封装、继承和多态三者是 面向对象 思想的灵魂，对于面向对象的编程语言来说，理解这三者是至关重要的。而且一旦掌握，当学习其他面向对象的编程语言时，就能快速代入，毕竟思想层面的东西的。

一、封装

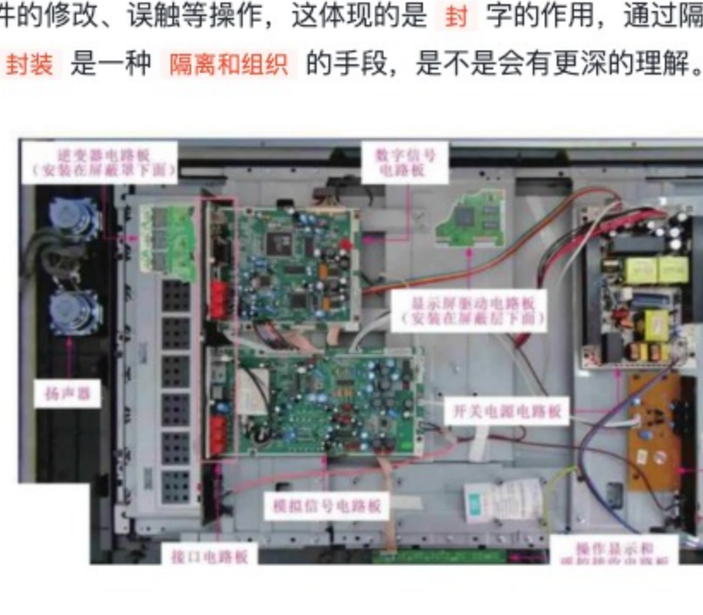
封装的思想在我们日常生活中比比皆是，你完全不用担心无法理解这个概念。封装，由两个字构成：其中 封 代表一种 "私密" 或 "密闭" 性，是一种 隔离手段 。比如 信封 将 信 和外界进行隔离，保证了 信 的私密性； 老坛 将 酸菜 在和外界进行隔离，内部对象相互作用，进行发酵，保证了密闭性。
装 代表一种 "包含" 或 "盛纳" 性，是一种 组织手段 。比如，把自行车部件装在一起，就是对零部件进行组织，使其成为一个更大的个体，完成特定的功能。所以封装的本质就是一种 隔离和组织 的手段。其实 函数 本身就是对语句的封装，对某些逻辑进行组织和盛纳。

1. 封装的优势

封装的优势，也就是说 隔离 和 组织 的优势。隔离，可以避免 污染 ，或说 减少外界对内部产生影响 ，在一定程度上可以保证内部功能运行的 稳定性 ，降低有外界误操作导致的，某些内部对象功能异常的可能性。

举个简单的例子，一台电视机，在卖给用户之前，必须经过 外壳的封装 。从 外壳 对于 电视机 的重要性，就能体会出封装的优势。可以试想一下，如果电视机没有外壳，并不会影响它的核心功能，那外壳有什么作用呢？

首先， 外壳 会作为一个 载体容器 ，为内部的零部件提供 "工作空间"，这体现的是 装 字的作用，对零件进行有序的 组织 ，使零件集合成为一个功能独立的个体。另外， 外壳 可以隔绝外界对内部零件的修改、误触等操作，这体现的是 封 字的作用，通过隔离来可以避免 污染 。现在再回想 封装 是一种 隔离和组织 的手段，是不是会有更深的理解。



上面两个方面是 封装 本质的优势，但在此之上会衍生出其他的优势。比如，封装在很大程度上可以屏蔽内部工作的细节，只暴露必要的方法来 间接 操作内部零件。这样用户在使用时，就无须面对复杂的内部结构，也无需理解内部的构成原理，就可以使用封装体的功能，这大大降低使用的门槛，简称 易用性 。这是 封 字优势的拓展。

另外， 封装体 是由若干个体进行组织形成的更大的个体。当在多处地方需要用到 封装体 ，我们只需要创建封装体对象执行功能即可，而不是在每处地方都写一遍相似的零件组织流程。也就是说，通过封装，我们只需要 买 电视，而不是 造电视 。这就是对个体进行有序组织的好处，可以更好地复用组织逻辑，简称 复用性 。这是 装 字的优势拓展。

2. 编程中的封装

其实，类的定义本身就是封装的过程； 类盛放着 成员变量和方法 ，这些就像是构成电视的零部件，类为它们提供工作空间， 并和外界进行隔离。这就是类封装的直观体现。但有一点千万不要忽略：类是对解决问题过程中，对参与角色的 数据和行为 的封装，也就是说，抛开问题来谈 类的封装 是没有意义的。
对于不同的问题，同类事物可以有不同的封装方式，解决问题是编程的最终目的。可能很多朋友写代码，问题中的角色关系完全不在乎，有什么数据、功能都不管，先写个 class 再说。除非你是大神，一切胸有成竹，下笔一气呵成，不然很容易把代码写得乱七八糟。

在封装的过程中，有一个很重要的环节：对一类事物数据和行为的提取。这种对特征的提取，可以称之为 "抽象" ，而且封装是一种横向提取的抽象过程。比如在绘制图形的这个问题中，对于矩形来说，重要的数据有 中心坐标 、 宽 、 高 ；重要的行为有 绘制 、 移动 、 旋转 等方法。

这样，我们可以定义如下的 Rectangle 类对这些数据和行为进行封装，将成员变量和成员方法组织起来，形成一个独立的，名为 Rectangle 的个体，这个过程就是 封装 。

```dart
---->[grammar/oop/feature/01_encapsulation.dart]----
class Rectangle{
  Vec2 center;
  double width;
  double height;

  Rectangle(this.center,this.height,this.width);

  void draw(){
    String info = "绘制矩形，中心点(${center.x},${center.y})"
        "宽:$height, 高: $width";
    print(info);
  }

  void move(){
    center.x += 10;
    center.y += 10;
  }

  void rotate(){

  }
}
```

3. 封装的缺点

世界上没有任何事物或概念是完美无缺的，有好处，必然有坏处。我们所要做的是 权衡利弊 ，做出当前场景下最有益的决策。封装，虽然可以简化用户的使用，但是由于用户无法感知和理解内部实现细节，一旦当前对象内部出现异常，无法工作。一般用户就手足无措，无法进行及时修复。这也是很多编程者在使用三方类库时，出现问题时的真实写照。

就像是现在交通出行逐渐便利，人们生活水平越来越高，这是好事。但另一方面，一部分人的运动量会变少，越来越懒惰，肥胖也会影响一些人的身体健康。客观条件虽然是好的，但由于一部分人的主观因素，可能导致反效果。封装的优势也可能出现这种现象。

一方面封装可以降低用户使用的门槛，但另一方面也会降低入门的水平，如果层次较低的朋友过于依赖别人的类库，在很大程度上会限制自身能力的成长。很多编程者工作三四年，也就只会调调三方库的方法。另外，还有一部分人，当可以轻松使用他自身水平之上的功能时，会带来盲目的自信，创造出一些傲慢的低能者，这是非常可怕的。

封装的本意是大家都可以发挥自己的创造力，促进共同进步。但事与愿违，能让别人做的为什么要自己做？这种普遍的思想造成了编程能力的两极格局。能力低的人可以依靠别人封装的类库实现功能，其中一部分人会安于现状，能用就行。从而缺乏主观能动性，导致两极分化严重。其实这本质上还是 个人态度 的问题， 封装 只是一个 诱因 。另外，不合理的过度封装，也会导致代码结构和逻辑的复杂，从而影响开发效率，这还是 个人能力 的问题 。

最后，就是封装对细节的屏蔽，在非常复杂的系统中，当创作者的离开，又未能留下相关说明，接手的人很难去理解这些细节。上一篇也说过，当一个封装体的内部细节变得无人可知，那就相当于它从 封装 到被 封死 了。当分工变得越来越细致， 个人 对整体的把握程度就越低，就像现在没有任何一个人能精通所有自然学科知识一样。一个庞大的封装体系的传承、维护和拓展本身就是一件非常困难的事。

二、继承

继承 作为一个耦合性极强的类间关系，在编程界饱受诟病。可以说 继承 是一种打破封装性，来共享自身 数据和行为 的举动。如果说 封装 是 面向对象 半壁江山，那么另外半壁就是 继承 。下面我们从梦的起点，来思考一下，对于面向对象而言，继承为什么是不可或缺的，它的存在是为了解决什么样的问题？

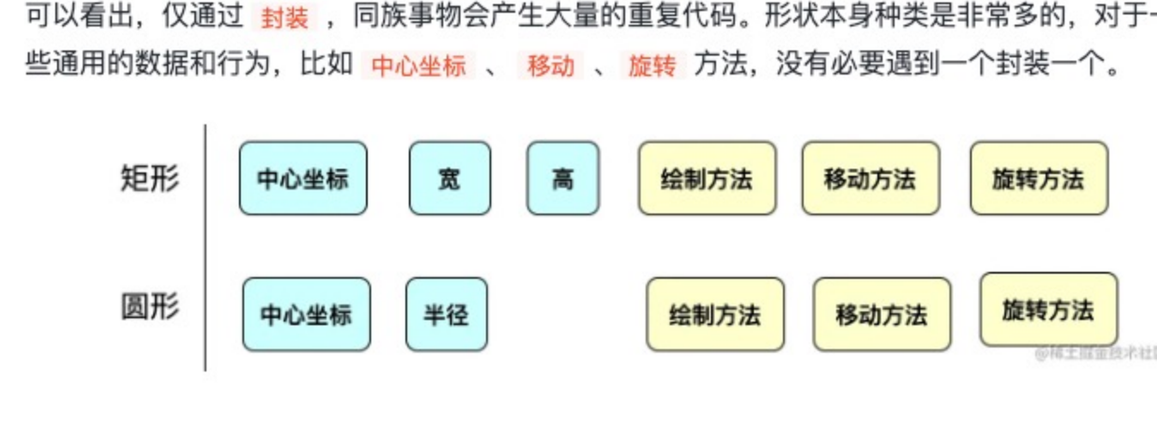
1. 继承存在的价值

前面说过 封装是一种横向提取的抽象过程 ，而 继承 和 封装 不在一个维度上。 继承 是一种纵向的抽象过程，考虑的是族系的派生。

这就有个问题，我们为什么需要定义基类，来对类进行派生呢？只用封装，遇到什么封装什么，有什么局限吗？ 封装 在一定程度上简化了使用，可以更好地复用。但是在类定义时，如果对同类事物各自封装，在代码表现上来看也会产生大量冗余。特别是一个体系类型众多，或未来可能在体系中引入新类时，仅靠 封装 就会显得 "捉襟见肘" 。打个比方，如果在没有继承的情况下，解决绘制 圆形 时，定义如下：

```dart
---->[grammar/oop/feature/02_extends.dart]----
class Circle{
  Vec2 center;
  double radius;


  Circle(this.center,this.radius);

  void draw(){
    String info = "绘制圆形，中心点(${center.x},${center.y})"
        "半径:$radius";
    print(info);
  }

  void move(){
    center.x += 10;
    center.y += 10;
  }

  void rotate(){

  }
}
```

可以看出，仅通过 封装 ，同族事物会产生大量的重复代码。形状本身种类是非常多的，对于一些通用的数据和行为，比如 中心坐标 、 移动 、 旋转 方法，没有必要遇到一个封装一个。



这时继承的优势就体现出来了。我们可以定义一个基类来维护族群间的通用数据和行为：

```dart
class Shape{
  Vec2 center;

  Shape(this.center);

  void move(){
    center.x += 10;
    center.y += 10;
  }

  void rotate(){

  }
}
```

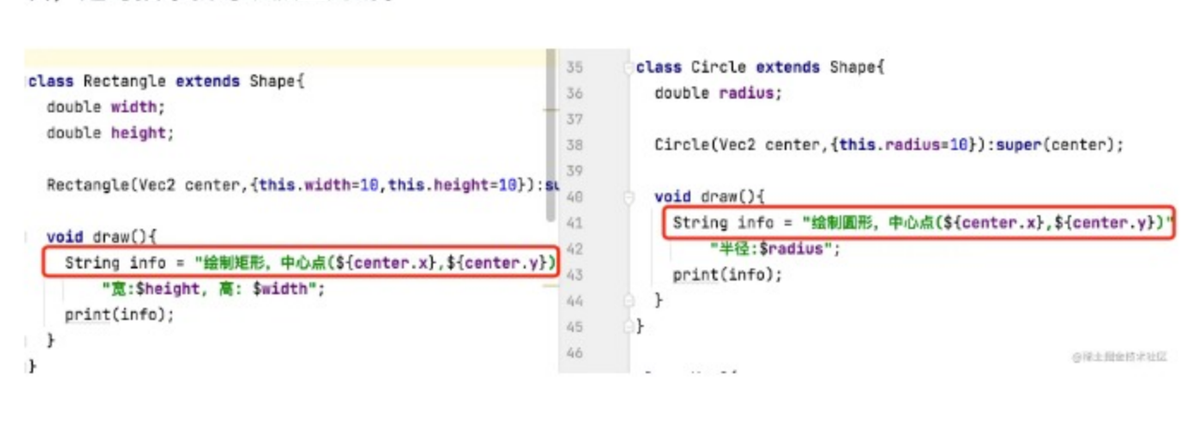
这样就可以通过继承减少重复的代码，让派生类更专注于自身的逻辑实现。比如下面的 Rectangle 类继承自 Shape ，就相当于在其中隐形地定义了 center 中心点成员，以及 move 和 rotate 方法。

```dart
class Rectangle extends Shape{
  double width;
  double height;

  Rectangle(Vec2 center,{this.width=10,this.height=10}):super(center);

  void draw(){
    String info = "绘制矩形，中心点(${center.x},${center.y})"
        "宽:$height, 高: $width";
    print(info);
  }
}
```

2.抽象类与继承

下一章会详细介绍 抽象类 相关的知识，这里简单介绍一下。对于继承而言， 抽象类 的存在非常重要的，不然基类中很难感知到派生类的行为，这会导致只有绝对独立的行为，才可以被提取到基类中。这会让对公共逻辑的提取产生很大的障碍，比如一类事物在处理某个行为时，有固定的流程，流程中有一小部分逻辑是子类特殊的行为，基类中无法实现。如果不存在抽象类，那基类就无法封装这种行为，这样某个派生类就不得不处理这些固定流程。

使用 抽象类 的价值在于，允许定义和访问未实现的方法，这种方法被称为 抽象方法 。比如上面 Rectangle 和 Circle 类中的 draw 方法，有一些公共逻辑，也有一些本类中特有的逻辑，这时抽象就可以派上用场。



比如下面，通过 abstract 来声明抽象类，其中定义了 drawInChild 抽象方法，但没有对该方法进行实现，并在 draw 方法中使用该方法。也就是说在抽象类中可以调用暂时不知道怎么实现的方法，这有助于整体逻辑的搭建，将具体的细节延迟到派生类中实现。

```dart
---->[grammar/oop/feature/03_abstract.dart]----
abstract class Shape{
  Vec2 center;

  Shape(this.center);

  void move(){
    center.x += 10;
    center.y += 10;
  }

  void draw(){
    String info = "绘制矩形，中心点(${center.x},${center.y})"
        "${drawInChild()}";
    print(info);
  }

  String drawInChild(); // dart

  void rotate(){

  }
}
```

继承自抽象类，派生类必须实现其中的抽象方法 ，这是语法级别的 铁律 。如下在 drawInChild 方法中进行关于矩形的绘制，这样我们就能进一步 浓缩 ，将和派生类无关点的公共逻辑进一步提取。注意一下，这里只是为了描述简洁，在 draw 方法中打印信息来表示绘制，重点是体会其中蕴含的思想，不要太纠结为什么 drawInChild 里返回个字符串就算是绘制了。

```dart
class Rectangle extends Shape{
  double width;
  double height;

  Rectangle(Vec2 center,{this.width=10,this.height=10}):super(center);

  @override
  String drawInChild() => "宽:$height, 高: $width";
}
```

3.派生类方法的覆写

覆写 的概念非常简单，派生类中实现与基类中的同名方法，称为 覆写 。通过覆写，可以自定义派生类的某种行为来完成特殊的需求。比如现在只想在 Rectangle 的绘制前后添加一句输出，如下 tag1 处，可以覆写 draw 方法来进行额外处理。另外，可以通过 super.XXX 来调用父类的方法。

```dart
---->[grammar/oop/feature/04_override.dart]----
class Rectangle extends Shape{
  double width;
  double height;

  Rectangle(Vec2 center,{this.width=10,this.height=10}):super(center);

  @override
  String drawInChild() => "宽:$height, 高: $width";

  @override
  void draw(){ // tag1
    print("绘制矩形前====");
    super.draw();
    print("绘制矩形后====");
  }
}
```

覆写可以通过 @override 注解进行标注，这不是强制性的，但是最好加一下，有利于区分覆写方法，提高可读性。这样，在 Rectangle 对象调用 draw 方法时，就会优先调用覆写的方法。如果没有覆写，就调用基类方法。

```dart
void main(){
 Rectangle rectangle = Rectangle(Vec2(10,10));
 rectangle.move();
 rectangle.draw();
}

---->[打印日志]----
绘制矩形前====
绘制矩形，中心点(20.0,20.0)
宽:10.0, 高: 10.0
绘制矩形后====
```

4. 继承的优缺点

很多人可能觉得继承不好，会让耦合性变高， 组合 应优于 继承 。其实我只想说，没有什么好与不好，只有适合与不适合。场景对于一件事物的优劣会起到至关重要的影响，比如你掉进一个洞穴里出不来，馒头要比黄金对你来说更重要。继承的缺点，往往是使用的场景或使用的方式有问题，这本质上是人的问题，并非工具的问题。编码者对关系把握不到位，写的乱七八糟，导致维护困难， 继承 表示这锅我不背。

有人会说 继承打破了封装性 是缺点，其实恰恰相反，正是因为封装对同族事物处理的缺陷，才需要的 继承 关系。 打破封装性 ，打破的是 封装 的局限性，这并不是缺点，这也是能够复用代码的前提，所以不要本末倒置。继承是从语法层面，提供的一套代码复用和类型派生的机制，是面向对象一个非常重要的特性。

你有没有想过，为什么说 封装 、 继承 、 多态 是面向对象的三大特性，而不是说 多态 、 继承 、 封装 呢？因为只有封装，才能产生类；只有存在类，才能说类与类间的关系，继承才有意义；只有存在 继承 体系，才能有多态的概念。这三者是环环相扣，不可逆序的。

三、多态

我们刚才说 封装 和 继承 都是面向对象的半壁江山。可能会有人疑惑，不是三大特性吗，那多态算什么？比喻一下，多态就相当于江山之上，世间的繁华万象。

1. 什么是多态

俗话说， 龙生九子，各有不同 ，这就是由基类派生的多样性。多态，就是指 基类 能以 多种派生类的形态 参与程序运行时。一个很明显的特征是： 声明的父类型对象，可以使用子类型进行赋值 ，通俗点说，就是: 派生类对象可以作为基类对象看待。

如下 tag1 和 tag2 所示，两个对象类型都是 Shape ，但其运行时可以是不同子类型的对象。因此在调用 Shape 对象的 draw 方法时，会根据运行时类型执行不同的逻辑。这就是多态最核心的体现：

```dart
---->[grammar/oop/feature/05_polymorphism.dart]----
void main(){
 Shape rectangle = Rectangle(Vec2(10,10)); // tag1
 rectangle.draw();

 Shape circle = Circle(Vec2(10,10)); // tag2
 circle.draw();
}
```

其实 多态 也很符合我们对显示的认知，比如说 人 是一种 生物 ，用伪代码表现就是：

```dart
Biology instance = Human(); // Biology 生物, Human 人类
```

这种 衍生类型 可视为 基类型 ，是一种很自然的思维。父类型可指向子类对象，但不能反过来，如果说 生物 是一种 人 , 就是逻辑的错误。另外， 植物 也是一种 生物 ，可以通过如下伪代码表示：

```dart
Biology instance = Plant(); // Biology 生物, Plant 植物
```

这样大家可以体会一下，一种类型，多种形态的意思。 其实 多态 的概念就和生物多样性一样：基于一个概念而派生出的其他子概念，这是继承的结果；而多态，是让子类型可以视为父类型。

2.多态的价值

多态，为面向对象提供了一个非常重要的功能：当面向基类编程时，其行为可以产生多样性。比如下面的 drawShape 方法，入参是 Shape 抽象类，在其中调用 shape 对象的 draw 方法。

```dart
void drawShape(Shape shape){
    shape.draw();
}
```

这样的好处在于，执行的具体逻辑会随着入参的具体对象而不同。对 drawShape 方法而言， shape 对象的行为就具有多样性 。对于 drawShape 方法而言，在使用时传入什么 Shape 的派生类就可以绘制什么，可以屏蔽派生类对象之间的差异性。

```dart
void main(){
 Shape rectangle = Rectangle(Vec2(10,10));
 drawShape(rectangle);

 Shape circle = Circle(Vec2(10,10));
 drawShape(circle);
}
```

这在现实生活中也很好理解，比如老师对你说： 再喊两个同学过来帮忙搬资料 。这里的 同学 是一个抽象的概念，喊过来的同学拥有能力的具体实现者。对于搬资料这个问题而言，在乎的是同学拥有搬资料的能力，而不是同学是哪个人。这就是 抽象 的作用，我们在意的是事物的 能力 ，而非具体事物。

基于多态性的特点，可以对通用逻辑更好地进行逻辑封装，让一个功能更加灵活，并具有拓展性，甚至可以应对未来未知的变化。比如 drawShape 方法的入参是 Shape ，未来可能需要绘制正多边形，只需要实现相关派生类即可。而 drawShape 本身并不关心具体绘制的是什么，只要是 Shape 的派生类都能画。所以 drawShape 方法本身，通过多态的特性可以应对某些需求的变化，避免代码的改动。

3.多态的限制

多态在 屏蔽派生类对象之间差异性 的同时，也会 屏蔽掉派生类各自的特点 。在 tag1 处，将 Rectangle 对象声明为 Shape 对象，那么你就无法访问到矩形的宽高。同样在 drawShape 中，入参是 Shape 对象，在方法体中就无法访问派生类中定义的额外成员。

```dart
Shape rectangle = Rectangle(Vec2(10,10)); //tag1
// rectangle.width // Error, 无法访问!

void drawShape(Shape shape){
    shape.draw();
}
```

这算不上是缺陷，只能说是限制，在实际编写代码中权衡一下即可。如果实在想通过基类对象访问派生类的成员，也有途径，可以通过 is 关键字来校验是否是 Rectangle ，在执行相关代码：

```dart
void drawShape(Shape shape){
    shape.draw();
    if(shape is Rectangle){
      print("绘制矩形: 宽高(${shape.width},${shape.height})");
    }
}
```

到这里面向对象的三大特性： 封装 、 继承 和 多态 就介绍完了。总的来看， 封装 是 继承 的基础、 继承 是 多态 的前提、而 多态 丰富 封装 的内容。这三者不是相互独立的，而是彼此依存，共同作用的。理解这三个特性，是面向对象编程最重要的，没有之一。下一章，我们将认识一下，建立在 封装 、 继承 、 多态 基础之上的，三个更为抽象的概念： 抽象 、 接口 和 混入 。

留言

输入评论（Enter换行，⌘ + Enter发送）

发表评论

全部评论（10）

ObliviateOnline | 安卓开发工程师 @ 某监... | 3月前
在开发中自觉的就这么使用了， 回顾这些概念，作者通过不同的表达方式，将一些已经印象模糊的知识再次加强印象，瞬间更清晰了
点赞 回复

那物已非 | 前端 | 5月前
最后总结 "这三者不是相互独立的，而是彼此易存。" 这里应该是笔误，易存 ->依存
点赞 2

> 张风捷特烈 （作者） | 5月前
> 已更正
> 1 回复
>
> 那物已非 回复 张风捷特烈 | 5月前
> 速度真快，下一章都没看完。
> "已更正"
> 点赞 回复

w___ | 5月前
覆写 的概念非常简单，派生类中实现与基类中的同名方法，成为 覆写(称为?)
点赞 1

> 张风捷特烈 （作者） | 5月前
> 已更正
> 点赞 回复

mzhi | 5月前
这种 衍生类型 可是为 基类型 （可视为？）
点赞 1

> 张风捷特烈 （作者） | 5月前
> 已更正
> 点赞 回复

mzhi | 5月前
没有不要遇到一个封装一个（没有必要？）
点赞 1

> 张风捷特烈 （作者） | 5月前
> 已更正
> 点赞 回复