Golang与Java全方位对比总结

原创 腾讯程序员 腾讯技术工程 2023-04-06 18:01 发表于广东

作者：xindong

> 本文针对Golang与Java的基础语法、结构体函数、异常处理、并发编程及垃圾回收、资源消耗等各方面的差异进行对比总结，有不准确、不到位的地方还请大家不吝赐教。

一、基础语法

Golang: 编码风格及可见域规则严格且简单；Java: 来说层次接口清晰、规范，主要表现有以下这些。

1、变量

a、变量声明及使用

在Java中：变量可以声明了却不使用

```
public static String toString(int num) { int data = num; return String.valueOf(num); }
```

Golang中：声明的变量必须被使用，否则需要使用_来替代掉变量名，表明该变量不会比使用到

```
func toString(num int) string {
 data := num     // data没有使用者，无法编译
 return strconv.Itoa(num)
}

func toString(num int) string {
 _ := num       // 正常编译
 return strconv.Itoa(num)
}
```

b、变量声明及初始化

在Java中：如果在方法内部声明一个变量但不初始化，在使用时会出现编译错误；

```
public void compareVariable() {
        int age;
        Object object;
        System.out.println(age); // 编译错误
        System.out.println(object); // 编译错误
}
```

在Golang中：对于基本类型来讲，声明即初始化;对于引用类型，声明则初始化为nil。

```
func compareVariable() {
 var age int
 var hashMap *map[string]int
 fmt.Println(num) // num = 0
 fmt.Println(hashMap) //  &hashMap== nil
}
```

2、作用域规则

Java: 对方法、变量及类的可见域规则是通过private、protected、public关键字来控制的,具体如下

作用域	当前类	同一**package**	子孙类	其他**package**
public	√	√	√	√
protected	√	√	√	×
default（无修饰词）	√	√	×	×
private	√	×	×	×

Golang: 控制可见域的方式只有一个，当字段首字母开头是大写时说明其是对外可见的、小写时只对包内成员可见。

3、逗号 ok 模式

在使用Golang编写代码的过程中，许多方法经常在一个表达式返回2个参数时使用这种模式：,ok，第一个参数是一个值或者nil，第二个参数是true/false或者一个错误error。在一个需要赋值的if条件语句中，使用这种模式去检测第二个参数值会让代码显得优雅简洁。这种模式在Golang编码规范中非常重要。这是Golang自身的函数多返回值特性的体现。例如：

```
if _, ok := conditionMap["page"]; ok {
      //
}
```

4、结构体、函数以及方法

a、结构体声明及使用

在Golang中区别与Java最显著的一点是，Golang不存在“类”这个概念，组织数据实体的结构在Golang中被称为结构体。函数可以脱离“类”而存在，函数可以依赖于结构体来调用或者依赖于包名调用。Golang中的结构体放弃了继承、实现等多态概念，结构体之间可使用组合来达到复用方法或者字段的效果。

Golang 声明一个结构体并使用：

```
// User  定义User结构体
type User struct {
 Name string
 Age  int
}

// 使用一个结构体
func main() {
 personPoint := new(User) // 通过new方法创建结构体指针
 person1 := User{}        // 通过Person{}创建默认字段的结构体
 person2 := User{
  Name: "xiaoHong",
  Age:  21,
 }
 fmt.Println(personPoint) // &{ 0 }
 fmt.Println(person1)     // { 0 }
 fmt.Println(person2)     // {xiaoHong 21 }
}
```

Java声明实体并使用：

```
class User {
    private String name;
    private int age;
```

```
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public String print() {
        return "{name = " + name + ",age = " + age + "}";
    }
}

public class Demo {
    public static void main(String[] args) {
        User user = new User("xiaohong", 29);
        System.out.println("user信息：" + user.print());
    }
}
//执行结果
user信息：{name = xiaohong,age = 29}
```

b、函数和方法的区别

在Java中：所有的“函数”都是基于“类”这个概念构建的，也就是只有在“类”中才会包含所谓的“函数”，这里的“函数”被称为“方法”，可见上方声明实体并使用。

在Golang中：“函数”和“方法”的最基本区别是：函数不基于结构体而是基于包名调用，方法基于结构体调用。如下实例：

```
package entity

import "fmt"
```

```
type User struct {
 Name string
 Age  int
}

// User结构体/指针可调用的"方法"，属于User结构体
func (user *User) Solve() {
 fmt.Println(user)
}

// 任何地方都可调用的"函数"，不属于任何结构体，可通过entity.Solve调用
func Solve(user *User) {
 fmt.Println(user)
}
func main() {
 userPoint := new(entity.User) // 通过new方法创建结构体指针
 entity.Solve(userPoint) // 函数调用
 userPoint.Solve()    // 方法调用
}
```

5、值类型、引用类型以及****指针

Java：在Java中不存在显式的指针操作；8种基本数据类型是值类型，数组和对象属于引用类型。

Golang：而Golang中存在显式的指针操作，但是Golang的指针不像C那么复杂，不能进行指针运算；所有的基本类型都属于值类型，但是有几个类型比较特殊，表现出引用类型的特征，分别是slice、map、channel、interface，除赋值以外它们都可以当做引用类型来使用，因此当我们这样做时,可以直接使用变量本身而不用指针。

注：slice与数组的区别为是否有固定长度，slice无固定长度，数组有固定长度。值得注意的是，在Golang中，只有同长度、同类型的数组才可视为“同一类型”，譬如[]int和[3]int则会被视为不同的类型，这在参数传递的时候会造成编译错误。

a、数组对比

在Java中：当向方法中传递数组时，可以直接通过该传入的数组修改原数组内部值（浅拷贝）。 在Golang中：则有两种情况：在不限定数组长度(为slice)时也直接改变原数组的值，当限定数组长度时会完全复制出一份副本来进行修改（深拷贝）：

Java的数组实践:

```
    public static void main(String[] args) {
        int[] array = {1, 2, 3};
        change(array);
        System.out.println(Arrays.toString(array)); // -1,2,3
    }

    private static void change(int[] array) {
        array[0] = -1;
    }
```

Golang的数组实践:

```
// 不限定长度（即slice）:
func main() {
 var array = []int{1, 2, 3}
 change(array)
 fmt.Println(array)    // [-1 2 3]
}

func change(array []int) {
 array[0] = -1
}
// 限定长度（即数组）:
func main() {
 var array = [3]int{1, 2, 3}
 change(array)
 fmt.Println(array)    //[1 2 3]
}

func change(array [3]int) {
 array[0] = -1
}
```

b、对象对比

在Golang中：传入函数参数的是原对象的一个全新的copy（有自己的内存地址）;go对象之间赋值是把对象内存的 内容（字段值等） copy过去，所以才会看到globalUser修改前后的地址不变，但是对象的内容变了。 在Java中：传入函数参数的是原对象的引用的copy（指向的是同样的内存地址）; Java对象之间的赋值是把对象的引用 copy过去，因为引用指向的地址变了，所以对象的内容也变了。

Golang的对象实践：

```
//User 定义User结构体
type User struct {
   Name string
   Age int
}

// 定义一个全局的User
var globalUser = User {
   "xiaoming",
   28,
}

// modifyUser 定义一个函数，参数为User结构体“对象”，将全局globalUser指向传递过来的User结构体“对象”
func modifyUser(user User) {
   fmt.Printf("参数user的地址 = %p\n",&user)
   fmt.Printf("globalUser修改前的地址 = %p\n",&globalUser)
   fmt.Println("globalUser修改前 = ",globalUser)
   // 修改指向
   globalUser = user
   fmt.Printf("globalUser修改后的地址 = %p\n",&globalUser)
   fmt.Println("globalUser修改后 = ",globalUser)
}

func main() {
   var u User = User {
      "xiaohong",
      29,
   }
   fmt.Printf("将要传递的参数u的地址 = %p\n",&u)
   modifyUser(u)
}

// 执行结果
将要传递的参数u的地址 = 0xc0000ac018
参数user的地址 = 0xc0000ac030
globalUser修改前的地址 = 0x113a270
globalUser修改前 = {xiaoming 28}
globalUser修改后的地址 = 0x113a270
globalUuser修改后 = {xiaohong 29}
```

```
Java的对象实践验证：class User {
    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public String print() {
        return "{name = " + name + ",age = " + age + "}";
    }
}

public class Demo {
    private static User globalUser = new User("xiaoming",28);
    public static void modifyUser(User user) {
        System.out.println("参数globalUser的地址 = " + user);
        System.out.println("globalUser修改前的地址 = " + globalUser);
        System.out.println("globalUser修改前 = " + globalUser.print());
        globalUser = user;
        System.out.println("globalUser修改后的地址 = " + globalUser);
        System.out.println("globalUser修改后 = " + globalUser.print());
    }
    public static void main(String[] args) {
        User user = new User("xiaohong", 29);
        System.out.println("将要传递的参数user的地址 = " + user);
        modifyUser(user);
    }
}

//执行结果
将要传递的参数user的地址 = com.example.demo.User@5abca1e0
参数globalUser的地址 = com.example.demo.User@5abca1e0
globalUser修改前的地址 = com.example.demo.User@2286778
```

```
globalUser修改前 = {name = xiaoming,age = 28}
globalUser修改后的地址 = com.example.demo.User@5abca1e0
globalUser修改后 = {name = xiaohong,age = 29}
```

c、指针的区别

在Java中：如果传递了引用类型（对象、数组等）会复制其指针进行传递 在Golang中：必须要显式传递Person的指针，不然只是传递了该对象的一个副本。

Golang的指针：

```
// User  定义User结构体
type User struct {
 Name string
 Age  int
}

func main() {
 p1 := User{
  Name: "xiaohong",
  Age:  21,
 }
 changePerson(p1)
 fmt.Println(p1.Name) // xiaohong
 changePersonByPointer(&p1)
 fmt.Println(p1.Name) // xiaoming
}

func changePersonByPointer(user *User) {
 user.Name = "xiaoming"
}

func changePerson(user User) {
 user.Name = "xiaoming"
}
```

Java的指针：

```
public class Demo {
```

```
    public static void changePerson(User user) {
        user.setName("xiaoming");
    }
    public static void main(String[] args) {
        User user = new User("xiaohong", 29);
        changePerson(user);
        System.out.println("user信息：" + user.getName());      // xiaoming
    }

}


class User {
    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}
```

二、面向对象

在Golang中：没有明确的OOP概念，*Go*语言只提供了两个关键类型：struct，interface。 在Java中： 面向对象语言的封装、继承、多态的特性以及“继承（extends）、实现（implements）”等关键字。

1、Java的OOP与Golang的结构体组合

假设有这么一个场景：动物（Animal）具备名字（Name）、年龄（Age）的基本特性，现在需要实现一个狗(Dog)，且Dog需要具备Animal所需的所有特性，并且自身具备犬吠（bark()）的动作。

首先来看看最熟悉的Java要如何写，很简单，使用抽象类描述Animal作为所有动物的超类，Dog extends Animal：

```
public abstract class Animal {
    String name;
    int age;
}

public class Dog extends Animal {
    public void bark() {
        System.out.println(age + "岁的" + name + "在汪汪叫");
    }
}

public class Demo {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.name = "小龙";
        dog.age = 2;
        dog.bark(); // 2岁的小龙在汪汪叫
    }
}
```

在Golang中，可以这样通过结构体的组合来实现：

```
package main

import "fmt"

type Animal struct {
 Name string
 Age  int
}

type Dog struct {
 *Animal
}
```

```
func (dog *Dog) Bark() {
 fmt.Printf("%d岁的%s在汪汪叫", dog.Age, dog.Name)
}

func main() {
 dog := &Dog{&Animal{
  Name: "小龙",
  Age:  2,
 }}
 dog.Bark() // 2岁的小龙在汪汪叫...
}
```

2、侵入式与非侵入式接口

在Java中：接口主要作为不同组件之间的契约存在。对契约的实现是强制的，你必须声明你的确实现了该接口。这类接口我们称为侵入式接口。“侵入式”的主要表现在于实现类需要明确声明自己实现了某个接口。

在Golang中：非侵入式接口不需要通过任何关键字声明类型与接口之间的实现关系，只要一个类型实现了接口的所有方法，那么这个类型就是这个接口的实现类型。

Java：管理狗的行为，可以通过以下接口实现：

```
public interface Dog {
    void Bark();
}

public class DogImpl implements Dog{

    @Override
    public void Bark() {
        System.out.println("汪汪叫");
    }
}

public class Demo {

    public static void main(String[] args) {
        Dog dog = new DogImpl();
        dog.Bark();    // 汪汪叫
    }

}
```

Golang: 假设现在有一个Factory接口，该接口中定义了Produce()方法及Consume()方法，CafeFactory结构体作为其实现类型，那么可以通过以下代码实现：

```
package entity

type Factory interface {
 Produce() bool
 Consume() bool
}

type CarFactory struct {
 ProductName string
}

func (c *CarFactory) Produce() bool {
 fmt.Printf("CarFactory生产%s成功", c.ProductName)
 return true
}

func (c *CarFactory) Consume() bool {
 fmt.Printf("CarFactory消费%s成功", c.ProductName)
```

```
 return true
}

// --------------
package main

func main() {
 factory := &entity.CarFactory{"Car"}
 doProduce(factory)
 doConsume(factory)
}


func doProduce(factory entity.Factory) bool {
 return factory.Produce()
}

func doConsume(factory entity.Factory) bool {
 return factory.Consume()
}
```

Golang的非侵入式接口优点：简单、高效、按需实现

在Go中，类没有继承的概念，只需要知道这个类型实现了哪些方法，每个方法是啥行为。

实现类型的时候，只需要关心自己应该提供哪些方法，不用再纠结接口需要拆得多细才合理。接口由使用方按需定义，而不用事前规划

减少包的引入，因为多引用一个外部的包，就意味着更多的耦合。接口由使用方按自身需求来定义，使用方无需关心是否有其他模块定义过类似的接口

Java的侵入式接口优点: 层次结构清晰，对类型的动作行为有严格的管理

三、异常处理

在Java中: 通过try..catch..finally的方式进行异常处理，有可能出现异常的代码会被try块给包裹起来，在catch中捕获相关的异常并进行处理，最后通过finally块来统一执行最后的结束操作（释放资源）。

在Golang中：错误处理方式有两种方式：**,ok模式** 与 **defer**、**panic**及**recover**的组合

1、Java的异常处理：

```java
public class ExceptionTest {
    public static void main(String[] args) {
        FileInputStream fileInputStream = null;
        try{
            fileInputStream = new FileInputStream("test.txt");
        }catch (IOException e){
            System.out.println(e.getMessage());
            e.printStackTrace();
            return;
        }finally {
            if(fileInputStream!=null){
                try {
                    fileInputStream.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("回收资源");
        }
    }
}
```

2、Golang的异常处理：

Golang的**,ok模式**。所有可能出现异常的方法或者代码直接把错误当作第二个响应值进行返回，程序中对返回值进行判断，非空则进行处理并且立即中断程序的执行。 优点：这种比Java的简单很多，是Golang在异常处理方式上的一大特色。

缺点：代码冗余，所有的异常都需要通过 if err != nil {} 去做判断和处理，不能做到统一捕捉和处理，容易遗漏。

```go
func main() {
 value, err := Bark()

 if err != nil {
  // 返回了异常，进行处理
  log.error("...异常: ", err)
  return err
 }

 // Bark方法执行正确，继续执行后续代码
```

```
 Process(value)
}
```

Golang的**defer**、**panic**及**recover**

defer是Golang错误处理中常用的关键字，pannic及recover是Golang中的内置函数，通常与defer结合进行错误处理，它们各自的用途为:

defer的作用是延迟执行某段代码，一般用于关闭资源或者执行必须执行的收尾操作，无论是否出现错误defer代码段都会执行，类似于Java中的finally代码块的作用；defer也可以执行函数或者是匿名函数:

```
defer func() {
 // 清理工作
} ()

// 这是传递参数给匿名函数时的写法
var num := 1
defer func(num int) {
 // 做你复杂的清理工作
} (num)

 需要注意的是，defer使用一个栈来维护需要执行的代码，所以defer函数所执行的顺序是和defer声明的顺序相反的。
defer fmt.Println(a)
defer fmt.Println(b)
defer fmt.Println(c)

执行结果：
c
b
a
```

panic的作用是抛出错误，制造系统运行时恐慌，当在一个函数执行过程中调用panic()函数时，正常的函数执行流程将立即终止，但函数中之前使用defer关键字延迟执行的语句将正常展开执行，之后该函数将返回到调用函数，并导致逐层向上执行 panic流程，直至所属的goroutine中所有正在执行的函数被终止，panic和Java中的throw关键字类似，用于抛出错误，阻止程序执行。

recover的作用是捕捉panic抛出的错误并进行处理，需要联合defer来使用，类似于Java中的catch代码块：

```
func main() {
      fmt.Println("main begin")
      // 必须要先声明defer，否则不能捕获到panic异常
      defer func() {
        fmt.Println("defer begin")
        if err := recover(); err != nil {
            // 这里的err其实就是panic传入的内容
            fmt.Println(err)
        }
         fmt.Println("defer end")
      }()
      test()
      // test中出现错误，这里开始下面代码不会再执行
      fmt.Println("main end")
}

func test() {
   fmt.Println("test begin")
   panic("error")
   //这里开始下面代码不会再执行
   fmt.Println("test end")
}


//执行结果
main begin
test begin
defer begin
error
defer end
```

注：利用recover处理panic指令，defer必须在panic之前声明，否则当panic时，recover无法捕获到panic。

四、并发编程

Java 中 CPU 资源分配对象是 Thread，Go 中 CPU 资源分配对象是 goroutine。Java Thread 与系统线程为一一对应关系，goroutine 是 Go 实现的用户级线程，与系统线程是 m:n 关系。

1、Java 和 Golang 的基本实现：

在 Java 中，如要获得 CPU 资源并异步执行代码单元，需要将代码单元包装成 Runnable，并创建可以运行代码单元的 Thread ,执行 start 方法启动线程。

```
Runnable task = ()-> System.out.println("task running");
Thread t = new Thread(task);
t.start();
```

Java 应用一般使用线程池集中处理任务，以避免线程反复创建回收带来的开销。

```
Runnable task = ()-> System.out.println("task running");
Executor executor = Executors.newCachedThreadPool();
executor.execute(task);
```

在 Golang 中，则需要将代码包装成函数。使用 go 关键字调用函数之后，便创建了一个可以运行代码单元的 goroutine。一旦 CPU 资源就绪，对应的代码单元便会在 goroutine 中执行。

```
go func() {
  fmt.Println("test task running")
}()
```

2、Java 和 Golang 的区别：

Golang语言采用了CSP（Communicating Sequential Processes）的模型，其中以goroutine和channel作为主要实现手段。 Java则采用了多线程模型，其中以Thread和Synchronization作为主要实现手段。 Golang语言的goroutine是一种轻量级的线程，它们的创建和销毁速度比Java中的线程快得多。在Java中，创建和销毁线程都需要相当大的开销。

Golang语言的channel是一种同步数据传递的机制，它可以方便地解决多道程序之间的通信问题。Java中则需要使用同步工具（如Semaphore、CountDownLatch等）来解决多线程之间的通信问题。

Java 和 **Go** 官方库中同步方式的对应关系

	Java	Golang
锁	synchronized,ReentrantLock	sync.Mutex, one unit buffered channel
读写锁	ReentrantReadWriteLock, StampedLock	sync.RWMutex
条件变量	condition	sync.Cond
CAS/Atomic	Varhandle、volatile，Atomic 类	atomic.Value，atomic 包
once	单例模式	sync.Once

a、Java synchronized 与Golang Mutex

Java synchronized：线程 A 在 t1 时刻释放 JVM 锁后（monitor exit)，在随后的 t2 时刻，若任意线程 B 获取到 JVM 锁（monintor enter），则线程 A 在 t1 时刻之前发生的所有写入均对 B 可见。synchronized 是 JVM 内置锁实现，写入 volatile 变量相当于 monitor exit，读取 volatile 变量相当于 monintor enter。（即一把锁只能同时被一个线程获取，没有获得锁的线程只能阻塞等待）

synchronized的使用： 修饰一个代码块，被修饰的代码块称为同步代码块，作用范围是大括号{}括起来的代码；

```
public void method()
{
   synchronized(this) {
     // todo some thing
   }
}
```

修饰一个方法，被修饰的方法称为同步方法，其作用范围是整个方法；

```
public synchronized void method()
{
   // todo some thing
}
```

修改一个静态方法，作用范围是整个静态方法；

```
public synchronized static void method() {
   // todo some thing
}
```

修改一个类，作用范围是synchronized后面括号括起来的部分。

```
class DemoClass {
   public void method() {
      synchronized(DemoClass.class) {
         // todo some thing
      }
   }
}
```

Go Mutex：Go 并未像 Java 一样提供 volatile 这样基础的关键字，但其 Mutex 相关内存模型和 synchronized 或 Java 官方库 Lock 实现有十分接近语义。若 goroutine A 在 t1 时刻释放 sync.Mutex 或 sync.RWMutex 后，在随后的 t2 时刻，若任意 goroutine B 获取到锁，则 goroutine A 在 t1 时刻之前发生的所有写入均对 B 可见。

Mutex的使用：

修饰关键代码：每次只有一个线程对这个关键变量进行修改，避免多个线程同时这个关键代码进行操作。

```
func main() {

 var mutex sync.Mutex

 count := 0

 for i := 0; i < 100; i++ {
  go func() {
   mutex.Lock() // 加锁
   count += 1
   mutex.Unlock() // 解锁
  }()
 }

 // 休眠，等待2s
 time.Sleep(time.Second * 2)
 // 100，没有加锁结果不正确
 fmt.Println("count = ", count)
}
```

修饰结构体： 带锁结构体初始化后，直接调用对应的线程安全函数就可以。

```
type count struct {
 lock  sync.Mutex
 value int
}

// 结构体对应的结构方法
func (receiver *count) countOne() {
 receiver.lock.Lock()
 defer receiver.lock.Unlock()
 receiver.value++
}

func main() {

 c := count{
  lock:  sync.Mutex{},
  value: 0,
 }
 group := sync.WaitGroup{}
 for i := 0; i < 10; i++ {
  group.Add(1)
  go func(count2 *count) {
   defer group.Done()
   for i := 0; i < 100; i++ {
    count2.countOne()
   }
  }(&c)
 }
 group.Wait()
 fmt.Printf("The count value is %d", c.value)    // The count value is 1000
}
```

b、条件变量

Java 和 Golang 相似点：一般来说，条件变量衍生于锁，不同条件变量只是同一锁空间下的不同等待队列。Java 可以使用 synchronized 代码块保护特定代码路径，兼而可以在 synchronized 代码块中使用 Object wait 和 notify、notifyall 方法实现单一条件等待。如果需要多个条件，可以使用官方库提供的 Lock 实现和 Condition 实现。

Java 和 Golang 区别点：Java 创建条件变量的方式是调用 Lock 接口 newCondition 方法。Go sync.Cond 结构体需设置 sync.Mutex 字段才能工作，挂起方法为 Wait，唤醒方法为 Braodcast。Go 语言里面条件变量的通知 Signal() 和 Broadcast()，并没有在锁的保护下执行，而是在 Unlock() 之后执行。

c、CAS/Atomic

原子性：一个或者多个操作在 CPU 执行的过程中不被中断的特性，称为原子性（atomicity）。CAS是乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。Java 和 Go 均支持 CAS 及原子操作。

在Java中： CAS 操作由 volatile 关键字和 VarHandle（9 之前是 UnSafe）支持，在此基础上有了 Atomic 类和并发包中的大量无锁实现（如 ConcurrentHashMap, AQS 队列等）。

在Golang中：atomic.Value 提供了 CAS 操作基础，它保证任意类型（interface {}) 的 Load 和 Store 为原子操作，在此基础上有 atomic 包。

d、Once 与单例模式

sync.Once 是 Golang 标准库提供的使函数只执行一次的实现，常应用于单例模式，例如初始化配置、保持数据库连接等。它有 2 个特性：

保证程序运行期间某段代码只会执行一次

如果多个 goroutine 同时执行 Once 守护代码，只有 1 个 goroutine 会获得执行机会，其他 goroutine 会阻塞直至代码执行完毕。

```
func main() {
  var once = sync.Once{}
  f := func() {
    time.Sleep(10 * time.Millisecond)
    fmt.Println("do once")
  }
  go func() {
    fmt.Println("do once start")
    once.Do(f)
    fmt.Println("do once finish")
  }()
  time.Sleep(1 * time.Millisecond)
  for i := 0; i < 2; i++ {
    go func() {
      fmt.Println("block...")
      once.Do(f)
      fmt.Println("resume")
    }()
  }
  time.Sleep(10 * time.Millisecond)
```

```
}//~
do once start
block...
block...
do once
do once finish
resume
resume
```

java中单例模式的写法有好几种，主要是懒汉式单例、饿汉式单例。

懒汉式单例: 懒汉式单例的实现没有考虑线程安全问题，需要结合synchronized，保证线程安全

```
//懒汉式单例类.在第一次调用的时候实例化自己
public class Singleton {
    private Singleton() {}
    private static Singleton single=null;
    //静态工厂方法
    public static synchronized Singleton getInstance() {
         if (single == null) {
             single = new Singleton();
         }
        return single;
    }
}
```

饿汉式单例：饿汉式在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以天生是线程安全的。

```
//饿汉式单例类.在类初始化时，已经自行实例化
public class Singleton {
    private Singleton() {}
    private static final Singleton single = new Singleton();
    //静态工厂方法
    public static Singleton getInstance() {
        return single;
    }
}
```

五、垃圾回收

GC(Garbage Collection)垃圾回收是一种自动管理内存的方式，支持GC的语言无需手动管理内存，程序后台自动判断对象是否存活并回收其内存空间，使开发人员从内存管理上解脱出来。 因为支持更多的特性和更灵活多样的GC策略, 比如分代,对象可移动,各种参数调节等等. 而Go只做了一种GC方案,不分代,不可移动,没什么参数能调节,而且更注重暂停时间的优化,执行GC的时机更频繁, 所以Go通常更占更少的内存,但代价就是GC性能比JVM差了不少。

1、Java的垃圾回收体系

Java基于JVM完成了垃圾收集的功能，其体系很庞大，包括了垃圾回收器（G1、CMS、Serial、ParNew等）、垃圾回收算法(标记-清除、标记-整理、复制、分代收集)、可达性算法(可达性分析、引用计数法)、引用类型、JVM内存模型等内容。经过多代发展，Java的垃圾回收机制较为完善，Java划分新生代、老年代来存储对象。对象通常会在新生代分配内存，多次存活的对象会被移到老年代，由于新生代存活率低，产生空间碎片的可能性高，通常选用“标记-复制”作为回收算法，而老年代存活率高，通常选用“标记-清除”或“标记-整理”作为回收算法，压缩整理空间。

2、Golang GC特征

三色标记、并发标记和清扫、非分代、非紧缩、写屏障

a、三色标记

a、程序开始时有黑白灰三个集合，初始时所有对象都是白色； b、从root对象开始标记，将所有可达对象标记为灰色； c、从灰色对象集合取出对象，将其引用对象标记为灰色，放入灰色集合，并将自己标记为黑色； d、重复第三步，直到灰色集合为空，即所有可达对象全部都被标记； e、标记结束后，不可达白色对象即为垃圾，对内存进行迭代清扫，回收白色对象； f、重置GC状态；

b、非分代

Java采用分代回收（按照对象生命周期长短划分不同的代空间，生命周期长的放入老年代，短的放入新生代，不同代有不同的回收算法和回收频率），Golang没有分代，一视同仁；

c、非紧缩

在垃圾回收之后不会进行内存整理以清除内存碎片；

d、写屏障

在并发标记的过程中，如果应用程序修改了对象图，就可能出现标记遗漏的可能，写屏障是为了处理标记遗漏的问题。

六、资源消耗对比

在内存利用效率上，Go语言确实比Java做得更好，在4个不同的角度来总结：

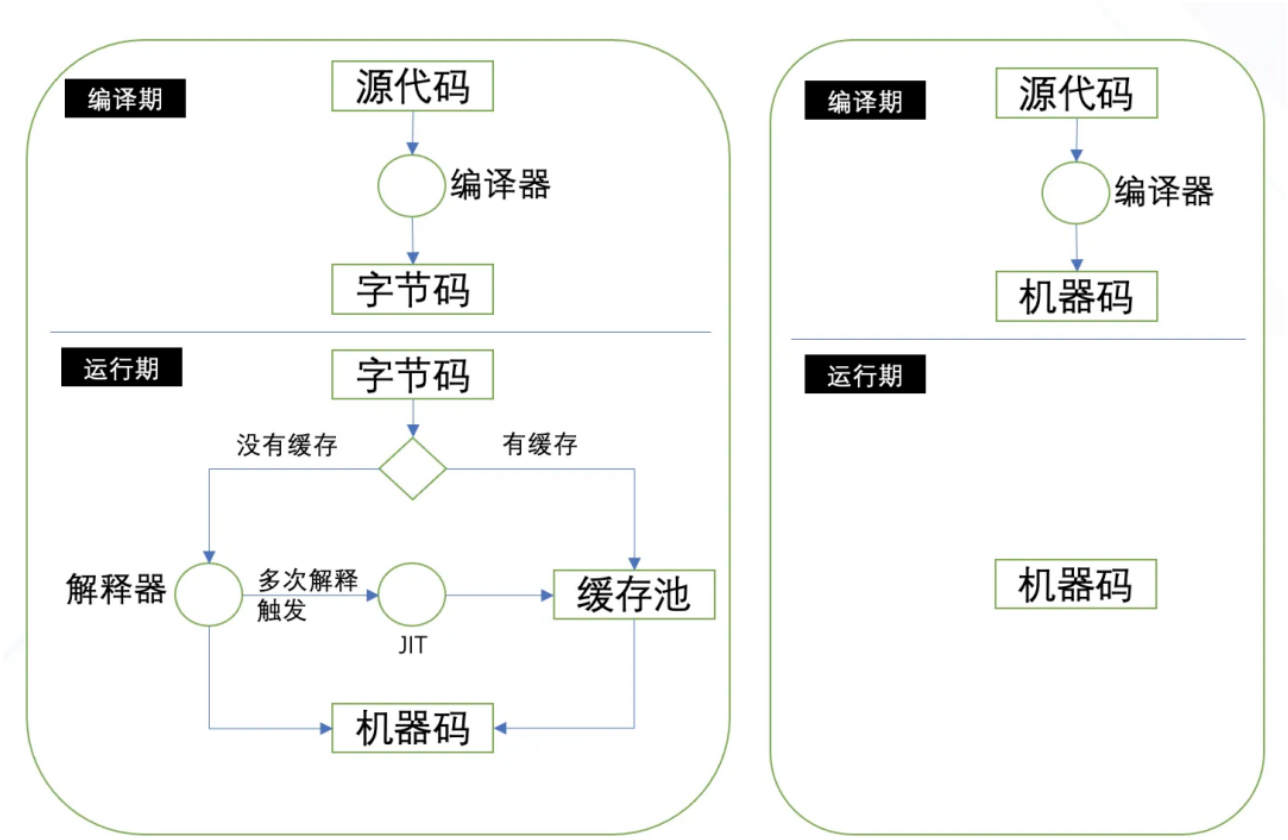
1、Java的JIT策略比Golang的AOT策略

Java在运行时相比Golang多占用了一些内存。原因在于：

Java运行态中包含了一个完整的解释器、一个JIT编译期以及一个垃圾回收器，这会显著地增加内存。

Golang语言直接编译到机器码，运行态只包含机器码和一个垃圾回收器。

因此Golang的运行态相对消耗内存较少。



2、内存分配和垃圾回收器

Java确实在起步占用上偏多，毕竟jvm需要更多内存做jit，默认的gc算法对内存要求偏高，但这不能代表后续占用仍然线性增长。如果目标是启动成百上千个内存需求较少的进程，那Java确实不擅长。

3、并发

协程模型比线程模型更加节省内存。

4、反射

Golang的反射更加简单，导致框架的内存消耗Golang程序比Java程序优秀。主要是因为：Java的框架实现中大量使用反射，并使用hashmap缓存信息，这2个都是极度消耗内存的行为。 Golang的框架中也使用reflect、map。但是Golang是面向interface和值类型的，这导致Golang的反射模型要比Java的反射模型简单非常多，反射过程要产生的对象数量也少非常多。

七、生态

Java 在生态这方面简直是无敌的存在，这主要得益于 Spring 全家桶，Spring 让 Java 走上了神座。Golang 语言知名的框架也很多，但是远远没有 Spring 影响那么大。

总结

	优点	缺点
Golang	代码简洁性 静态类型可编译成机器码直接运行 天生多核并行 垃圾收集 跨平台且不依赖运行时环境 简洁的泛型	有限的库支持 泛型不够完善 灵活度没Java高(这个可算优点也可算缺点)
java	优秀的生态 优秀的三方库 多线程 灵活性高 平台独立性 完善的语言特性 代码结构层次清晰	大量冗余的陈旧实现导致性能不佳 生态的复杂性 复杂的继承机制

腾讯程序员 2763

腾讯程序员 642

喜欢此内容的人还喜欢

MyBatis 批量插入别再乱用 foreach 了，5000 条数据花了 14 分钟。。。

macrozheng

Meta开源Rust编写的高性能构建系统

OSC开源社区

分布式锁用 Redis 还是 Zookeeper？哪家强？

Java后端技术