Трансформеры монад

Кольцов Максим
Мат-Мех

5 декабря 2019 года

BIOCAD
Biotechnology Company

Монада Reader

Монада Reader

```haskell
data AppConfig = AppConfig …

loadConfig :: IO AppConfig
loadFiles ::  AppConfig -> IO [String]
doWork ::     AppConfig -> String -> IO Int
saveResult :: AppConfig -> [Int] -> IO ()
```

Монада Reader

```haskell
data AppConfig = AppConfig …

loadConfig :: IO AppConfig
loadFiles ::  AppConfig -> IO [String]
doWork ::     AppConfig -> String -> IO Int
saveResult :: AppConfig -> [Int] -> IO ()

main :: IO ()
main = do
  cfg <- loadConfig
  files <- loadFiles cfg
  results <- mapM (doWork cfg) files
  saveResult cfg results
```

Монада Reader

```haskell
data AppConfig = AppConfig …

loadConfig :: IO AppConfig
loadFiles ::  AppConfig -> IO [String]
doWork ::     AppConfig -> String -> IO Int
saveResult :: AppConfig -> [Int] -> IO ()

main :: IO ()
main = do
  cfg <- loadConfig
  files <- loadFiles cfg
  results <- mapM (doWork cfg) files
  saveResult cfg results
```

Монада Reader

```haskell
data AppConfig = AppConfig …

loadConfig :: IO AppConfig
loadFiles ::  AppConfig -> IO [String]
doWork ::     AppConfig -> String -> IO Int
saveResult :: AppConfig -> [Int] -> IO ()

main :: IO ()
main = do
  cfg <- loadConfig
  files <- loadFiles cfg
  results <- mapM (doWork cfg) files
  saveResult cfg results
```

Монада Reader

```haskell
newtype Reader r a = Reader (r -> a)

instance Monad (Reader r) where
  (Reader x) >>= f = Reader $ \r -> let Reader y = f (x r) in y r

ask :: Reader r r
ask = Reader id
```

Монада Reader

```haskell
newtype Reader r a = Reader (r -> a)

instance Monad (Reader r) where
  (Reader x) >>= f = Reader $ \r -> let Reader y = f (x r) in y r

ask :: Reader r r
ask = Reader id
```

```haskell
foo :: Reader Int String
foo = do
  i <- ask
  return $ show $ i * 2
```

Монада Reader

```haskell
newtype Reader r a = Reader (r -> a)

instance Monad (Reader r) where
  (Reader x) >>= f = Reader $ \r -> let Reader y = f (x r) in y r

ask :: Reader r r
ask = Reader id
```

IO ?

```haskell
foo :: Reader Int String
foo = do
  i <- ask
  return $ show $ i * 2
```

Композиция функторов

Композиция функторов

```haskell
m1 :: Maybe Int
m1 = Just 21

m2 :: Maybe Int
m2 = fmap (* 2) m1
```

Композиция функторов

```haskell
m1 :: Maybe Int
m1 = Just 21

m2 :: Maybe Int
m2 = fmap (* 2) m1
```

```haskell
l1 :: [Int]
l1 = [21, 22]

l2 :: [Int]
l2 = fmap (* 2) l2
```

Композиция функторов

```haskell
m1 :: Maybe Int
m1 = Just 21

m2 :: Maybe Int
m2 = fmap (* 2) m1

lm :: [Maybe Int]
lm = [Just 21]

lm2 :: [Maybe Int]
lm2 = fmap (fmap (* 2)) lm
```

```haskell
l1 :: [Int]
l1 = [21, 22]

l2 :: [Int]
l2 = fmap (* 2) l2
```

Композиция функторов

```haskell
m1 :: Maybe Int
m1 = Just 21

m2 :: Maybe Int
m2 = fmap (* 2) m1

lm :: [Maybe Int]
lm = [Just 21]

lm2 :: [Maybe Int]
lm2 = fmap (fmap (* 2)) lm
```

```haskell
l1 :: [Int]
l1 = [21, 22]

l2 :: [Int]
l2 = fmap (* 2) l2

ml :: Maybe [Int]
ml = Just [21, 22]

ml2 :: Maybe [Int]
ml2 = fmap (fmap (* 2)) ml
```

Композиция функторов

```haskell
newtype Compose f g a = Compose (f (g a))

instance (Functor f, Functor g) => Functor (Compose f g) where
  fmap f (Compose x) = Compose $ fmap (fmap f) x
```

Композиция функторов

```haskell
newtype Compose f g a = Compose (f (g a))

instance (Functor f, Functor g) => Functor (Compose f g) where
  fmap f (Compose x) = Compose $ fmap (fmap f) x
```

```haskell
lmC :: Compose Maybe [] Int
lmC = Compose $ Just [21]

lmC2 :: Compose Maybe [] Int
lmC2 = fmap (* 2) lmC
```

Композиция монад

```haskell
mM :: Maybe Int
mM = m1 >>= (\i -> return $ i * 2)

lM :: [Int]
lM = l1 >>= (\i -> return $ i * 2)
```

Композиция монад

```haskell
mM :: Maybe Int
mM = m1 >>= (\i -> return $ i * 2)

lM :: [Int]
lM = l1 >>= (\i -> return $ i * 2)

mlM :: Maybe [Int]
mlM = ml >>= _
```

Композиция монад

```haskell
mM :: Maybe Int
mM = m1 >>= (\i -> return $ i * 2)

lM :: [Int]
lM = l1 >>= (\i -> return $ i * 2)

mlM :: Maybe [Int]
mlM = ml >>= (\li -> _)
```

Композиция монад

```haskell
mM :: Maybe Int
mM = m1 >>= (\i -> return $ i * 2)

lM :: [Int]
lM = l1 >>= (\i -> return $ i * 2)

mlM :: Maybe [Int]
mlM = ml >>= (\li -> return (li >>= _))
```

Композиция монад

```haskell
mM :: Maybe Int
mM = m1 >>= (\i -> return $ i * 2)

lM :: [Int]
lM = l1 >>= (\i -> return $ i * 2)

mlM :: Maybe [Int]
mlM = ml >>= (\li -> return (li >>= \i -> return $ i * 2))
```

Композиция монад

```haskell
instance (Monad f, Monad g) => Monad (Compose f g) where
  (Compose x) >>= f = _
```

Композиция монад

```haskell
instance (Monad f, Monad g) => Monad (Compose f g) where
  (Compose x) >>= f = _

  bindLM :: [Maybe a] -> (a -> [Maybe b]) -> [Maybe b]
  bindLM lm f = do
    x <- lm
    case x of
      Nothing -> return Nothing
      Just y -> f y

  bindML :: Maybe [a] -> (a -> Maybe [b]) -> Maybe [b]
  bindML ml f = ml >>= (return . concat . mapMaybe f)
```

Композиция монад

```
λ> :t bindLM
bindLM
  :: Monad m => m (Maybe a) -> (a -> m (Maybe b)) -> m (Maybe b)
```

Композиция монад

```
λ> :t bindLM
bindLM
  :: Monad m => m (Maybe a) -> (a -> m (Maybe b)) -> m (Maybe b)
```

Композиция монад

```
λ> :t bindLM
bindLM
  :: Monad m => m (Maybe a) -> (a -> m (Maybe b)) -> m (Maybe b)
```

```haskell
newtype MaybeT m a = MaybeT (m (Maybe a))

instance Monad m => Monad (MaybeT m) where
  MaybeT v >>= f = MaybeT $ do
    x <- v
    case x of
      Nothing -> return Nothing
      Just y -> let MaybeT r = f y in r
```

ReaderT

```haskell
newtype ReaderT r m a = ReaderT (r -> m a)
instance Monad m => Monad (ReaderT s m)

runReaderT :: r -> ReaderT r m a -> m a
runReaderT r (ReaderT f) = f r
```

ReaderT

```haskell
newtype ReaderT r m a = ReaderT (r -> m a)
instance Monad m => Monad (ReaderT s m)

runReaderT :: r -> ReaderT r m a -> m a
runReaderT r (ReaderT f) = f r


newtype Reader  r   a = Reader  (r ->   a)
```

ReaderT

```haskell
loadFilesR :: ReaderT AppConfig IO [String]
doWorkR :: String -> ReaderT AppConfig IO Int
saveResultR :: [Int] -> ReaderT AppConfig IO ()

appR :: IO ()
appR = do
  cfg <- loadConfig
  runReaderT cfg $ do
    files <- loadFilesR
    results <- mapM doWorkR files
    saveResultR results
```

ReaderT

```haskell
loadFilesR :: ReaderT AppConfig IO [String]
doWorkR :: String -> ReaderT AppConfig IO Int
saveResultR :: [Int] -> ReaderT AppConfig IO ()

appR :: IO ()
appR = do
  cfg <- loadConfig
  runReaderT cfg $ do
    files <- loadFilesR
    results <- mapM doWorkR files
    saveResultR results
```

StateT

```haskell
newtype StateT s m a = StateT (s -> m (s, a))

instance Monad m => Monad (StateT s m)


get :: Monad m => StateT s m s
get = StateT $ \s -> return (s, s)

put :: Monad m => s -> StateT s m ()
put s = StateT $ \_ -> return (s, ())
```

ReaderT + StateT

```haskell
type Handler a = ReaderT Request (StateT Headers IO) a
```

```haskell
handle :: Handler
handle = do
  req <- ask
  -- do something...
  return Response
```

ReaderT + StateT

```haskell
addHeader :: String -> String -> Headers -> Headers
addHeader = _

addHeaderS :: Monad m => String -> String -> StateT Headers m ()
addHeaderS k v = do
  h <- get
  put $ addHeader k v h
```

ReaderT + StateT

```haskell
addHeaderH :: String -> String -> Handler ()
addHeaderH k v = do
  h <- _
```

MonadTrans

MonadTrans

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

MonadTrans

```haskell
class MonadTrans t where
  lift :: Monad m => m a -> t m a

instance MonadTrans (ReaderT r) where
  lift m = ReaderT $ \_ -> m

instance MonadTrans (StateT s) where
  lift m = StateT $ \s -> fmap (\a -> (s, a)) m
```

MonadTrans

```haskell
addHeaderH :: String -> String -> Handler ()
addHeaderH k v = do
  h <- lift get
  lift $ put $ addHeader k v h
```

MonadTrans

```haskell
addHeaderH :: String -> String -> Handler ()
addHeaderH k v = do
  h <- lift get
  lift $ put $ addHeader k v h
```

MonadTrans

```haskell
addHeaderH :: String -> String -> Handler ()
addHeaderH k v = do
  h <- lift get
  lift $ put $ addHeader k v h

handle :: Handler Response
handle = do
  req <- ask
  -- do something...
  addHeaderH "Content-Type" "application/json"
  return Response
```

MonadIO

```haskell
class MonadIO m where
  liftIO :: IO a -> m a

instance MonadIO IO where
  liftIO = id

instance (Monad m, MonadIO m) => MonadIO (ReaderT r m) where
  liftIO = lift . liftIO
```

mtl

```haskell
addHeaderS :: Monad m => String -> String -> StateT Headers m ()
addHeaderS k v = do
  h <- get
  put $ addHeader k v h
```

```haskell
class Monad m => MonadState s m where
  get' :: m s
  put' :: s -> m ()

instance Monad m => MonadState s (StateT s m)

addHeaderMS :: MonadState Headers m => String -> String -> m ()
addHeaderMS k v = do
  h <- get'
  put' $ addHeader k v h
```

mtl

```haskell
addHeaderS :: Monad m => String -> String -> StateT Headers m ()
addHeaderS k v = do
  h <- get
  put $ addHeader k v h
```

```haskell
class Monad m => MonadState s m where
  get' :: m s
  put' :: s -> m ()

instance Monad m => MonadState s (StateT s m)

addHeaderMS :: MonadState Headers m => String -> String -> m ()
addHeaderMS k v = do
  h <- get'
  put' $ addHeader k v h
```

mtl

```
type Handler a = ReaderT Request (StateT Headers IO) a
```

mtl

```haskell
type Handler a = ReaderT Request (StateT Headers IO) a

instance MonadState s m => MonadState s (ReaderT r m) where
  get' = lift get'
  put' = lift . put'
```

mtl

```haskell
type Handler a = ReaderT Request (StateT Headers IO) a

instance MonadState s m => MonadState s (ReaderT r m) where
  get' = lift get'
  put' = lift . put'
```

n^2 инстансов :(

Эффекты

Трансформер	Эффект
`ReaderT`	Общая конфигурация
`StateT`	Изменяемое состояние
`WriterT`	Дополняемый лог
`MaybeT`	Вычисление с ошибкой
`ExceptT`	Вычисление с информацией о ошибке
`ContT`	Вычисление с прерыванием

Альтернативы

- capability
- free
- freer
- polysemy
- fused-effects
- …

Альтернативы

- capability
- free
- freer
- polysemy
- fused-effects
- ...

```
handle
  :: (Has (Reader Request) sig m, Has (State Headers) sig m)
  => m Response
```

mtl-style

mtl-style

```haskell
class MonadDB where
  getUser :: Int -> m User
  addPost :: Post -> m ()

class MonadLog where
  logInfo :: String -> m ()
```

mtl-style

```haskell
class MonadDB where
  getUser :: Int -> m User
  addPost :: Post -> m ()

class MonadLog where
  logInfo :: String -> m ()


handle
  :: (MonadDB m, MonadLog m, MonadError WebError m)
  => Request
  -> m ()
```