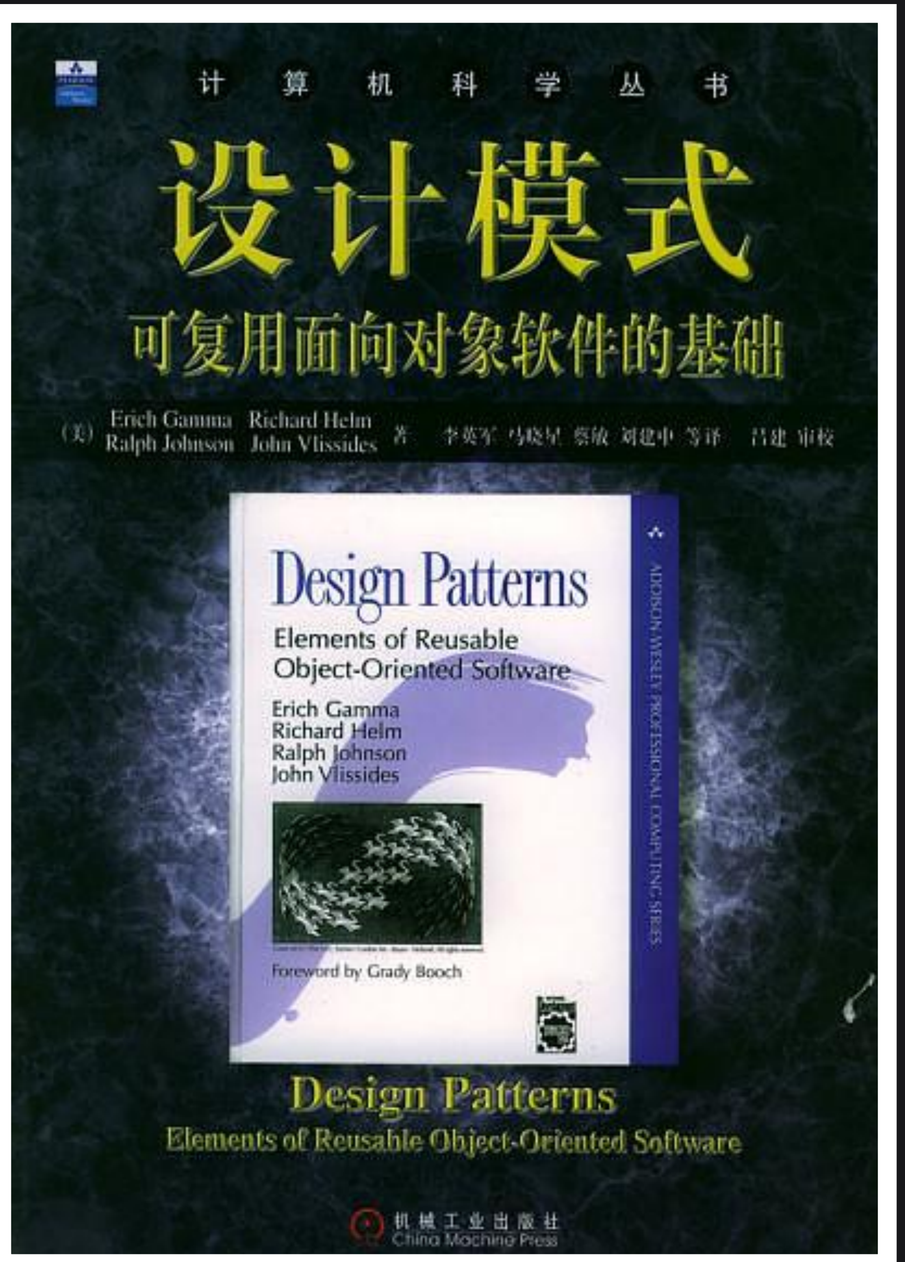
动态语言一时爽，代码重构火葬场?

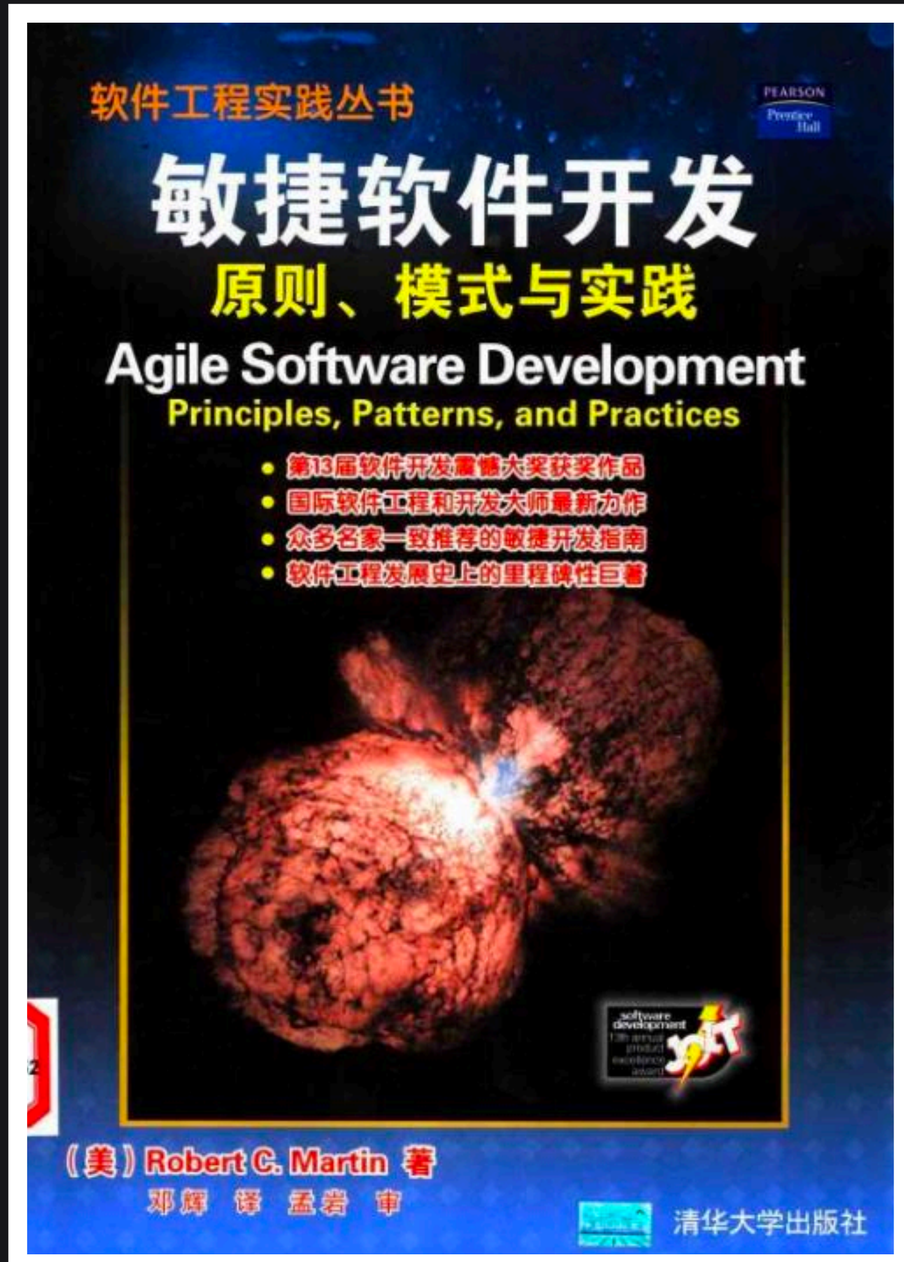
聊聊编程原则

姚钢强 · 10/14/2018

姚钢强

- 2013 年加入知乎，14 - 15 年负责社区管理的技术研发，期间开发了举报，审核等服务
- 16 - 17 年担任首页 feed 流技术负责人，期间通过构架优化使响应时间 P95 从 1.6s 降低到 700ms，使稳定性由 99.9% 提升到 99.99%，计算资源节省 50%
- 2018 年开始负责社区架构组，使用 Golang 重写了部分主要的业务模块，节省机器资源 75%

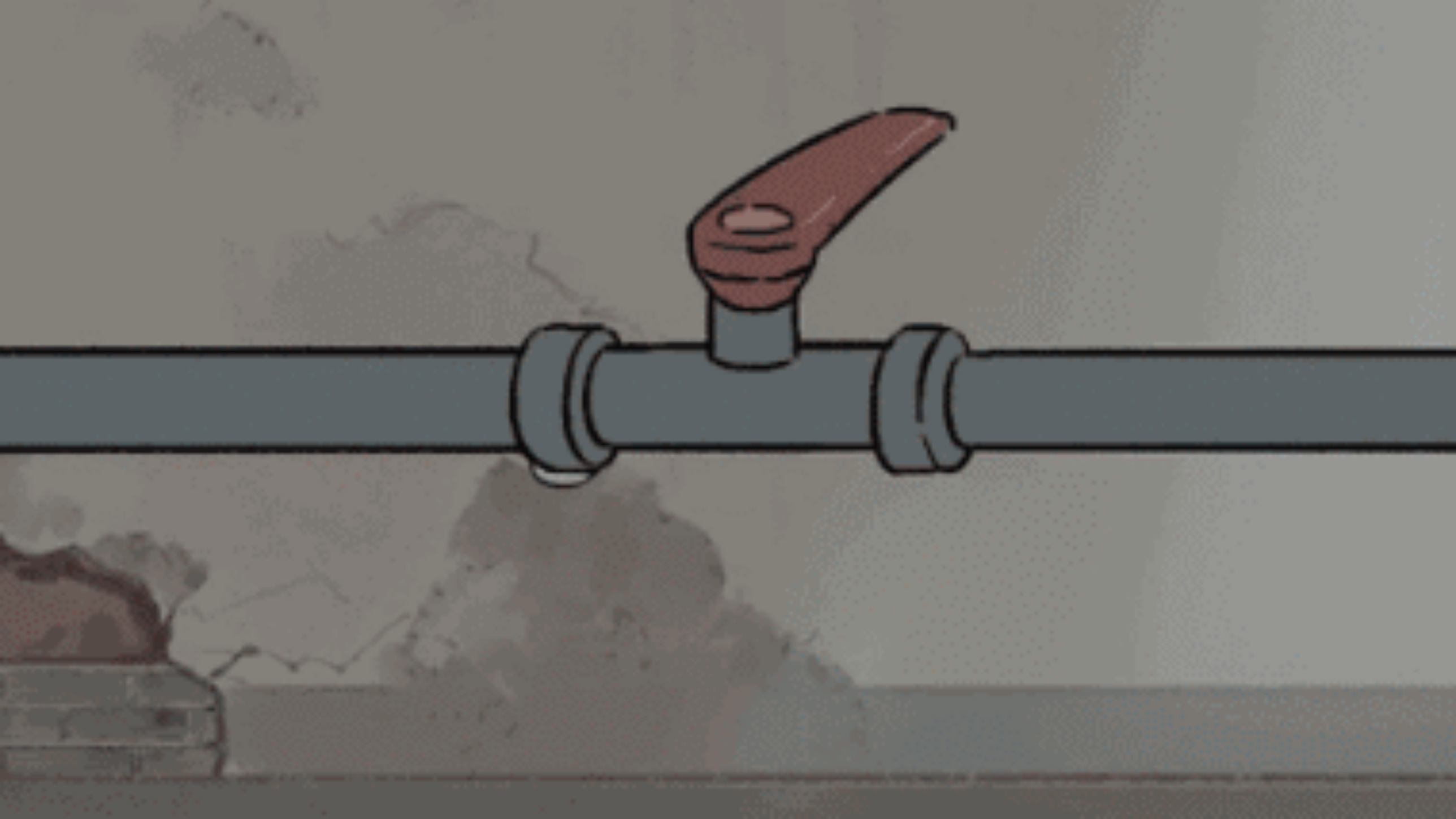

计 算 机 科 学 丛 书
设计模式
可复用面向对象软件的基础
(美) Erich Gamma Richard Helm Ralph Johnson John Vlissides 著 李英军 马晓星 蔡敏 刘建中 等译 吕建 审校
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
Design Patterns
Elements of Reusable Object-Oriented Software
机械工业出版社
China Machine Press


软件工程实践丛书
PEARSON
Prentice Hall
敏捷软件开发
原则、模式与实践
Agile Software Development
Principles, Patterns, and Practices
第13届软件开发震撼大奖获奖作品
国际软件工程和开发大师最新力作
众多名家一致推荐的敏捷开发指南
软件工程发展史上的里程碑性巨著
(美) Robert C. Martin 著
邓辉 译 孟岩 审
清华大学出版社

Don’t Believe Me!

动画时间


FOOD GROUPS



如何让项目更容易维护呢?

Code Review?

好代码是什么样子?

- 可读性高
- 逻辑清晰
- 高内聚
- 低耦合
- 易测试
- …

说的真好，如何做到呢?

从基础概念入手

面向对象特征

SOLID Principles

- 封装
- 继承
- 多态

面向对象特征

SOLID Principles

- 封装
- 继承
- 多态

Encapsulation

Encapsulation refers to the bundling of data with the methods that operate on that data.

Encapsulation is *used to hide the values or state of a structured data object inside a class*, preventing unauthorized parties' direct access to them.

——From Wikipedia

```
class Person(object):

    def __init__(self, birth_day, sex, children, lover):
        self.birth_day = birth_day
        self.sex = sex
        self.children = children
        self.lover = lover
        self.age = self.compute_age()

    def compute_age(self):
        today = date.today()
        if (today.month, today.day) > (self.birth_day.month,
self.birth_day.day):
            return today.year - self.birth_day.year
        else:
            return today.year - self.birth_day.year - 1
```

```
person = Person(date(1990, 1, 1), u"male", [], u"新垣结衣")

person.birth_day = date(2020, 1, 1)
person.children = [1]
person.sex = u"female"
person.lover = u"刘亦菲"
person.age = 18
```

一个人的生日能随便乱改?

生日确定了，年龄能随便改?

性别貌似也不行?

孩子更不行?

除了女朋友都不行

```python
class Person(object):

    def __init__(self, birth_day, sex, children, lover):
        self._birth_day = birth_day
        self._sex = sex
        self._children = children
        self.lover = lover

    @property
    def birth_day(self):
        return self._birth_day

    @property
    def age(self):
        today = date.today()
        if (today.month, today.day) > (self._birth_day.month, self._birth_day.day):
            return today.year - self._birth_day.year
        else:
            return today.year - self._birth_day.year - 1

    @property
    def sex(self):
        return self._sex

    @property
    def children(self):
        return self._children
```

```python
class Person(object):

    def __init__(self, birth_day, sex, children, lover):
        self._birth_day = birth_day
        self._sex = sex
        self._children = children
        self.lover = lover

    @property
    def children(self):
        return self._children

    @children.setter
    def children(self, children):
        self._children = children
```

孩子只准增加

不准减少和替换

```python
class Person(object):

    def __init__(self, birth_day, sex, children, lover):
        self._birth_day = birth_day
        self._sex = sex
        self._children = children
        self.lover = lover

    @property
    def children(self):
        return self._children

    def add_child(self, child):
        self._children.append(child)
```

“What’s the simplest thing that could possibly work?” The idea is to focus on the goal.

Implementing setters and getters up front is a distraction from the goal.

In Python, we can simply use public attributes knowing we can change them to properties later, if the need arises.

——From 《Fluent Python》

封装

- 尽可能隐藏一个模块的实现细节（属性名称，属性是否可变，算法，数据结构，数据类型）
- 访问控制只是为了防止程序员的无意误用，不打算，也无法防止程序员的故意破坏

面向对象特征

SOLID Principles

- 封装
- 继承
- 多态

Inheritance

Inheritance is the mechanism of basing an object or class upon another object (prototypical inheritance) or class (class-based inheritance), retaining similar implementation.

Duck 继承于 Bird，Ostrich 继承于 Bird，Bird 继承于 Animal，Human 继承 Animal，is-a is-a-kind-of 的关系?

```
class Bird(object):

    name = "Bird"

    def fly(self):
        pass

    def run(self):
        print(self.name + " run")


class Ostrich(Bird):

    name = "Ostrich"

    def laid_eggs(self):
        pass
```

```python
class Stack(list):

    def push(self, item):
        self.append(item)

    # 直接使用 list 的 pop 方法
    # def pop(self, index: int = ...):
    #     pass
```

试试组合?

```
class Stack(object):

    def __init__(self):
        self._items = []

    def push(self, item):
        self._items.append(item)

    def pop(self):
        self._items.pop()
```

```
class DoppelDict(dict):

    def __setitem__(self, key, value):
        super().__setitem__(key, [value] * 2)


dd = DictString(one=1)
dd["two"] = "2"
print(dd)

{'one': 1, 'two': ['2', ‘2']}
```

继承

- 继承使用不当会破坏封装，造成信息泄露
- 先考虑组合，在考虑继承
- 继承是 behaves-like-a, is-substitutable-for 的关系，不是 is-a 或 is-a-kind-of 的关系

面向对象特征

SOLID Principles

- 封装
- 继承
- 多态

Polymorphism

Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types

```
class Development(object):
    __metaclass__ = abc.ABCMeta

    def run(self):
        self.rfc_review()
        self.write_code()
        self._test()
        self.release()

    @abc.abstractmethod
    def rfc_review(self):
        pass

    @abc.abstractmethod
    def write_code(self):
        pass

    @staticmethod
    def _test():
        print("test")

    @abc.abstractmethod
    def release(self):
        pass
```

```
class LiveDevelopment(Development):

    def rfc_review(self):
        print("MySQL, HBase")

    def write_code(self):
        print("Java")

    def release(self):
        print("build APP, upload")


LiveDevelopment().run()
```

多态

相同的实现代码适用不同的场合

不同的实现代码适用相同的场合

面向对象特征 → SOLID Principles

- Single responsibility principle
- Open–closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

面向对象特征 → SOLID Principles

- Single responsibility principle
- Open–closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Single responsibility principle

A module should have one, and only one, reason to change.

A module should be responsible to one, and only one, actor.

```python
class Question(object):

    def __init__(self, question_id, data_formatting):
        self._question_id = question_id
        self._data_formatting = data_formatting

    def get(self):
        # connect_MySQL
        # get_data_from_MySQL
        # connect_Redis
        # get_data_from_Redis
        return self._format_data()

    def _format_data(self):
        # do some thing
        pass
```

```python
class Question(object):

    def __init__(self, question_id, data_formatting):
        self._question_id = question_id
        self._data_formatting = data_formatting

    def get(self):
        self._get_data()
        return self._format_data()

    def _get_data(self):
        # connect_MySQL
        # get_data_from_MySQL
        # connect_Redis
        # get_data_from_Redis
        pass

    def _format_data(self):
        # do something
        pass
```

```python
class Question(object):

    def __init__(self, question_id, data_formatting):
        self._question_id = question_id
        self._data_formatter = DataFormatter(data_formatting)

    def get(self):
        self._get_data()
        return self._data_formatter.format()

    def _get_data(self):
        # connect_MySQL
        # get_data_from_MySQL
        # connect_Redis
        # get_data_from_Redis
        pass


class DataFormatter(object):

    def __init__(self, data_formatting):
        self._data_formatting = data_formatting

    def format(self):
        # step1
        # step2
        pass


Question(question_id=1, data_formatting="PDF").get()
```

如何理解 responsibility？

* 职责是从外部角度定义的
* 职责可能不是一件事，而是很多有相同目标的事情

* 后端微服务

* 客户端，前端组件化

* 架构上的层次的划分，知乎 NIGINX 双层，一层负责安全策略，一层负责流量转发

Single responsibility principle

每个软件模块都有且只有一个需要被改变的理由

面向对象特征 → SOLID Principles

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Dependency inversion principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

```python
class Question(object):

    def __init__(self, question_id, data_formatting):
        self._question_id = question_id
        self._data_formatter = DataFormatter(data_formatting)

    def get(self):
        self._get_data()
        return self._data_formatter.format()

    def _get_data(self):
        # connect_MySQL
        # get_data_from_MySQL
        # connect_Redis
        # get_data_from_Redis
        pass


class DataFormatter(object):

    def __init__(self, data_formatting):
        self._data_formatting = data_formatting

    def format(self):
        # step1
        # step2
        pass


Question(question_id=1, data_formatting="PDF").get()
```

DataFormatter 增加新的参数 color 怎么办?


Question
DataFormatter


Question
DataFormatter

```python
class Question(object):

    def __init__(self, question_id, data_formatter):
        self._question_id = question_id
        self._data_formatter = data_formatter

    def get(self):
        self._get_data()
        return self._data_formatter.format()

    def _get_data(self):
        # connect_MySQL
        # get_data_from_MySQL
        # connect_Redis
        # get_data_from_Redis
        pass


class DataFormatter(object):

    def __init__(self, data_formatting):
        self._data_formatting = data_formatting

    def format(self):
        pass

data_formatter = DataFormatter(data_formatting="PDF")
Question(question_id=1, data_formatter=data_formatter).get()
```

类的定义

```
Class Car{
    private Framework framework;

    Car(){
        this.framework=new Framework();
    }

    public void run(){...}
}
```

依赖

```
Class Framework{
    private Bottom bottom;

    Framework(){
        this.bottom=new Bottom();
    }

}
```

依赖

```
Class Bottom{
    private Tire tire;

    Bottom(){
        this.tire= new Tire();
    }

}
```

依赖

```
Class Tire{
    private int size;

    Tire(){
        this.size=30;
    }

}
```

初始化车

```
Car mycar = new Car();
```

运行车

```
mycar.run();
```

https://www.zhihu.com/question/23277575/answer/24259844

类的定义

```
Class Car{
    private Framework framework;

    Car(int size){
        this.framework=new Framework(size);
    }

    public void run(){...}
}
```

依赖

```
Class Framework{
    private Bottom bottom;

    Framework(int size){
        this.bottom=new Bottom(size);
    }

}
```

依赖

```
Class Bottom{
    private Tire tire;

    Bottom(int size){
        this.tire= new Tire(size);
    }

}
```

依赖

```
Class Tire{
    private int size;

    Tire(int size){
        this.size=size;
    }

}
```

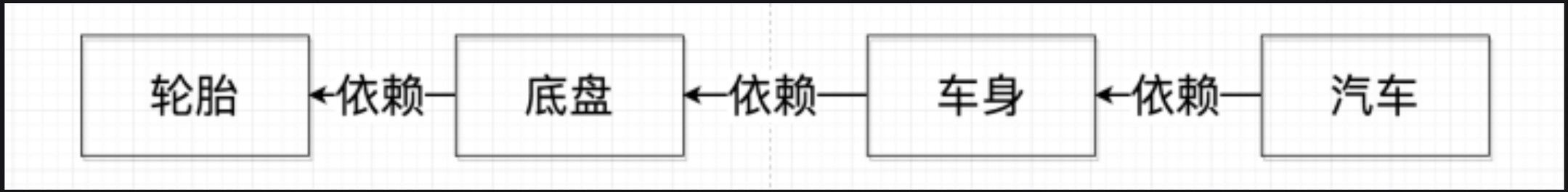
初始化车

```
int size = 40;
Car mycar = new Car(size);
```

运行车

```
mycar.run();
```

https://www.zhihu.com/question/23277575/answer/24259844



https://www.zhihu.com/question/23277575/answer/24259844

类的定义

```
Class Car{
    private Framework framework;

    Car(Framework framework){
        this.framework=framework;
    }

    public void run(){...}
}
```

注入

```
Class Framework{
    private Bottom bottom;

    Framework(Bottom bottom){
        this.bottom=bottom;
    }

}
```

注入

```
Class Bottom{
    private Tire tire;

    Bottom(Tire tire){
        this.tire=tire;
    }

}
```

注入

```
Class Tire{
    private int size;

    Tire(int size){
        this.size=size;
    }

}
```

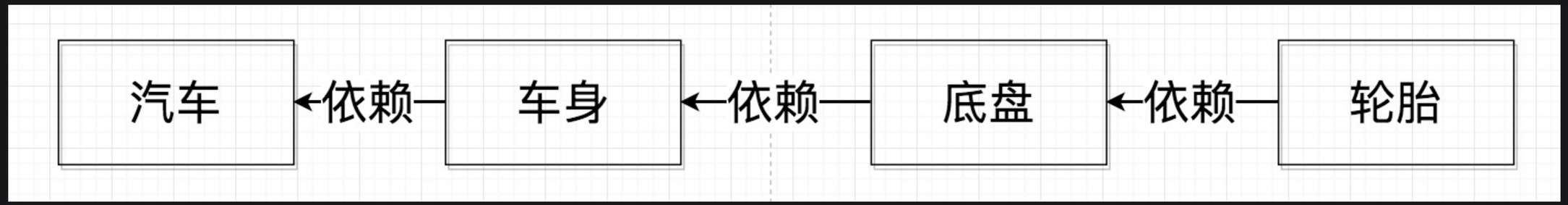
初始化车

```
int size = 40;
Tire tire = new Tire(size);
Bottom bottom = new Bottom(tire);
Framework framework = new Framework(bottom);
Car mycar = new Car(framework);
```

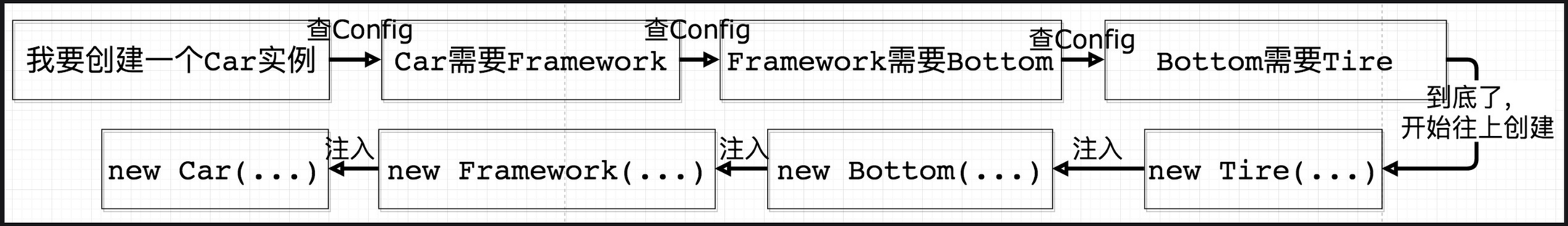
运行车

```
mycar.run();
```

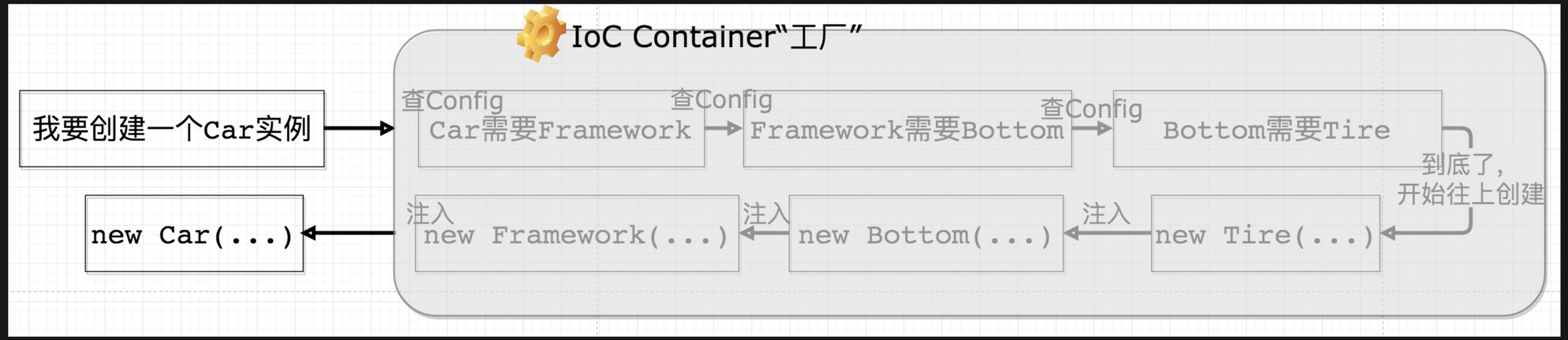
https://www.zhihu.com/question/23277575/answer/24259844



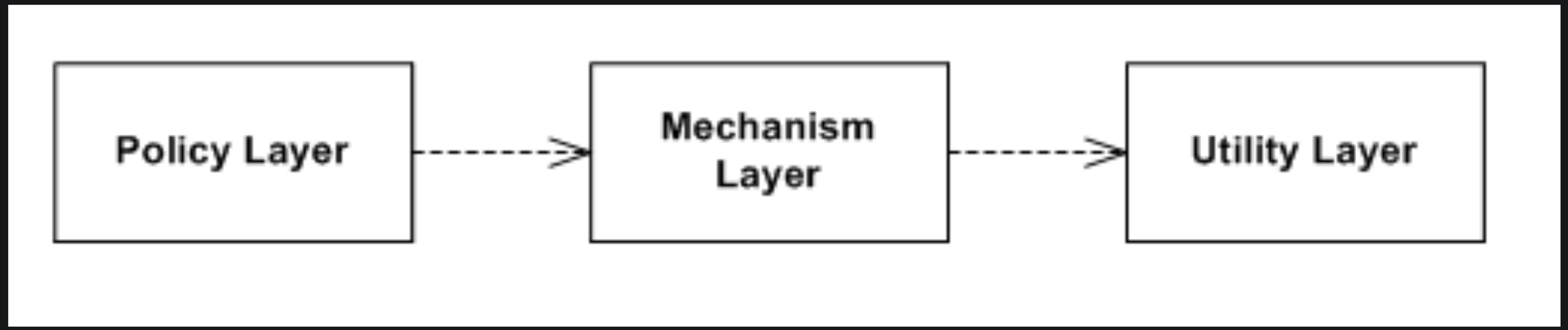
https://www.zhihu.com/question/23277575/answer/24259844



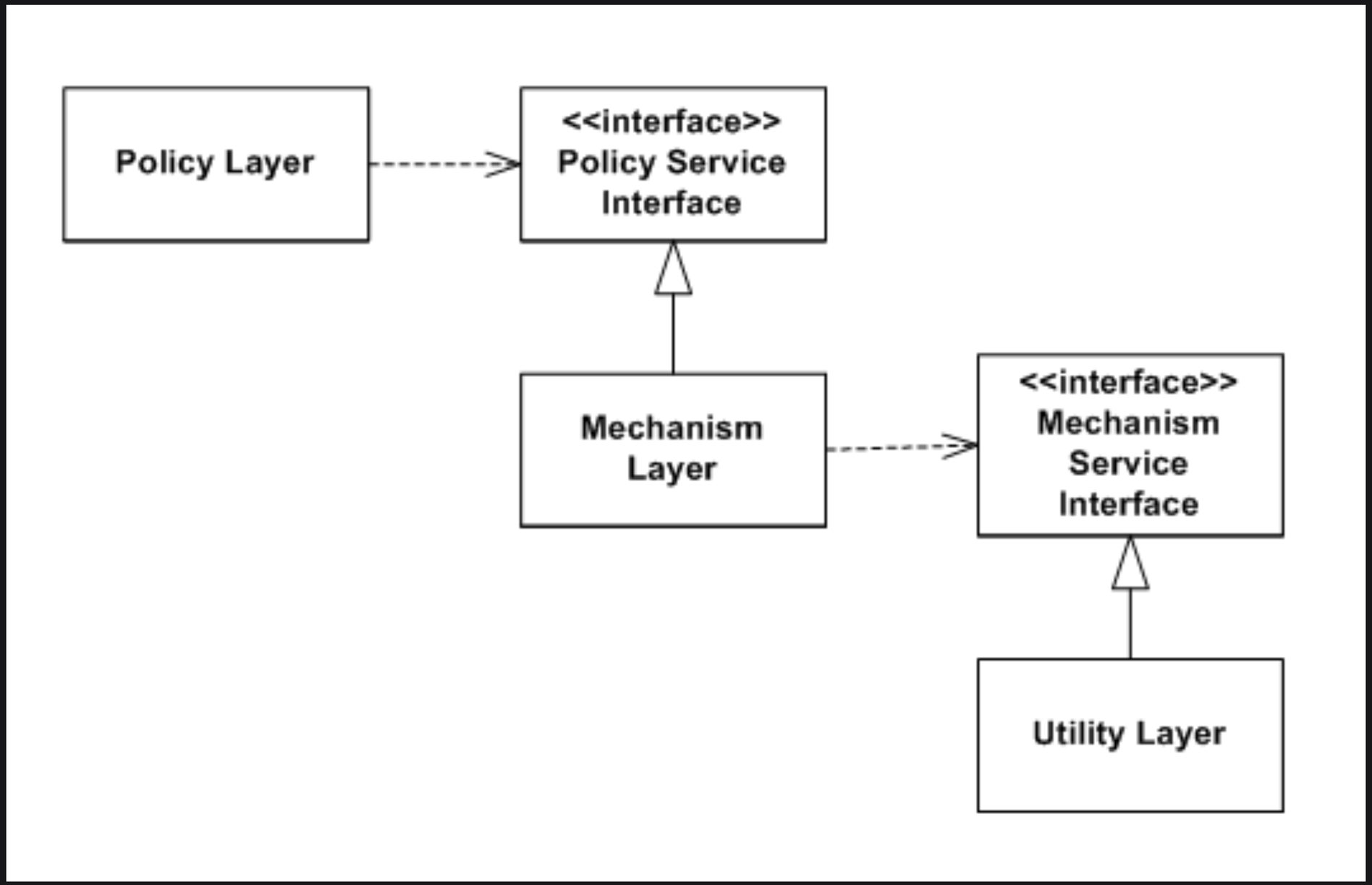
https://www.zhihu.com/question/23277575/answer/24259844



https://www.zhihu.com/question/23277575/answer/24259844



https://www.wikiwand.com/en/Dependency_inversion_principle



https://www.wikiwand.com/en/Dependency_inversion_principle

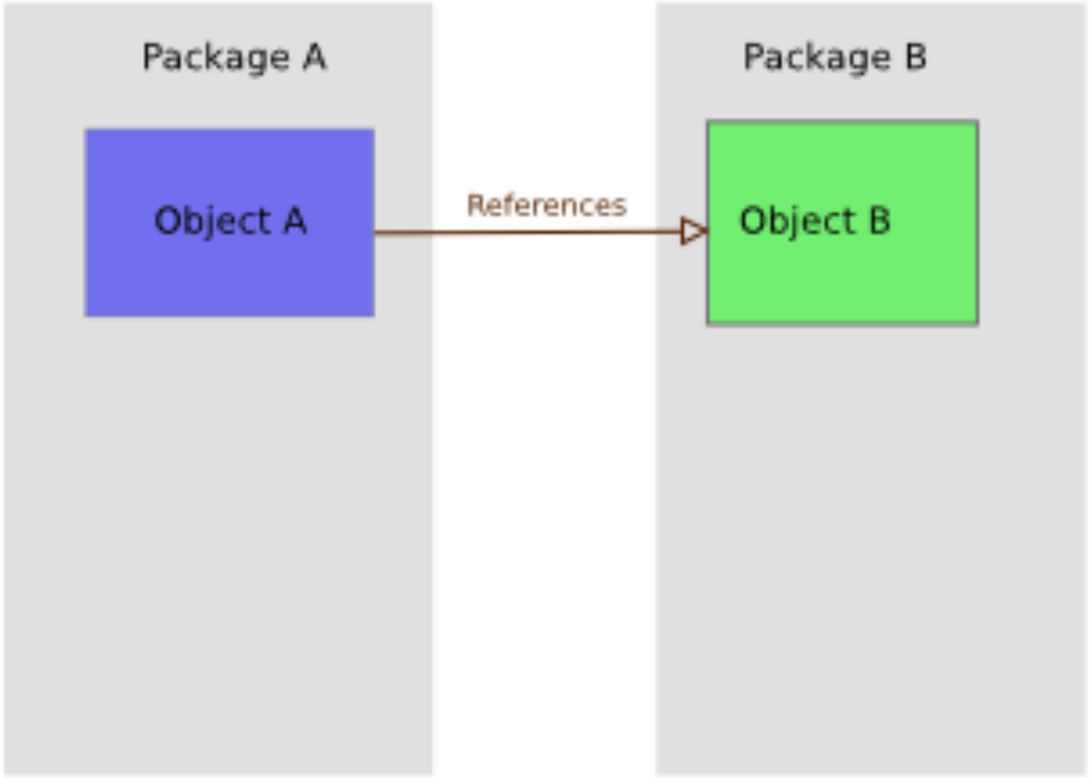


Figure 1

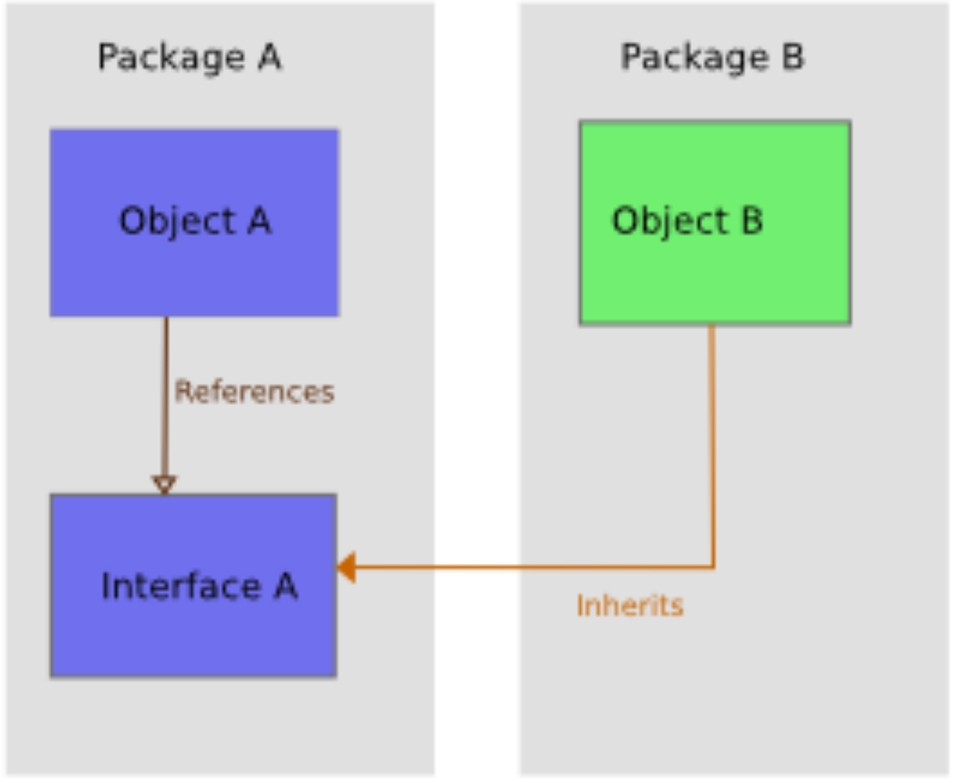
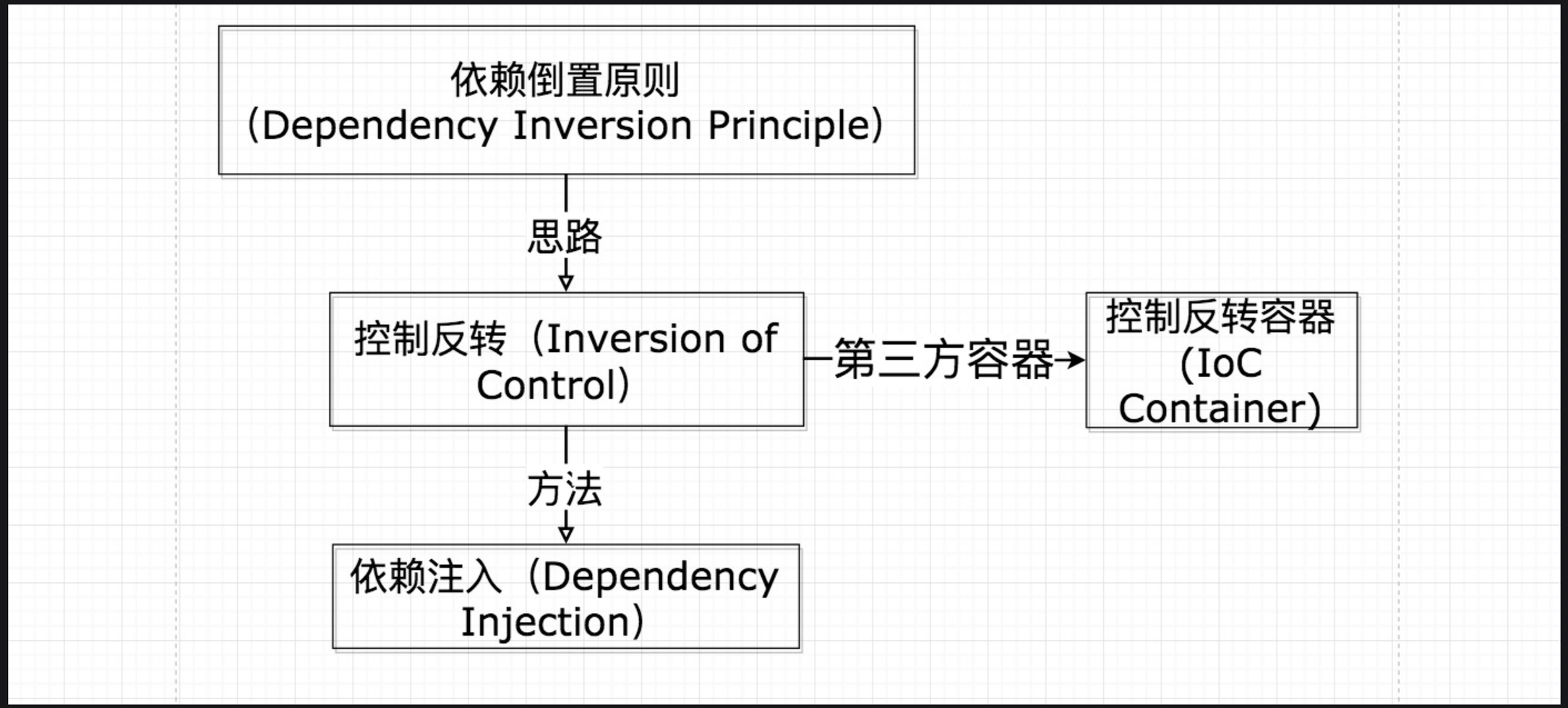


Figure 2

https://www.wikiwand.com/en/Dependency_inversion_principle



https://www.zhihu.com/question/23277575/answer/24259844

HTTP 协议

SQL

Dependency inversion principle

高层策略性代码不要依赖实现底层细节

底层细节的代码应该依赖高层策略性代码

面向对象特征 → SOLID Principles

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Liskov substitution principle

What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T

Liskov substitution principle

Subtypes must be substitutable for their base types.

```
class DataFormatter(object):

    def __init__(self, data_formatting):
        self._data_formatting = data_formatting

    def format(self):
        # do something
        pass


class ThriftFormatter(DataFormatter):
    def format(self):
        # do something
        pass


class JsonFormatter(DataFormatter):
    def format(self):
        if self._data_formatting != "txt":
            # do something
            pass
        else:
            raise Exception("")


data_formatter = JsonFormatter(data_formatting="PDF")
Question(question_id=1, data_formatter=data_formatter).get()
```

```python
class Question(object):

    def __init__(self, question_id, data_formatter):
        self._question_id = question_id
        self._data_formatter = data_formatter

    def get(self):
        self._get_data()
        if isinstance(self._data_formatter, JsonFormatter) and self._data_formatter._data_formatting ==
"txt":
            # do some thing
            return
        else:
            return self._data_formatter.format()
        # try:
        #     return self._data_formatter.format()
        # except Exception:
        #     # do something
        #     return

    def _get_data(self):
        # connect_MySQL
        # get_data_from_MySQL
        # connect_Redis
        # get_data_from_Redis
        pass
```

使用可替换的组件来构建软件系统

同一层次的组件遵守同一个约定，以方便替换

面向对象特征 → SOLID Principles

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Interface Segregation Principle

No client should be forced to depend on modules it does not use.

Interface Segregation Principle

在设计中避免不必要的依赖

软件系统不应该依赖其不直接使用的组件

面向对象特征 → SOLID Principles

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Open–closed principle

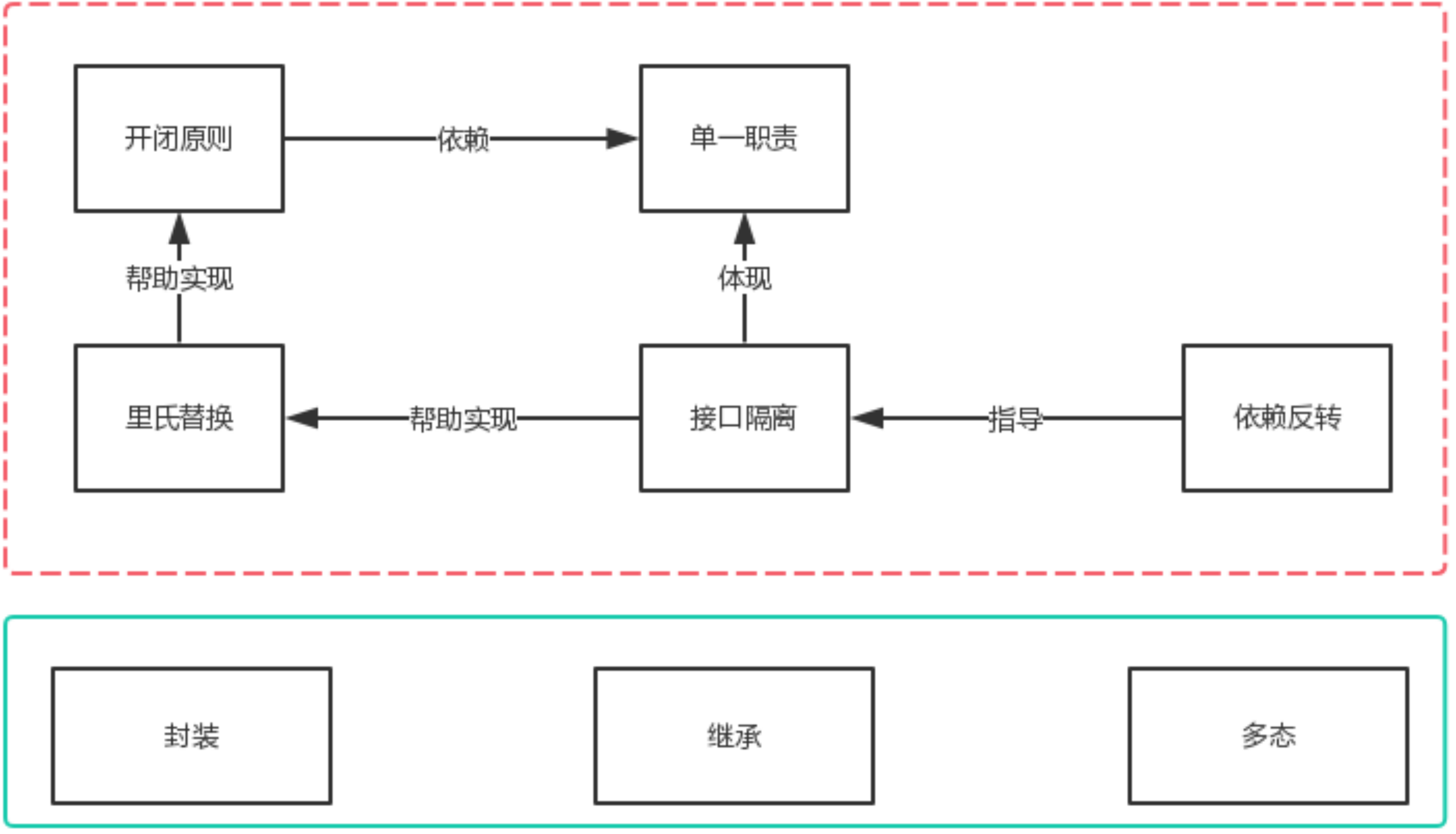
A software artifact should be open for extension but closed for modification.

```python
class Question(object):

    def __init__(self, question_id, data_formatter):
        self._question_id = question_id
        self._data_formatter = data_formatter

    def get(self):
        self._get_data()
        return self._data_formatter.format()

    def _get_data(self):
        # connect_MySQL
        # get_data_from_MySQL
        # connect_Redis
        # get_data_from_Redis
        pass
```

```python
class DataFormatter(object):

    def __init__(self, data_formatting):
        self._data_formatting = data_formatting

    def format(self):
        # do something
        pass


class ThriftFormatter(DataFormatter):
    def format(self):
        # do something
        pass


class JsonFormatter(DataFormatter):
    def format(self):
        # do something
        pass


data_formatter = JsonFormatter(data_formatting="PDF")
Question(question_id=1, data_formatter=data_formatter).get()
```

Open–closed principle

通过新增代码修改系统行为，而非修改原来的代码


开闭原则
依赖
单一职责
帮助实现
体现
里氏替换
帮助实现
接口隔离
指导
依赖反转
封装
继承
多态

总结

- 封装，尽可能掩盖模块内部实现细节，方便进行迭代和替换
- 继承，遵循 LSP，非 is-a 或 is-a-kind-of 的关系，而是 hehaves-like-a, is-substitutable-for 的关系
- SRP，一个模块只负责一类事情，有且只有一个被修改的理由
- DIP，高层策略性代码不要依赖实现底层细节，底层细节的代码应该依赖高层策略性代码
- LSP，使用可替换的组件来构建软件系统，同一层次的组件遵守同一个行为约定，以方便替换
- SRP，不要依赖其不直接使用的组件
- OCP，通过新增代码修改系统行为，而非修改原来的代码

- 模块一定要隐藏实现细节，只暴露必要接口
- 优先考虑组合，后考虑在行为一致的基础上使用继承
- 高层策略性代码不要依赖实现底层细节，底层细节的代码应该依赖高层策略性代码

- 模块一定要隐藏实现细节，只暴露必要接口
- 优先考虑组合，后考虑在行为一致的基础上使用继承
- 高层策略性代码不要依赖实现底层细节，底层细节的代码应该依赖高层策略性代码

- 模块一定要隐藏实现细节，只暴露必要接口
- 优先考虑组合，后考虑在行为一致的基础上使用继承
- 高层策略性代码不要依赖实现底层细节，底层细节的代码应该依赖高层策略性代码

- 模块一定要隐藏实现细节，只暴露必要接口
- 优先考虑组合，后考虑在行为一致的基础上使用继承
- 高层策略性代码不要依赖实现底层细节，底层细节的代码应该依赖高层策略性代码

Reference

- 《敏捷软件开发》
- 《设计模式》
- 《冒号课堂》
- 《Clean Architecture》
- 《面向对象葵花宝典：思想、技巧与实践》
- 《Fluent Python》
- 《代码大全》
- 《松本行弘的程序世界》
- Encapsulation is not information hiding
- ENCAPSULATION WITH EXAMPLE AND BENEFITS IN JAVA & OOP
- Introduction IoC, DIP, DI and IoC Container

- 写了这么多年代码，你真的了解设计模式么?
- 写了这么多年代码，你真的了解SOLID吗?
- Python SOLID
- 正交设计，OO与SOLID
- 解密“设计模式”
- 为何大量设计模式在动态语言中不适用?
- Is Liskov Substitution Principle incompatible with Introspection or Duck Typing?
- Spring IoC有什么好处呢?
- SOLID Python: SOLID principles applied to a dynamic programming language
- SOLID
- 谈谈面向对象编程

Q & A

姚钢强

但行好事，莫問前程。

216	3304	621
创作	赞同	感谢

扫码来知乎找我

知乎

发现更大的世界