TOUR OF SCALA

VARIANCES

Variance is the correlation of subtyping relationships of complex types and the subtyping relationships of their component types. Scala supports variance annotations of type parameters of generic classes, to allow them to be covariant, contravariant, or invariant if no annotations are used. The use of variance in the type system allows us to make intuitive connections between complex types, whereas the lack of variance can restrict the reuse of a class abstraction.

```
class Foo[+A] // A covariant class
class Bar[-A] // A contravariant class
class Baz[A]  // An invariant class
```

Covariance

A type parameter `T` of a generic class can be made covariant by using the annotation `+T`. For some `class List[+T]`, making `T` covariant implies that for two types `A` and `B` where `B` is a subtype of `A`, then `List[B]` is a subtype of `List[A]`. This allows us to make very useful and intuitive subtyping relationships using generics.

Consider this simple class structure:

```
abstract class Animal {
  def name: String
}
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal
```

Both `Cat` and `Dog` are subtypes of `Animal`. The Scala standard library has a generic immutable `sealed abstract class List[+A]` class, where the type parameter `A` is covariant. This means that a `List[Cat]` is a `List[Animal]`. A `List[Dog]` is also a `List[Animal]`. Intuitively, it makes sense that a list of cats and a list of dogs are each lists of animals, and you should be able to use either of them for in place of `List[Animal]`.

In the following example, the method `printAnimalNames` will accept a list of animals as an argument and print their names each on a new line. If `List[A]` were not covariant, the last two method calls would not compile, which would severely limit the usefulness of the `printAnimalNames` method.

```
def printAnimalNames(animals: List[Animal]): Unit =
  animals.foreach {
    animal => println(animal.name)
  }

val cats: List[Cat] = List(Cat("Whiskers"), Cat("Tom"))
val dogs: List[Dog] = List(Dog("Fido"), Dog("Rex"))

// prints: Whiskers, Tom
printAnimalNames(cats)

// prints: Fido, Rex
printAnimalNames(dogs)
```

Contravariance

A type parameter `A` of a generic class can be made contravariant by using the annotation `-A`. This creates a subtyping

A type parameter `A` of a generic class can be made contravariant by using the annotation `-A`. This creates a subtyping relationship between the class and its type parameter that is similar, but opposite to what we get with covariance. That is, for some `class Printer[-A]`, making `A` contravariant implies that for two types `A` and `B` where `A` is a subtype of `B`, `Printer[B]` is a subtype of `Printer[A]`.

Consider the `Cat`, `Dog`, and `Animal` classes defined above for the following example:

```
abstract class Printer[-A] {
  def print(value: A): Unit
}
```

A `Printer[A]` is a simple class that knows how to print out some type `A`. Let's define some subclasses for specific types:

```
class AnimalPrinter extends Printer[Animal] {
  def print(animal: Animal): Unit =
    println("The animal's name is: " + animal.name)
}

class CatPrinter extends Printer[Cat] {
  def print(cat: Cat): Unit =
    println("The cat's name is: " + cat.name)
}
```

If a `Printer[Cat]` knows how to print any `Cat` to the console, and a `Printer[Animal]` knows how to print any `Animal` to the console, it makes sense that a `Printer[Animal]` would also know how to print any `Cat`. The inverse relationship does not apply, because a `Printer[Cat]` does not know how to print any `Animal` to the console. Therefore, we should be able to use a `Printer[Animal]` in place of `Printer[Cat]`, if we wish, and making `Printer[A]` contravariant allows us to do exactly that.

```
def printMyCat(printer: Printer[Cat], cat: Cat): Unit =
  printer.print(cat)

val catPrinter: Printer[Cat] = new CatPrinter
val animalPrinter: Printer[Animal] = new AnimalPrinter

printMyCat(catPrinter, Cat("Boots"))
printMyCat(animalPrinter, Cat("Boots"))
```

The output of this program will be:

```
The cat's name is: Boots
The animal's name is: Boots
```

Invariance

Generic classes in Scala are invariant by default. This means that they are neither covariant nor contravariant. In the context of the following example, `Container` class is invariant. A `Container[Cat]` is *not* a `Container[Animal]`, nor is the reverse true.

```
class Container[A](value: A) {
  private var _value: A = value
  def getValue: A = _value
  def setValue(value: A): Unit = {
    _value = value
  }
}
```

It may seem like a `Container[Cat]` should naturally also be a `Container[Animal]`, but allowing a mutable generic class to be covariant would not be safe. In this example, it is very important that `Container` is invariant. Supposing `Container` was actually covariant, something like this could happen:

```
val catContainer: Container[Cat] = new Container(Cat("Felix"))
val animalContainer: Container[Animal] = catContainer
```

```scala
val animalContainer: Container[Animal] = catContainer
animalContainer.setValue(Dog("Spot"))
val cat: Cat = catContainer.getValue // Oops, we'd end up with a Dog assigned to a Cat
```

Fortunately, the compiler stops us long before we could get this far.

Other Examples

Another example that can help one understand variance is `trait Function1[-T, +R]` from the Scala standard library. `Function1` represents a function with one parameter, where the first type parameter `T` represents the parameter type, and the second type parameter `R` represents the return type. A `Function1` is contravariant over its parameter type, and covariant over its return type. For this example we'll use the literal notation `A => B` to represent a `Function1[A, B]`.

Assume the similar `Cat`, `Dog`, `Animal` inheritance tree used earlier, plus the following:

```scala
abstract class SmallAnimal extends Animal
case class Mouse(name: String) extends SmallAnimal
```

Suppose we're working with functions that accept types of animals, and return the types of food they eat. If we would like a `Cat => SmallAnimal` (because cats eat small animals), but are given a `Animal => Mouse` instead, our program will still work. Intuitively an `Animal => Mouse` will still accept a `Cat` as an argument, because a `Cat` is an `Animal`, and it returns a `Mouse`, which is also a `SmallAnimal`. Since we can safely and invisibly substitute the former with the latter, we can say `Animal => Mouse` is a subtype of `Cat => SmallAnimal`.

Comparison With Other Languages

Variance is supported in different ways by some languages that are similar to Scala. For example, variance annotations in Scala closely resemble those in C#, where the annotations are added when a class abstraction is defined (declaration-site variance). In Java, however, variance annotations are given by clients when a class abstraction is used (use-site variance).

← **previous** **next** →

Contributors to this page:

martinjaime mikkelmilo ckipp01 raindev manishbansal8843 SethTisue
andreamarconi mlachkar emonmishra ashawley fabiopakk IDispose
Okina1Raion jbalintbiro heathermiller

DOCUMENTATION
Getting Started
API
Overviews/Guides
Language Specification

DOWNLOAD
Current Version
All versions

COMMUNITY
Community
Mailing Lists
Chat Rooms & More
Libraries and Tools
The Scala Center

CONTRIBUTE
How to help
Report an Issue

SCALA
Blog
Code of Conduct
License

SOCIAL
GitHub
Twitter