架构设计&分布式&数据结构与算法面试题（2020最新版）

原创 ThinkWon 2020-04-30 22:20:45 48606 收藏 546 原力计划 版权

分类专栏： Java面试总结 文章标签： 架构设计 分布式 数据结构鱼算法

文章目录

- 架构设计
  - 请列举出在JDK中几个常用的设计模式?
  - 什么是设计模式？你是否在你的代码里面使用过任何设计模式?
  - 静态代理、JDK动态代理以及CGLIB动态代理
    - 静态代理
    - 动态代理
    - cglib代理
  - 单例模式
  - 工厂模式
  - 观察者模式
  - 装饰器模式
  - 秒杀系统设计
- 分布式
  - 分布式概述
    - 分布式
    - 集群
    - 微服务
    - 多线程
    - 高并发
  - 分布式系统设计理念
    - 分布式系统的目标与要素
    - 分布式系统设计两大思路：中心化和去中心化
    - 分布式与集群的区别是什么?
  - CAP定理
- CAP定理的证明
  - BASE理论
    - BASE理论的核心思想
    - BASE理论三要素
      1. 基本可用
      2. 软状态
      3. 最终一致性
- 数据结构与算法
  - 冒泡排序
  - 选择排序
  - 快速排序
  - 递归

二分查找
一致性Hash算法
概述
一致性Hash算法原理

Java面试总结汇总，整理了包括Java基础知识，集合容器，并发编程，JVM，常用开源框架Spring，MyBatis，数据库，中间件等，包含了作为一个Java工程师在面试中需要用到或者可能用到的绝大部分知识。欢迎大家阅读，本人见识有限，写的博客难免有错误或者疏忽的地方，还望各位大佬指点，在此表示感激不尽。文章持续更新中…

序号	内容	链接地址
1	Java基础知识面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104390612
2	Java集合容器面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104588551
3	Java异常面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104390689
4	并发编程面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104863992
5	JVM面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104390752
6	Spring面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104397516
7	Spring MVC面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104397427
8	Spring Boot面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104397299
9	Spring Cloud面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104397367
10	MyBatis面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/101292950
11	Redis面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/103522351
12	MySQL数据库面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104778621
13	消息中间件MQ与RabbitMQ面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104588612
14	Dubbo面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104390006
15	Linux面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104588679
16	Tomcat面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104397665
17	ZooKeeper面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104397719
18	Netty面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/104391081
19	架构设计&分布式&数据结构与算法面试题（2020最新版）	https://thinkw on.blog.csdn.net/article/details/105870730

架构设计

请列举出在JDK中几个常用的设计模式?

单例模式（Singleton pattern）用于Runtime，Calendar和其他的一些类中。工厂模式（Factory pattern）被用于各种不可变的类如Boolean，像Boolean.valueOf，观察者模式（Observer pattern）被用于 Swing 和很多的事件监听中。装饰器设计模式（Decorator design pattern）被用于多个 Java IO 类中。

什么是设计模式？你是否在你的代码里面使用过任何设计模式?

设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。设计模式是代码可用性的延伸

设计模式分类：创建型模式，结构型模式，行为型模式

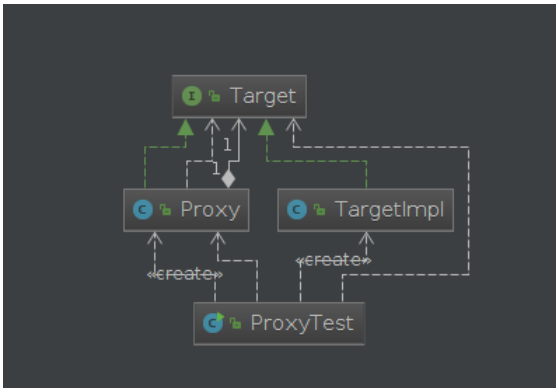
静态代理、JDK动态代理以及CGLIB动态代理

代理模式是java中最常用的设计模式之一，尤其是在spring框架中广泛应用。对于java的代理模式，一般可分为：静态代理、动态代理、以及CGLIB实现动态代理。

对于上述三种代理模式，分别进行说明。

静态代理

静态代理其实就是在程序运行之前，提前写好被代理方法的代理类，编译后运行。在程序运行之前，class已经存在。
下面我们实现一个静态代理demo:



定义一个接口Target

```
package com.test.proxy;

public interface Target {

    public String execute();
}
```

TargetImpl 实现接口Target

```
package com.test.proxy;

public class TargetImpl implements Target {

    @Override
    public String execute() {
        System.out.println("TargetImpl execute! ");
        return "execute";
    }
}
```

代理类

```
package com.test.proxy;

public class Proxy implements Target{

    private Target target;

    public Proxy(Target target) {
```

```
            this.target = target;
        }

        @Override
        public String execute() {
            System.out.println("perProcess");
            String result = this.target.execute();
            System.out.println("postProcess");
            return result;
        }
    }
```

测试类:

```
    package com.test.proxy;

    public class ProxyTest {

        public static void main(String[] args) {

            Target target = new TargetImpl();
            Proxy p = new Proxy(target);
            String result =  p.execute();
            System.out.println(result);
        }

    }
```

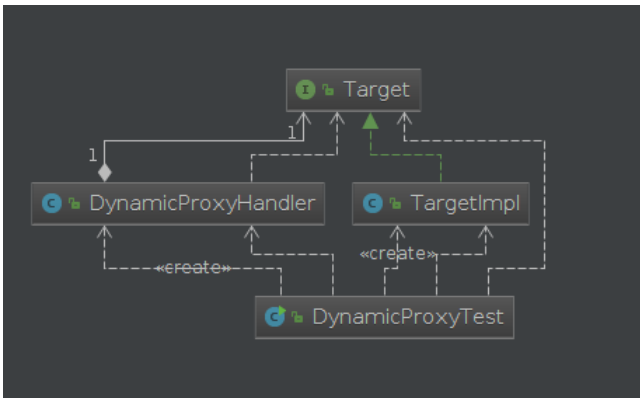
运行结果:

```
    perProcess
    TargetImpl execute!
    postProcess
    execute
```

静态代理需要针对被代理的方法提前写好代理类，如果被代理的方法非常多则需要编写很多代码，因此，对于上述缺点，通过动态代理的方式进行了弥补。

动态代理

动态代理主要是通过反射机制，在运行时动态生成所需代理的class.



接口

```
package com.test.dynamic;

public interface Target {

    public String execute();
}
```

实现类

```
package com.test.dynamic;

public class TargetImpl implements Target {

    @Override
    public String execute() {
        System.out.println("TargetImpl execute! ");
        return "execute";
    }
}
```

代理类

```
package com.test.dynamic;


import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class DynamicProxyHandler implements InvocationHandler{

    private Target target;

    public DynamicProxyHandler(Target target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("========before==========");
        Object result = method.invoke(target,args);
        System.out.println("========after===========");
        return result;
    }
}
```

测试类

```
package com.test.dynamic;

import java.lang.reflect.Proxy;

public class DynamicProxyTest {

    public static void main(String[] args) {
        Target target = new TargetImpl();
        DynamicProxyHandler handler = new DynamicProxyHandler(target);
        Target proxySubject = (Target)
Proxy.newProxyInstance(TargetImpl.class.getClassLoader(),TargetImpl.class.getInterfaces(),handler);
        String result = proxySubject.execute();
        System.out.println(result);
```

```
    }

}
```

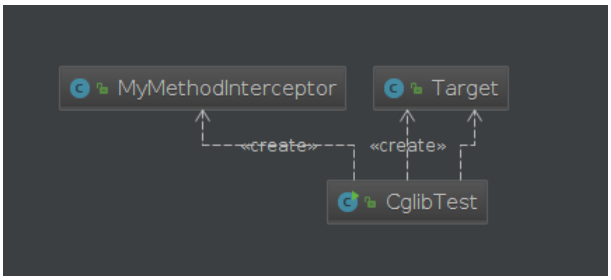
运行结果：

```
========before==========
TargetImpl execute!
========after===========
execute
```

无论是动态代理还是静态带领，都需要定义接口，然后才能实现代理功能。这同样存在局限性，因此，为了解决这个问题，出现了第三种代理方式：cglib代理。

cglib代理

CGLib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。



目标类

```
package com.test.cglib;

public class Target {

    public String execute() {
        String message = "-----------test------------";
        System.out.println(message);
        return message;
    }
}
```

通用代理类

```
package com.test.cglib;

import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class MyMethodInterceptor implements MethodInterceptor{

    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {
        System.out.println(">>>>MethodInterceptor start...");
        Object result = proxy.invokeSuper(obj,args);
        System.out.println(">>>>MethodInterceptor ending...");
```

```
            return "result";
        }
    }
```

测试类

```
    package com.test.cglib;

    import net.sf.cglib.proxy.Enhancer;

    public class CglibTest {

        public static void main(String[] args) {
            System.out.println("***************");
            Target target = new Target();
            CglibTest test = new CglibTest();
            Target proxyTarget = (Target) test.createProxy(Target.class);
            String res = proxyTarget.execute();
            System.out.println(res);
        }

        public Object createProxy(Class targetClass) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(targetClass);
            enhancer.setCallback(new MyMethodInterceptor());
            return enhancer.create();
        }

    }
```

执行结果:

```
    ***************
    >>>>MethodInterceptor start...
    -----------test------------
    >>>>MethodInterceptor ending...
    result
```

代理对象的生成过程由Enhancer类实现，大概步骤如下：

1. 生成代理类Class的二进制字节码；
2. 通过Class.forName加载二进制字节码，生成Class对象；
3. 通过反射机制获取实例构造，并初始化代理类对象。

单例模式

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

懒汉式，线程安全

代码实例：

```
    public class Singleton2 {
```

```
    private static Singleton2 instance;

    private Singleton2() {}

    public static synchronized Singleton2 getInstance() {
        if (instance == null) {
            instance = new Singleton2();
        }

        return instance;
    }

}
```

饿汉式，线程安全

代码实例:

```
public class Singleton3 {

    private static Singleton3 instance = new Singleton3();

    private Singleton3() {}

    public static Singleton3 getInstance() {
        return instance;
    }

}
```

双检锁/双重校验锁 + volatile关键字

代码实例:

```
public class Singleton7 {

    private static volatile Singleton7 instance = null;

    private Singleton7() {}

    public static Singleton7 getInstance() {
        if (instance == null) {
            synchronized (Singleton7.class) {
                if (instance == null) {
                    instance = new Singleton7();
                }
            }
        }

        return instance;
    }

}
```

工厂模式

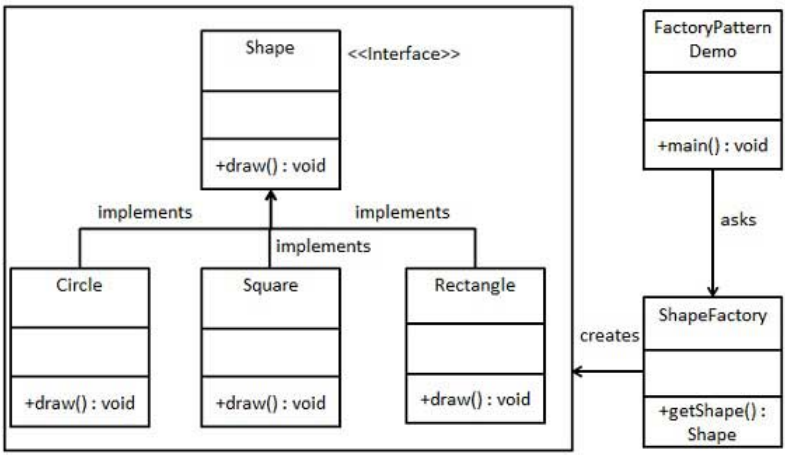
工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

主要解决：主要解决接口选择的问题。

我们将创建一个 *Shape* 接口和实现 *Shape* 接口的实体类。下一步是定义工厂类 *ShapeFactory*。

FactoryPatternDemo，我们的演示类使用 *ShapeFactory* 来获取 *Shape* 对象。它将向 *ShapeFactory* 传递信息（*CIRCLE / RECTANGLE / SQUARE*），以便获取它所需对象的类型。



步骤 1

创建一个接口。

Shape.java

```
public interface Shape {
    void draw();
}
```

步骤 2

创建实现接口的实体类。

Rectangle.java

```
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```

Square.java

```
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}
```

Circle.java

```
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}
```

步骤 3

创建一个工厂，生成基于给定信息的实体类的对象。

ShapeFactory.java

```
public class ShapeFactory {

    //使用 getShape 方法获取形状类型的对象
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }
        shapeType = shapeType.toLowerCase();

        switch (shapeType) {
            case "circle":
                return new Circle();
            case "rectangle":
                return new Rectangle();
            case "square":
                return new Square();
            default:
                return null;
        }

    }

}
```

步骤 4

使用该工厂，通过传递类型信息来获取实体类的对象。

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        //获取 Circle 的对象，并调用它的 draw 方法
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        //调用 Circle 的 draw 方法
        shape1.draw();

        //获取 Rectangle 的对象，并调用它的 draw 方法
        Shape shape2 = shapeFactory.getShape("RECTANGLE");
        //调用 Rectangle 的 draw 方法
        shape2.draw();

        //获取 Square 的对象，并调用它的 draw 方法
        Shape shape3 = shapeFactory.getShape("SQUARE");
        //调用 Square 的 draw 方法
```

```
            shape3.draw();
        }

    }
```

步骤 5

验证输出。

```
    Inside Circle::draw() method.
    Inside Rectangle::draw() method.
    Inside Square::draw() method.
```

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。观察者模式属于行为型模式。

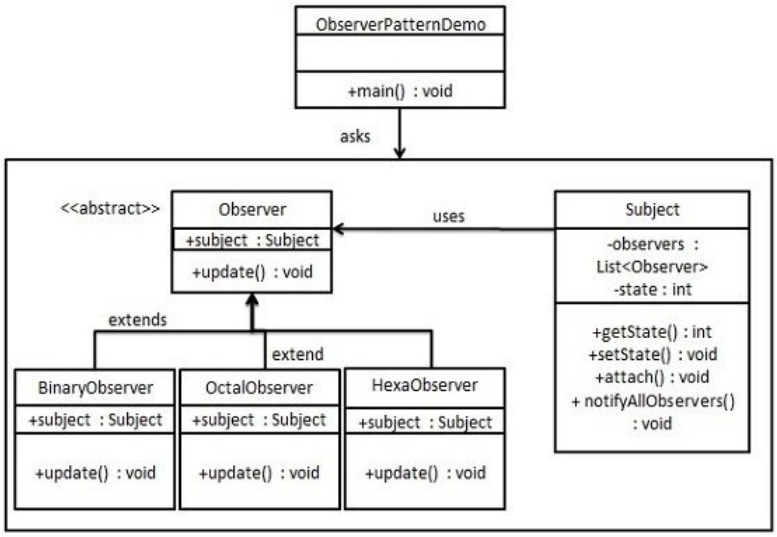
意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

实现

观察者模式使用三个类 Subject、Observer 和 Client。Subject 对象带有绑定观察者到 Client 对象和从 Client 对象解绑观察者的方法。我们创建 *Subject* 类、*Observer* 抽象类和扩展了抽象类 *Observer* 的实体类。

ObserverPatternDemo，我们的演示类使用 *Subject* 和实体类对象来演示观察者模式。



步骤 1

创建 Subject 类。

Subject.java

```
    public class Subject {

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

        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);
    }

    public void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }

}
```

步骤 2

创建 Observer 类。

Observer.java

```
public abstract class Observer {

    protected Subject subject;

    public abstract void update();

}
```

步骤 3

创建实体观察者类。

BinaryObserver.java

```
public class BinaryObserver extends Observer {

    public BinaryObserver(Subject subject) {
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println("Binary String: "
                + Integer.toBinaryString(subject.getState()));
    }

}
```

OctalObserver.java

```
public class OctalObserver extends Observer {
```

```
    public OctalObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Octal String: "
                + Integer.toOctalString( subject.getState() ) );
    }

}
```

HexaObserver.java

```
public class HexaObserver extends Observer {

    public HexaObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Hex String: "
                + Integer.toHexString( subject.getState() ).toUpperCase() );
    }

}
```

步骤 4

使用 *Subject* 和实体观察者对象。

ObserverPatternDemo.java

```
public class ObserverPatternDemo {

    public static void main(String[] args) {
        Subject subject = new Subject();

        new BinaryObserver(subject);
        new HexaObserver(subject);
        new OctalObserver(subject);

        System.out.println("First state change: 15");
        subject.setState(15);
        System.out.println();

        System.out.println("Second state change: 10");
        subject.setState(10);
    }

}
```

步骤 5

验证输出。

```
First state change: 15
Binary String: 1111
Hex String: F
Octal String: 17

Second state change: 10
Binary String: 1010
Hex String: A
Octal String: 12
```

装饰器模式

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

意图：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

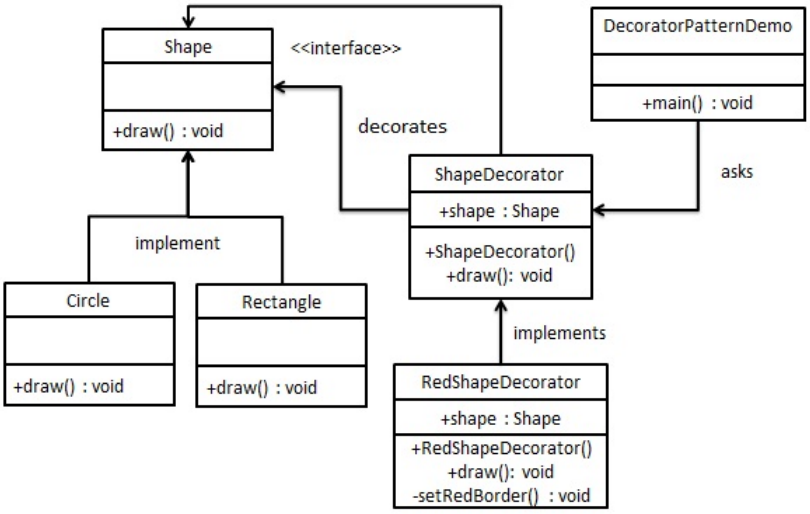
主要解决：一般的，我们为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀。

实现

我们将创建一个 *Shape* 接口和实现了 *Shape* 接口的实体类。然后我们创建一个实现了 *Shape* 接口的抽象装饰类 *ShapeDecorator*，并把 *Shape* 对象作为它的实例变量。

RedShapeDecorator 是实现了 *ShapeDecorator* 的实体类。

DecoratorPatternDemo，我们的演示类使用 *RedShapeDecorator* 来装饰 *Shape* 对象。



步骤 1

创建一个接口。

Shape.java

```
public interface Shape {

    void draw();

}
```

步骤 2

创建实现接口的实体类。

Rectangle.java

```
public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Shape: Rectangle");
    }

}
```

Circle.java

```
public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Shape: Circle");
    }

}
```

步骤 3

创建实现了 *Shape* 接口的抽象装饰类。

ShapeDecorator.java

```
public abstract class ShapeDecorator implements Shape {

    protected Shape decoratorShape;

    public ShapeDecorator(Shape decoratorShape) {
        this.decoratorShape = decoratorShape;
    }

    @Override
    public void draw() {
        decoratorShape.draw();
    }
}
```

步骤 4

创建扩展了 *ShapeDecorator* 类的实体装饰类。

RedShapeDecorator.java

```
public class RedShapeDecorator extends ShapeDecorator {

    public RedShapeDecorator(Shape decoratorShape) {
        super(decoratorShape);
    }

    @Override
    public void draw() {
```

```
            decoratorShape.draw();
            setRedBorder(decoratorShape);
        }

        private void setRedBorder(Shape decoratorShape) {
            System.out.println("Border Color: Red");
        }

    }
```

步骤 5

使用 *RedShapeDecorator* 来装饰 *Shape* 对象。

DecoratorPatternDemo.java

```
    public class DecoratorPatternDemo {

        public static void main(String[] args) {
            Shape circle = new Circle();
            Shape redCircle = new RedShapeDecorator(new Circle());
            Shape redRectangle = new RedShapeDecorator(new Rectangle());

            System.out.println("Circle with normal border");
            circle.draw();

            System.out.println("\nCircle of red border");
            redCircle.draw();

            System.out.println("\nRectangle of red border");
            redRectangle.draw();

        }

    }
```

步骤 6

验证输出。

```
    Circle with normal border
    Shape: Circle

    Circle of red border
    Shape: Circle
    Border Color: Red

    Rectangle of red border
    Shape: Rectangle
    Border Color: Red
```

秒杀系统设计

什么是秒杀

通俗一点讲就是网络商家为促销等目的组织的网上限时抢购活动

业务特点

- 高并发：秒杀的特点就是这样**时间极短**、 **瞬间用户量大。**

- 库存量少：一般秒杀活动商品量很少，这就导致了只有极少量用户能成功购买到。
- 业务简单：流程比较简单，一般都是下订单、扣库存、支付订单
- 恶意请求，数据库压力大

解决方案

前端：页面资源静态化，按钮控制，使用答题校验码可以防止秒杀器的干扰，让更多用户有机会抢到

nginx：校验恶意请求，转发请求，负载均衡；动静分离，不走tomcat获取静态资源；gzip压缩，减少静态文件传输的体积，节省带宽，提高渲染速度

业务层：集群，多台机器处理，提高并发能力

redis：集群保证高可用，持久化数据；分布式锁（悲观锁）；缓存热点数据（库存）

mq：削峰限流，MQ堆积订单，保护订单处理层的负载，Consumer根据自己的消费能力来取Task，实际上下游的压力就可控了。重点做好路由层和MQ的安全

数据库：读写分离，拆分事务提高并发度

秒杀系统设计小结

- 秒杀系统就是一个“三高”系统，即**高并发、高性能**和**高可用**的分布式系统
- 秒杀设计原则：**前台请求尽量少，后台数据尽量少，调用链路尽量短，尽量不要有单点**
- 秒杀高并发方法：**访问拦截、分流、动静分离**
- 秒杀数据方法：**减库存策略、热点、异步、限流降级**
- 访问拦截主要思路：通过CDN和缓存技术，尽量把访问拦截在离用户更近的层，尽可能地过滤掉无效请求。
- 分流主要思路：通过分布式集群技术，多台机器处理，提高并发能力。

分布式

分布式概述

分布式

分布式（distributed）是为了解决单个物理服务器容量和性能瓶颈问题而采用的优化手段，将一个业务拆分成不同的子业务，分布在不同的机器上执行。服务之间通过远程调用协同工作，对外提供服务。

该领域需要解决的问题极多，在不同的技术层面上，又包括：分布式缓存、分布式数据库、分布式计算、分布式文件系统等，一些技术如MQ、Redis、zookeeper等都跟分布式有关。

从理念上讲，分布式的实现有两种形式：

水平扩展：当一台机器扛不住流量时，就通过添加机器的方式，将流量平分到所有服务器上，所有机器都可以提供 相同的服务；

垂直拆分：前端有多种查询需求时，一台机器扛不住，可以将不同的业务需求分发到不同的机器上，比如A机器处理余票查询的请求，B机器处理支付的请求。

集群

集群（cluster）是指在多台不同的服务器中部署相同应用或服务模块，构成一个集群，通过负载均衡设备对外提供服务。

两个特点

可扩展性：集群中的服务节点，可以动态的添加机器，从而增加集群的处理能力。

高可用性：如果集群某个节点发生故障，这台节点上面运行的服务，可以被其他服务节点接管，从而增强集群的高可用性。

两大能力

负载均衡：负载均衡能把任务比较均衡地分布到集群环境下的计算和网络资源。

集群容错：当我们的系统中用到集群环境，因为各种原因在集群调用失败时，集群容错起到关键性的作用。

微服务

微服务就是很小的服务，小到一个服务只对应一个单一的功能，只做一件事。这个服务可以单独部署运行，服务之间通过远程调用协同工作，每个微服务都是由独立的小团队开发，测试，部署，上线，负责它的整个生命周期。

多线程

多线程（multi-thread）：多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务。多线程是为了提高CPU的利用率。

高并发

高并发（High Concurrency）是一种系统运行过程中发生了一种“短时间内遇到大量请求”的情况，高并发对应的是访问请求，多线程是解决高并发的方法之一，高并发还可以通过分布式，集群，算法优化，数据库优化等方法解决。

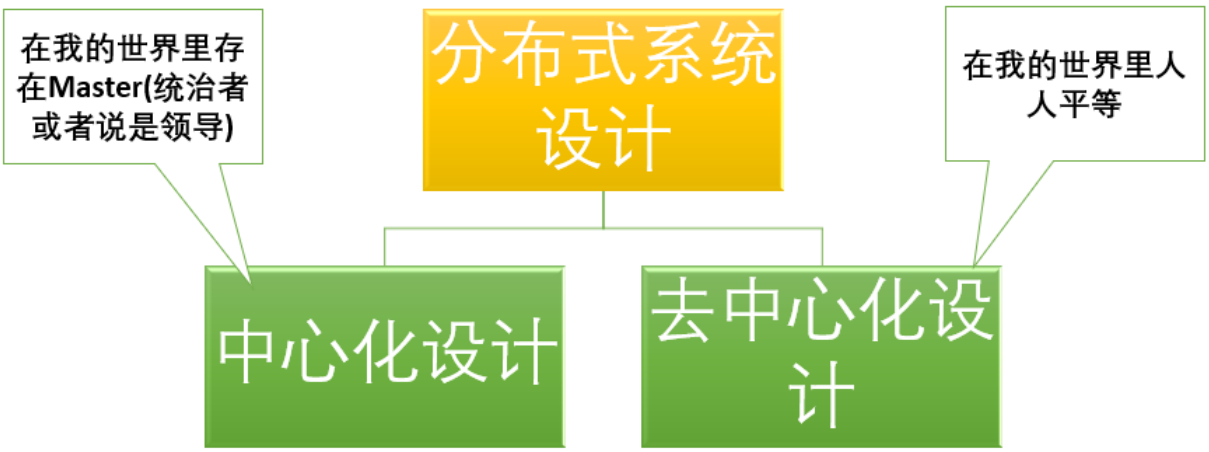
分布式系统设计理念

分布式系统的目标与要素

分布式系统的目标是提升系统的整体性能和吞吐量另外还要尽量保证分布式系统的容错性（假如增加10台服务器才达到单机运行效果2倍左右的性能，那么这个分布式系统就根本没有存在的意义）。

即使采用了分布式系统，我们也要尽力运用并发编程、高性能网络框架等等手段提升单机上的程序性能。

分布式系统设计两大思路：中心化和去中心化



中心化设计

- **两个角色：** 中心化的设计思想很简单，分布式集群中的节点机器按照角色分工，大体上分为两种角色：**“领导”**和**“干活的”**
- **角色职责：** “领导”通常负责分发任务并监督“干活的”，发现谁太闲了，就想发设法地给其安排新任务，确保没有一

个“干活的”能够偷懒，如果“领导”发现某个“干活的”因为劳累过度而病倒了，则是不会考虑先尝试“医治”他的，而是一脚踢出去，然后把他的任务分给其他人。其中微服务架构 **Kubernetes** 就恰好采用了这一设计思路。

- 中心化设计的问题
    1. 中心化的设计存在的最大问题是“领导”的安危问题，如果“领导”出了问题，则群龙无首，整个集群就奔溃了。但我们难以同时安排两个“领导”以避免单点问题。
    2. 中心化设计还存在另外一个潜在的问题，既“领导”的能力问题：可以领导10个人高效工作并不意味着可以领导100个人高效工作，所以如果系统设计和实现得不好，问题就会卡在“领导”身上。

- **领导安危问题的解决办法：** 大多数中心化系统都采用了主备两个“领导”的设计方案，可以是热备或者冷备，也可以是自动切换或者手动切换，而且越来越多的新系统都开始具备自动选举切换“领导”的能力，以提升系统的可用性。

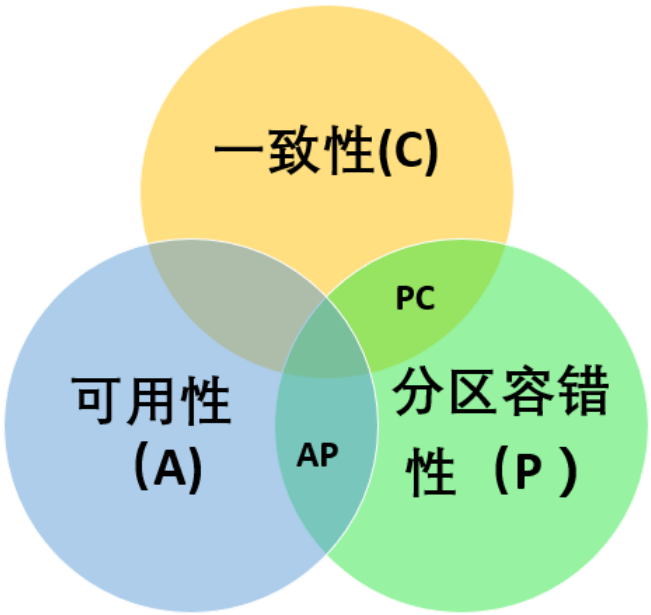
去中心化设计

- **众生地位平等：** 在去中心化的设计里，通常没有“领导”和“干活的”这两种角色的区分，大家的角色都是一样的，地位是平等的，全球互联网就是一个典型的去中心化的分布式系统，联网的任意节点设备宕机，都只会影响很小范围的功能。
- **“去中心化”不是不要中心，而是由节点来自由选择中心。** （集群的成员会自发的举行“会议”选举新的“领导”主持工作。最典型的案例就是ZooKeeper及Go语言实现的Etcd）
- **去中心化设计的问题：** 去中心化设计里最难解决的一个问题是 **“脑裂”问题** ，这种情况的发生概率很低，但影响很大。脑裂指一个集群由于网络的故障，被分为至少两个彼此无法通信的单独集群，此时如果两个集群都各自工作，则可能会产生严重的数据冲突和错误。一般的设计思路是，当集群判断发生了脑裂问题时，规模较小的集群就“自杀”或者拒绝服务。

分布式与集群的区别是什么?

- **分布式：** 一个业务分拆多个子业务，部署在不同的服务器上
- **集群：** 同一个业务，部署在多个服务器上。比如之前做电商网站搭的redis集群以及solr集群都是属于将redis服务器提供的缓存服务以及solr服务器提供的搜索服务部署在多个服务器上以提高系统性能、并发量解决海量存储问题。

CAP定理



在理论计算机科学中，CAP定理（CAP theorem），又被称作布鲁尔定理（Brewer's theorem），它指出对于一个分布式计算系统来说，不可能同时满足以下三点：

选项	描述
Consistency（一致性）	指数据在多个副本之间能够保持一致的特性（严格的一致性）
Availability（可用性）	指系统提供的服务必须一直处于可用的状态，每次请求都能获取到非错的响应（不保证获取的数据为最新数据）
Partition tolerance（分区容错性）	分布式系统在遇到任何网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务，除非整个网络环境都发生了故障

Spring Cloud在CAP法则上主要满足的是A和P法则，Dubbo和Zookeeper在CAP法则主要满足的是C和P法则

CAP仅适用于原子读写的NOSQL场景中，并不适合数据库系统。现在的分布式系统具有更多特性比如扩展性、可用性等等，在进行系统设计和开发时，我们不应该仅仅局限在CAP问题上。

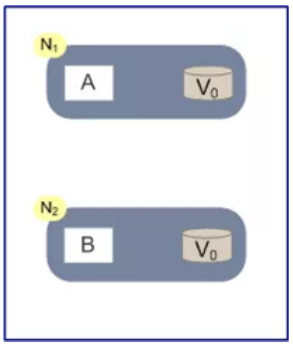
注意：不是所谓的3选2（不要被网上大多数文章误导了）

现实生活中，大部分人解释这一定律时，常常简单的表述为："一致性、可用性、分区容忍性三者你只能同时达到其中两个，不可能同时达到"。实际上这是一个非常具有误导性质的说法，而且在CAP理论诞生12年之后，CAP之父也在2012年重写了之前的论文。

当发生网络分区的时候，如果我们要继续服务，那么强一致性和可用性只能2选1。也就是说当网络分区之后P是前提，决定了P之后才有C和A的选择。也就是说分区容错性（Partition tolerance）我们是必须要实现的。

CAP定理的证明

关于CAP这三个特性我们就介绍完了，接下来我们试着证明一下**为什么CAP不能同时满足。**

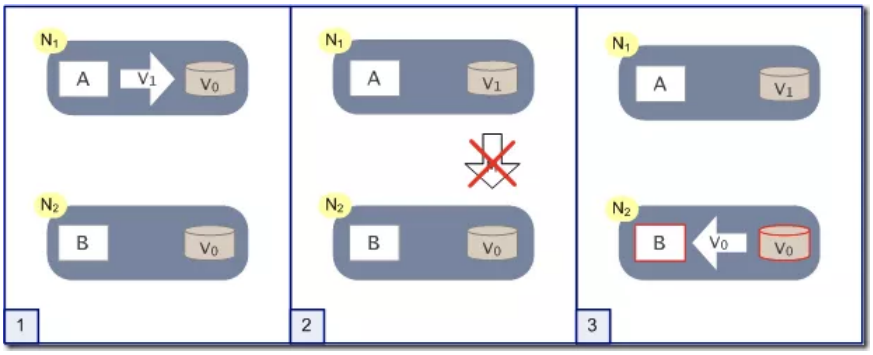


为了简化证明的过程，我们假设整个集群里只有两个N1和N2两个节点，如下图：

N1和N2当中各自有一个应用程序AB和数据库，当系统满足一致性的时候，我们认为N1和N2数据库中的数据保持一致。在满足可用性的时候，我们认为无论用户访问N1还是N2，都可以获得正确的结果，在满足分区容错性的时候，我们认为无论N1还是N2宕机或者是两者的通信中断，都不影响系统的运行。

我们假设一种极端情况，假设某个时刻N1和N2之间的网络通信突然中断了。如果系统**满足分区容错性**，那么显然可以支持这种异常。问题是在此前提下，一致性和可用性是否可以做到不受影响呢？

我们做个假象实验，如下图，突然某一时刻N1和N2之间的关联断开：



有用户向N1发送了请求更改了数据，将数据库从V0更新成了V1。由于网络断开，所以N2数据库依然是V0，如果这个时候有一个请求发给了N2，但是N2并没有办法可以直接给出最新的结果V1，这个时候该怎么办呢？

这个时候无法两种方法，**一种是将错就错，将错误的V0数据返回给用户。第二种是阻塞等待，等待网络通信恢复，N2中的数据更新之后再返回给用户**。显然前者牺牲了一致性，后者牺牲了可用性。

这个例子虽然简单，但是说明的内容却很重要。在分布式系统当中，CAP三个特性我们是无法同时满足的，必然要舍弃一个。三者舍弃一个，显然排列组合一共有三种可能。

BASE理论

BASE理论由eBay架构师Dan Pritchett提出，在2008年上被分表为论文，并且eBay给出了他们在实践中总结的基于BASE理论的一套新的分布式事务解决方案。

BASE 是 **Basically Available（基本可用）** 、**Soft-state（软状态）** 和 **Eventually Consistent（最终一致性）** 三个短语的缩写。BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结，是基于CAP定理逐步演化而来的，它大大降低了我们对系统的要求。

BASE理论的核心思想

即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。也就是牺牲数据的一致性来满

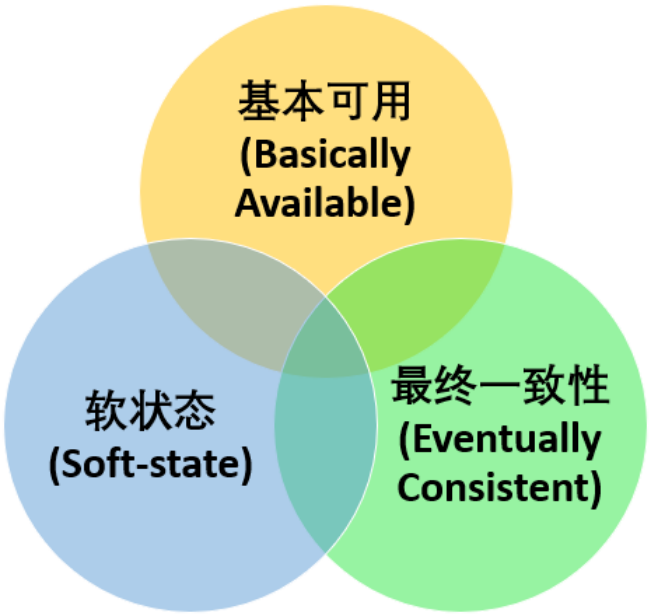
足系统的高可用性，系统中一部分数据不可用或者不一致时，仍需要保持系统整体“主要可用”。

针对数据库领域，BASE思想的主要实现是对业务数据进行拆分，让不同的数据分布在不同的机器上，以提升系统的可用性，当前主要有以下两种做法：

- 按功能划分数据库
- 分片（如开源的Mycat、Amoeba等）。

由于拆分后会涉及分布式事务问题，所以eBay在该BASE论文中提到了如何用最终一致性的思路来实现高性能的分布式事务。

BASE理论三要素



1. 基本可用

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性。但是，这绝不等价于系统不可用。

比如：

- **响应时间上的损失**：正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障，查询结果的响应时间增加了1~2秒
- **系统功能上的损失**：正常情况下，在一个电子商务网站上进行购物的时候，消费者几乎能够顺利完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面

2. 软状态

软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时

3. 最终一致性

最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

数据结构与算法

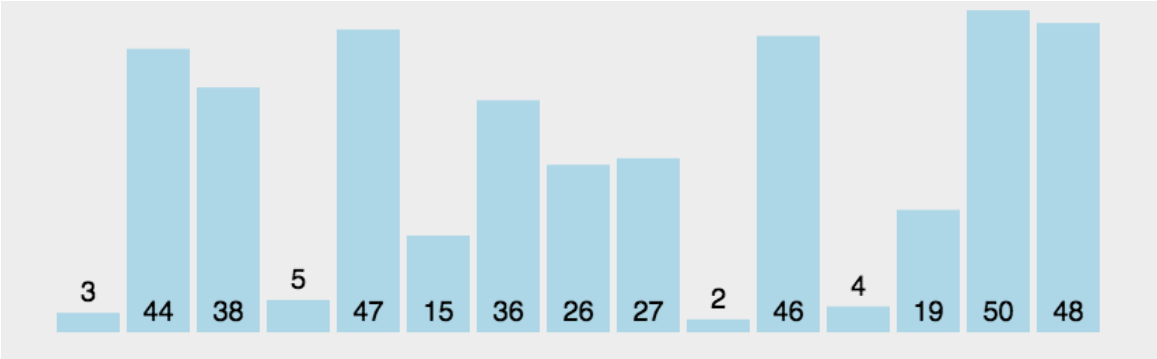
冒泡排序

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，依次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

算法描述

- 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
- 针对所有的元素重复以上的步骤，除了最后一个；
- 重复步骤1~3，直到排序完成。

动图演示



代码实现

下面的排序算法统一使用的测试代码如下，源码GitHub链接

```
public static void main(String[] args) {
    int[] array = {3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48};
    // 只需要修改成对应的方法名就可以了
    bubbleSort(array);

    System.out.println(Arrays.toString(array));
}


/**
 * Description:冒泡排序
 *
 * @param array 需要排序的数组
 * @author JourWon
 * @date 2019/7/11 9:54
 */
public static void bubbleSort(int[] array) {
    if (array == null || array.length <= 1) {
        return;
    }

    int length = array.length;
```

```
        // 外层循环控制比较轮数i
        for (int i = 0; i < length; i++) {
            // 内层循环控制每一轮比较次数，每进行一轮排序都会找出一个较大值
            // (array.length - 1)防止索引越界，(array.length - 1 - i)减少比较次数
            for (int j = 0; j < length - 1 - i; j++) {
                // 前面的数大于后面的数就进行交换
                if (array[j] > array[j + 1]) {
                    int temp = array[j + 1];
                    array[j + 1] = array[j];
                    array[j] = temp;
                }
            }
        }

    }
```

算法分析

最佳情况：T(n) = O(n) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)

选择排序

表现**最稳定的排序算法之一**，因为**无论什么数据进去都是O(n2)的时间复杂度**，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。理论上讲，选择排序可能也是平时排序一般人想到的最多的排序方法了吧。

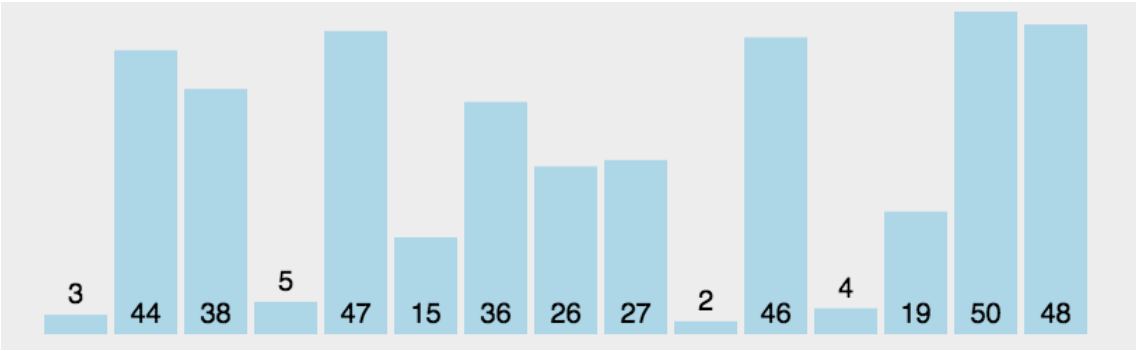
选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

算法描述

n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。具体算法描述如下：

- 初始状态：无序区为R[1…n]，有序区为空；
- 第i趟排序(i=1,2,3…n-1)开始时，当前有序区和无序区分别为R[1…i-1]和R(i…n）。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1…i]和R[i+1…n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
- n-1趟结束，数组有序化了。

动图演示



代码实现

下面的排序算法统一使用的测试代码如下，源码GitHub链接

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

```
        int[] array = {3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48};
        // 只需要修改成对应的方法名就可以了
        selectionSort(array);

        System.out.println(Arrays.toString(array));
    }


    /**
     * Description: 选择排序
     *
     * @param array
     * @return void
     * @author JourWon
     * @date 2019/7/11 23:31
     */
    public static void selectionSort(int[] array) {
        if (array == null || array.length <= 1) {
            return;
        }

        int length = array.length;

        for (int i = 0; i < length - 1; i++) {
            // 保存最小数的索引
            int minIndex = i;

            for (int j = i + 1; j < length; j++) {
                // 找到最小的数
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                }
            }

            // 交换元素位置
            if (i != minIndex) {
                swap(array, minIndex, i);
            }
        }

    }

    /**
     * Description: 交换元素位置
     *
     * @param array
     * @param a
     * @param b
     * @return void
     * @author JourWon
     * @date 2019/7/11 17:57
     */
    private static void swap(int[] array, int a, int b) {
        int temp = array[a];
        array[a] = array[b];
        array[b] = temp;
    }
```

算法分析

最佳情况：T(n) = O(n2) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)

快速排序

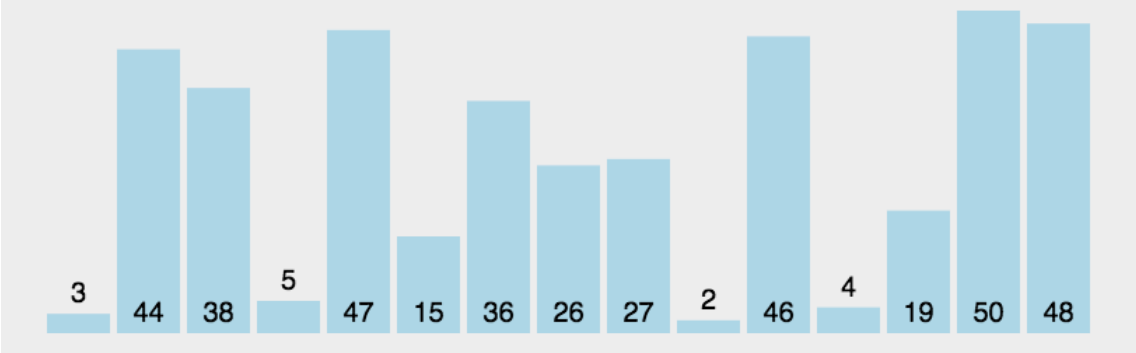
快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

算法描述

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

- 从数列中挑出一个元素，称为 “基准”（**pivot**）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
- 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

动图演示



代码实现

下面的排序算法统一使用的测试代码如下，源码GitHub链接

```
public static void main(String[] args) {
    int[] array = {3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48};
    // 只需要修改成对应的方法名就可以了
    quickSort(array);

    System.out.println(Arrays.toString(array));
}


/**
 * Description: 快速排序
 *
 * @param array
 * @return void
 * @author JourWon
 * @date 2019/7/11 23:39
 */
public static void quickSort(int[] array) {
    quickSort(array, 0, array.length - 1);
}


private static void quickSort(int[] array, int left, int right) {
    if (array == null || left >= right || array.length <= 1) {
        return;
```

```
        }
        int mid = partition(array, left, right);
        quickSort(array, left, mid);
        quickSort(array, mid + 1, right);
    }


    private static int partition(int[] array, int left, int right) {
        int temp = array[left];
        while (right > left) {
            // 先判断基准数和后面的数依次比较
            while (temp <= array[right] && left < right) {
                --right;
            }
            // 当基准数大于了 arr[left]，则填坑
            if (left < right) {
                array[left] = array[right];
                ++left;
            }
            // 现在是 arr[right] 需要填坑了
            while (temp >= array[left] && left < right) {
                ++left;
            }
            if (left < right) {
                array[right] = array[left];
                --right;
            }
        }
        array[left] = temp;
        return left;
    }
```

算法分析

最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(nlogn)

递归

什么叫递归

递归函数就是直接或间接调用自身的函数，也就是自身调用自己。

一般什么时候使用递归?

递归是常用的编程技术，其基本思想就是“自己调用自己”，一个使用递归技术的方法即是直接或间接的调用自身的方法。递归方法实际上体现了“以此类推”、“用同样的步骤重复”这样的思想。

还有些数据结构如二叉树，结构本身固有递归特性；此外，有一类问题，其本身没有明显的递归结构，但用递归程序求解比其他方法更容易编写程序。

需满足的两个条件

1. 有反复执行的过程(调用自身)
2. 有跳出反复执行过程的条件(递归出口)

经典问题：阶乘

```
    递归阶乘n! = n * (n-1) * (n-2) * ...* 1(n>0)


    public static Integer recursionMulity(Integer n) {
```

```
        if (n == 1) {
            return 1;
        }
        return n * recursionMulity(n - 1);
    }
```

经典问题：不死神兔（斐波那契数列）

3个月起每个月都生一对兔子，小兔子长到第三个月后每个月又生一对兔子，假如兔子都不死，问每个月的兔子总数为多少?

分析：首先我们要明白题目的意思指的是每个月的兔子总对数；假设将兔子分为小中大三种，兔子从出生后三个月后每个月就会生出一对兔子，

那么我们假定第一个月的兔子为小兔子，第二个月为中兔子，第三个月之后就为大兔子，那么第一个月分别有1、0、0，第二个月分别为0、1、0，

第三个月分别为1、0、1，第四个月分别为,1、1、1，第五个月分别为2、1、2，第六个月分别为3、2、3，第七个月分别为5、3、5……

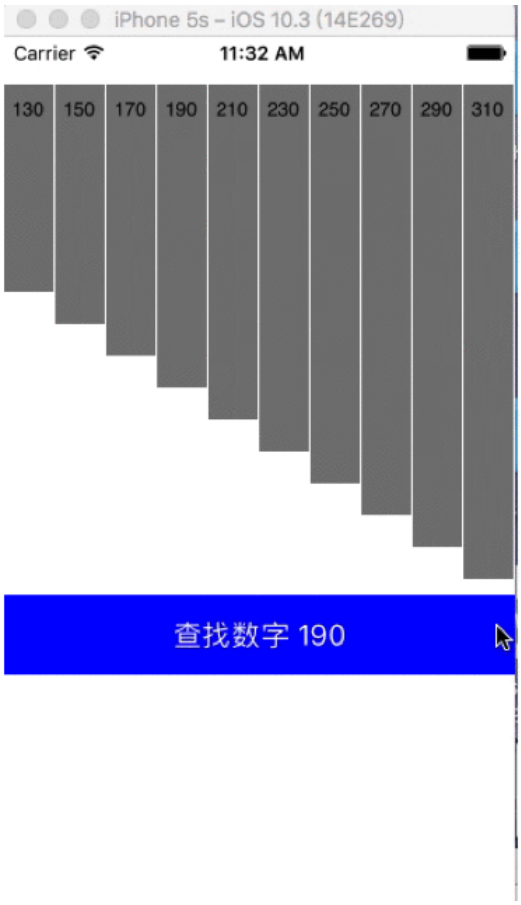
兔子总数分别为：1、1、2、3、5、8、13……

于是得出了一个规律，从第三个月起，后面的兔子总数都等于前面两个月的兔子总数之和，即为斐波那契数列。

```
    public static int fib(int mon) {
        if (mon < 2) {
            return 1;
        } else {
            return fib(mon - 1) + fib(mon - 2);
        }
    }
```

二分查找

在数组[130,150,170,190,210,230,250,270,290,310]中查找数字190，红色为二分线(折半线)，灰色为查找区域，黑色为排除区域。



二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法，前提是**数据结构必须先排好序**，时间复杂度可以表示O(h)=O(log2n)，以2为底，n的对数。其缺点是要求待查表为有序表，且插入删除困难。

左加右不加，找右缩左，找左缩右

```
public class BinarySearch {
    public static void main(String[] args) {
        int[] arr = {5, 12, 23, 43, 66, 98, 100};
        System.out.println(binarySort(arr, 23));
    }

    /**
     * 循环实现二分查找
     *
     * @param arr
     * @param key
     * @return
     */
    public static int binarySearch(int[] arr, int key) {
        //第一个下标
        int low = 0;
        //最后一个下标
        int high = arr.length - 1;
        int mid = 0;
        //防越界
        if (key < arr[low] || key > arr[high] || low > high) {
            return -1;
        }
        while (low <= high) {
            mid = (low + high) >>> 1;
```

```
                if (key < arr[mid]) {
                    high = mid - 1;
                } else if (key > arr[mid]) {
                    low = mid + 1;
                } else {
                    return mid;
                }
            }
            return -1;
        }
    }
```

二分查找中中间值的计算

这是一个经典的话题，如何计算二分查找中的中值？大家一般给出了两种计算方法：

- 算法一： `mid = (low + high) / 2`
- 算法二： `mid = low + (high – low)/2`

乍看起来，算法一简洁，算法二提取之后，跟算法一没有什么区别。但是实际上，区别是存在的。算法一的做法，在极端情况下，(low + high)存在着溢出的风险，进而得到错误的mid结果，导致程序错误。而算法二能够保证计算出来的mid，一定大于low，小于high，不存在溢出的问题。

一致性Hash算法

概述

一致性Hash是一种特殊的Hash算法，由于其均衡性、持久性的映射特点，被广泛的应用于负载均衡领域和分布式存储，如nginx和memcached都采用了一致性Hash来作为集群负载均衡的方案。

普通的Hash函数最大的作用是**散列**，或者说是将一系列在形式上具有相似性质的数据，打散成随机的、均匀分布的数据。**不难发现，这样的Hash只要集群的数量N发生变化，之前的所有Hash映射就会全部失效。**如果集群中的每个机器提供的服务没有差别，倒不会产生什么影响，但对于分布式缓存这样的系统而言，映射全部失效就意味着之前的缓存全部失效，后果将会是灾难性的。**一致性Hash通过构建环状的Hash空间**代替线性Hash空间的方法解决了这个问题。

良好的分布式cahce系统中的一致性hash算法应该满足以下几个方面：

- **平衡性(Balance)**

平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。

- **单调性(Monotonicity)**

单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲区加入到系统中，那么哈希的结果应能够保证原有已分配的内容可以被映射到新的缓冲区中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

- **分散性(Spread)**

在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。

- **负载(Load)**

负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区

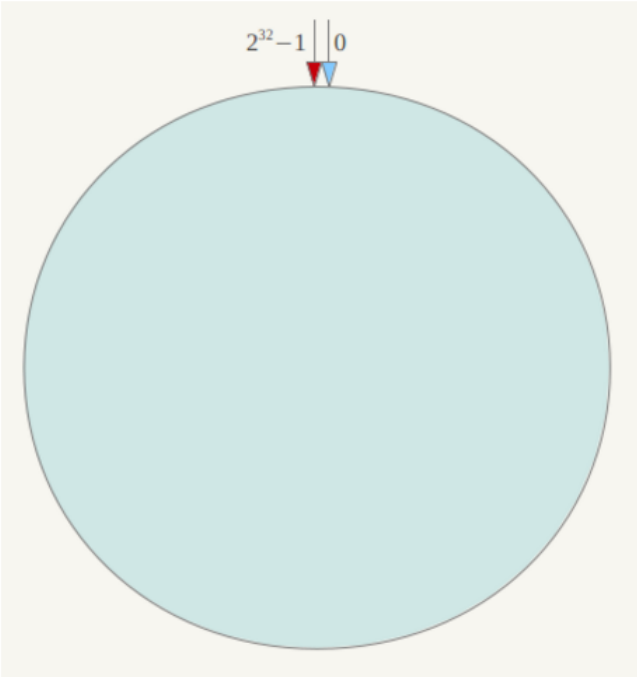
而言，也可能被不同的用户映射为不同的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

- **平滑性(Smoothness)**

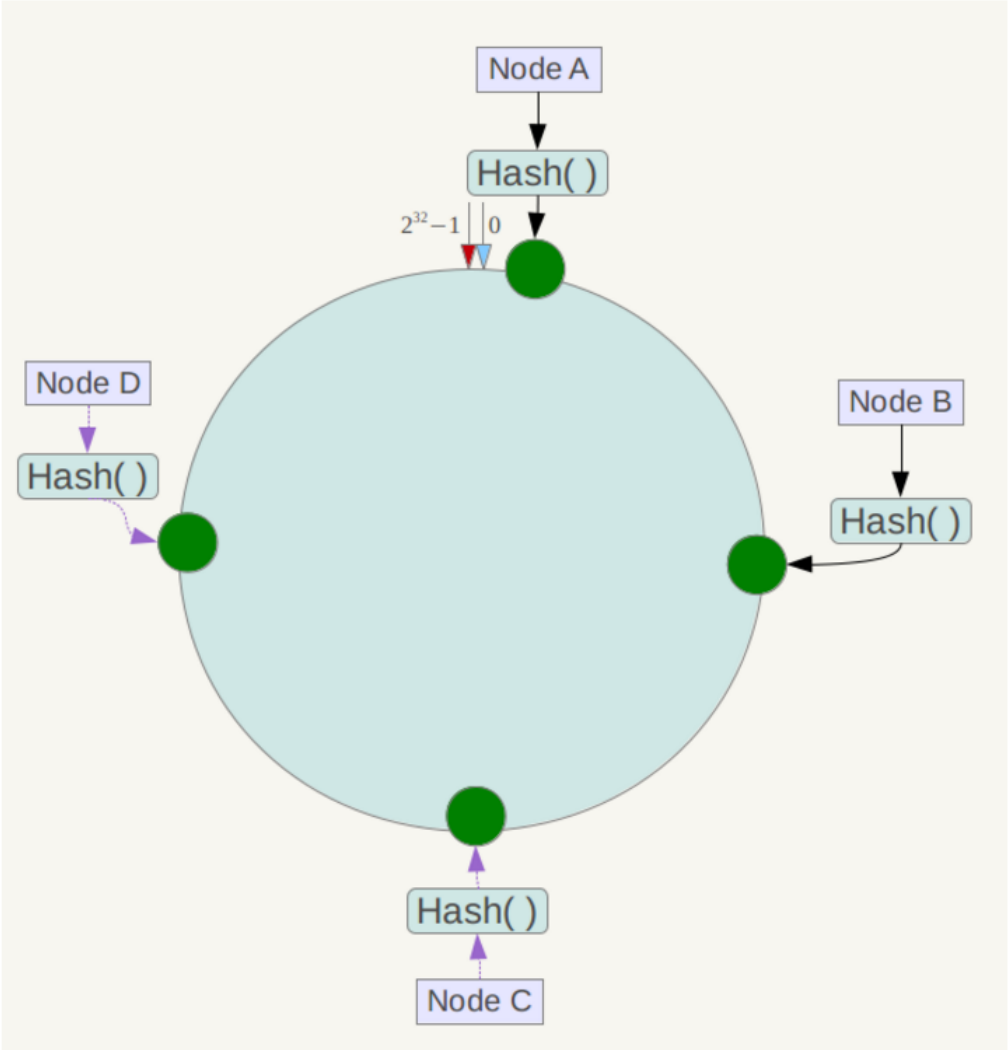
平滑性是指缓存服务器的数目平滑改变和缓存对象的平滑改变是一致的。

一致性Hash算法原理

简单来说，一致性哈希将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H的值空间为0-2^{32-1}（即哈希值是一个32位无符号整形），整个哈希空间环如下：整个空间按顺时针方向组织。0和2^{32-1}在零点中方向重合。

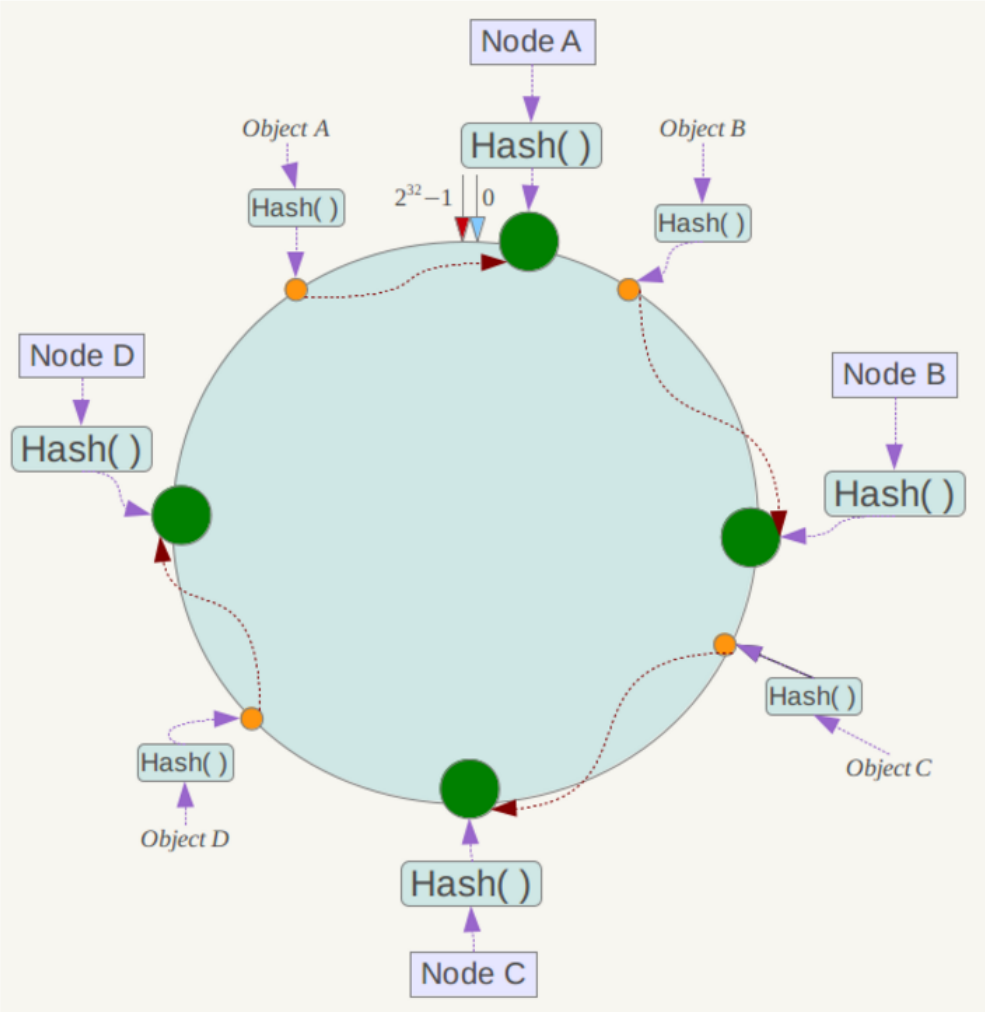


下一步将各个服务器使用Hash进行一次哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中四台服务器使用ip地址哈希后在环空间的位置如下：



接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器。

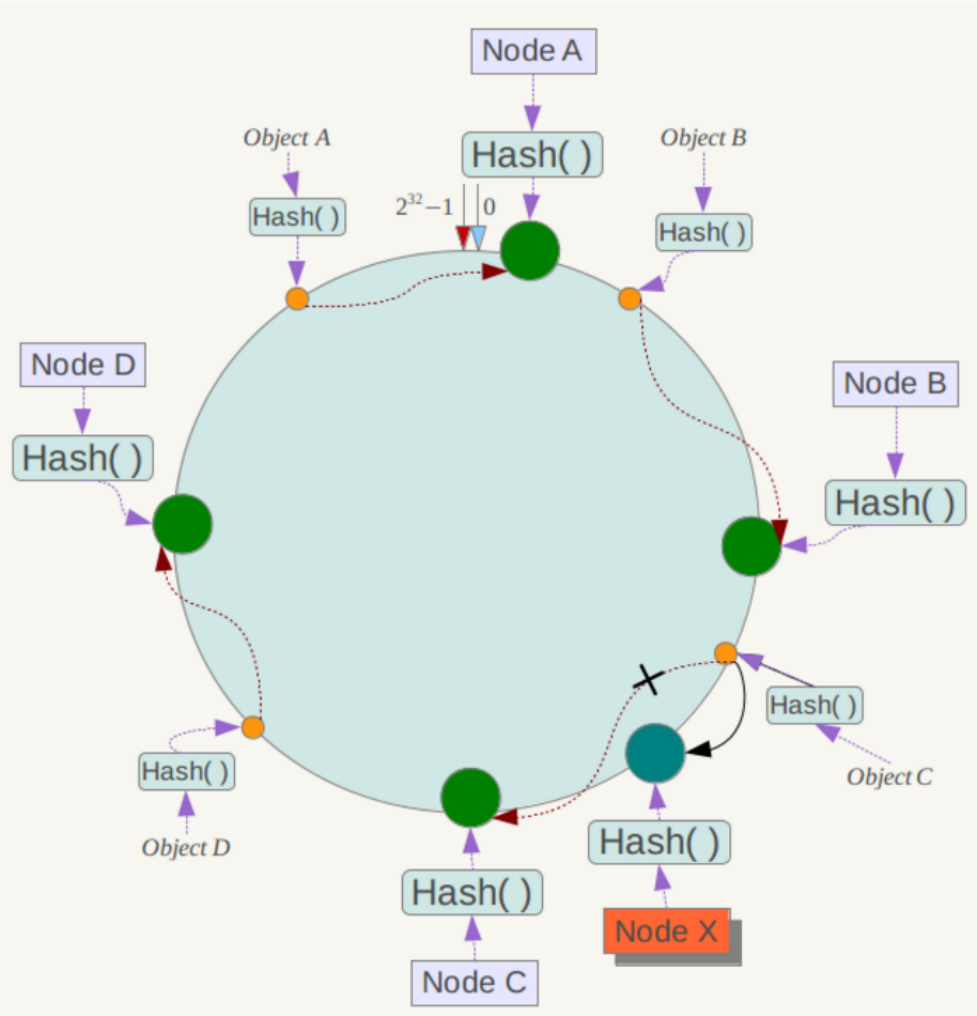
例如我们有Object A、Object B、Object C、Object D四个数据对象，经过哈希计算后，在环空间上的位置如下：



根据一致性哈希算法，数据A会被定为到Node A上，B被定为到Node B上，C被定为到Node C上，D被定为到Node D上。

下面分析一致性哈希算法的容错性和可扩展性。现假设Node C不幸宕机，可以看到此时对象A、B、D不会受到影响，只有C对象被重定位到Node D。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

下面考虑另外一种情况，如果在系统中增加一台服务器Node X，如下图所示：



此时对象Object A、B、D不受影响，只有对象C需要重定位到新的Node X。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。

综上所述，一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

Java代码实现

```
public class ConsistentHash<T> {

    /**
     * 节点的复制因子,实际节点个数 * numberOfReplicas = 虚拟节点个数
     */
    private final int numberOfReplicas;
    /**
     * 存储虚拟节点的hash值到真实节点的映射
     */
    private final SortedMap<Integer, T> circle = new TreeMap<Integer, T>();

    public ConsistentHash(int numberOfReplicas, Collection<T> nodes) {
        this.numberOfReplicas = numberOfReplicas;
        for (T node : nodes) {
            add(node);
        }
```

```
}

public void add(T node) {
    for (int i = 0; i < numberOfReplicas; i++) {
        // 对于一个实际机器节点 node, 对应 numberOfReplicas 个虚拟节点
        /*
         * 不同的虚拟节点(i不同)有不同的hash值,但都对应同一个实际机器node
         * 虚拟node一般是均衡分布在环上的,数据存储在顺时针方向的虚拟node上
         */
        String nodestr = node.toString() + i;
        int hashcode = nodestr.hashCode();
        System.out.println("hashcode:" + hashcode);
        circle.put(hashcode, node);

    }
}

public void remove(T node) {
    for (int i = 0; i < numberOfReplicas; i++) {
        circle.remove((node.toString() + i).hashCode());
    }
}


/**
 * 获得一个最近的顺时针节点,根据给定的key 取Hash
 * 然后再取得顺时针方向上最近的一个虚拟节点对应的实际节点
 * 再从实际节点中取得 数据
 *
 * @param key
 * @return
 */
public T get(Object key) {
    if (circle.isEmpty()) {
        return null;
    }
    // node 用String来表示,获得node在哈希环中的hashCode
    int hash = key.hashCode();
    System.out.println("hashcode----->:" + hash);
    //数据映射在两台虚拟机器所在环之间,就需要按顺时针方向寻找机器
    if (!circle.containsKey(hash)) {
        SortedMap<Integer, T> tailMap = circle.tailMap(hash);
        hash = tailMap.isEmpty() ? circle.firstKey() : tailMap.firstKey();
    }
    return circle.get(hash);
}

public long getSize() {
    return circle.size();
}

/**
 * 查看表示整个哈希环中各个虚拟节点位置
 */
public void testBalance() {
    //获得TreeMap中所有的Key
    Set<Integer> sets = circle.keySet();
    //将获得的Key集合排序
    SortedSet<Integer> sortedSets = new TreeSet<Integer>(sets);
    for (Integer hashCode : sortedSets) {
        System.out.println(hashCode);
    }
```

```
        System.out.println("----each location 's distance are follows: ----");
        /*
         * 查看相邻两个hashCode的差值
         */
        Iterator<Integer> it = sortedSets.iterator();
        Iterator<Integer> it2 = sortedSets.iterator();
        if (it2.hasNext()) {
            it2.next();
        }
        long keyPre, keyAfter;
        while (it.hasNext() && it2.hasNext()) {
            keyPre = it.next();
            keyAfter = it2.next();
            System.out.println(keyAfter - keyPre);
        }
    }

    public static void main(String[] args) {
        Set<String> nodes = new HashSet<String>();
        nodes.add("A");
        nodes.add("B");
        nodes.add("C");

        ConsistentHash<String> consistentHash = new ConsistentHash<String>(2, nodes);
        consistentHash.add("D");

        System.out.println("hash circle size: " + consistentHash.getSize());
        System.out.println("location of each node are follows: ");
        consistentHash.testBalance();

        String node = consistentHash.get("apple");
        System.out.println("node----------->:" + node);
    }

}
```