什么是设计模式

用来解决普遍存在问题的方法论，就是为了**把代码中变化的部分和不变的部分给尽可能的剥离开**，使得代码的**可维护性**大大增强。设计模式是为了让程序具有更好的**代码重用性**、**可读性**、**可扩展性**、**可靠性**、使程序呈现**高内聚，低耦合**的特性

设计模式的六大原则（理解异地开单）

里氏替换原则

子类型必须能够替换掉它们的基类型

继承应当保证基类在程序中可以被任意派生类替换使用，而程序的行为不会发生变化

要求派生类在逻辑上必须是基类的一个“真子集”

接口隔离原则

不应该强迫客户依赖于它们不使用的方法（每个接口中不存在子类用不到却必须实现的方法）

要求将庞大臃肿的接口拆分成更小的和更具体的接口，这样客户就会依赖于它们实际调用的方法

有助于降低系统的耦合度，提高系统的灵活性和可维护性

依赖倒转原则

细节应该依赖抽象

抽象不应该依赖细节

高层次的模块不应该依赖低层次的模块，它们都应该依赖其抽象

迪米特法则（最少知识原则）

一个软件实体应当尽可能少地与其他实体发生相互作用

每一个软件单位对另一个单位的了解应该尽可能少，只限于那些与该单位紧密相关的软件单位

旨在减少软件实体之间的通信，降低系统的复杂度，提高系统的可维护性

开闭原则（总原则）

软件实体（类、模块、函数等）应该对扩展开放，对修改关闭

软件实体应该可以通过扩展来应对需求的变化，而不是通过修改现有的代码

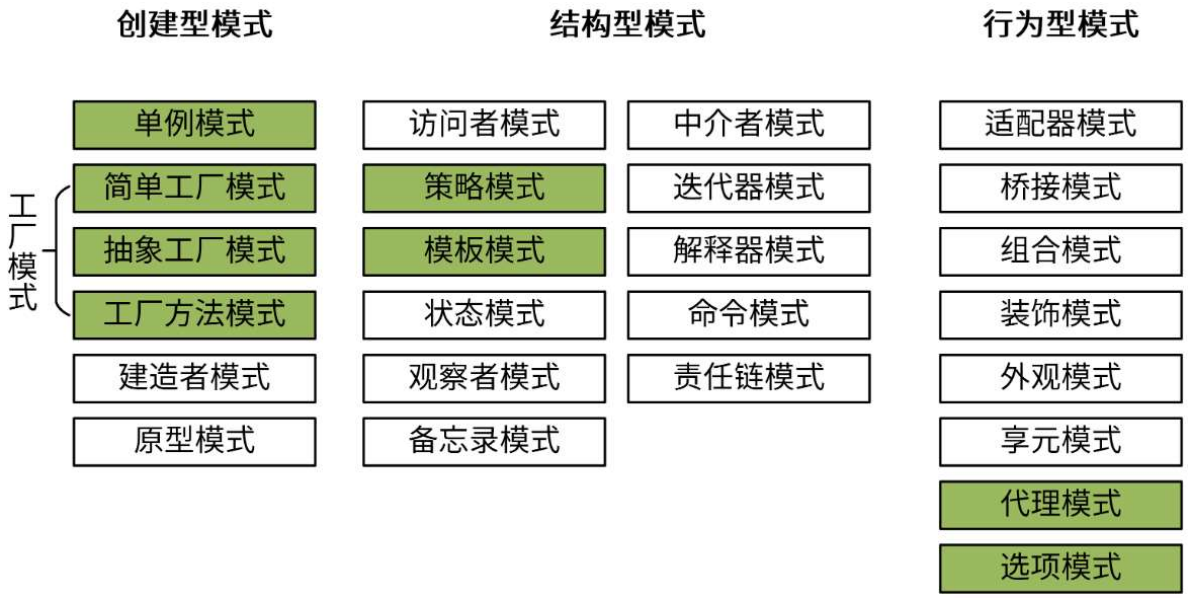
旨在增强软件的灵活性和可维护性

单一职责原则

一个类应该仅有一个引起它变化的原因。简单来说，一个类应该负责一项职责

如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。因此，应当尽量将类的职责单一化

设计模式的分类



常见设计模式

创建型模式

创建型模式（Creational Patterns），它提供了一种**在创建对象的同时隐藏创建逻辑**的方式，而不是使用new运算符直接实例化对象

单例模式

单例模式（Singleton Pattern），是**最简单的一个模式。**在Go中，单例模式指的是全局只有一个实例，并且它负责创建自己的对象。单例模式不仅有利于减少内存开支，还有减少系统性能开销、防止多个实例产生冲突等优点

因为单例模式保证了实例的全局唯一性，而且只被初始化一次，所以比较适合**全局共享一个实例，且只需要被初始化一次的场景**，例如数据库实例、全局配置、全局任务池等

单例模式又分为**饿汉方式**和**懒汉方式。**饿汉方式指全局的单例实例在包被加载时创建，而懒汉方式指全局的单例实例在第一次被使用时创建。你可以看到，这种命名方式非常形象地体现了它们不同的特点

饿汉方式

```
package singleton

type singleton struct {
}

var ins *singleton = &singleton{}

func GetInsOr() *singleton {
    return ins
}
```

实例是在包被导入时初始化的，所以如果初始化耗时，会导致程序加载时间比较长

懒汉方式

```
package singleton

type singleton struct {
}

var ins *singleton

func GetInsOr() *singleton {
    if ins == nil {
        ins = &singleton{}
    }

    return ins
}
```

懒汉方式是开源项目中使用最多的，但它的缺点是非并发安全，在实际使用时需要加锁

当多个线程几乎同时检查到`ins == nil`时，它们都会尝试进入创建实例的代码块。如果没有锁来确保只有一个线程能进入这个代码块，那么每个线程都可能创建一个新的实例

懒汉方式+锁

```
import "sync"

type singleton struct {
}

var ins *singleton
var mu sync.Mutex

func GetIns() *singleton {
    if ins == nil {
        mu.Lock()
        if ins == nil {
            ins = &singleton{}
        }
        mu.Unlock()
    }
    return ins
}
```

只有在创建时才会加锁，既提高了代码效率，又保证了并发安全

更优雅的实现方式：基于`sync.Once`的单例模式

```
package singleton

import (
    "sync"
)

type singleton struct {
```

```
}

var ins *singleton
var once sync.Once

func GetInsOr() *singleton {
    once.Do(func() {
        ins = &singleton{}
    })
    return ins
}
```

使用once.Do可以确保ins实例全局只被创建一次，once.Do函数还可以确保当同时有多个创建动作时，只有一个创建动作在被执行

工厂模式

工厂模式（Factory Pattern）是面向对象编程中的常用模式。在Go项目开发中，你可以通过使用多种不同的工厂模式，来使代码更简洁明了。Go中的结构体，可以理解为面向对象编程中的类，例如 Person 结构体（类）实现了Greet方法

简单工厂模式

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

func (p Person) Greet() {
  fmt.Printf("Hi! My name is %s", p.Name)
}

func NewPerson(name string, age int) *Person {
  return &Person{
    Name: name,
    Age: age,
  }
}
```

简单工厂模式、抽象工厂模式、工厂方法模式这三种工厂模式中，简单工厂模式是最常用、最简单的。它就是一个接受一些参数，然后返回Person实例的函数

和`p := &Person {}`这种创建实例的方式相比，简单工厂模式可以确保我们创建的实例具有需要的参数，进而保证实例的方法可以按预期执行

例如，通过`NewPerson`创建Person实例时，可以确保实例的name和age属性被设置

抽象工厂模式

```
// 定义了一个不可导出的结构体`person`，在通过NewPerson创建实例的时候返回的是接口，而不是结构体
type Person interface {
  Greet()
}
```

```
type person struct {
  name string
  age int
}

func (p person) Greet() {
  fmt.Printf("Hi! My name is %s", p.name)
}

// Here, NewPerson returns an interface, and not the person struct itself
func NewPerson(name string, age int) Person {
  return person{
    name: name,
    age: age,
  }
}
```

抽象工厂模式，和简单工厂模式的唯一区别，就是它返回的是接口而不是结构体

通过返回接口，可以**在你不公开内部实现的情况下，让调用者使用你提供的各种功能**

```
// 定义了一个名为Doer的接口，它有一个Do方法，该方法接受一个http.Request对象并返回一个http.Response对象和一个错误值。这个接口模拟了http.Client的Do方法
type Doer interface {
    Do(req http.Request) (http.Response, error)
}

// NewHTTPClient函数返回一个指向http.Client结构体的指针，这个结构体是Go标准库net/http包中的一个HTTP客户端
func NewHTTPClient() Doer {
    return &http.Client{}
}

// 定义了一个名为mockHTTPClient的结构体，它将用于模拟HTTP客户端
type mockHTTPClient struct{}

// mockHTTPClient的Do方法实现，它创建一个新的httptest.ResponseRecorder对象，这个对象用于记录HTTP响应。然后调用Result方法来生成一个http.Response对象，并返回这个响应和nil错误值
func (mockHTTPClient) Do(req http.Request) (http.Response, error) {
    res := httptest.NewRecorder()
    return res.Result(), nil
}

// NewMockHTTPClient函数返回一个mockHTTPClient的实例，这个实例可以用来模拟HTTP请求
func NewMockHTTPClient() Doer {
    return &mockHTTPClient{}
}
```

NewHTTPClient和NewMockHTTPClient都返回了同一个接口类型Doer，这使得二者可以互换使用。当你想测试一段调用了Doer接口Do方法的代码时，这一点特别有用。因为你可以使用一个Mock的HTTP客户端，从而避免了调用真实外部接口可能带来的失败

```
// QueryUser函数接受一个Doer接口类型的参数，创建一个HTTP GET请求，并使用传入的Doer来执行这个请求。如果请求失败，它会返回错误
```

```
func QueryUser(doer Doer) error {
    req, err := http.NewRequest("GET",
"http://iam.api.marmotedu.com:8080/v1/secrets", nil)
    if err != nil {
        return err
    }

    _, err := doer.Do(req)
    if err != nil {
        return err
    }

    return nil
}
// TestQueryUser是一个测试函数，它使用testing包来测试QueryUser函数。它创建了一个模拟的HTTP客户端NewMockHTTPClient，并调用QueryUser函数。如果QueryUser返回错误，测试将失败
func TestQueryUser(t *testing.T) {
    doer := NewMockHTTPClient()
    if err := QueryUser(doer); err != nil {
        t.Errorf("QueryUser failed, err: %v", err)
    }
}
```

- **函数名前面加上“mock”通常表示这个函数是一个模拟（mock）对象或模拟（mock）函数。模拟是一种在软件测试中广泛使用的技术，特别是在单元测试和集成测试中。它的目的是创建一个可控的替代物（即mock对象或mock函数），以便在测试过程中替代掉真实对象或函数的行为**

在使用简单工厂模式和抽象工厂模式返回实例对象时，都可以返回指针。在实际开发中，建议返回非指针的实例，因为主要是想通过创建实例，调用其提供的方法，而不是对实例做更改。如果需要对实例做更改，可以实现`SetXXX`的方法。通过返回非指针的实例，可以确保实例的属性，避免属性被意外/任意修改

- **返回Person：由于返回的是结构体的副本，因此接收者无法修改原始结构体的内容。任何对返回值的修改都不会影响到原始结构体。**
- **返回*Person：通过返回指针，接收者可以访问和修改原始结构体的内容。这允许函数通过返回值来影响外部状态**

工厂方法模式

```
type Person struct {
    name string
    age int
}

func NewPersonFactory(age int) func(name string) Person {
    return func(name string) Person {
        return Person{
            name: name,
            age: age,
        }
    }
}
```

```
newBaby := NewPersonFactory(1)
baby := newBaby("john")

newTeenager := NewPersonFactory(16)
teen := newTeenager("jill")
```

在**简单工厂模式**中，依赖于唯一的工厂对象，如果我们需要实例化一个产品，就要向工厂中传入一个参数，获取对应的对象；如果要增加一种产品，就要在工厂中修改创建产品的函数。这会导致耦合性过高，这时我们就可以使用**工厂方法模式**

在**工厂方法模式**中，依赖工厂函数，我们可以通过实现工厂函数来创建多种工厂，将对象创建从由一个对象负责所有具体类的实例化，变成由一群子类来负责对具体类的实例化，从而将过程解耦

结构型模式

结构型模式（Structural Patterns），它的特点是**关注类和对象的组合**

策略模式

```
package strategy

// 策略模式

// 定义一个策略类
type IStrategy interface {
    do(int, int) int
}

// 策略实现：加
type add struct{}

func (*add) do(a, b int) int {
    return a + b
}

// 策略实现：减
type reduce struct{}

func (*reduce) do(a, b int) int {
    return a - b
}

// 具体策略的执行者
type Operator struct {
    strategy IStrategy
}

// 设置策略
func (operator *Operator) setStrategy(strategy IStrategy) {
    operator.strategy = strategy
}

// 调用策略中的方法
func (operator *Operator) calculate(a, b int) int {
```

```
    return operator.strategy.do(a, b)
}
```

在上述代码中，定义了策略接口 IStrategy，还定义了 add 和 reduce 两种策略。最后定义了一个策略执行者，可以设置不同的策略，并执行，例如：

```
func TestStrategy(t *testing.T) {
    operator := Operator{}

    operator.setStrategy(&add{})
    result := operator.calculate(1, 2)
    fmt.Println("add:", result)

    operator.setStrategy(&reduce{})
    result = operator.calculate(2, 1)
    fmt.Println("reduce:", result)
}
```

可以随意更换策略，而不影响Operator的所有实现

模板模式

模板模式就是将一个类中能够公共使用的方法放置在抽象类中实现，将不能公共使用的方法作为抽象方法，强制子类去实现，这样就做到了将一个类作为一个模板，让开发者去填充需要填充的地方

```
package template

import "fmt"

type Cooker interface {
    fire()
    cooke()
    outfire()
}

// 类似于一个抽象类
type CookMenu struct {
}

func (CookMenu) fire() {
    fmt.Println("开火")
}

// 做菜，交给具体的子类实现
func (CookMenu) cooke() {
}

func (CookMenu) outfire() {
    fmt.Println("关火")
}

// 封装具体步骤
func doCook(cook Cooker) {
    cook.fire()
```

```
    cook.cooke()
    cook.outfire()
}

type XiHongShi struct {
    CookMenu
}

func (*XiHongShi) cooke() {
    fmt.Println("做西红柿")
}

type ChaoJiDan struct {
    CookMenu
}

func (ChaoJiDan) cooke() {
    fmt.Println("做炒鸡蛋")
}
```

```
func TestTemplate(t *testing.T) {
    // 做西红柿
    xihongshi := &XiHongShi{}
    doCook(xihongshi)

    fmt.Println("\n=====> 做另外一道菜")
    // 做炒鸡蛋
    chaojidan := &ChaoJiDan{}
    doCook(chaojidan)

}
```

行为型模式

代理模式

代理模式 (Proxy Pattern)，可以为另一个对象提供一个替身或者占位符，以控制对这个对象的访问

```
package proxy

import "fmt"

type Seller interface {
    sell(name string)
}

// 火车站
type Station struct {
    stock int //库存
}

func (station *Station) sell(name string) {
    if station.stock > 0 {
        station.stock--
```

```
        fmt.Printf("代理点中：%s买了一张票,剩余：%d \n", name, station.stock)
    } else {
        fmt.Println("票已售空")
    }

}

// 火车代理点
type StationProxy struct {
    station *Station // 持有一个火车站对象
}

func (proxy *StationProxy) sell(name string) {
    if proxy.station.stock > 0 {
        proxy.station.stock--
        fmt.Printf("代理点中：%s买了一张票,剩余：%d \n", name, proxy.station.stock)
    } else {
        fmt.Println("票已售空")
    }
}
```

上述代码中，StationProxy代理了Station，代理类中持有被代理类对象，并且和被代理类对象实现了同一接口

选项模式

Go不支持给参数设置默认值，使用选项模式可以实现：既能够创建带默认值的实例，又能够创建自定义参数的实例，创建一个带有默认值的struct变量，并选择性地修改其中一些参数的值

```
package options

import (
    "time"
)

type Connection struct {
    addr    string
    cache   bool
    timeout time.Duration
}

const (
    defaultTimeout = 10
    defaultCaching = false
)

type options struct {
    timeout time.Duration
    caching bool
}

// Option overrides behavior of Connect.
type Option interface {
    apply(*options)
}
```

```
type optionFunc func(*options)

func (f optionFunc) apply(o *options) {
    f(o)
}

func WithTimeout(t time.Duration) Option {
    return optionFunc(func(o *options) {
        o.timeout = t
    })
}

func WithCaching(cache bool) Option {
    return optionFunc(func(o *options) {
        o.caching = cache
    })
}

// Connect creates a connection.
func NewConnect(addr string, opts ...Option) (*Connection, error) {
    options := options{
        timeout: defaultTimeout,
        caching: defaultCaching,
    }

    for _, o := range opts {
        o.apply(&options)
    }

    return &Connection{
        addr:    addr,
        cache:   options.caching,
        timeout: options.timeout,
    }, nil
}
```

在上面的代码中，首先我们定义了`options`结构体，它携带了timeout、caching两个属性。接下来，我们通过`NewConnect`创建了一个连接，`NewConnect`函数中先创建了一个带有默认值的`options`结构体变量，并通过调用

```
for _, o := range opts {
    o.apply(&options)
}
```

来修改所创建的`options`结构体变量

需要修改的属性，是在`NewConnect`时，通过Option类型的选项参数传递进来的

总结

分类	设计模式	常见应用场景
创建型模式	单例模式	全局共享一个实例，且只需要被初始化一次的场景
	工厂模式	• 简单工厂模式可以传入参数并返回一个结构体的实例； • 抽象工厂模式可以返回一个接口，通过返回接口，在不公开内部实现的情况下，让调用者使用你提供的各种功能。 • 工厂方法模式将对象创建从由一个对象负责所有具体类的实例化，变成由一群子类来负责对具体类的实例化，从而将过程解耦
结构型模式	策略模式	需要采用不同策略的场景
	模版模式	需要在不改变算法框架的情况下，改变算法执行效果的场景
行为型模式	代理模式	需要一个替身或者占位符，以控制对这个对象的访问的场景
	选项模式	• 结构体参数很多，期望创建一个携带默认值的结构体变量，并选择性修改其中一些参数的值； • 结构体参数经常变动，变动时又不想修改创建实例的函数

创建型模式

单例模式

确保一个类只有一个实例，而且自行实例化并向整个系统提供这个实例

饿汉方式

类加载的时候就进行实例化

懒汉方式

类加载的时候不进行实例化，在第一次使用的时候再进行实例化

工厂模式

简单工厂模式

在简单工厂模式中，可以根据参数的不同返回不同类的实例

简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类

抽象工厂模式

和工厂模式的区别在于，工厂模式只生产一大类对象，而抽象工厂可以生产好几大类对象，就是所谓1和n的区别

工厂方法模式

定义一个用于创建对象的接口，让子类决定实例化哪个类

工厂方法使一个类的实例化延迟到其子类

结构型模式

策略模式

定义一组算法，将每个算法都封装起来，并且使它们之间可以互换

策略模式让算法独立于使用它的客户而变化，也称为政策模式

模板模式

定义一个操作中的算法的框架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤

行为型模式

代理模式

为其他对象提供一种代理以控制对这个对象的访问

选项模式

既能够创建带默认值的实例，又能够创建自定义参数的实例