

Extensible Exception

kbkz.tech #10

吉村 優

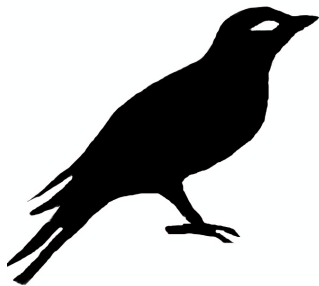
https://twitter.com/_yyu_

<http://qiita.com/yyu>

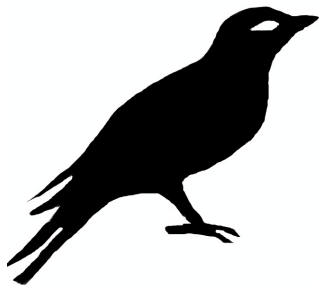
<https://github.com/y-yu>

July 16, 2016

自己紹介

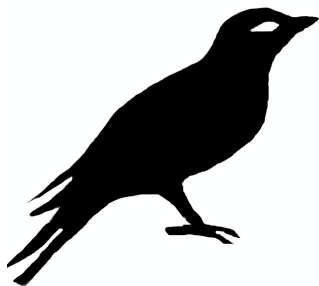


自己紹介



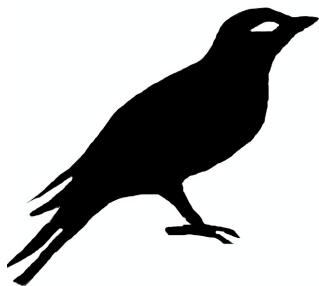
- 筑波大学 情報科学類 学士
(COINS11)

自己紹介



- 筑波大学 情報科学類 学士 (COINS11)
- 現在は Scala を書く仕事に従事

自己紹介



- 筑波大学 情報科学類 学士 (COINS11)
- 現在は Scala を書く仕事に従事
- エラー処理に関する話をします

エラー値とは？

エラー値とは？

エラー値

エラー値とは？

エラー値

- エラーであることを表す値

エラー値とは？

エラー値

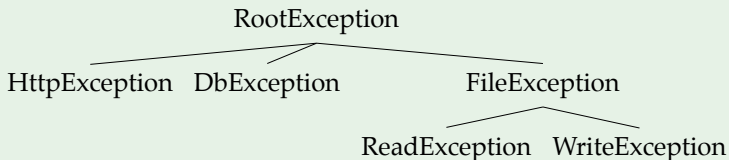
- エラーであることを表す値
- 階層構造（木構造）になるのが一般的

エラー値とは？

エラー値

- エラーであることを表す値
- 階層構造（木構造）になるのが一般的

エラー値の階層構造の例

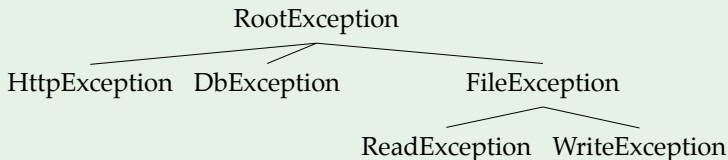


エラー値とは？

エラー値

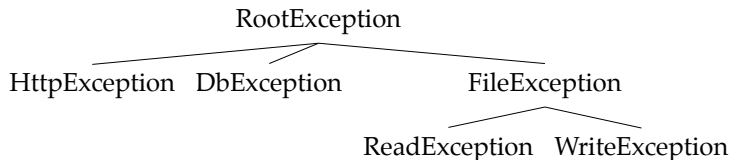
- エラーであることを表す値
- 階層構造（木構造）になるのが一般的

エラー値の階層構造の例

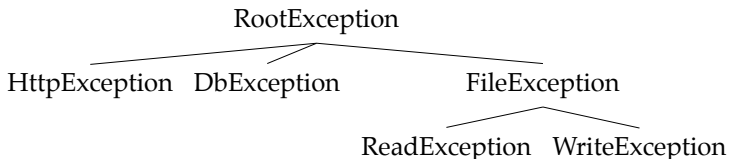


どうやって階層構造を作る？

継承を用いた表現

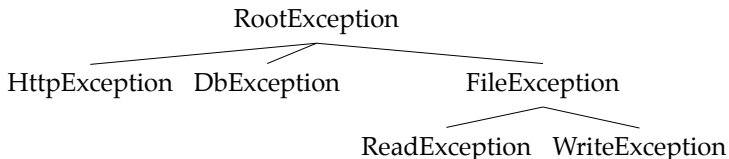


継承を用いた表現



```
trait RootException extends Throwable
case class DbException(m: String) extends RootException
case class HttpException(m: String) extends RootException
trait FileException extends RootException
case class ReadException(m: String) extends FileException
case class WriteException(m: String) extends FileException
```

継承を用いた表現



```
trait RootException extends Throwable
case class DbException(m: String) extends RootException
case class HttpException(m: String) extends RootException
trait FileException extends RootException
case class ReadException(m: String) extends FileException
case class WriteException(m: String) extends FileException
```

どうして継承を使うの？

サブタイプ多相

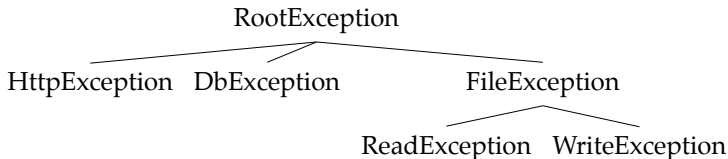
“ 型 A が型 B の *subtype* (部分型) のとき、型 B の式を書くべきところに、型 A の式を書いても良い。 ”

筑波大学 プログラム言語論 [1]

サブタイプ多相

“ 型 A が型 B の *subtype* (部分型) のとき、型 B の式を書くべきところに、型 A の式を書いても良い。 ”

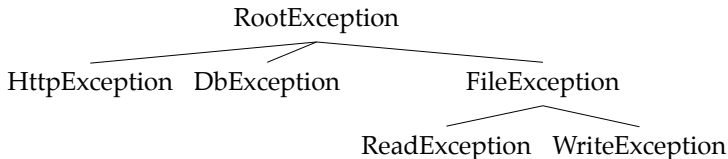
筑波大学 プログラム言語論 [1]



サブタイプ多相

“ 型 A が型 B の *subtype* (部分型) のとき、型 B の式を書くべきところに、型 A の式を書いても良い。 ”

筑波大学 プログラム言語論 [1]

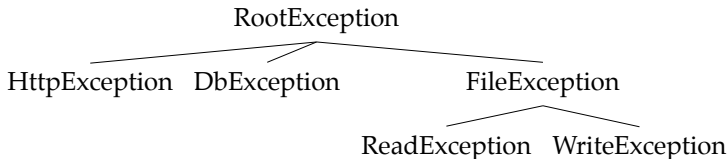


例

サブタイプ多相

“ 型 A が型 B の *subtype* (部分型) のとき、型 B の式を書くべきところに、型 A の式を書いても良い。 ”

筑波大学 プログラム言語論 [1]



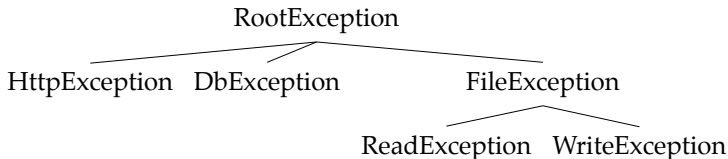
例

- `RootException` を書くべきところに `HttpException` を書く

サブタイプ多相

“ 型 A が型 B の *subtype* (部分型) のとき、型 B の式を書くべきところに、型 A の式を書いても良い。 ”

筑波大学 プログラム言語論 [1]



例

- `RootException` を書くべきところに `HttpException` を書く
- `RootException` を書くべきところに `ReadException` を書く

Either

Either

```
trait Either[+A, +B] {  
  def flatMap[AA >: A, Y](f: B => Either[AA, Y]) =  
    this match {  
      case Left(a)   => Left(a)  
      case Right(b)  => f(b)  
    }  
}  
  
case class Left[+A, +B](a: A) extends Either[A, B]  
case class Right[+A, +B](b: B) extends Either[A, B]
```

Either

```
trait Either[+A, +B] {  
  def flatMap[AA >: A, Y](f: B => Either[AA, Y]) =  
    this match {  
      case Left(a)   => Left(a)  
      case Right(b)  => f(b)  
    }  
}  
  
case class Left [ +A, +B](a: A) extends Either[A, B]  
case class Right[ +A, +B](b: B) extends Either[A, B]
```

- flatMapの型パラメータで $AA >: A$ を取る

Either

```
trait Either[+A, +B] {  
  def flatMap[AA >: A, Y](f: B => Either[AA, Y]) =  
    this match {  
      case Left(a)   => Left(a)  
      case Right(b)  => f(b)  
    }  
}  
case class Left [ +A, +B](a: A) extends Either[A, B]  
case class Right[ +A, +B](b: B) extends Either[A, B]
```

- `flatMap`の型パラメータで `AA >: A` を取る
- `AA >: A` は `AA` が `A` のスーパータイプであることを表す

Either

```
trait Either[+A, +B] {  
  def flatMap[AA >: A, Y](f: B => Either[AA, Y]) =  
    this match {  
      case Left(a)   => Left(a)  
      case Right(b)  => f(b)  
    }  
}  
case class Left[+A, +B](a: A) extends Either[A, B]  
case class Right[+A, +B](b: B) extends Either[A, B]
```

- flatMapの型パラメータで $AA >: A$ を取る
- $AA >: A$ は AA が A のスーパータイプであることを表す

```
// Left[FileNotFoundException]  
for {  
  a <- Left(WriteException("file write error"))  
  b <- Left(ReadException("file read error"))  
} yield ()
```

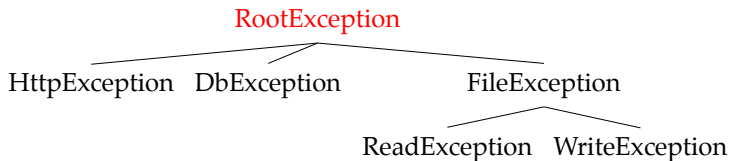

Either

```
for {  
  a <- Left(HttpException("http error"))  
  b <- Left(DbException("db error"))  
} yield ()
```

Either

```
for {  
  a <- Left(HttpException("http error"))  
  b <- Left(DbException("db error"))  
} yield ()
```

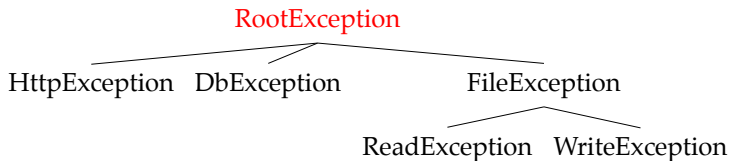
この型は `Left[RootException]` になる



Either

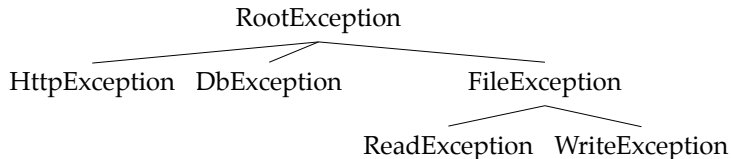
```
for {  
  a <- Left(HttpException("http error"))  
  b <- Left(DbException("db error"))  
} yield ()
```

この型は `Left[RootException]` になる

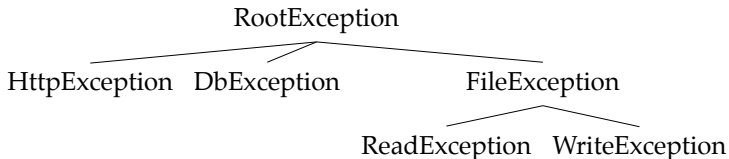


型の上では `FileException` と区別できなくなった！

階層構造の拡張

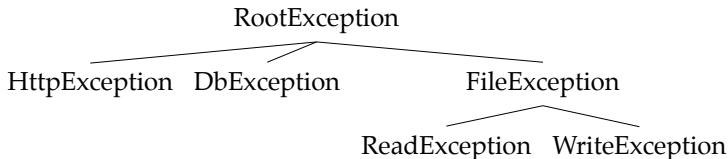


階層構造の拡張

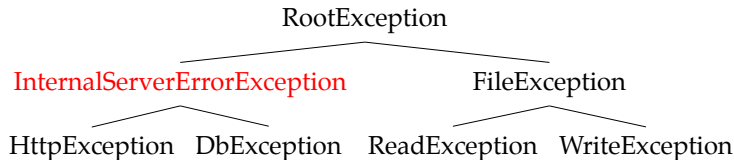


後からこの階層構造を変更できる？

階層構造の拡張



後からこの階層構造を変更できる？



無理では？

無理では？

継承でやるのはよくない？

無理では？

継承でやるのはよくない？

型クラスでやろう！



階層構造の拡張

- ① 新しい型クラス `:~>` (Transform) を導入

階層構造の拡張

- ① 新しい型クラス `:~>` (Transform) を導入
- ② 新しいエラー値を定義

階層構造の拡張

- ① 新しい型クラス `:~>` (Transform) を導入
- ② 新しいエラー値を定義
- ③ 型クラス `:~>` のインスタンスを定義

階層構造の拡張

- ① 新しい型クラス `:~>` (Transform) を導入
- ② 新しいエラー値を定義
- ③ 型クラス `:~>` のインスタンスを定義
- ④ 型クラス `:~>` のインスタンスを使うように `Either` を拡張

新しい型クラス:~>の定義

新しい型クラス:~>の定義

変換を表す型クラス:~>

型 A から型 B への変換ができることを表す型クラス

新しい型クラス:~>の定義

変換を表す型クラス:~>

型 A から型 B への変換ができることを表す型クラス

```
trait ~>[-A, +B] {  
  def apply(a: A): B  
}
```


新しい型クラス:~>の定義

変換を表す型クラス:~>

型 A から型 B への変換ができることを表す型クラス

```
trait ~>[-A, +B] {  
  def apply(a: A): B  
}
```

例

DbException から RootException への変換ができることを表す

```
implicit val db = new (DbException ~> RootException) {  
  def apply(a: DbException): RootException =  
    new RootException { ??? }  
}
```

新しいエラー値の定義

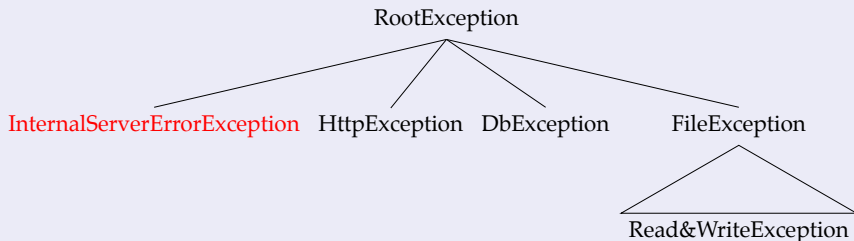
新しいエラー値の定義

```
case class InternalServerErrorException(m: String)
  extends RootException
```

新しいエラー値の定義

```
case class InternalServerErrorException(m: String)
  extends RootException
```

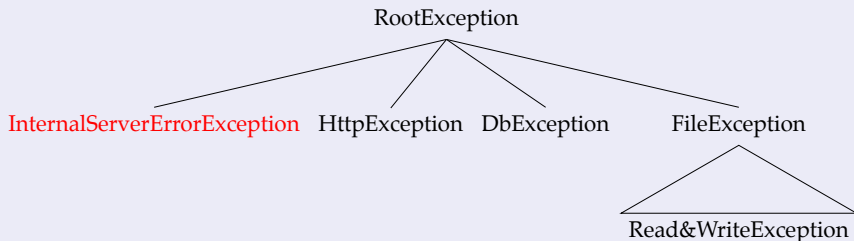
継承に基づく階層構造



新しいエラー値の定義

```
case class InternalServerErrorException(m: String)
  extends RootException
```

継承に基づく階層構造



- この時点で、HttpExceptionとDbExceptionはInternalServerErrorExceptionと関係がない

型クラス:~>のインスタンス定義

型クラス:~>のインスタンス定義

```
object InternalServerErrorException {  
  implicit val databaseException =  
    new (DbException :~> InternalServerErrorException) {  
      def apply(a: DbException): InternalServerErrorException =  
        InternalServerErrorException(s"database: ${a.m}")  
    }  
  
  implicit val httpException =  
    new (HttpException :~> InternalServerErrorException) {  
      def apply(a: HttpException): InternalServerErrorException =  
        InternalServerErrorException(s"http: ${a.m}")  
    }  
}
```

型クラス:~>のインスタンス定義

```
object InternalServerErrorException {  
  implicit val databaseException =  
    new (DbException :~> InternalServerErrorException) {  
      def apply(a: DbException): InternalServerErrorException =  
        InternalServerErrorException(s"database: ${a.m}")  
    }  
  
  implicit val httpException =  
    new (HttpException :~> InternalServerErrorException) {  
      def apply(a: HttpException): InternalServerErrorException =  
        InternalServerErrorException(s"http: ${a.m}")  
    }  
}
```

次のインスタンスを定義する

型クラス:~>のインスタンス定義

```
object InternalServerErrorException {  
  implicit val databaseException =  
    new (DbException :~> InternalServerErrorException) {  
      def apply(a: DbException): InternalServerErrorException =  
        InternalServerErrorException(s"database: ${a.m}")  
    }  
  
  implicit val httpException =  
    new (HttpException :~> InternalServerErrorException) {  
      def apply(a: HttpException): InternalServerErrorException =  
        InternalServerErrorException(s"http: ${a.m}")  
    }  
}
```

次のインスタンスを定義する

- DbException から InternalServerErrorException への変換

型クラス:~>のインスタンス定義

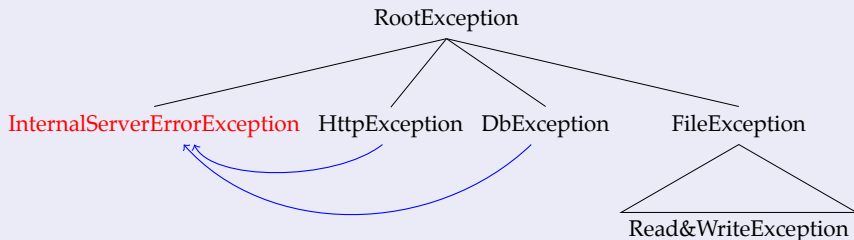
```
object InternalServerErrorException {  
  implicit val databaseException =  
    new (DbException :~> InternalServerErrorException) {  
      def apply(a: DbException): InternalServerErrorException =  
        InternalServerErrorException(s"database: ${a.m}")  
    }  
  
  implicit val httpException =  
    new (HttpException :~> InternalServerErrorException) {  
      def apply(a: HttpException): InternalServerErrorException =  
        InternalServerErrorException(s"http: ${a.m}")  
    }  
}
```

次のインスタンスを定義する

- DbException から InternalServerErrorException への変換
- HttpException から InternalServerErrorException への変換

型クラス:~>のインスタンス定義

継承に基づく階層構造とインスタンス



Either の拡張

既存の Either

```
trait Either[+A, +B] {  
  def flatMap[AA >: A, Y](f: B => Either[AA, Y]) =  
    this match {  
      case Left(a)  => Left(a)  
      case Right(b) => f(b)  
    }  
}  
case class Left [A, B](a: A) extends Either[A, B]  
case class Right[+A, +B](b: B) extends Either[A, B]
```

Either の拡張

既存の Either

```
trait Either[+A, +B] {  
  def flatMap[AA >: A, Y](f: B => Either[AA, Y]) =  
    this match {  
      case Left(a)  => Left(a)  
      case Right(b) => f(b)  
    }  
}  
case class Left[+A, +B](a: A) extends Either[A, B]  
case class Right[+A, +B](b: B) extends Either[A, B]
```

Pimp my Library パターンで Either を拡張

```
implicit class ExceptionEither[L1, R1](val ee: Either[L1, R1]) {  
  ???  
}
```

Either の拡張

既存の Either

```
trait Either[+A, +B] {  
  def flatMap[AA >: A, Y](f: B => Either[AA, Y]) =  
    this match {  
      case Left(a)   => Left(a)  
      case Right(b)  => f(b)  
    }  
}  
case class Left[+A, +B](a: A) extends Either[A, B]  
case class Right[+A, +B](b: B) extends Either[A, B]
```

Pimp my Library パターンで Either を拡張

```
implicit class ExceptionEither[L1, R1](val ee: Either[L1, R1]) {  
  ???  
}
```

- map と flatMap を拡張

Either の拡張

既存の Either

```
trait Either[+A, +B] {  
  def flatMap[AA >: A, Y](f: B => Either[AA, Y]) =  
    this match {  
      case Left(a)   => Left(a)  
      case Right(b)  => f(b)  
    }  
}  
case class Left[+A, +B](a: A) extends Either[A, B]  
case class Right[+A, +B](b: B) extends Either[A, B]
```

Pimp my Library パターンで Either を拡張

```
implicit class ExceptionEither[L1, R1](val ee: Either[L1, R1]) {  
  ???  
}
```

- map と flatMap を拡張
- 新しいメソッド as を導入

mapとflatMapの拡張

mapとflatMapの拡張

```
def map[L2, R2](f: R1 => R2)
    (implicit F: L1 :~> L2): Either[L2, R2] =
  ee match {
    case Left(e)  => Left(F(e))
    case Right(v) => Right(f(v))
  }

def flatMap[L2, R2](f: R1 => Either[L2, R2])
    (implicit F: L1 :~> L2): Either[L2, R2] =
  ee match {
    case Left(e)  => Left(F(e))
    case Right(v) => f(v)
  }
```

mapとflatMapの拡張

```
def map[L2, R2](f: R1 => R2)
    (implicit F: L1 :~> L2): Either[L2, R2] =
  ee match {
    case Left(e)  => Left(F(e))
    case Right(v) => Right(f(v))
  }

def flatMap[L2, R2](f: R1 => Either[L2, R2])
    (implicit F: L1 :~> L2): Either[L2, R2] =
  ee match {
    case Left(e)  => Left(F(e))
    case Right(v) => f(v)
  }
```

- 型クラス`:~>`のインスタンスを `implicit` パラメータで検索

mapとflatMapの拡張

```
def map[L2, R2](f: R1 => R2)
    (implicit F: L1 :~> L2): Either[L2, R2] =
  ee match {
    case Left(e)   => Left(F(e))
    case Right(v)  => Right(f(v))
  }

def flatMap[L2, R2](f: R1 => Either[L2, R2])
    (implicit F: L1 :~> L2): Either[L2, R2] =
  ee match {
    case Left(e)   => Left(F(e))
    case Right(v)  => f(v)
  }
```

- 型クラス: $\sim>$ のインスタンスをimplicitパラメータで検索
- Leftの場合、型クラス: $\sim>$ のインスタンスを用いて変換

mapとflatMapの拡張

```
def map[L2, R2](f: R1 => R2)
    (implicit F: L1 :~> L2): Either[L2, R2] =
  ee match {
    case Left(e)  => Left(F(e))
    case Right(v) => Right(f(v))
  }

def flatMap[L2, R2](f: R1 => Either[L2, R2])
    (implicit F: L1 :~> L2): Either[L2, R2] =
  ee match {
    case Left(e)  => Left(F(e))
    case Right(v) => f(v)
  }
```

- 型クラス: $\sim>$ のインスタンスを implicit パラメータで検索
- Left の場合、型クラス: $\sim>$ のインスタンスを用いて変換
- Right の場合は元の Either と同様

新しいメソッド `as` の導入

新しいメソッド `as` の導入

```
def as[L2](implicit F: L1 :~> L2): Either[L2, R1] =  
  ee match {  
    case Left(e)  => Left(F(e))  
    case Right(v) => Right(v)  
  }
```

新しいメソッド `as` の導入

```
def as[L2](implicit F: L1 :~> L2): Either[L2, R1] =  
  ee match {  
    case Left(e)  => Left(F(e))  
    case Right(v) => Right(v)  
  }
```

- 型クラス `:~>` のインスタンスを `implicit` パラメータで検索

新しいメソッド `as` の導入

```
def as[L2](implicit F: L1 => L2): Either[L2, R1] =  
  ee match {  
    case Left(e)  => Left(F(e))  
    case Right(v) => Right(v)  
  }
```

- 型クラス `:=>` のインスタンスを `implicit` パラメータで検索
- 型クラス `:=>` のインスタンスを用いて変換

新しいメソッド `as` の導入

```
def as[L2](implicit F: L1 :~> L2): Either[L2, R1] =  
  ee match {  
    case Left(e)  => Left(F(e))  
    case Right(v) => Right(v)  
  }
```

- 型クラス `:~>` のインスタンスを `implicit` パラメータで検索
- 型クラス `:~>` のインスタンスを用いて変換

これの何が便利なの？

新しいメソッド `as` の導入

```
def as[L2](implicit F: L1 => L2): Either[L2, R1] =  
  ee match {  
    case Left(e) => Left(F(e))  
    case Right(v) => Right(v)  
  }
```

- 型クラス `:=>` のインスタンスを `implicit` パラメータで検索
- 型クラス `:=>` のインスタンスを用いて変換

これの何が便利なの？

後で使います

例

```
import InternalServerErrorException._  
val e1 = Left(DbException("db error"))  
val e2 = Left(HttpException("http error"))  
  
// Left[InternalServerErrorException]  
for {  
  a <- e1  
  b <- e2.as[InternalServerErrorException]  
} yield ()
```

既存の階層構造との互換性

既存の階層構造との互換性

自らへのインスタンスがない

```
val e1 = Left(DbException("db error"))

// compile time error!
for {
  a <- e1
} yield ()
```

既存の階層構造との互換性

自らへのインスタンスがない

```
val e1 = Left(DbException("db error"))

// compile time error!
for {
  a <- e1
} yield ()
```

継承関係との互換性がない

```
val e3 = Left(WriteException("file write error"))
val e4 = Left(ReadException("file read error"))

// compile time error!
for {
  a <- e3
  b <- e4
} yield ()
```

自明なインスタンスの導入

自明なインスタンスの導入

self（自らへのインスタンス）

```
implicit def self[A] = new (A :=> A) {  
  def apply(a: A): A = a  
}
```


自明なインスタンスの導入

self (自らへのインスタンス)

```
implicit def self[A] = new (A :~> A) {  
  def apply(a: A): A = a  
}
```

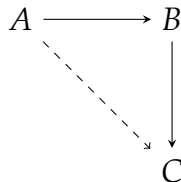
superclass (スーパータイプへのインスタンス)

```
implicit def superclass[A, B >: A] = new (A :~> B) {  
  def apply(a: A): B = a  
}
```

Transitive

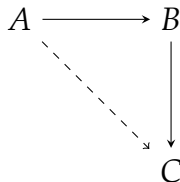
Transitive

$A : \sim > B$ と $B : \sim > C$ から $A : \sim > C$ というインスタンスを生成



Transitive

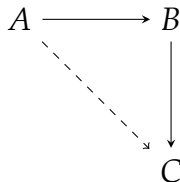
$A : \sim > B$ と $B : \sim > C$ から $A : \sim > C$ というインスタンスを生成



```
implicit def transitive[A, B, C]
  (implicit F: A :~> B, G: B :~> C): A :~> C = new (A :~> C) {
    def apply(a: A): C = G(F(a))
  }
```

Transitive

A :~> Bと B :~> Cから A :~> Cというインスタンスを生成

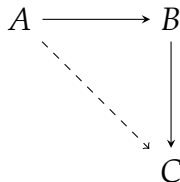


```
implicit def transitive[A, B, C]
  (implicit F: A :~> B, G: B :~> C): A :~> C = new (A :~> C) {
    def apply(a: A): C = G(F(a))
  }
```

これで動くの？

Transitive

A :~> B と B :~> C から A :~> C というインスタンスを生成



```
implicit def transitive[A, B, C]
  (implicit F: A :~> B, G: B :~> C): A :~> C = new (A :~> C) {
    def apply(a: A): C = G(F(a))
  }
```

これで動くの？

diverging implicit expansion for type :~>[HttpException,B]

ワンステップの変換を表す型クラスの定義

ワンステップの変換を表す型クラスの定義

: ->

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


ワンステップの変換を表す型クラスの定義

: ->

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

- 型クラス `:->` (`Transform1`) は推移を含まないワンステップの変換を表す

ワンステップの変換を表す型クラスの定義

`: ->`

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

- 型クラス `: ->` (Transform1) は推移を含まないワンステップの変換を表す
- 型クラス `: ->` のインターフェースは `: ~>` と全く同じ

:~>のインスタンスを:->に変更

:~>のインスタンスを:->に変更

Before

```
object InternalServerErrorException {  
  implicit val databaseException =  
    new (DbException :~> InternalServerErrorException) {  
      def apply(a: DbException): InternalServerErrorException =  
        InternalServerErrorException(s"database: ${a.m}")  
    }  
  
  implicit val httpException =  
    new (HttpException :~> InternalServerErrorException) {  
      def apply(a: HttpException): InternalServerErrorException =  
        InternalServerErrorException(s"http: ${a.m}")  
    }  
}
```

:~>のインスタンスを:->に変更

After

```
object InternalServerErrorException {  
  implicit val databaseException =  
    new (DbException :-> InternalServerErrorException) {  
      def apply(a: DbException): InternalServerErrorException =  
        InternalServerErrorException(s"database: ${a.m}")  
    }  
  
  implicit val httpException =  
    new (HttpException :-> InternalServerErrorException) {  
      def apply(a: HttpException): InternalServerErrorException =  
        InternalServerErrorException(s"http: ${a.m}")  
    }  
}
```

Transitive の定義

Transitive の定義

Transitive

```
implicit def transitive[A, B, C]
  (implicit F: A :-> B, G: B :~> C): A :~> C = new (A :~> C) {
    def apply(a: A): C = G(F(a))
  }
```

Transitive の定義


Transitive

```
implicit def transitive[A, B, C]
  (implicit F: A :-> B, G: B :~> C): A :~> C = new (A :~> C) {
    def apply(a: A): C = G(F(a))
  }
```

- ワンステップの変換 $:->$ を使って発散を防止

参考文献

 亀山幸義.
プログラム言語論 オブジェクト指向, 2015.

 Simon Marlow.
An extensible dynamically-typed hierarchy of exceptions.
In Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell,
Haskell '06, pp. 96–106, New York, NY, USA, 2006. ACM.

目次

- 1 自己紹介
- 2 エラー値とは？
- 3 サブタイピングと Either
- 4 階層構造の拡張
- 5 新しい型クラスと Either の拡張
- 6 自明なインスタンスの導入
- 7 Transitive

Thank you for listening!
Any question?