Dart 语言里的类型体系

Dart 是类型安全的编程语言：Dart 使用静态类型检查和 运行时检查 的组合来确保变量的值始终与变量的静态类型或其他安全类型相匹配。尽管类型是必需的，但由于 类型推断，类型的注释是可选的。

静态类型检查的一个好处是能够使用 Dart 的 静态分析器 在编译时找到错误。

可以向泛型类添加类型注释来修复大多数静态分析错误。最常见的泛型类是集合类型 `List<T>` 和 `Map<K,V>` 。

例如，在下面的代码中，`main()` 创建一个列表并将其传递给 `printInts()`，由 `printInts()` 函数打印这个整数列表。

```
X static analysis: failure                                                    dart
  void printInts(List<int> a) => print(a);

  void main() {
    final list = [];
    list.add(1);
    list.add('2');
    printInts(list);
  }
```

上面的代码在调用 `printInts(list)` 时会在 `list` （高亮提示）上产生类型错误：

```
  error - The argument type 'List<dynamic>' can't be assigned to the parameter type 'List<int>'. -
```

高亮错误是因为产生了从 `List<dynamic>` 到 `List<int>` 的不正确的隐式转换。 `list` 变量是 `List<dynamic>` 静态类型。这是因为 `list` 变量的初始化声明 `var list = []` 没有为分析器提供足够的信息来推断比 `dynamic` 更具体的类型参数。`printInts()` 函数需要 `List<int>` 类型的参数，因此导致类型不匹配。

在创建 `list` 时添加类型注释 `<int>`（代码中高亮显示部分）后，分析器会提示无法将字符串参数分配给 `int` 参数。删除 `list.add("2")` 中的字符串引号使代码通过静态分析并能够正常执行。

```
✓ static analysis: success                                                    dart
  void printInts(List<int> a) => print(a);

  void main() {
    final list = <int>[];
    list.add(1);
    list.add(2);
    printInts(list);
  }
```

尝试在 DartPad 中练习.

什么是类型安全

类型安全是为了确保程序不会进入某些无效状态。安全的类型系统意味着程序永远不会进入表达式求值为与表达式的静态类型不匹配的值的状态。例如，如果表达式的静态类型是 `String` ，则在运行时保证在评估它的时候只会获取字符串。

Dart 的类型系统，同 Java 和 C＃中的类型系统类似，是安全的。它使用静态检查（编译时错误）和运行时检查的组合来强制执行类型安全。例如，将 `String` 分配给 `int` 是一个编译时错误。如果对象不是字符串，使用 `as String` 将对象转换为字符串时，会由于运行时错误而导致转换失败。

类型安全的好处

安全的类型系统有以下几个好处：

- 在编译时就可以检查并显示类型相关的错误。
  安全的类型系统强制要求代码明确类型，因此在编译时会显示与类型相关的错误，这些错误可能在运行时可能很难发现。

- 代码更容易阅读。
  代码更容易阅读，因为我们信赖一个拥有指定类型的值。在类型安全的 Dart 中，类型是不会骗人的。因为一个拥有指定类型的值是可以被信赖的。

- 代码可维护性更高。
  在安全的类型系统下，当更改一处代码后，类型系统会警告因此影响到的其他代码块。

- 更好的 AOT 编译。
  虽然在没有类型的情况下可以进行 AOT 编译，但生成的代码效率要低很多。

静态检查中的一些技巧

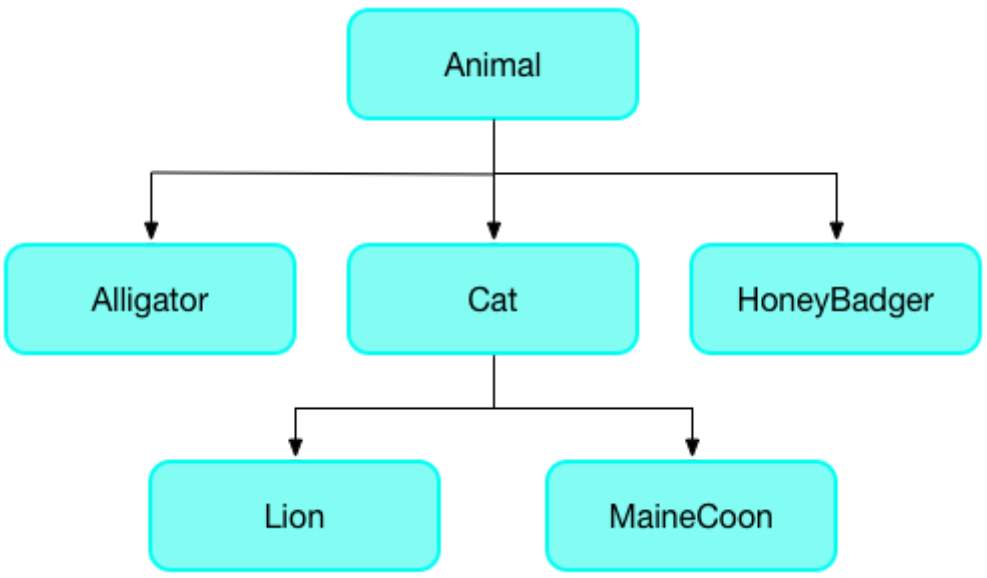
大多数静态类型的规则都很容易理解。下面是一些不太明显的规则：

- 重写方法时，使用类型安全返回值。

- 重写方法时，使用类型安全的参数。

- 不要将动态类型的 List 看做是有类型的 List。

让我们通过下面示例的类型结构，来更深入的了解这些规则：



重写方法时，使用类型安全的返回值

子类方法中返回值类型必须与父类方法中返回值类型的类型相同或其子类型。考虑 Animal 类中的 Getter 方法：

```dart
class Animal {
  void chase(Animal a) { ... }
  Animal get parent => ...
}
```

父类 Getter 方法返回一个 Animal 。在 HoneyBadger 子类中，可以使用 HoneyBadger（或 Animal 的任何其他子类型）替换 Getter 的返回值类型，但不允许使用其他的无关类型。

```dart
✓ static analysis: success
class HoneyBadger extends Animal {
  @override
  void chase(Animal a) { ... }

  @override
  HoneyBadger get parent => ...
}
```

```dart
✗ static analysis: failure
class HoneyBadger extends Animal {
  @override
  void chase(Animal a) { ... }

  @override
  Root get parent => ...
}
```

重写方法时，使用类型安全的参数。

子类方法的参数必须与父类方法中参数的类型相同或是其参数的父类型。不要使用原始参数的子类型，替换原有类型，这样会导致参数类型"收紧"。

> ⓘ 提示
>
> **提示：** 如果有合理的理由使用子类型，可以使用 covariant 关键字。

考虑 Animal 的 `chase(Animal)` 方法：

```dart
class Animal {
  void chase(Animal a) { ... }
  Animal get parent => ...
}
```

`chase()` 方法的参数类型是 Animal 。一个 HoneyBadger 可以追逐任何东西。因此可以在重写 `chase()` 方法时将参数类型指定为任意类型 (Object) 。

```dart
✓ static analysis: success
class HoneyBadger extends Animal {
  @override
  void chase(Object a) { ... }

  @override
  Animal get parent => ...
}
```

Mouse 是 Animal 的子类，下面的代码将 `chase()` 方法中参数的范围从 Animal 缩小到 Mouse 。

```dart
✗ static analysis: failure
class Mouse extends Animal { ... }

class Cat extends Animal {
  @override
  void chase(Mouse a) { ... }
}
```

下面的代码不是类型安全的，因为 a 可以是一个 cat 对象，却可以给它传入一个 alligator 对象。

```dart
Animal a = Cat();
a.chase(Alligator()); // Not type safe or feline safe.
```

不要将动态类型的 List 看做是有类型的 List

当期望在一个 List 中可以包含不同类型的对象时，动态列表是很好的选择。但是不能将动态类型的 List 看做是有类型的 List 。

这个规则也适用于泛型类型的实例。

下面代码创建一个 Dog 的动态 List，并将其分配给 Cat 类型的 List，表达式在静态分析期间会产生错误。

```dart
✗ static analysis: failure
void main() {
  List<Cat> foo = <dynamic>[Dog()]; // Error
  List<dynamic> bar = <dynamic>[Dog(), Cat()]; // OK
}
```

运行时检查

运行时检查工具会处理分析器无法捕获的类型安全问题。

例如，以下代码在运行时会抛出异常，因为将 Dog 类型的 List 赋值给 Cat 类型的 List 是错误的：

```dart
✗ runtime: failure
void main() {
  List<Animal> animals = <Dog>[Dog()];
  List<Cat> cats = animals as List<Cat>;
}
```

类型推断

分析器 (analyzer) 可以推断字段，方法，局部变量和大多数泛型类型参数的类型。当分析器没有足够的信息来推断出一个特定类型时，会使用 dynamic 作为类型。

下面是在泛型中如何进行类型推断的示例。在此示例中，名为 arguments 的变量包含一个 Map，该 Map 将字符串键与各种类型的值配对。

如果显式键入变量，则可以这样写：

```dart
Map<String, dynamic> arguments = {'argA': 'hello', 'argB': 42};
```

或者，使用 var 让 Dart 来推断类型：

```dart
var arguments = {'argA': 'hello', 'argB': 42}; // Map<String, Object>
```

Map 字面量从其条目中推断出它的类型，然后变量从 Map 字面量的类型中推断出它的类型。在此 Map 中，键都是字符串，但值具有不同的类型（String 和 int，它们具有共同的上限类型 Object）。因此，Map 字面量的类型为 Map<String, Object>，也就是 arguments 的类型。

字段和方法推断

重写父类的且没有指定类型的字段或方法，继承父类中字段或方法的类型。

没有声明类型且不存在继承类型的字段，如果在声明时被初始化，那么字段的类型为初始化值的类型。

静态字段推断

静态字段和变量的类型从其初始化程序中推断获得。需要注意的是，如果推断是个循环，推断会失败（也就是说，推断变量的类型取决于知道该变量的类型）。

局部变量推断

在不考虑连续赋值的情况下，局部变量如果有初始化值的情况下，其类型是从初始化值推断出来的。这可能意味着推断出来的类型会非常严格。如果是这样，可以为他们添加类型注释。

```dart
✗ static analysis: failure
var x = 3; // x is inferred as an int.
x = 4.0;
```

```dart
✓ static analysis: success
num y = 3; // A num can be double or int.
y = 4.0;
```

参数类型推断

构造函数调用的类型参数和 泛型方法 调用是根据上下文的向下信息和构造函数或泛型方法的参数的向上信息组合推断的。如果推断没有按照意愿或期望进行，那么你可以显式的指定他们的参数类型。

```dart
✓ static analysis: success
// Inferred as if you wrote <int>[].
List<int> listOfInt = [];

// Inferred as if you wrote <double>[3.0].
var listOfDouble = [3.0];

// Inferred as Iterable<int>.
var ints = listOfDouble.map((x) => x.toInt());
```

在最后一个示例中，根据向下信息 `x` 被推断为 `double` 。闭包的返回类型根据向上信息推断为 `int` 。在推断 `map()` 方法的类型参数：`<int>` 时，Dart 使用此返回值的类型作为向上信息。

替换类型

当重写方法时，可以使用一个新类型（在新方法中）替换旧类型（在旧方法中）。类似地，当参数传递给函数时，可以使用另一种类型（实际参数）的对象替换现有类型（具有声明类型的参数）要求的对象。什么时候可以用具有子类型或父类型的对象替换具有一种类型的对象那?

从_消费者_和_生产者_的角度有助于我们思考替换类型的情况。消费者接受类型，生产者产生类型。

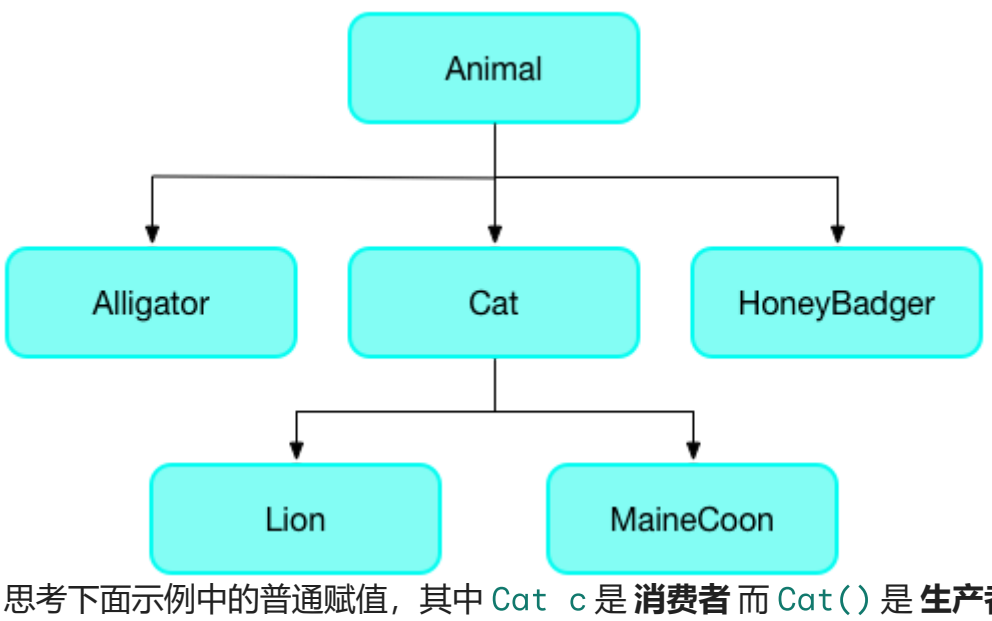
可以使用父类型替换消费者类型，使用子类型替换生产者类型。

下面让我们看一下普通类型赋值和泛型类型赋值的示例。

普通类型赋值

将对象赋值给对象时，什么时候可以用其他类型替换当前类型？答案取决于对象是消费者还是生产者。

分析以下类型层次结构：



思考下面示例中的普通赋值，其中 `Cat c` 是 **消费者** 而 `Cat()` 是 **生产者**:

```dart
Cat c = Cat();
```

在消费者的位置，任意类型（`Animal`）的对象替换特定类型（`Cat`）的对象是安全的。因此使用 `Animal c` 替换 `Cat c` 是允许的，因为 Animal 是 Cat 的父类。

```dart
✓ static analysis: success
Animal c = Cat();
```

但是使用 `MaineCoon c` 替换 `Cat c` 会打破类型的安全性，因为父类可能会提供一种具有不同行为的 Cat ， 例如 Lion：

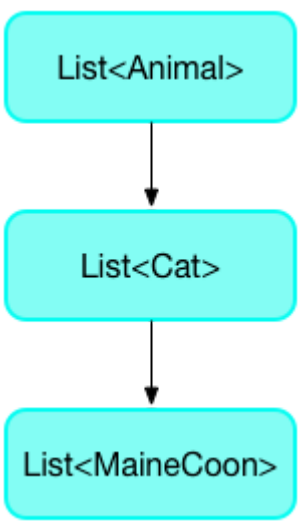
```dart
✗ static analysis: failure
MaineCoon c = Cat();
```

在生产者的位置，可以安全地将生产类型 (Cat) 替换成一个更具体的类型 (MaineCoon) 的对象。因此，下面的操作是允许的：

```dart
✓ static analysis: success
Cat c = MaineCoon();
```

泛型赋值

上面的规则同样适用于泛型类型吗？是的。考虑动物列表的层次结构— Cat 类型的 List 是 Animal 类型 List 的子类型，是 MaineCoon 类型 List 的父类型。



在下面的示例中，可以将 `MaineCoon` 类型的 List 赋值给 `myCats` ，因为 `List<MaineCoon>` 是 `List<Cat>` 的子类型：

```dart
✓ static analysis: success
List<MaineCoon> myMaineCoons = ...
List<Cat> myCats = myMaineCoons;
```

从另一个角度看，可以将 `Animal` 类型的 List 赋值给 `List<Cat>` 吗?

```dart
✗ static analysis: failure
List<Animal> myAnimals = ...
List<Cat> myCats = myAnimals;
```

这个赋值不能通过静态分析，因为它创建了一个隐式的向下转型 (downcast)，这在非 `dynamic` 类型中是不允许的，比如 `Animal`。

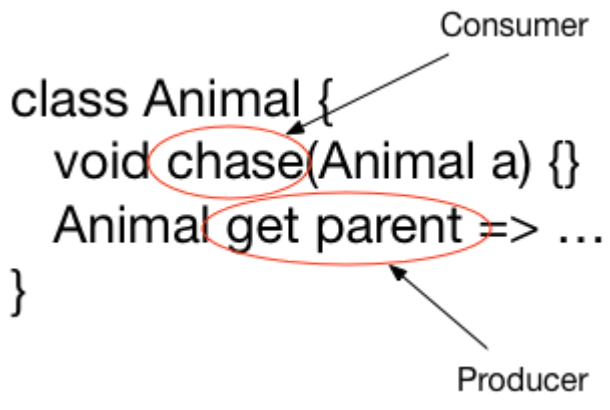
若要这段代码能够通过静态分析，需要使用一个显式转换，这可能会在运行时导致失败。

```dart
List<Animal> myAnimals = ...
List<Cat> myCats = myAnimals as List<Cat>;
```

不过，显式转换在运行时仍然可能会失败，这取决于转换被转换内容的实际类型 (此处是 `myAnimals`)。

方法

在重写方法中，生产者和消费者规则仍然适用。例如：



对于使用者（例如 `chase(Animal)` 方法），可以使用父类型替换参数类型。对于生产者（例如 父类 的 Getter 方法），可以使用子类型替换返回值类型。

有关更多信息，请参阅 重写方法时，使用类型安全的返回值 以及 重写方法时，使用类型安全的参数。

其他资源

以下是更多关于 Dart 类型安全的相关资源：

- 修复常见类型问题 - 编写类型安全的 Dart 代码时可能遇到的错误，以及解决错误的方法。
- 修复类型转换错误 - 了解和学习如何修复类型转换错误
- 健全的空安全 - 学习关于如何撰写健全的空安全代码。
- Customizing static analysis - 如何使用分析配置文件设置及自定义分析器和 linter。