## 步骤：

什么是BigTable？

BigTable的特性有哪些？

为什么要使用BigTable？

如何使用BigTable？

BigTable的实现原理是什么？

BigTable的优缺点是什么？

参考资料

## UML

### 关系图：

<http://www.uml.org.cn/oobject/201609062.asp>

#### 1. 泛化（Generalization）

【泛化关系】：是一种继承关系，表示一般与特殊的关系，它指定了子类如何特化父类的所有特征和行为。例如：老虎是动物的一种，即有老虎的特性也有动物的共性。

【箭头指向】：带三角箭头的实线，箭头指向父类



#### 2. 实现（Realization）

【实现关系】：是一种类与接口的关系，表示类是接口所有特征和行为的实现.

【箭头指向】：带三角箭头的虚线，箭头指向接口



#### 3. 关联（Association)

【关联关系】：是一种拥有的关系，它使一个类知道另一个类的属性和方法；如：老师与学生，丈夫与妻子关联可以是双向的，也可以是单向的。双向的关联可以有两个箭头或者没有箭头，单向的关联有一个箭头。

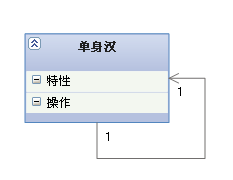
【代码体现】：成员变量

【箭头及指向】：带普通箭头的实心线，指向被拥有者



上图中，老师与学生是双向关联，老师有多名学生，学生也可能有多名老师。但学生与某课程间的关系为单向关联，一名学生可能要上多门课程，课程是个抽象的东西他不拥有学生。

下图为自身关联：



#### 4. 聚合（Aggregation）

【聚合关系】：是整体与部分的关系，且部分可以离开整体而单独存在。如车和轮胎是整体和部分的关系，轮胎离开车仍然可以存在。

聚合关系是关联关系的一种，是强的关联关系；关联和聚合在语法上无法区分，必须考察具体的逻辑关系。

【代码体现】：成员变量

【箭头及指向】：带空心菱形的实心线，菱形指向整体



#### 5. 组合(Composition)

【组合关系】：是整体与部分的关系，但部分不能离开整体而单独存在。如公司和部门是整体和部分的关系，没有公司就不存在部门。

组合关系是关联关系的一种，是比聚合关系还要强的关系，它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期。

【代码体现】：成员变量

【箭头及指向】：带实心菱形的实线，菱形指向整体



#### 6. 依赖(Dependency)

【依赖关系】：是一种使用的关系，即一个类的实现需要另一个类的协助，所以要尽量不使用双向的互相依赖.

【代码表现】：局部变量、方法的参数或者对静态方法的调用

【箭头及指向】：带箭头的虚线，指向被使用者



各种关系的强弱顺序：

泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖

下面这张UML图，比较形象地展示了各种类图关系：



## 工厂模式：

什么是BigTable？

BigTable的特性有哪些？

为什么要使用BigTable？

如何使用BigTable？

BigTable的实现原理是什么？

BigTable的优缺点是什么？

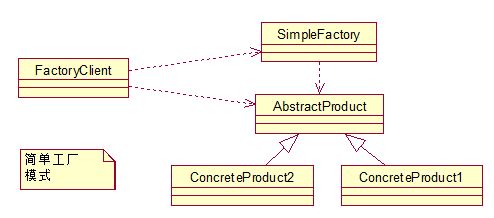
参考资料

#### 简单工厂模式：

这个模式本身很简单而且使用在业务较简单的情况下。一般用于小项目或者具体产品很少扩展的情况（这样工厂类才不用经常更改）。

它由三种角色组成：  
**工厂类角色：**这是本模式的核心，含有一定的商业逻辑和判断逻辑，根据逻辑不同，产生具体的工厂产品。如例子中的Driver类。  
**抽象产品角色**：它一般是具体产品继承的父类或者实现的接口。由接口或者抽象类来实现。如例中的Car接口。  
**具体产品角色**：工厂类所创建的对象就是此角色的实例。在java中由一个具体类实现，如例子中的Benz、Bmw类。

UML类图：（Abstract：抽象 ， Concrete: 具体）



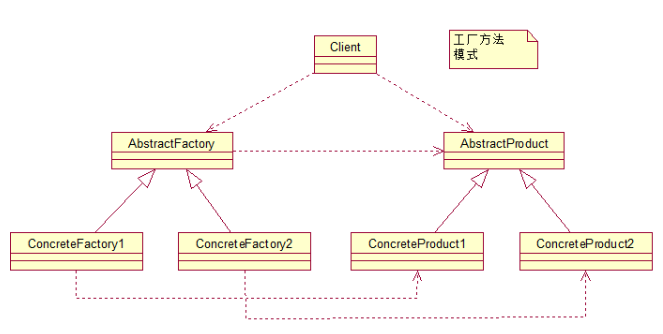
|  |
| --- |
| //抽象产品  **abstract** **class** Car{  **private** String name;    **public** **abstract** **void** drive();    **public** String getName() {  **return** name;      }  **public** **void** setName(String name) {  **this**.name = name;      }  }  //具体产品  **class** Benz **extends** Car{  **public** **void** drive(){          System.out.println(**this**.getName()+"----go-----------------------");      }  }    **class** Bmw **extends** Car{  **public** **void** drive(){          System.out.println(**this**.getName()+"----go-----------------------");      }  }    //简单工厂  **class** Driver{  **public** **static** Car createCar(String car){          Car c = **null**;  **if**("Benz".equalsIgnoreCase(car))              c = **new** Benz();  **else** **if**("Bmw".equalsIgnoreCase(car))              c = **new** Bmw();  **return** c;      }  }    //老板测试类  **public** **class** BossSimplyFactory {    **public** **static** **void** main(String[] args) **throws** IOException {          //老板告诉司机我今天坐奔驰          Car car = Driver.createCar("benz");          car.setName("benz");           //司机开着奔驰出发          car.drive();      } |

#### 工厂模式：

它由四种角色组成：

**抽象工厂角色：** 这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在java中它由抽象类或者接口来实现。  
**具体工厂角色**：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。在java中它由具体的类来实现。  
**抽象产品角色**：它是具体产品继承的父类或者是实现的接口。在java中一般有抽象类或者接口来实现。  
**具体产品角色：**具体工厂角色所创建的对象就是此角色的实例。在java中由具体的类来实现。

UML类图：（Abstract：抽象 ， Concrete: 具体）



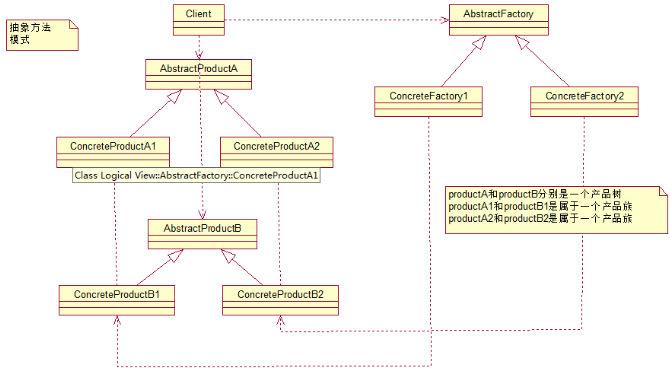
|  |
| --- |
| //抽象产品  **abstract** **class** Car{  **private** String name;    **public** **abstract** **void** drive();    **public** String getName() {  **return** name;      }  **public** **void** setName(String name) {  **this**.name = name;      }  }  //具体产品  **class** Benz **extends** Car{  **public** **void** drive(){          System.out.println(**this**.getName()+"----go-----------------------");      }  }  **class** Bmw **extends** Car{  **public** **void** drive(){          System.out.println(**this**.getName()+"----go-----------------------");      }  }      //抽象工厂  **abstract** **class** Driver{  **public** **abstract** Car createCar(String car) **throws** Exception;  }  //具体工厂（每个具体工厂负责一个具体产品）  **class** BenzDriver **extends** Driver{  **public** Car createCar(String car) **throws** Exception {  **return** **new** Benz();      }  }  **class** BmwDriver **extends** Driver{  **public** Car createCar(String car) **throws** Exception {  **return** **new** Bmw();      }  }    //老板  **public** **class** Boss{    **public** **static** **void** main(String[] args) **throws** Exception {          Driver d = **new** BenzDriver();          Car c = d.createCar("benz");          c.setName("benz");          c.drive();      }  } |

#### ****抽象工厂模式：****

所以抽象工厂模式一般用于具有产品树和产品族的场景下。

**抽象工厂角色：** 这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在java中它由抽象类或者接口来实现。  
**具体工厂角色：**它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。在java中它由具体的类来实现。  
**抽象产品角色：**它是具体产品继承的父类或者是实现的接口。在java中一般有抽象类或者接口来实现。  
**具体产品角色：**具体工厂角色所创建的对象就是此角色的实例。在java中由具体的类来实现。

UML类图：（Abstract：抽象 ， Concrete: 具体）



|  |
| --- |
| //抽象产品（Bmw和Audi同理）  **abstract** **class** BenzCar{  **private** String name;    **public** **abstract** **void** drive();    **public** String getName() {  **return** name;      }  **public** **void** setName(String name) {  **this**.name = name;      }  }  //具体产品（Bmw和Audi同理）  **class** BenzSportCar **extends** BenzCar{  **public** **void** drive(){          System.out.println(**this**.getName()+"----BenzSportCar-----------------------");      }  }  **class** BenzBusinessCar **extends** BenzCar{  **public** **void** drive(){          System.out.println(**this**.getName()+"----BenzBusinessCar-----------------------");      }  }    **abstract** **class** BmwCar{  **private** String name;    **public** **abstract** **void** drive();    **public** String getName() {  **return** name;      }  **public** **void** setName(String name) {  **this**.name = name;      }  }  **class** BmwSportCar **extends** BmwCar{  **public** **void** drive(){          System.out.println(**this**.getName()+"----BmwSportCar-----------------------");      }  }  **class** BmwBusinessCar **extends** BmwCar{  **public** **void** drive(){          System.out.println(**this**.getName()+"----BmwBusinessCar-----------------------");      }  }    **abstract** **class** AudiCar{  **private** String name;    **public** **abstract** **void** drive();    **public** String getName() {  **return** name;      }  **public** **void** setName(String name) {  **this**.name = name;      }  }  **class** AudiSportCar **extends** AudiCar{  **public** **void** drive(){          System.out.println(**this**.getName()+"----AudiSportCar-----------------------");      }  }  **class** AudiBusinessCar **extends** AudiCar{  **public** **void** drive(){          System.out.println(**this**.getName()+"----AudiBusinessCar-----------------------");      }  }      //抽象工厂  **abstract** **class** Driver3{  **public** **abstract** BenzCar createBenzCar(String car) **throws** Exception;    **public** **abstract** BmwCar createBmwCar(String car) **throws** Exception;    **public** **abstract** AudiCar createAudiCar(String car) **throws** Exception;  }  //具体工厂  **class** SportDriver **extends** Driver3{  **public** BenzCar createBenzCar(String car) **throws** Exception {  **return** **new** BenzSportCar();      }  **public** BmwCar createBmwCar(String car) **throws** Exception {  **return** **new** BmwSportCar();      }  **public** AudiCar createAudiCar(String car) **throws** Exception {  **return** **new** AudiSportCar();      }  }  **class** BusinessDriver **extends** Driver3{  **public** BenzCar createBenzCar(String car) **throws** Exception {  **return** **new** BenzBusinessCar();      }  **public** BmwCar createBmwCar(String car) **throws** Exception {  **return** **new** BmwBusinessCar();      }  **public** AudiCar createAudiCar(String car) **throws** Exception {  **return** **new** AudiBusinessCar();      }  }    //老板  测试类  **public** **class** BossAbstractFactory {    **public** **static** **void** main(String[] args) **throws** Exception {            Driver3 d = **new** BusinessDriver();          AudiCar car = d.createAudiCar("");          car.drive();      }  } |

抽象工厂模式的缺点：如果需要增加新的产品树，那么就要新增三个产品类，比如VolvoCar，VolvoSportCar,VolvoSportCar，并且要修改三个工厂类。这样大批量的改动是很丑陋的做法。

所以可以用简单工厂配合反射来改进抽象工厂：

|  |
| --- |
| abstract class BenzCar{      private String name;        public abstract void drive();        public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }  }  class BenzSportCar extends BenzCar{      public void drive(){          System.out.println(this.getName()+"----BenzSportCar-----------------------");      }  }  class BenzBusinessCar extends BenzCar{      public void drive(){          System.out.println(this.getName()+"----BenzBusinessCar-----------------------");      }  }    abstract class BmwCar{      private String name;        public abstract void drive();        public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }  }  class BmwSportCar extends BmwCar{      public void drive(){          System.out.println(this.getName()+"----BmwSportCar-----------------------");      }  }  class BmwBusinessCar extends BmwCar{      public void drive(){          System.out.println(this.getName()+"----BmwBusinessCar-----------------------");      }  }    abstract class AudiCar{      private String name;        public abstract void drive();        public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }  }  class AudiSportCar extends AudiCar{      public void drive(){          System.out.println(this.getName()+"----AudiSportCar-----------------------");      }  }  class AudiBusinessCar extends AudiCar{      public void drive(){          System.out.println(this.getName()+"----AudiBusinessCar-----------------------");      }  }      /\*\*   \* 简单工厂通过反射改进抽象工厂及其子工厂   \* @author Administrator   \*   \*/  class Driver3{      public static BenzCar createBenzCar(String car) throws Exception {          return (BenzCar) Class.forName(car).newInstance();      }        public static BmwCar createBmwCar(String car) throws Exception {          return (BmwCar) Class.forName(car).newInstance();      }        public static AudiCar createAudiCar(String car) throws Exception {          return (AudiCar) Class.forName(car).newInstance();      }  }  //客户端  public class SimpleAndAbstractFactory {        public static void main(String[] args) throws Exception {            AudiCar car = Driver3.createAudiCar("com.java.pattendesign.factory.AudiSportCar");          car.drive();      }  } |

## 建造者模式：

建造者模式：是将一个复杂的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。

 建造者模式通常包括下几个角色：   
1. builder（抽象建造者）：给出一个抽象结论，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的那些部分的创建，并不涉及具体的对象部件的创建。   
2. ConcreteBuilder（具体建造者）：实现Builder接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。在构造过程完成后，提供产品的实例。   
3. Director（指导者）：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。   
4. Product（产品类）：要创建的复杂对象。

举个简单例子（车的建造过程，包括造轮子，造发动机和造车身结构）：   
1 产品类

|  |
| --- |
| public class Car  {  private String wheel;  private String skeleton;  private String engine;  // 省略getter和setter方法  } |

2 抽象建造者

|  |
| --- |
| public interface ICarBuilder  {  public void buildWheel();  public void buildSkeleton();  public void buildEngine();  Car buildCar();  } |

3 具体建造者

|  |
| --- |
| public class ConcreteBuilder implements ICarBuilder  {  Car car;  public ConcreteBuilder()  {  car = new Car();  }  @Override  public void buildWheel()  {  car.setWheel("轮子");  }  @Override  public void buildSkeleton()  {  car.setSkeleton("车身结构");  }  @Override  public void buildEngine()  {  car.setEngine("发动机");  }  @Override  public Car buildCar()  {  return this.car;  }  } |

4 指导者

|  |
| --- |
| public class CarDirector  {  public Car constructCar(ICarBuilder builder)  {  builder.buildEngine();  builder.buildSkeleton();  builder.buildWheel();  return builder.buildCar();  }  } |

10

5 测试代码

|  |
| --- |
| public class MainTest  {  public static void main(String[] args)  {  CarDirector director = new CarDirector();  Car car = director.constructCar(new ConcreteBuilder());  System.out.println(car.getWheel());  System.out.println(car.getEngine());  System.out.println(car.getSkeleton());  }  } |

**适用场景**

需要生产的产品对象有复杂的内部结构。

需要生产的产品对象的属性相互依赖，建造者模式可以强迫生成顺序。

在对象创建过程中会使用到系统中的一些其它对象，这些对象在产品对象的创建过程中不易得到。

**优点**

使用建造者模式可以使客户端不必知道产品内部的组成细节。（封装性）

具体的建造者之间是相互独立的，对系统的扩展非常有利。（扩展性）

由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他模块产生任何影响。

**与抽象工厂的区别：**

在建造者模式里，有个指导者，由指导者来管理建造者，用户是与指导者联系的，指导者联系建造者最后得到产品。即建造模式可以强制实行一种分步骤进行的构造过程。工厂模式是将对象的全部创建过程封装在工厂类中，由工厂类想客户端提供最终的产品。而在建造者模式中，建造者类一般只提供产品类中各个组件的建造，而将具体建造过程交付给指导者，由指导者负责将各个组件按照特定的规则组建为产品，然后将组建好的产品交付给客户端。

分页查询：

Mysql：

**select \* from table limit (pageNO-1)\*pageSize ,pageSize**

pageNO:表示第几页

pageSize：每页的条数

**select \* from table limit 8,4;**

表示： 返回第3页的数据，返回4条。

**select \* from table limit start,pageNum**

start：表示从第几条开始，

pageNum：表示连续返回几条

**select \* from table limit 8,4;**

表示：从第9个数字开始，连续4条，即9-12位置

Oracel：

select \* from (select rownum,name from table where rownum <=endIndex ) where rownum > startIndex

SELECT \* FROM (SELECT ROWNUM R,t1.\* From Sys\_option where rownum < 30 ) t2   
Where t2.R >= 10

表示：

从表Sys\_option(主键为sys\_id)中从10条记录还是检索20条记录，

DB2：

select \* from (select rownumber() over(order by id asc ) as rowid from table where rowid <=endIndex ) where rowid > startIndex

SELECT \* FROM (Select 字段1,字段2,字段3,rownumber() over(ORDER BY 排序用的列名 ASC) AS rn from 表名) AS a1 WHERE a1.rn BETWEEN 10 AND 20

以上表示提取第10到20的纪录