Scala第八章节

章节目标

1. 能够使用trait独立完成适配器, 模板方法, 职责链设计模式
2. 能够独立叙述trait的构造机制
3. 能够了解trait继承class的写法
4. 能够独立完成程序员案例

---

1. 特质入门

1.1 概述

有些时候, 我们会遇到一些特定的需求, 即: 在不影响当前继承体系的情况下, 对某些类(或者某些对象)的功能进行加强, 例如: 有猴子类和大象类, 它们都有姓名, 年龄, 以及吃的功能, 但是部分的猴子经过马戏团的训练后, 学会了骑独轮车. 那骑独轮车这个功能就不能定义到父类(动物类)或者猴子类中, 而是应该定义到`特质`中. 而Scala中的特质, 要用关键字`trait`修饰.

1.2 特点

- 特质可以提高代码的复用性.
- 特质可以提高代码的扩展性和可维护性.
- 类与特质之间是继承关系, 只不过类与类之间只支持`单继承`, 但是类与特质之间, `既可以单继承, 也可以多继承`.
- Scala的特质中可以有普通字段, 抽象字段, 普通方法, 抽象方法.

> 注意:
>
> 1. 如果特质中只有抽象内容, 这样的特质叫: 瘦接口.
> 2. 如果特质中既有抽象内容, 又有具体内容, 这样的特质叫: 富接口.

1.3 语法

定义特质

```
trait 特质名称 {
    // 普通字段
    // 抽象字段

    // 普通方法
    // 抽象方法
}
```

继承特质

```
class 类 extends 特质1 with 特质2 {
    // 重写抽象字段
    // 重写抽象方法
}
```

注意

- scala中不管是类还是特质, 继承关系用的都是`extends`关键字
- 如果要继承多个特质(trait)，则特质名之间使用`with`关键字隔开

1.4 示例: 类继承单个特质

需求

1. 创建一个Logger特质，添加`log(msg:String)`方法
2. 创建一个ConsoleLogger类，继承Logger特质，实现log方法，打印消息
3. 添加main方法，创建ConsoleLogger对象，调用log方法.

参考代码

```
//案例: Trait入门之类继承单个特质
object ClassDemo01 {
  //1. 定义一个特质.
  trait Logger {
    def log(msg:String)   //抽象方法
  }

  //2. 定义一个类, 继承特质.
  class ConsoleLogger extends Logger {
    override def log(msg: String): Unit = println(msg)
  }

  def main(args: Array[String]): Unit = {
    //3. 调用类中的方法
    val cl = new ConsoleLogger
    cl.log("trait入门: 类继承单个特质")
  }
}
```

1.5 示例: 类继承多个trait

需求

1. 创建一个MessageSender特质，添加`send(msg:String)`方法
2. 创建一个MessageReceiver特质，添加`receive()`方法
3. 创建一个MessageWorker类, 继承这两个特质, 重写上述的两个方法
4. 在main中测试，分别调用send方法、receive方法

参考代码

```
//案例: 类继承多个trait
object ClassDemo02 {
  //1. 定义一个特质: MessageSender, 表示发送信息.
  trait MessageSender {
    def send(msg:String)
  }
  //2. 定义一个特质: MessageReceiver, 表示接收信息.
  trait MessageReceiver {
    def receive()
  }
  //3. 定义一个类MessageWorker, 继承两个特质.
  class MessageWorker extends MessageSender with MessageReceiver {
```

```
    override def send(msg: String): Unit = println("发送消息: " + msg)

    override def receive(): Unit = println("消息已收到, 我很好, 谢谢!...")
  }

  //main方法, 作为程序的主入口
  def main(args: Array[String]): Unit = {
    //4. 调用类中的方法
    val mw = new MessageWorker
    mw.send("Hello, 你好啊!")
    mw.receive()
  }
}
```

1.6 示例: object继承trait

需求

1. 创建一个Logger特质，添加 `log(msg:String)` 方法
2. 创建一个Warning特质, 添加 `warn(msg:String)` 方法
3. 创建一个单例对象ConsoleLogger，继承Logger和Warning特质, 重写特质中的抽象方法
4. 编写main方法，调用单例对象ConsoleLogger的log和warn方法

参考代码

```
//案例: 演示object单例对象继承特质
object ClassDemo03 {
  //1. 定义一个特质Logger, 添加log(msg:String)方法.
  trait Logger{
    def log(msg:String)
  }

  //2. 定义一个特质Warning, 添加warn(msg:String)方法.
  trait Warning{
    def warn(msg:String)
  }

  //3. 定义单例对象ConsoleLogger, 继承上述两个特质, 并重写两个方法.
  object ConsoleLogger extends Logger with Warning{
    override def log(msg: String): Unit = println("控制台日志信息: "  + msg)

    override def warn(msg: String): Unit = println("控制台警告信息: " + msg)
  }

  //main方法, 作为程序的入口
  def main(args: Array[String]): Unit = {
    //4. 调用ConsoleLogger单例对象中的两个方法.
    ConsoleLogger.log("我是一条普通日志信息!")
    ConsoleLogger.warn("我是一条警告日志信息!")
  }
}
```

1.7 示例: 演示trait中的成员

需求

1. 定义一个特质Hero, 添加具体字段name(姓名), 抽象字段arms(武器), 具体方法eat(), 抽象方法toWar()
2. 定义一个类Generals, 继承Hero特质, 重写其中所有的抽象成员.
3. 在main方法中, 创建Generals类的对象, 调用其中的成员.

参考代码

```
//案例: 演示特质中的成员
object ClassDemo04 {
  //1. 定义一个特质Hero
  trait Hero{
    var name = ""      //具体字段
    var arms:String    //抽象字段

    //具体方法
    def eat() = println("吃肉喝酒, 养精蓄锐!")

    //抽象方法
    def toWar():Unit
  }

  //2. 定义一个类Generals, 继承Hero特质, 重写其中所有的抽象成员.
  class Generals extends Hero {
    //重写父特质中的抽象字段
    override var arms: String = ""

    //重写父特质中的抽象方法
    override def toWar(): Unit = println(s"${name}带着${arms}, 上阵杀敌!")
  }

  //main方法, 作为程序的入口
  def main(args: Array[String]): Unit = {
    //3. 创建Generals类的对象.
    val gy = new Generals

    //4. 测试Generals类中的内容.
    //给成员变量赋值
    gy.name = "关羽"
    gy.arms = "青龙偃月刀"
    //打印成员变量值
    println(gy.name, gy.arms)
    //调用成员方法
    gy.eat()
    gy.toWar()
  }
}
```

2. 对象混入trait

有些时候, 我们希望在不改变类继承体系的情况下, 对对象的功能进行临时增强或者扩展, 这个时候就可以考虑使用`对象混入`技术了. 所谓的`对象混入`指的就是: 在scala中, 类和特质之间无任何的继承关系, 但是通过特定的关键字, 却可以让该类对象具有指定特质中的成员.

2.1 语法

```
val/var 对象名 = new 类 with 特质
```

2.2 示例

需求

1. 创建Logger特质, 添加log(msg:String)方法
2. 创建一个User类, 该类和Logger特质之间无任何关系.
3. 在main方法中测试, 通过对象混入技术让User类的对象具有Logger特质的log()方法.

参考代码

```
//案例: 演示动态混入
object ClassDemo05 {
  //1. 创建Logger特质,  添加log(msg:String)方法
  trait Logger {
    def log(msg:String) = println(msg)
  }

  //2. 创建一个User类, 该类和Logger特质之间无任务关系.
  class User

  //main方法, 作为程序的入口
  def main(args: Array[String]): Unit = {
    //3. 在main方法中测试, 通过对象混入技术让User类的对象具有Logger特质的log()方法.
    val c1 = new User with Logger     //对象混入
    c1.log("我是User类的对象, 我可以调用Logger特质中的log方法了")
  }
}
```

3. 使用trait实现适配器模式

3.1 设计模式简介

概述

设计模式（Design Pattern）是前辈们对代码开发经验的总结，是解决特定问题的一系列套路。它并不是语法规定，而是一套用来提高代码可复用性、可维护性、可读性、稳健性以及安全性的解决方案。

分类

设计模式一共有23种, 分为如下的3类:

1. 创建型

   `指的是: 需要创建对象的. 常用的模式有: 单例模式, 工厂方法模式`

2. 结构型

   `指的是: 类,特质之间的关系架构. 常用的模式有: 适配器模式, 装饰模式`

3. 行为型

   `指的是: 类(或者特质)能够做什么. 常用的模式有:模板方法模式, 职责链模式`

3.2 适配器模式

当特质中有多个抽象方法, 而我们只需要用到其中的某一个或者某几个方法时, 不得不将该特质中的所有抽象方法给重写了, 这样做很麻烦. 针对这种情况, 我们可以定义一个抽象类去继承该特质, 重写特质中所有的抽象方法, 方法体为空. 这时候, 我们需要使用哪个方法, 只需要定义类继承抽象类, 重写指定方法即可. 这个抽象类就叫: 适配器类. 这种设计模式(设计思想)就叫: 适配器设计模式.

结构

```
trait 特质A{
    //抽象方法1
    //抽象方法2
    //抽象方法3
    //...
}
abstract class 类B extends A{    //适配器类
    //重写抽象方法1, 方法体为空
    //重写抽象方法2, 方法体为空
    //重写抽象方法3, 方法体为空
    //...
}
class 自定义类C extends 类B {
    //需要使用哪个方法, 重写哪个方法即可.
}
```

需求

1. 定义特质PlayLOL, 添加6个抽象方法, 分别为: top(), mid(), adc(), support(), jungle(), schoolchild()

   > 解释: top: 上单, mid: 中单, adc: 下路, support: 辅助, jungle: 打野, schoolchild: 小学生

2. 定义抽象类Player, 继承PlayLOL特质, 重写特质中所有的抽象方法, 方法体都为空.
3. 定义普通类GreenHand, 继承Player, 重写support()和schoolchild()方法.
4. 定义main方法, 在其中创建GreenHand类的对象, 并调用其方法进行测试.

参考代码

```
//案例: 演示适配器设计模式.
object ClassDemo06 {
  //1. 定义特质PlayLOL, 添加6个抽象方法, 分别为: top(), mid(), adc(), support(),
jungle(), schoolchild()
  trait PlayLOL {
    def top()           //上单
    def mid()           //中单
    def adc()           //下路
    def support()       //辅助
    def jungle()        //打野
    def schoolchild()   //小学生
  }

  //2. 定义抽象类Player, 继承PlayLOL特质, 重写特质中所有的抽象方法, 方法体都为空.
  //Player类充当的角色就是: 适配器类.
  class Player extends PlayLOL {
    override def top(): Unit = {}
    override def mid(): Unit = {}
    override def adc(): Unit = {}
    override def support(): Unit = {}
    override def jungle(): Unit = {}
    override def schoolchild(): Unit = {}
```

```
  }

  //3. 定义普通类GreenHand, 继承Player, 重写support()和schoolchild()方法.
  class GreenHand extends Player{
    override def support(): Unit = println("我是辅助, B键一扣, 不死不回城!")
    override def schoolchild(): Unit = println("我是小学生, 你骂我, 我就挂机!")
  }

  //4. 定义main方法, 在其中创建GreenHand类的对象, 并调用其方法进行测试.
  def main(args: Array[String]): Unit = {
    //创建GreenHand类的对象
    val gh = new GreenHand
    //调用GreenHand类中的方法
    gh.support()
    gh.schoolchild()
  }
}
```

4. 使用trait实现模板方法模式

在现实生活中, 我们会遇到论文模板, 简历模板, 包括PPT中的一些模板等, 而在面向对象程序设计过程中，程序员常常会遇到这种情况：设计一个系统时知道了算法所需的关键步骤，而且确定了这些步骤的执行顺序，但某些步骤的具体实现还未知，或者说某些步骤的实现与具体的环境相关。

例如，去银行办理业务一般要经过以下4个流程：取号、排队、办理具体业务、对银行工作人员进行评分等，其中取号、排队和对银行工作人员进行评分的业务对每个客户是一样的，可以在父类中实现，但是办理具体业务却因人而异，它可能是存款、取款或者转账等，可以延迟到子类中实现。这就要用到模板方法设计模式了.

4.1 概述

在Scala中, 我们可以先定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤, 这就是: 模板方法设计模式.

优点

1. 扩展性更强.

   `父类中封装了公共的部分, 而可变的部分交给子类来实现.`

2. 符合开闭原则。

   `部分方法是由子类实现的, 因此子类可以通过扩展方式增加相应的功能.`

缺点

1. 类的个数增加, 导致系统更加庞大, 设计也更加抽象。

   `因为要对每个不同的实现都需要定义一个子类`

2. 提高了代码阅读的难度。

   `父类中的抽象方法由子类实现, 子类执行的结果会影响父类的结果, 这导致一种反向的控制结构.`

4.2 格式

```
class A {        //父类, 封装的是公共部分
    def 方法名(参数列表) = {    //具体方法, 在这里也叫: 模板方法
        //步骤1, 已知.
```

```
        //步骤2, 未知, 调用抽象方法
        //步骤3, 已知.
        //步骤n...
    }

    //抽象方法
}

class B extends A {
    //重写抽象方法
}
```

> 注意: 抽象方法的个数要根据具体的需求来定, 并不一定只有一个, 也可以是多个.

4.3 示例

需求

1. 定义一个模板类Template, 添加code()和getRuntime()方法, 用来获取某些代码的执行时间.
2. 定义类ForDemo继承Template, 然后重写code()方法, 用来计算打印10000次"Hello,Scala!"的执行时间.
3. 定义main方法, 用来测试代码的具体执行时间.

参考代码

```
//案例: 演示模板方法设计模式
object ClassDemo07 {

  //1. 定义一个模板类Template, 添加code()和getRuntime()方法, 用来获取某些代码的执行时间.
  abstract class Template {
    //定义code()方法, 用来记录所有要执行的代码
    def code()

    //定义模板方法, 用来获取某些代码的执行时间.
    def getRuntime() = {
      //获取当前时间毫秒值
      val start = System.currentTimeMillis()
      //具体要执行的代码
      code()
      //获取当前时间毫秒值
      val end = System.currentTimeMillis()
      //返回指定代码的执行时间.
      end - start
    }
  }

  //2. 定义类ForDemo继承Template, 然后重写getRuntime()方法, 用来计算打印10000
次"Hello,Scala!"的执行时间.
  class ForDemo extends Template {
    override def code(): Unit = for(i <- 1 to 10000) println("Hello, Scala!")
  }

  def main(args: Array[String]): Unit = {
    //3. 测试打印10000次"Hello, Scala!"的执行时间
    println(new ForDemo().getRuntime())
  }
}
```

5 使用trait实现职责链模式

5.1 概述

多个trait中出现了同一个方法, 且该方法最后都调用了super.该方法名(), 当类继承了这多个trait后, 就可以依次调用多个trait中的此同一个方法了, 这就形成了一个调用链。

执行顺序为:

1. 按照`从右往左`的顺序依次执行.

   `即首先会先从最右边的trait方法开始执行，然后依次往左执行对应trait中的方法`

2. 当所有子特质的该方法执行完毕后, 最后会执行父特质中的此方法.

这种设计思想就叫: 职责链设计模式.

> 注意: 在Scala中, 一个类继承多个特质的情况叫`叠加特质`.

5.2 格式

```
trait A {            //父特质
    def show()       //假设方法名叫: show
}

trait B extends A { //子特质, 根据需求可以定义多个.
    override def show() = {
        //具体的代码逻辑.
        super.show()
    }
}

trait C extends A {
    override def show() = {
        //具体的代码逻辑.
        super.show()
    }
}

class D extends B with C {  //具体的类, 用来演示: 叠加特质.
     def 方法名() = {         //这里可以是该类自己的方法, 不一定非的是show()方法.
        //具体的代码逻辑.
        super.show()         //这里就构成了: 调用链.
    }
}
/*
    执行顺序为:
        1. 先执行类D中的自己的方法.
        2. 再执行特质C中的show()方法.
        3. 再执行特质B中的show()方法.
        4. 最后执行特质A中的show()方法.
*/
```

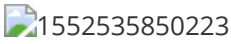
5.3 示例

需求

通过Scala代码, 实现一个模拟支付过程的调用链.

> 解释:
>
> 我们如果要开发一个支付功能，往往需要执行一系列的验证才能完成支付。例如:
>
> 1. 进行支付签名校验
> 2. 数据合法性校验
> 3. ...
>
> 如果将来因为第三方接口支付的调整，需要增加更多的校验规则，此时如何不修改之前的校验代码，来实现扩展呢?
>
> 这就需要用到: 职责链设计模式了.

图解

 1552535850223

步骤

1. 定义一个Handler特质, 添加具体的handle(data:String)方法，表示处理数据(具体的支付逻辑)
2. 定义一个DataValidHandler特质，继承Handler特质.
    - 重写handle()方法，打印"验证数据", 然后调用父特质的handle()方法
3. 定义一个SignatureValidHandler特质，继承Handler特质.
    - 重写handle()方法, 打印"检查签名", 然后调用父特质的handle()方法
4. 创建一个Payment类, 继承DataValidHandler特质和SignatureValidHandler特质
    - 定义pay(data:String)方法, 打印"用户发起支付请求", 然后调用父特质的handle()方法
5. 添加main方法, 创建Payment对象实例, 然后调用pay()方法.

参考代码

```
//案例: 演示职责链模式(也叫: 调用链模式)
object ClassDemo08 {
  //1. 定义一个父特质 Handler, 表示处理数据(具体的支付逻辑)
  trait Handler {
    def handle(data:String) = {
      println("具体处理数据的代码(例如: 转账逻辑)")
      println(data)
    }
  }
  //2. 定义一个子特质 DataValidHandler, 表示 校验数据.
  trait DataValidHandler extends Handler {
    override def handle(data:String) = {
      println("校验数据...")
      super.handle(data)
    }
  }
  //3. 定义一个子特质 SignatureValidHandler, 表示 校验签名.
  trait SignatureValidHandler extends Handler {
    override def handle(data:String) = {
      println("校验签名...")
      super.handle(data)
    }
  }
  //4. 定义一个类Payment, 表示: 用户发起的支付请求.
  class Payment extends DataValidHandler with SignatureValidHandler {
    def pay(data:String) = {
```

```
      println("用户发起支付请求...")
      super.handle(data)
    }
  }
  def main(args: Array[String]): Unit = {
    //5. 创建Payment类的对象, 模拟: 调用链.
    val pm = new Payment
    pm.pay("苏明玉给苏大强转账10000元")
  }
}

// 程序运行输出如下:
// 用户发起支付请求...
// 校验签名...
// 校验数据...
// 具体处理数据的代码(例如: 转账逻辑)
// 苏明玉给苏大强转账10000元
```

6. trait的构造机制

6.1 概述

如果遇到一个类继承了某个父类且继承了多个父特质的情况，那该类(子类), 该类的父类, 以及该类的父特质之间是如何构造的呢?

要想解决这个问题, 就要用到接下来要学习的`trait的构造机制`了.

6.2 构造机制规则

- 每个特质只有`一个无参数`的构造器。

  `也就是说: trait也有构造代码，但和类不一样，特质不能有构造器参数.`

- 遇到一个类继承另一个类、以及多个trait的情况，当创建该类的实例时，它的构造器执行顺序如下:

  1. 执行父类的构造器
  2. 按照`从左到右`的顺序, 依次执行trait的构造器
  3. 如果trait有父trait，则先执行父trait的构造器.
  4. 如果多个trait有同样的父trait，则父trait的构造器只初始化一次.
  5. 执行子类构造器

6.3 示例

需求

- 定义一个父类及多个特质，然后用一个类去继承它们.
- 创建子类对象, 并测试trait的构造顺序

步骤

1. 创建Logger特质，在构造器中打印"执行Logger构造器!"
2. 创建MyLogger特质，继承自Logger特质，，在构造器中打印"执行MyLogger构造器!"
3. 创建TimeLogger特质，继承自Logger特质，在构造器中打印"执行TimeLogger构造器!"
4. 创建Person类，在构造器中打印"执行Person构造器!"
5. 创建Student类，继承Person类及MyLogger, TimeLogge特质，在构造器中打印"执行Student构造器!"

6. 添加main方法，创建Student类的对象，观察输出。

参考代码

```
//案例: 演示trait的构造机制.
object ClassDemo09 {
  //1. 创建Logger父特质
  trait Logger {
    println("执行Logger构造器")
  }
  //2. 创建MyLogger子特质, 继承Logger特质
  trait MyLogger extends Logger {
    println("执行MyLogger构造器")
  }
  //3. 创建TimeLogger子特质, 继承Logger特质.
  trait TimeLogger extends Logger {
    println("执行TimeLogger构造器")
  }
  //4. 创建父类Person
  class Person{
    println("执行Person构造器")
  }
  //5. 创建子类Student, 继承Person类及TimeLogger和MyLogger特质.
  class Student extends Person with TimeLogger with MyLogger {
    println("执行Student构造器")
  }
  //main方法, 程序的入口.
  def main(args: Array[String]): Unit = {
    //6. 创建Student类的对象，观察输出。
    new Student
  }
}

// 程序运行输出如下:
// 执行Person构造器
// 执行Logger构造器
// 执行TimeLogger构造器
// 执行MyLogger构造器
// 执行Student构造器
```

7. trait继承class

7.1 概述

在Scala中, trait(特质)也可以继承class(类)。特质会将class中的成员都继承下来。

7.2 格式

```
class 类A {          //类A
    //成员变量
    //成员方法
}

trait B extends A { //特质B
}
```

7.3 示例

需求

1. 定义Message类. 添加printMsg()方法, 打印"学好Scala, 走到哪里都不怕!"
2. 创建Logger特质，继承Message类.
3. 定义ConsoleLogger类, 继承Logger特质.
4. 在main方法中, 创建ConsoleLogger类的对象, 并调用printMsg()方法.

参考代码

```
//案例: 演示特质继承类
object ClassDemo10 {
  //1. 定义Message类. 添加printMsg()方法, 打印"测试数据..."
  class Message {
    def printMsg() = println("学好Scala, 走到哪里都不怕!")
  }
  //2. 创建Logger特质，继承Message类.
  trait Logger extends Message
  //3. 定义ConsoleLogger类, 继承Logger特质.
  class ConsoleLogger extends Logger

  def main(args: Array[String]): Unit = {
    //4. 创建ConsoleLogger类的对象, 并调用printMsg()方法.
    val cl = new ConsoleLogger
    cl.printMsg()
  }
}
```

8. 案例: 程序员

8.1 需求

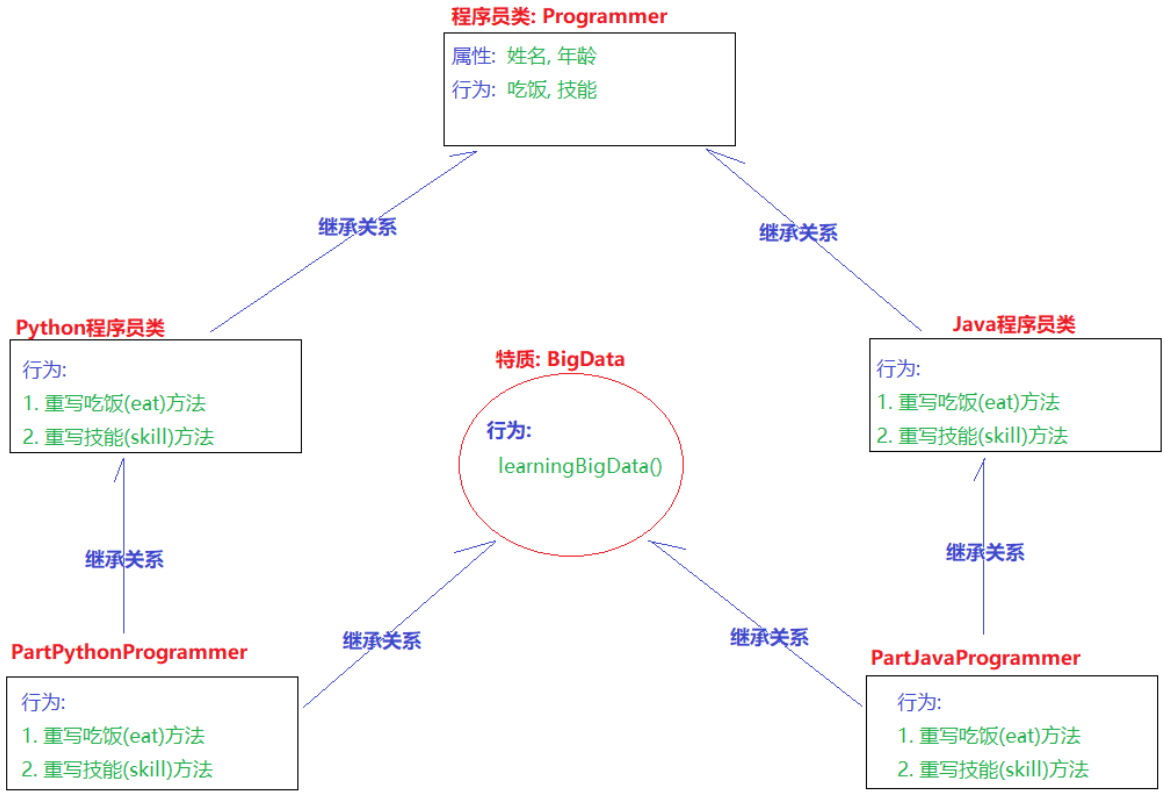
现实生活中有很多程序员, 例如: Python程序员, Java程序员, 他们都有姓名和年龄, 都要吃饭, 都有自己所掌握的技能(skill). 不同的是, 部分的Java程序员和Python程序员来`黑马程序员`培训学习后, 掌握了大数据技术, 实现更好的就业. 请用所学, 模拟该场景.

8.2 目的

- 考察`特质`, `抽象类`的相关内容

> 简单记忆: 整个继承体系的共性内容定义到父类中, 整个继承体系的扩展内容定义到父特质中.

8.3 分析



8.4 参考代码

```
//案例: 程序员案例
object ClassDemo11 {
  //1. 定义程序员类Programmer.
  abstract class Programmer {
    //属性: 姓名和年龄
    var name = ""
    var age = 0

    //行为: 吃饭和技能
    def eat()
    def skill()
  }
  //2. 定义特质BigData.
  trait BigData {
    def learningBigData() = {
      println("来到黑马程序员之后: ")
      println("我学会了: Hadoop, Zookeeper, HBase, Hive, Sqoop, Scala, Spark,
Flink等技术")
      println("我学会了: 企业级360°全方位用户画像, 千亿级数据仓, 黑马推荐系统, 电信信号强度
诊断等项目")
    }
  }
  //3. 定义Java程序员类JavaProgrammer.
  class JavaProgrammer extends Programmer {
    override def eat(): Unit = println("Java程序员吃大白菜, 喝大米粥.")

    override def skill(): Unit = println("我精通Java技术.")
  }

  //4. 定义Python程序员类PythonProgrammer.
  class PythonProgrammer extends Programmer {
    override def eat(): Unit = println("Python程序员吃小白菜, 喝小米粥.")
```

```scala
    override def skill(): Unit = println("我精通Python技术.")
  }

  //5. 定义PartJavaProgrammer类, 表示精通: Java和大数据的程序员.
  class PartJavaProgrammer extends JavaProgrammer with BigData {
    override def eat(): Unit = println("精通Java和大数据的程序员, 吃牛肉, 喝牛奶")

    override def skill(): Unit = {
      super.skill()     //本身就掌握的内容.
      learningBigData() //来到黑马之后学习的技能.
    }
  }

  //6. 定义PartPythonProgrammer类, 表示精通: Python和大数据的程序员.
  class PartPythonProgrammer extends PythonProgrammer with BigData {
    override def eat(): Unit = println("精通Python和大数据的程序员, 吃羊肉, 喝羊奶")

    override def skill(): Unit = {
      super.skill()     //本身就掌握的内容.
      learningBigData() //来到黑马之后学习的技能.
    }
  }

  def main(args: Array[String]): Unit = {
    //7. 测试.
    //7.1 测试普通Java程序员.
    val jp = new JavaProgrammer
    jp.name = "张三"
    jp.age = 23
    println(s"我叫${jp.name}, 年龄为${jp.age}, 我是普通的Java程序员")
    jp.eat()
    jp.skill()
    println("-" * 15)   //分割线

    //7.2 测试精通Java + 大数据的程序员
    val pjp = new PartJavaProgrammer
    pjp.name = "李四"
    pjp.age = 24
    println(s"我叫${pjp.name}, 年龄为${pjp.age}, 我精通Java技术和大数据技术")
    pjp.eat()
    pjp.skill()

    //7.3 测试普通的Python程序员, 自行测试.
    //7.4 测试精通Python + 大数据的程序员, 自行测试.
  }
}
```