【济南中心】JAVA 编程阶梯：基础篇之第二十五章

多线程(单例设计模式)：

* 单例设计模式：保证类在内存中只有一个对象。

* 如何保证类在内存中只有一个对象呢？

    * (1)控制类的创建，不让其他类来创建本类的对象。private

    * (2)在本类中定义一个本类的对象。Singleton s;

    * (3)提供公共的访问方式。 public static Singleton getInstance(){return s}

* 单例写法两种：

    * (1)饿汉式 开发用这种方式。

```
//饿汉式
class Singleton {
        //1,私有构造函数
        private Singleton(){}
        //2,创建本类对象
        private static Singleton s = new Singleton();
        //3,对外提供公共的访问方法
        public static Singleton getInstance() {
                return s;
        }

        public static void print() {
                System.out.println("11111111111");
        }
}
```

    * (2)懒汉式 面试写这种方式。多线程的问题？

```
//懒汉式,单例的延迟加载模式
                class Singleton {
```

```
	//1,私有构造函数
	private Singleton(){}
	//2,声明一个本类的引用
	private static Singleton s;
	//3,对外提供公共的访问方法
	public static Singleton getInstance() {
		if(s == null)
			//线程 1,线程 2
			s = new Singleton();
		return s;
	}

	public static void print() {
		System.out.println("11111111111");
	}
}
```

* (3)第三种格式

```
class Singleton {
	private Singleton() {}

	public static final Singleton s = new Singleton();//final 是最终的意思,被 final 修饰的变量不可以被更改
}
```

多线程(Runtime 类)：

* Runtime 类是一个单例类

```
Runtime r = Runtime.getRuntime();
//r.exec("shutdown -s -t 300");		//300 秒后关机
r.exec("shutdown -a");			//取消关机
```

多线程(Timer)：

* Timer 类:计时器

```
public class Demo5_Timer {
	/**
	 * @param args
	 * 计时器
	 * @throws InterruptedException
	 */
	public static void main(String[] args) throws InterruptedException {
		Timer t = new Timer();
```

```
                t.schedule(new MyTimerTask(), new
Date(114,9,15,10,54,20),3000);

                while(true) {
                        System.out.println(new Date());
                        Thread.sleep(1000);
                }
        }
}
class MyTimerTask extends TimerTask {
        @Override
        public void run() {
                System.out.println("起床背英语单词");
        }

}
```

多线程(两个线程间的通信)：

* 1.什么时候需要通信

    * 多个线程并发执行时，在默认情况下 CPU 是随机切换线程的

    * 如果我们希望他们有规律的执行，就可以使用通信，例如每个线程执行一次打印

* 2.怎么通信

    * 如果希望线程等待，就调用 wait()

    * 如果希望唤醒等待的线程，就调用 notify();

    * 这两个方法必须在同步代码中执行，并且使用同步锁对象来调用

多线程(三个或三个以上间的线程通信)：

* 多个线程通信的问题

    * notify()方法是随机唤醒一个线程

    * notifyAll()方法是唤醒所有线程

* JDK5 之前无法唤醒指定的一个线程

* 如果多个线程之间通信，需要使用 notifyAll()通知所有线程，用 while 来反复判断条件

多线程(线程间的通信注意的问题)：

* 1.在同步代码块中,用哪个对象锁,就用哪个对象调用 wait 方法

* 2.为什么 wait 方法和 notify 方法定义在 Object 这个类中?

* 锁对象可以是任意对象,那么任意对象对应的类都是 Object 类的子类,

* 也就是 Object 是所有的类的基类,所以将将方法定义在 Object 这个类中就会让任意对象对其调用所以 wait 方法和 notify 方法需要定义在 Object 这个类中

* 3.sleep 方法和 wait 方法的区别?

* sleep 在同步代码块或者同步函数中,不释放锁

* wait 在同步代码块或者同步函数中,释放锁

* sleep 方法必须传入参数,参数其实就是时间,时间到了自动醒来

* wait 方法可以传入参数,也可以不传入参数

* 如果给 wait 方法传入时间参数,用法与 sleep 相似,时间到就停止等待(通常用的都是没有参数的 wait 方法)

多线程(JDK1.5 的新特性互斥锁)：

* 1.同步

* 使用 ReentrantLock 类的 lock()和 unlock()方法进行同步

* 2.通信

* 使用 ReentrantLock 类的 newCondition()方法可以获取 Condition 对象

* 需要等待的时候使用 Condition 的 await()方法，唤醒的时候用 signal()方法

* 不同的线程使用不同的 Condition，这样就能区分唤醒的时候找哪个线程了

多线程(线程组的概述和使用)：

* A:线程组概述

* Java 中使用 ThreadGroup 来表示线程组，它可以对一批线程进行分类管理，Java 允许程序直接对线程组进行控制。

* 默认情况下，所有的线程都属于主线程组。

```
public final ThreadGroup getThreadGroup()//通过线程对象获取他所属于的组
public final String getName()//通过线程组对象获取他组的名字
```

* 我们也可以给线程设置分组

* 1,ThreadGroup(String name) 创建线程组对象并给其赋值名字

* 2,创建线程对象

* 3,Thread(ThreadGroup?group, Runnable?target, String?name)

* 4,设置整组的优先级或者守护线程

* B:案例演示

* 线程组的使用,默认是主线程组

```
MyRunnable mr = new MyRunnable();
        Thread t1 = new Thread(mr, "张三");
        Thread t2 = new Thread(mr, "李四");
        //获取线程组
        // 线程类里面的方法：public final ThreadGroup getThreadGroup()
        ThreadGroup tg1 = t1.getThreadGroup();
        ThreadGroup tg2 = t2.getThreadGroup();
        // 线程组里面的方法：public final String getName()
        String name1 = tg1.getName();
        String name2 = tg2.getName();
        System.out.println(name1);
        System.out.println(name2);
        // 通过结果我们知道了：线程默认情况下属于 main 线程组
        // 通过下面的测试，你应该能够看到，默任情况下，所有的线程都属于同一个组
        System.out.println(Thread.currentThread().getThreadGroup().getName());
```

* 自己设定线程组

```
// ThreadGroup(String name)
        ThreadGroup tg = new ThreadGroup("这是一个新的组");

        MyRunnable mr = new MyRunnable();
        // Thread(ThreadGroup group, Runnable target, String name)
        Thread t1 = new Thread(tg, mr, "张三");
        Thread t2 = new Thread(tg, mr, "李四");

        System.out.println(t1.getThreadGroup().getName());
        System.out.println(t2.getThreadGroup().getName());

        //通过组名称设置后台线程，表示该组的线程都是后台线程
        tg.setDaemon(true);
```

多线程(线程的五种状态)：

* 看图说话

* 新建,就绪,运行,阻塞,死亡

多线程(线程池的概述和使用)：

* A:线程池概述

* 程序启动一个新线程成本是比较高的，因为它涉及到要与操作系统进行交互。而使用线程池可以很好的提高性能，尤其是当程序中要创建大量生存期很短的线程时，更应该考虑使用线程池。线程池里的每一个线程代码结束后，并不会死亡，而是再次回到线程池中成为空闲状态，等待下一个对象来使用。在 JDK5 之前，我们必须手动实现自己的线程池，从 JDK5 开始，Java 内置支持线程池

* B:内置线程池的使用概述

* JDK5 新增了一个 Executors 工厂类来产生线程池，有如下几个方法

* public static ExecutorService newFixedThreadPool(int nThreads)

* public static ExecutorService newSingleThreadExecutor()

* 这些方法的返回值是 ExecutorService 对象，该对象表示一个线程池，可以执行 Runnable 对象或者 Callable 对象代表的线程。它提供了如下方法

* Future<?> submit(Runnable task)

* <T> Future<T> submit(Callable<T> task)

* 使用步骤：

* 创建线程池对象

* 创建 Runnable 实例

* 提交 Runnable 实例

* 关闭线程池

* C:案例演示

* 提交的是 Runnable

```
// public static ExecutorService newFixedThreadPool(int nThreads)
        ExecutorService pool = Executors.newFixedThreadPool(2);

        // 可以执行 Runnable 对象或者 Callable 对象代表的线程
        pool.submit(new MyRunnable());
        pool.submit(new MyRunnable());

        //结束线程池
        pool.shutdown();
```

多线程(多线程程序实现的方式 3)：

* 提交的是 Callable

```
// 创建线程池对象
        ExecutorService pool = Executors.newFixedThreadPool(2);

        // 可以执行 Runnable 对象或者 Callable 对象代表的线程
        Future<Integer> f1 = pool.submit(new MyCallable(100));
        Future<Integer> f2 = pool.submit(new MyCallable(200));

        // V get()
        Integer i1 = f1.get();
        Integer i2 = f2.get();

        System.out.println(i1);
        System.out.println(i2);

        // 结束
        pool.shutdown();

        public class MyCallable implements Callable<Integer> {

            private int number;

            public MyCallable(int number) {
```

```
                this.number = number;
        }

        @Override
        public Integer call() throws Exception {
                int sum = 0;
                for (int x = 1; x <= number; x++) {
                        sum += x;
                }
                return sum;
        }

}
```

* 多线程程序实现的方式 3 的好处和弊端

* 好处：

  * 可以有返回值

  * 可以抛出异常

* 弊端：

  * 代码比较复杂，所以一般不用

设计模式(简单工厂模式概述和使用)：

* A:简单工厂模式概述

  * 又叫静态工厂方法模式，它定义一个具体的工厂类负责创建一些类的实例

* B:优点

  * 客户端不需要在负责对象的创建，从而明确了各个类的职责

* C:缺点

  * 这个静态工厂类负责所有对象的创建，如果有新的对象增加，或

者某些对象的创建方式不同，就需要不断的修改工厂类，不利于后期的维护

* D:案例演示

* 动物抽象类：public abstract Animal { public abstract void eat(); }

* 具体狗类：public class Dog extends Animal {}

* 具体猫类：public class Cat extends Animal {}

* 开始，在测试类中每个具体的内容自己创建对象，但是，创建对象的工作如果比较麻烦，就需要有人专门做这个事情，所以就知道了一个专门的类来创建对象。

```
public class AnimalFactory {
                        private AnimalFactory(){}

                        //public static Dog createDog() {return new Dog();}
                        //public static Cat createCat() {return new Cat();}

                        //改进
                        public static Animal createAnimal(String animalName) {
                                if(“dog”.equals(animalName)) {}
                                else if(“cat”.equals(animale)) {

                                }else {
                                        return null;
                                }
                        }
                }
```

设计模式(工厂方法模式的概述和使用)：

* A:工厂方法模式概述

* 工厂方法模式中抽象工厂类负责定义创建对象的接口，具体对象的创建工作由继承抽象工厂的具体类实现。

* B:优点

    * 客户端不需要在负责对象的创建，从而明确了各个类的职责，如果有新的对象增加，只需要增加一个具体的类和具体的工厂类即可，不影响已有的代码，后期维护容易，增强了系统的扩展性

* C:缺点

    * 需要额外的编写代码，增加了工作量

* D:案例演示

    动物抽象类：public abstract Animal { public abstract void eat(); }

    工厂接口：public interface Factory {public abstract Animal createAnimal();}

    具体狗类：public class Dog extends Animal {}

    具体猫类：public class Cat extends Animal {}

    开始，在测试类中每个具体的内容自己创建对象，但是，创建对象的工作如果比较麻烦，就需要有人专门做这个事情，所以就知道了一个专门的类来创建对象。发现每次修改代码太麻烦，用工厂方法改进，针对每一个具体的实现提供一个具体工厂。

    狗工厂：public class DogFactory implements Factory {

        public Animal createAnimal() {...}

            }

    猫工厂：public class CatFactory implements Factory {

        public Animal createAnimal() {...}

            }

GUI(如何创建一个窗口并显示)：

* Graphical User Interface(图形用户接口)。

*

```
Frame  f = new Frame(“my window”);
f.setLayout(new FlowLayout());//设置布局管理器
f.setSize(500,400);//设置窗体大小
f.setLocation(300,200);//设置窗体出现在屏幕的位置
f.setIconImage(Toolkit.getDefaultToolkit().createImage("qq.png"));
f.setVisible(true);
```

GUI(布局管理器)：

* FlowLayout（流式布局管理器）
    * 从左到右的顺序排列。
    * Panel 默认的布局管理器。
* BorderLayout（边界布局管理器）
    * 东，南，西，北，中
    * Frame 默认的布局管理器。
* GridLayout（网格布局管理器）
    * 规则的矩阵
* CardLayout（卡片布局管理器）
    * 选项卡

* GridBagLayout（网格包布局管理器）

    * 非规则的矩阵

GUI(窗体监听)：

```
Frame f = new Frame("我的窗体");
//事件源是窗体,把监听器注册到事件源上
//事件对象传递给监听器
f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
                //退出虚拟机,关闭窗口
            System.exit(0);
        }
});
```

设计模式(适配器设计模式)：

* a.什么是适配器

    * 在使用监听器的时候，需要定义一个类事件监听器接口.

    * 通常接口中有多个方法，而程序中不一定所有的都用到，但又必须重写，这很繁琐.

    * 适配器简化了这些操作，我们定义监听器时只要继承适配器，然后重写需要的方法即可.

* b.适配器原理

    * 适配器就是一个类，实现了监听器接口，所有抽象方法都重写了，但是方

法全是空的.

* 适配器类需要定义成抽象的,因为创建该类对象,调用空方法是没有意义的

* 目的就是为了简化程序员的操作，定义监听器时继承适配器，只重写需要的方法就可以了.