# Go 语⾔言从⼊入⻔门到实战



**扫码订阅/试看**

**《Go语⾔言从⼊入⻔门到实战》**

## 内容概述

|  |  |
| --- | --- |
| **Go 语⾔言基础** | 基本程序结构 |
| 常⽤用集合 |
| 函数式编程 |
| ⾯面向对象编程 |
| 错误处理理 |
| 模块化及依赖管理理 |
| **进阶与实战** | 并发编程模式 |
| 常⻅见并发任务 |
| 深⼊入测试 |
| 反射和 Unsafe |
| 常⻅见架构模式的实现 |
| 性能调优 |
| ⾼高可⽤用性服务设计 |

Go 语⾔言简介

软件开发的新挑战

1. 多核硬件架构
2. 超⼤大规模分布式计算集群
3. Web 模式导致的前所未有的开发规模和更更新速度

## Go 的创始⼈人



Rob Pike

Unix 的早期开发者

UTF-8 创始⼈人

Ken Thompson Unix 的创始⼈人 C 语⾔言创始⼈人1983 年年获图灵奖

Robert Griesemer

Google V8 JS Engine 开发者

Hot Spot 开发者

## Go 语⾔言发展

*Subject: Re: prog lang discussion From: Rob ‘Commander' Pike*

*Date*：*Tue,Sep 25, 2007 at 3:12 PM To: Robert Griesemer, Ken Thompson*

*i had a couple of thoughts on the drive home. 1.name*

*‘go'.you can invent reasons for this name but it has nice properties. it's short,easy to type.tools:goc,gol,goa.if there's an interactive*

*debugger/interpreter it could just be called‘go'.the suffix is .go*

...

简单

## C Go

37 25

C++ 84

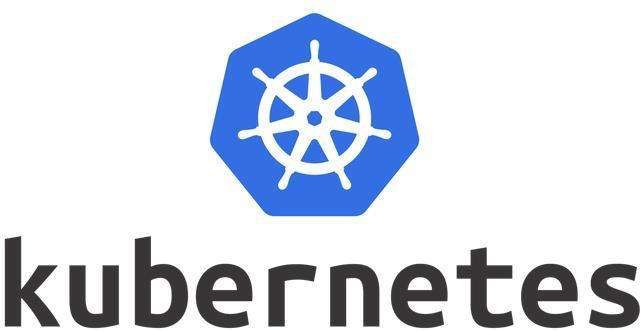
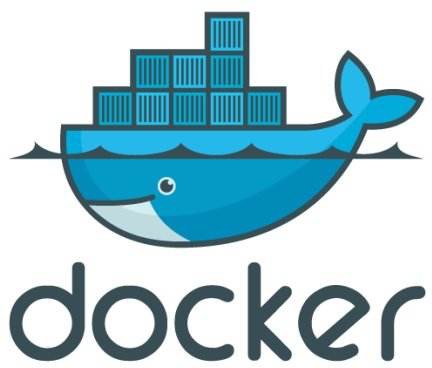
⾼高效

## 垃圾回收 指针

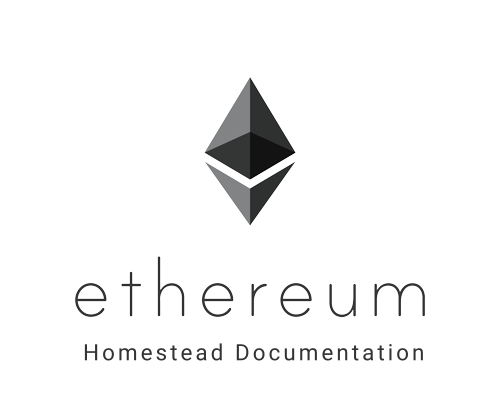
⽣生产⼒力力

复合 **vs** 继承

## 云计算语⾔言



区块链语⾔言



准备开始 Go 冒险之旅

下载安装 Go 语⾔言

https://golang.org/doc/install https://golang.google.cn/dl/

安装 IDE

Atom: https://atom.io + Package: go-plus

# 编写第⼀一个 Go 程序

## 开发环境构建

GOPATH

1. 在 1.8 版本前必须设置这个环境变量量
2. 1.8 版本后（含 1.8）如果没有设置使⽤用默认值

在 *Unix* 上默认为 *$HOME/go ,* 在 *Windows* 上默认为 *%USERPROFILE%/go*

在 *Mac* 上 *GOPATH* 可以通过修改 ～*/.bash\_profile* 来设置

## 基本程序结构

package main //包，表明代码所在的模块（包） import "fmt" //引⼊入代码依赖

//功能实现 func main() {

fmt.Println("Hello World!")

}

## 应⽤用程序⼊入⼝口

1. 必须是 main 包：package main
2. 必须是 main ⽅方法：func main()
3. ⽂文件名不不⼀一定是 main.go

## 退出返回值

#### 与其他主要编程语⾔言的差异

* Go 中 main 函数不不⽀支持任何返回值
* 通过 os.Exit 来返回状态

## 获取命令⾏行行参数

#### 与其他主要编程语⾔言的差异

* main 函数不不⽀支持传⼊入参数

func main(arg []string)

* 在程序中直接通过 os.Args 获取命令⾏行行参数

# 变量量与常量量

*The master has failed more times than the beginner has tried.*

## 编写测试程序

1. 源码⽂文件以 \_test 结尾：xxx\_test.go
2. 测试⽅方法名以 Test 开头：func TestXXX(t \*testing.T) {…}

## 实现 Fibonacci 数列列

1, 1, 2, 3, 5, 8, 13, ….

变量量赋值

#### 与其他主要编程语⾔言的差异

* 赋值可以进⾏行行⾃自动类型推断
* 在⼀一个赋值语句句中可以对多个变量量进⾏行行同时赋值

## 常量量定义

#### 与其他主要编程语⾔言的差异

**快速设置连续值**

const ( const (

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Monday = | iota | + | 1 | Open = 1 << iota |
| Tuesday |  |  |  | Close |
| Wednesday |  |  |  | Pending |
| Thursday |  |  |  | ) |
| Friday |  | | | |
| Saturday |
| Sunday |

)

# 数据类型

## 基本数据类型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bool | | | | |
| string | | | | |
| int | int8 | int16 | int32 | int64 |
| uint uint8 uint16 uint32 uint64 uintptr | | | | |
| byte // alias for uint8 | | | | |
| rune // alias for int32,represents a Unicode code point | | | | |
| float32 float64 | | | | |
| complex64 complex128 | | | | |

类型转化

#### 与其他主要编程语⾔言的差异

1. Go 语⾔言不不允许隐式类型转换
2. 别名和原有类型也不不能进⾏行行隐式类型转换

## 类型的预定义值

1. math.MaxInt64
2. math.MaxFloat64
3. math.MaxUint32

指针类型

与其他主要编程语⾔言的差异

* 1. 不不⽀支持指针运算
  2. string 是值类型，其默认的初始化值为空字符串串，⽽而不不是 nil

# 运算符

## 算术运算符

|  |  |  |
| --- | --- | --- |
| **运算符 描述 实例例** | | |
| + | 相加 | A + B 输出结果 30 |
| - | 相减 | A - B 输出结果 -10 |
| \* | 相乘 | A \* B 输出结果 200 |
| / | 相除 | B / A 输出结果 2 |
| % | 求余 | B % A 输出结果 0 |
| ++ | ⾃自增 | A++ 输出结果 11 |
| -- | ⾃自减 | A-- 输出结果 9 |

Go 语⾔言没有前置的 ++，- -，（++a）

## ⽐比较运算符

|  |  |  |
| --- | --- | --- |
| **运算符 描述 实例例** | | |
| == | 检查两个值是否相等，如果相等返回 True 否则返回 False。 | (A == B) 为 False |
| != | 检查两个值是否不不相等，如果不不相等返回 True 否则返回 False。 | (A != B) 为 True |
| > | 检查左边值是否⼤大于右边值，如果是返回 True 否则返回 False。 | (A > B) 为 False |
| < | 检查左边值是否⼩小于右边值，如果是返回 True 否则返回 False。 | (A < B) 为 True |
| >= | 检查左边值是否⼤大于等于右边值，如果是返回 True 否则返回 False。 | (A >= B) 为 False |
| <= | 检查左边值是否⼩小于等于右边值，如果是返回 True 否则返回 False。 | (A <= B) 为 True |

⽤用 == ⽐比较数组

* 相同维数且含有相同个数元素的数组才可以⽐比较
* 每个元素都相同的才相等

## 逻辑运算符

|  |  |  |
| --- | --- | --- |
| **运算符 描述 实例例** | | |
| && | 逻辑 AND 运算符。 如果两边的操作数都是 True，则条件 True，否则为 False。 | (A && B) 为 False |
| || | 逻辑 OR 运算符。 如果两边的操作数有⼀一个 True，则条件 True，否则为 False。 | (A || B) 为 True |
| ! | 逻辑 NOT 运算符。 如果条件为 True，则逻辑 NOT 条件 False，否则为 True。 | !(A && B) 为 True |

|  |  |  |
| --- | --- | --- |
| **运算符 描述 实例例** | | |
| & | 按位与运算符"&"是双⽬目运算符。 其功能是参与运算的两数各对应的⼆二进位相与。 | (A & B) 结果为 12, ⼆二进制为 0000 1100 |
| | | 按位或运算符"|"是双⽬目运算符。 其功能是参与运算的两数各对应的⼆二进位相或 | (A | B) 结果为 61, ⼆二进制为 0011 1101 |
| ^ | 按位异或运算符"^"是双⽬目运算符。 其功能是参与运算的两数各对应的⼆二进位相异或，当两对应的⼆二进位相异时，结果 | (A ^ B) 结果为 49, ⼆二进制为 0011 0001 |
| << | 左移运算符”<<"是双⽬目运算符。左移 n 位就是乘以 2 的 n  次⽅方。 其功能把"<<"左边的运算数的各⼆二进位全部左移若  ⼲干位，由"<<"右边的数指定移动的位数，⾼高位丢弃，低位补 | A << 2 结果为 240 ，⼆二进制为 1111 0000 |
| >> | 右移运算符”>>"是双⽬目运算符。右移 n 位就是除以 2 的 n 次⽅方。 其功能是把">>"左边的运算数的各⼆二进位全部右移若⼲干位，">>"右边的数指定移动的位数。 | A >> 2 结果为 15 ，⼆二进制为 0000 1111 |

#### 与其他主要编程语⾔言的差异

&^ 按位置零

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | &^ | 0 | -- | 1 |
| 1 | &^ | 1 | -- | 0 |
| 0 | &^ | 1 | -- | 0 |
| 0 | &^ | 0 | -- | 0 |

# 编写结构化程序

## 循环

#### 与其他主要编程语⾔言的差异

Go 语⾔言仅⽀支持循环关键字 for

for ( j := 7; j <= 9; j+

+ )

## 代码示例例

**while 条件循环**

**while （n<5)**

n := 0

for n < 5 { n++

fmt.Println(n)

}

**⽆无限循环**

**while (true)**

n := 0

for {

…

}

if condition {

// code to be executed if condition is true

} else {

// code to be executed if condition is false

}

if condition-1 {

// code to be executed if condition-1 is true

} else if condition-2 {

// code to be executed if condition-2 is true

} else {

// code to be executed if both condition1 and condition2 are false

}

#### 与其他主要编程语⾔言的差异

1. condition 表达式结果必须为布尔值
2. ⽀支持变量量赋值：

if var declaration; condition {

// code to be executed if condition is true

}

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| switch os := runtime.GOOS; | os | { | switch { |  | | | | |
| case "darwin": |  |  | case | 0 <= | Num | && Num | <= | 3: |

fmt.Println("OS X.”)

//break case "linux":

fmt.Println("Linux.") default:

// freebsd, openbsd,

// plan9, windows... fmt.Printf("%s.", os)

}

fmt.Printf("0-3")

case 4 <= Num && Num <= 6: fmt.Printf("4-6")

case 7 <= Num && Num <= 9: fmt.Printf("7-9")

}

## switch条件

#### 与其他主要编程语⾔言的差异

1. 条件表达式不不限制为常量量或者整数；
2. 单个 case 中，可以出现多个结果选项, 使⽤用逗号分隔；
3. 与 C 语⾔言等规则相反，Go 语⾔言不不需要⽤用break来明确退出⼀一个 case；
4. 可以不不设定 switch 之后的条件表达式，在此种情况下，整个 switch 结构与多个 if…else… 的逻辑作⽤用等同

# 数组和切⽚片

## 数组的声明

var a [3]int //声明并初始化为默认零值 a[0] = 1

b := [3]int{1, 2, 3} //声明同时初始化

c := [2][2]int{{1, 2}, {3, 4}} //多维数组初始化

数组元素遍历

与其他主要编程语⾔言的差异

func TestTravelArray(t \*testing.T) {

a := [...]int{1, 2, 3, 4, 5} //不不指定元素个数 for idx/\*索引\*/, elem/\*元素\*/ := range a {

fmt.Println(idx, elem)

}

}

## 数组截取

a[开始索引(包含), 结束索引(不不包含)]

a := [...]int{1, 2, 3, 4, 5}

a[1:2] //2

a[1:3] //2,3

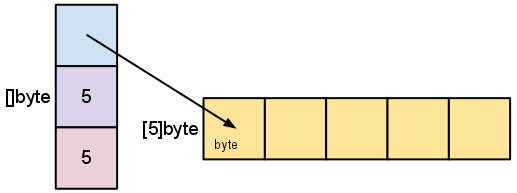
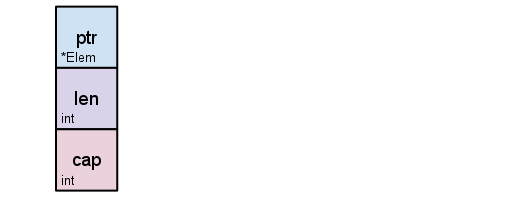
a[1:len(a)] //2,3,4,5

a[1:] //2,3,4,5

a[:3] //1,2,3

切⽚片内部结构

**len: 元素的个数**



**cap: 内部数组的容量量**

## 切⽚片声明

var s0 []int

s0 = append(s0, 1) s := []int{}

s1 := []int{1, 2, 3}

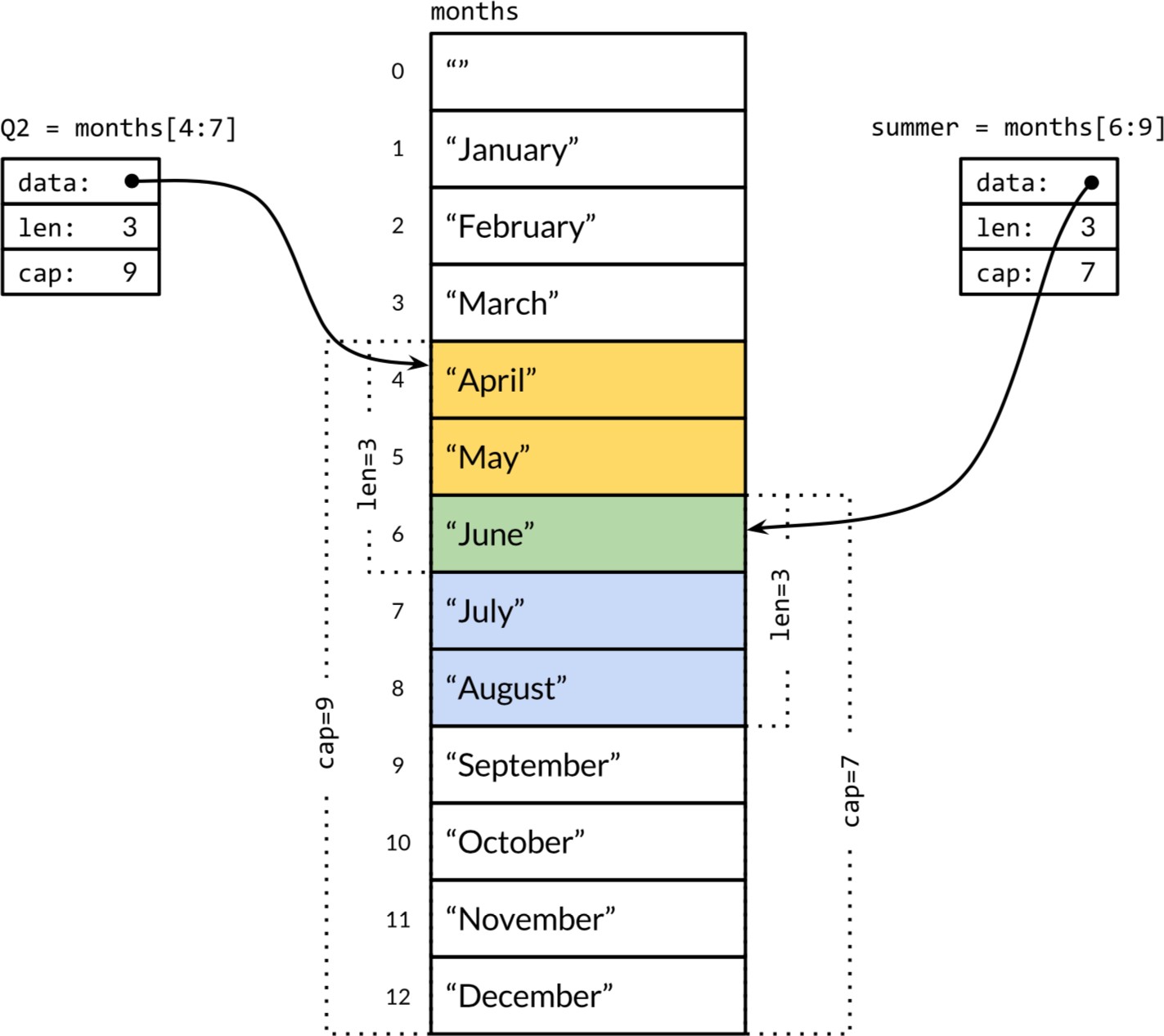
s2 := make([]int, 2, 4)

/\*[]type, len, cap

其中len个元素会被初始化为默认零值，未初始化元素不不可以访问

\*/

## 切⽚片共享存储结构



数组 vs. 切⽚片

1. 容量量是否可伸缩
2. 是否可以进⾏行行⽐比较

# Map 基础

## Map 声明

m := map[string]int{"one": 1, "two": 2, "three": 3} m1 := map[string]int{}

m1["one"] = 1

m2 := make(map[string]int, 10 /\*Initial Capacity\*/)

//为什什么不不初始化len？

## Map 元素的访问

#### 与其他主要编程语⾔言的差异

在访问的 Key 不不存在时，仍会返回零值，不不能通过返回 nil 来判断元素是否存在

if v, ok := m["four"]; ok {

t.Log("four", v)

} else {

t.Log("Not existing")

}

## Map 遍历

m := map[string]int{"one": 1, "two": 2, "three": 3} for k, v := range m {

t.Log(k, v)

}

Map 扩展

Map 与⼯工⼚厂模式

* Map 的 value 可以是⼀一个⽅方法
* 与 Go 的 Dock type 接⼝口⽅方式⼀一起，可以⽅方便便的实现单⼀一⽅方法对象的⼯工⼚厂模式

## 实现 Set

Go 的内置集合中没有 Set 实现， 可以 map[type]bool

1. 元素的唯⼀一性
2. 基本操作
   1. 添加元素
   2. 判断元素是否存在
   3. 删除元素
   4. 元素个数

# 字符串串与字符编码

## 字符串串

#### 与其他主要编程语⾔言的差异

1. string 是数据类型，不不是引⽤用或指针类型
2. string 是只读的 byte slice，len 函数可以它所包含的 byte 数
3. string 的 byte 数组可以存放任何数据

## Unicode UTF8

1. Unicode 是⼀一种字符集（code point）
2. UTF8 是 unicode 的存储实现 （转换为字节序列列的规则）

## 编码与存储

字符

Unicode UTF-8

string/[]byte

“中”

0x4E2D

0xE4B8AD [0xE4,0xB8,0xAD]

常⽤用字符串串函数

1. strings 包 (https://golang.org/pkg/strings/)
2. strconv 包 (https://golang.org/pkg/strconv/)

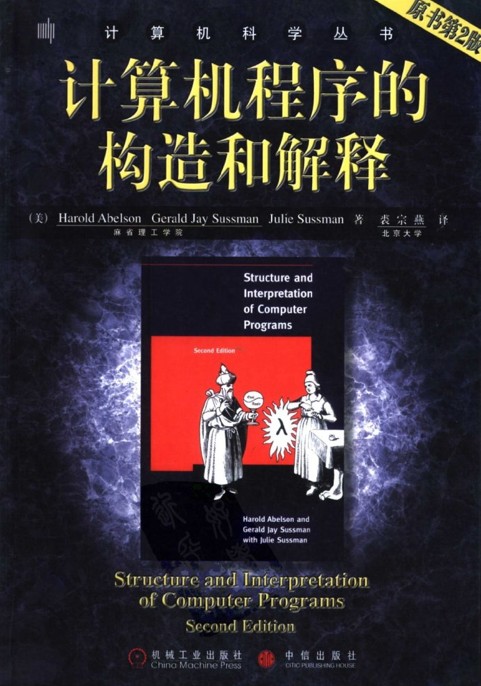
函数：⼀一等公⺠民

函数是⼀一等公⺠民

#### 与其他主要编程语⾔言的差异

1. 可以有多个返回值
2. 所有参数都是值传递：slice，map，channel 会有传引⽤用的错觉
3. 函数可以作为变量量的值
4. 函数可以作为参数和返回值

## 学习函数式编程



函数：可变参数及 defer

可变参数

|  |  |  |  |
| --- | --- | --- | --- |
| func sum(ops | ...int) | int | { |
| s := 0 |  |  |  |
| for \_, op | := range | ops | { |
| s += op |  |  |  |

}

return s

}

defer 函数

func TestDefer(t \*testing.T) { defer func() {

t.Log("Clear resources")

}()

t.Log("Started")

panic("Fatal error”) //defer仍会执⾏行行

}

# ⾯面向对象编程

### Is Go an object-oriented language?

**Yes and no**. Although Go has types and methods and allows an object- oriented style of programming, there is **no type hierarchy**. **The concept of “interface” in Go provides a different approach that we believe is easy to use and in some ways more general.**

Also, the lack of a type hierarchy makes “objects” in Go feel much more lightweight than in languages such as C++ or Java.

https://golang.org/doc/faq

# 封装数据和⾏行行为

## 结构体定义

#### type Employee struct { Id string

Name string

Age int

}

实例例创建及初始化

e := Employee{"0", "Bob", 20}

e1 := Employee{Name: "Mike", Age: 30}

e2 := new(Employee) //注意这⾥里里返回的引⽤用/指针，相当于 e := &Employee{} e2.Id = “2" //与其他主要编程语⾔言的差异：通过实例例的指针访问成员不不需要使⽤用-> e2.Age = 22

e2.Name = “Rose"

## ⾏行行为（⽅方法）定义

#### 与其他主要编程语⾔言的差异

type Employee struct { Id string

Name string

Age int

//第⼀一种定义⽅方式在实例例对应⽅方法被调⽤用时，实例例的成员会进⾏行行值复制func (e Employee) String() string {

return fmt.Sprintf("ID:%s-Name:%s-Age:%d", e.Id, e.Name, e.Age)

}

}

//通常情况下为了了避免内存拷⻉贝我们使⽤用第⼆二种定义⽅方式func (e \*Employee) String() string {

return fmt.Sprintf("ID:%s/Name:%s/Age:%d", e.Id, e.Name, e.Age)

}

# 定义交互协议

接⼝口与依赖 Programmer.java

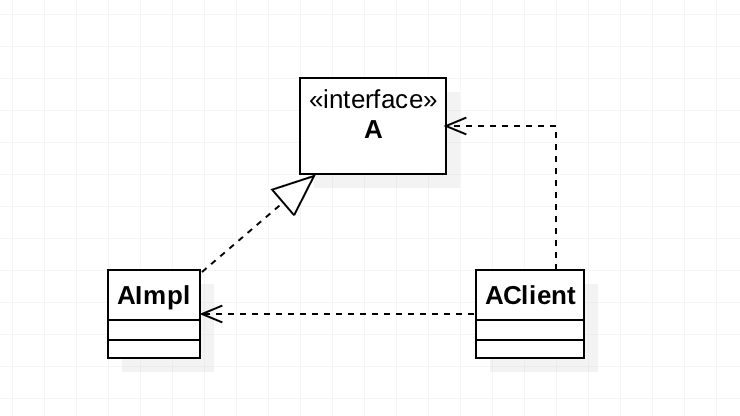
public interface Programmer {

String WriteCodes() ;

}

GoProgrammer.java

public class GoProgrammer implements Programmer { @Override



public String WriteCodes() {

return "fmt.Println(\"Hello World\")";

}

}

Task.java

public class Task{

public static void main(String[] args) { Programmer prog = new GoProgrammer(); String codes = prog.WriteCodes(); System.out.println(codes);

}

}

## Duck Type 式接⼝口实现

##### 接⼝口定义 接⼝口实现

type Programmer interface { WriteHelloWorld() Code

type GoProgrammer struct {

}

}

func (p \*GoProgrammer) WriteHelloWorld() Code { return "fmt.Println(\"Hello World!\")"

}

## Go 接⼝口

#### 与其他主要编程语⾔言的差异

1. 接⼝口为⾮非⼊入侵性，实现不不依赖于借⼝口定义
2. 所以接⼝口的定义可以包含在接⼝口使⽤用者包内

## 接⼝口变量量

var prog Coder = &GoProgrammer{}

type GoProgrammer struct {

|  |  |  |  |
| --- | --- | --- | --- |
| prog  类型 | | |  |
|  | 数据 |  |
|  |  |
|  | | |

}

&GoProgrammer{}

## ⾃自定义类型

1. type IntConvertionFn func(n int) int
2. type MyPoint int

# 扩展与复⽤用

## 复合

#### 与其他主要编程语⾔言的差异

Go 不不⽀支持继承，但可以通过复合的⽅方式来复⽤用

## 匿匿名类型嵌⼊入

#### 与其他主要编程语⾔言的差异

它**不不是继承**，如果我们把“内部 struct ”看作⽗父类，把“外部 struct” 看作⼦子类，

会发现如下问题：

1. 不不⽀支持⼦子类替换
2. ⼦子类并不不是真正继承了了⽗父类的⽅方法
   * ⽗父类的定义的⽅方法⽆无法访问⼦子类的数据和⽅方法

# 多态与空接⼝口

## 多态

type Programmer interface { WriteHelloWorld() Code

}

func writeFirstProgram(p Programmer) { fmt.Printf("%T %v\n", p, p.WriteHelloWorld())

}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | |  | | |
|  |  | | | |  |
| type GoProgrammer struct {  } | |  | | type JavaProgrammer struct {  } | |
| func (p \*GoProgrammer) WriteHelloWorld() Code { return "fmt.Println(\"Hello World!\")"  } | | func (p \*JavaProgrammer) WriteHelloWorld() Code { return "System.out.Println(\"Hello World!\")"  } | |

空接⼝口与断⾔言

1. 空接⼝口可以表示任何类型
2. 通过断⾔言来将空接⼝口转换为制定类型

v, ok := p.(int) //ok=true 时为转换成功

## Go 接⼝口最佳实践

倾向于使⽤用⼩小的接⼝口定义，很多接

⼝口只包含⼀一个⽅方法

type Reader interface {

Read(p []byte) (n int, err error)

}

type Writer interface {

Write(p []byte) (n int, err error)

较⼤大的接⼝口定义，可以由多个⼩小接

⼝口定义组合⽽而成

只依赖于必要功能的最⼩小接⼝口

}

type ReadWriter interface { Reader

Writer

}

func StoreData(reader Reader) error {

…

}

# 编写好的错误处理理

## Go 的错误机制

#### 与其他主要编程语⾔言的差异

1. 没有异常机制
2. error 类型实现了了 error 接⼝口

type error interface { Error() string

1. 可以通过 errors.New 来快速创建错误实例例

}

errors.New("n must be in the range [0,100]")

定义不不同的错误变量量，以便便于判断错误类型

var LessThanTwoError error = errors.New("n must be greater than 2")

var GreaterThanHundredError error = errors.New("n must be less than 100")

**…**

func TestGetFibonacci(t \*testing.T) { var list []int

list, err := GetFibonacci(-10) if err == LessThanTwoError {

t.Error("Need a larger number")

}

if err == GreaterThanHundredError { t.Error("Need a larger number")

}

…

}

及早失败，避免嵌套！

# panic 和 recover

## panic

• panic ⽤用于不不可以恢复的错误

• panic 退出前会执⾏行行 defer 指定的内容

## panic vs. os.Exit

* os.Exit 退出时不不会调⽤用 defer 指定的函数
* os.Exit 退出时不不输出当前调⽤用栈信息

|  |  |  |
| --- | --- | --- |
| Java |  | C++ |
| try{  …  }catch(Throwable | t){ | try{  …  }catch(…){ |
| } |  | } |

defer func() {

if err := recover(); err != nil {

//恢复错误

}

}()



defer func() {

if err := recover(); err != nil { log.Error(“recovered panic”,err)

}

}()

* + 形成僵⼫尸服务进程，导致 health check 失效。
  + “Let it Crash!” 往往是我们恢复不不确定性错误的最好⽅方法。

# 构建可复⽤用模块：包

1. 基本复⽤用模块单元

以⾸首字⺟母⼤大写来表明可被包外代码访问

1. 代码的 package 可以和所在的⽬目录不不⼀一致
2. 同⼀一⽬目录⾥里里的 Go 代码的 package 要保持⼀一致
   1. 通过 go get 来获取远程依赖
      * go get -u 强制从⽹网络更更新远程依赖
   2. 注意代码在 GitHub 上的组织形式，以适应 go get
      * 直接以代码路路径开始，不不要有 src

示例例： https://github.com/easierway/concurrent\_map

## init ⽅方法

* + 在 main 被执⾏行行前，所有依赖的 package 的 init ⽅方法都会被执⾏行行
  + 不不同包的 init 函数按照包导⼊入的依赖关系决定执⾏行行顺序
  + 每个包可以有多个 init 函数
  + 包的每个源⽂文件也可以有多个 init 函数，这点⽐比较特殊

# 依赖管理理

## Go 未解决的依赖问题

1. 同⼀一环境下，不不同项⽬目使⽤用同⼀一包的不不同版本
2. ⽆无法管理理对包的特定版本的依赖

## vendor 路路径

随着 Go 1.5 release 版本的发布，vendor ⽬目录被添加到除了了 GOPATH 和GOROOT 之外的依赖⽬目录查找的解决⽅方案。在 Go 1.6 之前，你需要⼿手动的设置环境变量量

查找依赖包路路径的解决⽅方案如下：

1. 当前包下的 vendor ⽬目录
2. 向上级⽬目录查找，直到找到 src 下的 vendor ⽬目录
3. 在 GOPATH 下⾯面查找依赖包
4. 在 GOROOT ⽬目录下查找

## 常⽤用的依赖管理理⼯工具

godep https://github.com/tools/godep

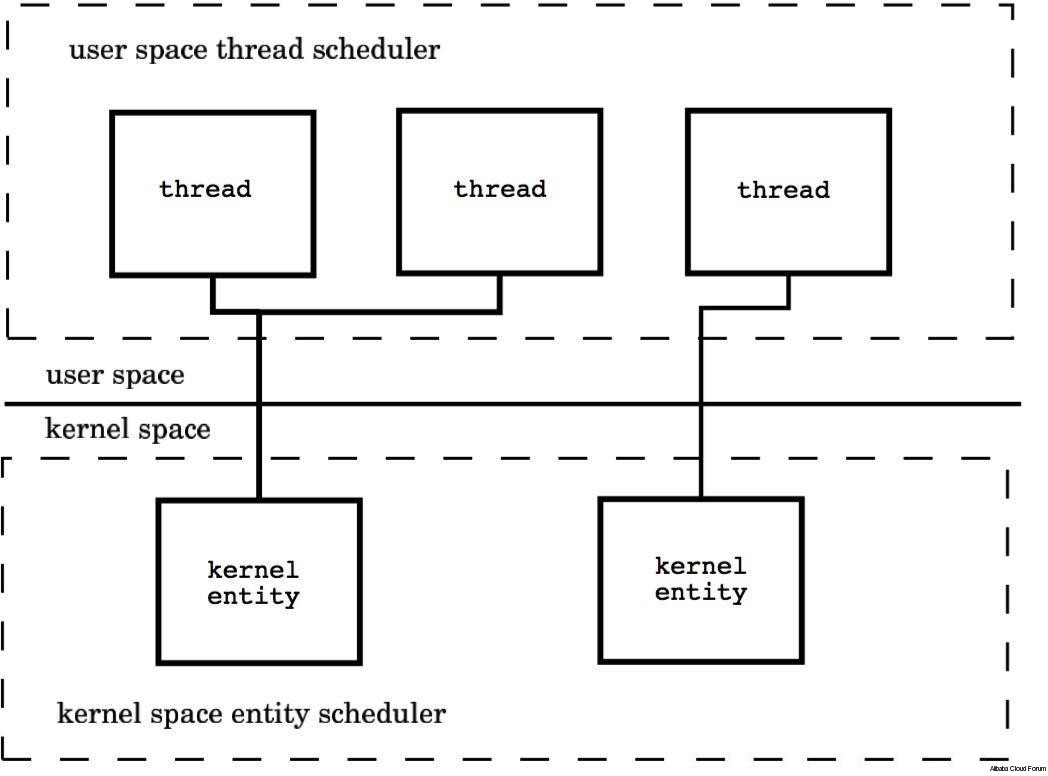
glide https://github.com/Masterminds/glide

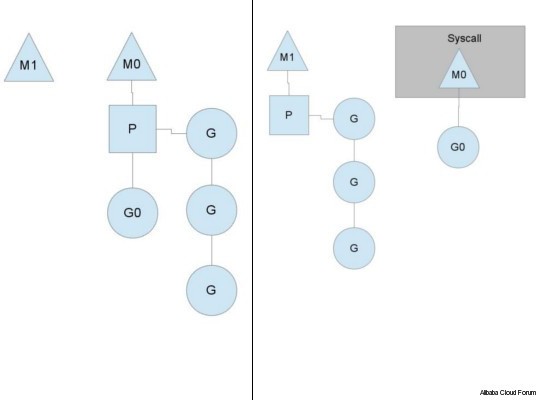
dep https://github.com/golang/dep

# 协程机制

## Thead vs. Groutine

1. 创建时默认的 stack 的⼤大⼩小
   * JDK5 以后 Java Thread stack 默认为1M
   * Groutine 的 Stack 初始化⼤大⼩小为2K
2. 和 KSE （Kernel Space Entity) 的对应关系
   * Java Thread 是 1:1
   * Groutine 是 M:N





**M System Thread P Processor**

**G Goroutine**

# 共享内存并发机制

## Lock

Lock lock = ...;

lock.lock(); try{

**package sync**

Mutex RWLock

//process (thread-safe)

}catch(Exception ex){

}finally{

lock.unlock();

}

## WaitGroup

var wg sync.WaitGroup

for i := 0; i < 5000; i++ { wg.Add(1)

go func() { defer func() {

wg.Done()

}()

…

}()

}

wg.Wait()

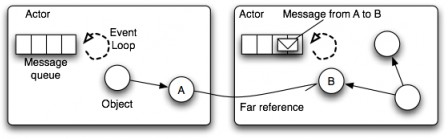
# CSP 并发机制

## CSP



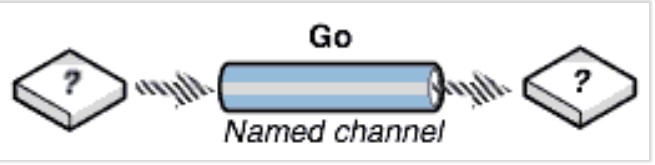
**Tony Hoare**

Actor Model

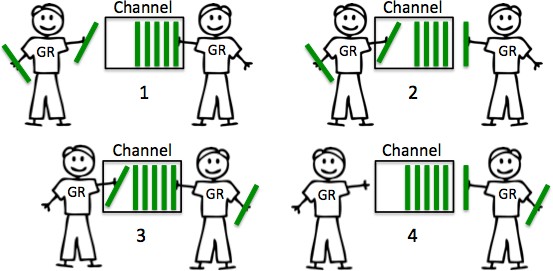
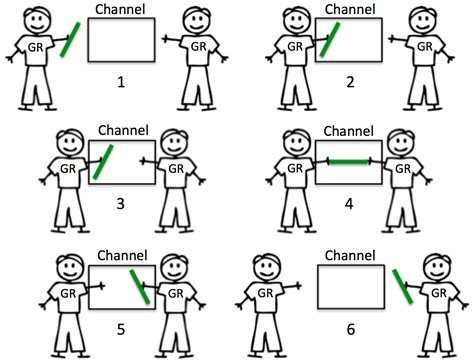


CSP vs. Actor

* + 和Actor的直接通讯不不同，CSP模式则是通过Channel进⾏行行通讯的，更更松耦合⼀一 些。
  + Go中channel是有容量量限制并且独⽴立于处理理Groutine，⽽而如Erlang，Actor模式中的mailbox容量量是⽆无限的，接收进程也总是被动地处理理消息。



## Channel



异步返回

private static FutureTask<String> service() {

FutureTask<String> task = new FutureTask<String>(()->"Do something");

new Thread(task).start();

return task;

}

FutureTask<String> ret = service(); System.out.println("Do something else”); System.out.println(ret.get());

# 多路路选择和超时控制

## select

#### 多渠道的选择 超时控制

select {

case ret := <-retCh1: t.Logf("result %s", ret)

case ret :=<-retCh2:

t.Logf("result %s", ret)

default:

t.Error(“No one returned”)

select {

case ret := <-retCh: t.Logf("result %s", ret)

case <-time.After(time.Second \* 1): t.Error("time out")

}

}

# channel 的关闭和⼴广播

## channel 的关闭

* + 向关闭的 channel 发送数据，会导致 panic
  + v, ok <-ch; ok 为 bool 值，true 表示正常接受，false 表示通道关闭
  + 所有的 channel 接收者都会在 channel 关闭时，⽴立刻从阻塞等待中返回且上述 ok 值为 false。这个⼴广播机制常被利利⽤用，进⾏行行向多个订阅者同时发送信号。如：退出信号。

# 任务的取消

## 获取取消通知

func isCancelled(cancelChan chan struct{}) bool { select {

case <-cancelChan: return true

default:

return false

}

}

## 发送取消消息

func cancel\_1(cancelChan chan struct{}) { cancelChan <- struct{}{}

}

## 通过关闭 Channel 取消

func cancel\_2(cancelChan chan struct{}) { close(cancelChan)

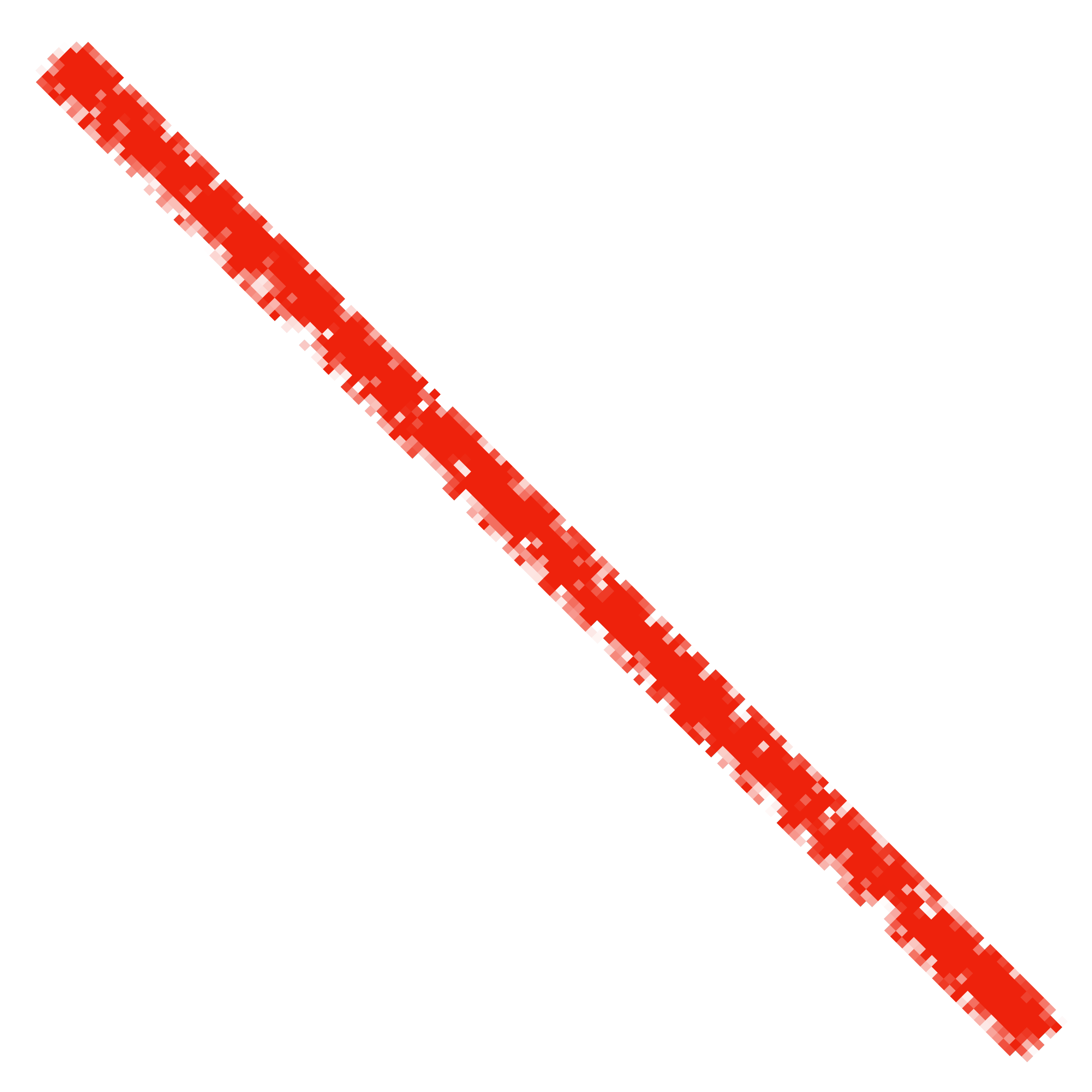
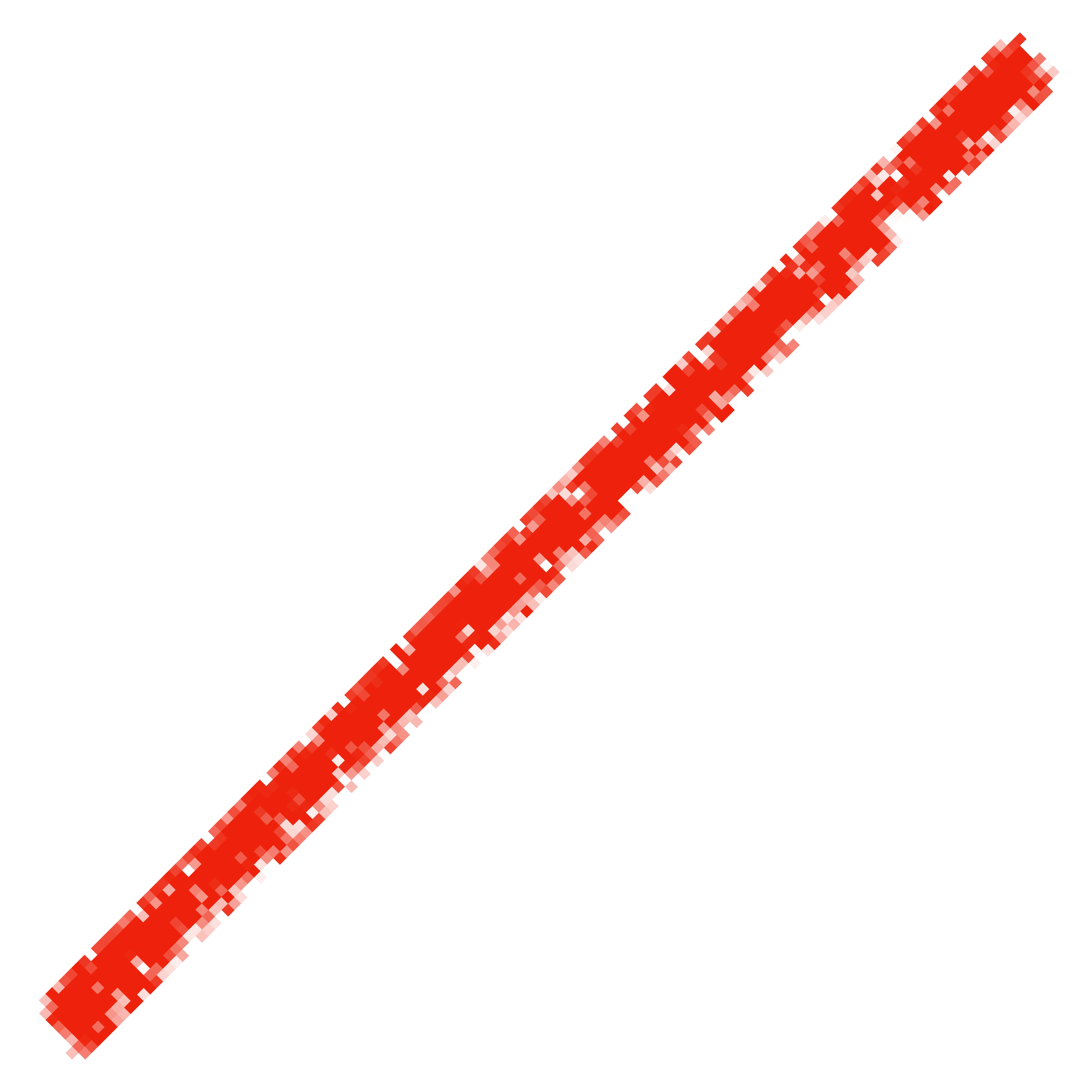
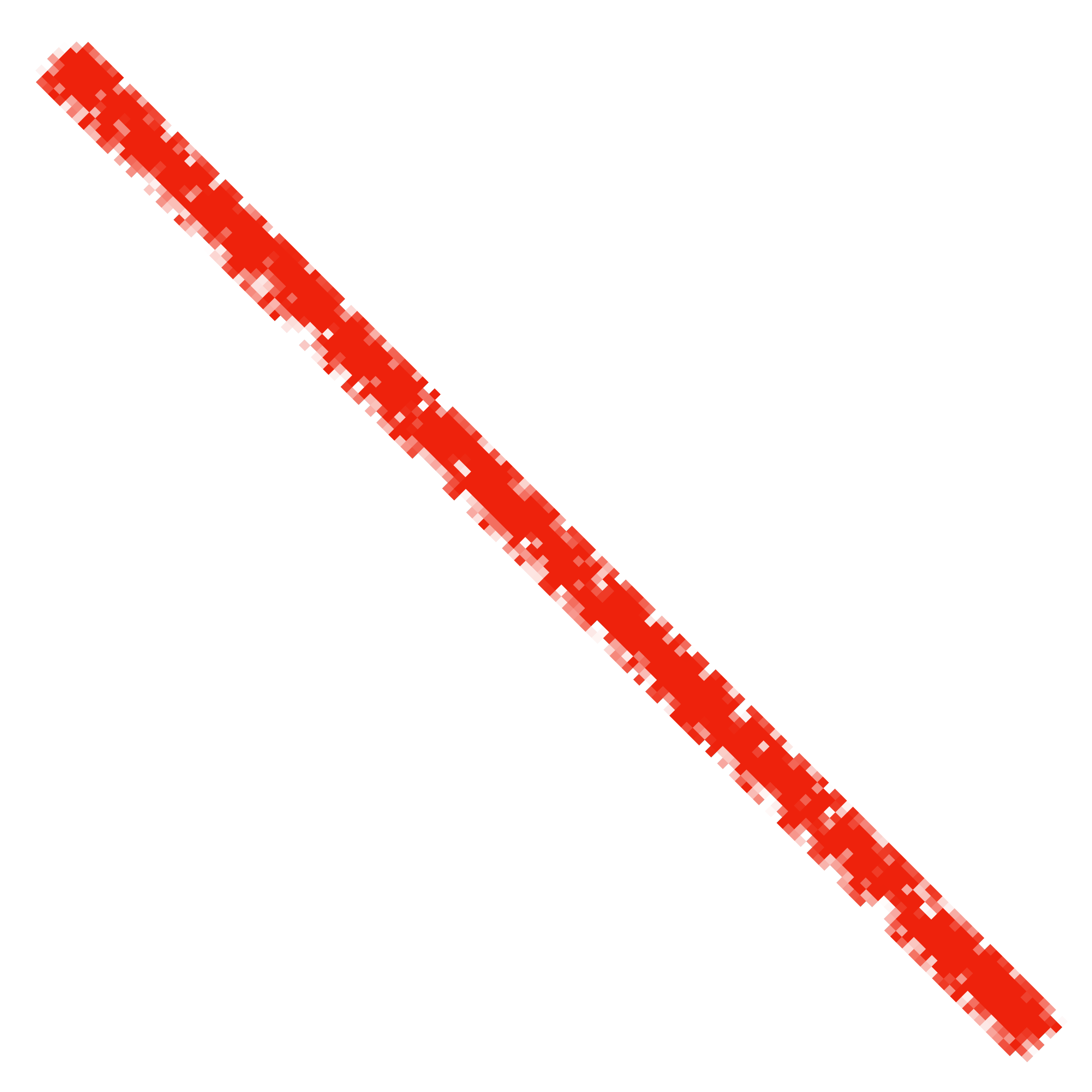
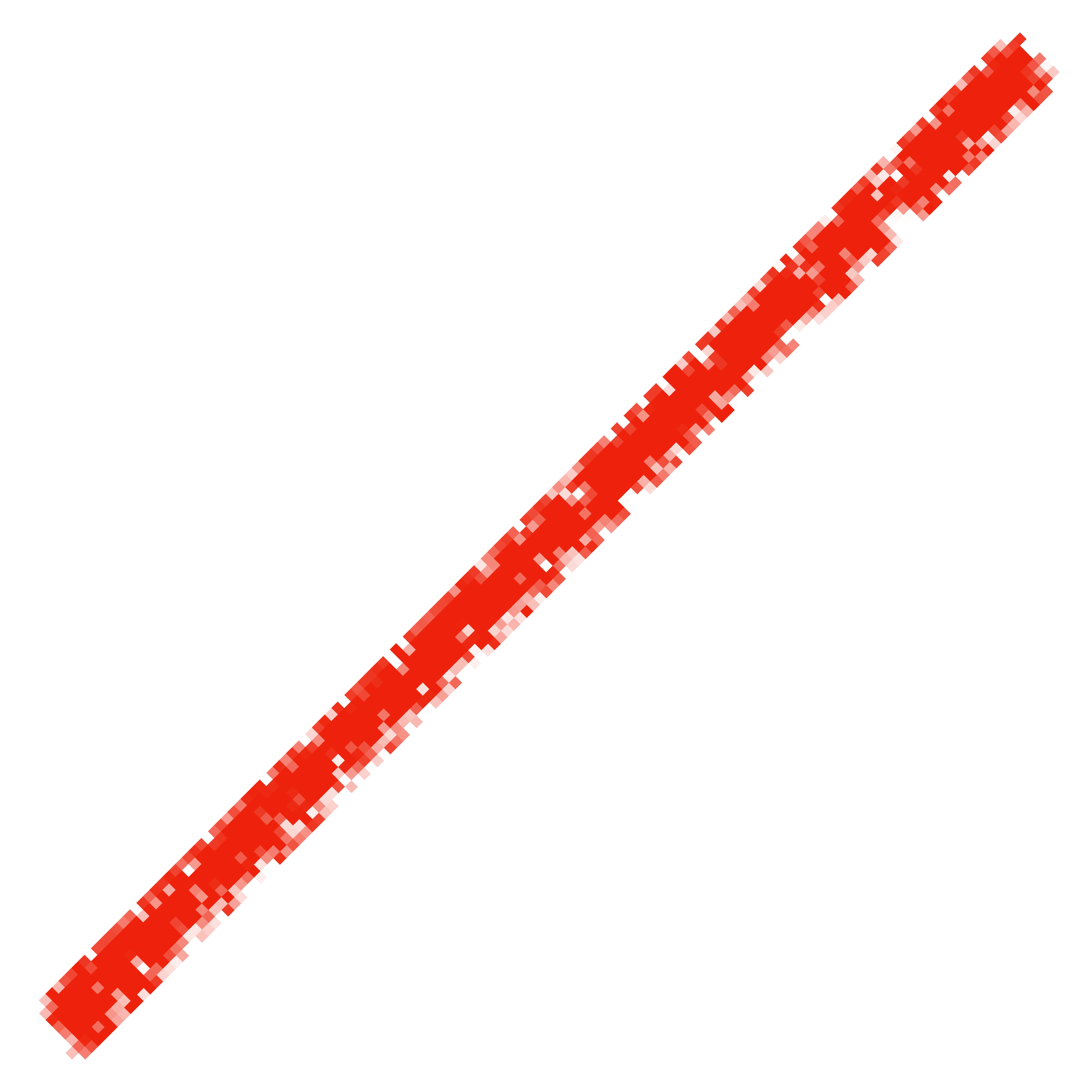
}

# Context 与任务取消

## 关联任务的取消

Main

Handle(Req1) Handle(Req2)



Search(B)

Search(A)

Search(B)

Search(C)

Search(A)

Search(C)

## Context

* + 根 Context：通过 context.Background () 创建
  + ⼦子 Context：context.WithCancel(parentContext) 创建
    - ctx, cancel := context.WithCancel(context.Background())
  + 当前 Context 被取消时，基于他的⼦子 context 都会被取消
  + 接收取消通知 <-ctx.Done()

# 常⻅见并发任务

仅执⾏行行⼀一次

public class Singleton {

private static Singleton INSTANCE=null; private Singleton(){}

public static Singleton getIntance(){

if(INSTANCE==null){ synchronized (Singleton.class){

if(INSTANCE==null){

INSTANCE = new Singleton();

}

}

}

return INSTANCE;

}

}

**var** once sync.Once

**var** obj \*SingletonObj

**func** GetSingletonObj() \*SingletonObj { once.Do(func() {

fmt.Println("Create Singleton obj.")

obj = &SingletonObj{}

})

**return** obj

}

# 仅需任意任务完成

必需所有任务完成

对象池

## 使⽤用buﬀered channel实现对象池

归还对象

获取对象

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

# sync.Pool 对象缓存

## sync.Pool 对象获取

* + - 尝试从私有对象获取
    - 私有对象不不存在，尝试从当前 Processor 的共享池获取
    - 如果当前 Processor 共享池也是空的，那么就尝试去其他

Processor 的共享池获取

* + - 如果所有⼦子池都是空的，最后就⽤用⽤用户指定的 New 函数产⽣生⼀一个新的对象返回

Processor Processor

Processor

共享池

私有对象

**协程安全 协程不不安全**

## sync.Pool 对象的放回

* + - 如果私有对象不不存在则保存为私有对象

Processor Processor

Processor

共享池

私有对象

* + - 如果私有对象存在，放⼊入当前 Processor ⼦子池的共享池中

**协程安全 协程不不安全**

## 使⽤用 sync.Pool

pool := &sync.Pool{

New: func() interface{} { return 0

},

}

arry := pool.Get().(int)

… pool.Put(10)

## sync.Pool 对象的⽣生命周期

* + - GC 会清除 sync.pool 缓存的对象
    - 对象的缓存有效期为下⼀一次GC 之前

## sync.Pool 总结

* + - 适合于通过复⽤用，降低复杂对象的创建和 GC 代价
    - 协程安全，会有锁的开销
    - ⽣生命周期受 GC 影响，不不适合于做连接池等，需⾃自⼰己管理理⽣生命周期的资源的池化

# 测试

单元测试

* + - Fail, Error: 该测试失败，该测试继续，其他测试继续执⾏行行
    - FailNow, Fatal: 该测试失败，该测试中⽌止，其他测试继续执⾏行行
    - 代码覆盖率

go test -v - cover

* + - 断⾔言

https://github.com/stretchr/testify

# Benchmark

func BenchmarkConcatStringByAdd(b \*testing.B) {

//与性能测试⽆无关的代码b.ResetTimer()

for i := 0; i < b.N; i++ {

//测试代码

}

b.StopTimer()

//与性能测试⽆无关的代码

}

go test -bench=. -benchmem

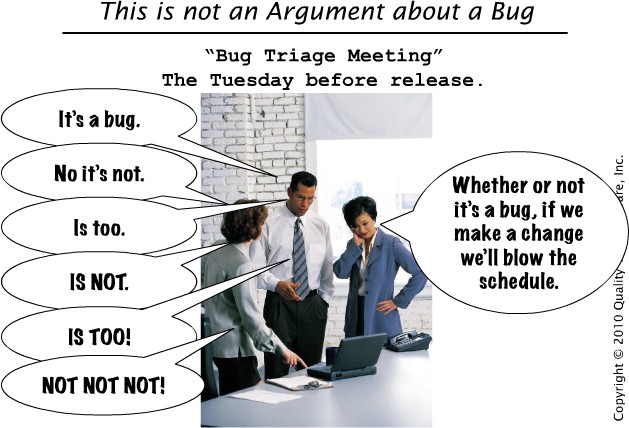


-bench=<相关benchmark测试>

Windows 下使⽤用 go test 命令⾏行行时，-bench=.应写为-bench="."

# Behavior Driven Development

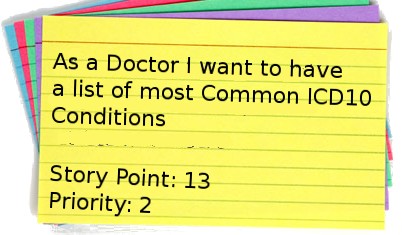
## 让业务领域的专家参与开发



*“I believe that the hardest part of software projects, the most common source of project failure, is* ***communication*** *with the customers and users of that software. By providing a clear yet precise language to deal with domains, a DSL can help improve this communication.” – Martin Fowler.*

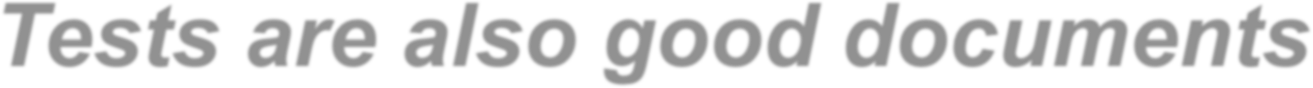
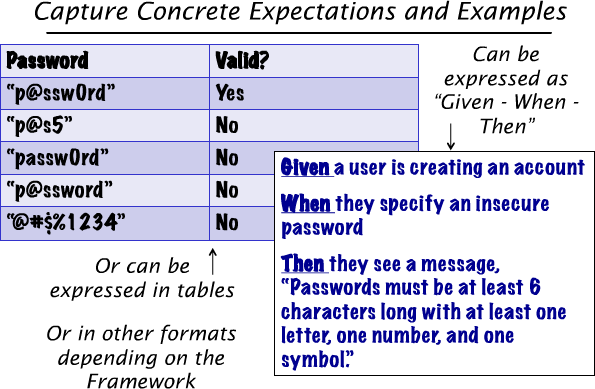


你知道 *Story Card* 背⾯面应该写什什么吗？



What’s on the back of the cards?

## ⽤用业务领域的语⾔言来描述



***Tests are also good documents***

BDD in Go

项⽬目⽹网站

https://github.com/smartystreets/goconvey

安装

go get -u github.com/smartystreets/goconvey/convey

启动 WEB UI

$GOPATH/bin/goconvey

# 反射编程

## reflect.TypeOf vs. reflect.ValueOf

* + reflect.TypeOf 返回类型 (reflect.Type)
  + reflect.ValueOf 返回值 (reflect.Value)
  + 可以从 reflect.Value 获得类型
  + 通过 kind 的来判断类型

## 判断类型—Kind()

const (

Invalid Kind = iota Bool

Int Int8 Int16 Int32 Int64 Uint Uint8 Uint16 Uint32 Uint64

…)

## 利利⽤用反射编写灵活的代码

按名字访问结构的成员

reflect.ValueOf(\*e).FieldByName("Name")

按名字访问结构的⽅方法

reflect.ValueOf(e).MethodByName("UpdateAge").Call([]reflect.Value{reflect.ValueOf(1)})

## Struct Tag

type BasicInfo struct {

Name string `json:"name"` Age int `json:"age"`

*Struct Tag*

}

访问 StructTag

if nameField, ok := reflect.TypeOf(\*e).FieldByName("Name"); !ok { t.Error("Failed to get 'Name' field.")

} else {

t.Log("Tag:format", nameField.Tag.Get("format"))

}

Reflect.Type 和 Reflect.Value 都有 FieldByName ⽅方法，注意他们的区别

# “万能”程序

## 关于“反射”你应该知道的

* + 提⾼高了了程序的灵活性
  + 降低了了程序的可读性
  + 降低了了程序的性能

# “不不安全”编程

## “不不安全”⾏行行为的危险性

i := 10

f := \*(\*float64)(unsafe.Pointer(&i))



合理理的冒险

* + 合理理类型转化
  + 原⼦子操作 (atomic)



# 构建⾼高可扩展性的软件架构

## 架构模式

An architectural pattern is a general, **reusable solution** to a commonly occurring problem in software architecture within a given context.

— wikipedia

Pipe-Filter

Pipe-Filter 架构

Ads SSP

Filling

Ranking

Filtering

Verification

Params Processing

Request Parsing

Architecture Style

-- Pipe-Filter Pattern

Pump

Sink

Filter

Filter

Filter

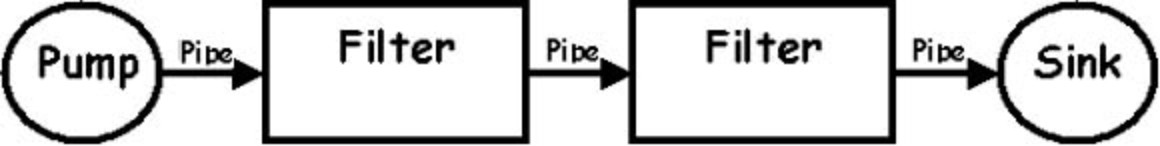
Filter

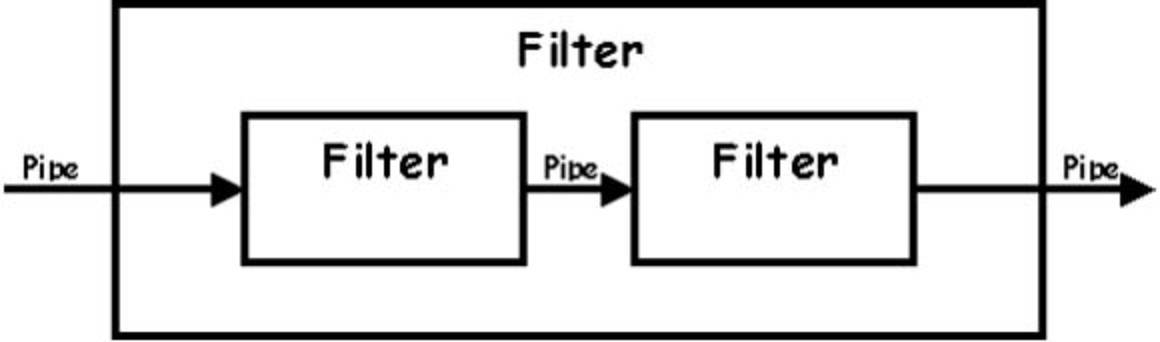
Filter

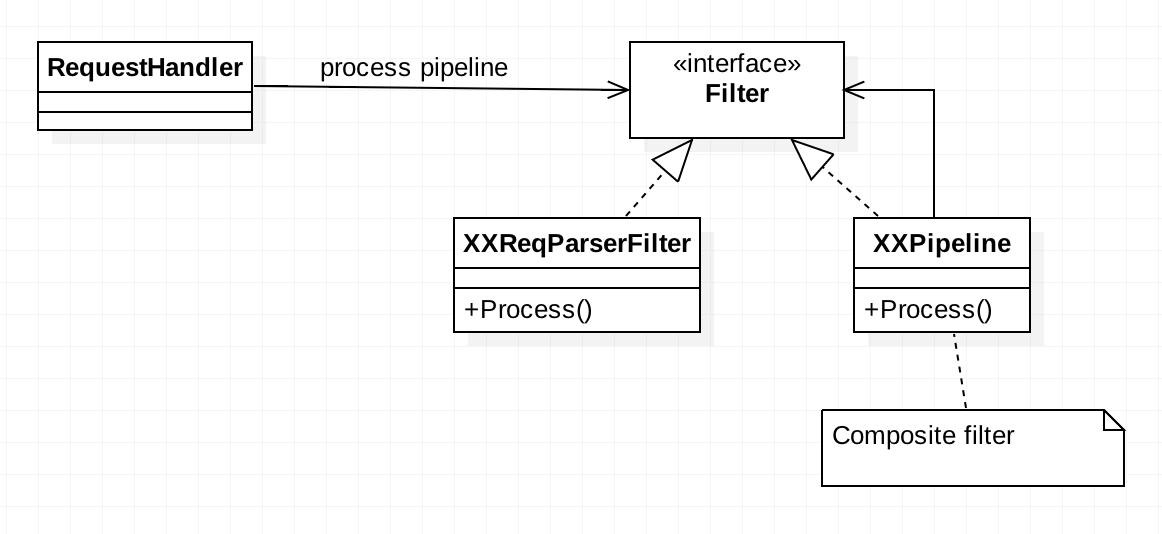
Pipe-Filter 模式

* + ⾮非常适合与数据处理理及数据分析系统
  + Filter封装数据处理理的功能
  + Pipe⽤用于连接Filter传递数据或者在异步处理理过程中缓冲数据 流
    - 进程内同步调⽤用时，pipe演变为数据在⽅方法调⽤用间传递
  + 松耦合：Filter只跟数据（格式）耦合

## Filter和组合模式







示例例

**“1,2,3”**

SplitFilter

**[“1”,”2”,”3”]**

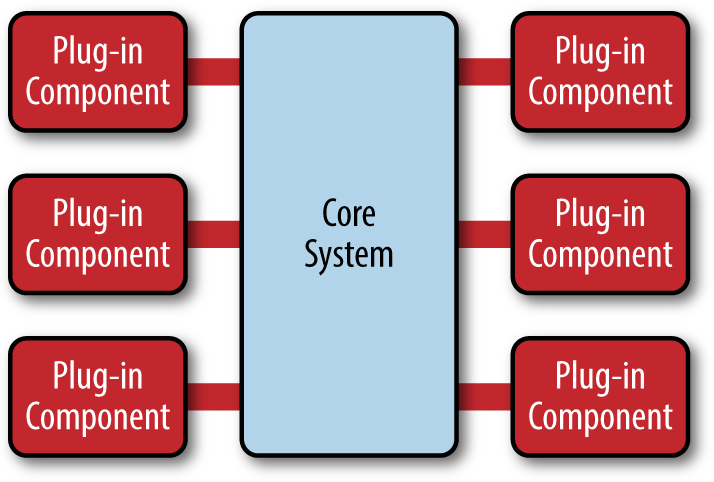
**2,3]**

|  |  |
| --- | --- |
| ToIntFilter | |
|  | **[1,** |
| SumFilter | |
|  |  |

**6**

# Micro-Kernel

## Micro Kernel

* + - 特点
      * 易易于扩展
      * 错误隔离
      * 保持架构⼀一致性
    - 要点
      * 内核包含公共流程或通⽤用逻辑
      * 将可变或可扩展部分规划为扩展点
      * 抽象扩展点⾏行行为，定义接⼝口
      * 利利⽤用插件进⾏行行扩展

## 示例例

**Extension Point**

<<Kernel>>

Agent

**…**

<<Plugin>>

AppCollector

<<Plugin>> FileCollector

<<Plugin>> ProcessCollector

# 常⻅见任务

JSON 解析

## 内置的JSON解析

利利⽤用反射实现，通过FeildTag来标识对应的json 值

**type** BasicInfo **struct** { Name string `json:"name"` Age int `json:"age"`

}

**type** JobInfo **struct** {

Skills []string `json:"skills"`

}

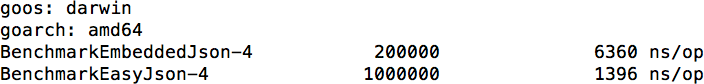
**type** Employee **struct** {

BasicInfo BasicInfo `json:"basic\_info"` JobInfo JobInfo `json:"job\_info"`

}

## 更更快的JSON解析

EasyJSON 采⽤用代码⽣生成⽽而⾮非反射



安装

go get -u github.com/mailru/easyjson/...

使⽤用

easyjson -all <结构定义>.go

# HTTP Server

## Handler

type Handler interface {

ServeHTTP(ResponseWriter, \*Request)

}

## 路路由规则

* + URL 分为两种，末尾是 /：表示⼀一个⼦子树，后⾯面可以跟其他⼦子路路径； 末尾不不是 /，表示⼀一个叶⼦子，固定的路路径
    - 以/ 结尾的 URL 可以匹配它的任何⼦子路路径，⽐比如 /images 会匹配 /images/ cute-cat.jpg
  + 它采⽤用最⻓长匹配原则，如果有多个匹配，⼀一定采⽤用匹配路路径最⻓长的那个进⾏行行处 理理
  + 如果没有找到任何匹配项，会返回 404 错误

## Default Router

func (sh serverHandler) ServeHTTP(rw ResponseWriter, req \*Request) { handler := sh.srv.Handler

if handler == nil {

handler = DefaultServeMux //使⽤用缺省的Router

}

if req.RequestURI == "\*" && req.Method == "OPTIONS" { handler = globalOptionsHandler{}

}

handler.ServeHTTP(rw, req)

}

## 更更好的 Router

https://github.com/julienschmidt/httprouter

func Hello(w http.ResponseWriter, r \*http.Request, ps httprouter.Params) { fmt.Fprintf(w, "hello, %s!\n", ps.ByName("name"))

}

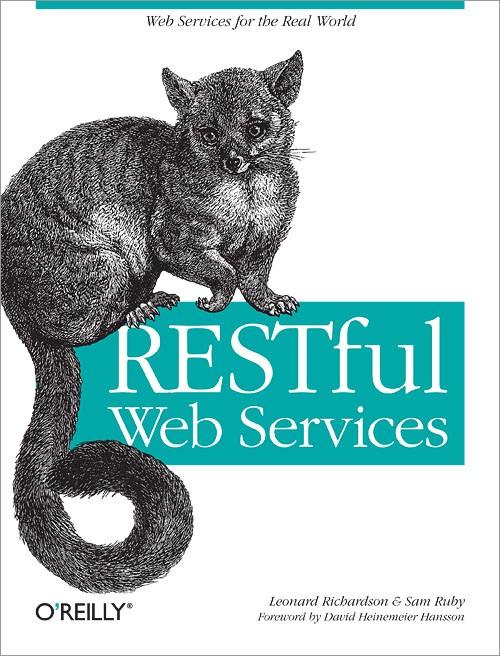
func main() {

router := httprouter.New() router.GET("/", Index) router.GET("/hello/:name", Hello)

log.Fatal(http.ListenAndServe(":8080", router))

}

## ⾯面向资源的架构（Resource Oriented Architecture）

In software engineering, a **resource**-**oriented architecture** (ROA) is a style of software **architecture** and programming paradigm for supportive designing and developing software in the form of Internetworking of **resources** with "RESTful" interfaces.

**www.demo\_portal.com/employee/*{name}***

# 性能分析

性能分析⼯工具

## 准备⼯工作

* + 安装 graphviz
    - brew install graphviz
  + 将 $GOPATH/bin 加⼊入 $PATH
    - Mac OS: 在 .bash\_profile 中修改路路径
  + 安装 go-torch
    - go get github.com/uber/go-torch
    - 下载并复制 flamegraph.pl （https://github.com/brendangregg/FlameGraph）⾄至 $GOPATH/bin 路路径下
    - 将 $GOPATH/bin 加⼊入 $PATH

## 通过⽂文件⽅方式输出 Profile

* + 灵活性⾼高，适⽤用于特定代码段的分析
  + 通过⼿手动调⽤用 runtime/pprof 的 API
  + API 相关⽂文档 https://studygolang.com/static/pkgdoc/pkg/runtime\_pprof.htm
  + go tool pprof [binary] [binary.prof]

## 通过 HTTP ⽅方式输出 Profile

* 简单，适合于持续性运⾏行行的应⽤用
* 在应⽤用程序中导⼊入 import \_ "net/http/pprof"，并启动 http server 即可
* http://<host>:<port>/debug/pprof/
* go tool pprof http://<host>:<port>/debug/pprof/profile?seconds=10 （默认值为30秒）
* go-torch -seconds 10 http://<host>:<port>/debug/pprof/profile

## Go ⽀支持的多种 Profile

go help testflag https://golang.org/src/runtime/pprof/pprof.go

性能调优示例例

性能调优过程

S

优化瓶颈点

E

分析系统瓶颈点

设定优化⽬目标

常⻅见分析指标

* + Wall Time
  + CPU Time
  + Block Time
  + Memory allocation
  + GC times/time spent

go test 输出 profile

|  |  |  |
| --- | --- | --- |
| go | test | -bench=. -cpuprofile=cpu.prof |
| go | test | -bench=. -blockprofile=block.prof |
| go | tool | pprof cpu.prof |

go help testflag

示例例

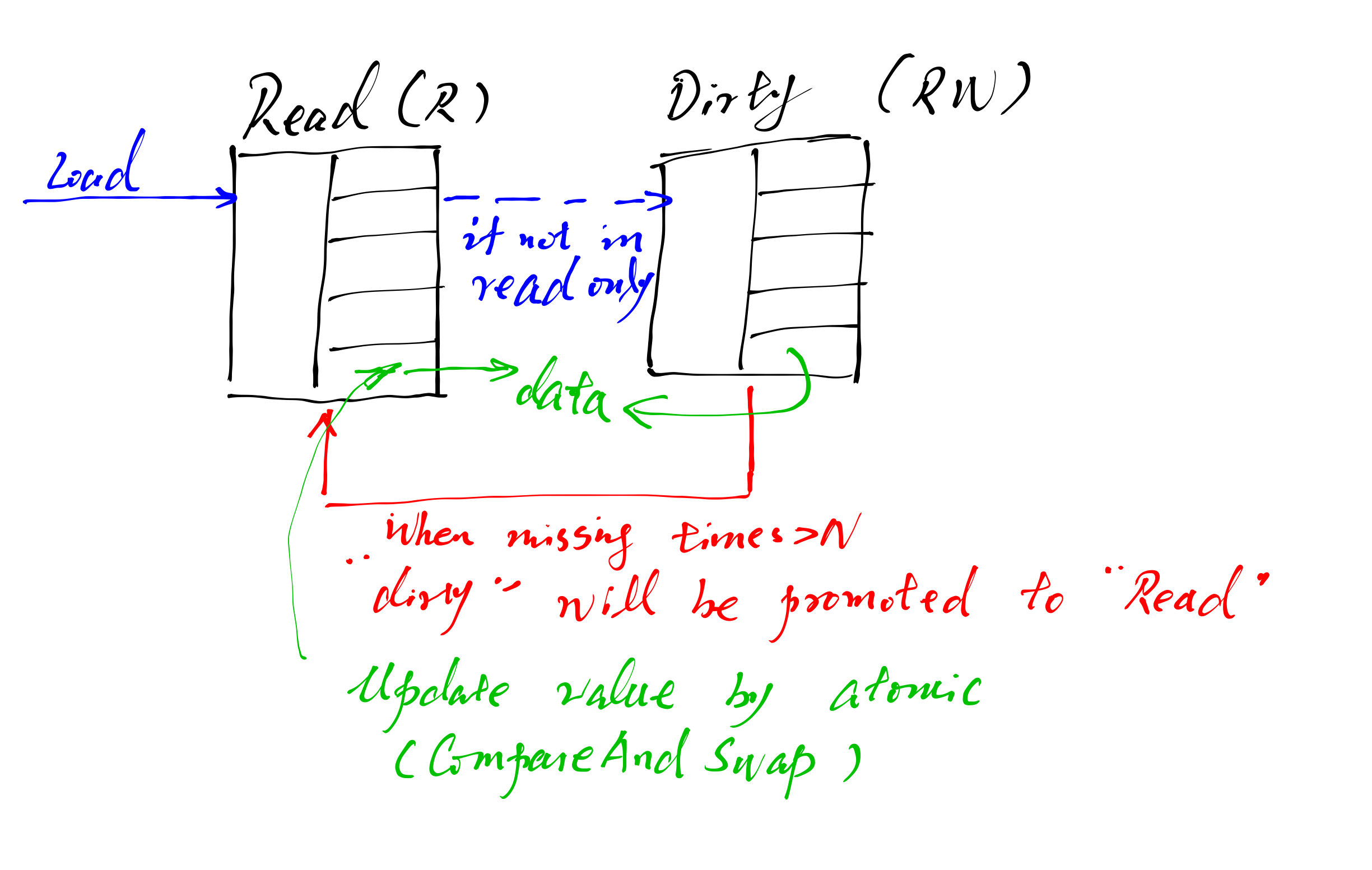
go test -bench=. -cpuprofile=cpu.prof go tool pprof cpu.prof

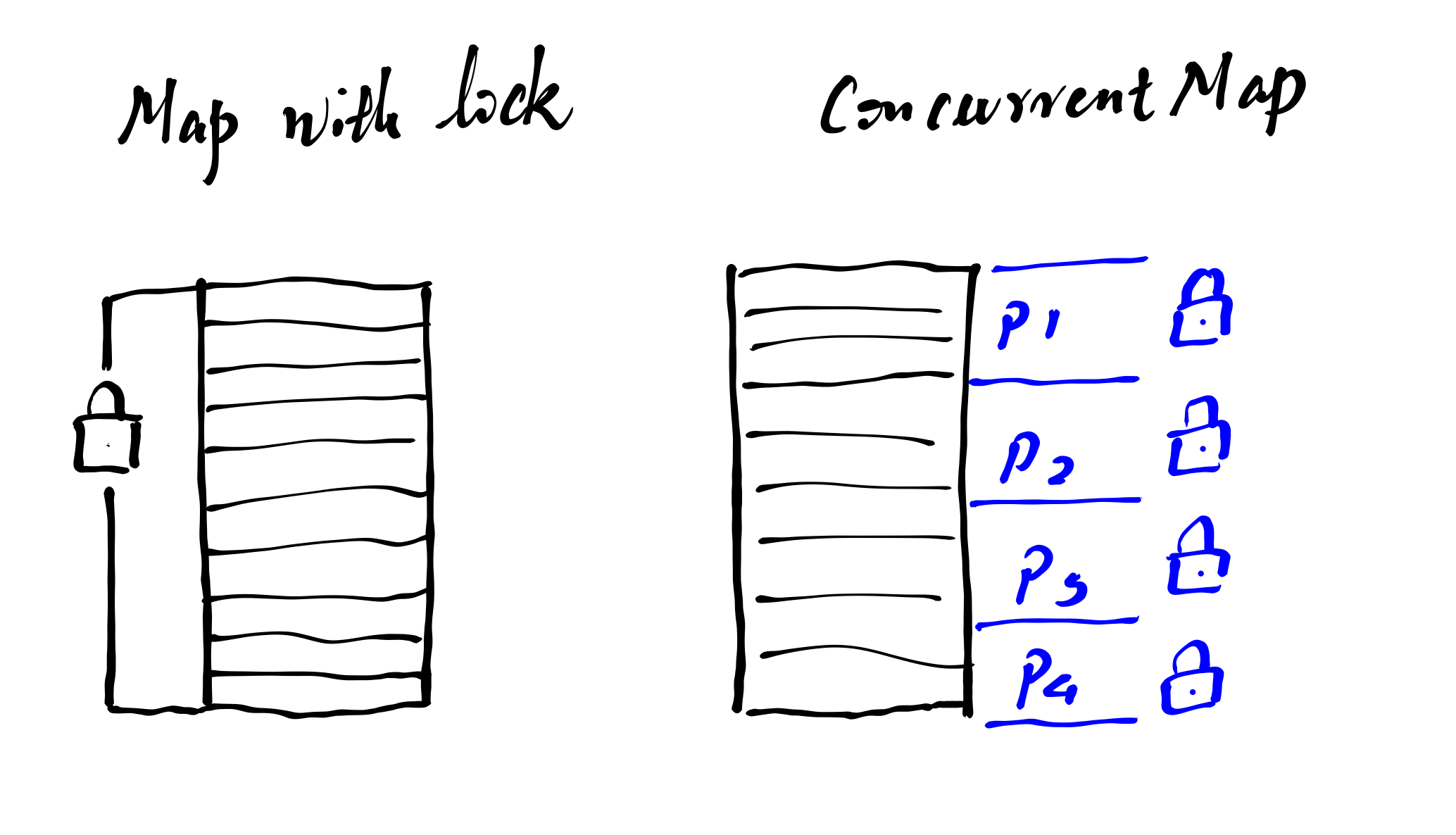
go-torch cpu.prof

# 编写⾼高性能的Go程序

别让性能被“锁”住

* 适合读多写少，且 Key 相对稳定的环境
* 采⽤用了了空间换时间的⽅方案，并且采⽤用指针的⽅方式间接实现值的映射，所以存储空间会较 built-in map ⼤大

**https://my.oschina.net/qiangmzsx/blog/1827059**

* 适⽤用于读写都很频繁的情况



**https://github.com/easierway/concurrent\_map**

## 别让性能被“锁”住

* + 减少锁的影响范围
  + 减少发⽣生锁冲突的概率
    - sync.Map
    - ConcurrentMap
  + 避免锁的使⽤用
    - LAMX Disruptor：https://martinfowler.com/articles/lmax.html

# GC 友好的代码

## 避免内存分配和复制

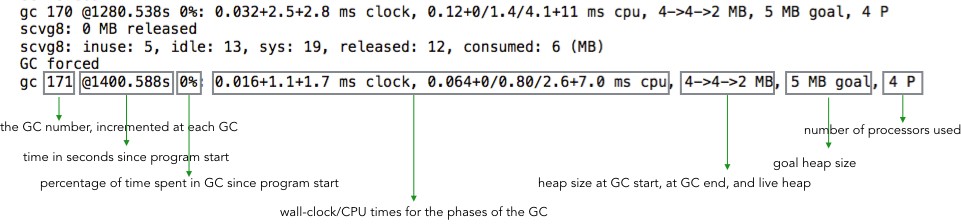
* 复杂对象尽量量传递引⽤用
  + 数组的传递
  + 结构体传递

## 打开 GC ⽇日志

只要在程序执⾏行行之前加上环境变量量 GODEBUG=gctrace=1 ， 如：GODEBUG=gctrace=1 go test -bench=.

GODEBUG=gctrace=1 go run main.go

##### ⽇日志详细信息参考： https://godoc.org/runtime



go tool trace

**普通程序输出 trace 信息 测试程序输出 trace 信息**

package main import (

"os"

"runtime/trace"

go test -trace trace.out

)

func main() {

f, err := os.Create("trace.out") if err != nil {

panic(err)

**可视化 trace 信息**

go tool trace trace.out

}

defer f.Close()

err = trace.Start(f) if err != nil {

panic(err)

}

defer trace.Stop()

// Your program here

}

## 避免内存分配和复制

* 初始化⾄至合适的⼤大⼩小
  + ⾃自动扩容是有代价的
* 复⽤用内存

# ⾼高效的字符串串连接

⾼高可⽤用性架构设计

⾯面向错误的设计

“Once you accept that failures will happen, you have the ability to design your system’s reaction to the failures.”

隔离

隔离错误 — 设计

**Extension Point**

**Extension Point**

**Extension Point**

Micro Kernel

**Extension Point**

**Extension Point**

**Extension Point**

plug-in

plug-in

plug-in

plug-in

plug-in

plug-in

plug-in

plug-in

plug-in

plug-in

plug-in

plug-in

隔离错误 — 部署

Cart Service

Ordering Service

Checkout Service

Payment Service

Fulfillment Service

重⽤用 vs 隔离

**Critical Data Collector Log Data Collector**

Data Transform Service

Log Data Collector

Critical Data Collector

**Data Transform Service Data Transform Service**

逻辑结构的重⽤用 vs 部署结构的隔离

冗余

冗余

Load Balancer

Load Balancer

**Online Service Standby Service Online Redundancy**

## 单点失效

**QPS 1500**

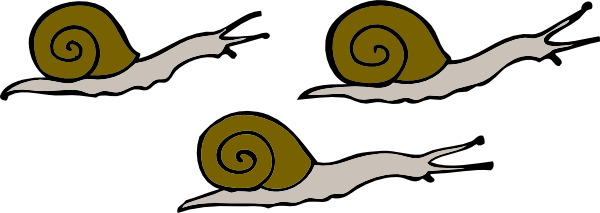
Load Balancer

**Max QPS 1000**

## 限流

慢响应

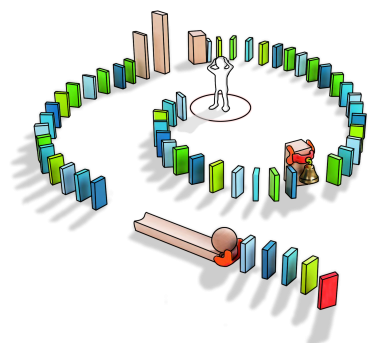
### A quick rejection is better than a slow response.

*Pooled resources are exhausted!*

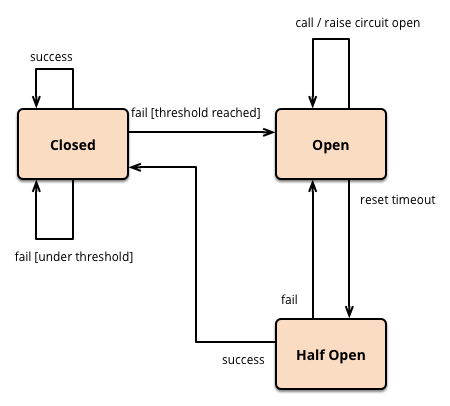
## 不不要⽆无休⽌止的等待



给阻塞操作都加上⼀一个期限

错误传递

断路路器器



⾯面向恢复的设计

“A priori prediction of all failure modes is not possible.”

健康检查

* 注意僵⼫尸进程
  + 池化资源耗尽
  + 死锁

## Let it Crash!

defer func() {

if err := recover(); err != nil { log.Error(“recovered panic”,err)

}

}()

## 构建可恢复的系统

* 拒绝单体系统
* ⾯面向错误和恢复的设计
  + 在依赖服务不不可⽤用时，可以继续存活
  + 快速启动
  + ⽆无状态

## 与客户端协商

服务器器: “我太忙了了，请慢点发送数据” Client: “好，我⼀一分钟后再发送”



# Chaos Engineering

“If something hurts, do it more often!”

* 如果问题经常发⽣生⼈人们就会学习和思考解决它的⽅方法

Chaos under control

**Inject latency**

**Terminate host**

**Inject failure**

Chaos Engineering 原则

* + Build a Hypothesis around Steady State Behavior
  + Vary Real-world Events
  + Run Experiments in Production
  + Automate Experiments to Run Continuously
  + Minimize Blast Radius

[**http://principlesofchaos.org**](http://principlesofchaos.org/)

相关开源项⽬目

https://github.com/Netflix/chaosmonkey

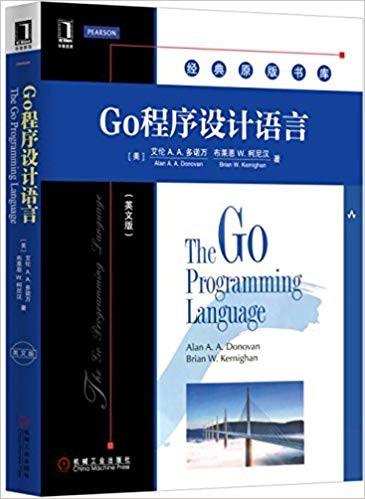
https://github.com/easierway/service\_decorators/blob/master/README.md

# 结束语

是结束，更更是开始！

The master has failed more times than the beginner has tried.

图书推荐







**扫码订阅/试看**

**《Go语⾔言从⼊入⻔门到实战》**