Please visit website: http://cxyroad.com

一文读懂 SOLID 原则

> 大家好，我是孔令飞，字节跳动云原生开发专家、前腾讯云原生技术专家、[云原生实战营](http://cxyroad.com/ "https://konglingfei.com") 知识星球星主、《企业级 Go 项目开发实战》作者。欢迎我的公众号【令飞编程】，Go、云原生、AI 领域技术干货不错过。

在 Go 项目开发中，你经常会听到软件开发要遵循 SOLID 原则。另外，在面试过程中，也经常有面试官问到 SOLID 原则。在我的职业生涯中，就遇到过 2 个面试官问我什么是 SOLID 原则。所以，作为开发者，掌握 SOLID 原则及开发方式是一项必备的技能。

那么 SOLID 原则是什么？如何遵循 SOLID 原则呢？本文详细为你解答这些疑问。

SOLID 原则介绍

SOLID 原则是由罗伯特·C·马丁在 21 世纪早期引入，指代了面向对象编程和面向对象设计的五个基本原则。遵循SOLID 原则可以确保我们设计的代码是易维护、易扩展、易阅读的。SOLID 原则同样也适用于 Go 程序设计。具体 SOLID 编码原则见下表：

简写	全称	中文描述
SRP	The Single Responsibility Principle	单一功能原则
OCP	The Open Closed Principle	开闭原则
LSP	The Liskov Substitution Principle	里氏替换原则
DIP	The Dependency Inversion Principle	依赖倒置原则
ISP	The Interface Segregation Principle	接口分离原则

Single Responsibility Principle：单一功能原则

单一功能原则：一个类或者模块只负责完成一个职责（或功能）。

简单来说就是保证我们在设计函数、方法时做到功能单一，权责明确，当发生改变时，只有一个改变它的原因。如果函数/方法承担的功能过多，就意味着很多功能会相互耦合，这样当其中一个功能发生改变时，可能会影响其它功能。单一功能原则，可以使代码后期的维护成本更低、改动风险更低。

例如，有以下代码，用来创建一个班级，班级包含老师和学生，代码如下：

```
package srp

type Class struct {
Teacher *Teacher
Student *Student
}

type Teacher struct {
Name  string
Class int
}

type Student struct {
Name  string
Class int
}

func createClass(teacherName, studentName string, class int) (*Teacher,
*Student) {
teacher := &Teacher{
Name:  teacherName,
Class: class,
}
student := &Student{
Name:  studentName,
Class: class,
}

return teacher, student
}

func CreateClass() *Class {
teacher, student := createClass(”colin”, ”lily”, 1)
return &Class{
```

```
Teacher: teacher,
Student: student,
}
}
```

上面的代码段通过 `createClass` 函数创建了一个老师和学生，老师和学生属于同一个班级。但是现在因为老师资源不够，要求一个老师管理多个班级。这时候，需要修改 `createClass` 函数的 `class` 参数，因为创建学生和老师是通过 `createClass` 函数的 `class` 参数偶合在一起，所以修改创建老师的代码，势必会影响创建学生的代码，其实，创建学生的代码我们是压根不想改动的。这时候 `createClass` 函数就不满足单一功能原则。需要修改为满足单一功能原则的代码，修改后代码段如下：

```
package srp

type Class struct {
Teacher *Teacher
Student *Student
}

type Teacher struct {
Name  string
Class int
}

type Student struct {
Name  string
Class int
}

func CreateStudent(name string, class int) *Student {
return &Student{
Name:  name,
Class: class,
}
}

func CreateTeacher(name string, classes []int) *Teacher {
return &Teacher{
Name:  name,
Class: classes,
}
}
```

```
func CreateClass() *Class {
teacher := CreateTeacher(”colin”, []int{1, 2})
student := CreateStudent(”lily”, 1)
return &Class{
Teacher: teacher,
Student: student,
}
}
```

上述代码，我们将 `createClass` 函数拆分成 2 个函数 `CreateStudent` 和 `CreateTeacher`，分别用来创建学生和老师，各司其职，代码互不影响。

Open / Closed Principle：开闭原则

开闭原则：软件实体应该对扩展开放、对修改关闭。

简单来说就是通过在已有代码基础上扩展代码，而非修改代码的方式来完成新功能的添加。开闭原则，并不是说完全杜绝修改，而是尽可能不修改或者以最小的代码修改代价来完成新功能的添加。

以下是一个满足开闭原则的代码段：

```
type IBook interface {
GetName() string
GetPrice() int
}

// NovelBook 小说
type NovelBook struct {
Name   string
Price  int
}

func (n *NovelBook) GetName() string {
return n.Name
}
```

```
func (n *NovelBook) GetPrice() int {
return n.Price
}
```

上述代码段，定义了一个 `Book` 接口和 `Book` 接口的一个实现：`NovelBook`（小说）。现在有新的需求，对所有小说打折统一打 5 折，根据开闭原则，打折相关的功能应该利用扩展实现，而不是在原有代码上修改，所以，新增一个 `OffNovelBook` 接口，继承 `NovelBook`，并重写 `GetPrice` 方法。

```
type OffNovelBook struct {
NovelBook
}

// 重写GetPrice方法
func (n *OffNovelBook) GetPrice() int {
return n.NovelBook.GetPrice() / 5
}
```

Liskov Substitution Principle：里氏替换原则

里氏替换原则：如果 S 是 T 的子类型，则类型 T 的对象可以替换为类型 S 的对象，而不会破坏程序。

简单来说，里氏替换原则要求子类（派生类）能够替换父类（基类）并且不影响程序的行为。也就是说，子类应该继承父类的所有属性和行为，并且可以在不改变程序逻辑的情况下进行扩展。在 Go 开发中，里氏替换原则可以通过接口来实现。

例如，以下是一个符合里氏替换原则的代码段：

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

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

type ReadWriter interface {
Reader
Writer
}

func Write(w Writer, p []byte) (int, error) {
return w.Write(p)
}
```

我们可以将 `Write` 函数中的 `Writer` 参数替换为其子类型 `ReadWriter`，而不影响已有程序：

```
func Write(rw ReadWriter, p []byte) (int, error) {
return rw.Write(p)
}
```

Dependency Inversion Principle：依赖倒置原则

依赖倒置原则：依赖于抽象而不是一个实例，其本质是要面向接口编程，不要面向实现编程。

以下是一个不符合依赖倒置原则的示例：

```
package main

import ”fmt”

// 定义一个高层模块
type HighLevelModule struct {
    lowLevelModule LowLevelModule
```

```
}

func (hlm HighLevelModule) DoSomething() {
    hlm.lowLevelModule.DoSomething()
}

// 定义一个低层模块
type LowLevelModule struct{}

func (llm LowLevelModule) DoSomething() {
    fmt.Println("Doing something in low level module...")
}

func main() {
    llm := LowLevelModule{}
    hlm := HighLevelModule{lowLevelModule: llm}
    hlm.DoSomething()
}
```

在上面的示例中，`HighLevelModule` 依赖于 `LowLevelModule`，而且在 `HighLevelModule` 中直接实例化了 `LowLevelModule`。这不符合依赖倒置原则的原因是高层模块应该依赖于抽象而不是具体的实现，而且高层模块不应该直接依赖于低层模块的具体实现。

为了符合依赖倒置原则，我们可以通过将 `LowLevelModule` 抽象成接口，并在 `HighLevelModule` 中依赖于该接口，从而实现依赖倒置。以下是优化后的示例：

```
package main

import "fmt"

// 定义一个低层模块接口
type LowLevelModule interface {
    DoSomething()
}

// 定义一个高层模块
type HighLevelModule struct {
    lowLevelModule LowLevelModule
}
```

```
func (hlm HighLevelModule) DoSomething() {
    hlm.lowLevelModule.DoSomething()
}

// 实现低层模块
type ConcreteLowLevelModule struct{}

func (cllm ConcreteLowLevelModule) DoSomething() {
    fmt.Println("Doing something in low level module...")
}

func main() {
    cllm := ConcreteLowLevelModule{}
    hlm := HighLevelModule{lowLevelModule: cllm}
    hlm.DoSomething()
}
```

在优化后的示例中，我们定义了 `LowLevelModule` 接口来抽象低层模块，并在 `HighLevelModule` 中依赖于该接口。同时，我们实现了 `ConcreteLowLevelModule` 结构体来实现 `LowLevelModule` 接口。这样就符合了依赖倒置原则，高层模块依赖于抽象接口，而不是具体的实现，降低了模块之间的耦合度。

Interface Segregation Principle：接口隔离原则

接口隔离原则：是指客户端不应该依赖它不需要的接口，即一个类对另一个类的依赖应该建立在最小的接口上。具体来说，接口隔离原则要求程序员尽量将臃肿庞大的接口拆分成更小的和更具体的接口，让接口中只包含客户感兴趣的方法。

以下是一个不符合接口隔离原则的示例：

```
package main

import "fmt"

// 定义一个接口
type Machine interface {
    Print()
```

```
    Scan()
}

// 实现接口
type MultiFunctionMachine struct{}

func (mfm MultiFunctionMachine) Print() {
    fmt.Println("Printing...")
}

func (mfm MultiFunctionMachine) Scan() {
    fmt.Println("Scanning...")
}

func main() {
    mfm := MultiFunctionMachine{}
    mfm.Print()
    mfm.Scan()
}
```

在上面的示例中，我们定义了一个 `Machine` 接口，包含 `Print()` 和 `Scan()` 两个方法。然后我们实现了一个 `MultiFunctionMachine` 结构体来实现这个接口。这个示例不符合接口隔离原则的原因是， `MultiFunctionMachine` 结构体实现了一个包含打印和扫描功能的接口，但是在实际使用中，可能某些设备只需要其中的一个功能，而不需要同时实现接口中的所有方法。

为了符合接口隔离原则，我们可以将 `Machine` 接口拆分为两个单一职责的接口，分别表示打印和扫描功能。然后根据需要实现对应的接口。以下是优化后的示例：

```
package main

import "fmt"

// 定义打印机接口
type Printer interface {
    Print()
}

// 定义扫描仪接口
type Scanner interface {
    Scan()
```

```
}

// 实现打印机
type SimplePrinter struct{}

func (sp SimplePrinter) Print() {
    fmt.Println("Printing...")
}

// 实现扫描仪
type SimpleScanner struct{}

func (ss SimpleScanner) Scan() {
    fmt.Println("Scanning...")
}

func main() {
    sp := SimplePrinter{}
    sp.Print()

    ss := SimpleScanner{}
    ss.Scan()
}
```

在优化后的示例中，我们将 `Machine` 接口拆分为 `Printer` 和 `Scanner` 两个单一职责的接口，分别表示打印和扫描功能。然后我们分别实现了 `SimplePrinter` 和 `SimpleScanner` 结构体来实现这两个接口，每个结构体只实现了对应的功能。这样就遵循了接口隔离原则，将接口按照单一职责进行拆分，避免了一个类需要实现不需要的方法。

---

* 您的支持是我写作的最大动力！如果这篇文章对您有帮助，感谢点赞和；
* 强烈推荐我写的一个优秀的 Go + 云原生项目开发脚手架：[OneX](http://cxyroad.com/ "https://konglingfei.com/onex/")。可以让你轻松开发 Go 项目，解放双手，提高开发效率！

原文链接: https://juejin.cn/post/7366154691032039458