编写可测试的代码

四个使代码难以测试的设计缺陷

按空格进入下一页 →

关于本指南

本指南确定了四个使代码难以测试的主要缺陷：

1. **构造函数执行实际工作**
2. **深入协作者**
3. **脆弱的全局状态和单例**
4. **类职责过多**

这些缺陷经常出现在现实世界的代码中，使单元测试变得困难或不可能。 理解这些模式有助于我们编写更易维护和可测试的代码。

重要概念解释

在深入学习这四个缺陷之前，我们需要先理解一些重要概念：

- **协作者(Collaborator)**：类为了完成其职责而需要与其他类进行交互，这些交互的类被称为协作者。
- **依赖注入(Dependency Injection)**：通过构造函数或方法参数将依赖传递给类，而不是在类内部创建依赖。这使代码更加灵活和可测试，因为我们可以在测试时传入模拟对象。
- **模拟对象(Mock)**：在测试中用来替代真实对象的特殊对象，可以验证方法调用和设置期望行为。模拟对象帮助我们隔离被测试的代码单元，使测试更加可靠。
- **得墨忒耳定律(Law of Demeter)**：一个对象应该只与直接朋友交谈，不与陌生人的陌生人交谈，避免链式调用。

缺陷 #1: 构造函数执行实际工作

问题所在

- 构造函数应该只将参数分配给字段
- 当构造函数执行实际工作时，会使得：
  - 在测试中创建实例变得困难
  - 用测试替身替换协作者变得困难
  - 理解类依赖关系变得困难

缺陷 #1: 构造函数执行实际工作 (继续)

警告信号及说明

- **在构造函数中或字段声明中使用 `new` 关键字**

```
class UserService {
  // 在字段声明中使用new
  private: UserRepository* repo = new DatabaseUserRepository();
};
```

说明：这种方式将创建依赖的职责放在了类内部，使得难以替换为测试替身，也隐藏了类的真实依赖。

缺陷 #1: 构造函数执行实际工作 (继续)

警告信号及说明 (继续)

- **在构造函数中或字段声明中调用静态方法**

```
class OrderService {
  private: Logger logger;
  public: OrderService() : logger(LoggerFactory::getLogger()) {}
};
```

说明：静态方法调用创建了隐式依赖，同样难以替换为测试替身，且使类与特定实现紧密耦合。

缺陷 #1: 构造函数执行实际工作 (继续)

警告信号及说明 (继续)

- **构造函数中除了字段赋值之外的任何操作**

```
class EmailService {
  public: EmailService(string configPath) {
    // 除了字段赋值还有其他操作
    ConfigReader reader(configPath);
    this->config = reader.read();  // 复杂操作
    this->initSmtp();              // 方法调用
  }
};
```

说明：构造函数应该只负责初始化字段，而不是执行复杂的业务逻辑或调用其他方法。

缺陷 #1: 构造函数执行实际工作 (继续)

警告信号及说明 (继续)

- **构造函数完成后对象未完全初始化**

```
class PaymentProcessor {
  public: PaymentProcessor() {
    // 构造函数为空，需要额外调用init()
  }
  public: void init() { /* 初始化逻辑 */ }
};
```

说明：对象应该在构造函数完成后就处于完全可用状态，不需要额外的初始化步骤。

缺陷 #1: 构造函数执行实际工作 (继续)

警告信号及说明 (继续)

- **构造函数中存在控制流（条件或循环逻辑）**

```
class ReportGenerator {
  public: ReportGenerator(ReportType type) {
    // 构造函数中有条件逻辑
    if (type == ReportType::PDF) {
      this->formatter = new PdfFormatter();
    } else if (type == ReportType::CSV) {
      this->formatter = new CsvFormatter();
    }
  }
};
```

说明：构造函数中包含条件逻辑使类的行为变得复杂且难以预测，也增加了测试的复杂性。

缺陷 #1: 构造函数执行实际工作 (继续)

警告信号及说明 (继续)

- **在构造函数中进行复杂的对象图构造**

```
class ShoppingCart {
  public: ShoppingCart(User user) {
    // 复杂的对象图构造
    this->user = user;
    this->discountService = new DiscountService(
      new UserDiscountProvider(user),
      new SeasonalDiscountProvider(),
      new CouponDiscountProvider()
    );
  }
};
```

说明：在构造函数中构造复杂的对象图使得类承担了过多职责，也使测试需要创建大量依赖对象。

缺陷 #1: 构造函数执行实际工作 (继续)

警告信号及说明 (继续)

- **添加或使用初始化块**

```
class DataProcessor {
  private: vector<string> filters;

  // 初始化块
  public: DataProcessor() {
    filters.push_back("filter1");
    filters.push_back("filter2");
    filters.push_back("filter3");
  }
};
```

说明：初始化块中的逻辑应该移到专门的方法中，保持构造函数的简洁。

缺陷 #1: 示例

之前：难以测试

```
class EmailSender {
private:
  SmtpClient client;  // 在构造函数中创建
  Logger logger;

public:
  EmailSender(string configPath) {
    // 构造函数中执行实际工作！
    Config config =
      new ConfigFileReader(configPath).read();
    this.client = new SmtpClient(
        config.getHost(),
        config.getPort());
    this.logger = new FileLogger("email.log");
    if (!client.isConnected()) {
      throw new ConnectionException();
    }
  }
};
```

问题：

- 无法模拟 "SmtpClient" 或 "Logger"
- 无法在没有网络连接的情况下进行测试
- 难以轻松测试错误处理路径

缺陷 #1: 示例

之后：可测试的设计

```
class EmailSender {
private:
  SmtpClient& client;  // 注入的依赖
  Logger& logger;      // 注入的依赖

public:
  // 构造函数只分配字段
  EmailSender(SmtpClient& client, Logger& logger)
    : client(client), logger(logger) {
  }

  void sendEmail(Email email) {
    logger.log("Sending email");
    client.send(email);
  }
};
```

优势：

- 易于在测试中注入模拟对象
- 构造函数中没有实际工作
- 依赖关系清晰

测试示例

```
// 模拟对象
class MockSmtpClient : public SmtpClient {
public:
    MOCK_METHOD(void, send, (Email email), (override));
};

class MockLogger : public Logger {
public:
    MOCK_METHOD(void, log, (const std::string& message), (override));
};
```

```
// 测试用例
TEST(EmailSenderTest, SendEmailLogsAndSends) {
    // Arrange (准备)
    MockSmtpClient mockClient;
    MockLogger mockLogger;
    EmailSender sender(mockClient, mockLogger);
    Email testEmail("test@example.com", "Test Subject", "Test Body");

    // 设置期望
    EXPECT_CALL(mockLogger, log("Sending email"));
    EXPECT_CALL(mockClient, send(testEmail));

    // Act (执行)
    sender.sendEmail(testEmail);

    // Assert (断言)
    // 期望已经被验证
}
```

```cpp
// 测试异常
TEST(EmailSenderTest, SendEmailHandlesException) {
    // Arrange
    MockSmtpClient mockClient;
    MockLogger mockLogger;
    EmailSender sender(mockClient, mockLogger);
    Email testEmail("test@example.com", "Test Subject", "Test Body");

    // 设置期望并抛出异常
    EXPECT_CALL(mockLogger, log("Sending email"));
    EXPECT_CALL(mockClient, send(testEmail))
        .WillOnce(testing::Throw(std::runtime_error("Network error")));
    EXPECT_CALL(mockLogger, log("Failed to send email: Network error"));

    // Act & Assert
    sender.sendEmail(testEmail);
}
```

说明：

通过构造函数注入依赖，而不是在构造函数中执行实际工作，可以轻松地测试类。

缺陷 #2: 深入协作者

问题所在

- 类需要其他对象只是为了获取更多的对象（深入协作者）
- 违反得墨忒耳定律
- 创建类之间的紧密耦合
- 使测试变得更加困难，因为您需要创建复杂的对象图

缺陷 #2: 深入协作者 (继续)

警告信号及说明

- **传入的对象从未直接使用（仅用于获取其他对象）**

```
class OrderService {
  public: void processOrder(Order order, DatabaseManager dbManager) {
    // dbManager仅用于获取Connection
    Connection conn = dbManager.getConnection();
    OrderRepository repo = new OrderRepository(conn);
    repo.save(order);
  }
};
```

说明：传入 "DatabaseManager" 只是为了获取 "Connection"，这表明类与 "DatabaseManager" 的耦合度过高，应该直接依赖所需的对象。

- **违反得墨忒耳定律：方法调用链通过对象图走过不止一个点(.)**

```
class UserService {
  public: void sendNotification(User user) {
    // 违反得墨忒耳定律的链式调用
    user.getProfile().getPreferences().getNotificationSettings().getEmail();
  }
};
```

说明：这种链式调用称为"火车残骸"，增加了代码的脆弱性，任何一个环节的改变都可能影响整个调用链。

- **参数或字段中的可疑名称**

```
class ReportGenerator {
  private: ApplicationContext context;  // 可疑的"上下文"名称

  public: void generateReport() {
    // 深入协作者
    ReportConfig config = context.getConfiguration().getReportSettings();
  }
};
```

说明：像 "context"、"environment"、"manager" 这样的名称通常表明类可能在深入协作者，应该明确需要的具体依赖。

缺陷 #2: 示例对比

之前：难以测试

```
class UserRegistration {
private:
  DatabaseManager dbManager;

public:
  UserRegistration(DatabaseManager dbManager)
    : dbManager(dbManager) {
  }

  void registerUser(UserData userData) {
    // 深入协作者：
    // 通过dbManager获取ConnectionPool，再获取Connection
    Connection conn = dbManager
      .getConnectionPool()
      .getConnection();
    UserRepository repo = new UserRepository(conn);
    repo.save(userData);
  }
};
```

问题：

- 需要模拟复杂的对象图 (DatabaseManager → ConnectionPool → Connection)
- 类之间紧密耦合
- 难以隔离测试

缺陷 #2: 示例对比 (继续)

之后：可测试的设计

```
class UserRegistration {
private:
  UserRepository& userRepository;

public:
  UserRegistration(UserRepository& userRepository)
    : userRepository(userRepository) {
  }

  void registerUser(UserData userData) {
    // 直接使用协作者
    userRepository.save(userData);
  }
};
```

优势：

- 清晰的单一依赖
- 易于模拟 UserRepository
- 遵循得墨忒耳定律

缺陷 #3: 脆弱的全局状态和单例

问题所在

- 全局状态使代码不可预测
- 单例创建隐藏依赖
- 测试变得依赖顺序
- 难以并行运行测试
- 难以隔离被测系统

缺陷 #3: 脆弱的全局状态和单例 (继续)

警告信号及说明

- **添加或使用单例**

```
class UserService {
  public: void createUser(User user) {
    // 使用单例
    DatabaseConnection conn = DatabaseManager::getInstance().getConnection();
    conn.save(user);
  }
};
```

说明：单例模式隐藏了类的依赖关系，使得难以替换为测试替身，也使得测试之间相互影响。

缺陷 #3: 脆弱的全局状态和单例 (继续)

警告信号及说明 (继续)

- **添加或使用静态字段或静态方法**

```
class OrderService {
  private: static Cache cache;  // 静态字段

  public: Order getOrder(int id) {
    // 使用静态方法
    return CacheManager::getCachedOrder(id);
  }
};
```

说明：静态字段和方法创建了全局状态，使得测试之间相互影响，难以并行运行。

缺陷 #3: 脆弱的全局状态和单例 (继续)

警告信号及说明 (继续)

- **添加或使用静态初始化块**

```
class Logger {
  private: static Logger instance;

  // 静态初始化块
  static {
    instance = new Logger();
    instance.setLevel(LogLevel.INFO);
    instance.setFile("app.log");
  }
};
```

说明：静态初始化块使得类的行为在测试间难以控制和修改。

缺陷 #3: 脆弱的全局状态和单例 (继续)

警告信号及说明 (继续)

- **添加或使用注册表**

```
class PaymentService {
  public: void processPayment(Payment payment) {
    // 使用注册表
    PaymentProcessor processor = ServiceRegistry.get("paymentProcessor");
    processor.process(payment);
  }
};
```

说明：注册表和服务定位器隐藏了真实的依赖关系，使得难以理解类的实际需求。

缺陷 #3: 脆弱的全局状态和单例 (继续)

警告信号及说明 (继续)

- **添加或使用服务定位器**

```
class NotificationService {
  public: void sendNotification(Notification notification) {
    // 使用服务定位器
    EmailService emailService = ServiceLocator.getEmailService();
    emailService.send(notification);
  }
};
```

说明：服务定位器虽然比单例稍好，但仍然隐藏了依赖关系，不利于测试。

缺陷 #3: 示例

之前：难以测试

```
class OrderProcessor {
public:
  void processOrder(Order order) {
    // 使用全局状态和单例
    PaymentService
      .getInstance()
      .charge(order.getAmount());
    InventoryManager
      .getInstance()
      .updateStock(order.getItems());
    Logger
      .getLogger()
      .log("Order processed: " + order.getId());
  }
};
```

问题：

- 无法模拟单例实例
- 测试通过全局状态相互影响
- 难以轻松测试错误场景
- 难以并行运行测试

之后：可测试的设计

```
class OrderProcessor {
private:
  PaymentService& paymentService;
  InventoryManager& inventoryManager;
  Logger& logger;

public:
  OrderProcessor(PaymentService& paymentService,
                 InventoryManager& inventoryManager,
                 Logger& logger)
    : paymentService(paymentService),
      inventoryManager(inventoryManager),
      logger(logger) {
  }

  void processOrder(Order order) {
    paymentService.charge(order.getAmount());
    inventoryManager.updateStock(order.getItems());
    logger.log("Order processed: " + order.getId());
  }
};
```

优势：

- 依赖关系明确
- 易于注入模拟对象
- 无全局状态
- 测试可以独立运行

补充：为什么单例模式会带来问题？

很多人都听说过全局变量是魔鬼。

因为全局变量带来了两个问题：

- 全局变量是全局的
- 全局变量影响范围不可控, 难以追踪状态

单例模式解决了什么？

- 单例模式的状态是私有的 √
- 单例模式影响范围不可控, 难以追踪状态 ×

补充：替代单例模式的方法

工厂模式 + 非单例对象 + 依赖注入 + 接口控制反转

```
// 定义接口（抽象基类）

// 支付服务接口
class IPaymentService {
public:
    virtual ~IPaymentService() = default;
    virtual void charge(double amount) = 0;
};

// 库存管理接口
class IInventoryManager {
public:
    virtual ~IInventoryManager() = default;
    virtual void updateStock(const std::vector<Item>& items) = 0;
};

// 日志接口
class ILogger {
public:
    virtual ~ILogger() = default;
    virtual void log(const std::string& message) = 0;
};
```

```cpp
// 具体实现
class PaymentService : public IPaymentService {
public:
    void charge(double amount) override {
        // 实际支付逻辑
    }
};

class InventoryManager : public IInventoryManager {
public:
    void updateStock(const std::vector<Item>& items) override {
        // 实际库存更新逻辑
    }
};

class Logger : public ILogger {
public:
    void log(const std::string& message) override {
        // 实际日志记录逻辑
    }
};
```

```cpp
// 工厂接口
class IServiceFactory {
public:
    virtual ~IServiceFactory() = default;
    virtual std::shared_ptr<IPaymentService> getPaymentService() = 0;
    virtual std::shared_ptr<IInventoryManager> getInventoryManager() = 0;
    virtual std::shared_ptr<ILogger> getLogger() = 0;
};
```

```cpp
// 具体工厂实现 - 缓存已创建的对象，避免重复创建
class ServiceFactory : public IServiceFactory {
private:
    // 缓存对象
    std::shared_ptr<IPaymentService> paymentService;
    std::shared_ptr<IInventoryManager> inventoryManager;
    std::shared_ptr<ILogger> logger;

public:
    // 获取或创建支付服务（单例模式）
    std::shared_ptr<IPaymentService> getPaymentService() override {
        if (!paymentService) {
            paymentService = std::make_shared<PaymentService>();
        }
        return paymentService;
    }

    // 获取或创建库存管理服务（单例模式）
    std::shared_ptr<IInventoryManager> getInventoryManager() override {
        // 省略创建逻辑
    }

    // 获取或创建日志服务（单例模式）
    std::shared_ptr<ILogger> getLogger() override {
        // 省略创建逻辑
    }
};
```

依赖注入

```
// 使用依赖注入的订单处理器
class OrderProcessor {
private:
    std::shared_ptr<IPaymentService> paymentService;
    std::shared_ptr<IInventoryManager> inventoryManager;
    std::shared_ptr<ILogger> logger;

public:
    // 注入服务对象
    OrderProcessor(std::shared_ptr<IPaymentService> paymentService,
                   std::shared_ptr<IInventoryManager> inventoryManager,
                   std::shared_ptr<ILogger> logger)
        : paymentService(paymentService),
          inventoryManager(inventoryManager),
          logger(logger) {
    }

    void processOrder(const Order& order) {
        paymentService->charge(order.getAmount());
        inventoryManager->updateStock(order.getItems());
        logger->log("Order processed: " + order.getId());
    }
};
```

说明：

这种组合方式结合了多种优秀的设计模式，适用于替代传统的单例模式：

1. **接口抽象**：通过接口（抽象类）定义服务契约，实现控制反转
2. **工厂模式**：使用工厂管理对象创建和生命周期
3. **对象缓存**：工厂缓存已创建的对象，避免重复创建，实现类似单例的效果
4. **依赖注入**：通过构造函数注入依赖，便于测试和替换
5. **智能指针**：使用 `std::shared_ptr` 管理对象生命周期，自动引用计数

优势：

- **可测试性**：可以轻松注入模拟对象进行测试
- **对象复用**：工厂缓存已创建的对象，避免重复创建开销
- **高度解耦**：类依赖于接口而非具体实现
- **灵活配置**：可以在运行时决定使用哪种工厂实现
- **生命周期管理**：使用智能指针自动管理内存
- **符合开闭原则**：添加新服务类型不需要修改现有代码

可以继续改进的地方：

- **工厂代码复用**: 如果可以使用模板元编程，则可以使用模板元编程来实现缓存对象的工厂类

测试示例：

```
// 在测试中使用模拟对象
class MockPaymentService : public IPaymentService {
public:
    MOCK_METHOD(void, charge, (double amount), (override));
};

class MockInventoryManager : public IInventoryManager {
public:
    MOCK_METHOD(void, updateStock, (const std::vector<Item>& items), (override));
};

class MockLogger : public ILogger {
public:
    MOCK_METHOD(void, log, (const std::string& message), (override));
}
```

```
// 测试代码
TEST(OrderProcessorTest, ProcessOrderChargesPayment) {
    auto mockPayment = std::make_shared<MockPaymentService>();
    auto mockInventory = std::make_shared<MockInventoryManager>();
    auto mockLogger = std::make_shared<MockLogger>();

    // 设置期望
    EXPECT_CALL(*mockPayment, charge(100.0));

    // 注入模拟对象
    OrderProcessor processor(mockPayment,
                             mockInventory,
                             mockLogger);

    Order order(100.0, items);
    processor.processOrder(order);
}
```

缺陷 #4: 类职责过多

问题所在

- 具有多个职责的类难以理解
- 难以测试所有场景
- 一个职责的更改可能破坏其他职责
- 违反单一职责原则

缺陷 #4: 类职责过多 (继续)

警告信号及说明

- **总结类的作用时包含"和"字**

```
// 这个类方法的作用是验证用户、保存用户和发送欢迎邮件。
class UserService {
  public:
    void registerUser(UserData data) {
      // 验证用户
      validateUserData(data);
      // 保存用户
      saveUser(data);
      // 发送邮件
      sendWelcomeEmail(data);
    }
};
```

说明：当需要用"和"来描述类的职责时，表明类承担了过多职责，应该拆分为多个专注的类。

- **新团队成员难以阅读并快速理解类的作用**

  这个类到底是做什么？

```
class OrderManager {
  // 包含太多方法，职责不清晰
  public:
    void createOrder() { /* ... */ } // 创建订单
    void calculateTax() { /* ... */ } // 计算税
    void generateInvoice() { /* ... */ } // 生成发票
    void sendConfirmation() { /* ... */ } // 发送确认邮件
    void updateInventory() { /* ... */ } // 更新库存
    void processPayment() { /* ... */ } // 处理支付
};
```

  说明：类应该有清晰、专注的职责，使新成员能够快速理解其作用。

- **类中的字段只在某些方法中使用**

  是不是强行把多个字段放在同一个类中？

```
class ReportGenerator {
  private:
    EmailService emailService;     // 只在sendReport中使用
    FileService fileService;       // 只在saveReport中使用
    DatabaseService databaseService; // 只在loadData中使用

  public:
    void loadData() { /* 只使用databaseService */ }
    void saveReport() { /* 只使用fileService */ }
    void sendReport() { /* 只使用emailService */ }
};
```

  说明：如果字段只在部分方法中使用，表明类可能承担了多个职责，应该拆分。

- **类中有只操作参数的静态方法**

  把工具类放进了一个有状态的类中。

```
class User {
  private:
    std::string firstName;
    std::string lastName;
    int age;

  // 静态方法只操作参数，与类的状态无关
  public:
    static bool isValidEmail(string email) { /* ... */ }
    static bool isAdult(int age) { /* ... */ }
    static string formatName(string firstName, string lastName) { /* ... */ }
};
```

  说明：只操作参数的静态方法应该移到更合适的工具类中，或者成为相关类的实例方法。

缺陷 #4: 示例

之前：难以测试

```
class UserService {
private:
  DatabaseConnection conn;
  EmailService emailService;
  UserValidator validator;
  Cache cache;

public:
  User createUser(string name, string email) {
    // 验证逻辑
    if (!validator.isValidEmail(email)) {
      throw new ValidationException();
    }
    // 数据库逻辑
    User user = new User(name, email);
    conn.save(user);
    // 通知逻辑
    emailService.sendWelcomeEmail(user);
    return user;
  }
};
```

问题：

- 多个职责：验证、持久化、通知
- 难以隔离测试
- 每个测试需要复杂的设置

之后：可测试的设计

```cpp
// 拆分为专注的类
class UserValidator {
public:
  bool isValidEmail(string email) { /* 只有验证逻辑 */ }
};

class UserRepository {
private:
  DatabaseConnection& conn;
public:
  UserRepository(DatabaseConnection& conn)
    : conn(conn) {}
  void save(User user) { conn.save(user); }
};

class WelcomeEmailSender {
private:
  EmailService& emailService;
public:
  WelcomeEmailSender(EmailService& emailService)
    : emailService(emailService) {}
  void sendWelcomeEmail(User user) {
    emailService.sendWelcomeEmail(user);
  }
};
```

优势：

- 每个类都有单一职责
- 易于单独测试每个类
- 依赖关系清晰

测试的好处

当我们消除这些缺陷后，测试变得容易得多：

- **快速**：测试运行迅速，不依赖外部资源
- **隔离**：测试之间互不影响，可以独立运行
- **具体**：当测试失败时，我们知道确切的问题所在
- **清晰**：测试代码易于理解，表达明确的意图
- **可维护**：实现更改时测试不会中断，除非行为确实改变

总结

1. **构造函数应该只将参数分配给字段**
   - 注入依赖而不是创建它们
   - 保持构造函数简洁
2. **直接请求依赖**
   - 不要深入了解协作者以获取其他对象
   - 遵循得墨忒耳定律
3. **避免全局状态和单例**
   - 使依赖关系明确
   - 使用依赖注入
4. **遵循单一职责原则**
   - 一个类，一个职责
   - 保持类小而专注

> "防止 Bug 最有效的方法是编写可测试的代码。" – Miško Hevery

参考资料

- Miško Hevery的编写可测试代码指南
- Google 测试博客
- Robert C. Martin的《代码整洁之道》
- Kent Beck的《测试驱动开发》

感谢聆听