Seiri，或组织（想想英语中的“排序”）。 使用适当的命名等方法了解事物的位置至关重要。 您认为命名标识符不重要吗？ 请继续阅读以下章节。

• Seiton，或整洁（想想英语中的“系统化”）。 美国有一句古老的谚语：万物皆有其所在，一切各就其位。 一段代码应该在您期望找到它的地方，如果没有，您应该重构以将其找到。

• Seiso 或清洁（想想英语中的“shine”）：保持工作场所没有悬挂的电线、油脂、碎屑和废物。 对于在代码中乱扔记录历史或对未来的愿望的注释和注释掉的代码行，作者有何看法？ 摆脱他们。

• Seiketsu，或标准化：小组就如何保持工作场所清洁达成一致。 您认为这本书是否谈到了在团队内保持一致的编码风格和实践集？ 这些标准从何而来？ 请继续阅读。 • Shutsuke，或纪律（自律）。 这意味着要遵守实践、经常反思自己的工作并愿意改变。

童子军规则

使用透露意图的名称

匈牙利表示法

* **不要为名称添加不必要的上下文**，避免冗余信息。
* 通过类名、模块名等隐式上下文来简化名称。
* 在必要时（如名称冲突、全局变量等）才添加上下文。

1. **函数应该小**：
   * 函数的首要规则是**要小**，其次是要**更小**。
   * 作者通过自己几十年的编程经验，总结出函数应该尽可能短小。

**. 遵循单一职责原则**

* 每个函数只做一件事。
* 如果一个函数做了多件事，应该将其拆分为多个小函数。

**2. 使用描述性名称**

* 函数名应该清晰地表达函数的功能。
* 通过名称可以让调用者理解函数的作用，而不需要查看实现细节。

**3. 减少嵌套层级**

* 避免深层嵌套（如多层 if-else 或 for 循环）。
* 通过提取方法将嵌套逻辑拆分为独立的函数。

**4. 限制函数长度**

* 通常建议一个函数的长度不超过 **20 行**。
* 如果函数超过 20 行，考虑是否可以将部分逻辑提取到新函数中。

**5. 避免副作用**

* 函数应该只做它声明的事情，避免修改全局状态或输入参数。
* 副作用会增加函数的复杂性，使其难以理解和测试。

这意味着 if 语句、else 语句、while 语句等中的块应该是一行长。 也许该行应该是一个函数调用。 这不仅使封闭函数保持较小，而且还增加了文档价值，因为在块内调用的函数可以有一个很好的描述性名称。

我们希望代码读起来像自上而下的叙述。5我们希望每个函数后面都有下一个抽象级别的函数，这样我们就可以阅读程序，在向下阅读时一次下降一个抽象级别。 函数列表。 我称之为“降级规则”。

函数的理想参数数量为零 (niladic)。 接下来是一个（一元），紧接着是两个（二元）。 应尽可能避免三个参数（三元组）。 超过三个（多元）需要非常特殊的理由，然后无论如何都不应该使用。

标志参数是丑陋的。 将布尔值传递给函数是一种非常糟糕的做法。 它立即使方法的签名变得复杂，大声宣称这个函数不仅仅做一件事。 如果标志为真，它会做一件事，如果标志为假，它会做另一件事！

\*\*时间耦合（Temporal Coupling）\*\*的，即函数的行为依赖于调用它的时机。这种设计会导致代码难以理解和维护，尤其是当这种依赖关系被隐藏在副作用中时。你提到的例子中，checkPassword 函数不仅检查密码，还会初始化会话（session），这导致了两个问题：

在面向对象编程中，输出参数通常是不必要的，因为对象本身可以封装状态和行为。通过将操作绑定到对象上，我们可以避免输出参数，从而使代码更加清晰、符合面向对象的设计原则，并提高可读性和可维护性。如果必须修改状态，尽量让函数修改其所属对象的状态，而不是通过输出参数来实现。

**输出参数（Output Arguments）** 是指通过函数参数传递的变量，函数会修改该变量的值。换句话说，输出参数是函数用来返回结果的一种方式，而不是通过返回值直接返回结果。

每个系统都是由程序员设计的用于描述该系统的特定领域语言构建的。 函数是该语言的动词，类是名词。 这并不是回到可怕的旧观念，即需求文档中的名词和动词是对系统的类和功能的第一个猜测。 相反，这是一个更古老的真理。 编程的艺术一直都是语言设计的艺术。

程序员大师将系统视为要讲述的故事，而不是要编写的程序。 他们使用所选编程语言的功能来构建更丰富、更具表现力的语言来讲述这个故事。 该领域特定语言的一部分是描述该系统内发生的所有操作的功能层次结构。 在巧妙的递归行为中，这些操作被编写为使用它们定义的特定于领域的语言来讲述自己的故事的一小部分。

本章讨论了编写好函数的机制。 如果您遵循此处的规则，您的函数将会简短、命名良好且组织良好。 但永远不要忘记，您的真正目标是讲述系统的故事，并且您编写的函数需要干净地组合成一种清晰而精确的语言，以帮助您讲述这个故事。

正确使用注释是为了弥补我们在代码中表达自己的失败。 请注意，我使用了“失败”这个词。 我是认真的。 评论总是失败的。 我们必须拥有它们，因为没有它们我们总是无法弄清楚如何表达自己，但它们的使用并不值得庆祝。

**1. 变量命名**

* **小驼峰命名法（camelCase）**：变量名以小写字母开头，后续单词首字母大写。

cpp

**2. 函数命名**

* **小驼峰命名法（camelCase）**：

**3. 类命名**

* **大驼峰命名法（PascalCase）**：类名以大写字母开头，后续单词首字母大写。

**4. 常量命名**

* **全大写字母 + 下划线分隔（UPPER\_CASE）**：

**5. 枚举命名**

* **枚举类型名**：使用大驼峰命名法（PascalCase）。

**6. 命名空间命名**

* **小写字母**：命名空间名通常使用小写字母。

**8. 成员变量命名**

* **前缀或后缀**：为了区分成员变量和局部变量，常见的做法是使用前缀或后缀。

几乎所有代码都是从左到右、从上到下读取的。 每行代表一个表达式或一个子句，每组行代表一个完整的思想。 这些想法应该用空行彼此分开。

 有空行分隔包声明、导入和每个函数。 这个极其简单的规则对代码的视觉布局有着深远的影响。 每个空白行都是一个视觉提示，标识一个新的、独立的概念。 当您向下浏览列表时，您的眼睛会被空行后面的第一行吸引。

因为我们的函数非常短，所以局部变量应该出现在每个函数的顶部，

**1. C++ 中的“剪刀规则”（Scissors Rule）**

在 C++ 中，剪刀规则建议将实例变量放在类的底部。这种风格的逻辑是：

* **关注点分离**：将公共接口（方法）放在类的顶部，便于阅读和使用。
* **实现细节隐藏**：将私有成员（实例变量）放在底部，强调它们是实现细节。

另一方面，我没有在函数名称和左括号之间添加空格。 这是因为函数及其参数密切相关。 将它们分开使它们看起来是脱节的而不是连在一起的。 我在函数调用括号内分隔参数以强调逗号并表明参数是分开的。

隐藏实现不仅仅是在变量之间放置一层函数的问题。 隐藏实现是关于抽象！ 类并不是简单地通过 getter 和 setter 推出它的变量。 相反，它公开了抽象接口，允许用户操作数据的本质，而无需知道其实现。

过程代码（使用数据结构的代码）可以轻松添加新功能，而无需更改现有数据结构。 另一方面，OO 代码可以轻松添加新类，而无需更改现有函数。

过程代码使得添加新的数据结构变得困难，因为所有函数都必须更改。 OO 代码使得添加新函数变得困难，因为所有类都必须更改。

**1. 迪米特法则的核心规则**

迪米特法则可以用一句话概括：**“只与直接的朋友通信”**。具体来说：

* 一个对象的方法只能调用以下对象的方法：
  1. 它自身的方法。
  2. 它的成员对象的方法。
  3. 它作为参数传入的对象的方法。
  4. 它创建的对象的方法。
* 一个对象的方法**不应该调用由其他方法返回的对象的方法**（即避免链式调用）。

对象公开行为并隐藏数据。 这使得在不改变现有行为的情况下添加新类型的对象变得容易。 它还使得向现有对象添加新行为变得困难。 数据结构公开数据并且没有显着的行为。 这使得向现有数据结构添加新行为变得容易，但向现有函数添加新数据结构变得困难。

在任何给定的系统中，我们有时会希望能够灵活地添加新的数据类型，因此我们更喜欢系统的该部分的对象。 其他时候，我们希望能够灵活地添加新行为，因此在系统的这一部分中，我们更喜欢数据类型和过程。 优秀的软件开发人员会不带偏见地理解这些问题，并选择最适合手头工作的方法。

**TDD（测试驱动开发，Test-Driven Development）的三条规则** 是 TDD 的核心原则。这些规则不仅要求我们在编写生产代码之前先写单元测试，还规定了如何逐步编写测试和代码。

**1. 第一条规则：在编写生产代码之前，必须先编写一个失败的单元测试**

* **含义**：在编写任何生产代码之前，必须先编写一个单元测试，并且这个测试必须失败（因为没有实现对应的功能）。
* **目的**：确保测试是有效的，并且能够检测到代码的变化。

**2. 第二条规则：单元测试的代码量应该刚好足够失败，编译失败也算失败**

* **含义**：单元测试的代码量应该尽可能少，只要能够触发失败即可。即使是编译失败（例如函数未定义）也算作失败。
* **目的**：避免编写过多的测试代码，确保测试的简洁性和针对性。

**3. 第三条规则：生产代码的代码量应该刚好足够通过当前失败的测试**

* **含义**：在编写生产代码时，只编写能够通过当前失败测试的最少代码，不要过度实现。
* **目的**：避免编写不必要的代码，确保代码的简洁性和可维护性。

**1. TDD 的基本流程**

TDD 的工作流程可以总结为以下三个步骤，通常被称为 **“红-绿-重构”** 循环：

**1. 红（Red）：编写一个失败的测试**

* 在编写任何生产代码之前，先编写一个单元测试。
* 这个测试应该描述你想要实现的功能。
* 运行测试，测试会失败（因为没有实现对应的功能）。

**2. 绿（Green）：编写最少的生产代码使测试通过**

* 编写刚好足够通过测试的生产代码。
* 运行测试，测试应该通过。

**3. 重构（Refactor）：优化代码**

* 在测试通过的基础上，优化代码的结构和设计。
* 确保代码简洁、可读、可维护。
* 运行测试，确保重构没有引入新的问题。

快速测试应该很快。 他们应该跑得快。 当测试运行缓慢时，您不会想频繁运行它们。 如果您不经常运行它们，您将无法及早发现问题并轻松修复它们。 您将无法自由地清理代码。 最终代码将开始腐烂。

快速测试应该很快。 他们应该跑得快。 当测试运行缓慢时，您不会想频繁运行它们。 如果您不经常运行它们，您将无法及早发现问题并轻松修复它们。 您将无法自由地清理代码。 最终代码将开始腐烂。

可重复的测试应该在任何环境中都是可重复的。 您应该能够在生产环境、QA 环境中以及在没有网络的情况下乘坐火车回家时在笔记本电脑上运行测试。 如果你的测试在任何环境下都不可重复，那么你总会有借口来解释为什么它们失败了。 当环境不可用时，您还会发现自己无法运行测试。

自我验证测试应该有一个布尔输出。 他们要么通过，要么失败。 您不必通过阅读日志文件来判断测试是否通过。 您不必手动比较两个不同的文本文件来查看测试是否通过。 如果测试不能自我验证，那么失败可能会变得主观，并且运行测试可能需要长时间的手动评估。

及时 需要及时编写测试。 单元测试应该写在使单元测试通过的生产代码之前。 如果您在生产代码之后编写测试，那么您可能会发现生产代码很难测试。 您可能认为某些生产代码太难测试。 您可能无法将生产代码设计为可测试的。

**1. C++ 类成员的常见排序规则**

C++ 的类成员排序通常遵循以下顺序：

**（1）公共成员（Public Members）**

* **公共常量（Public Constants）**：例如 public static const 或 constexpr。
* **公共函数（Public Functions）**：包括构造函数、析构函数、接口方法等。

**（2）保护成员（Protected Members）**

* **保护变量（Protected Variables）**：通常用于继承层次结构中。
* **保护函数（Protected Functions）**：子类可以访问的函数。

**（3）私有成员（Private Members）**

* **私有常量（Private Constants）**：例如 private static const。
* **私有变量（Private Variables）**：类的内部状态。
* **私有函数（Private Functions）**：工具函数或辅助函数。

**（4）友元声明（Friend Declarations）**

* 如果需要，可以在类的末尾声明友元类或友元函数。

事实上，命名可能是帮助确定班级规模的第一种方法。 如果我们不能为一个类派生一个简洁的名称，那么它可能太大了。 类名越模糊，它就越有可能承担过多的职责。

**SRP（单一职责原则，Single Responsibility Principle）** 是面向对象设计（OOD）中的一个核心原则，它指出：**一个类应该只有一个引起它变化的原因**，或者说**一个类应该只有一个职责**

类应该有少量的实例变量。 类的每个方法都应该操作一个或多个这些变量。 一般来说，方法操作的变量越多，该方法与其类的内聚力就越大。 每个方法都使用每个变量的类具有最大的内聚性。

类的内聚性（Cohesion）**的概念。内聚性是指一个类中的方法和变量之间的关联程度。高内聚性意味着类中的方法和变量紧密相关，共同完成一个明确的任务或职责。这是面向对象设计中的一个重要原则，与**单一职责原则（SRP）\*\*密切相关。

**. 什么是内聚性？**

内聚性描述了一个类中的方法和变量之间的关系。高内聚性的类具有以下特点：

* **方法和变量紧密相关**：类中的方法主要操作类的实例变量。
* **职责单一**：类的职责明确，不包含无关的功能。
* **逻辑完整**：类是一个独立的、自包含的逻辑单元。

**2. 如何实现构造与使用的分离？**

**（1）在 main 函数中构造对象**

* 在 main 函数中创建所有需要的对象，并将它们注入到应用程序中。

**（2）应用程序只使用对象**

* 应用程序的类只负责使用对象，而不负责创建对象。

**（3）依赖注入**

* 通过构造函数或方法参数将依赖对象注入到应用程序中。

系统也必须干净。 侵入式架构压倒了领域逻辑并影响敏捷性。 当领域逻辑模糊时，质量就会受到影响，因为错误更容易隐藏，而故事变得更难以实现。

如果敏捷性受到损害，生产力就会受到影响，并且 TDD 的优势也会丧失。 在所有抽象级别，意图都应该明确。 仅当您编写 POJO 并使用类似方面的机制以非侵入性方式合并其他实现问题时，才会发生这种情况。 无论您是设计系统还是单个模块，永远不要忘记使用最简单、可行的东西。

**1. 什么是侵入性架构？**

侵入性架构是指系统的架构设计过于复杂，强制要求领域逻辑与特定的技术细节（如框架、库、基础设施）紧密耦合。这种架构会导致：

* **领域逻辑被掩盖**：业务逻辑与技术细节混杂在一起，难以分离。
* **系统难以扩展**：每次技术栈变更都需要大量修改代码。
* **测试困难**：领域逻辑无法独立测试，必须依赖特定的技术环境。

**1. 通过所有测试（Runs All the Tests）**

* **含义**：代码必须通过所有的测试用例，包括单元测试、集成测试和系统测试。
* **目的**：确保代码的正确性和可靠性

**2. 消除重复（Contains No Duplication）**

* **含义**：代码中不应该有重复的逻辑或代码片段。
* **目的**：减少代码量，提高可维护性。

**3. 表达程序员的意图（Expresses the Intent of the Programmer）**

* **含义**：代码应该清晰地表达其意图，易于理解和维护。
* **目的**：提高代码的可读性和可维护性。

**4. 最小化类和方法的数量（Minimizes the Number of Classes and Methods）**

* **含义**：在满足功能需求的前提下，尽量减少类和方法的数量。
* **目的**：避免过度设计，保持代码简洁

并发是一种解耦策略。 它帮助我们将完成的事情与完成的时间分离。 在单线程应用程序中，什么以及何时耦合得如此紧密，以至于整个应用程序的状态通常可以通过查看堆栈回溯来确定。 调试此类系统的程序员可以设置一个断点或一系列断点，并了解命中断点的系统状态。

将内容与时间解耦可以显着提高应用程序的吞吐量和结构。 从结构的角度来看，该应用程序看起来像许多小型协作计算机，而不是一个大的主循环。 这可以使系统更容易理解，并提供一些强大的方法来分离关注点。

• 并发总是可以提高性能。 并发有时可以提高性能，但前提是有大量等待时间可以在多个线程或多处理器之间共享。 这两种情况都不是微不足道的。

• 编写并发程序时设计不会改变。 事实上，并发算法的设计与单线程系统的设计有很大不同。 什么与何时的解耦通常会对系统的结构产生巨大的影响。

• 在使用Web 或EJB 容器等容器时，了解并发问题并不重要。 事实上，你最好知道你的容器在做什么，以及如何防范本章后面描述的并发更新和死锁问题。

• 并发会产生一些开销，无论是在性能方面还是在编写额外代码方面。

• 正确的并发性很复杂，即使对于简单的问题也是如此。

• 并发错误通常不可重复，因此它们常常被视为一次性问题2，而不是真正的缺陷。 • 并发性通常需要对设计策略进行根本性的改变。

并发防御原则

下面是一系列的原则和技术，用于保护您的系统免受并发代码问题的影响。

不幸的是，将并发实现细节直接嵌入到其他生产代码中的情况太常见了。 以下是需要考虑的一些事项：

• 与并发相关的代码有其自己的开发、更改和调优生命周期。

• 并发相关代码有其自身的挑战，这些挑战不同于非并发相关代码，而且通常比非并发相关代码更困难。

• 错误编写的基于并发的代码可能会失败的方式有很多，这使其在不增加周围应用程序代码负担的情况下具有足够的挑战性。

Recommendation: Keep your concurrency-related code separate from other code.

**1. 为什么线程应尽可能独立？**

**（1）减少同步需求**

* 如果线程之间不共享数据，就不需要使用锁或其他同步机制，从而避免竞态条件、死锁等问题。

**（2）简化代码**

* 独立的线程更容易理解和维护，因为它们的逻辑是自包含的。

**（3）提高性能**

* 减少同步操作可以提高性能，因为线程不需要等待锁或竞争资源。

**（4）增强可测试性**

* 独立的线程可以更容易地进行单元测试，因为它们的逻辑不依赖于其他线程的状态。

**1. Bound Resources（受限资源）**

* **定义**：在并发环境中使用的固定大小或数量的资源。
* **示例**：数据库连接、固定大小的读写缓冲区。
* **重要性**：受限资源的管理是并发编程中的关键问题，因为资源的有限性可能导致竞争和性能瓶颈。

**2. Mutual Exclusion（互斥）**

* **定义**：一次只有一个线程可以访问共享数据或资源。
* **实现方式**：通常通过锁（如互斥锁）来实现。
* **重要性**：互斥是防止数据竞争和确保线程安全的基本机制。

**3. Starvation（饥饿）**

* **定义**：一个线程或一组线程被长时间或永久禁止继续执行。
* **示例**：如果总是优先让运行速度快的线程执行，可能会导致运行时间较长的线程长时间得不到执行。
* **重要性**：饥饿会影响系统的公平性和性能，需要合理的调度策略来避免。

**4. Deadlock（死锁）**

* **定义**：两个或多个线程相互等待对方释放资源，导致所有线程都无法继续执行。
* **示例**：线程 A 持有资源 X 并等待资源 Y，而线程 B 持有资源 Y 并等待资源 X。
* **重要性**：死锁会导致系统完全停止响应，需要通过合理的资源分配和锁顺序来避免。

**5. Livelock（活锁）**

* **定义**：线程在相互“礼让”中不断尝试执行，但由于共振效应，无法取得进展。
* **示例**：两个线程在遇到对方时都主动让出资源，导致双方都无法继续执行。
* **重要性**：活锁虽然不会导致系统完全停止，但会严重影响性能，需要通过调整策略来解决。

**6. 执行模型（Execution Models）**

在并发编程中，有几种常见的执行模型，用于管理线程和资源的交互：

* **线程池模型**：使用固定数量的线程来处理任务，适用于任务数量多且执行时间短的情况。
* **生产者-消费者模型**：生产者线程生成任务并放入队列，消费者线程从队列中取出任务并执行。
* **读写锁模型**：允许多个读线程同时访问资源，但写线程需要独占访问。
* **事件驱动模型**：通过事件循环处理异步事件，适用于高并发的 I/O 操作。

生产者-消费者9

一个或多个生产者线程创建一些工作并将其放入缓冲区或队列中。 一个或多个消费者线程从队列中获取该工作并完成它。 生产者和消费者之间的队列是一个绑定资源。 这意味着生产者必须在写入之前等待队列中的可用空间，而消费者必须等到队列中有东西可以使用。 生产者和消费者之间通过队列进行的协调涉及生产者和消费者相互发送信号。 生产者写入队列并发出信号表明队列不再为空。 消费者从队列中读取数据并发出队列不再满的信号。 两者都可能等待通知何时可以继续。

读者-作者10

当您拥有一个主要用作读者信息源但偶尔由作者更新的共享资源时，吞吐量就是一个问题。 强调吞吐量可能会导致饥饿和陈旧信息的积累。 允许更新可能会影响吞吐量。 协调读者，使他们不阅读作者正在更新的内容，反之亦然，这是一个艰难的平衡行为。 写入者往往会长时间阻塞许多读取者，从而导致吞吐量问题。

挑战在于平衡读取器和写入器的需求，以满足正确的操作、提供合理的吞吐量并避免饥饿。 一个简单的策略是让写入者等到没有读取者才允许写入者执行更新。 然而，如果有持续不断的读者，作家就会挨饿。 另一方面，如果有频繁的写入者并且给予他们优先级，则吞吐量将会受到影响。 问题所要解决的就是找到这种平衡并避免并发更新问题。

**1. 哲学家进餐问题的描述**

* **哲学家**：代表线程。
* **叉子**：代表资源（如锁、文件句柄等）。
* **规则**：
  + 每个哲学家需要两把叉子才能进餐。
  + 每把叉子只能被一个哲学家使用。
  + 如果哲学家无法同时拿到两把叉子，就必须等待。

**2. 问题的核心挑战**

* **死锁（Deadlock）**：如果所有哲学家同时拿起左边的叉子，然后尝试拿右边的叉子，就会导致所有哲学家都无法继续进餐。
* **活锁（Livelock）**：哲学家们可能会不断尝试拿叉子，但由于相互礼让而无法取得进展。
* **吞吐量和效率**：不合理的资源分配可能导致系统吞吐量下降和效率降低。

**1. 什么是临界区？**

* **临界区**：一段需要被保护的代码，确保在同一时间只有一个线程可以执行。
* **目的**：防止多个线程同时访问共享资源，导致数据竞争和不一致。

让我让你安心吧。 我并不是简单地以当前的形式从头到尾编写这个程序。 更重要的是，我并不期望您能够一次性编写出干净而优雅的程序。 如果说我们在过去几十年里学到了什么的话，那就是编程与其说是一门科学，不如说是一门手艺。 要编写干净的代码，必须先编写脏代码，然后再清理它。

大多数新生程序员（像大多数小学生一样）并没有很好地遵循这个建议。 他们认为首要目标是让该计划发挥作用。 一旦“工作”，他们就会继续执行下一个任务，让“工作”程序保持最终“工作”的状态。 大多数经验丰富的程序员都知道这是职业自杀。

.评论

. C1：不当信息

C2：过时的评论

C3：多余的注释

C4：评论写得不好

C5：注释掉的代码

环境

E1：构建需要多个步骤

E2：测试需要多个步骤

函数

F1：参数太多函数应该有少量参数。

F2：输出参数

F3：标记参数

F4：死函数

综合

G1：一个源文件中的多种语言

G2：明显的行为未实现

G3：边界处的不正确行为

G4：超越安全性

G5：复制

G6：抽象级别错误的代码

G7：基类取决于其派生类

G8：信息太多

G9：死代码

G10：垂直分离

G11：不一致

G12：杂乱

G13：人工耦合

G14：功能羡慕

G15：选择器参数

G16：模糊的意图

错位的责任

G18：不适当的静电

G19：使用解释变量

解释变量的简单使用可以清楚地表明第一个匹配组是键，第二个匹配组是值。 这很难做得太过分。 更多的解释变量通常比更少的解释变量更好。 值得注意的是，只需将计算分解为命名良好的中间值，一个不透明的模块就可以突然变得透明。

G20：函数名称应该说明它们的作用

G21：理解算法

G22：使逻辑依赖成为物理依赖

如果一个模块依赖于另一个模块，那么这种依赖关系应该是物理的，而不仅仅是逻辑的。 依赖模块不应该对其依赖的模块做出假设（换句话说，逻辑依赖）。 相反，它应该明确地向该模块询问它所依赖的所有信息。

G23：比起 If/Else 或 Switch/Case 更喜欢多态

G24：遵循标准约定

G25：用命名常量替换幻数

G26：精确

G27：结构优于惯例

G28：封装条件

G29：避免否定条件