Modern C++ Camp

现代C++系统研发骨干特训营

李建忠 Boolan

Boolan
高端IT咨询与教育平台

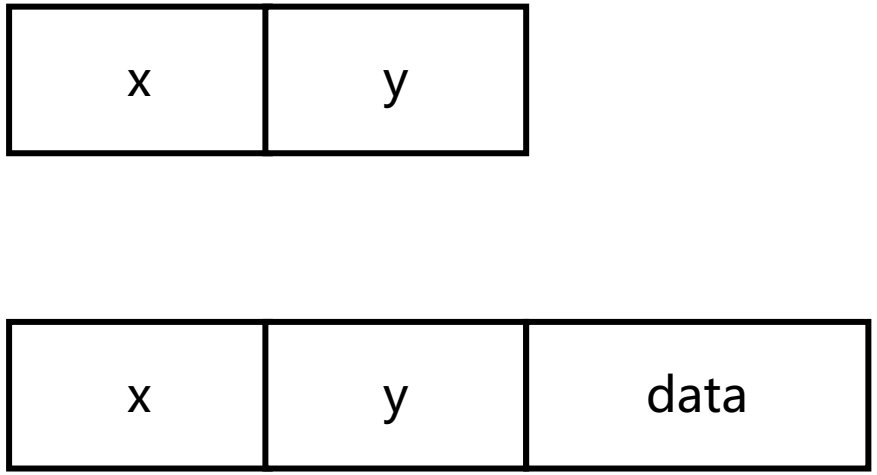
现代C++系统研发骨干特训营

模块二、C++面向对象编程

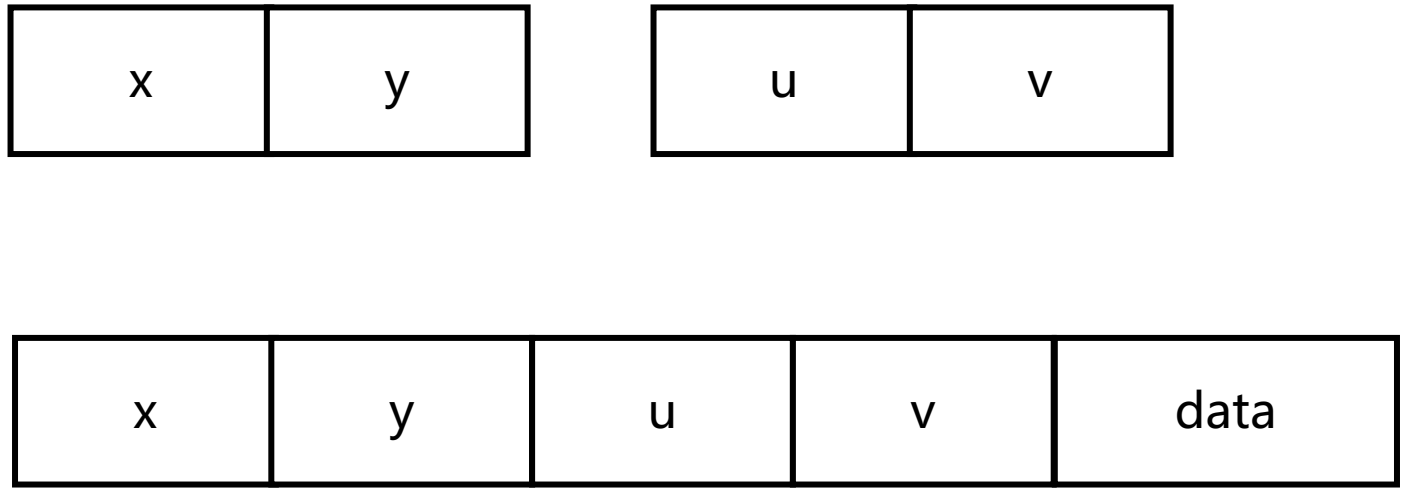
C++ 对象内存模型——继承

- 单继承模型
  - 数据成员、函数成员的继承
  - 构造函数、析构函数的调用链
  - 子父类对象赋值：注意对象切片效应
- 继承 VS. 组合
  - 内存模型对比
  - 组合优于继承
- 多继承模型
  - 多继承内存模型

继承内存模型



单继承模型

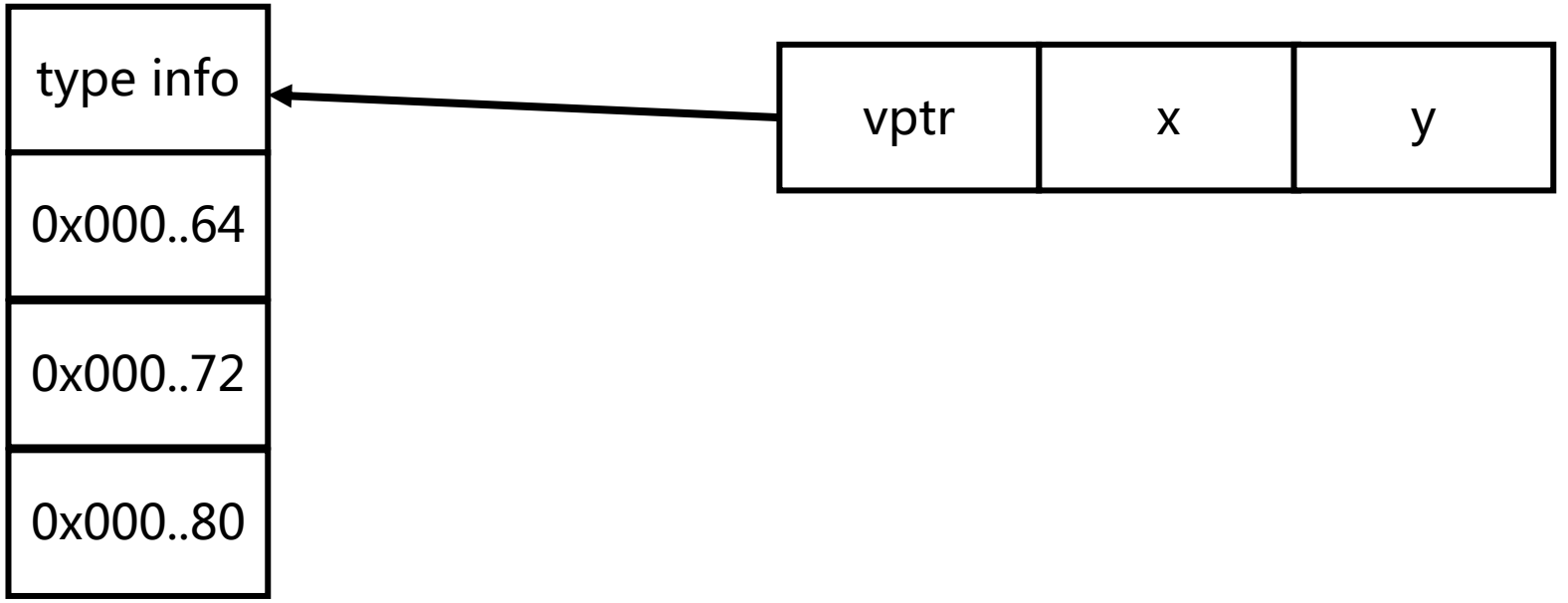


多继承模型

C++ 对象内存模型——多态

- 动态多态的基石——虚函数
  - override与final
- 虚函数的内存模型
  - 虚函数表结构
  - 运行时类型信息（RTTI）
- 虚析构函数：基类析构函数必须为虚，否则会有资源泄漏危险。
- 继承的两层含义
  - 实现继承：复用父类定义的数据成员和函数成员
  - 接口继承：通过父类定义的虚函数来约定行为合同
- 编译时绑定 VS. 运行时绑定

虚函数内存模型



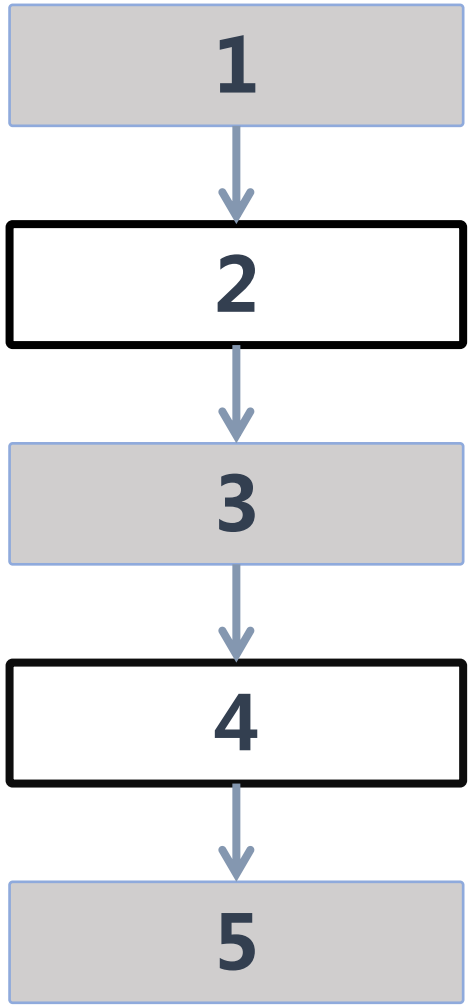
Template Method 模式

动机（Motivation）

- 在软件构建过程中，对于某一项任务，它常常有稳定的整体操作结构，但各个子步骤却有很多改变的需求，或者由于固有的原因（比如框架与应用之间的关系）而无法和任务的整体结构同时实现。

- 如何在确定稳定操作结构的前提下，来灵活应对各个子步骤的变化或者晚期实现需求？

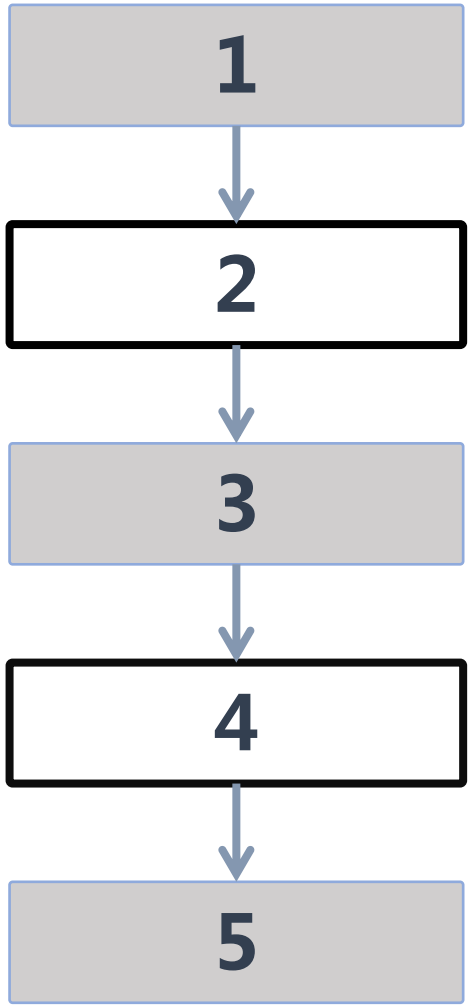
结构化软件设计流程



Library开发人员：
(1) 开发1、3、5 三个步骤

Application开发人员
(1) 开发2、4两个步骤
(2) 程序主流程

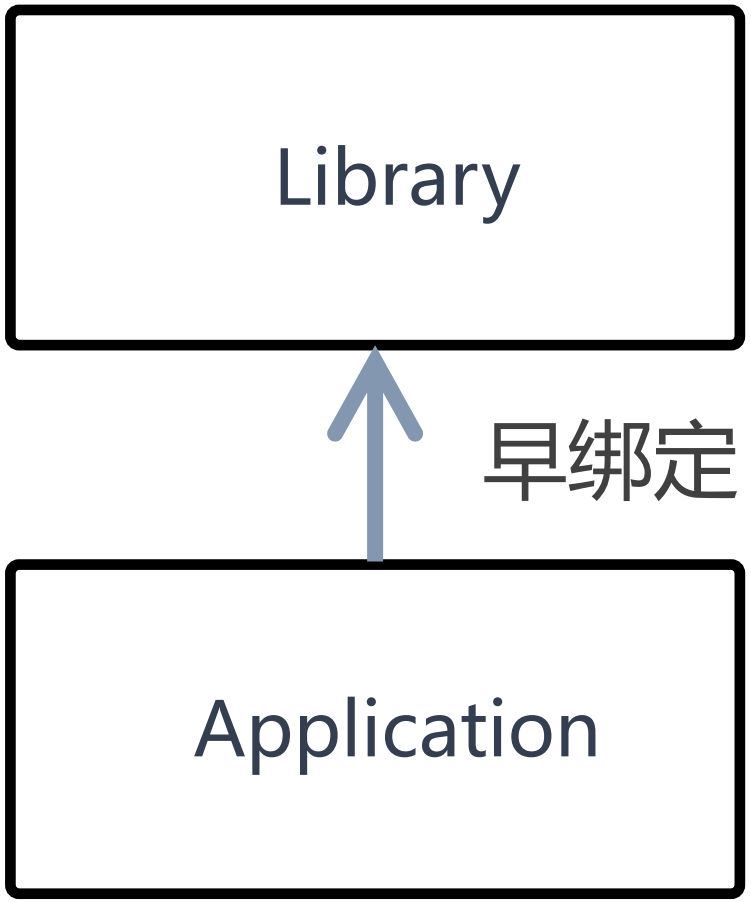
面向对象软件设计流程

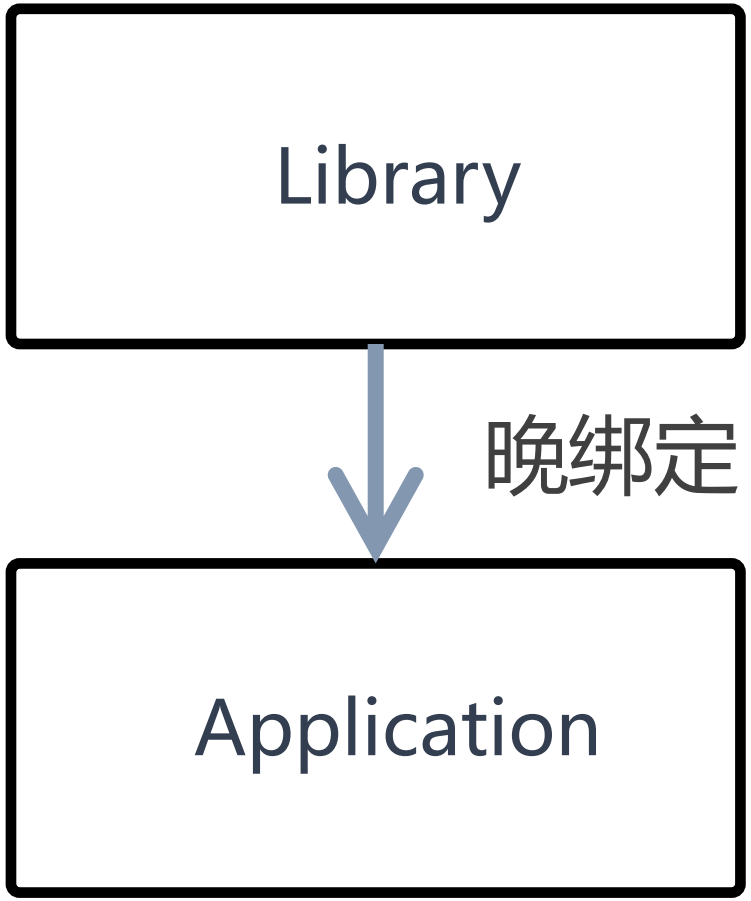


Library开发人员：
(1) 开发1、3、5 三个步骤
(2) 程序主流程

Application开发人员
(1) 开发2、4两个步骤

早绑定与晚绑定



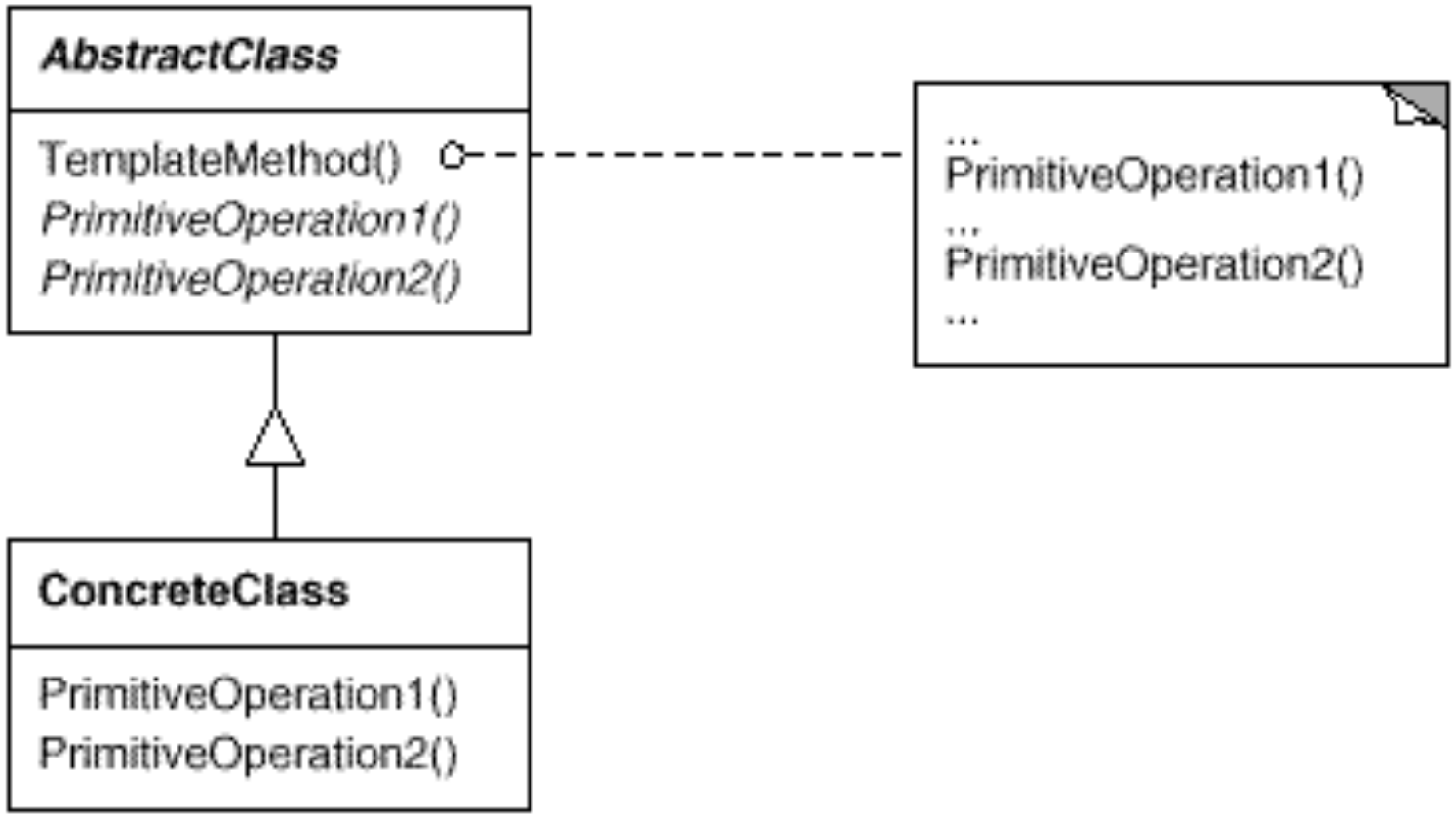


模式定义

定义一个操作中的算法的骨架 (稳定)，而将一些步骤延迟(变化)到子类中。Template Method使得子类可以不改变(复用)一个算法的结构即可重定义(override 重写)该算法的某些特定步骤。

——《设计模式》GoF

结构（Structure）



要点总结

- Template Method模式是一种非常基础性的设计模式，在面向对象系统中有着大量的应用。它用最简洁的机制（虚函数的多态性）为很多应用程序框架提供了灵活的扩展点，是代码复用方面的基本实现结构。

- 除了可以灵活应对子步骤的变化外，“不要调用我，让我来调用你”的反向控制结构是Template Method的典型应用。

- 在具体实现方面，被Template Method调用的虚方法可以具有实现，也可以没有任何实现（抽象方法、纯虚方法），但一般推荐将它们设置为protected方法。

Strategy 策略模式

动机（Motivation）

- 在软件构建过程中，某些对象使用的算法可能多种多样，经常改变，如果将这些算法都编码到对象中，将会使对象变得异常复杂；而且有时候支持不使用的算法也是一个性能负担。

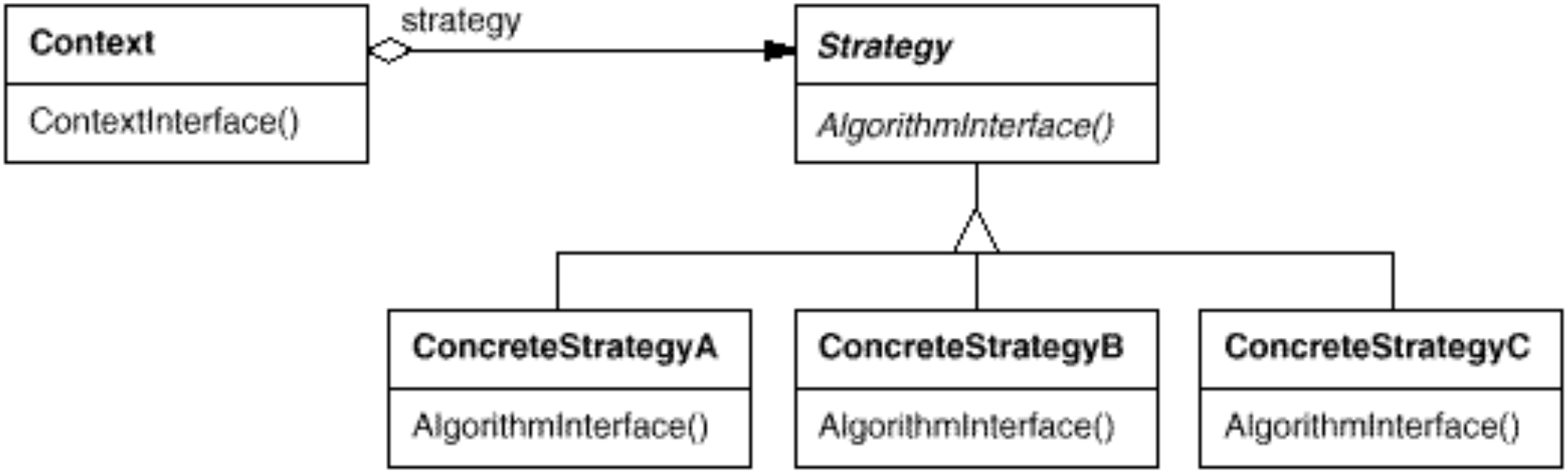
- 如何在运行时根据需要透明地更改对象的算法？将算法与对象本身解耦，从而避免上述问题？

模式定义

定义一系列算法，把它们一个个封装起来，并且使它们可互相替换（变化）。该模式使得算法可独立于使用它的客户程序(稳定)而变化（扩展，子类化）。

——《设计模式》GoF

结构（Structure）



要点总结

- Strategy及其子类为组件提供了一系列可重用的算法，从而可以使得类型在运行时方便地根据需要在各个算法之间进行切换。

- Strategy模式提供了用条件判断语句以外的另一种选择，消除条件判断语句，就是在解耦合。含有许多条件判断语句的代码通常都需要Strategy模式。

- 如果Strategy对象没有实例变量，那么各个上下文可以共享同一个Strategy对象，从而节省对象开销。

软件设计——解决复杂性的艺术

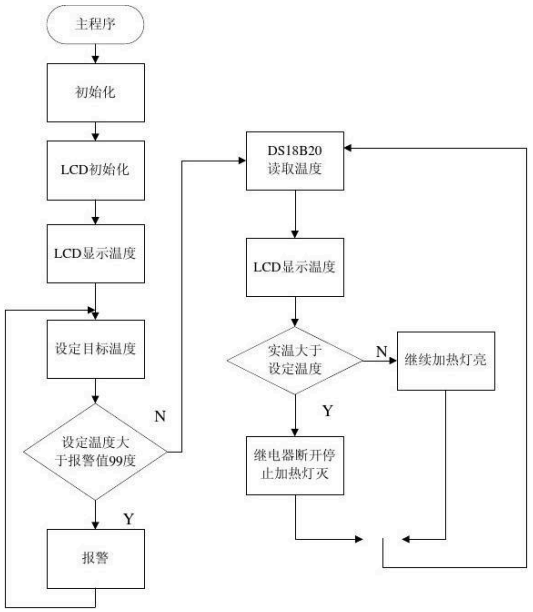
- 分解
  - 人们面对复杂性有一个常见的做法：即分而治之，将大问题分解为多个小问题，将复杂问题分解为多个简单问题。

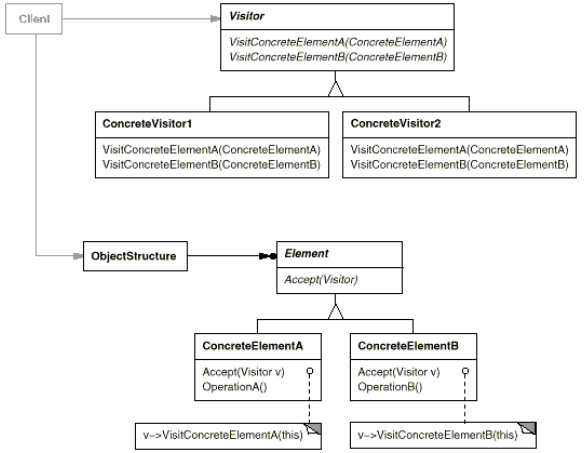
- 抽象
  - 更高层次来讲，人们处理复杂性有一个通用的技术，即抽象。由于不能掌握全部的复杂对象，我们选择忽视它的非本质细节，而去处理泛化和理想化了的对象模型。

分解 V.S. 抽象

- 分解：结构化设计——系统中的每个模块表示某个总体进程中的主要一步。

- 抽象：面向对象设计——根据问题域中的关键抽象来分解系统。





面向对象——拥有责任的抽象

- 理解隔离变化
  - 从宏观层面来看，面向对象的构建方式更能适应软件的变化，能将变化所带来的影响减为最小
- 各司其职
  - 从微观层面来看，面向对象的方式更强调各个类的“责任”
  - 由于需求变化导致的新增类型不应该影响原来类型的实现——是所谓各负其责
- 对象是什么？
  - 从语言实现层面来看，对象封装了代码和数据。
  - 从规格层面讲，对象是一系列可被使用的公共接口。
  - 从概念层面讲，对象是某种拥有责任的抽象。

面向对象设计的精髓——“抽象”

- 向上：深刻把握面向对象机制所带来的抽象意义，理解如何使用这些机制来表达现实世界，掌握什么是“好的面向对象设计”

- 对象通过“抽象”来管理复杂性
  - 数据抽象
  - 单一责任
  - 模块化
  - 分治管理

重新认识“封装”

封装责任，隔离变化

- 使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良的影响，从而实现层次间的松耦合。
- 封装就是建立责任的边界！

重新认识“继承”

优先使用对象组合，而不是类继承

- 继承需要严格满足子类-父类的“IS-A”抽象原则（接口合约）；组合只需要具备“HAS-A”的拥有或者使用关系。
- 类继承通常为“白箱复用”，对象组合通常为“黑箱复用”。
- 继承在某种程度上破坏了封装性，子类父类耦合度高；而对象组合则只要求被组合的对象具有良好定义的接口，耦合度低。

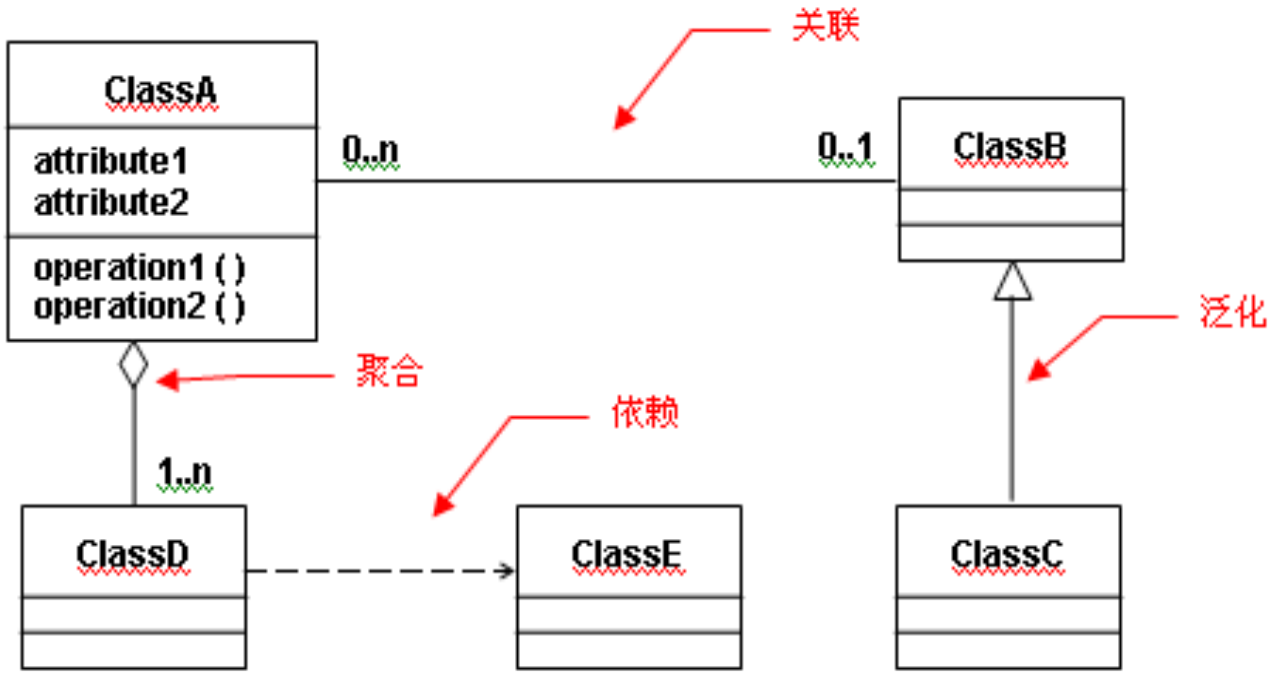
重新认识“多态”

针对接口编程，而不是针对实现编程

- 不将变量类型声明为某个特定的具体类，而是声明为某个接口。
- 客户程序无需获知对象的具体类型，只需要知道对象所具有的接口。
- 减少/规范系统中各部分的依赖关系，从而实现“高内聚、松耦合”的类型设计方案。

审视类的依赖关系

- 1. **泛化**（generalization）抽象事物(父类)与特殊事物(子类)之间的关系，IS-A。
- 2. **依赖**（dependency）又称使用关系,即A使用B来完成某项任务（局部变量，参数）。
- 3. **关联**（association）对象间的实例结构对应关系（实例变量）。
- 4. **聚合**（aggregation）特殊的关联关系。A由B组成，整体与部分的关系(实例变量)。



将设计原则提升为设计经验

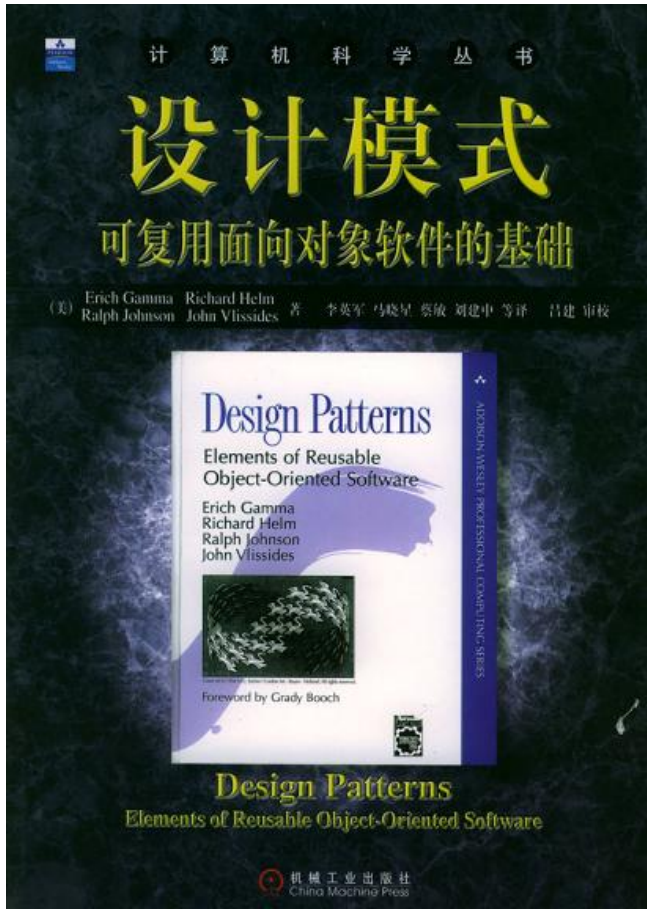
- 1. 设计原则 Design Principles
  - Design Principles 描述与特定编程语言无关的，通用性的，适用于各种粒度的设计原则。
- 2. 设计习语 Design Idioms
  - Design Idioms 描述与特定编程语言相关的特定规范、技巧、惯用法。
- 3. 设计模式 Design Patterns
  - Design Patterns主要描述的是“类与相互通信的对象之间的组织关系，包括它们的角色、职责、协作方式等方面。

软件设计的宗旨

对象/组件/系统内部，对象/组件/系统外部

高内聚，低耦合

GOF 设计模式



- 历史性著作《设计模式：可复用面向对象软件的基础》一书中描述了23种经典面向对象设计模式，创立了模式在软件设计中的地位。

- 由于《设计模式》一书确定了设计模式的地位，通常所说的设计模式隐含地表示“面向对象设计模式”。但这并不意味“设计模式”就等于“面向对象设计模式”。

GOF-23 模式分类

- 从目的来看：
  - 创建型（Creational）模式：将对象的部分创建工作延迟到子类或者其他对象，从而应对需求变化为对象创建时具体类型实现引来的冲击。
  - 结构型（Structural）模式：通过类继承或者对象组合获得更灵活的结构，从而应对需求变化为对象的结构带来的冲击。
  - 行为型（Behavioral）模式：通过类继承或者对象组合来划分类与对象间的职责，从而应对需求变化为多个交互的对象带来的冲击。
- 从范围来看：
  - 类模式处理类与子类的静态关系。
  - 对象模式处理对象间的动态关系。

从封装变化角度对模式分类

Boolan
高端IT咨询与教育平台

- 组件协作：
  - Template Method
  - Strategy
  - Observer / Event
- 对象创建:
  - Factory Method
  - Abstract Factory
  - Prototype
  - Builder
- 单一职责：
  - Decorator
  - Bridge
- 对象性能：
  - Singleton
  - Flyweight
- 接口隔离:
  - Façade
  - Proxy
  - Mediator
  - Adapter
- 状态变化：
  - Memento
  - State
- 数据结构：
  - Composite
  - Iterator
  - Chain of Resposibility
- 行为变化：
  - Command
  - Visitor
- 领域问题：
  - Interpreter

Observer 观察者模式

动机（Motivation）

- 在软件构建过程中，我们需要为某些对象建立一种“通知依赖关系”——一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知。如果这样的依赖关系过于紧密，将使软件不能很好地抵御变化。

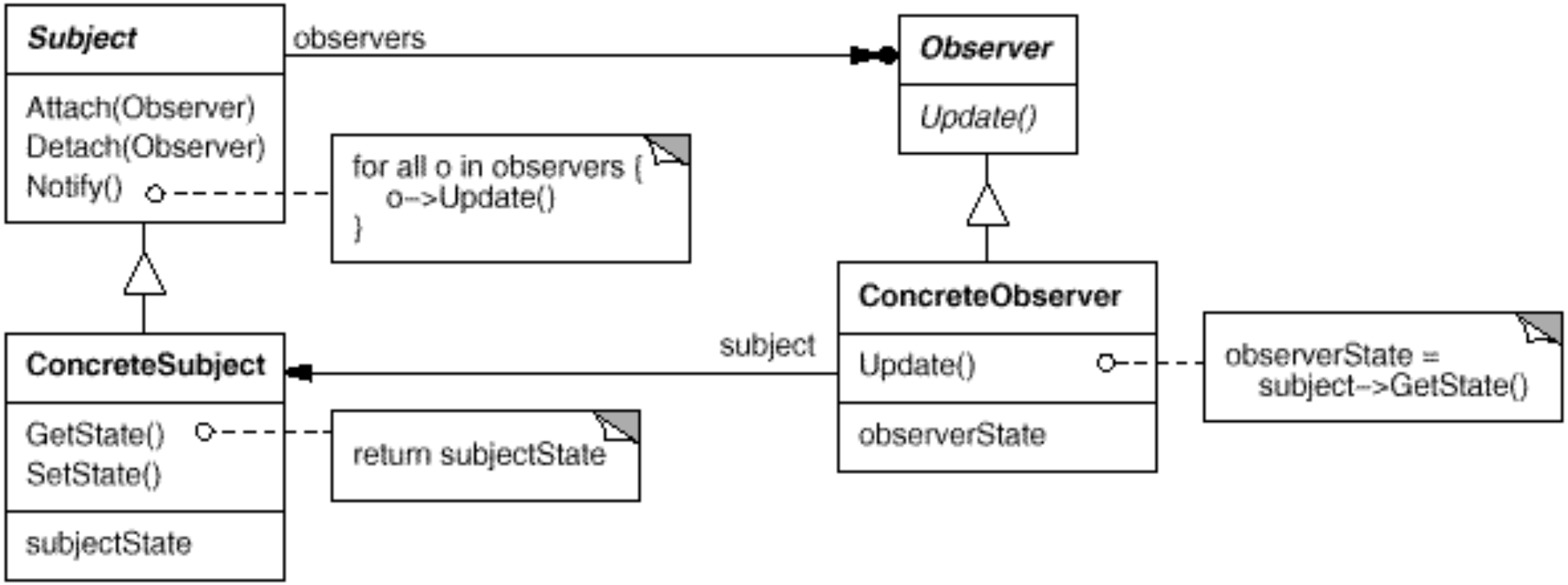
- 使用面向对象技术，可以将这种依赖关系弱化，并形成一种稳定的依赖关系。从而实现软件体系结构的松耦合。

模式定义

定义对象间的一种一对多（变化）的依赖关系，以便当一个对象(Subject)的状态发生改变时，所有依赖于它的对象都得到通知并自动更新。

——《设计模式》GoF

结构（Structure）



要点总结

- 使用面向对象的抽象，Observer模式使得我们可以独立地改变目标与观察者，从而使二者之间的依赖关系达致松耦合。
- 目标发送通知时，无需指定观察者，通知（可以携带通知信息作为参数）会自动传播。
- 观察者自己决定是否需要订阅通知，目标对象对此一无所知。
- Observer模式是基于事件的UI框架中非常常用的设计模式，也是MVC模式的一个重要组成部分。

Pimp设计习语

- Pimpl, Pointer to Implementation，将类数据成员声明为指针，指向某具体实现类 或抽象类。
- **编译依赖**：消除.h头文件对实现类的依赖（只需声明、而无需定义的不完整类型）。因为指针指向不完整类型是合法的。虽然.cpp实现文件内仍然需要实现类的定义（延迟依赖）。
- **延迟分发**：如果使用指针指向抽象类，则由于虚函数的多态分发，将函数调用延迟到运行时，将具有更好的弹性。
- **资源管理**：在构造器中初始化实现类(new)，在析构器中释放实现类(delete)。裸指针可考虑替换为智能指针（unique_ptr），需要特别谨慎处理其中的编译依赖关系。

性能指南——继承与多态

- 优先使用具体类型而不是类继承层次
- 如非必要，不随意将函数定义为虚函数（代价：对象大小、函数间接绑定、无法内联）
- 避免使用dynamic_cast和typeid等RTTI机制，使用虚函数或编译时模板替换
- 考虑使用基于模板的policy设计替代虚函数设计
- 多态类应当避免或者删除公开的移动/复制操作，以防止传值带来的对象切片效应
- 利用空基类优化：如果是空类，继承时对象size为零；但组合时，至少占用一个byte，加上对齐效应，通常更大
- 多态类的深拷贝；优先采用虚函数 clone 来替代公开复制构造/赋值
- 禁止把指向派生类对象数组的指针赋值给指向基类的指针

异常机制

- 异常是程序库(library)和 应用(application)针对违例的通知机制
  - throw 抛出异常， try 包含可能抛异常的语句块，catch 语句捕获异常
  - catch 异常对象时，使用引用避免异常对象拷贝。
  - 无论异常是否出现，函数调用栈上的对象会确保释放（析构函数）
- C++11废弃了异常规约（列举函数可能抛出的异常），而采用noexcept表示是否抛出异常
  - 只有确定不抛异常的函数才标记noexcept
  - 如果函数标记为noexcept又抛出异常，则程序立即终止。
- 什么情况使用异常机制
  - 错误罕见无预期、直接调用者无法处理（必须层层交给最终调用者）
  - 错误码没有合适路径返回、或者返回错误码低效、混乱
- 什么情况使用返回错误码？
  - 错误是常规预期会出现的、并且调用者能够合理处理（例如文件打开错误）

性能指南——异常处理

- 使用RAII，尽量避免对try/catch 的显式使用
- 将不抛异常的函数声明为noexcept，是很多代码优化的前提
  - 编译器优化可以省掉展开栈的相关成本
  - 只有移动操作不抛出异常，才使用移动；否则使用拷贝替代
  - 针对异常中立的函数（自己不抛，但调用的函数可能会抛），不要试图捕获所有异常，来实现noexcept
  - 默认情况下，内存释放函数、析构函数为noexcept
- 只要启用异常，代码一般会有5%～15%的膨胀
- 异常不抛出时没有额外性能负担，只有抛出时才有（比错误码大）
- 只有特殊的硬实时系统对时间开销（异常抛出时），或者嵌入式系统对空间开销（代码膨胀）极度敏感时，才关闭异常（决策前测量）