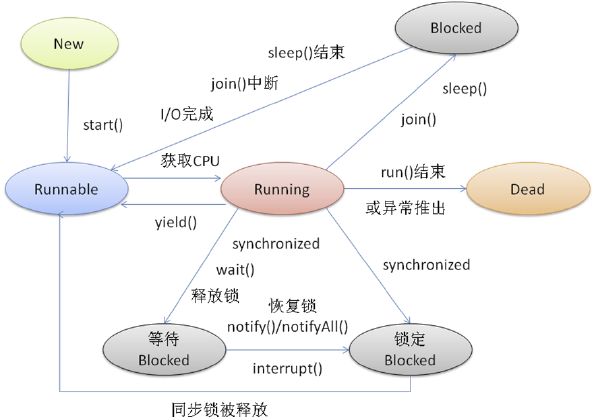
# 并行和并发

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；

### 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

# 垃圾回收

# 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;

}

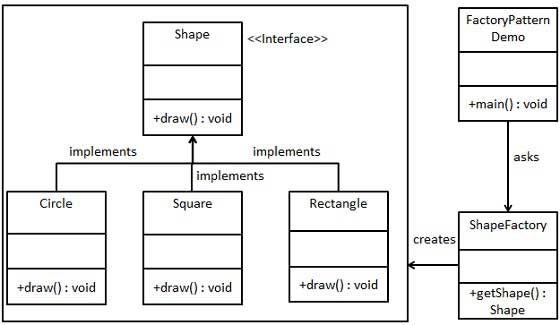
}

## 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**();

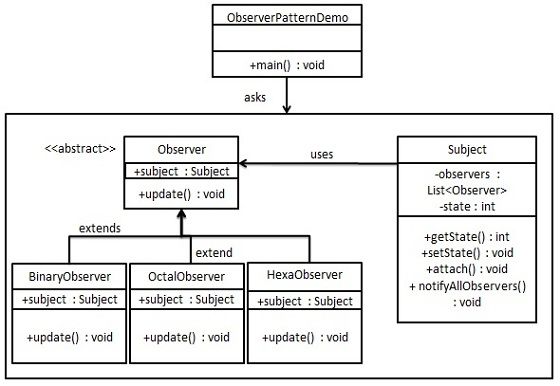
Shape shape = factory.**getShape**("circle");//Shape是interface

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

# 关键字

## static

主要有四种用法：

1. 用来修饰成员变量，将其变为**类的成员**，从而实现所有对象对于该成员的共享；
2. 用来修饰成员方法，将其变为类方法，可以直接使用“**类名.方法名”的方式调用**，常用于工具类；
3. 静态块用法，将多个类成员放在一起初始化，使得程序更加规整，其中理解对象的初始化过程非常关键；
4. 静态导包用法，将类的方法直接导入到当前类中，从而直接使用“方法名”即可调用类方法，更加方便。
5. **修饰成员变量**

static最常用的功能就是修饰类的属性和方法，让他们成为类的成员属性和方法，我们通常将用static修饰的成员称为类成员或者静态成员，意思是static修饰的变量就不是由对象管理了，而是那个类管理，多个对象对应同一个变量，一个对象修改了这个变量，其他对象都会受影响。

public class **Person**{

String name;

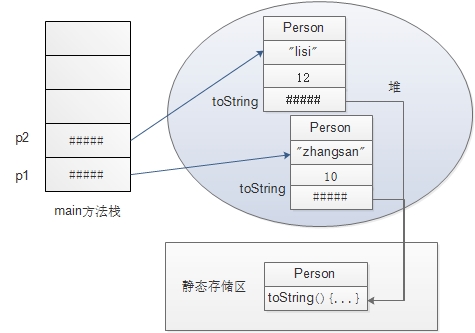
int age;

public String **toString**() {

return "Name:" + name + ", Age:" + age;

}

}



p1和p2两个变量引用的对象分别存储在内存中堆区域的不同地址中，所以他们之间相互不会干扰。两个Person对象的方法实际上只是指向了同一个方法定义。这个方法定义是位于内存中的一块不变区域（由jvm划分），我们暂称它为静态存储区。这一块存储区不仅存放了方法的定义，实际上从更大的角度而言，它存放的是各种类的定义，当我们通过new来生成对象时，会根据这里定义的类的定义去创建对象。多个对象仅会对应同一个方法，这里有一个让我们充分信服的理由，那就是不管多少的对象，他们的方法总是相同的，尽管最后的输出会有所不同，但是方法总是会按照我们预想的结果去操作，即不同的对象去调用同一个方法，结果会不尽相同。

public class **Person**{

String name;

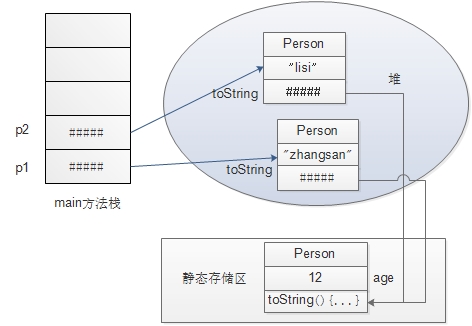
static int age;

public String **toString**() {

return "Name:" + name + ", Age:" + age;

}

}



给age加上了static关键字之后，age交给了Person类管理，对象不再拥有age属相。

1. **修饰成员方法**

相比于修饰成员属性，修饰成员方法对于数据的存储上面并没有多大的变化，因为我们从上面可以看出，方法本来就是存放在类的定义当中的。static修饰成员方法最大的作用，就是可以使用"类名.方法名"的方式操作方法，避免了先要new出对象的繁琐和资源消耗。在一个static修饰的方法中不能使用非static修饰的成员变量和方法。

1. **静态块**

class **Book**{

public **Book**(String msg) {

System.out.**println**(msg);

}

}

public class **Person** {

Book book1 = new **Book**("book1成员变量初始化");

static Book book2 = new **Book**("static成员book2成员变量初始化");

public **Person**(String msg) {

System.out.**println**(msg);

}

Book book3 = new **Book**("book3成员变量初始化");

static Book book4 = new **Book**("static成员book4成员变量初始化");

public static void **funStatic**() {

System.out.**println**("static修饰的funStatic方法");

}

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

Person.**funStatic**();

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

Person p1 = new **Person**("p1初始化");

}

*/\*\*Output*

*\* static成员book2成员变量初始化*

*\* static成员book4成员变量初始化*

*\* static修饰的funStatic方法*

*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\* book1成员变量初始化*

*\* book3成员变量初始化*

*\* p1初始化*

*\*/*

}

可以看到static修饰的在main函数执行之前就已经执行了。

1. 静态导包

import static com.\*\*\*.Helper.\*

static导包之后，Helper类下面所有的static方法都可以直接调用，而不需要用类名.方法名的方式调用。

## final

1. 修饰类

用final修饰一个类时，表明这个类不能被继承。也就是说，如果一个类你永远不会让他被继承，就可以用final进行修饰。final类中的成员变量可以根据需要设为final，但是要注意final类中的所有**成员方法都会被隐式地指定为final方法**。

1. 修饰方法

final修饰方法可以把方法锁定，以防任何继承类修改它的含义，想明确禁止该方法在子类中被覆盖的情况下才将方法设置为final。类的private方法会隐式被指定为final。

1. 修饰变量

对于一个final变量，如果是**基本数据类型**的变量，则其数值一旦在初始化之后便不能更改；如果是**引用类型**的变量，则在对其初始化之后便不能再让其指向另一个对象。

final String str1 = "ja";

String str2 = "ja";

String str3 = str1 + "va";

String str4 = str2 + "va";

String str5 = "java";

System.out.**println**(str3 == str5);//true

System.out.**println**("ja" + "va" == str5);//true

System.out.**println**(str4 == str5);//false

当final变量是基本数据类型以及String类型时，如果在**编译期间能知道它的确切值**（这个是前提，如果final String str1 = getja()这种方式得到的，结果就不一样了），则编译器会把它当做**编译期常量使用**（和c中宏替换差不多一个意思），用到该final变量的地方，相当于直接访问的这个常量，不需要在运行时确定。而不是final修饰的修饰的str2，在访问的时候要通过链接来进行。

final修饰的引用变量初始化之后就不可以再指向其他对象，但是该对象中的成员变量还是可以被修改的。

## violate

synchronized除了互斥的作用，还具有可见性的作用。是指在一个线程中修改变量的值以后，在其他线程中能够看到这个值。synchronized保证了synchronized块中的变量的可见性，而volatile保证了所修饰变量的可见性。因为volatile只是保证同一个变量在多线程中的可见性，所以它更多是用于修饰作为开关状态的变量。

int i; public int **getI**(){return i};

volatile int j; public int **getJ**(){return j};

int k; public synchronized int **getK**(){return k};

getI调用获取的是**当前线程中的副本**，这个值不一定是最新的值。j是volatile修饰的，对于JVM来说这个变量不会有线程的本地副本，只会放在主存中，所以得到的一定是最新的。getK是synchronized修饰的，保证线程的本地副本和主存的同步，所以也会得到最新的值。

# 堆栈

栈的位置仅次于CPU中的寄存器，所以存取速度很快。但是栈中的数据大小与生存周期必须是确定的，缺乏灵活性，此外栈的数据可以共享。java中所有的基本数据类型和引用类型都是在栈中。

堆可以动态的分配内存大小，所以存取较慢，垃圾回收器会自动收走不再使用的数据。

java中有两种数据类型，基本类型（8种：char、byte、short、int、long、float、double、boolean），比如定义int a = 3，a是一个指向int类型的引用，指向3这个字面值。这些字面值数据大小可知，生存期可知，所以都存放在栈中。当我们定义int a = 3, b = 3时，编译器首先在栈中查找是否有字面值为3的地址，如果有，将a指向这个地址，如果没有会在栈中开辟一个地址。同样b便会指向字面值为3的地址（这就是为什么说栈中数据共享）。但是我们执行a++之后，a会指向字面值为4的地址，具体过程和上面一样，先在栈中查找，如果没有找到会在栈中创建。

通过字面值的引用修改其值，不会导致另一个指向此字面值的引用的值改变（上面解释过）。但是如果两个类对象的引用同时指向同一个对象，一个对象引用变量修改了这个对象的内部状态，另一个对象会得到修改后的状态。

另一种是包装类数据，比如Integer、String、Double等将相应的基本数据类型包装起来的类，这些类数据都存在堆中。所以new语句创建的数据对象都存放在堆中。

## String str = "abc"

上面这个语句执行的过程：

1. 定义一个名为str的String类对象的引用，引用是在栈上的；
2. 在栈中查找有没有值为"abc"的地址，如果没有，在栈中开辟一个存放值为"abc"的地址，接着创建了一个新的String类型的对象o，并将o的字符串值指向栈中"abc"的地址，在栈中这个地址旁边记下这个引用的对象o。如果已经有了值为"abc"的地址，则直接查找对象o，并返回o的地址；
3. **将str指向对象o的地址**。（需要注意的是str没有指向栈中存放"abc"的地址，而是指向了一个对象o，而o是指向"abc"的）

String str1 = "abc"

String str2 = "abc"

System.out.**println**(str1 == str2); *//True*

这里没有用str1.equals(str2)，==比较的是对象的地址。

String str1 = "abc"

String str2 = new **String**("abc")

System.out.**println**(str1 == str2); *//False*

new会在堆中创建的对象，所以两个地址肯定不一样。

我们在使用诸如String str = "abc"；的格式定义类时，总是想当然地认为，我们创建了String类的对象str。担心陷阱！对象可能并没有被创建！唯一可以肯定的是，指向 String类的引用被创建了。至于这个引用到底是否指向了一个新的对象，必须根据上下文来考虑，除非你通过new()方法来显要地创建一个新的对象。因 此，更为准确的说法是，我们创建了一个指向String类的对象的引用变量str，这个对象引用变量指向了某个值为"abc"的String类。清醒地认 识到这一点对排除程序中难以发现的bug是很有帮助的。

使用String str = "abc"；的方式，可以在一定程度上提高程序的运行速度，因为JVM会自动根据栈中数据的实际情况来决定是否有必要创建新对象。而对于String str = new String("abc")；的代码，则一概在堆中创建新对象，而不管其字符串值是否相等，是否有必要创建新对象，从而加重了程序的负担。这个思想应该是 享元模式的思想，但JDK的内部在这里实现是否应用了这个模式，不得而知。

# 抽象类和接口

接口是对**动作的抽象**，而抽象类是对**根源的抽象**。比如男人，女人这两个类，那我们可以为这两个类设计一个更高级别的抽象类--人。对于接口，我们可以坐着吃饭，可以站着吃饭，可以用筷子吃饭，可以用叉子吃饭，甚至可以学三哥一样用手抓着吃饭，那么可以把这些吃饭的动作抽象成一个接口--吃饭。所以一个类只能继承一个抽象类，而可以实现多个接口。

1. 抽象类和接口都不能被直接实例化，如果二者要实例化，就涉及到多态。如果抽象类要实例化，那么抽象类定义的变量必须指向一个子类对象，这个子类继承了这个抽象类并实现了这个抽象类的**所有抽象方法**。如果接口要实例化，那么这个接口定义的变量要指向一个子类对象，这个子类必须实现了这个接口**所有的方法**。**抽象类要被子类继承，接口要被子类实现。**
2. **接口里面只能对方法进行声明**，抽象类既可以对方法进行声明也可以对方法进行实现。
3. 抽象类里面的抽象方法必须全部被子类实现，如果子类不能全部实现，那么子类必须也是抽象类。接口里面的方法也必须全部被子类实现，如果子类不能实现那么子类必须是抽象类。
4. 抽象类里面可以没有抽象方法。如果一个类里面有抽象方法，那么这个类一定是抽象类。
5. 抽象类中的方法都要被实现，所以抽象方法不能是静态的static，也不能是私有的private。
6. 接口（类）可以继承接口，甚至可以继承多个接口。但是类只能继承一个类。
7. 抽象类主要是用来抽象类别，接口主要是用来抽象方法功能。当你关注事物的本质的时候，请用抽象类；当你关注一种操作的时候，用接口。
8. 抽象类的功能应该要远多于接口，但是定义抽象类的代价较高。因为高级语言一个类只能继承一个父类，即你在设计这个类的时候必须要抽象出所有这个类的子类所具有的共同属性和方法；但是类（接口）却可以继承多个接口，因此每个接口你只需要将特定的动作方法抽象到这个接口即可。也就是说，接口的设计具有更大的可扩展性，而抽象类的设计必须十分谨慎。

# Hash

## HashSet，HashMap，HashTable

HashSet和HashMap都是collection框架的一部分，HashSet是set的一个实现类，hashMap是Map的一个实现类。collection框架主要有Set接口、List接口和Queue接口。

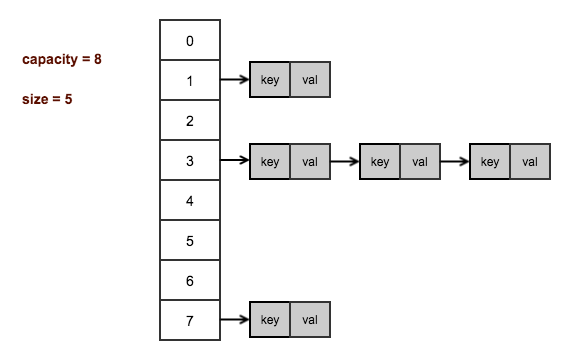
HashSet实现了Set接口，不允许集合中有重复值。把对象存储在HashSet之前要确保对象重写equals和hashCode方法，这样才能保证set中没有存储相同的对象。HashSet实现利用了HashMap，可以少一个参数。

HashMap继承自AbstractMap，实现了Map接口，Map中不允许重复的键（put重复的键不会报错，只是后面的会把前面的覆盖）。Map接口有两个基本的实现，HashMap和TreeMap，TreeMap保存了对象的排列次序，而HashMap则不能。HashMap允许键和值为null。**HashMap是非synchronized的**，但collection框架提供方法能保证HashMap synchronized，这样多个线程同时访问HashMap时，能保证只有一个线程更改Map。

HashTable继承自Dictionary，也实现了Map接口。**Hashtable 中的方法是Synchronize的**。

HashMap把Hashtable的contains方法去掉了，改成containsValue和containsKey，因为contains方法容易让人引起误解。 Hashtable则保留了contains，containsValue和containsKey三个方法，其中contains和containsValue功能相同。

HashMap是支持null键和null值的，而HashTable在遇到null时，会抛出NullPointerException异常。这并不是因为HashTable有什么特殊的实现层面的原因导致不能支持null键和null值，这仅仅是因为HashMap在实现时对null做了特殊处理，将null的hashCode值定为了0，从而将其存放在哈希表的第0个bucket中。



HashMap和HashTable存放数据的方式是一致的。

## ConcurrentHashMap

HashTable是线程安全的，使用synchronized锁住整张Hash表来实现线程安全，**每次锁住整张表让线程独占**。ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了**锁分离**技术。使用了多个锁来控制对hash表的不同部分进行的修改。ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的Hashtable，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。

有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

# 枚举类

# 问题

## 自动装箱、拆箱

Integer a = new **Integer**(2);

Integer b = 2; *//将2自动装箱成Integer*

int c = 2;

System.out.**println**(a == b); *//false,两个对象地址不同*

System.out.**println**(a == c); *//true,a自动拆箱得到int型值*

Integer d = 2;

Integer e = 150, f = 150;

System.out.**println**(b == d); *// true*

System.out.**println**(e == f); *// false*

Integer g = new **Integer**(2);

System.out.**println**(a == g); *// false*

整型字面值的值在-128到127之间，自动装箱的Integer类型不会new新的Integer对象，而是直接引用常量池中的Integer对象。但是new Integer()肯定是在堆空间申请新的空间。

## switch

switch(expr)中，expr只能是byte、short、char、int、enum、String

# 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>