

# Go 语言从入门到实战



扫码订阅/试看

《Go语言言从入门到实战》

# 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
	常见见并发任务
	深入入测试
	反射和 Unsafe
	常见见架构模式的实现
	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



Rob Pike  
Unix 的早期开发者  
UTF-8 创始人



Ken Thompson  
Unix 的创始人  
C 语言创始人  
1983 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

C++

37

25

84



高高效

垃圾回收

指针

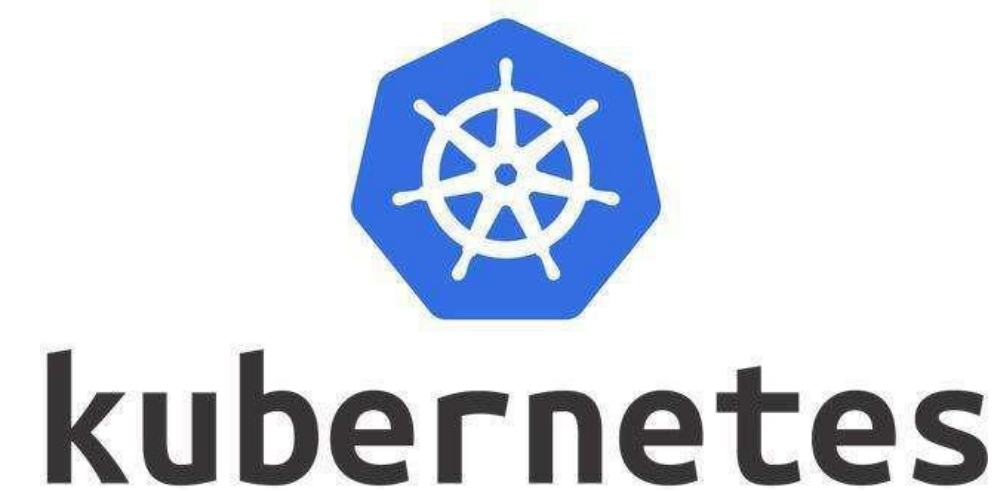
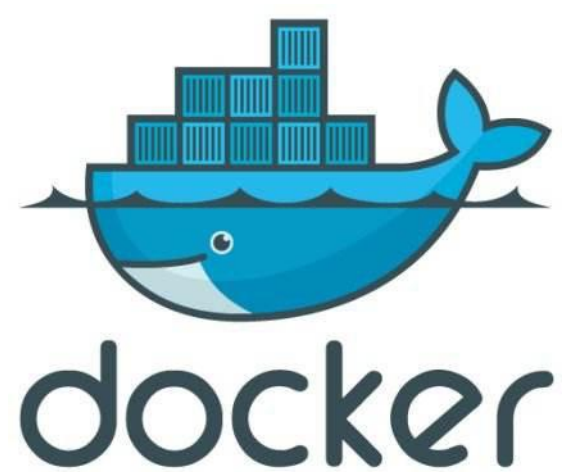
生产力

复合

**VS**

继承

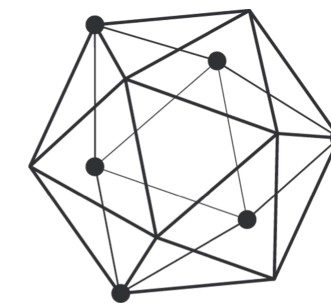
# 云计算语言言



# 区块链语言言



ethereum  
Homestead Documentation



**HYPERLEDGER**

# 准备开始 Go 冒险之旅

下载安装 Go 语言

<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
2. 1.8 版本后（含 1.8）如果没有设置使用默认值

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" //引入代码依赖
```

```
//功能实现
```

```
func MAIN() {  
    fmt.Println("Hello World!")  
}
```



# 应用程序入口

1. 必须是 **main** 包: `PACKAGE MAIN`
2. 必须是 **main** 方法: `func MAIN()`
3. 文件名不一定是 `MAIN.GO`

# 退出返回值

## 与其他主要编程语言言的差异

- Go 中 `MAIN` 函数不不支持任何返回值
- 通过 `os.Exit` 来返回状态

# 获取命令行行参数

## 与其他主要编程语言的差异

- main 函数不支持传入参数

```
func MAIN (ARG []string)
```

- 在程序中直接通过 `os.Args` 获取命令行行参数

# 变量与常量

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

# 编写测试程序

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

# 实现 Fibonacci 数列

1, 1, 2, 3, 5, 8, 13, ....

# 变量赋值

## 与其他主要编程语言的差异

- 赋值可以进行行级自动类型推断
- 在一个赋值语句中可以对多个变量进行同时赋值



# 常量定义

## 与其他主要编程语言的差异

### 快速设置连续值

```
const (  
    MONDAY = IOTA + 1  
    TUESDAY  
    WEDNESDAY  
    THURSDAY  
    FRIDAY  
    SATURDAY  
    SUNDAY  
)
```

```
const (  
    Open = 1 << IOTA  
    Close  
    Pending  
)
```

# 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的差异

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

# 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的差异

1. 不支持指针运算
2. `string` 是值类型，其默认的初始化值为空字符串，而不是 `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 <code>True</code> 否则返回 <code>False</code> 。	<code>(A == B)</code> 为 <code>False</code>
!=	检查两个值是否不不相等，如果不不相等返回 <code>True</code> 否则返回 <code>False</code> 。	<code>(A != B)</code> 为 <code>True</code>
>	检查左边值是否大大于右边值，如果是返回 <code>True</code> 否则返回 <code>False</code> 。	<code>(A &gt; B)</code> 为 <code>False</code>
<	检查左边值是否小小于右边值，如果是返回 <code>True</code> 否则返回 <code>False</code> 。	<code>(A &lt; B)</code> 为 <code>True</code>
>=	检查左边值是否大大于等于右边值，如果是返回 <code>True</code> 否则返回 <code>False</code> 。	<code>(A &gt;= B)</code> 为 <code>False</code>
<=	检查左边值是否小小于等于右边值，如果是返回 <code>True</code> 否则返回 <code>False</code> 。	<code>(A &lt;= B)</code> 为 <code>True</code>

# 用 == 比较数组

- 相同维数且含有相同个数元素的数组才可以比较
- 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

# 位运算符

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

# 位运算符

## 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 条件

```
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  
}
```

# if 条件

## 与其他主要编程语言的差异

1. `condition` 表达式结果必须为布尔值
2. 支持变量赋值:

```
if  VAR DECLARATION;  condition {  
    // code to be executed if condition is true  
}
```

# switch 条件

```
switch os := runtime.GOOS; os {  
CASE "DARWIN":  
    fmt.Println("OS X.")  
    //BREAK  
CASE "linux":  
    fmt.Println("Linux.")  
DEFAULT:  
    // freebsd, openbsd,  
    // PLAN9, windows...  
    fmt.Printf("%s.", os)  
}
```

```
switch {  
    CASE 0 <= Num && Num <= 3:  
        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的差异

1. 条件表达式不限制为常量或者整数；
2. 单个 **case** 中，可以出现多个结果选项，使用逗号分隔；
3. 与 C 语言等规则相反，Go 语言不需要用 **break** 来明确退出一个 **case**；
4. 可以不设定 **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...** 的逻辑作用等同

# 数组和切片

# 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
```

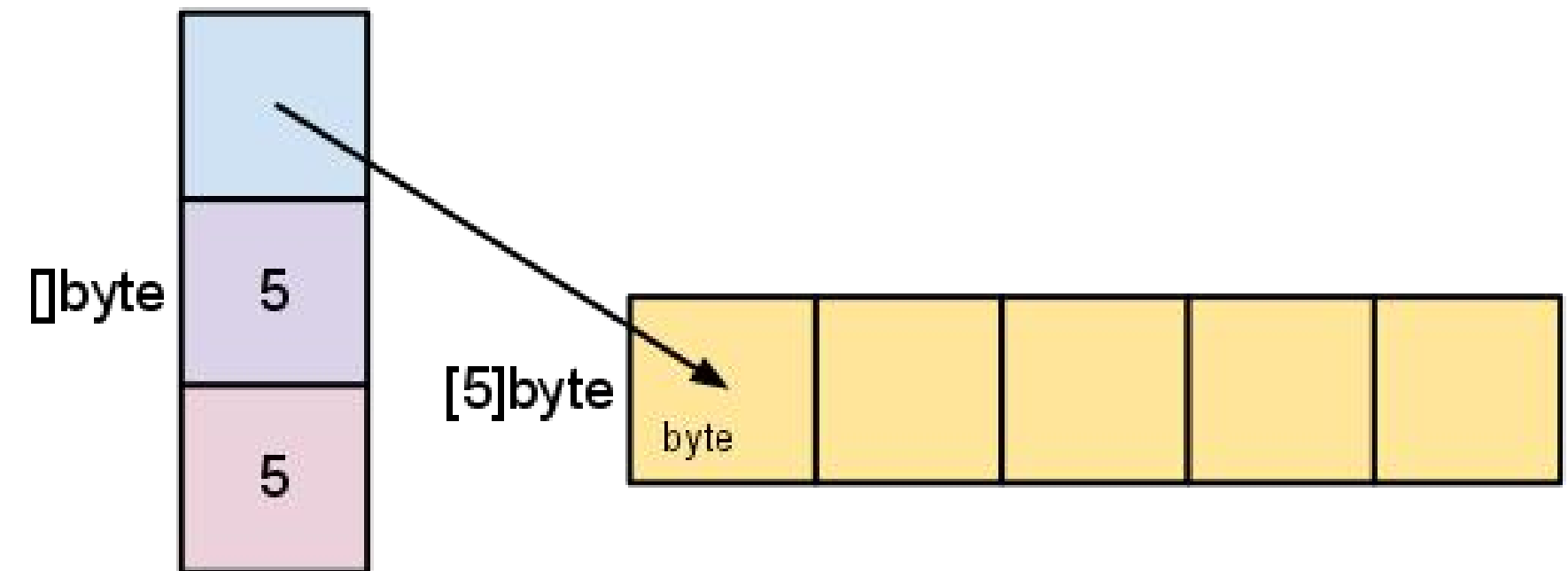
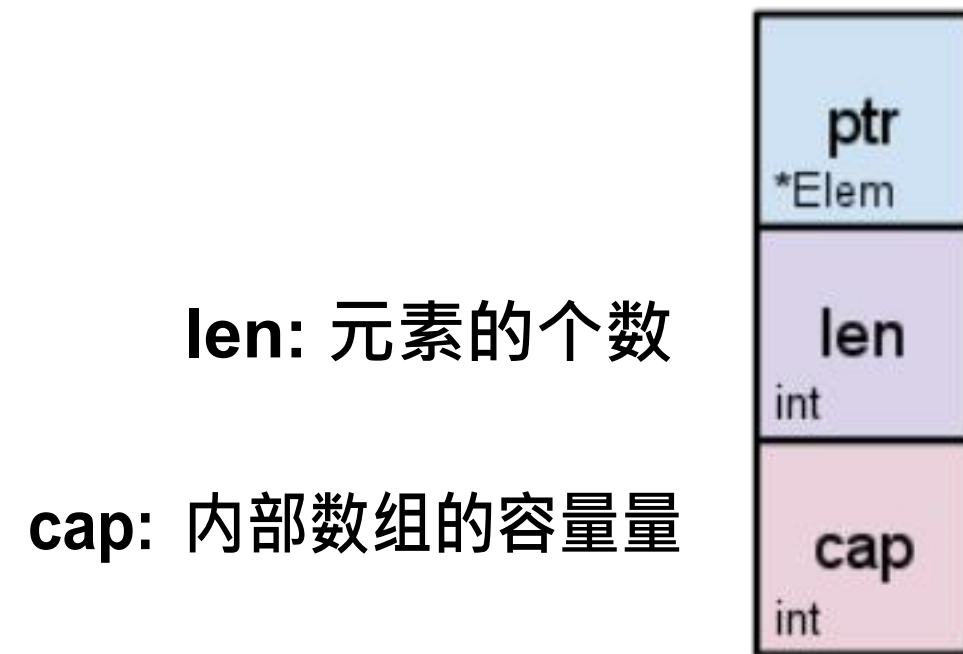
```
A[1:LEN(A)] //2,3,4,5
```

```
A[1:] //2,3,4,5
```

```
A[:3] //1,2,3
```



# 切片内部结构



# 切片声明

```
VAR s0 []int  
s0 = APPEND(s0, 1)
```

```
s := []int{}
```

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

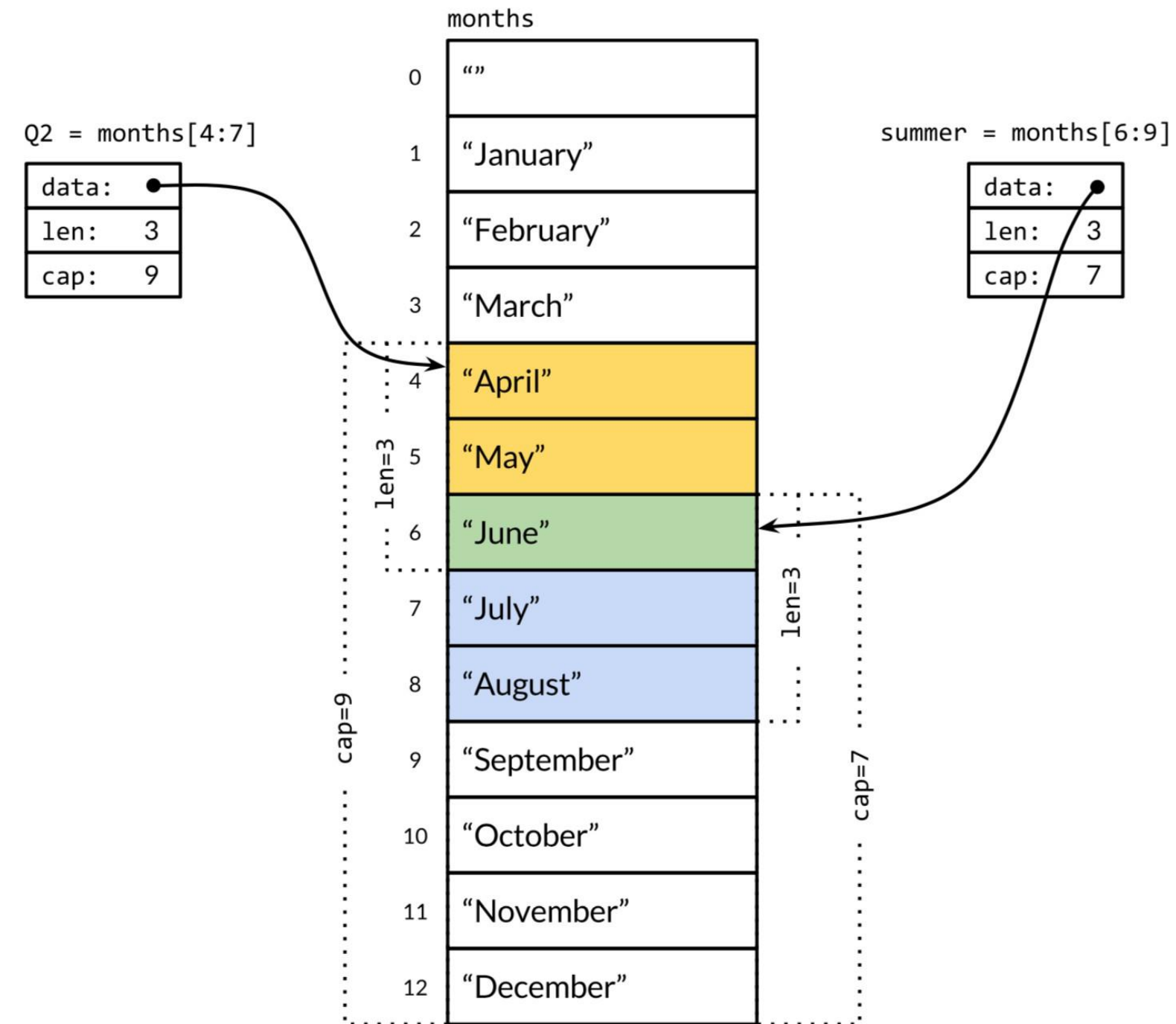
```
s2 := MAKE([]INT, 2, 4)
```

```
/* []type, len, CAP
```

其中len个元素会被初始化为默认零值，未初始化元素不可以访问

```
*/
```

# 切片共享存储结构



# 数组 vs. 切片

1. 容量是否可伸缩
2. 是否可以进行行比较

# 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 */) 
```

//为什么不初始化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** 不存在时，仍会返回零值，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 与工厂模式

- Map 的 value 可以是一个方法
- 与 Go 的 Dock type 接口方式一起，可以方便地实现单一方法对象的工厂模式

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

# 编码与存储

字符

“中”

Unicode

0x4E2D

UTF-8

0xE4B8AD

string/[]byte

[0xE4, 0xB8, 0xAD]

# 常用字符串函数

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



函数：一一等公民

# 函数是一等公民

## 与其他主要编程语言的差异

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

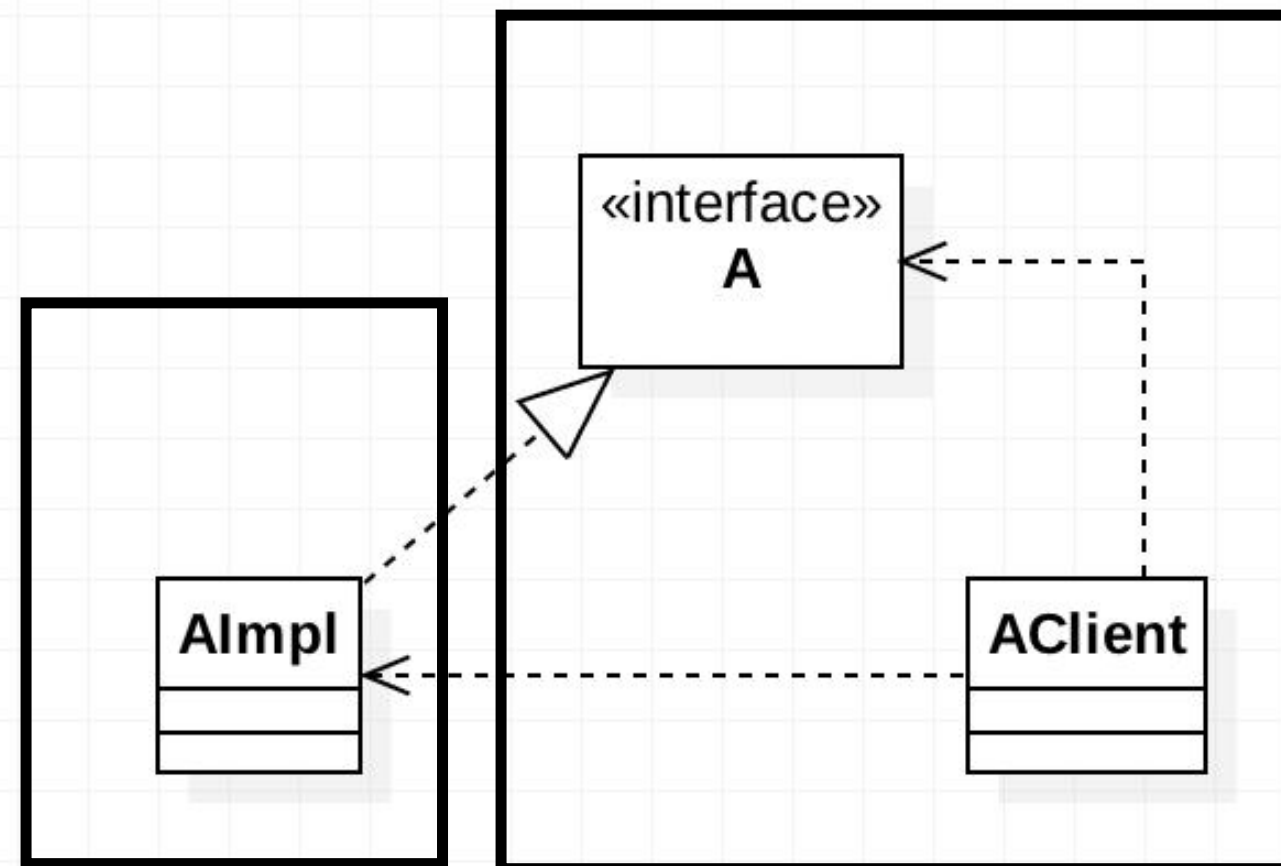
```
func
(e Employee) String() string {
    return FMT.Sprintf("ID:%S-NAME:%S-AGE:%D", e.Id, e.NAME, e.Age)
}
```

//通常情况下为了了避免内存拷贝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 式接口实现

## 接口定义

```
type PROGRAMMER INTERFACE
    { WriteHelloWorld() Code
    }
```

## 接口实现

```
type GoPROGRAMMER struct {
}

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

# Go 接口

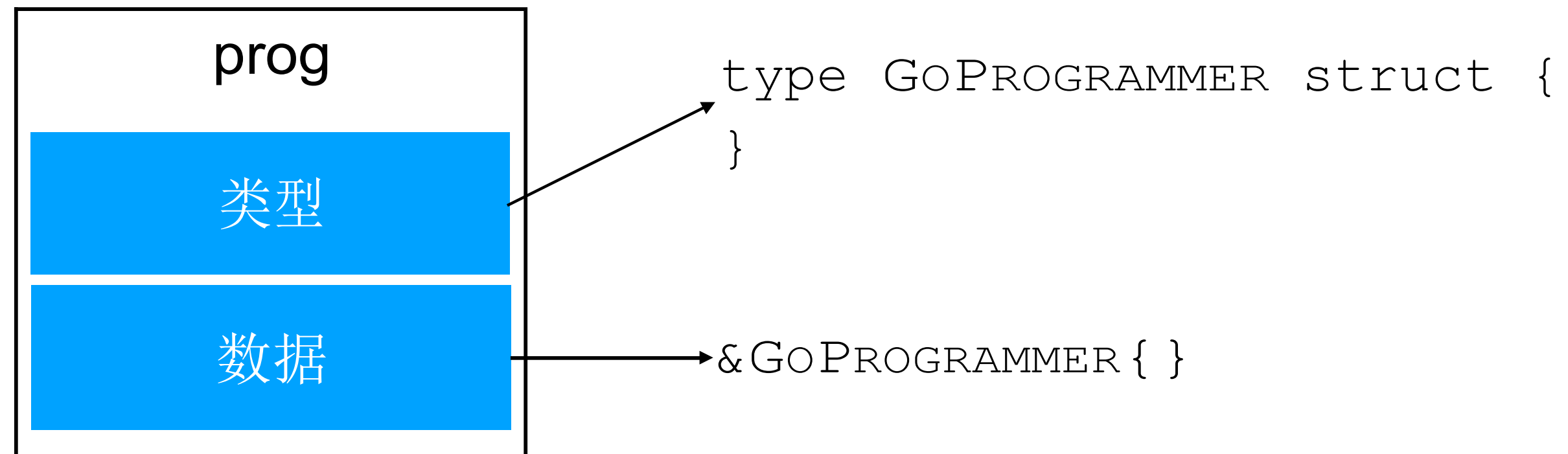
## 与其他主要编程语言的差异

1. 接口为非侵入性，实现不依赖于接口定义
2. 所以接口的定义可以包含在接口使用者包内



# 接口变量

```
var prog Coder = &GoProgrammer{ }
```



# 自定义类型

```
1. type IntConversionFn func(n int) int
```

```
2. type MyPoint int
```

# 扩展与复用

# 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式来复用用

# 匿名类型嵌入

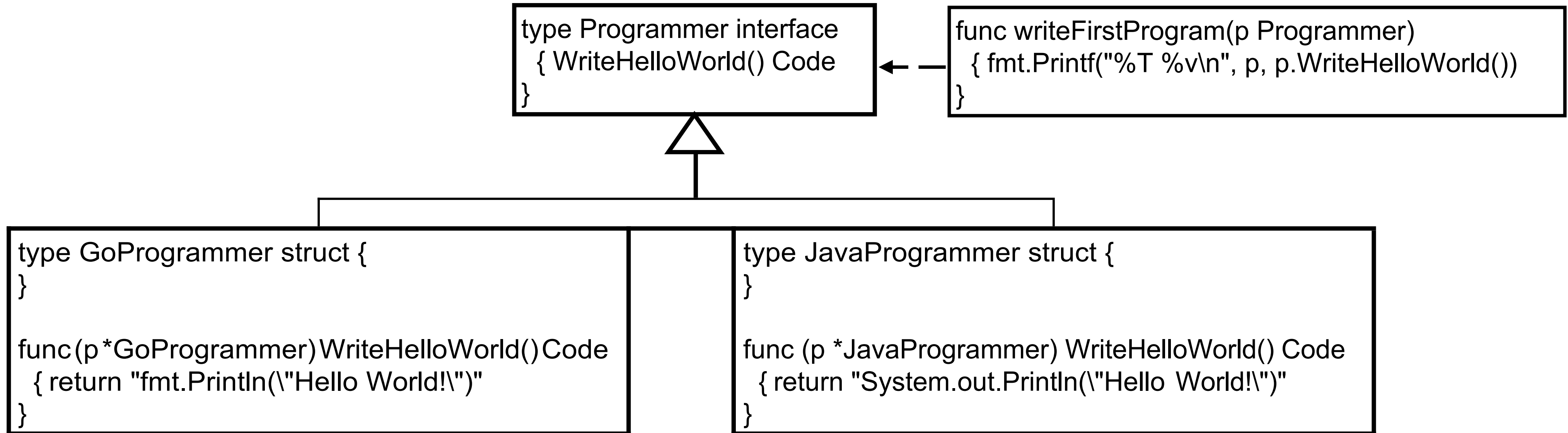
## 与其他主要编程语言的差异

它**不是继承**，如果我们把“内部 struct ”看作父类，把“外部 struct” 看作子类，会发现如下问题：

1. 不**支持**子类替换
2. 子类并不是真正继承了父类的方法
  - 父类的定义的方法无法访问子类的数据和方法

# 多态与空接口

# 多态



# 空接口与断言

1. 空接口可以表示任何类型
2. 通过断言来将空接口转换为制定类型

```
v, ok := p.(int) //ok=true 时为转换成功
```



# Go 接口最佳实践

倾向于使用用小小的接口定义，很多接口只包含一个方法

```
type Reader INTERFACE {  
    Read(p []byte) (n int, err error)  
}
```

```
type Writer INTERFACE {  
    Write(p []byte) (n int, err error)  
}
```

---

较大的接口定义，可以由多个小小接口定义组合而成

```
type ReaderWriter INTERFACE  
{ Reader  
  Writer  
}
```

---

只依赖于必要功能的最小接口

```
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的差异

1. 没有异常机制

2. `error` 类型实现了 `error` 接口

```
type error INTERFACE
    { Error() string
    }
```

3. 可以通过 `errors.New` 来快速创建错误实例

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

# 最佳实践

定义不同的错误变量量, 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")
    }
    ...
}
```

# 最佳实践

及早失败，避免嵌套！

panic 和 recover

# panic

- panic 用于不可恢复的错误
- panic 退出前会执行行 `defer` 指定的内容

# panic vs. os.Exit

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



# recover

## Java

```
try{  
    ...  
} CATCH (THROWABLE t) {  
  
}
```

## C++

```
try{  
    ...  
} CATCH (...) {  
  
}
```

# recover

```
defer func() {  
    if err := recover(); err != nil {  
        //恢复错误  
    }  
}()
```

# 最常见见的“错误恢复”

```
defer func() {  
    if err := recover(); err != nil  
    { log.Error("recovered PANIC", ERR)  
    }  
}()
```

# 当心心！ recover 成为恶魔

- 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法。



# 构建可复用模块：包

# package

## 1. 基本复用模块单元

以首字母大写来表明可被包外代码访问

## 2. 代码的 `package` 可以和所在的目录不一致

## 3. 同一目录里的 Go 代码的 `package` 要保持一致

# package

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](https://github.com/easierway/concurrent_map)

# init 方法

- 在 `main` 被执行行行前，所有依赖的 `package` 的 `init` 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

# 常用的依赖管理工具

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

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

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

# 协程机制

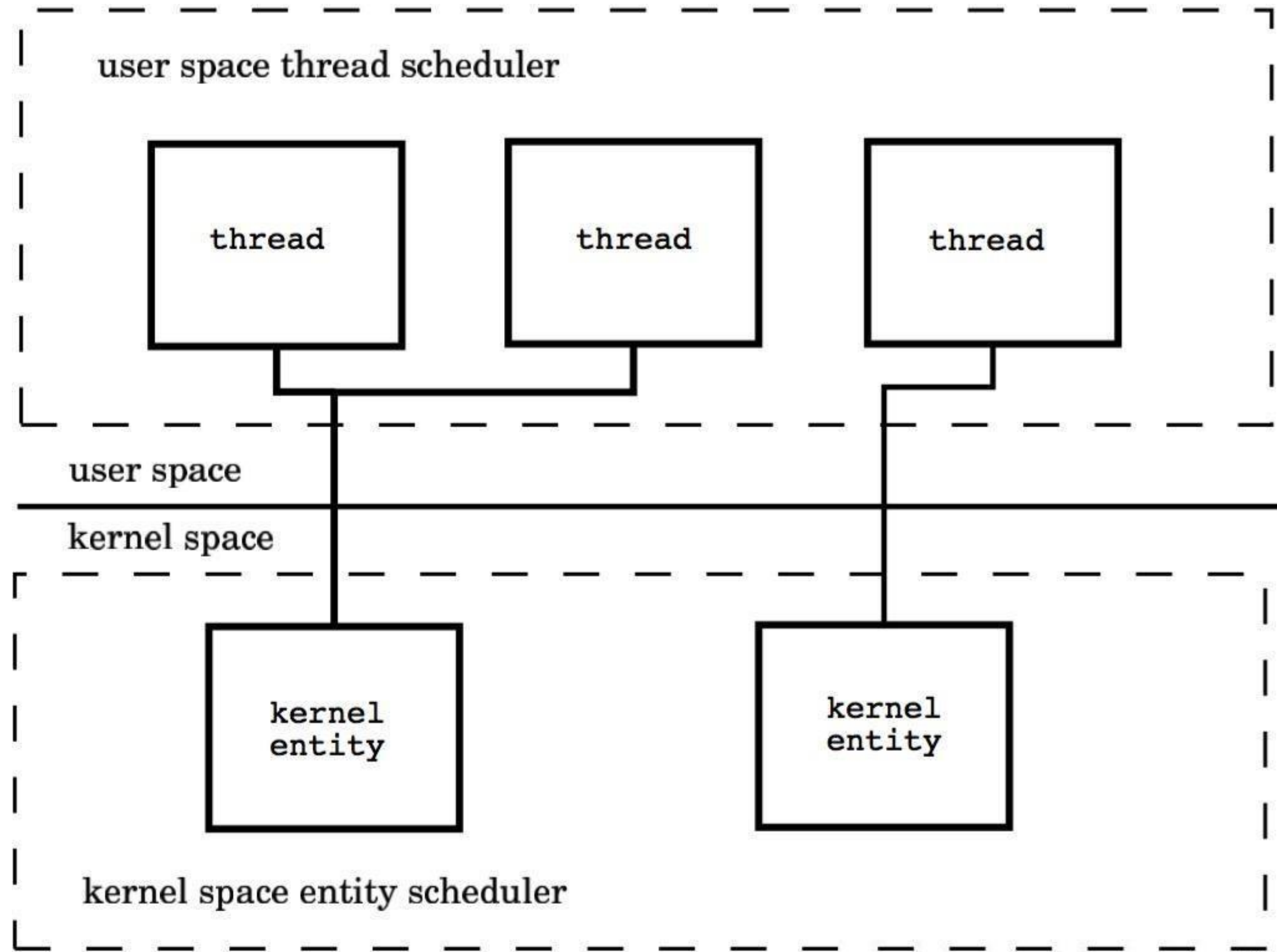
# Thead vs. Groutine

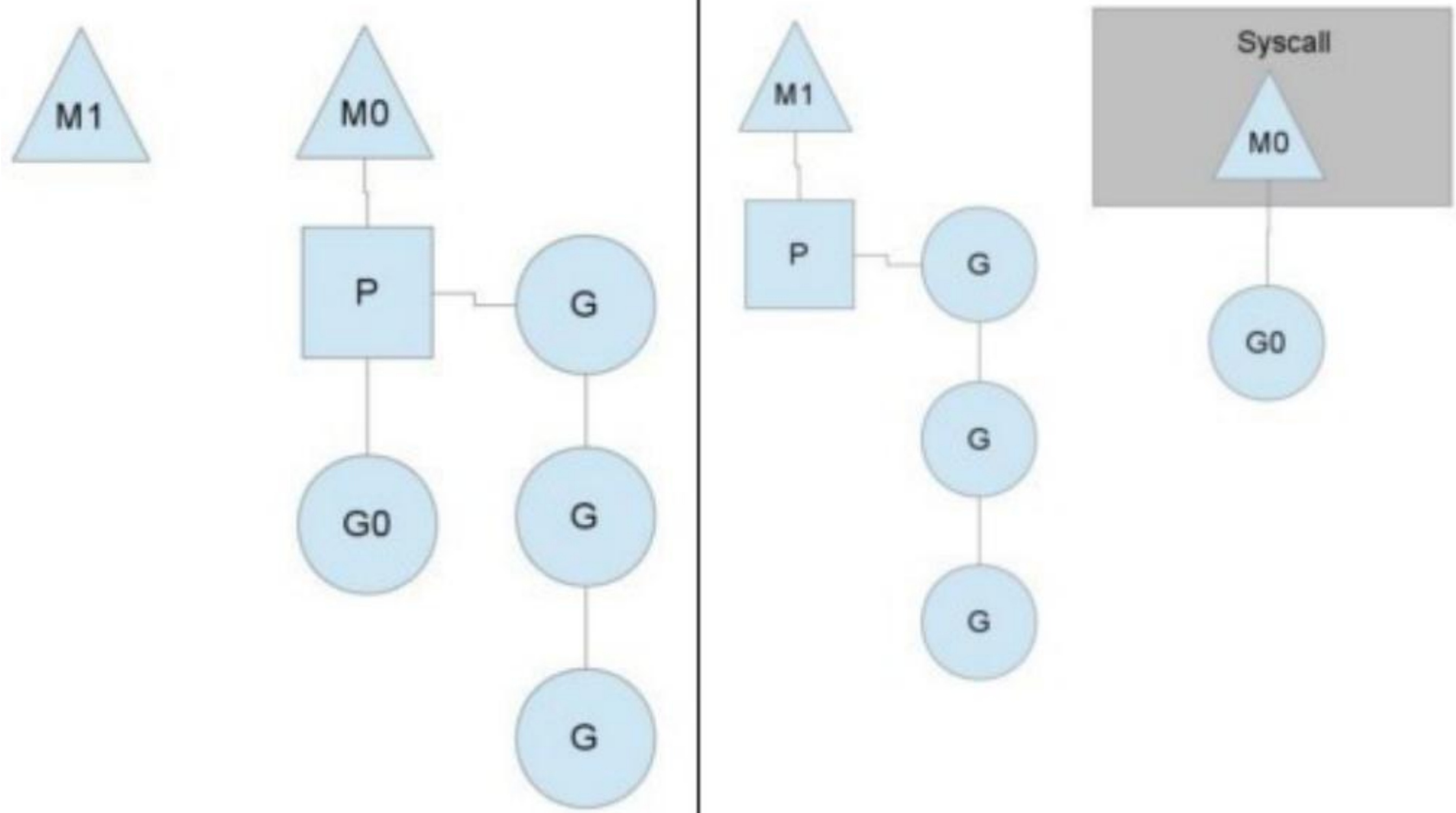
## 1. 创建时默认的 stack 的大小

- JDK5 以后 Java Thread stack 默认为1M
- Groutine 的 Stack 初始化大小为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{  
    //process (THREAD-SAFE)  
}CATCH (EXCEPTION ex) {  
  
} FINALLY {  
    lock.unlock();  
}
```

**package sync**

Mutex


RWLock

# 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



WIKIPEDIA  
The Free Encyclopedia

[Main page](#)  
[Contents](#)  
[Featured content](#)  
[Current events](#)  
[Random article](#)  
[Donate to Wikipedia](#)  
[Wikipedia store](#)

[Interaction](#)

[Help](#)  
[About Wikipedia](#)  
[Community portal](#)  
[Recent changes](#)  
[Contact page](#)

[Tools](#)

[What links here](#)  
[Related changes](#)

Not logged in [Talk](#) [Contributions](#) [Create account](#) [Log in](#)

Article [Talk](#)

Read [Edit](#) [View history](#)

## Communicating sequential processes

From Wikipedia, the free encyclopedia

In [computer science](#), **communicating sequential processes (CSP)** is a [formal language](#) for describing [patterns](#) of [interaction](#) in [concurrent systems](#).<sup>[1]</sup> It is a member of the family of mathematical theories of concurrency known as process algebras, or [process calculi](#), based on [message passing](#) via [channels](#). CSP was highly influential in the design of the [occam](#) programming language,<sup>[1][2]</sup> and also influenced the design of programming languages such as [Limbo](#),<sup>[3]</sup> [RaftLib](#), [Go](#)<sup>[4]</sup>, [Crystal](#), and [Clojure](#)'s [core.async](#).

CSP was first described in a 1978 paper by [Tony Hoare](#),<sup>[5]</sup> but has since evolved substantially.<sup>[6]</sup> CSP has been practically applied in industry as a tool for [specifying and verifying](#) the concurrent aspects of a variety of different systems, such as the T9000 [Transputer](#),<sup>[7]</sup> as well as a secure ecommerce system.<sup>[8]</sup> The theory of CSP itself is also still the subject of active research, including work to increase its range of practical applicability (e.g., increasing the scale of the systems that can be tractably analyzed).<sup>[9]</sup>

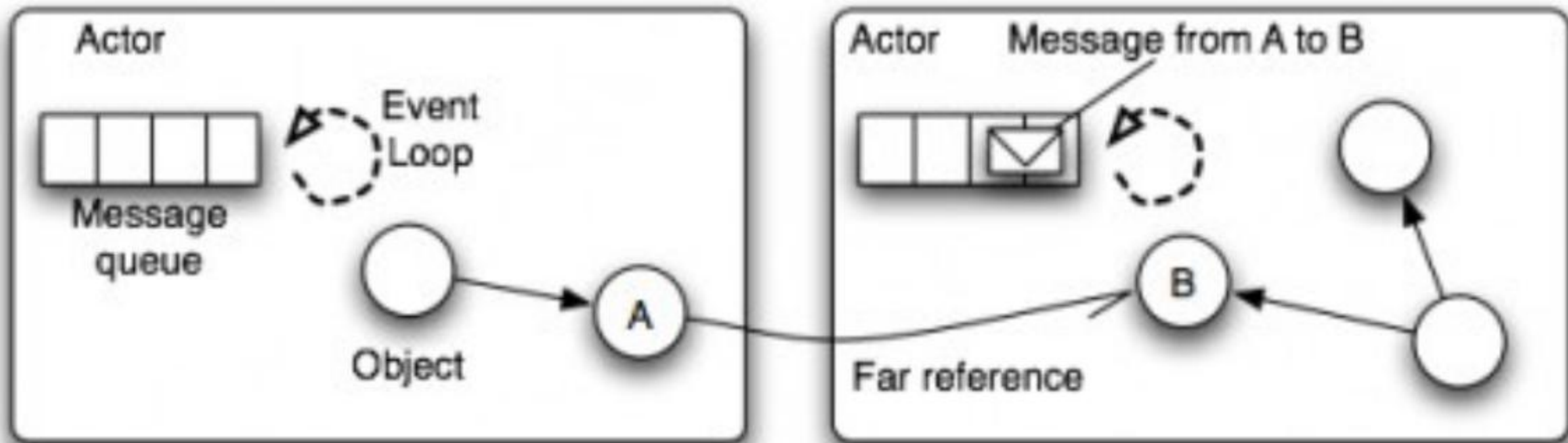
### Contents [\[hide\]](#)

- [History](#)
  - [Applications](#)
- [Informal description](#)
  - [Primitives](#)



Tony Hoare

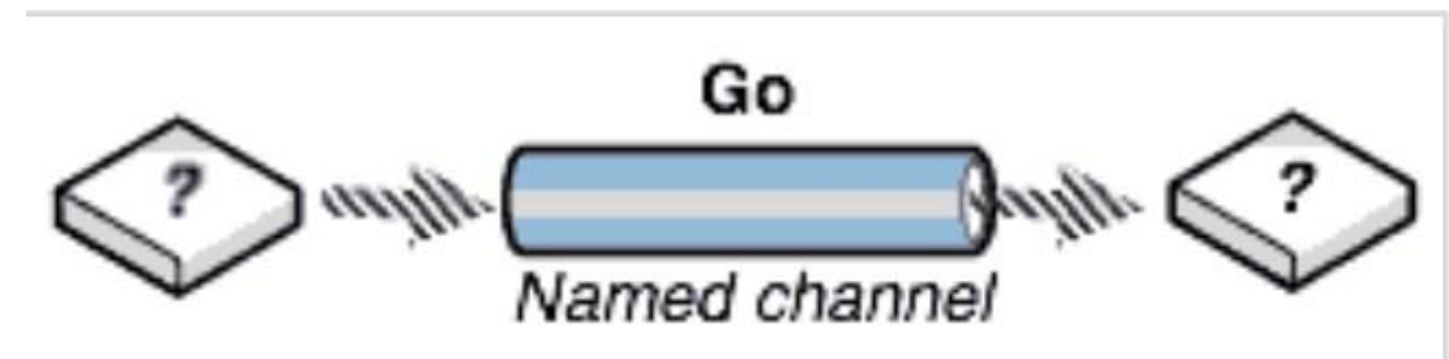
# Actor Model



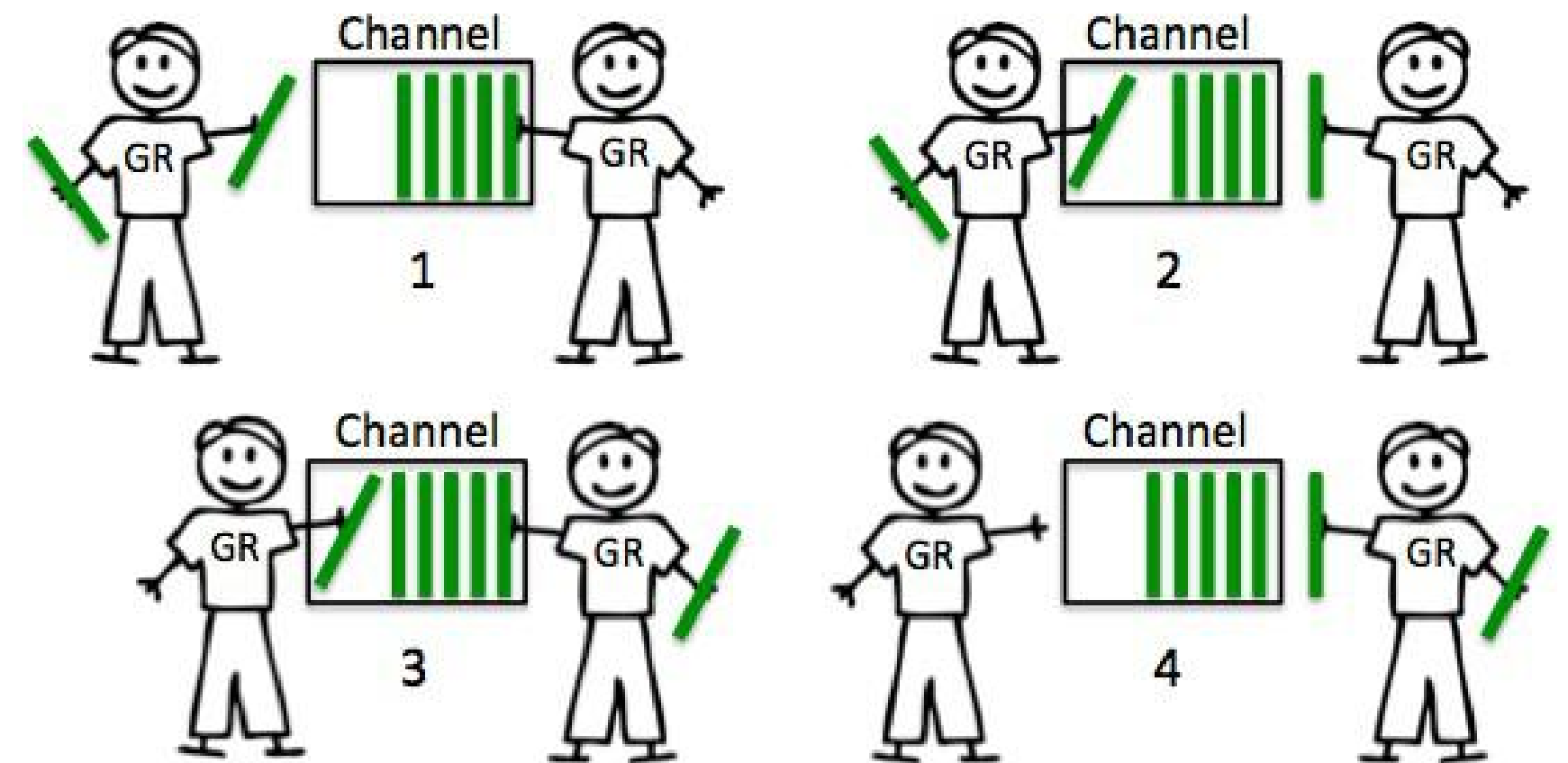
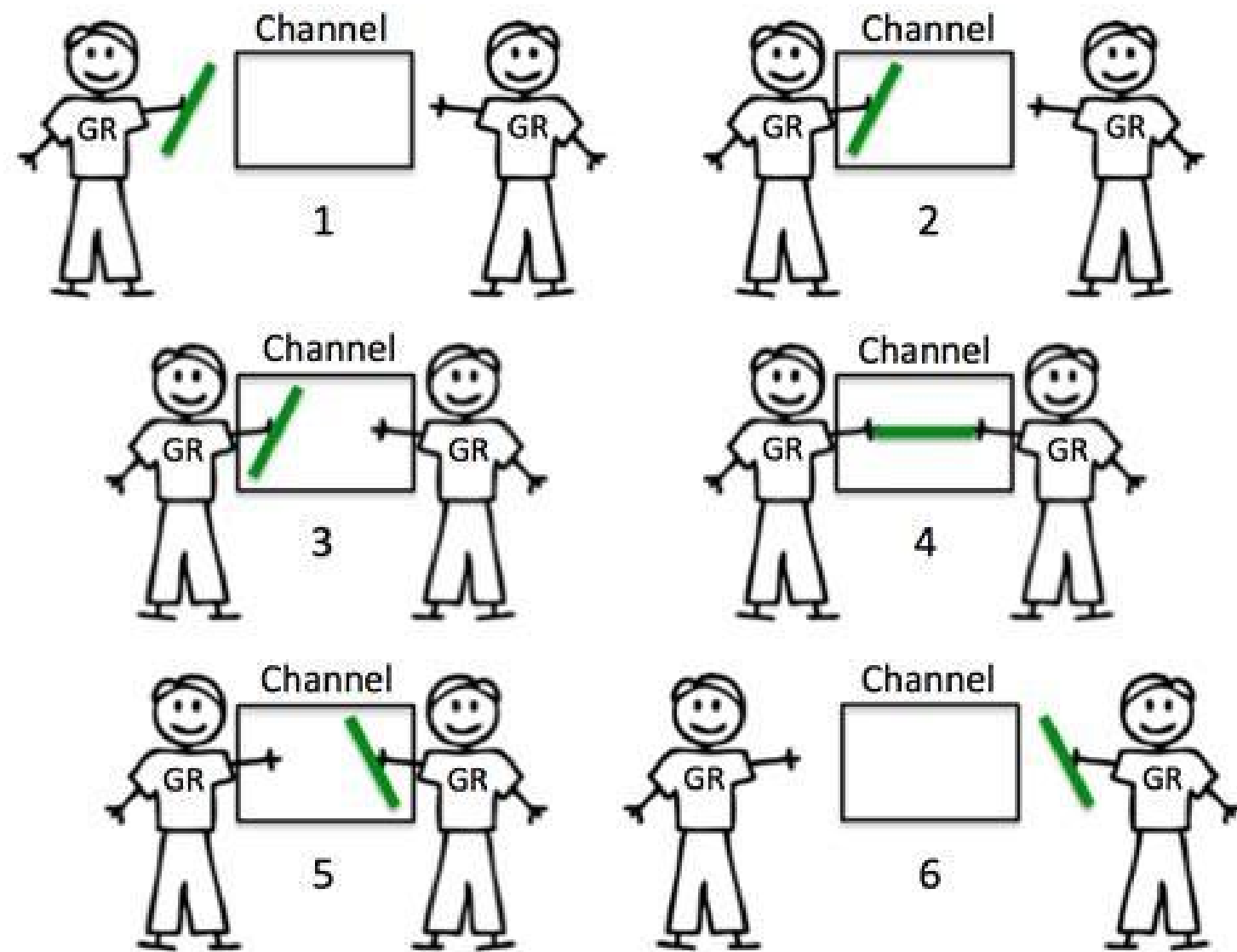


# CSP vs. Actor

- 和Actor的直接通讯不同，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());
```

# 多路选择和超时控制

# select

## 多渠道的选择

```
select {
CASE ret := <-retCh1:
    t.Logf("result %s", ret)
CASE ret :=<-retCh2:
    t.Logf("result %s", ret)
DEFAULT:
    t.Error("No one returned")
}
```

## 超时控制

```
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，立刻从阻塞等待中返回且上述 `ok` 值为 `false`。这个广播机制常被利用，进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{ }{}  
    }
```

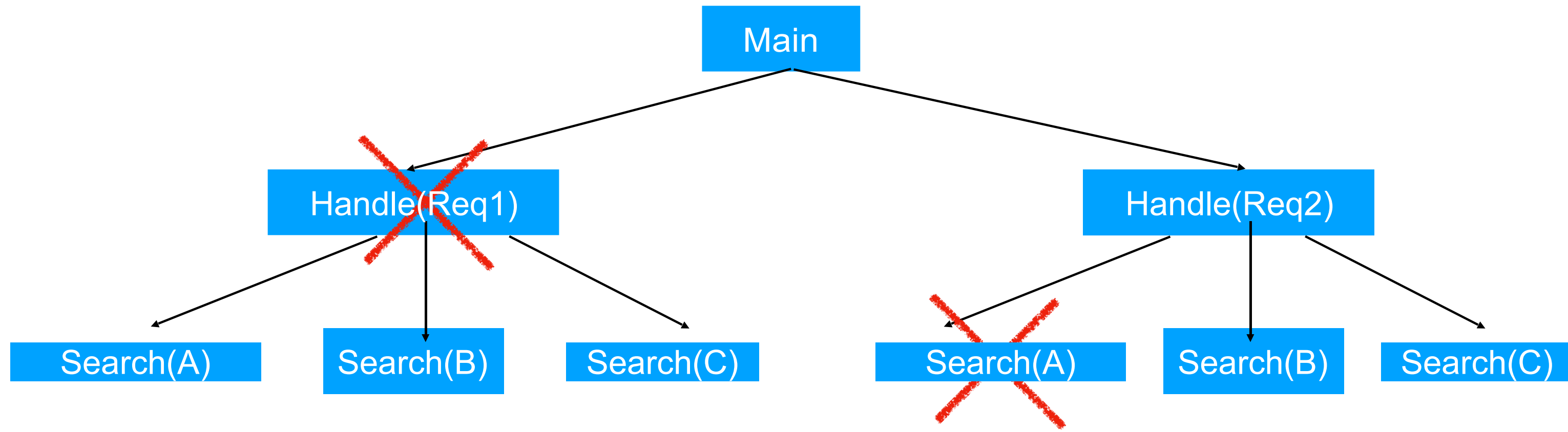


# 通过关闭 Channel 取消

```
func CANCEL_2 (CANCELCHAN chan struct{})  
    { CLOSE (CANCELCHAN)  
    }
```

# Context 与任务取消

# 关联任务的取消



# Context

- 根 Context: 通过 `context.Background ()` 创建
- 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 context 都会被取消
- 接收取消通知 `<-ctx.Done()`

# 常见并发任务

仅执行行行一一次

# 单例例模式 （懒汉式，线程安全）

```
public class Singleton {  
    private static Singleton INSTANCE=null;  
    private Singleton(){}  
    public static Singleton getInstnace(){  
        if(INSTANCE==null){ synchronize  
            d (Singleton.class){  
                if(INSTANCE==null){  
                    INSTANCE = new Singleton();  
                }  
            }  
        }  
        return INSTANCE;  
    }  
}
```

# 单例例模式（懒汉式，线程安全）

```
var once sync.Once
var obj *SingletonObj

func GetSingletonObj() *SingletonObj
{
    once.Do(func() {
        fmt.Println("Create Singleton obj.")
        obj = &SingletonObj{}
    })
    return obj
}
```

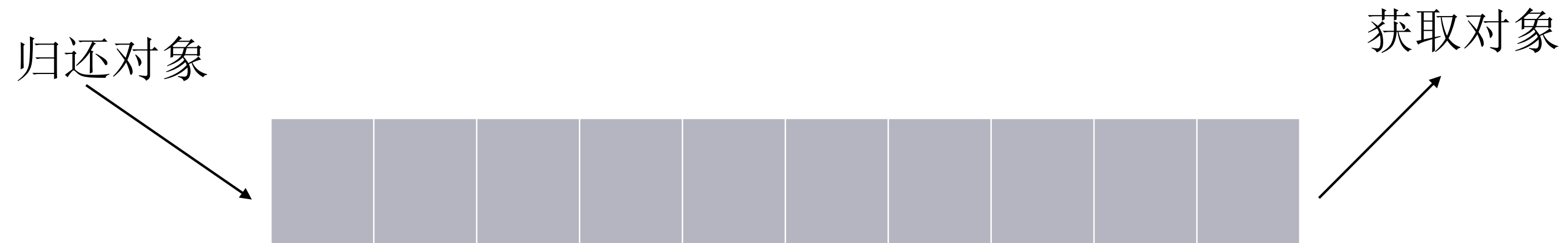


仅需任意任务完成

必需所有任务完成

# 对象池

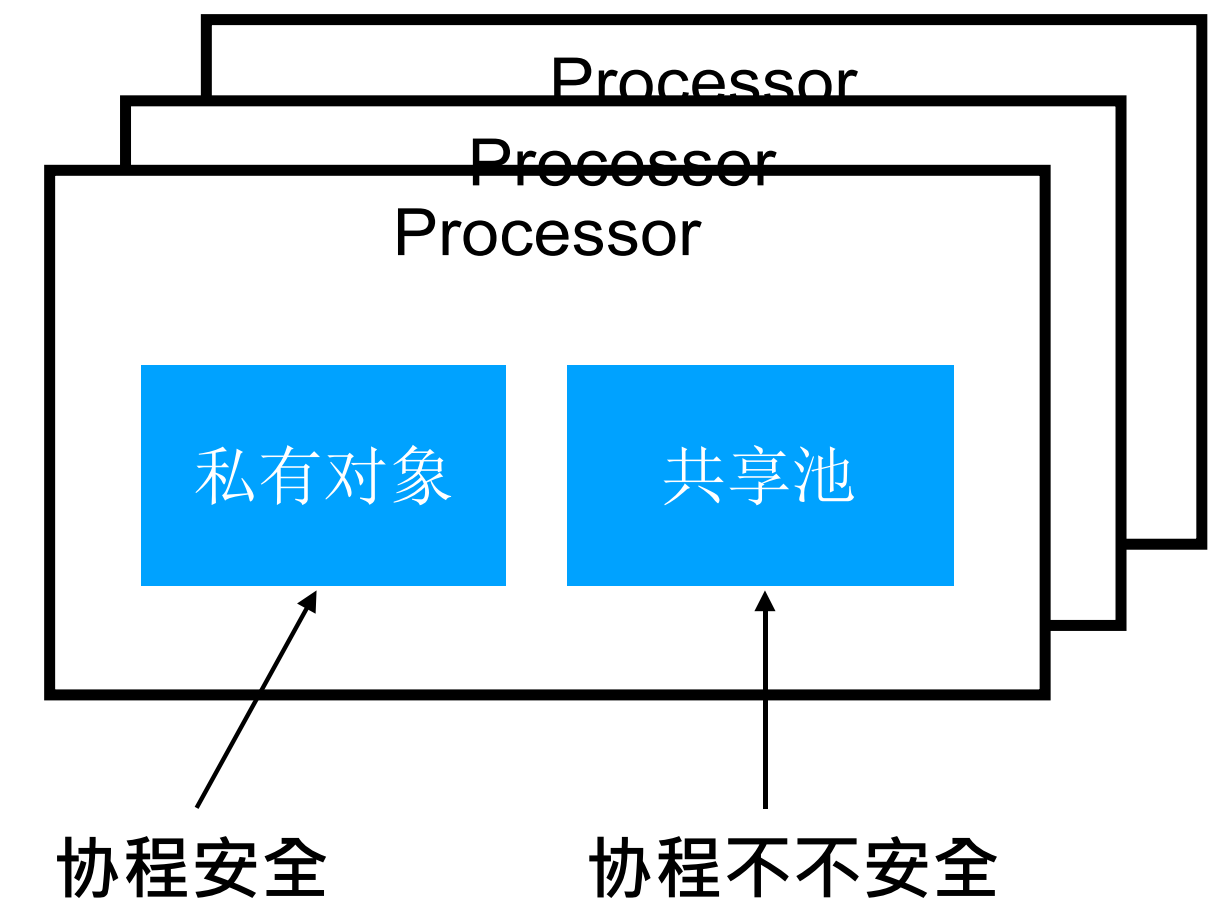
# 使用用buffered channel实现对象池



# sync.Pool 对象缓存

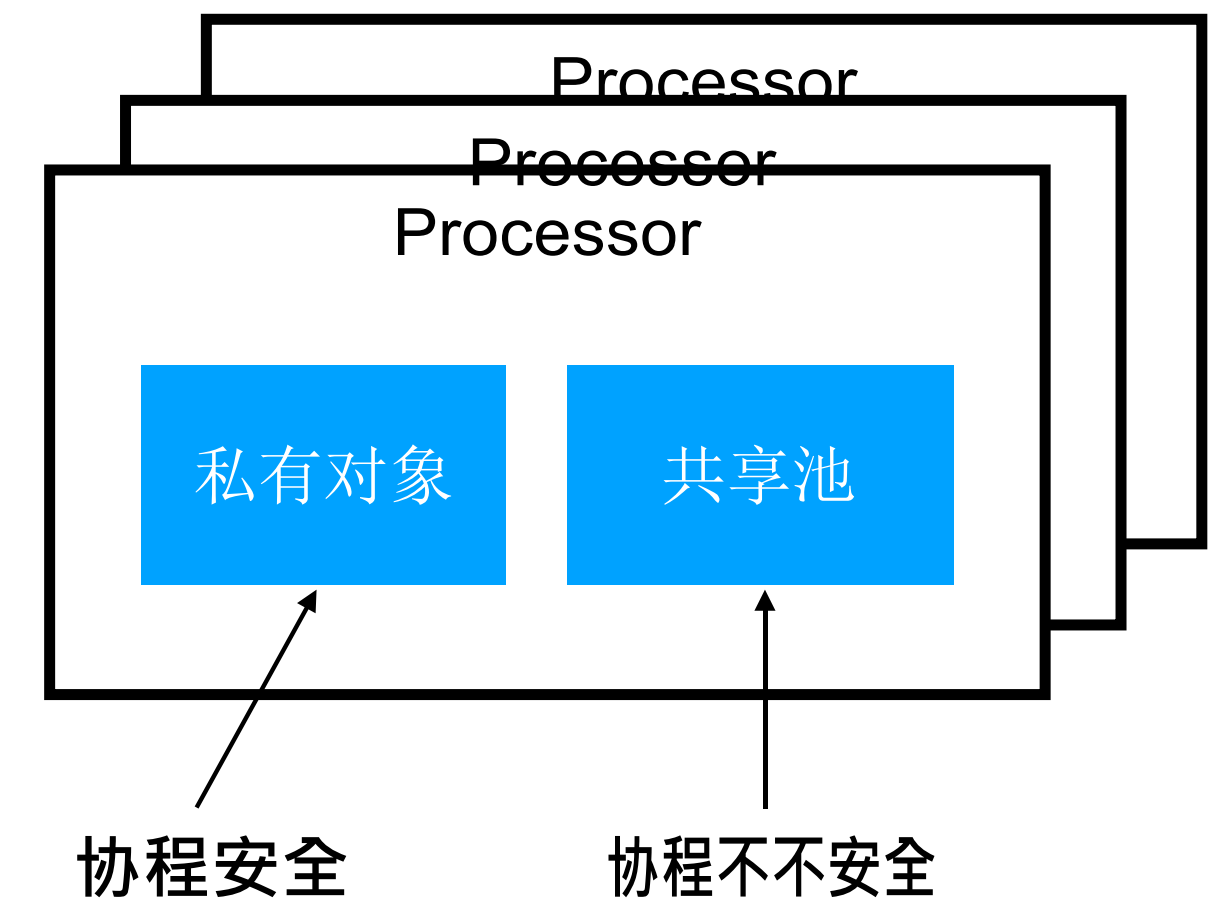
# sync.Pool 对象获取

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



# sync.Pool 对象的放回

- 如果私有对象不存在则保存为私有对象
- 如果私有对象存在，放入入当前 Processor 子子池的共享池中



# 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次GC 之前

# sync.Pool 总结

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

# 测试

# 单元测试

# 内置单元测试框架

- **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

# 内置单元测试框架

- 代码覆盖率

`go test -v -cover`

- 断言言

<https://github.com/stretchr/testify>

# Benchmark

# 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  
}
```



# Benchmark

```
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

*This is not an Argument about a Bug*

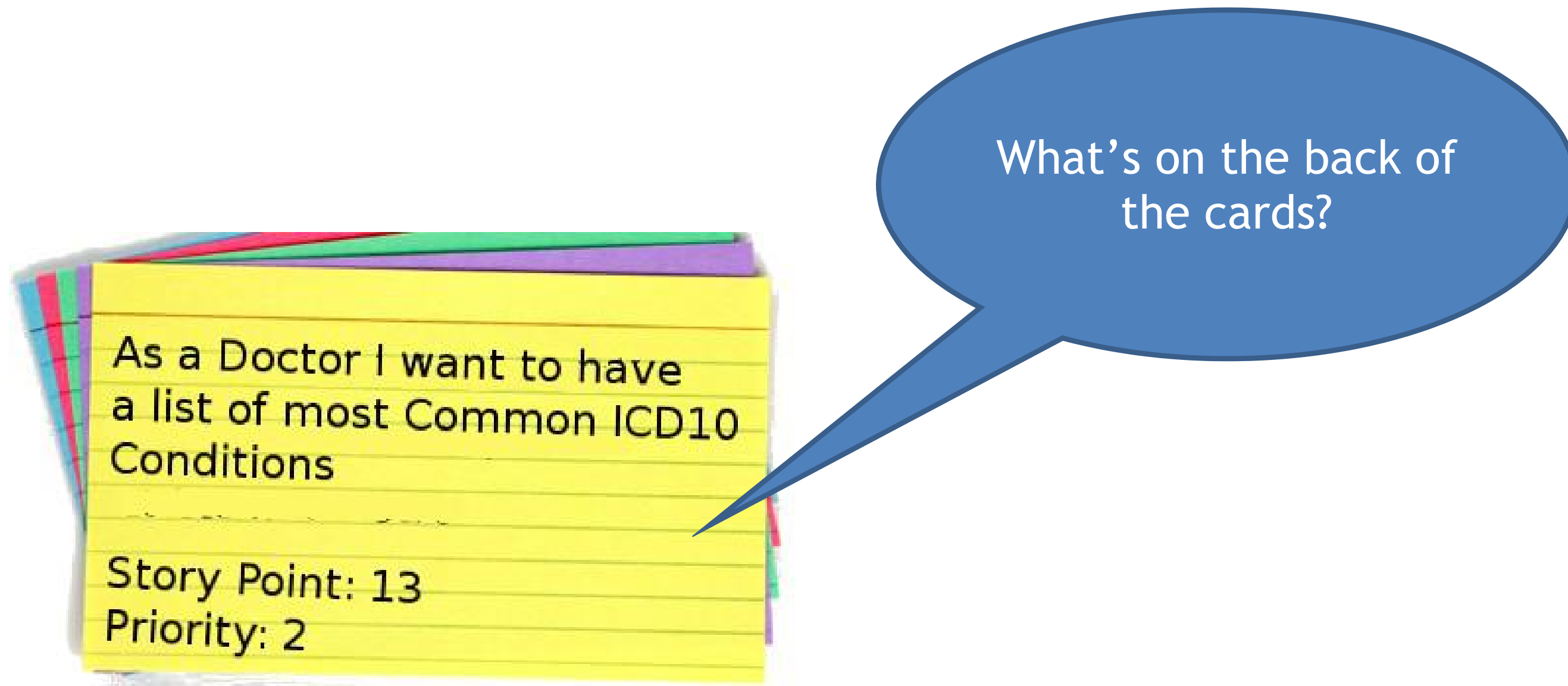
"Bug Triage Meeting"  
The Tuesday before release.



*“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* 背面应该写什么吗？



# 用业务领域的语言来描述

## Capture Concrete Expectations and Examples

Password	Valid?	Can be expressed as "Given - When - Then"
"p@ssw0rd"	Yes	
"p@s5"	No	
"passw0rd"	No	<p>↓</p> <p><b>Given</b> a user is creating an account</p> <p><b>When</b> they specify an insecure password</p> <p><b>Then</b> they see a message, "Passwords must be at least 6 characters long with at least one letter, one number, and one symbol."</p>
"p@ssword"	No	
"@#s%1234"	No	

Or can be ↑  
expressed in tables

Or in other formats  
depending on the  
Framework

*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")
```

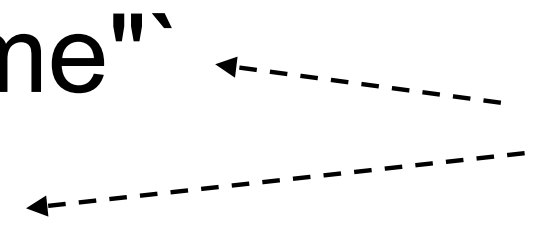
## 按名字访问结构的方法

```
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 方法，注意他们的区别

# “万能”程序

# 关于“反射”你应该知道的

- 提高了程序的灵活性
- 降低了程序的可读性
- 降低了程序的性能

# “不不安全”编程



# “不安全”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)

Take the risk  
or lose the  
chance.

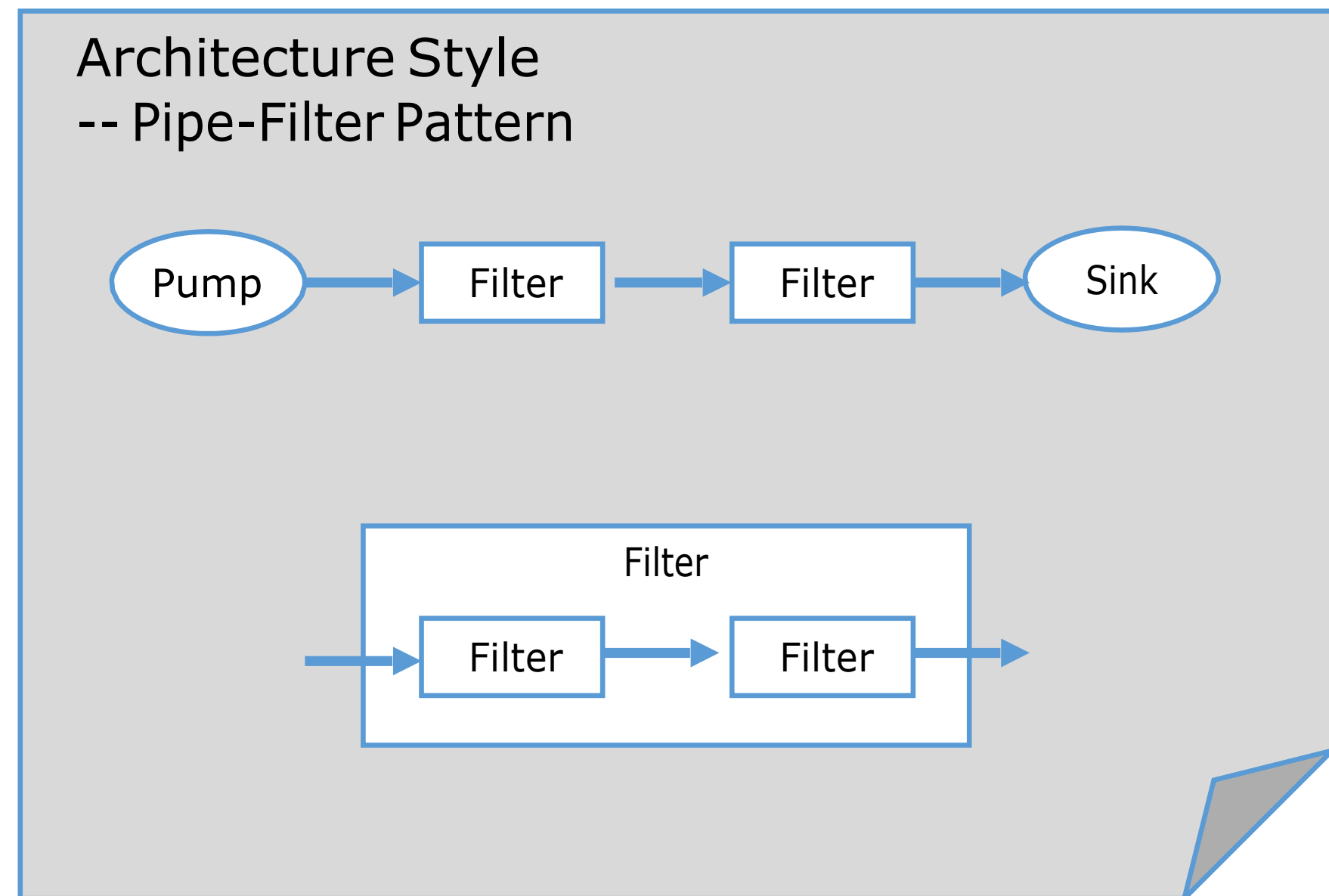
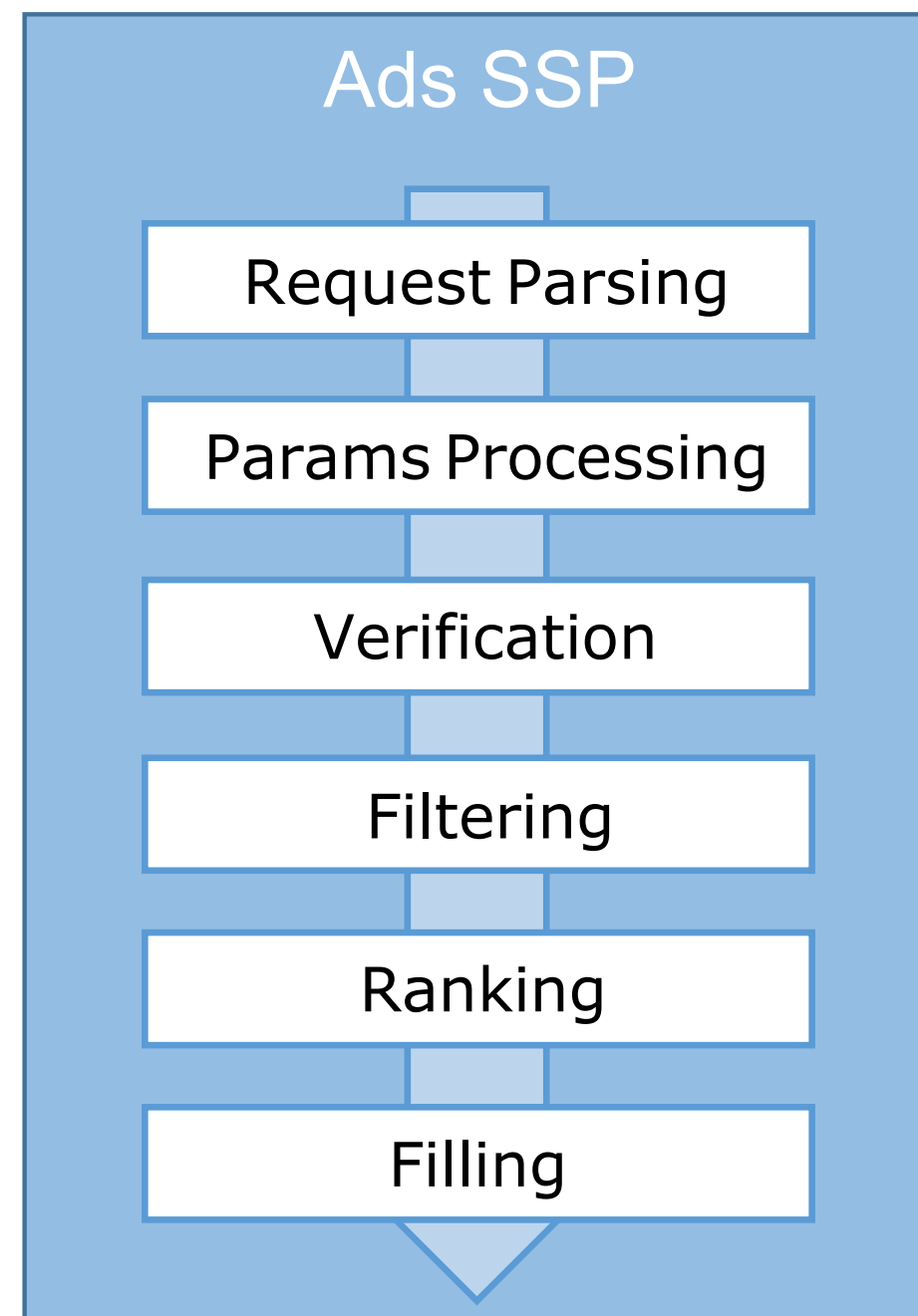
# 构建高可扩展性的软件架构

# 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

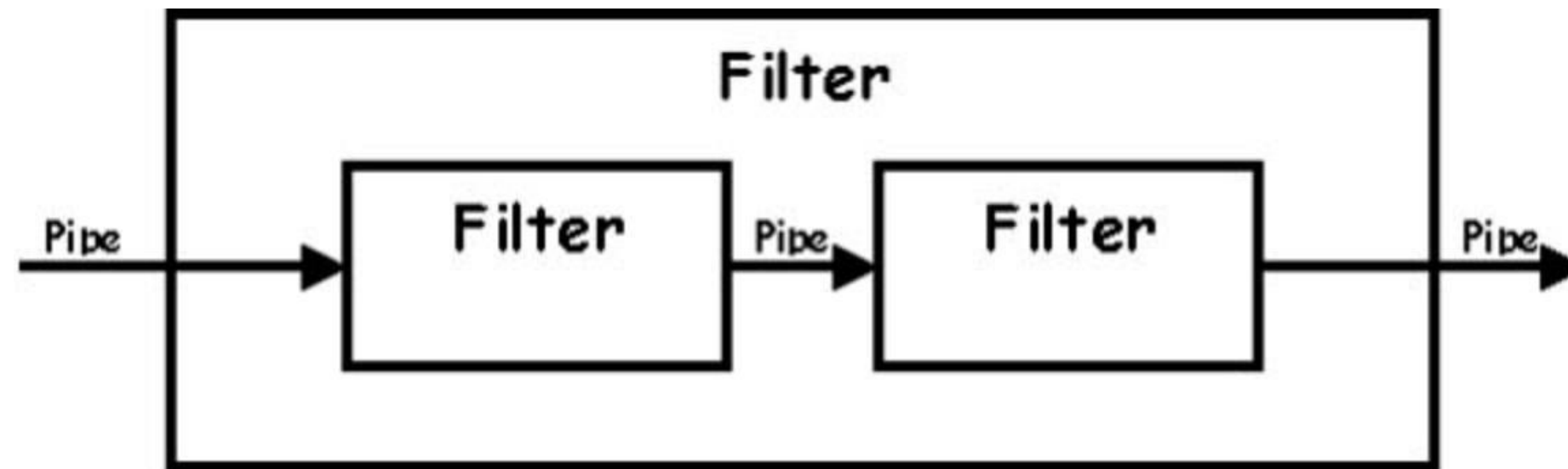
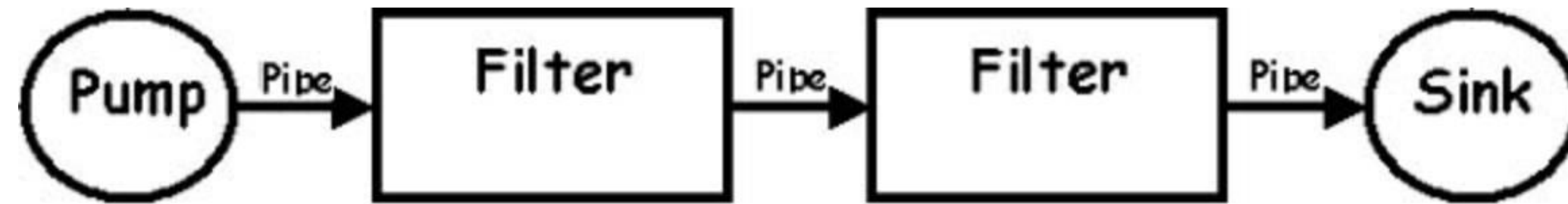
# Pipe-Filter 架构



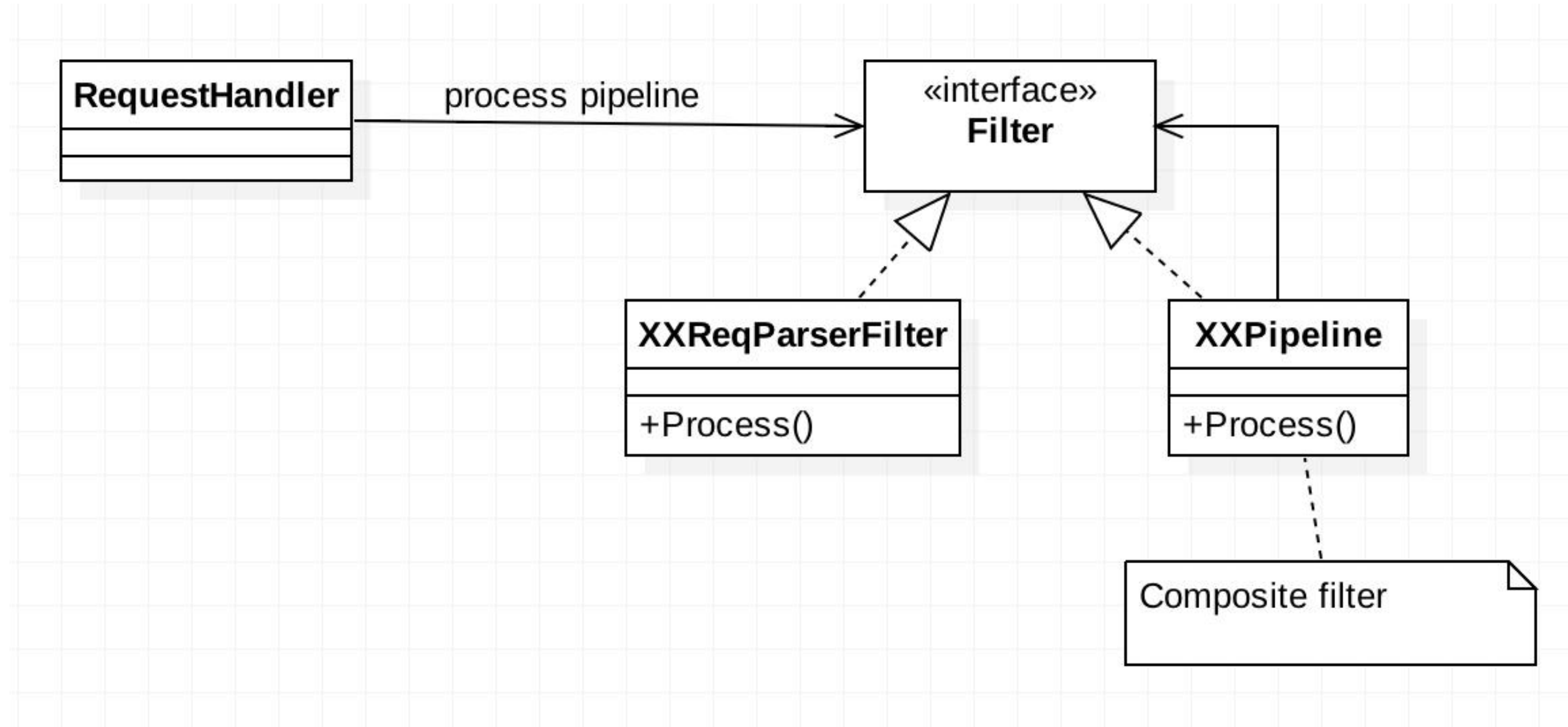
# Pipe-Filter 模式

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

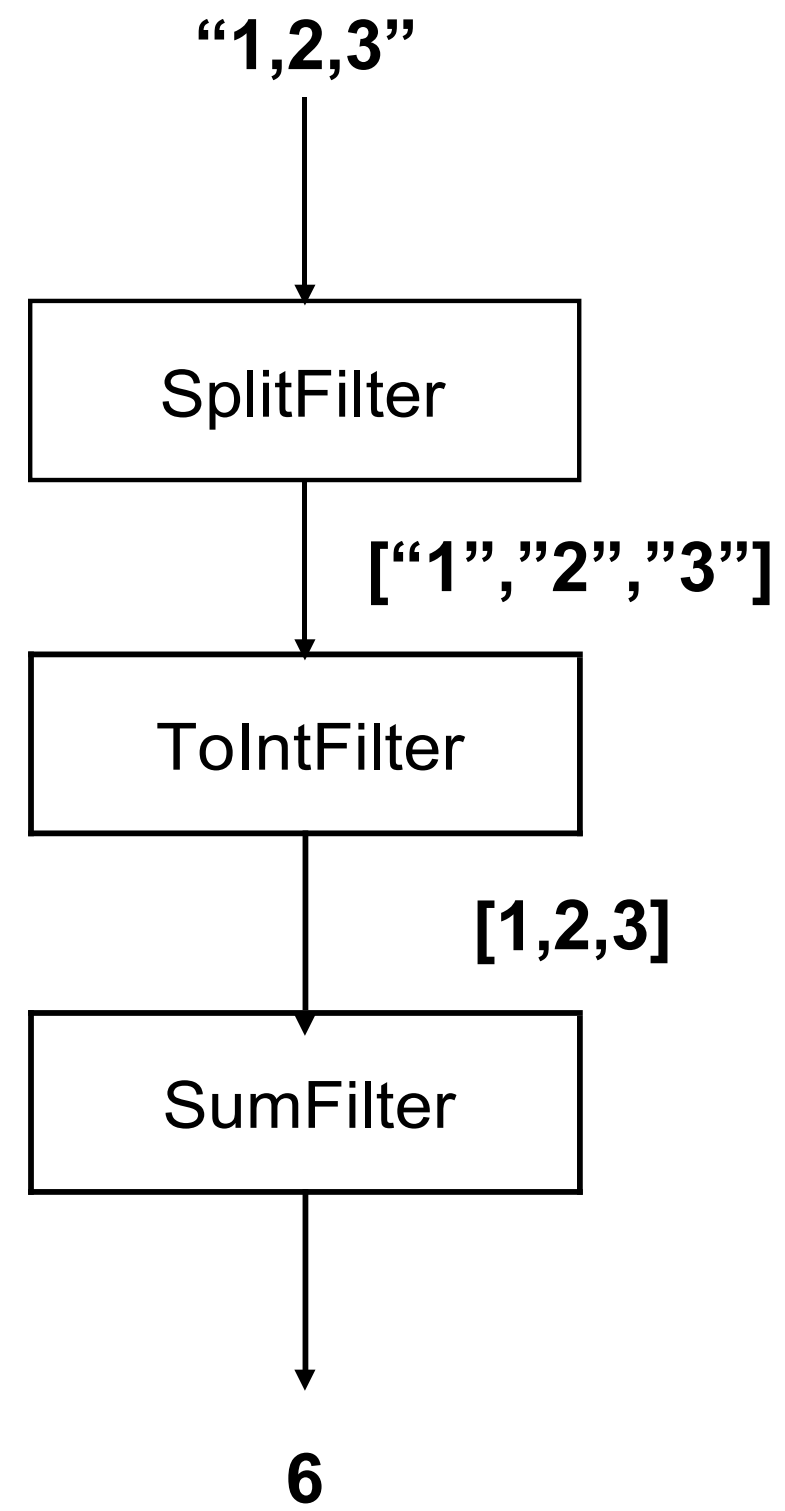
# Filter和组合模式







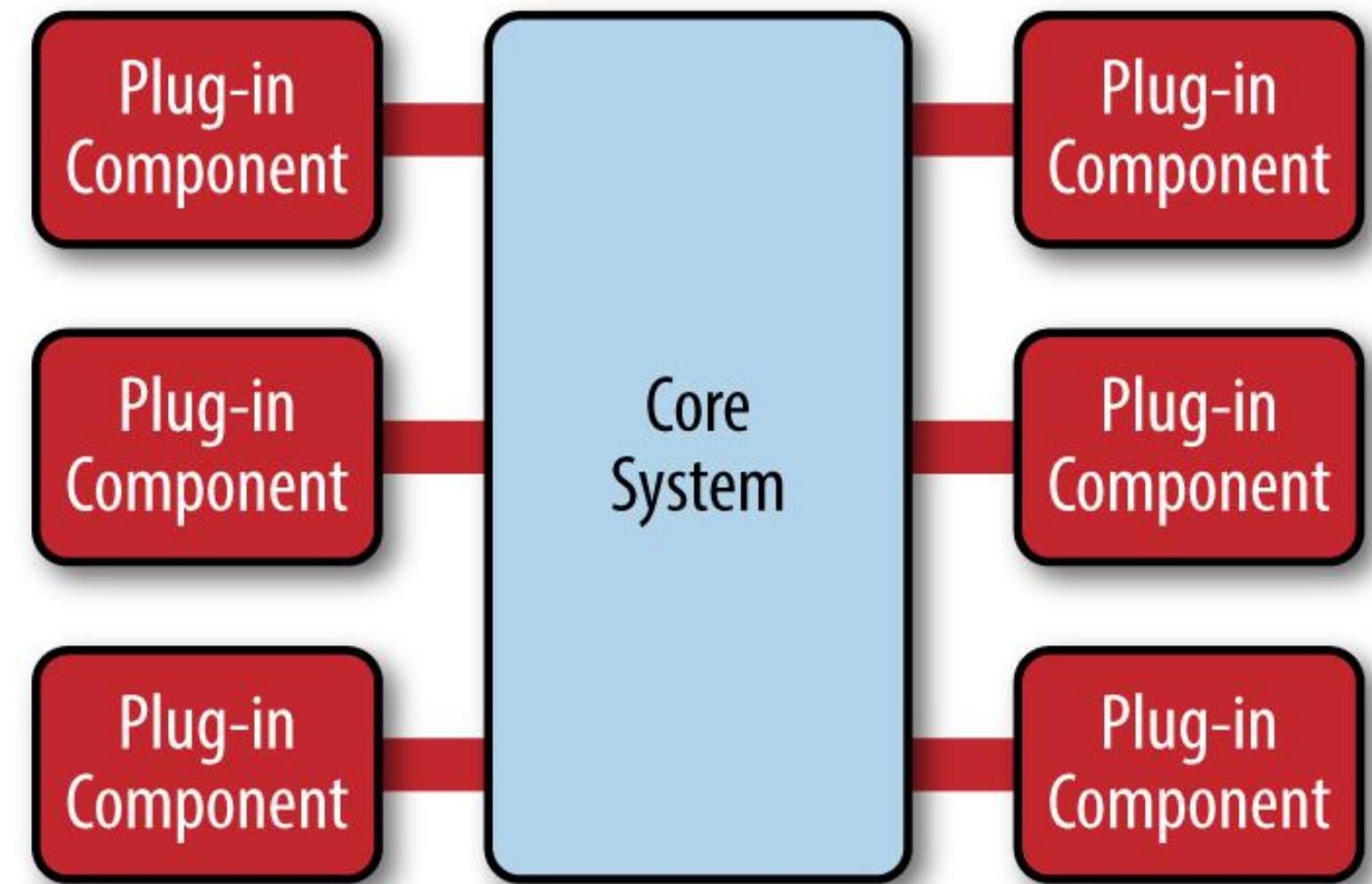
# 示例例



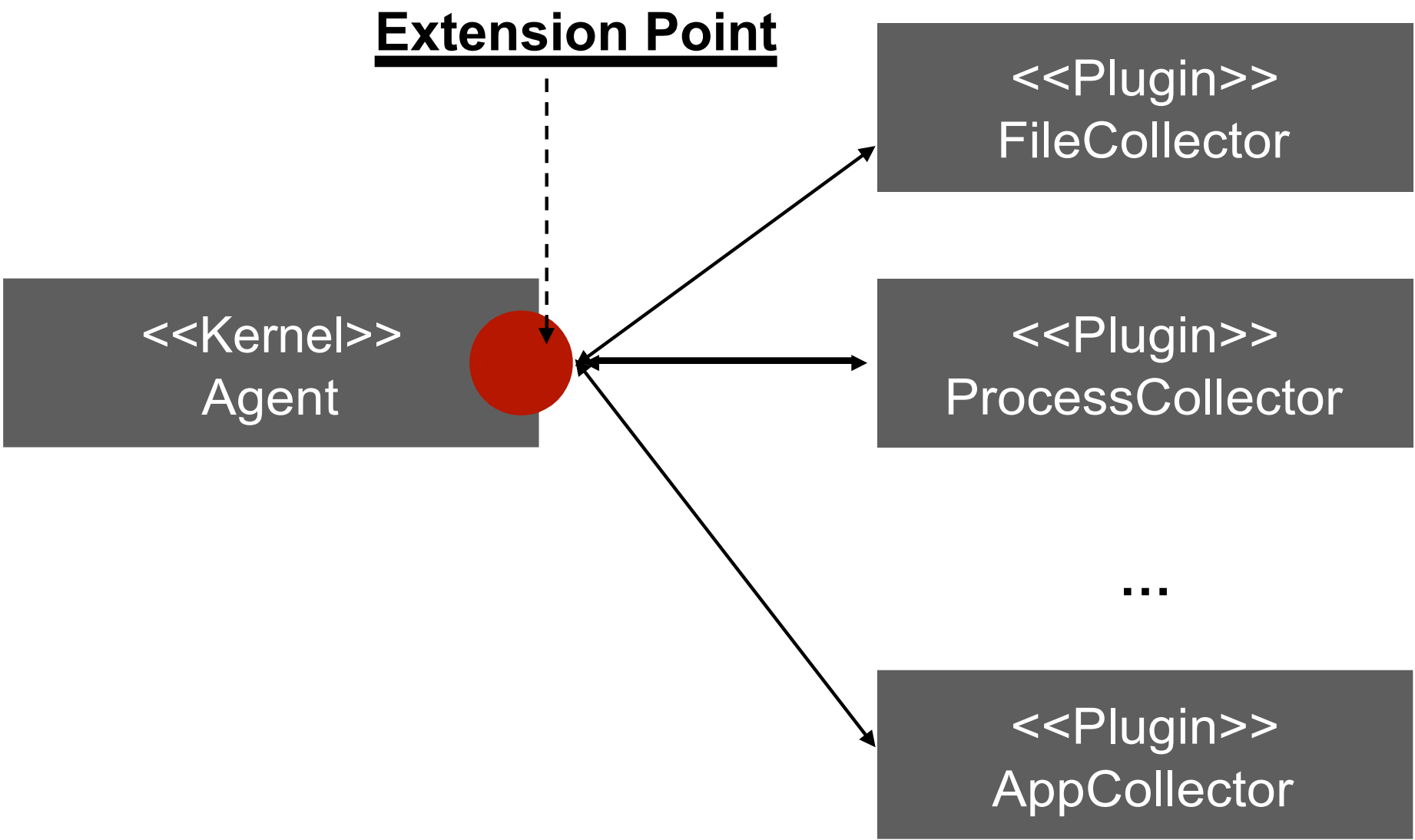
# Micro-Kernel

# Micro Kernel

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



# 示例例



# 常见任务

# JSON 解析

# 内置的JSON解析

- 利用反射实现，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 采用用代码生成而非非反射

```
goos: darwin
goarch: amd64
BenchmarkEmbeddedJson-4          200000      6360 ns/op
BenchmarkEasyJson-4             1000000     1396 ns/op
```

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)  
}
```

# 路由规则

- URL 分为两种，末尾是 `/`：表示一个子树，后面可以跟其他子路径； 末尾不是 `/`，表示一个叶子，固定的路径
  - 以 `/` 结尾的 URL 可以匹配它的任何子路径，比如 `/images` 会匹配 `/images/cute-cat.jpg`
- 它采用最长匹配原则，如果有多个匹配，一定采用匹配路径最长的那个进行处理
- 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"))
  }
```

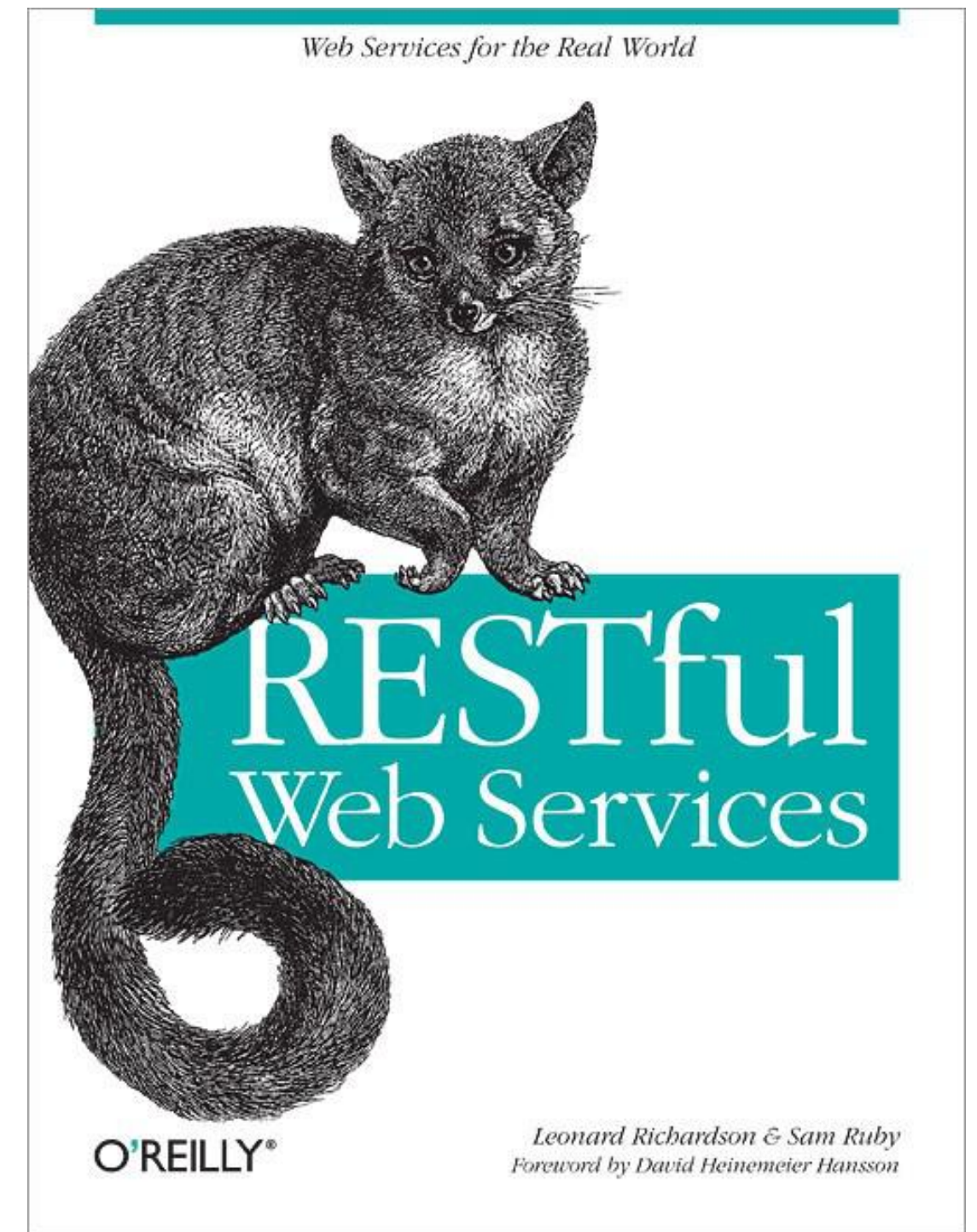
```
func main() {
    router := httprouter.New()
    router.GET("/", Index)
    router.GET("/hello/:name", Hello)

    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



# 性能分析工具

# 准备工作

- 安装 graphviz
  - `brew install graphviz`
- 将 `$GOPATH/bin` 加入 `$PATH`
  - Mac OS: 在 `.BASH_PROFILE` 中修改路径
- 安装 go-torch
  - `go get github.com/uber/go-torch`
  - 下载并复制 `FLAMEGRAPH.PL` (<https://github.com/brendangregg/FlameGraph>) 至 `$GOPATH/bin` 路径下
  - 将 `$GOPATH/bin` 加入 `$PATH`

# 通过文件方式输出 Profile

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

# 通过 HTTP 方式输出 Profile

- 简单，适合于持续性运行行的应用用
- 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`

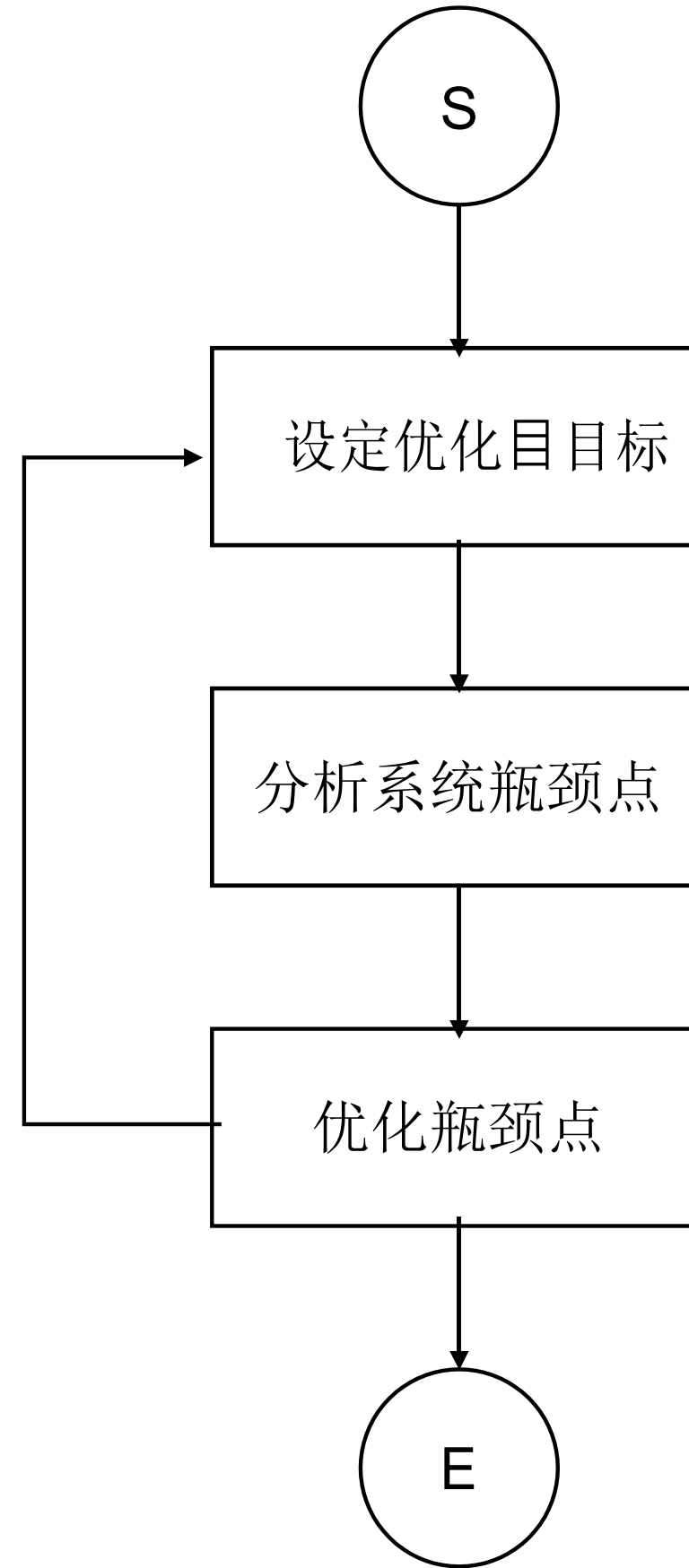
# Go 支持的多种 Profile

`go help testflag`

<https://golang.org/src/runtime/pprof/pprof.go>

# 性能调优示例例

# 性能调优过程



# 常见分析指标

- 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
```

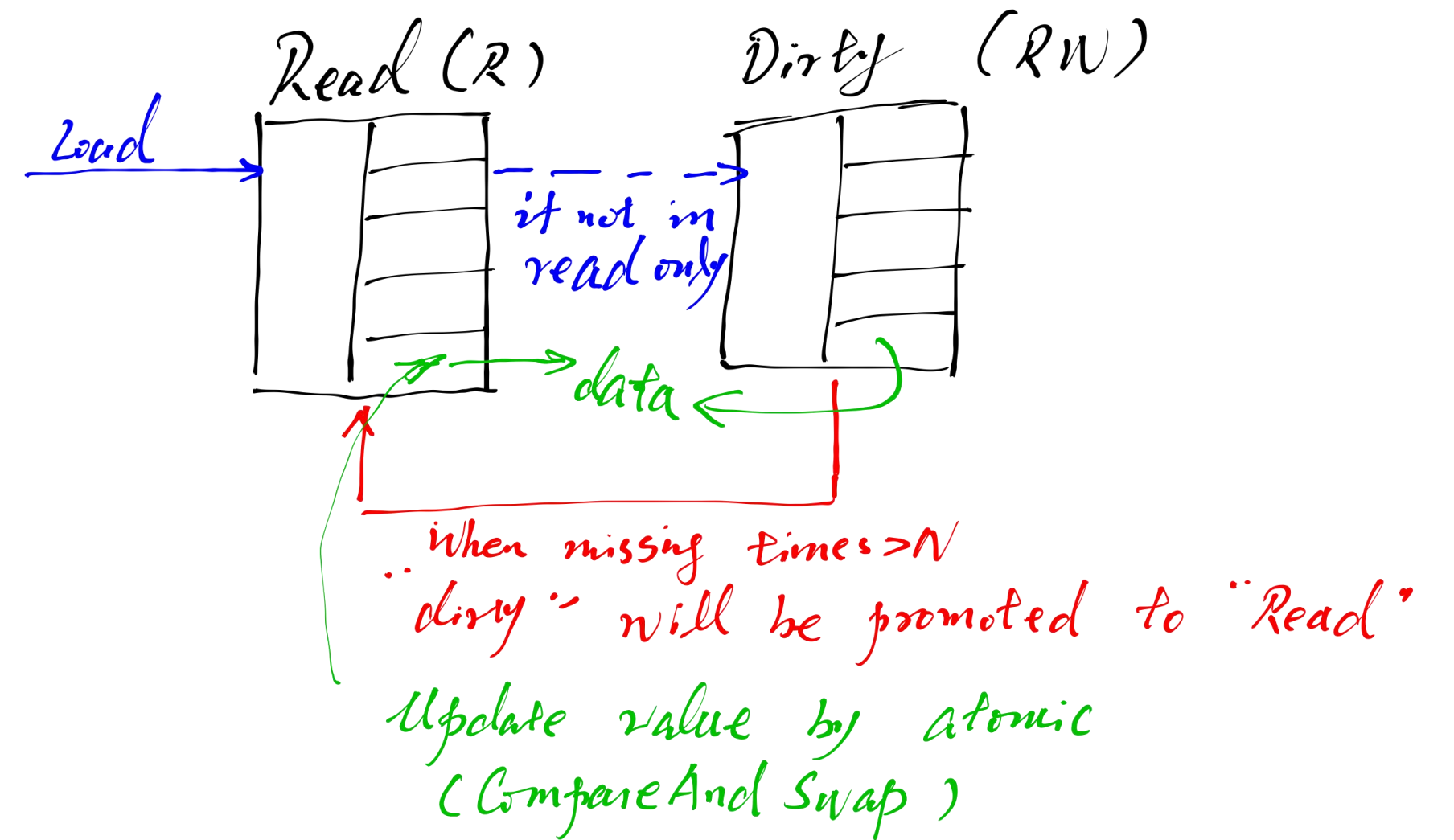
```
go-torch cpu.prof
```

# 编写高性能的Go程序

# 别让性能被“锁”住

# sync.Map

- 适合读多写少，且 **Key** 相对稳定的环境
- 采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

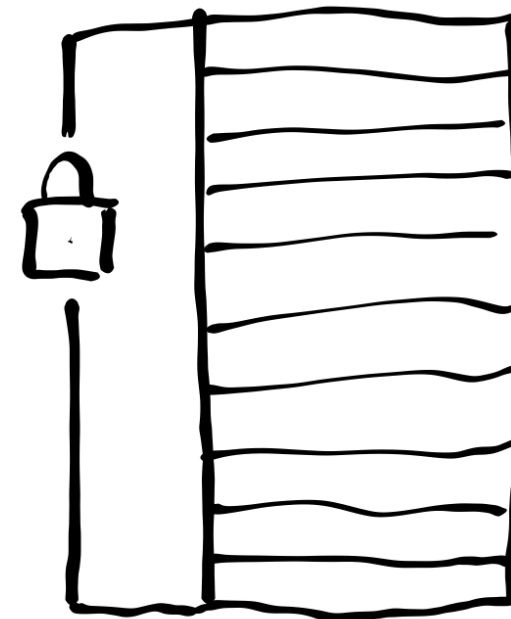


<https://my.oschina.net/qiangmzsx/blog/1827059>

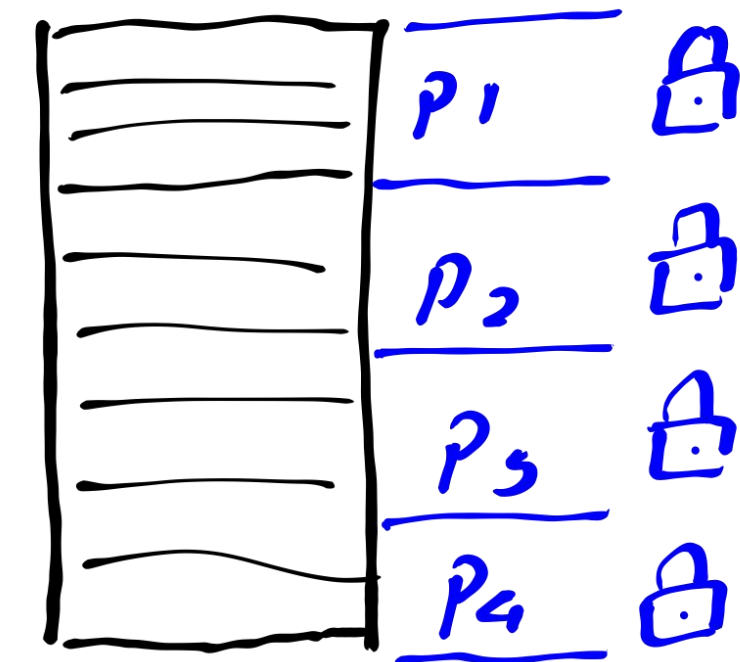
# Concurrent Map

- 适用于读写都很频繁的情况

*Map with lock*



*Concurrent Map*



BenchmarkSyncmap/map\_with\_RWLock-4  
BenchmarkSyncmap/sync.map-4  
BenchmarkSyncmap/concurrent\_map-4

1  
1  
1

8705993615 ns/op  
10499648643 ns/op  
4847699924 ns/op

[https://github.com/easierway/concurrent\\_map](https://github.com/easierway/concurrent_map)

# 别让性能被“锁”住

- 减少锁的影响范围
- 减少发生锁冲突的概率
  - sync.Map
  - ConcurrentMap
- 避免锁的使用
  - 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

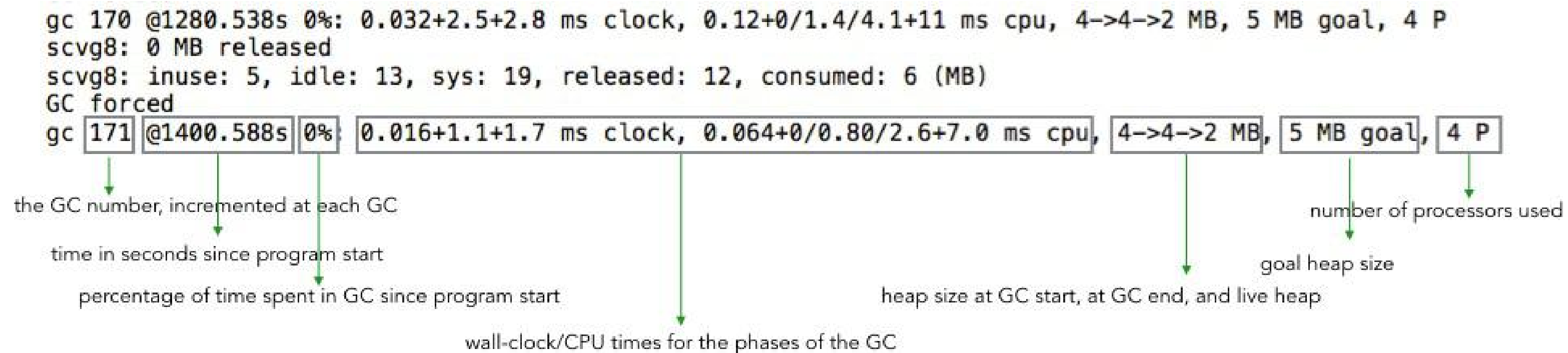
# 打开 GC 日志

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`

日志详细信息参考 : <https://golang.org/doc/runtime>



# go tool trace

## 普通程序输出 **trace** 信息

```
PACKAGE MAIN
import (
    "os"
    "runtime/trace"
)

func MAIN() {
    f, err := os.Create("TRACE.OUT")
    if err != nil {
        PANIC(err)
    }
    defer f.Close()

    err = trace.Start(f)
    if err != nil {
        PANIC(err)
    }
    defer trace.Stop()
    // Your PROGRAM here
}
```

## 测试程序输出 **trace** 信息

```
go test -trace trace.out
```

## 可视化 **trace** 信息

```
go tool trace trace.out
```

# 避免内存分配和复制

- 初始化至合适的大小
  - 自自动扩容是有代价的
- 复用内存

# 高高效的字符串串连接

# 高高可用用性架构设计

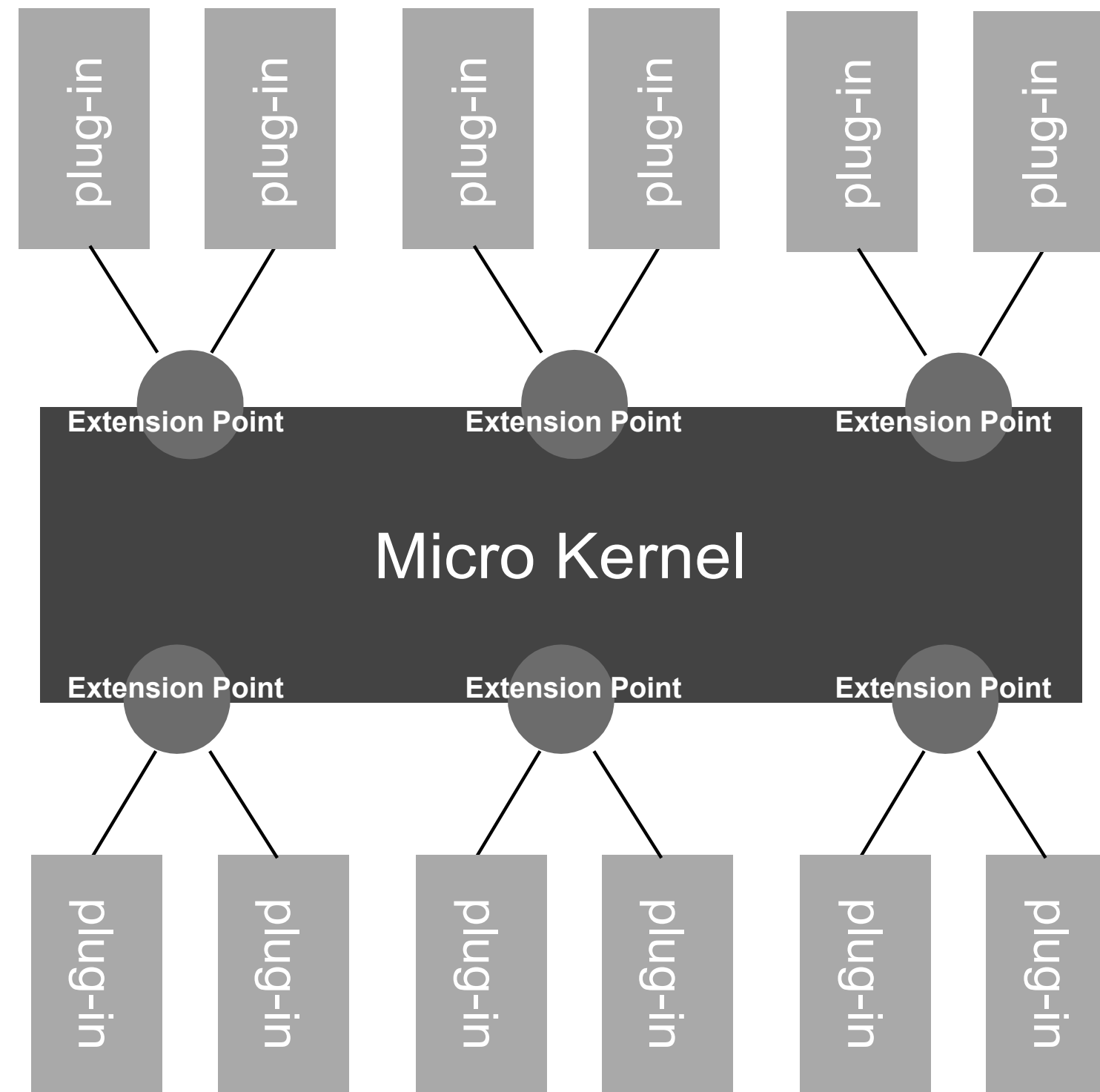
# 面面向错误的设计

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

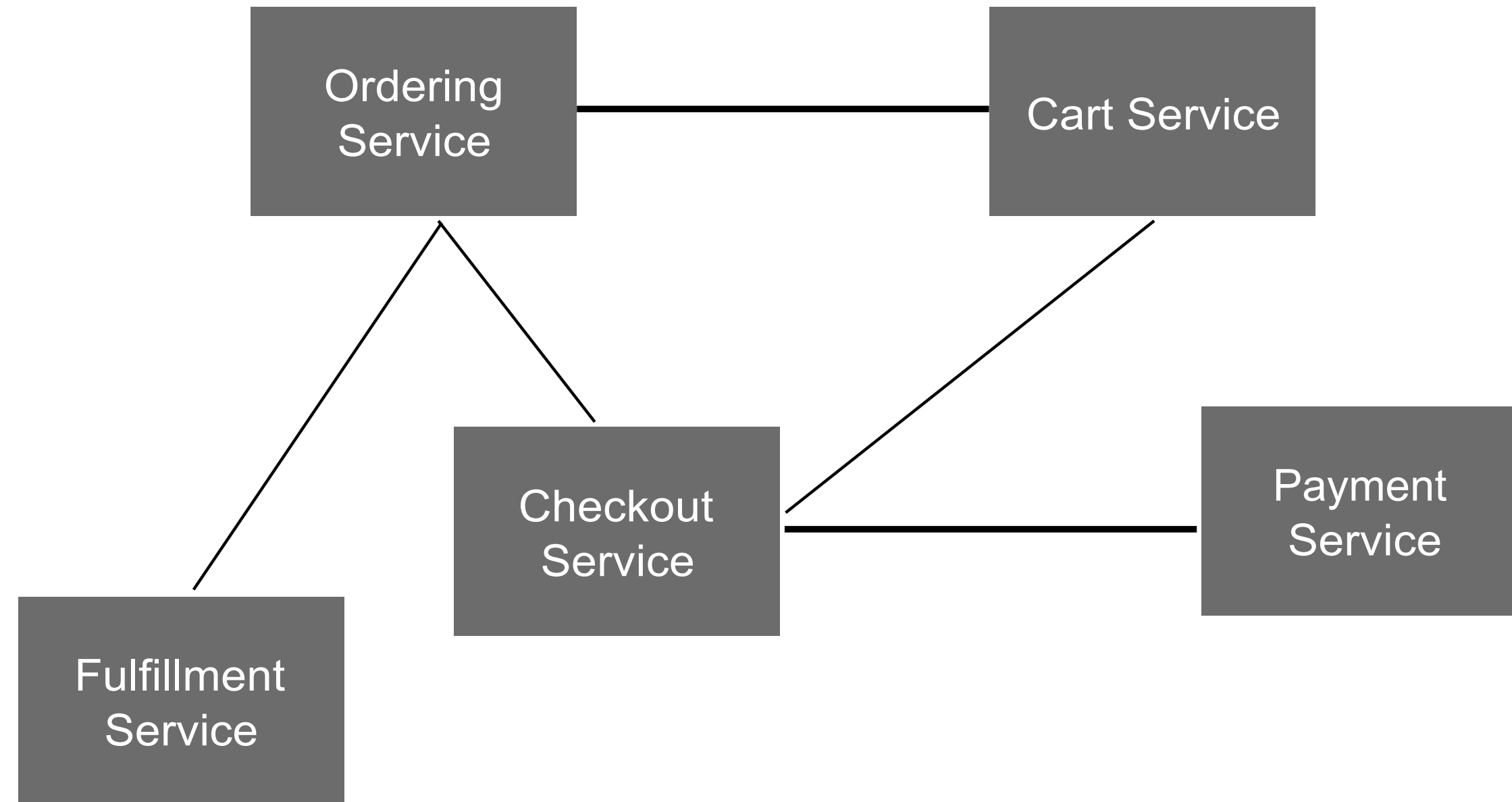


# 隔离

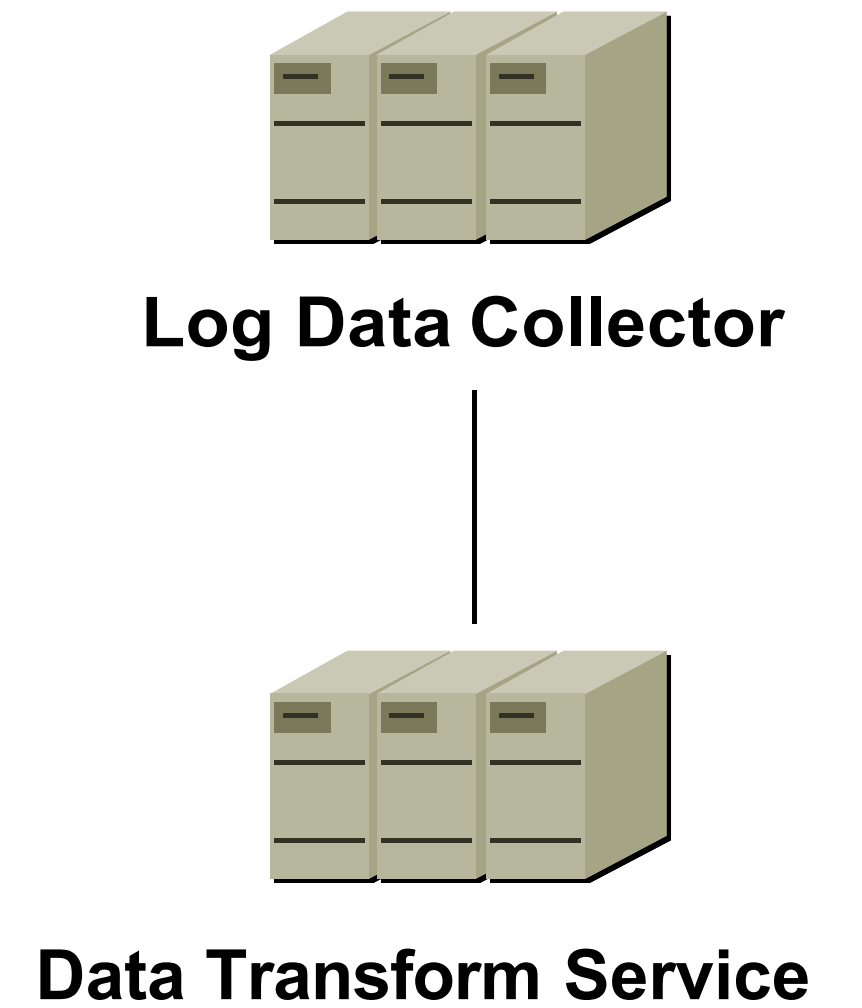
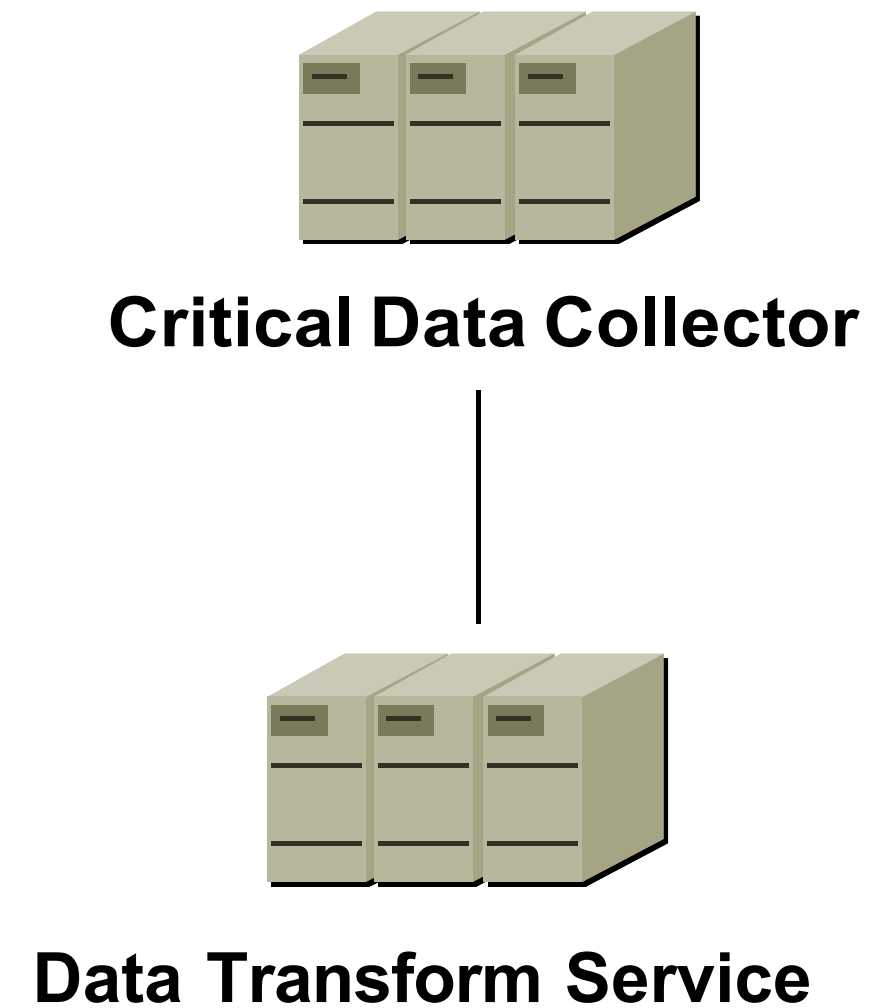
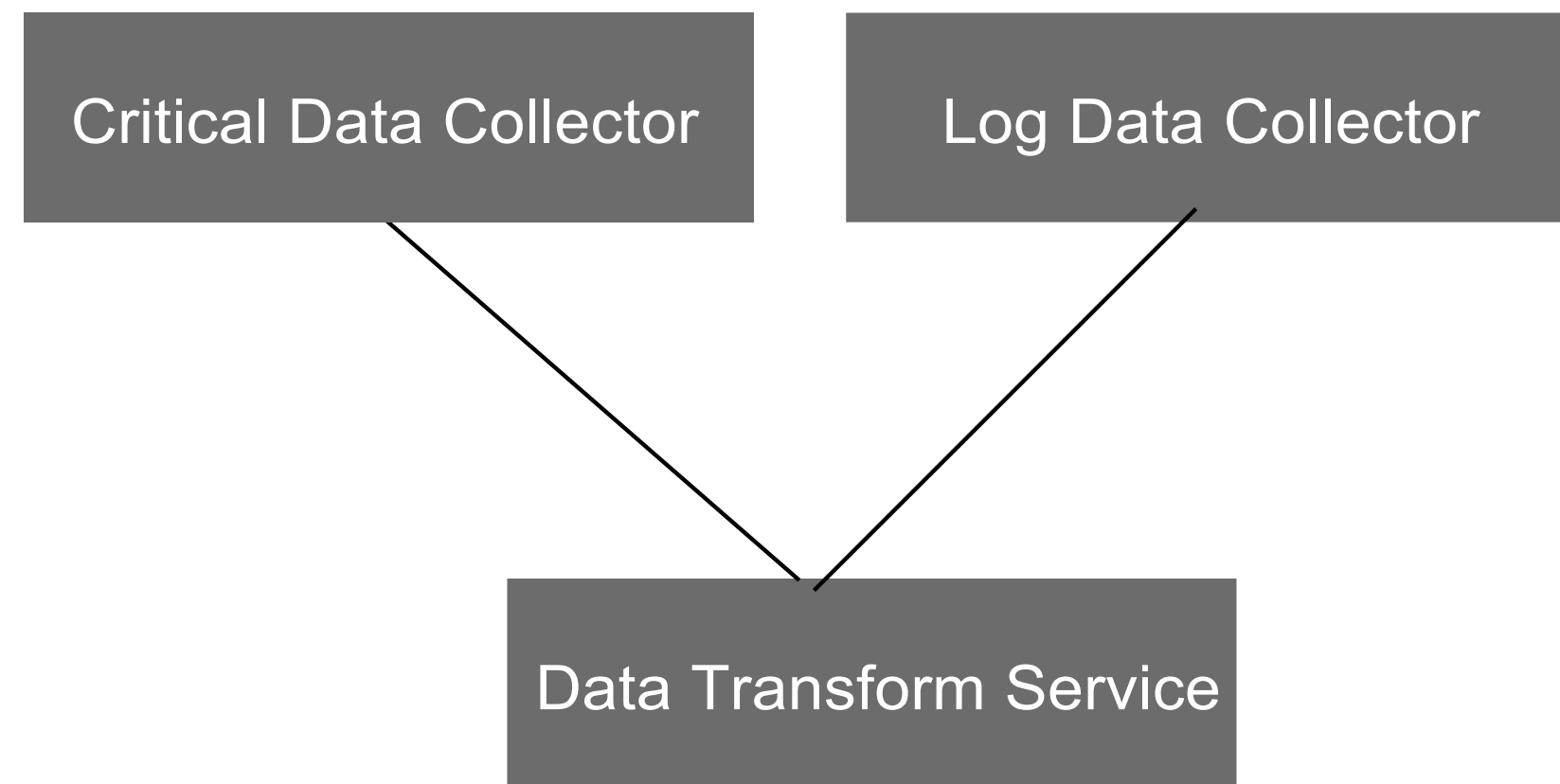
# 隔离错误 — 设计



# 隔离错误 — 部署



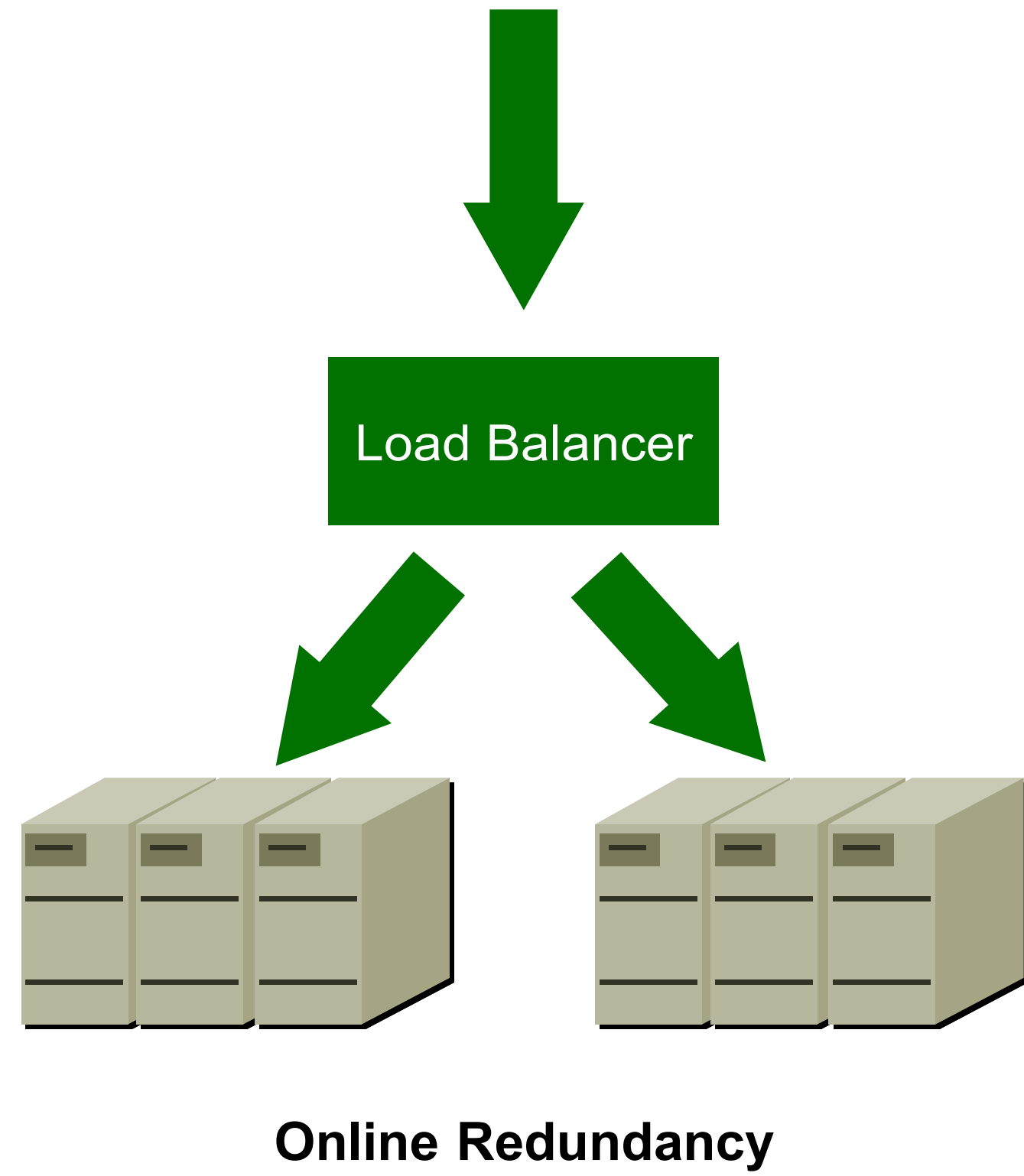
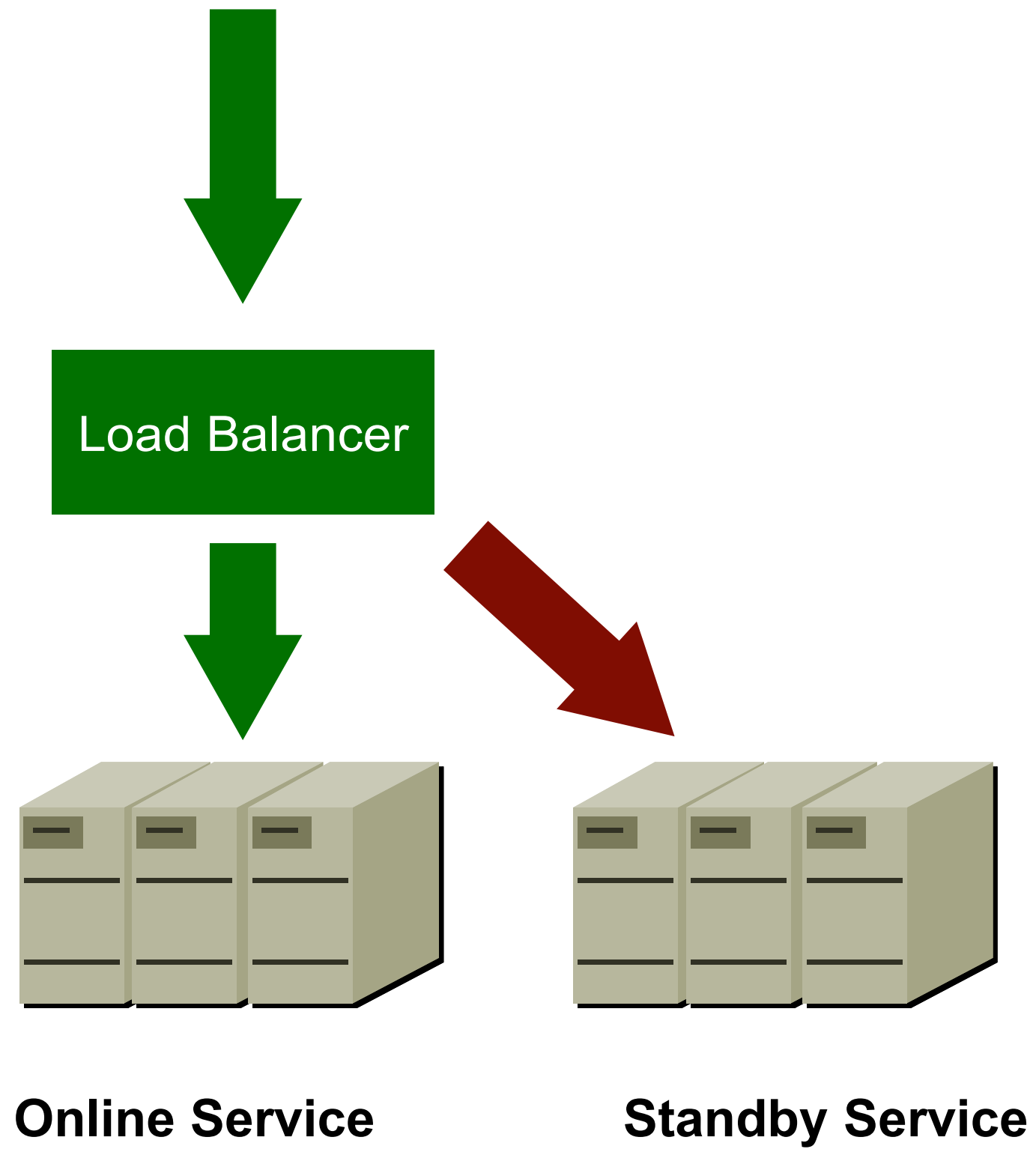
# 重用 vs 隔离



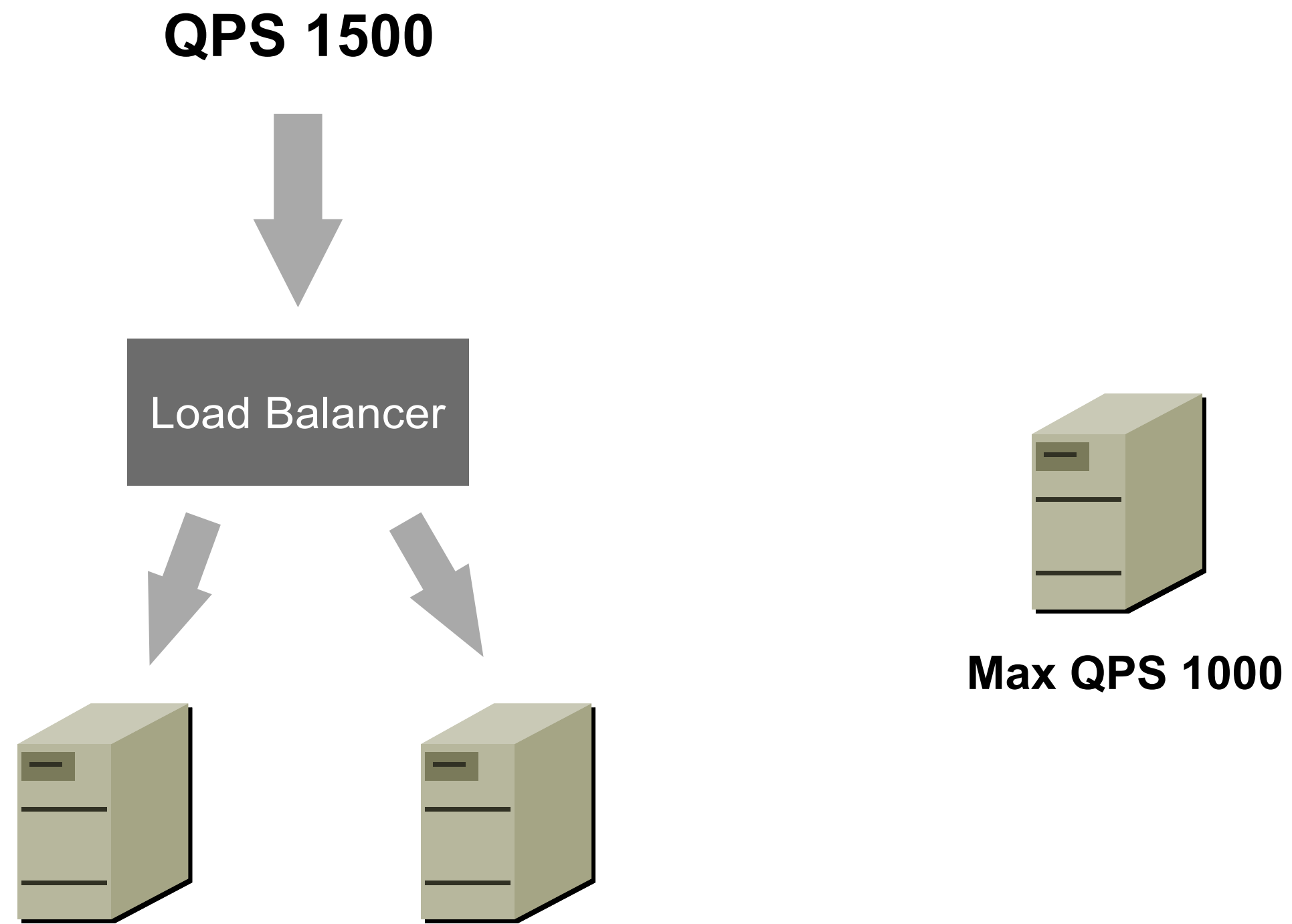
逻辑结构的重用 vs 部署结构的隔离

冗余

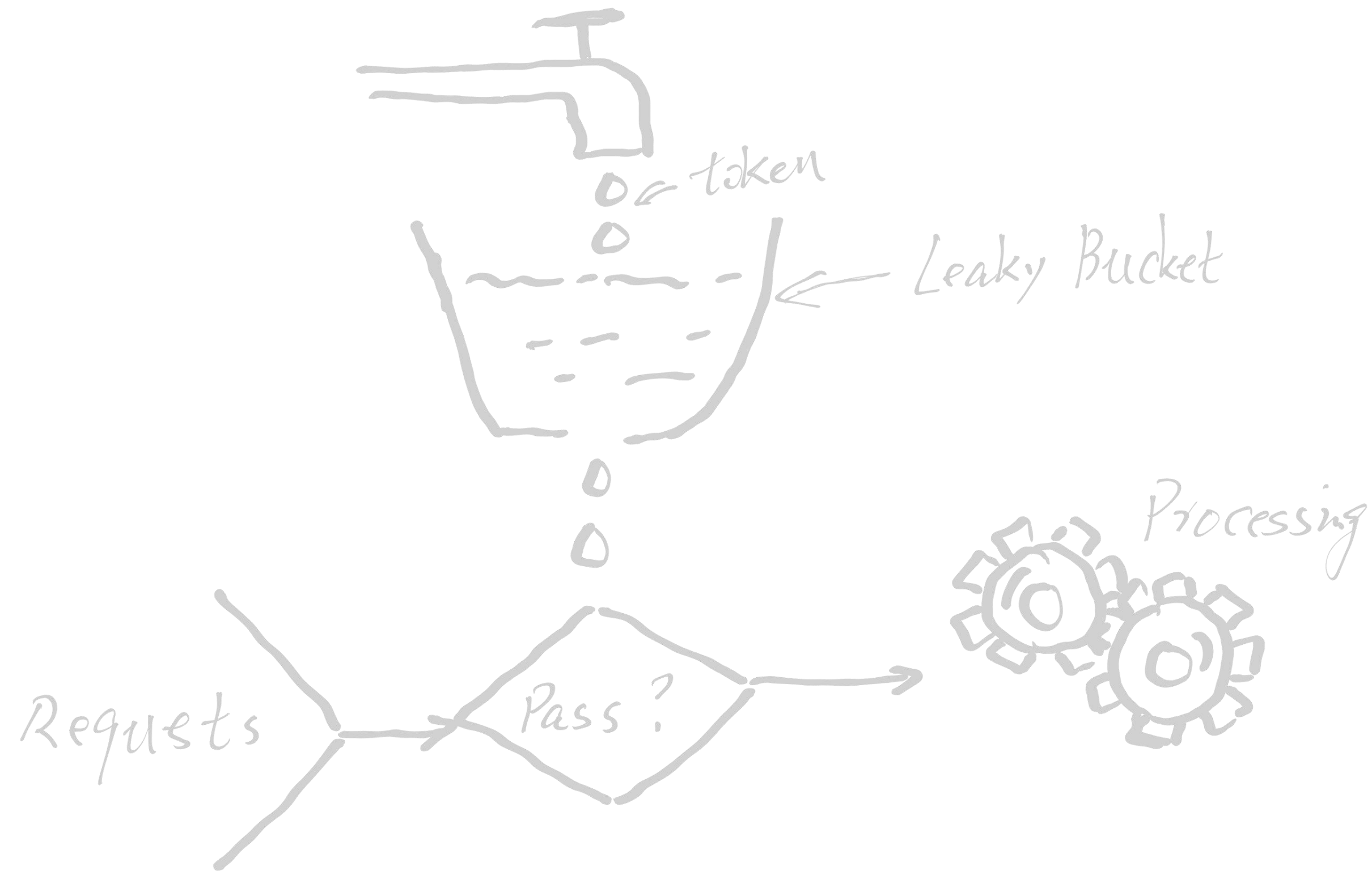
冗余



# 单点失效



# 限流

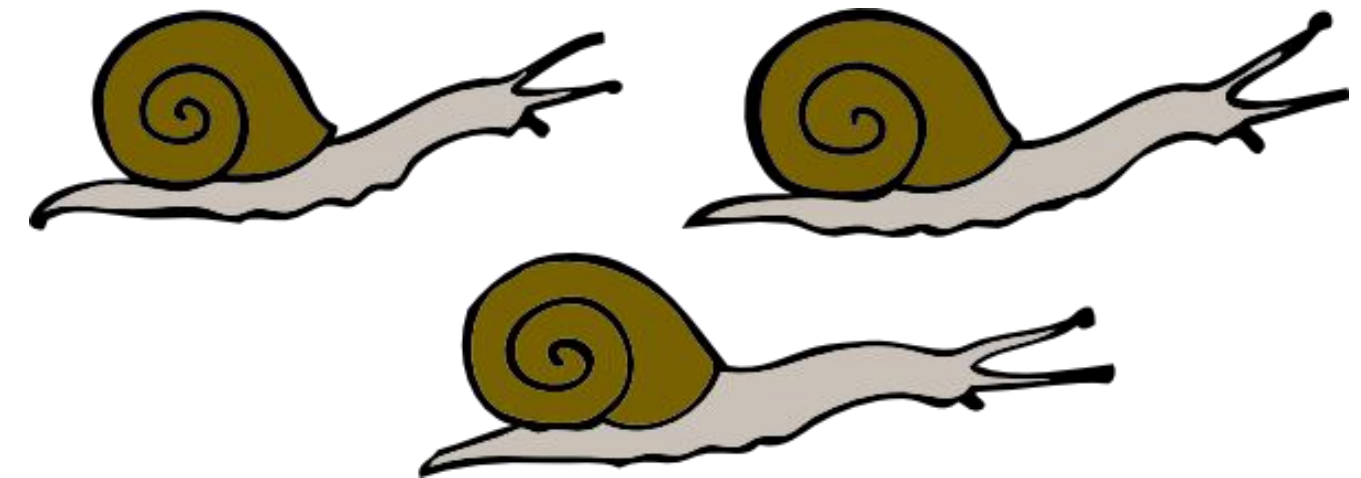




# 慢响应

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

*Pooled resources are exhausted!*

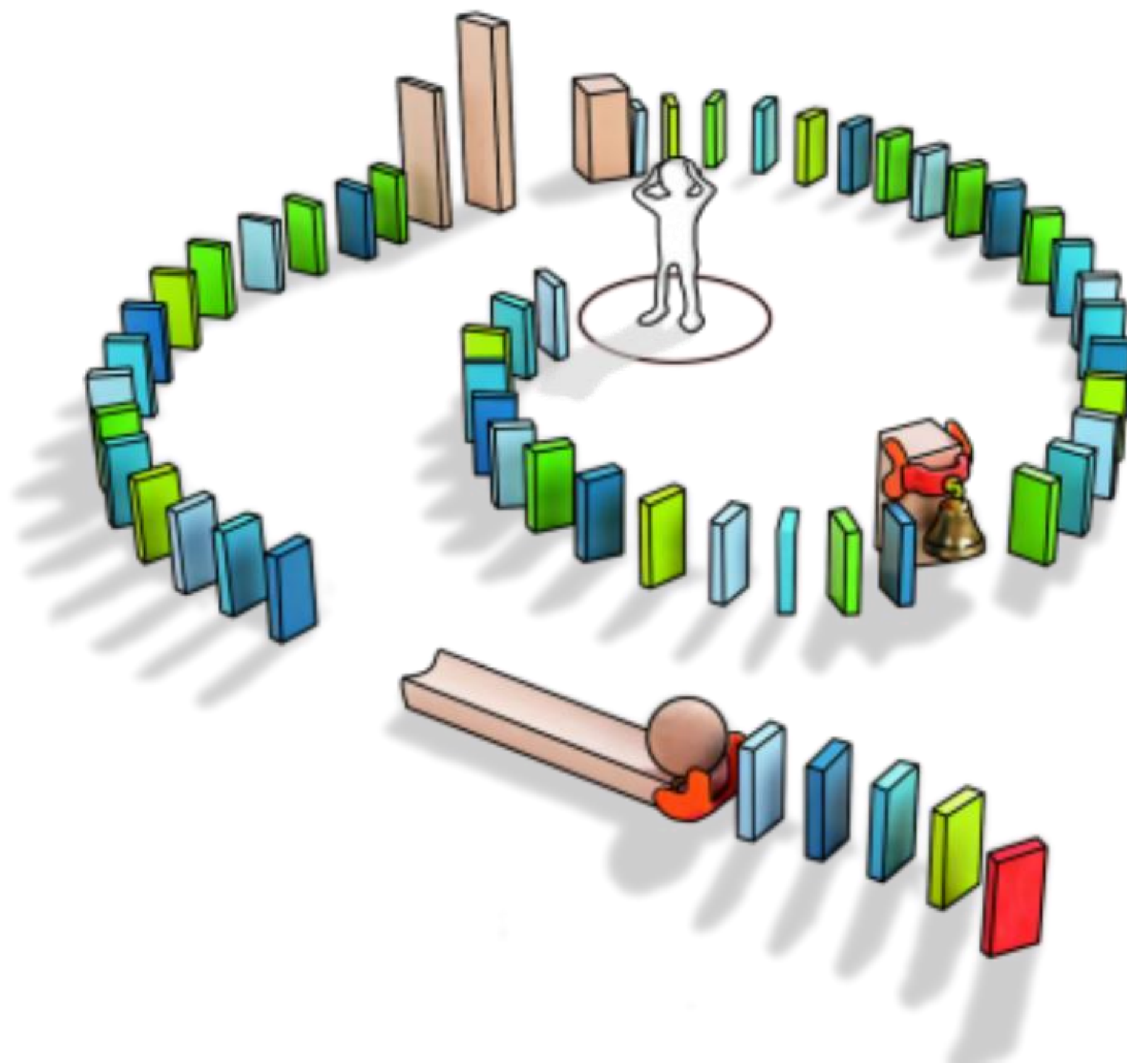


不要无休止的等待

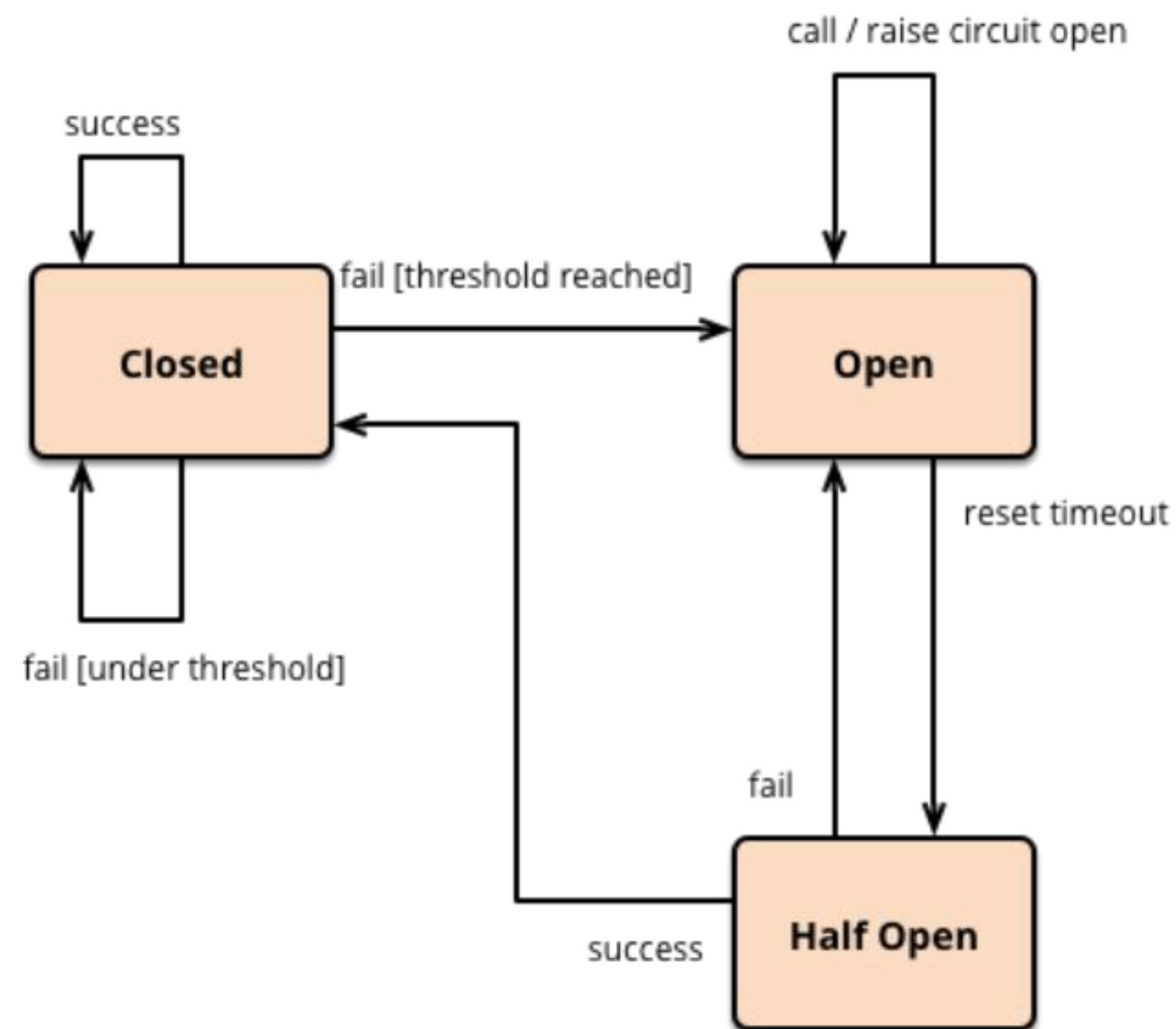


给阻塞操作都加上一个期限

# 错误传递



# 断路器



# 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
  - 在依赖服务不可用时，可以继续存活
- 快速启动
- 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!”**

# Chaos Engineering

- 如果问题经常发生生人人们就会学习和思考解决它的方法

## Chaos under control

Terminate host

Inject latency

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>**

# 相关开源项目目

<https://github.com/Netflix/chaosmonkey>

[https://github.com/easierway/service\\_decorators/blob/master/README.md](https://github.com/easierway/service_decorators/blob/master/README.md)

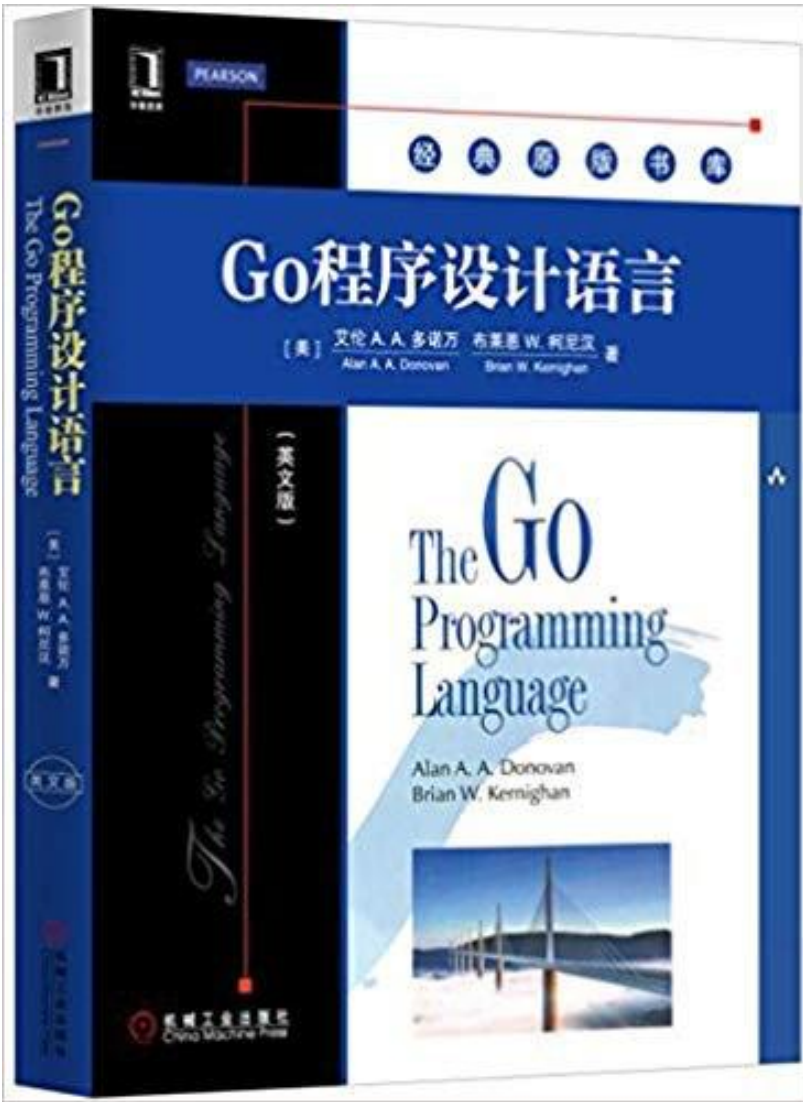
# 结束语



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语言言从入入门门到实战》