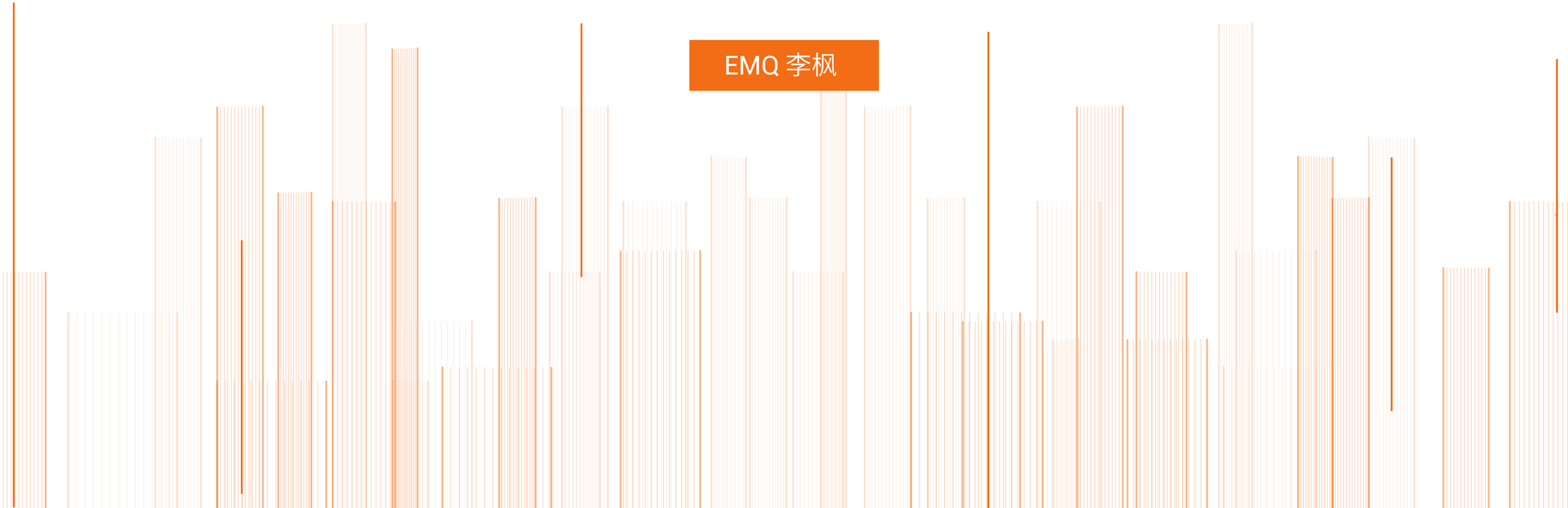
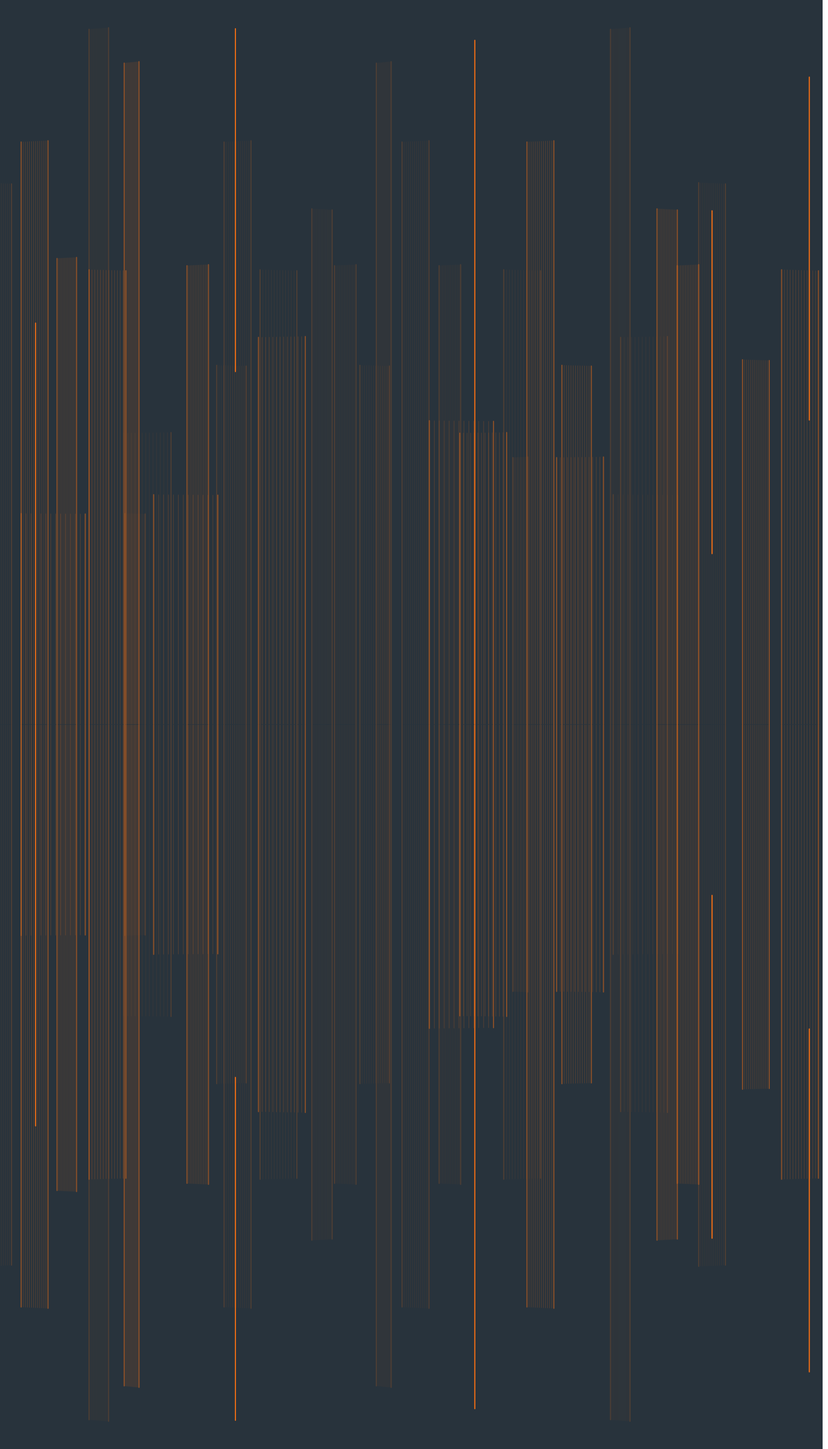
Hamler - 面向 IoT&5G 应用的开源函数编程语言

Haskell-style functional programming language running on Erlang VM



EMQ 李枫



目录

01 Why Hamler?

02 Hamler 概述

03 编译器架构

04 安装与使用

05 函数式编程

06 并发编程

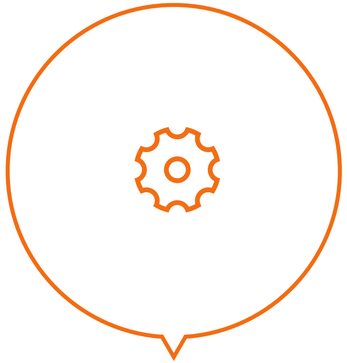
07 Hamler 未来展望

Erlang 编程语言

Erlang/OTP 与 Beam 虚拟机是工程学的杰作

EMQ X 与 Erlang 编程语言

近十年来，我们一直在开发基于 Erlang/OTP 的软件系统，特别是我们的核心产品可伸缩分布式开源 MQTT 服务器 - EMQ X



Erlang VM 是工程学杰作

我们一直认为 Erlang/OTP，尤其是 Beam 虚拟机是工程学的杰作。它具有出色的并发性、分布性和容错性，是少数正确处理高并发和软实时的通用语言平台

IoT & 5G 应用的重要开发平台

Erlang VM 非常适合 5G、IoT、云计算和边缘计算等未来潜力领域，构建下一代高并发、高可靠、可扩展、具备软实时支持应用

关于 Actor Model

1974年，卡尔-休伊特教授发表了论文《Actor model of computation》。文中，他阐述了 Actor 作为一个计算实体，它会对收到的消息作出回应，并可以并发地进行以下操作：

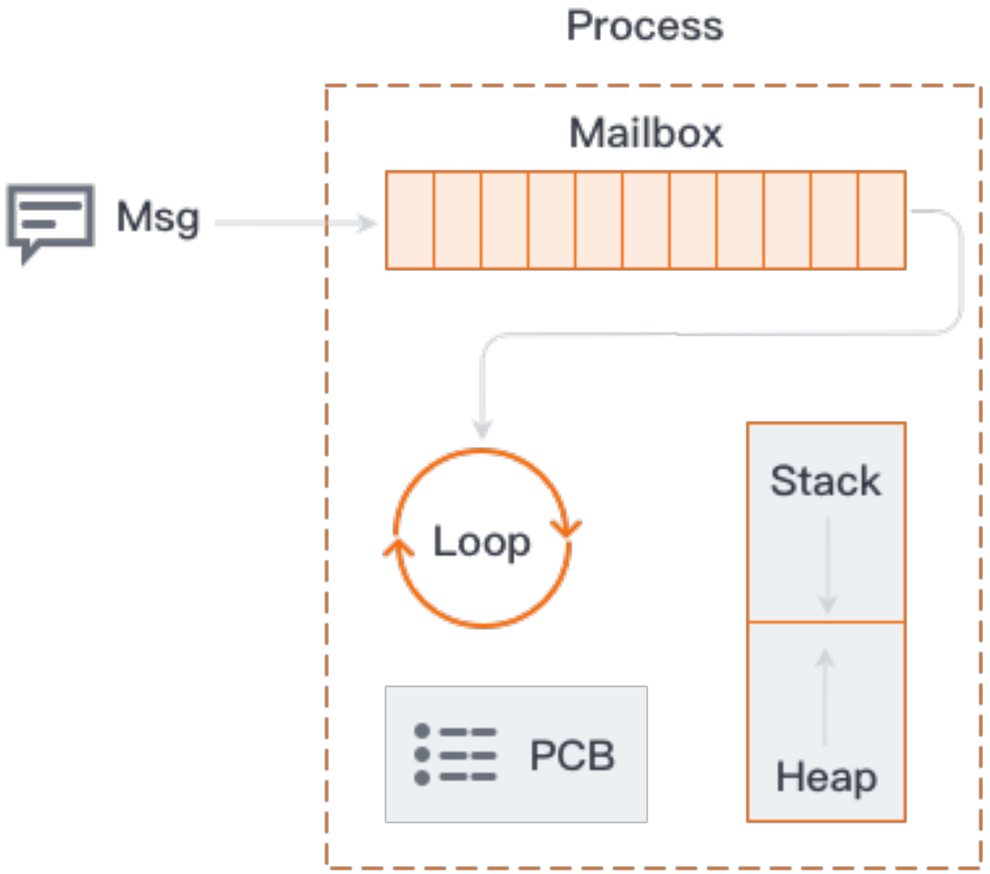
01 向其他 Actor 发送有限数量的消息

02 创建有限数量的新 Actor

03 指定下一个收到的消息所要使用的行为

随着多核计算和大规模分布式系统的兴起，Actor 模型因其天然的并发性、并行性和分布式变得越来越重要。

Erlang Process & Mailbox



Erlang 中的 Actor 被定义为一个进程，它的工作方式就像一个 OS 进程。

每个进程都有自己的内存，由一个 Mailbox、一个 Heap、一个 Stack 和一个包含进程信息的 Process Control Block(PCB) 组成。

Erlang 中的进程非常轻量，可以在一个正在运行的 Erlang 虚拟机上快速创建数百万个进程。

Erlang 编程语言存在的问题

业界过去20年的改进努力

01 为 Erlang 引入类型系统

Philip Wadler 教授和 Simon Marlow 在 2000 年前后，为 Erlang 引入了类型标注和 **Dialyzer** 静态类型检查工具：

Simon Marlow & Philip Wadler (1997): A practical subtyping system for Erlang

Philip Wadler (2002): The great type hope

02 Erlang VM 上开发新的语言

2008 年后，产业界有近 20 个项目，不断地尝试解决类型系统和友好语法的问题。Elixir 项目引入了 Ruby 语法，吸引了部分 Ruby On Rails 社区开发者，却没有类型系统支持。

此外还有 LFE 引入了 Lisp 语法，Alpaca 、Efene 、Elchemy 、Gleam 等项目试图引入 ML 风格语法和静态类型，目前大部分仍处于很早期的开发中。

03 JVM 上实现 Erlang/OTP 架构

Akka 项目在 JVM 上模拟实现了 Erlang/OTP ，但丧失了 Erlang/OTP 的软实时特性。

Well-Typed 公司的 Cloud Haskell 项目试图在 Haskell 上模拟实现 Erlang/OTP，目前项目已经停滞。

关于新语言构想

Haskell 风格编程语言运行在 Erlang 虚拟机

- BEAM is a really awesome VM
- Bring Haskell style language to BEAM
- Introduce the Type System, ADT
- Functional Programming & Concurrency

Hamler 的诞生

一门构建在 Erlang 虚拟机(VM)上的 Haskell 风格的强类型(strongly-typed)编程语言，独特地结合了编译时的类型检查推导，与对运行时高并发和软实时能力的支持。

函数式编程

类 Haskell 和 ML 的友好语法

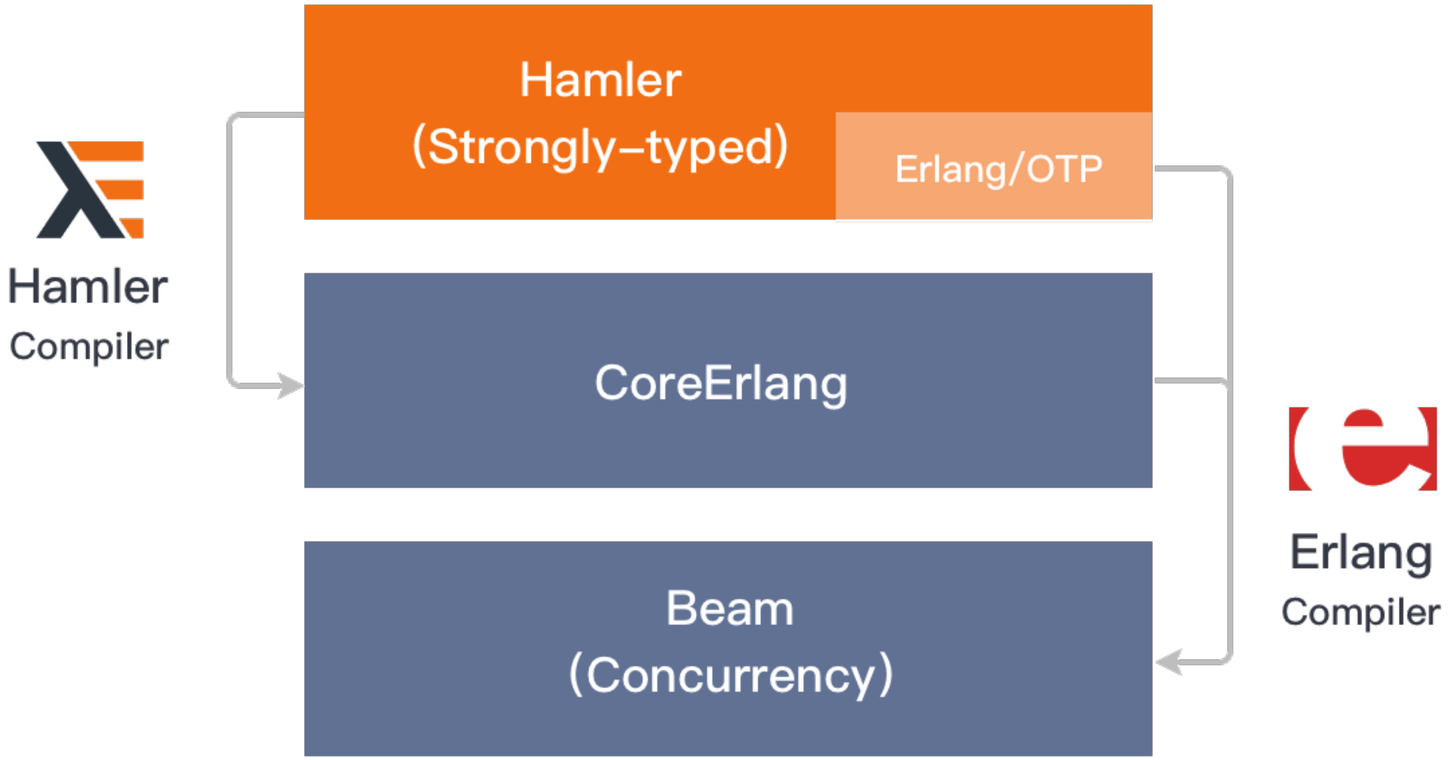
强类型语言

编译时的类型检查与类型推导

并发运行时

运行时的高并发、软实时支持

Hamler 编译器架构



Hamler 编译器设计

Hamler 0.1 编译器最初尝试基于 GHC 8.10.1 实现，后改为基于 Purescript 0.13.6 实现

Hamler 源码

ML (Meta Language) 的友好语法风格

抽象语法树

解析语法树 -> 抽象语法树，类型检查

CoreFn

Purescript 的通用 CoreFn 实现

CoreErlang

Erlang 编译器的 IR 代码，编译为 Beam

安装 Hamler

01 Homebrew (macOS)

```
brew tap hamler-lang/hamler
brew install hamler
```

02 Centos / Redhat

```
$ rpm -ivh hamler-$version-1.el7.x86_64.rpm
```

03 Debian / Ubuntu

```
$ dpkg -i hamler_$version_amd64.deb
```

Hello World!


```
module Main where

import Prelude

main :: IO ()
main = println "Hello, world!"
```

A Sexy QuickSort


```
quickSort :: forall a. Ord a => [a] -> [a]
quickSort [] = []
quickSort [x|xs] = quickSort [v|v <- xs, v < x] ++ [x] ++ quickSort [v|v <- xs, v >= x]
```

Hamler REPL


```
$ hamler repl
> -- List, range and enums
> [1,2,3]
> [1..10]
> ['a'..'z']

> -- erlang style maps
> import Data.Map as Map
> -- New map
> m = #{"foo" => "bar", "bar" => "foo"}
> -- Match Map
> #{"foo" := a, "bar" := b} = m
> -- get, put
> Map.get "foo" m -- a = "bar"
> Map.get "bar" m -- b = "foo"
> m1 = Map.put "key" "val"
> -- keys, values
> keys = Map.keys m
> values = Map.values m
```

基本数据类型

Type	Values	Description
Atom	:ok, :error	Erlang Atom type
Boolean(Bool)	true | false	Boolean type
Char	'c', 'x'	UTF-8 character
String	"hello"	List of UTF-8 character
Integer(Int)	1, 2, -10	Integer type
Float(Double)	3.14	Float type
List	[1,2,3,4]	[Integer]
Tuple	(1, true)	
Map	#{"k" => "v"}	Erlang Map
Record		

函数、递归、高阶函数、闭包、Lambda


```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n - 1)

> fact 10
3628800
> fact 5
120

fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

> fib 10
89
> fib 5
8
```

```
apply :: forall a b. (a -> b) -> a -> b
apply f x = f x

compose :: forall a b. (b -> c) -> (a -> b) -> a -> c
compose g f x = g (f x)
```

Lambda (Anonymous Function)

```
multBy :: Integer -> Integer -> Integer
multBy n = \m -> m * n

mean :: Integer -> Integer -> Integer
mean = \x y -> (x + y) `div` 2  -- f = (\x -> \y -> (x + y) `div` 2)
```

Currying and partial application

Currying

```
-- uncurried
plus :: (Integer, Integer) -> Integer
plus (x, y) = x + y

-- sum is the curried version of plus
sum :: Integer -> Integer -> Integer
sum x y = x + y
```

Partial application

```
sum 1  2      :: Integer
sum 1 (2 + 3) :: Integer

add2 = sum 2  :: Integer -> Integer -- partially applied
x    = add2 3 :: Integer        -- x = 5
```

Pattern matching, and Guards

Function Pattern Matching

```
(x, y) = (1, 2)

-- function declartion via pattern matching
allEmpty [] = True
allEmpty _ = False

-- pattern matching stops when it finds the first match
```

Guarded Equations

```
abs n | n > 0     = n
      | otherwise = -n
```

List comprehension

A list comprehension consists of four types of elements: *generators*, *guards*, *local bindings*, and *targets*.

```
-- examples
[x*2 | x <- [1,2,3]]   -- [2,4,6]

[x * x | x <- [1..10]] -- [1,4,9,16,25,36,49,64,81,100]

-- multiple generators
[(x,y) | x <- [1,2,3], y <- [4,5]]

-- dependent generators
[(x,y) | x <- [1..3], y <- [x..3]]

-- conditions
even i = 0 == i % 2
[x | x <- [1..10], even x]
```

代数数据类型(ADT)


```
-- type synonym
type Name = String
"Miles" :: Name
"Miles" :: String

newtype UInt8 = UInt8 Integer
1 :: Integer
UInt8 1 :: UInt8

-- sum datatype
data Color = Red | Green | Blue
Blue :: Color

-- product datatype
data Pair = Pair Integer Integer
Pair 3 4 :: Pair

-- record product datatype
data Person = Person {
  name :: String
  age :: Integer
  address :: String
}
Person {name = "Miles", age = 50, address = "NY"} :: Person

-- generic datatype (maybe for example)
data Maybe a = Just a | None
data Result val err = Ok val | Error err

-- recursive datatype
data Tree = Leaf Integer | Node Tree Tree
```

Spawn a new process

In Hamler, a new process is created via the `spawn` functions, which are defined in `Control.Process.Spawn` module.

```
-- | Create a process
spawn :: forall a. IO a -> Process Pid

-- | Create and link a process
spawnLink :: forall a. IO a -> Process Pid

-- | Create and monitor a process
spawnMonitor :: forall a. IO a -> Process (Pid, Ref)
```

Send/Receive message


```
go :: Process ()
go = do
  pid <- spawn recv
  pid ! :msg

recv :: Process ()
recv = receive x -> printf "recv: %s" (showAny x)
```

```
go :: Process ()
go = do
  pid <- spawn selectiveRecv
  pid ! :bar
  pid ! :foo

selectiveRecv :: Process ()
selectiveRecv = do
  receive :foo -> println "foo"
  receive :bar -> println "bar"
```

```
go :: Process ()
go = do
  pid <- spawn recvAfter
  pid ! :foo

recvAfter :: Process ()
recvAfter =
  receive
    :bar -> println "recv bar"
  after
    1000 -> println "timeout"
```

A Ping/Pong Example

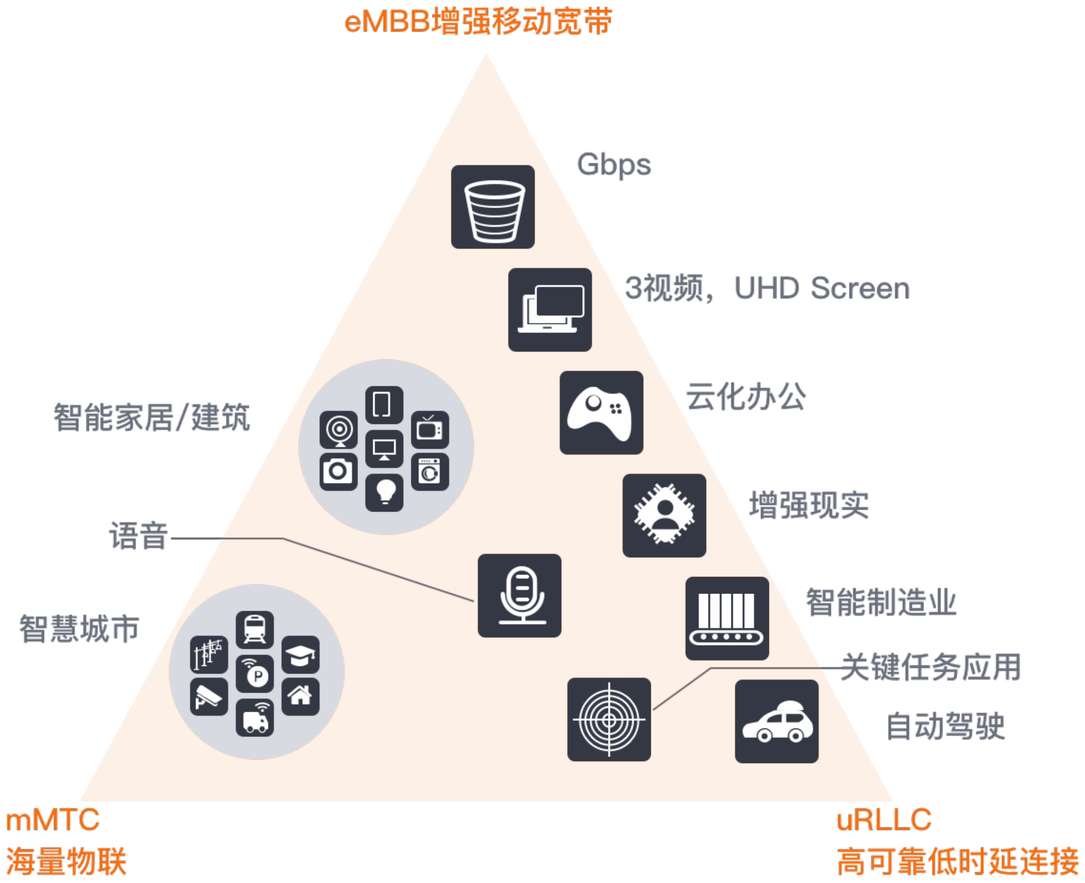

```
import Prelude

go :: Process ()
go = do
  self <- getSelf
  pid <- spawn loop
  pid ! (self, :ping)
  receive
    :pong -> println "Pong!"
  pid ! :stop

loop :: Process ()
loop =
  receive
    (from, :ping) -> do
      println "Ping!"
      from ! :pong
      loop
    :stop -> return ()
```

Hamler 未来展望

Hamler 将赋予5G、IoT、云计算和边缘计算等未来潜力领域，构建下一代高可靠、高并发、可扩展、软实时应用的能力。



eMBB 增强移动宽带

控制平面应用

mMTC 海量物联

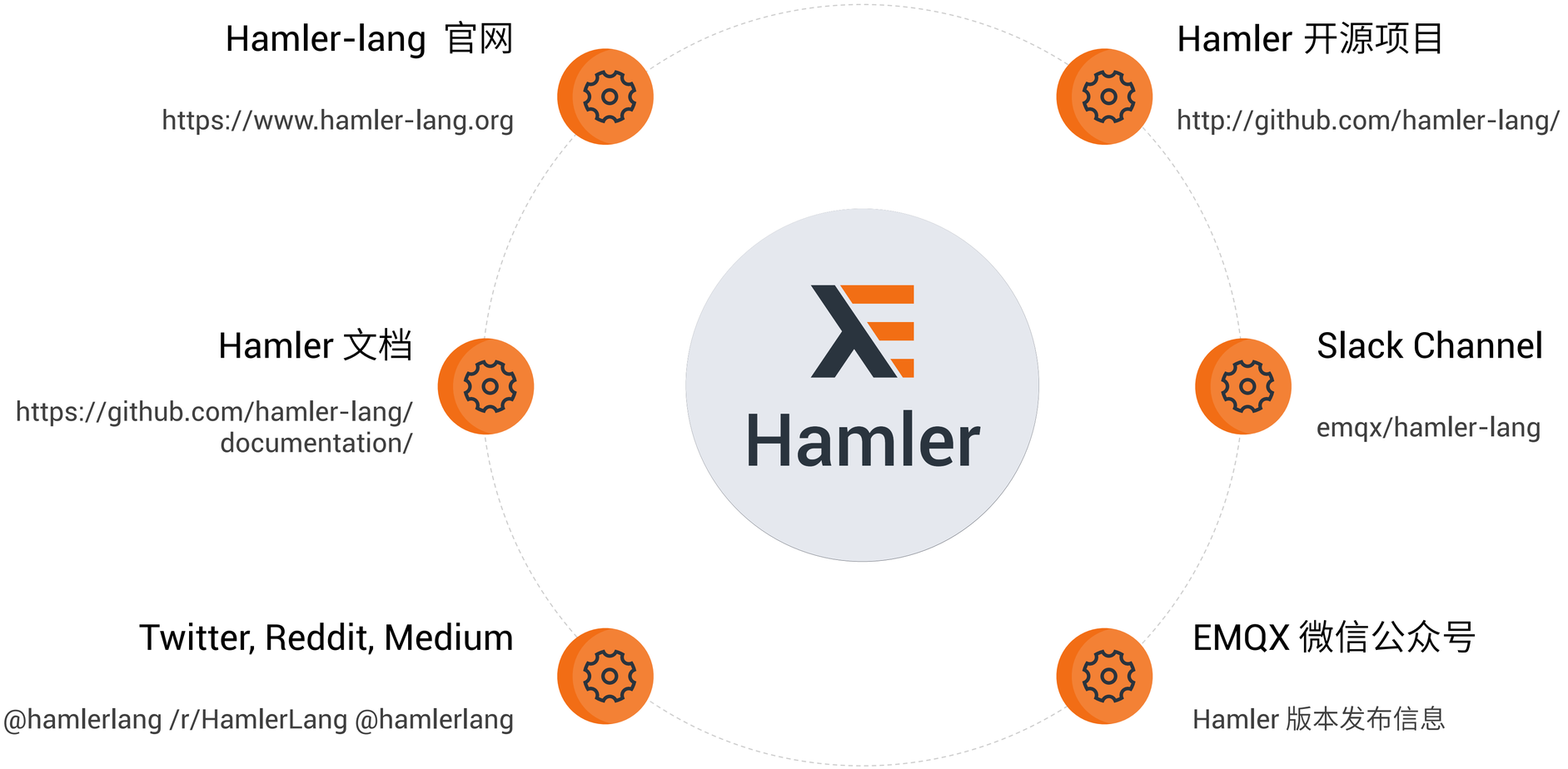
海量物联网连接处理

uRLLC 高可靠低时延连接

软实时低时延应用

欢迎加入 Hamler 编程语言社区

Hamler 函数编程语言从发起即是一个开源项目，项目托管在 GitHub: https://github.com/hamler-lang/



Thanks