形而上学

用函数式语言探究面向对象的语义

琳琳的小狗 @scriptfans

WTF! 形而上学?

- MetaPhysics
- τὰ μετὰ τὰ φυσικὰ βιβλία (ta meta ta physika biblia)

A. 亚里士多德《物理学》之后的一本没有名字的书

B. 探索物理学背后的哲学问题

“形而上者谓之道，形而下者谓之器。”

——《易传·系辞上》

形而上学

- 道：思想、实现原理
- 器：语法、细枝末叶
- ~~孤立、片面、静止的看待事物~~

主题内容

1. 对象？面向对象编程？？
2. 最基本的的Clojure语言构件
3. 以Ruby为蓝本，实现一套简单的（内部）DSL

用Clojure谈对象!

干嘛要折腾?

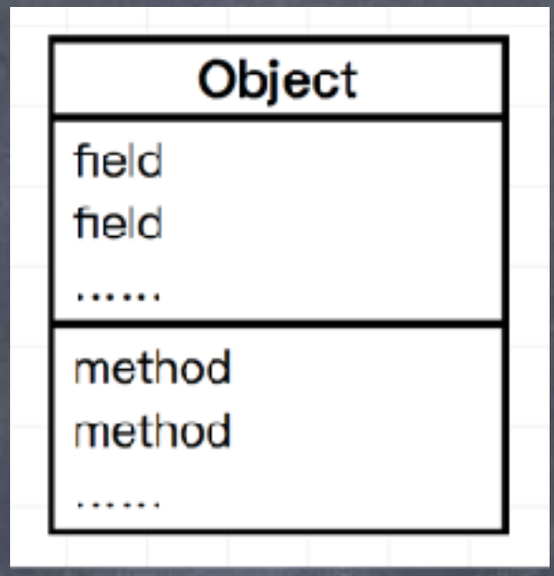
- 对象光“谈”是不够的
- 脱掉对象的外衣，让关系更进一步!
- Just for fun

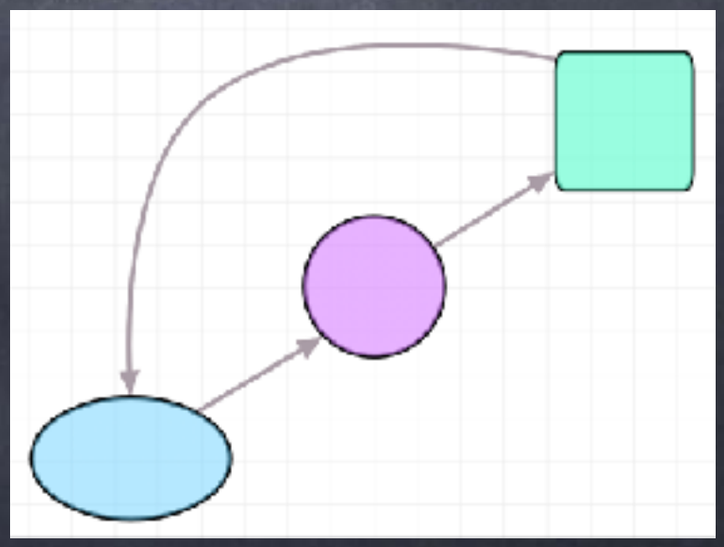
面向对象编程

1. 将现实世界建模为抽象的对象
2. 通过编排这些对象之间的交互，来解决实际的问题
3. 三要素：封装、继承、多态

对象：object

- 静态视角：
  - 封装数据信息（属性）
  - 实现操作逻辑（方法）
- 动态视角：
  - 消息通讯（发送/接收）
  - 协作实现复杂算法





Ruby

Untitled.rb — Edited

Ruby | Language | Run | Stop | Run Settings... | View

Untitled.rb

```ruby

class Pet
  def initialize(name)
    @name = name
  end
end

class Dog < Pet
  def bark
    "#{@name}: 汪~汪汪~汪汪汪~"
  end
end

dog = Dog.new "旺财"
puts dog.bark

doggie = Dog.send :new, "来福"
puts doggie.send(:bark)
```

```
旺财: 汪~汪汪~汪汪汪~
来福: 汪~汪汪~汪汪汪~
```

Run Succeeded | Time 66 ms | bark | Tab Size: 2 | Line 18, Column 24

用 Clojure 来造一个?

Resources

- https://gist.github.com/scriptfans/6fb52e4c5b61b336991893b26af35e69

- https://gist.github.com/scriptfans/caf62351320e01982303caa96ebbf15

用到的 Clojure 构件

- var：符号，可以用def绑定到某个值
  - (def aVar “I’m the value of aVal”)
- 关键字，以冒号开头
  - :key，求值结果为它自己
- let：声明局部绑定
  - (let [名字1 值，名字2 值2，…] 表达式序列)
- fn：定义匿名函数
  - (fn [形参1 形参2 & 可变参数] 函数体)
  - #(%1 %2 %3 … %&)

- defn：宏，用于定义命名函数
  - (def plus-one (fn [n] (+ 1 n)))
  - (def plus-one #(+ 1 %))
  - (defn plus-one [n] (+ 1 n))

- 函数调用
  - （函数名称 实参1 实参2 ...）
    - (plus-one 232)
- (apply 函数名称 实参1..实参n 剩余的实参序列)
  - (apply + 1 2 [3 4 5])

- quote：防止代码求值
  - A ；RuntimeException，符号未绑定
  - (quote A)；返回结果为符号A
  - ‘A
  - `(+ 1 2 3)；(+ 1 2 3)

- Unquote:
  - `(+ 1 2 (+ 3 4)) ; (+ 1 2 (+ 3 4))
  - `(+ 1 2 ~(+ 3 4)) ; (+ 1 2 7)
  - (eval `(+ 1 2 (+ 3 4))) ; 10

- 元数据：metadata
  - 可以附加在符号、值、对象上的额外信息
  - (with-meta 源对象 元数据)
    - (with-meta {} {:name “I’m a Map”})
    - (meta object-with-metadata) ;返回元数据

递归

- 将复杂问题规约为可重复的步骤
- 调用函数自身直到满足某个原子条件
- 示例：计算阶乘

解法一

```clojure
(defn factorial-1 [n]
  (if (or (= n 0) (= n 1))
    1
    (*' n (factorial-1 (dec n)))))

(println (factorial-1 100))
```

解法二：尾递归

```clojure
(defn factorial-2
  (
    [n acc]
    (if (or (= n 0) (= n 1))
      acc
      (recur (- n 1) (*' n acc)))
  )

  (
    [n] (factorial-2 n 1)
  )
)

(println (factorial-2 100))
```

解法三

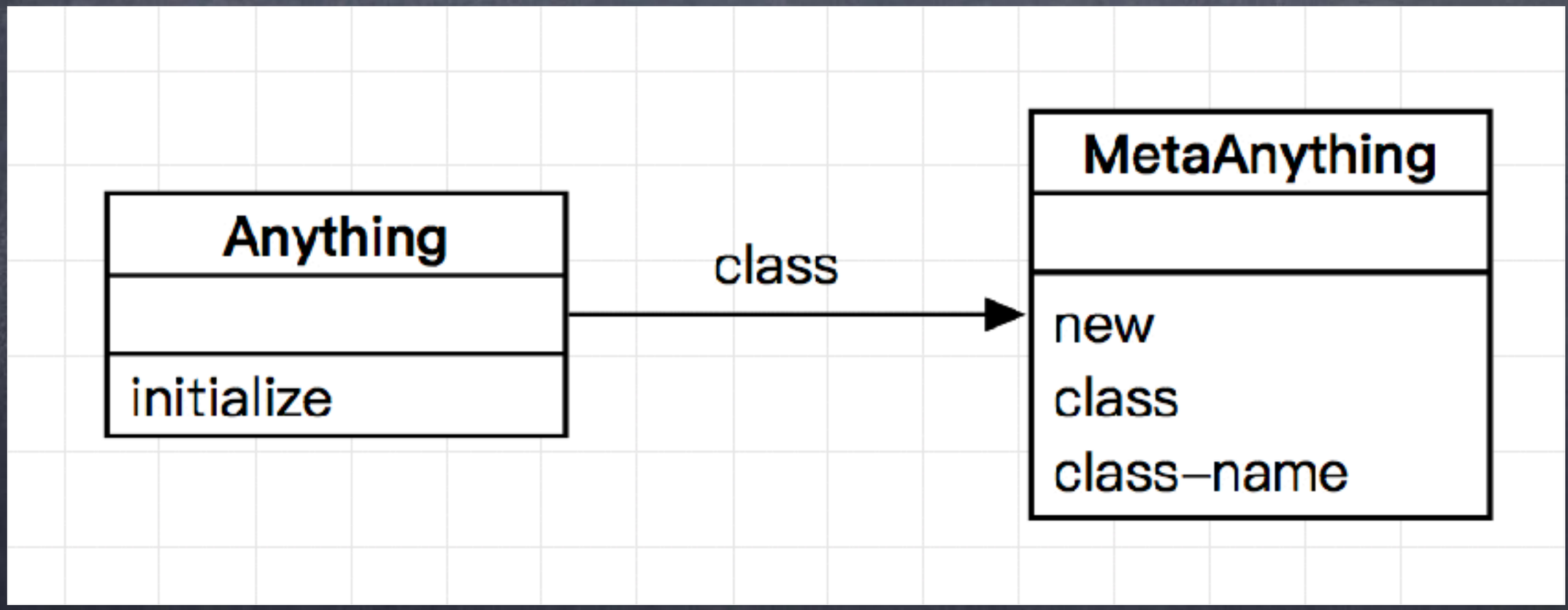
```
(def factorial-3 (fn [n]
  (apply *' (range 1 (+ n 1)))))

(println (factorial-3 100))
```

实现对象模型

语义设计

- 用map表示对象，通过元数据持有其class的引用
- 实例变量保存在对象中，实例方法保存在对象的类中
  - 类也是对象，也有自己的class
  - 类方法保存在类的类中（metaclass）
- 通过给对象发送消息来调用方法
  - (send-to object message args…)
  - 方法声明，第一个参数为实例对象本身，类似Python
- 单继承，根类型为Anything

根类型


Anything
initialize
class
MetaAnything
new
class
class-name

对象实例化过程

1. 分配内存：send-to

2. 添加元数据：class.new

3. 初始化实例变量：instance.initialize

Anything

```
(def MetaAnything (with-meta {
  :new (fn [self & args] (let [
    instance (with-meta {} {:class self})
  ] (apply send-to instance :initialize args)))

  :class (fn [self] self)

  :class-name (fn [self] (:class-name (meta self)))
} {:class-name "MetaAnything"}))
```

```
(def Anything (with-meta { ;实例方法
  ;构造函数，提供默认实现
  :initialize (fn [self] self)
} { ; 元数据，保存类的引用，以及自己的名称
  :class MetaAnything, :class-name "Anything"
}))
```

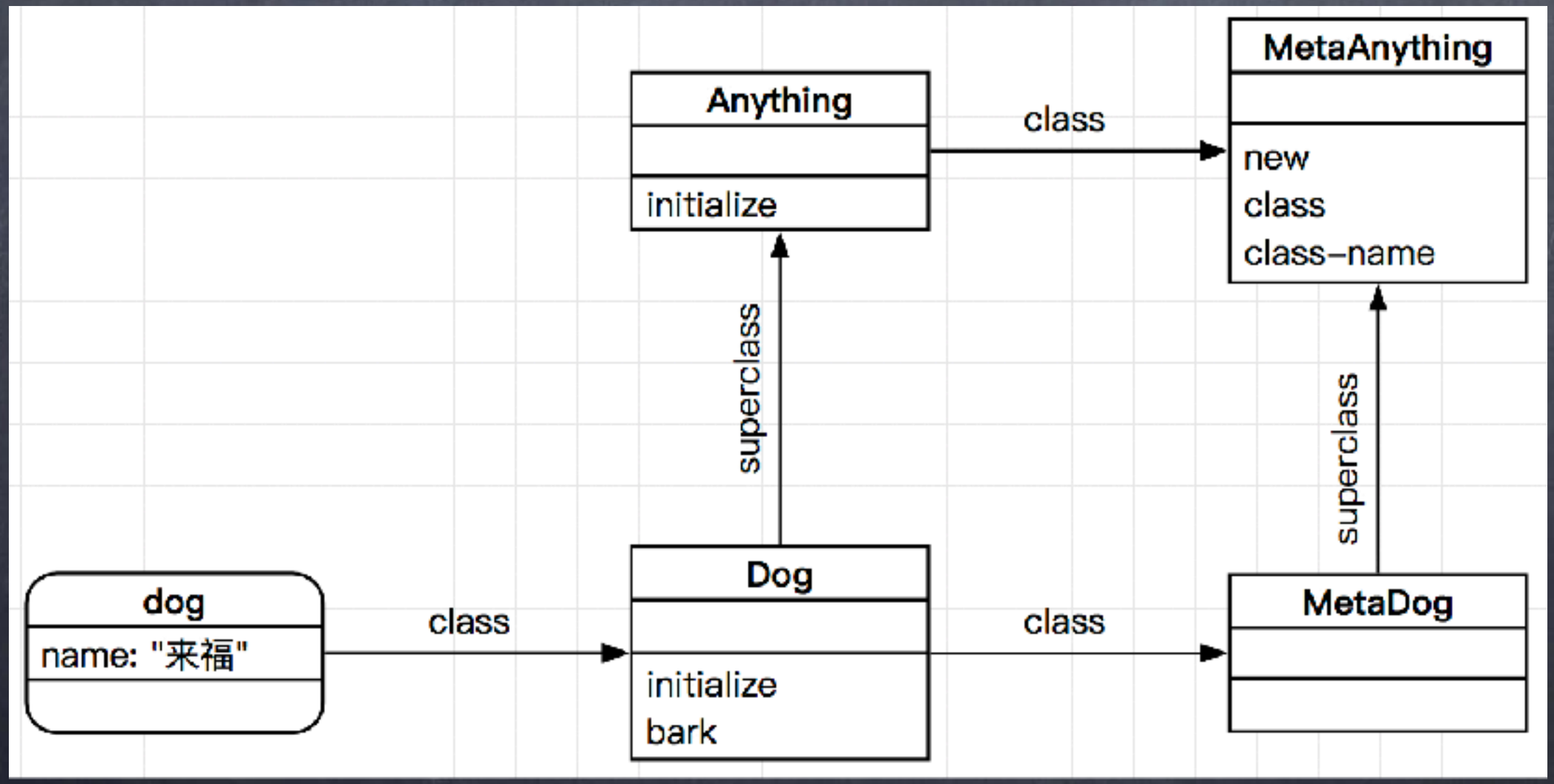
类定义示例

```
(def MetaDog (with-meta {

} {:class Anything, :class-name "MetaDog", :superclass MetaAnything}))

(def Dog (with-meta {

  :initialize (fn [self name] (merge self {:name name}))

  :bark (fn [self] (str "汪汪~" ", 我叫" (:name self)))

} {:class MetaDog, :class-name "Dog", :superclass Anything}))
```

```
(send-to Dog :new "来福") ; => {:name "来福"}
```

```
(def dog (send-to Dog :new "来福"))

(send-to dog :bark) ; => "汪汪~, 我叫来福"
```

Dog


MetaAnything
new
class
class-name
Anything
initialize
class
superclass
superclass
dog
name: "来福"
class
Dog
initialize
bark
class
MetaDog

消息发送

```
(defn send-to [object message & args]
  (let [
    method (instance-method (class-of object) message)
  ] (apply method object args))
)
```

辅助函数

```
(defn class-of [object]
  (let [
    metadata (meta object)
    class (:class metadata)
  ] class)
)

(defn super-class-of [class] (:superclass (meta class)))

(defn instance-method [class message]
  (if (nil? class)
    nil
    (let [
      method (message class)
      superclass (super-class-of class)
    ] (if (not (nil? method))
      method
      (instance-method superclass message)
    ))
  )
)
```

简化类声明

```
(define-class 'Pig Anything {} { ; 声明类的同时，自动生成对应的 MetaClass: MetaPig
  :say (fn [self] "我是逼格猪")
})

(def pig (send-to Pig :new))

(send-to pig :say) ; => "我是逼格猪"
```

define-meta-class

```
(defn define-meta-class [class-name superclass instance-methods]
  (let [
    meta-class-name (symbol (str "Meta" class-name))
    metaclass (with-meta instance-methods {
      :class Anything, :superclass superclass, :class-name (str meta-class-name)
    })
  ] (eval `(def ~meta-class-name ~metaclass)) metaclass)
)
```

define-class

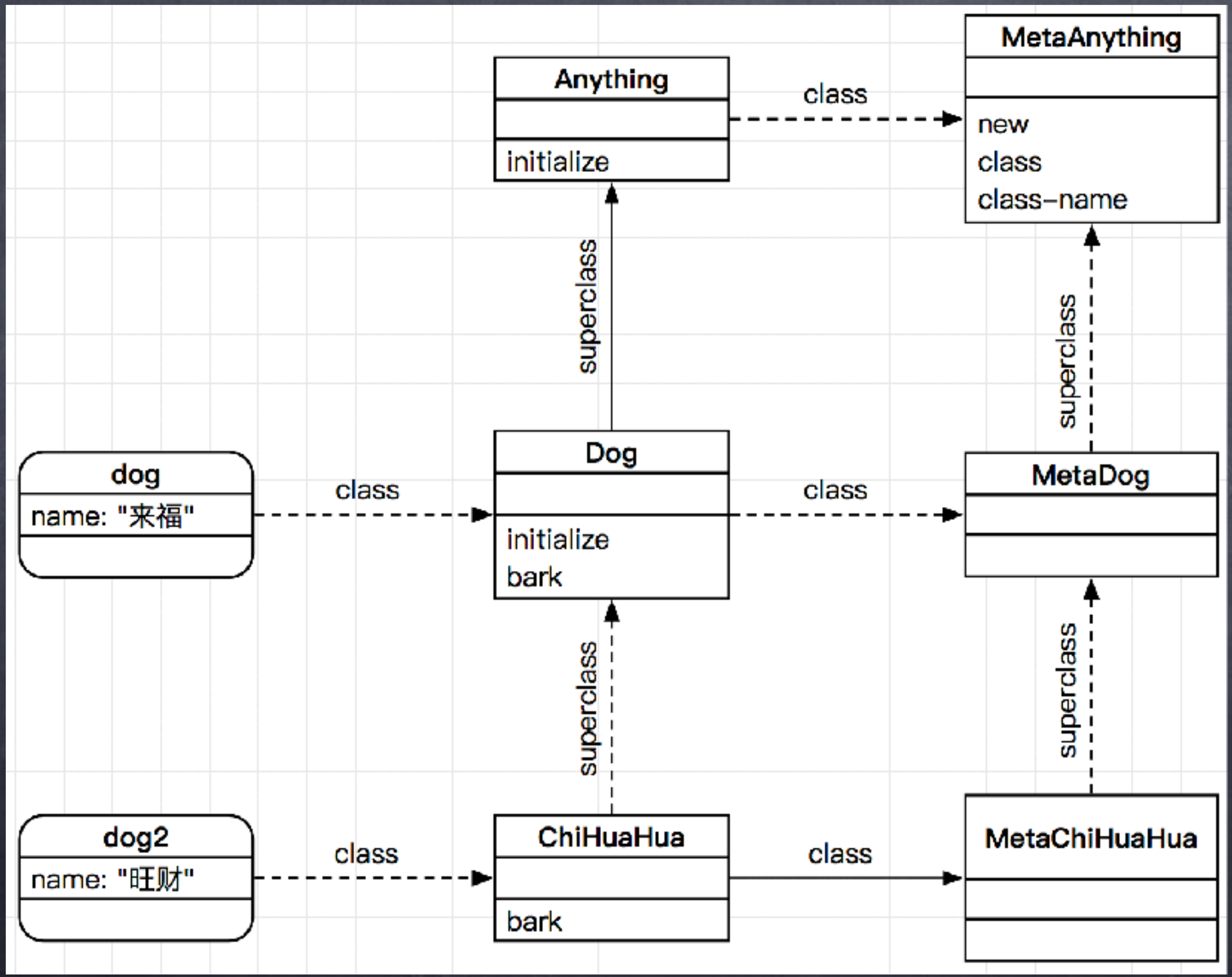
```
(defn define-class [class-name superclass class-methods instance-methods]
  (let [
    metaclass (
      define-meta-class class-name (or (class-of superclass) MetaAnything) class-methods
    )
    class (with-meta instance-methods {
      :class-name (str class-name), :superclass superclass, :class metaclass
    })
  ] (eval `(def ~(symbol class-name) ~class)) class)
)
```

完善继承

调用超类同名方法

super-method

```
(defn super-method [class message]
  (instance-method (super-class-of class) message)
)

(define-class 'ChiHuaHua Dog {} {
  :bark (fn [self]
    ;调用超类同名方法
    (str ((super-method (class-of self) :bark) self) ", 我是一只吉娃娃")
  )
})

(def dog2 (send-to ChiHuaHua :new "旺财"))

(send-to dog2 :bark) ; => “汪汪~, 我叫旺财, 我是一只吉娃娃"
```


Anything
initialize
class
MetaAnything
new
class
class-name
superclass
superclass
dog
name: "来福"
class
Dog
initialize
bark
class
MetaDog
superclass
superclass
dog2
name: "旺财"
class
ChiHuaHua
bark
class
MetaChiHuaHua

回头看 Ruby

- Metaclass是隐式的
- 具有Class类，使得创建类对象与普通对象保持高度一致
- self在方法内部直接可用
- 使用super关键字即可调用父类同名方法
- 各种hook，比如method_missing
  - 这有何难!
  - Tip: instance-method
  - 但是……

你们总是想实现更完美的轮子

可对象已经很累了

它不想被折磨

它想尽快结束被人围观

你关心过这些吗?

没有！你只关心你自己!

感谢您的倾听，请多指导