Dart是一个面向对象的语言，面向对象中非常重要的概念就是类，类产生了对象。

**1.1. 类的定义**

在Dart中，定义类用class关键字。

类通常有两部分组成：成员（member）和方法（method）。

定义类的伪代码如下：

class 类名 {

类型 成员名;

返回值类型 方法名(参数列表) {

方法体

}

}

编写一个简单的Person类：

* **这里有一个注意点:** 我们在方法中使用属性(成员/实例变量)时，并没有加this；
* Dart的开发风格中，在方法中通常使用属性时，会省略this，但是有命名冲突时，this不能省略；

class Person {

String name;

eat() {

print('$name在吃东西');

}

}

我们来使用这个类，创建对应的对象：

* 注意：从Dart2开始，new关键字可以省略。

main(List<String> args) {

// 1.创建类的对象

var p = new Person(); // 直接使用Person()也可以创建

// 2.给对象的属性赋值

p.name = 'why';

// 3.调用对象的方法

p.eat();

}

**1.2. 构造方法**

**1.2.1. 普通构造方法**

我们知道, 当通过类创建一个对象时，会调用这个类的构造方法。

* 当类中没有明确指定构造方法时，将默认拥有一个无参的构造方法。
* 前面的Person中我们就是在调用这个构造方法.

我们也可以根据自己的需求，定义自己的构造方法:

* **注意一：**当有了自己的构造方法时，默认的构造方法将会失效，不能使用
  + 当然，你可能希望明确的写一个默认的构造方法，但是会和我们自定义的构造方法冲突；
  + 这是因为Dart本身不支持函数的重载（名称相同, 参数不同的方式）。
* **注意二：**这里我还实现了toString方法

class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

@override

String toString() {

return 'name=$name age=$age';

}

}

另外，在实现构造方法时，通常做的事情就是通过**参数**给**属性**赋值

为了简化这一过程, Dart提供了一种更加简洁的语法糖形式.

上面的构造方法可以优化成下面的写法：

Person(String name, int age) {

this.name = name;

this.age = age;

}

// 等同于

Person(this.name, this.age);

**1.2.2. 命名构造方法**

但是在开发中, 我们确实希望实现更多的构造方法，怎么办呢？

* 因为不支持方法（函数）的重载，所以我们没办法创建相同名称的构造方法。

我们需要使用**命名构造方法**:

class Person {

String name;

int age;

Person() {

name = '';

age = 0;

}

// 命名构造方法

Person.withArgments(String name, int age) {

this.name = name;

this.age = age;

}

@override

String toString() {

return 'name=$name age=$age';

}

}

// 创建对象

var p1 = new Person();

print(p1);

var p2 = new Person.withArgments('why', 18);

print(p2);

在之后的开发中, 我们也可以利用命名构造方法，提供更加便捷的创建对象方式:

* 比如开发中，我们需要经常将一个Map转成对象，可以提供如下的构造方法

// 新的构造方法

Person.fromMap(Map<String, Object> map) {

this.name = map['name'];

this.age = map['age'];

}

// 通过上面的构造方法创建对象

var p3 = new Person.fromMap({'name': 'kobe', 'age': 30});

print(p3);

**1.2.3. 初始化列表**

我们来重新定义一个类Point, 传入x/y，可以得到它们的距离distance:

class Point {

final num x;

final num y;

final num distance;

// 错误写法

// Point(this.x, this.y) {

// distance = sqrt(x \* x + y \* y);

// }

// 正确的写法

Point(this.x, this.y) : distance = sqrt(x \* x + y \* y);

}

上面这种初始化变量的方法, 我们称之为初始化列表(Initializer list)

**1.2.4. 重定向构造方法**

在某些情况下, 我们希望在一个构造方法中去调用另外一个构造方法, 这个时候可以使用重定向构造方法：

* 在一个构造函数中，去调用另外一个构造函数（注意：是在冒号后面使用this调用）

class Person {

String name;

int age;

Person(this.name, this.age);

Person.fromName(String name) : this(name, 0);

}

**1.2.5. 常量构造方法**

在某些情况下，传入相同值时，我们希望返回同一个对象，这个时候，可以使用常量构造方法.

默认情况下，创建对象时，即使传入相同的参数，创建出来的也不是同一个对象，看下面代码:

* 这里我们使用identical(对象1, 对象2)函数来判断两个对象是否是同一个对象:

main(List<String> args) {

var p1 = Person('why');

var p2 = Person('why');

print(identical(p1, p2)); // false

}

class Person {

String name;

Person(this.name);

}

但是, 如果将构造方法前加const进行修饰，那么可以保证同一个参数，创建出来的对象是相同的

* 这样的构造方法就称之为常量构造方法。

main(List<String> args) {

var p1 = const Person('why');

var p2 = const Person('why');

print(identical(p1, p2)); // true

}

class Person {

final String name;

const Person(this.name);

}

常量构造方法有一些注意点:

* **注意一：**拥有常量构造方法的类中，所有的**成员变量必须是final修饰**的.
* **注意二:** 为了可以通过常量构造方法，创建出相同的对象，不再使用 **new**关键字，而是使用const关键字
  + 如果是将结果赋值给const修饰的标识符时，const可以省略.

**1.2.6. 工厂构造方法**

Dart提供了factory关键字, 用于通过工厂去获取对象

main(List<String> args) {

var p1 = Person('why');

var p2 = Person('why');

print(identical(p1, p2)); // true

}

class Person {

String name;

static final Map<String, Person> \_cache = <String, Person>{};

factory Person(String name) {

if (\_cache.containsKey(name)) {

return \_cache[name];

} else {

final p = Person.\_internal(name);

\_cache[name] = p;

return p;

}

}

Person.\_internal(this.name);

}

**1.3. setter和getter**

默认情况下，Dart中类定义的属性是可以直接被外界访问的。

但是某些情况下，我们希望监控这个类的属性被访问的过程，这个时候就可以使用setter和getter了

main(List<String> args) {

final d = Dog("黄色");

d.setColor = "黑色";

print(d.getColor);

}

class Dog {

String color;

String get getColor {

return color;

}

set setColor(String color) {

this.color = color;

}

Dog(this.color);

}

**1.4. 类的继承**

面向对象的其中一大特性就是继承，继承不仅仅可以减少我们的代码量，也是多态的使用前提。

Dart中的继承使用extends关键字，子类中使用super来访问父类。

父类中的所有成员变量和方法都会被继承,，但是构造方法除外。

main(List<String> args) {

var p = new Person();

p.age = 18;

p.run();

print(p.age);

}

class Animal {

int age;

run() {

print('在奔跑ing');

}

}

class Person extends Animal {

}

子类可以拥有自己的成员变量, 并且可以对父类的方法进行重写：

class Person extends Animal {

String name;

@override

run() {

print('$name在奔跑ing');

}

}

子类中可以调用父类的构造方法，对某些属性进行初始化：

* 子类的构造方法在执行前，将隐含调用父类的无参默认构造方法（没有参数且与类同名的构造方法）。
* 如果父类没有无参默认构造方法，则子类的构造方法必须在初始化列表中通过super显式调用父类的某个构造方法。

class Animal {

int age;

Animal(this.age);

run() {

print('在奔跑ing');

}

}

class Person extends Animal {

String name;

Person(String name, int age) : name=name, super(age);

@override

run() {

print('$name在奔跑ing');

}

@override

String toString() {

return 'name=$name, age=$age';

}

}

**1.5. 抽象类**

我们知道，继承是多态使用的前提。

所以在定义很多通用的**调用接口**时, 我们通常会让调用者传入父类，通过多态来实现更加灵活的调用方式。

但是，父类本身可能并不需要对某些方法进行具体的实现，所以父类中定义的方法,，我们可以定义为**抽象方法**。

什么是 **抽象方法**? 在Dart中没有具体实现的方法(没有方法体)，就是抽象方法。

* 抽象方法，必须存在于抽象类中。
* 抽象类是使用abstract声明的类。

下面的代码中, Shape类就是一个抽象类, 其中包含一个抽象方法.

abstract class Shape {

getArea();

}

class Circle extends Shape {

double r;

Circle(this.r);

@override

getArea() {

return r \* r \* 3.14;

}

}

class Reactangle extends Shape {

double w;

double h;

Reactangle(this.w, this.h);

@override

getArea() {

return w \* h;

}

}

注意事项:

* **注意一：**抽象类不能实例化.
* **注意二：**抽象类中的抽象方法必须被子类实现, 抽象类中的已经被实现方法, 可以不被子类重写.

**1.6. 隐式接口**

Dart中的接口比较特殊, 没有一个专门的关键字来声明接口.

默认情况下，定义的每个类都相当于默认也声明了一个接口，可以由其他的类来实现(因为Dart不支持多继承)

在开发中，我们通常将用于给别人实现的类声明为抽象类:

abstract class Runner {

run();

}

abstract class Flyer {

fly();

}

class SuperMan implements Runner, Flyer {

@override

run() {

print('超人在奔跑');

}

@override

fly() {

print('超人在飞');

}

}

**1.7. Mixin混入**

在通过implements实现某个类时，类中所有的方法都必须被重新实现(无论这个类原来是否已经实现过该方法)。

但是某些情况下，一个类可能希望直接复用之前类的原有实现方案，怎么做呢?

* 使用继承吗？但是Dart只支持单继承，那么意味着你只能复用一个类的实现。

Dart提供了另外一种方案: **Mixin混入的方式**

* 除了可以通过class定义类之外，也可以通过mixin关键字来定义一个类。
* 只是通过mixin定义的类用于被其他类混入使用，通过with关键字来进行混入。

main(List<String> args) {

var superMan = SuperMain();

superMan.run();

superMan.fly();

}

mixin Runner {

run() {

print('在奔跑');

}

}

mixin Flyer {

fly() {

print('在飞翔');

}

}

// implements的方式要求必须对其中的方法进行重新实现

// class SuperMan implements Runner, Flyer {}

class SuperMain with Runner, Flyer {

}

**1.8. 类成员和方法**

前面我们在类中定义的成员和方法都属于对象级别的, 在开发中, 我们有时候也需要定义类级别的成员和方法

在Dart中我们使用static关键字来定义:

main(List<String> args) {

var stu = Student();

stu.name = 'why';

stu.sno = 110;

stu.study();

Student.time = '早上8点';

// stu.time = '早上9点'; 错误做法, 实例对象不能访问类成员

Student.attendClass();

// stu.attendClass(); 错误做法, 实现对象补鞥呢访问类方法

}

class Student {

String name;

int sno;

static String time;

study() {

print('$name在学习');

}

static attendClass() {

print('去上课');

}

}

**1.9. 枚举类型**

枚举在开发中也非常常见, 枚举也是一种特殊的类, 通常用于表示固定数量的**常量值**。

**1.9.1. 枚举的定义**

枚举使用enum关键字来进行定义:

main(List<String> args) {

print(Colors.red);

}

enum Colors {

red,

green,

blue

}

**1.9.2. 枚举的属性**

枚举类型中有两个比较常见的属性:

* index: 用于表示每个枚举常量的索引, 从0开始.
* values: 包含每个枚举值的List.

main(List<String> args) {

print(Colors.red.index);

print(Colors.green.index);

print(Colors.blue.index);

print(Colors.values);

}

enum Colors {

red,

green,

blue

}

枚举类型的注意事项:

* 注意一: 不能子类化、混合或实现枚举。
* 注意二: 不能显式实例化一个枚举