第 1 章　新型敏捷硬件开发语言——Chisel 和 Scala

1.1　最好的宿主——什么是 Scala

“如果今天让我在 Java 之外选一门语言，我会选 Scala。”[1]

——James Gosling，Java 之父

在今天众多的编程语言中，Java 常常是软件开发者的首选语言。而能让 Java 之父给出如此评价的 Scala，想必有其吸引人之处。那么，Scala 究竟是一门什么样的语言呢？

Scala 是一门基于 JVM（Java Virtual Machine，Java 虚拟机）运行的语言，并且兼容现有的 Java 程序，在设计之初就考虑了与 Java 的无缝衔接。Scala 代码不需要任何特殊的语法、显式的接口描述，就可以直接调用 Java 方法、访问 Java 字段、从 Java 类继承、实现 Java 接口。Java 代码也可以调用 Scala 代码，不过由于 Scala 的语义比 Java 更为丰富，因此有些更为先进的 Scala 特性映射在 Java 前需要先被编码。但是 Scala 设计者的目的是创造一门比 Java 更好用、更高效、更优秀的语言。从运行机制上讲，Scala 会被编译成与 Java 一样的字节码，交由 JVM 运行，所以其运行时的速度通常与 Java 程序不分上下。从实用性来看，它的形式比 Java 简洁得多，语法功能更加强大，代码量往往比相同功能的 Java 少得多。

一方面，Scala 是一门面向对象的函数式语言。时至今日，面向对象已经成为大多数编程语言都支持的主要特性。另一方面，Scala 没有选择更多人熟悉的指令式编程风格，而是选择了更为小众的函数式编程理念。对于熟悉 C/C++、Java、Python 等语言的读者来说，可能从未接触过函数式编程。但只需要基本的学习，读者便能掌握基本的函数式编程，并会逐步发现函数式编程的妙处。Scala 提倡使用者使用函数式编程，但也预留了指令式编程的余地。

正如它名字取自的“Scalable”一样，这也是一门可以自由伸缩的语言：既能裁剪已有的类库，又能扩展自定义类库；既能用于编写一个简单的脚本，又足以胜任复杂、庞大的软件系统编程任务。Scala 的语法比 Python 更为简洁，抽象能力也比 C++更为高级，因此，Scala 的学习曲线并不是平滑的，而是阶梯状的。也正因此，如果读者能耐心学习 Scala，并逐步掌握它提供的高级语法，深入理解其编程理念，就会发现这是一种让你爱不释手、相见恨晚的编程语言。

Scala 最大的优势就是其各种语法便利造就的强大伸缩性，进而使其成为一种优秀的宿主语言。换句话说，开发者可以方便地利用自定义 Scala 类库，快速开发出“新”语言，专门用于某一特殊用途。

1.2　敏捷开发——什么是 Chisel

对每个数字电路工程师而言，Verilog HDL（Verilog Hardware Description Language，Verilog 硬件描述语言）是再熟悉不过的了。然而，Verilog HDL 是 C 语言时代的产物。现如今，其开

发效率低下的问题越来越明显。与之相比，软件的开发效率早已随着各种高级语言和先进的设计验证方法学的出现而突飞猛进。同时，晶体管密度也随摩尔定律水涨船高，而数字集成电路最前端的 HDL（Hardware Description Language，硬件描述语言）似乎还在原地踏步。早在二三十年前，人们就认为 Verilog HDL 快要过时了。当时的开发者主要分成三大派：一派主张应该改进 Verilog HDL，另外两派则主张把 HDL 转移到软件语言（一派主张 C++，另一派主张 Java），这样就能获得开源的综合工具和仿真器。最终，Verilog HDL 的改进版获胜了，也就是 Verilog 的后续标准——SystemVerilog。这是因为，一方面支持改进 Verilog 的人占多数，另一方面 SystemVerilog 也得到了某些大公司的支持。

SystemVerilog 的出现，使硬件描述语言迈向面向对象和验证驱动的方向。但在很长一段时间里，SystemVerilog 更多地被用于验证而非设计。其复杂的语法结构和较低的抽象层级，让设计人员在大规模系统开发中依旧感到烦琐。然而，SystemVerilog 毕竟是业界的主流标准，几乎所有 EDA 工具都提供了良好的支持，这也为后来的新型硬件描述语言提供了兼容的目标。

C++阵营的产物是 SystemC。事实上，SystemC 也就是用 C++定义的一堆类库。SystemC 的开发多数仍用于事务级，即编写硬件模型。用 SystemC 直接开发硬件并不多见，因为目前 EDA 工具支持得不够好，相比 Verilog HDL，开发出来的电路的优化性能并不好。

主张 Java 的一派呢？我们已经看到了基于 Java 平台的 HDL 语言——Chisel（Constructing Hardware In a Scala Embedded Language）出现，尽管 Scala 不是 Java，但是它也是基于 JVM 的。Chisel 是一门以 Scala 为宿主语言开发的硬件描述语言，它是由加州大学伯克利分校的研究团队发布的一种新型硬件开发语言。据团队成员之一 Krste Asanovic 教授介绍，早在三十多年前还没有硬件描述语言的时候，他们就已经开始构想这样一种语言了。最开始 Chisel 是基于 Ruby 语言的，但是后来发现 Scala 更适合构建 Chisel。因为 Scala 有诸多特性适合描述电路，比如它是静态语言，适合转换成 Verilog HDL/VHDL；再如它的操作符即方法、柯里化、纯粹的面向对象、强大的模式匹配、便捷的泛型编写、特质混入、函数式编程等特性，使得用 Scala 开发 DSL（Domain Specific Language，领域专用语言）很方便。

早期版本的 Chisel 通过 FIRRTL 编译器把 Chisel 文件转换成 FIRRTL 文件，再由后端工具生成 Verilog HDL/VHDL 文件。如今，随着 CIRCT 项目的引入，Chisel 已经可以直接生成高质量的 SystemVerilog 代码。这一改进由 firtool 工具实现，它将 FIRRTL 转换为标准的 SystemVerilog 输出，并与主流 EDA 工具（如 VCS、Verilator、Vivado 等）无缝兼容。也就是说，Chisel 现在不再是 SystemVerilog 的替代品，而是其高层构建语言（High-Level Construction Language）——设计者可以在更抽象的层次描述电路逻辑，而生成的结果依旧是符合工业标准的 SystemVerilog 文件。

因此，Chisel 的地位从“新语言实验”转变为“SystemVerilog 的上层抽象”，既能充分利用 Scala 语言的现代化语义，又能直接落地到成熟的 EDA 工具链。这使得 Chisel 成为一种既面向未来，又兼容现实工程生态的硬件开发语言。

Chisel 解决了 Verilog HDL 的一些痛点。首先，也是最重要的，就是在硬件电路设计中引入了面向对象的特性。其次，减少了很多不必要的语法，改进了有瑕疵的语法。Verilog HDL 的初衷本就是用于电路验证，而不是设计，因此存在很多不可综合的语法。在 Chisel 转换成 SystemVerilog 的过程中，不会采用这些不可综合的语法，因此编写 Chisel 时不用担心无法生成电路，对硬件新手而言是很便利的。再次，Verilog HDL 的 reg 不一定指代寄存器，这常常被新手误解，而 Chisel 的 Reg 就是寄存器，没有歧义。最后，利用 Scala 的模式匹配、特质混入、类继承等特性，能够迅速改变电路结构。对于日益庞大的 SoC 系统设计而言，这是非常重要的。

加州大学伯克利分校在设计著名的 RISC-V 处理器时引入了 Chisel 语言，是因为其 RISC-V 团队仅有十几名成员，如果使用传统的硬件描述语言实现 CPU，开发过程会是漫长、艰难的，且效率极低。然而，他们借助 Chisel，仅花了一年时间和 100 万美元，就完成了从 RISC-V 指令集设计到 Rocket 芯片的成功流片。这不得不让人好奇，Chisel 这把“凿子”（chisel 的中文意思）究竟有多大的魔力？接下来，我们就来细致地学习 Scala/Chisel。

自 Chisel 问世以来，已经历了多个重要的发展阶段。早期版本 Chisel 2 采用较为原始的编译框架，其生成 Verilog HDL 的路径依赖较多，使用门槛相对较高。2016 年以后，随着 FIRRTL（Flexible Intermediate Representation for RTL）中间语言的引入，Chisel 3 正式成为主流版本。FIRRTL 使得 Chisel 在语言前端与硬件后端之间建立了明确的抽象层，极大地提高了扩展性与工具链兼容性。

2019 年以后，Chisel 正式纳入 CHIPS Alliance（由 Google、SiFive、Intel、Western Digital 等企业支持），此后发展得更加迅速。各版本的主要改进如下。

- **Chisel 3.4～3.6 阶段（2020—2022 年）**：完善了类型系统，增强了 Vec 与 Bundle 等复杂类型的推导机制；
- **Chisel 5.x（2022—2023 年）**：重构了编译流程，引入 ChiselStage 与测试框架 chiseltest；
- **Chisel 6.x（2024 年）**：与 CIRCT（Circuit IR Compilers and Tools）项目深度集成，支持 firtool 作为默认 SystemVerilog 生成后端；
- **Chisel 7.x（2025 年）**：Chisel 7.x 逐步完善对 Scala 3 的支持（仍以 Scala 2.13 为主流配置），生态库正在陆续适配中，在工程实践中仍建议优先使用 Scala 2.13.x，支持 Layer（层次化设计）理念，改进元编程接口和硬件生成器模式。

Chisel 7.3 是当前的稳定主线版本，它的主要特性包括以下内容。

- **更完善的类型与位宽推导机制**，大幅减少显式声明；
- **统一的 FIRRTL/CIRCT 后端**，通过内置 CIRCT firtool 直接生成.sv 后缀的 SystemVerilog 文件（不再使用传统.v），可与现有的 Verilog 代码无缝混用；
- **新增 Layer 抽象**，支持在不同设计层次（行为层、结构层、物理层）进行灵活组合；
- **增强的调试与仿真接口**，内置与 Verilator 和 VCS 兼容的测试框架；
- **改进的工程化支持**，可直接通过 sbt 或 scala-cli 进行快速编译与验证。

Chisel 从学术研究走向工业实践，已经成为 RISC-V 生态中最重要的硬件描述语言之一。无论是 Rocket、BOOM，还是 SiFive 的商业核，都采用 Chisel 进行设计。随着 Chisel 7.3 的发布，这门语言已从科研原型工具成长为成熟的敏捷硬件开发平台，具备支撑现代 SoC 与 AI 加速芯片设计的能力。

1.3　Scala 入门——让你的代码跑起来

1.3.1　Scala 的安装方法

要使用 Scala，首先需要保证已经安装好了 Java。Chisel 7.3 推荐使用 JDK 17 或更高版本，因为新版 Scala 3 与工具链（sbt、scala-cli）都要求较新的 JVM 环境。

对于 Linux 操作系统，可以执行以下命令安装 Java：

```
sudo apt update
```

```
sudo apt install openjdk-17-jdk
```

安装完成后，在终端中输入：

```
java -version
```

若输出类似以下信息，则说明 Java 安装成功：

```
openjdk version "17.0.11" 2025-04-16
OpenJDK Runtime Environment (build 17.0.11+9-Ubuntu-1)
OpenJDK 64-Bit Server VM (build 17.0.11+9-Ubuntu-1, mixed mode, sharing)
```

接下来，从 Scala 官方网站下载 Scala 2.13.17 版本的安装包。Chisel 7.x 官方只支持 Scala 2.13 系列，推荐使用与官方模板一致的 Scala 2.13.x（如 2.13.16/2.13.17）。本书采用 Scala 2.13.17。Scala 3.3.x 虽可在部分简单项目中完成编译，但由于部分宏插件与 chiseltest 尚未完全兼容，暂不推荐在生产环境中使用。Linux 系统可根据发行版选择对应的.deb 或.rpm 安装包。

提示：本书中为 Chisel 7.3 配套使用 sbt 1.9.9 作为项目构建工具，并通过 firtool（来自 CIRCT 项目）自动生成 SystemVerilog 代码。仿真工具请源码安装 Verilator 5.024，Ubuntu 仓库版本过旧会导致 firtool 生成文件无法通过仿真。

1.3.2 使用 Scala 解释器

安装完成后，在终端中运行以下命令即可进入 Scala 解释器（REPL 环境）：

```
scala
```

出现如下提示表示成功：

```
Welcome to Scala 2.13.17 (OpenJDK 64-Bit Server VM, Java 17.0.11)
Type in expressions for evaluation. Type :help for more information.
scala>
```

此时，光标会自动移动到“scala>”的后面，接下来就可以输入 Scala 代码，按回车键运行。例如：

```
scala> 1 + 2
val res0: Int = 3
```

如果代码不足以构成一条语句，那么按回车键后并不会运行，而是自动跳转到下一行，等待其余代码的输入。例如：

```
scala> println(
     | "Hello, Chisel!"
     | )
Hello, Chisel!
```

要退出解释器，只需输入以下命令：

```
scala> :q
```

终端返回命令行提示符，即退出解释器。

1.3.3 运行 Scala 脚本

Scala 文件的后缀名为“.scala”。创建一个新的脚本文件，如 hello.scala，输入以下内容并保存：

```
// hello.scala
println("Hello, Chisel 7.3!")
```

在该文件的路径下打开终端，运行以下命令：

```
scala hello.scala
```

输出结果如下：

```
Hello, Chisel 7.3!
```

此方法适用于简单脚本或实验性测试。对于大型项目，推荐使用 sbt 或 scala-cli 构建工具。

1.3.4　编译非脚本文件

非脚本文件通常以类（class）、对象（object）或包（package）定义结尾。创建一个文件 Hello.scala，内容如下：

```
object Hello extends App {
  val hw = "Hello, Chisel!"
  def display(): Unit = println(hw)
  display()
}
```

在终端中运行以下命令编译：

```
scalac Hello.scala
```

编译完成后，会在当前目录中生成 Hello.class 文件。然后使用解释器调用：

```
scala -classpath . Hello
```

就能执行 Scala 程序。

说明：在现代 Chisel 工程中，通常使用 sbt 构建系统运行 Scala 代码。命令 sbt "runMain mypackage.Main"可直接生成 SystemVerilog 文件。

1.3.5　使用 IDEA 开发 Scala 项目

IntelliJ IDEA 是一款适用于 JVM 平台的集成开发环境（IDE），它支持 Java、Kotlin、Scala 等语言。Chisel 的官方开发团队推荐使用 IntelliJ IDEA Community 版本（免费）。

本书以 Ubuntu 22.04 为例，介绍如何配置 IntelliJ IDEA 以开发 Chisel 7.3 项目。

1. 软件配置

IntelliJ IDEA 分为 Ultimate 版（收费版）和 Community 版（免费版）。安装 Community 版即可满足学习和科研需求。

（1）在 JetBrains 官网或系统应用商店下载并安装 IntelliJ IDEA Community Edition。

（2）在 GitHub 下载最新版的 Chisel 模板工程：

```
git clone https://github.com/chipsalliance/chisel-template.git
```

（3）启动 IntelliJ IDEA，在菜单栏选择 File → Settings → Plugins，搜索并安装 Scala 插件，安装完成后重启 IDEA。

（4）打开刚刚下载的 chisel-template 工程。依次选择 File → Project Structure → Project，在右侧的 Project SDK 下选择 Add SDK → Download JDK，在弹出的 Download JDK 对话框中选择 Version：17，然后单击 Download 和 OK 按钮，如图 1-1 所示。

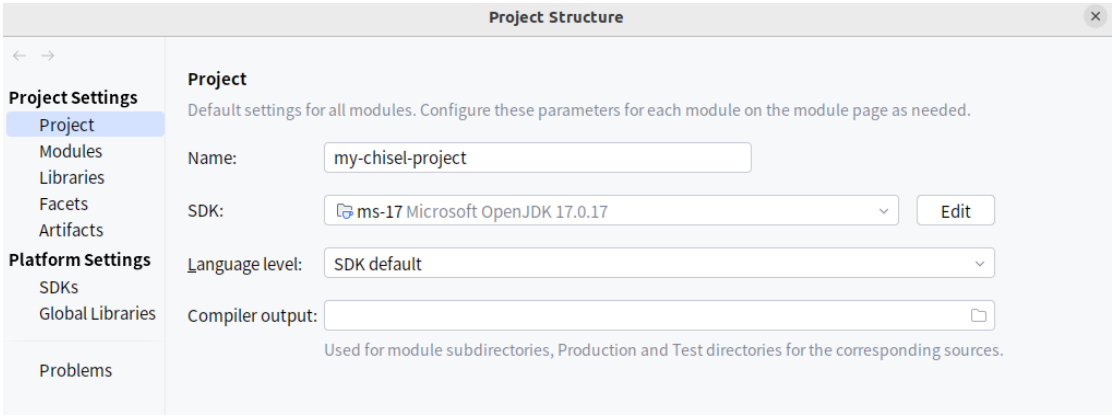

图 1-1　软件配置

（5）在菜单栏选择 Build → Build Project。IDEA 将自动下载 sbt、Scala、Chisel、CIRCT 等依赖包。构建完成后，在底部 sbt shell 中输入 test，若输出 All tests passed 并显示 success，则说明配置成功。

提示：Chisel 7.3 使用 firtool 作为默认后端，无须单独安装。如果希望使用本地版本，可以通过环境变量 CHISEL_FIRTOOL_PATH 指定路径。

（6）对于硬件开发和仿真，还需安装以下工具：Verilator（开源仿真器）、GTKWave（波形查看器）、Git（版本管理）、g++（编译器）。

2. 快捷键操作

IntelliJ IDEA 拥有丰富的快捷键，可在 File → Settings → Keymap 中修改。对于习惯使用 Eclipse、VS Code 或 Sublime Text 的用户，可选择相应的快捷键模板。如果部分快捷键与输入法或系统热键冲突，可在 Keymap 中重新绑定。

3. 运行调试

在新建工程前，依次选择 File → Project Structure → Global Libraries，单击“+”按钮，选择 Scala SDK 并确认安装。然后依次选择 File→New→Project，在弹出的对话框左侧选择 Scala，右侧选择 sbt 或 Mill 构建方式。单击 Next → Finish 按钮即可新建项目。

新建后，在 src/main/scala 目录下创建一个对象文件，例如：

```
object TestRun {
  def main(args: Array[String]): Unit = {
    for(a <- 1 until 6) {
      println(s"Value of a: $a")
    }
  }
}
```

在代码区右击 Run 'TestRun' 或使用快捷键 Ctrl + Shift + F10 运行。

控制台输出如下：

```
Value of a: 1
```

```
Value of a: 2
Value of a: 3
Value of a: 4
Value of a: 5
```

单击行号右侧即可设置断点，在 Debug 模式下使用 F7（Step Into）进行单步调试，底部窗口会显示当前变量的实时值。

1.3.6　总结

本节介绍了 Scala 的安装、使用解释器（REPL）、脚本编写与编译运行方法，并结合 IntelliJ IDEA 完成了从环境配置到工程运行的全过程。

读者可以选择以下 4 种方式运行 Scala 代码：

（1）在解释器中直接编写并运行；

（2）使用脚本文件（.scala）运行；

（3）编译非脚本文件并在解释器中调用；

（4）使用 IntelliJ IDEA 或 sbt 工具构建完整项目。

在 Chisel 7.3 环境下，推荐的现代开发流程如下。

（1）使用 sbt 或 scala-cli 管理依赖与编译。

（2）使用 firtool 生成标准 SystemVerilog 文件。

（3）在 Chisel 7.3 中，官方推荐使用 ChiselSim（基于 svsim）+ Verilator/VCS 等仿真器进行验证。早期的 chiseltest 库仍可用于部分旧工程，但已不再由核心开发团队维护。

（4）通过 IDEA 或 VS Code 进行工程化开发与调试。

这些工具的协同使得硬件设计与软件开发的体验逐渐趋同，真正实现了“敏捷硬件开发”（Agile Hardware Design）的目标。

1.4　章节安排

本书旨在让读者系统掌握如何使用 Chisel 语言实现敏捷硬件开发。由于 Chisel 是构建在 Scala 语言之上的 DSL（领域专用语言），因此学习路径将从 Scala 的编程基础逐步过渡到 Chisel 的硬件建模与生成机制，最终实现硬件系统的设计、验证与工程化。

与第 1 版不同，本书依据 Chisel 7.3 的最新特性和现代工具链（Scala 2.13.17 + sbt 1.9.x + CIRCT/firtool）对全书结构进行了重新规划，使内容更加系统、连贯，既可作为教学中的教材，也适用于工程实践。

全书共 13 章，具体安排如下。

（1）新型敏捷硬件开发语言——Chisel 和 Scala：阐释二者的协同关系，搭建开发入门基础。

（2）Chisel 入门与 Scala 变量函数基础：讲解 Scala 核心语法，完成 Chisel 环境搭建与编译入门。

（3）Chisel 数据类型与 Scala 集合应用：解析硬件数据类型体系，结合 Scala 集合实现灵活的硬件描述。

（4）Chisel 硬件设计中的操作、控制与分层：梳理硬件基础操作，借助 Scala 控制结构实

现分层设计。

（5）Chisel 中的面向对象设计与类型系统解析：运用 Scala 面向对象特性，实现硬件模块化与参数化。

（6）Chisel 硬件设计原语与高级生成器：介绍常用硬件组件，掌握参数化生成器的设计与应用。

（7）Chisel 设计的生成、优化与测试：覆盖 Verilog 生成、性能优化，以及验证方法。

（8）黑盒与多时钟域设计：讲解黑盒调用技巧，解决多时钟域设计与跨时钟域信号同步问题。

（9）Scala 隐式机制与 Chisel 函数抽象：利用 Scala 隐式特性，实现 Chisel 硬件逻辑的高效抽象。

（10）Chisel 工程化开发规范与版本迁移：明确工程化命名与设计规范，提供版本升级实操指南。

（11）riscv-mini 处理器的 Chisel 实现与参数化机制：剖析三阶段流水线架构，理解参数化设计核心。

（12）Chisel 标准库深入与常用设计模式：深挖标准库组件用法，总结工程化常用硬件设计模式。

（13）Chisel 实践与应用案例：通过加速器、混合仿真等实例，落地工程化优化与实践技巧。

通过以上章节安排，读者可逐步掌握 Scala 与 Chisel 的协同使用技巧，理解敏捷硬件开发与传统 HDL 设计的差异，熟练运用 Chisel 7.3 工具生态完成从硬件建模、验证到工程化落地的全流程工作。

1.5 芯片自主创新：历史传承、产业实践与技术路径

在当今全球半导体产业格局中，集成电路已成为国家战略科技力量的核心载体。2023 年，全球半导体市场规模达 5740 亿美元，中国进口芯片价值超 4000 亿美元，凸显自主可控的紧迫性。Chisel 作为基于 Scala 的硬件描述语言，以高级抽象与参数化设计赋能芯片敏捷开发，但自主创新的核心更在于产业生态、人才体系与工程文化的协同构建，以下从三个方面展开分析。

1.5.1 古代工程技术中的系统思维

现代芯片设计的系统工程思维，在中国古代技术中已有雏形。西周偃师的机械人偶，通过凸轮、连杆与齿轮的复合传动实现预设动作，其“复杂行为分解为基本单元”的思路，与现代处理器指令集架构理念相通；东汉水排利用水力驱动鼓风，通过能量转换效率优化满足冶铁需求，与芯片低功耗设计的能效追求异曲同工；北宋水运仪象台以“动力源—传动系统—功能模块”的分层架构，结合擒纵机构实现精准时序控制，其模块化与同步机制对现代 SoC 设计仍具启发。这些案例传递的需求导向、资源约束优化、系统协同等思维，为当代芯片创新提供了方法论借鉴。

1.5.2 芯片产业技术实践的多维观察

中国芯片产业的发展是多环节协同的集体实践。设计环节从依赖架构授权到掌握指令集

定义、流水线优化等核心技术，需通过多代微架构迭代与“约束随机验证+形式验证”的方法学积累；EDA 工具链自主化遵循“点工具突破—平台化整合—核心算法攻关”路径，依赖长期数学基础研究；工艺制造方面，先进节点需通过多重曝光、AI 缺陷检测、统计过程控制等技术，实现精细的良率与稳定性管控；人才培养强调“做中学”模式，需构建跨学科知识体系与工程文化传承机制，形成合理的人才梯队结构。

1.5.3　Chisel 学习与工程实践方法

Chisel 学习需构建“理论—实践—质量保障”的系统化路径。基础层面要夯实数字电路与 Scala 语言核心特性，建立生成式设计思维；实践中遵循“基础组件—模块接口—系统集成—验证环境”的分层路径，强化接口规范与系统协同能力；专业开发需建立统一代码规范，通过形式化验证、性能建模、可测试性设计保障工程质量；同时可通过参与开源项目，从文档改进到核心功能开发逐步积累经验，培养技术深度、系统广度与前瞻性。

芯片自主创新是专业知识、工程经验与系统思维的综合体现，需通过持续积累与协作演进，推动产业生态完善。本书后续章节将提供从基础语法到复杂系统设计的完整知识体系，助力读者夯实专业基础，为投身集成电路自主创新事业做好准备。

1.6　参考文献

[1] MARTIN O，LEX S，BILL V. Scala 编程[M]. 高宇翔，译. 5 版. 北京：电子工业出版社，2022.

1.7　课后练习

1. 书中提到 Chisel 是“嵌入在 Scala 中的语言”。请思考：当我们编写一个 Chisel 程序（如 my_design.scala）并在终端运行它时，计算机的 CPU 实际上是在直接执行“硬件电路逻辑”，还是在执行“Scala 代码”？如果是后者，那么“硬件电路”是在什么时候产生的？

2. 第 1 章提到 Scala 是基于 JVM（Java 虚拟机）运行的。请问：最终生成的 SystemVerilog 文件（.sv）内是否包含 Java 字节码？如果我们将生成的 SystemVerilog 文件交给晶圆厂去流片生产芯片，这颗芯片在通电工作时，还需要 JVM 的支持吗？为什么？

3. 在 Scala 语法中，val 定义的是不可变变量（Immutable）。请根据书中“Scala 是硬件构建语言”的描述推断：当我们用“val a=…”来定义一个硬件组件时，这里的“不可变”是指“这个硬件组件里的电压/数值永远不能变”，还是指“变量 a 永远指向这个特定的硬件组件，不能再指向别的组件”？

4. 书中强调 Chisel 是“硬件构建语言”（Hardware Construction Language），而 Verilog 是“硬件描述语言”。请举例说明：如果我们要设计一个包含 100 个重复模块的电路，使用“硬件描述语言”（Verilog）通常需要怎么做？（提示：是否需要写很多重复代码？）使用“硬件构建语言”（Chisel/Scala）可以利用 Scala 的什么特性（如循环、函数）来“构建”这 100 个模块？这体现了 Chisel 的什么优势？

5. 根据 1.2 节所述，Chisel 7.3 引入了 CIRCT 项目和 firtool 工具。请简述：从我们编写的“.scala”源代码，到最终生成工业标准的“.sv”（SystemVerilog）文件，中间经历了哪些

主要步骤？为什么要引入 firtool 而不是直接由 Scala 打印出 Verilog？

6. Scala 语言中有 if-else 语句和 for 循环语句。请根据“生成器”的概念推断：如果在 Chisel 代码的“for (i <- 0 until 4)”循环中定义了硬件逻辑，那么生成的电路中会包含一个“正在循环执行的计数器”吗？还是会并行地生成 4 份同样的硬件结构？

7. 书中引用了 Java 之父 James Gosling 对 Scala 的评价，并提到 Scala 可以无缝调用 Java 类库。请思考：在 Chisel 的“构建阶段”（生成电路图的时候），我们能否使用 Scala/Java 的数学库（如“Math.sin”）来预先计算好一个正弦波查找表（ROM）的初始值？这给硬件设计带来了哪些便利？

8. 1.1 节提到 Scala 是一门“面向对象”的语言。Verilog 虽然也有模块（Module），但缺乏继承、多态等特性。请结合书中 RISC-V 的案例思考：如果我们要设计一系列处理器（有的带浮点运算单元，有的不带；有的带缓存，有的不带），使用 Scala 的“类继承”特性相比于传统 Verilog 的“宏定义”（ifdef）或“复制–粘贴”有什么潜在优势？