

# 设计模式-面试课

## 问题1: 说一说单例模式?

### 1.1 什么是单例模式？

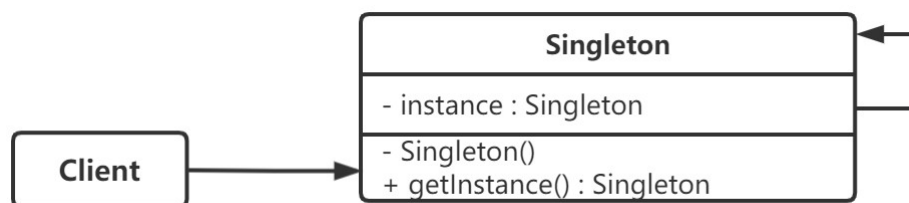
#### 1) 定义

单例模式是一种创建型设计模式，它确保一个类只有一个实例，并提供了一种全局访问点以访问该实例。

单例模式也比较好理解，比如一个人一生当中只能有一个真实的身份证号，一个国家只有一个政府，类似的场景都是属于单例模式。

单例模式通常用于那些需要在应用程序中仅存在一个实例的情况，例如配置管理器、线程池、数据库连接池等。通过使用单例模式，可以确保该类的唯一实例在整个应用程序中被共享和使用。

#### 2) 单例模式通常包含以下几个关键元素：



1. 私有的构造函数，以防止类的直接实例化。
2. 提供一个私有的静态成员变量，用于存储类的唯一实例。
3. 一个公共的静态方法，用于返回类的唯一实例。该方法负责创建实例（如果尚未创建）并返回该实例的引用。

### 1.2 饿汉式与懒汉式区别？

1. **初始化时机不同**：饿汉式单例模式在类加载时就立即初始化并创建单例对象，而懒汉式单例模式则是在第一次使用时才会进行初始化，并创建单例对象。
2. **线程安全性不同**：饿汉式单例模式在类加载时就创建了单例对象，因此天生就是线程安全的。而懒汉式单例模式在第一次使用时才会创建单例对象，如

果多个线程同时调用getInstance()方法，可能会创建出多个单例对象，因此需要进行同步控制来保证线程安全。

3. **内存占用不同**：饿汉式单例模式在类加载时就创建了单例对象，因此占用的内存较多。而懒汉式单例模式在第一次使用时才会创建单例对象，因此占用的内存相对较少。

## 饿汉式

在类加载期间初始化静态实例,保证 instance 实例的创建是线程安全的 (实例在类加载时实例化, 有JVM保证线程安全).

特点: 不支持延迟加载实例(懒加载), 此中方式类加载比较慢, 但是获取实例对象比较快

问题: 该对象足够大的话, 而一直没有使用就会造成内存的浪费。

```
public class Singleton_01 {

    private byte[] data1 = new byte[1024*1024];

    private byte[] data2 = new byte[1024*1024];

    private byte[] data3 = new byte[1024*1024];

    //1. 私有构造方法
    private Singleton_01(){

    }

    //2. 在本类中创建私有静态的全局对象
    private static Singleton_01 instance = new
Singleton_01();

    //3. 提供一个全局访问点,供外部获取单例对象
    public static Singleton_01 getInstance(){

        return instance;
    }

}
```

## 懒汉式(线程不安全)

此种方式的单例实现了懒加载,只有调用getInstance方法时 才创建对象.但是如果是多线程情况,会出现线程安全问题.

```
public class Singleton_02 {

    //1. 私有构造方法
    private Singleton_02(){

    }

    //2. 在本类中创建私有静态的全局对象
    private static Singleton_02 instance;

    //3. 通过判断对象是否被初始化,来选择是否创建对象
    public static Singleton_02 getInstance(){

        if(instance == null){

            instance = new Singleton_02();
        }
        return instance;
    }

    public static void main(String[] args) {

        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                instance = getInstance();
            }).start();
        }
    }

    /*
    * 打印结果, 构造方法被多次调用
    * Thread-0
    * Thread-3
    * Thread-2
    * Thread-1
    *
    */
}
```

```
*/  
  
}
```

假设在单例类被实例化之前，有两个线程同时在获取单例对象，线程A在执行完if (instance == null) 后，线程调度机制将 CPU 资源分配给线程B，此时线程B在执行 if (instance == null)时也发现单例类还没有被实例化，这样就会导致单例类被实例化两次。为了防止这种情况发生，需要对 getInstance() 方法同步处理。改进后的懒汉模式。

## 懒汉式(线程安全)

原理: 使用同步锁 `synchronized` 锁住 创建单例的方法，防止多个线程同时调用，从而避免造成单例被多次创建

1. 即，`getInstance()` 方法块只能运行在1个线程中
2. 若该段代码已在1个线程中运行，另外1个线程试图运行该块代码，则 **会被阻塞而一直等待**
3. 而在这个线程安全的方法里我们实现了单例的创建，保证了多线程模式下单例对象的唯一性

```
public class Singleton_03 {  
  
    //1. 私有构造方法  
    private Singleton_03(){  
  
    }  
  
    //2. 在本类中创建私有静态的全局对象  
    private static Singleton_03 instance;  
  
    //3. 通过添加synchronize,保证多线程模式下的单例对象的唯一性  
    public static synchronized Singleton_03 getInstance(){  
  
        if(instance == null){  
            instance = new Singleton_03();  
        }  
        return instance;  
    }  
  
}
```

懒汉式的缺点也很明显，我们给 `getInstance()` 这个方法加了一把大锁（`synchronized`），导致这个函数的并发度很低。量化一下的话，并发度是 1，也就相当于串行操作了。而这个函数是在单例使用期间，一直会被调用。如果这个单例类偶尔会被用到，那这种实现方式还可以接受。但是，如果频繁地用到，那频繁加锁、释放锁及并发度低等问题，会导致性能瓶颈，这种实现方式就不可取了。

## 1.3 DCL单例模式为什么要进行两次校验？

### 双重校验锁（DCL）

饿汉式不支持延迟加载，懒汉式有性能问题，不支持高并发。那我们再来看一种既支持延迟加载、又支持高并发的单例实现方式，也就是双重校验锁（Double Check Lock）实现方式。

#### 为什么要两次判断

- 第一次判断singleton是否为null

第一次判断是在`Synchronized`同步代码块外进行判断，由于单例模式只会创建一个实例，并通过`getInstance`方法返回singleton对象，所以，第一次判断，是为了在singleton对象已经创建的情况下，避免进入同步代码块，提升效率。

- 第二次判断singleton是否为null

第二次判断是为了避免以下情况的发生。

(1)假设：线程A已经经过第一次判断，判断`singleton=null`，准备进入同步代码块。

(2)此时线程B获得时间片，犹豫线程A并没有创建实例，所以，判断`singleton`仍然`=null`，所以线程B创建了实例`singleton`。

(3)此时，线程A再次获得时间片，由于刚刚经过第一次判断`singleton=null`（不会重复判断），进入同步代码块，这个时候，我们如果不加入第二次判断的话，那么线程A又会创建一个实例`singleton`，就不满足我们的单例模式的要求，所以第二次判断是很有必要的。

#### 在双重检查锁模式中为什么需要使用 `volatile` 关键字？

Volatile禁止JVM对指令进行重排序。所以创建对象的过程仍然会按照指令1-2-3的有序执行。

- **保证变量的可见性**：当一个被volatile关键字修饰的变量被一个线程修改的时候，其他线程可以立刻得到修改之后的结果。
- **屏蔽指令重排序**：指令重排序是编译器和处理器为了高效对程序进行优化的手段，它只能保证程序执行的结果时正确的，但是无法保证程序的操作顺序与代码顺序一致。这在单线程中不会构成问题，但是在多线程中就会出现

```
//3. 提供全局访问点，供外部获取单例对象
public static Singleton03 getInstance(){

    // 第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实例
    if(instance == null){
        // 第二次判断 抢到锁之后再次进行判断是否为null，避免多线程下重复创建
        synchronized(Singleton03.class){
            if(instance == null){
                instance = new Singleton03();
            }
        }
    }

    return instance;
}
```

**线程A**

**线程B**

线程B在执行第一次判断，singleton03不为空，直接返回，但是单例对象并未完成初始化，使用就会报错

1) 执行了 1 - 3，然后线程B来了

1) 为singleton03分配内存空间  
2) 调用构造函数，初始化singleton03  
3) 将singleton03对象指向分配的内存空间

以上三步，第2步与第3步的顺序是不能保证的，假设最终执行的顺序是 1 - 3 - 2，这个时候就会发生问题

在java内存模型中，volatile 关键字作用可以是保证可见性或者禁止指令重排。这里是因为 singleton = new Singleton()，它并非是一个原子操作，事实上，在JVM中上述语句至少做了以下这3件事：

- 第一步是给 singleton 分配内存空间；
- 第二步开始调用 Singleton 的构造函数等，来初始化 singleton；
- 第三步，将 singleton 对象指向分配的内存空间（执行完这步 singleton 就不是 null 了）。

这里需要留意一下 1-2-3 的顺序，因为存在指令重排序的优化，也就是说第2步和第3步的顺序是不能保证的，最终的执行顺序，可能是 1-2-3，也有可能是 1-3-2。

如果是 1-3-2，那么在第3步执行完以后，singleton 就不是 null 了，可是这时第2步并没有执行，singleton 对象未完成初始化，它的属性的值可能不是我们所预期的值。假设此时线程2进入 getInstance 方法，由于 singleton 已经不是 null 了，所以会通过第一重检查并直接返回，但其实这时的 singleton 并没有完成初始化，所以使用这个实例的时候会报错。

## 实现步骤:

1. 在声明变量时使用了 volatile 关键字
2. 将同步方法改为同步代码块. 在同步代码块中使用二次检查, 以保证其不被重复实例化 同时在调用getInstance()方法时不进行同步锁, 效率高。

```
public class Singleton03 {  
  
    private Singleton03() {  
        System.out.println(Thread.currentThread().getName());  
    }  
  
    private volatile static Singleton03 instance;  
  
    //3.提供全局访问点, 供外部获取单例对象  
    public static Singleton03 getInstance(){  
  
        //第一次判断,如果instance不为null,不进入抢锁阶段,直接返回实例  
        if(instance == null){  
            //第二次判断 抢到锁之后再次进行判断是否为null,避免多线程下重复创建  
            synchronized(Singleton03.class){  
                if(instance == null){  
                    instance = new Singleton03();  
                }  
            }  
        }  
  
        return instance;  
    }  
  
    public static void main(String[] args) {  
  
        for (int i = 0; i < 10; i++) {  
            new Thread(() -> {  
                instance = getInstance();  
            }).start();  
        }  
    }  
}
```

## 1.4 有没有比DCL模式更简洁的方式？

### 静态内部类

- 原理 根据 **静态内部类** 的特性(外部类的加载不影响内部类)，同时解决了按需加载、线程安全的问题，同时实现简洁

1. 在静态内部类里创建单例，在装载该内部类时才会去实例化单例
2. 线程安全：类是由 `JVM` 加载，而 `JVM` 只会加载1遍，保证只有1个单例

```
public class Singleton04 {  
  
    //静态内部类的特点，外部类的加载不影响内部类，同时解决了 按需加  
    载、线程安全的问题，并实现了代码简洁  
    private static class SingletonHandler{  
  
        private static Singleton04 instance = new  
Singleton04();  
    }  
  
    private Singleton04() {  
        System.out.println(Thread.currentThread().getName());  
    }  
  
    public static Singleton04 getInstance(){  
        return SingletonHandler.instance;  
    }  
  
    public static void main(String[] args) {  
  
        for (int i = 0; i < 10; i++) {  
            new Thread(() -> {  
                Singleton04 instance = getInstance();  
            }).start();  
        }  
    }  
}
```



## 1.5 如何阻止反射对于单例的破坏?

### 反射对于单例的破坏

反射的概念: JAVA反射机制是在运行状态中, 对于任意一个类, 都能够知道这个类的所有属性和方法; 对于任意一个对象, 都能够调用它的任意方法和属性; 这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

反射技术过于强大, 它可以通过 `setAccessible()` 来修改构造器, 字段, 方法的可见性。单例模式的构造方法是私有的, 如果将其可见性设为 `public`, 那么将无法控制对象的创建。

```
public class Test_Reflect {

    public static void main(String[] args) {

        try {

            //反射中, 欲获取一个类或者调用某个类的方法, 首先要获取到
            该类的Class 对象。
            Class<Singleton_05> clazz = Singleton_05.class;

            //getDeclaredXxx: 不受权限控制的获取类的成员。
            Constructor c =
            clazz.getDeclaredConstructor(null);

            //设置为true, 就可以对类中的私有成员进行操作了
            c.setAccessible(true);

            Object instance1 = c.newInstance();
            Object instance2 = c.newInstance();

            System.out.println(instance1 == instance2);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

解决方法之一: 在单例类的构造方法中 添加判断 `instance != null` 时, 直接抛出异常

```

public class Singleton_05 {

    private static class SingletonHandler{
        private static Singleton_05 instance = new
Singleton_05();
    }

    private Singleton_05(){
        if(SingletonHandler.instance != null){
            throw new RuntimeException("不允许非法访问!");
        }
    }

    public static Singleton_05 getInstance(){
        return SingletonHandler.instance;
    }
}

```

上面的这种方式使代码简洁性遭到破坏,设计不够优雅.

## 枚举(推荐方式)

枚举单例方式是 <<Effective Java>> 作者推荐的使用方式,这种方式

因为枚举类型是线程安全的,并且只会装载一次,设计者充分的利用了枚举的这个特性来实现单例模式,枚举的写法非常简单,而且枚举类型是所用单例实现中唯一一种不会被破坏的单例实现模式。

特点: 满足单例模式所需的 **创建单例、线程安全、实现简洁的需求**

```

public enum Singleton_05 {

    INSTANCE;

    public static Singleton_05 getInstance(){

        return INSTANCE;
    }

}

```

//我们用一个枚举实现单个数据源例子来简单验证一下: 声明一个枚举,用于获取数据库连接。

```

public enum DataSourceEnum {
    DATASOURCE;
    private DBConnection connection = null;
    private DataSourceEnum() {
        connection = new DBConnection();
    }
    public DBConnection getConnection() {
        return connection;
    }
}

```

使用反编译工具，查看枚举类的代码

```

C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.19045.2604]
(c) Microsoft Corporation。保留所有权利。

I:\MSB\msb_work\frame_design_pattern\design_pattern\target\classes\com\marshibing\singleton>jad -sjava Singleton05.class
Parsing Singleton05.class... Generating Singleton05.java

I:\MSB\msb_work\frame_design_pattern\design_pattern\target\classes\com\marshibing\singleton>

```

发现枚举类本质也是一个class类，并且只有一个带两个参数的构造方法。

```

public final class Singleton05 extends Enum
{
    public static Singleton05[] values()
    {
        return (Singleton05[])$VALUES.clone();
    }

    public static Singleton05 valueOf(String name)
    {
        return (Singleton05)Enum.valueOf(com/mashibing/singleton/Singleton05, name);
    }

    private Singleton05(String s, int i)
    {
        super(s, i);
    }

    public Object getData()
    {
        return data;
    }
}

```

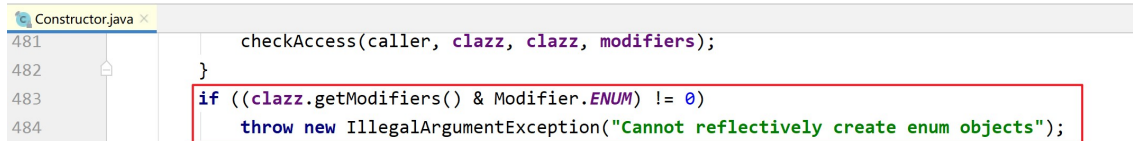
**问题1: 为什么枚举类可以阻止反射的破坏?**

1. 首先枚举类中是没有空参构造方法的,只有一个带两个参数的构造方法.

```
protected Enum(String name, int ordinal) {  
    this.name = name;  
    this.ordinal = ordinal;  
}
```

2. 真正原因是: 反射方法中不予许使用反射创建枚举对象

异常: 不能使用反射方式创建enum对象



```
481 checkAccess(caller, clazz, clazz, modifiers);  
482 }  
483 if ((clazz.getModifiers() & Modifier.ENUM) != 0)  
484     throw new IllegalArgumentException("Cannot reflectively create enum objects");
```

## 1.6 单例模式总结

单例模式是一种创建型设计模式，其主要目的是确保类只有一个实例，并提供一个全局访问点。以下是单例模式的优缺点和应用场景：

### 优点：

1. 确保一个类只有一个实例：在某些情况下，例如日志记录器或数据库连接器等，确保只有一个实例非常重要。
2. 提供一个全局访问点：单例模式可以提供一个全局访问点，让其他对象可以方便地访问该实例。
3. 确保对象的唯一性：单例模式可以确保对象的唯一性，避免出现多个相同对象的问题。
4. 减少资源消耗：由于只有一个实例，可以减少资源消耗，提高程序的效率。

### 缺点：

1. 违反单一职责原则：单例模式通常会将对象的创建和访问分离，这可能会导致代码的复杂性增加，违反单一职责原则。
2. 可能会引起并发问题：如果多个线程同时访问单例对象，可能会引起并发问题。需要使用同步机制来确保线程安全。
3. 难以测试：由于单例模式的对象是全局唯一的，所以在测试时可能会有一些困难。

### 应用场景：

1. 配置文件读取器：当程序需要读取配置文件时，可以使用单例模式确保只有一个读取器实例，以避免重复读取配置文件。

2. 日志记录器：当程序需要记录日志时，可以使用单例模式确保只有一个日志记录器实例，以避免日志记录器被多个线程同时使用，引起并发问题。
3. 数据库连接池：当程序需要访问数据库时，可以使用单例模式来管理数据库连接池，以确保只有一个连接池实例，避免资源的浪费和数据的不一致性。
4. 线程池：当程序需要管理多个线程时，可以使用单例模式来管理线程池，以确保只有一个线程池实例，避免线程池的资源竞争和管理复杂度。
5. GUI组件：当程序需要使用一些全局的GUI组件时，可以使用单例模式来管理这些组件的实例，以确保它们的一致性和协作。

## 问题2: 说一说你对代理模式的理解？

### 1.1 什么是代理模式

代理模式的定义: 对一个对象提供一个代理对象,使用代理对象控制对原对象的引用



代理模式的目的

- **透明的控制对象的访问**，代理模式的作用是隐藏对象的复杂性，控制对对象的访问，并在必要时增加一些额外的功能。

### 1.2 代理模式的分类

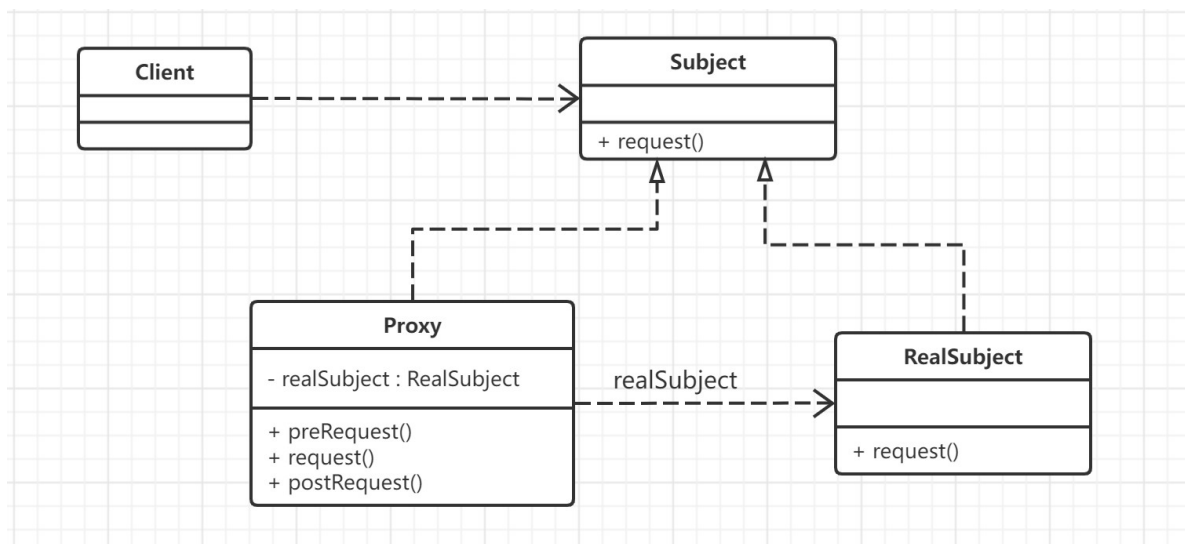
- **静态代理**: 是在编译期间就已经确定了代理关系，也就是说代理类和被代理类的关系在程序运行之前就已经确定了。静态代理需要自己定义一个代理类，这个代理类需要实现和被代理类一样的接口，然后在代理类中调用被代理类的方法，并且可以在方法前后添加一些额外的逻辑。
- **动态代理**: 是在运行期间动态生成代理类，也就是说代理类和被代理类的关系是在程序运行时动态生成的。

动态代理需要使用Java提供的Proxy类和InvocationHandler接口，通过实现InvocationHandler接口的invoke()方法来动态生成代理类。在调用代理类的方法时，会自动调用InvocationHandler的invoke()方法，在这个方法中可以对被代理类的方法进行拦截，并且可以在方法前后添加一些额外的逻辑。

## 1.3 代理模式结构图

1. 抽象主题（Subject）：定义了真实主题和代理主题的共同接口，这样代理就可以在任何时候代替真实主题。
2. 真实主题（Real Subject）：定义了代理所代表的对象，是代理所要代理的对象。
3. 代理（Proxy）：提供了与真实主题相同的接口，同时还负责创建和删除真实主题对象，使得客户端不直接与真实主题进行交互。
4. 客户端（Client）：通过代理访问真实主题对象。

在代理模式中，代理与真实主题实现相同的接口，从而使得客户端可以透明地使用代理来访问真实主题对象。代理可以在真实主题操作之前或之后执行一些附加操作，从而对真实主题进行控制。



## 1.4 代理模式的优缺点和应用场景

优点：

- 代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；

- 代理对象可以扩展目标对象的功能；
- 代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度；

#### 缺点：

- 增加了系统的复杂度；

#### 代理模式使用场景

- 功能增强

当需要对一个对象的访问提供一些额外操作时,可以使用代理模式

- 远程 (Remote) 代理

**实际上，RPC 框架也可以看作一种代理模式**，GoF 的《设计模式》一书中把它称作远程代理。通过远程代理，将网络通信、数据编解码等细节隐藏起来。**客户端在使用 RPC 服务的时候，就像使用本地函数一样，无需了解跟服务器交互的细节。**除此之外，RPC 服务的开发者也只需要开发业务逻辑，就像开发本地使用的函数一样，不需要关注跟客户端的交互细节。

- 防火墙 (Firewall) 代理

当你将浏览器配置成使用代理功能时，防火墙就将你的浏览器的请求转给互联网；当互联网返回响应时，代理服务器再把它转给你的浏览器。

- 保护 (Protect or Access) 代理

控制对一个对象的访问，如果需要，可以给不同的用户提供不同级别的使用权限。

## 问题3: 静态代理与动态代理的区别？

静态代理和动态代理都是实现代理模式的方式，但它们在实现上有很大的不同。下面是它们的主要区别：

### 1. 实现方式不同：

静态代理是在编译期就已经确定代理对象的类型，代理类需要手动编写。而动态代理是在运行时动态生成代理对象，代理类不需要手动编写，而是由框架自动生成。

### 1. 适用范围不同：

静态代理只适用于代理对象类型固定、接口较少的情况下。因为每增加一个被代理的接口，就需要编写一个新的代理类。而动态代理则可以代理任意的接口，无需编写新的代理类，因此更加灵活。

### 1. 性能表现不同：

由于静态代理在编译期就已经确定代理对象的类型，因此在运行时执行效率较高。而动态代理在运行时需要进行额外的代理对象生成、方法调用转发等操作，因此会存在一定的性能损失。

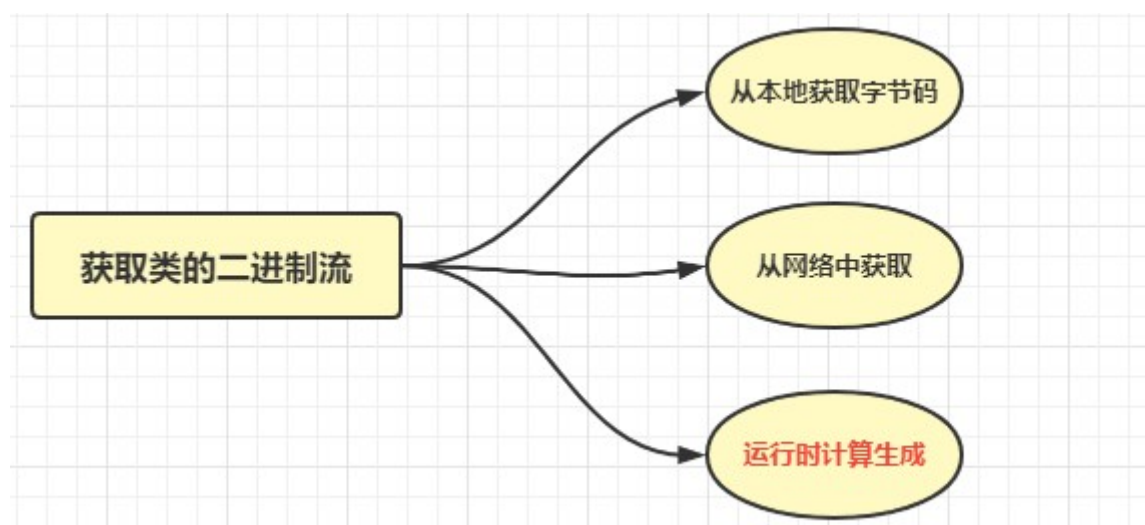
总的来说，静态代理适用于代理对象类型固定、接口较少、性能要求较高的情况。而动态代理则适用于代理对象类型不固定、接口较多、灵活性要求较高的情况。

## 类是如何动态生成的

Java虚拟机类加载过程主要分为五个阶段：加载、验证、准备、解析、初始化。其中加载阶段需要完成以下3件事情：

1. 通过一个类的全限定名来获取定义此类的二进制字节流
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
3. 在内存中生成一个代表这个类的 `java.lang.Class` 对象，作为方法区这个类的各种数据访问入口

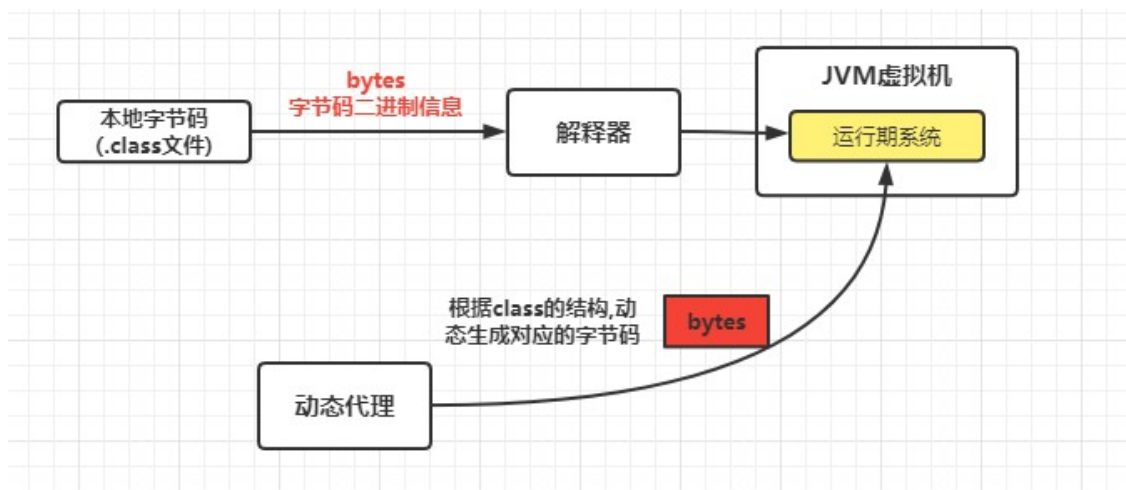
由于虚拟机规范对这3点要求并不具体，所以实际的实现是非常灵活的，关于第1点，**获取类的二进制字节流**（class字节码）就有很多途径：



- 从本地获取: Java虚拟机可以从本地文件系统加载类的字节码文件。



- 从网络中获取: : Java虚拟机也可以从网络中加载类的字节码数据。使用类加载器的getResourceAsStream()方法, 通过指定URL的方式来获取InputStream, 然后通过读取InputStream获取字节码数据。
- **运行时计算生成**, 这种场景使用最多的是动态代理技术, 在 java.lang.reflect.Proxy 类中, 就是用了 ProxyGenerator.generateProxyClass 来为特定接口生成形式为 `*$Proxy` 的代理类的二进制字节流



所以, **动态代理就是想办法, 根据接口或目标对象, 计算出代理类的字节码, 然后再加载到JVM中使用**

## 问题4: JDK动态代理与CGLIB动态代理的区别?

JDK动态代理和CGLIB动态代理是Java语言中实现动态代理的两种方式。它们之间的主要区别如下:

1. **基于的技术不同**: JDK动态代理是基于Java的反射机制实现的, 而CGLIB动态代理则是使用字节码生成技术实现的。
2. **被代理类的要求不同**: JDK动态代理只能代理实现了接口的类, 而CGLIB动态代理可以代理没有实现接口的类。
3. **代理性能不同**: JDK动态代理生成的代理类性能相对较低, 因为它是基于反射实现的, 而CGLIB动态代理生成的代理类性能相对较高, 因为它是基于字节码生成技术实现的。
4. **代理方式不同**: JDK动态代理是基于接口实现的, 因此代理类必须实现被代理接口, 而CGLIB动态代理是基于继承实现的, 因此代理类继承被代理类。

综上所述，JDK动态代理适合代理接口方法，而CGLIB动态代理适合代理普通类方法。如果被代理类实现了接口，那么优先选择JDK动态代理；如果被代理类没有实现接口，那么只能使用CGLIB动态代理。

JDK生成代理对象依赖的是反射, 动态代理类对象 继承了 Proxy 类，并且实现了被代理的所有接口

```
public final class $Proxy0 extends Proxy implements IUserDao {
    private static Method m3;

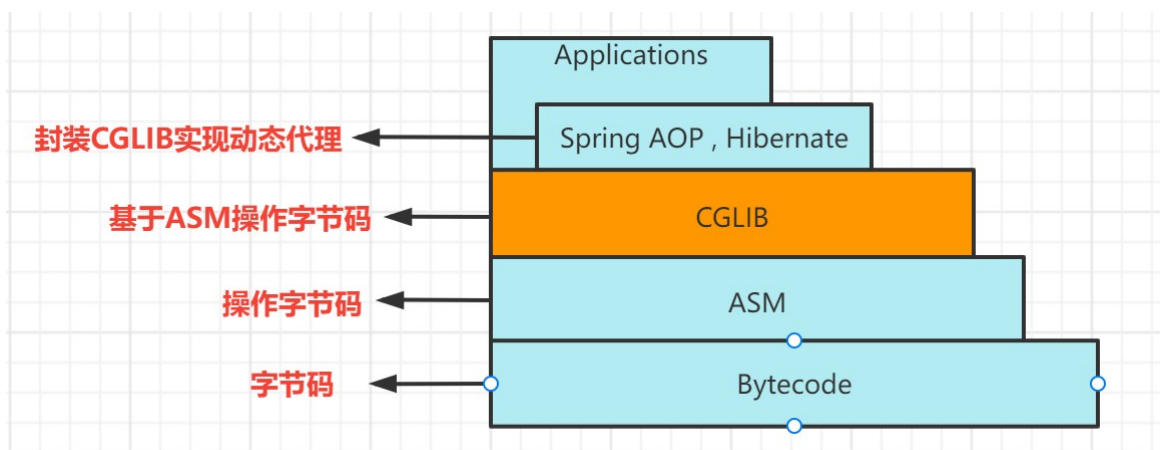
    public $Proxy0(InvocationHandler invocationHandler) {
        super(invocationHandler);
    }

    static {
        try {
            m3 = Class.forName("com.mashibing.proxy.example01.IUserDao").getMethod("save", new Class[0]);
            return;
        }
    }

    public final void save() {
        try {
            this.h.invoke(this, m3, null);
            return;
        }
    }
}
```

JDK生成代理对象依赖的是反射

cglib (Code Generation Library ) 是一个第三方代码生成类库，运行时在内存中动态生成一个子类对象从而实现对目标对象功能的扩展。cglib 为没有实现接口的类提供代理，为JDK的动态代理提供了很好的补充。



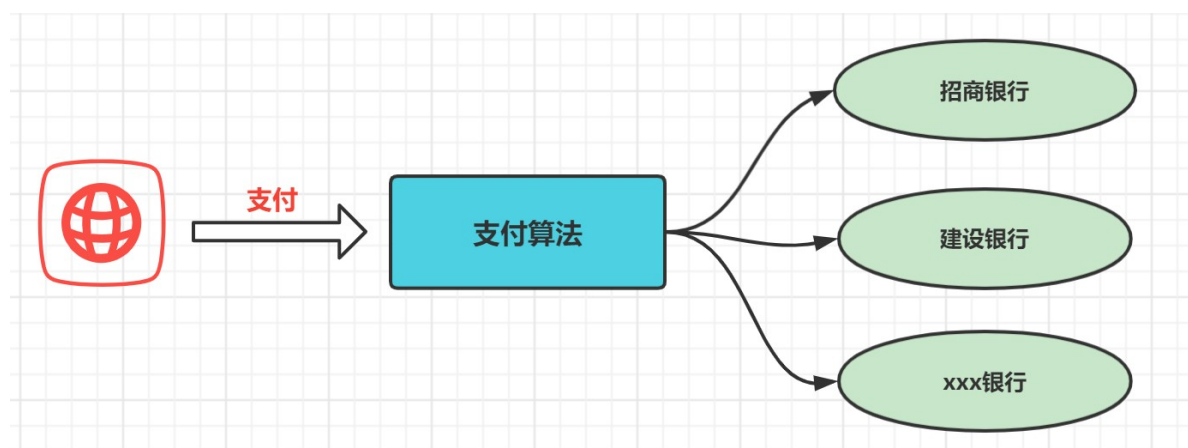
- 最底层是字节码
- ASM是操作字节码的工具
- cglib基于ASM字节码工具操作字节码（即动态生成代理，对方法进行增强）
- SpringAOP基于cglib进行封装，实现cglib方式的动态代理

## 问题5: 什么是策略模式?

策略模式(strategy pattern)的原始定义是: 定义一系列算法, 将每一个算法封装起来, 并使它们可以相互替换。策略模式让算法可以独立于使用它的客户端而变化。

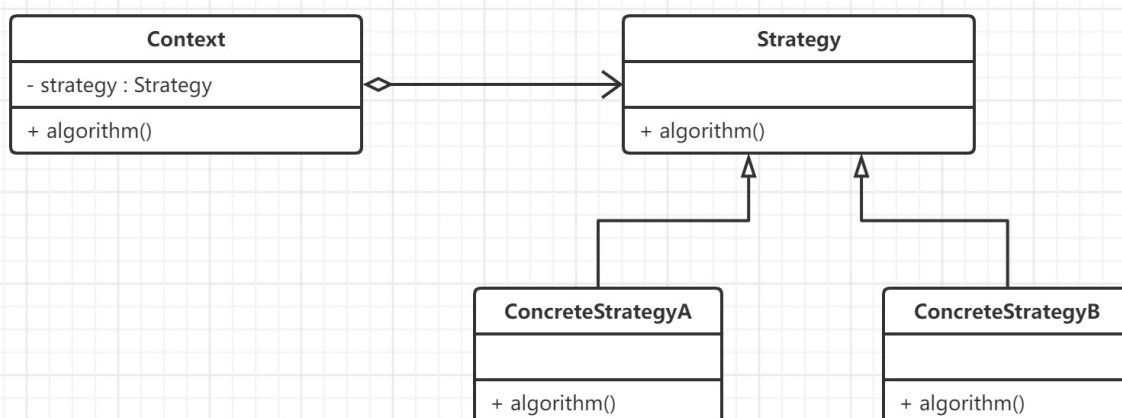
在软件开发中也会遇到相似的情况, 当实现某一个功能存在多种算法或者策略, 我们可以根据环境或者条件的不同选择不同的算法或者策略来完成该功能。

比如网购, 你可以选择工商银行、农业银行、建设银行等等, 但是它们提供的算法都是一致的, 就是帮你付款。



在策略模式中可以定义一些独立的类来封装不同的算法, 每一个类封装一种具体的算法, 在这里每一个封装算法的类都可以被称为一种策略, 为了保证这些策略在使用时具有一致性, 一般会提供一个抽象的策略类来做算法的声明. 而每种算法对应一个具体的策略类.

策略模式结构



策略模式的主要角色如下：

- 抽象策略（Strategy）类：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。
- 具体策略（Concrete Strategy）类：实现了抽象策略定义的接口，提供具体的算法实现或行为。
- 环境或上下文（Context）类：是使用算法的角色，持有一个策略类的引用，最终给客户端调用。

```
public interface Strategy {

    void algorithm();
}

public class ConcreteStrategyA implements Strategy {

    @Override
    public void algorithm() {
        System.out.println("执行策略A");
    }
}

public class ConcreteStrategyB implements Strategy {

    @Override
    public void algorithm() {
        System.out.println("执行策略B");
    }
}

/**
 * 环境类
 * @author spikeCong
 * @date 2022/10/13
 */
public class Context {

    //维持一个对抽象策略类的引用
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
}
```

```

        //调用策略类中的算法
        public void algorithm(){
            strategy.algorithm();
        }
    }

    public class Client {

        public static void main(String[] args) {

            Strategy strategyA = new ConcreteStrategyA();
            Context context = new Context(strategyA); //可以在运行时指定类型,通过配置文件+反射机制实现
            context.algorithm();
        }
    }

```

#### 优点:

1. 易于扩展和维护。由于不同的算法被封装在不同的类中，所以我们可以很容易地添加新的算法或修改已有算法，而不需要修改客户端的代码。
2. 提高代码的可读性。将不同的算法封装在不同的类中，使得代码更加模块化，易于理解和维护。
3. 可以消除大量的条件语句。使用策略模式，我们可以将不同的算法替换成不同的类，从而消除大量的if-else语句，使得代码更加简洁和易于理解。

#### 缺点:

1. 需要额外的类和接口。使用策略模式，我们需要为每个算法都创建一个独立的类，从而增加了代码的复杂度。
2. 客户端需要知道所有的策略类。使用策略模式，客户端需要知道所有的策略类，以便在运行时选择合适的策略。这可能会增加代码的复杂度。

#### 应用场景:

##### 策略模式适用于以下场景:

1. 需要根据不同的条件选择不同的算法时。例如，计算器程序需要根据用户输入的运算符选择相应的计算方法。
2. 需要在运行时动态地选择算法时。例如，某个系统需要根据用户的配置或环境变量来选择合适的算法。

3. 需要将算法的实现细节与客户端代码分离时。例如，某个系统需要根据不同的数据来源来解析数据，但是客户端并不关心数据的解析细节。

## 问题6: 如何用设计模式消除代码中的if-else(你在工作中使用过哪些设计模式)

### 1) 不使用设计模式

我们先看一段代码,这是一个请假审批流程功能,包含员工类, 请假单类和审核类.

```
public class Employee {  
  
    private String name;    //姓名  
  
    private int level;    //级别: P6 P7 P8  
}  
  
//请假单  
public class LeaveForm {  
  
    private com.mashibing03.old.pojo.Employee employee; //员工  
  
    private String reason;    //请假原因  
  
    private int days;    //天数  
  
    private int type;    //类型: 0-病假 , 1-婚丧假 , 2-年假  
  
}
```

```
//业务类  
public class LeaveService {  
  
    public void audit(LeaveForm leaveForm){  
  
        //3天以下婚丧假,自动通过  
        if(leaveForm.getDays() <= 3 && leaveForm.getType() ==  
1){  
            System.out.println("三天以下婚丧假 无需审批自动通  
过!");  
        }  
}
```

```

        if(leaveForm.getDays() > 3 && leaveForm.getType() ==
1){
            System.out.println("三天以上婚丧假 进入上级审批流
程!");
        }
        if(leaveForm.getEmployee().getLevel() == 9){
            System.out.println("总经理请假无需审批自动通过!");
        }
        if(leaveForm.getDays() == 1 && leaveForm.getType() ==
0){
            System.out.println("一天病假无需审批自动通过!");
        }
        if(leaveForm.getDays() > 1 && leaveForm.getType() ==
0){
            System.out.println("一天以上病假进入审批流程!");
        }
    }
}

```

## 2) 使用策略模式进行优化

通过策略模式, 将所有的if-else分支的业务逻辑抽取为各种策略类,判断条件和执行逻辑分装到对应的策略类中, 让客户端去依赖策略接口,保证具体策略类的改变不影响客户端.

- 策略接口

```

/**
 * 审核策略接口
 * @author spikeCong
 * @date 2023/3/21
 */
public interface AuditStrategy {

    //判断条件是否匹配
    public boolean isSupport(LeaveForm leaveForm);

    //审核业务逻辑
    public void audit(LeaveForm leaveForm);

    //规则冲突时的优先级
    public int getPriority();
}

```

```
//规则名称
public String getName();
}
```

- 具体策略类

```
public class AuditStrategyImpl_1 implements AuditStrategy {

    @Override
    public boolean isSupport(LeaveForm leaveForm) {
        return leaveForm.getDays() <= 3 &&
leaveForm.getType() == 1;
    }

    @Override
    public void audit(LeaveForm leaveForm) {
        System.out.println(leaveForm);
        System.out.println("三天以下婚丧假 无需审批自动通过!");
    }

    @Override
    public int getPriority() {
        return 0;
    }

    @Override
    public String getName() {
        return "三天以下婚假审批规则";
    }
}

public class AuditStrategyImpl_2 implements AuditStrategy {

    @Override
    public boolean isSupport(LeaveForm leaveForm) {
        return leaveForm.getDays() > 3 && leaveForm.getType()
== 1;
    }

    @Override
    public void audit(LeaveForm leaveForm) {
        System.out.println(leaveForm);
        System.out.println("三天以上婚丧假 进入上级审批流程!");
    }
}
```



```

    }

    @Override
    public int getPriority() {
        return 0;
    }

    @Override
    public String getName() {
        return "三天以上婚丧假审批规则";
    }
}

public class AuditStrategyImpl_3 implements AuditStrategy {

    @Override
    public boolean isSupport(LeaveForm leaveForm) {
        //级别9 总经理
        return leaveForm.getEmployee().getLevle() == 9;
    }

    @Override
    public void audit(LeaveForm leaveForm) {
        System.out.println(leaveForm);
        System.out.println("总经理请假无需审批自动通过!");
    }

    @Override
    public int getPriority() {
        return 999;
    }

    @Override
    public String getName() {
        return "总经理请假审批规则";
    }
}

```

- 策略工厂: 有了这个策略规则之后, 创建策略工厂, 用于策略规则的存储和管理

```

/**
 * 策略工厂

```

```

* @author spikeCong
* @date 2023/3/22
**/
public class AuditStrategyFactory {

    //私有静态全局唯一 工厂对象
    private final static AuditStrategyFactory factory = new
AuditStrategyFactory();

    //使用集合存储策略信息
    private List<AuditStrategy> auditStrategyList = new
ArrayList<>();

    //私有构造,用于注册规则
    private AuditStrategyFactory(){

        auditStrategyList.add(new AuditStrategyImpl_1());
        auditStrategyList.add(new AuditStrategyImpl_2());
        auditStrategyList.add(new AuditStrategyImpl_3());
        auditStrategyList.add(new AuditStrategyImpl_4());
        auditStrategyList.add(new AuditStrategyImpl_5());
    }

    //全局访问点 获取单例对象
    public static AuditStrategyFactory getInstance(){

        return factory;
    }

    //获取匹配的策略
    public AuditStrategy getAuditStrategy(LeaveForm
leaveForm){

        AuditStrategy auditStrategy = null;
        for (AuditStrategy strategy : auditStrategyList) {
            //找到匹配的规则
            if(strategy.isSupport(leaveForm)){
                if(auditStrategy == null){
                    auditStrategy = strategy;
                }else{
                    //优先级高的规则 替换现有的规则
                    //例如： 总经理请病假10,总经理优先级999,普通员工0

```

```

        if(strategy.getPriority() >
auditStrategy.getPriority()){
            auditStrategy = strategy;
        }
    }
}

if(auditStrategy == null){
    throw new RuntimeException("没有匹配到请假审核规
则");
}else{
    return auditStrategy;
}
}
}

```

- 业务类

```

public class LeaveServiceNew {

    public void audit(LeaveForm leaveForm){
        AuditStrategyFactory factory =
AuditStrategyFactory.getInstance();

        //返回符合条件的策略
        AuditStrategy strategy =
factory.getAuditStrategy(leaveForm);

        //通过策略类进行审核
        strategy.audit(leaveForm);
    }
}

```

- 测试

```

public class Client {

    public static void main(String[] args) {

        LeaveServiceNew leaveServiceNew = new
LeaveServiceNew();
    }
}

```

```

        LeaveForm form1 = new LeaveForm(new Employee("李总经理", 9), "甲流发烧", 10, 0);
        leaveServiceNew.audit(form1);

        LeaveForm form2 = new LeaveForm(new Employee("打工人1", 2), "甲流发烧", 2, 0);
        leaveServiceNew.audit(form2);

        LeaveForm form3 = new LeaveForm(new Employee("打工人2", 3), "结婚", 2, 1);
        leaveServiceNew.audit(form3);

        LeaveForm form4 = new LeaveForm(new Employee("打工人3", 4), "请年假,休息休息", 5, 2);
        leaveServiceNew.audit(form4);
    }
}

```

由于还没有添加年假规则,所以有报错

```

Exception in thread "main" java.lang.RuntimeException: 没有匹配到请假审核规则
    at com.mashibing03.strategy.AuditStrategyFactory.getAuditStrategy(AuditStrategyFactory.java:57)

```

- 添加新的规则, 只需要先建立对应的策略类,然后在工厂中添加一下规则即可

```

public class AuditStrategyImpl_6 implements AuditStrategy {

    @Override
    public boolean isSupport(LeaveForm leaveForm) {
        return leaveForm.getType() == 2;
    }

    @Override
    public void audit(LeaveForm leaveForm) {
        System.out.println(leaveForm);
        System.out.println("查询您的剩余年假天数...");
        System.out.println("剩余年假还有6天,进入审批流程");
    }

    @Override
    public int getPriority() {
        return 0;
    }
}

```

```
@Override
public String getName() {
    return "年假审批规则";
}
}
```

修改工厂代码

```
private AuditStrategyFactory(){

    auditStrategyList.add(new AuditStrategyImpl_1());
    auditStrategyList.add(new AuditStrategyImpl_2());
    auditStrategyList.add(new AuditStrategyImpl_3());
    auditStrategyList.add(new AuditStrategyImpl_4());
    auditStrategyList.add(new AuditStrategyImpl_5());

    //添加新的规则
    auditStrategyList.add(new AuditStrategyImpl_6());
}
```

经过上面的改造，我们已经消除了if-else的结构，每当新来了一种请假规则，只需要添加新的规则处理策略，并修改Factory中的集合。如果要使得程序符合开闭原则，则需要调整Factory中处理策略的获取方式，通过反射的方式，获取指定包下的所有Strategy实现类，然后放到字典集合中去。

## 基于InitializingBean的实现策略模式

InitializingBean接口提供了bean被BeanFactory设置了所有属性后的处理方法，它只包括afterPropertiesSet方法。也可用@PostConstruct注解实现，但接口实现方式调用效率上高一些，@PostConstruct确是通过反射机制调用的。

## 策略模式结构

