第2章 ArkTS

Outline

- ArkTS语言
  - 历史
  - 基础语法
  - 变量
  - 运算符
  - 语句
  - 方法
  - 类
  - 接口
  - 空安全
  - 模块
  - 类库

历史

历史

- Mozilla创造了JS
- Microsoft创建了TS
- Huawei进一步推出了ArkTS。

JS语言

- JS语言由Mozilla创造，最初主要是为了解决页面中的逻辑交互问题，它和HTML（负责页面内容）、CSS（负责页面布局和样式）共同组成了Web页面/应用开发的基础。
- 随着Web和浏览器的普及，以及Node.js进一步将JS扩展到了浏览器以外的环境，JS语言得到了飞速的发展。
- 在2015年相关的标准组织ECMA发布了一个主要的版本ECMAScript 6（简称ES6），这个版本具备了较为完整的语言能力，包括类（Class）、模块（Module）、相关的语言基础API增强（Map/Set等）、箭头函数（Arrow Function）等。
- 从2015年开始，ECMA每年都会发布一个标准版本，比如ES2016/ES2017/ES2018等，JS语言越来越成熟。

Web前端框架

- 为了提升应用的开发效率，相应的JS前端框架也不断地涌现出来。其中比较典型的有Facebook发起的React.js，以及个人开发者尤雨溪发起的Vue.js。React和Vue的主要出发点都是将响应式编程的能力引入到应用开发中，实现数据和界面内容的自动关联处理。

- 具体的实现方式上，React对JS做了一些扩展，引入了JSX（JavaScript XML）语法，可以将HTML的内容统一表示成JS来处理；Vue则是通过扩展的模板语法（Template）的方式来处理。

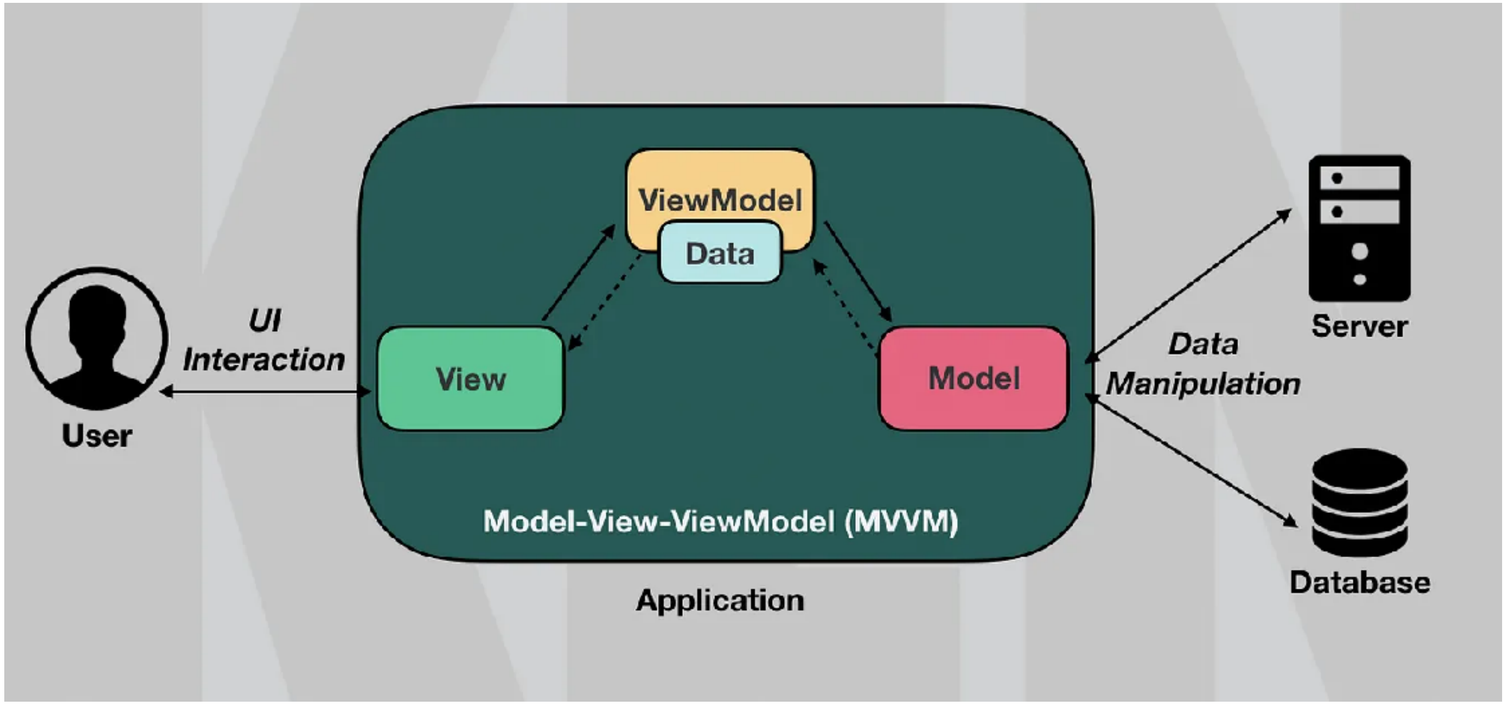
- 从运行时的维度来看，基于React以及Vue的应用都可运行在Web引擎上。为了进一步提升相应的性能体验，2015年Facebook在React基础上推出了React Native, 在渲染架构上没有采用传统的Web引擎渲染路径，而是桥接到相应OS平台的原生UI组件上。2019年Facebook引入全新实现的JS引擎Hermes，并推出一系列架构改进来进一步提升React Native的性能体验。2016年阿里巴巴开源的Weex则是基于Vue做了一些类似的改进，也是采用了桥接到原生UI组件的渲染路径。

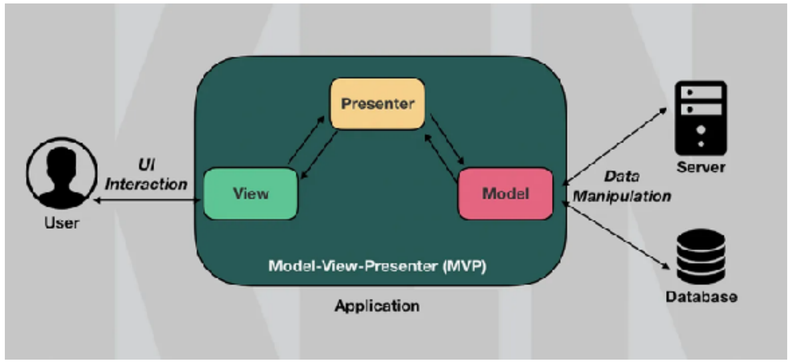
React

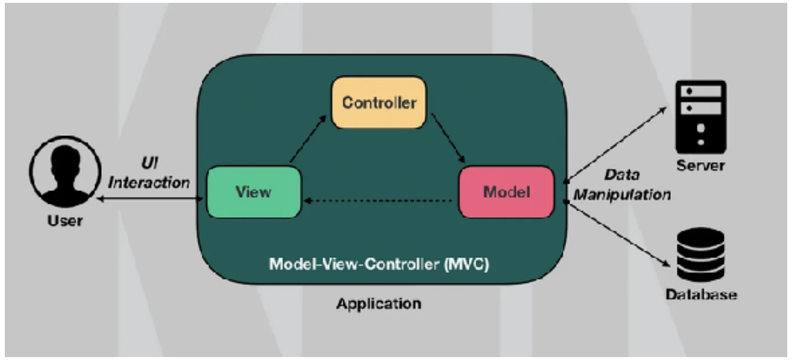
```
<div id="id01">Hello World!</div>

<script type="text/babel">
const name = 'John Doe';
ReactDOM.render(
    <h1>Hello {name}!</h1>,
    document.getElementById('id01'));
</script>
```

Vue

```
<div id="app">
    {{ message }}
</div>

<script>
var myObject = new Vue({
    el: '#app',
    data: {message: 'Hello Vue!'}
})

function myFunction() {
    myObject.message = "John Doe";
}
</script>
```


ViewModel
Data
UI
Interaction
User
View
Model
Data
Manipulation
Server
Database
Model-View-ViewModel (MVVM)
Application


Presenter
UI
Interaction
User
View
Model
Data
Manipulation
Server
Database
Model-View-Presenter (MVP)
Application


Controller
UI
Interaction
User
View
Model
Data
Manipulation
Server
Database
Model-View-Controller (MVC)
Application

```swift
// SwiftUI
struct MainView: View {
    //Variable declaration
    var count: Int = 0
    var body: some View {
        Text("SwiftUI")
            .font(.system(size: 15, weight: .bold))
        }
}

// Jetpack Compose
@Composable private fun MainView() {
    // Variable declaration
    var count: Int = 0

    Text(
        text = "Jetpack Compose",
        fontWeight = FontWeight.Bold,fontSize = 15.sp
    )
}
```

客户端UI框架：Swift UI vs Jetpack Compose

TS语言

- TS主要从以下几个方面做了进一步的增强：
  - 引入了类型系统，并提供了类型检查以及类型自动推导能力，可以进行编译时错误检查，有效的提升了代码的规范性以及错误检测范围和效率。
  - 在类型系统基础上，引入了声明文件（Declaration Files）来管理接口或其他自定义类型。声明文件一般是以d.ts的形式来定义模块中的接口，这些接口和具体的实现做了相应的分离，有助于各模块之间的分工协作。
  - 另外，TS通过接口，泛型（Generics）等相关特性的支持，进一步增强了设计复杂的框架所需的扩展以及复用能力。
- 在工具层面，TS也有相应的编辑器、编译器、IDE（Integrated Development Environment）插件等相关的工具，来进一步提升开发效率。
- TS在兼容JS生态方面也做了较好的平衡，TS应用通过相应编译器可以编译出纯JS应用，可以在标准的JS引擎上运行。同时，TS定位为JS的超集，即JS应用也是合法的TS应用。此外，在标准层面上，TS兼容ECMA的相应标准，并维护那些还未成为ECMA标准的新特性。

ArkTS语言

ArkTS是HarmonyOS优选的主力应用开发语言。ArkTS基于TypeScript（简称TS）语言扩展而来，是TS的超集。

- ArkTS继承了TS的所有特性。
- 当前，ArkTS在TS基础上主要扩展了声明式UI能力，让开发者以更简洁、更自然的方式开发高性能应用。当前扩展的声明式UI包括如下特性。
  - 基本UI描述：ArkTS定义了各种装饰器、自定义组件、UI描述机制，再配合UI开发框架中的UI内置组件、事件方法、属性方法等共同构成了UI开发的主体。
  - 状态管理：ArkTS提供了多维度的状态管理机制，在UI开发框架中，和UI相关联的数据，不仅可以在组件内使用，还可以在不同组件层级间传递，比如父子组件之间、爷孙组件之间，也可以是全局范围内的传递，还可以是跨设备传递。另外，从数据的传递形式来看，可分为只读的单向传递和可变更的双向传递。开发者可以灵活的利用这些能力来实现数据和 UI 的联动。
  - 动态构建UI元素：ArkTS提供了动态构建UI元素的能力，不仅可自定义组件内部的UI结构，还可复用组件样式，扩展原生组件。
  - 渲染控制：ArkTS提供了渲染控制的能力。条件渲染可根据应用的不同状态，渲染对应状态下的部分内容。循环渲染可从数据源中迭代获取数据，并在每次迭代过程中创建相应的组件。
  - 使用限制与扩展：ArkTS在使用过程中存在限制与约束，同时也扩展了双向绑定等能力。
- 未来，ArkTS会结合应用开发/运行的需求持续演进，逐步提供并行和并发能力增强、类型系统增强、分布式开发范式等更多特性。

ArkTS vs TypeScript

	TS	ArkTS
静态检查	可配置：开发人员可以使其更严格或更放松。 风险：如果我们错误配置了类型检查器，那么静态类型带来的好处都将丧失	完全静态：不能放松或关闭类型检查。 类型始终存在于代码中
编译	编译为JavaScript 编译后类型信息丢失	编译为字节码（特殊的运行时友好 执行格式，如JVM或.NET） 类型信息在编译后得到保留
运行	可以在任何能够运行JavaScript的引擎上运行 （V8、JavaScriptCore等） 引擎在运行JavaScript代码期间执行所有必要的运行时类型检查	在ArkRuntime上运行 由于全静态类型，运行时类型检查的数量大大减少（与Java或C#的水 平相同）

基础语法

变量声明 常量声明

- 以关键字let开头的声明引入变量，该变量在程序执行期间可以具有不同的值。
  - let hi: string = “hello"
  - hi = "hello, world”
- 以关键字const开头的声明引入只读常量，该常量只能被赋值一次
  - const hello: string = “hello"
- ArkTS规范中列举了所有允许自动推断类型的场景。以下示例中，两条声明语句都是有效的，两个变量都是string类型：
  - let hi1: string = "hello"
  - let hi2 = "hello, world"

基本类型

- Number： 任何整数和浮点数都可以被赋给此类型的变量。
- Boolean：由 true和false两个逻辑值组成。
- String：代表字符序列；可以使用 转义字符来表示字符。
- Void： 用于指定函数没有返回值。
- Object： 所有引用类型的基类型。
- Array： 由可赋值给数组声明中指 定的元素类型的数据组成的对象。
- Enum： 枚举类型，是预先定义的 一组命名值的值类型。
- Union ： 联合类型。
- Aliases： 匿名类型。

Number

ArkTS提供number和Number类型，任何整数和浮点数都可以被赋给此类型的变量。

数字字面量包括整数字面量和十进制浮点数字面量。

整数字面量包括以下类别：

- 由数字序列组成的十进制整数。例如：0、117、-345
- 以0x（或0X）开头的十六进制整数，可以包含数字（0-9）和字母a-f或A-F。例如：0x1123、0x00111、-0xF1A7
- 以0o（或0O）开头的八进制整数，只能包含数字（0-7）。例如：0o777
- 以0b（或0B）开头的二进制整数，只能包含数字0和1。例如：0b11、0b0011、-0b11

浮点字面量包括以下：

- 十进制整数，可为有符号数（即，前缀为“+”或“-”）；
- 小数点（“.”）
- 小数部分（由十进制数字字符串表示）
- 以“e”或“E”开头的指数部分，后跟有符号（即，前缀为“+”或“-”）或无符号整数。

Number

- let n1 = 3.14;
- let n2 = 3.141592;
- let n3 = .5;
- let n4 = 1e10;

- function factorial(n: number): number {
- if (n <= 1) {
- return 1;
- }
- return n * factorial(n - 1);
- }

Boolean

- let isDone: boolean = false;

- // ...

- if (isDone) {
- console.log ('Done!');
- }

String

- string代表字符序列；可以使用转义字符来表示字符。
- 字符串字面量由单引号（'）或双引号（"）之间括起来的零个或多个字符组成。字符串字面量还有一特殊形式，是用反向单引号（`）括起来的模板字面量。
- let s1 = 'Hello, world!\n';
- let s2 = 'this is a string';
- let a = 'Success';
- let s3 = `The result is ${a}`;

Void

- void类型用于指定函数没有返回值。 此类型只有一个值，同样是void。由于void是引用类型，因此它可以用于泛型类型参数。

- class Class<T> {
- //...
- }
- let instance: Class <void>

Object & Array

- Object类型
  - Object类型是所有引用类型的基类型。任何值，包括基本类型的值（它们会被自动装箱），都可以直接被赋给Object类型的变量。
- Array类型
  - array，即数组，是由可赋值给数组声明中指定的元素类型的数据组成的对象。 数组可由数组复合字面量（即用方括号括起来的零个或多个表达式的列表，其中每个表达式为数组中的一个元素）来赋值。数组的长度由数组中元素的个数来确定。数组中第一个元素的索引为0。
  - 以下示例将创建包含三个元素的数组：
  - let names: string[] = ['Alice', 'Bob', 'Carol'];

Enum

- enum类型，又称枚举类型，是预先定义的一组命名值的值类型，其中命名值又称为枚举常量。 使用枚举常量时必须以枚举类型名称为前缀。

- enum ColorSet { Red, Green, Blue }
- let c: ColorSet = ColorSet.Red;

- 常量表达式可以用于显式设置枚举常量的值。

- enum ColorSet { White = 0xFF, Grey = 0x7F, Black = 0x00 }
- let c: ColorSet = ColorSet.Black;

Union类型

- union类型，即联合类型，是由多个类型组合成的引用类型。联合类型包含了变量可能的所有类型。

- class Cat {
- // ...
- }
- class Dog {
- // ...
- }
- class Frog {
- // ...
- }
- type Animal = Cat | Dog | Frog | number
- // Cat、Dog、Frog是一些类型（类或接口）

- let animal: Animal = new Cat();
- animal = new Frog();
- animal = 42;
- // 可以将类型为联合类型的变量赋值为任何组成类型的有效值

Union类型

- 可以用不同的机制获取联合类型中特定类型的值。
- class Cat { sleep () {}; meow () {} }
- class Dog { sleep () {}; bark () {} }
- class Frog { sleep () {}; leap () {} }
- type Animal = Cat | Dog | Frog | number
- let animal: Animal = new Frog();
- if (animal instanceof Frog) {
- let frog: Frog = animal as Frog; // animal在这里是Frog类型
- animal.leap();
- frog.leap();
- // 结果：青蛙跳了两次
- }
- animal.sleep (); // 任何动物都可以睡觉

Aliases类型

- Aliases类型为匿名类型（数组、函数、对象字面量或联合类型）提供名称，或为已有类型提供替代名称。
- type Matrix = number[][];
- type Handler = (s: string, no: number) => string;
- type Predicate <T> = (x: T) => Boolean;
- type NullableObject = Object | null;

赋值运算符

- 赋值运算符=，使用方式如x=y。
- 复合赋值运算符将赋值与运算符组合在一起，其中x op = y等于x = x op y。
- 复合赋值运算符列举如下：+=、-=、*=、/=、%=、<<=、>>=、>>>=、&=、|=、^=。

比较运算符

- ==
- 如果两个操作数相等，则返回true。
- !=
- 如果两个操作数不相等，则返回true。
- >
- 如果左操作数大于右操作数，则返回true。
- >=
- 如果左操作数大于或等于右操作数，则返回true。
- <
- 如果左操作数小于右操作数，则返回true。
- <=
- 如果左操作数小于或等于右操作数，则返回true。

一元、二元、位、逻辑

一元运算符为`-`、`+`、`--`、`++`。

二元运算符列举如下：

运算符	说明
`+`	加法
`-`	减法
`*`	乘法
`/`	除法
`%`	除法后余数

运算符	说明
`a & b`	按位与：如果两个操作数的对应位都为1，则将这个位设置为1，否则设置为0。
`a | b`	按位或：如果两个操作数的相应位中至少有一个为1，则将这个位设置为1，否则设置为0。
`a ^ b`	按位异或：如果两个操作数的对应位不同，则将这个位设置为1，否则设置为0。
`~ a`	按位非：反转操作数的位。
`a << b`	左移：将a的二进制表示向左移b位。
`a >> b`	算术右移：将a的二进制表示向右移b位，带符号扩展。
`a >>> b`	逻辑右移：将a的二进制表示向右移b位，左边补0。

运算符	说明
`a && b`	逻辑与
`a ‖ b`	逻辑或
`! a`	逻辑非

If 语句

- if语句用于需要根据逻辑条件执行不同语句的场景。当逻辑条件为真时，执行对应的一组语句，否则执行另一组语句（如果有的话）。 else部分也可能包含if语句。
- if语句如下所示：

- if (condition1) {
- // 语句1
- } else if (condition2) {
- // 语句2
- } else {
- // else语句
- }

- 条件表达式可以是任何类型。但是对于boolean以外的类型，会进行隐式类型转换：

- let s1 = 'Hello';
- if (s1) {
- console.log(s1); // 打印“Hello”
- }

- let s2 = 'World';
- if (s2.length != 0) {
- console.log(s2); // 打印“World”
- }

switch

- 使用switch语句来执行与switch表达式值匹配的代码块。
- switch语句如下所示：

- switch (expression) {
- case label1: // 如果label1匹配，则执行
- // ...
- // 语句1
- // ...
- break; // 可省略
- case label2:
- case label3: // 如果label2或label3匹配，则执行
- // ...
- // 语句23
- // ...
- break; // 可省略
- default:
- // 默认语句
- }
- 如果switch表达式的值等于某个label的值，则执行相应的语句。
- 如果没有任何一个label值与表达式值相匹配，并且switch具有default子句，那么程序会执行default子句对应的代码块。
- break语句（可选的）允许跳出switch语句并继续执行switch语句之后的语句。
- 如果没有break语句，则执行switch中的下一个label对应的代码块。

条件表达式

- 条件表达式由第一个表达式的布尔值来决定返回其它两个表达式中的哪一个。
- 示例如下：

- condition ? expression1 : expression2
- typescript

- 如果condition的为真值（转换后为true的值），则使用expression1作为该表达式的结果；否则，使用expression2。
- 示例：

- let isValid = Math.random() > 0.5 ? true : false;
- let message = isValid ? 'Valid' : 'Failed';

for语句

- for语句会被重复执行，直到循环退出语句值为false。
- for语句如下所示：
- for ([init]; [condition]; [update]) {
- statements
- }
- for语句的执行流程如下：

- 1、 执行init表达式（如有）。此表达式通常初始化一个或多个循环计数器。
  2、 计算condition。如果它为真值（转换后为true的值），则执行循环主体的语句。如果它为假值（转换后为false的值），则for循环终止。
  3、 执行循环主体的语句。
  4、 如果有update表达式，则执行该表达式。
  5、 回到步骤2。
- 示例：
- let sum = 0;
- for (let i = 0; i < 10; i += 2) {
- sum += i;
- }

for of

- 使用for-of语句可遍历数组或字符串。示例如下：

- for (forVar of expression) {
- statements
- }

- 示例：
- for (let ch of 'a string object') {
- /* process ch */
- }

While

- 只要condition为真值（转换后为true的值），while语句就会执行statements语句。示例如下：

- while (condition) {
- statements
- }
- 示例：

- let n = 0;
- let x = 0;
- while (n < 3) {
- n++;
- x += n;
- }

Do while

- Do-while语句
- 如果condition的值为真值（转换后为true的值），那么statements语句会重复执行。示例如下：

- do {
- statements
- } while (condition)

- 示例：

- let i = 0;
- do {
- i += 1;
- } while (i < 10)

break

- 使用break语句可以终止循环语句或switch。
- 示例：

```
let x = 0;
while (true) {
  x++;
  if (x > 5) {
    break;
  }
}
```

- 如果break语句后带有标识符，则将控制流转移到该标识符所包含的语句块之外。
- 示例：

```
let x = 1
label: while (true) {
  switch (x) {
    case 1:
      // statements
      break label // 中断while语句
  }
}
```

Continue

- Continue语句
- continue语句会停止当前循环迭代的执行，并将控制传递给下一个迭代。
- 示例：

- let sum = 0;
- for (let x = 0; x < 100; x++) {
- if (x % 2 == 0) {
- continue
- }
- sum += x;
- }
-

Throw和Try语句

- throw语句用于抛出异常或错误：
- throw new Error('this error')
- try语句用于捕获和处理异常或错误：
- try {
- // 可能发生异常的语句块
- } catch (e) {
- // 异常处理
- }
- 下面的示例中throw和try语句用于处理除数为0的错误：
- class ZeroDivisor extends Error {}
- function divide (a: number, b: number): number{
- if (b == 0) throw new ZeroDivisor();
- return a / b;
- }
- function process (a: number, b: number) {
- try {
- let res = divide(a, b);
- console.log('result: ' + res);
- } catch (x) {
- console.log('some error');
- }
- }
-

finally语句

```
function processData(s: string) {
  let error: Error | null = null;

  try {
    console.log('Data processed: ' + s);
    // ...
    // 可能发生异常的语句
    // ...
  } catch (e) {
    error = e as Error;
    // ...
    // 异常处理
    // ...
  } finally {
    if (error != null) {
      console.log(`Error caught: input='${s}', message='${error.message}'`);
    }
  }
}
```

函数

一个函数包含其名称、参数列表、返回类型和函数体。

以下示例是一个简单的函数：

```
function add(x:string):string{
    let z:string = x
    return z
}
```

函数声明

- 可选参数 function add(x?: string)： String
- 默认参数 function add(x: string=“hello”)： String
- Rest参数 function add(...xs: string[])： String

- 缺省返回值
  - 从函数体内推断出函数返回类型
    - // 显式指定返回类型
    - function foo(): string { return "foo" }
  - 不需要返回值的函数
    - // 推断返回类型为string
    - function goo() { return "goo" }
    - function hi1() { console.log("hi") }
    - function hi2(): void { console.log("hi")}

可选参数

- function hello(name?: string) {
- if (name == undefined) {
- console.log('Hello!');
- } else {
- console.log(`Hello, ${name}!`);
- }
- }

可选参数

- function multiply(n: number, coeff: number = 2): number {
- return n * coeff;
- }
- multiply(2); // 返回2*2
- multiply(2, 3); // 返回2*3

Rest参数

- 函数的最后一个参数可以是rest参数。使用rest参数时，允许函数或方法接受任意数量的实参。
- function sum(...numbers: number[]): number {
- let res = 0;
- for (let n of numbers)
- res += n;
- return res;
- }

- sum() // 返回0
- sum(1, 2, 3) // 返回6

函数的调用

- 函数中定义的变量和其他实例仅可以在函数内部访问，不能从外部访问。
- 如果函数中定义的变量与外部作用域中已有实例同名，则函数内的局部变量定义将覆盖外部定义。

- 调用函数以执行其函数体，实参值会赋值给函数的形参。
- 如果函数定义如下：

- function join(x: string, y: string): string {
- let z: string = `${x} ${y}`;
- return z;
- }

- 则此函数的调用需要包含两个string类型的参数：

- let x = join('hello', 'world');
- console.log(x);

函数类型

- 函数类型通常用于定义回调：

- type trigFunc = (x: number) => number // 这是一个函数类型

- function do_action(f: trigFunc) {
- f(3.141592653589); // 调用函数
- }

- do_action(Math.sin); // 将函数作为参数传入

箭头函数或Lambda函数

- 函数可以定义为箭头函数，例如：

- let sum = (x: number, y: number): number => {
- return x + y;
- }

- 箭头函数的返回类型可以省略；省略时，返回类型通过函数体推断。
- 表达式可以指定为箭头函数，使表达更简短，因此以下两种表达方式是等价的：

- let sum1 = (x: number, y: number) => { return x + y; }
- let sum2 = (x: number, y: number) => x + y

闭包

- 闭包是由函数及声明该函数的环境组合而成的。该环境包含了这个闭包创建时作用域内的任何局部变量。
- 在下例中，z是执行f时创建的g箭头函数实例的引用。g的实例维持了对它的环境的引用（变量count存在其中）。因此，当z被调用时，变量count仍可用。

- function f(): () => number {
- let count = 0;
- let g = (): number => { count++; return count; };
- return g;
- }

- let z = f();
- z(); // 返回：1
- z(); // 返回：2

闭包

- 在 ArkTS（ArkUI 的 TypeScript 编程语言）中，闭包的概念与普通 TypeScript 或 JavaScript 中的闭包概念非常相似。闭包是一种允许内部函数访问其外部函数作用域中的变量的机制。即使外部函数已经返回，内部函数仍然能够“记住”并使用这些变量。

- 闭包的核心思想是函数在定义时会捕获其词法环境，也就是它定义时所处的上下文，这样即使外部函数已经执行完毕，内部函数仍然能保持对外部函数中的变量的引用。这在状态保持、函数工厂、事件处理等场景中非常有用。

实践处理

- function createClickHandler(message: string): () => void {
- return (): void => {
- console.log(message);
- };
- }

- let clickHandler = createClickHandler("Button clicked!");
- // 绑定事件处理器到按钮点击事件
- button.addEventListener('click', clickHandler);
-

函数工厂

- function createMultiplier(factor: number): (number) => number {
- return (value: number): number => {
- return value * factor;
- };
- }

- let double = createMultiplier(2);
- let triple = createMultiplier(3);

- console.log(double(5)); // 输出: 10
- console.log(triple(5)); // 输出: 15

私有变量

```
function createBankAccount(initialBalance: number) {
  let balance = initialBalance;

  return {
    deposit(amount: number): void {
      balance += amount;
    },
    withdraw(amount: number): void {
      if (amount <= balance) {
        balance -= amount;
      } else {
        console.log("Insufficient funds");
      }
    },
    getBalance(): number {
      return balance;
    }
  };
}

let account = createBankAccount(100);
console.log(account.getBalance()); // 输出: 100
account.deposit(50);
console.log(account.getBalance()); // 输出: 150
account.withdraw(70);
console.log(account.getBalance()); // 输出: 80
```

延迟执行

- function delayedGreeting(name: string, delay: number): () => void {
- return (): void => {
- setTimeout(() => {
- console.log(`Hello, ${name}!`);
- }, delay);
- };
- }

- let greet = delayedGreeting("Alice", 2000);
- greet(); // 在 2 秒后输出: "Hello, Alice!"
-

函数的重载（Overloading）

- 我们可以通过编写重载，指定函数的不同调用方式。具体方法为，为同一个函数写入多个同名但签名不同的函数头，函数实现紧随其后。

- function foo(x: number): void;　　/* 第一个函数定义 */
- function foo(x: string): void;　　/* 第二个函数定义 */
- function foo(x: number | string): void {　/* 函数实现 */
- }

- foo(123);　　// OK，使用第一个定义
- foo('aa'); // OK，使用第二个定义

- 不允许重载函数有相同的名字以及参数列表，否则将会编译报错。

类

- 类声明引入一个新类型，并定义其字段、方法和构造函数。
- 在以下示例中，定义了Person类，该类具有字段name和surname、构造函数和方法fullName：

- class Person {
- name: string = ''
- surname: string = ''
- constructor (n: string, sn: string) {
- this.name = n;
- this.surname = sn;
- }
- fullName(): string {
- return this.name + ' ' + this.surname;
- }
- }

- 定义类后，可以使用关键字new创建实例：

- let p = new Person('John', 'Smith');
- console.log(p.fullName());
- typescript

- 或者，可以使用对象字面量创建实例：
- class Point {
- x: number = 0
- y: number = 0
- }
- let p: Point = {x: 42, y: 42};
-

实例字段

- 实例字段存在于类的每个实例上。每个实例都有自己的实例字段集合。
- 要访问实例字段，需要使用类的实例。

```
class Person {
  name: string = ''
  age: number = 0
  constructor(n: string, a: number) {
    this.name = n;
    this.age = a;
  }
  getName(): string {
    return this.name;
  }
}
```

```
let p1 = new Person('Alice', 25);
p1.name;
let p2 = new Person('Bob', 28);
p2.getName();
```

静态字段

- 使用关键字static将字段声明为静态。静态字段属于类本身，类的所有实例共享一个静态字段。
- 要访问静态字段，需要使用类名：

- class Person {
- static numberOfPersons = 0
- constructor() {
- // ...
- Person.numberOfPersons++;
- // ...
- }
- }

- Person.numberOfPersons;

字段初始化

- 为了减少运行时的错误和获得更好的执行性能， ArkTS要求所有字段在声明时或者构造函数中显式初始化。这和标准TS中的strictPropertyInitialization模式一样。
- 以下代码是在ArkTS中不合法的代码。

- class Person {
- name: string // undefined
-
- setName (n:string): void {
- this.name = n;
- }
-

- getName(): string {
- // 开发者使用"string"作为返回类型，这隐藏了name可能为"undefined"的事实。
- // 更合适的做法是将返回类型标注为"string | undefined"，以告诉开发者这个API所有可能的返回值。
- return this.name;
- }
- }

- let jack = new Person();
- // 假设代码中没有对name赋值，例如调用"jack.setName('Jack')"
- jack.getName().length; // 运行时异常：name is undefined

正确写法

- class Person {
- **name: string = ''**
-
- setName(n:string): void {
- this.name = n;
- }
-
- // 类型为'string'，不可能为"null"或者"undefined"
- getName(): string {
- return this.name;
- }
- }
-

- let jack = new Person();
- // 假设代码中没有对name赋值，例如调用"jack.setName('Jack')"
- jack.getName().length; // 0, 没有运行时异常

正确写法

- class Person {
- name?: string // 可能为`undefined`
- setName(n:string): void {
- this.name = n;
- }
- // 编译时错误：name可以是"undefined"，所以将这个API的返回值类型标记为string
- getNameWrong(): string {
- return this.name;
- }
- getName(): string | undefined { // 返回类型匹配name的类型
- return this.name;
- }
- }
- let jack = new Person();
- // 假设代码中没有对name赋值，例如调用"jack.setName('Jack')"
- // 编译时错误：编译器认为下一行代码有可能会访问undefined的属性，报错
- jack.getName().length; // 编译失败
- jack.getName()?.length; // 编译成功，没有运行时错误

getter setter

- setter和getter可用于提供对对象属性的受控访问。
- 在以下示例中，setter用于禁止将age属性设置为无效值：

- class Person {
- name: string = ''
- private _age: number = 0
- get age(): number { return this._age; }
- set age(x: number) {
- if (x < 0) {
- throw Error('Invalid age argument');
- }
- this._age = x;
- }
- }

- let p = new Person();
- p.age; // 输出0
- p.age = -42; // 设置无效age值会抛出错误

- 在类中可以定义getter或者setter。

实例方法

- 以下示例说明了实例方法的工作原理。
- calculateArea方法通过将高度乘以宽度来计算矩形的面积：

- class RectangleSize {
- private height: number = 0
- private width: number = 0
- constructor(height: number, width: number) {
- // ...
- }
- calculateArea(): number {
- return this.height * this.width;
- }
- }
- 必须通过类的实例调用实例方法：

- let square = new RectangleSize(10, 10);
- square.calculateArea(); // 输出：100

静态方法

- 使用关键字static将方法声明为静态。静态方法属于类本身，只能访问静态字段。
- 静态方法定义了类作为一个整体的公共行为。
- 必须通过类名调用静态方法：

```
class Cl {
  static staticMethod(): string {
    return 'this is a static method.';
  }
}
console.log(Cl.staticMethod());
```

继承

- 一个类可以继承另一个类（称为基类），并使用以下语法实现多个接口：

- class [extends BaseClassName] [implements listOfInterfaces] {
- // ...
- }

- 继承类继承基类的字段和方法，但不继承构造函数。继承类可以新增定义字段和方法，也可以覆盖其基类定义的方法。
- 基类也称为“父类”或“超类”。继承类也称为“派生类”或“子类”。
- 示例：

- class Person {
- name: string = ''
- private _age = 0
- get age(): number {
- return this._age;
- }
- }
- class Employee extends Person {
- salary: number = 0
- calculateTaxes(): number {
- return this.salary * 0.42;
- }
- }

实现接口

- 包含implements子句的类必须实现列出的接口中定义的所有方法，但使用默认实现定义的方法除外。

```
interface DateInterface {
  now(): string;
}
class MyDate implements DateInterface {
  now(): string {
    // 在此实现
    return 'now is now';
  }
}
```

父类访问

- 关键字super可用于访问父类的实例字段、实例方法和构造函数。在实现子类功能时，可以通过该关键字从父类中获取所需接口：

```
class RectangleSize {
  protected height: number = 0
  protected width: number = 0

  constructor (h: number, w: number) {
    this.height = h;
    this.width = w;
  }

  draw() {
    /* 绘制边界 */
  }
}
class FilledRectangle extends RectangleSize {
  color = ''
  constructor (h: number, w: number, c: string) {
    super(h, w); // 父类构造函数的调用
    this.color = c;
  }

  draw() {
    super.draw(); // 父类方法的调用
    // super.height -可在此处使用
    /* 填充矩形 */
  }
}
```

方法重写（Overriding）

- 子类可以重写其父类中定义的方法的实现。重写的方法必须具有与原始方法相同的参数类型和相同或派生的返回类型。

- class RectangleSize {
- // ...
- area(): number {
- // 实现
- return 0;
- }
- }
- class Square extends RectangleSize {
- private side: number = 0
- area(): number {
- return this.side * this.side;
- }
- }

方法重载（Overloading）

- 通过重载签名，指定方法的不同调用。具体方法为，为同一个方法写入多个同名但签名不同的方法头，方法实现紧随其后。

- class C {
- foo(x: number): void; /* 第一个签名 */
- foo(x: string): void; /* 第二个签名 */
- foo(x: number | string): void { /* 实现签名 */
- }
- }
- let c = new C();
- c.foo(123); // OK，使用第一个签名
- c.foo('aa'); // OK，使用第二个签名

- 如果两个重载签名的名称和参数列表均相同，则为错误。

构造函数

- 类声明可以包含用于初始化对象状态的构造函数。
- 构造函数定义如下：
- constructor ([parameters]) {
- // ...
- }
- 如果未定义构造函数，则会自动创建具有空参数列表的默认构造函数，例如：
- class Point {
- x: number = 0
- y: number = 0
- }
- let p = new Point();
- 在这种情况下，默认构造函数使用字段类型的默认值来初始化实例中的字段。

派生类的构造函数

- 构造函数函数体的第一条语句可以使用关键字super来显式调用直接父类的构造函数。

- class RectangleSize {
- constructor(width: number, height: number) {
- // ...
- }
- }
- class Square extends RectangleSize {
- constructor(side: number) {
- super(side, side);
- }
- }

构造函数重载签名

- 我们可以通过编写重载签名，指定构造函数的不同调用方式。具体方法为，为同一个构造函数写入多个同名但签名不同的构造函数头，构造函数实现紧随其后。

- class C {
- constructor(x: number) /* 第一个签名 */
- constructor(x: string) /* 第二个签名 */
- constructor(x: number | string) { /* 实现签名 */
- }
- }
- let c1 = new C(123); // OK，使用第一个签名
- let c2 = new C('abc'); // OK，使用第二个签名

可见性

- 可见性修饰符
- 类的方法和属性都可以使用可见性修饰符。
- 可见性修饰符包括：private、protected和public。默认可见性为public。

对象字面量

- 对象字面量是一个表达式，可用于创建类实例并提供一些初始值。它在某些情况下更方便，可以用来代替new表达式。
- 对象字面量的表示方式是：封闭在花括号对({})中的'属性名：值'的列表。

- class C {
- n: number = 0
- s: string = ''
- }

- let c: C = {n: 42, s: 'foo'};

- ArkTS是静态类型语言，如上述示例所示，对象字面量只能在可以推导出该字面量类型的上下文中使用。其他正确的例子：

- class C {
- n: number = 0
- s: string = ''
- }
- function foo(c: C) {}

- let c: C
- c = {n: 42, s: 'foo'}; // 使用变量的类型
- foo({n: 42, s: 'foo'}); // 使用参数的类型

- function bar(): C {
- return {n: 42, s: 'foo'}; // 使用返回类型
- }

- 也可以在数组元素类型或类字段类型中使用：

- class C {
- n: number = 0
- s: string = ''
- }
- let cc: C[] = [{n: 1, s: 'a'}, {n: 2, s: 'b'}];
-

Record类型的对象字面量

- 泛型Record<K, V>用于将类型（键类型）的属性映射到另一个类型（值类型）。常用对象字面量来初始化该类型的值：

- let map: Record<string, number> = {
- 'John': 25,
- 'Mary': 21,
- }

- map['John']; // 25

- 类型K可以是字符串类型或数值类型，而V可以是任何类型。

- interface PersonInfo {
- age: number
- salary: number
- }
- let map: Record<string, PersonInfo> = {
- 'John': { age: 25, salary: 10},
- 'Mary': { age: 21, salary: 20}
- }

接口

- 接口声明引入新类型。接口是定义代码协定的常见方式。
- 任何一个类的实例只要实现了特定接口，就可以通过该接口实现多态。
- 接口通常包含属性和方法的声明
- 示例：

- interface Style {
- color: string // 属性
- }
- interface AreaSize {
- calculateAreaSize(): number // 方法的声明
- someMethod(): void; // 方法的声明
- }
- 实现接口的类示例：

- // 接口：
- interface AreaSize {
- calculateAreaSize(): number // 方法的声明
- someMethod(): void; // 方法的声明
- }

- // 实现：
- class RectangleSize implements AreaSize {
- private width: number = 0
- private height: number = 0
- someMethod(): void {
- console.log('someMethod called');
- }
- calculateAreaSize(): number {
- this.someMethod(); // 调用另一个方法并返回结果
- return this.width * this.height;
- }
- }
-

接口属性

- 接口属性可以是字段、getter、setter或getter和setter组合的形式。
- 属性字段只是getter/setter对的便捷写法。以下表达方式是等价的：

- interface Style {
- color: string
- }
- interface Style {
- get color(): string
- set color(x: string)
- }
- 实现接口的类也可以使用以下两种方式：

- interface Style {
- color: string
- }

- class StyledRectangle implements Style {
- color: string = ''
- }

- interface Style {
- color: string
- }

- class StyledRectangle implements Style {
- private _color: string = ''
- get color(): string { return this._color; }
- set color(x: string) { this._color = x; }
- }
-

泛型类和接口

- 类和接口可以定义为泛型，将参数添加到类型定义中，如以下示例中的类型参数Element：

- class CustomStack<Element> {
- public push(e: Element):void {
- // ...
- }
- }

- 要使用类型CustomStack，必须为每个类型参数指定类型实参：

- let s = new CustomStack<string>();
- s.push('hello');

- 编译器在使用泛型类型和函数时会确保类型安全。参见以下示例：

- let s = new CustomStack<string>();
- s.push(55); // 将会产生编译时错误

泛型约束

- 泛型类型的类型参数可以绑定。例如，HashMap<Key, Value>容器中的Key类型参数必须具有哈希方法，即它应该是可哈希的。

- interface Hashable {
- hash(): number
- }
- class HasMap<Key extends Hashable, Value> {
- public set(k: Key, v: Value) {
- let h = k.hash();
- // ...其他代码...
- }
- }
- 在上面的例子中，Key类型扩展了Hashable，Hashable接口的所有方法都可以为key调用。

泛型函数

- 使用泛型函数可编写更通用的代码。比如返回数组最后一个元素的函数：

- function last(x: number[]): number {
- return x[x.length - 1];
- }
- last([1, 2, 3]); // 3

- 如果需要为任何数组定义相同的函数，使用类型参数将该函数定义为泛型：

- function last<T>(x: T[]): T {
- return x[x.length - 1];
- }
- 现在，该函数可以与任何数组一起使用。
- 在函数调用中，类型实参可以显式或隐式设置：

- // 显式设置的类型实参
- last<string>(['aa', 'bb']);
- last<number>([1, 2, 3]);

- // 隐式设置的类型实参
- // 编译器根据调用参数的类型来确定类型实参
- last([1, 2, 3]);

泛型默认值

- 泛型类型的类型参数可以设置默认值。这样可以不指定实际的类型实参，而只使用泛型类型名称。下面的示例展示了类和函数的这一点。

- class SomeType {}
- interface Interface <T1 = SomeType> { }
- class Base <T2 = SomeType> { }
- class Derived1 extends Base implements Interface { }
- // Derived1在语义上等价于Derived2
- class Derived2 extends Base<SomeType> implements Interface<SomeType> { }

- function foo<T = number>(): T {
- // ...
- }
- foo();
- // 此函数在语义上等价于下面的调用
- foo<number>();

空安全

- 严格空值检查模式（strictNullChecks），但规则更严格。
- 在下面的示例中，所有行都会导致编译时错误：

- let x: number = null; // 编译时错误
- let y: string = null; // 编译时错误
- let z: number[] = null; // 编译时错误

- 可以为空值的变量定义为联合类型T | null。

- let x: number | null = null;
- x = 1; // ok
- x = null; // ok
- if (x != null) { /* do something */ }

非空断言运算符

- 后缀运算符!可用于断言其操作数为非空。
- 应用于空值时，运算符将抛出错误。否则，值的类型将从T | null更改为T：

- class C {
- value: number | null = 1;
- }

- let c = new C();
- let y: number;
- y = c.value + 1; // 编译时错误：无法对可空值作做加法
- y = c.value! + 1; // ok，值为2

空值合并运算符

- 空值合并二元运算符??用于检查左侧表达式的求值是否等于null。如果是，则表达式的结果为右侧表达式；否则，结果为左侧表达式。
- 换句话说，a ?? b等价于三元运算符a != null ? a : b。
- 在以下示例中，getNick方法如果设置了昵称，则返回昵称；否则，返回空字符串：

```
class Person {
  // ...
  nick: string | null = null
  getNick(): string {
    return this.nick ?? '';
  }
}
```

可选链

- 在访问对象属性时，如果该属性是undefined或者null，可选链运算符会返回undefined。

- class Person {
- nick: string | null = null
- spouse?: Person

- setSpouse(spouse: Person): void {
- this.spouse = spouse;
- }

- getSpouseNick(): string | null | undefined {
- return this.spouse?.nick;
- }

- constructor(nick: string) {
- this.nick = nick;
- this.spouse = undefined;
- }
- }

- 说明：getSpouseNick的返回类型必须为string | null | undefined，因为该方法可能返回null或者undefined。
- 可选链可以任意长，可以包含任意数量的?.运算符。
- 在以下示例中，如果一个Person的实例有不为空的spouse属性，且spouse有不为空的nickname属性，则输出spouse.nick。否则，输出undefined：

- class Person {
- nick: string | null = null
- spouse?: Person

- constructor(nick: string) {
- this.nick = nick;
- this.spouse = undefined;
- }
- }

- let p: Person = new Person('Alice');
- p.spouse?.nick; // undefined

模块

- 程序可划分为多组编译单元或模块。
- 每个模块都有其自己的作用域，即，在模块中创建的任何声明（变量、函数、类等）在该模块之外都不可见，除非它们被显式导出。
- 与此相对，从另一个模块导出的变量、函数、类、接口等必须首先导入到模块中。

导出

- 可以使用关键字export导出顶层的声明。
- 未导出的声明名称被视为私有名称，只能在声明该名称的模块中使用。
- 注意：通过export方式导出，在导入时要加{}。

- export class Point {
- x: number = 0
- y: number = 0
- constructor(x: number, y: number) {
- this.x = x;
- this.y = y;
- }
- }
- export let Origin = new Point(0, 0);
- export function Distance(p1: Point, p2: Point): number {
- return Math.sqrt((p2.x - p1.x) * (p2.x - p1.x) + (p2.y - p1.y) * (p2.y - p1.y));
- }

导入

- 导入声明用于导入从其他模块导出的实体，并在当前模块中提供其绑定。导入声明由两部分组成：
- 导入路径，用于指定导入的模块；
- 导入绑定，用于定义导入的模块中的可用实体集和使用形式（限定或不限定使用）。
- 导入绑定可以有几种形式。
- 假设模块具有路径“./utils”和导出实体“X”和“Y”。
- 导入绑定* as A表示绑定名称“A”，通过A.name可访问从导入路径指定的模块导出的所有实体：

- import * as Utils from './utils'
- Utils.X // 表示来自Utils的X
- Utils.Y // 表示来自Utils的Y

- 导入绑定{ ident1, ..., identN }表示将导出的实体与指定名称绑定，该名称可以用作简单名称：

- import { X, Y } from './utils'
- X // 表示来自utils的X
- Y // 表示来自utils的Y

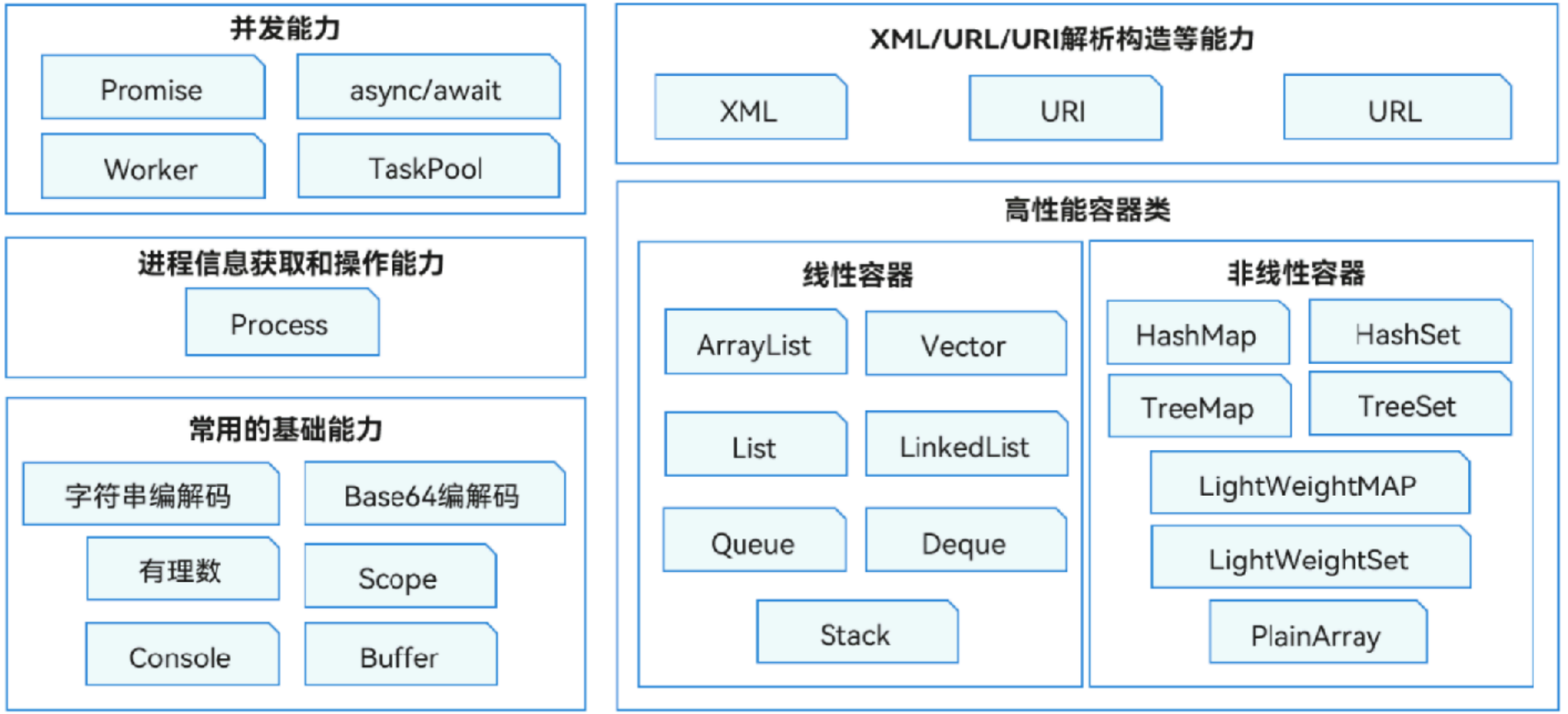
- 如果标识符列表定义了ident as alias，则实体ident将绑定在名称alias下：

- import { X as Z, Y } from './utils'
- Z // 表示来自Utils的X
- Y // 表示来自Utils的Y
- X // 编译时错误：'X'不可见

程序入口

- 程序（应用）的入口是顶层主函数。主函数应具有空参数列表或只有string[]类型的参数。

- function main() {
- 	console.log('this is the program entry');
- }



ArkTS语言基础类库能力示意图

ArkTS容器类库概述

- 容器类库：用于存储各种数据类型的元素，并具备一系列处理数据元素的方法
- 容器类采用了类似静态语言的方式来实现，并通过对存储位置以及属性的限制，让每种类型的数据都能在完成自身功能的基础上去除冗余逻辑，保证了数据的高效访问，提升了应用的性能。
- 当前提供了线性和非线性两类容器，共14种。

线性容器库

- ArrayList 、Vector 、List 、LinkedList 、Deque、 Queue 、Stack

- // 导入ArrayList模块

- import ArrayList from '@ohos.util.ArrayList';

- let arrayList = new ArrayList();

- arrayList.add('a');

- arrayList.add(1); // 增加元素

- console.info(`result: ${arrayList[0]}`); // 访问元素

- arrayList[0] = 'one'; // 修改元素

- console.info(`result: ${arrayList[0]}`);

- // 导入Stack模块

- import Stack from '@ohos.util.Stack';

- let stack = new Stack();

- stack.push('a');

- stack.push(1); // 增加元素

- console.info(`result: ${stack[0]}`); // 访问元素

- stack.pop(); // 弹出元素

- console.info(`result: ${stack.length}`);

// Vector

```
import Vector from '@ohos.util.Vector'; // 导入Vector模块
let vector = new Vector();
vector.add( 'a');
let b1 = [1, 2, 3];
vector.add(b1);
vector.add(false); // 增加元素
console.info(`result: ${vector[0]}`); // 访问元素
 console.info(`result: ${vector.getFirstElement()}`); // 访问元素
```

// Deque

```
import Deque from '@ohos.util.Deque'; // 导入Deque模块
let deque = new Deque;
deque.insertFront('a');
deque.insertFront(1); // 增加元素
console.info(`result: ${deque[0]}`); // 访问元素
deque[0] = 'one'; // 修改元素
console.info(`result: ${deque[0]}`);
```

// List

```
import List from '@ohos.util.List'; // 导入List模块
let list = new List;
list.add( 'a');
list.add(1);
let b2 = [1, 2, 3];
list.add(b2); // 增加元素
console.info(`result: ${list[0]}`); // 访问元素
console.info(`result: ${list.get(0)}`); // 访问元素
```

非线性容器库

- HashMap 、HashSet 、TreeMap 、TreeSet、LightWeightMap 、LightWeightSet 、PlainArray

- // HashMap
- // 导入HashMap模块
- import HashMap from '@ohos.util.HashMap';
- let hashMap = new HashMap();
- hashMap.set('a', 123);
- hashMap.set(4, 123); // 增加元素
- // 判断是否含有某元素
- console.info(`result: ${hashMap.hasKey(4)}`);
- // 访问元素console.info(`result:
- ${hashMap.get('a')}`);

- // HashSet
- import HashSet from '@ohos.util.HashSet';
- let hashSet = new HashSet();
- hashSet.add(123);
- hashSet.add(222); // 增加元素
- // 打印全部元素
- console.info(`result: ${hashSet.values()}`);

- // HashMap
- HashMap底层使用数组+链表+红黑树的方式实现，查询、插入和删除的效率都很高。HashMap 存储内容基于key-value的键值对映射，不能有重复的key，且一个key只能对应一个value。
- HashMap和TreeMap相比，HashMap依据键的hashCode存取数据，访问速度较快。而 TreeMap是有序存取，效率较低。
- HashSet基于HashMap实现。HashMap的输入参数由key、value两个值组成。在HashSet中，　只对value对象进行处理。
- 推荐使用场景：　需要快速存取、删除以及插入键值对数据时，推荐使用HashMap。

- import HashMap from '@ohos.util.HashMap'; // 导入HashMap模块

- let hashMap = new HashMap();
- hashMap.set('a', 123);
- hashMap.set(4, 123); // 增加元素
- console.info(`result: ${hashMap.hasKey(4)}`); // 判断是否含有某元素
- console.info(`result: ${hashMap.get('a')}`); // 访问元素

// TreeMap

TreeMap可用于存储具有关联关系的key-value键值对集合，存储元素中key值唯一，每个key对 应一个value。

TreeMap底层使用红黑树实现，可以利用二叉树特性快速查找键值对。key值有序存储，可以实 现快速的插入和删除。

TreeMap和HashMap相比，HashMap依据键的hashCode存取数据，访问速度较快。而 TreeMap是有序存取，效率较低。

推荐使用场景： 一般需要存储有序键值对的场景，可以使用TreeMap。

- import TreeMap from '@ohos.util.TreeMap'; // 导入TreeMap模块
-
- let treeMap = new TreeMap();
- treeMap.set('a', 123);
- treeMap.set('6', 356); // 增加元素
- console.info(`result: ${treeMap.get('a')}`); // 访问元素
- console.info(`result: ${treeMap.getFirstKey()}`); // 访问首元素
- console.info(`result: ${treeMap.getLastKey()}`); // 访问尾元素

// LightWeightMap

- LightWeightMap可用于存储具有关联关系的key-value键值对集合，存储元素中key值唯一，每 个key对应一个value。
- LightWeightMap依据泛型定义，采用轻量级结构，初始默认容量大小为8，每次扩容大小为原始 容量的两倍。
- 集合中key值的查找依赖于hash算法，通过一个数组存储hash值，然后映射到其他数组中的key 值及value值。
- LightWeightMap和HashMap都是用来存储键值对的集合，LightWeightMap占用内存更小。 推荐使用场景： 当需要存取key-value键值对时，推荐使用占用内存更小的LightWeightMap。

- import LightWeightMap from '@ohos.util.LightWeightMap'; // 导入 LightWeightMap模块

- let lightWeightMap = new LightWeightMap();
- lightWeightMap.set('x', 123);
- lightWeightMap.set('8', 356); // 增加元素
- console.info(`result: ${lightWeightMap.get('a')}`); // 访问元素
- console.info(`result: ${lightWeightMap.get('x')}`); // 访问元素
- console.info(`result: ${lightWeightMap.getIndexOfKey('8')}`); // 访 问元素

// PlainArray

- PlainArray可用于存储具有关联关系的key-value键值对集合，存储元素中key值唯一，key值类　型为number类型，每个key对应一个value。
- PlainArray依据泛型定义，采用轻量级结构，集合中key值的查找依赖于二分查找算法，然后映射 到其他数组中的value值。
- PlainArray和LightWeightMap都是用来存储键值对，且均采用轻量级结构，但PlainArray的key　值类型只能为number类型。
- 推荐使用场景：　当需要存储key值为number类型的键值对时，可以使用PlainArray。

- import PlainArray from '@ohos.util.PlainArray' // 导入PlainArray模块

```
• let plainArray = new PlainArray();
• plainArray.add(1, 'sdd');
• plainArray.add(2, 'sff'); // 增加元素
• console.info(`result: ${plainArray.get(1)}`); // 访问元素
• console.info(`result: ${plainArray.getKeyAt(1)}`); // 访问元素
```

案例讲解- ArkTSAlgorithm

本章总结

- 介绍了ArkTS语言