设计模式

# 简易说明

1. 简单工厂模式

|  |
| --- |
|  |

1. 策略模式

|  |
| --- |
| 策略模式解析：  \* 是一种定义一系列算法的方法，从概念上看，所有这些算法完成的都是相同的工作，只是实现不同，他可以以相同方式调用所有的算法，减少了各种算法类与  \* 使用算法类之间的耦合。  \* 策略模式的strategy类层次（就是那个父类）为context定义了一系列可充用的算法或者行为，继承有助于析取出这些算法中的公共功能  \* 策略模式的优点：简化了单元测试，因为每个算法都有自己的类，可以通过自己的接口进行单独测试  \* 策略模式可以用来封装他几乎任何类型的规则，只要在分析过程中听到需要在不同时间应用不同的业务规则，就可以考虑使用策略模式 |

1. 原则

|  |
| --- |
| **单一职责原则**：就一个类而言，应该仅有一个引起他变化的原因  如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力，  这种耦合会导致脆弱的设计，当变化发生时，设计会遭受到意想不到的破坏。  **开放-封闭原则**：软件实体（类、模块、函数等等）应该可以扩展，但是不可修改  对于扩展是开放的，对于更改是封闭的  开闭原则是面向对象设计的核心所在，遵循这个原则可以带来面向对象技术所声称的巨大好处，也就是可维护，可扩展可服用，灵活性好  开发人员应该进队程序中呈现出频繁变化的那些部分做出抽象，然而，对于应用程序中每个部分都可以的进行抽象并不是好主意，拒绝不成熟的抽象和抽象本身一样重要  **里氏代换原则**：子类型必须能够替换掉他们的父类型  子类继承付了，表示子类拥有父类所有非private的行为和属性  鸟类会飞，作为父类，企鹅是无法继承鸟类的，尽管企鹅在生物学上是年鸟类，但是由于不会飞，在编程上是无法继承的  只有当子类可以替换掉父类，软件单位的功能不受到影响时，父类才能够真正被复用而子类也能够在父类基础上增加新的行为依赖  **依赖倒转原则**：  A：高层模块不应该依赖底层模块，两个都应该依赖抽象（接口或者抽象类）  B：抽象不应该依赖细节，细节应该依赖抽象 |

1. 装饰者模式

|  |
| --- |
| 装饰模式：动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更加灵活    class Program  {  static void Main(string[] args)  {  ConcreteComponent c = new ConcreteComponent();  ConcreteDecoratorA d1 = new ConcreteDecoratorA();  ConcreteDecoratorB d2 = new ConcreteDecoratorB();  d1.SetComponent(c);  d2.SetComponent(d1);  d2.Operation();  Console.Read();  }  }  abstract class Component  {  public abstract void Operation();  }  class ConcreteComponent : Component  {  public override void Operation()  {  Console.WriteLine("具体对象的操作");  }  }  abstract class Decorator : Component  {  protected Component component;  public void SetComponent(Component component)  {  this.component = component;  }  public override void Operation()  {  if (component != null)  {  component.Operation();  }  }  }  class ConcreteDecoratorA : Decorator  {  private string addedState;  public override void Operation()  {  base.Operation();  addedState = "New State";  Console.WriteLine("具体装饰对象A的操作");  }  }  class ConcreteDecoratorB : Decorator  {  public override void Operation()  {  base.Operation();  AddedBehavior();  Console.WriteLine("具体装饰对象B的操作");  }  private void AddedBehavior()  {  }  }  如果只有一个ConcreteComponent类，而没有抽象的component类，那么Decorator类可以是ConcreteComponent的一个子类，同理如果只有一个ConcreteDecorator类，就没有必要建立一个单独的Decorator类    总结：装饰模式是为已有功能动态的添加更多功能的一种方式，普通方式：当系统需要新的功能时，是向旧的类中添加新的代码，这些新加的代码通常装饰了原有类的核心职责或主要行为，在主类中加入了新的字段，新的方法和新的逻辑，从而增加了主类的复杂度。装饰模式却提供了一个非常好的解决方案，它把每个要装饰的功能放在单独的类中，并让这个类包装它所需要包装的类，因此，当需要执行特殊行为时，客户代码就可以在运行时根据需要有选择地，有顺序的使用装饰功能包装对象了 |

1. 代理模式

|  |
| --- |
| 为其他对象提供一种代理以控制这个对象的访问 |

1. 工厂方法模式

|  |
| --- |
| 简单工厂模式的最大优点在于工厂类中包含了必要的逻辑判断，根据客户端的选择条件动态实例化相关的类，对于客户端来说，去除了与具体产品的依赖。  工厂方法模式：定义一个用于创建对象的接口，让子类决定实例化哪一个类，工厂方法是的类的实例化延迟到其子类，工厂类抽象出一个接口，这个接口只有一个方法，就是创建抽象产品的工厂方法，然后所有要生产的具体类的工厂，就去实现这个接口。  工厂方法模式实现时，客户端需要决定实例化哪个工厂来实现运算类，选择判断的问题还是存在的，也就是说，工厂方法把简单工厂的内部逻辑判断移到了客户端代码来进行。 |

1. 原型模式

|  |
| --- |
| 用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象    深复制与浅复制  默认的Cloneable接口的clone方法是浅复制，如果字段是值类型，则对该字段执行逐位复制，如果字段是引用类型，则复制引用但不复制引用的对象，因此原对象及其复本引用同一对象。被复制的对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用都仍然指向原来的对象。  深复制：把要复制的对象所应用的对象都复制一遍，即如果要克隆a，就要把a里面的所有成员对象的引用全部克隆。 |

1. 模板方法模式

|  |
| --- |
| 模板方法模式：定义一个操作中的算法的骨架，而将一些步骤延迟到子类，模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。    class Program  {  static void Main(string[] args)  {  AbstractClass c;  c = new ConcreteClassA();  c.TemplateMethod();  c = new ConcreteClassB();  c.TemplateMethod();  Console.Read();  }  }  abstract class AbstractClass  {  public abstract void PrimitiveOperation1();  public abstract void PrimitiveOperation2();  public void TemplateMethod()  {  PrimitiveOperation1();  PrimitiveOperation2();  Console.WriteLine("");  }  }  class ConcreteClassA : AbstractClass  {  public override void PrimitiveOperation1()  {  Console.WriteLine("具体类A方法1实现");  }  public override void PrimitiveOperation2()  {  Console.WriteLine("具体类A方法2实现");  }  }  class ConcreteClassB : AbstractClass  {  public override void PrimitiveOperation1()  {  Console.WriteLine("具体类B方法1实现");  }  public override void PrimitiveOperation2()  {  Console.WriteLine("具体类B方法2实现");  }  }  模板方法模式的特点：  通过把不变行为搬移到超类，去除子类中的重复代码来体现它的优势，当不变的和可变的行为在方法的子类实现中混合在一起的时候，不变的行为就是在子类中重复出现，我们通过模板方法模式吧这些行为搬移到单一的地方，这样就帮助子类拜托重复的不变行为的纠缠。 |

1. 迪米特法则

|  |
| --- |
| 迪米特法则：最少知识原则，如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用，如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用    外观模式：为子系统中的一组接口提供一个一致的界面，此模式定义了一个高层接口，这个接口是的这一子系统更加容易使用  何时使用外观模式：  首先，在设计初期阶段，应该要有意识的将不同的两个层分离，其次在开发阶段，子系统往往因为不断的重构演化而变得越来越复杂，大多数的模式使用时，也都会产生很多很小的类，增加外观Façade可以提供一个简单的接口，减少他们之间的依赖，第三，在维护一个遗留的大型系统时，可能这个系统已经非常难以维护和扩展了，但因为它包含非常重要的功能，新的需求开发必须依赖于他，此时用外观模式Façade也是最合适的 |

1. 建造者模式

|  |
| --- |
| 建造者模式：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示    建造者模式主要是用于创建一些复杂的对象，这些对象内部构建间的建造顺序通常是稳定的，但对象内部构建通常面临着复杂的变化。  建造者模式的好处就是使得建造代码与表示代码分离，由于建造者隐藏该产品是如何组装的，所以若需要改变一个产品的内部表示，只需要在定义一个具体的建造者就可以了  class Program  {  static void Main(string[] args)  {  Director director = new Director();  Builder b1 = new ConcreteBuilder1();  Builder b2 = new ConcreteBuilder2();  director.Construct(b1);  Product p1 = b1.GetResult();  p1.Show();  director.Construct(b2);  Product p2 = b2.GetResult();  p2.Show();  Console.Read();  }  }  class Director  {  public void Construct(Builder builder)  {  builder.BuildPartA();  builder.BuildPartB();  }  }  abstract class Builder  {  public abstract void BuildPartA();  public abstract void BuildPartB();  public abstract Product GetResult();  }  class ConcreteBuilder1 : Builder  {  private Product product = new Product();  public override void BuildPartA()  {  product.Add("部件A");  }  public override void BuildPartB()  {  product.Add("部件B");  }  public override Product GetResult()  {  return product;  }  }  class ConcreteBuilder2 : Builder  {  private Product product = new Product();  public override void BuildPartA()  {  product.Add("部件X");  }  public override void BuildPartB()  {  product.Add("部件Y");  }  public override Product GetResult()  {  return product;  }  }  class Product  {  IList<string> parts = new List<string>();  public void Add(string part)  {  parts.Add(part);  }  public void Show()  {  Console.WriteLine("\n产品 创建 ----");  foreach (string part in parts)  {  Console.WriteLine(part);  }  }  } |

1. 观察者模式

|  |
| --- |
| **观察者模式**：一种一对多的依赖关系，让多格观察者对象同时监听某一个主题对象。这个主题对象在状态发生变化时，会通知所有观察者对象，使他们能够自动更新自己    将一个系统分割成一系列相互协作的类有一个很不好的副作用，那就是需要维护相关对象间的一致性，我们不希望为了维持一致性而是各类紧密耦合，这样会给维护、扩展和重用都带来不便。而观察者模式的关键对象是subject和observer，一个subject可以有任意个observer，一旦subject的状态发生了改变，所有的observer都可以收到通知，subject发出通知并不需要知道谁是它的观察者，任意一个观察者也不需要知道其他观察者的存在  何时需要使用观察者模式：当一个对象的改变需要同时改变其他相关对象的时候，而且它不知道具体有多少对象有待改变时，应该使用观察者模式  事件委托说明  委托就是一种引用方法的类型，一旦为委托分配了方法，委托将与该方法具有完全相同的行为，委托方法的使用可以像其他任何方法一样，具有参数和返回值，委托课以看做是对函数的抽象，是函数的类，委托的实例将代表一个具体的函数  一个委托可以搭载多个方法，所有方法被一次唤起，更重要的是，他可以使得委托对象搭载的方法并不需要属于同一类，委托对象所搭载的所有方法必须具有相同的原型和形式，也就是有相同的参数列表和返回值 |

1. 抽象工厂模式

|  |
| --- |
| 抽象工厂模式：提供一个个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类    AbstractA和B是两个抽象产品，之所以为抽象，是因为她们都有可能有两个不同的实现，就刚才的例子来说，就是User和Department，而ProductA1和2等就是两个抽象产品的具体分类的实现。IFactory是一个抽象工厂的接口，它里面应该包含所有的产品创建的抽象方法。而ConcreteFactory1和2就是具体的工厂了。通常是在运行时刻在创建一个ConcreteFactory类的实例，这个具体的工厂再创建具有特定实现的产品对象，也就是说，为创建不同的产品对象，客户端使用不同的具体工厂  优点与缺点：  ①于交换产品系列，由于具体工厂类，在一个应用中只需要在初始化的时候出现一次，这就使得改变一个应用的具体工厂变得非常容易，它只需要改变具体共产即可使用不同的产品配置。  ②它让具体的创建实例过程与客户端分离，客户端是通过它们的抽象接口操纵实例，产品的具体类名也被具体工厂的实现分离，不会出现在客户代码中。  通过反射机制可以动态的访问类的属性，方法以及字段 |

1. 状态模式

|  |
| --- |
| 当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类  状态模式主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况，把状态的判断逻辑转移到表示不同状态的一系列类当中，可以把复杂的判断逻辑简化 |

1. 适配器模式

|  |
| --- |
| 将一个类的接口转换成客户希望的另外一个接口，adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作  使用情况：系统的数据和行为都正确，但接口不符，我们应该考虑用适配器，目的是使控制范围之外的一个原油对象与某个接口匹配，适配器模式主要应用于希望复用一些现存的类，但是接口又与复用环境要求不一致的情况。    class Program  {  static void Main(string[] args)  {  Target target = new Adapter();  target.Request();  Console.Read();  }  }  class Target  {  public virtual void Request()  {  Console.WriteLine("普?通ª¡§请?求¨®");  }  }  class Adaptee  {  public void SpecificRequest()  {  Console.WriteLine("特¬?殊ºa请?求¨®");  }  }  class Adapter : Target  {  private Adaptee adaptee = new Adaptee();  public override void Request()  {  adaptee.SpecificRequest();  }  }  在双方都不太容易修改的手再使用适配器模式适配，而不是一有不同就使用它 |

1. 备忘录模式

|  |
| --- |
| 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样以后就可以将该对象恢复到原先保存的状态    class Program  {  static void Main(string[] args)  {  Originator o = new Originator();  o.State = "On";  o.Show();  Caretaker c = new Caretaker();  c.Memento = o.CreateMemento();  o.State = "Off";  o.Show();  o.SetMemento(c.Memento);  o.Show();  Console.Read();  }  }  class Originator  {  private string state;  public string State  {  get { return state; }  set { state = value; }  }  public Memento CreateMemento()  {  return (new Memento(state));  }  public void SetMemento(Memento memento)  {  state = memento.State;  }  public void Show()  {  Console.WriteLine("State=" + state);  }  }  class Memento  {  private string state;  public Memento(string state)  {  this.state = state;  }  public string State  {  get { return state; }  }  }  class Caretaker  {  private Memento memento;  public Memento Memento  {  get { return memento; }  set { memento = value; }  }  }  Memento模式比较适用于功能比较复杂的，但需要维护或记录属性历史的类，或者需要保存的属性只是众多的属性中的一小部分时，originator可以根据保存的memento信息还原到前一个状态 |

1. 组合模式

|  |
| --- |
| 将对象组合成树形结构以表示“部分-整体”的层次结构关系使得用户对单个对象和组合对象的使用具有一致性    class Program  {  static void Main(string[] args)  {  Composite root = new Composite("root");  root.Add(new Leaf("Leaf A"));  root.Add(new Leaf("Leaf B"));  Composite comp = new Composite("Composite X");  comp.Add(new Leaf("Leaf XA"));  comp.Add(new Leaf("Leaf XB"));  root.Add(comp);  Composite comp2 = new Composite("Composite XY");  comp2.Add(new Leaf("Leaf XYA"));  comp2.Add(new Leaf("Leaf XYB"));  comp.Add(comp2);  root.Add(new Leaf("Leaf C"));  Leaf leaf = new Leaf("Leaf D");  root.Add(leaf);  root.Remove(leaf);  root.Display(1);  Console.Read();  }  }  abstract class Component  {  protected string name;  public Component(string name)  {  this.name = name;  }  public abstract void Add(Component c);  public abstract void Remove(Component c);  public abstract void Display(int depth);  }  class Composite : Component  {  private List<Component> children = new List<Component>();  public Composite(string name)  : base(name)  { }  public override void Add(Component c)  {  children.Add(c);  }  public override void Remove(Component c)  {  children.Remove(c);  }  public override void Display(int depth)  {  Console.WriteLine(new String('-', depth) + name);  foreach (Component component in children)  {  component.Display(depth + 2);  }  }  }  class Leaf : Component  {  public Leaf(string name)  : base(name)  { }  public override void Add(Component c)  {  Console.WriteLine("Cannot add to a leaf");  }  public override void Remove(Component c)  {  Console.WriteLine("Cannot remove from a leaf");  }  public override void Display(int depth)  {  Console.WriteLine(new String('-', depth) + name);  }  }  透明方式：在component中声明所有用来管理子对象的方法，其中包括add，remove等，这样实现component接口的所有子类都具备了add和remove，这样做的好处是叶节点和枝节点对于外界没有区别，他们具备完全一致的行为接口，但问题也很明显，因为Leaf类本生不具备add和remove的功能，所以实现它没有意义  安全方式：在component接口汇总不去声明add和remove方法，那么子类的leaf也就不需要去实现它，而是在composite声明所有用来管理子类对象的方法，这样做就不会出现刚才提到的问题，不过由于不够透明，所以树叶和树枝类将不具有相同的接口，客户端的调用需要做相应的判断。  何时要用组合模式：  当你发现需求中是体现部分和整体层次的结构时，以及你希望用户可以忽略组合对象和单个对象的不同，统一的使用组合结构中的所有对象时，就应该考虑使用组合模式了 |

1. 迭代器模式

|  |
| --- |
| 提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。 |

1. 单例模式

|  |
| --- |
| 单例模式：保证一个类仅有一个实例，并提供了一个访问他的全局访问点  通常我们可以让一个全局变量使得一个对象被访问，但他不能防止你实例化多个对象，一个最好的办法就是，让类自身负责保存它的唯一实例，这个类可以保证没有其他实例可以被创建，并且它自己可以提供一个访问该实例的方法 |

1. 桥接模式

|  |
| --- |
| 合成/聚合原则：尽量使用合成/聚合，尽量不用继承  聚合表示一种弱的拥有关系，体现的是A对象可以包含B对象，但B对象不是A对象的一部分；组合就是一种强的拥有关系，体现了严格的部分和整体的关系，部分和整体的生命周期一样。大雁和雁群，雁群包含大雁，但是这个大雁并不是那个燕群的，大雁和翅膀，整体和部分，生命周期一致。  组合和聚合的好处：优先使用对象的组合和聚合有助于保持每个类被封装，并被集中在单个任务上。这样类和类继承层次会保持较小规模，并且不太可能增长为不可控制的庞然大物  桥接模式：将抽象部分和他的实现部分分离，使他们可以独立化 |

1. 命令模式

|  |
| --- |
| 命令模式：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作  命令模式的优点：  ①它能比较容易的设计一个命令队列；  ②在需要的情况下，可以较容易的将命令记入日志  ③允许接收请求的一方决定是否要否决请求  ④可以容易的实现请求的撤销和重做  ⑤由于加进新的具体命令类不影响其他的类，因此增加新的具体命令类很容易，命令模式把请求一个操作的对象与知道怎么执行一个操作的对象分隔开 |

1. 职责链模式

|  |
| --- |
| 职责链模式：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这个对象连成一条链，并沿着这条链传递该请求，知道有一个对象处理它为止    责任链的好处：当客户提交一个请求时，请求是沿着链传递直至有一个handler对象负责处理它。使得接受着和发送者都没有对方的明确信息，且链中的对象自己也不知道链的结构，结果是职责链可简化对象的相互连接，它们仅需保持一个指向其后继者的引用，而不需要保持它所有的候选接受者的引用。 |

1. 中介者模式

|  |
| --- |
| 中介者模式：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立的改变他们之间的交互  中介者模式的优点：mediator的出现减少了各个colleague的耦合，是的可以独立的改变和复用各个colleague和mediator，比如任何国家的改变不会影响到其他国家，而只是与安理会发生变化。其次，由于把对象如何协作进行了抽象，将中介作为一个独立的概念并将其封装在一个对象中，这样关注的对象就从对象各自本身的行为转移到他们之间的交互上来，也就是站在一个更宏观的角度看带系统。 |

1. 享元模式

|  |
| --- |
| 运用共享技术有效的支持大量细粒度的对象    在享元对象内部并且并且不会随着环境的改变而改变的共享部分，可以称为是享元对象的内部状态，而随环境改变而改变的，不可以共享的状态就是外部状态。事实上，享元模式可以避免大量非常相似类的开销，在程序设计中，有时需要生成大量细粒度的类实例来表示数据。如果能发现这些实例除了几个参数外基本都是相同的，有时就能够很大幅度的减少需要实例化的类的数量。如果能把哪些参数移到类实例的外面，在方法调用时将他们传递进来，就可以通过共享大幅度的减少单个实例的数目。  享元模式的应用：  如果一个应用程序使用了大量的对象，而大量的这些对象造成了很大的开销时就应该考虑使用；还有就是对象的大多数状态可以外部状态，如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象，此时考虑使用享元模式  因为用了享元模式，所以有了共享对象，实例总数就大大减少了，如果共享的对象越多，存储节约也就越多，节约量随着共享状态的增多而增大  这就是享元模式的应用，两个字符串是相同的实例  这么写就不是同一个实例 |

1. 解释器模式

|  |
| --- |
| 解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子    如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个势力表述为一个简单语言中的句子，这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。 通常当有一个语言需要解释执行，并且你可以将改语言中的句子表示为一个抽象语法树事，可使用解释器模式  解释器为文法中的每一条规则至少定义了一个类，因此包含许多规则的文法可能难以管理和维护，建议当文法非常复杂时，使用其他的技术如语法分析程序或者编译期生成器来处理 |

# CSDN设计模式原则

## 单一职责原则

定义：不要存在多于一个导致类变更的原因，通俗的说，即一个类只负责一项职责

问题由来：类T负责两个不同的职责，职责P1和职责P2.当由于职责P1需求发生变化而需要改变类T的时候，有可能导致原本运行的正常的职责P2功能发生故障。

解决方案：可遵循单一职责原则。分别建立两个类T1，T2，使得T1完成职责P1的功能，T2完成P2功能。这样，当修改类T1的时候，不会使得职责P2发生故障危险；同理当修改T2时，也不会使职责P1发生故障。

虽然单一职责原则如此简单，但是即使是经验丰富的程序员写出的程序，也会违背这一原则、因为有职责扩散。所谓职责扩散，就是因为某种原因，职责P被分化为粒度更细的职责P1和P2. 比如类T只负责一个职责P，这样设计是符合单一职责原则的。后来由于某种原因，需求变更了，要将职责P分解为更加细的职责P1和P2，这时如果要使程序遵循单一职责原则，需要将类T也分解为两个T1和T2，分别负责P1、P2两个职责。但是在程序写好的情况下，这样太浪费时间。所以，简单的修改类T，用它来负责两个职责是一个不错的选择，虽然这样是不错的选择，但是违背了单一原则（风险：可能P将来还会分解为P1，P2，P3等，所以记住，在职责扩散到我们无法控制的程度时，立刻对代码进行重构）

针对上述情况，代码修改有三种：

1. 遵循单一原则
2. 在类中加入if条件判断
3. 在类中写另外的方法，不改变原来的方法

真实情况中，选择哪种很难说，可以这样，只有逻辑足够简单，才可以在代码级别上②违反单一原则；只有类中方法数量足够少③，才可以在方法级别上违反单一职责原则。

单一职责的优点：可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；提高类的可读性，提高系统的可维护性；变更引起的风险降低，变更是，必然的，如果单一职责做的好，当修改一个功能时，可以显著降低对其他功能的影响。

需要说明一点，单一职责原则不只是面向对象编程思想所特有的，只要是模块化程序设计，都是用单一职责原则。

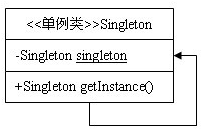
# CSDN设计模式整理

## 单例模式

定义：确保类只有一个实例，而且自行实例化并向整个系统提供这个实例。

类型：创建对象型模式

类图：



单例模式要素：

1. 私有的构造方法
2. 指向自己的实例的私有静态引用
3. 以自己实例为返回值的静态的共有的方法

单例模式根据实例化对象时机分为不同的两种：一种是饿汉式单例，一种是懒汉式单例。

饿汉式单例：在单例类被加载的时候，就实例化一个对象交给自己的引用；

|  |
| --- |
| public class Singleton {      private static Singleton singleton = new Singleton();      private Singleton(){}      public static Singleton getInstance(){          return singleton;      }  } |

懒汉式单例在调用取得**实例方法**的时候才会实例化对象。

|  |
| --- |
| public class Singleton {      private static Singleton singleton;      private Singleton(){}        public static synchronized Singleton getInstance(){          if(singleton==null){              singleton = new Singleton();          }          return singleton;      }  } |

单例模式的优点：

1. 在内存中只有一个对象，节省内存空间
2. 避免频繁创建销毁对象，可以提高性能
3. 避免对共享资源的多重占用
4. 可以全局访问

适用场景：由于单例模式的以上优点，所以是编程中用的比较多的一种设计模式。

1. 需要频繁实例化然后销毁对象
2. 创建对象时耗时过多或耗资源过多，但友经常用到对象
3. 有状态的工具类对象
4. 频繁访问数据库或文件的对象。

注意事项：

1. 只要能使用单例类提供的方法得到单例对象，不要使用反射，否则会实例化一个新对象
2. 不要做断开单例类对象与类中静态引用的危险操作。
3. 多线程使用单例共享资源时，注意线程安全问题

Java中的一些争议：

单例模式的对象长时间不用会被jvm垃圾收集器收集嘛？

在hotspot虚拟机1.6版本中，除非人为的断开单例模式中静态引用到单例对象的连接，否则jvm垃圾收集器是不会回收单例对象的。

在一个jvm中会出现多个单例吗？

在分布式系统，多个类加载器、以及序列化的情况下，会产生多个单例，这是毋庸置疑的。那么在同一jvm中，会不会产生多个单例？使用单例提供的getInstance（）方法只能得到一个单例，除非是使用反射机制，将会得到新的单例。

|  |
| --- |
| Class c = Class.forName(Singleton.class.getName());  Constructor ct = c.getDeclaredConstructor();  ct.setAccessible(true);  Singleton singleton = (Singleton)ct.newInstance(); |

这样，每次运行都会产生新的单例对象，所以运用单例模式时，一定要注意不要使用反射产生新的单例对象。

懒汉式单例线程安全吗？

经过测试后，发现加synchronized关键字修饰后，虽然对性能有部分影响，但却是线程安全的，不会产生多个实例对象的情况。

单例类可以被继承吗？

饿汉式和懒汉式单例的构造方法都是private的，所以她们都是不可继承的，但是其他很懂单例模式是可以继承的，例如登记式单例

饿汉式单例好还是懒汉式单例好？

在java中，饿汉式单例要优于懒汉式单例，C++中一般使用懒汉式单例