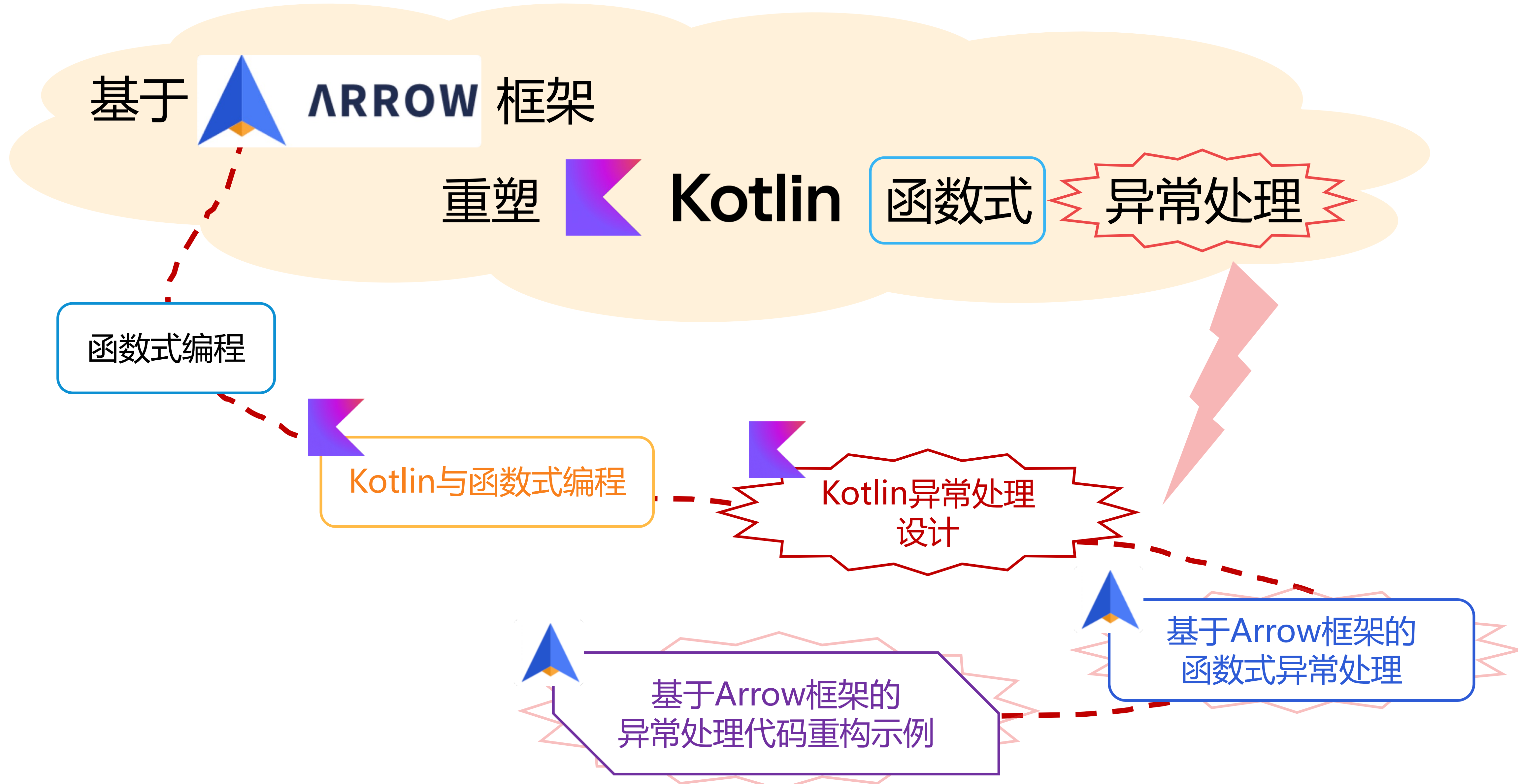
基于Arrow框架重塑Kotlin函数式异常处理

2024/06/20

Xiaochun Han

Rakuten China Development Center Co., Ltd.
Rakuten Group, Inc.

本期分享路经



函数式编程

主流编程范式之一，它的主要特征包括：

- 函数是“头等公民” (First-class citizens)
  - 高阶函数 (Higher-Order Functions)
  - Lambda 表达式
- 纯函数 (Pure Functions)
  - 没有副作用 (No Side Effect)
- 不可变数据 (Immutability)
- 函数组合 (Function composition)

函数式编程的所带来的好处：

- 无副作用的处理执行 -> 安全性
- 更好的并发适应性和可扩展性
- 更好的可读性和可测试性

https://en.wikipedia.org/wiki/Functional_programming
(引用时间: 2024-06-20)

Traditional Imperative Loop:

```
const numList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let result = 0;
for (let i = 0; i < numList.length; i++) {
  if (numList[i] % 2 === 0) {
    result += numList[i] * 10;
  }
}
```

Functional Programming with higher-order functions:

```
const result = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
               .filter(n => n % 2 === 0)
               .map(a => a * 10)
               .reduce((a, b) => a + b, 0);
```

Kotlin与函数式编程

Kotlin语言原生地支持函数式编程。

- ✓ 在Kotlin中，函数是 “头等公民”
  - 函数可以作为变量类型、函数参数和返回值
  - ✓ 支持 “高阶函数” 和 Lambda表达式
- ✓ Kotlin编程习惯中，推荐使用 “不可变数据”
  - 声明局部变量尽可能地选择val，而不是var
- ✓ Kotlin支持inline函数，可以减少函数调用的栈开销
- ✓ Kotlin scope函数(let, run, with, apply, also)可以帮助构建链式调用

不可变数据

```
val user = User(1, ”aaa”)
val newUser = user.copy(name = ”bbb”)
```

函数类型

```
(Request) -> Response
```

作为变量类型

```
val httpHandler: (Request) -> Response
```

作为函数参数

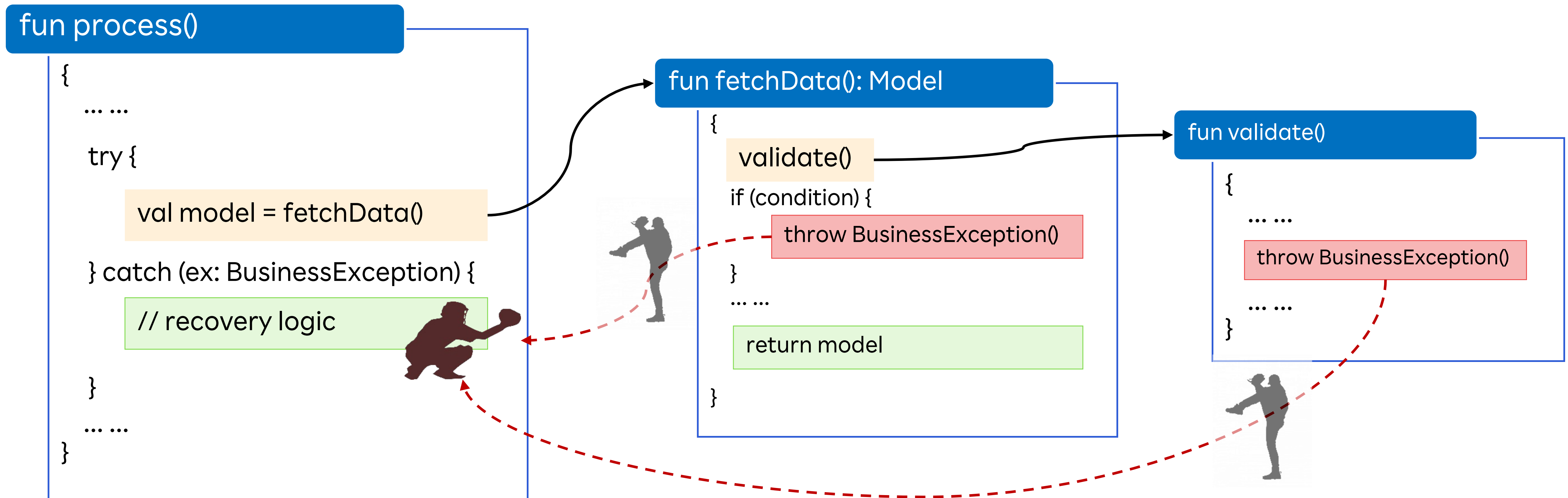
```
fun filter(httpHandler: (Request) -> Response)
```

作为函数返回

```
fun selectHandler(): (Request) -> Response
```

Kotlin异常处理策略 (1 / 3) – 基于Exception的逻辑控制

- 沿袭于Java风格的异常处理方式
- 基于Exception的控制流程往往被认为是bad practice
- 由于Kotlin没有提供checked exceptions，难以从函数签名中快速获悉可能抛出的异常，这增加了理解代码的难度

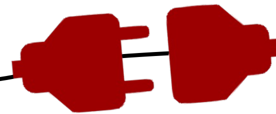


Kotlin异常处理策略 (2 / 3) – 自定义的结果类型封装

- 利用sealed class对函数处理的各种结果（成功 or 失败）进行封装
- 通过显式得定义错误类型，提供编译时的类型安全保证
- 美中不足之处在于，成功和失败的处理通道因此混杂在了一起

```
sealed class FetchResult {
    data class Success(val model: Model) : FetchResult()
    data object NotFound : FetchResult()
    data object InvalidData : FetchResult()
    … …
}
```

```
fun process()
{
  … …
  val result = fetchData()

  when (result) {
    is FetchResult.Success -> ...
    is FetchResult.NotFound -> ...
    is FetchResult.InvalidData -> ...
  }

  … …
}
```



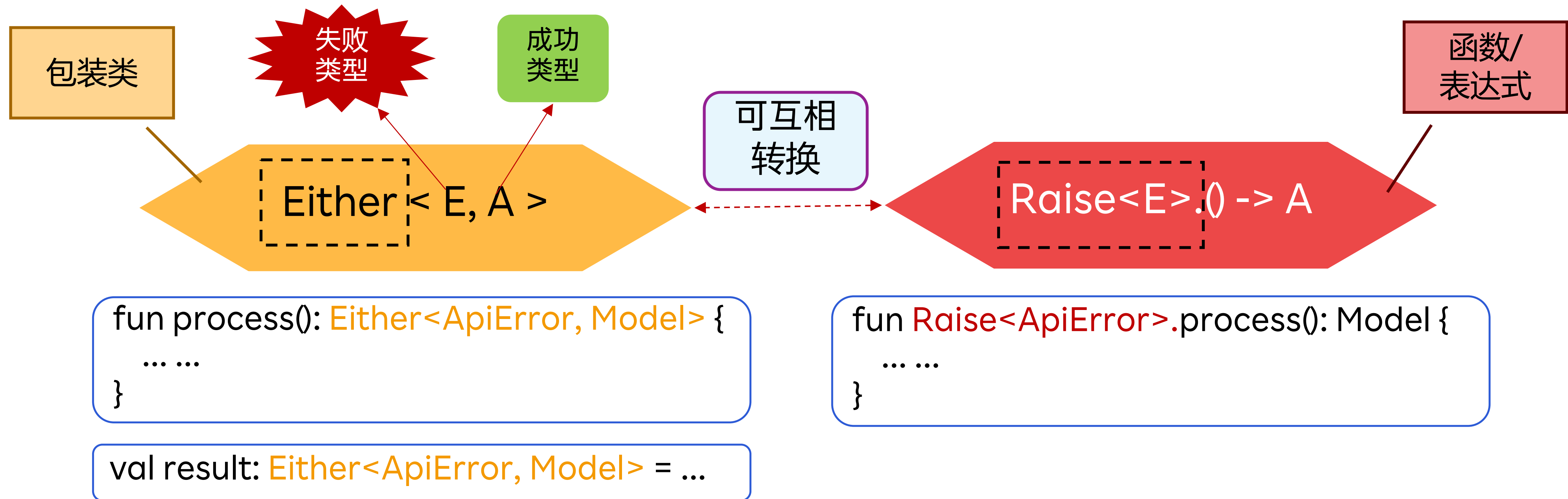
```
fun fetchData(): FetchResult
{
  if (condition) {
    return FetchResult.NotFound
  }

  … …

  return FetchResult.Success(model)
}
```

Kotlin异常处理策略 (3 / 3) – 基于Arrow框架的结构化异常类型

Arrow框架是一个基于Kotlin的工具库，它提供了两种不同风格的结构化异常类型（Typed Errors）的表现方式

- 表现方式 #1：通过引入Either<E, A>类型，将“成功” or “业务逻辑失败”封装在一个结构体中
- 表现方式 #2：通过为函数添加Raise<E>的前缀 (receiver)，标记该函数可能会引发的“业务逻辑失败”

通过函数签名，可以获悉和感知各种成功和失败的可能性，并提供编译时检查，增强代码的可读性和可维护性



Arrow框架结构化异常处理 – Either类型的本质

Either<E, A>

- 一个包装了成功和失败类型的sealed class
  - Left data class代表失败类型
  - Right data class代表成功类型
  - 灵感来自于Scala的Either类型

```kotlin
public sealed class Either<out A, out B> {

    // The left side of the disjoint union, as opposed to the Right side.
    public data class Left<out A> constructor(val value: A) : Either<A, Nothing>() {
        override val isLeft = true
        override val isRight = false

        override fun toString(): String = "Either.Left($value)"

        public companion object {...}
    }

    // The right side of the disjoint union, as opposed to the Left side.
    public data class Right<out B> constructor(val value: B) : Either<Nothing, B>() {
        override val isLeft = false
        override val isRight = true

        override fun toString(): String = "Either.Right($value)"

        public companion object {...}
    }
```

Arrow框架结构化异常处理 – 如何创建和使用Either对象实例

Either<E, A>

1) 可以用 left() 函数包装"失败"的实例

```
data class User(val id: Long)
object UserNotFound
```

```
val error: Either<UserNotFound, User> = UserNotFound.left()
```

2) 可以用 right() 函数包装"成功"的实例

```
val user: Either<UserNotFound, User> = User(id: 1).right()
```

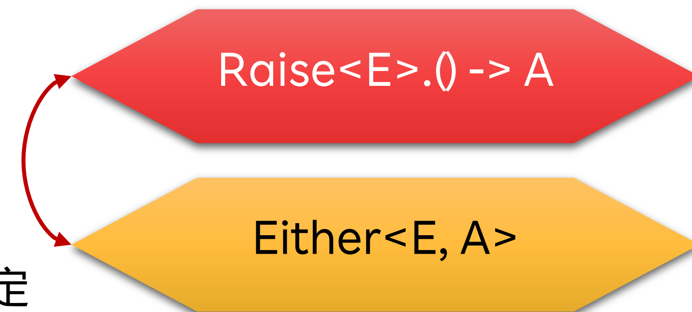
3) 通过类型比较，可以判断Either对象到底是代表了失败？还是代表了成功？

```
println(error is Either.Left)
println(user is Either.Right)
```

4) 可以用fold()函数进行成功和失败结果转换

```
val maybeUser = listOf(error, user).random()
println(maybeUser.fold(
    { HttpStatus.NOT_FOUND to User(id: 0) },
    { HttpStatus.OK to it }
))
```

Arrow框架结构化异常处理 – Either风格 v.s. Raise风格



- Either风格代码的痛点
  - 代码中充斥着 left(), right() 装箱操作，以及Left代表失败，Right代表成功的隐式约定
- Raise风格提供了更加简洁、更易于使用的表现方式

- 可以使用 **either()** 函数作为桥梁，将两种代码风格串联起来

```
fun <E, A> either(block: Raise<E>.() → A): Either<E, A>
```

```
val result: Either<ApiError, Model> = either {
    process()
}
```

```
fun Raise<ApiError>.process(): Model {
  … …
}
```

Arrow框架结构化异常处理 – 如何编写Raise风格的函数

Raise<E>.() -> A

1) 按原有的方式去编写"正常" / "成功"的处理

```
fun Raise<UserNotFound>.user(): User = User( id: 1)
```

<对比> Either风格，需要额外的装箱操作:

```
val user: Either<UserNotFound, User> = User( id: 1).right()
```

2) 可以使用 **raise()** 函数抛出"业务逻辑失败"

```
fun Raise<UserNotFound>.error(): User = raise(UserNotFound)
```

<对比> Either风格，需要额外的装箱操作:

```
val error: Either<UserNotFound, User> = UserNotFound.left()
```

3) 如果要把 Either对象 带入到Raise风格的处理中时，需要使用 **bind()** 函数绑定

```
val maybeUser: Either<UserNotFound, User> = listOf(error, user).random()
fun Raise<UserNotFound>.process(): User = maybeUser.bind()
```

Arrow框架结构化异常处理 – Raise风格的内在实现

Raise<E>.() -> A

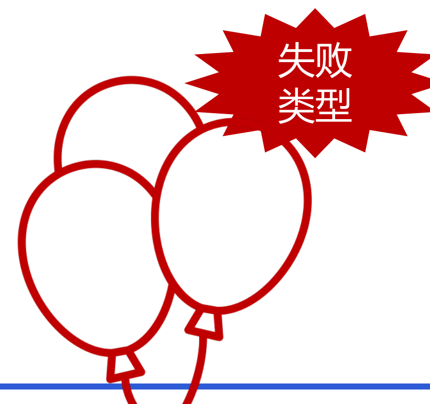
- Raise<E>是一个Arrow框架提供的接口，它包含一个**raise()** 方法，用来抛出“业务逻辑失败”

```
public interface Raise<in Error> {
    @RaiseDSL
    public fun raise(r: Error): Nothing
```

- raise()方法的默认实现，实质上是通过throw Exception的方式，将“失败”冒泡

```
internal class DefaultRaise(@PublishedApi internal val isTraced: Boolean) : Raise<Any?> {
    override fun raise(r: Any?): Nothing = when {
        isActive.value -> throw if (isTraced) Traced(r, raise: this) else NoTrace(r, raise: this)
        else -> throw RaiseLeakedException()
    }
}
```

- 把函数标记为扩展Raise<E>接口之后（也就是采用Raise风格）
  - 该函数的接口将明确展示 “可能抛出的错误”
  - 可以在该函数内调用raise()方法（和其他Raise DSL方法）



```
fun Raise<Error>.process(): Model {
        … …
        raise(Error)
}
```

Arrow框架结构化异常处理 – Raise Scope

Raise<E>.() -> A

- 在Raise风格函数体内，可以调用Raise DSL方法，比如raise()
  - 我们可以称Raise风格函数体的内部为**Raise Scope**
- 类似于Kotlin Coroutine的设计，Arrow框架也提供了几种不同的函数，用来开启Raise Scope
  - **either()** builder函数，或者更通用的 **fold()** 函数都可以开启 **Raise Scope**
- 在Raise Scope中，可以进一步调用其他Raise风格的函数

Kotlin Coroutine

```
fun mainFun() = runBlocking {
    launch { process() }
}
```

Coroutine Scope

```
suspend fun process() {
    delay(1000L)
    subProcess()
}
```

```
suspend fun subProcess() { … }
```

Arrow Raise

```
fun mainFunc() = either {
    process()
}
```

Raise Scope

```
fun Raise<Error>.process(): Model {
    if (failed) raise(Error)
    subProcess()
}
```

```
fun Raise<Error>.subProcess(): Model { … }
```

Arrow框架结构化异常处理 – fold()它到底做了什么?

Raise<E>.() -> A

Fold.kt

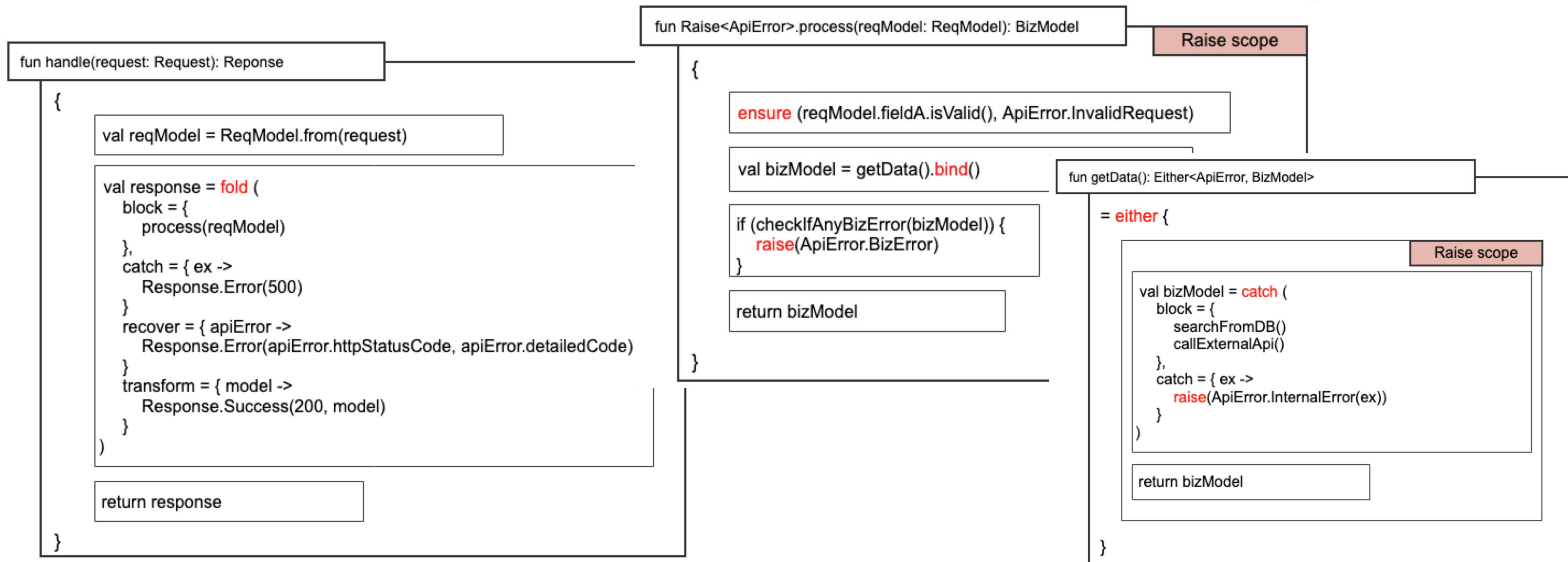
```kotlin
@OptIn(DelicateRaiseApi::class)
@JvmName( name: "_fold")
public inline fun <Error, A, B> fold(
    @BuilderInference block: Raise<Error>.() -> A,
    catch: (throwable: Throwable) -> B,
    recover: (error: Error) -> B,
    transform: (value: A) -> B,
): B {
    contract {
        callsInPlace(catch, AT_MOST_ONCE)
        callsInPlace(recover, AT_MOST_ONCE)
        callsInPlace(transform, AT_MOST_ONCE)
    }
    val raise = DefaultRaise( isTraced: false)
    return try {
        val res = block(raise)
        raise.complete()
        transform(res)
    } catch (e: RaiseCancellationException) {
        raise.complete()
        recover(e.raisedOrRethrow(raise))
    } catch (e: Throwable) {
        raise.complete()
        catch(e.nonFatalOrThrow())
    }
}
```

- fold() 函数可以用来开启 Raise Scope
  - either() 等builder函数，实际上内部实现也依赖于 fold() 函数
- fold() 函数的内部实现采用try-catch block（详见左图）
  - 通过block函数参数(#1)，执行Raise风格的函数或Lambda表达式
  - 通过catch函数参数(#2)，捕捉真正的非预期异常并恢复
  - 通过recover函数参数(#3)，从业务逻辑失败中恢复
  - 通过transform函数参数(#4)，进行成功结果的类型转换 (A -> B)
- recover, catch, transform 都有着共同的目标!
  - 将最终结果转换为一个新的返回类型 -> B

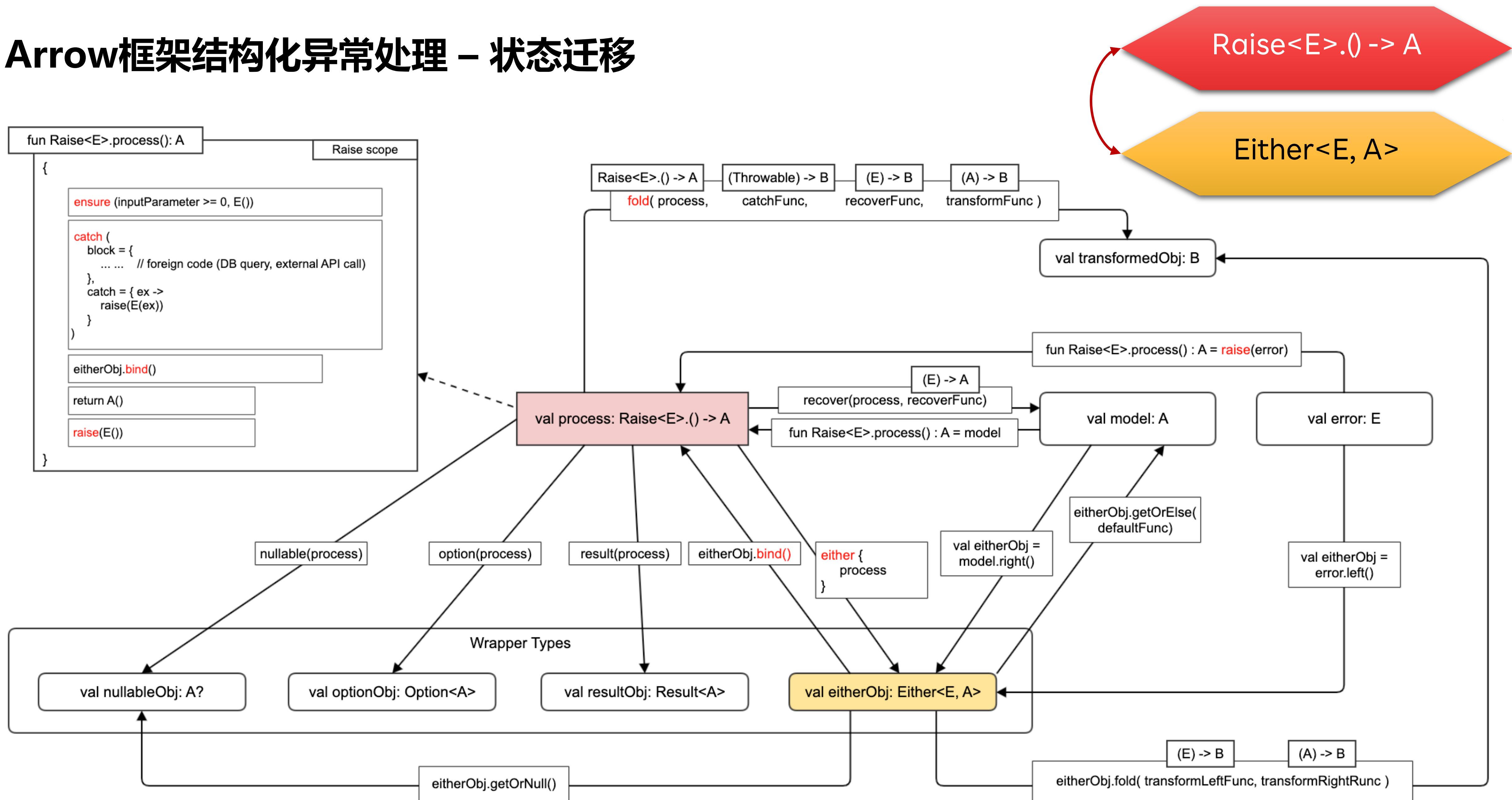
Arrow框架结构化异常处理 – 开启Raise Scope的完整例子

Raise<E>.() -> A

- 通过 **fold()** 或者 **either()** 等函数开启 **Raise Scope**，在其中可以层级调用Raise风格的函数
- 在Raise Scope内，可以使用一些Raise DSL方法 (例如: raise, ensure, bind, catch)



Arrow框架结构化异常处理 – 状态迁移



代码演示

- 业务场景及功能描述
  - 一个简化版的会员管理用API服务，包含以下功能：
    - 用户注册
    - 用户登录
    - 用户查询

- 所使用的技术栈
  - Spring Boot
  - JPA + H2 database
  - Arrow framework

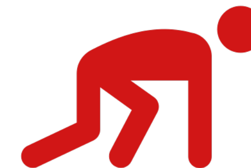
使用Arrow框架的一些体会

Arrow框架带来了什么?

- ✓ 开箱即用的结构化异常类型
- ✓ 代码可读性和可维护性的提升
  - 可预测的业务逻辑失败
  - 对于业务逻辑失败的编译期类型安全检查
- ✓ Kotlin函数式编程的完美拼图

如何有效地学习?

- ✓ 理解核心设计(Either & Raise)
- ✓ 熟悉DSL方法及其内在本质
  - raise()
  - bind()
  - either()
  - fold()



如何在项目中使用?

- ✓ 全局策略设计
  - 找到合适的引入Raise Scope的切入点
  - 制定Recovery策略
- ✓ 代码风格统一
- ✓ Enjoy !

Rakuten