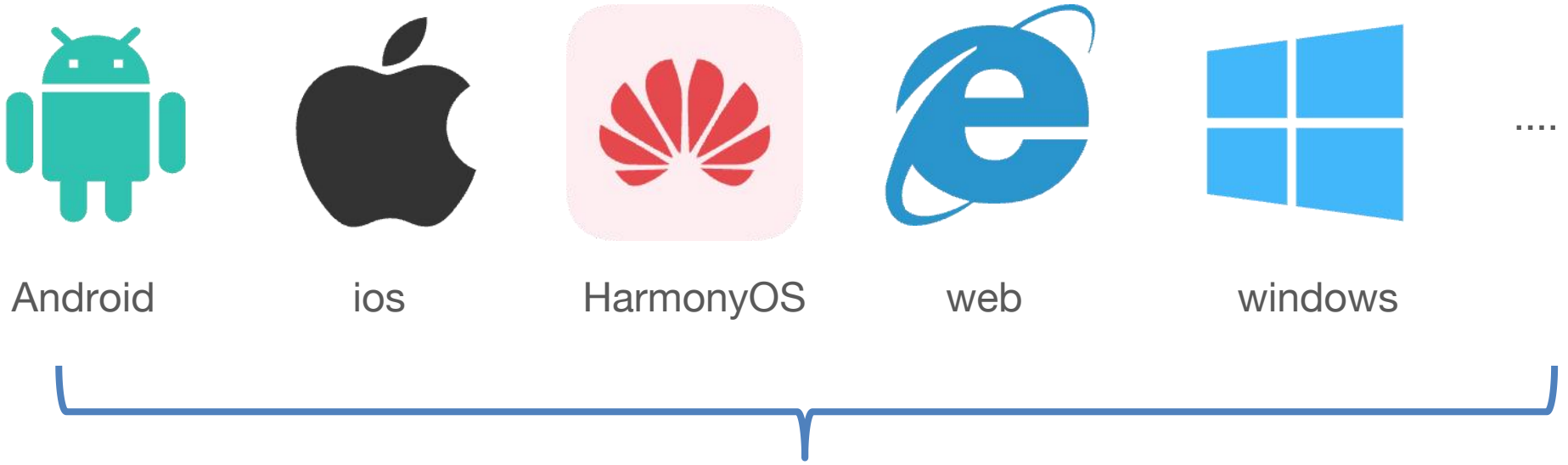
Flutter从入门到实战

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

Flutter是什么?

Flutter 是 Google 开源的一个用于构建高性能、高质量原生界面应用程序的软件开发工具包



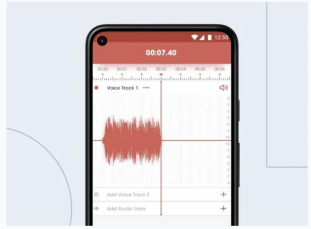
Flutter 目的是用一套代码同时为以上多个平台开发原生质量的应用程序

为什么要学Flutter?

开发效率：只需要一套Dart代码，即可编译生成原生性能的各类平台（Android/IOS/HarmonyOS/..）应用

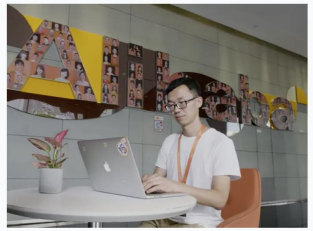
市场份额：Flutter 以 46% 的开发者采用率稳居跨平台框架首位

企业应用：全球已经发布了50w+款应用采用Flutter技术开发，其中15%占比为大厂应用



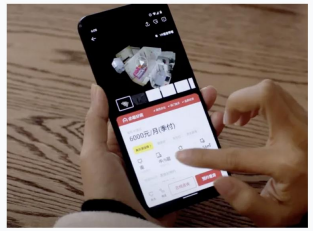
阿比路录音室

用 Flutter 重塑歌曲创作



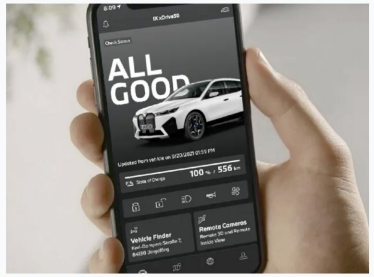
阿里巴巴集团

阿里集团通过 Flutter 构建和扩展自己的二手交易平台：闲鱼



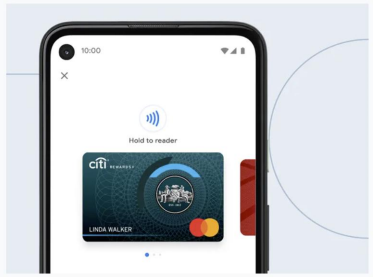
贝壳找房

贝壳找房携手 Flutter，为三亿家庭提供更好的居住服务



BMW

BMW 集团使用 Flutter 加大以客户为中心的产品投入



Google Pay

借助 Flutter，Google Pay 正在走向全球

字节跳动

字节跳动使用 Flutter 提升了 33% 的研发效率

本次课程亮点

最新Flutter版本

- Dart最新版本
- 适配鸿蒙系统

渐进式知识脉络

- 语言基础
- 框架基础
- 综合案例

渐进式环境配置

- Dart基础环境
- Flutter开发环境
- Web运行环境
- Android
- IOS
- HarmonyOS

- AI编辑器Trae辅助开发
- 紧跟随市场需求更新
- 依托企业用人需求研发
- 对零基础同学更友好

课程安排

1.Dart语言基础

2.Flutter组件核心

3.综合案例

Dart语言基础

数据类型	运算符	流程控制	函数	类	异步编程
变量/常量-const/final	算术运算符	if分支	函数定义	类的定义	事件循环
字符串-String	比较运算符	三元运算符	必传参数	默认构造函数	Future
布尔-bool	赋值运算符	switch/case	可选位置参数	命名构造函数	链式调用
数字-int/double/num	逻辑运算符	while循环	可选命名参数	私有属性	async/await
列表-List		for循环	匿名函数	继承/多态	
字典-Map			箭头函数	混入/泛型	
动态类型- dynamic					
空安全					

Flutter核心

Flutter工程	组件	路由	组件通信	网络请求	弹层及动画
搭建Flutter开发环境	无状态组件	命名路由	父传子	Dio安装	信息弹窗
工程目录解析	有状态组件	非命名式路由	子传父	基础使用	确认弹窗
启动文件	组件生命周期	路由传参	Provider	发送请求	进度条弹窗
页面结构	Container组件	路由拦截	Getx	拦截器	动画
Material风格体系	文本组件	404		处理异常	
	图片组件				
	SIngleChildScrollView				
	ListView				
	GridView				
	CustomScrollView				

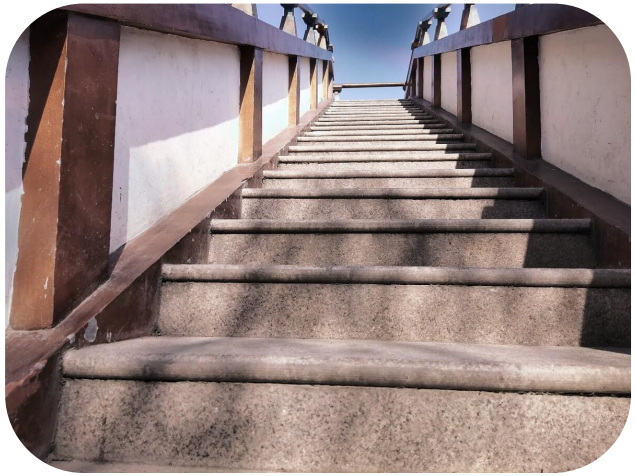
综合案例

- 搭建项目
- 目录结构
- 基础布局
- 网络请求封装
- 加载数据
- 无限滚动
- 性能优化

适配-安卓/苹果/鸿蒙/web

课程说明

阶梯式学习



渐进式配置环境

避免一开始就配置大量环境，导致入门到放弃

不间断式更新

响应市场进行小知识点类更新

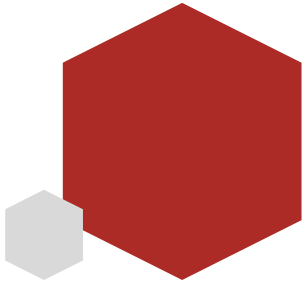
适合什么样的人学习?

- 有一定的编程语言基础
- 想要做跨端应用却对安卓和ios望而却步
- 拥抱新技术，拥抱新生态

一键三连，加关注！

多评论，多互动，互相督促！

您的支持就是我们更新的动力！



Dart语言基础

安装Dart基础学习环境

DartSDK

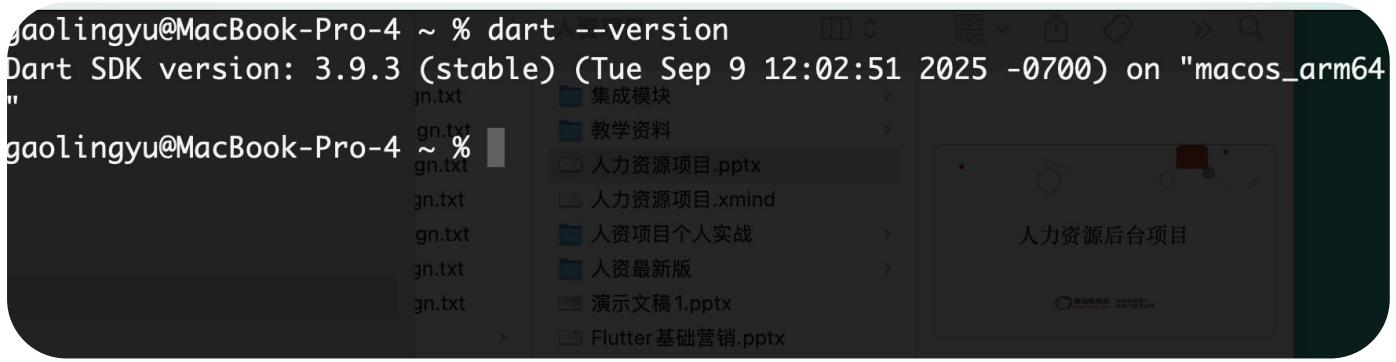
Dart SDK 是 Dart 语言的官方工具包，它提供了一整套完整的工具和库，用于帮助编写、编译、调试和运行 Dart 程序

- windows: 使用安装包安装（资料中提供）
- mac:

```
brew install dart-sdk
```

- 验证sdk:

```
dart --version
```

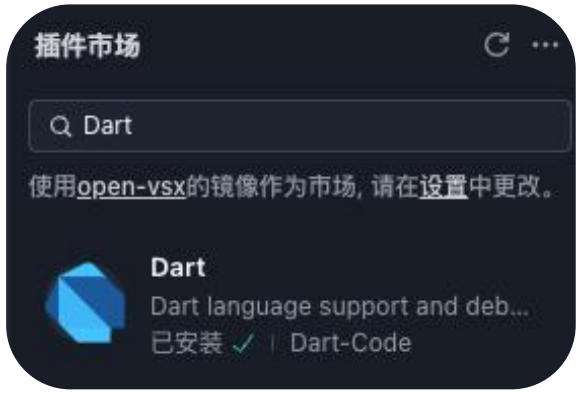


AI编辑器–Trae

- Trae: 支持各类编程语言的IDE开发工具

  下载链接： https://www.trae.cn/

- 安装Dart插件-Dart代码提示



AI编辑器–Trae两种AI模式–Tab模式

●Tab键模式： 开启Tab提示之后，按Tab自动填入代码

AI编辑器–Trae两种AI模式–Builder协作模式

●Builder模式：在对话框中输入需求，直接帮助我们写代码

Dart语言注意事项

- 文件后缀： dart语言文件后缀以.dart结尾
- 入口：dart文件的入口方法为main方法
- 分号：dart文件中绝大部分语句都需要加分号结尾，像 '{}' 后通常不加分号

Dart的变量和常量

Dart中的变量声明–var

- 场景：当一个人的年龄从20岁变成21岁，存储数据发生变化，需要用变量来进行声明
- 关键字：var
- 语法：var 变量名 = 值/表达式;
- 注意：使用var声明的变量，其类型在第一次赋值之后确定，不能再赋值其他类型的值

Dart中的常量声明–const

- 场景：当做圆形计算的时候，π的值一开始就是被确定的，且不允许更改，使用const进行声明
- 关键字：const
- 语法：const 属性名 = 值/表达式;
- 特点：const是代码编译前被确定，不允许表达式中有变量存在，必须为常量或者固定值

```dart
Run | Debug
void main(List<String> args) {
  const num = 3.1415926; // π
  const length = 2 * num * 10; // 圆的周长
}
```

Dart中的常量声明–final

- 场景：当我们需要当前时间作为当前唯一的操作时间，时间一旦被确定就不可以被修改。
- 关键字：final
- 语法：final 属性名 = 值/表达式;
- 特点：final变量在运行时被初始化，其值设置后不可更改

```
Run | Debug
1  void main(List<String> args) {
2    final time = DateTime.now();
3    print(time);
4  }
5
```

总结

- 变量：当需要存储一个变化的数据需要使用var来声明变量
- 编译时常量：当需要存储一个不变的数据，且在编译时就确定，需要使用const声明常量
- 运行时常量：当需要存储一个不变的数据，但是在运行时才确定，需要使用final声明常量

Dart的常用数据类型

Dart中的常用数据类型

- 字符串-String
- 数字-int/double/num
- 布尔-bool
- 列表-List
- 字典-Map
- 动态类型-dynamic

Dart中的常用类型–String

- 场景：当我们需要用一个变量来描述一段文本，就可以使用String来声明。
- 关键字：String
- 语法：String 属性名 = ‘文本内容’;
- 特点：引号支持双引号或者单引号，支持拼接及模板字符串

```
Run | Debug
1  void main(List<String> args) {
2    String text = '今天是个好日子';
3    print(text);
4    text = "明天同样是个好天气";
5    print(text);
6  }
7
```

Dart中的常用类型–String–模板字符串

- 场景：当String类型的变量当前时间来显示，可以使用模板字符串或者拼接实现。
- 需求：打印出我要在(具体时间)的时候吃早饭
- 语法：String 属性名 = '文本内容$变量名'; 或 String 变量名 = '文本内容${变量名}';
- 注意：当存在模板中的内容是一个表达式的时候需要使用${}, 更推荐使用${}

Dart中的数字类型–int/num/double

- 场景：当我们需要描述一个数字类型的时候，需要使用int/num/double
- 区别：int-整型数字，num-可整型可小数，double-小数
- 语法：int/num/double 属性名 = 数值;

```dart
void main(List<String> args) {
  int friendCount = 3; // 我有几个朋友
  print('我有$friendCount个朋友');
  double appleCount = 3.4; // 我买了3.4斤苹果
  print('我买了$appleCount斤苹果');
  num rest = 1.5; // 我有几个月的假期
  print('我今年有$rest月的假期');
}
```

Dart中的数字类型–赋值关系

- 注意：数字之间的赋值关系

  double和int不能直接赋值

  num不能直接给double赋值

  double可直接给num赋值

```
Run | Debug
void main(List<String> args) {
  int friendCount = 3; // 我有几个朋友
  print('我有$friendCount个朋友');
  double appleCount = 3.4; // 我买了3.4斤苹果
  print('我买了$appleCount斤苹果');
  num rest = 1.5; // 我有几个月的假期
  print('我今年有$rest月的假期');
  // friendCount = appleCount; // int和double不允许直接赋值
  friendCount = appleCount.toInt(); // int和double 转化之后可以赋值
  appleCount = friendCount.toDouble();
  rest = appleCount; // double可以直接给num赋值
  appleCount = rest.toDouble(); // num给double赋值需要转化
}
```

Dart中的布尔类型–bool

- 场景：当我们需要一个属性来表示当前为真(true)或假(false)的时候，需要使用bool关键字声明
- 需求：声明当前自己是否已经完成作业
- 语法：bool 属性名 = true/false;

```
Run | Debug
1  void main(List<String> args) {
2    bool isFinishWork = false;
3    print('我完成作业的状态为$isFinishWork');
4  }
5
```

问题 2　输出　调试控制台　终端　GitLens

```
我完成作业的状态为false

Exited.
```

Dart中的列表类型–List

- 场景：当一个变量需要存储多个值的时候，可以使用列表类型List
- 需求：一个班级的学生用List存储，支持对学生的查找、新增、删除、循环
- 语法：List 属性名 = [‘学生1’, ‘学生2’];

```
Run | Debug
1  void main(List<String> args) {
2    List students = ["张三", "李四", "王五"];
3    print(students);
4  }
5
```

问题 2　输出　调试控制台　终端　GitLens　筛选器

```
[张三, 李四, 王五]

Exited.
```

Dart中的列表类型–List的常用操作方法

- 在尾部添加-add(内容)
- 在尾部添加一个列表-addAll(列表)
- 删除满足内容的第一个-remove(内容)
- 删除最后一个-removeLast()
- 删除索引范围内数据-removeRange(start,end)

注意：end不包含在删除范围内

```
un | Debug
void main(List<String> args) {
  List students = ["张三", "李四", "王五"];
  students.add("新同学"); // 在尾部添加
  // 在尾部添加一个数组
  students.addAll([
    "新班的同学1",
    "新班的同学2",
  ]);
  // 删除满足内容的第一个
  students.remove("新同学");
  // 删除最后一个
  students.removeLast();
  // 删除索引范围内的数据
  students.removeRange(0, 2);
```

Dart中的列表类型–List的常用操作方法和属性

- 循环-forEach((item) {});
- 是否都满足条件-every((item) { return 布尔值 });
- 筛选出满足条件的数据-where((item) { return 布尔值 });
- 列表的长度(属性)-length
- 最后一个元素(属性)-last
- 第一个元素(属性)-first
- 是否为空(属性)-isEmpty

```dart
void main(List<String> args) {
  students.forEach((item) {
    print(item);
  });
  // 是否都满足条件
  print(students.every((item) {
    return item.toString().startsWith("新");
  }));
  // 筛选出所有满足条件的数据
  print(students.where((item) {
    return item.toString().startsWith("新");
  }).toList());
  // 列表的长度
  print(students.length);
  // 最后一个元素
  print(students.last);
  // 第一个元素
  print(students.first);
  // 是否为空列表
  print(students.isEmpty);
}
```

Dart中的字典类型–Map

- 场景：当存储的英文需要找到对应的中文描述，需要使用键值对类型Map
- 需求：需要定义一个英文对应中文的描述，并可以通过英文找到对应的中文
- 语法1：Map 属性名 = { key: value };
- 语法2： 字典[key] 可以取值和赋值

```
Run | Debug
1  void main(List<String> args) {
2    Map transMap = {"lunch": "午饭", "morning": "早上", "hello": "你好"};
3    print(transMap["hello"]);
4    transMap["hello"] = "你非常好";
5    print(transMap["hello"]);
6  }
7
```

```
问题 2  输出  调试控制台  终端  GitLens  筛选器(例如 text、!exclude)
你好
你非常好

Exited.
```

Dart中的字典类型–Map的常用操作方法

- 循环-forEach
- 在添加一个字典-addAll
- 是否包含某个key-containsKey
- 删除某个key-remove
- 清空-clear

```dart
main
void main(List<String> args) {
  // 循环
  transMap.forEach((key, value) {
    print('$key,$value');
  });
  // 添加一个字典
  transMap.addAll({"fine": "很好"});
  // 是否包含某个key
  transMap.containsKey("fine");
  // 删除某个key
  transMap.remove("fine");
  // 清空字典
  transMap.clear();
}
```

Dart中的动态类型–dynamic

- 定义：Dart语言中，dynamic用来声明动态类型
- 特点：允许变量运行时自由改变类型，同时绕过编译时的静态检查
- 语法1：dynamic 属性名 = 值;

```
Run | Debug
void main(List<String> args) {
  // 自由设置类型
  dynamic free = 1;
  free = "";
  free = false;
  free = [];
  free = {};
}
```

Dart中的动态类型–dynamic和var的区别

- dynamic： 运行时可自由改变类型，无编译检查，方法和属性直接调用
- var: 根据初始值进行推断类型，确定类型后类型确定，有编译检查，仅限推断的属性和方法

```
Run | Debug
void main(List<String> args) {
  // 自由设置类型
  var text = 1;
  text.toDouble();
  dynamic text1 = 1;
  text1.startsWith(""); // 虽然运行报错，但是绕过编译，不会提示
}
```

Dart的空安全机制

Dart中的空安全机制

- 定义：在Dart语言中，通过编译静态检查将运行时空指针提前暴露
- 特点：将空指针异常从运行时提前至编译时，减少线上崩溃
- 常用空安全操作符

操作符	符号	作用	示例
可空类型	?	声明可空变量	String?→ 允许 String或 null
安全访问	?.	对象为 null时跳过操作，返回 null	user?.name→ 若 user为 null则返回 null
非空断言	!.	开发者保证变量非空（否则运行时崩溃）	name!.length→ 断言 name非空
空合并	??	左侧为 null时返回右侧默认值	name ?? "Guest"→ name为 null时返回 "Guest"

Dart中的空安全机制–安全操作符

```dart
Run | Debug
void main(List<String> args) {
  // 可空类型
  String? username = null;
  username?.length; // 安全访问
  username!.length; // 非空断言
  username ?? "老高"; // 空合并
}

```

Dart的运算符

Dart中的常用运算符

- 算术运算符
- 赋值运算符
- 比较运算符
- 逻辑运算符

Dart中的常见算术运算符

- 定义：在Dart语言中，对数字进行加减乘除运算采用算术运算符

运算符	作用
+	加
-	减
*	乘
/	除
~/	整除
%	取余数

```dart
Run | Debug
void main(List<String> args) {
  double item = 10.99; // 商品单价
  double allPrice = item * 4; // 乘法
  double money = 100; //
  double lastMoney = money - allPrice; // 减法
  // double everyMoney = lastMoney / 4; // 除法
  int everyMoney = lastMoney ~/ 4; // 整除
  // int everyMoney = 10 % 4; // 取余数
  print(everyMoney);
}
```

Dart中的常见赋值运算符

- 定义：在Dart语言中，对数据进行赋值运算采用赋值运算符

运算符	作用
=	赋值操作
+=	加等，a += b 相等于 a = a + b
-=	减等, a -= b 相等于 a = a - b
*=	乘等, a *= b 相等于 a = a * b
/=	除等, a /= b 相等于 a = a / b

```
un | Debug
void main(List<String> args) {
  double a = 1;
  a += 2;
  print(a);
  a -= 1;
  print(a);
  a *= 2;
  print(a);
  a /= 2;
  print(a);
}
```

Dart中的常见比较运算符

- 定义：在Dart语言中，对数值进行比较操作实用比较运算符
- 特点：比较运算符的结果都是布尔类型

运算符	作用
==	判断两个值是否相等
!=	判断两个值是否不等
>	判断左侧值是否大于右侧值
>=	判断左侧值是否大于等于右侧值
<	判断左侧值是否小于右侧值
<=	判断左侧值是否小于等于右侧值

```dart
Run | Debug
void main(List<String> args) {
  int a = 1;
  int b = 2;
  print(a == b);
  print(a != b);
  print(a > b);
  print(a >= b);
  print(a < b);
  print(a <= b);
}
```

Dart中的逻辑运算符

- 定义：在Dart语言中，需要对于bool类型的值进行逻辑运算，需要使用逻辑运算符

运算符	作用
&&	逻辑与，a && b, a和b同时true，得true
||	逻辑或，a || b，a和b有一个true，得true
!	逻辑非，!a，对a变量进行取反

```
Run | Debug
void main(List<String> args) {
  bool isOpenDoor = false; // 是否开门
  bool isOpenLight = true; // 是否开灯
  print(isOpenDoor && isOpenDoor);
  print(isOpenDoor || isOpenDoor);
  print(!isOpenDoor);
}
```

- 注意： 使用逻辑运算符必须保证参与的变量为布尔类型

Dart的流程控制

Dart中的流程控制

- if分支语句
- 三元运算符
- switch/case
- 循环语句

Dart中的if分支语句

- 定义：在Dart语言中，if分支语句可以进行不同逻辑的判断和处理

if分支	作用
单分支	单个条件判断
双分支	两个条件判断
多分支	多个条件判断

```
| Debug
void main(List<String> args) {
  // 单分支
  int score = 61;
  if (score > 60) {
    print("考试及格");
  }
  // 双分支
  bool isMarry = false;
  if (isMarry) {
    print("恭喜你结婚啦");
  } else {
    print("还没结婚哦");
  }
  // 多分支
  if (score > 80) {
    print("优秀");
  } else if (score > 60) {
    print("及格");
  } else {
    print("不及格");
  }
```

Dart中的三元运算符

- 定义：在Dart语言中，三元运算符是一种简化版本的双分支语句
- 语法：表达式(布尔值) ? 结果1 : 结果2;

```dart
oj.dart › main
void main(List<String> args) {
  int score = 61;
  print(score > 60 ? "及格" : "不及格");
}
```

Dart中的switch/case

- 定义：在Dart语言中，如果分支条件很多，且条件是判断相等，可以使用switch case语句

语法：switch(变量) {

case 值1:

逻辑1;

break;

case 值2:

逻辑2;

break;

default:

默认逻辑;

```dart
Run | Debug
void main(List<String> args) {
  int state = 1;
  switch (state) {
    case 1:
      print("待付款");
      break;
    case 2:
      print("待发货");
      break;
    case 3:
      print("待收货");
      break;
    case 4:
      print("待评价");
      break;
    default:
      print("默认值");
  }
}
```

Dart中的循环语句–while

- 定义：在Dart语言中，循环主要使用while循环和for循环

  while循环语法：

```
while(条件) {
  逻辑;
}
```

- 特点：只有条件满足，括号中逻辑一直执行，想要跳出执行使用break或者continue

  注意： break是跳出整个while循环，continue是跳出当前迭代，进入下一次迭代

Dart中的循环语句–while

```dart
Run | Debug
void main(List<String> args) {
  List foods = [
    "第一个包子",
    "第二个包子",
    "第三个包子",
    "第四个包子",
    "第五个包子",
  ];
  int index = 0;
  while (index < foods.length) {
    if (index == 2) {
      index += 1;
      continue; // 跳出此次迭代 继续下个迭代
      // break; // 跳出整个循环
    }
    print(foods[index]);
    index += 1;
  }
}
```

Dart中的循环语句–for

- 定义：在Dart语言中，for循环可以对列表类型进行遍历

  for循环语法：

```
for(int  i =0; i < 10; i++) {
   逻辑;
 }
```

- 特点：只有小括号逻辑满足，大括号逻辑一直执行，想要跳出执行使用break或者continue

  注意： break是跳出整个while循环，continue是跳出当前迭代，进入下一次迭代

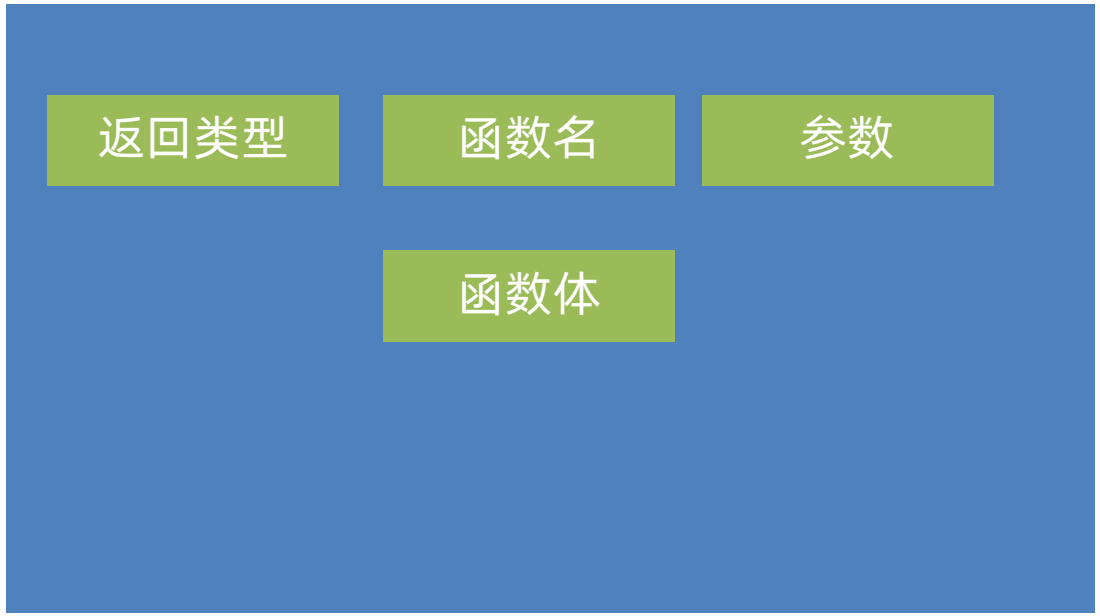
Dart中的循环语句–for

```
Run | Debug
void main(List<String> args) {
  List foods = [
    "第一个包子",
    "第二个包子",
    "第三个包子",
    "第四个包子",
    "第五个包子",
  ];
  for (var i = 0; i < foods.length; i++) {
    if (i == 2) {
      continue; // 跳出此次迭代 继续下个迭代
      // break; // 跳出整个循环
    }
    print(foods[i]);
  }
}
```

Dart的函数

Dart中的函数定义

- 定义：在Dart语言中，函数是代码组合和复用的核心单元
- 场景：应用程序中很多地方需要做两数之和的运算，可以定义函数来组织代码




```
int add (int a, int b) {
  return a + b;
}
```

Dart中的函数的返回值

- 分类：函数返回值分为有返回值和无返回值
- 有返回值： 具体类型 函数名称() {}
- 无返回值： void 函数名称() {}
- 注意： 返回值类型可省略，Dart会自动推断类型为dynamic
- 推荐： 虽然返回值可省略，明确返回值类型是更推荐的编程习惯

```
3  // 带返回值
4  int add(int a, int b) {
5    return a + b;
6  }
7
8  // 无返回值
9  void test() {
10   print("测试无返回值");
11 }
12
13 // 无明确类型，自动推断
14 getValue() {
15   return 1 + 2;
16 }
```

函数的参数–必传参数

- 分类：函数的参数分为必传参数，可选位置参数，可选命名参数
- 特点：必传参数不能为空

函数的参数–可选位置参数

- 特点：可选位置参数必须位于必传参数后面，采用中括号包裹
- 语法：函数名(String a, [String? b, ...])，传递时按照顺序传递
- 适用场景：参数少且顺序固定时
- 默认值：可选参数可以设置默认值

```
Run | Debug
void main(List<String> args) {
  print(combine("1"));
}

String combine(String a, [String? b = "b", String? c = "c"]) {
  return a + (b ?? "默认值b") + (c ?? "默认值c");
}
```

函数的参数–可选命名参数

- 特点：可选命名参数必须位于必传参数后面，采用大括号包裹
- 语法：函数名(String a, { String? b, ...})，传递时按照 参数名:值的方式进行传递，无需关注顺序
- 适用场景：参数多且需明确含义时
- 默认值：可选参数可以设置默认值

```dart
Run | Debug
void main(List<String> args) {
  showPerson("张三");
}

void showPerson(String username, {int? age = 10, String? sex = "男"}) {
  print('姓名:$username, 年龄: $age, 性别: $sex');
}
```

匿名函数

- 特点：可以声明一个没有名称的函数赋值给变量，进行调用
- 语法： Function 变量名 = () { };
- 注意：函数的类型使用Function来声明

```
Run | Debug
1  void main(List<String> args) {
2    onTest(test); // 调用函数 传递匿名函数
3  }
4
5  // 声明匿名函数
6  Function test = () {
7    print("测试");
8  };
9  // 定义一个参数接受一个函数
10 void onTest(Function callBack) {
11   callBack();
12 }
```

箭头函数

- 特点：当函数体只有一行代码时，可以使用箭头函数编写
- 语法：函数名 () => 代码逻辑
- 注意：使用箭头函数可以省略return关键字

```dart
Run | Debug
void main(List<String> args) {
  print(add(1, 2));
}

int add(int a, int b) => a + b;
```

Dart的类

Dart中的类–class

- 定义：Dart语言中，类(class)是面向对象编程的核心，类包含属性和方法来定义对象的行为和状态
- 需求： 定义一个Person类，属性包括姓名、年龄、性别，包括学习的方法
- 定义类语法： class Person { 属性 方法 }
- 实例化对象： Person 变量 = Person();
- 属性和方法： 变量.属性/方法()

```dart
Run | Debug
void main(List<String> args) {
  // 实例化类
  Person p = Person();
  p.name = "老高";
  p.age = 35;
  p.study();
}

// 定义类
class Person {
  String name = "";
  int age = 0;
  String sex = "男";
  void study() {
    print('$name在学习');
  }
}
```

Dart中类的构造函数–默认构造函数

- 定义：实例化对象的时候，使用构造函数直接给对象中的属性赋值
- 常见分类： 默认构造函数、命名构造函数、构造函数的语法糖
- 定义语法：

```
class 类名 {
    类名(可选命名参数) {
    }
}
```

- 实例化语法：

```
Person p = new Person(属性: 值)
```

```dart
| Debug
void main(List<String> args) {
  Person p = Person(name: '张三');
  p.study();
}

class Person {
  String? name;
  int? age;
  String? sex;
  Person({String? name, int? age, String? sex}) {
    this.name = name;
    this.age = age;
    this.sex = sex;
  }
  void study() {
    print('$name在学习');
  }
```

Dart中类的构造函数–命名构造函数

- 定义：构造函数可以采用命名的方式，返回一个实例化对象
- 定义语法：

```
class 类名 {
    类名.构造函数名(可选命名参数) {
    }
}
```

- 实例化语法：

```
Person p = Person.构造函数名(属性: 值)
```

- 注意：默认构造函数和命名构造函数可同时存在

```dart
Run | Debug
void main(List<String> args) {
  Person p = Person.createPerson(name: '张三');
  p.study();
}

class Person {
  String? name;
  int? age;
  String? sex;
  // 命名构造函数
  Person.createPerson({String? name, int? age, String? sex}) {
    this.name = name;
    this.age = age;
    this.sex = sex;
  }

  void study() {
    print('$name在学习');
  }
}
```

Dart中类的构造函数–构造函数语法糖

- 定义：同名构造函数和命名构造函数都支持简写写法
- 语法：

```
class 类名 {
  类名({ this.属性1，this.属性2 });
  或
  类名.命名函数({ this.属性1，this.属性2 });
}
```

```dart
| Debug
void main(List<String> args) {
  Person p = Person.createPerson(name: '张三');
  p.study();
  Person pp = Person(name: '张三');
  pp.study();
}

class Person {
  String? name;
  int? age;
  String? sex;
  // 命名构造函数
  Person({this.name, this.age, this.sex}); // 简写
  Person.createPerson({this.name, this.age, this.sex}); // 命名函数简写

  void study() {
    print('$name在学习');
  }
```

Dart中类的公有属性和私有属性

- 公有属性，提供自身或者其他外部文件和类使用的属性和方法
- 私有属性，仅供自身使用的属性和方法，其他外部文件和类无法访问
- 语法：私有属性以下划线开头， 如_name，其余均为公有属性

```dart
class Person {
  String? _name;
  int? _age;
  String? _sex;
  // 命名构造函数
  Person.createPerson({String? name, int? age, String? sex}) {
    this._name = name;
    this._age = age;
    this._sex = sex;
  }

  void study() {
    print('$_name在学习');
  }
```

```dart
import './类的私有属性.dart';

Run | Debug
void main(List<String> args) {
  Person p = Person.createPerson(name: '张三');
  p._name = ""; // 私有属性无法被调用
}
```

Dart中类的继承

- 定义：继承是拥有父类的属性和方法
- 特点：dart属于单继承，一个类只能拥有一个直接父类， 子类拥有父类所有的属性和方法
- 语法：class 类名 extends 父类
- 重写：子类可通过@override注解重写父类方法，扩展其行为
- 注意：子类不会继承父类构造函数，子类必须通过super关键字调用父类构造函数确保父类正确初始化
- super语法： 子类构造函数(可选命名参数) : super({ 参数 })

```dart
class Parent {
  String? name;
  int? age;
  Parent({this.name, this.age});
  void study() {
    print('$name父类学习');
  }
}
```

```dart
class Child extends Parent {
  Child({String? name, int? age}) : super(name: name, age: age);
  // 重写方法
  @override
  void study() {
    // TODO: implement study
    // super.study();
    print('$name子类学习');
  }
}
```

```dart
Run | Debug
void main(List<String> args) {
  Child c = Child(name: '张三', age: 18);
  c.study();
}
```

Dart中类的多态–继承和方法重写

- 定义：Dart中的类的是指同一操作作用于不同的对象，可以产生不同的执行效果
- 场景：微信和支付宝都遵循同样支付接口，但实现逻辑不同，即同一个支付操作拥有不同的支付效果
- 实现方式：1.继承和方法重写 、2. 抽象类和接口
- 需求：定义一个父类，分别实现微信和支付宝支付类，重写得到不同的支付逻辑

```dart
// 声明基础支付类
class PayBase {
  pay() {
    print("基础支付");
  }
}
```

```dart
// 微信支付类
class WxPay extends PayBase {
  @override
  pay() {
    print("微信支付");
  }
}
```

```dart
// 支付宝支付类
class AliPay extends PayBase {
  @override
  pay() {
    print("支付宝支付");
  }
}
```

```dart
// 调用支付
Run | Debug
void main(List<String> args) {
  PayBase p = WxPay();
  p.pay();
}
```

Dart中类的多态–抽象类和接口实现

- 方式：使用abstract关键字声明一个抽象类(没有实现体)
- 方式：使用implements关键字继承并实现抽象类

```dart
// 定义抽象类
abstract class PayBase {
  void pay();
}
```

```dart
// 微信支付类
class WxPay implements PayBase {
  @override
  void pay() {
    print("微信支付");
  }
}
```

```dart
// 支付宝支付类
class AliPay implements PayBase {
  @override
  void pay() {
    print("支付宝支付");
  }
}
```

```dart
// 调用支付
Run | Debug
void main(List<String> args) {
  PayBase p = WxPay();
  p.pay();
}
```

Dart中类的混入

- 定义：Dart允许在不使用传统继承的情况下，向类中添加新的功能
- 方式：使用mixin关键字定义一个对象
- 方式：使用with关键字将定义的对象混入到当前对象
- 特点：一个类支持with多个mixin，调用优先级遵循“后来居上”原则，即后混入的会覆盖先混入的同名方法
- 需求：让一个学生类和一个老师类都拥有唱歌的方法

```dart
// 定义混入对象
mixin Base {
  void song(String name) {
    print("$name我会唱歌");
  }
}
```

```dart
// 混入Base对象
class Student with Base {
  String? name;
  int? age;
  Student({this.name, this.age});
}

class Teacher with Base {
  String? name;
  int? age;
  Teacher({this.name, this.age});
}
```

```dart
Run | Debug
void main(List<String> args) {
  Student s = Student(name: "小高同学");
  s.song(s.name!);
  Teacher t = Teacher(name: "老张老师");
  t.song(t.name!);
}
```

Dart中泛型

- 定义：Dart允许使用类型参数，限定类型的同时又让类型更加灵活，让代码更加健壮和维护性更强
- 场景：List类型中只想存储String类型怎么办？函数中返回值希望和参数一个类型怎么办？
- 常见分类： 泛型集合、泛型方法、泛型类

```dart
// 定义一个泛型List
List<String> list = [];
list.add("张三");
list.add("李四");
list.add("王五");
print(list);
```

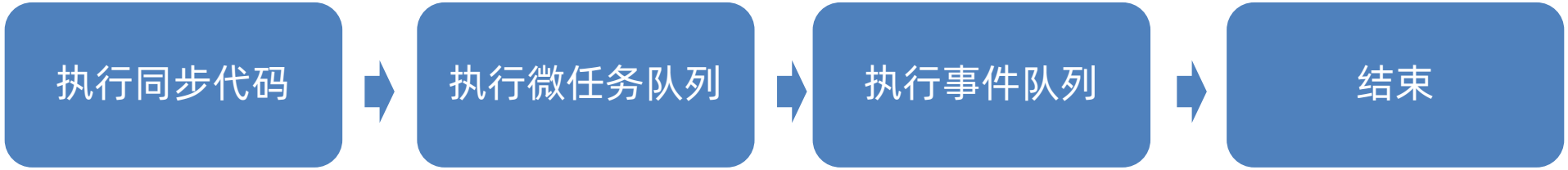
```dart
// 定义一个泛型方法
void printList<T>(List<T> list) {
  for (var i = 0; i < list.length; i++) {
    print(list[i]);
  }
}
```

```dart
// 泛型类
class Student<T> {
  T? name;
  int? age;
  Student({this.name, this.age});
}
```

Dart的异步编程

Dart中异步编程–事件循环

- 介绍：Dart是单线程语言，即同时只能做一件事，遇到耗时任务就会造成程序阻塞，此时需要异步编程
- 定义：Dart采用单线程 + 事件循环的机制完成耗时任务的处理
- 事件循环：



- 微任务队列：Future.microtask()
- 事件队列：Future、Future.delayed()、I/O 操作(文件、网络) 等
- 微任务队列 + 事件队列都属于异步任务

Dart中异步编程–Future

- 介绍：Future代表一个异步操作的最终结果.
- 状态：Uncompleted（等待）、Completed with a value (成功)、Completed with a error(失败)
- 创建： Future(() {})
- 执行成功：不抛出异常-成功状态-then(() {})
- 执行失败： throw Exception()-失败状态-catchError(() {})

```
Run | Debug
1  void main(List<String> args) {
2    // 定义一个Future对象
3    Future<String> f = Future(() {
4      // return "hello world";
5      throw Exception("错误");
6    }); // Future
7    // 调用then方法
8    f.then((value) {
9      print(value);
10   });
11   f.catchError((error) {
12     print(error);
13   });
14 }
```

Dart中异步编程–Future链式调用

- 介绍：Future可以通过链式的方式连续得到异步的结果
- 语法：通过Future().then() 拿到执行成功的结果
- 语法：通过Future().catchError() 拿到执行失败的结果
- 注意：在上一个then返回对象会在下一个then中接收
- 需求：执行三个异步任务，按照顺序排列，最后一次任务抛出异常

```
Run | Debug
1  void main(List<String> args) {
2    // 定义一个Future对象
3    // 调用then方法
4    Future(() {
5      return "hello world";
6    }).then((value) {
7      return Future(() => "test1");
8    }).then((value) {
9      return Future(() => "test2");
10   }).then((value) {
11     return Future(() => "test3");
12   }).then((value) {
13     print(value);
14     throw Exception("错误");
15   }).catchError((error) {
16     print(error.toString());
17   });
18 }
```

Dart中异步编程–Future–async/await

- 介绍：除了通过then/catchError的方式，还可以通过async/await来实现异步编程
- 特点：await 总是等到后面的Future执行成功，才执行下方逻辑，async必须配套await出现
- 语法：

```
函数名 () async {
  try {
    await Future();
    // Future执行成功才执行的逻辑
  }
  catch(error) {
    // 执行失败的逻辑
  }
}
```

await.dart > test

Run | Debug

```dart
void main(List<String> args) {
  test();
}

void test() async {
  try {
    await Future(() {
      // return "测试";
      throw Exception();
    }); // Future
    print("执行成功逻辑");
  } catch (e) {
    print("执行失败逻辑");
  }
}
```