EDA 系统软件分析和设计方法学

讲义

2026 年春季

毕朝日

复旦大学集成电路与微纳电子创新学院

目录

第四讲　软件错误与测试

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> — 艾兹赫尔 • 迪杰斯特拉（Edsger W. Dijkstra）

上一讲我们沿着历史的轨迹，讲述了软件工程方法从混乱走向秩序的过程。我们得出的核心结论是：简单是可靠的前提——过程模型的演进在追求更简洁的组织方式，模块化设计在将复杂系统分解为简单单元，结构化编程在用三种简单的控制结构取代无序的跳转。但即便我们掌握了最好的方法论，遵循了最严格的工程纪律，一个令人不安的事实依然存在：**软件中总是有错误的。**

1947 年 9 月 9 日，哈佛大学的工程师们在 Mark II 计算机的继电器中发现了一只飞蛾。工程师们将这只飞蛾用胶带粘在工作日志上，旁边写道：“*First actual case of bug being found.*”格蕾丝 • 霍珀（Grace Hopper）后来广泛传播了这一轶事，使其成为计算机史上最著名的故事之一。虽然爱迪生早在 1878 年就用“bug”来形容技术故障，但这只真实的虫子赋予了这个词永恒的生命力。

这个故事之所以经久不衰，不仅因为它有趣，更因为它揭示了一个深刻的隐喻：软件中的错误，就像藏在继电器缝隙中的飞蛾——它们不是被设计出来的，而是在你不经意间悄悄潜入的。你无法通过**禁止**飞蛾来解决问题，你只能通过系统化的**检查**来发现它们。这正是本讲要讨论的核心问题：错误是什么？错误从哪里来？我们如何处理错误？又如何通过测试来发现错误？

4.1　程序错误的本质与分类

4.1.1　一个抽奖程序的启示

在进入严肃的技术讨论之前，让我们先看一个轻松的故事。

某互联网公司的年会上，一位程序员负责编写抽奖程序。三等奖抽出后，台下的程序员们开始窃窃私语：“你觉不觉得这个抽奖程序写得有点问题啊？”“很可能有 *bug*。”

写程序的同事不服气：“抽奖程序是我写的。哪有问题？你讲清楚？”

于是，台下的程序员们开始了一场即兴的代码审查。“你考虑到工号是不连续的吗？”“当然，我重新组织了一个 *list*。”“已经抽中的人要从 *list* 里移走，你写了吗？”“废话！……这要忘？我还写啥程序！”“三等奖有 *50* 个，你是连续取了 *50* 次还是一次抽出来 *50* 个数？”“当然是一次取 *50* 个啊。”“你随机数怎么生成的，种子哪来的？”“肯定没问题，我为了可靠都没自己写随机数，直接用的 *random.org* 的 *API*，没话说了吧？”

这组漫画生动地展示了一个看似简单的程序背后潜藏的复杂性。一个抽奖程序，功能描述只需要一句话——“从 N 个员工中随机抽取 K 个中奖者”——但实现时需要考虑的问题远比预期的多：工号不连续怎么办？中奖者需要去重吗？随机数的质量够不够？网络 API 调用失败了怎么处理？每一个未被考虑到的问题，都是一个潜在的 bug。

这个故事还揭示了另一个重要的现象：**写代码的人往往看不到自己代码中的问题**。不是因为他们不够聪明，而是因为他们的思维路径和代码的逻辑路径是同构的——他们沿着自己预设的正常路径思考，而 bug 往往藏在那些“没想到”的异常路径上。这就是为什么软件测试需要独立的第三方，为什么代码审查（Code Review）是现代软件工程中不可或缺的实践。

4.1.2　错误的三层分类

程序中的错误可以按照它们被发现的时机分为三个层次。

第一层是**编译时错误**和**连接时错误**。这是最“友好”的错误——编译器和连接器会直接告诉你哪里出了问题。语法错误——比如少写了一个分号、括号不匹配——编译器会精确地指出错误的位置。类型错误——比如把一个字符串赋值给整数变量——类型检查系统会在编译阶段拦截。函数重定义或函数不存在——连接器会报告符号冲突或未定义的引用。这类错误虽然烦人，但代价最低，因为它们在代码运行之前就被发现了。

一个值得深思的问题是：为什么现代编程语言的设计者投入如此大的精力来增强类型系统？从 *C* 语言的弱类型到 *C++* 的强类型，从 *Java* 的泛型到 *Rust* 的所有权系统，每一代语言都在试图把更多的错误从运行时“前移”到编译时。原因很简单：编译时发现的错误，修复成本近乎为零；运行时才暴露的错误，修复成本可能是天文数字。类型系统的本质，是用编译器的计算能力来替代程序员的注意力。

第二层是**运行时错误**。程序通过了编译和连接，成功生成了可执行文件，但在运行过程中崩溃或产生异常。运行时错误又可以细分为三类：由计算机硬件或操作系统检测出的错误——比如除以零、访问非法内存地址、栈溢出；由标准库或第三方库检测出的错误——比如数组越界访问（C++ 的 `vector::at()` 会抛出 `out_of_range` 异常）、文件打开失败；由用户程序自身的检查代码检测出的错误——比如前置条件不满足、输入数据超出合法范围。

第三层是**逻辑错误**。这是最危险的错误——程序能够正常运行并得到输出结果，但

结果是错的。逻辑错误之所以危险，是因为它不会触发任何崩溃或异常，不会有任何错误提示，程序看起来一切正常——只是结果不对。产生逻辑错误的原因有很多：你所理解的程序逻辑本身就是错误的；你编写的程序不是你所设想的程序（比如运算符优先级搞错了）；某条语句出现了“低级错误”（比如把 == 写成了 =）。

我们来看一个具体的例子。下面这段程序用于计算一天中每小时气温的最高值、最低值和平均值：

```
double high_temp = 0; double low_temp = 0;

while (cin >> temp) { ... if (temp > high_temp) high_temp = temp;

  if (temp < low_temp) low_temp = temp; }
```

这段程序能正常编译、正常运行、正常输出——但结果很可能是错的。问题在于 high_temp 和 low_temp 都被初始化为 0。如果所有气温都是正数，low_temp 将永远是 0，因为没有任何气温比 0 更低；如果所有气温都是负数，high_temp 将永远是 0。正确的做法是用第一个读入的气温值来初始化这两个变量，或者使用 std::numeric_limits 的极值。此外，当没有输入数据时，程序会执行除以零操作——又一个运行时错误。

这个例子说明了一个重要的原则：逻辑错误往往不是因为程序员不够聪明，而是因为程序员对边界条件和初始状态的假设出了问题。最好的程序员不是那些从不犯错的人，而是那些能系统性地思考“什么情况下这段代码会出错”的人。

4.1.3 正确性与健壮性

讨论程序错误，首先需要澄清两个容易混淆的概念：**正确性**和**健壮性**。

一个正确的程序，是指对于所有合法输入，都能输出正确的结果。正确性关注的是“正常路径”——当一切按预期进行时，程序是否给出了正确的答案。一个健壮的程序，是指对于所有非法输入，都能输出有意义的错误信息而不是崩溃。健壮性关注的是“异常路径”——当意外发生时，程序是否能优雅地处理。

一个只有正确性而没有健壮性的程序，就像一辆只能在晴天平坦公路上行驶的汽车——在理想条件下表现完美，但一遇到坑洼路面就散架了。在产品级软件中，**错误处理代码的规模往往会超过正常流程的代码**。一个简单的文件复制程序，正常逻辑只有几行——打开源文件、打开目标文件、逐字节复制、关闭文件。但要让它成为健壮的产品级软件，你需要考虑：源文件不存在怎么办？目标文件已存在是否覆盖？磁盘空间不足怎么办？复制过程中源文件被删除了怎么办？权限不足怎么办？这些异常路径的处理代码，可能是正常逻辑代码量的十倍。

4.1.4　Ariane 5 的四十秒

1996 年 6 月 4 日，欧洲航天局的 Ariane 5 火箭在法属圭亚那库鲁航天中心发射升空。约 40 秒后，火箭偏离航线，自毁系统启动，价值约 3.7 亿美元的火箭和它搭载的四颗卫星化为一团火球。

事故调查委员会的报告揭示了一个令人震惊的原因：火箭的惯性导航系统中，一个 64 位浮点数被转换为 16 位有符号整数时发生了溢出。具体来说，与水平速度相关的内部变量 BH（Horizontal Bias）超出了 16 位整数的表示范围（−32768 到 32767），转换操作触发了一个 `Operand Error` 异常，这个异常没有被捕获，导致惯性导航系统停止工作，进而导致飞行控制系统接收到错误的姿态数据，最终导致火箭偏航自毁。

这个故事中最讽刺的部分是：这段有问题的代码是从 Ariane 4 火箭中直接复用的。在 Ariane 4 的飞行剖面下，水平速度值永远不会超出 16 位整数的范围，所以这段代码在 Ariane 4 上运行了多年从未出过问题。但 Ariane 5 的飞行剖面不同——它的初始加速度更大，水平速度值在发射后 37 秒就超出了 16 位整数的范围。

*Ariane 5 事故是软件错误放大效应的极端案例。一个类型转换溢出——从技术角度看这是最基础的运行时错误——造成了 3.7 亿美元的损失。更深层的教训是：**在新的上下文中复用旧代码时，旧代码中所有隐含的假设都需要被重新验证**。Ariane 4 的代码隐含了一个假设——水平速度值永远不会超过 32767——这个假设在 Ariane 4 的飞行剖面下是成立的，但在 Ariane 5 的飞行剖面下不再成立。没有人去检查这个假设，因为没有人意识到这个假设的存在。*

对 EDA 领域而言，这个教训同样适用。EDA 工具中的代码经常在不同的工艺节点、不同的设计规模之间复用。一个在 65nm 工艺下运行良好的 DRC 规则检查器，在 7nm 工艺下可能因为多重图案化（Multi-Patterning）带来的新约束而产生漏报。一个在百万门级设计上验证过的时序分析工具，在十亿门级的 SoC 上可能因为数据结构的规模假设而出现数值精度问题或内存溢出。EDA 工具的每一个 bug，其后果不仅是工具本身需要修复——如果这个 bug 导致芯片设计者做出了错误的设计决策，那么代价可能是一次失败的流片。在先进工艺节点上，一次流片的成本可能超过一千万美元，周期可能长达三到六个月。

4.2　错误的处理与防御

4.2.1　四种基本策略

程序在运行过程中发现错误后，可以采取四种基本的处理策略，它们的英文缩写恰好构成了一个容易记忆的组合：Abort、Retry、Ignore、Fail。年长一些的程序员对这四个词不会陌生——在 MS-DOS 时代，当软驱读取失败时，系统会弹出那个经典的提示：“*Abort, Retry, Ignore, Fail?*”

终止程序运行（Abort）是最简单粗暴的策略：输出错误信息，然后退出程序。当程序遇到一个它根本无法处理的严重错误时——比如关键的配置文件不存在、内存分配失败——终止运行是合理的选择。虽然这看起来不够“优雅”，但一个果断终止的程序，比一个在错误状态下继续运行、产生不可预测结果的程序要好得多。在 EDA 工具中，当 SPICE 仿真器检测到网表中存在浮空节点（floating node）时，立即报错终止通常比尝试“猜测”用户的意图要安全——一个基于错误网表的仿真结果，可能误导设计者做出灾难性的决策。

修复错误并继续运行（Retry）是最理想的策略：如果程序能够合理地应对错误，纠正问题后继续执行。最典型的场景是用户输入错误——当用户输入了一个非法的数值时，程序提示“无效输入，请重新输入”然后等待合法的输入。在 EDA 工具中，许多布局布线工具在遇到局部拥塞时，会自动回退几步、调整策略后重新尝试——这就是 Retry 策略在算法层面的应用。

忽略错误并继续运行（Ignore）是最危险的策略。忽略错误会使程序处于不正常的运行状态，并且很可能引发更严重的后续错误。因此，程序不应简单地忽略错误。然而在实践中，“忽略”有时是经过深思熟虑的选择——例如在音视频流播放中，丢弃一个损坏的帧比停止播放要好。关键在于区分“有意识地容忍”和“无意识地忽视”。

返回错误状态，交由上一级程序处理（Fail）是最常用的策略。当一个函数发现了错误但自己无法合理地处理时，它应当把错误信息传递给调用者，让更高层的代码来决定如何应对。这是软件分层架构中错误处理的基本原则：**每一层只处理自己能处理的错误，其余的向上传递。**

4.2.2 从 errno 到异常：错误传递机制的演进

如何将错误信息从发现错误的函数传递给调用者？这个看似简单的问题，在编程语言的历史上经历了漫长的演进。

C 语言采用了两种机制：返回错误码和全局变量 `errno`。标准库函数在出错时返回一个特殊值（比如 `fopen` 在失败时返回 `NULL`，`read` 在失败时返回 -1），同时设置全局变量 `errno` 来指示具体的错误类型。这种机制简单直接，但有三个严重的缺陷。其一，许多函数并不具有合理的“错误值”——如果一个函数的返回值本身就可能是任何整数，那么用哪个整数来表示“出错了”？其二，全局变量 `errno` 在早期实现中是多线程不安全的——不同线程可能同时修改它。虽然 POSIX 标准和 C11 标准后来要求 `errno` 为线程局部存储（thread-local），解决了多线程安全问题，但 `errno` 作为全局状态的设计理念本身仍然增加了模块间的耦合。其三，也是最致命的缺陷：**调用者可以简单地忽略返回值。**如果调用者不检查 `fopen` 的返回值就直接使用返回的指针，程序将在解引用空指针时崩溃——而这个崩溃点可能远离错误的真正源头，使得调试极为困难。

当错误需要跨越多层函数调用传递时，C 语言的错误码机制变得更加笨拙。假设函

数 f 调用 g，g 调用 h，而 h 中发生了错误。在 C 语言中，h 返回错误码给 g，g 必须检查这个错误码并决定是否继续向上传递给 f，f 再检查并决定如何处理。每一层都需要显式地编写错误检查和传递代码——这些代码与业务逻辑交织在一起，既影响可读性，又容易遗漏。

C++ 引入了**异常机制**来解决这些问题。异常的核心思想是**将错误检测和错误处理分离**：如果一个函数发现了自己不能处理的错误，它不是正常返回，而是**抛出**（throw）一个异常。这个异常会沿着调用栈向上传播，直到被某个调用者的 catch 块捕获。如果没有任何调用者捕获这个异常，程序终止运行。

异常机制的优势在于：中间层的函数不需要编写任何错误传递代码——异常会自动“穿透”它们。在上面的例子中，h 抛出异常，g 不需要做任何事情（异常自动穿过 g），f 在 try-catch 块中捕获并处理。C++ 标准库定义了一系列异常类型：out_of_range 用于数组越界，runtime_error 用于一般性运行时错误，bad_alloc 用于内存分配失败。程序员也可以定义自己的异常类型，使错误信息更加精确。

异常并非没有争议。在性能敏感的代码中——比如 EDA 工具的核心算法循环——异常的开销可能不可接受。异常的控制流是隐式的，可能使代码的执行路径难以追踪。此外，异常安全（exception safety）——确保在异常发生时程序状态保持一致——是 C++ 中最复杂的话题之一。因此，异常的使用需要遵循一条重要的规范：**仅用异常处理意外情况，不要用它控制正常的程序流程。**

4.2.3　现代错误处理：从异常到类型系统

异常机制虽然比错误码先进，但它有一个根本性的问题：**编译器不强制你处理异常。**一个函数可能抛出异常，但调用者可以完全不写 try-catch——编译器不会报错，只是运行时程序会崩溃。换句话说，异常将“是否处理错误”的决定权留给了程序员的自律，而非语言的强制。

近年来，编程语言的设计者们开始探索一种更根本的思路：**用类型系统来强制错误处理。**

C++17 引入的 std::optional<T> 是这一思路的起步。一个查找函数不再返回“找到则返回结果、找不到则返回 -1”这样的约定，而是返回 std::optional<T>——它要么包含一个有效值，要么为空。调用者在使用返回值之前，必须先检查它是否为空。这比用魔术数字（magic number）表示“没找到”要安全得多。C++23 提案中的 std::expected<T, E> 更进一步：它不仅表示“可能没有值”，还携带了具体的错误信息——返回值要么是成功的结果（类型 T），要么是一个错误描述（类型 E）。

Rust 语言将这一思路推到了极致。在 Rust 中，一个可能失败的函数返回 Result<T, E> 类型——与 C++ 的 std::expected 理念相似，但有一个关键区别：Rust 的编译器**强制**调用者处理两种可能。你无法“忘记”处理错误，因为不处理就无法获取返回值

——编译器会直接报错。换句话说，Rust 把“是否处理错误”从程序员的自律问题变成了编译器的强制约束。

*错误处理机制的演进——从 C 语言的 errno 到 C++ 的异常，再到 Rust 的 Result 和 C++ 的 std::expected——揭示了一条清晰的趋势：**让错误在编译时而非运行时被发现，让错误处理成为代码结构的一部分而非程序员的自律**。这与第二讲讨论的平台演进逻辑一脉相承：每一层抽象的进步，都在试图将更多的负担从人脑转移到工具。*

4.2.4 assert：调试期的卫兵

C 标准库的 `<assert.h>` 定义了 `assert` 宏，它是程序员在开发和调试阶段最忠实的卫兵。`assert` 接受一个布尔表达式，如果表达式为假，`assert` 会输出详细的错误信息——包括失败的表达式文本、源文件名和行号——然后终止程序运行。

`assert` 的典型用法是检查**程序员自己的假设**——那些“不应该发生”但“万一发生就说明有 bug”的条件。比如一个排序函数在返回之前，可以用 `assert` 检查输出数组是否真的有序；一个内存池管理器在分配内存后，可以用 `assert` 检查返回的指针是否在合法范围内。

`assert` 有一个巧妙的设计：在软件的调试阶段（Debug 配置），`assert` 正常工作，帮助程序员尽早发现错误；当软件最终发布时（Release 配置），通过定义 `NDEBUG` 宏，所有的 `assert` 语句会被编译器完全移除，不会产生任何运行时开销。这种“调试时严格、发布时高效”的策略，在 EDA 工具的开发中尤为重要——EDA 算法的核心循环可能执行数十亿次，任何额外的检查开销都会累积成不可忽视的性能损失。

`assert` 和异常的使用场景不同：`assert` 用于检查**不应该发生**的编程错误（即 bug），异常用于处理**可能发生**的运行时意外（如文件不存在、网络断开）。前者是程序员的错误——应当在测试阶段就被消灭；后者是外部环境的不确定性——需要在产品中被优雅地处理。

4.2.5 前置条件与后置条件：契约式设计

函数是完成独立计算任务的基本单位。每个函数对其参数和返回值都有一定的要求——参数必须满足什么条件函数才能正确工作？函数执行完毕后保证返回什么样的结果？这些要求被称为函数的**前置条件**（pre-condition）和**后置条件**（post-condition）。

以一个计算长方形面积的函数为例：前置条件是 `length` 和 `width` 都必须是正整数；后置条件是返回值必须是一个正整数（面积）。在函数的注释中明确说明前置条件和后置条件，并增加相应的检查语句，是应对程序错误的重要手段：

```
int area(int length, int width)

// 前置条件：length 和 width 是正整数
```

```
// 后置条件：返回一个正整数
{ if (length <= 0 || width <= 0)
    throw runtime_error(“area() pre-condition”);
  int a = length * width;
  if (a <= 0) throw runtime_error(“area() post-condition”);
  return a; }
```

这种思想被伯特兰•迈耶（Bertrand Meyer）在 1986 年系统化为**契约式设计**（Design by Contract）。迈耶在设计 Eiffel 语言时，将前置条件和后置条件作为语言的一等公民——每个函数的契约由编译器强制执行，而非依赖程序员的自律。虽然 C++ 没有原生支持契约式设计（C++20 曾将契约纳入标准草案，但因设计争议于 2019 年被移除，目前计划纳入 C++26），但其核心思想——**显式声明函数的输入输出约束并加以验证**——已经成为高质量软件开发的标准实践。

4.2.6　输入验证与边界检查

外部输入的正确性是程序不能控制的因素。无论用户是通过键盘输入、文件读取还是网络接收数据，程序都应当对输入进行严格的验证。这至少包括两个方面：检查输入数据是否在合法范围内，以及检查数据是否成功输入（比如期望读取数字时用户输入了字母）。

边界检查是另一个重要的防线。C 语言的数组不做下标越界检查——出于性能考虑，这是语言设计者有意为之的选择。C++ 的 `vector` 提供了两种访问元素的方式：`v[i]` 不检查下标越界（与 C 数组一致），`v.at(i)` 在下标越界时抛出 `out_of_range` 异常。选择哪种方式，取决于性能和安全之间的权衡——在 EDA 工具的核心算法循环中，数组访问可能发生数十亿次，使用 `at()` 带来的额外检查开销可能不可接受；但在接口层和数据预处理阶段，使用 `at()` 来确保安全是明智的选择。

4.2.7　静态分析与动态分析：让工具替人找错误

前面讨论的 assert、契约、输入验证、边界检查，都依赖于程序员主动编写防御性代码——这要求程序员预见到可能出现的错误，并在正确的位置插入检查。但人的注意力是有限的，总有遗漏。现代软件工程的一个重要趋势是**用自动化工具来系统性地发现错误**。

静态分析工具在不运行程序的情况下，通过分析源代码来发现潜在的错误。编译器本身就是最基本的静态分析工具——开启`-Wall -Wextra` 等警告选项后，编译器会报告许多潜在的问题，如未使用的变量、隐式的类型转换、可能的空指针解引用。专门的静态分析工具更加强大：Clang Static Analyzer 能检测内存泄漏和使用已释放的内存；

Cppcheck 能发现数组越界和未初始化变量的使用；PVS-Studio 能检测数百种 C++ 代码模式中的潜在错误。

C++ Core Guidelines 是由 C++ 语言的创造者比雅尼·斯特劳斯特鲁普（Bjarne Stroustrup）和赫布·萨特（Herb Sutter）主导编写的编码规范，它定义了一系列可被工具自动检查的规则。例如："*Bounds.1: Don't use pointer arithmetic*"——不要使用指针算术，因为指针算术极易出现越界访问；"*Bounds.2: Only index into arrays using constant expressions*"——只使用常量表达式作为数组下标，因为运行时计算的下标难以静态验证；"*Always initialize variables*"——总是初始化变量，因为使用未初始化的变量是未定义行为。Visual Studio 的代码分析工具可以直接检查这些规则的违反情况。

动态分析工具在程序运行时监控其行为，发现静态分析无法检测的问题。Valgrind 能检测内存泄漏、使用已释放的内存、未初始化内存的读取——这些问题在 C/C++ 中极为常见，而且往往不会立即导致崩溃，而是在程序运行很久之后才以不可预测的方式表现出来。Google 开发的 Sanitizer 家族——AddressSanitizer 检测内存错误，ThreadSanitizer 检测数据竞争，UndefinedBehaviorSanitizer 检测未定义行为——已经成为现代 C++ 开发的标准配置。

静态分析和动态分析是互补的。静态分析能覆盖所有代码路径（包括那些难以通过测试触发的路径），但它基于保守的近似分析，可能产生误报。动态分析只能覆盖实际执行的代码路径，但它检测到的问题都是真实存在的。在产品级软件的开发流程中，两者通常同时使用——静态分析在代码提交前自动运行，动态分析在测试执行时同步运行。

在现代软件工程实践中，这些工具被集成到 CI/CD（持续集成/持续部署）流水线中。每一次代码提交都会自动触发编译、静态分析、单元测试和动态分析——错误在进入代码库的几分钟内就被发现，而非等到数周后的集成测试阶段。这种"持续预防"的模式，与第三讲讨论的敏捷方法论一脉相承：**小步快跑，快速反馈，尽早发现问题**。

4.3 软件测试的方法与实践

前两节讨论的所有手段——错误处理代码、assert、契约式设计、静态分析——本质上都是**防御性**的：它们在错误已经发生或即将发生时做出反应。但还有一个更根本的问题：**如何在软件交付之前，主动地、系统性地去发现那些尚未暴露的错误？**这正是软件测试要回答的问题。

4.3.1 测试的哲学基础

本章的开篇格言来自迪杰斯特拉："*Testing shows the presence, not the absence of bugs.*"——测试能证明 bug 的存在，但不能证明 bug 的不存在。这句话初听令人沮丧，但它蕴含着深刻的方法论意义。

要理解这句话，需要认识到一个基本事实：穷举测试是不可能的。一个接受两个 32

位整数作为输入的函数，其输入空间有 2^{64} 种可能——即使每秒测试一百万种组合，也需要超过五十万年才能穷举。对于任何有实际意义的程序，其输入空间都是天文数字级别的，穷举测试在物理上不可行。因此，测试永远是一个**抽样**过程——我们从天文数字的输入空间中选取有限数量的测试用例，希望这些样本能代表整个空间。

由此引出软件测试的几条基本准则。测试的依据是用户需求——测试的标准不是“程序员认为程序应该怎样工作”，而是“用户期望程序怎样工作”。应该远在测试开始之前就制定出测试计划——测试计划应当在需求分析和设计阶段就开始制定，而非等到编码完成后才临时拼凑。应从小规模的测试开始，逐步进行大规模的测试——先确保每个零件都能工作，再确保零件组装后的系统能工作。应由独立的第三方从事测试工作——如前面抽奖程序的故事所示，代码的作者往往看不到自己代码中的问题。最后，也是最重要的一条：**测试决不能证明程序是正确的**——一个通过了所有测试的程序，只能说“在已测试的条件下没有发现错误”，不能说“没有错误”。

那么，如果测试不能证明正确性，它的价值是什么？答案是：测试的价值在于**增加信心**。每一个通过的测试用例，都在缩小 bug 可能隐藏的空间。一个好的测试方案，不是追求“覆盖所有情况”（这是不可能的），而是追求“用最少的测试用例覆盖最多的错误类型”。测试方案的设计——选择哪些输入作为测试用例——因此成为测试阶段的核心技术问题。

4.3.2　黑盒测试与白盒测试

根据测试者是否了解程序的内部结构，软件测试方法分为两大类。

黑盒测试（又称功能测试）将程序看作一个黑盒子，完全不考虑程序的内部结构和处理过程，只在程序的接口进行测试。测试者关心的是：程序的功能是否能按照规格说明书的规定正常使用？程序是否能适当地接收输入数据并产生正确的输出信息？程序运行过程中能否保持外部信息（如数据库或文件）的完整性？黑盒测试的核心问题是：如何从庞大的输入空间中选取最有代表性的测试用例？

白盒测试（又称结构测试）将程序看作一个透明的盒子，测试者完全知道程序的结构和处理算法，按照程序内部的逻辑测试程序。白盒测试的核心问题是：如何选择测试用例，使得程序中的主要执行通路都能按预定要求正确工作？这引出了“覆盖”的概念——不同的测试用例“覆盖”了程序逻辑的不同部分。

4.3.3　白盒测试：覆盖标准的层次

白盒测试的核心在于**覆盖标准**——按照测试用例覆盖源程序语句的详尽程度，可以定义从弱到强的多个层次。

语句覆盖是最弱的标准：选择足够多的测试用例，使得被测程序中的每条语句至少执行一次。语句覆盖看起来合理——如果一条语句从未被执行过，其中的错误自然无法

被发现。但语句覆盖的问题在于，它对程序的分支逻辑几乎没有检验能力。一个包含 `if-else` 的程序，可能只需要一个测试用例就能覆盖所有语句——但这并不意味着两个分支都被正确地测试了。

判定覆盖（又称分支覆盖）要求每个判定的每种可能结果至少执行一次——即每个 `if` 的 true 分支和 false 分支都至少被走过一次。判定覆盖比语句覆盖更强，但它仍有盲区：一个复合条件（如 `A > 1 AND B = 0`）在判定覆盖下只需要整个表达式取 true 一次、取 false 一次，但条件 `A > 1` 和 `B = 0` 各自取 true 和 false 的组合并没有被完整测试。

条件覆盖要求判定表达式中的每个条件都取到各种可能的结果。例如对于条件 `A > 1 AND B = 0`，条件覆盖要求 `A > 1` 和 `A <= 1` 各出现至少一次，`B = 0` 和 `B != 0` 也各出现至少一次。条件覆盖比判定覆盖更细致，但有时满足条件覆盖的测试用例却不满足判定覆盖——条件的各种取值可能恰好使判定始终为同一个结果。因此，实践中常使用**判定/条件覆盖**——同时满足判定覆盖和条件覆盖。

条件组合覆盖要求每个判定表达式中条件的各种可能组合都至少出现一次。对于包含两个条件的判定，这意味着需要测试 $2 \times 2 = 4$ 种组合。条件组合覆盖是一种相当严格的标准，它自然满足判定覆盖和条件覆盖。

路径覆盖是最强的标准：程序的每条可能路径（从开始到结束）都至少执行一次。若程序包含循环，则要求每个循环至少经过一次。路径覆盖在理论上是最完整的，但在实践中往往不可行——一个包含 n 个独立判定的程序，其路径数量可能达到 2^n，呈指数增长。

覆盖标准的层次从语句覆盖到路径覆盖，反映了一个普遍的工程权衡：更高的覆盖标准意味着更强的错误检测能力，但也意味着更多的测试用例和更高的测试成本。在实际项目中，100% 的路径覆盖几乎不可能实现，通常的目标是达到 90% 以上的语句覆盖或判定覆盖，并对关键模块进行条件组合覆盖或路径覆盖。

4.3.4 黑盒测试：等价划分与边界值分析

黑盒测试不关心程序的内部结构，而是着重测试软件的功能。其核心挑战是：在不可能穷举的输入空间中，如何选择最有效的测试用例？两种经典的方法是**等价划分**和**边界值分析**。

等价划分的基本思想是：如果把所有可能的输入（有效的和无效的）划分成若干个等价类，则可以合理地假定——每类中的一个典型值在测试中的作用与这一类中所有其他值的作用相同。因此，可以从每个等价类中只取一组数据作为测试用例。这样选取的测试用例最具有代表性，最可能发现程序中的错误。

边界值分析基于一个经验法则：处理边界情况时程序最容易发生错误。因此，使程序运行在边界情况附近的测试方案，暴露出程序错误的可能性更大一些。

以一个实际的例子来说明：某网上支付平台一次允许支付的金额范围为 0.01 到 5000.00 元。为这一输入数据，我们可以划分出四个等价类：小于 0.01（无效）、0.01 到 5000.00 之间（有效）、大于 5000.00（无效）、以及非数字输入（无效）。结合等价划分和边界值分析方法，我们设计出如下测试方案：选取 −1.00（等价类 1）、0.00（等价类 1 的边界值）、0.01（等价类 2 的边界值）、0.02（临近边界值）、一个中间值如 1764.42（等价类 2）、4999.99（临近边界值）、5000.00（等价类 2 的边界值）、5000.01（等价类 3 的边界值）、9999.99（等价类 3）、以及一个非数字输入如 3.14+42（等价类 4）。用仅仅十个测试用例，就覆盖了四个等价类和所有关键的边界值。

4.3.5　测试层次：从单元到系统

软件测试不是一步到位的，而是按照从小到大的层次逐步展开。

单元测试（又称模块测试）是最底层的测试，保证每个模块作为一个单元能正确运行。单元测试主要使用白盒测试技术，由开发者自己编写和执行。单元测试的作用远不止“检查代码中的错误”——它还有两个常被忽视的价值。其一，**改进设计**：在设计代码时考虑其可测试性，会自然地降低模块间的耦合，使接口更清晰——难以测试的代码，往往也是设计不良的代码。其二，**作为开发文档**：一组好的单元测试，展示了软件模块如何使用的最全面的实例，而且这份“文档”始终反映软件代码的最新状态，永远不会过时。

集成测试包括子系统测试和系统测试。子系统测试把经过单元测试的模块放在一起形成一个子系统来测试，重点测试模块之间的接口。系统测试把经过测试的子系统装配成一个完整的系统来测试，验证系统确实能提供需求说明书中指定的功能。

验收测试（又称确认测试）把软件系统作为单一的实体进行测试，在用户积极参与下进行。对于大批量出售的软件产品，验收测试又分为 Alpha 测试和 Beta 测试——Alpha 测试由用户在开发者的场所进行，在开发者的指导下进行；Beta 测试由最终用户在开发者不能控制的真实环境下进行。

4.3.6　单元测试框架

手工编写和维护单元测试非常繁琐。每个测试用例都需要：设置测试环境、调用被测函数、比较实际结果和预期结果、报告测试结果。为此，软件工程社区发展出了一系列**单元测试框架**——组织和运行单元测试的工具包。

JUnit 是最著名的单元测试框架，由埃里希 • 伽玛（Erich Gamma）和肯特 • 贝克（Kent Beck）于 1997 年设计。伽玛是“四人帮”（Gang of Four）之一，*Design Patterns* 的合著者；贝克是极限编程（XP）和测试驱动开发（TDD）的提出者。JUnit 最初为 Java 设计，后来被移植到数十种其他语言——C++ 的 CppUnit、Python 的 unittest、JavaScript 的 Jest 等。这种“编写代码来测试代码”的理念，从根本上改变了软件开发

的实践。

C++ 的单元测试框架选择丰富，其中 Google Test（gtest）是目前最广泛使用的。它提供了丰富的断言宏（`EXPECT_EQ`、`ASSERT_THROW` 等）、测试用例的自动注册和发现、测试结果的格式化输出，以及与 CI/CD 流水线的无缝集成。

4.3.7 现代测试实践

教科书中的测试方法——黑盒、白盒、覆盖标准——构成了测试的经典理论基础。但在过去二十年中，软件测试实践经历了深刻的变革，产生了若干值得关注的新方法。

测试驱动开发（Test-Driven Development，TDD）是肯特·贝克在 1999 年的 *Extreme Programming Explained* 中作为 XP 核心实践推广、并在 2003 年的专著 *Test Driven Development: By Example* 中系统化的一种开发方法论。TDD 的核心主张“反直觉”：**先写测试，再写代码**。开发者在编写任何功能代码之前，先编写一个描述该功能预期行为的测试用例——这个测试必然会失败，因为功能代码还不存在。然后，开发者编写最少量的代码使测试通过。最后，在测试的保护下重构代码以改善设计。这个“红（测试失败）—绿（测试通过）—重构”的循环，每次只前进一小步，但每一步都有测试作为安全网。TDD 的深层价值不在于测试本身，而在于它**迫使开发者在动手写代码之前先思考接口和行为**——这与契约式设计的思想不谋而合。

模糊测试（Fuzzing）是一种用随机或半随机的输入来自动发现程序错误的方法。与传统测试中人工设计测试用例不同，模糊测试器（Fuzzer）自动生成大量的随机输入，观察程序是否崩溃、挂起或产生异常行为。现代的模糊测试器——如 Google 的 AFL（American Fuzzy Lop）和 libFuzzer——使用覆盖率引导的策略：它们监控被测程序的代码覆盖率，优先变异那些能触发新代码路径的输入，从而高效地探索程序的输入空间。Google 通过 OSS-Fuzz 等模糊测试基础设施，在 Chrome 及众多开源项目中发现了数万个 bug，其中包括数千个安全漏洞。模糊测试在 EDA 工具的测试中同样有潜力——通过随机生成网表和约束文件来测试布局布线工具的鲁棒性，比人工编写测试用例要高效得多。

4.3.8 Pentium FDIV Bug：四亿七千五百万美元的查找表

1994 年，英特尔（Intel）发布了备受期待的 Pentium 处理器。数学教授托马斯·奈斯利（Thomas Nicely）在弗吉尼亚州林奇堡学院枚举孪生素数并计算布伦常数（Brun's constant）时，发现 Pentium 的浮点除法结果在某些特定操作数下出错。例如，$4195835.0 \div 3145727.0$ 的正确结果应为 $1.333820449\ldots$，但 Pentium 给出的结果是 $1.333739068\ldots$——从第四位有效数字开始就出错了。

调查发现，错误的根源在于 Pentium 浮点除法单元中一个查找表的编程失误。浮点除法单元的 PLA 查找表有 2048 个单元，其中 1066 个需要填入有效值。因脚本错误，

有 16 个条目被遗漏，其中 5 个会在特定操作数组合下被访问，导致除法结果出错。英特尔最初试图淡化这个问题，声称“一般用户每 27000 年才会遇到一次这个错误”。但公众和媒体的反应远超英特尔的预期——IBM 暂停了搭载 Pentium 处理器的电脑出货，学术界和金融界对计算结果的可靠性表达了严重担忧。最终，英特尔宣布无条件更换所有有缺陷的 Pentium 处理器，这次召回的财务影响约为 4.75 亿美元。

Pentium FDIV Bug 的教训是多层次的。从技术层面看，它是一个经典的逻辑错误——*2048* 个查找表单元中有 *16* 个因脚本错误被遗漏，其中 *5* 个会影响计算结果，但在特定输入下就会表现出来。从测试层面看，它暴露了等价划分的局限性——如果测试者把所有浮点数视为一个等价类而只测试几个典型值，这 *5* 个错误条目对应的输入很可能永远不会被测试到。英特尔后来大幅加强了处理器的形式验证和穷举测试投入。从商业层面看，它证明了软件（和硬件）质量问题的代价可以远远超出技术修复的成本——*4.75* 亿美元的召回费用，加上无法估量的品牌声誉损失，远远超过了在发布前进行更充分测试的成本。

4.3.9　从软件测试到芯片验证

软件测试的思想和方法，在芯片设计验证领域有着深刻的映射。理解这些映射，对于 EDA 工具的开发者而言尤为重要——因为 EDA 验证工具本身就是将软件测试理论应用于硬件验证的产物。而对于芯片设计者而言，理解这些映射有助于看清验证方法论的底层逻辑，而非仅仅把验证当作一组需要学习的工具和流程。

仿真验证是芯片验证中最传统的方法，它与软件的动态测试直接对应。验证工程师编写测试向量（testbench），驱动被验证的设计（DUT），检查输出是否符合预期。仿真验证面临的根本挑战与软件测试一样：输入空间太大，无法穷举。一个拥有数百个输入引脚的芯片，其输入组合数是天文级别的，而每次仿真可能需要数小时甚至数天。

早期的仿真验证完全依赖人工编写定向测试（directed test）——验证工程师根据对设计的理解，手工构造输入序列来覆盖特定的功能场景。这与软件测试中人工设计测试用例的思路完全一致，面临的困境也完全一致：人工方法能覆盖的场景数量有限，而且严重依赖验证工程师对设计的理解——如果工程师没有想到某个边角情况，那个情况就不会被测试到。Pentium FDIV Bug 的教训在硬件验证中同样适用：如果你不知道查找表的第 1054 个条目可能有问题，你就不会去写一个测试来验证它。

受约束的随机验证（Constrained Random Verification）是芯片验证领域的一次方法论革命，它的思想与软件测试中的模糊测试有异曲同工之妙。验证工程师不再逐一编写定向测试，而是定义输入信号的约束条件（比如“地址必须对齐到 4 字节边界”“数据长度在 1 到 256 之间”），然后让验证平台自动生成大量满足约束的随机激励。每一次仿真运行使用不同的随机种子，产生不同的输入序列。这种方法能探索到人工难以想到的输入组合，大幅提高了验证的覆盖率。

但随机验证带来了一个新的问题：如何知道随机激励已经“足够好”？你可能运行了十万次随机仿真，但怎么知道这十万次覆盖了设计的哪些方面、遗漏了哪些方面？这正是**覆盖率驱动验证**（Coverage-Driven Verification）要解决的问题。

覆盖率驱动验证是芯片验证领域对软件覆盖标准的延伸和深化。芯片验证中的覆盖率分为两类。**代码覆盖率**（code coverage）直接对应白盒测试的覆盖标准——行覆盖、分支覆盖、条件覆盖、表达式覆盖、有限状态机覆盖等概念在 RTL 代码中有精确的对应物。EDA 验证工具（如 Synopsys 的 VCS、Cadence 的 Xcelium）可以自动收集仿真过程中的代码覆盖率数据，验证工程师通过分析覆盖率报告来发现哪些代码路径没有被执行到。

功能覆盖率（functional coverage）则对应黑盒测试的等价划分。验证工程师根据设计规格书，定义一系列需要被验证的功能点（coverpoint）和功能点之间的交叉组合（cross coverage）。例如，对于一个缓存控制器，功能覆盖率可能包括：每种缓存操作（读命中、读缺失、写命中、写缺失、替换）都至少发生过一次；缓存操作与总线仲裁状态的所有组合都至少出现过一次；连续两次缺失紧接着一次命中的序列至少出现过一次。SystemVerilog 语言原生支持 `covergroup` 和 `coverpoint` 的定义，这使得功能覆盖率的定义、收集和分析可以完全自动化。

代码覆盖率和功能覆盖率是互补的。代码覆盖率回答“代码的哪些部分被执行过”，但不能回答“设计的功能是否被正确验证”——100% 的代码覆盖率并不意味着所有功能都被测试了。功能覆盖率回答“哪些功能场景被覆盖了”，但需要验证工程师对设计有深入的理解才能定义出有意义的功能点——遗漏的功能点不会被覆盖率报告显示。在实际的芯片验证项目中，验证团队通常要求代码覆盖率达到 95% 以上，功能覆盖率达到 100%（所有定义的功能点都被覆盖），才认为验证“收敛”。

形式验证（Formal Verification）则走了一条与测试完全不同的道路。它不是通过运行设计来发现错误，而是通过数学证明来验证设计满足某些性质——这是测试永远做不到的。形式验证的两个最重要的应用是模型检验和等价性检验。

模型检验（Model Checking）可以穷举地验证一个有限状态机的所有可达状态是否满足某个性质。验证工程师用断言（assertion）描述设计必须满足的性质——例如“请求信号拉高后，应答信号必须在 10 个时钟周期内拉高”“FIFO 永远不会同时为空和为满”——模型检验工具自动探索所有可能的状态，证明断言成立或给出一个反例。如果工具找到了一个违反断言的输入序列，它会报告这个序列的每一步状态转换，使得调试极为高效——这相当于不仅告诉你“有 bug”，还告诉你“怎么触发这个 bug”。

2008 年，英特尔的形式验证团队报告了一个著名的案例[1]：在 Core i7 处理器的执行引擎验证中，形式验证发现了多个仿真验证运行数月都未能发现的深层 bug。这些 bug 需要极其特殊的输入序列才能触发——特殊到随机仿真在统计上几乎不可能碰到。形式

[1]Kaivola, R. et al., “Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation,” *Proc. CAV*, pp. 414–429, 2009.

验证通过穷举状态空间找到了这些序列，证明了它在发现“深层 bug”方面的独特价值。

等价性检验（Equivalence Checking）可以证明两个设计在功能上完全等价——最常见的应用是验证综合后的门级网表与原始的 RTL 描述功能一致。在现代芯片设计流程中，等价性检验已经成为签核（signoff）的必要步骤：如果综合工具引入了功能错误，等价性检验会在流片之前将其拦截。Synopsys 的 Formality 和 Cadence 的 Conformal 是这一领域最常用的商业工具。

形式验证的局限在于**状态空间爆炸**——当设计的规模超过一定阈值后，需要验证的状态数量呈指数增长，形式验证工具无法在可接受的时间内完成。一个包含几百个触发器的模块通常可以被完整地形式验证，但一个包含数百万触发器的 SoC 就超出了当前工具的能力范围。因此，形式验证在实践中通常应用于三类场景：协议接口模块（总线协议、仲裁逻辑等状态机密集的模块）、关键控制逻辑（中断控制器、电源管理状态机）、以及综合后的等价性检验。对于完整 SoC 的全局功能验证，仿真仍然是不可替代的。

形式验证和仿真验证的关系，恰好对应迪杰斯特拉格言的两面：仿真验证“可以用来证明 bug 的存在”——它擅长快速地发现大量表层错误；形式验证“可以证明 bug 的不存在”——它擅长在有限范围内提供数学级别的正确性保证。两者不是替代关系，而是互补关系。现代芯片验证流程同时使用两种方法：仿真验证用于覆盖广度，形式验证用于覆盖深度。

可测试性设计（Design for Testability，DFT）解决的是另一个层面的问题：设计验证确认芯片的逻辑设计是正确的，但制造出来的物理芯片是否与设计一致？制造过程中可能引入缺陷——短路、断路、粒子沾污——这些缺陷需要在芯片出厂前被检测出来。DFT 在芯片中插入额外的测试结构来实现这一目标。

扫描链（scan chain）是最基本的 DFT 结构。它将芯片中所有的触发器串联成一条或多条移位寄存器链，使得测试设备可以从外部直接控制和观测芯片内部的每一个触发器状态。在正常工作模式下，这些触发器按照设计逻辑工作；在测试模式下，它们变成移位寄存器，测试向量可以被逐位移入，测试结果可以被逐位移出。扫描链的插入使得组合逻辑的测试变得可控——没有扫描链的芯片，内部状态对外部不可见，测试覆盖率受到严重限制。

内建自测试（Built-In Self-Test，BIST）更进一步——它在芯片内部集成了测试向量生成器和结果比较器，使得芯片能够自己测试自己，不依赖外部测试设备。存储器 BIST 是最常见的应用：芯片中的 SRAM 通常配备专门的 BIST 电路，能在上电时自动执行 March 算法来检测存储单元的各种缺陷模式。

DFT 的核心思想与软件中“为测试而设计”的理念一脉相承。软件中，我们通过降低模块耦合、暴露清晰接口来提高可测试性；硬件中，我们通过插入扫描链和 BIST 来提高可观测性和可控制性。两者的底层逻辑是相同的：**一个系统如果在设计时没有考虑可测试性，那么无论后期投入多少测试资源，都无法有效地验证它。**一个不可测试的芯片，就像一个接口不清晰、状态不可观测的软件模块——即使它的设计是正确的，你也

无法确认这一点。

软件测试和芯片验证的这种深层对应关系，揭示了一个更本质的命题：无论是软件还是硬件，验证的核心挑战都是同一个——在有限的时间和资源内，尽可能增加对设计正确性的信心。定向测试对应人工编写测试用例，随机验证对应模糊测试，覆盖率驱动验证对应白盒/黑盒覆盖标准，形式验证对应数学证明，可测试性设计对应为测试而设计——这些策略在软件和硬件两个领域中平行发展、相互借鉴，构成了一套完整的验证方法学。

4.4 本讲总结

本讲从程序错误的分类出发，讨论了错误的处理机制和测试方法。

错误的本质是什么？程序错误分为编译时错误、运行时错误和逻辑错误三个层次。编译时错误最友好、修复代价最低；逻辑错误最危险、修复代价最高。程序的正确性关注正常路径，健壮性关注异常路径——在产品级软件中，异常路径的处理代码往往超过正常流程的代码。Ariane 5 的悲剧告诉我们，一个基础的类型转换错误就足以造成数亿美元的损失；Pentium FDIV Bug 则证明，查找表中少数被遗漏的条目，在特定条件下就会导致计算结果出错。

如何系统性地应对错误？错误处理机制的演进——从 C 语言的 errno 到 C++ 的异常，再到 Rust 的 `Result` 类型和 C++ 的 `std::expected`——遵循一条清晰的趋势：让编译器而非程序员来保证错误不被遗漏。契约式设计将函数的输入输出约束显式化，assert 在调试阶段充当卫兵，静态分析和动态分析工具将人工代码审查的能力自动化并规模化。

如何通过测试发现错误？测试是一个抽样过程，其核心问题是如何用最少的测试用例覆盖最多的错误类型。白盒测试通过覆盖标准——从语句覆盖到路径覆盖——指导测试用例的选取；黑盒测试通过等价划分和边界值分析从功能角度选取代表性输入。测试驱动开发将测试从“事后检查”提升为“驱动设计”的力量，模糊测试用自动化手段探索人工难以覆盖的输入空间。在芯片验证领域，仿真验证、覆盖率驱动验证、形式验证和可测试性设计，是软件测试思想在硬件世界的映射和延伸。

贯穿本讲的底层哲学，是布莱恩·克尼根（Brian Kernighan）和菲利普·普劳格（P.J. Plauger）在 *The Elements of Programming Style* 中的信条：**不要评论不好的代码——重写它**。这句话的深意远超代码注释本身。面对错误，有两种态度：一种是在问题代码上打补丁、加注释、写绕行方案（workaround），让它“勉强能用”；另一种是承认问题的根源在于设计本身，有勇气重写、重构、从源头消除错误的温床。错误处理机制的演进，本质上是在追求“让错误无处藏身”的设计；测试方法的层层递进，本质上是在逼近“让错误无所遁形”的理想；静态分析工具的进步，本质上是在实现“让机器替人找错误”的自动化。三条线索汇聚于同一个结论：**好的软件工程师不是不犯错的人，而**

是有能力、有勇气、有工具去系统性地发现和消灭错误的人。

如果说第一讲的 DRY 原则回答了“不该做什么”——不要重复自己，第二讲的 YAGNI 原则回答了“什么时候做”——面临确凿需求时才实现，第三讲的简单性原则回答了“怎么做”——用最简单的方式做正确的事，那么本讲的**“不要评论不好的代码——重写它”**回答了面对错误时应当持有的态度——**与其掩盖问题，不如直面问题。**

课后思考

1. Ariane 5 的类型转换溢出和 Pentium FDIV 的查找表遗漏，分别属于哪种类型的错误？如果你是当时的测试负责人，你会设计什么样的测试方案来发现它们？

2. 你使用过的 EDA 工具（如 Vivado、Cadence Virtuoso、HSPICE 等）是如何报告和处理错误的？它的错误处理策略更接近 Abort、Retry、Ignore 还是 Fail？你认为这种策略合理吗？

3. 迪杰斯特拉说“测试只能证明有 bug，不能证明没有 bug”。形式验证可以在数学上证明正确性，那为什么形式验证没有取代测试成为芯片验证的主流方法？在你看来，测试和形式验证各自的适用边界在哪里？

参考文献

[1] Kernighan, B. W. and Plauger, P. J., *The Elements of Programming Style*, 2nd ed., McGraw-Hill, 1978.

[2] Dijkstra, E. W., “Notes on Structured Programming,” in *Structured Programming*, Academic Press, pp. 1–82, 1972.

[3] Lions, J. L., et al., “Ariane 5 Flight 501 Failure: Report by the Inquiry Board,” European Space Agency, 1996.

[4] Nicely, T. R., “Pentium FDIV Flaw FAQ,” `https://www.trnicely.net/pentbug/pentbug.html`, 1995.

[5] Meyer, B., “Applying ‘Design by Contract’,” *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[6] Stroustrup, B. and Sutter, H., “C++ Core Guidelines,” `https://github.com/isocpp/CppCoreGuidelines`, 2015–present.

[7] Beck, K., *Test-Driven Development: By Example*, Addison-Wesley, 2003.

[8] Gamma, E. and Beck, K., “JUnit: A Cook’s Tour,” *Java Report*, vol. 4, no. 5, 1999.

[9] Nethercote, N. and Seward, J., "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proc. PLDI*, pp. 89–100, 2007.

[10] Serebryany, K., et al., "AddressSanitizer: A Fast Address Sanity Checker," *Proc. USENIX ATC*, pp. 309–318, 2012.

[11] Zalewski, M., "American Fuzzy Lop (AFL) Technical Whitepaper," `https://lcamtuf.coredump.cx/afl/technical_details.txt`, 2014.

[12] Clarke, E. M., Grumberg, O., and Peled, D. A., *Model Checking*, MIT Press, 1999.

[13] Bergeron, J., *Writing Testbenches: Functional Verification of HDL Models*, 2nd ed., Springer, 2003.

[14] Kaivola, R., et al., "Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation," *Proc. CAV*, pp. 414–429, 2009.

[15] 赵文庆, 周学功, 《软件设计和开发》, 复旦大学出版社, 2013.