Flutter中的abstract、implements、extends、mixin关键字

为什么划船不靠桨

1、abstract

使用关键字 `abstract` 标识一个类可以让类成为抽象类，抽象类将无法被实例化,也就是说不能直接使用抽象类，只能使用抽象类的子类。

抽象类定义的方法只定义不进行实现，就是抽象方法（还有抽象属性），需要子类进行方法实现，已经实现的方法就是普通方法。

```
abstract class Animal{
  void updateChildren();
}
class Dog extends Animal{
  @override
  void updateChildren() {
    // 已经实现的方法子类不强制实现(普通方法),只有定义没有实现的是抽象方法子类必须实现
  }
}
// Animal a = Animal();//报错，抽象类没办法直接实例化一个方法(抽象类不能直接使用,要使用抽象类的子类)
```

2、implements

一个类可以通过关键字 `implements` 来实现一个或多个接口并实现每个接口定义的 API，例如想要创建一个 A 类支持调用 B 类的 API 且不想继承 B 类，则可以实现 B 类的接口。

```
class Person {
  final String _name;

  Person(this._name);

  String greet(String who) => 'Hello, $who. I am $_name.';
}

class Impostor implements Person {
  String get _name => '';

  String greet(String who) => 'Hi $who. Do you know who I
am?';
}

String greetBob(Person person) => person.greet('Bob');

void main() {
  print(greetBob(Person('Kathy')));
  print(greetBob(Impostor()));
}
```

Hello, Kathy who. I am Kathy_name
Hi who. Do you know who I am?

如果需要实现多个类接口，可以使用逗号分割每个接口类

```
class Point implements Comparable, Location {...}
```

3、extends

使用 `extends` 关键字来创建一个继承自某个类的子类，并可使用 `super` 关键字引用一个父类。

```
class Television {
  void turnOn() {
  }
}

class SmartTelevision extends Television {
  @override
  void turnOn() {
    super.turnOn();
  }
}
```

如果要复用抽象类里面的方法，并且要用抽象方法约束子类的话就用`extends`继承抽象类。
如果只是把抽象类当作标准的话就用`implement`实现抽象类。
需要注意`implements` 的`abstract` 需要实现接口方法，但是`extends` 如果具备同名方法就无需重写。看下面的例子：

```
abstract class First {
  void doPrint() {
    print('First');
  }
}
```

```
abstract class Second {
  void doPrint() {
    print('Second');
  }
}

class Father {
  void doPrint() {
    print('Father');
  }
}

class Son extends Father implements First,Second {}
```

调用：

```
Son().doPrint();
```

打印：

```
Father
```

因为已经在父类已经实现了同名方法，所以不需要再子类重新实现`doPrint()`方法。这时如果在`Second`或者`First`中再加入其他方法，结果我们可以想到，肯定会报错。

4、mixin和with

mixin 是Dart 2.1 加入的特性，以前版本通常使用abstract class代替。Dart为了支持多重继承，引入了mixin关键字，它的特点是：
1.mixin定义的类不能有构造方法，这样可以避免继承多个类而产生的父类构造方法冲突。
2.可以混入多个类。
3.with关键字能够实现mixin,可以想象成多继承，而且是以类似于栈的形式实现（最后一个混入的 mixins 会覆盖前面一个 mixins 的特性，这里需要注意)。
4.with和 implements的区别就是with不需要实现继承接口方法。

想要实现一个 Mixin，请创建一个继承自 Object且未声明构造函数的类。除非你想让该类与普通的类一样可以被正常地使用，否则请使用关键字 mixin 替代 class。

```
mixin Musical {    }   //使用 mixin 关键字创建一个混入类
```

可以使用关键字 on 来指定哪些类可以使用该 Mixin 类。比如有 Mixin 类 MusicalPerformer，但是 MusicalPerformer只能被 Musician 类使用，则可以这样定义 MusicalPerformer。

```
class Musician {}
//使用 on 关键字规定混入类 MusicalPerformer 只能在子类继承自
Musician 的时候才能使用。
```

```
//
mixin MusicalPerformer on Musician {}
class SingerDancer extends Musician with MusicalPerformer
{}
```

on关键字会把使用了混入类的自动关联为父子关系，看下面的例子：

```
class Father {
  void init() {
    print('Father init');
  }
}
//当使用on关键字，则表示该mixin只能在那个类的子类使用了，
// 那么结果显然的，mixin中可以调用那个类定义的方法、属性
mixin FirstMixin on Father {
  void init() {
    print('FirstMixin init start');
    super.init();
    print('FirstMixin init end');
  }
}

mixin SecondMixin on Father {
  void init() {
    print('SecondMixin init start');
    super.init();
    print('SecondMixin init end');
  }
}

class Son2 extends Father with FirstMixin, SecondMixin {
```

```
  @override
  void init() {
    print('Son2 init start');
    super.init();
    print('Son2 init end');
  }
}
```

方法调用及打印：

```
Son2().init();

flutter: Son2 init start
flutter: SecondMixin init start
flutter: FirstMixin init start
flutter: Father init
flutter: FirstMixin init end
flutter: SecondMixin init end
flutter: Son2 init end
```

这里super调用会发现，with关键字把FirstMixin和SecondMixin以及Father自动关联为父子，这一切都是基于on关键字，且对Father这个家族（可以理解为整个家族）增加束缚，Son2这个类也只能继承Father，如果增加其他家族的mixin，就会报错。

```
class OtherFamily{
  void init(){
    print('其他家族');
```

```
  }
}

mixin OtherFamilyMixin on OtherFamily{
@override
  void init() {
    // TODO: implement init
    super.init();
  }
}

class Son2 extends Father with FirstMixin,
SecondMixin ,OtherFamily{
   //这里如果 with 最后面跟的是 OtherFamilyMixin，那么肯定就会报
错，
  //原因上面已经说过了，继承自Father类，只能使用FirstMixin和
SecondMixin两个混入类。
  @override
  void init() {
    print('Son2 init start');
    super.init();
    print('Son2 init end');
  }
}
```

打印：

```
flutter: Son2 init start
flutter: 其他家族
flutter: Son2 init end
```

这里发生了有点混乱的情况，`super`直接指向`OtherFamily`,但是他又不忘记继承的`Father`，还是依旧能调用`Father`里的方法。

`on`关键字后边不止可以跟一个类,其实可以跟多个类，但是跟了多个类，则在使用混入类的时候，需要在`with`关键字的后面先跟上除继承类的其他类，否则如果先跟的是混入类就会报错

```
//新建一个类
class Mother{
  void initMother() {}
}
//将原来的混入类由原来的Father改为Father,Mother
mixin SecondMixin on Father,Mother {
  void init() {
    print('SecondMixin init start');
    super.init();
    print('SecondMixin init end');
  }
}

class Son2 extends Father with
FirstMixin,Mother,SecondMixin {
  //这里再进行混入的时候，需要先写FirstMixin或Mother,将
SecondMixin写在最后，否则会报错
  @override
  void init() {
    print('Son2 init start');
    super.init();
    print('Son2 init end');
  }
```

```
}
```

还可以像下面这样使用

```
class A { void a(){ print("a"); } }
class A1 { void a(){ print("a1"); } }
class B with A,A1{ }
class B1 with A1,A{ }
class B2 with A,A1{ void a(){ print("b2"); } }
class C { void a(){ print("c"); } }
class B3 extends C with A,A1{ }
class B4 extends C with A1,A{ }
class B5 extends C with A,A1{ void a(){ print("b5"); } }

B b = B();      b.a();    //a1
B1 b1 = B1();   b1.a();   //a
B2 b2 = B2();   b2.a();   //b2
B3 b3 = B3();   b3.a();   //a1
B4 b4 = B4();   b4.a();   //a
B5 b5 = B5();   b5.a();   //b5
```

这里可以对某一个类进行拆解进行分析

```
class B3 extends C with A,A1{}
//B3类的结构可以分解为
class CA = C with A;
class CAA1 = CA with A1;
class B3 extends CAA1{ }
```

如果新建一个类，将上面讲的关键字都合在一起

```
class D extends C with A implements A1{ }
//创建对象，调用方法
D().a(); //a
```

这里说明一下，`A1`是接口，`with`关键字依旧遵守上边将的性质，首先实现`A`中的方法（`with`相当于实现了可以省略不写，但是同名，也相当于重写了implements的接口），如果把`A`中的`a()`方法注释掉，则会打印`c`。

看下面这个例子

```
mixin _CounterStateMixin < T extends StatefulWidget> on
State<T> {
  int _counter = 0;
  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
}
class _MyHomePageState extends State<MyHomePage> with
_CounterStateMixin {

  @override
  void initState() {
    _incrementCounter();
    super.initState();
  }
```

```
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```