集成 Java

1. `scala` 代码经常和大型 `Java` 程序以及框架一起使用。由于 `Scala` 和 `Java` 高度兼容，因此大部分时间你结合这两种语言时并不需要太多顾虑。

一、通用规则

1. `Scala` 的实现方式是将代码翻译成标准的 `Java` 字节码。 `Scala` 的特性会尽可能地直接映射为相应的 `Java` 特性。如：`Scala` 的类、方法、字符串、异常都和它们在 `Java` 中的概念一样编译成 `Java` 字节码。

   为实现这一点，在设计 `Scala` 的过程中有时需要作出艰难的抉择。如：如果能够在运行期类型解析从而确定重载的方法，而不是在编译期决定，可能会更好。但是这样的设计会破坏 `Java` 的重载解析，使得无法混用 `Java` 和 `Scala` 代码。在这个问题上， `Scala` 和 `Java` 的重载解析保持一致。

2. `Scala` 在某些特性上有自己的设计。例如：特质 `trait` 在 `Java` 中就没有与之相对应的概念。

   同样地，虽然 `Scala` 和 `Java` 都有泛型，但是这两个特性在细节上存在冲突。

   对于类似这些语言特性， `Scala` 代码无法直接映射为 `Java` 语言结构，因此它必须结合 `Java` 现有的特性来进行编码。对于这些无法直接映射的特性，编码规则并不是固定的。可以使用类似 `javap` 这样的工具查看 `.class` 文件来获知当前的 `Scala` 编译器使用的翻译规则。

二、特殊规则

1. 除了通用规则之外，还有一些特殊规则。

2.1 值类型

1. 类似于 `Int` 这样的值类型翻译成 `Java` 有两种不同的方式。
   - 只要可能，编译器就会将 `Scala` 的 `Int` 翻译成 `Java` 的 `int` 从而获得更好的性能。
   - 但有时候做不到，因为编译器不确定它正在翻译的是一个 `Int` 还是另外某种数据类型。如：尽管 `List[Any](1,2,3)` 只包含 `Int` 类型的元素，但是编译器无法确认这一点。

     对于这样的情况，编译器不确定某个对象是不是值类型，因此会选择使用对象类型并依赖相应的包装类（例如 `java.lang.Integer` ）。

2.2 单例对象

1. `Java` 并没有单例对象的确切概念，不过`Java` 有静态方法。 `Scala` 对单例对象的翻译采取了静态方法和实例方法相结合的方式。

   对于每个`Scala` 单例对象，编译器都为这个对象创建了一个名称加美元符号 `$` 的 `Java` 类。例如：对于名叫 `App` 的单例对象，编译器产出一个名为 `App$` 的`Java` 类。这个 `Java` 类拥有 `Scala` 单例对象的所有方法和字段。另外，这个 `Java` 类还有一个名为 `MODULE$` 的静态字段，它保存了该类在运行期间创建的一个实例。

   例如， `Scala` 单例对象：

```
object App{
  def main(args: Array[String]) ={
    println("Hello world")
  }
}
```

将被翻译成一个 `Java` 类 `App$` ：

```
public final class App$ extends java.lang.Object
  implements scala.ScalaObject{
    public static final App$ MODULE$;     // 保存 APP$ 运行期间创建的一个实例
    public static {};
    public App$();
    public void main(java.lang.String[]);
    public int $tag();
}
```

2. 通常情况下，`Scala` 的单例对象翻译如前所述。但是一个重要的特例是：当面对一个"独立的"单例对象时（即没有同名的类与之对应），此时编译器将会创建一个同名的 `Java` 类，这个类对于每个 `Scala` 单例对象的方法都有一个静态方法与之对应。

   如我们有一个 `App2` 的 `Scala` 单例对象，但是没有 `App2` 的 `Scala` 类与之对应：

```
object App2{
  def main(args: Array[String]) ={
    println("Hello world")
  }
}
```

   则它将被翻译为：

```
public final class App2 extends java.lang.Object{
  public static final int $tag();
  public static final void main(java.lang.String[]);
}
```

   相反，如果你的 `Scala` 代码中有一个名为 `App2` 的类，那么 `Scala` 会创建相应的 `Java` 类 `App2$` 来保存你定义的成员。此时，它不会添加任何转发到同名单例对象的方法， `Java` 代码必须通过 `MODULE$` 字段来访问这个单例。

2.3 接口

1. 编译任何特质都会创建一个同名的 `java` 接口，这个接口可以作为 `Java` 类型使用。
2. 在 `Java` 中实现特质是另外一回事。通常情况下，这样做不切实际。
3. 如果你写的 `scala trait` 仅包含抽象方法，那么这个特质会直接翻译成 `java` 接口。否则可能会包含额外的代码。

三、注解

1. 有一些注解编译器在针对 `Java` 平台编译时会产生额外的信息。当编译器看到这样的注解时，它会首先根据一般的 `scala` 原则去处理，然后针对 `scala` 做一些额外的工作。

- `deprecated`：对于任何标记为 `@deprecated` 的方法或类，编译器会为产出的代码添加 `Java` 自己的过期注解。因此 `Java` 编译器能够在 `Java` 代码访问过期 `scala` 方法时给出过期告警。
- `volatile`：`scala` 中标记为 `@volative` 的字段会在产出的代码中添加 `java` 的 `volative` 修饰符。因此 `scala` 中的 `volatile` 字段和 `java` 的处理机制完全一致。
- 序列化：`scala` 中的三个标准序列化注解全部都被翻译成 `Java` 中对应的语法结构。
  - 标记为`@serializable` 的类会被加上 `Java` 的 `Seriablizable` 接口。
  - `@serialVersion UID(1234>)` 会被转换成如下的 `Java` 字段定义：

    ```
    private final static long SerialVersionUID = 1234L
    ```

  - 标记为 `@transient` 的变量会被加上 `Java` 的 `transient` 修饰符。

2. `scala` 并不检查抛出的异常是否被代码捕获。也就是说：`Scala` 的方法并没有与 `Java` 中 `throws` 声明相对应的定义。所有 `Scala` 方法都被翻译成没有声明任何抛出异常的 `Java` 方法。

   `scala` 之所以这么做，是因为在 `java` 中人们对于 `throws` 声明的体验并不全是正面的。很多开发者为了能够通过编译，直接在代码里吃掉并丢弃异常处理的代码。他们可能也想在之后强化异常处理的逻辑，但是经验表明：由于时间压力，大多数程序员几乎从来不会回过头来增加正确的异常处理。这样带来的扭曲的结果就是这个本意很好的特性成为了负担。

   因此，大量生产环境下的 `Java` 代码都吃掉并隐藏了运行期的异常，仅仅是为了让编译器通过而已。

   如果你希望在 `scala` 中开启异常，则可以使用 `@throws` 注解来标注你的方法：

   ```
   class A(name: String){
     ...
     @throws(classOf[IOException]) // 翻译成 java 的 throws 声明
     def read() = ...
   }
   ```

3. `Java` 框架中的注解可以直接在 `Scala` 代码中使用，任何 `Java` 框架都可以看到你编写的注解，就好像你是用 `Java` 编写的一样。

四、通配类型

1. 所有 `Java` 类型在 `Scala` 中都有对等的概念。这是必要的，因为只有这样 `Scala` 代码才能访问任何合法的 `Java` 类。但是在某些情况下，如 `Java` 中的 `Iterator<?>` 或者 `Iterator<? Extends Component>` 的 `Java` 通配类型，`scala` 使用一种额外的叫做通配类型 `wildcard type` 来表示。

   `scala` 中的通配类型的编写方式是：通过占位符语法，类似函数字面量的简写方式。在函数字面量中，可以通过下划线 `_` 代替表达式，如 `( _ + 1)` 等价于 `( x => x + 1)`。通配类型也是相同的理念，但是它针对的是类型而不是表达式。如果你写下 `Iterator[_]`，则这里的下划线 `_` 就是代表某个类型。这个类型声明表示的是一个元素类型未知的 `Iterator`。

2. 你也可以在占位符语法中插入类型上界或类型下界，只需要在下划线之后添加即可，使用和类型参数相同的 `<:` 语法：

```
Iterator[ _ <: Component] // 表示一个元素类型未知的 Iterator，但是元素类型必须是 Component 的子类
```

3. 在简单的使用中，可以忽略通配符，直接调用集合的方法：

```
// java 代码
public class ABC{
  public Collection<?> contents(){
    Collection<String> s = new Vector<String> ();
    s.add("a");
    s.add("b");
    return s;
  }
}
```

在 `Scala` 中可以直接使用：

```
val contents = (new ABC).contents // 返回 java.util.Collection[_]
```

对于更复杂的情况，需要做更多额外的工作：

```
val iter = (new Wild).contents.iterator
val set = scala.collections.mutable.Set.empty[] // 这里应该填什么类型？不知道
while(iter.hasMore){
  set += iter.next()
}
```

有两个技巧来解决该问题：

- 给方法分配一个类型参数来表示这个通配类型。
- 不用从方法返回通配符，而是返回一个对每个占位符都定义了抽象成员的对象

```
abstract class CCC{
  type Elem
  val set:scala.collections.mutable.Set[Elem]
}

def f(javaSet: java.util.Collection[T]): CCC = {
  val s = scala.collection.mutable.Set.empty[T]   // 现在可以用 T 表示通配类型
  val iter = javaSet.contents.iterator
  while(iter.hasMore){
    set += iter.next()
  }
  return new CCC{  // 返回一个对象
    type Elem = T
    val set = s
  }
}
```

由于 `scala` 中通配类型的使用比较繁琐，因此你也可以在一开始就是使用抽象成员而避免使用通配类型。

五、同时编译 Scala 和 Java

1. 当编译依赖 `Java` 代码的 `Scala` 代码时，你首先将 `Java` 代码构建成 `.class` 文件。然后再编译 `Scala` 代码，并将 `Java` 代码的 `class` 文件放到 `class path` 中。

   不过，这种方式对于 `Java` 代码反过来引用 `Scala` 代码的情况就不行了。此时，无论你采用何种顺序编译代码，其中的某一方都会有未被满足的外部引用。

   为支持这样的场景，`Scala` 允许同时面对 `Java` 代码和 `Java class` 文件做编译。你只需要将 `Java` 代码放在命令行中，就当做它们是 `Scala` 文件那样。`Scala` 编译器不会编译这些 `Java` 文件，不过会扫描它们，看看它们包含了哪些内容。

   以下是同时编译 `Scala` 和 `Java` 的流程：

   - 首先利用 `Java` 源文件编译 `Scala` 代码：（`-d` 表示存放编译后的 `class` 文件的位置）

     ```
     scalac -d bin A.scala B.java C.java
     ```

   - 然后利用 `Scala` 编译出的 `class` 文件来编译 `Java`： （`-cp` 表示指定依赖的其它 `class` 的路径，等价于 `-classpath` ； `-d` 表示编译后的 `class` 文件的位置）

     ```
     javac -cp bin -d bin B.Java C.java D.java
     ```

六、Scala2.12 和 Java 8 的集成

1. `Java8` 对 `Java` 语言和字节码做了一些改进，而 `Scala 2.12` 开始用到了这些改进。通过利用 `Java8` 的这些新特性，`Scala2.12` 的编译器可以生成更小的 `class` 文件和 `jar` 文件，同时改善了特质的二进制兼容性。
2. 从 `Scala` 程序员的角度看，`Scala2.12` 中与 `Java8` 相关的最显著的改进是：`Scala` 函数字面量现在可以像 `Java8` 中的 `lambda` 表达式那样当做匿名类的实例的精简形式来使用。

   在 `Java8` 之前，为了将某种 `action` 传入某个方法中，`Java` 通常都会定义匿名内部类的实例，如：

   ```
   JButton button = new JButton();
   button.addActionListener(
     new ActionListener(){
       public void actionPerformed(ActionEvent event){
         System.out.println("pressed!");
       }
     }
   )
   ```

   这里一个匿名的 `ActionListener` 的实例被创建出来，然后传递给 `JButton.addActionListener` 方法。

   在 `Java8` 中，任何需要某个只包含单个抽象方法（又称作 `single abstract method:SAM` ）的类或接口的实例的地方，都可以使用 `lambda` 表达式。 `ActionListener` 就是这样一个接口，因为它只包含单个抽象方法 `actionPerformed` 。因此在 `Java8` 中上述代码可以调整为：

```
JButton button = new JButton();
button.addActionListener(
  event -> System.out.println("pressed!")
)
```

而在 `Scala` 中，我们可以进一步使用函数字面量：

```
val button = new JButton
button.addActionListener( _ => println("pressed!"))
```

- 在 `Scala 2.12` 之前，为了写出上述代码，我们需要定义一个从 `ActionEvent => Unit` 函数类型到 `ActionListener` 类型的隐式转换，从而支持这种风格的代码。
- 在 `Scala 2.12` 及其以后，我们可以直接使用函数字面量，而不需要定义这个隐式类型转换。

3. 在 `Scala 2.12` 中，允许在任何要求某个 `SAM` 的类或者特质的实例的地方使用函数类型。

   注意：只有函数字面量可以被适配成 `SAM` 类型，而不能使用函数类型的表达式。如：

   ```
   val button = new JButton
   val f = (event : ActionEvent) => println("pressed!") // 函数类型的表达式
   button.addActionListener(f)  // 不支持
   ```

4. `Scala` 的函数类型定义为包含具体方法的特质，而 `Scala 2.12` 会将特质编译为带有默认方法的 `Java` 接口（`Java 8` 的新特性）。因此，在 `Java` 看来，`Scala` 的函数类型跟 `SAM` 没什么两样。

   因此如果在 `Java` 代码中调用 `Scala` 方法，那么 `Scala` 方法中需要函数类型的地方，`Java` 都可以传入 `lambda` 表达式。