约定

《约定》前言

在程序员的日常生活中，我们不断面对代码、接口、配置、文档，试图将混乱的世界编织为有序的系统。而这种秩序的建立，几乎全靠一个概念支撑：约定（Convention）。

写下一个函数，是一次约定；设计一个类，是一次约定；向同事交付接口，是一次约定；启动一个微服务，是多个系统间的约定在共同生效。

本书的写作缘起，正是来自一个朴素的问题：为什么有些人能快速掌握系统本质，而有些人始终感到混乱？

我逐渐意识到，这不仅是技术熟练度的问题，而是——

是否拥有“识别约定、理解约定、制定约定”的能力。

编程不是无规则的自由表达，而是规则中构建秩序的艺术。

本书试图从编程的细节出发，系统梳理我们在语言、框架、协作、系统、架构中所依赖的各种显性与隐性约定，并引导读者从“模糊”走向“确信”。

愿它成为你思考系统之美、代码之清晰、团队之协作的桥梁。

写代码，是表达；守约定，是尊重；建系统，是构建秩序。

愿你在混乱中看见规则，在规则中获得自由。

📚《约定》目录

内容

《约定》前言 1

第 1 章：从模糊到确信的起点 7

1.1 从“起名”说起 8

1.2 什么是“约定”？ 8

1.3 为什么“约定”如此重要？ 9

1.4 本书的目的与结构 9

1.5 约定是一种思维方式 10

第 2 章：什么是“约定”？ 10

2.1 日常生活中的约定 10

2.2 编程中的“约定” 10

2.3 约定的特征 13

2.5 小结 14

第 3 章：计算机语言是一系列“约定” 14

3.1 自然语言与形式语言的对比 14

3.2 编程语言的“约定本质” 15

3.3 为什么编程必须依赖约定？ 16

3.4 语法也是约定：强制规则的体现 17

3.5 语义也是约定：动作的含义 17

3.6 小结 18

第 4 章：约定的基本单位 —— 类、变量、方法、语句 18

4.1 类是一种约定 19

4.2 变量是一种约定 20

4.3 方法是一种约定 21

4.4 语句是一种约定 22

4.5 小结 23

第 5 章：约定的描述叫文档 24

5.1 文档的本质：约定的“名文化” 24

5.2 Java 中的文档体系 25

5.3 文档是一种约定的共识中介 26

5.4 除了 JavaDoc，还有哪些文档？ 27

5.5 如何写出好的文档 28

5.6 小结 29

第 6 章：Spring Framework —— 约定优于配置的实践 29

6.1 什么是“约定优于配置”？ 30

6.2 项目结构的默认约定 30

6.3 自动配置与组件扫描 31

6.4 Web 开发的路由约定 32

6.5 数据库访问的约定 32

6.6 配置文件中的默认项 33

6.7 自定义 Bean 的默认注入方式 33

6.8 Spring Boot 的自动装配原理（深入） 34

6.9 约定带来的益处 35

6.10 小结 35

第 7 章：框架与库中的“隐形约定” 36

7.1 Hibernate 与 JPA 的命名与映射约定 36

7.2 JSON 序列化的属性约定（Jackson） 37

7.3 RESTful API 的方法与语义约定 38

7.4 前后端开发中的非明确约定 39

7.5 JavaScript 框架中的约定（Vue、React） 40

7.6 跨平台工具中的约定 40

7.7 如何识别与应对隐形约定？ 41

7.8 小结 41

第 8 章：如何制定团队中的约定 42

8.1 为什么需要团队约定？ 42

8.2 团队约定的典型内容 43

8.3 怎么制定这些约定？ 44

8.4 如何写一份“约定文档”？ 44

8.5 如何让大家遵守？ 45

8.6 小结 46

9.1 什么是架构层面的约定？ 47

9.2 接口层的约定：API 契约 48

9.3 服务之间的协作约定 48

9.4 事件驱动架构中的约定 49

9.5 系统治理与元规则的约定 50

9.6 架构图与文档：可视化约定 50

9.7 如何推动架构约定落地？ 51

9.8 小结 51

第 10 章：约定的哲学 —— 程序世界的秩序之源 52

10.1 程序员的世界：一场人与规则的博弈 52

10.2 什么是“约定哲学”？ 52

10.3 为什么“自由编码”不是理想？ 53

10.4 编程世界中的“秩序创造模型” 53

10.5 人类社会也是“约定驱动”的 54

10.6 如何建立“约定思维”？ 55

10.7 结语：约定即秩序，秩序生自由 55

第 11 章：互联网是一个约定的世界 56

11.1 网络世界的“契约精神” 56

11.2 IP 与 DNS：互联网的户籍系统 57

11.3 TCP/IP 与可靠性协议 58

11.4 HTTP 与 Web 的会话契约 58

11.5 HTML/CSS/JS 的前端约定 59

11.6 搜索引擎与 robots.txt：协议之外的默契 59

11.7 HTTPS 与信任体系 60

11.8 WebSocket、gRPC 与新型协议 60

11.9 分布式与服务间的契约 60

11.10 哲学视角：秩序的隐形编织 61

第 1 章：从模糊到确信的起点

在学习编程的旅途中，我们经常会遭遇各种各样的"模糊"。这种模糊，可能来自语言的语法不熟悉、框架的行为不理解、工具的使用方式不明确，也可能来自人与人之间协作时的误解。这种模糊常常令人焦虑，使我们感到无所适从，甚至怀疑自己是否适合这条道路。

但有一种力量，能带我们从模糊走向确信，那就是：**约定**。

1.1 从“起名”说起

我们在《从模糊到确信》中讲到，一个重要的清晰化过程，是“给事物起名”。命名，是对混沌世界的第一次理解和掌控，是心智对信息的第一次组织。

而“约定”，正是这种命名行为的延伸。

在计算机世界中，“约定”是一种比“命名”更具结构性的存在。它不仅包含了名称，还规定了行为、格式、顺序、权限、职责。

你写下 public static void main(String[] args)，这不仅是起名，而是一种与 Java 虚拟机之间的“约定”——你遵守它，程序才能运行；你违反它，编译器就会拒绝你的代码。

1.2 什么是“约定”？

约定，是一种双方达成的协议，在人类社会中，它是社交的基础；在软件系统中，它是协作的基石。

约定可以是口头的、书面的、默认的，也可以是强制的。它们的共同点是：**一旦形成，就必须被遵守，否则后果自负。**

例如：

- 你和朋友约好晚上 7 点见面，这是口头约定。
- 程序员之间约定函数名用小写驼峰，这是社区习惯的约定。
- Java 语法规定类名必须与文件名一致，这是强制性的语言约定。

本书中，我们主要探讨的是**计算机语言与开发过程中的各种约定**，它们涵盖了：

- 编程语言本身的语法与语义
- 编程风格与命名规则
- 框架和工具的默认行为
- 团队协作中的规范文档
- 自动化构建和部署流程的目录结构

1.3 为什么“约定”如此重要？

没有约定的世界，是混乱的。

- 你无法预测函数名是否能正常调用
- 你不清楚文件应该放在哪个目录才会被识别
- 你不确定框架是否能自动加载这个组件

而有了约定之后，一切变得有迹可循：

- 只要放对位置、写对名字、用对语法，一切就能自动运行
- 团队成员之间可以无缝协作，不用每次都写文档解释接口
- 编译器、构建工具、运行环境都知道该怎么做

这就是“约定优于配置”（Convention over Configuration）所强调的精神：**让规则成为习惯，让协作变得高效**。

1.4 本书的目的与结构

本书旨在揭示隐藏在程序世界背后的“约定系统”。这些约定像看不见的道路，把程序员从一个模糊的草地，引向一条清晰可行的公路。

我们不仅要告诉你“有哪些约定”，更要告诉你：

- 它们为何存在？
- 它们是怎样被人类和机器共同遵守的？
- 它们又怎样影响了编程的每一个角落？

我们将会逐步从最基础的语言规则，到项目结构的组织方式，再到框架级别的默认机制，甚至再往外拓展到团队协作和企业开发规范，构建出一个“约定”的地图。

在阅读这本书的过程中，你会学会：

- 识别约定：哪些地方隐含了重要规则？
- 利用约定：怎样减少重复劳动，提升效率？
- 制定约定：在团队中建立良好的协作规范

1.5 约定是一种思维方式

最后，我们希望你明白，**约定不仅是一种工具，更是一种思维方式。**

当你开始主动寻找约定、依赖约定、利用约定，你就不再是一个“模糊的程序员”，你开始进入“确信的程序世界”。

本书的全部内容，都是在帮助你回答一句话：

"面对这个不确定的系统，我是否可以确信它会如我所愿地运作？"

而回答的方式，正是：我是否理解并遵守了它的**约定**。

第 2 章：什么是“约定”？

2.1 日常生活中的约定

我们每个人每天都在使用“约定”，即使我们没有意识到。

早上八点上班，是一种时间约定；红灯停绿灯行，是一种交通规则约定；写电子邮件要写主题、称呼、正文，是一种交流格式的约定。

“约定”这个词，在日常生活中意味着：**双方对某个行为达成的共识**。

- 它可以是口头的，也可以是书面的
- 可以是正式的合同，也可以是默认的习惯

比如：

- “我们下周四开会” 是一种口头时间约定
- “请使用模板写日报” 是一种团队工作格式约定

它们存在的意义是：**减少沟通成本、提高预期确定性、避免混乱**。

2.2 编程中的“约定”

编程世界里，同样充满了“约定”，甚至可以说：

编程语言，就是一整套约定的集合。

在程序中，约定更像是“规则”或“合同”：你必须遵守它，否则程序无法运行，或者运行结果与你所期待的不一样。

这些约定可以分为几种层级：

（1）语言级约定（语法与语义）

- 变量声明必须指定类型（在 Java 中）
- 方法名必须唯一（在同一个类中）
- 控制结构如 if 和 for 有固定格式

例如：

```
int age = 18;  // 约定：变量必须声明类型
if (age > 18) {
    System.out.println("成年人");
}  // 约定：条件表达式必须加括号，语句块必须加花括号
```

（2）风格级约定（命名和排版规范）

- 类名首字母大写，驼峰格式：StudentInfo
- 方法名小写开头，驼峰格式：getName
- 常量全部大写，用下划线分隔：MAX_SIZE

例如：

```
public class StudentInfo {
    private String name;

    public String getName() {
        return name;
    }
}
```

这类约定不影响程序运行，但影响可读性与协作性。好的命名习惯，可以让他人无需解释就理解代码含义。

（3）结构级约定（文件与目录）

- Java 类文件必须与类名相同
- 主类文件必须在 src/main/java 中
- 测试文件必须在 src/test/java 中

```
src/
  main/
    java/
      com/example/DemoApplication.java
  test/
    java/
      com/example/DemoApplicationTest.java
```

如果你打破这个结构，Maven 或 Gradle 等构建工具可能就找不到对应的类。

（4）框架级约定（Spring、Hibernate 等）

Spring Boot 的很多功能依赖“命名”和“注解”的约定：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello!";
    }
}
```

在这里：

- @RestController 是告诉框架：这是 Web API
- @GetMapping("/hello") 是告诉框架：这个方法处理 GET 请求

如果不写注解，Spring 就不知道该怎么做。

2.3 约定的特征

特征一：可预期性

如果遵守约定，行为结果就应该是可预测的。

比如：你写了一个类 DemoApplication，只要你放在正确目录下，并加上 @SpringBootApplication 注解，它就会成为应用的启动类。这种稳定性来源于约定。

特征二：高协作性

当多人协作开发时，大家都遵守相同的约定，代码才不会混乱。

比如：你接手一个项目时，看到测试代码都在 src/test/java 中，配置文件都在 resources 文件夹中，你会很快上手，因为这些都是共识。

特征三：强制与非强制并存

有些约定是语言规定的（不遵守就编译错误），有些是建议性的（不遵守也能跑，但不推荐）。

类型	示例	是否强制
语法	if () {}	强制
命名	类名大写	非强制
注解	@Autowired	框架约定（强制）
结构	src/main/java	工具约定（强制）

2.4 约定与模糊的关系

模糊的地方，就是没有共识的地方。

比如：

- 不明确类名代表什么 → 命名不清晰

- 不知道这个文件是干什么的 ➡ 结构不清晰
- 方法返回值不明 ➡ 类型不清晰

这些问题的根源，正是**缺乏约定**。

一旦我们建立起一套合理的约定，代码就有了规则，协作就有了秩序，错误就可以被预防，思维也会更加清晰。

2.5 小结

在本章中，我们初步定义了“约定”的意义和分类，说明了它在编程中的核心作用。我们也认识到：

- 编程语言本质上是规则集合
- 程序运行依赖于这些规则的遵守
- 程序员之间的协作依赖于共识和规范

下一章，我们将从更基础的角度开始探讨：

计算机语言为何必须依赖约定，它与形式语言之间的关系是什么？

我们将进入“语言本质与约定结构”的领域。

第 3 章：计算机语言是一系列“约定”

3.1 自然语言与形式语言的对比

在我们学习编程语言之前，先来看看我们熟悉的自然语言。

自然语言是模糊而灵活的，而计算机语言则是精确而严格的。

例如：

- 你对朋友说：“下午我们去那边那个地方吃点东西吧？”
    - 朋友可能知道你指的是那家常去的拉面馆
- 你对计算机说：“运行点什么吧？”
    - 抱歉，它会一脸茫然地报错

自然语言的特点：

- 容错性高（说错也能猜个大概）
- 上下文依赖强（得靠对方补全意义）
- 歧义多（同一句话可多种解释）

形式语言的特点：

- 严格的词法规则（必须写成“对”的格式）
- 精确的语法结构（缺一个分号都不行）
- 不能含糊、不猜测、不冗余

编程语言就是一种“形式语言”，它的目的不是沟通，而是控制计算机执行。

计算机不能理解人类的“模糊语言”，所以我们必须用一种形式化的表达来“说清楚每一步”。

这就带来了一个问题：

怎样让程序员“规范地说话”？

答案是：**制定规则，也就是制定“约定”。**

3.2 编程语言的“约定本质”

如果你仔细观察一门编程语言，你会发现它几乎全是各种各样的“规则集合”：

- 关键字（if, while, class）的使用方式
- 数据类型（int, double, boolean）的约定
- 函数和变量的定义规则
- 运算符的含义和优先级
- 程序的结构和作用域规则

举个例子：

```
public class Hello {

    public static void main(String[] args) {
```

```
        System.out.println("Hello, world!");
    }
}
```

这个程序可以运行，是因为它满足了一整套“约定”：

1. 类名必须与文件名一致（Hello.java）
2. main 方法必须是 public static void 的形式
3. 参数类型是 String[] args
4. 所有语句必须写在方法体 {} 里
5. 每条语句必须以 ; 结尾

你哪怕错一个，比如不加 static，都不会编译通过。

所以我们可以说：编程语言 = 一套语法规则 + 一套语义约定

这些“语法”与“语义”，本质上就是约定的集合。

3.3 为什么编程必须依赖约定？

1. 让机器理解我们的意思

计算机只能识别明确无误的命令。如果你模糊表达，它根本“不会猜”。

所以必须有规则告诉它：

- 什么叫变量声明
- 什么叫方法定义
- 什么是控制语句

2. 保证程序员之间协作的一致性

一个人可以随便写代码，但当团队协作时，如果每个人的命名、缩进、文件结构都不同，会导致代码不可维护。

统一的命名规范、类结构、目录层级，都是“约定”的体现。

3. 提高效率（框架与工具能自动识别）

如果你遵守了 Maven 的目录结构，它就能自动找到源代码与测试代码；
如果你在 Spring 中遵守了注解规则，它就能自动加载你的 Bean。

这些都是约定给你带来的效率红利。

3.4 语法也是约定：强制规则的体现

编译器是什么？

编译器其实就是“约定检查器”。

- 它检查你有没有写错拼写（词法约定）
- 检查你有没有写错结构（语法约定）
- 检查你调用的是否是合法的东西（语义约定）

你写下：

```
int x = "abc";
```

编译器立刻提示：类型不匹配。这说明它在检查你的代码是否遵守类型系统的约定。

再比如：

```
if x > 0:
    doSomething()
```

在 Java 中，这种写法根本不合法，因为它没有遵守 Java 的 if（条件）{语句} 的结构约定。

所以语法不是随便的，它是语言设计者制定的“法律文本”。

3.5 语义也是约定：动作的含义

不仅是怎么写，还有写了之后“是什么意思”也是一种约定。

举个例子：

```
int x = 5 / 2;
```

这个表达式的结果是 2，而不是 2.5，为什么？

- 因为在 Java 中，整数除以整数的结果还是整数。这是一种**隐性约定**。

如果你要得到小数结果，必须显式地告诉它：

```
double x = 5.0 / 2;
```

这时候结果才是 2.5。

再比如：

```
String s1 = "abc";

String s2 = new String("abc");

System.out.println(s1 == s2);
```

输出是 false，不是因为“内容不同”，而是 == 比较的是引用地址，这又是一种语义层级的约定。

很多初学者的困惑，往往来自“语义约定”不了解。

3.6 小结

我们本章论证了一个核心观点：

编程语言就是一系列被定义好的“约定”，语法是强制性的约定，语义是行为上的约定。

这些约定是：

- 与机器之间达成的协议（你写法对，它才执行）
- 与人之间达成的共识（大家都这么写，才易于协作）
- 与工具之间达成的契约（自动构建与部署的前提）

在下一章中，我们将深入每一种具体的语言结构：类、变量、函数、语句……看看它们各自是如何体现“约定”的。这将是对编程语言“约定解剖”的第一步。

第 4 章：约定的基本单位 —— 类、变量、方法、语句

在本章，我们将系统地拆解编程语言中的核心结构，逐个分析它们内部隐藏的“约定”。你将会看到：**每一个语法元素，不只是写法的规则，更是“意义”的约定，是程**

序构建的最小单元。

我们以 Java 语言为例，逐步分析类（Class）、变量（Variable）、方法（Method）、语句（Statement）等要素所体现的约定。

4.1 类是一种约定

类是什么？

类（Class）是面向对象编程的核心结构，它是一种**模板**，定义了对象应该具备哪些数据（属性）和行为（方法）。

```
public class Person {

    private String name;

    private int age;


    public void sayHello() {

        System.out.println("Hello, I am " + name);

    }

}
```

类的约定体现在哪？

1. **文件名必须与类名一致（大小写敏感）**
   Person.java 中必须定义 public class Person，否则编译失败。
2. **类名首字母大写（命名约定）**
   Java 社区默认约定，便于区分类与方法。
3. **构造器命名必须与类名一致（结构约定）**

```
public Person(String name, int age) {

    this.name = name;

    this.age = age;
```

}

4. **类只能有一个 public 修饰符的外部声明（文件级约定）**
5. **类中定义的成员可见性、作用域等规则，属于语义级约定**

类是一份“对象的合约”。对象被创建时，自动继承这份“约定”。

4.2 变量是一种约定

变量是存储数据的容器，但它并不是任意的。变量的命名、作用域、类型、初始化等，都蕴含着大量的“约定”。

示例：

```
int score = 100;

String name = "Alice";

final double PI = 3.1415;
```

常见变量约定：

规则	**说明**	**示例**
类型必须明确	Java 是强类型语言	int score
命名应语义清晰	不要用 a, b, x1	String userName
常量大写加下划线	final 值不可变	PI, MAX_SIZE
初始值应匹配类型	类型不匹配无法编译	int x = "abc"; // 错误

作用域遵守块级规则 在 {} 内声明的变量只在其中有效 for (int i = 0; ...)

特殊变量约定：

- 静态变量（static）属于类，不属于对象
- 全局变量（public static final）要谨慎使用

变量的设计约定，不只是为了程序正确执行，更是为了让“代码含义清晰、行为可预

期”。

4.3 方法是一种约定

方法（Method）是行为的定义，它不仅包含执行的语句，还定义了输入与输出的规范，是程序员之间协作的“契约”。

示例：

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

方法的约定包括：

项目	约定说明
方法名	使用小驼峰格式，如 getName
参数名	使用有语义的词，如 price, quantity
返回值	明确指定返回类型，不可模糊
void	无返回值的方法必须声明 void
方法注释	推荐使用 JavaDoc 注解方法目的与参数

重载与覆盖的约定：

- **重载（Overload）**：同名不同参
- **覆盖（Override）**：子类重写父类方法，必须加 @Override

```
@Override
public String toString() {
    return "Person: " + name;
}
```

@Override 是一种**显式约定**，告诉编译器你有意重写它。如果不加，写错签名也不会报错，可能引发 bug。

方法签名约定：

- 访问修饰符（public/private） + 返回类型 + 方法名 + 参数列表
- 不支持默认参数（不像 Python），所以通常使用重载

方法的存在，就意味着：我承诺，如果你传入这些参数，我将返回某种结果。这是一种契约思维。

4.4 语句是一种约定

程序是由语句构成的。每一句代码语句，背后都是一种“执行的约定”。

基本语句约定：

- 所有语句必须以分号 ; 结尾

```
int a = 5;
System.out.println(a);
```

- 多条语句必须包裹在代码块 {} 中

```
if (a > 0) {
    System.out.println("正数");
    a--;
}
```

- 条件表达式必须用括号包住

```
while (true) {
    break;
}
```

结构性语句的约定：

结构	关键词	约定
条件判断	if, else	必须加 () 与 {}
循环	for, while	条件格式严格
跳出	break, continue	只能用于循环结构内
分支选择	switch	case 必须带 break（除非故意穿透）

错误示例（违反约定）：

```
if x > 0
    print(x);
```

以上在 Java 中完全非法，必须写成：

```
if (x > 0) {
    System.out.println(x);
}
```

每一句语法，都不是“随意的自由”，而是“规则下的创造”。

4.5 小结

本章中，我们逐一分析了 Java 编程语言的几个基础构造元素，并揭示了它们背后的“约定精神”：

- 类是关于“结构”的约定
- 变量是关于“数据表达”的约定
- 方法是关于“行为契约”的约定
- 语句是关于“执行逻辑”的约定

这些看似微小的书写规范，其实构成了整个语言的骨架，定义了编译器的判断逻辑、工具的智能分析方式、程序员之间的合作方式。

下一章，我们将继续探索这些“约定”的扩展形式——**当它们被抽象为“文档”时，如何在人与人之间传达清晰的规则？**

第 5 章：约定的描述叫文档

我们已经看到，程序世界的每一部分都充满了“约定”：语言的语法、类的结构、方法的使用、框架的运作，甚至目录和命名，也都是一种默契与规矩。

但机器不需要“注释”，它只执行代码。

那么，“文档”到底是给谁看的？它存在的意义又是什么？

答案是：**文档是程序的“人类解释版本”，是给人看的约定说明书。**

在这一章，我们来揭示文档与约定的关系，以及文档的种类、规范与写法。

5.1 文档的本质：约定的“名文化”

文档，就像是语言中“约定”的翻译官。

你可能写了一个方法：

```
public int getTotalPrice(int quantity, int pricePerUnit) {
    return quantity * pricePerUnit;
}
```

从代码本身可以理解逻辑，但读者仍然可能有这些疑问：

- quantity 是什么单位？是件数？还是重量？
- pricePerUnit 是人民币还是美元？是否含税？
- 会不会抛出异常？传负数怎么办？

这些问题不能靠代码表达，只能靠人写出来给人看，这就是**文档的职责**。

代码是写给机器执行的，文档是写给人理解的。

代码是动作，文档是解释。

就像法律条文是约定，而法律说明书和案例注释是文档。

5.2 Java 中的文档体系

5.2.1 JavaDoc 注释

Java 中最重要的文档机制是 JavaDoc。
它是一种特殊格式的注释，可以被工具解析为 HTML 文档。

示例：

```
/**
 * 计算总价
 * @param quantity 商品数量，单位为件
 * @param pricePerUnit 每件商品的单价，单位为元
 * @return 返回总价，单位为元
 */
public int getTotalPrice(int quantity, int pricePerUnit) {
    return quantity * pricePerUnit;
}
```

- @param：说明参数含义
- @return：说明返回值
- @throws：说明可能抛出的异常（如果有）

优点：

- 自动生成网页文档
- IDE（如 IntelliJ, Eclipse）可自动显示提示

约定：

- 方法的每个参数都应写 @param

- 所有公共方法都应写注释
- 注释内容应清晰、简洁、避免重复代码

5.2.2 类文档注释

```
/**
 * 表示一个用户对象
 * 包含用户名、年龄等信息
 */
public class User {
    ...
}
```

文档应回答：这个类是干什么的？在系统中扮演什么角色？是否有特殊限制？

5.3 文档是一种约定的共识中介

文档连接人类与代码的思维差距

程序员写代码，读者读的是“意图”。
而机器不关心你的意图，只看语法对不对。

文档的作用就是：**把代码的目的、假设、限制、行为写清楚**，避免误用。

文档是多人协作的前提

没有文档，代码就是黑盒。一个团队如果没有公共的接口文档、模块说明、目录结构说明，很容易出错：

- 不知道参数含义
- 搞错了调用顺序
- 重复造轮子

文档是测试和设计的基础

好的文档，可以作为测试编写的基础——你知道函数应该做什么，就知道该测什么。

甚至，**测试用例本身也可以是一种文档**。

5.4 除了 JavaDoc，还有哪些文档？

5.4.1 README 文件

项目根目录中的 README.md 是“入口文档”，告诉他人这个项目是做什么的，怎么运行。

典型内容：

- 项目介绍
- 安装步骤
- 使用示例
- 依赖版本
- 作者联系方式

5.4.2 API 文档

用于描述接口之间的调用规则，尤其是前后端分离架构中。

- 请求方法（GET/POST）
- 路由路径（/api/user/{id}）
- 请求参数格式
- 返回数据结构
- 错误码含义

可通过 Swagger 自动生成，也可用 Postman 文档共享。

5.4.3 设计文档（Design Doc）

- 模块设计说明书
- 架构决策记录（ADR）

- 数据库结构设计
- 枚举说明、业务规则说明等

这些文档的共同目标是：**用文字清晰表达系统背后的逻辑和边界**。

5.5 如何写出好的文档

避免冗余

坏的文档：

```
/** 设置名称 */
public void setName(String name) {
    this.name = name;
}
```

这就是重复了代码，属于“无效注释”。

补充隐藏信息

好的文档：

```
/**
 * 设置用户名，长度不能超过 20 个字符，不能包含空格
 * @param name 用户名，非空
 * @throws IllegalArgumentException 如果参数不合法
 */
public void setName(String name) {
    ...
}
```

它提供了代码没有直接体现的“行为边界”与“前置条件”。

写给“使用者”而不是写给“自己”

文档不是给你自己看的，是给未来阅读你代码的人看的，哪怕那个“未来的人”也是现在的你。

示例、格式、清晰结构

- 给出调用示例
- 语句简洁，避免废话
- 用列表、段落、强调来分清重点

5.6 小结

本章我们讲解了文档的本质，它是“约定的文本形式”，是人与代码之间的桥梁，是约定可以被传达的前提。

我们介绍了：

- JavaDoc 的使用与约定规范
- 各种开发文档的分类与作用
- 如何写出有用、清晰、有意义的文档

在后面的章节，我们将进入框架和项目实践层面，看一看 Spring Framework 等技术体系是如何将“约定”扩展为整个系统自动化和配置的基础的。

写代码是创造，写文档是对创造的解释与共享。
只有共享的规则，才能创造协作的奇迹。

第 6 章：Spring Framework —— 约定优于配置的实践

在前几章中，我们已经理解了“约定”作为编程语言和协作工具的底层机制，而到了现代开发框架中，这一理念被发扬到了极致。

最具代表性的框架之一就是 **Spring Framework**，特别是其子项目 **Spring Boot**。

它们的核心哲学之一就是：

约定优于配置（Convention over Configuration）

本章我们将通过实战的方式，一步步揭示 Spring 是如何用“约定”来大幅降低配置负担、提高开发效率的。

6.1 什么是“约定优于配置”？

传统的 Java 开发常常需要繁琐的 XML 配置：

- 每个 Bean 需要手动注册
- 每个路径需要声明映射
- 每个数据库需要声明连接参数

而 Spring Boot 提供了一种机制：

如果你按照它的“默认方式”来命名、组织、注解，它就会自动配置好一切。

如果你不想用默认的，也可以手动配置，但默认就能跑，这就是它的魔力。

6.2 项目结构的默认约定

```
src/
  main/
    java/
      com/example/demo/DemoApplication.java
    resources/
      application.properties
```

- DemoApplication.java 是主类，必须位于根包下
- resources/ 目录放置配置文件、模板、静态资源
- application.properties 是默认配置文件，Spring Boot 会自动读取

你不需要告诉 Spring 哪是配置目录，它通过“路径约定”就知道了。

6.3 自动配置与组件扫描

默认扫描规则

主类所在的包会自动成为扫描根目录：

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

只要你写在这个包及其子包下的类：

```
@Component
public class MyService {}
```

Spring Boot 会自动注册成 Bean。

约定：你把组件写在主类子包下，并加上正确注解，它就会被扫描。

注解驱动的约定

注解	含义
@Component	通用组件
@Service	业务逻辑层
@Repository	数据访问层
@Controller	Web 控制器

它们只是语义不同，但底层功能一样。这是一种 **命名上的约定**，帮助开发者理解结构。

6.4 Web 开发的路由约定

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello!";
    }
}
```

- @RestController 表示这是一个返回 JSON 的控制器（自动包含 @ResponseBody）
- @GetMapping("/hello") 表示处理 GET 请求 /hello 的方法

这些注解是一种约定性的描述，告诉框架“怎么处理这个类和方法”。你不需要手动写 Servlet 或 XML 映射。

默认端口

- Spring Boot 默认使用端口 8080
- 如果你不改配置，访问 http://localhost:8080/hello 就能看到结果

6.5 数据库访问的约定

Spring Data JPA 提供了一种强大的“基于方法名”的约定查询：

```
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByName(String name);
    List<User> findByAgeBetween(int min, int max);
}
```

你不需要写 SQL，方法名就代表查询逻辑：

- findByName → SELECT * FROM user WHERE name = ?
- findByAgeBetween → WHERE age BETWEEN ? AND ?

这是约定的极致表现：方法命名 = 查询语句

当然，如果你不喜欢这种方式，也可以自己写 @Query 来配置查询逻辑。

6.6 配置文件中的默认项

Spring Boot 使用 application.properties 或 application.yml 进行配置。

但你不写任何配置，它也能跑。

示例：无需配置即可运行的 Web 项目

- 默认端口：8080
- 默认日志：console 输出
- 默认模板路径：resources/templates
- 默认静态资源路径：resources/static

如果你想改变行为，只需要覆写默认值：

```
server.port=9090
spring.application.name=myapp
```

配置的存在不是必须的，而是可选项。这就是“优于配置”的精髓。

6.7 自定义 Bean 的默认注入方式

```
@Service
public class OrderService {
    private final OrderRepository orderRepository;
```

```
    public OrderService(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }
}
```

- 构造函数只有一个时，Spring 会自动注入参数
- 不用写 @Autowired，也能正常注入

这是 **“构造函数注入的简化约定”**，推荐用法比冗长配置更清晰。

6.8 Spring Boot 的自动装配原理（深入）

Spring Boot 会在启动时自动加载许多配置类：

- 例如：WebMvcAutoConfiguration，DataSourceAutoConfiguration 等
- 条件注解如 @ConditionalOnClass，@ConditionalOnMissingBean 决定是否启用

这些类是通过 spring.factories 配置加载的：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,
...
```

开发者不需要关心这个过程，只需“符合条件”就能自动生效。

“只要你符合条件，框架就给你装好一切。” 这就是基于“约定”的自动化。

6.9 约定带来的益处

优点	说明
降低配置负担	写更少的代码和配置文件
减少出错机会	避免手动 wiring 的错误
更清晰的项目结构	遵循统一标准易于协作
学习门槛更低	新人只要遵守格式就能上手
有助于生成文档和测试	注解 + 命名有利于自动化处理

6.10 小结

Spring Framework，尤其是 Spring Boot，是“约定优于配置”的现代实践典范。

它的设计理念是：

- 如果你不说，我就用默认
- 如果你想说，我也支持你定制
- 但你要改变默认，得清楚告诉我

通过这一机制，Spring 实现了：

- 自动化组件扫描
- 控制器与路由的快速建立
- 数据库访问逻辑的自动推导
- 构建与部署流程的零配置起步

你写得少，做得多——这不是魔法，而是“约定”的力量。

下一章我们将拓展到更广阔的编程世界：看一看在 Web 开发、前后端协作、API 设计、持续集成等领域中，还有哪些“隐形的约定”在背后支撑着系统正常运行。

第 7 章：框架与库中的“隐形约定”

如果说 Spring 是显式约定的大师，那么在许多其他框架与第三方库中，还有大量“隐形”的、**非强制却影响深远的约定**，它们不一定写在文档里，但不遵守就容易出错或产生误解。

这些隐形约定广泛存在于：

- ORM 框架（如 Hibernate, JPA）
- JSON 序列化库（如 Jackson, Gson）
- 前后端接口（如 REST, OpenAPI）
- Web 前端与后端的协同开发

本章我们将深入揭示这些“沉默的契约”，并教你如何识别它们、利用它们。

7.1 Hibernate 与 JPA 的命名与映射约定

默认字段命名映射

在使用 JPA 时，我们常写：

```
@Entity
public class User {
    @Id
    private Long id;
    private String email;
    private LocalDateTime createdAt;
}
```

没有配置的情况下，JPA 会默认做如下假设：

- 表名为 user
- 列名为 id, email, created_at

为什么最后一个字段变成了下划线形式？

因为 JPA 实现（如 Hibernate）会默认将 Java 的驼峰命名转换为数据库的下划线风格，这是一个“隐式命名策略”。

如果你不理解这个规则，可能会导致数据库建表结构与代码不匹配。

你可以用注解显式说明：

```
@Column(name = "created_at")
private LocalDateTime createdAt;
```

表名大小写敏感

某些数据库（如 PostgreSQL）会区分大小写，这时 User 与 user 是不同表。
JPA 默认使用类名的小写作为表名，但你可以指定：

```
@Table(name = "User")
```

7.2 JSON 序列化的属性约定（Jackson）

当你用 Spring Boot 开发 REST API 时，Jackson 默认负责对象与 JSON 的转换。

你写下：

```
public class Product {
    private String productName;
    private int price;

    // getters and setters
}
```

返回的 JSON 默认是：

```
{
  "productName": "Book",
```

```
  "price": 1000
}
```

这是因为 Jackson 遵循以下约定：

- 字段名根据 getter 方法推导出来
- 属性名直接映射为 JSON 键
- 支持驼峰形式（可通过配置改为下划线）

你也可以用注解更改它：

```
@JsonProperty("product_name")
private String productName;
```

隐含的约定：如果没有提供 getter/setter，属性可能不会被序列化。

还包括：

- null 值默认会序列化除非你关闭
- 日期格式默认是 Timestamp，非 ISO8601

7.3 RESTful API 的方法与语义约定

REST 本身就是一套“资源导向的约定协议”。

常见的 URL + 方法约定：

动作	方法	路径
查询资源列表	GET	/users
查询单个资源	GET	/users/{id}
创建新资源	POST	/users
更新资源	PUT	/users/{id}

动作	方法	路径
删除资源	DELETE	/users/{id}

如果你在 GET 请求里提交数据，或者用 POST 做查询，就违反了 REST 的约定，虽然可以工作，但不符合设计规范。

框架如 Spring RESTController 也默认会据此行为映射：

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) {...}
```

7.4 前后端开发中的非明确约定

很多前后端项目的失败，不是因为技术实现出错，而是因为**缺乏清晰的“默认协作规则”**。

常见的约定有：

- JSON 字段风格统一（驼峰 or 下划线）
- 日期格式统一（ISO vs 时间戳）
- 错误响应格式固定（包含 code, message, data）
- Token 授权写在 Authorization header 中
- 上传文件使用 multipart/form-data

这些都是**接口协议中约定俗成**的内容，最好在 API 文档或团队文档中明确下来。

否则：

- 前端以为字段是 user_name，后端发的是 userName
- 前端期待时间是 "2024-01-01T10:00:00Z"，后端发的是 1704098400000
- 错误码字段不一致，前端无法统一处理异常

7.5 JavaScript 框架中的约定（Vue、React）

前端也有大量“隐形约定”：

- Vue 的单文件组件默认导出一个对象或函数
- React 的组件名必须大写首字母，否则 JSX 不会识别为组件
- 模板绑定中默认使用 v-model 约定为双向绑定

```
<input v-model="username" />
```

相当于：

```
:value="username" @input="username = $event.target.value"
```

你可以不使用它，但使用它的开发效率会高很多。

这些“语法糖”都是隐式约定的包装，让复杂操作变简单。

7.6 跨平台工具中的约定

Maven 的目录结构

```
src/
  main/java/          → Java 源文件
  main/resources/     → 配置文件、模板
  test/java/          → 测试代码
```

只要你遵循这个结构，Maven 就能自动打包、测试、运行。

Git 的约定

- .gitignore 约定哪些文件不入库
- README.md, LICENSE 是开源仓库默认识别的信息文件
- 提交信息格式：feat: 增加新功能，fix: 修复 bug（配合 CI 工具如 semantic-release）

这些都不是语法规定，而是社区行为的共识。

7.7 如何识别与应对隐形约定？

1. **看框架的默认行为**
    - 阅读官方示例，找出哪些地方“你没写它却自动生效了”
2. **阅读源码或文档源码注释**
    - 特别是注解、构造函数、注入行为
3. **看开源项目怎么写**
    - 观察他们的结构、命名、注释、调用方式
4. **与团队达成共识**
    - 用文档明确你们自己的“默认规则”，如 JSON 命名风格、错误码格式、认证方式等
5. **工具辅助验证**
    - 使用 Lint 工具（如 ESLint、Checkstyle）来自动检查是否遵守规则

7.8 小结

在现代软件工程中，除了语言层面的强制约定，还有大量“隐形规则”潜藏在框架、库、工具和团队习惯之中。

这些规则不会报错，却决定了：

- 系统是否易维护
- 团队是否高效协作
- 接口是否能顺畅联通

你能否敏锐察觉这些隐性约定，并主动使用和明确它们，将直接影响你能否成为一个“确信级”的开发者。

下一章，我们将从“写代码”扩展到“写规范”：

如何将这些约定转化为团队制度、文化与标准？如何制定规则而不让人反感？如何建立一份“约定文档”？

第 8 章：如何制定团队中的约定

前几章我们介绍了各种语言和框架中的约定，从显式的语法规则到隐式的协作默契。而当项目发展到多人协作阶段，最关键的问题变成了：

我们团队要不要制定自己的约定？怎么制定？谁来制定？如何让每个人都遵守？

这一章，我们将从技术与文化两个层面深入探讨：**如何打造属于你们团队的“编程宪法”——约定文档。**

8.1 为什么需要团队约定？

理由 1：减少沟通成本

- 命名方式统一 → 无需解释含义
- 项目结构统一 → 一眼识别代码职责
- 异常处理格式统一 → 前后端无需反复对齐

理由 2：提升协作效率

- 新人加入更快上手
- Code Review 更聚焦逻辑而非风格
- 自动化工具可以根据规范自动检查

理由 3：降低出错概率

- 一致的约定 = 一致的预期 = 更少的误解

约定，就是为了**在不需要询问的情况下，也能做对的事情**。

8.2 团队约定的典型内容

1. 命名规范

- 变量、类、方法的命名风格（驼峰、下划线）
- 是否允许缩写，哪些单词必须写全
- 枚举、常量、错误码的命名规则

2. 代码结构

- 每个模块放在哪个包
- Controller/Service/Repository 的分层规则
- 公共类是否放在 common 包中

3. 异常与返回值格式

- 所有接口是否都统一返回 Result<T>?
- 异常统一包装成 BusinessException?
- 错误码是否有一份集中定义？

4. 接口规范（前后端）

- JSON 命名规则（userName or user_name）
- 日期格式（ISO 8601 or 时间戳）
- HTTP 状态码与错误处理方式

5. 注释与文档

- 哪些类/方法必须写注释？
- 是否强制使用 JavaDoc?
- 是否要求补充使用示例？

6. Git 提交规范

- 提交信息格式（如 Conventional Commits）

- 分支命名（如 feature/xxx, bugfix/xxx）
- 是否使用 Pull Request + Review

7. 测试与部署流程

- 必须写单元测试吗？覆盖率要求多少？
- 每次合并前是否跑 CI？
- 发布流程中是否有验收步骤？

8.3 怎么制定这些约定？

方法 1：从已有项目中提炼

观察已有代码：大家**已经在默默遵守的规则**，可以整理成文档。

方法 2：从开源项目中借鉴

优秀开源项目如 Spring、Vue、React 都有明确的贡献指南、格式规范，可以作为参考。

方法 3：由核心成员主导初版

初期可以由经验丰富的开发者草拟初版规范，供团队评审。

方法 4：定期回顾与优化

规范不能一成不变，要每季度或每轮迭代后评估、更新。

8.4 如何写一份“约定文档”？

一个好的约定文档，应具备：

项目　描述

明确　不要用模糊词，如“尽量”、“最好”

项目　描述

示例　每一条规则附代码示例更易理解

可执行 可配合工具检查（如 ESLint, Checkstyle）

简洁　一页纸就能说明基本规则最好

分级　将规则分为“必须”、“建议”、“可选”

示例格式：

命名规范

- 变量使用小驼峰：userName

- 类名使用大驼峰：UserService

- 常量大写下划线：MAX_LENGTH

控制层结构

- 所有 Controller 放在 `controller` 包

- URL 映射统一小写，动词+名词结构，如 `/getUserList`

建议将该文档命名为：

- CONTRIBUTING.md
- team-style-guide.md
- 开发约定手册（v1.0）

8.5 如何让大家遵守？

规则写出来是一回事，让人遵守是另一回事。

建议 1：让约定“可检查”

- Java → Checkstyle + PMD + SpotBugs

- JS/TS → ESLint + Prettier
- Git 提交 → CommitLint + Husky

通过工具约束，可以减少人工口头提醒。

建议 2：将规范嵌入流程

- 每次 Pull Request 触发 CI 检查
- 不通过规范自动退回 PR
- 编译失败时提示规则违反点

建议 3：用 Code Review 培养习惯

- 评审时指出违反规范的地方
- 每月总结典型规范问题，集体讨论改进

建议 4：鼓励团队共建规范

- 允许团队成员提建议修改规则
- 建立规则变更记录（changelog）
- 激励大家参与“规则制定”，建立共识

规则不是约束，而是“信任的外部化”。

8.6 小结

在本章中，我们从实践角度讲解了：

- 为什么团队需要统一约定？
- 哪些内容是必须规范的？
- 如何制定并更新规则？
- 如何通过工具、流程、文化确保规则生效？

好的团队规范不是“限制开发者自由”，而是“解放他们的创造力”。

当团队成员不必在琐碎的选择上反复沟通，就能把精力集中在真正重要的设计、架构与创新上。

下一章，我们将把“约定”上升到更高层次：

架构设计与系统治理，也是一种约定思维。我们该如何构建可扩展、可协作的大系统？

第 9 章：架构与系统设计中的约定

到目前为止，我们讨论的“约定”多半发生在代码、工具与团队协作的层面。但随着系统变得复杂，组件变得众多、协作者变得广泛，我们会遇到新的问题：

- 服务与服务之间如何通信？
- 不同模块之间如何解耦？
- 如何演进架构而不破坏旧有逻辑？

这些问题，不再是“语法”层面的，也不再仅靠“注解”和“配置”能解决。

于是我们进入了“架构约定”与“系统治理”的世界。

大型系统的可扩展性，本质上依赖一套清晰而一致的“架构级约定”。

9.1 什么是架构层面的约定？

我们将架构级约定定义为：

在系统多个模块、服务或参与者之间，为了保持可协作、可演化、可维护而形成的“跨越模块边界”的规则与接口定义”。

这些约定通常表现为：

- 接口风格与协议（REST / GraphQL / gRPC）
- 微服务通信模式（同步 / 异步 / 事件驱动）
- 数据共享策略（API vs 数据复制 vs Event）
- 服务注册与发现机制

- 安全与鉴权标准（OAuth2, JWT）
- 可观测性协议（日志格式、追踪 ID）

9.2 接口层的约定：API 契约

每一个服务对外提供的 API，本质上是一种“契约”。

以 REST 为例：

```
GET /orders/{id}
Authorization: Bearer <token>
Accept: application/json
```

这条接口的隐含约定包括：

- 请求必须附带身份令牌（安全约定）
- 返回 JSON 格式数据（序列化约定）
- 路由风格统一（命名与分层约定）
- HTTP 状态码定义业务行为（如 404 表示“资源不存在”）

接口是一种显性“输入输出”约定，违背它就无法通信。

推荐实践：使用 OpenAPI（Swagger）描述接口契约文档。

9.3 服务之间的协作约定

微服务架构中，各服务间协作不能依赖“人记得”。必须靠协议与规范约束。

常见的协作约定：

约定类型	内容
数据一致性	是用事务？幂等性？补偿机制？

约定类型	内容
服务发现方式	静态配置？注册中心？DNS？
调用方式	HTTP REST？gRPC？消息队列？
接口版本管理	REST v1/v2？GraphQL schema？
认证机制	统一登录？Token 转发？

这些都必须**事先约定**，否则后续维护如同走钢丝。

微服务间最大的隐形 bug，就是“协作假设不一致”。

9.4 事件驱动架构中的约定

在事件驱动架构（EDA）中，服务不直接调用彼此，而是通过“事件”异步通知：

```
{
  "eventType": "OrderCreated",
  "data": {
    "orderId": 123,
    "userId": 456
  },
  "timestamp": "2025-06-01T10:00:00Z"
}
```

事件流的约定包括：

- 事件名命名规范（统一大写驼峰）
- 消息字段结构（数据字段必须固定）
- 是否需要幂等处理（消费重复事件怎么办）
- 消息持久化机制（是否允许补发）

可以使用 JSON Schema 或 Apache Avro 描述事件格式。

推荐：事件规范文档 + Schema Registry + 消费者契约测试

9.5 系统治理与元规则的约定

架构设计不是一次性的，系统必须不断演进。这要求我们建立一套“元级别”的约定机制。

典型治理项：

- **API 网关统一入口**（认证、限流、监控）
- **统一日志结构**（TraceId, Timestamp, Level）
- **服务健康检查标准**（Spring Actuator, /health）
- **错误码体系统一**（ErrorCode + Message）
- **配置统一中心管理**（如 Nacos, Spring Config Server）

这些都是系统生存性、扩展性、可维护性的基础规则。

治理 = 管控混乱，靠的是“元约定”。

9.6 架构图与文档：可视化约定

所有架构级约定都应有文档支撑，理想状态下还包括图示：

- 接口关系图
- 服务拓扑图
- 数据流图 / 事件流图
- 架构演进路线图

推荐使用工具：

- PlantUML

- Archimate
- Draw.io / Excalidraw
- Structurizr DSL

这些图和文档构成了“系统约定的视觉地图”。

9.7 如何推动架构约定落地？

1. **设定清晰标准文档**：不要只口头说，要写下来，集中管理
2. **制定规则责任人**：架构师或 Tech Leader 负责解释与修订
3. **配合自动化工具验证**：如 API 测试、日志格式校验、监控系统校验
4. **定期架构回顾**：每个季度或大版本做架构约定回顾和调整
5. **培养“契约思维”文化**：让每位成员理解“约定不是限制，而是合作语言”

9.8 小结

架构是“约定”的高维表达。

从 API 接口、服务通信，到日志、异常、部署流程，所有良好运作的大型系统背后，都是一套可识别、可遵循、可持续演进的“架构约定体系”。

本章强调了：

- 架构不仅是设计图，更是规则集合
- 系统治理本质上是约定治理
- 架构的可演进能力，来自于约定的“弹性与稳定平衡”

在最后一章，我们将进一步抽象与升华：

从语言到协作，从代码到文化，约定如何构成整个程序世界的秩序基础？

第 10 章：约定的哲学 —— 程序世界的秩序之源

我们已经从最基础的语法约定、命名规则，一路走到了架构设计、系统治理。在最后这一章，我们将不再讲具体的技术细节，而试图揭示更深层次的本质：

为什么“约定”如此重要？它背后是否代表着一种更高层次的认知方式？

10.1 程序员的世界：一场人与规则的博弈

程序员的日常是什么？

- 是写代码？调 bug？学新框架？

其实更本质地说：

程序员是在不断“接受规则、遵守规则、制定规则、优化规则”。

你学 Java，要理解语法规则；你写后端，要遵守 RESTful API 协议；你加入团队，要服从项目结构；你做系统架构，要设计服务间协作协议。

每一层都是“规则”，每一步都是“约定”。

而真正优秀的程序员，不是对抗规则，而是：

通过理解规则，掌握自由。

10.2 什么是“约定哲学”？

我们可以这样定义：

约定，是人类为了达成可协作、可复用、可演进的系统，而在语言、行为、结构层面设立的共识规则。

它是一种：

- **形式化**（精确定义的）
- **共享性**（所有人都能理解的）

- **可验证**（能否遵守可以检测的）
- **可继承**（后人可以沿用的）

的语言单位。

约定 ≠ 束缚，它是共识的工具，是混乱世界的秩序之源。

10.3 为什么“自由编码”不是理想？

许多初学者误解：

“写代码不就是想怎么写就怎么写嘛。”

但实际上：

- 自由编写 = 后期维护困难
- 不设规则 = 无法协作
- 无命名规范 = 看不懂别人的代码
- 无接口契约 = 服务无法联通

自由之上，如果没有秩序，那只是混乱；而有规则的自由，才是创造。

约定就是这种秩序的体现，它：

- 给出边界（你可以写的范围）
- 提供预期（别人如何理解你）
- 帮助自动化（让工具知道你在干什么）

10.4 编程世界中的“秩序创造模型”

我们可以这样比喻：

层级	行为	对应约定
语言	输入必须精确	语法、类型系统
函数	调用必须符合契约	方法签名、返回值
项目	文件必须组织好	模块划分、结构规范
团队	协作要有边界	接口设计、代码规范
系统	服务间通信要可控	API 契约、架构协议

每一层的良好运作，都依赖于**可识别的约定**。

如果没有这些，每一层都会失控。

10.5 人类社会也是“约定驱动”的

程序世界源自人类社会。

- 语言是约定（“红色”这个词是我们共同赋予的）
- 时间是约定（一天 24 小时是人为定义）
- 货币是约定（一张纸上印了￥100 就有价值）
- 法律是约定（宪法条文是一种权力规则）

所以，**人类的协作文明，就是一层层“约定叠加”的结果。**

计算机系统，只是这种“规则协作”的极端形式：

- 它不容模糊
- 它无法猜测
- 它必须规则先行

程序员，就是这个“约定社会”的设计师与执行者。

10.6 如何建立“约定思维”？

1. **主动识别隐形约定**
    - 阅读框架源码
    - 注意“为什么这样写就生效”？
2. **善于制定自己的微型规则**
    - 自己习惯的注释方式
    - 文件命名与分层结构
    - 参数顺序与返回值结构
3. **不断抽象出可复用的约定**
    - 把“做对一次”的经验总结成规则
    - 写进 Wiki、CONTRIBUTING.md、代码模板
4. **尊重他人的约定生态**
    - 使用别人开源项目，要先读他们的规范
    - 进入新团队，先熟悉已有习惯
5. **将约定作为认知路径**
    - 学习任何一个新技术，不是“看它能做什么”，而是“看它假设了什么”

10.7 结语：约定即秩序，秩序生自由

我们用了整整十章，构建了一个看似技术性的主题：《约定》。但它最终不只是关于代码的写法、结构的规范、工具的用法。

它其实是：

- 程序员如何理解系统
- 程序员如何构建秩序

- 程序员如何与他人协作

的一种世界观。

约定，是程序员的语言，是程序世界的秩序，也是确信的基础。

写下代码的那一刻，你就在与未来的你、与你的同事、与你的系统约定一场行为。

愿你从此以后，

看懂别人留下的约定，也能写下别人愿意遵守的约定。

愿你的系统，

稳定、优雅、清晰、可维护。

因为你心中有规则，眼中有结构，手中有确信。

第 11 章：互联网是一个约定的世界

互联网并不是一个混乱的“自由空间”，而是一个靠无数约定构建起来的巨大系统。它像一个运转稳定的城市，表面上充满了无限可能，其实内部每一个动作，每一次通信，都严格遵守了一套套被全球共同接受的“约定”。在这一章中，我们将深入探讨这些约定背后的本质、形成过程、演进轨迹与哲学意义。

11.1 网络世界的“契约精神”

约定，是现实社会得以协作的基础。在互联网世界，这种“契约精神”体现得更加精密、严格、体系化。

每一封电子邮件，每一次网页访问，每一次直播、下载、搜索、支付、上传，背后都有数十甚至上百个“协定”在无声地运作。例如：

- IP 地址确保设备身份不重复
- DNS 把人类语言转换成机器语言
- TCP/IP 规定了传输顺序与可靠性

- HTTP 决定客户端如何请求内容
- TLS/SSL 保障数据安全传输

这些并不是法律强制，而是工程师们在“共识”基础上建立起的事实标准（de facto standards）。这种看似松散、实则高效的标准化过程，是互联网文明最伟大的成就之一。

11.2 IP 与 DNS：互联网的户籍系统

IP 地址：地址是身份

每一台连接互联网的设备都需要一个“地址”，这就是 IP。IPv4 是最早的协议（如 192.168.1.1），但由于地址耗尽，IPv6（如 2001:0db8:85a3::8a2e:0370:7334）逐步推广。

IP 是网络通信中最底层的身份标识：

- 没有 IP，数据包无法被路由
- 通过 IP 可以定位设备
- 但 IP 不等于用户（NAT 等机制存在）

DNS：从人类到机器的翻译官

人类记住域名，机器识别 IP。DNS 的存在就是为了协调两者。它基于一个分层架构：

1. 根域（.）
2. 顶级域（.com，.org，.jp 等）
3. 二级域（example.com）

当你访问某网页时，浏览器会先向 DNS 服务器发起查询，这一过程有多层缓存（本地、ISP、公网等）来加速。

DNS 是互联网最成功的“分布式约定系统”之一。

11.3 TCP/IP 与可靠性协议

互联网不是“瞬时通道”，而是“信息邮局”。你发出一份数据包，它可能：

- 被截断
- 被延迟
- 顺序错乱
- 重复发送

TCP（Transmission Control Protocol）就是解决这一切问题的“约定”：

- 三次握手：建立连接的礼节
- 顺序编号：恢复数据顺序
- 超时重传：保障可靠到达

相比之下，UDP 是“不可靠协议”，但它有利于直播、游戏等对速度敏感的场景。

11.4 HTTP 与 Web 的会话契约

网页不是图画，是一种“结构化信息协议”。HTTP 规定了浏览器与服务器如何对话。

请求结构：

```
GET /index.html HTTP/1.1
Host: www.example.com
```

响应结构：

```
HTTP/1.1 200 OK
Content-Type: text/html
```

每一个响应码（200、404、500）都代表一次会话的“裁决”。

随着需求增长，HTTP 演化出了：

- HTTP/2：多路复用、压缩头部

- HTTP/3：基于 QUIC（UDP）的高速通道

每一代都代表约定的升级。

11.5 HTML/CSS/JS 的前端约定

浏览器不会“理解”，它只是“解释”。HTML 是文档结构的约定语言，CSS 是样式层的描述语言，JS 是行为逻辑的执行语言。

例如：

```
<!DOCTYPE html>
<html>
<head><title>约定世界</title></head>
<body><h1>Hello, Internet!</h1></body>
</html>
```

不写 <!DOCTYPE>？浏览器会进入怪异模式（quirks mode），很多渲染行为变得不可预测。这也是一种“你若不守约，我就不保证行为”的机制。

11.6 搜索引擎与 robots.txt：协议之外的默契

搜索引擎爬虫（crawler）会访问你的网站。

- 想被收录？提供 sitemap.xml
- 不想被抓？用 robots.txt 阻止

例如：

```
User-agent: *
Disallow: /private/
```

这是不具强制力的协议，但所有主流搜索引擎都会遵守。

这体现了“非强制性约定”的力量。

11.7 HTTPS 与信任体系

加密并不是魔法，它是一套“信任协定”：

- 浏览器信任“根证书颁发机构”
- 这些机构签发证书（如 *.example.com）
- 客户端用公钥解密服务端返回的信息

如果证书无效，浏览器会警告用户。
这就是用户与站点之间的“加密契约”。

11.8 WebSocket、gRPC 与新型协议

WebSocket：实现长连接，约定了通信格式（帧、ping/pong）
gRPC：使用 protobuf 进行结构化通信，高性能、跨语言

这些协议构建在 HTTP/2 或 TCP 之上，体现了“协议之上的协议”。

11.9 分布式与服务间的契约

现代系统是服务组成的网：A 调用 B，B 调用 C。
每个服务之间的调用都需遵守：

- 接口定义（OpenAPI、protobuf）
- 认证授权（OAuth、JWT）
- 响应规范（错误码、字段类型）

服务之间的稳定，靠的不是“信任”，而是“契约”。

11.10 哲学视角：秩序的隐形编织

互联网的成功，在于其“约定自治”。它没有一位绝对权威者，但依然形成了稳定、高效的运行机制。

它就像一个全球“契约社会”，程序员是其中的守约者与制定者。

这不仅是技术设计，更是社会设计的智慧。

小结

互联网从未真正混乱，它只是秩序太过精妙以至于被忽视。理解这些“隐形约定”，让我们在技术的表象下看到真正的文明核心：规则、尊重与共识。