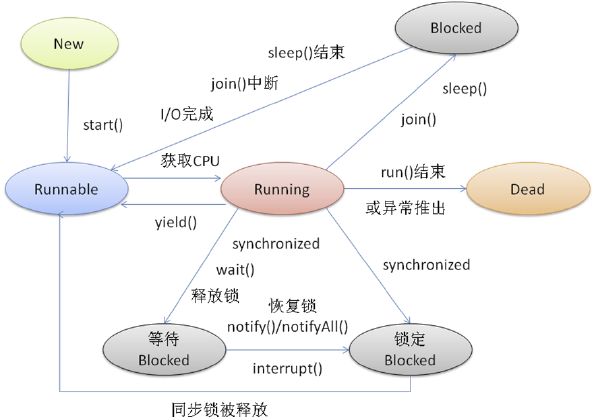
# 并行和并发

并行：多个cpu实例或者多台机器同时执行一段处理逻辑，是真正的同时。

并发：通过cpu调度算法，让用户看上去同时执行，实际上从cpu操作层面不是真正的同时。并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈，我们会用TPS或者QPS来反应这个系统的处理能力。

## 线程的状态



## Thread

线程简单实现的三种方式：

1. 继承Thread类覆盖run()方法；（java类只能继承一个父类）
2. 实现Runnable接口并实现run()方法；
3. 实现Callable接口。

### sleep

sleep让线程睡眠，交出CPU，让CPU去执行其他的任务。但是sleep方法不会释放锁，也就是说如果当前线程持有对某个对象的锁，则即使调用sleep方法，其他线程也无法访问这个对象。sleep时间满了之后线程不一定立即能得到执行，因为CPU可能在执行其他的任务。sleep方法相当于让线程进入阻塞状态。

### yield

yield方法也是让当前线程交出CPU权限，让CPU去执行其他的线程，和sleep一样不会释放对象锁。yield方法不会让线程进入阻塞状态，而是让线程重新回到就绪状态，重新等待获取CPU时间。

### join

假如在main线程中调用thread.join方法，main线程会等待thread线程执行完毕或者等待一定时间，具体看join方法的参数。

## Synchronized

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。

## 线程池

线程池可以帮我们管理线程，避免创建大量的线程增加开销。除了降低开销以外，线程池也可以提高响应速度。jvm创建对象的过程大概经过：

1. 检查对应的类是否已经被加载、解析和初始化；
2. 类加载后，为新生对象分配内存；
3. 将分配到的内存空间初始为 0；
4. 对对象进行关键信息的设置，比如对象的哈希码等；
5. 然后执行 init 方法初始化对象。

## Atomic

concurrent包中提供了一些Atomic开头的类，比如AtomicInteger，这些类主要提供一些相关的**原子操作**。

private int counter = 0;

public int **increase**(){

**sychronized**(this){

counter = counter + 1;

return counter;

}

}

private AtomicInteger counter = new **AtomicInteger**();

public int **increase**(){

return counter.**incrementAndGet**();

}

使用Atomic性能能够得到提升，而且是比较显著的提升。

## ReentranLock

## wait、notify、notifyAll

## CountDownLatch

## CyclicBarrier

## Exchanger

## Future、FutureTask

## 字节码增强

# 设计模式

## 单例模式

class **SingletonObject** {

private **SingletonObject**(){}

private static SingletonObject instance;

public static SingletonObject **getInstance**(){

if(instance == null){

synchronized (SingletonObject.class){

if(instance == null)

instance = new **SingletonObject**();

}

}

return instance;

}

}

线程安全的，而且在多线程情况下能保持高性能，相比之下下面的方式性能不高，因为synchronized对getInstance函数加锁，每次调用都加锁造成性能的降低。实际上只有在instance为null的时候才需要加锁。

class **SingletonObject** {

private **SingletonObject**(){}

private static SingletonObject instance;

public static synchronized SingletonObject **getInstance**(){

if(instance == null){

instance = new **SingletonObject**();

}

return instance;

}

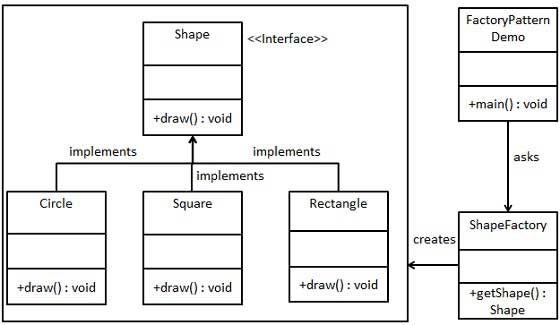
}

## 工厂模式

在工厂模式中，创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

1、日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方。

2、数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时。



getShape()方法传入想要创建对象的名称，比如"Circle"，根据传入的对象名称返回相应的对象。由于所有的子类实现的相同的接口，所以不管得到的对象是哪一个子类，接下来都可以执行相同的方法。

//Shape.java

public interface **Shape** {

void **draw**();

}

//Circle.java

public class **Circle** implements **Shape**{

@Override

public void **draw**() {

System.out.**println**("Circle");

}

}

//ShapeFactory.java

public class **ShapeFactory** {

public Shape **getShape**(String shapeType){

if(shapeType == null)

return null;

if("Circle".**equalsIgnoreCase**(shapeType))

return new **Circle**();

else if("Square".**equalsIgnoreCase**(shapeType))

return new **Square**();

return null;

}

}

//main.java

public static void **main**(String[] args) {

ShapeFactory factory = new **ShapeFactory**();

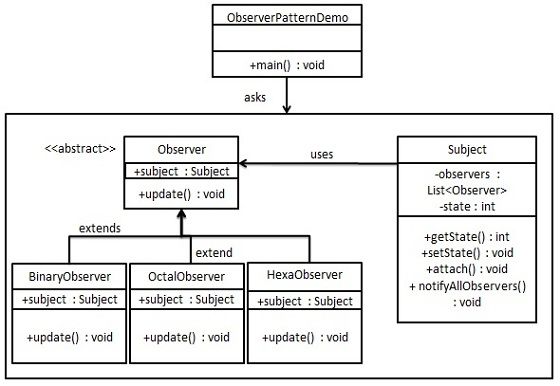
Shape shape = factory.**getShape**("circle");//Shape是interface

shape.**draw**();

}

## 观察者模式

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。



*//Observer.java*

public abstract class **Observer** {

protected Subject subject;

abstract void **update**();

}

*//Subject.java*

public class **Subject** {

private List<Observer> observers = new ArrayList<Observer>();

private int state;

public int **getState**(){

return state;

}

public void **setState**(int state){

this.state = state;

**notifyAllObservers**();

}

public void **attach**(Observer observer){

observers.**add**(observer);

}

private void **notifyAllObservers**() {

for(Observer observer : observers)

observer.**update**();

}

}

*//BinaryObserver.java*

public class **BinaryObserver** extends **Observer**{

public **BinaryObserver**(Subject subject){

this.subject = subject;

this.subject.**attach**(this);

}

@Override

void **update**() {

System.out.**println**("Binary:" + Integer.**toBinaryString**(subject.**getState**()));

}

}

*//HexaObserver.java*

public class **HexaObserver** extends **Observer**{

public **HexaObserver**(Subject subject){

this.subject = subject;

this.subject.**attach**(this);

}

@Override

void **update**() {

System.out.**println**("Hexa:" + Integer.**toBinaryString**(subject.**getState**()));

}

}

*//main.java*

public class **Main** {

public static void **main**(String[] args){

Subject subject = new **Subject**();

new **BinaryObserver**(subject);

new **HexaObserver**(subject);

subject.**setState**(10);

subject.**setState**(2);

}

}

## 代理模式、动态代理

# 参考

java中的多线程你只要看这一篇就够了

<https://www.cnblogs.com/wxd0108/p/5479442.html>

java堆、栈、堆栈的区别

<https://www.cnblogs.com/iliuyuet/p/5603618.html>

设计模式

<http://www.runoob.com/design-pattern/design-pattern-tutorial.html>

ConcurrentHashMap原理分析

<https://my.oschina.net/hosee/blog/639352>