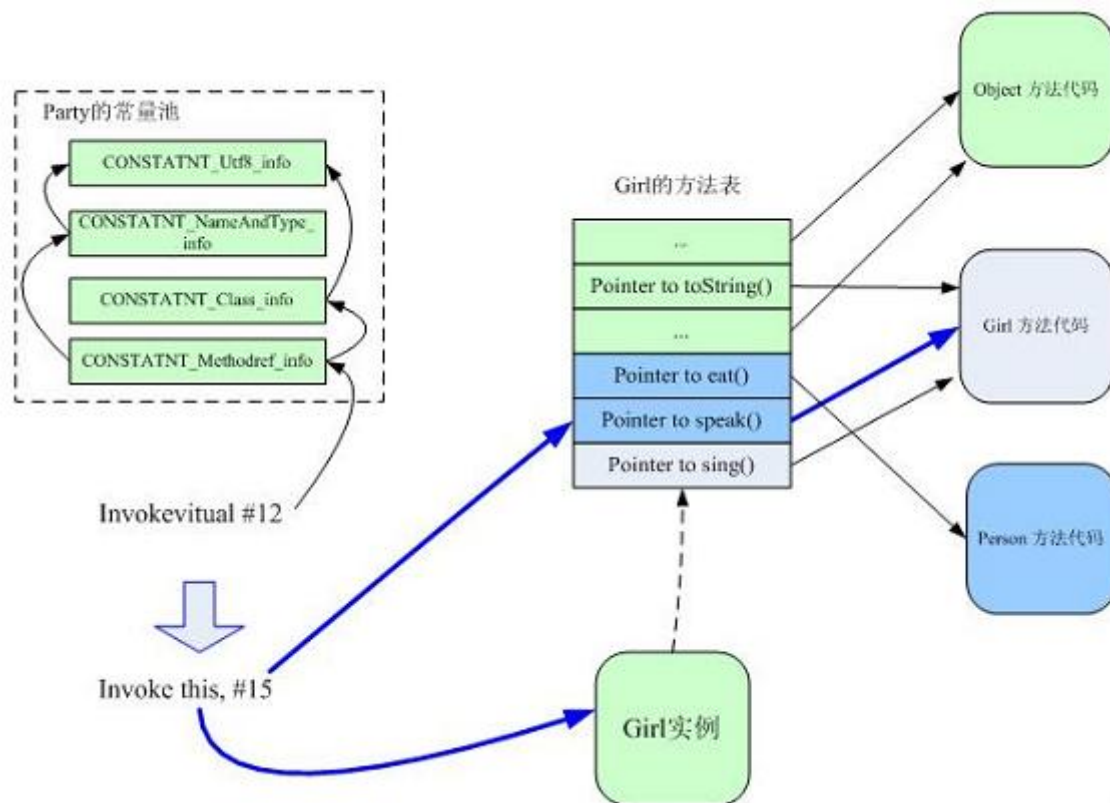
class Party{
...
    void happyHour(){
        Person girl = new Girl();
        girl.speak();
    }
}

```

当编译 Party 类的时候，生成 girl.speak() 的方法调用假设为：

设该调用代码对应着 `girl.speak()`; #12 是 `Party` 类的常量池的索引。JVM 执行该调用指令的过程如下所示：

图 4. 解析调用过程



JVM 首先查看 `Party` 的常量池索引为 12 的条目（应为 `CONSTANT_Methodref_info` 类型，可视为方法调用的符号引用），进一步查看常量池（`CONSTANT_Class_info`, `CONSTANT_NameAndType_info`, `CONSTANT_Utf8_info`）可得出要调用的方法是 `Person` 的 `speak` 方法（注意引用 `girl` 是其基类 `Person` 类型），查看 `Person` 的方法表，得出 `speak` 方法在该方法表中的偏移量 15（offset），这就是该方法调用的直接引用。

当解析出方法调用的直接引用后（方法表偏移量 15），JVM 执行真正的方法调用：根据实例方法调用的参数 `this` 得到具体的对象（即 `girl` 所指向的位于堆中的对象），据此得到该对象对应的方法表（`Girl` 的方法表），进而调用方法表中的某个偏移量所指向的方法（`Girl` 的 `speak()` 方法的实现）。

接口调用

因为 `Java` 类是可以同时实现多个接口的，而当用接口引用调用某个方法的时候，情况就有所不同了。`Java` 允许一个类实现多个接口，从某种意义上来说相当于多继承，这样同样的方法在基类和派生类的方法表的位置就可能不一样了。

```

interface IDance{
    void dance();
}

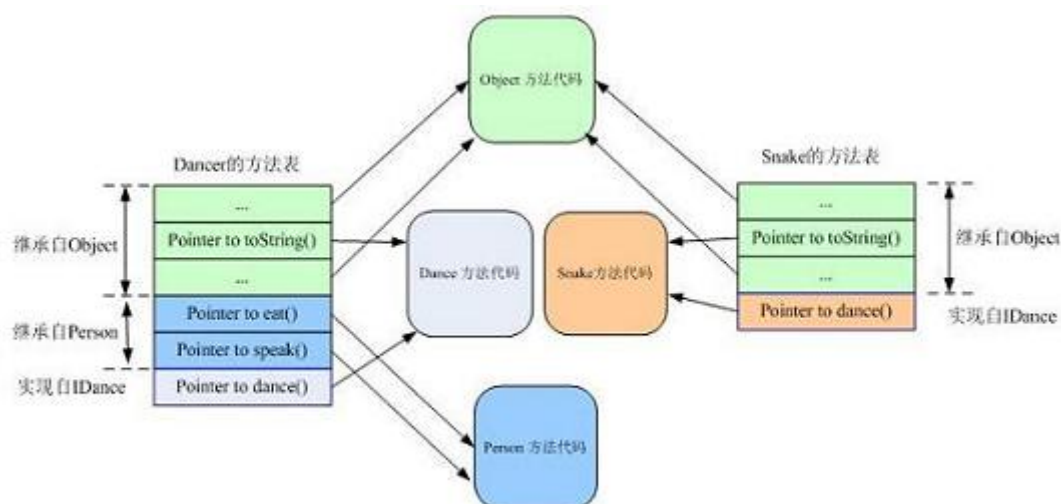
class Person {
    public String toString(){
        return "I'm a person.";
    }
    public void eat(){}
    public void speak(){}
}

class Dancer extends Person
implements IDance {
    public String toString(){
        return "I'm a dancer.";
    }
    public void dance(){}
}

class Snake implements IDance{
    public String toString(){
        return "A snake.";
    }
    public void dance(){
        //snake dance
    }
}

```

图 5.Dancer 的方法表 (查看大图)



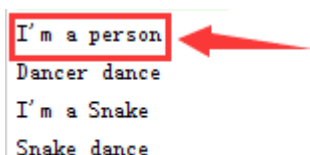
可以看到，由于接口的介入，继承自于接口 `IDance` 的方法 `dance()` 在类 `Dancer` 和 `Snake` 的方法表中的位置已经不一样了，显然我们无法通过给出方法表的偏移量来正确调用 `Dancer` 和 `Snake` 的这个方法。这也是 Java 中调用接口方法有其专有的调用指令 (`invokeinterface`) 的原因。

Java 对于接口方法的调用是采用搜索方法表的方式，对如下的方法调用

invokeinterface #13

JVM 首先查看常量池，确定方法调用的符号引用（名称、返回值等等），然后利用 this 指向的实例得到该实例的方法表，进而搜索方法表来找到合适的方法地址。

因为每次接口调用都要搜索方法表，所以从效率上来说，接口方法的调用总是慢于类方法的调用的。



执行结果如下：

可以看到

System.out.println(dancer); 调用的是Person的toString方法。

继承的实现原理

Java 的继承机制是一种复用类的技术，从原理上来说，是更好的使用了组合技术，因此要理解继承，首先需要了解类的组合技术是如何实现类的复用的。

使用组合技术复用类 假设现在的需求是要创建一个具有基本类型，String 类型以及一个其他非基本类型的对象。该如何处理呢？

对于基本类型的变量，在新类中成员变量处直接定义即可，但对于非基本类型变量，不仅需要在类中声明其引用，并且还需要手动初始化这个对象。

这里需要注意的是，编译器并不会默认将所有的引用都创建对象，因为这样的话在很多情况下会增加不必要的负担，因此，在合适的时机初始化合适的对象，可以通过以下几个位置做初始化操作：

在定义对象的地方，先于构造方法执行。在构造方法中。在正要使用之前，这个被称为惰性初始化。使用实例初始化。

```
class Soap {
    private String s;
    Soap() {
        System.out.println("Soap()");
        s = "Constructed";
    }
    public String tiString(){
        return s;
    }
}

public class Bath {
    // s1 初始化先于构造函数
    private String s1 = "Happy", s2 = "Happy", s3, s4;
```

```

private Soap soap;
private int i;
private float f;

public Both() {
    System.out.println("inSide Both");
    s3 = "Joy";
    f = 3.14f;
    soap = new Soap();
}

{
    i = 88;
}

public String toString() {
    if(s4 == null){
        s4 = "Joy"
    }
    return "s1 = " + s1 + "\n" +
           "s2 = " + s2 + "\n" +
           "s3 = " + s3 + "\n" +
           "s4 = " + s4 + "\n" +
           "i = " + i + "\n" +
           "f = " + f + "\n" +
           "soap = " + soap;
}
}

```

继承 Java 中的继承由 extend 关键字实现，组合的语法比较平实，而继承是一种特殊的语法。当一个类继承自另一个类时，那么这个类就可以拥有另一个类的域和方法。

```

class Cleanser{
    private String s = "Cleanser";

    public void append(String a){
        s += a;
    }
    public void apply(){
        append("apply");
    }
    public void scrub(){
        append("scrub");
    }
    public String toString(){
        return s;
    }
    public static void main(String args){
        Cleanser c = new Cleanser();

        c.apply();
        System.out.println(c);
    }
}

```

```

}

public class Deter extends Cleanser{
    public void apply(){
        append("Deter.apply");
        super.scrub();
    }
    public void foam(){
        append("foam");
    }
    public static void main(String args){
        Deter d = new Deter();

        d.apply();
        d.scrub();
        d.foam();
        System.out.println(d);
        Cleanser.main(args);
    }
}

```

上面的代码中，展示了继承语法中的一些特性：

子类可以直接使用父类中公共的方法和成员变量（通常为了保护数据域，成员变量均为私有）子类中可以覆盖父类中的方法，也就是子类重写了父类的方法，此时若还需要调用被覆盖的父类的方法，则需要用到 `super` 来指定是调用父类中的方法。子类中可以自定义父类中没有的方法。可以发现上面两个类中均有 `main` 方法，命令行中调用的哪个类就执行哪个类的 `main` 方法，例如：`java Deter`。继承语法的原理 接下来我们将通过创建子类对象来分析继承语法在我们看不到的地方做了什么样的操作。

可以先思考一下，如何理解使用子类创建的对象呢，首先这个对象中包含子类的所有信息，但是也包含父类的所有公共的信息。

下面来看一段代码，观察一下子类在创建对象初始化的时候，会不会用到父类相关的方法。

```

class Art{
    Art() {
        System.out.println("Art Construct");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing Construct");
    }
}

public class Cartoon extends Drawing {
    public Cartoon() {
        System.out.println("Cartoon construct");
    }
}

```

```

    }
    public void static main(String args) {
        Cartoon c = new Cartoon();
    }
}
/*output:
Art Construct
Drawing Construct
Cartoon construct
*/

```

通过观察代码可以发现，在实例化Cartoon时，事实上是从最顶层的父类开始向下逐个实例化，也就是最终实例化了三个对象。编译器会默认在子类的构造方法中增加调用父类默认构造方法的代码。

因此，继承可以理解为编译器帮我们完成了类的特殊组合技术，即在子类中存在一个父类的对象，使得我们可以用子类对象调用父类的方法。而在开发者看来只不过是使用了一个关键字。

注意：虽然继承很接近组合技术，但是继承拥有其他更多的区别于组合的特性，例如父类的对象我们是不可见的，对于父类中的方法也做了相应的权限校验等。

那么，如果类中的构造方法是带参的，该如何操作呢？（使用super关键字显示调用）

见代码：

```

class Game {
    Game(int i){
        System.out.println("Game Construct");
    }
}

class BoardGame extends Game {
    BoardGame(int j){
        super(j);
        System.out.println("BoardGame Construct");
    }
}

public class Chess extends BoardGame{
    Chess(){
        super(99);
        System.out.println("Chess construct");
    }
    public static void main(String args) {
        Chess c = new Chess();
    }
}
/*output:
Game Construct
BoardGame Construct

```

重载和重写的实现原理

刚开始学习Java的时候，就了解了Java这个比较有意思的特性：重写 和 重载。开始的有时候从名字上还总是容易弄混。我相信熟悉Java这门语言的同学都应该了解这两个特性，可能只是从语言层面上了解这种写法，但是jvm是如何实现他们的呢？

重载官方给出的介绍：

一. overload: The Java programming language supports overloading methods, and Java can distinguish between methods with different method signatures. This means that methods within a class can have the same name if they have different parameter lists .

Overloaded methods are differentiated by the number and the type of the arguments passed into the method.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

首先看一段代码，来看看代码的执行结果：

```
public class OverrideTest {  
  
    class Father{  
  
    class Sun extends Father {  
  
    public void doSomething(Father father){  
        System.out.println("Father do something");  
    }  
  
    public void doSomething(Sun father){  
        System.out.println("Sun do something");  
    }  
  
    public static void main(String [] args){  
        OverrideTest overrideTest = new OverrideTest();  
        Father sun = overrideTest.new Sun();  
        Father father = overrideTest.new Father();  
        overrideTest.doSomething(father);  
        overrideTest.doSomething(sun);  
    }  
}
```

```
}  
}
```

看下这段代码的执行结果，最后会打印：

Father do something Father do something

为什么会打印出这样的结果呢？首先要介绍两个概念：静态分派和动态分派

静态分派：依赖静态类型来定位方法执行版本的分派动作称为静态分派

动态分派：运行期根据实际类型确定方法执行版本的分派过程。

他们的区别是：

1. 静态分派发生在编译期，动态分派发生在运行期；
2. private,static,final 方法发生在编译期，并且不能被重写，一旦发生了重写，将会在运行期处理。
3. 重载是静态分派，重写是动态分派

回到上面的问题，因为重载是发生在编译期，所以在编译期已经确定两次 doSomething 方法的参数都是Father类型，在class文件中已经指向了Father类的符号引用，所以最后会打印两次Father do something。

二. override: An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method that it overrides. An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a covariant return type.

还是上面那个代码，稍微改动下

```
public class OverrideTest {  
  
    class Father{  
  
    class Sun extends Father {  
  
    public void doSomething(){  
        System.out.println("Father do something");  
    }  
}
```



```
public void doSomething(){
    System.out.println("Sun do something");
}

public static void main(String [] args){
    OverrideTest overrideTest = new OverrideTest();
    Father sun = overrideTest.new Sun();
    Father father = overrideTest.new Father();
    overrideTest.doSomething();
    overrideTest.doSomething();
}
}
```

最后会打印：

Father do something

Sun do something

相信大家都会知道这个结果，那么这个结果jvm是怎么实现的呢？

在编译期，只会识别到是调用Father类的doSomething方法，到运行期才会真正找到对象的实际类型。

首先该方法的执行，jvm会调用invokevirtual指令，该指令会找栈顶第一个元素所指向的对象的实际类型，如果该类型存在调用的方法，则会走验证流程，否则继续找其父类。这也是为什么子类可以直接调用父类具有访问权限的方法的原因。简而言之，就是在运行期才会去确定对象的实际类型，根据这个实际类型确定方法执行版本，这个过程称为动态分派。override 的实现依赖jvm的动态分派。

参考文章

https://blog.csdn.net/dj_dengjian/article/details/80811348

<https://blog.csdn.net/chenssy/article/details/12757911>

<https://blog.csdn.net/fan2012huan/article/details/51007517>

<https://blog.csdn.net/fan2012huan/article/details/50999777>

<https://www.cnblogs.com/serendipity-fly/p/9469289.html>

https://blog.csdn.net/m0_37264516/article/details/86709537

微信公众号

Java技术江湖