目 录

[1 单例模式（Singleton Pattern） 1](#_Toc10992792)

[1.1 定义 1](#_Toc10992793)

[1.2 使用场景 1](#_Toc10992794)

[2 工厂方法（Factory method pattern） 2](#_Toc10992795)

[2.1 定义 2](#_Toc10992796)

[2.2 使用场景 3](#_Toc10992797)

[3 抽象工厂模式（Abstract factory pattern） 3](#_Toc10992798)

[3.1 定义 3](#_Toc10992799)

[3.2 使用场景 4](#_Toc10992800)

[4 模板方法模式（Template Pattern） 4](#_Toc10992801)

[4.1 定义 4](#_Toc10992802)

[4.2 使用场景 5](#_Toc10992803)

[5 建造者模式（Builder Pattern） 5](#_Toc10992804)

[5.1 定义 5](#_Toc10992805)

[5.2 使用场景 6](#_Toc10992806)

[6 代理模式（Proxy Pattern） 6](#_Toc10992807)

[6.1 定义 6](#_Toc10992808)

[6.2 使用场景 7](#_Toc10992809)

[7 原型模式（Prototype Pattern） 8](#_Toc10992810)

[7.1 定义 8](#_Toc10992811)

[7.2 使用场景 8](#_Toc10992812)

[8 中介者模式（Mediator Pattern） 9](#_Toc10992813)

[8.1 定义 9](#_Toc10992814)

[8.2 使用场景 9](#_Toc10992815)

[9 命令模式（Command Pattern） 10](#_Toc10992816)

[9.1 定义 10](#_Toc10992817)

[9.2 使用场景 10](#_Toc10992818)

[10 责任链模式（Chain of Responsibility Pattern） 10](#_Toc10992819)

[10.1 定义 10](#_Toc10992820)

[10.2 使用场景 11](#_Toc10992821)

[11 装饰模式（Decorator Pattern） 11](#_Toc10992822)

[11.1 定义 11](#_Toc10992823)

[11.2 使用场景 12](#_Toc10992824)

[12 策略模式（Strategy Pattern） 12](#_Toc10992825)

[12.1 定义 12](#_Toc10992826)

[12.2 使用场景 13](#_Toc10992827)

[13 适配器模式（Adapter Pattern） 13](#_Toc10992828)

[13.1 定义 13](#_Toc10992829)

[13.2 使用场景 13](#_Toc10992830)

[14 迭代器模式（Iterator Pattern） 14](#_Toc10992831)

[14.1 定义 14](#_Toc10992832)

[14.2 使用场景 14](#_Toc10992833)

[15 组合模式（Composite Pattern） 14](#_Toc10992834)

[15.1 定义 14](#_Toc10992835)

[15.2 使用场景 15](#_Toc10992836)

[16 观察者模式（Observer Pattern） 15](#_Toc10992837)

[16.1 定义 15](#_Toc10992838)

[16.2 使用场景 16](#_Toc10992839)

[17 门面模式（Facade Pattern） 16](#_Toc10992840)

[17.1 定义 16](#_Toc10992841)

[17.2 使用场景 17](#_Toc10992842)

[18 备忘录模式（Memento Pattern） 17](#_Toc10992843)

[18.1 定义 17](#_Toc10992844)

[18.2 使用场景 18](#_Toc10992845)

[19 访问者模式（Visitor Pattern） 18](#_Toc10992846)

[19.1 定义 18](#_Toc10992847)

[19.2 使用场景 19](#_Toc10992848)

[20 状态模式（Sate Pattern） 19](#_Toc10992849)

[20.1 定义 19](#_Toc10992850)

[20.2 使用场景 20](#_Toc10992851)

[21 解释器模式（Interpreter Pattern） 20](#_Toc10992852)

[21.1 定义 20](#_Toc10992853)

[21.2 使用场景 21](#_Toc10992854)

[22 享元模式（Flyweight Pattern） 21](#_Toc10992855)

[22.1 定义 21](#_Toc10992856)

[22.2 使用场景 22](#_Toc10992857)

[23 桥梁模式（Bridge Pattern） 22](#_Toc10992858)

[23.1 定义 22](#_Toc10992859)

[23.2 使用场景 23](#_Toc10992860)

[参 考 文 献 24](#_Toc10992861)

# 1 单例模式（Singleton Pattern）

## 1.1 定义

单例模式是一种设计模式，它限制类实例化其多个对象。它是一种定义课程的方式。类的定义方式是在完成程序或项目的执行时只创建一个类的实例。它用于在整个执行过程中只需要一个类的单个实例来控制操作的情况。单例类在任何情况下都不应该具有多个实例且不惜任何代价。 Singleton类用于日志记录，驱动程序对象，缓存和线程池，数据库连接。

单例类的实现应具有以下属性：

它应该只有一个实例：这是通过在类中提供类的实例来完成的。应该阻止外部类或子类来创建实例。这是通过在java中使构造函数私有来完成的，这样任何类都不能访问构造函数，因此无法实例化它。

实例应该是全局可访问的：单例类的实例应该是全局可访问的，以便每个类都可以使用它。在Java中，它是通过使实例的访问说明符为public来完成的。

UML图如图1.1.1所示：



图1.1.1 单例模式

## 1.2 使用场景

单例模式有很多应用，如缓存内存，数据库连接，驱动程序，日志记录。例如：

硬件接口访问：单例的使用取决于要求。 Singleton类也用于防止类的并发访问。在需要外部硬件资源使用限制的情况下，可以使用实际的单例。硬件打印机，其中打印假脱机程序可以成为单例，以避免多个并发访问并创建死锁。

**记录器**：Singleton类用于日志文件生成。日志文件由记录器类对象创建。假设一个应用程序，其中日志记录实用程序必须根据从用户收到的消息生成一个日志文件。如果有多个客户端应用程序使用此日志记录实用程序类，则可能会创建此类的多个实例，并且在并发访问同一记录器文件期间可能会导致问题。我们可以将logger实用程序类用作单例，并提供全局参考点，以便每个用户都可以使用此实用程序，并且没有2个用户同时访问它。

**配置文件**：这是Singleton模式的另一个潜在候选者，因为它具有性能优势，因为它可以防止多个用户重复访问和读取配置文件或属性文件。它创建了一个配置文件的单个实例，可以同时访问多个调用，因为它将提供加载到内存中对象的静态配置数据。应用程序仅首次从配置文件中读取，然后从第二次调用开始，客户端应用程序从内存中对象读取数据。

缓存：我们可以将缓存用作单个对象，因为它可以具有全局参考点，并且对于将来对缓存对象的所有调用，客户端应用程序将使用内存中对象。

# 2 工厂方法（Factory method pattern）

## 2.1 定义

工厂方法是创建设计模式，即与对象创建有关。 在Factory模式中，我们创建对象而不将创建逻辑暴露给客户端，并且客户端使用相同的公共接口来创建新类型的对象。

我们的想法是使用静态成员函数（静态工厂方法）来创建和返回实例，从而隐藏用户的类模块的细节。

工厂模式是创建对象的核心设计原则之一，允许客户端以某种方式创建库的对象（如下所述），使其不与库的类层次结构紧密耦合。

工厂方法创建对象，然后子类可以覆盖。指定将要创建的派生产品类型。工厂方法模式依赖于继承，作为对象创建被委托给实现的子类，工厂方法来创建对象。

UML图如图2.1.1所示：



图2.1.1 工厂方法

## 2.2 使用场景

比如说，在“绘图”系统中，根据用户的输入，可以绘制不同的图片，如方形，矩形，圆形。 在这里，我们可以使用工厂方法根据用户的输入创建实例。 要添加新类型的形状，无需更改客户端的代码。

另一个例子：在旅游网站，我们可以预订火车票以及公交车票和机票。 在这种情况下，用户可以将他的旅行类型称为“公共汽车”，“火车”或“航班”。

这里我们有一个带有静态成员函数'GetObject'的抽象类'AnyTravel'，它取决于用户的旅行类型，将创建并返回'BusTravel'或'TrainTravel'的对象。 “BusTravel”或“TrainTravel”具有常用功能，如乘客姓名，Origin，destinationparameters。

# 3 抽象工厂模式（Abstract factory pattern）

## 3.1 定义

抽象工厂设计模式是Creational模式之一。 抽象工厂模式几乎与工厂模式类似，被视为工厂模式的另一层抽象。抽象工厂模式实现为我们提供了一个框架，允许我们创建遵循一般模式的对象。 因此，在运行时，抽象工厂与任何可以创建所需类型的对象的工厂相结合。

* AbstractFactory：声明用于创建抽象产品对象的操作的接口。
* ConcreteFactory：实现AbstractFactory中声明的操作以创建具体的产品对象。
* Product：定义由相应的具体工厂创建的产品对象，并实现AbstractProduct接口。
* Client：仅使用AbstractFactory和AbstractProduct类声明的接口。

UML图如图3.1.1所示：



图3.2.1 抽象工厂模式

## 3.2 使用场景

让我们举一个例子，假设我们想建立一个全球汽车工厂。 如果是工厂设计模式，那么它适用于单个位置。 但对于这种模式，我们需要多个位置和一些关键的设计变更。

我们需要在每个地点建立汽车工厂，如IndiaCarFactory，USACarFactory和DefaultCarFactory。 现在，我们的应用程序应该足够智能，以确定其使用位置，因此我们应该能够使用适当的汽车工厂，甚至不知道将在内部使用哪个汽车工厂实施。 这也使我们免于某人为特定地点打电话给错误的工厂。

在这里，我们需要另一层抽象，它将识别位置并在内部使用正确的汽车工厂实现，甚至不向用户提供单一提示。 这正是问题抽象工厂模式解决的问题。

# 4 模板方法模式（Template Pattern）

## 4.1 定义

模板方法设计模式是将算法定义为操作的框架，并将细节留给子类实现。 父类保留算法的整体结构和顺序。

模板意味着预设格式，如HTML模板，具有固定的预设格式。类似于模板方法模式，我们有一个预设的结构方法，称为模板方法，由步骤组成。这个步骤可以是抽象方法，将由其子类实现。

这种行为设计模式是最容易理解和实现的模式之一。 这种设计模式在框架开发中广泛使用。 这有助于避免代码重复。

UML图如图4.1.1所示：



图4.1.1 模板方法模式

## 4.2 使用场景

模板方法用于framework中，每个框架都实现了域架构的不变部分，为自定义选项留下了“占位符”。

使用模板方法的原因如下：

* 让子类实现不同的行为（通过方法覆盖）
* 避免在代码中重复，一般工作流结构在抽象类的算法中实现一次，并且在子类中实现必要的变体。
* 控制允许在哪些点进行子类化。 与简单的多态覆盖相反，其中基本方法将被完全重写，允许彻底改变工作流，只允许改变工作流的特定细节。

# 5 建造者模式（Builder Pattern）

## 5.1 定义

构建器模式旨在“将复杂对象的构造与其表示分离，以便相同的构造过程可以创建不同的表示。”它用于逐步构造复杂对象，最后一步将返回对象。 构造对象的过程应该是通用的，以便它可以用于创建相同obj的不同表示。

* Product - product类定义构建器模式生成的复杂对象的类型。
* 构建器 - 此抽象基类定义了为正确创建产品必须采取的所有步骤。每个步骤通常都是抽象的，因为构建器的实际功能是在具体的子类中执行的。 GetProduct方法用于返回最终产品。构建器类通常用简单的接口替换。
* ConcreteBuilder可能有许多继承自Builder的具体构建器类。这些类包含创建特定复杂产品的功能。
* Director-director类控制生成最终产品对象的算法。实例化一个director对象，并调用其Construct方法。该方法包括用于捕获将用于生成产品的特定具体构建器对象的参数。然后，Director以正确的顺序调用具体构建器的方法以生成产品对象。完成此过程后，构建器对象的GetProduct方法可用于返回产品。

UML图如图5.1.1所示：



图5.1.1 建造者模式

## 5.2 使用场景

考虑建造一个家。 Home是作为构造过程的输出返回的最终最终产品（对象）。 它将有许多步骤，如地下室建筑，墙体施工等屋顶结构。 最后返回整个主对象。 在这里使用相同的过程，您可以建立具有不同属性的房屋。

# 6 代理模式（Proxy Pattern）

## 6.1 定义

代理意味着'代替'，代表'或'代替'或'代表'代表字面含义，直接解释代理设计模式。

代理也称为代理，句柄和包装。 它们在结构上与适配器和装饰器密切相关，但不是目的。

当我们需要创建一个包装器来覆盖客户端主对象的复杂性时，就会使用代理模式。

一个真实世界的例子可以是支票或信用卡是我们银行账户中的内容的代理。 它可以用来代替现金，并提供在需要时获取现金的方法。 而这正是代理模式所做的“控制和管理对他们正在保护的对象的访问”。

UML图如图6.1.1所示：



图6.1.1 代理模式

## 6.2 使用场景

代理模式可以在虚拟代理场景中使用—当有多个数据库调用来提取大尺寸图像。由于这是一个昂贵的操作，所以在这里我们可以使用代理模式，它将创建多个代理并指向大尺寸内存消耗对象以进行进一步处理。仅当客户端首先请求/访问对象时才创建真实对象，之后我们可以引用代理来重用该对象。这避免了对象的重复，从而节省了内存。

代理模式可用于保护代理方案---它充当授权层，以验证实际用户是否具有访问适当内容的权限。例如，代理服务器在办公室中提供对互联网访问的限制。只允许有效的网站和内容，其余的将被阻止。

它可用于远程代理方案---可以考虑远程代理RPC调用中的存根。远程代理提供对象的本地表示，该对象存在于不同的地址位置。另一个示例可以是为诸如web服务或REST资源之类的远程资源提供接口。

代理模式可用于智能代理方案—智能代理通过在访问对象时插入特定操作来提供额外的安全层。例如，要在访问真实对象之前检查它是否被锁定，以便其他对象不能更改它。

# 7 原型模式（Prototype Pattern）

## 7.1 定义

原型允许我们隐藏从客户端创建新实例的复杂性。 这个概念是复制现有对象而不是从头开始创建新实例，这可能包括昂贵的操作。 现有对象充当原型并包含对象的状态。 只有在需要时，新复制的对象才可以更改相同的属性。 这种方法节省了昂贵的资源和时间，特别是在对象创建过程繁重时。

原型模式是一种创造性的设计模式。 当对象创建耗时且操作成本高时，需要原型模式，因此我们使用现有对象本身创建对象。 从现有对象创建对象的最佳方法之一是clone（）方法。 克隆是实现原型模式的最简单方法。 但是，您可以根据业务模型决定如何复制现有对象。

原型设计参与者：

* prototype：这是实际对象的原型。
* Prototype注册表：这用作注册表服务，可以使用简单的字符串参数访问所有原型。
* client：客户端将负责使用注册表服务来访问原型实例。

UML图如图7.1.1所示：



图7.1.1 原型模式

## 7.2 使用场景

* 通过动态加载或避免构建与产品或类的层次结构相似的工厂的类层次结构。
* 当一个类的实例只能有几个不同的状态组合之一时。 安装相应数量的原型并克隆它们可能更方便，而不是每次都使用适当的状态手动实例化类。

# 8 中介者模式（Mediator Pattern）

## 8.1 定义

中介设计模式是重要且广泛使用的行为设计模式之一。 介体通过在介于两者之间引入一个层来实现对象的解耦，以便通过层发生对象之间的交互。 如果对象直接相互交互，则系统组件彼此紧密耦合，这使得可维护性成本更高并且不难扩展。 中介模式侧重于在对象之间提供中介以进行通信，并有助于实现对象之间的失耦。

空中交通管制员是调解员模式的一个很好的例子，其中机场控制室作为不同航班之间通信的调解员。 Mediator作为对象之间的路由器，它可以拥有自己的逻辑来提供通信方式。中介模式的设计组件如下：

* mediator：它定义了同事对象之间的通信接口。
* ConcreteMediator：它实现了中介接口并协调同事对象之间的通信。
* colleague：它定义了与其他同事沟通的界面。
* ConcreteColleague：它实现同事界面并通过其中介与其他同事进行通信。

UML图如图8.1.1所示：



图8.1.1 中介者模式

## 8.2 使用场景

它通常用于基于消息的系统，同样是聊天应用程序。当这组对象以复杂但明确定义的方式进行通信时。

# 9 命令模式（Command Pattern）

## 9.1 定义

命令模式将请求封装为对象，从而允许我们使用不同的请求，队列或日志请求参数化其他对象，并支持可撤销操作。命令模式表示“将请求封装在对象下作为命令并将其传递给调用者对象.Interoker对象查找可以处理此命令的相应对象，并将该命令传递给相应的对象，该对象执行命令。

UML图如图9.1.1所示：



图9.1.1 命令模式

## 9.2 使用场景

* 当需要根据动作执行参数化对象时。
* 当需要在不同时间创建和执行请求时。
* 当需要支持回滚，日志记录或事务功能时。

# 10 责任链模式（Chain of Responsibility Pattern）

## 10.1 定义

责任链模式用于在软件设计中实现松散耦合，其中来自客户端的请求被传递到对象链以处理它们。 之后，链中的对象将自行决定谁将处理请求以及是否需要将请求发送到链中的下一个对象。责任链模式表示只是“通过为多个对象提供处理请求的机会，避免将请求的发送者与其接收者耦合”。 例如，ATM在货币提供过程中使用责任链设计模式。

UML图如图10.1.1所示：



图10.1.1 责任链模式

## 10.2 使用场景

* 当想要解耦请求的发送方和接收方时。
* 在运行时确定的多个对象是处理请求的候选对象。
* 当不想在代码中明确指定处理程序时。
* 如果要向多个对象之一发出请求而未明确指定接收方。

# 11 装饰模式（Decorator Pattern）

## 11.1 定义

装饰器模式动态地附加对象的附加职责。 装饰器为子类化提供了灵活的替代扩展功能。Decorator Pattern表示只是“动态地将附加责任附加到对象上”。他使用组合而不是继承来在运行时扩展对象的功能。

UML图如图11.1.1所示：



图11.1.1 装饰模式

## 11.2 使用场景

* 当想要透明地动态地向对象添加职责而不影响其他对象时。
* 当想要为将来可能要更改的对象添加职责时。
* 通过子类扩展功能已不再实用。

# 12 策略模式（Strategy Pattern）

## 12.1 定义

策略模式（也称为策略模式）是一种软件设计模式，可以在运行时选择算法的行为。 策略模式定义了一系列算法，封装了每个算法，以及使算法在该族中可互换。

UML图如图12.1.1所示：



图12.1.1 策略模式

## 12.2 使用场景

* 当多个类的行为不同时，如 Servlet API。
* 当需要不同的算法变体时使用策略模式。

# 13 适配器模式（Adapter Pattern）

## 13.1 定义

适配器帮助两个不兼容的接口一起工作。这是适配器的实际定义。接口可能不兼容，但是内部功能应该满足需要。适配器设计模式通过将一个类的接口转换为客户端期望的接口，允许其他不兼容的类一起工作。

UML如图13.1.1所示：



图13.1.1 适配器模式

## 13.2 使用场景

适配器模式适用于解决以下问题:

* 希望用另一个类替换一个类，而接口不匹配。
* 希望创建一个可以与其他类交互的类，而不需要在设计时知道它们的接口。

例如：两个设备之间数据传输时遵循不同的数据传输协议，需要在其中一个设备上实现适配器模式，将同一意义，不同名称数据进行转换，规化成所需格式。

# 14 迭代器模式（Iterator Pattern）

## 14.1 定义

在面向对象编程中，迭代器用于遍历容器并访问容器的元素。迭代器模式将算法与容器解耦。但是，在某些情况下，算法必须是特定于容器的，因此不能解耦，例如，随机访问只能作用于向量型数据结构，不能作用于链表。

UML图如14.1.1所示：



图14.1.1 迭代器模式

## 14.2 使用场景

可以用于底层数据结构，对外提供透明的接口服务。例如，在C++的STL中，常见的数据结构，list、map、vector、stack、queue等，提供了操作数据结构的函数，对于向量型数据，我们可以通过随机访问获取元素，但是对于链表却不能，迭代器可以帮助我们比较方便地访问链式数据结构。

# 15 组合模式（Composite Pattern）

## 15.1 定义

组合模式允许编程者使用组合树结构来表示整个部分层次结构，以便客户端能够统一地处理单个对象和对象的组合。我们通常构建树结构，其中一些节点是其他节点的容器，而其他节点是“叶子”。组合模式不需要创建单独的客户机代码来管理每种类型的节点，而是让客户机使用相同的代码进行工作。

UML图如图15.1.1所示：



图15.1.1 组合模式

## 15.2 使用场景

在GUI编程中，各种窗体的组合可以用到组合模式，主窗体和子窗体之间不存在泛化关系时，可以利用组合模式设计。

# 16 观察者模式（Observer Pattern）

## 16.1 定义

提供一种方法定义对象之间一对多的依赖关系，当一个对象的状态改变时，所有的依赖于这个对象的对象都会被通知并且自动更新。这个对象包含数据独立于显示这个数据的对象，观察者对象观测数据的更改。

UML图如图16.1.1所示：



图16.1.1 观察者模式

## 16.2 使用场景

多个客户端同时接收来自用一个数据源的数据并进行展示，多个客户端作为多个观察者，实现观察者接口。在多人同时游戏时，每个人的客户端作为观察者，感知并更新当前游戏局面。

# 17 门面模式（Facade Pattern）

## 17.1 定义

一个系统往往包含诸多小的功能，在运行时，实现一个大功能需要调用一系列小的子功能，调用这些子功能的方式对于用户来讲有时候显得很晦涩，引入一个代理—“门面”，用户只需要与门面打交道，屏蔽了复杂性，也在一定程度上保证了系统的安全。

UML图如图17.1.1所示：



图17.1.1 门面模式

## 17.2 使用场景

在使用word打印功能时，对于用户只需要简单地点击“打印”按钮即可，操作系统内部会产生中断，调用中断处理程序，处理打印任务。

# 18 备忘录模式（Memento Pattern）

## 18.1 定义

保存一个对象当前的状态，不能更改这个状态或者破坏该状态的封闭性。在下次使用这个对象时，可以恢复到现有状态。

UML类图如图18.1.1所示：



18.1.1 备忘录模式

## 18.2 使用场景

在游戏平台中，对于用户的当前游戏记录的保存是基于备忘录的，如果用户数量较多，作为游戏运营公司，可以考虑使用数据库，但是一定要保证存储游戏记录的完整性、一致性、原子性、持久性。

# 19 访问者模式（Visitor Pattern）

## 19.1 定义

将算法和对象结构相分离，这种分离可以保证不改变类的结构的前提下，添加新的操作到现有的类上，遵循了开放封闭原则。

UML图如图19.1.1所示：



图19.1.1 访问者模式

## 19.2 使用场景

对于一个关系型数据库，有不同的需求，使用这个数据库中的不同的列，我们可以定义不同的函数访问不同的列，无需改变这个数据库。也可以在代码中封装DML语句，利用数据库查询的高效性的优势。

# 20 状态模式（Sate Pattern）

## 20.1 定义

一个类的对象在处于不同状态时，拥有不同的行为。

UML类图如图20.1.1所示：



图20.1.1 状态模式

## 20.2 使用场景

例如描述一个生产线的生产过程时，我们可以定义在正常、负载不足、超负荷运行时的生产场景。在定义网络协议时，我们可以定义在拥塞情形时的各个网络组件采取的发包策略。

# 21 解释器模式（Interpreter Pattern）

## 21.1 定义

定义一种语言模式，对于输入进行相应的解释，按照字符串的内容、输入格式等匹配，执行不同的函数。

UML类图如图21.1.1所示：



图21.1.1 解释器模式

## 21.2 使用场景

对于可编程计算器，我们可以构造一个有限自动机，利用有限自动机得出LALR预测分析表，构造移进-规约分析程序，实现可编程计算器。

# 22 享元模式（Flyweight Pattern）

## 22.1 定义

当用户有返回一个对象的新需求时，如果当前内存中有这个对象，则无需开辟新空间定义新的对象，直接返回这个对象即可，但要注意并发访问时，可能会引起数据不一致的问题。

UML类图如图22.1.1所示：



图22.1.1 享元模式

## 22.2 使用场景

在文本编辑软件中，如果当前文档中已经有相应的字，则直接应用这个字显示即可，无需重新开辟空间存储这个字，当然，如果重新存储带来的空间消耗小于引用，则可以重新存。

# 23 桥梁模式（Bridge Pattern）

## 23.1 定义

抽取多个类的本质特征，找到这些特征的交集，形成接口，包装为一个包含诸多接口的类，成为抽象类，具体类通过实现这些接口实现达到功能定义和实现的解耦。在使用这些功能时，可以让桥梁两端的实体独立地扩展功能。

UML图如图23.1.1所示：



图23.1.1 桥梁模式

## 23.2 使用场景

在Java访问数据库的工具JDBC驱动器中，JDBC对程序员提供接口（API），通过调用这些接口API程序员及时不是很深入地了解数据库也可以通过JDBC访问并操作数据库。既屏蔽了操作数据库的复杂性，也避免了因为不了解数据库产生误操作，导致严重后果。

JDBC驱动器可以解决同一个应用程序基于不同数据库可能会出现的兼容性问题，如果有新的数据库被开发出来，数据库厂商可以开发对应的JDBC驱动器，提供给Java开发者。具有良好的扩展性。

# 参 考 文 献

[1] 刘海. *软件项目管理实用教程*. 北京: 人民邮电出版社, 2015.

[2] "Project Management Body of Knowledge." Wikipedia, The Free Encyclopedia (English).