类

一、类的定义

1.1 字段和方法

1. 类的定义可以包含字段`field`和方法`method`，它们统称为类的成员`member`。
   - 通过`val`或者`var`定义字段，通过`def`定义方法。
   - 字段保留了对象的状态（或者说数据），方法用这些数据来执行计算。
   - 当实例化一个类时，`runtime`会分配一些内存来保存对象的数据。
   - 字段又被称作实例变量，因为每个实例都有自己的变量。

```
class C1{
    val field1 = 0;          // 字段
    var field2 = "hello";    // 字段
    def method1(){};         // 方法
}
```

2. `Scala`中，禁止同一个类中使用相同名字的字段和方法，而这在`Java`中是允许的。

```
class C1{  // 无法编译
    val f = 0 ;       // 字段
    def f = 1 ;       // 方法
}
```

3. `Scala`中，字段和方法属于同一个命名空间。事实上，`Scala`只有两个命名空间用于定义，而`Java`有四个。
   - `Java`的四个命名空间：字段、方法、`Class`、`package`。
   - `Scala`的两个命名空间：值（字段、方法、`package`、单例对象）、类型（`Class`和`Traits`）。
4. 可以对字段设置缺省值`_`。

```
class C1{
    val field1: Int = _
}
```

字段的`= _`初始化代码会给这个字段赋一个零值。具体零值取决于字段的类型：数值类型的零值是 0，布尔值的零值是`false`，引用类型的零值是`null`。这跟`Java`中没有初始化代码的变量效果一样。

注意：并不能简单的去掉`= _`。如果是`val field1: Int`会定义一个抽象变量，而不是一个未被初始化的变量。

1.2 对象实例化

1. 在`Scala`中可以使用`new`来实例化对象，这和`Java`保持一致。当实例化对象时，可以使用值和类型来对其进行参数化`parameterize`。

- 参数化的意义是：在创建实例时，对其进行“配置”。
- 值的参数化示例：在构造方法的圆括号中传入对象参数。

```
val int1 = new java.math.BigInteger("1234")
```

- 类型的参数化示例：在方括号中给出一个或者多个类型。

```
val string1 = new Array[String](3)
```

- 当同时使用类型参数化和值参数化时，首先是以方括号给出的类型参数，然后是圆括号给出的值参数。

2. 类型参数实际上是实例类型的一部分（而值参数不是），你也可以显式的给出实例的类型：

```
val string1: Array[String] = new Array[String](3)
```

3. 如果希望对类型同时实例化多个对象，且参数相同，则可以简化：

```
val a = new C
val b = new C
val c = new C
// 简化为： val a, b, c = new C
```

1.3 可变性

1. 如果包含`var`字段，则对象是可变的。

```
class C1{
    val field1 = 0;          // 不可变字段
    var field2 = "hello";    // 可变字段
    // 其它定义
}
val obj1 = new C1;
var obj2 = new C1;
```

- `obj1`是`val`变量，因此不能对其重新赋值。但是可以修改`obj1`的状态：

```
// obj1 = new C1; // 错误：obj1 是 val
obj1.field2 = "world"
```

- `obj2`是`var`变量，可以重新对其赋值：

```
obj2 = new C1; // 再次创建对象
```

2. 可变对象与不可变对象的选择：

- 不可变对象的优势：不会改变状态，因此可以自由地传递不可变对象，也可以在多线程中安全使用。同时不可变对象可被安全地用作哈希表里的键。
- 不可变对象的劣势：有时需要拷贝很大的数据内容而实际上一个很小的局部更新就能满足要求。

因此类库对于不可变的类也提供了可变的版本，如`StringBuilder`就是`String`类的可变版本。

1.4 构造方法

1. 一个类如果没有定义体，则并不需要给出空的花括号。当然你也可以给出花括号，这不是必须的。
2. 类名后的圆括号中的参数称作类参数`class parameter`，`Scala`编译器会收集类参数，并且创建一个主构造方法`primary constructor`，由主构造方法接收类参数。

```
class C(n:Int,s:String) // 省略花括号
val c = new C(1,"hello")
```

   - 这与`Java`不同。
     - `Java`中，类有构造方法，构造方法可以接收参数。而`Scala`中，类可以直接接收参数，更为精简。
     - 在`Scala`中，类定义体内可以直接使用类参数，不需要定义字段并编写构造方法将参数赋值给该字段。
   - `Scala`编译器会将你在类定义体中给出的非字段定义、非方法定义的代码编译进类的主构造方法中。

```
class C(n:Int,s:String)
{
    println("n:"+n+";s:"+s)  // 将被添加到类的主构造方法中
}
```

3. 虽然类定义体内可以直接使用类参数，但是在类外无法直接访问类参数。如果希望在类外访问，则需要将它做成字段。

```
class C(n:Int,s:String)
{
    require(n>=0)
    val num:Int = n
    val name:String = s
    override def toString = num + ";" + name // 也可以使用 n 和 s
}
val c = new C(1,"zhang san")
println(c.num+";"+c.name) //输出：1;zhang san
// println(c.n+";"+c.s) 编译错误
```

   尽管`n`和`s`是类参数，它们只会出现在类定义体中，`Scala`编译器并不会为它们生成字段。

4. `Scala`编译器会根据字段出现的先后顺序来依次初始化，因此如果字段之间有依赖关系，则需要仔细安排其位置。

```
class C(m:Int,n:Int)
{
    private val g = min(m,n)  // 首先被初始化
    val num1 = m/g            // 必须在 g 初始化之后才能初始化
    val num2 = n/g            // 必须在 g 初始化之后才能初始化
}
```

5. 有时需要给某个类定义多个构造方法。在`Scala`中，主构造方法之外的构造方法称作辅助构造方法`auxiliary constructor`。

   - `Scala`的辅助构造方法以`def this(...)`开始。
   - `Scala`中，每个辅助构造方法都必须首先调用同一个类的另一个构造方法。即：每个辅助构造方法的第一条语句都必须是这样的形式：`this(...)`。

     被调用的构造方法要么是主构造方法，要么是另一个定义在调用方之前的另一个辅助构造方法。这使得`Scala`的每个构造方法最终都会调用到类的主构造方法，从而主构造方法成为类的单一入口。
   - `Java`中，构造方法要么调用同一个类的另一个构造方法，要么调用超类的构造方法。而`Scala`中，只有主构造方法才能调用超类的构造方法。这是一种更严格的限制。

```
class C(n:Int,s:String)
{
    val num:Int = n
    val name:String = s
    def this(n:Int) = this(n,"zhang san") // 辅助构造方法
}
```

6. 可以对主构造方法定义一个前置条件`precondition`，前置条件是对传入方法或者构造方法的参数值的约束，是方法调用者必须要满足的。

   - 实现前置条件的方式是采用`require`。它是定义在`Predef`这个单例对象中，而`Scala`源文件中默认自动引入`Predef`的成员。
   - `require`方法接收一个`boolean`的参数。如果传入的参数为`true`，则它正常返回。如果传入的参数为`false`，则它抛出`IllegalArgumentException`，这会阻止对象的构建。

```
class C(n:Int,s:String)
{
    require(n>=0)  // 主构造方法必须满足的前置条件
    override def toString =  n + ";" + s
}
```

1.5 this 关键字

1. 在类定义体中，关键字`this`指向当前对象。如果方法需要返回当前对象，则必须显式给出`this`，否则`this`大多数情况下可以省略。

```
class C(n:Int,s:String)
{
    require(n>=0)
    val num:Int = n
    val name:String = s
    def lessThan(that:C) = this.num < that.num
    // 也可以忽略 this，写作： num < that.num

    def max(that:C) = if(lessThan(that)) that else this // 必须显式指定 this
}
```

1.6 无参方法

1. `Scala` 中，如果类的方法没有参数列表，则可以采用无参方法的模式（省略了空的参数列表）：

```
class C(n:Int, s:String)
{
    def width: Int = s.length // 无参方法
}
```

带空的圆括号定义的方法，如`def width():Int = s.length`被称作空圆括号方法。

2. `Scala` 推荐对没有参数的、且不改变对象的状态的方法尽可能使用无参方法。这种做法支持“统一访问原则”：使用方代码不应该受到某个属性是字段还是方法实现的影响。

   如：要想把`width`实现为字段而不是方法，则只需要简单将`def`修改为`val`：

```
class C(n:Int, s:String)
{
    val width: Int = s.length // 修改为字段
}
```

从使用方来看，这个定义完全等价。区别在于：

- 字段访问的速度比方法略快，因为字段值在对象初始化时就被预先计算好。而方法需要每次调用时都计算一次。
- 字段需要为对象分配额外的内存空间，而方法不需要。

因此属性实现为字段好还是方法好，这取决于类的用法。而用法可以随时间变化，核心点在于：类的使用方不应该被内部实现的变化所影响，使用方并不需要关心究竟是哪一种实现。

3. 不仅在方法定义时可以通过无参方法省略空的圆括号，还可以在方法调用时省略掉空的圆括号。

```
val c = new C(1,"hello")
val c_width = c.width // 省略掉空的圆括号
```

- 从原理上讲，可以对`Scala`所有无参函数调用都去掉空括号，但是建议对于有副作用的调用时给出空括号。这些场景包括：方法执行`I/O`、修改了某个`var`变量（可能是类内，可能是类外）。

  这时候空的参数列表给出一个视觉上的线索：这里发生了副作用。

```
"hello".length    // 可以去掉 ()，因为没有副作用
println()         // 建议不要去掉 ()，因为有副作用
```

- 这种省略空括号的方法调用，使得使用方调用方法和访问字段的行为完全保持一致。

```
class C(n:Int, s:String)
{
    def width: Int = s.length // 无参方法
    val height: Int = n       // 字段
}
val c = new C(1,"hello")
val c_width = c.width  // 调用方法
val c_width = c.height // 访问字段
```

由于这种访问一致性，使用方因此可以不关心属性背后的实现是通过方法、还是通过字段。

1.7 参数化字段

1. 有时候类的参数的唯一作用是为类的字段赋值，此时可以通过参数化字段来简写。

```
class A(s:String){
    val name = s        // s 的作用是为 name 字段赋值
}
```

参数化字段 `parametric field` 是在类的参数之前添加一个 `val` 或者 `var`，这可以同时定义参数和一个同名的字段。

```
class A(val name:String){
  // 现在 A 拥有一个名为 name 的字段，以及一个名为 name 的参数
}
```

这里 `var` 和 `val` 的区别在于：`var` 的字段是可重新赋值的，`val` 的字段是不可重新赋值的。

2. 你也可以为参数化字段添加修饰符，如 `private、protected、overried`，就像你对其它类成员做的那样。

1.8 私有构造方法

1. `Java` 中，我们可以通过标记为 `private` 来隐藏构造方法。但是在 `Scala` 中，主构造方法并没有显式的定义，它是通过类参数和类定义体隐式地定义的。

   可以在参数列表前加上 `private` 修饰符来隐藏主构造方法：

```
class C private(
    private val name: String,
    private val age: Double
)
val c = new C("hello", 20.0)  // 编译失败
```

一旦隐藏了主构造方法，则该构造方法只能从类本身及其伴生对象访问。类名依然是公有的，因此可以将其作为类型来用。但是不能调用其构造方法。

2. 如果隐藏了主构造方法，则有两种方式来创建类的实例：
   - 通过辅助构造方法来创建：

```
class C private(
                private val name: String,
                private val age: Double)
{
  def this(n: String) = this(n, 20.0)  // 辅助构造方法
}
val c = new C("hello")
```

   - 通过伴生对象的工厂方法来创建：

```
class C private(
                private val name: String,
                private val age: Double)
object C {
  def apply(name: String, age: Double) = new C(name,age)
}
val c = C("hello", 20.0)
```

通过伴生对象的 `apply` 方法来创建类的一个实例。

3. 一种极端情况是：利用伴生对象来隐藏类本身。

```
trait A {
    /*
    定义了一些接口
    */
}
object A {
  def apply(name: String, age: Double):A = new AImpl(name,age)  // 创建对象
  private class AImpl(private val name: String,private val age: Double)
 extends A{
       /*
    定义了一些接口
    */
  }
}
```

现在类 `CImpl` 被完整的隐藏了，因为它是一个私有类。

二、单例对象

1. `Scala` 的类不允许有 `static` 静态成员，为了提供类似 `Java` 静态成员的功能，`Scala` 提供了单例对象 `singleton object` 。

   单例对象是 `Scala` 中的一等公民，其地位与 `class` 相等，并不是其附属。定义单例对象并不会定义类型。

2. 单例对象的定义看起来和类定义很像，只是 `class` 关键字被 替换成了 `object` 关键字。
   - 可以将单例对象视作存放 `Java` 中的静态字段和静态方法的地方。
   - 可以用 `单例对象名.成员名` 的方式来访问单例对象的成员。
3. 类和单例对象的区别：
   - 单例对象不接收参数，而类可以。因为无法采用 `new` 实例化单例对象，因此没有任何手段来向它传参。
   - 每个单例对象都是通过一个静态变量引用合成类 `synthetic class` 的实例来实现的（这个合成类的名字是 `对象名+美元符$` ），因此单例对象从初始化的语义上跟 `Java` 静态成员是一致的：单例对象在有代码首次访问时才被初始化。
4. `Scala` 在每个源码文件都隐式的引入了 `java.lang` 和 `scala` 包的成员，以及名为 `Predef` 的单例对象的所有成员。

   `Predef` 单例对象位于 `scala` 包中，它包含许多有用的方法。
   - 如：在 `Scala` 源码中使用 `println` 时，实际调用了 `Predef.println` 方法，而该方法实际调用的是 `Console.println` 来执行具体操作。

- 如：在`Scala`源码中使用`assert`时，实际调用了`Predef.assert`方法。

2.1 伴生对象

1. 当单例对象跟某个类共用同一个名字时，该单例对象被称作这个类的伴生对象`companion object`，这个类被称作该单例对象的伴生类`companion class`。
   - 必须在同一个源码文件中定义类和类的伴生对象。
   - 类和它的伴生对象可以相互访问对方的私有成员。

```
class C{              // 伴生类
    // 这里是类定义
}
object C{             // 伴生对象
    // 这里是单例对象的定义
}
```

2.2 孤立对象

1. 没有伴生类的单例对象称作孤立对象`standalone object`。孤立对象有很多用途：作为工具方法的收集者、定义`Scala`应用程序的入口等。
2. 要运行一个`Scala`程序，必须提供一个单例对象的名称，该单例对象必须包含一个`main`方法，该方法必须接收一个`Array[String]`作为参数，该方法的类型必须为`Unit`。

   任何带有满足正确签名的`main`方法的单例对象都能被用作应用程序的入口：

```
object A{
    def main(args:Array[String]) = {
        for（arg <- args)
          println(arg)
    }
}
```

3. `scala`提供了一个特质`scala.App`来便捷地定义`main`方法：

```
object A extends App{
    for（arg <- args)
       println(arg)
}
```

   首先在单例对象的名字后面添加`extends App`，然后将`main`方法里的代码直接写在单例对象的花括号里。可以通过`args`的`Array[String]`来访问命令行参数。

2.3 工厂对象

1. 工厂对象包含了创建其它对象的方法。通常建议采用工厂对象的方法来构建对象，而不是直接采用`new`来构建对象，这些方法称作工厂方法。

   这种做法的优点是：对象创建的逻辑可以被集中管理，而对象是如何用具体的类来表示的细节可以隐藏起来。
   - 一方面，你的类库更容易被使用方所理解。
   - 另一方面，因为暴露的细节少，这样就提供了更多机会让你在未来不破坏使用方代码的前提下改变类库的实现。

2. 工厂方法最直接的方案是创建类的伴生对象。

```
abstract class C(name:String) { // 抽象父类
    // 类的定义体
}
class Child1(name:String,age:Int) extends C{ // 第一级子类
    // 类的定义体
}
class Child2(name:String,age:Int,job:String) extends Child1{ // 第二级子类
    // 类的定义体
}

object C {
    def make_C(n:String,m:Int) : C = new Child1(n,m)
    def make_C(n:String,m:Int,j:String) : C = new Child2(n,m,j)
}
```

`C` 对象包含三个重载的`make_C` 工厂方法，每个工厂方法创建不同的类的对象（它们要么是`C` 的对象，要么是`C` 子类的对象）。

3. 一旦有了工厂方法，就可以把目标类及其子类变成私有的，使得无法通过 `new` 来创建其对象。

   将类变成私有的方式之一是：将它们放在工厂对象中，并声明为私有的。

```
abstract class C(name:String) { // 抽象父类
    // 类的定义体
}

object C {
  private class Child1(name:String,age:Int) extends C{ // 第一级子类
      // 类的定义体
  }
  private class Child2(name:String,age:Int,job:String) extends Child1{ //
第二级子类
      // 类的定义体
  }

    def make_C(n:String,m:Int) : C = new Child1(n,m)
    def make_C(n:String,m:Int,j:String) : C = new Child2(n,m,j)
}
```

现在子类 `Child1` 和 `Child2` 都无法在外部访问，但是可以通过单例对象`C` 的工厂方法访问。

三、访问级别

1. 在 `Scala` 中，默认的访问级别是 `public` 。通过`private` 修饰符可以将字段或者方法变成私有。

```
class C(n:Int,s:String)
{
    private val num:Int = n      //私有字段
    private val name:String = s
}
```

2. 包、类和对象的成员可以标上`private`和`protected`这样的访问修饰符。这些修饰符对成员的访问限定在特定的代码区域。
3. `Scala`对访问修饰符的处理大体上与`Java`保持一致，但是也有一些重要区别。

3.1 private

1. `Scala`对私有成员的处理跟`Java`类似：标记为`private`的成员只有在包含该定义的类或者对象的内部可见。

```
class Outer{
    class Inner {
        private def f() = println("Inner f")
        class InnerMost {
            f()       // 内部访问：可以调用
        }
    }
    (new Inner).f() // 外部访问：编译错误，无法访问  f
}
```

   - 在`Scala`中，这个规则同样适用于内部类。`Scala`在一致性上做的比`Java`更好。

     如上例所示：

     - 第一次调用`f`是包含在`Inner`类的内部，因此可以访问。
     - 第二次调用`f`时发生在`Inner`类的外部，因此无法访问。
   - 在`Java`中，两种访问都可以进行。因为在`Java`中，可以从外部类访问其内部类的私有成员。

3.2 protected

1. 和`Java`相比，`Scala`对`protected`成员的访问也更加严格。在`Scala`中，`protected`成员只能从定义该成员的子类、或者该成员本身访问。而在`Java`中，允许同一个包内的其它类访问这个类的受保护成员。

```
package p {
    class Parent {
        protected def f() = println("Parent f")
    }
    class Child extends Parent {
        f()       // 子类可以访问父类的 protected 成员
    }
    class Other {
        (new Parent).f() // 同一个包内，在 Java 中允许访问；  在 Scala 中无法访问
    }
}
```

3.3 public

1. `Scala`并没有专门的修饰符来标记公有成员，任何未被标记为`private`或`protected`的成员都是公有的。

   公有成员可以从任何位置访问到。

3.4 保护的范围

1. 我们可以通过限定词对 `Scala` 中的访问修饰符机制进行增强。形如 `private[X]` 或者 `protected[X]` 的含义是：对此成员的访问限制"上至" `X` 都是私有的或者受保护的。其中 `X` 表示某个包含该定义的包、类或者单例对象。
2. 带限定词的访问修饰符允许我们对成员的可见性做非常细粒度的控制，尤其是允许我们表达 `Java` 中访问限制的语义，如：包内私有、包内受保护、到最外层嵌套类范围内私有等。

   通过这种方法，我们还可以表达 `Java` 中无法表示的访问规则。

   这种机制对于横跨多个包的大工程非常有用。可以定义对工程中某些子包可见、但是对外部不可见的实体。这在 `Java` 中是无法做到的，因为 `Java` 中一旦定义越过了包的边界，它就对整个世界可见了。
3. 示例：

```
package root_p

package first_p {
    private[root_p] class Outer {        // Outer 对包 root_p 内所有类和对象可见
        protected[first_p] def f1()={}   // f1 对包 first_p 内所有类和对象可见
        class Inner{
            private[Outer] val num1 = 0  // num1 对类 Outer 内所有类和对象可见
            private[Inner] val num2 = 0  // num2 对类 Inner 内所有类和对象可见
        }
        private[this] val num3 = 0       // num3 仅在当前对象内访问
    }
}

package second_p {
    import first_p._  // 导入 first_p 所有成员
    object Obj {
        private[second_p] val value = new Outer
    }
}
```

   - `private` 限定词的用法：
     - `private[root_p] Outer`： 类 `Outer` 对 `root_p` 包内的所有类和对象都可见，但是对于 `root_p` 包之外的代码都无法访问 `Outer` 。

       其公共访问范围为： `root_p` 包内访问。
     - `private[first_p] def f1()={}` ： 方法 `f1` 对 `first_p` 包内的所有类和对象都可见，但是对于 `first_p` 包之外的代码都无法访问。

       其公共访问范围为： `first_p` 包内访问。它等价于 `Java` 的 `package` 可见性。
     - `private[Outer] val num1`： 变量 `num1` 对 `Outer` 类内的所有类和对象都可见，但是对于 `Outer` 类外的代码都无法访问。

       其公共访问范围为： `Outer` 类内访问。它等价于 `Java` 的 `private` 。
     - `private[Inner] val num2`： 变量 `num2` 对 `Inner` 类内的所有类和对象都可见，但是对于 `Inner` 类外的代码都无法访问。

       其公共访问范围为： `Inner` 类内访问。它等价于 `Scala` 的 `private` 。

- `private[this] val num3`：变量 `num3` 仅在包含该定义的同一个对象内访问，这样的定义被称作是对象私有的`object-private`。

  这意味着所有对它的访问不仅必须来自于`Outer`类的内部，还必须是来自于`Outer`的同一个实例。因此将一个成员标记为`private[this]`保证了它不会被同一个类的其它对象看到。

```
class C(n:Int)
{
  private val age:Int = n
  def is_less(other:C) = this.age <= other.age  // other.age 可以访问
}

println(new C(10).is_less(new C(20))) // 输出： true

class C2(n:Int)
{
  private[this] val age:Int = n
  def is_less(other:C) = this.age <= other.age  // other.age 无法访问
}
```

  - `protected`限定词的用法：所有的限定词都可以应用在`protected`上，跟`private`上的用法一样。

    即：如果我们在类`C`中使用`protected[X]`这个修饰符，则`C`的所有子类，以及`X`表示的包、类或对象中，都能访问这个被标记的定义。

    如：`protected[first_p] def f1()`：方法`f1`对包`first_p`内的所有类和对象都可见，也对`Outer`任何子类内部的所有类和对象可见。因此其含义就和`Java`的`protected`完全一样。

4. 在`Java`中，静态成员和实例成员同属于一个类，因此访问修饰符对它们的应用方式是统一的。

   由于`Scala`没有静态成员，而是用伴生对象来承载那些只存在一次的成员。因此`Scala`的访问规则在`private`和`protected`的处理上给伴生对象和类保留了特权。

   一个类会将它的所有访问权限和它的伴生对象共享，反之亦然。即：一个对象可以访问它的伴生类的所有私有成员，一个类也可以访问它的伴生对象的所有私有成员。

5. `Scala`和`Java`在修饰符的方面的确很相似，但是有一个重要例外：`protected static`。
   - `Java`中，类`C`的`protected static`成员可以被`C`的所有子类访问。
   - 对于`Scala`的伴生对象而言，`protecte`成员没有任何意义，因为单例对象没有子类。

四、继承

1. 类的组合：一个类可以包含对另一个类的引用，利用这个被引用类来帮助它完成任务。组合代表了两个类的关系是：`has-a`。

   类的继承：超类与子类的关系。继承代表了两个类的关系是：`is-a`

2. 组合和继承是两种用已有的类来定义新类的两种方式。如果追求代码复用，通常优先选择组合而不是继承。因为继承会遇到脆弱基类问题：在修改超类时会不小心破坏子类的代码。

4.1 继承

4.1.1 抽象类

1. 通过修饰符 `abstract` 可以表明类是抽象的，这种类称作抽象类。不可以直接实例化一个抽象类。

```
abstract class Element {
    def contents: Array[String]
}
```

其中 `contents` 方法是一个没有实现的方法，这种方法称作抽象成员。一个包含抽象成员的类必须声明为抽象类。这与 `Java` 不同， `Java` 中抽象成员也需要添加修饰符 `abstract`。

4.1.2 继承

1. `Scala` 通过关键字 `extends` 来声明类的继承。

```
class ArrayElement(conts:Array[String]) extends Element{
    def contents: Array[String] = conts
}
```

`extends` 有两个作用：

- 子类从父类中继承所有的非私有成员。
- 子类成为父类的子类型 `subtype` 。子类型的意思是：任何需要超类对象的地方，都可以用子类对象来替代。

这里 `ArrayElement` 是 `Element` 的子类， `Element` 是 `ArrayElement` 的超类。

2. 如果没有 `extends` 语句，则 `Scala` 的类默认继承自 `scala.AnyRef`，这对应于 `Java` 的 `java.lang.Object` 类。
3. 类的继承：子类从父类中继承所有成员，除了以下两个例外：
   - 子类并不会继承父类的私有成员。
   - 如果子类已经实现了父类中相同名称和参数的成员，则该成员不会被继承。这种情况称作：子类的成员重写 `override` 了父类的成员。

     更进一步的，如果子类的成员是具体的，而父类的成员是抽象的，则称子类的具体的成员实现 `implement` 了那个抽象的成员。

4.1.3 父类构造函数

1. 如果需要调用超类的构造方法，则只需要将你打算传入的入参放在超类名称后的圆括号里即可。

```
class Child(name: String) extends Parent(name){
   // 类的定义
}
```

4.2 重载

1. `Scala` 支持方法重载：同一个方法名可以用于多个方法，这些方法的参数列表不同。在方法调用时， `Scala` 根据入参类型来选取合适的重载版本。与 `Java` 类似， `Scala` 根据最匹配的入参类型来选择。

   如果找不到合适的重载版本，则编译器提示 `ambiguous reference` 错误。

```
class C(n:Int,s:String)
{
    private val num:Int = n
    private val name:String = s
    def f(num:Int) = this.num <= num        // 版本一
    def f(name:String) = this.name <= name // 版本二
}
```

2. 类默认继承了 `java.lang.Object` 类的 `toString` 实现，该实现只是简单的打印出类名、一个 `@` 符、一个十六进制数字。

   通常会重写类的 `toString` 方法来获取更有价值的信息，从而方便调试。重写方法通过关键字 `override` 实现：

```
class C(n:Int,s:String)
{
    override def toString = n + ";" + s // 精简了花括号、return、空参数、返回类型
}
```

   `override` 表示父类的该方法被重写覆盖了。

3. `Scala` 并没有操作符重载，因为它并没有传统意义上的操作符。类似 `+,-,*,/` 这样的字符可以被作为方法名。因此 `1+2` 这种表达式实际上调用了 `Int` 对象 `1` 上的、一个叫做 `+` 的方法，`2` 是该方法的参数。

   你也可以显式写作：`1.+(2)` 。

4. `Scala` 中，字段和方法属于同一个命名空间，这使得字段重写无参方法成为可能。
   - 可以通过 `val` 字段重写父类同名的 `val` 字段或者无参方法。
   - 也可以通过无参方法重写父类同名的 `val` 字段或者无参方法。
5. `Scala` 要求在所有重写了父类具体成员的成员之前加上 `override` 修饰符。
   - 如果重写的是父类的抽象成员，则可以加、也可以不加上 `override` 。

- 如果没有重写父类成员，但是添加了 `override`，则编译报错。
  - 如果应该添加而未添加 `override`，则编译器也报错。

6. 如果希望某个成员（方法或者字段）不能够被子类重载，则可以在成员之前添加 `final` 修饰符。

   如果子类 `override` 了父类的一个 `final` 成员，则编译报错。

```
class C // 父类
{
    final def echo = println("This is C")  // 不需要 echo 被子类重载
}

class Child extends C // 子类
{
    override def echo = println("This is Child") // 编译报错
}
```

7. 如果希望整个类没有子类，则可以简单的将类声明为 `final` 的，做法是在类声明之前添加 `final` 修饰符。

```
final class C // 不希望 C 拥有子类
{
    def echo = println("This is C")
}
```

4.3 多态和动态绑定

1. 一个类型为`C`的变量可以指向任何一个`C`的子类型的对象，这种现象称作多态`polymorphism`。
2. 对变量和表达式的方法调用是动态绑定的：实际被调用的方法实现是在运行时基于对象的类型来决定的，而不是变量或者表达式的类型决定的。

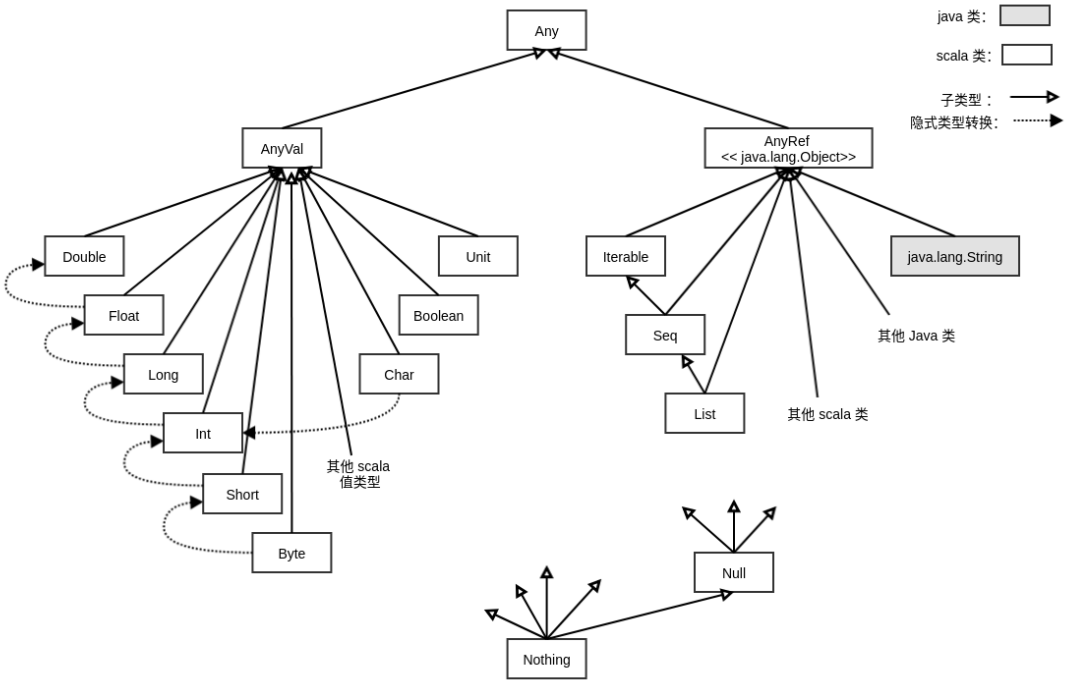
```
class C // 父类
{
    def echo = println("This is C")
}

class Child extends C // 子类
{
    override def echo = println("This is Child")
}

def f1():C = new C              // 函数 f1 返回类型为 C
def f2():C = new Child          // 函数 f2 返回类型为 C

val c1 = f1()  // c1 类型 C
val c2 = f2()  // c2 类型 C
c1.echo        // 输出：This is C
c1.echo        // 输出：This is Child
```

例子中，虽然函数`f2`返回的静态类型是`C`，但是其动态类型是`C`的子类`Child`。其`echo`方法调用的是子类中的`echo`方法。

4.4 继承体系

1. `Scala`中，所有类都继承自同一个名为`Any`的超类。由于任何类都是`Any`的子类，因此`Any`中定义的方法是"全类型"的：它们可以在任何对象上被调用。
2. `Scala`还在继承关系的底部定义了`Null`类和`Nothing`类，它们本质上是作为通用子类存在的。

   `Nothing`类是每个其它类的子类。



4.4.1 Any 类

1. `Any` 类是所有继承关系的顶部，它定义了如下方法：
   - 相等和不等比较：

     ```
     final def ==(that:Any) : Boolean
     final def !=(that:Any) : Boolean
     def equals(that:Any) : Boolean
     ```

     - `==` 和 `!=` 是 `final` 的，因此不能被子类重写。
     - `==` 方法本质上等同于 `equals`，而 `!=` 一定是 `equals` 的反义。因此可以通过重写 `equals` 方法来定制 `==` 和 `!=` 的含义。
     - `Scala` 的相等性 `==` 被设计为：对类型的实际表现是透明的：对于值类型而言，它表示的是自然的数值相等性；对于引用类型而言，它表示的是从 `Object` 继承的 `equals` 方法的别名。
   - 哈希函数：

     ```
     def ## : Int
     def hasCode : Int
     ```

   - 格式化成字符串：

     ```
     def toString : String
     ```

2. 在 `Scala` 中，如果希望使用引用相等性而不是内容上的相等性，则使用 `eq` 方法。`AnyRef` 类定义了 `eq,ne` 方法来给出引用相等性和引用不等性。
3. `Any` 类有两个子类：`AnyVal` 和 `AnyRef`：`AnyVal` 是所有值类型的父类；`AnyRef` 是所有引用类型的父类。

   在 `Java` 平台上，`AnyRef` 事实上只是 `java.lang.Object` 的一个别名，因此 `Java` 编写的类和 `Scala` 编写的类都集成自 `AnyRef`。虽然可以在面向 `Java` 平台的 `Scala` 程序中任意切换 `Object` 和 `AnyRef`，但是推荐采用 `AnyRef`。

4.4.2 AnyVal 类

1. 可以通过继承 `AnyVal` 来定义自己的值类型。
2. `Scala` 提供了九个内建的值类型：
   `Byte,Short,Char,Int,Long,FLoat,Double,Boolean,Unit`。
   - 其中前八个对应 `Java` 的基本类型，它们的 `runtime` 是采用 `Java` 基本类型来表示的。

     这些类的实例在 `Scala` 中统统写作字面量，不能用 `new` 来创建这些类的实例。这是通过将这些值类型定义为 `abstract` 同时也是 `final` 这个技巧来完成的。
   - 最后一个 `Unit` 粗略的对应到 `Java` 的 `void` 类型，它有且只有一个实例值，写作 `()` 。
3. 值类型以方法的形式支持通常的算术和布尔运算操作符，同时它们还支持 `Any` 类的所有方法。
4. 值类空间是扁平的：所有的值类型都是 `scala.AnyVal` 的子类，但是它们之间并没有子类关系。

   而不同值类型之间存在隐式类型转换。
5. 可以对 `Int` 类型调用 `min,max,until,to,abs` 等方法。原理是：存在从 `Int` 类到 `RichInt` 类的隐式转换。

   只要对 `Int` 调用的方法没有在 `Int` 类中定义，而 `RichInt` 类刚好定义了该方法，则隐式转换就自动应用。

   其它的值类型也有类似的辅助类和隐式转换。

4.4.3 底类型

1. `Null` 类是 `null` 引用的类型，它是每个引用类的子类。

   `Null` 类并不兼容值类型，如：你无法将 `null` 赋值给一个整数变量。
2. `Nothing` 位于 `scala` 类继承关系的底部，它是任何其它类型的子类。

   实际上并不存在这个 `Nothing` 类型的任何值，其用途之一是给出非正常终止信号。

   如：

```
def error(message: String) : Nothing =
  throw new RuntimeException(message) // Nothing 表明该方法并不会正常返回

def f(x:Int,y:Int): Int =
  if (y!=0) x/y
  else error("can't divide by 0")
```

   由于 `error` 分支的类型为 `Nothing`，兼容于 `Int`，因此能够成功编译。

4.4.4 自定义值类型

1. 可以自定义值类型来对内建的值类型进行扩充。和内建的值类型一样，自定义的值类型通常也会编译成那种不使用包装类的 `Java` 字节码。

   而在需要包装类的环境中（如泛型），值类型将被自动装箱和拆箱。
2. 要使得某个类成为值类，该类必须满足以下条件：
   - 该类继承自 `AnyVal`。
   - 该类必须有且仅有一个参数，且在该唯一参数之前加上 `val`（参数化字段）。这可以让该参数作为字段被外界访问。
   - 该类的内部除了 `def` 之外不能有任何其它东西。
   - 该类不能有子类。

- 该类不能重写`equals`和`hashCode`。

```
class RMB(val amount:Int) extends AnyVal
{
    override def toString() = "$" + amount
}
```

`RMB`在`Scala`源码中的类型为`RMB`，但是编译后`Java`字节码中直接使用`Int`。

3. 一个良好的习惯是：对每个领域概念定义一个新的类，哪怕复用相同的类来实现不同用途也是可以的。甚至这样的类是细微类：既没有方法，也没有字段。

   这个方法可以有助于编译器在更多的地方帮助你。如：

```
def ROI(cost:Double, revenue:Double): Double = (revenue-cost)/cost
```

   这个函数用于计算企业的`ROI`。如果由于疏忽，将成本传给了`revenue`参数、将收入传给了`cost`参数，则计算结果是错误的。此时编译器并不会错误，因为它们都是`Double`类型。

   此时可以定义值类：

```
class Cost(val value: Double) extends AnyVal
class Revenue(val value: Double) extends AnyVal
def ROI(cost:Cost, revenue:Revenue) : Double = (revenue.value-
cost.value)/cost.value
```

五、Trait

1. `Scala`和`Java`中，不支持多重继承，任何一个类都继承自单个父类。`C++`支持多重继承，一个类可以继承自多个父类。

   为了实现类似多重继承的效果，`Java`采用接口`interface`来实现，而`Scala`采用特质`trait`来实现。

   `trait`是`Scala`代码复用的基础单元：`trait`将方法、字段封装起来，然后通过将它们混入`mix in`类的方式来实现复用。

   特质很像`Java`接口，但是特质比接口更强大：它不仅可以定义方法，还可以定义字段来保持状态。

2. 通过`trait`关键字来定义特质，其定义除了关键字不一样，其它都和类的定义相同。

```
trait T {
    def f = println("This is a trait")
}
```

   在特质定义中你可以做任何在类定义中做的事，语法也完全相同。除了以下两种情况：

   - 特质不能有任何“类”参数，即：那些传入类的主构造方法的参数。

     你可以这样定义一个类：`class C:(x:Int,y:Int)`，但是无法这样定义一个特质：`trait C:(x:Int,y:Int)`。

   - 类中的`super`调用是静态绑定的，而在特质中`super`是动态绑定的。

     - 在类中编写`super.toString`这样的代码时，你可以明确知道实际调用的是哪个实现。

- 在特质中编写 `super.toString` 这种代码时，你无法知道实际调用的是哪个实现，因为定义特质时 `super` 并没有被定义。

  具体哪个实现被调用，在每次该特质混入到某个具体类时，都会重新判定。

  这种 `super` 的奇怪行为是特征能够实现可叠加修改 `stackable modification` 的关键。

3. 特质也有继承关系，如果未明确给出超类，则它默认继承自 `AnyRef` 。
4. 一旦定义好特质，就可以通过 `extends` 或者 `with` 关键字将其混入 `mix in` 到类中。

   一般是混入特质，而不是继承特质。因为混入和继承有很多重要区别。
5. 可以通过 `extends` 来混入特质，这种情况下，类会隐式的继承了特质的超类。从特质继承的方法跟从超类继承的方法用起来一样。

```
class Parent{
    def f1 = println("This is Parent class")
}

trait T extends Parent{
    def f2 = println("This is a trait")
}

class Child extends T{ // Child 隐式继承了 Parent，Child 还可以使用 T 的方法
    def f3 = println("This is Child class")
}

val c = new Child()
c.f1          // 打印：This is Parent class
c.f2          // 打印：This is a trait
c.f3          // 打印：This is Child class
```

同时特质也定义了一个类型，但是有几个约束：

- 不能直接 `new` 一个特质。
- 只能使用特质中定义、以及特质从它的超类中继承而来的字段和方法。此时也能够支持多态和动态绑定。

```
val c: T = new Child()  // c 的类型是 T
c.f3                    // 编译失败，因为 T 没有 f3 成员
c.f2                    // 打印：This is a trait
c.f1                    // 打印：This is Parent class
val t = new T()         // 编译失败，无法直接 new 一个特质
```

6. 如果类已经通过 `extends` 继承自某个超类，如果还希望混入特质，则通过 `with` 关键字。

```
class Parent{
    def f1 = println("This is Parent class")
}
trait T {
    def f2 = println("This is a trait")
}

class Child extends Parent with T { // Child 显式继承了 Parent，并混入了 T
```

```
    def f3 = println("This is Child class")
}

val c = new Child()
c.f1          // 打印：This is Parent class
c.f2          // 打印：This is a trait
c.f3          // 打印：This is Child class
```

7. 如果希望混入多个特质，则可以通过多个 `with` 子句添加。
   - 如果显式继承自父类，则父类采用`extends`，每个特质一个 `with` 子句。
   - 如果没有显式继承，则第一个特质采用`extends`，后面每个特质一个 `with` 子句。

```
class Parent{
    def f1 = println("This is Parent class")
}
trait T1 {
    def f2 = println("This is a trait1")
}
trait T2 {
    def f3 = println("This is a trait2")
}

class Child extends Parent with T1 with T2 { // Child 显式继承了 Parent，并混入了 T1,T2
    def f4 = println("This is Child class")
}
```

8. 类可以重写特质的成员，语法和重写超类中的成员一样。
9. 特质的一个主要用途是：自动给类添加基于已有方法的新方法。即：特质可以丰富一个瘦接口，使其成为一个富接口。

   瘦接口和富接口代表了面向对象设计中经常面临的取舍，在接口实现方和接口使用方之间的权衡。
   - 富接口有很多方法，对调用方很方便，但是实现方需要做更多的工作。
   - 瘦接口方法较少，因此实现起来更容易，但是需要调用方编写更多的代码。

   `Java` 通常采用瘦接口，而`Scala` 倾向于采用富接口。因为`Scala` 可以通过给特质添加具体方法的形式，而这种投入是一次性的。你只需要在特质中实现这些方法一次，然后混入类中。你不需要在每个混入该特质的类中重新实现一遍。

   因此和没有特质的语言相比，`Scala` 中实现富接口的代价更小。
10. 要用特质来实现富接口，只需要定义一个拥有为数不多的抽象方法（接口中的瘦的部分）和可能数量很多的具体方法（这些具体方法基于那些抽象方法编写）的特质。然后你可以将这个特质混入到某个类，在类中实现接口中瘦的部分，最终得到一个拥有完整富接口实现的类。

```
abstract trait T{
    /***** 瘦接口部分：抽象 *****/
    def interface1(n:Int):Int
    def interface2(name:String):String

    /***** 富接口部分：具体 *****/
    def f1(m:Int) = {/* 调用 interface1 和 interface2 的具体实现 */}
    def f2(m:Int,n:String) = {/* 调用 interface1 和 interface2 的具体实现 */}
```

```
}

class C extends T{ // C 拥有了富接口
    /******  实现瘦接口  ******/
    override def interface1(n:Int) = {/* 具体实现 */}
    override def interface2(name:String) = {/* 具体实现 */}
}
```

11. 当需要实现某个可复用的行为集合时，都需要决定是用特质还是抽象类。有一些参考意见：
    - 如果某个行为不会被复用，则用具体的类。
    - 如果某个行为可能被用于多个互不相关的类，则用特质。
    - 如果想要从`Java`代码中继承某个行为，用抽象类。

      从`Java`类继承和从`Scala`类继承几乎一样，唯有一个例外：如果某个`Scala`只有抽象方法，则它会被翻译成`Java`的接口。
    - 如果计划将某个行为以编译好的形式分发，且预期会有外部的组织编写继承自它的类，则倾向于使用抽象类。因为当某个特质增加或者减少成员时，任何继承自该特质的类都需要被重新编译，哪怕这些类并没有任何改变。

      如果外部的使用方只是调用到这个行为，而没有继承，则特质也是可以的。
    - 如果没有任何线索，则推荐从特质开始。因为特质能让你保留更多选择。
12. 许多有经验的`Scala`程序员都在实现的初期采用特质。每个特质可以描述整个概念的一小块。随着设计逐步固化和稳定，这些小块可以通过特质混入，被组合成更完整的概念。

5.1 Ordered Trait

1. 特质的一个应用场景是比较大小。通常为了支持大小比较操作，我们会定义一些方法：

```
class C
{
    def < (that:C)  =  {/* 小于比较 */ }
    def > (that:C)  =  that < this
    def <= (that:C) =  (this < that) || (this == that)
    def >= (that:C) =  (this > that) || (this == that)
}
```

   实际上这个场景太普遍，因此`Scala`提供了专用的特质来解决，这个特质叫`Ordered`。

2. `Ordered`特质定义了`<,>,<=,>=`，你需要实现`compare`方法，而这些方法将基于`compare`方法来实现。

   因此`Ordered`特质允许你通过只实现一个`compare`方法来增强某个类，从而使其具有完整的比较操作。

```
class C(n:Int) extends Ordered[C]{
    val num = n
    def compare(that:C) = this.num -  that.num
}
```

   - `Ordered`特质要求混入时提供一个类型参数，如这里的`Ordered[C]`，其中`C`是你要比较元素的类。
   - `compare`方法用于比较调用者（这里是`this`），和传入对象（这里是`that`）。

- 如果二者相等，则返回 0 。
- 如果调用者较小，则返回负值。
- 如果调用者较大，则返回正值。

3. 注意：`Ordered` 特质并不会帮你定义 `equals` 方法，因为它做不到。因为当你用 `compare` 来实现 `equals` 时，需要检查传入对象的类型。而由于 `Java` 的类型擦除机制，`Ordered` 特质自己无法完成这个检查。

   因此你需要自定义 `equals` 方法，哪怕你已经继承了 `Ordered` 。

5.2 可叠加的修改

1. 特质除了用于将瘦接口转化成富接口之外，还可以用于为类提供可叠加的修改。
2. 一个典型的例子：对整数队列进行修改：
   - 可以将所有放入队列的整数翻倍。
   - 可以将所有放入队列的整数加一。
   - 可以将队列中的负数去掉。

```
import scala.collection.mutable.ArrayBuffer

abstract class MyQueue{
    def get(): Int
    def put(x:Int)
}
class BasicQueue extends MyQueue{
    private val buf = new ArrayBuffer[Int]
    def get() = buf.remove(0)
    def put(x:Int) = {buf += x}
}

val queue = new BasicQueue
queue.put(10)
println(queue.get())  // 打印结果：10


trait DoubleOp extends MyQueue{
    abstract override def put(x:Int) = {super.put(2*x)}
}
trait IncOp extends MyQueue{
    abstract override def put(x:Int) = {super.put(x+1)}
}

class DoubleBasicQueue extends BasicQueue with DoubleOp

val queue2 = new DoubleBasicQueue
queue2.put(10)
println(queue2.get())  // 打印结果：20
```

这里有两个要点：

- `DoubleOp` 声明了一个超类 `MyQueue` ，因此该特质只能够被混入那些同样继承自 `MyQueue` 的类。
- 该特质在一个声明为抽象的方法里做了一个 `super` 的调用。

- 对于普通的类而言，这样的调用是非法的，因为它们在运行时必定会失败。因为父类的`put`方法是抽象的，并未实现。
- 对于特质来说，这样的调用实际上可以成功。因为特质中的`super`是动态绑定的，只要在给出了该方法具体定义的特质或类之后混入，`DoubleOp`特质里的`super`调用就可以正常工作。

为了告诉编译器你是特意如此，必须将这样的方法标记为`abstract override`。这样的修饰符组合只允许用在特质的成员上，不允许用在类的成员上。其含义是：该特质必须混入某个拥有该方法具体定义的类中。

3. 如果类仅仅是继承然后混入特质，而并没有任何新的代码，这时候可以直接`new`而不必定义一个有名字的类。

```
class DoubleBasicQueue extends BasicQueue with DoubleOp
val queue2 = new DoubleBasicQueue
```

可以简化为：

```
val queue2 = new BasicQueue with DoubleOp
```

4. 这里的特质主要用作是代表某种修改`modification`，因为它们修改了底层类的状态，而不是定义并修改自己的状态。
   - 这些特质是可叠加的`stackable`。
   - 可以从这些特质中任意选择，将它们混入类。
5. 混入特质的顺序是重要的。大致上讲，越靠右出现的特质越先起作用。

   当你调用某个带有混入的类的方法时，最靠右端的特质中的方法最先被调用。如果该方法调用`super`，则它将调用左侧紧挨着它的那个特质的方法，以此类推。

5.3 特质和多重继承

1. 特质是一种从多个像类一样结构继承的方式，但是它和`C++`中的多重继承有重大区别。其中一个尤为重要的区别是：对`super`的解读。
   - 在多重继承中，`super`调用的方法在调用的地方就确定了。
   - 在特质中，`super`调用的方法取决于类和混入类的特质的线性化`linearization`。
2. 以一个简单例子来说明：

```
import scala.collection.mutable.ArrayBuffer

abstract class MyQueue{
    def get(): Int
    def put(x:Int)
}
class BasicQueue extends MyQueue{
    private val buf = new ArrayBuffer[Int]
    def get() = buf.remove(0)
    def put(x:Int) = {buf += x}
}

trait DoubleOp extends MyQueue{
    abstract override def put(x:Int) = {super.put(2*x)}
}
```

```
trait IncOp extends MyQueue{
    abstract override def put(x:Int) = {super.put(x+1)}
}
```

对于代码：

```
val q = new BasicQueue with DoubleOp with IncOp
q.put(1)
```

- 如果是多重继承，则需要考虑`q.put`究竟调用的是哪一个`put`方法。

  如果规则是最后一个特质胜出，则`DoubleOp`中的`put`会被执行；如果规则是第一个特质胜出，则`IncOp`中的`put`会被执行。

  因此，这种方式无法实线可叠加的修改。

- `Scala`的规则是：线性化。

  当用`new`实例化一个类的时候，`Scala`会将类及其所有继承的类和特质都拿出来，将其线性的排列在一起。

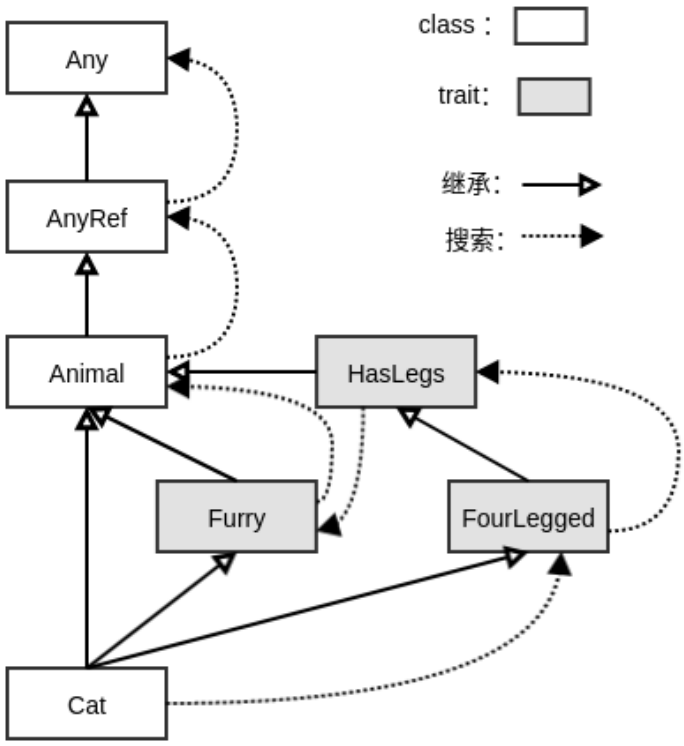
  当你在某个类中调用`super`的时，被调用的方法是这个链条中向上最近的那个。如果除了最后一个方法外，所有的方法都调用了`super`，则最终的结果就是叠加在一起的行为。

3. 在任何线性化中，类总是位于所有它的超类和混入的特质之前。因此当你写下调用`super`的方法时，该方法绝对是在修改超类和混入特质的行为。
4. `Scala`线性化的主要性质可以通过下面的例子说明：

```
class Animal
{
    def f = println("This is Animal")
}

trait Furry extends Animal
{
    override abstract def f = {
        println("This is Furry")
        super.f
    }
}
trait HasLegs extends Animal
{
    override abstract def f = {
        println("This is HasLegs")
        super.f
    }
}
trait FourLegged extends HasLegs
{
    override abstract def f = {
        println("This is FourLegged")
        super.f
    }
}
class Cat extends Animal with Furry with FourLegged
{
```

```
    override def f = {
        println("This is Cat")
        super.f
    }
}
```

其继承体系为：



注意：

- 即使`Animal`的`f`是具体的，`trait`中的`f`也可以是抽象的。
- 具体的类`Cat`中，`f`必须是具体的，也就是不能有`abstract`。

执行代码：

```
var cat = new Cat
cat.f
/*
输出为：
This is Cat
This is FourLegged
This is HasLegs
This is Furry
This is Animal
/*
```

`Cat`(`class Cat extends Animal with Furry with FourLegged`)的线性化从后到前计算过程如下：

- 最后一部分是超类`Animal`的线性化。这段线性化被直接复制过来而不加修改。

  由于`Animal`并不显式扩展自某个超类，也没有混入任何特质，因此其线性化看起来是这样的：`Animal -> AnyRef -> Any`。

- 线性化倒数第二部分是首个混入特质 `Furry` 的线性化。因为所有已经出现在 `Animal` 线性化中的类都不再重复出现，每个类在 `Cat` 的线性化当中只出现一次。因此结果为：

  `Furry ->Animal -> AnyRef -> Any` 。

- 接下来是 `FourLegged` 的线性化。同样的，任何之前出现的类都不再重复出现，因此结果为：

  `FourLegged -> HasLegs -> Furry ->Animal -> AnyRef -> Any` 。

- 最后，`Cat` 线性化中的第一个类是它自己，最终 `Cat` 的线性化结果为：

  `Cat -> FourLegged -> HasLegs -> Furry ->Animal -> AnyRef -> Any` 。

当这些类和特质中的任何一个通过 `supper` 调用某个方法时，被调用的是在线性化链条中出现在其右侧的首个实现。

如果定义为：

```
class Cat extends Animal with Furry with FourLegged
{
    override def f = {
        println("This is Cat")
    }
}
```

因为 `Cat` 中没有 `super` 调用，则并不会执行线性化链条搜索。

```
var cat = new Cat
cat.f
/*
输出为:
This is Cat
*/
```

五、包和导入

1. 在大型程序中，通常以模块化的风格编写代码，将程序切分成若干个较小的模块，从而降低耦合。

5.1 包

1. `Scala` 代码存在于 `Java` 平台全局的包层次结构当中。在 `Scala` 中，可以通过两种方式将代码放进带名字的包中：

   - 在文件顶部放置一个 `package` 子句，让整个文件的内容放进指定的包中。

     ```
     package com.xx.yy

     /* 现在整个文件的内容都在包中 */
     class C
     ```

   - 在 `package` 子句之后加上一段用花括号包裹起来的代码块，这个代码块包含了进入该包的定义。

     这个语法称作打包 `packaging`，类似于 `C#` 的命名空间。

这是个更加通用的表示法，允许我们在一个文件中包含多个包的内容。如：将某个类的测试代码和原始代码放置在同一个文件里，不过分成不同的包。

```
package com.xx.yy{
    /* 只有花括号中的内容在包中 */

    package model_a { // 原始 package
        class C       // 原始代码
    }
    package tests {   // 测试 package
        class C_Test  // 测试代码
    }
}
```

2. 由于`Scala`代码是`Java`生态的一部分，因此建议采用`Java`的包名习惯：将域名倒过来。
3. 采用包层次结构划分代码之后，不仅有助于人们浏览代码，同时也告诉编译器：同一个包中的代码之间存在某种相关性。
4. 在访问同一个包内的代码时，`Scala`允许我们采用简短的、不带限定前缀的名称。
    - 一个类不需要前缀就可以在自己的包内被访问。

```
package com.xx.yy {
    package model_a {
        class C1              // 在 model_a 内， C1 不需要前缀就可以访问
        class C2{
            val c = new C1    // 不需要 com.xx.yy.model_a.C1
        }
    }
}
```

    - 包自身也可以从包含它的包里不带前缀的访问到。

```
package com.xx.yy {
    package model_a {      // 在 com.xx.yy 内，model_a 不需要前缀就可以访问
         class C1
    }
    class C3 {
        val c = new model_a.C1 // 不需要 com.xx.yy.model_a
    }
}
```

    - 使用花括号打包语法时，所有在包外的作用域内可以被访问的名称，在包内也可以访问到。

```
package com.xx.yy {
    class C3
    package model_b { // model_b 可以直接访问 C3
        class C4 {
            // 由于外层 model_b 可以直接访问 C3，这里不需要 com.xx.yy.C3
            val c = new C3
        }
    }
}
```

- 这些访问规则只有当你显式的嵌套打包时才有效。如果你采用每个文件只有一个包的做法，则只有那些在当前包内定义的名称才直接可用。

```
package com.xx.yy {
    class C1
}

package com.xx.yy.model_a { // 现在 com.xx.yy.model_a 位于文件顶层
    class C2 {
        val c = new C1  // C1 不再可见，编译错误
    }
}
```

5. 如果花括号嵌套让你的代码缩进太多不方便阅读，则你可以用多个 `package` 子句但是不采用花括号。这称作链式包子句。

```
package com.xx.yy // 省略了花括号
class C1

package com.xx.yy // 省略了花括号
package model_a   // 省略了花括号
class C2{
    val c = new C1  // C1 可见
}
```

6. 有时候可能 `package` 名字相互遮挡。

```
// 位于文件 launch.scala 中
package launch {
    class Booster3
}

// 位于文件 mypackage.scala 中
package mypackage {
    package navigation {
        package launch {
            class Booster1
        }
        class MissionControl {
            val booster1 = new launch.Booster1
            val booster2 = new mypackage.launch.Booster2
            val booster3 = new _root_.launch.Booster3
```

```
        }
    }
    package launch {
        class Booster2
    }
}
```

- `MissionControl` 和 `Booster1` 的包 `launch` 位于同一个包，因此可以直接访问 `launch.Booster1` 。
- `navigation` 和 `Booster2` 的包 `launch` 位于同一个包，因此可以通过 `mypackage.launch.Booster2` 来访问。
- `Scala` 提供了一个名为 `_root_` 的包，这个包不会跟任何用户编写的包冲突。每个你编写的顶层包都被当作时 `_root_` 包的成员。因此 `_root_.launch.Booster3` 可以访问 `Booster3` 。

5.2 导入

1. 在 `Scala` 中，可以通过 `import` 子句导入包和包的成员。被导入的项可以通过 `File` 这样的简单名字访问，而不需要在前面加上包名或者对象名，如：`java.io.File` 。

```
package model_a

abstract class Fruit(
  val name: String,
  val color: String)

object Fruits {
    object Apple extends Fruit("apple","red")
    object Orange extends Fruit("orange","orange")
    val menu = List(Apple,Orange)
}
```

使用时可以通过四种方式导入：

- 导入包本身：

```
import java.util.regrex  // 导入 regrex 包
// 现在可以直接使用 regrex 这个名字了

class C {
    val pattern = regrex.Pattern.compile("[0-9]*abc")
}
```

- 导入单个类型：

```
import model_a.Fruit
// 现在可以直接使用 Fruit 这个名字了
```

- 导入包内的所有成员：

```
import model_a._
// 现在可以直接使用 Fruit, Fruits 等名字了，因为它们都是 model_a 的成员
```

这种方式与 `java` 稍有不同。`java` 中是星号 `*` ，而 `scala` 中是下划线 `_` 。因为 `scala` 中 `*` 是个合法的标识符。

- 导入对象的所有成员：

```
import model_a.Fruits._
// 现在可以直接使用 Apple,Orange,menu 等名字了，因为它们都是 Fruits 的成员
```

2. `Scala` 的导入比 `java` 的导入更加通用：
   - 导入可以出现在任何地方，不仅仅是在某个编译单元的最开始。
   - 可以导入任意对象或者包，而不仅仅是导入包。

```
def printFruit(fruit: Fruit) = {
    import fruit._                    // 导入可以出现在任何地方，可以导入对象
    println(name + "s are " + color)
}
```

这里导入了对象 `fruit` ，而不是包。它导入了对象 `fruit` 的所有成员，因此接下来的 `name,color` 等价于 `fruit.name` 和 `fruit.color` 。

   - 可以让你重命名并隐藏某些被导入的成员。做法是：在 `import` 中引入括在花括号中的导入选择器子句，这个子句跟在那个我们要导入成员的对象的后面。
     - 导入限定的成员：

```
import Fruits.{Apple,Orange}
// 这里只会从 Fruits 对象导入 Apple 和 Orange 这两个成员
```

     - 重命名被导入的成员：通过格式 `原名 => 新名` ：

```
import Fruits.{Apple => XXX,Orange}
// 这里只会从 Fruits 对象导入 Apple 和 Orange 这两个成员，但是 Apple 重命名为 XXX
```

     - 如果花括号里只有下划线，则等价于导入所有成员：

```
import Fruits.{_}
// 等价于 import Fruits._
```

     - 可以配合 `_` 和 `=>` ，此时表示：引入所有成员，但是将某些被导入成员重命名。

```
import Fruits.{Apple => XXX, _}
// 导入 Fruits 对象的所有成员，但是将 Apple 重命名为 XXX
```

     - 可以通过格式 `原名 => _` 来排除某个成员的导入。

```
import Fruits.{Apple => _, _}
//除了 Apple 之外， 导入 Fruits 的所有成员
```

3. `Scala` 导入选择器拥有巨大灵活性，可以包含：
   - 一个简单的名字 `x` ，这将把 `x` 包含在导入的名字集合里。

- 一个重命名子句 `x => y` ，这将让名字为`x` 的成员以名字 `y` 可见。
- 一个隐藏子句 `x => _`，这将会从导入的名字集合里排除 `x` 。
- 一个捕获所有的 `_` 。这将会导入除了之前规则中提到的成员之外的所有成员。

  如果希望捕获所有，则它必须出现在导入选择器列表的末尾。

前面给出的简单导入子句可以看作是带有选择器子句的导入子句的缩写。

如：`import p._` 等价于 `import p.{_}`； 而 `import p.n` 等价于 `import p.{n}` 。

4. `Scala` 对每个程序都隐式添加了一些导入。本质上，这就好像是为每个`.scala` 源码文件的顶部都添加了三行导入子句：

```
import java.lang._  // java.lang 包的全部内容
import scala._      // scala 包的全部内容
import Predef._     // Predef 对象的全部内容
```

   - `java.lang` 包包含了标准的`Java` 类。由于 `java.lang` 是隐式导入的，因此可以直接写 `Thread`，而不是 `java.lang.Thread` 。
   - `scala` 包包含了`Scala` 的标准类库。由于`scala` 是隐式导入的，因此可以直接写 `List`，而不是 `scala.List` 。
   - `Predef` 对象包含了许多类型、方法和隐式转换的定义，这些定义在`scala` 程序中经常被用到。

     由于`Predef` 是隐式导入的，因此可以直接写 `assert` ，而不是 `Predef.assert` 。
   - `Scala` 对这三个子句进行了一些特殊处理，后导入的会遮挡前面的。

     如：`scala` 包和 `java 1.5` 版本以后的 `java.lang` 包都定义了 `StringBuilder`类。由于 `scala` 的导入遮挡了 `java.lang` 的导入，因此 `StringBuilder` 这个名字会引用到 `scala.StringBuilder`，而不是 `java.lang.StringBuilder` 。

5.3 包对象

1. 可以添加到包里的代码有类、特质、孤立对象之外，还可以添加其它代码：任何能够放在`class level` 的定义，都能够放在`package level` 。
2. 如果你有某个希望在整个包都能用的`helper` 方法，可以将其放在包的顶层。具体做法是：将其定义放在包对象`package object`中 。

   每个包都允许有一个包对象，任何被放在包对象里的定义都被当作这个包本身的成员。
3. 包对象的语法和花括号"打包"很像，区别是包对象包含了一个`object` 关键字。包对象的花括号括起来的部分可以包含任何你希望添加的定义。
   - 任何包的任何其他代码都可以像引入类一样引入这些方法。
   - 包对象经常用于包级别的类型别名和隐式转换。
   - 顶层`scala` 包也有一个包对象，其中的定义对所有`Scala` 代码都可用。

```
// 位于文件 p1/package.scala 中
package object p1 {
    def printFruit(fruit: Fruit) = {
        import fruit._
        println(name + "s are " + color)
    }
}

// 位于文件 p2/package.scala 中
package p2
import p1.printFruit
/* 其它代码 */
```

4. 包对象会被编译成名为`package.class`的类文件，该文件位于它增强的包的对应目录下。

   源文件最好能保持相同的习惯，即：包对象的源码放在包名目录下的一个叫`package.scala`的文件中。

五、隐式类型转换

1. `Scala`支持隐式类型转换：这是一个定义为`implicit`的方法。

```
class C(n:Int,s:String)
{
    val num:Int = n
    val name:String = s
}
implicit def CtoInt(c:C) = c.num  // 隐式类型转换
val c = new C(123,"Hello")
println(1+c)  // 输出：124
```

   为了让隐式类型转换能够工作，它需要定义在作用域内。

2. 由于隐式类型转换是由编译器隐式的作用在代码上，而不是在代码中显式的给出。因此对于使用方程序员，究竟哪些地方隐式转换起了作用并不是那么直观。因此这会阻碍代码的可读性。
3. 隐式定义指的是我们允许编译器插入程序以解决类型错误的定义。例如：如果`x + y`无法通过编译，则编译器可能将它修改为`convert(x) + y`，其中`convert`是某种可用的隐式转换。

   如果`convert`是一种简单的转换函数，则不应该在代码里显式写出从而有助于澄清程序的主要逻辑。

4. 隐式转换规则：
   - 标记规则：只有标记为`implicit`的定义才可用。

     关键字`implicit`用于标记哪些声明可以被编译器用作隐式定义。可以用`implicit`来标记任何变量、函数或者对象的定义。编译器只会从那些显式标记为`implicit`的定义中选择。

   - 作用域规则：被插入的隐式转换必须是当前作用域的单个标识符，或者跟隐式转换的源类型或者目标类型有关联。

     `Scala`编译器只会考虑那些在当前作用域内的隐式转换，因此必须以某种方式将隐式转换定义引入到当前作用域。

除了一个例外，隐式转换在当前作用域中必须是单个标识符。如对于 `x + y`，编译器不会插入 `someVar.convert(x)` 这种形式的转换，你必须显式导入 `import someVar.convert` 来使用单个标识符 `convert`。事实上对于类库而言，通常提供一个包含了一些有用的隐式转换的 `Preamble` 对象，这样使用这个类库的代码就可以通过 `import Preamble._` 来访问该类库的隐式类型转换。

但是这个规则有一个例外：编译器还会在隐式转换的源类型或者目标类型的伴生对象中查找隐式定义。我们不需要在程序中 `import` 伴生对象。

- 每次一个规则：每次只能有一个隐式定义被插入。

  如：编译器绝对不会将 `x + y` 重写为 `convert1(convert2(x)) + y`。

  可以通过让隐式定义包含隐式参数的方式绕过这个限制。

- 显式优先原则：只要代码按照编写的样子能够通过类型检查，就不要尝试隐式定义。

  编译器不会对已经可以工作的代码做修改。

  这个规则必然可以得出结论：我们总是可以将隐式标识符替代成显式的，代码会更长但是歧义更少。这是一种折衷：

  - 如果代码看上去重复啰嗦，则隐式转换可以减少这种繁琐的代码
  - 如果代码看上去生硬晦涩，则可以显式的插入转换

  究竟是否采用隐式转换，这是代码风格问题。

5. 隐式转换可以使用任何名称。隐式转换的名称只有两种情况下重要：

   - 当你希望显式给出转换时
   - 当决定程序的哪些位置都有哪些隐式转换可用时。

   考虑一个带有两个隐式转换的对象：

   ```
   object MyConversions {
     implicit def stringWrapper(s :String) : IndexedSeq[Char] = ...
     implicit def intToString(x :Int): String = ...
   }
   ```

   如果你只是希望使用 `stringWrapper` 转换，并不希望使用 `intToString` 转换，则可以通过仅仅引入其中一个来实现：

   ```
   import MyConversions.stringWrapper
   ```

   在这里，隐式转换的名字很重要，因为只有这样才可以有选择的引入一个而不引入另外一个。

6. `Scala` 会在三个地方使用隐式定义：

   - 转换到一个预期的类型：在预期不同类型的上下文中使用某个类型。如你有一个 `String`，但是你要将它传递给一个要求 `IndexedSeq[Char]` 的方法。
   - 对某个（成员）选择接收端（即字段、方法调用）的转换：适配接收端的类型。

     如 `"abc".exists` 将转换为 `stringWrapper("abc").exists`，因为 `exists` 方法在 `String` 上不可用，但是在 `IndexedSeq` 上是可用的。

   - 隐式参数：用于给被调用函数提供更多关于调用者诉求的信息。

     隐式参数对于泛型函数尤其有用，被调用的函数可能完全不知道某个或某些入参的类型。

5.1 转换到一个预期的类型

1. 每当编译器看到一个 `X` 而它需要一个 `Y` 的时候，他就会查找一个能将 `X` 转换为 `Y` 的隐式转换。
2. 将 `Double` 隐式转换成 `Int` 可能并不是一个好主意，因为这会悄悄的丢掉精度。我们更推荐从一个受限的类型转换成更通用的类型。

   如：`scala.Predef` ，它是每个 `Scala` 程序都会隐式导入的对象，它定义了一些从 “更小” 的数值类型到 “更大” 的数值类型的隐式转换。如 `scala.Predef` 定义了：

```
implicit def int2double(x: Int): Double = x.toDouble
```

   `Scala` 中并没有什么强制类型转换，所有的类型转换都是通过这种隐式转换或者显式转换来实现的。

5.2 转换接收端

1. 隐式转换还能应用于方法调用的接收端，也就是调用方法的那个对象。这种隐式转换有两个用途：
   - 允许我们更平滑的将新类集成到已有的类继承关系图谱当中。
   - 支持在语言中编写（原生的）领域特定语言 `DSL` 。
2. 即如你写了 `obj.doIt`，而 `obj` 并没有一个 `doIt` 的成员，编译器会在放弃之前尝试插入转换。

   这里转换需要应用于接收端，也就是 `obj`。编译器会装作 `obj` 的预期“类型”为“拥有“名字为 `doIt` 的成员。这个”类型“ 并不是一个普通的 `scala` 类型，不过从概念上讲它是存在的。
3. 接收端转换的一个主要用途是：让新类型和已有类型的集成更为平滑。

   如定义了新类型：

```
class MyCls(n: Int, d: Int){
  ...
  def + (that: MyCls): MyCls = ...
  def + (that: Int): Mycls = ...
}
```

   则我们可以通过以下调用：

```
val obj = new MyCls(1,2)
obj + 1
```

   但是无法通过以下调用：

```
1 + obj
```

   因为作为接收端的 `1` 并没有一个合适的方法 `+ (MyCls)` 。为了允许这样的操作，我们可以定义一个 `Int` 到 `MyCls` 的隐式转换：

```
implicit def intToMyCls(x: Int) = new MyCls(x, 1)
1 + obj
```

背后的原理：`Scala` 编译器首先尝试对表达式 `1 + obj` 进行类型检查。虽然 `Int` 有多个 `+` 方法，但是没有一个是接收参数为 `MyCls` 类型的，因此类型检查失败。接着编译器检查一个从 `Int` 到另一个拥有可以应用 `MyCls` 参数的 `+` 方法的类型的隐式转换，编译器将会找到 `intToMyCls` 隐式转换并执行：

```
intToMyCls(1) + obj
```

4. 隐式转换的另一个主要用途是模拟添加新的语法。

   考虑创建一个 `Map` ：

   ```
   Map("a" -> 1, "b" -> 2)
   ```

   这里的 `->` 并不是 `scala` 的语法特性，而是 `ArrowAssoc` 类的方法。`ArrowAssoc` 是一个定义在 `scala.Predef` 对象这个 `scala` 标准前导 `preamble` 代码里的类。当写下 `"a" -> 1` 时，编译器将插入一个从 `"a"` 到 `ArrowAssoc` 的转换。

   下面是相关定义：

   ```
   package scala
   object Predef{
     class ArrowAssoc[A](x: A){
       def -> [B](y: B): Tuple2[A,B] = Tuple2(x,y)
     }
     implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] = new ArrowAssoc(x)
     ...
   }
   ```

   这种“富包装类”模式在给编程语言提供 `syntax-like` 的扩展的类库中非常常见。

   - 只要你看到有人调用了接受类中不存在的方法，那么很可能使用了隐式转换。
   - 如果你看到名为 `RichSomething` 的类（如：`RichInt,RichBoolean`），那么这个类很可能对 `Something` 类型增加了 `syntax-like` 的方法。

   富包装类的应用场景广泛，它可以让你做出以类库形式定义的内部 `DSL` 。

5. `Scala 2.10` 引入隐式类来简化富包装类的编写。隐式类是一个以 `implicit` 关键字开始的类。对这样的类，编译器会生成一个从类的构造方法参数到类本身的隐式转换。

   如：

   ```
   case class Rect(width: Int, height: Int)
   implicit class RectMaker(width: Int){
     def x(height: Int) = Rect(width, height)
   }
   val myRect = 3 x 4
   ```

   其调用过程为：

   - 由于 `Int` 类型没有一个名为 `x` 的方法，因此编译器会查找一个从 `Int` 到某个有该方法的类型的隐式转换。
   - 编译器将找到类 `RectMaker` 并执行转换，然后调用 `RectMaker` 的 `x` 方法。

   并不是所有的类都可以作为隐式类。

   - 隐式类不能是样例类。

- 隐式类的构造方法必须有且仅有一个参数。
- 隐式类必须存在于另一个对象、类或者特质里。

  在实际应用中，只要是用隐式类作为富包装类来给某个已有的类添加方法，该限制不是问题。

5.3 隐式参数

1. 编译器有时候将 `f(a)` 替换为 `f(a)(b)`，或者将 `new C(a)` 替换成 `new C(a)(b)`，通过追加一个参数列表的方式来完成某个函数调用。

   隐式参数调用提供的是整个最后一组柯里化的参数列表，而不仅仅是最后一个参数。如果 `f` 缺失的最后一个参数列表有三个参数，那么编译器将 `f(a)` 替换成 `f(a)(b,c,d)`。此时，不仅被插入的标识符，如 `b,c,d` 需要在定义时标记为 `implicit`，`f` 的最后一个参数列表在定义时也需要标记为 `implicit`。

2. 示例：

```
class A(val name: String)
{
  ...
}

object O{
  def f(s: String)(implicit a: A)={
    println(s + " " + a.name)
  }
}
```

   你可以显式调用：

```
val a = new A("a1")
O.f("hello")(a)
```

   也可以隐式调用，因为 `f` 的最后一个参数列表标记为 `implicit`：

```
implicit val imp_a = new A("implicit a")
O.f("hello")
```

   当调用 `O.f("hello")` 时，编译器自动填充最后一个参数列表。但是这里必须首先在作用域内找到一个符合要求的类型的 `implicit` 变量，这里为 `imp_a`。

   注意 `implict` 变量必须为当前作用域内的单个标识符，如：

```
object impObj{
  implicit val imp_a = new A("implict a")
}
O.f("hello")
```

   这种调用在编译期间报错，必须将 `imp_a` 引入到 `O.f("hello")` 的作用域。

3. 示例：

```
class A(val name: String)
```

```
{
  ...
}
class B(val name: String)
{
  ...
}
object O{
  def f(s: String)(implicit a: A, b: B)={
    println(s + " " + a.name + " " + b.name)
  }
}
object impObj{
  implicit val imp_a = new A("implict a")
  implicit val imp_b = new B("implict b")
}
import impObj._
O.f("hello")
```

这里 `implicit` 关键字是应用到整个参数列表而不是单个参数。

4. 隐式参数通常都是采用那些非常"稀有"或者"特别"的类型，防止意外匹配。
5. 隐式参数最常用的场景是：提供关于更靠前的那个参数列表中已经"显式"（与"隐式"相对应）提到的类型的信息。

   如：下面的函数给取出列表中的最大元素：

```
def maxElement[T](elements: List[T])(implicit ordering: Ordering[T]): T =
{
  elements match {
    case List() => throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxElement(rest)(ordering)
      if(ordering.gt(x, maxRest)) x else maxRest
  }
}
```

这里隐式参数 `ordering` 用于提供关于前面提到的 `elements` 中的类型 `T` 的排序信息。由于调用 `maxElements` 时必须给出 `elements`，因此编译器在编译时就会知道 `T` 是什么，因此就能确定类型 `Ordering[T]` 的隐式定义是否可用。如果可用，则它就可以隐式的作为 `ordering` 传入第二个参数列表。

```
maxElement(List(1,5,7,2))    // 编译器插入针对 Int 的 ordering
maxLement(List("one","two","three")) // 编译器插入针对 String 的 ordering
```

6. 从代码风格而言，最好是对隐式参数使用特殊的、定制化的类型。

   前面的例子也可以这样写：

```
def maxElement[T](elements: List[T])(implicit ordering: (T, T) =>
Boolean): T = {
  ...
}
```

这个版本的隐式参数 `ordering` 的类型为 `(T, T) => Boolean`，这是一个非常泛化的类型，覆盖了所有从两个 `T` 到 `Boolean` 的函数。这个类型并没有给出任何关于 `T` 类型的信息，它可以是相等性测试、小于等于测试、大于等于测试.... 。

而前面的例子：

```
def maxElement[T](elements: List[T])(implicit ordering: Ordering[T]): T =
{
  ...
}
```

它用一个类型为 `Ordering[T]` 的参数 `ordering`。这个类型中的单词 `Ordering` 明白无误的表达了这个隐式参数的作用：对类型为 `T` 的元素进行排序。由于这个类型更为特殊，因此在标准库中添加相关隐式定义不会带来什么麻烦。

与此相反，如果我们在标准库里添加一个类型为 `(T,T) => Boolean` 的隐式定义，则编译器广泛的自动传播这个定义，这会带来很多稀奇古怪的行为。

因此有这样的代码风格规则：在给隐式参数的类型命名时，使用一个能确定其职能的名字。

5.4 上下文界定

1. 当我们使用隐式参数时，编译器不仅会给这个参数提供一个隐式值，它还会将这个参数作为一个可以在方法体中使用的隐式定义。

   如前面的例子：

   ```
   def maxElement[T](elements: List[T])(implicit ordering: Ordering[T]): T =
   {
     elements match {
       case List() => throw new IllegalArgumentException("empty list!")
       case List(x) => x
       case x :: rest =>
         val maxRest = maxElement(rest) // 这里会隐式添加 (ordering)
         if(ordering.gt(x, maxRest)) x else maxRest // 这里必须显式给出
     }
   }
   ```

   编译器检查 `maxElement(rest)` 时发现类型不匹配，由于第二个参数列表是隐式的，因此编译器并不会立即放弃类型检查。它会查找合适类型的隐式参数，在上述代码中这个类型是 `Ordering[T]`。编译器找到了这样一个隐式参数，并将方法调用重写为 `maxList(rest)(ordering)` 。

2. 还有一种方法可以去掉对 `ordering` 的显式使用。这涉及标准类库中定义的方法：

   ```
   def implicitly[T](implicit t: T) = t
   ```

   调用 `implicitly[Foo]` 的作用是编译器会查找一个类型为 `Foo` 的隐式定义，然后编译器用这个对象来调用 `implicitly` 方法，该方法直接将这个隐式对象返回。

   因此，如果希望在当前作用域内找到类型为 `Foo` 的隐式对象，则可以直接写 `implicitly[Foo]` 。

   采用这个方式之后，上述例子改为：

```
def maxElement[T](elements: List[T])(implicit ordering: Ordering[T]): T =
{
  elements match {
    case List() => throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxElement(rest) // 这里会隐式添加 (ordering)
      if(implicitly[Ordering[T]].gt(x, maxRest)) x else maxRest // 使用
implicitly
  }
}
```

在这个版本中，方法体内没有任何地方提到`ordering`参数。

这个模式很常用，因此`Scala`允许我们省掉这个参数并使用上下文界定`context bound`来缩短方法签名。采用上下文界定的方式为：

```
def maxElement[T : Ordering](elements: List[T]): T = { // 使用上下文界定
  elements match {
    case List() => throw new IllegalArgumentException("empty list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxElement(rest) // 这里会隐式添加 (ordering)
      if(implicitly[Ordering[T]].gt(x, maxRest)) x else maxRest //
implicitly
  }
}
```

这里`[T: Ordering]`这样的语法是一个上下文界定，它完成两件事：

- 首先，它像平常那样引入一个类型参数`T`
- 其次，它添加了一个类型为`Ordering[T]`的隐式参数。至于这个隐式参数叫什么名字，你完全不需要知道。

  你只需要知道这个隐式参数的类型，并通过`implicitly[Ordering[T]`获得该隐式参数即可。

3. 直观上讲，可以将上下文界定想象为对类型参数做某种描述：
   - `[T <: Ordering[T]]`表示`T`是一个`Ordered[T]`类型或者其子类
   - `[T : Ordering]`并没有任何关于`T`是什么类型的定义，仅仅表示`T`带有某种形式的排序。

5.5 多个转换可用

1. 当作用域内存在多个隐式转换可用时，大部分场景`Scala`编译器都会拒绝插入转换。

   在`Scala 2.8`中对这个规则有所放宽：如果所有可用的转换中，某个转换比其它更为具体`more specific`，那么编译器就会选择这个更为具体的转换。

   如果满足下面任何一条，则我们就说某个隐式转换比另一个转换更为具体：

   - 前者的入参类型是后者入参类型的子类型。
   - 两者都是方法，而前者所在的类扩展自后者所在的类。

   增加这个修改规则的动机是：改进`Java`集合、`Scala`集合、字符串之间的互操作。

例如在 `Scala 2.8` 之前：

```
"abc".reverse.reverse == "abc"
```

这个表达式结果是 `false` 。原因是 `cba` 的类型是 `String` ， `Scala 2.8` 之前 `String` 没有 `.reverse` 操作，因此字符串被转换成了 `Scala` 的集合，而对集合的 `reverse` 返回的是另一个集合，因此表达式左侧返回一个集合，它不等于字符串。

在 `Scala 2.8` 之后， `Scala` 提供了一个更具体的从 `String` 转换到 `StringOps` 的隐式转换。 `StringOps` 有很多像 `reverse` 这样的方法，不过它们并不返回集合，而是返回字符串。

到 `StringOps` 的隐式转换直接定义在 `Predef` 中。

5.6 调试

1. 在调试时，可以将转换显式写出。如果显式写出还出错，你就能很快定位问题；如果显式写出不报错，则说明某个其它规则（如作用域规则）阻止了该隐式转换。
2. 可以通过 `-Xprint:typer` 编译器选项查看编译器插入的隐式转换。
3. 如果隐式转换被频繁使用，则会让代码变得难以阅读。因此在添加一个新的隐式转换之前，首先问自己能否通过其它手段达到相似的效果，比如继承、混入或者方法重载。

六、编译和执行

1. `Java` 和 `Scala` 的区别之一： `Java` 要求将类放入跟类同名的文件中，而 `Scala` 可以任意命名 `.scala` 文件。

   但是在非脚本的场景，推荐将类放入以类名命名的文件中，这便于程序员更容易的根据类名来定位到对应的文件。

2. 可以通过 `scalac` 这个编译器来编译 `Scala` 的类文件：

   ```
   scalac A.scala B.scala
   ```

   每次编译器启动时，它都会花时间扫描 `jar` 文件的内容以及执行其它一些初始化工作，然后才开始关注你提交的新的源码文件。

3. 也可以使用一个名为 `fsc` 的 `Scala` 编译器的守护进程：

   ```
   fsc A.scala B.scala
   ```

   - 首次运行 `fsc` 时，它将创建一个本地的服务器守护进程，绑定到计算机的某个端口上。然后它会通过这个端口将需要编译的文件发送给这个守护进程来编译。

     下次运行 `fsc` 时，守护进程已经在运行中，因此 `fsc` 会简单地将文件发送给这个守护进程，然后守护进程立即编译这些文件。

     因此只有在首次运行 `fsc` 时才需要等待 `Java runtime` 的启动。

   - 通过命令 `fsc -shutdown` 可以停止该守护进程。

4. 无论是通过 `scalac` 还是 `fsc` 命令，都会产出 `Java` 类文件，这些类文件可以用 `scala` 命令来运行。

   与之前执行脚本文件不同，这时候需要的是那个包含了符合正确签名要求的 `main` 方法的独立对象的名字。

   如： `scala A param1 param2` 。

注：`Scala` 的脚本文件必须以一个可以计算出结果的表达式结尾。

七、可变对象

1. 在 `Scala` 中可以定义带有可变状态的对象。当我们希望对真实世界中那些随着时间变化的对象进行建模时，自然而然就会想到这样的可变对象。
2. 不可变对象的一个特点是：当多次调用某个不可变对象的方法或者获取其字段时，总能得到相同的结果。如：

```
val list = List('a','b','c')
```

对 `list.head` 的调用总是会返回 `'a'` ，无论调用多少次、无论调用前后执行了什么其它操作。

3. 可变对象的一个特点是：方法调用或者字段访问的结果可能取决于该对象被执行了哪些操作。

```
class C {
    private var num: Int = 0
    def currentNum: Int = num
    def addN(n: Int) = {
        require(n>0)
        num += n
    }
    def subN(n: Int): Boolean = {
        if(n>num) false
        else:
        {
            num -= n
            true
        }
    }
}
val c = new C
```

对 `c.currentNum` 的同样的、多次调用结果可能会不同，这取决于调用前后是否执行了 `c.addN` 和 `c.subN` 操作。

4. 对象的可变和 `var` 通常成对出现，但是有时候不是那么清晰。
   - 一个类可能并没有定义或者继承任何 `var` 变量，但是它依然是可变的。因为该类将方法调用转发到其它带有可变状态的对象上。
   - 一个类可能包含了 `var` ，但是它依然是纯函数式的。如：

```
class Key {
    def computeKey: Int = ....          // 计算 key，这需要时间
}
class CachedKey extends Key {
    private var keyCache: Option[Int] = None
    def computeKey: Int = {
        if (!keyCache.isDefined) keyCache = Some(super.computeKey) //
写缓存
        keyCache.get            // 从缓存中读取
    }
}
```

通过使用 `CachedKey`，可以加速 `computeKey` 。除了速度上的提升，`Key` 类和 `CachedKey` 类的行为完全一致。如果说 `Key` 类是纯函数式的，那么 `CachedKey` 也是纯函数式的，即使它包含一个 `var` 变量。

5. 对一个可被重新赋值的变量，我们可以做两种基本操作：获取它的值，修改它的值。
   - 在诸如 `JavaBeans` 的类库中，这些操作通常被包装成单独的 `getter` 和 `setter` 方法，我们需要显式的定义这些方法。
   - 在 `Scala` 中，每个非私有的 `var` 成员都隐式定义了对应的 `getter` 和 `setter` 方法。不过这些 `getter` 方法和 `setter` 方法的命名和 `Java` 的习惯不一样，对于 `var x` ：
     - `getter` 方法的名字为 `x` 。
     - `setter` 方法的名字为 `x_=` 。

```
class C{
  var name:String = "hello"
}
val c = new C
println("get name:"+c.name)   // 输出: get name:hello
c.name = "world"
println("after setter: get name:"+c.name)// 输出: after setter: get
name:world
```

这里除了定义一个可被重新赋值的字段外，编译器还将生成一个名为 `name` 的 `getter` 和一个名为 `name_=` 的 `setter` 。

- 其中的字段总是被标记为 `private[this]` ，这意味着该字段只能从包含它的对象中访问。
- `getter` 和 `setter` 拥有跟原来的 `var` 相同的可见性。
  - 如果原来的 `var` 定义是公有的，则它的 `getter` 和 `setter` 也是公有的。
  - 如果原来的 `var` 定义是 `protected` 的，则它的 `getter` 和 `setter` 也是 `protected` 的。

因此上例中的代码等效于：

```
class C{
    private[this] var n = "hello"
    def name: String = n
    def name_=(x:String) = { n = x }
}
```

6. `var` 展开成 `getter` 和 `setter` 的机制的一个有趣点是：你可以逆向使用该机制，人工订制 `getter` 方法和 `setter` 方法。

```
class C{
  private[this] var n:Int = 0
  def age: Int = n
  def age_=(num:Int) = {
    require(0<= num && num<=120)
    n = num
  }
}
```

```
val c = new C
println("get age:"+c.age) // 输出: get age:0
c.age = 20
println("after setter: get age:"+c.age) // 输出: after setter: get age:20
c.age = -1  // 异常
```

其优点是：你可以在人工定制的 `setter` 方法里增加一些特殊的配置，如：增加参数限制、记录此次修改、发送通知、触发事件等。

其它语言有些特殊的语法来表示，如 `python` 语言的 `property` 语法。

7. `Scala` 允许我们定义不跟任何字段关联的 `getter` 和 `setter` 。

距离可以用 `米` 来 `千米` 来表示，因此我们内部存储的单位是 `米` ，但是 `getter` 和 `setter` 操作的是 `千米` 。

```
class Distance{
  private[this] var m:Double = 0           // 单位: 米
  def km: Double = m/1000                  // getter: 单位: 千米
  def km_=(num:Double) = { m = num*1000 }  // setter: 单位: 千米
}

val d = new Distance
println("get km:"+d.km)  // 输出: get km:0.0
d.km = 10
println("after setter: get km:"+ d.km) // 输出: after setter: get km:10.0
```

八、类型参数化

8.1 类型参数化

1. 类型参数化用于编写泛型的类和泛型的特质。泛型的意思是：我们用一个泛化的类或者特质来定义许多具体的类型。

```
trait Queue[T]{
    def head: T
    def tail: Queue[T]
    def enqueue(x: T): Queue[T]
}
```

这里的 `Queue` 是泛型的， 是一个泛型的特质，其定义为 `Queue[T]` 。

事实上 `Queue` 并不能当作一个类型来用，因此我们不能创建类型为 `Queue` 的变量：

```
def f(q: Queue) = {}
```

我们必须参数化 `Queue` ，指定参数化类型 `T` 。如： `Queue[Int],Queue[String],...` 。

```
def f(q: Queue[Int]) = {}
```

因此 `Queue` 也被称作类型构造方法，因为我们可以通过指定类型参数来构造一个类型。类型构造方法 `Queue` 能够"生成"一组类型，如： `Queue[Int],Queue[String],...` 。

8.2 型变注解

1. 泛型和子类型这两个概念放在一起，会产生一些非常有趣的问题。如：`Queue[String]` 应不应该被当作 `Queue[AnyRef]` 的子类型?

   更通俗的讲：如果 `S` 是类型 `T` 的子类型，那么 `Queue[S]` 是否应该作为 `Queue[T]` 的子类型?

   如果 `Queue[S]` 可以作为 `Queue[T]` 的子类型，则可以说特质 `Queue` 在类型参数 `T` 上是协变的 `convariant` 。由于 `Queue` 只有一个类型参数，因此可以简单的说 `Queue` 是协变的。这意味着以下的调用是允许的：

   ```
   def f(q: Queue[AnyRef]) = {}   // 函数参数类型: Queue[AnyRef]
   val q1: Queue[String] = ....
   f(q1)                          // 实参类型: Queue[String]
   ```

2. 在 `Scala` 中，泛型类型默认的子类型规则是非协变的 `nonvariant` 。也就是说，`Queue[String]` 不能当作 `Queue[AnyRef]` 来使用。

3. 可以通过在类型参数前面加上 `+` 来表示协变的：

   ```
   trait Queue[+T]{
       ...
   }
   ```

   通过 `+` ，我们告诉 `Scala` 我们要的效果是： `Queue[String]` 是 `Queue[AnyRef]` 的子类型。编译器会检查 `Queue` 的定义符合这种子类型关系的要求。

4. 也可以通过在类型参数前面加上 `-` 来表示逆协变的 `contravariance` ：

   ```
   trait MyTrait[-T]{
       ...
   }
   ```

   如果 `T` 是类型 `S` 的子类型，则 `MyTrait[S]` 是 `MyTrait[T]` 的子类型。也就是说，`MyTrait[AnyRef]` 是 `MyTrait[String]` 的子类型。

5. 类型参数是协变的、逆变的、还是不变的，这被称作类型参数的形变 `variance` 。可以放在类型参数旁边的 `+` 和 `-` 被称作类型注解 `variance annotation` 。

6. 在纯函数式的世界中，许多类型自然而然的是协变的。但是引入可变数据之后，情况就会发生变化。

   ```
   class Cell[+T](init: T) {      // 实际上无法通过编译
       private[this] var current = init
       def get = current
       def set(x: T) = { current = x }
   }
   ```

   假设 `Cell` 是协变的（实际上这段代码无法通过编译器的检查），则可以构建如下语句：

```
val c1 = new Cell[String]("hello")
val c2: Cell[Any] = c1          // Cell[String] 是 Cell[Any] 的子类型
c2.set(1)                       // Int 是 Any 的子类型
val s: String = c1.get          // current 现在是整数 1
```

这四行代码的效果是：将整数 `1` 赋值给了字符串 `s` 。这显然违背了类型约束。

7. 并不仅仅只有可变字段能让协变类型变得不可靠，泛型参数类型作为方法参数类型出现时，协变类型也不可靠。

```
class Cell[+T](init: T) {      // 实际上无法通过编译
    private[this] var current = init
    def get = current
    def set(x: T) = { current = x }
}

class MyCell extends Cell[Int]{
    override def set(x: Int) ={
        println(math.sqrt(x))
        super.set(x)
    }
}
```

因为 `Cell[Int]` 是 `Cell[Any]` 的子类，而 `MyCell` 又是 `Cell[Int]` 的子类，因此现在我们可以这么做：

```
val c: Cell[Any] = new MyCell
c.set("hello")
```

`c.set` 会执行 `MyCell` 的 `set` 方法，而该方法要求的参数类型是 `Int` 。

事实上，可变字段能让协变类型变得不可靠只是如下规则的特例：用 `+` 注解的类型参数不允许应用于方法的参数类型。

8. `Java` 中的数组是被当作协变来处理的。`Java` 在运行时会保存数组的元素类型。每当数组元素被更新时，都会检查新元素值是否满足类型要求。如果新元素不满足类型要求，则抛出 `ArrayStoreException` 异常。

```
// Java 代码，能够编译成功，但是运行异常
String[] a1 = { "abc" };
Object[] a2 = a1;
a2[0] = new Integer(17);
String s = a1[0]
```

`Java` 的这种设计是为了用一种简单的手段来泛化地处理数组。

而在 `Scala` 中，数组是不变的。因此 `Array[String]` 并不会被当作 `Array[Any]` 的子类来处理。

```
val a1 = Array("abc")
val a2: Array[Any] = a1 // 编译失败
```

但是 `Scala` 允许我们将元素类型为 `T` 的数组，经过类型转换成 `T` 的任意超类型的数组：

```
val a1 = Array("abc")
val a2: Array[Any] = a1.asInstanceOf[Array[Any]]
```

这个类型转换在编译时永远合法，且在运行时也永远成功。因为 `JVM` 的底层 `runtime` 模型对数组的处理都是协变的，跟 `Java` 语言一样。但是你可能在这之后遇到 `ArrayStoreException`，就跟 `Java` 一样：

```
val a1 = Array("abc")
val a2: Array[Any] = a1.asInstanceOf[Array[Any]]
a2(0) = 17        // 抛出 ArrayStoreException
```

8.3 逆变

1. 类型系统设计的一个通用原则：如果在任何需要类型 `U` 的值的地方，都能够用类型 `T` 的值替代，则可以安全的假定类型 `T` 是类型 `U` 的子类型。这称作李氏替换原则。

   如果 `T` 支持跟 `U` 一样的操作，而 `T` 的所有操作跟 `U` 中对应的操作相比，要求更少且提供更多的话，该原则就成立。

2. 有些场景需要逆变。

   ```
   trait OutputChannel[-T]{
       def write(x: T)
   }
   ```

   这里 `OutputChannel` 被定义为以 `T` 逆变，因此一个 `OutputChannel[AnyRef]` 是 `OutputChannel[String]` 的子类。

   这是因为 `OutputChannel[AnyRef]` 和 `OutputChannel[String]` 都支持 `write` 操作，而这个操作在 `OutputChannel[AnyRef]` 中的要求比 `OutputChannel[String]` 更少。更少的意思是：前者只要求入参是 `AnyRef`，后者要求入参是 `String` 。

3. 有时候逆变和协变会同时出现。一个典型的例子是 `Scala` 的 `Function` 特质。

   当我们写下函数类型 `A => B` 时，`Scala` 会将其展开成 `Function1[A, B]`。标准类库中的 `Function1` 同时使用了协变和逆变：函数入参类型 `A` 上进行逆变，函数结果类型 `B` 上进行协变。

   ```
   trait Function1[-A, +B]{
       def apply(x: A): B
   }
   ```

   这是因为对于函数调用，我们可以传入 `A` 的子类对象；而函数的返回值可以传递给 `B` 的超类对象。

8.3 检查型变注解

1. `Scala` 编译器会检查你添加在类型参数上的任何型变注解。如：如果你尝试声明一个类型参数为协变的（添加一个 `+` ），但是有可能引发潜在的运行时错误，则你的程序将无法通过编译。
2. 为了验证型变注解的正确性，`Scala` 编译器会对类或者特质定义中的所有能够出现类型参数的地点进行归类，归类为：协变的 `positive`、逆变的 `negative` 和不变的 `neutral` 。
   - 所谓的“地点”指的是：类或特质中，任何一个可以用到类型参数的地方。

- 编译器会检查类型参数的每一次使用：
  - 使用 `+` 注解的类型参数只能用在协变点。
  - 使用 `-` 注解的类型参数只能用在逆变点。
  - 没有型变注解的类型参数能够用在任何能够出现类型参数的点，因此这也是唯一的能用在不变点的类型参数。

3. 为了对类型参数点进行归类，编译器从类型参数声明开始，逐步深入到更深的嵌套层次。
   - 声明该类型参数的类的顶层的点被归类为协变点。
   - 更深的嵌套层次默认为跟包含它的层次相同，不过有一些例外情况归类会发生变化：
     - 值函数的参数的点被归类为：方法外的翻转：
       - 协变点的翻转是逆变点。
       - 逆变点的翻转是协变点。
       - 不变点的翻转仍然是不变点。
     - 当前的归类在方法的类型参数上也会翻转。

```
class C[-T,+U]
{   // T,U 在顶层都是协变点
    def func(t:T, u:U){}
    // func 为方法，因此翻转：T,U 都为逆变点
}
```

     - 当前的归类在类的类型参数上也会翻转。

```
class C[-T,+U]
{   // W 在顶层是协变点
    def func[W](){}
    // W 为类型，因此翻转：W 为逆变点
}
```

   - 要想跟踪型变点相当不容易。不过不用担心，`Scala` 编译器会帮助你做这个检查。
   - 一旦归类被计算出来，编译器会检查每个类型参数只被用在了正确的归类点。

4. `Scala` 的型变检查对于对象私有定义 `private[this]` 有一个特殊规则：在检查带有 `+` 或 `-` 的类型参数必须匹配相同型变归类点时，会忽略掉对象私有的定义。

8.4 上界/下界

1. 对于例子中：

```
trait Queue[T]{
    def head: T
    def tail: Queue[T]
    def enqueue(x: T): Queue[T]
}
```

由于 `T` 是以 `enqueue` 方法的参数出现，因此 `T` 不能是协变的。

事实上可以通过给 `enqueue` 一个类型参数，并对这个类型参数使用下界来实现多态：

```
trait Queue[T]{
    def head: T
    def tail: Queue[T]
    def enqueue[U >: T](x: U): Queue[U]
}
```

新的定义给`enqueue`添加了一个类型参数`U`，并且用`U >: T`这样的语法定义了`U`的下界为`T`。这样一来`U`必须是`T`的超类。

现在`enqueue`的参数类型为`U`而不是`T`，方法的返回值是`Queue[U]`而不是`Queue[T]`。

2. 超类和子类关系是反身的。即：一个类型同时是自己的超类和子类。对于`U >: T`，尽管`T`是`U`的下界，你仍然可以将一个`T`传入`enqueue`。
3. 上界的指定方式跟下界类似，只是不再采用表示下界的`>:`符号，而是采用`<:`符号。

   对于`U <: T`，要求类型参数`U`是`T`的子类型。
4. 型变注解和上、下界配合得很好。它们是类型驱动设计得绝佳范例。

8.5 类型推断

1. 对于方法调用`func(args)`，类型推断算法首先检查`func`的类型是否已知。
   - 如果已知，则这个类型信息就被用于推断入参的预期类型。

     如：`List(1,2,3,4).sortWith( _ > _)`中，调用对象的类型为`List[Int]`，因此类型推断算法知道`sortWith`的参数类型为：`(Int,Int) => Boolean`，并且返回类型为`List[Int]`。

     由于该参数类型已知，所以并不需要显式写出来。
   - 如果未知，则类型推断算法无法自动推断入参类型。

     ```
     def msort[T](less:(T,T) => Boolean)(xs: List[T]): List[T] ={
       ...
     }
     msort ( _ > _)(List(1,2,3,4))     // 错误
     msort[Int]( _ > _)(List(1,2,3,4)) // 正确
     msort[Int]( (x:Int, y:Int) => x > y )List(1,2,3,4)) // 正确
     ```

     `msort`是一个经过科里化的、多态的方法类型，它接收一个类型为`(T, T) => Boolean`的入参，产出一个从`List[T]`到`List[T]`的函数，其中`T`是当前未知的某个类型。`msort`需要先用一个类型参数实例化之后才能应用到它的入参上。

     由于`msort`确切实例类型未知，因此类型推断算法无法推断首个入参类型。此时类型推断算法尝试从入参类型来决定方法的正确实例类型。但是当它检查`_ > _`时，无法得到任何类型信息。

     - 一种方案是：显式传入类型参数，如`msort[Int]( _ > _)(List(1,2,3,4))`。
     - 另一种方案是：交换两个参数列表的位置：

       ```
       def msort[T](xs: List[T])(less: (T, T) => Boolean): List[T] ={
         ...
       }
       ```

当类型推断算法需要推断一个多态方法的类型参数时，它会考虑第一个参数列表里的所有入参的类型，但是不会考虑第二个、第三个等等参数列表的入参。

因此，当我们设计一个接收非函数的入参和接受函数入参时，将函数入参单独放在最后一个参数列表中。这样一来，方法的正确实例类型可以从非函数入参推断而来，而这个类型又可以继续用于对函数入参进行类型检查。这样的效果是：编写函数字面量作为入参时可以更简洁。

九、抽象成员

1. 如果类或者特质的某个成员在当前类中没有完整的定义，则它就是抽象的。抽象的本意是为了让声明该成员的类的子类来实现。不过 `Scala` 走的更远，它将这个概念完全泛化：除了可以声明抽象方法之外，还可以声明抽象字段甚至抽象类型作为类或特质的成员。

   下面这个特质声明了四种抽象成员：抽象类型 `T`、抽象方法 `transform`、抽象 `val` 字段 `initial`、抽象 `var` 字段 `current` 。

   ```
   trait Abstract{
     type T
     def transform(x: T): T
     val initial : T
     var current : T
   }
   ```

   因此 `Abstract` 特质的具体实现需要填充每个抽象成员的定义。

   ```
   class MyCls extends Abstract{
     type T = String
     def transform(x: String) = x + x
     val initial = "hi"
     val current = initial
   }
   ```

2. 抽象类型 `abstract type` 成员：用 `type` 关键字声明为某个类或者特质的成员、但是未给出定义的类型。

   注意：抽象类不等于抽象类型，抽象类型永远是类或者特质的成员。而抽象类指的是通过修饰符 `abstract` 定义的类。

   ```
   trait Abstract{
     type T  // 抽象类型
     ...
   }
   abstract class Element {  // 抽象类
       def contents: Array[String]
   }
   ```

   与抽象类型成员对应的是具体类型成员，具体类型成员可以认为是某个具体类型的别名。

   ```
   class MyCls extends Abstract{
     type T = String  // T 是 String 的别名
   }
   ```

使用类型成员的原因之一是：给名字冗长的类型、或者含义不清晰的类型一个简短的、含义明确的别名。

另一个原因是：声明子类必须定义的抽象类型（如这里的 `type T`）。

3. 抽象的 `val` 成员：

```
val initial : String
```

该声明给出了 `val` 的名称和类型，但是未给出具体值。这个值必须由子类中具体的 `val` 定义提供。

这种抽象成员的应用场景：不知道变量具体的值，但是明确的知道它在当前类的每个实例中都不可变时，可以采用抽象的 `val` 成员。

- 采用抽象 `val` 成员可以有如下保证：每次对 `.initial` 的引用都会交出相同的值。

  如果采用抽象的无参方法成员：

  ```
  def initial : String
  ```

  则每次方法调用无法给出这个保证。

- 因此抽象 `val` 限制了它的合法实现：每个子类的实现必须是由一个 `val` 定义，而不能是 `def` 或者 `var`。

  与此对比，抽象方法成员可以用具体的方法定义或者具体的 `val` 定义来实现。

```
abstract class Fruit{
  val v : String
  def m : String
}

class Apple extends Fruit{
  val v : String = "apple"
  val m : String = "example" // 用 val 重写 def 是 OK 的
}

class Orange extends Fruit{
  def v : String = "apple"  // 用 def 覆盖 val 是不允许的
  val m : String = "example"
}
```

4. 跟抽象的 `val` 成员类似，抽象的 `var` 成员也只是声明了名称和类型，但是并不给出初始值。

   - 声明为类成员的 `var` 都默认带上了 `getter` 方法和 `setter` 方法，这对于抽象的 `var` 成员也成立。

     ```
     trait AbstractTime {
       var hour: Int
       var minute: Int
     }
     ```

     这里等价于：

```
trait AbstractTime{
  def hour: Int         // hour 的 getter 方法
  def hour_=(x: Int)    // hour 的 setter 方法
  def minute: Int       // minute 的 getter 方法
  def minute_=(x: Int)  // minute 的 setter 方法
}
```

9.1 初始化抽象的 val

1. 抽象 `val` 成员有时候会承担超类的参数化功能：它允许我们在子类中提供那些在超类中缺失的细节。这对于特质尤其重要，因为特质并没有让我们传入参数的构造方法。

   因此，通常对于特质的参数化是通过在子类中实现抽象 `val` 成员来实现的。

   如：

   ```
   trait RationalTrait{  // 有理数
     val numerArg: Int   // 分子
     val denomArg: Int   // 分母
   }
   ```

   要实例化该特质的一个具体实例，需要实现抽象的 `val` 定义：

   ```
   new RationalTrait{
     val numerArg = 1
     val denomArg = 2
   }
   ```

   这个表达式交出的是一个混入了特质，并且由定义体 (由 `{}` 提供) 定义的匿名类 `anonymous class` 的实例。它和以下代码效果类似：

   ```
   class Rational(n: Int, d: Int){
     val numer: Int = n
     val denom: Int = d
   }
   new Rational(1,2)
   ```

   但是二者有细微区别：

   - `new Rational(expr1, expr2)` 中，`expr1` 和 `expr2` 这两个表达式会在 `Rational` 类初始化之前被求值，因此 `expr1` 和 `expr2` 的值在 `Rational` 类初始化过程中是可见的、已知的。
   - 对于特质来讲，情况不同。

     ```
     new RationalTrait{
       val numerArg = expr1
       val denomArg = expr2
     }
     ```

`expr1` 和 `expr2` 这两个表达式作为匿名类初始化过程的一部分被求值的，但是匿名类是在 `RationalTrait` 特质之后被初始化的。因此在 `RationalTrait` 初始化过程中，`numerArg, demoArg` 的值不可用。更准确的说，对于这两个值当中任何一个的选取都会交出类型 `Int` 的默认值 0 。

对于这里的例子，似乎并没有什么问题，因为特质的初始化过程并没有用到 `numerArg` 和 `denomArg` 。但是对于下面的示例，却很致命：

```
trait RationalTrait{  // 有理数
  val numerArg: Int   // 分子
  val denomArg: Int   // 分母
  require(denomArg != 0) // 前置条件
  private val g = gcd(numerArg, denomArg) // 最大公约数
  val numer = numerArg / g     // 归一化分子
  val denom = denomArg / g     // 归一化分母
  private def gcd(a:Int, b:Int) : Int= if(b == 0 ) a else gcd(b, a %
b) // 求最大公约数
  override def toString = numer + "/" + denom // 打印
}

val x = 2
new RationalTrait{
  val numerArg = 1 * x
  val denomArg = 2 * x
}
```

在初始化匿名类之前，编译器先初始化特质，此时 `1*x` 和 `2*x` 尚未求值，因此 `numerArg,denomArg` 为默认值 `0` ，因此特质的 `require` 前置条件不满足。

这个例子说明：类参数和抽象字段的初始化顺序并不相同。类参数在传入类构造方法之前被求值（传名参数除外），而在子类中实现的 `val` 抽象成员则在超类初始化之后被求值。

为解决后者的问题， `scala` 提供了两种方案：预初始化字段`pre-initialized field` 和 `lazy` 惰性的 `val` 。

2. 预初始化字段 `pre-initialized field` ：在超类被调用之前就初始化子类的字段。只需要在把字段定义放在超类的构造方法之前的花括号中即可。

```
val x = 2
new {
  val numerArg = 1 * x
  val denomArg = 2 * x
} with RationalTrait
```

初始化代码段出现在超类特质 `RationalTrait` 之前，用 `with` 隔开。

预初始化字段不仅局限于匿名类，也可以用于对象或者具名子类中。

```
object ABC extends{  // 用于对象中
  val numerArg = 2
  val denomArg = 3
} with RationalTrait

object MyClass(n: Int, d: Int) extends{ // 用于具名子类
  val numerArg = n
  val denomArg = d
}  with RationalTrait{
  def + (that: MyClass) = new MyClass(
    ....
  )
}
```

由于预初始化字段在超类的构造方法被调用之前初始化，因此它们的代码不能引用那个正在被构造的对象。因此，如果这样的初始化代码使用了 `this`，那么这个引用将指向包含当前被构造的类或对象的对象，而不是被构造的对象本身。

```
new {
  val numerArg = 2
  val denomArg = 3 * this.numerArg // 错误， this 指向的不是当前对象，因此没有 numerArg 属性
} with RationalTrait
```

这个例子无法编译，因为 `this.numerArg` 引用的是包含了 `new` 的对象，在解释器中对应于名为 `$iw` 的合成对象。

3. 预初始化字段这方面的行为类似于类构造方法的入参行为。我们可以通过预初始化字段来精确模拟类构造方法入参的初始化行为，但是有时候我们希望系统能自己搞定初始化顺序。

   可以将 `val` 定义为惰性的来实现。如果在 `val` 定义之前添加 `lazy` 修饰符，则右侧的初始化表达式只会在 `val` 第一次被用到时求值。

```
object Demo{
  lazy val x = {println("initialize x"); "done"}
}
```

这里 `Demo` 的初始化不涉及对 `x` 的初始化。对 `x` 的初始化延迟到第一次访问 `x` 的时候。

这和将 `x` 用 `def` 定义成无参方法的情况类似，区别在于：

- 不同于 `def`，惰性 `val` 永远不会被求值多次，只会被求值一次。

  事实上对惰性 `val` 首次求值之后其结果会被保存起来，在后续的使用中都会复用该值。

- `def` 每次使用时都会被求值。

因此一种修改方案为：

```
trait RationalTrait{  // 有理数
  val numerArg: Int   // 分子
  val denomArg: Int   // 分母
  lazy val numer = numerArg / g     // 归一化分子
  lazy val denom = denomArg / g     // 归一化分母
  private lazy val g = {
    require(denomArg != 0) // 前置条件
```

```
    gcd(numerArg, denomArg) // 最大公约数
  }
  private def gcd(a:Int, b:Int) : Int= if(b == 0 ) a else gcd(b, a % b) //
求最大公约数
  override def toString = numer + "/" + denom // 打印
}

val x = 2
val r = new RationalTrait{
  val numerArg = 1 * x
  val denomArg = 2 * x
}
println(r)
```

当 `RationalTrait` 匿名类的一个对象被创建时，`RationalTrait` 的初始化代码被执行。此时 `RationalTrait` 的初始化代码仅初始化 `numerArg` 和 `denomArg` 。

一旦后续调用 `println(r)`，则调用对象的 `toString` 方法。该方法用到 `denom` 这个 `lazy val` 于是 `denom` 被求值。而 `denom` 用到了 `lazy val g`，于是 `g` 被求值。

尽管 `g` 出现在 `numer, denom` 之后，但是它在 `numer, denom` 初始化之前初始化。因此对于惰性 `val`，其初始化顺序和定义顺序无关。这个优势仅在惰性的 `val` 的初始化既不产生副作用、也不依赖副作用时有效。

在有副作用参与时，初始化顺序就变得相当重要。这时候跟踪惰性 `val` 的初始化顺序非常困难。因此惰性 `val` 在函数式编程的使用场景比较合适，因为函数式对象的初始化顺序不重要。

9.2 抽象类型

1. 跟其它所有抽象声明一样，抽象类型声明是某种将会在子类中具体定义的东西的占位符：在类继承关系的下游中将被定义的类型，不同的子类可以提供不同的类型实现。
2. 下面的例子并不能编译通过：

```
class Food
abstract class Animal{
  def eat(food: Food)
}
class Grass extends Food
class Cow extends Animal{
  override def eat(food: Grass) ={} // 编译不通过
}
```

实际上 `Cow` 类的 `eat` 方法并没有重写 `Animal` 类的 `eat` 方法，因为它们的参数类型不同：一个是 `Food` 另一个是 `Grass`，虽然 `Grass` 是 `Food` 子类。

事实上如果上述代码能给通过编译，则容易写出下面的代码：

```
class Fish extends Food
val abc: Animal = new Cow
abc.eat(new Fish)
```

这明显是不合理的。

可以通过抽象类型来实现这段逻辑：

```
class Food
abstract class Animal{
  type T <: Food
  def eat(food: T)
}
```

这里 `Animal` 只能吃那些适合它吃的食物。至于什么食物是合适的，并不能在 `Animal` 类这个层级确定。这就是为什么 `T` 定义为一个抽象类型。这个抽象类型有个上界 `Food`，以 `<: Food` 表示。这意味着 `Animal` 每个子类对于 `T` 的实例化都必须是 `Food` 的子类。

```
class Grass extends Food
class Cow extends Animal{
  type T = Grass
  override def eat(food: Grass) = {}
}
```

9.3 路径依赖类型

1. 路径依赖类型的语法跟 `Java` 的内部类类型相似，不过有个重要区别：路径依赖类型用的是外部对象的名称，内部类用的是外部类的名称。在 `Scala` 中，内部类的寻址是通过 `Outer#Inner` 这样的表达式而不是 `Java` 的 `Outer.Inner`，`Scala` 的 `.` 语法只为对象保留。

```
class Outer{
  class Inner
}
val o1 = new Outer
val o2 = new Outer
```

`o1.Inner` 这样的类型称作路径依赖类型 `path-dependent type`。这里的路径指的是对对象的引用，它可以是一个简单的名称，比如 `o1`，也可以是更长的访问路径。

一般不同的路径（这里是不同的对象）催生出不同的类型，如 `o1.Inner` 和 `o2.Inner` 是两个路径依赖的类型（它们是不同的类型），这两个类型都是 `Outer#Inner` 类的子类型。

   - `o1.Inner` 指的是特定外部对象（即 `o1` 引用的那个对象）的 `Inner` 类。
   - `o2.Inner` 指的是特定外部对象（即 `o2` 引用的那个对象）的 `Inner` 类。
2. 跟 `Java` 一样，`Scala` 的内部类的实例会保存一个到外部类实例的引用。这允许内部类访问外部类的成员。因此我们在实例化内部类的时候必须以某种方式给出外部类实例。
   - 一种方式是在外部类的定义体中实例化内部类。此时通过 `this` 来访问外部类实例本身。
   - 另一种方式是采用路径依赖类型。如`o1.Inner` 这个类型是一个特定于外部对象，我们可以实例化它：

     ```
     new o1.Inner
     ```

     得到的内部对象将包含一个指向其外部对象（即 `o1`）的引用。

     于此对应，由于 `Outer#Inner` 并没有指明 `Outer` 的特定实例，因此不能创建它的实例。

9.4 改良类型

1. 当类从另一个类继承时，将前者称为后者的名义`nominal`子类。之所以称作`nomial`，是因为每个类型都有一个名称，而这些名称被显式声明为存在子类关系。

   除此之外`Scala`还支持结构`structural`子类，即：只要两个类型有兼容的成员就可以说它们之间存在子类关系。`Scala`实现结构子类的方式是改良类型`refinement type`。

2. 名义子类通常更方便使用，因此应该在任何新的设计中优先尝试名义子类。

   但是结构子类灵活性更高。比如，希望定义一个包含食草动物的"牧场"类：

   ```
   class Pasture {
     var animals: List[Animal {type SuitableFood = Grass}] = Nil
   }
   ```

   这里的改良类型只需要写基类型`Animal`，然后加上一系列用花括号括起来的成员即可。花括号中的成员进一步指定（或者说改良）了基类中的成员类型。

9.5 枚举

1. 别的语言，如`Java,C#`都有内建的语法结构来支持枚举类型，而`Scala`不需要特殊的语法来支持枚举。这是通过路径依赖类型来实现的。

   `Scala`在标准库中提供了一个类`scala.Enumeration`，可以通过定义一个扩展自该类的对象来创建新枚举。

   ```
   object Color extends Enumeration {
     val Red = Value
     val Green = Value
     val Blue = Value
   }
   ```

   `Scala`还允许我们用同一个右侧表达式来简化多个连续的`val`或者`var`定义，上述定义等价于：

   ```
   object Color extends Enumeration {
     val Red,Green,Blue = Value
   }
   ```

   这个对象定义了三个值：`Color.Red,Color.Green,Color.Blue`。

2. `Enumeration`定义了一个名为`Value`的内部类，跟这个内部类同名的、不带参数的`Value`方法每次都返回该类的全新实例。因此类似`Color.Red`的类型为`Color.Value`，而`Color.Value`是所有定义在`Color`对象中的`Value`的类型。

   这里面的关键点在于：这是一个完全的新类型，不同于其它所有类型。

   因此如果我们定义了另外一个枚举类型：

   ```
   object Direction extends Enumeration {
     val North, East, South, West = Value
   }
   ```

   那么`Direction.Value`将不同于`Color.Value`，因为这两个类型的路径部分是不同的。

3. `Scala`的`Enumeration`类型还提供了其它编程语言的枚举不支持的其它功能。

- 可以用一个重载的 `Value` 方法给枚举值关联特定的名称：

```
object Color extends Enumeration {
  val Red = Value("Red")
  val Green = Value("Green")
  val Blue = Value("Blue")
}
```

- 可以通过枚举的 `values` 方法返回的`Set` 来遍历枚举的 `Value`：

```
for (c <- Color.values) print(c + " ") // 打印结果： Red Green Blue
```

- 枚举的值从 `0` 开始编号，可以通过枚举`Value` 的 `id` 方法获取编号：

```
println(Color.Red.id) // 打印结果： 0
```

- 可以从一个非负整数编号获取对应的枚举值：

```
val c = Color(1) // c 为： Color.Green
```

十、模块化编程

1. `package` 和访问修饰符能够让你把包当做模块 `module` 来组织大型程序。这里的模块指的是具有良好定义的接口以及隐藏实现的 "小程序片段"。
2. 任何致力于模块化都需要满足一些最基本的要求：
   - 首先，应该有一个能够很好地分离接口和实现的模块结构
   - 其次，应该有方式可以替换具有相同接口的模块，而不需要改变或者重新编译依赖该模块的其它模块
   - 最后，应该有方式可以将模块连接在一起。这种连接操作可以认为是在配置该系统。
3. 解决模块化的一种思路是依赖注入`dependency injection`，这是一种通过框架来支持的、构建在 `Java` 平台上的技术。

   尽管也可以在 `Scala` 里使用 `Spring` ，从而以 `Spring` 的方式让`scala` 程序模块化，但是在 `scala` 中还有其它选择：将对象当做模块来使用，无须任何额外的框架从而达到模块化的目的。
4. `Scala` 用对象来表示模块，因此程序可以被切分成单例对象，每个单例对象表示一个模块。单例对象中的一些 `private` 元素是模块实现的一部分，这可以在不影响其他模块的情况下进行修改。
5. 混入特质时，可以通过 `self type` 来处理跨特质的引用。

```
abstract class Food(val name: String) {
  override def toString = name
}

object Orange extends Food("orange")

trait A{
  object Apple extends Food("apple")
  ...
}
```

```
trait B{
  val list = List(Apple,Orange) // Apple 不在作用域内
  ...
}
```

在特质 `B` 中引入 `Apple` 时出现问题，因为 `Apple` 作用在特质 `A` 中，因此超出了作用域。

此时可以在特质中引入 `self type`。从技术上讲，`self type` 是在类中提到 `this` 时，对 `this` 预期的类型。从实际应用来讲， `self type` 指定了对于特质能够混入的具体`class` 的要求。

例如：

```
trait B{
  this: A =>
  val list = List(Apple,Orange) // Apple 现在在作用域内
  ...
}
```

现在要求任何混入 `B` 的 `class` 也必须混入 `A` 。现在 `Apple` 被隐含地认为是 `this.Apple` 了。

注意：抽象子类和抽象特质不必遵循这个限制，因为它们不能被 `new` 实例化，所以并不存在 `this.Apple` 引用失败的风险。

6. 结尾为 `.type` 的类型表示单例类型。单例类型极为明确：这个类型只保存一个对象。通常这样的类型太过于确定，以至于没什么用处。

   通常它用于链式调用：

```
class A{
  def f1: A = this
}
class B extends A{
  def f2: B = this
}
val b = new B
b.f2.f1 // 可以调用
b.f1.f2 // 无法调用
```

其中 `b.f1.f2` 无法调用，因为 `b.f1` 返回的类型为 `A`，而 `A` 是不存在 `f2` 方法的。

解决的办法就是单例类型：

```
class A{
  def f1: this.type = this
}
class B extends A{
  def f2: this.type = this
}
val b = new B
b.f2.f1 // 可以调用
b.f1.f2 // 可以调用
```

现在 `b.f1` 返回的类型就是当前实例的单例类型，这个单例类型只要唯一的实例对象 `b` 。

十一、对象相等性

1. 在 `scala` 中，相等性的定义和 `java` 不同。`java` 有两种相等性比较：
   - `==` 操作符：对于值类型而言，这是很自然的相等性。对于引用类型而言，这表示对象的一致性，即两个名字是否指向同一个对象。
   - `equals` 方法：是用户自定义的、用于比较引用类型的相等性（而不是一致性）。

   这种做法是有问题的，因为 `==` 符号更为自然，但是并不总是对应到自然语义上的相等性。比如在 `java` 中， 对于字符串 `x` 和 `y`， 即使它们具有相同的有序字符，最终得到 `false` 也不奇怪。

   `scala` 也有一个相等性判断方法用于表示对象的一致性，但是用得并不多：`x eq y` ，当 `x` 和 `y` 引用同一个对象时为 `true` 。

   在 `scala` 中，`==` 相等性判断被用来表示每个类型"自然地"相等性。
   - 对于值类型而言， `scala` 中 `==` 是对值的比较，这和 `java` 一样。
   - 对于引用类型而言，`scala` 汇总 `==` 相当于 `java` 中的 `equals` 。

   对于引用类型而言，也可以重写新类型的 `equals` 方法从而重新定义 `==` 行为。

   `equals` 方法总是会从 `Any` 类继承下来。继承的 `equals` 方法除非被重写，否则默认是像 `Java` 那样判断对象是否一致。因此 `equals` 方法（以及 `==`）默认和 `eq` 是一样的，不过可以通过在定义的类中重写 `equals` 方法的方式改变其行为。

   我们没有办法直接重写 `==` ，因为它在 `Any` 类中定义为 `final` 方法。也就是在 `Anly` 中，`final` 的定义类似于：

```
final def == (that: Any): Boolean =
    if (null eq this) {null eq that} else { this euqals that}
```

   因此在 `scala` 中，对于新的类型我们需要重写 `equals` 方法。事实上 `2007` 年的一篇论文指出：几乎所有的 `equals` 方法的实现都有问题。这个问题很严重，因为很多其他代码逻辑严重依赖于相等性判断。

   重写 `equals` 方法有四个常见陷阱：
   - 定义 `equals` 方法时采用了错误的方法签名。
   - 修改了 `equals` 方法但是没有同时修改 `hashCode` 。
   - 使用可变字段定义 `equals` 方法。
   - 未能按照等同性关系定义 `equals` 方法。

11.1 错误的方法签名

1. 考虑如下的一个 `equals` 实现方法：

```
class Point(val x: Int, val y: Int) {
  def equals(other: Point): Boolean =
      this.x == other.x && this.y = other.y
}
```

   简单来看，这种相等性实现是 `OK` 的：

```
val p1, p2 = new Point(1,2)
val p3 = new Point(2,3)
p1 equals p2 // true
p1 equals p3 // false
```

但是，一单将`p1,p2,p3`放到集合中来时，问题出现了：

```
import scala.collection.mutable
val col1 = mutable.HashSet(p1)
col1 contains p2 //false
```

这里`p1`添加到集合，而`p1`和`p2`是两个相等对象，但是集合不包含`p2`。

进一步的，我们考虑以下比较：

```
val p4:Any = p2
p4 equals p1 //false
```

我们发现：`p1`等于`p2`，`p2`就是`p4`，但是`p4`不等于`p2`。

问题在于：前面的`equals`方法并未重写标准的`equals`方法，因为方法签名不同。在`Any`类中的`equals`方法的类型为：

```
def equals(other: Anly) :Boolean
```

而`Point`类中我们定义的`equals`方法类型为：

```
def equals(other: Point) :Boolean
```

因此这里并未重写`Any`类的`equals`方法，而是实现了重载的一个备选方法。

目前`scala`和`java`中的重载都是根据参数的静态类型，而不是运行时的动态类型来解析的。因此，只要参数的静态类型是`Point`，则调用的就是`Point`类中的`equals`方法。但是，一旦静态的参数类型是`Any`类型，则调用的就是`Any`类的`equals`方法。

因此，`Point`类并未重写`equals`方法，它仍然是通过比较对象是否一致来实现相等性比较。这就是为什么`p1`和`p4`的内容一致，但是`p1 equals p4`仍然返回`false`的原因。

这也是为什么`col1.contains(p2)`返回`false`的原因。由于它操作的是泛型集合，因此它调用的是`Object`类的`equals`方法，而不是`Point`中重载的变种。

2. 一个稍微好一点（但是仍有问题）的`equals`方法为：

```
override def equals(other: Any) = other match{
  case that: Point => this.x == that.x && this.y == that.y
  case _ => false
}
```

现在`equals`方法有了正确的签名，它以一个类型为`Any`的值作为参数，返回`Boolean`类型的结果。

3. 一个常见的陷阱是：使用错误的签名来定义`==`。

- 如果你希望重写 `==`，而且签名正确（即接收一个类型为 `Any` 的参数），则编译器会报错。因为你这是在重写 `Any` 中的 `final` 方法。
- 如果你希望重写 `==`，但是签名错误，则编译器会通过。因为这是实现了重载的一个备选方法。

```
def == (ohter: Point) :Boolean = // 虽然可以通过编译，但这不是重写，而是重载
```

这里用户定义的 `==` 方法被当做 `Any` 类中同名方法重载的变种，因此程序通过了编译。

11.2 未修改 hashCode

1. 使用了正确的方法签名之后，`equals` 能够正常工作。但是现在 `col1.contains(p2)` 还是返回 `false`：

```
class Point(val x: Int, val y: Int) {
  override def equals(other: Any) = other match{
      case that: Point => this.x == that.x && this.y == that.y
      case _ => false
  }
}
val p1, p2 = new Point(1,2)
val p3 = new Point(2,3)
val p4:Any = p2
p1 equals p2 // true
p1 equals p3 // false
p4 equals p1 // true : 现在结果是对的

import scala.collection.mutable
val col1 = mutable.HashSet(p1)
col1 contains p2 //false : 还是有问题
```

这里的问题是 `Point` 类重写了 `equals` 方法，但是没有重写 `hashCode` 方法。

这里我们使用了 `HashSet`，意味着集合类中的元素会依据它们的哈希码放到哈希桶里。`contains` 检测首先决定要找的桶，然后再将给定的元素和桶内的元素一一比较。

现在的情况是：`Point` 类重新定义了 `equals`，但是没有重新定义 `hashCode`。因此 `hashCode` 仍然是 `AnyRef` 类中的版本：对已分配对象地址的某种转换。

因此，`p1` 和 `p2` 的哈希码几乎肯定不同，尽管它们的内容完全相同。不同的哈希码意味着它们对应于集合中不同的哈希桶。因此 `Point` 违背了 `Any` 类中定义的 `hashCode` 方法的约定：如果两个对象根据 `equals` 方法是相等的，那么对它们调用 `hashCode` 方法都必须产出相同的整型结果。

事实上，`Java` 中的 `hashCode` 和 `equals` 应该总是一起定义，这是普遍的共识。除此之外，`hashCode` 只能依赖于 `equals` 方法依赖的字段。因此，可以对 `Point` 类重写 `hashCode` 方法：

```
class Point(val x: Int, val y: Int) {
  override def equals(other: Any) = other match{
      case that: Point => this.x == that.x && this.y == that.y
      case _ => false
  }
  override def hashCode = (x, y).##
}
```

注意：这里的 `##` 方法是计算基本类型、引用类型和 `null` 的哈希码的简写。当我们对集合或元组调用这个方法时，它会计算一个混合的哈希码，这个哈希码跟集合中所有元素的哈希码都相关。

11.3 可变字段

1. 考虑如下的实现：

```
class Point(var x: Int, var y: Int) {
  override def equals(other: Any) = other match{
      case that: Point => this.x == that.x && this.y == that.y
      case _ => false
  }
  override def hashCode = (x, y).##
}
```

唯一的变动是将 `val x` 和 `val y` 替换为 `var x` 和 `var y` 。由于 `equals` 和 `hashCode` 是基于这两个字段来实现的， 因此如果修改了 `x` 和 `y`，则 `equals/hashCode` 结果也会发生改变。

```
val p = new Point(1,2)
val col1 = collection.mutable.HashSet(p)
col1 contains p // true
p.x += 1 // 修改了 x
col1 contains p // false
col1.iterator contains p // true
```

现在发生了奇怪的现象：包含 `p` 的集合 `contains(p)` 结果为 `false` 。

原因是当对 `p` 进行修改之后（`p.x += 1`），`p` 的 `hashCode` 发生改变，使得 `p` 被放到 `col` 集合中错误的哈希桶中。从某种意义上讲，`p` 已经 “掉出了” `col1` 集合的 “视野”。

2. 如果 `equals` 和 `hashCode` 依赖于可变的状态，则会隐含着问题。如果需要将这样的对象放入集合中，则需要非常小心地避免修改被依赖的状态。

11.4 等同性关系

1. 根据 `scala.Any` 的 `equals` 方法的约定， `equals` 方法必须对非 `null` 对象实现等同性关系：
   - 反身性：对任何非 `null` 的对象 `x`，表达式 `x.equals(x)` 始终返回 `true`
   - 对称性：对任何非 `null` 的对象 `x, y` ，表达式 `x.equals(y)` 结果始终与 `y.equals(x)` 相同
   - 传递性：对于任何非 `null` 的对象 `x, y, z`，如果 `x.equals(y)` 返回 `true` 且 `y.equals(z)` 返回 `true`，则 `x.equals(z)` 也应该返回 `true`
   - 一致性：对于任何非 `null` 的对象 `x,y`，只要用于对象 `equals` 比较的信息未被修改过，则多次调用 `x.equals(y)` 的结果都是相同的，要么都是返回 `true`、要么都是返回 `false`。

- 非空性：对于任何非`null`的对象`x`，`x.equals(null)`始终返回`false`。

2. 前面定义的`Point`类的`equals`方法目前满足`equals`的约定。但是当我们考虑子类的时候，情况变得更为复杂：

```
class Point(val x: Int, val y: Int) {
  override def equals(other: Any) = other match{
      case that: Point => this.x == that.x && this.y == that.y
      case _ => false
  }
  override def hashCode = (x, y).##
}

object Color extends Enumeration{
  val Red, Orange, Yellow, Green, Blue, Indigo, Violet = Value
}

class ColoredPoint(x: Int, y: Int, val color: Color.Value) extends
Point(x, y){
  override def equals(other: Anly) = other match{
    case that: ColoredPoint => this.color = that.color &&
super.equals(that)
    case _ => false
  }
}
```

这里没有重写`hashCode`方法。因为根据约定：如果两个对象根据`equals`方法是相等的，那么对它们调用`hashCode`方法都必须产出相同的整型结果。如果两个`ColoredPoint`是相等的，则根据这里的`equals`定义则它们具有相同的`x, y, color`，则根据`Point`中定义的`hashCode`方法，它们的哈希码也是相同的。

考虑以下的比较：

```
val p = new Point(1,2)
val cp = new ColoredPoint(1,2,Color.Red)
p equals cp // true
cp equals p // false
```

由于第二个比较中，`p`不是`ColoredPoint`，因此比较结果返回`false`。可以看到这两个比较违反了对称性。

对集合而言，失去对称性会带来无法预期的后果，如：

```
collection.mutable.HashSet[Point](p) contains cp // true
collection.mutable.HashSet[Point](cp) contains p // false
```

要修复这种对称性问题有两种方式：

- 要么让`equals`比较对象之间的关系更为泛化。如：

```
class ColoredPoint(x: Int, y: Int, val color: COlor.Value) extends
Point(x, y){
  override def equals(other: Anly) = other match{
    case that: ColoredPoint => this.color = that.color &&
super.equals(that)
    case that: Point => that equals this
    case _ => false
  }
```

现在如果 `other` 对象是 `Point` 但不是 `ColoredPoint`，则判断逻辑会转到 `Point` 类的 `equals` 方法。这使得 `equals` 方法是对称的。

现在无论是 `cp equals p` 还是 `p equals cp` ，结果都是 `true`。但是现在的问题是：新的 `equals` 方法不满足传递性：

```
val cp1 = new ColoredPoint(1,2,Color.Red)
val cp2 = new ColoredPoint(1,2,Color.Blue)
val p = new Point(1,2)
cp1 equals p // true
p equals cp2 // true
cp1 equals cp2 // false
```

现在看来将 `equals` 的关系泛化行不通。

- 要么让 `equals` 比较对象之间的关系更为严格，总是将不同类型的对象当做是不同的。如：

```
class Point(val x: Int, val y: Int) {
  override def equals(other: Any) = other match{
      case that: Point => this.x == that.x && this.y == that.y &&
this.getClass == that.getClass
      case _ => false
  }
  override def hashCode = (x, y).##
}

object Color extends Enumeration{
  val Red, Orange, Yellow, Green, Blue, Indigo, Violet = Value
}

class ColoredPoint(x: Int, y: Int, val color: COlor.Value) extends
Point(x, y){
  override def equals(other: Any) = other match{
    case that: ColoredPoint => this.color = that.color &&
super.equals(that)
    case _ => false
  }
}
```

这里在 `Point` 的 `equals` 方法中增加： `this.getClass == that.getClass` 。

新的定义既满足对称性又满足传递性，因为现在不同类型的对象之间比较判断总是返回 `false`。因此一个 `ColoredPoint` 永远不会和一个 `Point` 相等。但是这种方式太过于严格。考虑如下的代码：

```
val pp = new Point(1,1){override val y = 2}
p equals pp //false
```

这里 `pp` 是 `Point` 的匿名子类，与 `p` 的 `Point` 类不是相同类型，因此即使它们的内容相同，比较结果也是 `false`。

11.5 CanEqual 方法

1. 可以在 `equals` 和 `hashCode` 方法之外再新增一个自定义的方法，如 `canEqual`。

```
class Point(val x: Int, val y: Int) {
  override def equals(other: Any) = other match{
      case that: Point =>
          (that canEqual this) &&
          (this.x == that.x) &&
          (this.y == that.y)
      case _ => false
  }
  override def hashCode = (x, y).##
  def canEqual(other: Any) = other.isInstanceOf[Point]
}
```

   - 如果 `other` 对象是 `Point` 的子类，且未重新定义 `canEqual`，则 `that canEqual this` 返回 `true`。
   - 如果 `other` 对象是 `Point` 的子类，且重新定义了 `canEqual`，则由这个子类的 `canEqual` 决定。
   - 如果 `other` 对象不是 `Point` 及其子类，则 `that canEqual this` 返回 `false`

2. 进一步地，我们考察 `ColoredPoint` 的实现：

```
object Color extends Enumeration{
  val Red, Orange, Yellow, Green, Blue, Indigo, Violet = Value
}

class ColoredPoint(x: Int, y: Int, val color: COlor.Value) extends
Point(x, y){
  override def hashCode = (super.hashCode, color).##
  override def equals(other: Any) = other match{
    case that: ColoredPoint =>
        (that canEqual this) && super.equals(that) && this.color =
that.color
    case _ => false
  }
  override def canEqual(color:Any) = other.isInstanceOf[ColoredPoint]
}
```

现在：

```
val p = new Point(1,2)
val cp = new ColoredPoint(1,2,Color.Red)
cp equals p // false，因为 p 不是 ColoredPoint 的实例，在 other match 处返回 false
p equals cp // false，因为 cp.canEqual(p) 返回 false
```

3. `Point` 的不同子类的对象可以相等，只要这些子类没有重新定义 `canEqual` 方法：

```
val p = new Point(1,2)
val cp = new ColoredPoint(1,2, Color.Red)
val pp = new Point(1,1){override val y = 2}
p equals pp // true，因为 pp.canEqual(p) 是成立的
pp equals p // true，因为 p.canEqual(pp) 是成立的
```

11.6 参数化类型

1. 考虑一个一个二叉树：

```
trait Tree[+T]{
  def elem: T
  def left: Tree[T]
  def right: Tree[T]
}

object EmptyTree extends Tree[Nothing]{
  def elem = throw new NoSunchElementException("EmptyTree.elem")
  def left = throw new NoSunchElementException("EmptyTree.left")
  def right = throw new NoSunchElementException("EmptyTree.right")
}

class Branch[+T](
  val elem:T,
  val left: Tree[T],
  val right: Tree[T]
)extends Tree[T]{
  override def equals(other: Any) = other match{
    case that: Branch[T] =>
        this.elem == that.elem &&
        this.left == that.left &&
        this.right == that.right
    case _ => false
  }
}
```

对于 `EmptyTree` 而言，不需要重写 `equals` 方法，因为 `EmptyTree` 默认从 `AnyRef` 继承的 `equals` 和 `hashCode` 就满足要求。毕竟 `EmptyTree` 只和自己相等，因此它的相等性就应该是引用相等，这也是从 `AnyRef` 继承下来的行为。

但是对于 `Branch` 而言，在重写 `equals` 过程中发现模式匹配有问题：编译器只能检测到 `other` 是某种 `Branch`，无法检测到具体的元素类型 `T` 。这是因为参数化类型的元素类型在编译器的擦除阶段被抹掉，这些信息在运行期无法被检查。

为解决该问题，可以在模式匹配中进行小小的修改：

```
override def equals(other: Any) = other match{
    case that: Branch[t] =>
        this.elem == that.elem &&
        this.left == that.left &&
        this.right == that.right
    case _ => false
  }
```

这里将`T`修改为`t`，即`t`表示未知类型。因此`case that: Branch[t]`会匹配任何`Branch`类型。

也可以将`t`替换为下划线，二者是等价的：

```
override def equals(other: Any) = other match{
    case that: Branch[_] =>
        this.elem == that.elem &&
        this.left == that.left &&
        this.right == that.right
    case _ => false
  }
```

2. 也可以在这里应用`canEqual`：

```
class Branch[+T](
  val elem:T,
  val left: Tree[T],
  val right: Tree[T]
)extends Tree[T]{
  override def equals(other: Any) = other match{
    case that: Branch[_] =>
        this.elem == that.elem &&
        this.left == that.left &&
        this.right == that.right
    case _ => false
  }
  def canEqual(other: Any) = other.isInstanceOf[Branch[_]]
}
```

注意，这里`other.isInstanceOf[Branch[_]]`中出现了`Branch[_]`，它并不是模式匹配，而是所谓的存在类型的简写。简单的说，这是一个通配类型。尽管技术上讲，下划线在模式匹配和方法调用的类型参数中代表两种不同的东西，但是它们本质含义是相同的：它让你将某些东西标记为未知的。

3. 最终的`Branch`版本为：

```
class Branch[+T](
  val elem:T,
  val left: Tree[T],
  val right: Tree[T]
)extends Tree[T]{
  override def equals(other: Any) = other match{
    case that: Branch[_] =>
        (that canEqual this) &&
```

```
        this.elem == that.elem &&
        this.left == that.left &&
        this.right == that.right
    case _ => false
  }
  def canEqual(other: Any) = other.isInstanceOf[Branch[_]]
  override def hashCode: Int = (elem, left, right).##
}
```

11.7 如何编写 equals 和 hashCode

1. 这里提供创建 `equals` 和 `hashCode` 方法的指导，这对于绝大多数情形而言是足够了的。
2. 编写 `equals` 的建议：
   - 如果要在非 `final` 类中重写 `equals` 方法，则应该创建 `canEqual` 方法。

     如果 `equals` 没有在类继承关系的父类被重新定义，则 `canEqual` 的定义将会是新的；否则这将重写父类中的同名方法的定义。

     如果要在 `final` 类中重写了继承自 `AnyRef` 的 `equals` 方法，则它们并不需要要定义 `canEqual`。

     注意：传递给 `canEqual` 的对象类型应该是 `Any`：

     ```
     def canEqual(other: Any): Boolean =
     ```

   - 如果参数对象是当前类的实例，则 `canEqual` 方法应该返回 `True`（即 `canEqual` 定义所在的类）；否则应该返回 `false`。

     ```
     other.isInstanceOf[XXX]
     ```

   - 在 `equals` 方法中，记得传入参数的类型为 `Any`：

     ```
     override def equals(other: Any): Boolean =
     ```

   - 将 `equals` 方法体写为单个 `match` 表达式，而 `match` 的选择器应该为传递给 `equals` 的参数。`match` 的表达式应该有两个 `case`：
     - 第一个应该声明为你定义 `equals` 方法的类型的类型模式。在这个 `case` 中编写一个表达式，把两个对象要相等的条件通过逻辑 `&&` 的关系组合起来。
       - 如果你需要调用父类的 `equals` 方法，则可以使用 `super.equals(that)`。
       - 如果是继承体系中第一个引入 `canEqual` 的 类定义的 `equals` 方法，则应该调用其 `canEqual` 方法，并将 `this` 传递进去：`that canEqual this`
       - 如果是重写的 `equals` 方法也应该包含 `canEqual` 的调用，除非它们包含了对 `super.equals` 的调用。在后者情况中，`canEqual` 会在超类中被调用。
       - 对于每个和相等性有关的字段，验证本对象的字段和传入参数对象的字段是相等的。
     - 对于第二个 `case`，用一个通配模式返回 `false`。

```
class XXX(val n:Int, val m:Int)={
  override def equals(other: Any): Boolean = {
  other match{
    case that : XXX =>
      (that canEqual this) &&
      (n == that.n) && (m.that.m)
    case _ => false
  }
}
```

3. 编写 `hashCode` 的建议：
   - 将对象中用在 `equals` 方法里计算相等性的每个相关字段都包含进来，创建一个包含这些字段的元组，然后对这个元组调用 `##` 方法。
   - 如果 `equals` 方法的实现过程中调用了 `super.equals(that)` ，则你应该在这个元组中包含 `super.hashCode` 逻辑。
   - 注意，有些字段的类型为集合类型，因此：
     - 对于 `Vector/List/Set/Map/元组` ，对这些对象调用 `hashCode` 会包含它们的元素的 `hash` 结果。因为这些类的 `equals/hashCode` 方法被重写过。
     - 对于 `Array` 类型，它的哈希码在计算时并未考虑包含的元素，所以你需要主动考虑每个元素的 `hashCode` 。
   - 如果你发现某个特定的 `hashCode` 计算影响到程序的性能，也可以考虑将 `hashCode` 缓存起来。如果对象是不可变的，可以在创建对象的时候计算 `hashCode` 并保存到一个字段中。可以简单地通过 `val` 而不是 `def` 重写 `hashCode` ：

     ```
     override val hashCode: Int = (n, m).##
     ```

4. 也可以将类定义为 `case class` ，这样编译器会自动地添加正确的、符合各项要求的 `equals` 方法和 `hashCode` 方法。