



# Подношение от типов: отношение подтипов

```
//> using dep dev.zio::zio:2.1.13
import zio.*

trait ServiceS:
  def getString: UIO[String]

trait ServiceB:
  def getBool: UIO[Boolean]
```

```
//> using dep dev.zio::zio:2.1.13
import zio.*

trait ServiceS:
  def getString: UIO[String]

trait ServiceB:
  def getBool: UIO[Boolean]

val program: RIO[ServiceS & ServiceB, Unit] =
  for
    svcS ← ZIO.service[ServiceS]
    str  ← svcS.getString
    _    ← Console.println(s"I got $str!")

    svcB ← ZIO.service[ServiceB]
    bool ← svcB.getBool
    _    ← Console.println(s"I got $bool!")
  yield ()
```

```
//> using dep dev.zio::zio:2.1.13
import zio.*

val program: RIO[String & Boolean, Unit] =
  for
    str  ← ZIO.service[String]
    _    ← Console.println(s"I got $str!")

    bool ← ZIO.service[Boolean]
    _    ← Console.println(s"I got $bool!")
  yield ()
```

```
//> using dep dev.zio::zio:2.1.13
import zio.*

val program: RIO[String & Boolean, Unit] =
  for
    str  ← ZIO.service[String]
    _    ← Console.println(s"I got $str!")

    bool ← ZIO.service[Boolean]
    _    ← Console.println(s"I got $bool!")
  yield ()

object App extends ZIOAppDefault:
  val run = program.provide(ZLayer.succeed("it"), ZLayer.succeed(true))
```

```
//> using dep dev.zio::zio:2.1.13
import zio.*

val program: RIO[String & Boolean, Unit] =
  for
    str  ← ZIO.service[String]
    _    ← Console.println(s"I got $str!")

    bool ← ZIO.service[Boolean]
    _    ← Console.println(s"I got $bool!")
  yield ()

object App extends ZIOAppDefault:
  val run = program.provide(ZLayer.succeed("it"), ZLayer.succeed(true))
```

```
I got it!
I got true!
```

`String & Boolean  $\cong$  Nothing`

```
//> using dep dev.zio::zio:2.1.13
import zio.*

val program: RIO[Nothing, Unit] =
  for
    str  ← ZIO.service[String]
    _    ← Console.println(s"I got $str!")
    bool ← ZIO.service[Boolean]
    _    ← Console.println(s"I got $bool!")

yield ()
```



```
//> using dep dev.zio::zio:2.1.13
import zio.*

val program: RIO[Nothing, Unit] =
  for
    str  ← ZIO.service[String]
    _    ← Console.println(s"I got $str!")
    bool ← ZIO.service[Boolean]
    _    ← Console.println(s"I got $bool!")

  yield ()

object App extends ZIOAppDefault:
  val run = someMapping(program).provide( // no legal someMapping
    ZLayer.succeed("it") and ZLayer.succeed(true)
  )
```

```
//> using dep dev.zio::zio:2.1.13
import zio.*

val program: RIO[Nothing, Unit] =
  for
    str ← ZIO.service[String]
    _   ← Console.println(s"I got $str!")
    bool ← ZIO.service[Boolean]
    _   ← Console.println(s"I got $bool!")
    cat  ← ZIO.service[Cat]
    _   ← Console.println(s"I got $cat!")
  yield ()

object App extends ZIOAppDefault:
  val run = someMapping(program).provide( // no legal someMapping
    ZLayer.succeed("it") and ZLayer.succeed(true)
  )
```

В ZIO контекст является необычным типом:

- нас не интересует есть ли **значения**, которые его населяют

В ZIO контекст является необычным типом:

- нас не интересует есть ли **значения**, которые его населяют
- нам важны только **структура и свойства** этого типа

В ZIO контекст является не обычным типом, а **фантомным**:

- нас не интересует есть ли **значения**, которые его населяют
- нам важны только **структура и свойства** этого типа

В ZIO контекст является не обычным типом, а **фантомным**:

- нас не интересует есть ли **значения**, которые его населяют
- нам важны только **структура и свойства** этого типа

Типы пересечения выбраны не случайно, у них есть прекрасные свойства!

# План

- Поговорим про свойства подтипирования и типов пересечений в Scala
- Посмотрим как под капотом реализованы ZIO-контексты
- Как еще применяется:
  - в нашей реализации общих `http`-клиентов
  - в других `opensource` библиотеках

Scala = OOP & FP



# Наследование

```
trait Animal:  
  def sound: String  
  
class Dog extends Animal:  
  override val sound = "woof"  
  
class Cat extends Animal:  
  override val sound = "meow"
```

# Наследование

```
trait Animal:  
  def sound: String  
  
class Dog extends Animal:  
  override val sound = "woof"  
  
class Cat extends Animal:  
  override val sound = "meow"  
  
def allSay(animals: Animal*): Unit =  
  animals.foreach(x => println(x.sound))  
  
allSay(new Dog, new Cat)
```

# Наследование

```
trait Animal:  
  def sound: String  
  
class Dog extends Animal:      // Dog <: Animal  
  override val sound = "woof"  
  
class Cat extends Animal:      // Cat <: Animal  
  override val sound = "meow"  
  
def allSay(animals: Animal*): Unit =  
  animals.foreach(x => println(x.sound))  
  
allSay(new Dog: Dog, new Cat: Cat)
```

# Наследование

```
trait Animal:  
  def sound: String  
  
class Dog extends Animal:      // Dog <: Animal  
  override val sound = "woof"  
  
class Cat extends Animal:     // Cat <: Animal  
  override val sound = "meow"  
  
def allSay(animals: Animal*): Unit =  
  animals.foreach(x => println(x.sound))  
  
allSay(new Dog: Dog, new Cat: Cat)
```

- Основной механизм для получения отношения подтипирования на типах

# Наследование

```
trait Animal:  
  def sound: String  
  
class Dog extends Animal:      // Dog <: Animal  
  override val sound = "woof"  
  
class Cat extends Animal:      // Cat <: Animal  
  override val sound = "meow"  
  
def allSay(animals: Animal*): Unit =  
  animals.foreach(x => println(x.sound))  
  
allSay(new Dog: Dog, new Cat: Cat)
```

- Основной механизм для получения отношения подтипирования на типах
- Но не единственный (structural types, type members, ...)

# Само собой

Любой тип является подтипом самого себя (рефлексивность):

```
trait A // A <: A
```

# Можно глубже

```
trait Animal  
  
trait Mammal extends Animal // Mammal <: Animal  
  
class Dog extends Mammal    // Dog <: Mammal
```

# Можно глубже

```
trait Animal  
  
trait Mammal extends Animal // Mammal <: Animal  
  
class Dog extends Mammal    // Dog <: Mammal
```

$\text{Dog} <: \text{Mammal}, \text{Mammal} <: \text{Animal} \implies \text{Dog} <: \text{Animal}$   
(транзитивность)



# Параметрический полиморфизм

Можно задавать баунды на типы-параметры:

```
def dummy[A >: Dog <: Animal](a: A): A = a
```

В качестве A можно подставить любой тип на отрезке Dog .. Animal

# Параметрический полиморфизм

Можно задавать баунды на типы-параметры:

```
def dummy[A >: Dog <: Animal](a: A): A = a
```

В качестве A можно подставить любой тип на отрезке Dog .. Animal

## Крайние случаи

```
def dummy1[A](a: A): A = a           // A >: Nothing <: Any
def dummy2[A <: Animal](a: A): A = a  // A >: Nothing <: Animal
def dummy3[A >: Dog](a: A): A = a     // A >: Dog <: Any
def dummy4[A >: Dog <: Dog](a: A): A = a // A == Dog
```

$A ::= B \iff A <: B, B <: A$  (антисимметричность)

```
type A >: Animal <: Animal // не тоже самое, что type A = Animal
```

```
summon[A ::= Animal]
```

$A ::= B \iff A <: B, B <: A$  (антисимметричность)

```
type A >: Animal <: Animal // не тоже самое, что type A = Animal
```

```
summon[A ::= Animal]
```

Отношение подтипирования образует **частичный порядок** на типах

# Nothing

$\forall A. \text{ Nothing } <: A$  (bottom type)

```
def get: Animal = (throw new Exception("(◡◡)┐└┐ ")): Nothing
```

# Nothing

$\forall A. \text{ Nothing } <: A$  (bottom type)

```
def get: Animal = (throw new Exception("(◡‿◡)└┬┘ ")): Nothing
```

```
// метод у ZIO
final def forever(implicit trace: Trace): ZIO[R, E, Nothing] = {
  lazy val loop: ZIO[R, E, Nothing] = self *> ZIO.yieldNow *> loop

  loop
}
```

# Nothing

$\forall A. \text{ Nothing } <: A$  (bottom type)

```
def get: Animal = (throw new Exception("(◡◡)└─┐ ")): Nothing
```

```
// метод у ZIO
final def forever(implicit trace: Trace): ZIO[R, E, Nothing] = {
  lazy val loop: ZIO[R, E, Nothing] = self *> ZIO.yieldNow *> loop

  loop
}
```

```
case object None extends Option[Nothing] // forall A. Option[Nothing] <: Option[A]
```

# Множественное наследование

```
trait Mammal  
trait Domestic  
  
class Dog extends Mammal, Domestic // Dog <: Mammal, Dog <: Domestic
```



# Множественное наследование

```
trait Mammal  
trait Domestic  
  
class Dog extends Mammal, Domestic // Dog <: Mammal & Domestic
```

# Типы-пересечения

$A \ \& \ B$  – тип, который населяют значения, одновременно населяющие  $A$  и  $B$

Правила, которые использует компилятор при выводе типов:

$$\forall A, B. \quad A \ \& \ B <: A, \quad A \ \& \ B <: B$$

# Типы-пересечения

$A \ \& \ B$  – тип, который населяют значения, одновременно населяющие  $A$  и  $B$

Правила, которые использует компилятор при выводе типов:

$$\forall A, B. \quad A \ \& \ B <: A, \quad A \ \& \ B <: B$$

$$\forall A, B, C. \quad A <: B, \quad A <: C \implies A <: B \ \& \ C$$

# Типы-пересечения

$A \ \& \ B$  – тип, который населяют значения, одновременно населяющие  $A$  и  $B$

Правила, которые использует компилятор при выводе типов:

$$\forall A, B. \quad A \ \& \ B <: A, \ A \ \& \ B <: B$$

$$\forall A, B, C. \quad A <: B, \ A <: C \implies A <: B \ \& \ C$$

**Следствия:**

$$1) \quad \forall A, B. \quad A \ \& \ B = B \ \& \ A \quad (\text{коммутативность})$$

# Типы-пересечения

$A \ \& \ B$  – тип, который населяют значения, одновременно населяющие  $A$  и  $B$

Правила, которые использует компилятор при выводе типов:

$$\forall A, B. \quad A \ \& \ B <: A, \quad A \ \& \ B <: B$$

$$\forall A, B, C. \quad A <: B, \quad A <: C \implies A <: B \ \& \ C$$

## Следствия:

- 1)  $\forall A, B. \quad A \ \& \ B = B \ \& \ A$  (коммутативность)
- 2)  $\forall A. \quad A \ \& \ A = A$  (идемпотентность)

# Типы-пересечения

$A \ \& \ B$  – тип, который населяют значения, одновременно населяющие  $A$  и  $B$

Правила, которые использует компилятор при выводе типов:

$$\forall A, B. \quad A \ \& \ B <: A, \ A \ \& \ B <: B$$

$$\forall A, B, C. \quad A <: B, \ A <: C \implies A <: B \ \& \ C$$

## Следствия:

- 1)  $\forall A, B. \quad A \ \& \ B = B \ \& \ A$  (коммутативность)
- 2)  $\forall A. \quad A \ \& \ A = A$  (идемпотентность)
- 3)  $\forall A, B, C. \quad A \ \& \ (B \ \& \ C) = (A \ \& \ B) \ \& \ C$  (ассоциативность)

# Типы-пересечения

$A \ \& \ B$  – тип, который населяют значения, одновременно населяющие  $A$  и  $B$

Правила, которые использует компилятор при выводе типов:

$$\forall A, B. \quad A \ \& \ B <: A, \quad A \ \& \ B <: B$$

$$\forall A, B, C. \quad A <: B, \quad A <: C \implies A <: B \ \& \ C$$

## Следствия:

- 1)  $\forall A, B. \quad A \ \& \ B = B \ \& \ A$  (коммутативность)
- 2)  $\forall A. \quad A \ \& \ A = A$  (идемпотентность)
- 3)  $\forall A, B, C. \quad A \ \& \ (B \ \& \ C) = (A \ \& \ B) \ \& \ C$  (ассоциативность)
- 4)  $\forall A, B : \quad A <: B \implies A \ \& \ B = A$

# Множество типов

$$\{ A, B, C \} = A \ \& \ B \ \& \ C$$



# Множество типов

$$\{ A, B, C \} = A \ \& \ B \ \& \ C$$

- добавление типа – пересечение с ним

$$X \ \& \ (A \ \& \ B \ \& \ C) = X \ \& \ A \ \& \ B \ \& \ C$$

# Множество типов

$$\{ A, B, C \} = A \ \& \ B \ \& \ C$$

- добавление типа – пересечение с ним

$$X \ \& \ (A \ \& \ B \ \& \ C) = X \ \& \ A \ \& \ B \ \& \ C$$

- проверка на вхождение

$$X \ >: (A \ \& \ B \ \& \ C)$$

# Множество типов

$$\{ A, B, C \} = A \ \& \ B \ \& \ C$$

- добавление типа – пересечение с ним

$$X \ \& \ (A \ \& \ B \ \& \ C) = X \ \& \ A \ \& \ B \ \& \ C$$

- проверка на вхождение

$$X \ >: (A \ \& \ B \ \& \ C)$$

- хорошо работает с подтипами:

$$X \ <: A, (A \ \& \ B \ \& \ C) \ \& \ X = X \ \& \ B \ \& \ C$$

$$X \ >: A, (A \ \& \ B \ \& \ C) \ \& \ X = A \ \& \ B \ \& \ C$$

# Пример: роли пользователя

```
case class UserRaw(  
  roles: Set[String],  
  emailOpt: Option[String],  
  emailVerifiedOpt: Option[Boolean],  
  phoneOpt: Option[String],  
  phoneVerifiedOpt: Option[Boolean],  
  deviceIdOpt: Option[String]  
)
```

# Пример: роли пользователя

```
case class UserRaw(  
  roles: Set[String],  
  emailOpt: Option[String],  
  emailVerifiedOpt: Option[Boolean],  
  phoneOpt: Option[String],  
  phoneVerifiedOpt: Option[Boolean],  
  deviceIdOpt: Option[String]  
)
```

Фактический набор полей **зависит** от того какие роли имеет пользователь:

- если есть роль `email`, то поля `emailOpt` и `emailVerifiedOpt` заданы
- если есть роль `phone`, то поля `phoneOpt` и `phoneVerifiedOpt` заданы
- если есть роль `device`, то поле `deviceIdOpt` задано

# Пример: роли пользователя

```
case class UserRaw(  
  roles: Set[String],  
  emailOpt: Option[String],  
  emailVerifiedOpt: Option[Boolean],  
  phoneOpt: Option[String],  
  phoneVerifiedOpt: Option[Boolean],  
  deviceIdOpt: Option[String]  
)
```

Фактический набор полей **зависит** от того какие роли имеет пользователь:

- если есть роль `email`, то поля `emailOpt` и `emailVerifiedOpt` заданы
- если есть роль `phone`, то поля `phoneOpt` и `phoneVerifiedOpt` заданы
- если есть роль `device`, то поле `deviceIdOpt` задано

При этом отсутствие роли не означает отсутствие соответствующих данных.

# Можно так

```
sealed trait User { ... }  
case class No(...) extends User { ... }  
case class OnlyEmail(...email: String...) extends User { ... }  
case class OnlyPhone(...phone: String....) extends User { ... }  
case class OnlyDeviceId(...deviceId: String...) extends User { ... }  
case class EmailAndPhone(...) extends User { ... }  
case class EmailAndDeviceId(...) extends User { ... }  
case class DeviceIdAndPhone(...) extends User { ... }  
case class DeviceIdAndPhoneAndEmail(...) extends User { ... }
```

# Можно так

```
sealed trait User { ... }  
case class No(...) extends User { ... }  
case class OnlyEmail(...email: String...) extends User { ... }  
case class OnlyPhone(...phone: String...) extends User { ... }  
case class OnlyDeviceId(...deviceId: String...) extends User { ... }  
case class EmailAndPhone(...) extends User { ... }  
case class EmailAndDeviceId(...) extends User { ... }  
case class DeviceIdAndPhone(...) extends User { ... }  
case class DeviceIdAndPhoneAndEmail(...) extends User { ... }
```

$2^{\text{roles.size}}$  моделей



# User[Roles]

```
object Role:
  type email
  type phone

case class User[+Roles] private (
  roles: Set[String],
  emailOpt: Option[String],
  emailVerifiedOpt: Option[Boolean],
  // ...
):
  // ...
  def email(using Roles <:: Role.email): String = emailOpt.get // get is safe here
  def emailVerified(using Roles <:: Role.email): Boolean = emailVerifiedOpt.get
```

# User[Roles]

```
object Role:
  type email
  type phone

case class User[+Roles] private (
  roles: Set[String],
  emailOpt: Option[String],
  emailVerifiedOpt: Option[Boolean],
  // ...
):
  // ...
  def setEmail(email: String, emailVerified: Boolean): User[Roles & Role.email] =
    User[Roles & Role.email](
      roles + "email",
      Some(email),
      Some(emailVerified),
      // ...
    )
```

- Естественное подтипирование

```
User[Roles.email & Roles.phone] <: User[Roles.phone]
```

- Естественное подтипирование

```
User[Roles.email & Roles.phone] <: User[Roles.phone]
```

- Добавление новой роли требует добавления/изменения константного числа методов

- Естественное подтипирование

```
User[Roles.email & Roles.phone] <: User[Roles.phone]
```

- Добавление новой роли требует добавления/изменения константного числа методов
- Валидации:

```
def validate[Roles](userRaw: UserRaw): Either[Error, User[Roles]] = ...
```

# Стоп, а как же Tuple?

```
import Tuple.*
import scala.compiletime.ops.boolean.*
import scala.compiletime.ops.any.*

case class User[Roles <: Tuple](
  roles: Set[String],
  emailOpt: Option[String],
  emailVerifiedOpt: Option[Boolean]
  // ...
):
  // ...
  def email(using (Contains[Roles, Role.email] ==> true)): String = emailOpt.get
```

# Стоп, а как же Tuple?

```
import Tuple.*
import scala.compiletime.ops.boolean.*
import scala.compiletime.ops.any.*

case class User[Roles <: Tuple](
  roles: Set[String],
  emailOpt: Option[String],
  emailVerifiedOpt: Option[Boolean]
  // ...
):
  // ...
  def email(using (Contains[Roles, Role.email] ==> true)): String = emailOpt.get
  def setEmail(
    email: String,
    emailVerified: Boolean
  ): User[Role.email *: Filter[Roles, [x] => ![x = Role.email]] =
    User(roles + "email", Some(email), Some(emailVerified) /*, ... */)

```

## Типы перечения

- + Естественное подтипирование
- + Совместимость с разными версиями Scala (2 / 3)

## Tuple

- + Поддержка на уровне языка (для Scala 3)
- + remove легко реализовать



## Типы перечения

- + Естественное подтипирование
- + Совместимость с разными версиями Scala (2 / 3)
- Без дополнительных макросов плохочитаемые ошибки
- С `remove` будут проблемы

## Tuple

- + Поддержка на уровне языка (для Scala 3)
- + `remove` легко реализовать
- Нет гарантии уникальности типов, надо отдельно за этим следить
- `shapeless` для Scala 2

# Пример: ZIO

```
//> using dep dev.zio::zio:2.1.13
import zio.*

val program: RIO[String & Boolean, Unit] =
  for
    str  ← ZIO.service[String]
    _    ← Console.println(s"I got $str!")
    bool ← ZIO.service[Boolean]
    _    ← Console.println(s"I got $bool!")
  yield ()

object App extends ZIOAppDefault:
  val run = program.provide(ZLayer.succeed("it") and ZLayer.succeed(true))
```

# DIY

```
val ctx: Ctx[String & Boolean] =  
  Ctx.empty  
    .add[String]("it")  
    .add[Boolean](true)  
  
val str = ctx.get[String]  
println(s"I got $str!") // prints "I got it!"  
  
val bool = ctx.get[Boolean]  
println(s"I got $bool!") // prints "I got true!"  
  
// ctx.get[Int]           // doesn't compile
```

# Ctx[R]

```
trait Ctx[+R]:  
  def add[X](x: X): Ctx[R & X]  
  def get[X >: R]: X
```

# Ctx[R]

```
trait Ctx[+R]:  
  def add[X](x: X): Ctx[R & X]  
  def get[X >: R]: X  
  
case class Impl[+R](map: Map[???, ???]) extends Ctx[R]  
  def add[X](x: X): Ctx[R & X] =  
    Impl[R & X](map + (??? → x))  
  def get[X >: R]: X =  
    map.get(???)
```

# Ctx[R]

```
//> using dep dev.zio::izumi-reflect:2.3.10
import izumi.reflect.Tag
import izumi.reflect.macrorrti.LightTypeTag

trait Ctx[+R]:
  def add[X: Tag](x: X): Ctx[R & X]
  def get[X >: R: Tag]: X

object Ctx:
  private case class Impl[+R](map: Map[LightTypeTag, Any]) extends Ctx[R]:
    def add[X: Tag](x: X): Ctx[R & X] =
      Impl[R & X](map + (Tag[X].tag → x))
    def get[X >: R: Tag]: X =
      map.getOrElse(Tag[X].tag, throw new Exception("shouldn't happen")).asInstanceOf[X]

  def empty: Ctx[Any] = new Impl[Any](Map())
```

```
val ctx: Ctx[String & Boolean] =  
  Ctx.empty  
    .add[String]("it")  
    .add[Boolean](true)  
  
val str = ctx.get[String]  
println(s"I got $str!") // prints "I got it!"  
  
val bool = ctx.get[Boolean]  
println(s"I got $bool!") // prints "I got true!"
```

```
val ctx: Ctx[String & Boolean] =  
  Ctx.empty  
    .add[String]("it")  
    .add[Boolean](true)  
  
val str = ctx.get[String]  
println(s"I got $str!") // prints "I got it!"  
  
val bool = ctx.get[Boolean]  
println(s"I got $bool!") // prints "I got true!"  
// ctx.get[Int] // doesn't compile
```



```
val ctx: Ctx[String & Boolean] =  
  Ctx.empty  
    .add[String]("it")  
    .add[Boolean](true)  
  
val str = ctx.get[String]  
println(s"I got $str!") // prints "I got it!"  
  
val bool = ctx.get[Boolean]  
println(s"I got $bool!") // prints "I got true!"  
// ctx.get[Int] // doesn't compile  
// ctx.get[String & Boolean] // Caused by: java.lang.Exception: shouldn't happen
```

# А можно через типы-объединения?

$A \mid B$  – тип, который населяют значения, населяющие  $A$  или  $B$ .

# А можно через типы-объединения?

$A \mid B$  – тип, который населяют значения, населяющие  $A$  или  $B$ .

Объединение также коммутативно, ассоциативно, идемпотентно.

# А можно через типы-объединения?

$A \mid B$  – тип, который населяют значения, населяющие  $A$  или  $B$ .

Объединение также коммутативно, ассоциативно, идемпотентно.

Разве не удобнее и естественнее использовать его для кодирования множеств?

$$\{A, B, C\} = A \mid B \mid C$$

# А можно через типы-объединения?

$A \mid B$  – тип, который населяют значения, населяющие  $A$  **или**  $B$ .

Объединение также коммутативно, ассоциативно, идемпотентно.

Разве не удобнее и естественнее использовать его для кодирования множеств?

$$\{A, B, C\} = A \mid B \mid C$$

Тогда  $\text{Ctx}[+R]$  пришлось бы делать контравариантным, иначе  $\text{Ctx}[A] \leq \text{Ctx}[A \mid B]$ , что менее удобно в данном случае

# Другие свойства

```
//> using dep dev.zio::zio:2.1.13
import zio.*

def program(
  flag: Boolean
): RIO[String & Boolean, Unit] = // RIO[String, Unit] | RIO[Boolean, Unit]
  flag match
    case true  => ZIO.service[String].flatMap(s => Console.println(s)) //: RIO[String, Unit]
    case false => ZIO.service[Boolean].flatMap(b => Console.println(b))//: RIO[Boolean, Unit]
```

# Другие свойства

```
//> using dep dev.zio::zio:2.1.13
import zio.*

def program(
  flag: Boolean
): RIO[String & Boolean, Unit] = // RIO[String, Unit] | RIO[Boolean, Unit]
  flag match
    case true  => ZIO.service[String].flatMap(s => Console.println(s)) //: RIO[String, Unit]
    case false => ZIO.service[Boolean].flatMap(b => Console.println(b))//: RIO[Boolean, Unit]
```

$$\forall F[-\_], A, B. \quad F[A] \mid F[B] = F[A \& B]$$

# Другие свойства

```
//> using dep dev.zio::zio:2.1.13
import zio.*

def program(
  flag: Boolean
): RIO[String & Boolean, Unit] = // RIO[String, Unit] | RIO[Boolean, Unit]
  flag match
    case true  => ZIO.service[String].flatMap(s => Console.println(s)) //: RIO[String, Unit]
    case false => ZIO.service[Boolean].flatMap(b => Console.println(b))//: RIO[Boolean, Unit]
```

$$\forall F[-\_], A, B. \quad F[A] \mid F[B] = F[A \& B]$$

$$\forall F[+_], A, B. \quad F[A] \mid F[B] = F[A \mid B], \quad F[A] \& F[B] = F[A \& B]$$



# Пример: kyo

```
//> using dep io.getkyo::kyo-core:0.14.1
import kyo.*
import java.io.IOException

val program: Unit < (IO & Abort[IOException] & Env[String] & Env[Boolean]) =
  for
    str  ← Env.get[String]
    _    ← Console.println(s"I got $str!")

    bool ← Env.get[Boolean]
    _    ← Console.println(s"I got $bool!")
  yield ()
```

# Пример: kyo

```
//> using dep io.getkyo::kyo-core:0.14.1
import kyo.*
import java.io.IOException

val program: Unit < (IO & Abort[IOException] & Env[String] & Env[Boolean]) =
  for
    str  ← Env.get[String]
    _    ← Console.println(s"I got $str!")

    bool ← Env.get[Boolean]
    _    ← Console.println(s"I got $bool!")
  yield ()

object App extends KyoApp:
  run(Env.run("it")(Env.run(true)(program)))
```

# Пример: kyo

```
//> using dep io.getkyo::kyo-core:0.14.1
import kyo.*
import java.io.IOException

val program: Unit < (IO & Abort[IOException] & Env[String & Boolean]) =
    // = Env[String] & Env[Boolean], так как Env[+R]

    for
        str  ← Env.get[String]
        _    ← Console.println(s"I got $str!")

        bool ← Env.get[Boolean]
        _    ← Console.println(s"I got $bool!")
    yield ()

object App extends KyoApp:
    run(Env.run("it")(Env.run(true)(program)))
```

# Пример: параметры отправки http запросов

```
val request: Request => IO[Error, Response] =
  clientBuilder.buildSafe[Request, Error, Response](
    TRequest((r: Request) => basicRequest.get(...).response { ... }) // sttp request
      .add[Loggables.Param](Loggables())
      .add[RequestTimeout.Param](RequestTimeout(10000.milliseconds))
      .add[Retries.Param](
        Retries.Enabled(
          strategy = Retries.Strategy.whenAny,
          config = Retries.Config
            .default(4)
            .withDelay(BackoffDelay.Exponent.Binary(1.second)),
        )
      )
  )
```

# Пример: параметры отправки http запросов

```
val request: Request => IO[Error, Response] =
  clientBuilder.buildSafe[Request, Error, Response](
    TRequest((r: Request) => basicRequest.get(...).response { ... }) // sttp request
      .add[Loggables.Param](Loggables()) // Loggables[Request, Error, Response]
      .add[RequestTimeout.Param](RequestTimeout(10000.milliseconds))
      .add[Retries.Param](
        Retries.Enabled(
          strategy = Retries.Strategy.whenAny,
          config = Retries.Config
            .default(4)
            .withDelay(BackoffDelay.Exponent.Binary(1.second)),
        )
      )
  )
```

# Пример: параметры отправки http запросов

```
val request: Request => IO[Error, Response] =
  clientBuilder.buildSafe[Request, Error, Response](
    TRequest((r: Request) => basicRequest.get(...).response { ... }) // sttp request
      .add[Loggables.Param](Loggables()) // Loggables[Request, Error, Response]
      .add[RequestTimeout.Param](RequestTimeout(10000.milliseconds))
      .add[Retries.Param](
        Retries.Enabled(
          strategy = Retries.Strategy.whenAny,
          config = Retries.Config
            .default(4)
            .withDelay(BackoffDelay.Exponent.Binary(1.second))
        )
      )
    // : TRequest[Request, Error, Response,
    //      Has[RequestTimeout.Param] & Has[Retries.Param] & Has[Loggables.Param]
    // ]
  )
```

# Пример: параметры отправки http запросов

```
val clientBuilder = ZioTClientBuilder.empty[ZEnv](backend)
  .append(TimeoutMiddleware.make[ZEnv].requiredAll)
  .append(LoggingObserver.default[UIO, URIO[ZEnv, *]])
  .append(RetryMiddleware.default[ZEnv])
// MetricsObserver, TraceMiddleware, CircuitBreakerMiddleware, CacheMiddleware ...
```

# Пример: параметры отправки http запросов

```
val clientBuilder = ZioTClientBuilder.empty[ZEnv](backend)
  .append(TimeoutMiddleware.make[ZEnv].requiredAll)
  .append(LoggingObserver.default[UIO, URIO[ZEnv, *]])
  .append(RetryMiddleware.default[ZEnv])
: ZioTClientBuilder[ZEnv, R,
  Mdt[RequestTimeout.Param] & Mdt[Loggables.Param] & Opt[Retries.Param]
]
```



# Пример: параметры отправки http запросов

```
val clientBuilder = ZioClientBuilder.empty[ZEnv](backend)
  .append(TimeoutMiddleware.make[ZEnv].requiredAll)
  .append(LoggingObserver.default[UIO, URIO[ZEnv, *]])
  .append(RetryMiddleware.default[ZEnv])
: ZioClientBuilder[ZEnv, R,
  Mdt[RequestTimeout.Param] & Mdt[Loggables.Param] & Opt[Retries.Param]
]
```

```
(Has[RequestTimeout.Param] & Has[Loggables.Param] & Has[Retries.Param])
  <:
(Mdt[RequestTimeout.Param] & Mdt[Loggables.Param] & Opt[Retries.Param])
```

## Пример: quantitative

Операции над физическими величинами требует того, чтобы размерности сходились:

$$s = ut + \frac{1}{2}at^2$$

# Пример: quantitative

Операции над физическими величинами требует того, чтобы размерности сходились:

$$s = ut + \frac{1}{2}at^2$$

```
//> using dep dev.soundness:quantitative-core:0.22.0
//> using option "-experimental"
import quantitative.*

type Velocity      = Quantity[Metres[1] & Seconds[-1]]
type Time          = Quantity[Seconds[1]]
type Acceleration  = Quantity[Metres[1] & Seconds[-2]]
type Distance      = Quantity[Metres[1]]

def s(u: Velocity, t: Time, a: Acceleration): Distance =
  u*t + 0.5*a*t*t
```

# Итоги

- Подтипирование в Scala хорошо дружит с другими фичами языка

# Итоги

- Подтипирование в Scala хорошо дружит с другими фичами языка
- При этом добавляются всяческие полезные свойства

# Итоги

- Подтипирование в Scala хорошо дружит с другими фичами языка
- При этом добавляются всяческие полезные свойства
- Если где-то вам требуется использовать множество типов, где
  - каждый тип уникален,
  - порядок не важен

попробуйте реализацию через типы пересечения

# Благодарности

- Денис Буздалов
- Иван Лягаев
- Петр и Александра Троицкие
- Ахтям Сакаев
- Михаил Чугунков



# Ссылки

- [Zionomicon](#) (Chapter 18: Advanced Dependency Injection)
- [Izumi Reflect: Scala Type System Model by Kai and Pavel Shirshov](#)
- [getkyo.io](#)
- [soundness.dev/quantitative](#)





# Ссылки

- [Zionomicon](#) (Chapter 18: Advanced Dependency Injection)
- [Izumi Reflect: Scala Type System Model by Kai and Pavel Shirshov](#)
- [getkyo.io](#)
- [soundness.dev/quantitative](#)



Спасибо!