IT's TINKOFF

Позвольте уточнить

Как писать new/refined типы железно и не очень

Алексей Троицкий

28.10.2023

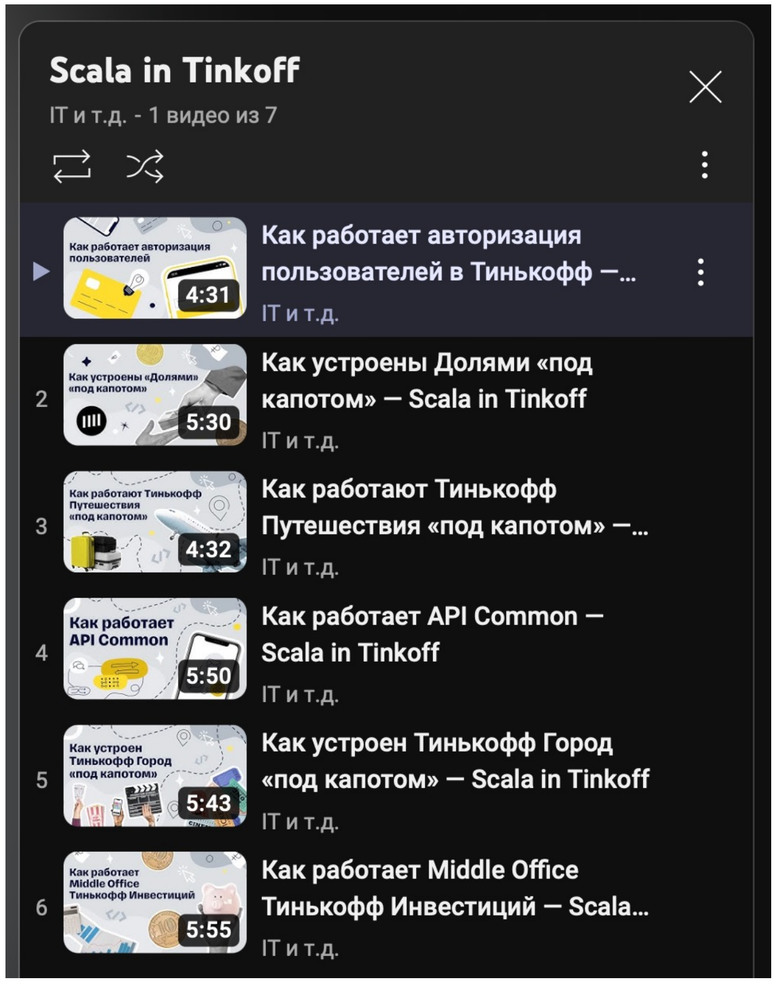
IT's TINKOFF

https://github.com/road21/talks

Алексей Троицкий

Telegram, github: road21

IT's TINKOFF



https://github.com/road21/talks

План

- Поставим проблему
- Решим ее используя **Scala 3**
- Решим с помощью библиотеки, посмотрим на преимущества
- Доработаем наше решение

Модель

```
case class Adult(
  id: UUID,
  name: String,
  email: Option[String],
  children: Vector[UUID]
)

case class Child(id: UUID, name: String)
```

Модель

```
case class Adult(
  id: UUID,
  name: String,
  email: Option[String],
  children: Vector[UUID]
)

case class Child(id: UUID, name: String)
```

```
val p = Child(UUID.randomUUID(), "Петя")


val s = Adult(UUID.randomUUID(), "Саша", None, Vector(p.id))
val l = Adult(UUID.randomUUID(),"Леша", Some("amtroitskiy@gmail.com"), Vector(p.id))
```

Модель

```
case class Adult(
  id: UUID,
  name: String,
  email: Option[String],
  children: Vector[UUID]
)

case class Child(id: UUID, name: String)
```

```
val p = Child(UUID.randomUUID(), "Петя")


val s = Adult(UUID.randomUUID(), "Саша", None, Vector(p.id))
val l = Adult(UUID.randomUUID(),"Леша", Some("amtroitskiy@gmail.com"), Vector(s.id))
```

Типы в помощь

```
case class Adult(
  id: AdultId,
  name: String,
  email: Option[String],
  children: Vector[ChildId]
)

case class Child(id: ChildId, name: String)
```

Типы в помощь

```
case class Adult(
  id: AdultId,
  name: String,
  email: Option[String],
  children: Vector[ChildId]
)

case class Child(id: ChildId, name: String)
```

- `UUID ~= AdultId ~= ChildId`
- `!(ChildId <: AdultId)`
- `!(AdultId <: ChildId)`

case class

```
case class AdultId(value: UUID)

case class ChildId(value: UUID)


case class Adult(
  id: AdultId,
  name: String,
  email: Option[String],
  children: Vector[ChildId]
)

case class Child(id: ChildId, name: String)
```

Инстансы

```
import io.circe.{Encoder, Decoder}

case class AdultId(value: UUID)

case class ChildId(value: UUID)


case class Adult(
  id: AdultId,
  name: String,
  email: Option[String],
  children: Vector[ChildId]
) derives Decoder, Encoder.AsObject


case class Child(id: ChildId, name: String)
  derives Decoder, Encoder.AsObject
```

Инстансы

```
import io.circe.{Encoder, Decoder}

case class AdultId(value: UUID)

case class ChildId(value: UUID)


case class Adult(
  id: AdultId,
  name: String,
  email: Option[String],
  children: Vector[ChildId]
) derives Decoder, Encoder.AsObject


case class Child(id: ChildId, name: String)
  derives Decoder, Encoder.AsObject
```

Инстансы

```scala
import io.circe.{Encoder, Decoder}
```

```scala
case class AdultId(value: UUID)
object AdultId:
  given Encoder[AdultId] = Encoder[UUID].contramap(_.value)
  given Decoder[AdultId] = Decoder[UUID].map(AdultId(_))
```

```scala
case class ChildId(value: UUID)
object ChildId:
  given Encoder[ChildId] = Encoder[UUID].contramap(_.value)
  given Decoder[ChildId] = Decoder[UUID].map(ChildId(_))
```

Больше инстансов

```
import io.circe.{Encoder, Decoder}
import logstage.LogstageCodec
import sttp.tapir.Schema

case class AdultId(value: UUID)
object AdultId:
  given Encoder[AdultId] = Encoder[UUID].contramap(_.value)
  given Decoder[AdultId] = Decoder[UUID].map(AdultId(_))
  given LogstageCodec[AdultId] = LogstageCodec[String].contramap(_.value.toString)
  given Schema[AdultId] = summon[Schema[UUID]].description("UUID identifier of adult user").as

case class ChildId(value: UUID)
object ChildId:
  given Encoder[ChildId] = Encoder[UUID].contramap(_.value)
  given Decoder[ChildId] = Decoder[UUID].map(ChildId(_))
  given LogstageCodec[ChildId] = LogstageCodec[String].contramap(_.value.toString)
  given Schema[ChildId] = summon[Schema[UUID]].description("UUID identifier of child user").as
```

Рефакторим

```scala
trait WrapsId:
  def value: UUID

abstract class IdWrapper[A <: WrapsId](entityName: String):
  def apply(uuid: UUID): A

  given Encoder[A] = Encoder[UUID].contramap(_.value)
  given Decoder[A] = Decoder[UUID].map(apply)
  given LogstageCodec[A] = LogstageCodec[String].contramap(_.value.toString)
  given Schema[A] = summon[Schema[UUID]].description(s"UUID identifier of $entityName").as
```

Рефакторим

```scala
trait WrapsId:
  def value: UUID

abstract class IdWrapper[A <: WrapsId](entityName: String):
  def apply(uuid: UUID): A

  given Encoder[A] = Encoder[UUID].contramap(_.value)
  given Decoder[A] = Decoder[UUID].map(apply)
  given LogstageCodec[A] = LogstageCodec[String].contramap(_.value.toString)
  given Schema[A] = summon[Schema[UUID]].description(s"UUID identifier of $entityName").as
```

```scala
case class AdultId(value: UUID) extends WrapsId
object AdultId extends IdWrapper[AdultId]("Adult")

case class ChildId(value: UUID) extends WrapsId
object ChildId extends IdWrapper[ChildId]("Child")
```

Рефакторим

```
trait WrapsId:
  def value: UUID

abstract class IdWrapper[A <: WrapsId](entityName: String):
  def apply(uuid: UUID): A

  given Encoder[A] = Encoder[UUID].contramap(_.value)
  given Decoder[A] = Decoder[UUID].map(apply)
  given LogstageCodec[A] = LogstageCodec[String].contramap(_.value.toString)
  given Schema[A] = summon[Schema[UUID]].description(s"UUID identifier of $entityName").as
```

```
case class AdultId(value: UUID) extends WrapsId
object AdultId extends IdWrapper[AdultId]("Adult")

case class ChildId(value: UUID) extends WrapsId
object ChildId extends IdWrapper[ChildId]("Child")
```

- *В рантайме аллоцируются инстансы* **AdultId, ChildId**

Value class

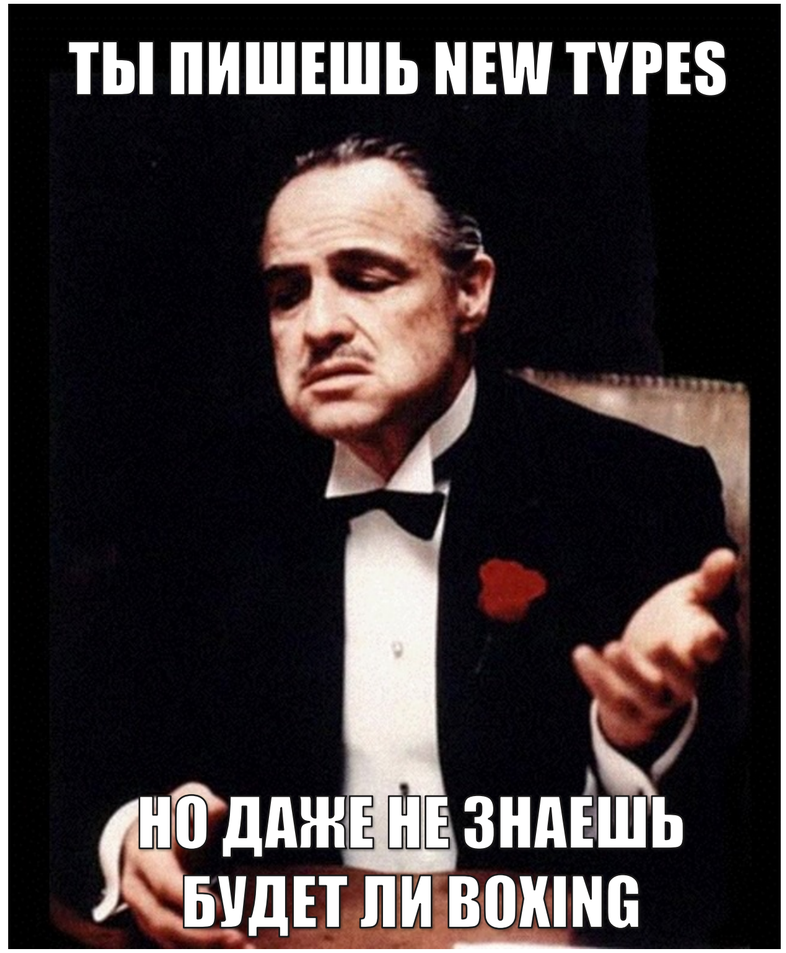
```scala
trait WrapsId extends Any:
  def value: UUID

abstract class IdWrapper[A <: WrapsId](entityName: String):
  def apply(uuid: UUID): A

  given Encoder[A] = Encoder[UUID].contramap(_.value)
  given Decoder[A] = Decoder[UUID].map(apply)
  given LogstageCodec[A] = LogstageCodec[String].contramap(_.value.toString)
  given Schema[A] = summon[Schema[UUID]].description(s"UUID identifier of $entityName").as
```

```scala
case class AdultId(value: UUID) extends AnyVal, WrapsId
object AdultId extends IdWrapper[AdultId]("Adult")

case class ChildId(value: UUID) extends AnyVal, WrapsId
object ChildId extends IdWrapper[ChildId]("Child")
```

Value class

```
trait WrapsId extends Any:
  def value: UUID

abstract class IdWrapper[A <: WrapsId](entityName: String):
  def apply(uuid: UUID): A

  given Encoder[A] = Encoder[UUID].contramap(_.value)
  given Decoder[A] = Decoder[UUID].map(apply)
  given LogstageCodec[A] = LogstageCodec[String].contramap...
  given Schema[A] = summon[Schema[UUID]].description...
```

```
case class AdultId(value: UUID) extends AnyVal, WrapsId
object AdultId extends IdWrapper[AdultId]("Adult")

case class ChildId(value: UUID) extends AnyVal, WrapsId
object ChildId extends IdWrapper[ChildId]("Child")
```



- *Могут все равно происходить аллокации, нет ошибки/ворнинга от компилятора*

Type members

```
type ChildId = ChildId.T
object ChildId:
  type T <: UUID
  def apply(s: UUID): T = s.asInstanceOf[T]

type AdultId = AdultId.T
object AdultId:
  type T <: UUID
  def apply(s: UUID): T = s.asInstanceOf[T]
```

Type members

```
type ChildId = ChildId.T
object ChildId:
  type T <: UUID
  def apply(s: UUID): T = s.asInstanceOf[T]

type AdultId = AdultId.T
object AdultId:
  type T <: UUID
  def apply(s: UUID): T = s.asInstanceOf[T]
```

Type members

```scala
type ChildId = ChildId.T
object ChildId:
  type T <: UUID
  def apply(s: UUID): T = s.asInstanceOf[T]

type AdultId = AdultId.T
object AdultId:
  type T <: UUID
  def apply(s: UUID): T = s.asInstanceOf[T]
```

Scala 2

```scala
import io.estatico.newtype.macros.newtype

@newtype case class ChildId(toUUID: UUID)
@newtype case class AdultId(toUUID: UUID)
```

Opaque types

```
type ChildId = ChildId.T
object ChildId:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s

type AdultId = AdultId.T
object AdultId:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s
```

Opaque types

```scala
type ChildId = ChildId.T
object ChildId:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s

type AdultId = AdultId.T
object AdultId:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s
```

Снаружи объекта **ChildId**:

```scala
type T <: UUID
```

Внутри объекта **ChildId**:

```scala
type T = UUID
```

Opaque types

```
type ChildId = ChildId.T
object ChildId:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s

type AdultId = AdultId.T
object AdultId:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s
```

Снаружи объекта **ChildId**:

```
type T <: UUID
```

Внутри объекта **ChildId**:

```
type T = UUID
```

Инстансы

```scala
type ChildId = ChildId.T
object ChildId:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s

  given Encoder[T] = Encoder[UUID].contramap(x => x)
  given Decoder[T] = Decoder[UUID].map(apply)
```

Инстансы

```scala
type ChildId = ChildId.T
object ChildId:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s

  given Encoder[T] = Encoder[UUID].contramap(x => x)
  given Decoder[T] = Decoder[UUID].map(apply)
```

```
[warn] Infinite loop in function body
[warn] io.circe.Encoder.apply[java.util.UUID](this.given_Encoder_T).contramap[
[warn]   java.util.UUID](
[warn]   {
[warn]     def $anonfun(x: java.util.UUID): java.util.UUID = x
[warn]     closure($anonfun)
[warn]   }
[warn] )
[warn]   given Encoder[T] = Encoder[UUID].contramap(x => x)
[warn]                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Инстансы

```
type ChildId = ChildId.T
object ChildId extends ChildInstances:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s

trait ChildInstances:
  given Encoder[ChildId] = Encoder[UUID].contramap(x => x)
  given Decoder[ChildId] = Decoder[UUID].map(ChildId.apply)
```

Больше инстансов

```
type ChildId = ChildId.T
object ChildId extends ChildInstances:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s
trait ChildInstances:
  given Encoder[ChildId] = Encoder[UUID].contramap(x => x)
  given Decoder[ChildId] = Decoder[UUID].map(ChildId.apply)
  given LogstageCodec[ChildId] = LogstageCodec[String].contramap(_.toString)
  given Schema[ChildId] = summon[Schema[UUID]].description(s"UUID identifier of child user").as

type AdultId = AdultId.T
object AdultId extends AdultInstances:
  opaque type T <: UUID = UUID
  def apply(s: UUID): T = s
trait AdultInstances:
  given Encoder[AdultId] = Encoder[UUID].contramap(x => x)
  given Decoder[AdultId] = Decoder[UUID].map(AdultId.apply)
  given LogstageCodec[AdultId] = LogstageCodec[String].contramap(_.toString)
  given Schema[AdultId] = summon[Schema[UUID]].description(s"UUID identifier of adult user").as
```

Рефакторим

```
trait NewTypeBase[A]:
  opaque type T <: A = A
  def apply(s: A): T = s
```

Рефакторим

```
trait NewTypeBase[A]:
  opaque type T <: A = A
  def apply(s: A): T = s

abstract class UUIDNewType(entityName: String) extends NewTypeBase[UUID]:
  given Encoder[T] = Encoder[UUID].contramap(x => x)
  given Decoder[T] = Decoder[UUID].map(apply)
  given LogstageCodec[T] = LogstageCodec[String].contramap(_.toString)
  given Schema[T] = summon[Schema[UUID]].description(s"UUID identifier of $entityName").as
```

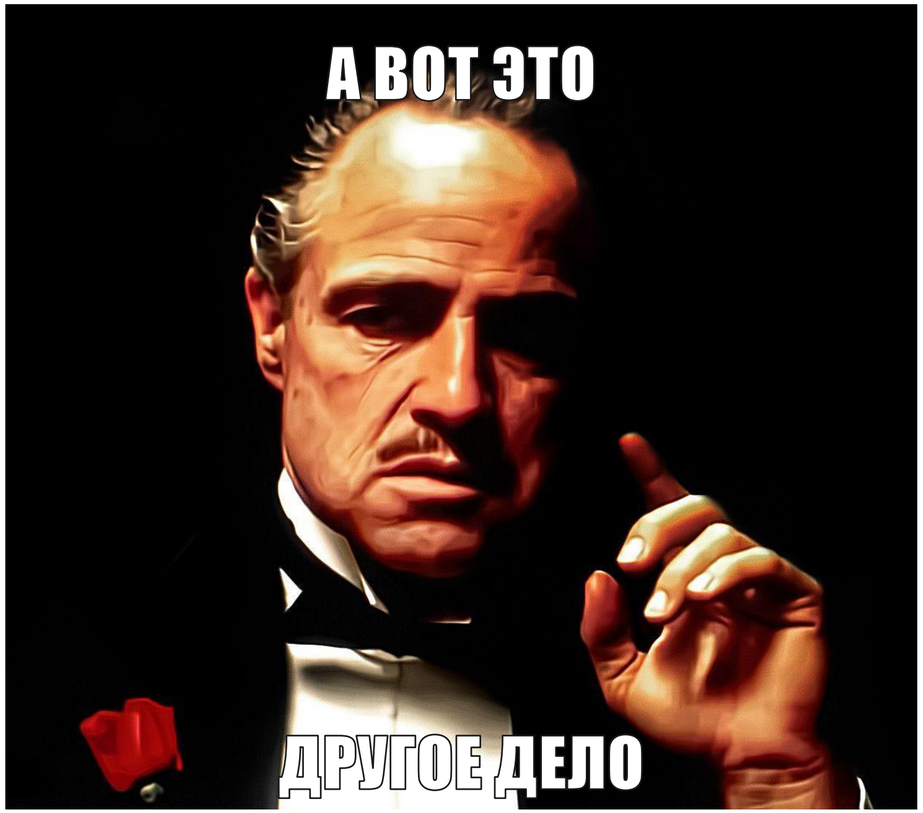
Рефакторим

```scala
trait NewTypeBase[A]:
  opaque type T <: A = A
  def apply(s: A): T = s

abstract class UUIDNewType(entityName: String) extends NewTypeBase[UUID]:
  given Encoder[T] = Encoder[UUID].contramap(x => x)
  given Decoder[T] = Decoder[UUID].map(apply)
  given LogstageCodec[T] = LogstageCodec[String].contramap(_.toString)
  given Schema[T] = summon[Schema[UUID]].description(s"UUID identifier of $entityName").as
```

```scala
type AdultId = AdultId.T
object AdultId extends UUIDNewType("Adult")

type ChildId = ChildId.T
object ChildId extends UUIDNewType("Child")
```

Рефакторим

```scala
trait NewTypeBase[A]:
  opaque type T <: A = A
  def apply(s: A): T = s

abstract class UUIDNewType(entityName: String) ...
  given Encoder[T] = Encoder[UUID].contramap(x => x)
  given Decoder[T] = Decoder[UUID].map(apply)
  given LogstageCodec[T] = LogstageCodec[String]...
  given Schema[T] = summon[Schema[UUID]]...
```




```scala
type AdultId = AdultId.T
object AdultId extends UUIDNewType("Adult")

type ChildId = ChildId.T
object ChildId extends UUIDNewType("Child")
```

- *Компактная запись*
- *Нет лишнего оверхеда*
- *Инстансы на месте*

Модель

```
type AdultId = AdultId.T
object AdultId extends UUIDNewType("Adult")

type ChildId = ChildId.T
object ChildId extends UUIDNewType("Child")

case class Adult(
  id: AdultId,
  name: String,
  email: Option[String],
  children: Vector[ChildId]
) derives Encoder.AsObject, Decoder

case class Child(
  id: ChildId,
  name: String
) derives Encoder.AsObject, Decoder
```

Модель

```scala
type AdultId = AdultId.T
object AdultId extends UUIDNewType("Adult")

type ChildId = ChildId.T
object ChildId extends UUIDNewType("Child")

case class Adult(
  id: AdultId,
  name: String,
  email: Option[String],
  children: Vector[ChildId]
) derives Encoder.AsObject, Decoder

case class Child(
  id: ChildId,
  name: String
) derives Encoder.AsObject, Decoder
```

Имя:

- Непустое
- Не начинается и не заканчивается на пробел
- Не больше 100 символов

Email:

- Удовлетворяет регулярному выражению
  `(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]...`

Refined

```scala
trait NewTypeBase[A]:
  opaque type T <: A = A
  def apply(s: A): T = s
```

```scala
trait RefinedBase[A]:
  opaque type T <: A = A
  def apply(s: A): Either[String, T]
```

Refined

```scala
trait NewTypeBase[A]:
  opaque type T <: A = A
  def apply(s: A): T = s
```

```scala
trait RefinedBase[A]:
  opaque type T <: A = A
  def apply(s: A): Either[String, T]
```

Refined

```scala
trait NewTypeBase[A]:
  opaque type T <: A = A
  def apply(s: A): T = s

abstract class UUIDNewType(entityName: String)
  extends NewTypeBase[UUID]:

  given Encoder[T] =
    Encoder[UUID].contramap(x => x)
  given Decoder[T] =
    Decoder[UUID].map(apply)
  given LogstageCodec[T] =
    LogstageCodec[String].contramap(_.toString)
  given Schema[T] =
    summon[Schema[UUID]]
      .description(
        s"UUID identifier of $entityName"
      ).as
```

```scala
trait RefinedBase[A]:
  opaque type T <: A = A
  def apply(s: A): Either[String, T]

trait StringRefined extends RefinedBase[String]:
  def description: String

  given Encoder[T] =
    Encoder[String].contramap(x => x)
  given Decoder[T] =
    Decoder[String].emap(apply)
  given LogstageCodec[T] =
    LogstageCodec[String].contramap(x => x)
  given Schema[T] =
    summon[Schema[String]]
      .description(description)
      .as
```

Refined

```scala
trait NewTypeBase[A]:
  opaque type T <: A = A
  def apply(s: A): T = s

abstract class UUIDNewType(entityName: String)
  extends NewTypeBase[UUID]:

  given Encoder[T] =
    Encoder[UUID].contramap(x => x)
  given Decoder[T] =
    Decoder[UUID].map(apply)
  given LogstageCodec[T] =
    LogstageCodec[String].contramap(_.toString)
  given Schema[T] =
    summon[Schema[UUID]]
      .description(
        s"UUID identifier of $entityName"
      ).as
```

```scala
trait RefinedBase[A]:
  opaque type T <: A = A
  def apply(s: A): Either[String, T]

trait StringRefined extends RefinedBase[String]:
  def description: String

  given Encoder[T] =
    Encoder[String].contramap(x => x)
  given Decoder[T] =
    Decoder[String].emap(apply)
  given LogstageCodec[T] =
    LogstageCodec[String].contramap(x => x)
  given Schema[T] =
    summon[Schema[String]]
      .description(description)
      .as
```

Refined

```
type Name = Name.T
object Name extends StringRefined:
  val maxLength = 100
  override val description =
    s"non empty, trimmed and not longer than
$maxLength symbols string"

  override def apply(s: String):
Either[String, Name.T] =
    Either.cond(
      s.nonEmpty && s.head != ' ' && s.last !=
' ' && s.length <= maxLength,
      s.asInstanceOf[T],
      s"Expected $description, got: '$s'"
    )
```

```
trait RefinedBase[A]:
  opaque type T <: A = A
  def apply(s: A): Either[String, T]

trait StringRefined extends RefinedBase[String]:
  def description: String

 given Encoder[T] =
    Encoder[String].contramap(x => x)
  given Decoder[T] =
    Decoder[String].emap(apply)
  given LogstageCodec[T] =
    LogstageCodec[String].contramap(x => x)
  given Schema[T] =
    summon[Schema[String]]
      .description(description)
      .as
```

Refined

```scala
type Name = Name.T
object Name extends StringRefined:
  val maxLength = 100
  override val description =
    s"non empty, trimmed and not longer than
$maxLength symbols string"

  override def apply(s: String):
Either[String, Name.T] =
    Either.cond(
      s.nonEmpty && s.head != ' ' && s.last !=
' ' && s.length <= maxLength,
      s.asInstanceOf[T],
      s"Expected $description, got: '$s'"
    )
```

```scala
trait RefinedBase[A]:
  opaque type T <: A = A
  def apply(s: A): Either[String, T]

trait StringRefined extends RefinedBase[String]:
  def description: String

 given Encoder[T] =
    Encoder[String].contramap(x => x)
  given Decoder[T] =
    Decoder[String].emap(apply)
  given LogstageCodec[T] =
    LogstageCodec[String].contramap(x => x)
  given Schema[T] =
    summon[Schema[String]]
      .description(description)
      .as
```

Refined

```
type Name = Name.T
object Name extends StringRefined:
  val maxLength = 100
  override val description =
    s"non empty, trimmed and not longer than
$maxLength symbols string"

  override def apply(s: String):
Either[String, Name.T] =
    Either.cond(
      s.nonEmpty && s.head != ' ' && s.last !=
' ' && s.length <= maxLength,
      s.asInstanceOf[T],
      s"Expected $description, got: '$s'"
    )
```

Непонятно какое именно ограничение нарушается в случае ошибки

```
trait RefinedBase[A]:
  opaque type T <: A = A
  def apply(s: A): Either[String, T]

trait StringRefined extends RefinedBase[String]:
  def description: String

 given Encoder[T] =
    Encoder[String].contramap(x => x)
  given Decoder[T] =
    Decoder[String].emap(apply)
  given LogstageCodec[T] =
    LogstageCodec[String].contramap(x => x)
  given Schema[T] =
    summon[Schema[String]]
      .description(description)
      .as
```

Refined

```
trait RefinedBase[A]:
  opaque type T <: A = A

  def predicates: Vector[(String, A => Boolean)]

  def apply(s: A): Either[String, T] =
    val errs = predicates.collect { case (descr, pred) if !pred(s) => descr }
    if (errs.isEmpty) Right(s)
    else Left("Following predicates are not hold: " + errs.mkString(","))
```

Refined

```
trait RefinedBase[A]:
  opaque type T <: A = A

  def predicates: Vector[(String, A => Boolean)]

  def apply(s: A): Either[String, T] =
    val errs = predicates.collect { case (descr, pred) if !pred(s) => descr }
    if (errs.isEmpty) Right(s)
    else Left("Following predicates are not hold: " + errs.mkString(","))
```

Refined


```
trait RefinedBase[A]:
  opaque type T <: A = A

  def predicates: Vector[(String, A => Boolean)]

  def apply(s: A): Either[String, T] =
    val errs = predicates.collect { case (descr, pred) if !pred(s) => descr }
    if (errs.isEmpty) Right(s)
    else Left("Following predicates are not hold: " + errs.mkString(","))
```

Refined

```
trait RefinedBase[A]:
  opaque type T <: A = A

  def predicates: Vector[(String, A => Boolean)]

  def apply(s: A): Either[String, T] =
    val errs = predicates.collect { case (descr, pred) if !pred(s) => descr }
    if (errs.isEmpty) Right(s)
    else Left("Following predicates are not hold: " + errs.mkString(","))
```

Refined

```
trait RefinedBase[A]:
  opaque type T <: A = A

  def predicates: Vector[(String, A => Boolean)]

  def apply(s: A): Either[String, T] =
    val errs = predicates.collect { case (descr, pred) if !pred(s) => descr }
    if (errs.isEmpty) Right(s)
    else Left("Following predicates are not hold: " + errs.mkString(","))

trait StringRefined extends RefinedBase[String]:
  def description: String =
    "String that satisfies predicates: " + predicates.map(_._1).mkString(", ")

  given Encoder[T] = Encoder[String].contramap(x => x)
  given Decoder[T] = Decoder[String].emap(apply)
  given LogstageCodec[T] = LogstageCodec[String].contramap(x => x)
  given Schema[T] = summon[Schema[String]].description(description).as
```

Refined

```
type Name = Name.T
object Name extends StringRefined:
  val maxLength = 100
  val predicates = Vector(
    "non empty" -> (_.nonEmpty),
    "trimmed" -> (x => !x.headOption.contains(' ') && !x.lastOption.contains(' ')),
    s"not longer than $maxLength" -> (_.length <= maxLength)
  )

type Email = Email.T
object Email extends StringRefined:
  val regex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+...
  val predicates = Vector(
    "matches email regex" -> (_.matches(regex))
  )
```

Refined

```
object Predicates:
  val nonEmpty: (String, String => Boolean) = ("non empty", _.nonEmpty)
  val trimmed: (String, String => Boolean) =
    ("trimmed", x => !x.headOption.contains(' ') && !x.lastOption.contains(' '))

  def maxLength(n: Int): (String, String => Boolean) =
    (s"not longer than $n", _.length <= n)
  def matches(regex: String): (String, String => Boolean) =
    (s"matches regex $regex", _.matches(regex))
```

Refined

```
object Predicates:
  val nonEmpty: (String, String => Boolean) = ("non empty", _.nonEmpty)
  val trimmed: (String, String => Boolean) =
    ("trimmed", x => !x.headOption.contains(' ') && !x.lastOption.contains(' '))

  def maxLength(n: Int): (String, String => Boolean) =
    (s"not longer than $n", _.length <= n)
  def matches(regex: String): (String, String => Boolean) =
    (s"matches regex $regex", _.matches(regex))
```

```
type Name = Name.T
object Name extends StringRefined:
  val predicates = Vector(nonEmpty, trimmed, maxLength(100))

type Email = Email.T
object Email extends StringRefined:
  val regex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...
  val predicates = Vector(matches(regex))
```

Или

```
trait Regex(regex: String) extends StringRefined:
  override def predicates =
    Vector(s"matches regex $regex" -> (_.matches(regex)))

trait NonEmpty extends StringRefined:
  override def predicates =
    Vector("non empty" -> (_.nonEmpty))

trait Trimmed extends StringRefined:
  override def predicates =
    Vector("trimmed" ->
      (x => !x.headOption.contains(' ') &&
!x.lastOption.contains(' '))
    )

trait MaxLength(length: Int) extends StringRefined:
  override def predicates =
    Vector(
      s"not longer than $length" -> (_.length <= length)
    )
```

Или

```
trait Regex(regex: String) extends StringRefined:
  override def predicates =
    Vector(s"matches regex $regex" -> (_.matches(regex)))

trait NonEmpty extends StringRefined:
  override def predicates =
    Vector("non empty" -> (_.nonEmpty))

trait Trimmed extends StringRefined:
  override def predicates =
    Vector("trimmed" ->
      (x => !x.headOption.contains(' ') &&
!x.lastOption.contains(' '))
    )

trait MaxLength(length: Int) extends StringRefined:
  override def predicates =
    Vector(
      s"not longer than $length" -> (_.length <= length)
    )
```

```
type Name = Name.T
object Name extends NonEmpty, Trimmed, MaxLength(100)

type Email = Email.T
object Email extends Regex(emailRegex)
```

Super

```scala
trait Regex(regex: String) extends StringRefined:
  override def predicates =
    super.predicates.prepended(s"matches regex $regex" -> (_.matches(regex)))

trait NonEmpty extends StringRefined:
  override def predicates =
    super.predicates.prepended(
      "non empty" -> (_.nonEmpty)
    )

trait Trimmed extends StringRefined:
  override def predicates =
    super.predicates.prepended("trimmed" ->
      (x => !x.headOption.contains(' ') && !x.lastOption.contains(' '))
    )

trait MaxLength(length: Int) extends StringRefined:
  override def predicates =
    super.predicates.prepended(s"not longer than $length" -> (_.length <= length))
```

```scala
type Name = Name.T
object Name extends NonEmpty, Trimmed, MaxLength(100)

type Email = Email.T
object Email extends Regex(emailRegex)
```

Что получилось

```
val emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...

type Name = Name.T
object Name extends NonEmpty, Trimmed, MaxLength(100)

type Email = Email.T
object Email extends Regex(emailRegex)

type ChildId = ChildId.T
object ChildId extends UUIDNewType

type AdultId = AdultId.T
object AdultId extends UUIDNewType

case class Adult(id: AdultId, name: Name, email: Option[Email], children: Vector[ChildId])
  derives Encoder.AsObject, Decoder

case class Child(id: ChildId, name: Name)
  derives Encoder.AsObject, Decoder
```

Что получилось

```
val emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...

type Name = Name.T
object Name extends NonEmpty, Trimmed, MaxLength(100)

type Email = Email.T
object Email extends Regex(emailRegex)

type ChildId = ChildId.T
object ChildId extends UUIDNewType

type AdultId = AdultId.T
object AdultId extends UUIDNewType

case class Adult(id: AdultId, name: Name, email: Option[Email], children: Vector[ChildId])
  derives Encoder.AsObject, Decoder

case class Child(id: ChildId, name: Name)
  derives Encoder.AsObject, Decoder
```

Что получилось

```scala
val emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...

type Name = Name.T
object Name extends NonEmpty, Trimmed, MaxLength(100)

type Email = Email.T
object Email extends Regex(emailRegex)

type ChildId = ChildId.T
object ChildId extends UUIDNewType

type AdultId = AdultId.T
object AdultId extends UUIDNewType

case class Adult(id: AdultId, name: Name, email: Option[Email], children: Vector[ChildId])
  derives Encoder.AsObject, Decoder

case class Child(id: ChildId, name: Name)
  derives Encoder.AsObject, Decoder
```

Выносим в конфиг

```
trait HasConfig[Config]:
  def get: Config

trait RegexConfig[Config](key: Config => String)(using conf: HasConfig[Config])
  extends StringRefined:

  lazy val regex = key(conf.get)

  override def predicates: Vector[(String, String => Boolean)] =
    super.predicates.prepended(
      s"matches regex $regex" -> (_.matches(regex))
    )
```

Выносим в конфиг

```scala
trait HasConfig[Config]:
  def get: Config

trait RegexConfig[Config](key: Config => String)(using conf: HasConfig[Config])
  extends StringRefined:

  lazy val regex = key(conf.get)

  override def predicates: Vector[(String, String => Boolean)] =
    super.predicates.prepended(
      s"matches regex $regex" -> (_.matches(regex))
    )


import pureconfig.{ConfigReader, ConfigSource}
case class Config(emailRegex: String, nameMaxLength: Int) derives ConfigReader

given HasConfig[Config] with
  lazy val get: Config = ConfigSource.file("application.conf").loadOrThrow[Config]
```

Выносим в конфиг

```scala
trait HasConfig[Config]:
  def get: Config

trait RegexConfig[Config](key: Config => String)(using conf: HasConfig[Config])
  extends StringRefined:

  lazy val regex = key(conf.get)

  override def predicates: Vector[(String, String => Boolean)] =
    super.predicates.prepended(
      s"matches regex $regex" -> (_.matches(regex))
    )


import pureconfig.{ConfigReader, ConfigSource}
case class Config(emailRegex: String, nameMaxLength: Int) derives ConfigReader

given HasConfig[Config] with
  lazy val get: Config = ConfigSource.file("application.conf").loadOrThrow[Config]
```

Выносим в конфиг

```
trait HasConfig[Config]:
  def get: Config

trait RegexConfig[Config](key: Config => String)(using conf: HasConfig[Config])
  extends StringRefined:

  lazy val regex = key(conf.get)

  override def predicates: Vector[(String, String => Boolean)] =
    super.predicates.prepended(
      s"matches regex $regex" -> (_.matches(regex))
    )


import pureconfig.{ConfigReader, ConfigSource}
case class Config(emailRegex: String, nameMaxLength: Int) derives ConfigReader

given HasConfig[Config] with
  lazy val get: Config = ConfigSource.file("application.conf").loadOrThrow[Config]
```

Что получилось

```
type Name = Name.T
object Name extends NonEmpty, Trimmed, MaxLengthConfig[Config](_.nameMaxLength)

type Email = Email.T
object Email extends RegexConfig[Config](_.emailRegex)

type ChildId = ChildId.T
object ChildId extends UUIDNewType

type AdultId = AdultId.T
object AdultId extends UUIDNewType

case class Adult(id: AdultId, name: Name, email: Option[Email], children: Vector[ChildId])
  derives Encoder.AsObject, Decoder

case class Child(id: ChildId, name: Name)
  derives Encoder.AsObject, Decoder
```

Что получилось

```
type Name = Name.T
object Name extends NonEmpty, Trimmed, MaxLengthConfig[Config](_.nameMaxLength)

type Email = Email.T
object Email extends RegexConfig[Config](_.emailRegex)

type ChildId = ChildId.T
object ChildId extends UUIDNewType

type AdultId = AdultId.T
object AdultId extends UUIDNewType

case class Adult(id: AdultId, name: Name, email: Option[Email], children: Vector[ChildId])
  derives Encoder.AsObject, Decoder

case class Child(id: ChildId, name: Name)
  derives Encoder.AsObject, Decoder
```

Что получилось

```
type Name = Name.T
object Name extends NonEmpty, Trimmed, MaxLengthConfig[Config](_.nameMaxLength)

type Email = Email.T
object Email extends RegexConfig[Config](_.emailRegex)

type ChildId = ChildId.T
object ChildId extends UUIDNewType

type AdultId = AdultId.T
object AdultId extends UUIDNewType

case class Adult(id: AdultId, name: Name, email: Option[Email], children: Vector[ChildId])
  derives Encoder.AsObject, Decoder

case class Child(id: ChildId, name: Name)
  derives Encoder.AsObject, Decoder
```

Iron

```
type Name = String :| (Not[Empty] & Trimmed & MaxLength[100])
```

```
type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...
```

```
type Email = String :| Match[emailRegex]
```

Iron

```
type Name = String :| (Not[Empty] & Trimmed & MaxLength[100])
```

```
type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...
```

```
type Email = String :| Match[emailRegex]
```

Iron

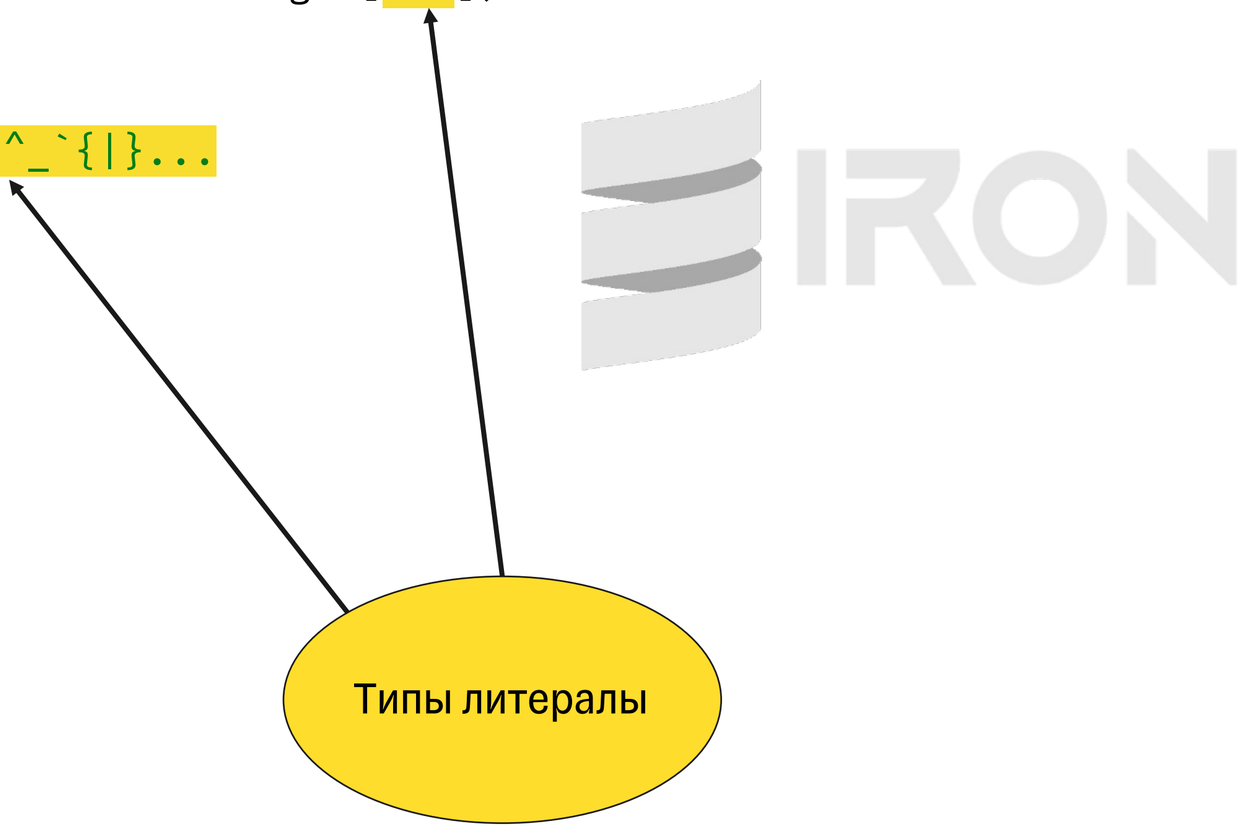
```
type Name = String :| (Not[Empty] & Trimmed & MaxLength[100])
```

```
type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...
```

```
type Email = String :| Match[emailRegex]
```

Iron

```
type Name = String :| (Not[Empty] & Trimmed & MaxLength[100])


type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...

type Email = String :| Match[emailRegex]
```



Iron

```
type Name = String :| (Not[Empty] & Trimmed & MaxLength[100])
```

```
type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...
```

```
type Email = String :| Match[emailRegex]
```

```
opaque type IronType[A, C] <: A = A

type :|[A, C] = IronType[A, C]
```

Iron

```scala
type Name = String :| (Not[Empty] & Trimmed & MaxLength[100])
```

```scala
type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...
type Email = String :| Match[emailRegex]
```

```scala
opaque type IronType[A, C] <: A = A

type :|[A, C] = IronType[A, C]
```

```scala
val bob: Name = "Bob" // ok
val notTrimmed: Name = " Bob " // not compiles

val email: Email = "amtroitskiy@gmail.com" // ok
val notValidEmail: Email = "amtroitskiy@@gmail.com" // not compiles
```

Iron

```scala
type Name = String :| (Not[Empty] & Trimmed & MaxLength[100])
```

```scala
type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...
type Email = String :| Match[emailRegex]
```

```scala
opaque type IronType[A, C] <: A = A

type :|[A, C] = IronType[A, C]
```

```scala
val bob: Name = "Bob" // ok
val notTrimmed: Name = " Bob " // not compiles
```

```scala
val email: Email = "amtroitskiy@gmail.com" // ok
val notValidEmail: Email = "amtroitskiy@@gmail.com" // not compiles
```

```scala
implicit inline def autoRefine[A, C](inline value: A)(
  using inline constraint: Constraint[A, C]
): A :| C =
  macros.assertCondition(value, constraint.test(value), constraint.message)
  IronType(value)
```

Iron

```
type Name = String :| (Not[Empty] & Trimmed & MaxLength[100])


type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...
type Email = String :| Match[emailRegex]


def nameInput: String = "Bob"
val name: Either[String, Name] = nameInput.refineEither
// Right(Bob)


def emailInput: String = "amtroitskiy@@gmail.com"
val email: Either[String, Email] =
  emailInput.refineEither
// Left(Should match (?:[a-z0-9!#$%&'*+/=?^_`{|}...
```

Инстансы

```scala
type Name = String :| (Not[Empty] & Trimmed & MaxLength[100])


import io.github.iltotore.iron.circe.given

summon[Decoder[Name]] // ok
summon[Encoder[Name]] // ok
```

Инстансы

```scala
type Name = Name.T
object Name:
  type C = Not[Empty] & Trimmed & MaxLength[100]
  type R = String :| C
  opaque type T = R // opaque type T >: R <: R = R

  given Encoder[T] = Encoder[String].contramap(x => x)
  given Decoder[T] = Decoder[String].emap(_.refineEither[C])
```

```scala
Encoder[Name] // compiles
Decoder[Name] // compiles
```

Больше инстансов

```
type Name = Name.T
object Name:
  type C = Not[Empty] & Trimmed & MaxLength[100]
  type R = String :| C
  opaque type T = R

  given Encoder[T] = Encoder[String].contramap(x => x)
  given Decoder[T] = Decoder[String].emap(_.refineEither[C])
  given LogstageCodec[T] = LogstageCodec[String].contramap(x => x)
  given Schema[T] = summon[Schema[String]].description(
    summon[RuntimeConstraint[String, C]].message
  ).as
```

Рефакторим

```
trait IronRefinedType[B, C](using r: RuntimeConstraint[B, C]):
  type R = B :| C
  opaque type T = R

  def apply(x: B): Either[String, T] = Either.cond(r.test(x), x.asInstanceOf[T], r.message)

  given (using Encoder[B]): Encoder[T] = Encoder[B].contramap(x => x)
  given (using LogstageCodec[B]): LogstageCodec[T] = LogstageCodec[B].contramap(x => x)
  given (using Decoder[B]): Decoder[T] = Decoder[B].emap(apply)
  given (using s: Schema[B]): Schema[T] = s.description(r.message).as
```

Рефакторим

```
trait IronRefinedType[B, C](using r: RuntimeConstraint[B, C]):
  type R = B :| C
  opaque type T = R

  def apply(x: B): Either[String, T] = Either.cond(r.test(x), x.asInstanceOf[T], r.message)

  given (using Encoder[B]): Encoder[T] = Encoder[B].contramap(x => x)
  given (using LogstageCodec[B]): LogstageCodec[T] = LogstageCodec[B].contramap(x => x)
  given (using Decoder[B]): Decoder[T] = Decoder[B].emap(apply)
  given (using s: Schema[B]): Schema[T] = s.description(r.message).as
```

```
type Name = Name.T
object Name extends IronRefinedType[String, Not[Empty] & Trimmed & MaxLength[100]]

type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...

type Email = Email.T
object Email extends IronRefinedType[String, Match[emailRegex]]
```

Что получилось

```
type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...

type Name = Name.T
object Name extends IronRefinedType[String, Not[Empty] & Trimmed & MaxLength[100]]

type Email = Email.T
object Email extends IronRefinedType[String, Match[emailRegex]]

type ChildId = ChildId.T
object ChildId extends IronRefinedType[UUID, Pure]

type AdultId = ChildId.T
object AdultId extends IronRefinedType[UUID, Pure]

case class Adult(id: AdultId, name: Name, email: Option[Email], children: Vector[ChildId])
  derives Encoder.AsObject, Decoder

case class Child(id: ChildId, name: Name) derives Encoder.AsObject, Decoder
```

Что получилось

```
type emailRegex = "(?:[a-z0-9!#$%&'*+/=?^_`{|}...

type Name = Name.T
object Name extends IronRefinedType[String, Not[Empty] & Trimmed & MaxLength[100]]

type Email = Email.T
object Email extends IronRefinedType[String, Match[emailRegex]]

type ChildId = ChildId.T
object ChildId extends IronRefinedType[UUID, Pure]

type AdultId = ChildId.T
object AdultId extends IronRefinedType[UUID, Pure]

case class Adult(id: AdultId, name: Name, email: Option[Email], children: Vector[ChildId])
  derives Encoder.AsObject, Decoder

case class Child(id: ChildId, name: Name) derives Encoder.AsObject, Decoder
```

А в конфиги можно?

```
type Email = Email.T
object Email extends IronRefinedType[String, RegexConfig["email-regex"]]
```

Config

```
type EmailRegex = "email-regex"
type NameMaxLength = "name-max-length"

type Keys = EmailRegex | NameMaxLength
```

Config

```scala
type EmailRegex = "email-regex"
type NameMaxLength = "name-max-length"

type Keys = EmailRegex | NameMaxLength
```

```scala
case class Config(nameMaxLength: Int, emailRegex: String) derives ConfigReader:
  inline def get[K <: Keys]: KeyMap[K] =
    inline erasedValue[K] match
      case _: EmailRegex => emailRegex
      case _: NameMaxLength => nameMaxLength
```

Config

```scala
type EmailRegex = "email-regex"
type NameMaxLength = "name-max-length"

type Keys = EmailRegex | NameMaxLength
```

```scala
case class Config(nameMaxLength: Int, emailRegex: String) derives ConfigReader:
  inline def get[K <: Keys]: KeyMap[K] =
    inline erasedValue[K] match
      case _: EmailRegex => emailRegex
      case _: NameMaxLength => nameMaxLength
```

Config

```
type EmailRegex = "email-regex"
type NameMaxLength = "name-max-length"

type Keys = EmailRegex | NameMaxLength
```

```
case class Config(nameMaxLength: Int, emailRegex: String) derives ConfigReader:
  inline def get[K <: Keys]: KeyMap[K] =
    inline erasedValue[K] match
      case _: EmailRegex => emailRegex
      case _: NameMaxLength => nameMaxLength
```

Config

```
type EmailRegex = "email-regex"
type NameMaxLength = "name-max-length"

type Keys = EmailRegex | NameMaxLength

type KeyMap[K <: Keys] = K match
  case EmailRegex => String
  case NameMaxLength => Int

case class Config(nameMaxLength: Int, emailRegex: String) derives ConfigReader:
  inline def get[K <: Keys]: KeyMap[K] =
    inline erasedValue[K] match
      case _: EmailRegex => emailRegex
      case _: NameMaxLength => nameMaxLength
```

Custom Constraint

```
final class RegexConfig[S <: Keys]
```

Custom Constraint

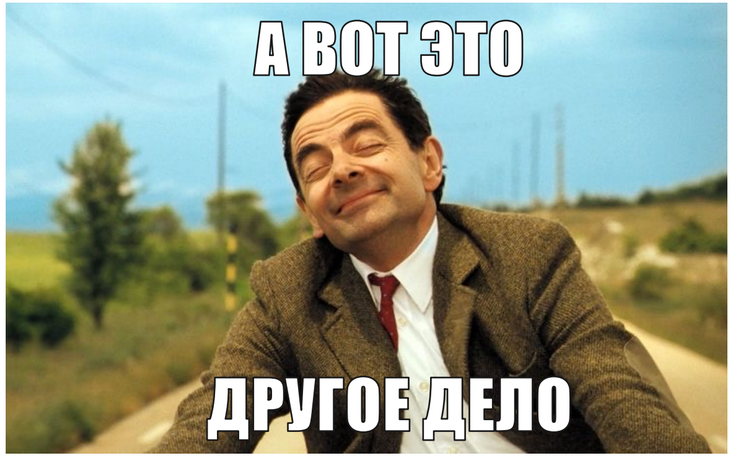
```scala
final class RegexConfig[S <: Keys]


given [K <: Keys](
  using v: ValueOf[K], h: HasConfig[Config]
): Constraint[String, RegexConfig[K]] with

  override inline def test(value: String): Boolean =
    value.matches(summonInline[KeyMap[K] =:= String].apply(h.get.get[K]))

  override inline def message: String = s"Should match regex by key ${v.value}"
```

Custom Constraint

```scala
final class RegexConfig[S <: Keys]


given [K <: Keys](
  using v: ValueOf[K], h: HasConfig[Config]
): Constraint[String, RegexConfig[K]] with

  override inline def test(value: String): Boolean =
    value.matches(summonInline[KeyMap[K] =:= String].apply(h.get.get[K]))

  override inline def message: String = s"Should match regex by key ${v.value}"
```

Что получилось

```
type Name = Name.T
object Name extends IronRefinedType[String, Not[Empty] & Trimmed & MaxLengthConfig[NameMaxLength]]

type Email = Email.T
object Email extends IronRefinedType[String, RegexConfig[EmailRegex]]

type ChildId = ChildId.T
object ChildId extends IronRefinedType[String, Pure]

type AdultId = ChildId.T
object AdultId extends IronRefinedType[String, Pure]

case class Adult(id: AdultId, name: Name, email: Option[Email], children: Vector[ChildId])
  derives Encoder.AsObject, Decoder

case class Child(id: ChildId, name: Name) derives Encoder.AsObject, Decoder
```



Iron

Плюсы:

- compile-time проверки
- OR для констрейнтов через типы пересечения

Минусы:

- Непонятно какой именно предикат нарушается в случае ошибки

Implication

```
type Age = Age.T
object Age extends IronRefinedType[Int, GreaterEqual[0]]

type ChildAge = ChildAge.T
object ChildAge extends IronRefinedType[Int, GreaterEqual[0] & Less[18]]

type AdultAge = AdultAge.T
object AdultAge extends IronRefinedType[Int, GreaterEqual[18]]
```

Implication

```scala
type Age = Age.T
object Age extends IronRefinedType[Int, GreaterEqual[0]]

type ChildAge = ChildAge.T
object ChildAge extends IronRefinedType[Int, GreaterEqual[0] & Less[18]]

type AdultAge = AdultAge.T
object AdultAge extends IronRefinedType[Int, GreaterEqual[18]]
```

```scala
val c: ChildAge = 16

c: Age // compiles
```

Implication

```
type Age = Age.T
object Age extends IronRefinedType[Int, GreaterEqual[0]]

type ChildAge = ChildAge.T
object ChildAge extends IronRefinedType[Int, GreaterEqual[0] & Less[18]]

type AdultAge = AdultAge.T
object AdultAge extends IronRefinedType[Int, GreaterEqual[18]]
```

```
val c: ChildAge = 16

c: Age // compiles

val a: AdultAge = 42

a: Age // not compiles
```

Implication

```scala
type Age = Age.T
object Age extends IronRefinedType[Int, GreaterEqual[0]]

type ChildAge = ChildAge.T
object ChildAge extends IronRefinedType[Int, GreaterEqual[0] & Less[18]]

type AdultAge = AdultAge.T
object AdultAge extends IronRefinedType[Int, GreaterEqual[18]]
```

```scala
val c: ChildAge = 16
summon[Implication[GreaterEqual[0] & Less[18], GreaterEqual[0]]] // compiles
c: Age // compiles

val a: AdultAge = 42
summon[Implication[GreaterEqual[18], GreaterEqual[0]]] // not compiles
a: Age // not compiles
```

Implication

```
type Age = Age.T
object Age extends IronRefinedType[Int, GreaterEqual[0]]

type ChildAge = ChildAge.T
object ChildAge extends IronRefinedType[Int, GreaterEqual[0] & Less[18]]

type AdultAge = AdultAge.T
object AdultAge extends IronRefinedType[Int, GreaterEqual[18]]

given [N <: Int, M <: Int](using (N >= M) =:= true)
  : Implication[GreaterEqual[N], GreaterEqual[M]] = Implication()

val c: ChildAge = 16
summon[Implication[GreaterEqual[0] & Less[18], GreaterEqual[0]]] // compiles
c: Age // compiles

val a: AdultAge = 42
summon[Implication[GreaterEqual[18], GreaterEqual[0]]] // compiles
a: Age // compiles
```

А можно самим такое?

```
type Age = Age.T
object Age:
  opaque type T <: Int = Int
  def apply(s: Int): Either[String, T] =
    Either.cond(s >= 0, s, "is less than 0, not a valid age")

type ChildAge = ChildAge.T
object ChildAge:
  opaque type T <: Age = Age
  def apply(s: Int): Either[String, T] =
    Age(s).flatMap(res => Either.cond(res < 18, res, "is greater or eq than than 18"))

type AdultAge = AdultAge.T
object AdultAge:
  opaque type T <: Age = Age
  def apply(s: Int): Either[String, T] =
    Age(s).flatMap(res => Either.cond(res >= 18, res, "is less than 18"))
```

А можно самим такое?

```
type Age = Age.T
object Age:
  opaque type T <: Int = Int
  def apply(s: Int): Either[String, T] =
    Either.cond(s >= 0, s, "is less than 0, not a valid age")

type ChildAge = ChildAge.T
object ChildAge:
  opaque type T <: Age = Age
  def apply(s: Int): Either[String, T] =
    Age(s).flatMap(res => Either.cond(res < 18, res, "is greater or eq than than 18"))

type AdultAge = AdultAge.T
object AdultAge:
  opaque type T <: Age = Age
  def apply(s: Int): Either[String, T] =
    Age(s).flatMap(res => Either.cond(res >= 18, res, "is less than 18"))
```

А можно самим такое?

```scala
type Age = Age.T
object Age:
  opaque type T <: Int = Int
  def apply(s: Int): Either[String, T] =
    Either.cond(s >= 0, s, "is less than 0, not a valid age")

type ChildAge = ChildAge.T
object ChildAge:
  opaque type T <: Age = Age
  def apply(s: Int): Either[String, T] =
    Age(s).flatMap(res => Either.cond(res < 18, res, "is greater or eq than than 18"))

type AdultAge = AdultAge.T
object AdultAge:
  opaque type T <: Age = Age
  def apply(s: Int): Either[String, T] =
    Age(s).flatMap(res => Either.cond(res >= 18, res, "is less than 18"))
```

```scala
summon[ChildAge <:< Age]
summon[AdultAge <:< Age]
summon[ChildAge <:< Int]
```

А можно самим такое?

```
type Age = Age.T
object Age extends Min[Int](0)

type ChildAge = ChildAge.T
object ChildAge extends Max[Age](17)

type AdultAge = AdultAge.T
object AdultAge extends Min[Age](18)
```

Итоги

- Писать удобные refined типы можно на **Scala 3** руками
- Писать удобные refined типы можно с помощью **iron**

- Если нужны compile-time проверки, то выбирайте **iron**
- Иначе проще написать самостоятельно решение под конкретную доменную область

Итоги

- Писать удобные refined типы можно на **Scala 3** руками
- Писать удобные refined типы можно с помощью **iron**

- Если нужны compile-time проверки, то выбирайте **iron**
- Иначе проще написать самостоятельно решение под конкретную доменную область

Благодарности

- Иван Лягаев
- Владислав Угрюмов
- Евгений Веретенников
- Михаил Чугунков
- Вадим Челышов
- Борис Потепун
- Анастасия Ермолаева
- Петр и Александра Троицкие
- Олег Нижников

https://github.com/road21/talks

Спасибо!