引言

面向对象编程，是基于 对象 间的相互协作，来解决问题的一种思考方式。它不像 面向 过程那样，凡事都要 亲力亲为 ；面向对象处理问题的过程中，通过 类 对事物的 数据 和 行为 进行封装，可以让各类事物的职能逻辑更加 "紧凑"，不至于零散地分处各地。这无形中会增加代码结构的稳定性，使其更易维护和拓展。

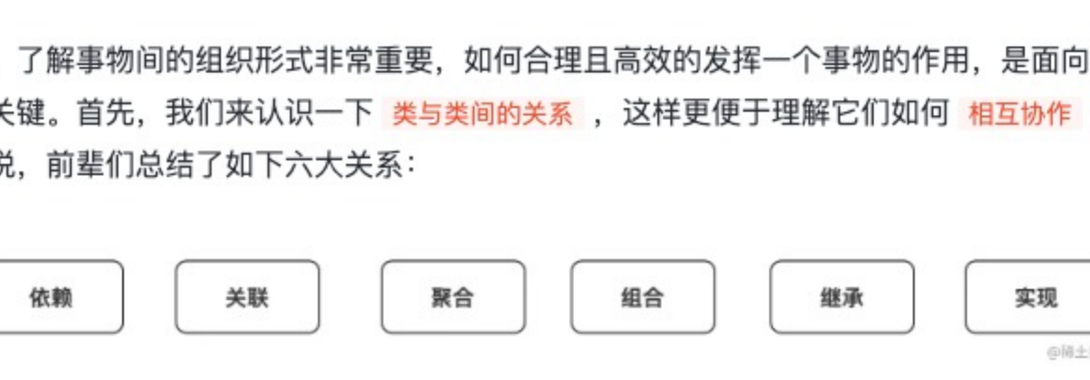
稳定的结构 是生物进化的前提，如果细胞间无法以一种稳定而高效的组织形式，维护各成员间的关系，内部的矛盾会导致庞大结构的解体。小到元素、细胞、大到国家、宇宙都是如此，想要集结大量的个体构成一个更巨大个体，组织形式至关重要。

对于一个 应用程序 而言，其本身就是大量 数据和函数 的集结体，想要让这个 巨大个体 存活或说进化。高效、稳定地对其进行管理是必不可少的，否则当一个应用的源码臃肿到难以维护，等待它的只有溃烂与死亡。

这里强调一下，我并不是说 面向对象 编程是唯一的 "良药"；也不是说，你用面向对象的思想来写代码，就一定能写出完美无缺，流传千古的程序。暂且不说你对思想的把握程度，人类编程史也不过六十几年，编程的思想现在也只能说在 萌芽阶段 ，面向对象也有它的弊端，这也是封装不可避免的弊端： 对细节的屏蔽 。

这是一把双刃剑：一方面 屏蔽细节 可以简化使用者的调用，但另一方面，也可能导致当创作者看不懂自己写的是什么，或创作者的离去，其他接盘侠难以胜任，就会成为 "幽灵代码"。从而导致上层建筑的繁荣，而底层根基的从 封装 到 封死 。这和 微观世界规律 非常类似，当最底层的细节无法窥探，很多技术就会陷入瓶颈，这和《三体》里智子锁死地球科技有些相像。

所以，了解事物间的组织形式非常重要，如何合理且高效的发挥一个事物的作用，是面向对象编程的关键。首先，我们来认识一下 类与类间的关系 ，这样更便于理解它们如何 相互协作 。一般来说，前辈们总结了如下六大关系：



---

1. 依赖关系

一般来说，在 B 类实现功能的逻辑中，使用到 A 类对象，称 B 依赖于 A 。比如下面代码中， Drawable 类在 draw 方法中使用了 Vec2 对象，所以 Drawable 的功能依赖于 Vec2 ，两者是依赖关系。我们一般通过如下的 单向虚线箭头 表示这种关系，其中箭头指向 被依赖方 。


```dart
---->[grammar/oop/relation/01_dependence.dart]----
class Drawable{
  void draw(Vec2 vec2){
    print("绘制向量(${vec2.x},${vec2.y})");
  }
}

class Vec2{
  double x;
  double y;

  Vec2(this.x, this.y);
}
```

可以看出 依赖关系 是比较容易形成的，关系比较薄弱，说专业一点就是 耦合性低 。而且关系不需要长期维持，是一种临时关系。比如快递小哥为我送餐，我们之间建立的是一种临时的依赖关系，我依赖于他的服务。

```dart
void main(){
  Drawable drawable = Drawable();
  Vec2 p0 = Vec2(3, 4);
  drawable.draw(p0);
}

---->[打印输出]----
绘制向量(3.0,4.0)
```

2.关联关系

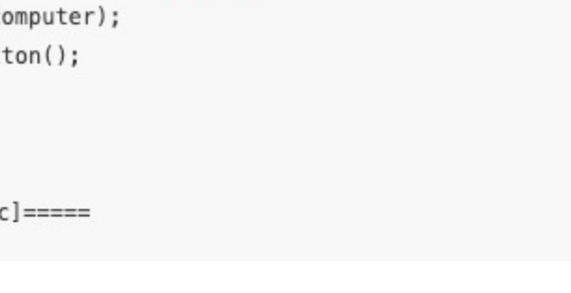
关联关系 指两类之间存在 固定 的对应关系，相对于 依赖 而言，关系要会更强。这种关系一般是长期的，稳定的。比如下面的 用户 User 和 电脑 Computer 的关系， User 持有 Computer 是一种强的依赖关系。被关联者通常以 成员变量 的形式存在于关联者类中。我们一般通过如下的 单向实线箭头 表示这种关系，其中箭头指向 被关联方 。


```dart
---->[grammar/oop/relation/02_association.dart]----
class User{
  User(this.computer);
  Computer computer;

  void pressStartButton(){
    computer.open();
  }
}

class Computer{
  Computer(this.name);
  String name;

  void open(){
    print("====电脑开机:[${name}]=====");
  }
}
```

如下，在使用时，提供构造方法对 User 中的 computer 成员变量进行赋值：

```dart
void main(){
  Computer computer = Computer("toly mac");
  User user = User(computer);
  user.pressStartButton();
}

---->[打印输出]----
====电脑开机:[toly mac]=====
```

通过线型也可以看出关系的强弱， 依赖关系 是虚线，表示关系较弱； 关联关系 是实线，表示关系稳固。另外，当两各类相互关联时，称 双向关联 ，这也很容易理解。比如 Computer 中现在添加一个 User 类型的 owner 对象，表示电脑的 主人 ，这时 User 和 Computer 就是双向的关联关系。用双向实线箭头表示：




```dart
class Computer{
  String name;
  User? owner;

  Computer(this.name);

  void open(){
    print("====电脑开机:[${name}]=====");
  }
}
```

---

3. 聚合关系

聚合 是一种 耦合性 更强的 关联关系 。一般 关联关系 的两个类，是相互独立的，在地位上平等。而 聚合关系 在语义上有 包含 的涵义，更强调 整体/个体 的区别。比如说，对于电脑而言，和 显卡 、 内存 等硬件间这就是一种 聚合关系 。 我们一般通过如下的 空心菱形实线箭头 表示这种关系，菱形指向 整体类 ，箭头指向 个体类 。


```dart
---->[grammar/oop/relation/03_aggregation.dart]----
class Computer{ // 电脑
  String name;
  GraphicsCard graphicsCard;
  Memory memory;

  Computer(this.name,this.graphicsCard,this.memory);

  void printConfig(){
    print("====电脑名称:[${name}]=====");
    print("====电脑显卡:[${graphicsCard.name}]=====");
    print("====电脑内存:[${memory.type}:${memory.size}GB]=====");
  }
}

class GraphicsCard{ // 显卡
  String name;
  GraphicsCard(this.name);
}

class Memory{ // 内存
  double size;
  String type;

  Memory(this.type,this.size);
}
```

可以看出 聚合关系 在代码上的表现，和 关联关系 一样，也是持有 个体 类作为成员变量。不过对于 聚合 来说，两类间的关系更加紧密，耦合性更强，有 整体/个体 的区别，在地位上是 不平等 的。

```dart
void main(){
  GraphicsCard graphicsCard = GraphicsCard("NVIDIA GeForce RTX 3060");
  Memory memory = Memory("DDR4", 32);
  Computer computer = Computer("toly mac",graphicsCard,memory);
  computer.printConfig();
}
```

---

4. 组合关系

组合 是一种 耦合性 比 聚合 更强的 关联关系 。它同样强调 整体/个体 间的关系，但要求 整体 与 个体 不可分割。这个 不可分割 体现在 个体 的生命周期被 整体 控制， 整体 对象的消亡，也会导致 个体 的消亡。

比如 学生 和 档案 之间的关系，随着 学生 学习阶段的变化，档案信息会随之变化。当学生死亡时，它的档案也就没有意义了。 我们一般通过如下的 实心菱形实线箭头 表示这种关系，菱形指向 整体类 ，箭头指向 个体类 。从实心和空心上也能感受到 组合 的关系更为紧密。


```dart
---->[grammar/oop/relation/04_composition.dart]----
class Student{
  Document document;

  Student(this.document);

  void event(String string){
    document.records.add(string);
  }

  String callDocument(){
    return document.records.join("\n");
  }

  void died(){
    document.records.clear();
  }
}

class Document{
  List<String> records = [];

  void addRecord(String string){
    records.add(string);
  }
}
```

如下所示，随着 Student 的生命活动，档案会进行变化。档案和学生两个事物紧紧绑定，体现着一种较强的附属关系。

```dart
void main(){
  Document document = Document();
  Student student = Student(document);
  student.event("2000 年 入学 XXXX 小学");
  student.event("2006 年 入学 XXXX 初中");
  student.event("2009 年 入学 XXXX 高中");
  student.event("2012 年 入学 XXXX 大学");
  String info = student.callDocument();
  print(info);
}

---->[打印输出]----
2000 年 入学 XXXX 小学
2006 年 入学 XXXX 初中
2009 年 入学 XXXX 高中
2012 年 入学 XXXX 大学
```

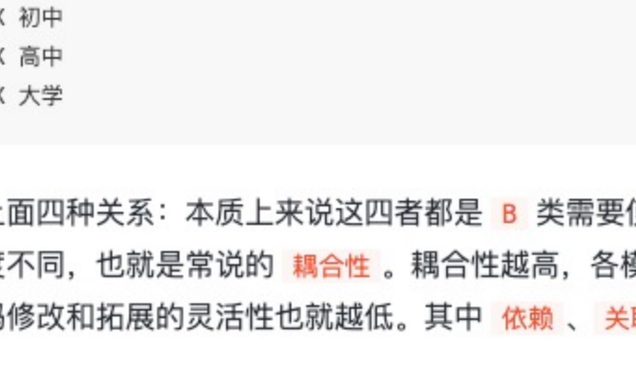
这里来小结一下上面四种关系：本质上来说这四者都是 B 类需要使用到 A 类对象，只不过类间关系的紧密程度不同，也就是常说的 耦合性 。耦合性越高，各模块间的联系越紧密，独立性就越差，对于代码修改和拓展的灵活性也就越低。其中 依赖 、 关联 、 聚合 、 组合 四者的耦合性依次增高：

- 依赖 是对 A 的使用，也被称为 use-a 。
- 关联 、 聚合 、 组合 都是持有 A 类成员，称为 has-a 。

5.继承关系

继承 作为 面向对象三个基本特征之一 ，被广泛地应用于程序开发中。前四种关系只是在 使用 、 持有 的层面， B 与 A 都是以 独立个体 存在的；而继承在语义上是 让 B 成为 A ，此时 B 与 A 将作为同一类型被对待，被成为 is-a ，这就表示 继承 的 耦合性 要远高于组合。在功能上， B 类将拥有 A 类的所有 成员属性/方法 ，这也是变相地让 B 类可以使用 A 类的功能。

继承的好处是可以建立一个类族系，从而更好地模拟现实中事物的关系。在代码层面，通过继承可以简化一个族系中公共成员属性及成员方法的逻辑实现，起到一定的便于复用价值。另外，上面四种关系在代码中的体现是在逻辑层面的，而 继承 是语法层次的特性支持。



如果 B 类继承自 A 类，我们一般称 A 为 基类 ，也有称为 超类 ，口语化中也被成为 父类 ； B 为 派生类 或 子类 。用什么称呼不用太纠结，自己喜欢就好，什么称呼大家都能理解。我们一般通过 单向实线三角空心箭头 表示这种关系，其中箭头指向 基类 。

```dart
---->[grammar/oop/relation/05_generalization.dart]----
class Shape {
  Vec2 center;

  Shape(this.center);

  void move() {
    center.x += 1;
    center.y += 1;
    print("$runtimeType:move(10,10)==> center:(${center.x},${center.y})");
  }
}

class Circle extends Shape{
  double radius;
  Circle(Vec2 center,{this.radius =10}) : super(center);
}

class Rectangle extends Shape{
  double width;
  double height;

  Rectangle(Vec2 center,{this.width=20,this.height=20}) : super(center);
}
```

如下， Circle 和 Rectangle 派生类对象都可以使用基类 Shape 的成员方法。在 move 方法中， center 成员变量被修改。另外通过 runtimeType 可以查看当前对象运行时类型：

```dart
void main() {
  Circle circle = Circle(Vec2(100,100),radius: 50);
  Rectangle rectangle = Rectangle(Vec2(50,50));
  circle.move();
  rectangle.move();
}

---->[打印输出]----
Circle:move(10,10)==> center:(110.0,110.0)
Rectangle:move(10,10)==> center:(60.0,60.0)
```

这里先简单认识一下 类与类 间继承关系的基本特点，另外基于 继承 而衍生出的其他特性，是个比较大的话题。在后面会单独提供一章，来详细介绍面向对象的 封装 、 继承 、 多态 的知识。

---

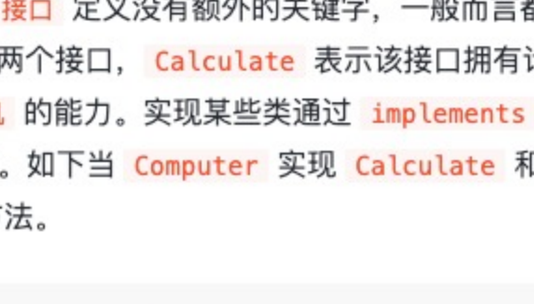
6. 实现关系

在一般面向对象的编程语言中，不支持一个类继承自多个类，因为这样可能一些歧义问题。如果两个类中的 成员变量 相同，或者 成员方法 的定义相同，子类未进行覆写，那子类在调用时无法选择使用那个，这就是 二义性 。但可以一个类可以 实现 多个 接口 ，因为 接口 不允许持有普通的成员变量，而且本身只定义 成员方法 ,要求派生类必须进行实现。这样接口就从本质上避免了 多继承 的歧义问题。

在 Dart 中，对于 接口 定义没有额外的关键字，一般而言都是抽象方法的抽象类可以被视为 接口 。如下定义了两个接口， Calculate 表示该接口拥有计算的能力， Electric 表示该接口拥有 开机 、 关机 的能力。实现某些类通过 implements 关键字完成，其后可以加多个接口，通过 , 号隔开。如下当 Computer 实现 Calculate 和 Electric 接口时，就必须实现接口中定义的全部方法。

```dart
---->[grammar/oop/relation/06_implementation.dart]----
abstract class Calculate{
  void calculate();
}

abstract class Electric{
  void open();
  void close();
}

class Computer implements Calculate, Electric{
  @override
  void calculate() {
    print("do calculate by computer");
  }

  @override
  void close() {
    print("computer close");
  }

  @override
  void open() {
    print("computer open");
  }
}
```

Human 也有计算能力，所以也可以实现 Calculate 。从这里可以看出 接口 可以对功能进行 隔离 ，这对于 封装 而言是非常重要的。我们一般通过 单向虚线三角空心箭头 表示这种关系，其中箭头指向 接口类 。




```dart
class Human implements Calculate{
  @override
  void calculate() {
    print("do calculate by Human");
  }
}
```

到这里，就介绍完了 类与类 间的六种关系，不需要死记硬背，对它们的理解是非常重要的。我们在解决问题，或说设计程序过程中，能认识各模块的功能关系，才能给出合理的结构。这对于新手来说过于遥远，不过这六种关系本身的理解本身并不是很难。

在日常开发中可以多思考一下，前辈们或者源码中如何维护类与类间的关系，也有利于我们自身的成长，更能成体系的去审视整个运作流程，而不是 仅 拘泥于一步一步的代码逻辑。下一章，将进入面向对象的灵魂，认识面向对象的三大特性： 封装 、 继承 与多态。