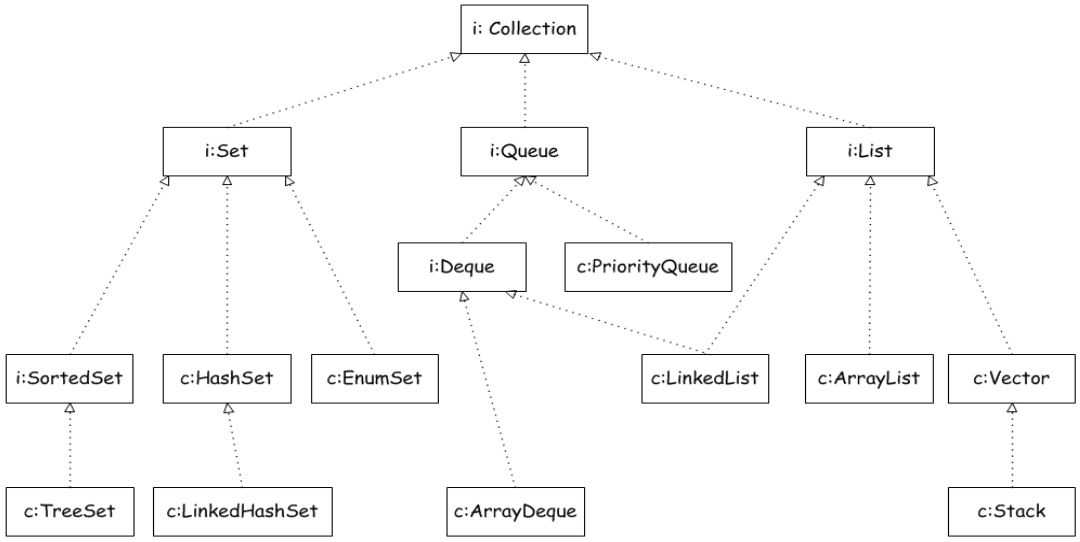
Java 基础

1 集合框架



1.1 Set 接口

Set 类继承了 Conllection 类：不允许出现重复数据；根据 equals 方法判断是否相同。

(1) HashSet 类

特征：实现 set 接口：①不允许出现重复数据；②可以出现 null；③数据是无序的。

原理：HashSet 的底层是是一个 hashmap；元素都存到 HashMap 键值对的 Key 上面，而 Value 时有一个统一的值 private static final Object PRESENT = new Object(); 添加时：当新放入的 key 与集合中原有的 key 相同（hashCode()返回值相等，通过 equals 比较也返回 true），则不添加；

(2) LinkedHashSet

特征：继承 hashset，唯一不同是有序（插入的顺序）。

原理：构造方法调用 LinkedHashMap，但它同时使用链表维护元素的次序。对集合迭代时，按增加顺序返回元素。添加性能略低于 HashSet，但迭代访问元素时会有好性能。

(3) **SortedSet 接口及 TreeSet 实现类**

特征：①不能写入 null；②数据是有序的（不是插入的顺序，是按关键字大小排序的）。

原理：底层是使用 TreeMap 来包含 Set 集合中的所有元素。默认对元素进行默认自然升序排序（String）。自己定义的类必须实现 Comparable 接口，并且覆写相应的 compareTo()函数，才可以正常使用。如果在比较的时候两个对象返回值为 0，那么元素重复。

1.2 List 接口

List 是一个元素有序的、可以重复、可以为 null 的集合，增加了与索引位置相关的操作。

(1) ArrayList

特征：①基于数组实现，长度是可变的；②查询速度快，增删较慢；③不同步

原理：ArrayList 默认会生成一个大小为 10 的 Object 类型的数组。也可以指定数组的大小。如果加入元素后数组大小不够会先进行扩容，每次扩容都将数组大小增大一半。

结论：建议在单线程中才使用 ArrayList，而在多线程中选择者 CopyOnWriteArrayList。

(2) LinkedList

特征：①基于双端链表实现。②增删速度快，查询较慢；③不同步

区别:①ArrayList 基于动态数组实现，LinkedList 基于双端链表实现;②查询，ArrayList 更优，因为 LinkedList 要移动指针；③增删，LinedList 更优，因为 ArrayList 要移动数据。

(3) Vector

特征：①基于数组实现的；②线程安全的。③性能会比 ArrayList 低。

结论：不推荐使用 Vector 类，即使需要考虑同步，即也可以通过其它方法实现。

1.3 Map 接口

(1) HashMap

(2) LinkedHashMap

特征：①继承 HashMap；②允许 key 为 null，value 为 null；③有序。④插入删除慢，

原理：LinkedHashMap 可以认为是 HashMap+LinkedList。通过维护一个双向链表，维护元素迭代的顺序。该迭代顺序可以是插入顺序或者是访问顺序（构造函数可以传入一个参数 accessOrder：false 按照插入的顺序排列；true 按照访问的顺序 LRU）。

构造方法中，实际调用了父类 HashMap 的相关构造方法来构造一个底层存放的 table 数组。LinkedHashMap 重新定义了 **Entry** 数据结构，在 hashmap 的 Entry 基础上，新增了 before，和 after。(next 是用于维护 HashMap 指定 table 位置上连接的 Entry 的顺序的) before、After 是用于维护 Entry 插入的先后顺序的。

(3) TreeMap

TreeMap 是 SortedMap 接口基于红黑树的实现，默认按照升序排列关键字。HashMap 是根据键的 HashCode 值存储数据，取得数据的顺序是完全随机的，HashMap 取值的速度更快。

(4) Hashtable

特征：①数据结构为哈希表，②同步的，③不允许 null 作为键和值。

原理：初始大小为 11，负载因子为 0.75 的 Entry 数组。而 Entry 实际上就是一个单向链表。当添加元素的时候，首先计算 key 的 hash 值，然后通过 hash 值确定在 table 数组中的索引位置，最后将 value 值替换或者插入新的元素，如果容器的数量达到阈值，就会进行扩容。

2 hashcode、equals

答：**两个对象 equals 相等那么 hashcode 是一定相等的；如果两个对象的 hashCode 相同，它们 equals 并不一定相同。所以二者必须同时重写。**

因为例如 HasmMap 中通过获取 key 的哈希值，然后通过 hash 方法计算出一个值，这个值作为比对的依据。也就是，**如果两个对象相等，那么他们的 hashCode 就一定要相等，不然这里这两个对象就会得到不同 hash 值从而存在不同的地方。**HashCode 说白了是地址值经过一系列的复杂运算得到的结果，而 Object 中的 equals 方法底层比较的就是地址值，所以 equals()相等，hashCode 必定相等，反 equals()不等，在 java 底层进行哈希运算的时候有一定的几率出现相等的 hashCode,所以 hashCode（）可等可不等。

3 final、finally 、finalize

(1) final：①被 final 修饰的类，就意味着不能再派生出新的子类，不能作为父类被继承。因此一个类不能被声明为 abstract，又被声明为 final。②将变量或方法声明为 final。可以保证他们在使用的时候不被改变。被声明为 final 的变量必须在声明时给出变量的初始值，而在以后的引用中只能读取。③被声明为 final 的方法也只能使用，不能重写。

(2) finally：在异常处理时提供 finally 块来执行任何清除操作。无论异常是否发生，finally 块都会被执行。

(3) finalize：finalize 是方法名。它是在 Object 类中定义的，因此，所有的类都继承了它。finalize()方法

是在垃圾收集器删除对象之前对这个对象调用的。

4 异常分类以及处理机制

Throwable：分为 Error 和 Exception。

(1) Error：当程序发生不可控的错误时，通常做法是通知用户并中止程序的执行。

(2) Exception:一般分为 Checked 异常和 Runtime 异常。

编译异常：没有处理 Checked 异常，该程序在编译时就会发生错误无法编译。处理方法：

①当前方法知道如何处理该异常，则用 try...catch 块来处理该异常。

②当前方法不知道如何处理，则在定义该方法声明时使用 throws 抛出该异常。

常见异常：EOFException、FileNotFoundException、IOException

运行时异常： ArrayIndexOutOfBoundsException 、 ArithmeticException 、 NullPointerException 、 ClassNotFoundException、IllegalArgumentException。

5 重载和重写的区别

(1) 重载 overloading

①方法重载是让类以统一的方式处理不同类型数据的一种手段。多个同名函数同时存在，具有不同的参数个数/类型。重载是一个类中多态性的一种表现。

②重载的时候，方法名一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准。

(2) 重写 Override

①参数列表相同。②返回的类型相同。

③子类方法大于等于重写方法的访问权限。

④子类方法不能抛出比父类方法更多的异常。

6 抽象类和接口有什么区别

抽象类和接口都不能被直接实例化。接口和抽象类的概念不一样。抽象类表示的是，这个对象是什么。接口表示的是，这个对象能做什么。

①接口中所有的方法都是抽象的。而抽象类可以进行声明也可以对方法进行实现。

②子类使用 extends 关键字来继承，如果子类不是抽象类，需要重写所有抽象方法；接口使用 implements 来实现，需要重写所有方法。

③接口可以多继承，抽象类只能继承一个类。

④抽象方法不能是静态的 static，接口方法可以是静态的；

7 static 关键字什么情况下使用

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。

8 GET 与 POST 的区别

①GET 参数通过 URL 传递，POST 放在 request body 中。

②GET 请求在 URL 中传递的参数是有长度限制的，最大是 2k，而 POST 理论上没有限制。

③GET 产生一个 TCP 数据包；POST 产生两个 TCP 数据包。GET 方式的请求，浏览器会把 http header 和 data 一并发送出去，服务器响应 200(返回数据); 对于 POST，浏览器先发送 header，服务器响应 100 continue，浏览器再发送 data，服务器响应 200 ok（返回数据）。

④GET 比 POST 更不安全，因为参数直接暴露在 URL 中，所以不能用来传递敏感信息。

⑤GET 请求只能进行 URL 编码，而 POST 支持多种编码方式。

9 session 与 cookie 区别

(1) 区别

①Cookie 和 Session 都是会话技术，Cookie 是运行在客户端，Session 是运行在服务器端。

②Cookie 有大小限制以及浏览器在存 cookie 的个数也有限制，Session 是没有大小限制和服务器的内存大小有关。

③Cookie 有安全隐患，通过拦截或本地文件找得到你的 cookie 后可以进行攻击。

④Session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能。

(2) 注意：服务端的 session 的实现对客户端的 cookie 有依赖关系的，服务端执行 session 机制会生成 session 的 id 值，这个 id 值会发送给客户端，客户端每次请求都会把这个 id 值放到 http 请求的头部发送给服务端，而这个 id 值在客户端会保存下来，保存的容器就是 cookie，因此当我们完全禁掉浏览器的 cookie 的时候，服务端的 session 也会不能正常使用。

10 session 分布式处理

(1) 分布式 Session 的几种实现方式

第一种：粘性 session

原理：粘性 Session 是指将用户锁定到某一个服务器上，比说说，用户第一次请求时，负载均衡器将用户的请求转发到了 A 服务器上，那么用户以后的每次请求都会转发到 A 服务器上，相当于把用户和 A 服务器粘到了一块，这就是粘性 Session 机制。

优点：简单，不需要对 session 做任何处理。

缺点：缺乏容错性，如果当前服务器发生故障，用户被转移到其他服务器时，session 信息都将失效。

适用场景：发生故障对客户产生的影响较小；服务器发生故障是低概率事件。

实现方式：以 Nginx 为例，在 upstream 模块配置 ip_hash 属性即可实现粘性 Session。

第二种：服务器 session 复制

原理：任何一个服务器上的 session 发生改变，该节点会把这个 session 的所有内容序列化，然后广播给所有其它节点，以此来保证 Session 同步。

优点：可容错，各个服务器间 session 能够实时响应。

缺点：网络负荷造成压力，如果 session 量大可能会造成网络堵塞，拖慢服务器性能。

实现方式：设置 tomcat，server.xml 开启 tomcat 集群功能；在 web.xml 中添加选项

<distributable/>支持分布式。

第三种：session 共享机制

使用分布式缓存方案比如 memcached、redis，但是要求 Memcached 或 Redis 必须是集群。

①粘性 session 处理方式

原理：不同的 tomcat 指定访问不同的主 memcached。多个 Memcached 之间信息是同步的，能主从备份和高可用。用户访问时首先在 tomcat 中创建 session，然后将 session 复制放到它对应的 memcahed 上。memcache 只起备份作用，读写都在 tomcat 上。当某一个 tomcat 挂掉后，集群将用户的访问定位到备 tomcat 上，然后根据 cookie 中存储的 SessionId 找 session，找不到时，再去相应的 memcached 上去 session，找到之后将其复制到备 tomcat 上。

②非粘性 session 处理方式

原理：memcached 做主从复制，写入 session 都往从 memcached 服务上写，读取都从主 memcached 读取，tomcat 本身不存储 session。

优点：可容错，session 实时响应。

第四种：session 持久化到数据库

原理：拿出一个数据库，专门用来存储 session 信息。保证 session 的持久化。

优点：服务器出现问题，session 不会丢失

缺点：如果网站的访问量很大，会对数据库造成很大压力，还需要增加额外的开销维护数据库。
第五种：利用 cookie 记录 session

11 equals 与 == 的区别

① == 是运算符，equals()是 Object 类中方法；
②对于基本数据类型：== 可以用来对比值是否相等，equals 不能。
③对于引用数据类型：== 和 equals 都比较的是对象的地址。如果不重写 equals 方法，底层调用的就是==；也可以像 String 类一样重写 equals 方法，用来对字符串的内容进行比较。

12 Int 和 integer 的区别

注意：给一个 Integer 对象赋一个 int 值的时候，会调用 Integer 类的静态方法 valueOf：会缓存-128 到 127，即整型字面量的值在-128 到 127 之间，将不会 new 新的 Integer 对象，而是直接引用常量池中的 Integer 对象。

13 &和&&的区别？

&运算符有两种用法：(1)按位与；(2)逻辑与。&&运算符是短路与运算。如果&&左边的表达式的值是 false，右边的表达式不会进行运算。

14 switch 类型

在 Java 5 以前，switch(expr)中，expr 只能是 byte、short、char、int。从 Java 5 开始，引入了枚举类型；从 Java 7 开始，引入字符串（String），但是长整型（long）在目前所有的版本中都是不可以的。

15 请用至少四种写法写单例模式

```
//1、饿汉模式
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance(){
        return instance;
    }
}
```

```
// 2、双重校验锁
public class Singleton {
    private volatile static Singleton instance;
    private Singleton (){}
    public static Singleton getInstance (){
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton ();
                }
            }
        }
        return singletonDemo7;
    }
}
```

```
//3、静态内部类
```

```
public class Singleton {
    private static class SingletonHolder{
        private static final Singleton INSTANCE = new Singleton ();
    }
    private Singleton (){}
    public static final Singleton getInsatance(){
        return SingletonHolder.INSTANCE;
    }
}
```

```
//4、枚举
public enum Singleton {
    INSTANCE;
}
```

双重原因:如果线程一执行到第二个 if 判断时,切换到线程二,这时 instance 仍是 null,则会出现两个 instance.

volatile 原因：instance = new Singleton();分解为三步：(1)分配对象的内存空间;(2)初始化对象;(3)设置 instance 指向刚分配的内存地址；2、3 会重排，导致获取到未初始化的非 null 对象。volatile 屏蔽指令重排序的语义在 JDK1.5 才被完全恢复。

16 类的实例化顺序

有父类的情况

(1) 加载父类：①静态属性；②静态初始化块（从上至下）

(2) 加载子类：①静态属性；②静态初始化块（从上至下）

(3) 加载父类构造器：①成员变量；②实例初始化块；③ 构造器内容

(4) 加载子类构造器：①成员变量；②实例初始化块；③ 构造器内容

17 生产者消费者

```
//1、方式一：synchronized、wait 和 notify
public class ProducerConsumerWithWaitNofity {
    public static void main(String[] args) {
        Resource resource = new Resource();
        ProducerThread p1 = new ProducerThread(resource);   //生产者线程
        ProducerThread p2 = new ProducerThread(resource);   //生产者线程
        ConsumerThread c1 = new ConsumerThread(resource);   //消费者线程
        p1.start();
        p2.start();
        c1.start();
    }
}

class Resource {   //公共资源类
    private int num = 0;     //当前资源数量
    private int size = 10;   //资源池中允许存放的资源数目

    public synchronized void remove() {     //从资源池中取走资源
```

```
        if (num > 0) {
            num--;
            System.out.println("消费者" + Thread.currentThread().getName() + "消耗一件资源，当前线程池
                                                                    有" + num + "个");
            notifyAll();     //通知生产者生产资源
        } else {
            try {
                wait();      //如果没有资源，则消费者进入等待状态
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public synchronized void add() {   //向资源池中添加资源
        if (num < size) {
            num++;
            System.out.println("生产者" + Thread.currentThread().getName() + "生产一件资源，
                                                            当前资源池有" + num + "个");
            notifyAll();    //通知等待的消费者
        } else {
            try {
                wait();//如果当前资源池中有 10 件资源，生产者进入等待状态
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class ConsumerThread extends Thread {   //消费者线程
    private Resource resource;
    public ConsumerThread(Resource resource) {
        this.resource = resource;
    }
    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource.remove();
        }
```

```
    }
}

class ProducerThread extends Thread {   //生产者线程
    private Resource resource;
    public ProducerThread(Resource resource) {
        this.resource = resource;
    }
    @Override
    public void run() {
        while (true) {   //不断地生产资源
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource.add();
        }
    }
}
```

```
// 2、使用阻塞队列 BlockingQueue 解决生产者消费者
public class BlockingQueueConsumerProducer {
    public static void main(String[] args) {
        Resource3 resource = new Resource3();
        ProducerThread3 p = new ProducerThread3(resource);  //生产者线程
        ConsumerThread3 c1 = new ConsumerThread3(resource);    //多个消费者
        ConsumerThread3 c2 = new ConsumerThread3(resource);
        ConsumerThread3 c3 = new ConsumerThread3(resource);
        p.start();
        c1.start();
        c2.start();
        c3.start();
    }
}

class ConsumerThread3 extends Thread {  //消费者线程
    private Resource3 resource3;

    public ConsumerThread3(Resource3 resource) {
        this.resource3 = resource;
    }

    public void run() {
        while (true) {
```

```
            try {
                Thread.sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource3.remove();
        }
    }
}

//生产者线程
class ProducerThread3 extends Thread {
    private Resource3 resource3;

    public ProducerThread3(Resource3 resource) {
        this.resource3 = resource;
    }

    public void run() {
        while (true) {
            try {
                Thread.sleep((long) (1000 * Math.random()));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource3.add();
        }
    }
}

class Resource3 {
    private BlockingQueue resourceQueue = new LinkedBlockingQueue(10);

    public void add() {   // 向资源池中添加资源
        try {
            resourceQueue.put(1);
            System.out.println("生产者" + Thread.currentThread().getName()
                    + "生产一件资源," + "当前资源池有" + resourceQueue.size() + "个资源");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public void remove() {  //向资源池中移除资源
        try {
```

```
            resourceQueue.take();
            System.out.println("消费者" + Thread.currentThread().getName() +
                    "消耗一件资源," + "当前资源池有" + resourceQueue.size() + "个资源");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
//3 使用 Condition
public class Mutil_Producer_ConsumerByCondition {
    public static void main(String[] args) {
        Resource r = new Resource();
        Mutil_Producer pro = new Mutil_Producer(r);
        Mutil_Consumer con = new Mutil_Consumer(r);
        Thread t0 = new Thread(pro);   //生产者线程
        Thread t1 = new Thread(pro);
        Thread t2 = new Thread(con);  //消费者线程
        Thread t3 = new Thread(con);
        t0.start();                    //启动线程
        t1.start();
        t2.start();
        t3.start();
    }
}

class Resource  {
    private String name;
    private int count = 1;
    private boolean flag = false;
    Lock lock = new ReentrantLock();//创建一个锁对象
    //通过已有的锁获取两组监视器，一组监视生产者，一组监视消费者。
    Condition producer_con = lock.newCondition();
    Condition consumer_con = lock.newCondition();

    public void product(String name) {    //生产
        lock.lock();
        try {
            while (flag) {
                try {
                    producer_con.await();
                } catch (InterruptedException e) {
                }
            }
            this.name = name + count;
```

```
            count++;
            System.out.println(Thread.currentThread().getName() + "...生产者..." + this.name);
            flag = true;
            consumer_con.signal();//直接唤醒消费线程
        } finally {
            lock.unlock();
        }
    }

    public void consume() {     //消费
        lock.lock();
        try {
            while (!flag) {
                try {
                    consumer_con.await();
                } catch (InterruptedException e) {
                }
            }
            System.out.println(Thread.currentThread().getName() + "...消费者......." + this.name);
            flag = false;
            producer_con.signal();//直接唤醒生产线程
        } finally {
            lock.unlock();
        }
    }
}

class Mutil_Producer implements Runnable {    //生产者线程
    private Resource  r;
    Mutil_Producer(Resource  r) {
        this.r = r;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            r.product("北京烤鸭");
        }
    }
}

//消费者线程
class Mutil_Consumer implements Runnable {
    private Resource  r;
    Mutil_Consumer(Resource  r) {
        this.r = r;
```

```
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            r.consume();
        }
    }
}
```

```
public void product(String name) {
      lock.lock();
      try{
            while(flag){
                  try{
                        producer_con.await();
                  }catch(InterruptedException e){}
            }
            this.name = name + count;
            count++;
            System.out.println(Thread.currentThread().getName()+"...生产者 5.0..."+this.name);
            flag = true;
            consumer_con.signal();//直接唤醒消费线程
      }finally{
            lock.unlock();
      }
}
```

```
public synchronized void product(String name){
      while(flag){
            //此时有烤鸭，等待
            try {
                  this.wait();
            } catch (InterruptedException e) {}
      }
      this.name=name+count;//设置烤鸭的名称
      count++;
      System.out.println(Thread.currentThread().getName()+"...生产者..."+this.name);
      flag=true;//有烤鸭后改变标志
      notifyAll();//通知消费线程可以消费了
}
```

18 线程交替打印例子

```
public static void main(String[] args) {
    Thread A = new Thread(new ThreadA());
    Thread B = new Thread(new ThreadB());
    A.start();
    B.start();
}

static class ThreadA implements Runnable{
  @Override
  public void run() {
    synchronized(lock) {
      for (int i = 0; i < 26; i++ ) {
          System.out.print(ch ++);
          lock.notify();
          try {
            lock.wait();
          } catch (InterruptedException e) {}
      }
    }
  }
}

static class ThreadB implements Runnable{
  @Override
  public void run() {
    synchronized(lock) {
      for (int i = 0; i < 26; i++ ) {
          System.out.print(array ++);
          lock.notify();
          try {
            lock.wait();
          } catch (InterruptedException e) {}
      }
    }
  }
}
```

18 设计模式

单例模式，**适配器模式**，工厂模式，**装饰模式**，模板方法模式，**代理模式**，责任链模式

19 JAVA1.8 版本增了哪些新功能

①接口的默认方法：允许添加非抽象方法的实现，加上关键字 default
②lambda 表达式：Arrays.asList("a", "b", "d").forEach(e -> System.out.println(e));
③函数式接口：用@FunctionalInterface 保证接口只有一个抽象方法（接收 lambda 表达式）
④函数式接口里的默认方法和静态方法
⑤方法的引用：使用 Class::new 引用构造函数;【Class::method； Class::static_method；instance::method】
⑥重复注解：重复注解机制本身必须用@Repeatable 注解
⑦Optional：解决空指针异常问题。ofNullable 允许参数为 null
⑧Stream
⑨时间

⑩并行（parallel）数组

20 你们的应用用的服务器是哪种？

Tomcat 是 Apache 软件基金会的一个核心项目，由 Apache、Sun 和其他一些公司及个人共同开发而成。因为 Tomcat 技术先进、性能稳定，而且免费，成为目前比较流行的 Web 应用服务器。目前最新版本是 9.0。Tomcat 运行时**占用的系统资源小，扩展性好，支持负载平衡与邮件服务**等开发应用系统常用的功能；支持 Servlet 和 JSP 等特性。

Resin 是 CAUCHO 公司的产品，是一个非常流行的 application server，对 servlet 和 JSP 提供了良好的支持，性能也比较优良，resin 自身采用 JAVA 语言开发。支持负载均衡。

1.先从是否收费上进行比较：

Resin 作为个人开发是免费的，如果需要将开发产品作为商业产品发布是需要收费的。

Tomcat 是开源的，免费。

2.Resin 在一台机器上配置多个运行实例时，稍显麻烦，不像 Tomcat 复制多份，修改个端口即可，完全独立。

二者在稳定性上都没有任何问题、性能在访问量不大的话，都没有多大的差别。目前选择 Tomcat 的较多。

21 QPS 计算

问题：每天 80%的访问集中在 20%的时间，每天有 300 万的 pv，而单台机器的 qps 为 50，需要几台机器？

原理：(总 PV 数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)

峰值时间每秒 QPS / 单台机器的 QPS = 需要的机器

解答：这台机器需要多少 QPS？ (3000000 * 0.8) / (86400 * 0.2) = 139 (QPS)

需要几台机器来支持？139 / 50 = 3

22 linux 常用命令

find /path -name '*string*' 查找文件名包含某个字符串的的文件

grep "被查找的字符串" 文件名 查询文件内容

-n 显示行数

-i 忽略大小写

-c 查找匹配的行数

find . -name '*.sql' -exec grep –i

rm –rf 递归强制删除

sudo chmod u+x g+w o+r filename 修改权限

tar –zxvf 文件名 解压 (-z：gzip 压缩文件；-v：显示压缩或解压缩过程；-f：使用档名；-x：解压)

netstat 显示网络状态信息

通信协议

1 说一下 TCP /IP 四层?

表1-1 TCP/IP四层模型和OSI七层模型对应表

OSI七层网络模型	Linux TCP/IP四层概念模型	对应网络协议
应用层（Application）	应用层	TFTP, FTP, NFS, WAIS
表示层（Presentation）		Telnet, Rlogin, SNMP, Gopher
会话层（Session）		SMTP, DNS
传输层（Transport）	传输层	TCP, UDP
网络层（Network）	网际层	IP, ICMP, ARP, RARP, AKP, UUCP
数据链路层（Data Link）	网络接口	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP
物理层（Physical）		IEEE 802.1A, IEEE 802.2到IEEE 802.11

2 常见的状态码

2 开头 （请求成功）表示成功处理了请求的状态代码。

200 （成功） 服务器已成功处理了请求。 通常，这表示服务器提供了请求的网页。

203 （非授权信息） 服务器已成功处理了请求，但返回的信息可能来自另一来源。

204 （无内容） 服务器成功处理了请求，但没有返回任何内容。

205 （重置内容） 服务器成功处理了请求，但没有返回任何内容。

206 （部分内容） 服务器成功处理了部分 GET 请求。

3 开头 （请求被重定向）表示要完成请求，需要进一步操作。

301 （永久移动） 请求的网页已永久移动到新位置。

302 （临时移动）

4 开头 （请求错误）这些状态代码表示请求可能出错，妨碍了服务器的处理。

400 （错误请求） 服务器不理解请求的语法。

401 （未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。

403 （禁止） 服务器拒绝请求。

404 （未找到） 服务器找不到请求的网页。

5 开头（服务器错误）这些状态代码表示服务器在尝试处理请求时发生内部错误。

500 （服务器内部错误） 服务器遇到错误，无法完成请求。

502 （错误网关） 服务器作为网关或代理，从上游服务器收到无效响应。

503 （服务不可用） 服务器目前无法使用（由于超载或停机维护）。

Mybatis

1 代码自动生成器：generatorConfiguration

① .xml：包含数据库驱动器；数据库连接；生成实体类名，指定包名以及生成的地址；生成 Dao 文件；对应数据库表；

② 新建文件夹

③java -jar C:\Users\Laura\Desktop\mybatis-generator\mybatis-generator\mybatis-generator-core-1.3.1.jar -configfile C:\Users\Laura\Desktop\mybatis-generator\generatorBp.xml –overwrite

2 缓存使用场景及选择策略

Mybatis 的一级缓存是指 SqlSession。一级缓存的作用域是一个 SqlSession。Mybatis 默认开启一级缓存。在同一个 SqlSession 中，执行相同的查询 SQL，第一次会去查询数据库，并写到缓存中；第二次直接从缓存中取。当执行 SQL 时两次查询中间发生了增删改操作，则 SqlSession 的缓存清空。

Mybatis 的二级缓存是指 mapper 映射文件。二级缓存的作用域是同一个 namespace 下的 mapper 映射文件内容，多个 SqlSession 共享。Mybatis 需要手动设置启动二级缓存。在同一个 namespace 下的 mapper 文件中，执行相同的查询 SQL，第一次会去查询数据库，并写到缓存中；第二次直接从缓存中取。当执行 SQL 时两次查询中间发生了增删改操作，则二级缓存清空。

开启：mybatis 文件：<setting name="cacheEnabled" value="true" />

在映射文件中，开启二级缓存：<cache/>

多表操作一定不能使用缓存.

3 MyBatis 的动态代理

当定义好一个 Mapper 接口(UserDao)里，我们并不需要去实现这个类，但 sqlSession.getMapper()最终会返回一个实现该接口的对象。这个对象是 Mybatis 利用 jdk 的动态代理实现的。这里将介绍这个代理对象的生成过程及其方法的实现过程。

①Mapper 代码对象的生成过程：获取每个 Mapper 接口对应一个 MapperProxyFactory 对象实例，然后调用 MapperRegistry.getMapper()方法，生成一个 MapperProxy 对象。

②MapperProxyFactory 的 newInstance 方法：创建一个 MapperProxy 对象，这个方法实现了 JDK 动态代理中的 InvocationHandler 接口，说明 Mapper 接口被代理了，这样子返回的对象就是 Mapper 接口的子类，方法被调用时会被 mapperProxy 拦截,也就是执行 mapperProxy.invoke()方法 。invoke 方法会拦截 Mapper 接口(UserDao)的所有方法， MapperProxy 会根据方法找到对应的 MapperMethod 对象来实现这次调用。

③MapperMethod 对应会读取方法中的注解，从 Configuration 中找到相对应的 MappedStatement 对象，再执行。

分布式

1 谈谈业务中使用分布式的场景

(1)首先，需要了解系统为什么使用分布式。

随着互联网的发展，传统单工程项目的很多性能瓶颈越发凸显，性能瓶颈可以有几个方面。**一、应用服务层**：随着用户量的增加，并发量增加，单项目难以承受如此大的并发请求导致的性能瓶颈。**二、底层数据库层**：随着业务的发展，数据库压力越来越大，导致的性能瓶颈。

(2)针对上面两点，我觉得可以从两方面解决。

应用系统集群最简单的就是服务器集群。应用系统集群的时候，比较凸显的问题是 **session 共享**，session 共享我们一是可以通过服务器插件来解决。另外一种也可以通过 redis 等中间件实现。

服务化拆分，是目前非常火热的一种方式。现在都在提微服务话。通过对传统项目进行服务化拆分，达到服务独立解耦，单服务又可以横向扩容。服务化拆分遇到的经典问题就是**分布式事务问题**。目前，比较常用的分布式事务解决方案有几种：消息最终一致性、TCC 补偿型事务、尽最大能里通知。

底层数据库层：如果系统的性能压力出现在数据库，那我们就可以读写分离、分库分表等方案进行解决。

2 Session 分布式方案

目前网上能找到的方案有：

1. 基于数据库的 Session 共享：

原理：就不用多说了吧，拿出一个数据库，专门用来存储 session 信息。保证 session 的持久化。

优点：服务器出现问题，session 不会丢失

缺点：如果网站的访问量很大，把 session 存储到数据库中，会对数据库造成很大压力，还需要增加额外的开销维护数据库。

2.客户端存储：

思路：将 session 存储到浏览器 cookie 中，每个端只要存储一个用户的数据了；

缺点：每次 http 请求都携带 session，占外网带宽；数据存储在端上，并在网络传输，存在泄漏、篡改、窃取等安全隐患；session 存储的数据大小受 cookie 限制。

3. 粘性 session

原理：**粘性 Session 是指将用户锁定到某一个服务器上**。用户第一次请求时，负载均衡器将用户的请求转发到了 A 服务器上，如果负载均衡器设置了粘性 Session 的话，那么用户以后的每次请求都会转发到 A 服务器上。

优点：简单，不需要对 session 做任何处理。

缺点：缺乏容错性，如果当前访问的服务器发生故障，用户被转移到第二个服务器上时， session 将失效。

适用场景：发生故障对客户产生的影响较小；服务器发生故障是低概率事件。

实现方式：以 Nginx 为例，在 upstream 模块配置 ip_hash 属性即可实现粘性 Session。

4. 服务器 session 复制

原理：任何一个服务器上的 session 发生改变，该节点会把这个 session 的所有内容序列化，然后广播给所有其它节点，不管其他服务器需不需要 session，以此来保证 Session 同步。

优点：可容错，各个服务器间 session 能够实时响应。

缺点：会对网络负荷造成一定压力，如果 session 量大的话可能会造成网络堵塞，拖慢服务器性能。

实现方式：① 设置 tomcat ，server.xml 开启 tomcat 集群功能；② 在应用里增加信息：通知应用当前处于集群环境中，支持分布式，在 web.xml 中添加选项 <distributable/>。

5. session 共享机制

使用分布式缓存方案比如 memcached、redis，但是要求 Memcached 或 Redis 必须是集群。

① 粘性 session 处理方式

原理：不同的 tomcat 指定访问不同的主 redis。多个 redis 之间信息是同步的，能主从备份和高可用。用户访问时首先在 tomcat 中创建 session，然后将 session 复制一份放到它对应的 redis 上。memcache 只起备份作用，读写都在 tomcat 上。当某一个 tomcat 挂掉后，集群将用户的访问定位到备 tomcat 上，然后根据 cookie 中存储的 SessionId 找 session，找不到时，再去相应的 redis 上去 session，找到之后将其复制到备 tomcat 上。

② 非粘性 session 处理方式

原理：redis 做主从复制，写入 session 都往从 redis 服务上写，读取都从主 redis 读取，tomcat 本身不存储 session。

优点：可容错，session 实时响应。

Spring Session + Redis 实现分布式 Session 共享：Redis Server 版本不低于 2.8

https://blog.csdn.net/zouxucong/article/details/53286748

添加依赖：spring-session-data-redis/jedis

Spring 配置：RedisHttpSessionConfiguration/JedisConnectionFactory

配置 web.xml 过滤器：配置过滤器 DelegatingFilterProxy

3 分布式锁的场景

比较敏感的数据比如金额修改，同一时间只能有一个人操作，想象下 2 个人同时修改金额，一个加金额一个减金额，为了防止同时操作造成数据不一致，需要锁。如果是数据库需要的就是行锁或表锁，如果是在集群里，多个客户端同时修改一个共享的数据就需要分布式锁。场景：例如秒杀。

4 分布式锁的实现方案

谈到分布式锁，有很多实现方式，如数据库、redis、ZooKeeper 等。

1. 数据库实现分布式锁-行锁

select * from lock where lock_name=xxx for update;

数据库的 lock 表，lock_name 是主键,通过 for update 操作，数据库就会对该行记录加上 record lock，从而阻塞其他人对该记录的操作。一旦获取到了锁，就可以开始执行业务逻辑，最后通过 connection.commit()操作来释放锁。

问题：首先性能不是特别高。通过数据库的锁来实现多进程之间的互斥，但是这貌似也有一个问题：就是 sql 超时异常的问题。jdbc 超时具体有 3 种超时：框架层的事务超时（不涉及）；jdbc 的查询超时（mysql 的 jdbc 驱动会向服务器发送 kill query 命令来取消查询）；Socket 的读超时（如果一旦出现 Socket 的读超时，对于如果是同步通信的 Socket 连接来说，该连接基本上不能使用了，需要关闭该连接，从新换用新的连接，因为会出现请求和响应错乱的情况，比如 jedis 出现的类型转换异常）。

2. redis 实现分布式锁

redis 通常可以使用 setnx 来实现分布式锁。setnx 来创建一个 key，如果 key 不存在则创建成功返回 1，如果 key 已经存在则返回 0。依照上述来判定是否获取到了锁，获取到锁的执行业务逻辑，完毕后删除 lock_key，来实现释放锁。

改进：一旦获取到锁的客户端挂了，没有执行上述释放锁的操作，则其他客户端就无法获取到锁了，所以在这种情况下有 2 种方式来解决：①为 lock_key 设置一个过期时间；②对 lock_key 的 value 进行判断是否过期：一旦发现 lock_key 的值已经小于当前时间了，说明该 key 过期了，然后对该 key 进行 getset 设置，一旦 getset 返回值是原来的过期值，说明当前客户端是第一个来操作的，代表获取到了锁，一旦 getset 返回值不是原来过期时间则说明前面已经有人修改了，则代表没有获取到锁。

问题：①lock timeout 的存在也使得失去了锁的意义，即存在并发的现象。一旦出现锁的租约时间，就意味着获取到锁的客户端必须在租约之内执行完毕业务逻辑，一旦业务逻辑执行时间过长，租约到期，就会引发并发问题。所以有 lock timeout 的可靠性并不是那么的高。②上述方式仅仅是 redis 单机情况下，还存在

redis 单点故障的问题。如果为了解决单点故障而使用 redis 的 sentinel 或者 cluster 方案，则问题更多。

3. ZooKeeper 实现分布式锁

https://www.jianshu.com/p/2d22df6eccf8

首先 zookeeper 创建个 PERSISTENT 持久节点，然后每个要获得锁的线程都会在这个节点下创建个临时顺序节点，然后规定节点最小的那个获得锁，所以每个线程首先都会判断自己是不是节点序号最小的那个，如果是则获取锁，如果不是则监听比自己小的上一个节点，如果上一个节点不存在了，然后会再一次判断自己是不是序号最小的那个节点，是则获得锁，不是重复上述动作。

1 获取锁

```
public void lock(){
    path = 在父节点下创建临时顺序节点
    while(true){
        children = 获取父节点的所有节点
        if(path 是 children 中的最小的){
            //代表获取了节点
            return;
        }else{
            //添加监控前一个节点是否存在的 watcher
            wait();
        }
    }
}
watcher 中的内容{
    notifyAll();
}
```

2 释放锁

```
public void release(){
    删除上述创建的节点
}
```

5 分布式事务

① 在说分布式事务之前，我们先从数据库事务说起。数据库事务的几个特性：原子性(Atomicity)、一致性(Consistency)、隔离性或独立性(Isolation)和持久性(Durabilily)，简称就是 **ACID**。

② **分布式理论- CAP 定理**：在分布式系统中，一致性（Consistency）、可用性（Availability）和分区容忍性（Partition Tolerance）3 个要素最多只能同时满足两个，不可兼得。其中，分区容忍性又是不可或缺的。

一致性：分布式环境下多个节点的数据是否强一致。

可用性：分布式服务能一直保证可用状态。当用户发出一个请求后，服务能在有限时间内返回结果。

分区容忍性：特指对网络分区的容忍性。

③ **BASE 理论**：在分布式系统中，我们往往追求的是可用性，它的重要程序比一致性要高，那么如何实现高可用性呢？提出来了另外一个理论，就是 BASE 理论，它是用来对 CAP 定理进行进一步扩充的。BASE 理论指的是：Basically Available（基本可用）、Soft state（软状态）、Eventually consistent（最终一致性）。BASE 理论是对 CAP 中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）。

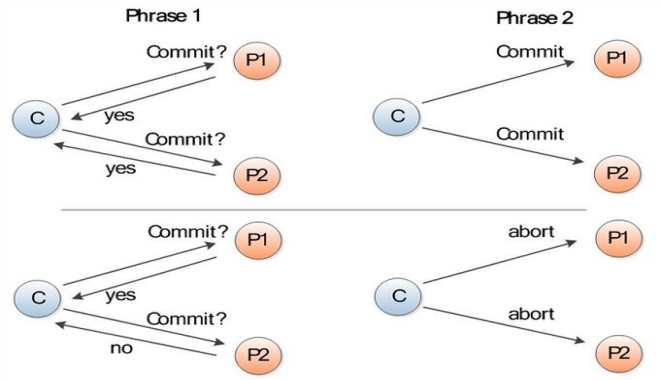
基本可用（BasicallyAvailable）：指分布式系统在出现故障时，允许损失部分的可用性来保证核心可用。

软状态（SoftState）：指允许分布式系统存在中间状态，该中间状态不会影响到系统的整体可用性。

最终一致性（EventualConsistency）：指分布式系统中的所有副本数据经过一定时间后，最终能够达到一致的状态。

④ **分布式事务：**简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

一、两阶段提交（2PC）：两阶段提交这种解决方案属于牺牲了一部分可用性来换取的一致性。



优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能 100%保证强一致）

缺点： 实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景。

二、补偿事务（TCC）：T 其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：① Try 阶段主要是对业务系统做检测及资源预留；② Confirm 阶段主要是对业务系统做确认提交，Try 阶段执行成功并开始执行 Confirm 阶段时，默认 Confirm 阶段是不会出错的。即：只要 Try 成功，Confirm 一定成功。③ Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。举个例子，假入 Bob 要向 Smith 转账，思路大概是：

1、首先在 Try 阶段，要先调用远程接口把 Smith 和 Bob 的钱给冻结起来。

2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。

3、如果第 2 步执行成功，转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。

优点：跟 2PC 比起来，实现以及流程相对简单了一些，但数据的一致性比 2PC 也要差一些

缺点：在 2,3 步中都有可能失败。所以需要程序员在实现的时候多写很多补偿的代码。

三、本地消息表（异步确保）：本地消息表这种实现方式应该是业界使用最多的，其核心思想是将分布式事务拆分成本地事务进行处理。基本思路就是：

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说他们要在一个数据库里面。然后消息会经过 MQ 发送到消息的消费方。如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

四、MQ 事务消息：有一些第三方的 MQ 是支持事务消息的，比如 RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交。

以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段 Prepared 消息，会拿到消息的地址。

第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发

送失败了 RocketMQ 会定期扫描消息集群中的事务消息，这时候发现了 Prepared 消息，它会向消息发送者确认，所以生产方需要实现一个 check 接口，RocketMQ 会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

6 集群与负载均衡的算法与实现

主要负载均衡方案：

一、HTTP 重定向负载均衡：这种负载均衡方式仅适合 WEB 服务器。用户发出请求时，负载均衡服务器会根据 HTTP 请求，重新计算出实际的 WEB 服务器地址，通过 302 重定向相应发送给用户浏览器。用户浏览器再根据 302 响应信息，对实际的 WEB 服务器发出请求。HTTP 重定向方案有点是比较简单，缺点是性能比较差，需要 2 次请求才能返回实际结果,还有就是仅适合 HTTP 服务器使用。

二、DNS 域名解析负载均衡：在 DNS 中存储了一个域名的多个主机地址，每次域名解析请求，都可以根据负载均衡算法返回一个不同的 IP 地址。这样多个 WEB 服务器就构成了一个集群，并由 DNS 服务器提供了负载均衡服务。DNS 域名解析负载均衡的优点是由 DNS 来完成负载均衡工作，服务本身不用维护负载均衡服务器的工作。缺点也是，由于负载均衡服务器不是自己维护，没法做精细控制，而且 DNS 在客户端往往带有缓存，服务器的变更很难及时反映到客户端上。

三、反向代理负载均衡：反向代理服务器位于实际的服务器之前，他能够缓存服务器响应，加速访问，同时也启到了负载均衡服务器的效果。反向代理服务器解析客户端请求，根据负载均衡算法转发到不同的后台服务器上。用户和后台服务器之间不再有直接的链接。请求，响应都由反向代理服务器进行转发。优点是和负载均衡服务集成在一起，部署简单。缺点是所有的请求回应都需要经过反向代理服务器。其本身可能会成为性能的瓶颈。著名的 Nginx 服务器就可以部署为反向代理服务器，实现 WEB 应用的负载均衡。

负载均衡算法：

1、轮询（默认）：每个请求按时间顺序逐一分配到不同的后端服务，如果后端某台服务器死机，自动剔除故障系统，使用户访问不受影响。

2、weight（轮询权值）：weight 的值越大分配到的访问概率越高，主要用于后端每台服务器性能不均衡的情况下。或者仅仅为在主从的情况下设置不同的权值，达到合理有效的地利用主机资源。

3、源地址哈希法：每个请求按访问 IP 的哈希结果分配，使来自同一个 IP 的访客固定访问一台后端服务器，并且可以有效解决动态网页存在的 session 共享问题。

4、随机法

5、最小连接数法：通过“最少连接"调度算法动态地将网络请求调度到已建立的链接数最少的服务器上如果集群系统的真实服务器具有相近的系统性能,采用“最小连接"调度算法可以较好地均衡负载

4、fair（第三方）【Nginx】：比 weight、ip_hash 更加智能的负载均衡算法，fair 算法可以根据页面大小和加载时间长短智能地进行负载均衡，也就是根据后端服务器的响应时间来分配请求，响应时间短的优先分配。Nginx 本身不支持 fair，需要安装 upstream_fair 模块。

5、url_hash（第三方）【Nginx】：按访问的 URL 的哈希结果来分配请求，使每个 URL 定向到一台后端服务器，可以进一步提高后端缓存服务器的效率。Nginx 本身不支持 url_hash，则安装 Nginx 的 hash 软件包。

7 说说分库与分表设计

垂直(纵向)拆分：是指按功能模块拆分，以解决表与表之间的 io 竞争。比如分为订单库、商品库、用户库...这种方式多个数据库之间的表结构不同。水平(横向)拆分：将同一个表的数据进行分块保存到不同的数据库中，来解决单表中数据量增长出现的压力。这些数据库中的表结构完全相同。

分库分表策略：1. 按照时间区间；2. 按照主键 ID 区间；3. 按照指定字段 hash 后再取模；4. 按照用户 ID 取模。

分库分表的策略相对于前边两种复杂一些，一种常见的路由策略如下：

1、中间变量 ＝ user_id%（库数量*每个库的表数量）;

2、库序号 ＝ 取整（中间变量 / 每个库的表数量）;

３、表序号 ＝ 中间变量％每个库的表数量;

例如：数据库有 256 个，每一个库中有 1024 个数据表，用户的 user_id＝262145，按照上述的路由策略，可得：

１、中间变量 ＝ 262145%（256*1024）= 1;

２、库序号 ＝ 取整（1 / 1024）= 0;

３、表序号 ＝ 1％1024 = 1;

这样的话，对于 user_id＝262145，将被路由到第 0 个数据库的第 1 个表中。

8 分库与分表带来的分布式困境与应对之策

分库分表需要解决的问题：

1、事务问题：

解决事务问题目前有两种可行的方案：分布式事务

方案一：使用分布式事务

优点：交由数据库管理，简单有效

缺点：性能代价高，特别是 shard 越来越多时

方案二：由应用程序和数据库共同控制

原理：将一个跨多个数据库的分布式事务分拆成多个仅处 于单个数据库上面的小事务，并通过应用程序来总控 各个小事务。

优点：性能上有优势

缺点：需要应用程序在事务控制上做灵活设计。

2、跨节点 Join、count、order by、group by 以及聚合函数问题

Join：分两次查询实现。在第一次查询的结果集中找出关联数据的 id,根据这些 id 发起第二次请求。

其他：分别在各个节点上得到结果后在应用程序端进行合并。可以并行执行。

3、数据迁移，容量规划，扩容等问题

利用对 2 的倍数取余具有向前兼容的特性（如对 4 取余得 1 的数对 2 取余也是 1）来分配数据，避免了行级别的数据迁移，但是依然需要进行表级别的迁移，同时对扩容规模和分表数量都有限制。

4、ID 问题

(1) UUID：UUID 是一个 128bit 长的数字，一般用 16 进制表示。全球唯一; UUID 往往是使用字符串存储，存储空间比较大，查询的效率比较低；不可读。

(2) Redis 生成 ID：利用 redis 的 lua 脚本执行功能，在每个节点上通过 lua 脚本生成唯一 ID。

(3)Twitter 的分布式自增 ID 算法 Snowflake：使用 41bit 作为毫秒数，10bit 作为机器的 ID，（5 个 bit 是数据中心，5 个 bit 的机器 ID），12bit 作为毫秒内的流水号，最前面还有一个符号位，永远是 0。

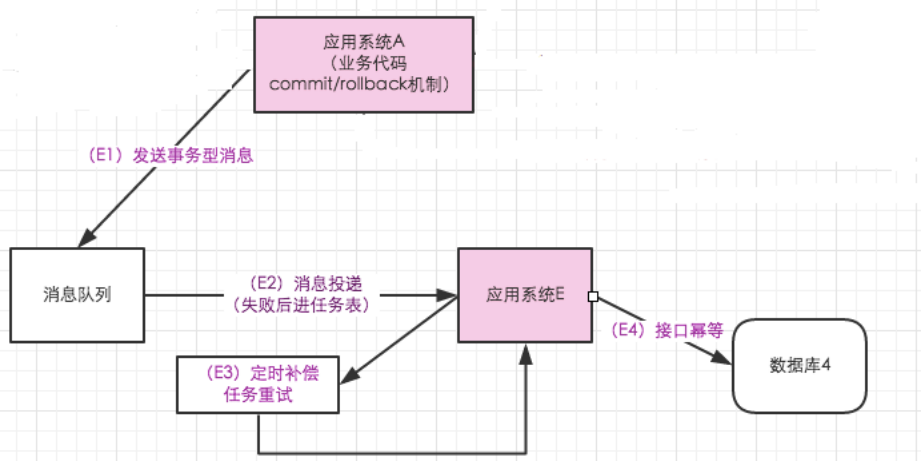
9 数据最终一致性方案

一、基于事务型消息队列的最终一致性

借助消息队列，在处理业务逻辑的地方，发送消息，业务逻辑处理成功后，提交消息，确保消息是发送成功的，之后消息队列投递来进行处理，如果成功，则结束，如果没有成功，则重试，直到成功，不过仅仅适用业务逻辑中，第一阶段成功，第二阶段必须成功的场景。

二、基于消息队列+定时补偿机制的最终一致性

前面部分和上面基于事务型消息的队列，不同的是，第二阶段重试的地方，不再是消息中间件自身的重试逻辑了，而是单独的补偿任务机制。其实在大多数的逻辑中，第二阶段失败的概率比较小，所以单独独立补偿任务表出来，可以更加清晰，能够比较明确的直到当前多少任务是失败的。



发送消息给消息中间件→消息中间件入库消息→消息中间件返回结果→业务操作→发送业务操作结果给消息中间件→更改存储中消息状态。

10 幂等性

幂等性：就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。更复杂的操作幂等保证是利用唯一交易号(流水号)实现。

1.查询：可以说是天然的幂等性，因为你查询一次和查询两次，对于系统来讲，没有任何数据的变更，所以，查询一次和查询多次一样的。

2.MVCC 方案：合理的选择乐观锁，通过 version 或者其他条件，来做乐观锁。乐观锁的实现方式多种多样可以通过 version 或者其他状态条件：①通过版本号实现；②2. 通过条件限制。

update table_xxx set name=#name#,version=version+1 where id=#id# and version=#version#

update table_xxx set amount= amount-#subAmount# where id=#id# and amount-#subAmount# >= 0

3.单独的去重表：如果涉及到的去重的地方特别多，可以单独搞一张去重表，在插入数据的时候，插入去重表，利用数据库的唯一索引特性，保证唯一的逻辑。

4. 分布式锁：在业务系统插入数据或者更新数据，获取分布式锁，然后做操作，之后释放锁

5. 删除操作：删除数据，仅仅第一次删除是真正的操作数据，第二次甚至第三次删除，直接返回成功。

6.插入数据的唯一索引：插入数据的唯一性，可以通过业务主键来进行约束，例如一个特定的业务场景，三个字段肯定确定唯一性，那么，可以在数据库表添加唯一索引来进行标示。

7. token 机制，防止页面重复提交：①数据提交前要向服务的申请 token，token 放到 redis 或 jvm 内存，token 有效时间；②提交后后台校验 token，同时删除 token，生成新的 token 返回

8. 对外提供接口的 api：对外提供接口为了支持幂等调用，接口有两个字段必须传，一个是来源 source，一个是来源方序列号 seq，这个两个字段在提供方系统里面做联合唯一索引，先在本方系统里面查询一下，是否已经处理过，返回相应处理结果；没有处理过，进行相应处理，返回结果。

9.状态机幂等：如果状态机已经处于下一个状态，这时候来了一个上一个状态的变更，理论上是不能够变更的，这样的话，保证了有限状态机的幂等。

11 消息队列丢消息怎么处理

12 zookeeper 选举算法

Spring

1 BeanFactory 和 ApplicationContext 有什么区别

BeanFactory：是 IOC 容器的核心接口，它定义了 IOC 的基本功能，我们看到它主要定义了 getBean 方法。getBean 方法是 IOC 容器获取 bean 对象和引发依赖注入的起点。方法的功能是返回特定的名称的 Bean。注意，BeanFactory 只能管理单例（Singleton）Bean 的生命周期。它不能管理原型(prototype,非单例)Bean 的生命周期。这是因为原型 Bean 实例被创建之后便被传给了客户端,容器失去了对它们的引用。

ApplicationContext：ApplicationContext 由 BeanFactory 派生而来，提供了更多面向实际应用的功能。在 BeanFactory 中，很多功能需要以编程的方式实现，而在 ApplicationContext 中则可以通过配置实现。ApplicationContext 还在功能上做了扩展，相较于 BeanFactorty，ApplicationContext 还提供了以下的功能：1.支持信息源，可以实现国际化。（实现 MessageSource 接口）； 2.访问资源。(实现 ResourcePatternResolver 接口，这个后面要讲)； 3.支持应用事件。(实现 ApplicationEventPublisher 接口)。

FactoryBean：在使用容器时，可以使用转义符“&”来得到 FactoryBean 本身，用来区分通过容器来获取 FactoryBean 产生的对象和获取 FactoryBean 本身。FactoryBean 不是简单的 Bean，是一个能产生或者修饰对象生成的工厂 Bean，**它的实现与设计模式中的工厂模式和修饰器模式类似。**

区别：1.BeanFactroy 采用的是延迟加载形式来注入 Bean 的，即只有在使用到某个 Bean 时(调用 getBean())，才对该 Bean 进行加载实例化。而 ApplicationContext 则相反，它是在容器启动时，一次性创建了所有的 Bean。占用内存空间。当应用程序配置 Bean 较多时，程序启动较慢。

2.BeanFactory 和 ApplicationContext 都支持 BeanPostProcessor、BeanFactoryPostProcessor 的使用，但两者之间的区别是：BeanFactory 需要手动注册，而 ApplicationContext 则是自动注册。

3.作用：BeanFactory 负责读取 bean 配置文档，管理 bean 的加载，实例化，维护 bean 之间的依赖关系，负责 bean 的声明周期。ApplicationContext 除了提供上述 BeanFactory 所能提供的功能之外，还提供了更完整的框架功能。

2 Spring Bean 的生命周期

1. spring 对 bean 进行实例化，由 BeanFactory 读取 Bean 定义文件，并生成各个实例，默认 bean 是单例；
2. 按照 Spring 上下文对实例化的 Bean 进行配置，也就是 IOC 注入；
3. 如果 Bean 实现了 BeanNameAware 接口，会调用它实现的 **setBeanName**(String beanId)方法，此处传递的是 Spring 配置文件中 Bean 的 ID；
4. 如果 Bean 实现了 BeanFactoryAware 接口，会调用它实现的 **setBeanFactory**()，将 BeanFactory 实例传进来，传递的是 Spring 工厂本身（可以用这个方法获取到其他 Bean）；
5. 如果 Bean 实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext)方法，传入 Spring 上下文；
6. 如果 Bean 实现了 BeanPostProcessor 接口，将会调用 **postProcessBeforeInitialization**(Object obj, String s)方法，BeanPostProcessor 经常被用作是 Bean 内容的更改，并且由于这个是在 Bean 初始化结束时调用 After 方法，也可用于内存或缓存技术；
7. 如果 Bean 在 Spring 配置文件中配置了 **init-method** 属性会自动调用其配置的初始化方法；
8. 如果 Bean 实现了 BeanPostProcessor 接口，将会调用 **postAfterInitialization**(Object obj, String s)方法；

注意：以上工作完成以后就可以用这个 Bean 了，那这个 Bean 是一个 single 的，所以一般情况下我们调用同一个 ID 的 Bean 会是在内容地址相同的实例

9. 当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现 DisposableBean 接口，调用其实现的 destroy 方法；
10. 最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置的销毁方法。

以上 10 步骤可以作为面试或者笔试的模板，另外我们这里描述的是应用 Spring 上下文 Bean 的生命周期，如果应用 Spring 的工厂也就是 BeanFactory 的话去掉第 5 步就 Ok 了。一旦把一个 Bean 纳入 Spring IOC

容器之中，这个 Bean 的生命周期就会交由容器进行管理，一般担当管理角色的是 BeanFactory 或者 ApplicationContext。

3 spring 的 initial bean 具体怎么实现的

(1)容器启动时：首先调用 BeanFactoryPostProcessor ->postProcessBeanFactory()
(2)getBean 时：实例化之后调用： InstantiationAwareBeanPostProcessor
->postProcessPropertyValues()
(3)初始化时：
①设置属性值；
②调用 Bean 中的 BeanNameAware.setBeanName()方法，如果(实现了 BeanNameAware 接口)；
③调用 Bean 中的 BeanFactoryAware.setBeanFactory()方法，如果(实现 BeanFactoryAware 接口)；
④调用 BeanPostProcessors.postProcessBeforeInitialization()方法；@PostConstruct 注解后的方法就是在这里被执行的；
⑤调用 Bean 中的 afterPropertiesSet 方法，如果该 Bean 实现了 InitializingBean 接口；
⑥调用 Bean 中的 init-method，通常是在配置 bean 的时候指定了 init-method，例如：<bean class="beanClass" init-method="init"></bean>
⑦调用 BeanPostProcessors.postProcessAfterInitialization()方法；
⑧如果该 Bean 是单例的，则当容器销毁并且该 Bean 实现了 DisposableBean 接口的时候，调用 destory 方法；
如果该 Bean 是 prototype，则将准备好的 Bean 提交给调用者，后续不再管理该 Bean 的生命周期。

4 容器初始化

5 Spring IOC

(1)什么是 IOC/DI：所谓 IoC，对于 spring 框架来说，就是由 spring 来负责控制对象的生命周期和对象间的关系。在传统的 java 应用开发中，我们要实现某一个功能至少需要两个或以上的对象来协作完成，在没有使用 Spring 的时候，每个对象在需要使用他的合作对象时，自己均要使用像 **new object()** 这样的语法

来将合作对象创建出来，这个合作对象是由自己主动创建出来的，创建合作对象的主动权在自己手上，自己需要哪个合作对象，就主动去创建，创建合作对象的主动权和创建时机是由自己把控的，而这样就会使得对象间的耦合度高了。而使用了 Spring 之后就不一样了，创建合作对象 B 的工作是由 Spring 来做的，Spring 创建好 B 对象，然后存储到一个容器里面，当 A 对象需要使用 B 对象时，Spring 就从存放对象的那个容器里面取出 A 要使用的那个 B 对象，然后交给 A 对象使用，至于 Spring 是如何创建那个对象，以及什么时候创建好对象的，A 对象不需要关心这些细节问题(你是什么时候生的，怎么生出来的我可不关心，能帮我干活就行)。

DI 是由 Martin Fowler 在 2004 年初的一篇论文中首次提出的。他总结：控制的什么被反转了？就是：**获得依赖对象的方式反转**了。IoC 的一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过 DI（Dependency Injection，依赖注入）来实现的。

(2) **IoC 容器的初始化**：IoC 容器的初始化是由 refresh()方法来启动的，这个方法标志着 IoC 容器的正式启动，具体包括①BeanDefinition 的 Resource 定位；②载入；③注册这三个基本的过程。第一个过程是 Resource 定位过程，这个 Resource 定位指的是 BeanDefinition 的资源定位，它由 ResourceLoader 通过统一的 Resource 接口来完成；第二个过程是 BeanDefinition 的载入，这个载入过程是把用户定义好的 Bean 表示成 IoC 容器内部的数据结构；第三个过程是想 IoC 容器注册这些 BeanDefinition 的过程，这个过程是通过调用 BeanDefinitionRegistry 接口的实现来完成（将 BeanDefinition 注入到一个 HashMap 中）。

下面以 FileSystemXmlApplicationContext 为例：

Resource 定位过程：①FileSystemXmlApplicationContext 构造器：首先，调用父类容器的构造方法为容器设置好 Bean 资源加载器；然后，再调用父类 AbstractRefreshableConfigApplicationContext 的方法设置 Bean 定义资源文件的定位路径。然后调用父类 AbstractApplicationContext. refresh()方法实现 BeanFactory 的更新；**②refresh()是一个模板方法**，refresh()方法的作用是：在创建 IoC 容器前，如果已经有容器存在，则需要把已有的容器销毁和关闭，以保证在 refresh 之后使用的是新建立起来的 IoC 容器。调用 **obtainFreshBeanFactory 函数载入**；③调用**子类 refreshBeanFactory** 函数；④refreshBeanFactory 首先判断 BeanFactory 是否存在，如果存在则先销毁 beans 并关闭 beanFactory，接着调用 createBeanFactory()创建 IoC 容器，使用的是 DefaultListableBeanFactory；然后调用**子类 AbstractXmlApplicationContext 的** loadBeanDefinitions(beanFactory)装载 bean；**⑤loadBeanDefinitions**：创建 **XmlBeanDefinitionReader**，即创建 Bean 读取器，容器使用该读取器读取 Bean 定义资源；调用其父类 AbstractBeanDefinitionReader 的 reader.loadBeanDefinitions 方法读取 Bean 定义资源；⑥loadBeanDefinitions：首先调用 DefaultResourceLoader 的 getResource 完成具体的 Resource 定位；然后调用其子类 XmlBeanDefinitionReader 的 loadBeanDefinitions 方法真正执行加载功能。

DefaultResourceLoader 中的 getSource()方法，因为 FileSystemXmlApplicationContext 本身就是 DefaultResourceLoader 的实现类，所以此时又回到了 FileSystemXmlApplicationContext 中来。

BeanDefinition 的载入和解析：①XmlBeanDefinitionReader 加载 Bean 定义资源。首先从特定 XML 文件中实际载入 Bean 定义资源，将 XML 文件转换为 DOM 对象，解析过程由 documentLoader 实现；然后按照 Spring 的 Bean 规则对 Document 对象解析的过程是在接口实现类 **DefaultBeanDefinitionDocumentReader** 中实现的。经过对 Spring Bean 定义资源文件转换的 Document 对象中的元素层层解析，将 XML 形式定义的 Bean 定义资源文件转换为 Spring IoC 所识别的数据结构——BeanDefinition。

注册：真正完成注册功能的是 DefaultListableBeanFactory。使用一个 HashMap 的集合对象存放。注册的过程中需要线程同步，以保证数据的一致性。

总结：①初始化的入口在容器实现中的 refresh()调用来完成；②对 bean 定义载入 IOC 容器使用的方法是 loadBeanDefinition,其中的大致过程如下：通过 ResourceLoader 来完成资源文件位置的定位，DefaultResourceLoader 是默认的实现，同时上下文本身就给出了 ResourceLoader 的实现，可以从类路径，文件系统, URL 等方式来定为资源位置。如果是 XmlBeanFactory 作为 IOC 容器，那么需要为它指定 bean 定义的资源，也就是说 bean 定义文件时通过抽象成 Resource 来被 IOC 容器处理的，容器通过 BeanDefinitionReader 来完成定义信息的解析和 Bean 信息的注册,往往使用的是 XmlBeanDefinitionReader

来解析 bean 的 xml 定义文件 - 实际的处理过程是委托给 BeanDefinitionParserDelegate 来完成的，从而得到 bean 的定义信息，这些信息在 Spring 中使用 BeanDefinition 对象来表示，容器解析得到 BeanDefinition 以后，需要把它在 IOC 容器中注册，这由 IOC 实现 BeanDefinitionRegistry 接口来实现。注册过程就是在 IOC 容器内部维护的一个 HashMap 来保存得到的 BeanDefinition 的过程。这个 HashMap 是 IoC 容器持有 bean 信息的场所，以后对 bean 的操作都是围绕这个 HashMap 来实现的。

(3)依赖注入：依赖注入在以下两种情况发生：①用户第一次通过 getBean 方法向 IoC 容索要 Bean 时，IoC 容器触发依赖注入。②当用户在 Bean 定义资源中为<Bean>元素配置了 lazy-init 属性，即让容器在解析注册 Bean 定义时进行预实例化，触发依赖注入。

①在 BeanFactory 中我们看到 getBean（String…）函数，它的具体实现在 AbstractBeanFactory 中，AbstractBeanFactory 通过 getBean 向 IoC 容器获取被管理的 Bean：如果 Bean 定义的单态模式(Singleton)，则容器在创建之前先从缓存中查找，确保整个容器中只存在一个实例对象。如果 Bean 定义的是原型模式，则容器每次都会创建一个新的实例对象。除此之外，Bean 定义还可以扩展为指定其生命周期范围。

②具体的 Bean 实例对象的创建过程由实现了 ObejctFactory 接口的匿名内部类的 createBean 方法完成。ObejctFactory 使用委派模式，具体的 Bean 实例创建过程交由其实现类 **AbstractAutowireCapableBeanFactory** 完成，调用 createBeanInstance 生成 Bean 所包含的 java 对象实例，**调用 populateBean 对 Bean 属性的依赖注入进行处理。**

③在 createBeanInstance 方法中，根据指定的初始化策略，使用静态工厂、工厂方法或者容器的自动装配特性生成 java 实例对象。调用相应的工厂方法或者参数匹配的构造方法即可完成实例化对象的工作，但是对于我们最常使用的默认无参构造方法就需要使用相应的初始化策略(JDK 的反射机制或者 CGLIB)来进行初始化了。

④populateBean：调用 applyPropertyValues 对属性进行注入：当属性值类型不需要转换时，直接准备进行依赖注入；当属性值需要进行类型转换时，如对其他对象的引用等，首先需要解析属性值，由 **resolveValueIfNecessary** 方法实现；然后对解析后的属性值通过 BeanWrapper.setPropertyValues 进行依赖注入，具体实现在 BeanWrapperImpl。

Spring IoC 容器是如何将属性的值注入到 Bean 实例对象中去的：

(1).对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性。

(2).对于非集合类型的属性，大量使用了 JDK 的反射和内省机制，通过属性的 getter 方法(reader method)获取指定属性注入以前的值，同时调用属性的 setter 方法(writer method)为属性设置注入后的值。看到这里相信很多人都明白了 Spring 的 **setter 注入**原理。

6 Autowired

容器对 Bean 实例对象的属性注入的处理发生在 AbstractAutoWireCapableBeanFactory 类中的 populateBean 方法中，我们通过程序流程分析 autowiring 的实现原理：

(1). AbstractAutoWireCapableBeanFactory 对 Bean 实例进行属性依赖注入：应用第一次通过 getBean 方法(配置了 lazy-init 预实例化属性的除外)向 IoC 容器索取 Bean 时，容器创建 Bean 实例对象，并且对 Bean 实例对象进行属性依赖注入，populateBean 方法就是实现 Bean 属性依赖注入的功能.

(2).Spring IoC 容器根据 Bean 名称或者类型进行 autowiring 自动依赖注入：

通过属性名进行自动依赖注入的相对比通过属性类型进行自动依赖注入要稍微简单一些，但是真正实现属性注入的是 DefaultSingletonBeanRegistry 类的 registerDependentBean 方法。

(3).DefaultSingletonBeanRegistry 的 registerDependentBean 方法对属性注入：

通过对 autowiring 的源码分析，我们可以看出，autowiring 的实现过程：

a.对 Bean 的属性迭代调用 getBean 方法，完成依赖 Bean 的初始化和依赖注入。

b.将依赖 Bean 的属性引用设置到被依赖的 Bean 属性上。

c.将依赖 Bean 的名称和被依赖 Bean 的名称存储在 IoC 容器的集合中。

7 Spring AOP

Spring 提供了两种方式来生成代理对象: **JDKProxy 和 Cglib**，具体使用哪种方式生成由 AopProxyFactory 根据 AdvisedSupport 对象的配置来决定。默认的策略是如果目标类是接口，则使用 JDK 动态代理技术，否则使用 Cglib 来生成代理。

①AbstractAutowireCapableBeanFactory 类的 initializeBean 方法会对创建完成的 Bean 进行初始化，然后调用 postProcessAfterInitialization()为容器产生的 Bean 实例对象添加 BeanPostProcessor 后置处理器，具体由 BeanPostProcessor 的子类实现。②AbstractAutoProxyCreator.java 类的 postProcessAfterInitialization 方法中，调用 wrapIfNecessary;③wrapIfNecessary 中 Spring 在这里先获取配置好的 Advisor 信息,然后调用 createProxy 方法为目标对象创建了代理，接着将创建的代理对象返回。

(1) **初始化 AopProxy：**①ProxyCreatorSupport.java 类的 createAopProxy 方法。这就是生成代理的入口，你会发现传入的参数是是 this，其实传入的就是 this 的父类 AdvisedSupport.java 中的 advisor 等生成代理的核心参数。②调用 AopProxyFactory 的 createAopProxy，实际调用子类 DefaultAopProxyFactory；③根据代理的目标对象是否实现了接口，来返回 JdkDynamicAopProxy 的动态代理或者 CGLIB 的代理，并且传入 advisor 核心参数（JdkDynamicAopProxy 这个实现了 InvocationHandler，要实现 invoke 的关键就是传入的 advisor）。

(2)获取代理对象：getProxy 方法，使用 java reflect 包提供的原生创建代理类方法 Proxy.newProxyInstance 方法创建并返回目标对象的代理对象。

6 动态代理（cglib 与 JDK 动态代理）

```
public interface Service {
    public void add();
}
```

```
public class UserServiceImpl implements Service {
    @Override
    public void add() {
        System.out.println("This is add service");
    }
}
```

```
public static void main(String[] args) {
    Service service = new UserServiceImpl();
    MyInvocatioHandler handler = new
                                MyInvocatioHandler(service);
        Service serviceProxy = (Service)handler.getProxy();
        serviceProxy.add();
}
```

```
public static void main(String[] args) {
    UserServiceImpl service = new UserServiceImpl();
    CGLIBProxy cglib = new CGLIBProxy();// 创建 CglibProxy 对象
    UserServiceImpl proxy = (UserServiceImpl)
                        cglib.getProxy(service);  //17 生成代理类
    proxy.add();
}
```

6.1 动态代理

```
class MyInvocatioHandler implements InvocationHandler {
    private Object target;
    public MyInvocatioHandler(Object 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("-----end-----");
```

```
            return result;
        }
        // 生成代理对象
        public Object getProxy() {
            ClassLoader loader = Thread.currentThread().getContextClassLoader();
            Class<?>[] interfaces = target.getClass().getInterfaces();
            return Proxy.newProxyInstance(loader, interfaces, this);
        }
}
```

它的好处理时可以为我们生成任何一个接口的代理类，并将需要增强的方法织入到任意目标函数。但它仍然具有一个局限性，就是**只有实现了接口的类，才能为其实现代理**。

6.2 CGLIB

```
public class CGLIBProxy implements MethodInterceptor {
    private Object target;
    public Object getProxy(Object target) {
        this.target = target;  //给业务对象赋值
        Enhancer enhancer = new Enhancer(); //创建加强器，用来创建动态代理类
        enhancer.setSuperclass(this.target.getClass());  //为加强器指定要代理的业务类（即：为下面生成的代理类指定父类）
        enhancer.setCallback(this);    //设置回调：对于代理类上所有方法的调用，都会调用 CallBack
        return enhancer.create();    // 创建动态代理类对象并返回
    }
    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {
        System.out.println("预处理————————");
        Object result = proxy.invokeSuper(obj, args);
        System.out.println("调用后操作————————");
        return result;
    }
}
```

CGLIB 解决了动态代理的难题，它通过生成目标类子类的方式来实现来实现代理，而不是接口，规避了接口的局限性。当然 CGLIB 也具有局限性，对于无法生成子类的类（final 类），肯定是没有办法生成代理子类的。

7 Spring 事务实现方式

①引入一系列的约束头文件以及标签；②配置 C3P0 数据源以及 DAO、Service；③配置事务管理器以及事务代理工厂 Bean。事务管理器：DataSourceTransactionManager

方式一：通过事务代理工厂 bean 进行配置[XML 方式]：事务代理工厂：注入事务管理器，隔离级别等

方式二：注解：@Transactional

方式三：Aspectj AOP 配置事务：<tx:advice>事务管理传播配置

8 Spring 事务

(1) 数据库事务的四大特性：

ACID，指数据库事务正确执行的四个基本要素的缩写。包含：

- ✧ 原子性（Atomicity）：原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚；
- ✧ 一致性（Consistency）：一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态；

- ✧ 隔离性（Isolation）：隔离性是当多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。
- ✧ 持久性（Durability）。持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

一个支持事务（Transaction）的数据库，必须要具有这四种特性。

(2) 数据库隔离级别：

- ✧ READ UNCIMMITTED（未提交读）：事务还没提交，而别的事务可以看到修改的数据，也就是脏读；
- ✧ READ COMMITTED（提交读）：只能看到已经完成的事务的结果，正在执行的，是无法被其他事务看到的。可防止脏读，但幻读和不可重复读仍可发生；
- ✧ REPEATABLE READ（可重复读）：对相同字段的多次读取是一致的，除非数据被事务本身改变。可防止脏、不可重复读，但幻读仍可能发生
- ✧ SERIALIZABLE（可串行化）：完全服从 ACID 的隔离级别，确保不发生脏、幻、不可重复读。这在所有的隔离级别中是最慢的，它是典型的通过完全锁定在事务中涉及的数据表来完成的。

MySQL 默认采用 REPEATABLE_READ 隔离级别；Oracle 默认采用 READ_COMMITTED 隔离级别

(3) 事务传播行为：（七种）

REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。

MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。

REQUIRES_NEW--新建事务，如果当前存在事务，把当前事务挂起。

NOT_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。

NESTED--如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与 REQUIRED 类似的操作。拥有多个可以回滚的保存点，内部回滚不会对外部事务产生影响。

9 Spring MVC 运行流程

①客户端发出一个 http 请求给 web 服务器，web 服务器对 http 请求进行解析，如果匹配 DispatcherServlet 的请求映射路径（在 web.xml 中指定），web 容器将请求转交给 **DispatcherServlet**。

②DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器，找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

③DispatcherServlet 调用 HandlerAdapter 处理器适配器。

④HandlerAdapter 经过适配调用具体的处理器(Controller，也叫后端控制器)。

⑤Controller 执行完成返回 ModelAndView。

⑥HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。

⑦DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。

⑧ViewReslover 解析后返回具体 View。

⑨DispatcherServlet 根据 View 进行渲染视图（即将模型数据填充至视图中）。

⑩DispatcherServlet 响应用户。

10 Spring 的单例实现原理

Spring 的依赖注入（包括 lazy-init 方式）都是发生在 **AbstractBeanFactory** 的 getBean 里。 getBean 的 doGetBean 方法调用 getSingleton 进行 bean 的创建。lazy-init 方式，在容器初始化时候进行调用，非 lazy-init 方式，在用户向容器第一次索要 bean 时进行调用。

1. **doGetBean**：(1)根据指定的名称获取被管理 Bean 的名称，如果指定的是别名，将别名转换为规范的 Bean 名称；(2)调用 **getSingleton**()先从缓存中取是否已经有被创建过的单态类型的 Bean(IoC 容器中只创建一次)；(3)如果指定名称的 Bean 在容器中已有单态模式的 Bean 被创建，直接返回；(4)如果没有，委托当前容

器的父级容器去查找，如果找到则返回；(5)如果一直找不到，就创建 bean：①先获取其父级的 Bean 定义，主要解决 B 继承时子类合并父类公共属性；②获取当前 Bean 所有依赖 Bean 的名称；③createBean()创建 bean。

2. **getSingleton**：使用了**双重判断加锁**的单例模式。(1)首先从缓存 singletonObjects（实际上是一个 map，存储的是完全实例化的 BeanName->Bean Instance）中获取 bean 实例；(2)如果为 null，但该 bean 正在创建过程中，则对缓存 singletonObjects **加锁**；(3) 然后再从尝试从 earlySingletonObjects 中获取。如果没有，则从 singletonFactories 中获取。这是因为 spring 创建单例 bean 的时候，存在循环依赖的问题，为了解决循环依赖的问题，spring 采取了一种将创建的 bean 实例提早暴露加入到缓存中，一旦下一个 bean 创建的时候需要依赖上个 bean，则直接使用 ObjectFactory 来获取 bean。提前暴露 bean 实例到缓存的时机是在 bean 实例创建（调用构造方法）之后，初始化 bean 实例（属性注入）之前。在从 singletonFactories 获取 bean 后，会将其存储到 earlySingletonObjects 中，然后从 singletonFactories 移除该 bean，之后在要获取该 bean 就直接从 earlySingletonObjects 获取。(4)如果找到就返回 bean 实例，否则返回 null。

3. **doCreateBean**：如果 Bean 定义的单态模式(Singleton)，则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。如果 Bean 定义的是原型模式(Prototype)，则容器每次都会创建一个新的实例对象。除此之外，Bean 定义还可以扩展为指定其生命周期范围。生成 Bean 所包含的 java 对象实例；对 Bean 属性的依赖注入进行处理。

11 Spring 框架中用到了哪些设计模式

(1) 单例模式：AbstractBeanFactory 通过 getBean 方法向 IoC 容器索要 Bean 时默认使用单例模式；

(2) 代理模式：Spring Aop 中 Jdk 动态代理就是利用代理模式技术实现的；

(3) 工厂模式：Spring Bean 的创建是典型的工厂模式。

(4) 装饰器：FactoryBean 是一个能产生或者修饰对象生成的工厂 Bean，它的实现与设计模式中的工厂模式和装饰器模式类似。

(5) 解释器:Spring 主要以 Spring Expression Language（SpEL）为例

(6) 模板模式：JdbcTemplate；

(7) 策略模式：FactoryBean 接口为 Spring 容器提供了一个很好的封装机制，具体的 getObject 有不同的实现类根据不同的实现策略来具体提供。

12 spring 注入 static 变量

(1) spring 不允许/不支持把值注入到静态变量中，@Value("${mongodb.db}")会得到 null；解决方法有：

①可以利用非静态 setter 方法注入静态变量。

```
@Value("${mongodb.db}")
public void setDatabase(String db) {
    DATABASE = db;
}
```

②使用配置文件的方式注入：

```
<bean class="TestStatic">
    <property name="from" value="abc"/>
</bean>
```

(2) 注入的是 static 类：

①配置文件

```
<bean id="mongoFileOperationUtil" class="com.*.*.MongoFileOperationUtil" init-method="init">
    <property name="dsForRW" ref="dsForRW"/>
</bean>
```

②@PostConstruct 方式实现：

```
@Component
public class MongoFileOperationUtil {
    @Autowired
    private static AdvancedDatastore dsForRW;
```

```
    private static MongoFileOperationUtil mongoFileOperationUtil;
    @PostConstruct
    public void init() {
        mongoFileOperationUtil = this;
        mongoFileOperationUtil.dsForRW = this.dsForRW;
    }
}
```

③3.set 方法上添加@Autowired 注解，类定义上添加@Component 注解；

```
    private static AdvancedDatastore dsForRW;
    @Autowired
    public void setDatastore(AdvancedDatastore dsForRW) {
        MongoFileOperationUtil.dsForRW = dsForRW;
    }
```

多线程

1 创建线程的方式及实现

Java 中创建线程主要有三种方式：

(1)继承 Thread 类创建线程类，并重写该类的 run 方法；

```
public class FirstThread extends Thread {
    private int i;
    public void run() {
        for (i = 0; i < 10; i++) {
            System.out.println(getName() + "    " + i);
        }
    }
    public static void main(String[] args) {
        new FirstThread ().start();
        new FirstThread ().start();
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + "    : " + i);
        }
    }
}
```

(2)实现 Runnable 接口创建线程类，并重写该接口的 run()方法；

```
public class RunnableThreadTest implements Runnable {
    private int i;
    public void run() {
        for (i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + " " + i);
        }
    }
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + " " + i);
        }
        RunnableThreadTest rtt = new RunnableThreadTest();
        new Thread(rtt, "新线程 1").start();
        new Thread(rtt, "新线程 2").start();
    }
}
```

(3) 通过 Callable 和 Future 创建线程；并实现 call()方法。

```
public class CallableThreadTest implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        int i = 0;
        for (; i < 10; i++) {
```

```
                System.out.println(Thread.currentThread().getName() + " " + i);
            }
            return i;
        }
        public static void main(String[] args) {
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.currentThread().getName() + "  的循环变量 i 的值  " + i);
            }
            CallableThreadTest ctt = new CallableThreadTest();
            FutureTask<Integer> ft = new FutureTask<Integer>(ctt);
            new Thread(ft, "有返回值的线程").start();
            try {
                System.out.println("子线程的返回值" + ft.get()); // get()方法来获得子线程执行结束后的返回值
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

对比：

(1) 采用接口的方式，还可以继承其他类。在这种方式下，多个线程可以共享同一个 target 对象，所以非常适合多个相同线程来处理同一份资源的情况。如果要访问当前线程，则必须使用 Thread.currentThread()方法。

(2) 使用继承 Thread 类的方式，如果需要访问当前线程，则无需使用 Thread.currentThread()方法，直接使用 this 即可获得当前线程。

2 sleep 、join、yield 有什么区别

(1) sleep

Thread 类的方法，必须带一个时间参数，它可以让当前正在执行的线程在指定的时间内暂停执行，进入阻塞状态并释放 CPU，提供其他线程运行的机会且不考虑优先级，但不会释放“锁标志”，也就是说如果有 synchronized 同步块，其他线程不能获得同步锁。

(2) yield

谦让：Thread 类的方法，类似 sleep 但无法指定时间并且只会提供相同或更高优先级的线程运行的机会，也不会释放“锁标志”，不推荐使用。

(3) wait

Object 类的方法，必须放在循环体和同步代码块中，执行该方法的线程会释放锁，进入线程等待池中等待被再次唤醒(notify 随机唤醒，notifyAll 全部唤醒，线程结束自动唤醒)即放入锁池中竞争同步锁；

线程 A 调用 wait()，A 就转为等待状态；线程 A 调用 notify，其他线程被唤醒争抢资源；

(4) join

一种特殊的 wait，当前运行线程调用另一个线程的 join 方法，当前线程进入阻塞状态直到另一个线程运行结束；在线程 A 中调用 B.join() 表示阻塞 A 线程，直到线程 B 运行结束或超时。

3 无锁操作-CAS

(1) Compare and Swap，即比较并交换。

java.util.concurrent 包中借助 CAS 实现了区别于 **synchronized** 同步锁的一种乐观锁。整个 AQS 同步组件、Atomic 原子类操作等等都是以 CAS 实现的。CAS 有 3 个操作数，内存值 V，旧的预期值 A，要修改的新值 B。当且仅当预期值 A 和内存值 V 相同时，将内存值 V 修改为 B，否则说明已经有其他线程做了更新，什么都

不做。

CAS 通过调用 JNI 的代码实现的。JNI:Java Native Interface 为 JAVA 本地调用，允许 java 调用其他语言。而 compareAndSwapInt(有四个参数，分别代表：对象、对象的地址、预期值、修改值)就是借助 C 来调用 CPU 底层指令实现的。因为 CAS 是一种系统原语，原语属于操作系统用语范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许被中断，也就是说 CAS 是一条 CPU 的原子指令，不会造成所谓的数据不一致问题。

(2) CAS 实现原子操作的三大问题

①ABA 问题：

CAS 需要检查值有没有发生变化，如果没有变化则更新，但是如果一个值**原来是 A，变成 B，又变成了 A**，那么使用 CAS 进行检查时会发现它的值没有发生变化，实际上变化了。

ABA 问题的解决思路就是：使用版本号。在变量前面追加上版本号，每次改变时加 1，即 A —> B —> A，变成 1A —> 2B —> 3A。

从 JDK1.5 开始，JDK 的 Atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。通过 Pair(AtomicStampedReference 的内部类)，主要用于记录引用和版本戳信息包装，从而避免 ABA 问题。这个类的 compareAndSet(有四个参数，分别表示：预期引用、更新后的引用、预期标志、更新后的标志)方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式设置更新值。

注：atomicStampedReference.compareAndSet(100, 110,

atomicStampedReference.getStamp(), atomicStampedReference.getStamp() + 1);

②循环时间长开销大

自旋 CAS 如果长时间不成功，会给 CPU 带来非常大的执行开销。

③只能保证一个共享变量的原子操作

对多个共享变量操作时，循环 CAS 无法保证操作的原子性，这个时候可以用锁。从 JDK1.5 开始，JDK 提供了 **AtomicReference 类来保证引用对象之间的原子性**，可以把多个变量放在一个对象里来进行 CAS 操作。

4 轻级的锁-volatile

(1) Volatile 是轻量级的 synchronized，它在多处理器开发中保证了共享变量的“可见性”。即当一个线程修改了一个被 volatile 修饰共享变量的值，新值可以被其他线程立即得知。确保所有线程看到这个变量的值是一致的。Volatile 不会引起线程上下文的切换和调度。

(2) **volatile 的特性：**①可见性：当读一个 volatile 变量时，JMM 会把该线程对应的本地内存置为无效。线程从主内存中读取共享变量。**②原子性**：对任意单个 volatile 变量的读/写具有原子性，但类似于 volatile++ 这种复合操作不具有原子性。**③禁止指令重排序优化【**1.5 才完全恢复**】**。

(3) Volatile 底层实现：在 JVM 底层 volatile 是采用“内存屏障”来实现的。volatile 关键字时，会多出一个 lock 前缀指令。lock 前缀指令其实就相当于一个内存屏障。内存屏障会提供 3 个功能：①如果第一个操作为 volatile 读，则不管第二个操作是啥，都不能重排序。当第二个操作为 volatile 写是，则不管第一个操作是啥，都不能重排序。当第一个操作 volatile 写，第二操作为 volatile 读时，不能重排序；②它会强制将对缓存的修改操作立即写入主存；③如果是写操作，它会导致其他 CPU 中对应的缓存行无效。

5 重量级的锁-synchronized

JavaSE1.6 对 synchronized 进行了各种优化，有些情况下他就并不那么重了。

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。

(1) **Java 中的每一个对象都可以作为锁**

对于普通同步方法，锁是当前实例对象

对于静态同步方法，锁是当前类的 class 对象

对于同步方法块，锁是 synchronized 括号里配置的对象

synchronized(this)以及非 static 的 synchronized 方法，只能防止多个线程同时执行同一个对象的同步代码段。**synchronized 锁住的是括号里的对象，而不是代码。对于非 static 的 synchronized 方法，锁的就是对象本身也就是 this。**

用 synchronized(Sync.class)实现了全局锁的效果。

static synchronized 方法，static 方法可以直接类名加方法名调用，方法中无法使用 this，所以它锁的不是 this，而是类的 Class 对象，所以，static synchronized 方法也相当于全局锁，相当于锁住了代码段。

(2) synchronized 底层原理

Java 虚拟机中的同步(Synchronization)基于进入和退出管程(Monitor)对象实现。同步代码块是使用 monitorenter 和 monitorexit 指令实现的，而同步方法并非如此，是由方法调用指令读取运行时常量池中方法的 ACC_SYNCHRONIZED 标志来隐式实现的。

①同步代码块：同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置，当执行 monitorenter 指令时，当前线程将试图获取 objectref(即对象锁) 所对应的 monitor 的持有权，当计数器为 0，那线程可以成功取得 monitor，并将计数器值设置为 1，取锁成功。如果当前线程已经拥持有权，那它可以重入这个 monitor，重入时计数器的值也会加 1。倘若其他线程已经拥有所有权，那当前线程将被阻塞，直到正在执行线程执行完毕，即 monitorexit 指令被执行，执行线程将释放 monitor(锁)并设置计数器值为 0 ，其他线程将有机会持有 monitor 。

②同步方法：JVM 可以从方法常量池中的方法表结构中的 ACC_SYNCHRONIZED 访问标志区分一个方法是否同步方法。当方法调用时，调用指令将会 检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先持有 monitor。

(3) Java 对象头

Synchronized 用的锁对象是存储在 Java 对象头里的，jvm 中采用 2 个字来存储对象头(如果对象是数组则会分配 3 个字，多出来的 1 个字记录的是数组长度)，其主要结构是由 Mark Word 和 Class Metadata Address 组成。

Java 对象头里的 mark word 里默认储存对象的 hashcode、分代年龄和锁标记位。mark word 里存储的数据会随着锁标志位的变化而变化。锁标志位分为：轻量级锁 00、重量级锁 10、GC 标记 11、偏向锁 01。

(4) Java 虚拟机对 synchronized 的优化

在 JavaSE1.6 中，**引入了轻量级锁和偏向锁**。锁一共有 4 中状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁、重量级锁。锁可以升级但不能降级。

①偏向锁：偏向锁就是在**无竞争的情况下把整个同步都消除掉**，连 CAS 都不做了。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时 Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，但是对于锁竞争比较激烈的场合，偏向锁就失效了，升级为轻量级锁。

②轻量级锁：轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

③自旋锁：轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是 50 个循环或 100 循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

(5) 其他

①Synchronized 是重入锁；②线程的中断操作对于正在等待获取的锁对象的 synchronized 方法或者代码块并

不起作用；如果一个线程在等待锁，要么它获得这把锁继续执行，要么它就保存等待，即使调用中断线程的方法，也不会生效；③ 等待唤醒 notify/notifyAll 和 wait 方法，必须处于 synchronized 代码块或者 synchronized 方法中；

6 队列同步器-AbstractQueuedSynchronizer

AQS 是基础组件，**只负责核心并发操作，如加入或维护同步队列，控制同步状态**等，而具体的**加锁和解锁操作交由子类完成。**

队列同步器 AQS 是用来构建锁或者其他同步组件的基础框架。它使用了一个 volatile int state 成员变量来控制同步状态。当 state=0 时，则说明没有任何线程占有共享资源的锁；当 state=1 时，则说明有线程目前正在使用共享变量，AQS 则会将当前线程以及等待状态等信息封装为 Node 结点调用 CAS 方法 compareAndSetTail()加入同步队列的尾部等待；在释放同步状态时，将会唤醒后继节点，后继节点将会在获取同步状态成功时将自己设置为首节点。

AQS 内部通过内部类 Node 构成 FIFO 的同步队列【双向链表】来完成线程获取锁的排队工作，同时利用内部类 ConditionObject 构建等待队列，当 Condition 调用 await()方法后，线程将会加入等待队列中，而当 Condition 调用 signal()方法后，线程将从等待队列转移动同步队列中进行锁竞争。注意这里涉及到两种队列，一种的同步队列，当线程请求锁而等待的后将加入同步队列等待，而另一种则是等待队列(可有多个)，通过 Condition 调用 await()方法释放锁后，将加入等待队列。

6.1 独占式同步状态获取

```
public final void acquire(int arg) {
   if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

①通过调用 aqs 的 acquire(int arg)方法获取同步状态，完成同步状态获取、节点构造、加入同步队列以及在同步队列中自旋等待的工作。

主要是：①先调用实现者的 tryAcquire(int arg)方法，该方法保证线程安全的获取同步状态。②如果获取失败，则构造同步节点，并通过 addWaiter(Node node)方法加入到同步队列的尾部；③调用 acquireQueued(node, int)方法，以死循环的方式获取同步状态，直到前驱节点的出列或阻塞线程被中断实现。

```
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);   //将请求失败的线程封装成结点
    Node pred = tail;
    if (pred != null) {     //如果非第一个结点则直接执行 CAS 入队操作，尝试在尾部快速添加
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {     //使用 CAS 尝试在尾部快速添加
            pred.next = node;
            return node;
        }
    }
    enq(node);                          //如果第一次加入或者 CAS 操作没有成功执行 enq 入队操作
    return node;
}
```

```
private Node enq(final Node node) {
    for (;;) {                                  //死循环
         Node t = tail;
         if (t == null) {                       //如果队列为 null，即没有头结点
```

```
            if (compareAndSetHead(new Node()))  //创建并使用 CAS 设置头结点
                tail = head;
        } else {                                    //队尾添加新结点
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

②addWaiter()：先通过快速尝试设置尾节点，如果失败，则调用 enq(Node node)方法设置尾节点。两个方法都是通过一个 CAS 方法 compareAndSetTail(Node expect, Node update)来设置尾节点，该方法可以确保节点是线程安全添加的。在 enq(Node node)方法中，AQS 通过“死循环”的方式来保证节点可以正确添加，只有成功添加后，当前线程才会从该方法返回，否则会一直执行下去。

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {                                  //自旋，死循环
           final Node p = node.predecessor();   //获取前驱结点
           if (p == head && tryAcquire(arg)) {  //当且仅当 p 为头结点才尝试获取同步状态
               setHead(node);                       //将 node 设置为头结点
               p.next = null;                       //清空原来头结点的引用便于 GC
                failed = false;
                return interrupted;
            }
                                                   //如果前驱结点不是 head，判断是否挂起线程
            if (shouldParkAfterFailedAcquire(p, node) && parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
           cancelAcquire(node);                 //最终都没能获取同步状态，结束该线程的请求
    }
}
```

③acquireQueued()：节点进入同步队列之后，就进入了一个自旋过程，当条件满足，获得同步状态。当前线程在“死循环”中尝试获取同步状态，而只有前驱节点是头结点才能够尝试获取同步状态。

6.2 共享式同步状态获取

①通过调用同步器的 acquireShared(int arg)方法可以共享式地获取同步状态。首先调用实现类的 tryAcquireShared(int arg)方法尝试获取同步状态。当返回值大于等于 0 时，表示能够获取到同步状态。因此，在共享式获取的自旋过程中，成功获取到同步状态并退出自旋的条件就是 tryAcquireShared(int arg)方法返回值大于等于 0。即当前节点的前驱为头结点时，尝试获取同步状态，如果返回值大于等于 0，表示该次获取同步状态成功并从自旋过程中退出。获取失败，自旋获取同步状态。

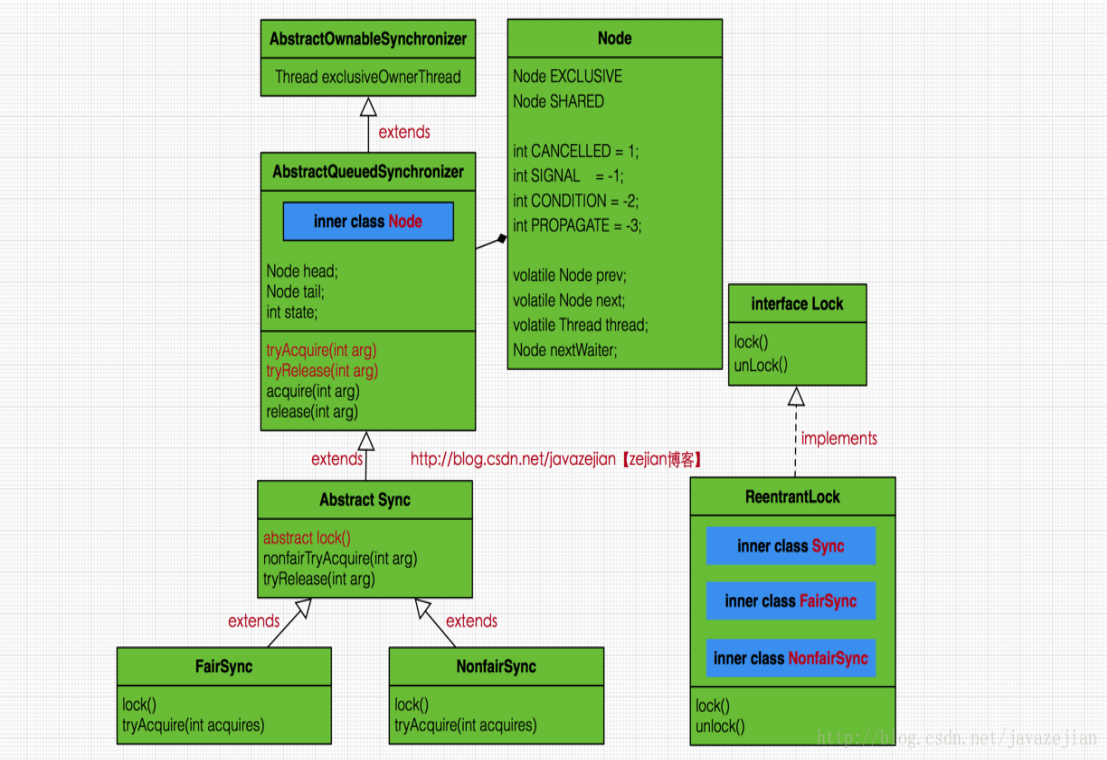
7 Lock 接口

在 java 中有两种方法实现锁机制，一种是 synchronized，而另一种是 Lock。Lock 确保当一个线程位于代码的临界区时，另一个线程不进入临界区， Lock 接口及其实现类提供了更加强大、灵活的锁机制。

public void test(){ synchronized(this){ //do something } }	Lock lock = new ReentrantLock(); lock.lock(); try{ //do something } **finally {** **lock.unlock();** **//注意：unlock()操作必须在 finally 中**，这样可确保即使 } //执行抛出异常，线程最终也能正常释放锁。

在 Java 1.5 中，官方在 concurrent 并发包中加入了 Lock 接口，该接口中提供了 lock()方法和 unLock()方法对显式加锁和显式释放锁操作进行支持。Lock 对象锁还提供了 synchronized 所不具备的其他同步特性，如尝试非阻塞地获取锁，能被中断锁的获取(synchronized 在等待获取锁时是不可中的)，超时锁的获取，等待唤醒机制的多条件变量 Condition 等。

8 独占锁- ReetrantLock



ReentrantLock，可重入锁，是一种独占锁。ReentrantLock 提供了公平锁也非公平锁的选择，构造方法接受一个可选的公平参数（默认非公平锁），**当设置为 true 时，表示公平锁，否则为非公平锁**。公平锁与非公平锁的区别在于公平锁的锁获取是有顺序的。

Sync 为 ReentrantLock 里面的一个内部类，它继承 AQS（AbstractQueuedSynchronizer），它有两个子类：公平锁 FairSync 和非公平锁 NonfairSync。

(1) 非公平锁

①获取锁

```
final void lock() {   //加锁
    if (compareAndSetState(0, 1))    //执行CAS操作，获取同步状态
   //成功则将独占锁线程设置为当前线程
      setExclusiveOwnerThread(Thread.currentThread());
    else
        acquire(1);    //否则再次请求同步状态
}
```

- 首先对同步状态执行 CAS 操作，尝试把 state 的状态从 0 设置为 1。如果返回 true 则代表获取同步状态成功，如果返回 false，获取锁失败，执行 acquire(1)方法，该方法是 AQS 中的方法；
- acquire(int arg)：首先会执行 tryAcquire(arg)方法，该方法在 AQS 中并没有具体实现，而是交由子类实现，因此该方法是由 ReetrantLock 类内部实现的；
- 非公平锁中实现 tryAcquire：①尝试再次获取同步状态，如果获取成功则将当前线程设置为 OwnerThread，返回 true；②判断当前线程 current 是否为 OwnerThread，如果是则属于重入锁，state 自增 1，并获取锁成功，返回 true；③反之，失败，返回 false；
- 回到 AQS 中 acquire 方法：如果 tryAcquire(arg)返回 true，获取锁成功；如果 tryAcquire(arg)返回 false，则会执行 addWaiter(Node.EXCLUSIVE)【ReentrantLock 属于独占锁，因此结点类型为 Node.EXCLUSIVE】加入同步队列队尾。如果第一次加入或者 CAS 入队失败，则执行 enq 入队操作：如果还没有初始同步队列则创建新结点并使用 compareAndSetHead 设置头结点，tail 也指向 head；如果队列已存在，则将新结点 node 添加到队尾。
- acquireQueued(addWaiter(Node.EXCLUSIVE), arg))：节点加入同步队列之后，就进入了一个自旋过程，当条件满足，获得同步状态。当前线程在“死循环”中尝试获取同步状态，而只有前驱节点是头结点才能够尝试获取同步状态。

②释放锁

```
public void unlock() {
    sync.release(1);
}
```

- unlock 内部使用 Sync 的 release(int arg)释放锁；该方法是 AQS 中的方法；
- release()：首先调用 tryRelease 尝试释放锁，此该方法是由 ReetrantLock 类内部实现的。如果释放锁成功，唤醒后继节点的线程；
- 非公平锁中实现 tryRelease：首先用一个临时变量 c 保存释放锁之后的状态值；然后判断状态是否为 0，如果是，则说明已释放同步状态，设置 Owner 为 null；这是更新同步状态，如果状态值为 0，返回 true；

(2) 公平锁

公平锁与非公平锁的区别在于获取锁的时候是否按照 FIFO 的顺序来。释放锁不存在公平性和非公平性。

①获取锁

比较非公平锁和公平锁获取同步状态的过程，会发现两者唯一的区别就在于公平锁在获取同步状态时多了一个限制条件：hasQueuedPredecessors()：即判断当前线程是否位于 CLH 同步队列中的第一个。如果是则返回 true，否则返回 false。如果该方法返回 true，则表示有线程比当前线程更早地请求获取锁，因此需要等待前驱线程获取并释放锁之后才能继续获取锁。

9 Condition

Condition 是一个接口类，通过 Condition 能够精细的控制多线程的休眠与唤醒。对于一个锁，我们可以为多个线程间建立不同的 Condition。Condition 的实现类是 AQS 的内部类 ConditionObject。

Condition 的实现原理：使用 Condition 前必须获得锁，同时在 Condition 的等待队列上的结点与前面同步队列的结点是同一个类即 Node，其结点的 waitStatus 的值为 CONDITION。

每个 Condition 都对应着一个等待队列，也就是说如果一个锁上创建了多个 Condition 对象，那么也就

存在多个等待队列。等待队列是一个 FIFO 的队列【单链表】，在队列中每一个节点都包含了一个线程的引用，而该线程就是 Condition 对象上等待的线程。当一个线程调用了 await()相关的方法，那么该线程将会释放锁，并构建一个 Node 节点封装当前线程的相关信息加入到等待队列中进行等待，直到被唤醒、中断、超时才从队列中移出。

等待：await()方法主要做了 3 件事，①调用 addConditionWaiter()方法将当前线程封装成 node 结点加入等待队列，②调用 fullyRelease(node)方法释放同步状态并唤醒后继结点的线程。③调用 isOnSyncQueue(node)方法判断结点是否在同步队列中，即是否被唤醒。注意是个 while 循环，如果同步队列中没有该结点就直接挂起该线程，需要明白的是如果线程被唤醒后就调用 acquireQueued(node, savedState)执行自旋操作争取锁，即当前线程结点从等待队列转移到同步队列并开始努力获取锁。

通知：调用 Condition 的 signal ()方法，将会唤醒在等待队列中等待时间最长的节点（首节点），在唤醒节点之前，会将节点移动到同步队列中。signal()方法做了两件事：①判断当前线程是否持有独占锁，没有就抛出异常【只有独占模式先采用等待队列，而共享模式下是没有等待队列的，也就没法使用 Condition】；②执行 doSignal(first)，唤醒等待队列的第一个结点，首节点移动到同步队列并唤醒节点中的线程。

doSignal(first)方法中做了两件事：①从条件等待队列移除被唤醒的节点，然后重新维护条件等待队列的 firstWaiter 和 lastWaiter 的指向。②将从等待队列移除的结点加入同步队列(在 transferForSignal()方法中完成的)，如果进入到同步队列失败并且条件等待队列还有不为空的节点，则继续循环唤醒后续其他结点的线程。

10 共享锁-Semaphore

共享锁模式允许同一个时刻多个线程可获取同步状态。

信号量维护了一个许可集，我们在初始化 Semaphore 时需要为这个许可集传入一个数量值，该数量值代表同一时间能访问共享资源的线程数量。线程可以通过 acquire()方法获取到一个许可，然后对共享资源进行操作，注意如果许可集已分配完了，那么线程将进入等待状态，直到其他线程释放许可才有机会再获取许可，线程释放一个许可通过 release()方法完成。

实现原理：Semaphore 内部同样存在继承自 AQS 的内部类 Sync 以及继承自 Sync 的公平锁(FairSync)和非公平锁(NofairSync)，AQS 是基础组件，只负责核心并发操作，如加入或维护同步队列，控制同步状态等，而具体的加锁和解锁操作交由子类完成。Semaphore 的内部类公平锁(FairSync)和非公平锁(NoFairSync)各自实现不同的获取锁方法即 tryAcquireShared(int arg)，而**释放锁 tryReleaseShared(int arg)的操作交由 Sync 实现，因为释放操作都是相同的。**

8.1 非公平锁中的共享锁

```
//默认创建公平锁，permits 指定同一时间访问共享资源的线程数
public Semaphore(int permits) {
        sync = new NonfairSync(permits);
}

public Semaphore(int permits, boolean fair) {
     sync = fair ? new FairSync(permits) : new NonfairSync(permits);
}
```

创建 Semaphore 对象传入许可数值时，最终会赋值给 state，state 的数值代表同一个时刻可同时操作共享数据的线程数量。每当一个线程请求到来时，如果 state 值代表的许可数足够使用，那么请求线程将会获取同步状态成功，并更新 state 的值(一般是对 state 值减 1)，直到 state 为 0 时，表示已没有可用的许可数，线程将被加入到同步队列等待并开启自旋操作，直到其他线程释放同步状态(一般是对 state 值加 1)才可能获取对共享资源的访问权。

```
public void acquire() throws InterruptedException {   //Semaphore 的 acquire()
```

```
      sync.acquireSharedInterruptibly(1);
}

//AQS 的 acquireSharedInterruptibly()方法
public final void acquireSharedInterruptibly(int arg)  throws InterruptedException {
    if (Thread.interrupted())   //判断是否中断请求
        throw new InterruptedException();
    if (tryAcquireShared(arg) < 0)                  //如果小于 0，则线程获取同步状态失败
        doAcquireSharedInterruptibly(arg);         //未获取成功加入同步队列等待
}
```

Semaphore 的 acquire()方法也是可中断的。**首先进行线程中断的判断**，如果没有中断，那么先尝试调用实现者 Sync 的 **tryAcquireShared(arg)** 方法获取同步状态，如果获取成功，那么方法执行结束，如果获取失败调用 doAcquireSharedInterruptibly(arg);方法加入同步队列等待。

```
protected int tryAcquireShared(int acquires) {     //Semaphore 中非公平锁 NonfairSync
   return nonfairTryAcquireShared(acquires);       //调用了父类 Sync 中的实现方法
}

abstract static class Sync extends AbstractQueuedSynchronizer {   //Syn 类中
    final int nonfairTryAcquireShared(int acquires) {
         for (;;) {   //使用死循环
             int available = getState();
             int remaining = available - acquires;
             //判断信号量是否已小于 0 或者 CAS 执行是否成功
             if (remaining < 0 || compareAndSetState(available, remaining))
                 return remaining;
         }
     }
}
```

tryAcquireShared(arg)是个模板方法，AQS 内部没有提供具体实现，由子类实现自己实现：首先获取 state，并执行减法操作，得到 remaining 值：如果 remaining **不小于 0**，那么线程获取同步状态成功，可访问共享资源，并更新 state 的值。如果小于 0，获取失败，则创建一个共享模式（Node.SHARED）的结点并加入同步队列，并进入自旋操作。只有前驱结点为 head 并获取同步状态成功，将当前线程设置为 head，并通知后续结点继续获取同步状态。

8.2 公平锁中的共享锁

与非公平锁 tryAcquireShared(int acquires)方法实现的唯一不同是：在尝试获取同步状态前，先调用了 hasQueuedPredecessors()方法判断同步队列中是否存在结点，如果存在则返回-1，即将线程加入同步队列等待。从而保证先到来的线程请求一定会先执行，也就是所谓的公平锁。至于其他操作，与前面分析的非公平锁一样。

8.3 释放锁

Semaphore 调用了 AQS 中的 releaseShared(int arg)方法。通过子类 Semaphore 实现的 tryReleaseShared(arg)方法【①获取当前 state；②释放状态 state 增加 releases；③通过 CAS 更新 state 的值，成功返回 true 并退出循环】尝试释放同步状态，如果释放成功，那么将调用 doReleaseShared()唤醒同步队列中后继结点。

9 ReentrantReadWriteLock

读写锁 ReentrantReadWriteLock【jdk1.5】在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，写操作对读操作保证可见性。把 State 状态作为一个读写锁的计数器，包括了重入的次数。

11 并发工具类

11.1 CountDownLatch

CountDownLatch 允许一个或多个线程一直等待其他线程完成操作。CountDownLatch 通过 AQS 里面的共享锁来实现的。在构造方法里，会创建一个 Sync[sɪŋk]对象。Sync 是 CountDownLatch 的内部私有类，继承 AQS。在创建 CountDownLatch 实例时，需要传递一个 int 型的参数 count，该参数为计数器的初始值，表示该共享锁可以获取的总次数，会赋值给了 AQS 中 state(private volatile long)。

当某个线程调用 await()方法时：程序首先判断 state 的值是否为 0，如果大于 0 的话则会一直等待直到为 0 为止。当其他线程调用 countDown()方法时，则执行释放共享锁状态，使 count 值 - 1。当在创建 CountDownLatch 时初始化的 count 参数，必须要有 count 线程调用 countDown 方法才会使计数器 count 等于 0，锁才会释放，前面等待的线程才会继续运行。注意 CountDownLatch 不能回滚重置。

```
//只有等所有的人到齐了才会开会
public class CountDownLatchTest2 {
    private static CountDownLatch countDown = null;
    private static final int THREAD_NUMBER = 5;

    public static void main(String[] args) {
        countDown = new CountDownLatch(THREAD_NUMBER);
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(THREAD_NUMBER);   //线程池
        for (int i = 0; i < THREAD_NUMBER; i++) {                                        //执行线程
            fixedThreadPool.execute(new ConsumeRunnable());
        }
        System.out.println("准备开会，参加会议人员总数为：" + countDown.getCount());
        try {
            countDown.await();
        } catch (InterruptedException e) {

        System.out.println("所有人员已经到达，会议开始.....");
    }

    private static class ConsumeRunnable implements Runnable {
        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() + "到达.....");
            countDown.countDown();
        }
    }
}
```

11.2 CyclicBarrier

CyclicBarrier 让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。

CyclicBarrier 的两个构造函数：

①CyclicBarrier(int parties)：默认构造方法会传 int 型参数，表示屏障拦截的线程数量，每个线程调用 await 方法表示已经到达了屏障，当前线程被阻塞。

②CyclicBarrier(int parties, Runnable barrierAction) ：用于在线程到达屏障时，优先执行 barrierAction。

```
//只有等所有的人到齐了才会开会
public class CyclicBarrierTest {
    private static CyclicBarrier cyclicBarrier;
    private static final int THREAD_NUMBER = 5;
    static class CyclicBarrierThread extends Thread{
        public void run() {
            System.out.println(Thread.currentThread().getName() + "到了");
            try {
                cyclicBarrier.await();    //等待
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args){
        cyclicBarrier = new CyclicBarrier(THREAD_NUMBER, new Runnable() {
            @Override
            public void run() {
                System.out.println("人到齐了，开会吧....");
            }
        });
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(THREAD_NUMBER);
        for (int i = 0; i < THREAD_NUMBER; i++) {          //执行线程
            fixedThreadPool.execute(new CyclicBarrierThread());
        }
    }
}
```

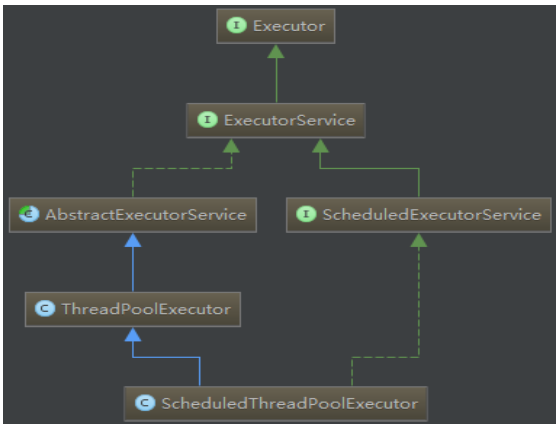
CyclicBarrier 和 CountDownLatch 的区别：CountDownLatch 的计数器只能使用一次，而 CyclicBarrier 的计数器可以使用 reset 方法重置。

CountDownLatch	CyclicBarrier
减计数方式	加计数方式
计算为 0 时释放所有等待的线程	计数达到指定值时释放所有等待线程

计数为 0 时，无法重置	计数达到指定值时，计数置为 0 重新开始
调用 countDown()方法计数减一，调用 await()方法只进行阻塞，对计数没任何影响	调用 await()方法计数加 1，若加 1 后的值不等于构造方法的值，则线程阻塞
不可重复利用	可重复利用

12 线程池

(1) Executor 的 UML 图



① Executor：一个接口，其定义了一个接收 Runnable 对象的方法 executor；
② ExecutorService：是一个比 Executor 使用更广泛的子类接口，其提供了生命周期管理的方法，以及可跟踪一个或多个异步任务执行状况返回 Future 的方法；
③ AbstractExecutorService：ExecutorService 执行方法的默认实现；
④ ScheduledExecutorService：一个可定时调度任务的接口；
⑤ ScheduledThreadPoolExecutor：一个可定时调度任务的线程池；
⑥ ThreadPoolExecutor：线程池，可以通过调用 Executors 以下静态工厂方法来创建线程池并返回一个 ExecutorService 对象：

(2) 线程池的实现原理

当提交一个新任务到线程池时，线程池的处理流程：
①首先判断**核心线程池**里的线程是否都在执行任务。如果不是(运行的线程少于 corePoolSize)，创建一个新的工作线程执行任务；否则，进入下一步；
②判断**工作队列**是否已满。没满，存储到这个工作队列中，否则，进入下一步；
③判断**线程池的线程**是否都在执行任务，如果不是(运行的线程少于 maximumPoolSize)，创建新的线程来处理任务;否则，采用拒绝策略来处理这个任务。

(3) 创建线程池

通过 ThreadPoolExecutor 来创建一个线程池：

```
public ThreadPoolExecutor( int corePoolSize,                //线程池的基本大小
                           int maximumPoolSize,       //线程池的最大数量
                           long keepAliveTime,          //多余线程活动保持的时间
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue,
                           ThreadFactory threadFactory,       //使用 ThreadFactory 创建新线程
                           RejectedExecutionHandler handler) //后两个参数为可选参数
```

- ✧ workQueue：任务队列，用于保存等待执行的任务的阻塞队列，可以选择：
  - ■ ArrayBlockingQueue：基于数组的有界阻塞队列，FIFO。
  - ■ LinkedBlockingQueue：基于链表的阻塞队列，Executors.newFixedThreadPool()
  - ■ SynchronousQueue：不存储元素的阻塞队列。Executors.newCachedThreadPool()
  - ■ PriorityBlockingQueue：具有优先级的无线阻塞队列。
- ✧ Handler：拒绝策略。
  - ■ AbortPolicy：直接抛出异常。
  - ■ CallerRunsPolicy：只用调用者所在线程来运行任务。
  - ■ DiscardOldestPolicy：丢弃队列里最近的一个任务，并指向当前任务。
  - ■ DiscardPolicy：不处理，丢弃掉。

(4)Executors

提供了一系列静态工厂方法用于创建各种线程池。

① newFixedThreadPool：创建可重用且**固定线程数**的线程池。在所有 corePoolSize 线程都忙时新任务将在队列中等待。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
     return new ThreadPoolExecutor( nThreads,  nThreads,
                                    0L,  TimeUnit.MILLISECONDS,
                                    new LinkedBlockingQueue<Runnable>());
}
```

② newSingleThreadExecutor：创建一个单线程的 Executor。在所有 corePoolSize 线程都忙时新任务将在队列中等待。

```
public static ExecutorService newSingleThreadExecutor() {
   return new FinalizableDelegatedExecutorService
     (new ThreadPoolExecutor(1, 1,
                             0L, TimeUnit.MILLISECONDS,
                             new LinkedBlockingQueue<Runnable>()));
}
```

③ newScheduledThreadPool：创建一个可延迟执行或定期执行的线程池。模拟心跳机制

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
          super(corePoolSize, Integer.MAX_VALUE,
               0, TimeUnit.NANOSECONDS,
               new DelayedWorkQueue());
}
```

④ newCachedThreadPool：创建可缓存的线程池，如果线程在 60 秒未被使用就将被移除。使用同步队列，将任务直接提交给线程。

```
public static ExecutorService newCachedThreadPool() {
       return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                     60L, TimeUnit.SECONDS,
                                     new SynchronousQueue<Runnable>());
}
```

13 ThreadLocal

ThreadLocal 类用来提供线程内部的局部变量。ThreadLocal 是**为每个线程创建一个单独的变量副本**，每个线程都可以修改自己所拥有的变量副本，而不会影响其他线程的副本. 其实这也是解决线程安全的问题的一种方法。

(1) ThreadLocal 是如何做到为每一个线程维护变量的副本的呢？

其实实现的思路很简单：在 ThreadLocal 类中有一个静态内部类 ThreadLocalMap(其类似于 Map)，用键值对的形式存储每一个线程的变量副本， key 为当前 ThreadLocal 对象，而 value 对应线程的变量副本，每个线程可能存在多个 ThreadLocal。ThreadLocal 实例本身是不存储值，它只是提供了一个在当前线程中找到副本值得 key。

(2) ThreadLocalMap

ThreadLocalMap 其内部利用 Entry 来实现 key-value 的存储，key 就是 ThreadLocal，而 value 就是值，同时，Entry 也继承 WeakReference，所以说 Entry 所对应 key 的引用为一个弱引用，这也导致了后续会产生内存泄漏问题的原因。

(3) ThreadLocal 为什么会内存泄漏

每个 thread 中都存在一个 ThreadLocal.ThreadLocalMap 的 map，该 Map 中的 key 为一个 threadlocal 实例，它为一个弱引用，弱引用有利于 GC 回收。**当把 threadlocal 实例置为 null 以后，没有任何强引用指向 threadlocal 实例,所以 threadlocal 将会被 gc 回收。**但是 value 却不一定能够被回收，因为他还与 Current Thread 存在一个强引用关系，由于存在这个强引用关系，会导致 value 无法回收。只有当前 thread 结束以后, current thread 就不会存在栈中,强引用断开, Current Thread, Map, value 将全部被 GC 回收。如果这个线程对象不会销毁那么这个强引用关系则会一直存在，就会出现内存泄漏情况。

(4) 怎么避免这个问题呢？

在 ThreadLocalMap 中的 setEntry()、getEntry()，如果遇到 key == null 的情况，会对 value 设置为 null。当然我们也可以显示调用 **ThreadLocal 的 remove()方**法进行处理。

(5)例子

```
public class TreadLocalTest {
    public static void main(String[] args) {
        TestThreadLocal t = new TestThreadLocal();
        new Thread(t, "Thread A").start();
        new Thread(t, "Thread B").start();
    }
}

class TestThreadLocal implements Runnable {
    ThreadLocal<Student> studentLocal = new ThreadLocal<Student>();
    @Override
    public void run() {
        String currentThreadName = Thread.currentThread().getName();
        System.out.println(currentThreadName + " is running...");
        Random random = new Random();
        int age = random.nextInt(100);
        Student s = getStudent();       //为每个线程都独立的 new 一个 student 对象
        s.setAge(age);
        System.out.println( currentThreadName + ":" + s.getAge());
    }
```

```
    public Student getStudent() {
        Student s = (Student) studentLocal.get();
        if (s == null) {
            s = new Student();
            studentLocal.set(s);
        }
        return s;
    }
}

class Student {
    private int age;
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

13 阻塞队列和非阻塞队列

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。

(1) 阻塞队列提供了四种处理方法:

方法/处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	**add(e)**	**offer(e)**	**put(e)**	**offer(e,time,unit)**
移除方法	**remove()**	**poll()**	**take()**	**poll(time,unit)**
检查方法	**element()**	**peek()**	**不可用**	**不可用**

✧ 抛出异常：当队列满时，再往队列里插入元素，抛出 IllegalStateException 异常
当队列空时，从队列里获取元素会抛出 NoSuchElementException 异常；

✧ 返回特殊值：当队列插入元素时，会返回元素是否插入成功，成功返回 true；
如果是移除方法，则是从队列里取出一个元素，如果没有则返回 null。

✧ 一直阻塞：一直阻塞，直到队列可用或者相应中断退出。

✧ 超时退出：队列会阻塞生产者线程一段时间；如果超过了指定时间，生产者线程退出。

如果是无界阻塞队列，队列不可能会出现满的情况，所以使用 put 和 offer 方法永远不被阻塞，而是 offer 方法时，返回的永远是 true。

(2) Java 里的阻塞队列

JDK7 提供了 7 个阻塞队列。分别是

①ArrayBlockingQueue 是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证公平的访问队列，可以创建一个公平的阻塞队列：ArrayBlockingQueue queue = new ArrayBlockingQueue(1000,**true**);

②LinkedBlockingQueue 是一个用链表实现的有界阻塞队列。此队列的默认和最大长度为 Integer.MAX_VALUE。此队列按照先进先出的原则对元素进行排序。

③PriorityBlockingQueue 是一个支持优先级的无界队列。默认情况下元素采取自然顺序排列，也可以通过比

较器 comparator 来指定元素的排序规则。元素按照升序排列。

④DelayQueue：DelayQueue 是一个支持延时获取元素的无界阻塞队列。队列使用 PriorityQueue 来实现。队列中的元素必须实现 Delayed 接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。

⑤**SynchronousQueue** 是一个不存储元素的阻塞队列。每一个 put 操作必须等待一个 take 操作，否则不能继续添加元素。吞吐量高于 LinkedBlockingQueue 和 ArrayBlockingQueue。

⑥**LinkedTransferQueue** 是一个由链表结构组成的无界阻塞 TransferQueue 队列。

⑦**LinkedBlockingDeque** 是一个由链表结构组成的双向阻塞队列。

(3) 阻塞队列实现原理

①BlockingQueue 通过 Condition 条件对象来实现阻塞。

②LinkedBlockingQueue：为什么添加完成后是继续唤醒在条件对象 notFull 上的添加线程而不是像 ArrayBlockingQueue 那样直接唤醒 notEmpty 条件对象上的消费线程？而又为什么要当 if (c == 0)时才去唤醒消费线程呢？

ArrayBlockingQueue 内部完成添加操作后，会直接唤醒消费线程对元素进行获取，这是因为 ArrayBlockingQueue 只用了一个 ReenterLock 同时对添加线程和消费线程进行控制，如果在添加完成后再次唤醒添加线程的话，消费线程可能永远无法执行。而 LinkedBlockingQueue 其内部对添加线程和消费线程分别使用了各自的 ReenterLock 锁对并发进行控制，也就是说添加线程和消费线程是不会互斥的。这也是为什么 LinkedBlockingQueue 的吞吐量要相对大些的原因。

为什么要判断 if (c == 0)时才去唤醒消费线程呢？这是因为消费线程一旦被唤醒是一直在消费的（前提是有数据），所以 c 值是一直在变化的，c 值是添加完元素前队列的大小，此时 c 只可能是 0 或 c>0，如果是 c=0，那么说明之前消费线程已停止，条件对象上可能存在等待的消费线程，添加完数据后应该是 c+1，那么有数据就直接唤醒等待消费线程，如果没有就结束啦，等待下一次的消费操作。那为什么不是条件 c>0 才去唤醒呢？消费线程一旦被唤醒是一直在消费的，如果添加前 c>0，那么很可能上一次调用的消费线程后，数据并没有被消费完，条件队列上也就不存在等待的消费线程了。

(4)非阻塞队列

ConcurrentLinkedQueue 是非阻塞队列。ConcurrentLinkedQueue 是一个基于链表实现的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。默认情况下 head 节点存储的元素为空

①入队列【CAS 添加到队列尾部直至成功，永远返回 true】

入队主要做两件事情，第一是将入队节点设置成当前队列的最后一个节点。第二是更新 tail 节点，如果 tail 节点的 next 节点不为空，则将入队节点设置成 tail 节点，如果 tail 节点的 next 节点为空，则将入队节点设置成 tail 的 next 节点【不更新 tail】，所以 **tail 节点不总是尾节点。**

HOPS 的设计意图

使用 hops 变量来控制并减少 tail 节点的更新频率，并不是每次节点入队后都将 tail 节点更新为尾节点，而是当 tail 节点和尾节点的距离大于等于常量 hops 的值（默认等于 1）时才更新 tail 节点。

②出队列

首先获取头节点的元素，然后判断头节点元素是否为空，如果不为空，使用 CAS 的方式将头节点的引用设置成 null 如果 CAS 成功，则直接返回头节点的元素，如果不成功，需要重新获取头节点。不会更新 head 节点。如果为空，则弹出 head 的 next 结点并更新 head 结点为原来 head 的 next 结点的 next 结点。

ConcurrentLinkedQueue 的线程安全是通过其插入、删除时采取 CAS 操作来保证的。

14 HashMap【1.7】

1. HashMap 的数据结构

HashMap 实际上是一个“链表散列”的数据结构，即数组和链表的结合体。HashMap 底层就是一个数

组结构，数组中的每一项又是一个链表。**Entry 就是数组中的元素**，每个 Map.Entry 其实就是一个 key-value 对，它持有一个指向下一个元素的引用，这就构成了链表。**HashMap 允许存放 null 键和 null 值**。

2. HashMap 的存取实现

```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;
    final int hash;  }
```

HashMap 中 put 元素的时候，首先根据该 key 的 hashCode() 返回值决定该 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() 返回值相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry 的 value，但 key 不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，那么在这个位置上的元素将以链表的形式存放，**新加入的放在链头**，最先加入的放在链尾。新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部。

3. HashMap 的 resize（rehash）

当 HashMap 中的元素个数超过数组大小*loadFactor 时，就会进行数组扩容，loadFactor 的默认值为 0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为 16，那么当 HashMap 中元素个数超过 16*0.75=12 的时候，就把数组的大小扩展为 2*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作。

4. HashMap 的死循环

多线程同时 put 时，如果同时触发了 rehash 操作，会导致 HashMap 中的链表中出现循环节点，进而使得后面 get 的时候，会死循环。

Put一个Key,Value对到Hash表中：

```
public V put(K key, V value)
{
    ......
    //算Hash值
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    //如果该key已被插入，则替换掉旧的value （链接操作）
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    //该key不存在，需要增加一个结点
    addEntry(hash, key, value, i);
    return null;
}
```

检查容量是否超标

```
void addEntry(int hash, K key, V value, int bucketIndex)
{
    Entry<K,V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    //查看当前的size是否超过了我们设定的阈值threshold，如果超过，需要resize
    if (size++ >= threshold)
        resize(2 * table.length);
}
```

新建一个更大尺寸的hash表，然后把数据从老的Hash表中迁移到新的Hash表中。

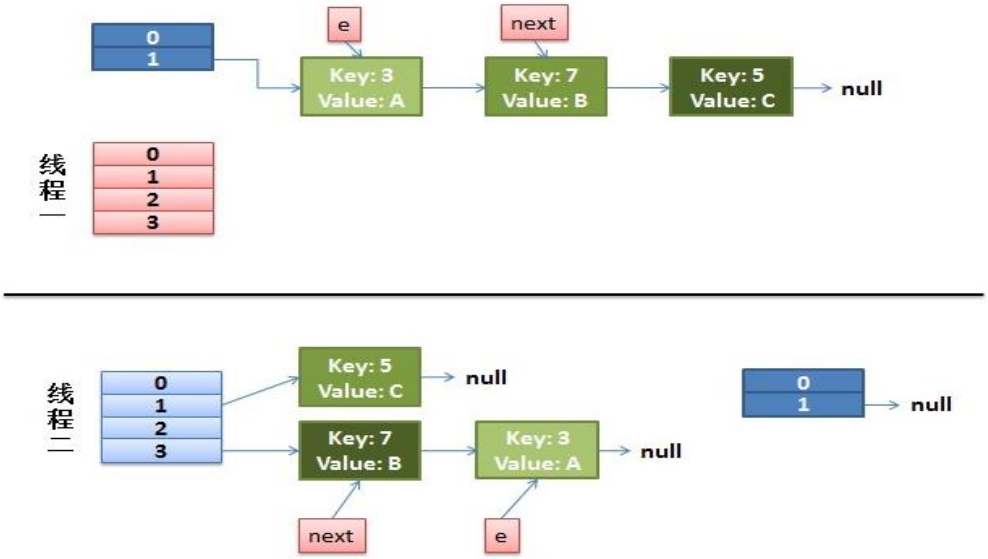
```
void resize(int newCapacity)
{
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    ......
    //创建一个新的Hash Table
    Entry[] newTable = new Entry[newCapacity];
    //将Old Hash Table上的数据迁移到New Hash Table上
    transfer(newTable);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);
}
```

```
void transfer(Entry[] newTable)
{
    Entry[] src = table;
    int newCapacity = newTable.length;
    //下面这段代码的意思是：
    //  从OldTable里摘一个元素出来，然后放到NewTable中
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
            src[j] = null;
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
                e = next;
            } while (e != null);
        }
    }
}
```

(1)线程 1、2 都进入 transfer 第三行,此时 map 为：

(2)线程 1 执行到【int i = indexFor(e.hash, newCapacity);】，线程 2 执行到【src[j] = null;】，此时 map 为：

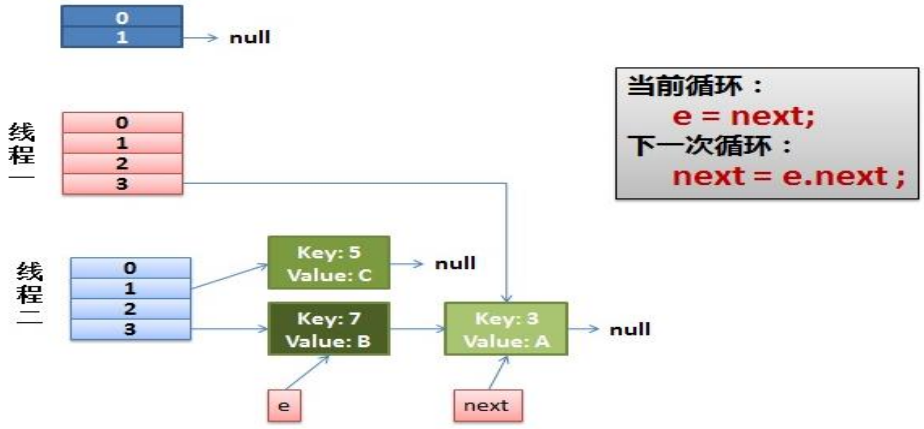
(3) 直接把 Thread2 执行完毕，此时 map 的内容为：



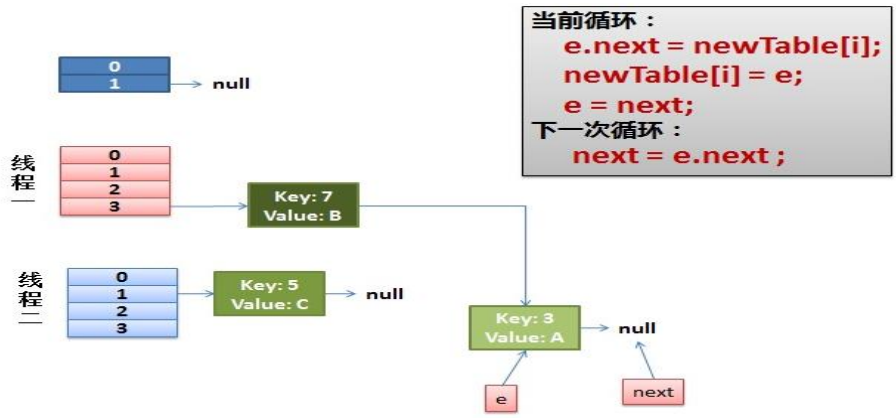
注意：因为 Thread1 的 **e 指向了 key(3)，而 next 指向了 key(7)**，其在线程二 rehash 后，指向了线程二重组后的链表。我们可以看到**链表的顺序被反转后**。

(4)切换回 Thread1，先是执行 newTalbe[i] = e;然后是 e = next，导致了 **e 指向了 key(7)**，

而下一次循环的 next = e.next 导致了 **next 指向了 key(3)**，此时 map 的内容为：



(5) Thread1 第 2 次循环后，把 key(7)摘下来，放到 newTable[i]的第一个，然后把 e 和 next 往下移。

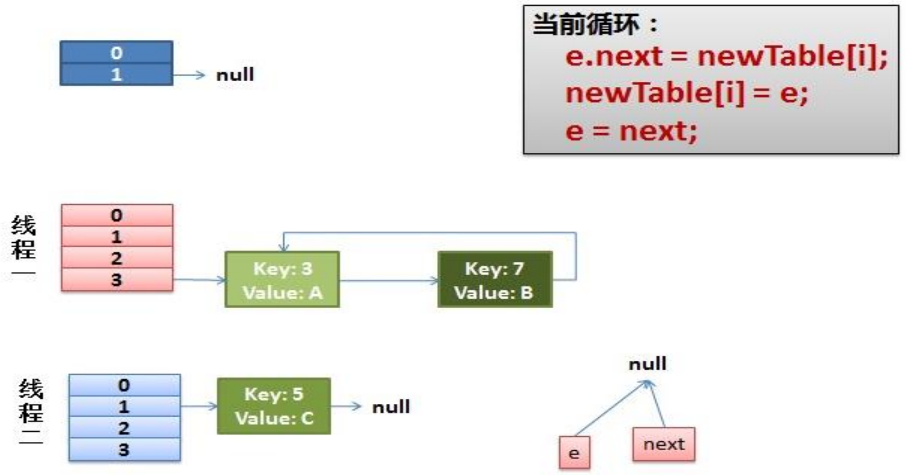


(5)环形链接出现。

e.next = newTable[i] 导致 key(3).next 指向了 key(7)

注意：此时的 key(7).next 已经指向了 key(3)， 环形链表就这样出现了。

于是，当我们的线程一调用到，HashTable.get(11)时，悲剧就出现了——Infinite Loop。



15 HashMap【1.8】

1.8 中 hashmap **声明两对指针，维护两个连链表**，依次在末端添加新的元素。因此不会因为多线程 put 导致死循环，但是依然有其他的弊端，比如数据丢失等等。因此多线程情况下还是建议使用 concurrenthashmap。

HashMap 采用的是**数组+链表+红黑树**的形式。数组是可以扩容的，链表也是转化为红黑树的。用户可以设置的参数：初始总容量默认 16，默认的加载因子 0.75。初始的数组个数默认是 16 容量 X 加载因子=阈值。一旦目前容量超过该阈值，则执行扩容操作。

HashMap 类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个 Node 的数组。Node 是 HashMap 的一个内部类，实现了 Map.Entry 接口，本质是就是一个映射(键值对)。

什么时候扩容？

1 当前容量超过阈值

2 **当链表中元素个数超过默认设定（8 个），当数组的大小还未超过 64 的时候，此时进行数组的扩容，如果超过则将链表转化成红黑树**

当**数组大小已经超过 64 并且链表中的元素个数超过默认设定（8 个）时，将链表转化为红黑树**。

1. Put 过程

- 根据 key 计算出 hash 值
- hash 值&（数组长度-1）得到所在数组的 index
  - 如果该 index 位置的 Node 元素不存在，则直接创建一个新的 Node
  - 如果该 index 位置的 Node 元素是 TreeNode 类型即红黑树类型了，则直接按照红黑树的插入方式进行插入
  - 如果该 index 位置的 Node 元素是非 TreeNode 类型则，则按照链表的形式进行插入操作
    链表插入操作完成后，判断是否超过阈值 TREEIFY_THRESHOLD（默认是 8），超过则要么数组扩容要么链表转化成红黑树
- 判断当前总容量是否超出阈值，如果超出则执行扩容

2. 扩容过程

按照 **2 倍**扩容的方式。这里为啥是 2 倍？因为 2 倍的话，更加容易计算他们所在的桶，并且各自不会相互干扰。桶 0 中的元素会被分到桶 0 和桶 4 中。

扩充 HashMap 的时候，不需要像 JDK1.7 的实现那样重新计算 hash，只需要看看原来的 hash 值新增的那个 bit 是 1 还是 0 就好了，是 0 的话索引没变，是 1 的话索引变成“原索引+oldCap”，JDK1.7 中 rehash 的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8 不会倒置。

3. get 过程

根据 key 计算 hash 值：计算出 hashCode 值，hash=hashCode^(hashCode>>>16);
hash 值&（数组长度-1）得到所在数组的 index
如果要找的 key 就是上述数组 index 位置的元素，直接返回该元素的值
如果该数组 index 位置元素是 TreeNode 类型，则按照红黑树的查询方式来进行查找 O(logn)
如果该数组 index 位置元素非 TreeNode 类型，则按照链表的方式来进行遍历查询 O(n)

4. 性能

随着 size 的变大，JDK1.7 的花费时间是增长的趋势，而 JDK1.8 是明显的降低趋势，并且呈现对数增长稳定。当一个**链表太长的时候，HashMap 会动态的将它替换成一个红黑树**，这样的话会将时间复杂度从 O(n)降为 O(logn)。

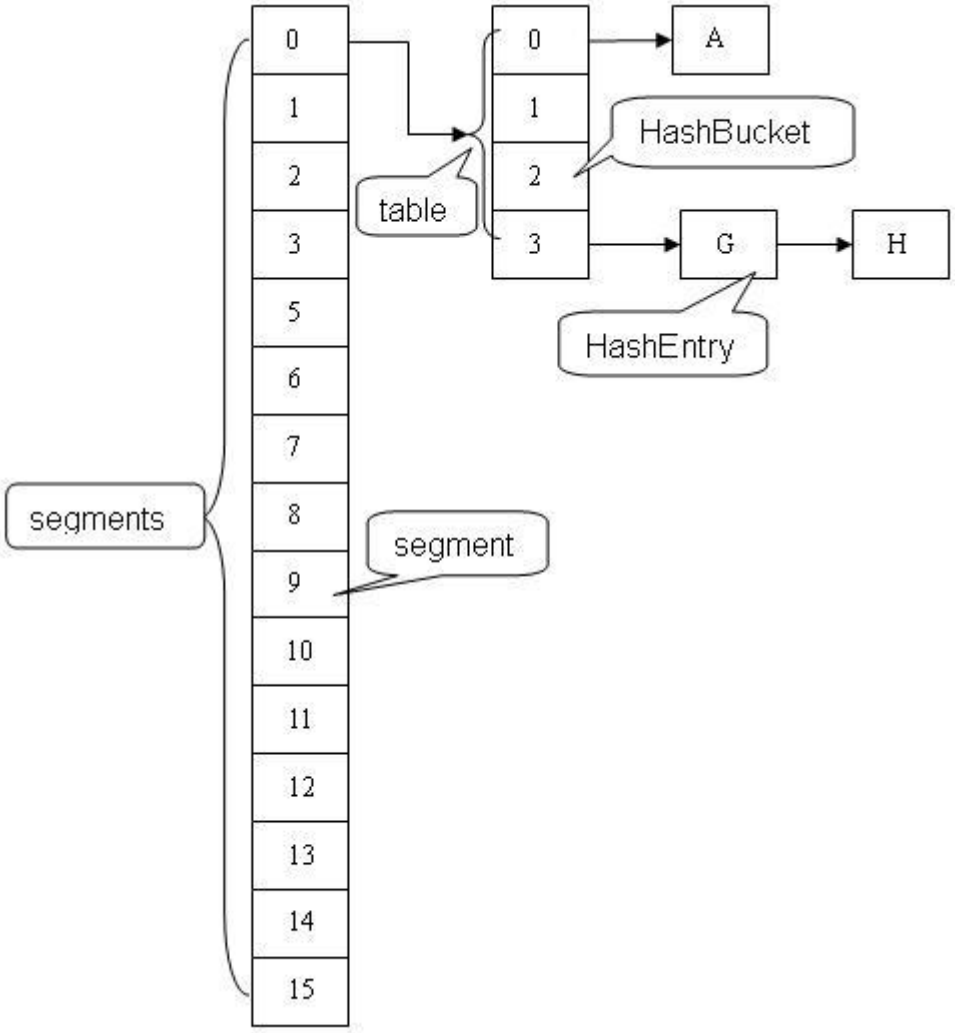
15 HashTable

HashTable 是一个线程安全的类，它使用 synchronized 来锁住整张 Hash 表来实现线程安全，即每次锁住整张表让线程独占。如线程 1 使用 put 进行元素添加，线程 2 不但不能使用 put 添加元素，也不能使用 get 获取元素。hashTable 不允许 key 和 value 为 null。

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。它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。

17 ConcurrentHashMap【1.7】



1. 结构

ConcurrentHashMap 采用了非常精妙的"分段锁"策略，ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成，主干是个 Segment 数组。**Segment 继承了 ReentrantLock**，所以它就是一种可重入锁（ReentrantLock)。HashEntry 则用于存储键值对数据。

在 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和 HashMap 类似，是一种数组和链表结构。一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，Segment 里维护了一个 HashEntry 数组。当对 HashEntry 数组的数据进行修改时，必须首先获得与它对应的 Segment 锁。（就按默认的 ConcurrentLeve 为 16 来讲，理论上就允许 16 个线程并发执行)。

2. 实现原理

ConcurrentHashMap 使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。

ConcurrentHashMap 内部分为很多个 Segment，Segment 继承了 ReentrantLock，表明每个 segment 都可以当做一个锁。对于一个 key，需要经过三次（）hash 操作，才能最终定位这个元素的位置，这三次 hash 分别为：

- 对于一个 key，先进行一次 hash 操作，得到 hash 值 h1，也即 h1 = hash1(key)；
- 将得到的 h1 的高几位进行第二次 hash，得到 hash 值 h2，也即 h2 = hash2(h1 高几位)，通过 **h2 能**

够确定该元素的放在哪个 **Segment**；

- 将得到的 h1 进行第三次 hash，得到 hash 值 h3，也即 h3 = hash3(h1)，通过 h3 能够**确定该元素放置在哪个 HashEntry**。

ConcurrentHashMap 实现技术是保证 HashEntry 几乎是不可变的。HashEntry 代表每个 hash 链中的一个节点，其结构如下所示：

```
static final class HashEntry<K,V> {
    final K key;
    final int hash;
    volatile V value;
    volatile HashEntry<K,V> next;
}
```

2.1 初始化 Segment

```
Segment(float lf, int threshold, HashEntry<K,V>[] tab) {
    this.loadFactor = lf;//负载因子
    this.threshold = threshold;//阈值
    this.table = tab;//主干数组即 HashEntry 数组
}
```

2.2 初始化 ConcurrentHashMap

```
public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {
    … …
    //MAX_SEGMENTS 为 1<<16=65536，也就是最大并发数为 65536
    if (concurrencyLevel > MAX_SEGMENTS)
       concurrencyLevel = MAX_SEGMENTS;
    //2 的 sshif 次方等于 ssize，例:ssize=16,sshift=4;ssize=32,sshif=5
    int sshift = 0;
    int ssize = 1;
    while (ssize < concurrencyLevel) {
        ++sshift;
        ssize <<= 1;
    }
    this.segmentShift = 32 - sshift;
    this.segmentMask = ssize - 1;
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
```

```
    //计算 cap 的大小，即 Segment 中 HashEntry 的数组长度，cap 也一定为 2 的 n 次方.
    int c = initialCapacity / ssize;
    if (c * ssize < initialCapacity)
        ++c;
    int cap = MIN_SEGMENT_TABLE_CAPACITY;
    while (cap < c)
      cap <<= 1;
    … …
}
```

传入的参数有 initialCapacity，loadFactor，concurrencyLevel 这三个。

- initialCapacity 表示新创建的这个 ConcurrentHashMap 的初始容量，也就是上面的结构图中的 **Entry 数量**。默认值为 static final int DEFAULT_INITIAL_CAPACITY = 16;
- loadFactor 表示**负载因子**，就是当 ConcurrentHashMap 中的元素个数大于 loadFactor * 最大容量时就需要 rehash，扩容。默认值为 static final float DEFAULT_LOAD_FACTOR = 0.75f;
- concurrencyLevel 表示**并发级别**，这个值用**来确定 Segment 的个数**。
- **Segment 数组的大小 ssize** 是由 concurrentLevel 来决定的，但是却不一定等于 concurrentLevel，**ssize 一定是大于或等于 concurrentLevel 的最小的 2 的 N 次幂。**比如：默认情况下 concurrentLevel 是 16，则 ssize 为 16；若 concurrentLevel 为 14，ssize 为 16；若 concurrentLevel 为 17，则 ssize 为 32。为什么 Segment 的数组大小一定是 2 的次幂？其实主要是便于通过按位与的散列算法来定位 Segment 的 index。
- segmentMask：段掩码，假如 **ssize** 为 16，则段掩码为 16-1=15；segments 长度为 32，段掩码为 32-1=31。这样得到的所有 bit 位都为 1，可以更好地保证散列的均匀性
  segmentShift：2 的 sshift 次方等于 ssize，segmentShift=32-sshift。若 segments 长度为 16，segmentShift=32-4=28。这里之所以用 32 是因为 ConcurrentHashMap 里的 hash()方法输出的最大数是 32 位的。无符号右移 segmentShift，则意味着只保留高几位（其余位是没用的），然后与段掩码 segmentMask 位运算来定位 Segment。
- Cap 就是 **segment 里 HashEntry 数组的长度**。计算每个 Segment 平均应该放置多少个元素，**这个值 c 是向上取整的值**。比如初始容量为 15，Segment 个数为 4，则每个 Segment 平均需要放置 4 个元素。Cap 最小是 2 或者大于等于 C 的最小的 2 的 N 次幂。
- Segment 的容量 threshold=(int)cap*loadFactor,默认情况下，并发级别 concurrentLevel=16，ssize=16，初始容量 initialCapacity=16，负载因子 loadFactor=0.75，通过运算 cap=1, threshold = 0。

2.3 定位 segment

在插入和获取数据元素的时候，必须先通过散列算法定位到 segment。首先使用 Wang/Jenkins hash 的变种算法对元素的 hashCode 进行一次再散列。之所以进行在散列，目的是减少散列冲突，使元素能够均匀地分布在不同的 Segment 上，从而提高容器的存取效率。如果不适用再散列，散列冲突会非常严重，只要地位一样，无论高位是什么数，起散列值总是一样。使用在散列，可以把每一位的数据都散列开了，通过这种再散列能让数字的每一位都参加到散列运算中，从而减少散列冲突。

3 put 操作

```
public V put(K key, V value) {
   Segment<K,V> s;
   if (value == null)
```

```
        throw new NullPointerException();
    int hash = hash(key);
    int j = (hash >>> segmentShift) & segmentMask;
    if ((s = (Segment<K,V>)UNSAFE.getObject (segments, (j << SSHIFT) + SBASE)) == null)
       s = ensureSegment(j);
    return s.put(key, hash, value, false);
}
```

操作步骤如下：

- 判断 value 是否为 null，如果为 null，直接抛出异常。
- key 通过一次 hash 运算得到一个 hash 值。
- 将得到 hash 值向右按位移动 segmentShift 位，然后再与 segmentMask 做&运算得到 segment 的索引 j。默认 segmentShift=28，segmentMask=15，hash 值向右移动 28 位就变成这个样子：0000 0000 0000 0000 0000 0000 0000 xxxx，然后再用这个值与 segmentMask 做&运算，也就是**取最后四位的值**。**这个值确定 Segment 的索引**。即 hash 值的最高 sshift 位。
- 使用 **Unsafe 的方式**从 Segment 数组中获取该索引对应的 Segment 对象。
- 调用 Segment 的 put 方法【put 操作是要加锁的】向这个 Segment 对象中 put 值，这个 put 操作也基本是一样的步骤（通过&运算获取 HashEntry 的索引，然后 set）。

4. get 操作

```
public V get(Object key) {
    Segment<K,V> s;
    HashEntry<K,V>[] tab;
    int h = hash(key);
    long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;
    if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
        (tab = s.table) != null) {
        for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile
                 (tab, ((long)(((tab.length - 1) & h)) << TSHIFT) + TBASE);
             e != null; e = e.next) {
            K k;
            if ((k = e.key) == key || (e.hash == h && key.equals(k)))
                return e.value;
        }
    }
    return null;
}
```

操作步骤为：

- 和 put 操作一样，先通过 key 进行**两次 hash 确定应该去哪个 Segment** 中取数据。
- 使用 **Unsafe 获取对应的 Segment**，然后再进行一次&运算得到 HashEntry 链表的位置，然后从链表头开始遍历整个链表（因为 Hash 可能会有碰撞，所以用一个链表保存），如果找到对应的 key，则返回对应的 value 值，如果链表遍历完都没有找到对应的 key，则说明 Map 中不包含该 key，返回 null。get 操作是不需要加锁的（如果 value 为 null，会调用 readValueUnderLock，只有这个步骤会加锁），通过前面提到的 volatile 和 final 来确保数据安全。

5. size 操作

size 操作与 put 和 get 操作最大的区别在于，**size 操作需要遍历所有的 Segment 才能算出整个 Map 的大小**，而 put 和 get 都只关心一个 Segment。假设我们当前遍历的 Segment 为 SA，那么在遍历 SA 过程中其他的 Segment 比如 SB 可能会被修改，于是这一次运算出来的 size 值可能并不是 Map 当前的真正大小。牛逼的作者还有一个更好的 Idea：**先给 3 次机会，不 lock 所有的 Segment**，遍历所有 Segment，累加各个 Segment 的大小得到整个 Map 的大小，如果**某相邻的两次计算获取的所有 Segment 的更新的次数**（每个 Segment 都有一个 modCount 变量，这个变量在 Segment 中的 Entry 被修改时会加一，通过这个值可以得到每个 Segment 的更新操作的次数）**是一样的**，说明计算过程中没有更新操作，则直接返回这个值。如果这三次不加锁的计算过程中 Map 的更新次数有变化，则之后的计算先对所有的 Segment 加锁，再遍历所有 Segment 计算 Map 大小，最后再解锁所有 Segment。

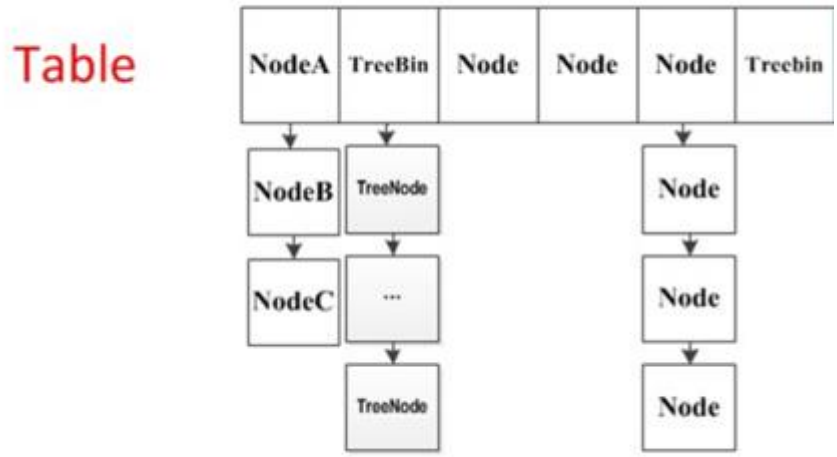
```
for (;;) {
  if (retries++ == RETRIES_BEFORE_LOCK) { //2
      for (int j = 0; j < segments.length; ++j)
           ensureSegment(j).lock(); // force creation
  }
  sum = 0L;
  size = 0;
  overflow = false;
  for (int j = 0; j < segments.length; ++j) {
       Segment<K,V> seg = segmentAt(segments, j);
       if (seg != null) {
           sum += seg.modCount;
           int c = seg.count;
           if (c < 0 || (size += c) < 0)
                overflow = true;
        }
  }
  if (sum == last)
      break;
  last = sum;
}
```

containsValue 操作采用了和 size 操作一样的想法。

6. 注意事项

- ConcurrentHashMap 中的 key 和 value 值都不能为 null，HashMap 中 key 可以为 null，HashTable 中 key 不能为 null。
- ConcurrentHashMap 是线程安全的类并不能保证使用了 ConcurrentHashMap 的操作都是线程安全的！
- ConcurrentHashMap 的 get 操作不需要加锁，put 操作需要加锁

18 ConcurrentHashMap【1.8】



1.8 的实现已经抛弃了 Segment 分段锁机制，利用 CAS+Synchronized 来保证并发更新的安全，底层依然采用数组+链表+红黑树的存储结构。

主要设计上的变化有以下几点:

- JDK1.8 的实现已经摒弃了 Segment 的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，锁住 node 来实现减小锁粒度。如果还是采用单向列表方式，那么查询某个节点的时间复杂度为 O(n)；因此，对于个数超过 8(默认值)的列表，jdk1.8 中采用了红黑树的结构，那么查询的时间复杂度可以降低到 O(logN)，可以改进性能。
- 设计了 MOVED 状态 当 resize 的中过程中 线程 2 还在 put 数据，线程 2 会帮助 resize。
- 使用 3 个 CAS 操作来确保 node 的一些操作的原子性，这种方式代替了锁。
- sizeCtl 的不同值来代表不同含义，起到了控制的作用。
- 并发控制使用 Synchronized 和 CAS 来操作。至于为什么 JDK8 中使用 synchronized 而不是 ReentrantLock，可能是 JDK8 中对 synchronized 有了足够的优化吧。
- 在其他方面也有一些小的改进，比如新增字段 transient volatile CounterCell[] counterCells; 可方便的计算 hashmap 中所有元素的个数，性能大大优于 jdk1.7 中的 size()方法。

1. 属性

下面看一下基本属性：

```
// node 数组最大容量
private  static  final  int  MAXIMUM_CAPACITY =  1  <<  30;
// 默认初始值，必须是 2 的幕数
private  static  final  int  DEFAULT_CAPACITY =  16;
// 负载因子
private  static  final  float  LOAD_FACTOR =  0.75f;
// 链表转红黑树阀值,> 8 链表转换为红黑树
static  final  int  TREEIFY_THRESHOLD =  8;
//树转链表阀值，小于等于 6
static  final  int  UNTREEIFY_THRESHOLD =  6;
```

```
private  transient  volatile  int  sizeCtl;
```

2. 重要概念

在开始之前，有些重要的概念需要介绍一下：

1. **table**：默认为 null，**初始化发生在第一次插入操作**，默认大小为 16 的数组，用来存储 Node 节点数据，扩容时大小总是 2 的幂次方。
2. **nextTable**：默认为 null，扩容时新生成的数组，其大小为原数组的两倍。
3. **sizeCtl** ：默认为 0，用来控制 table 的初始化和扩容操作，具体应用在后续会体现出来。
    - **-1 代表 table 正在初始化**
    - **-N 表示有 N-1 个线程正在进行扩容操作**
    - 其余情况：

      1、如果 table 未初始化，表示 table 需要初始化的大小。

      2、如果 table 初始化完成，表示 table 的容量，默认是 table 大小的 0.75 倍，居然用这个公式算 0.75（n - (n >>> 2)）。
4. **Node**：保存 key，value 及 key 的 hash 值的数据结构。

```
class Node<K,V> implements Map.Entry<K,V> {
  final int hash;
  final K key;
  volatile V val;
  volatile Node<K,V> next;
  ... 省略部分代码
}
```

其中 value 和 next 都用 volatile 修饰，保证并发的可见性。

5. **ForwardingNode**：一个特殊的 Node 节点，hash 值为-1，其中存储 nextTable 的引用。只有 table 发生扩容的时候，ForwardingNode 才会发挥作用，作为一个占位符放在 table 中表示当前节点为 null 或则已经被移动。

ConcurrentHashMap 在构造函数中只会初始化 sizeCtl 值，并不会直接初始化 table，而是延缓到第一次 put 操作。

3. 实例初始化

实例化 ConcurrentHashMap 时带参数时，会根据参数调整 table 的大小，假设参数为 **100，最终会调整成 256**，确保 **table 的大小总是 2 的幂次方**.

```
tableSizeFor(initialCapacity + a + 1));
private static final int tableSizeFor(int c) {
    int n = c - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    int resule = (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
    return resule;
}
```

4. table 初始化

table 初始化操作**会延缓到第一次 put 行为**。但是 put 是可以并发执行的，Doug Lea 是如何实现 table 只初始化一次的？sizeCtl 默认为 0，执行第一次 put 操作的线程会执行 Unsafe.compareAndSwapInt 方法修改 sizeCtl 为-1，有且只有一个线程能够修改成功，其它线程通过 Thread.yield()让出 CPU 时间片等待 table 初始化完成。如果一个线程发现 sizeCtl<0，意味着另外的线程执行 CAS 操作成功，当前线程只需要让出 cpu 时间片。

5. put 操作

put 操作采用 CAS+synchronized 实现并发插入或更新操作，具体实现如下。

- **hash 算法**

  int hash = spread(key.hashCode());
- **table 中定位索引位置，n 是 table 的大小**

  int index = (n - 1) & hash
- **获取 table 中对应索引的元素 f。**

Doug Lea 采用 Unsafe.getObjectVolatile 来获取。虽然 table 是 volatile 修饰的，但不能保证线程每次都拿到 table 中的最新元素，Unsafe.getObjectVolatile 可以直接获取指定内存的数据，保证了每次拿到数据都是最新的。

- 如果 f 为 null，说明 table 中这个位置第一次插入元素，利用 **Unsafe.compareAndSwapObject** 方法插入 Node 节点。
  - 如果 CAS 成功，说明 Node 节点已经插入，随后 addCount(1L, binCount)方法会检查当前容量是否需要进行扩容。
  - 如果 CAS 失败，说明有其它线程提前插入了节点，自旋重新尝试在这个位置插入节点。
- **如果 f 的 hash 值为-1，**说明当前 f 是 ForwardingNode 节点，**意味有其它线程正在扩容，**则一起进行扩容操作。
- 其余情况把新的 Node 节点**按链表或红黑树的方式插入到合适的位置**，这个过程采用同步内置锁 synchronized 实现并。
- 在节点 f 上进行同步，节点插入之前，再次利用 tabAt(tab, i) == f 判断，防止被其它线程修改。
  - 如果 f.hash >= 0，说明 f 是链表结构的头结点，遍历链表，如果找到对应的 node 节点，则修改 value，否则在链表尾部加入节点。
  - 如果 f 是 TreeBin 类型节点，说明 f 是红黑树根节点，则在树结构上遍历元素，更新或增加节点。
  - 如果链表中节点数 binCount >= TREEIFY_THRESHOLD(默认是 8)，则把链表转化为红黑树结构。

6. table 扩容

当 table 容量不足的时候，即 table 的元素数量达到容量阈值 sizeCtl，需要对 table 进行扩容。整个扩容分为两部分：

- 构建一个 nextTable，大小为 table 的两倍。
- 把 table 的数据复制到 nextTable 中。

ConcurrentHashMap 是支持并发插入的，扩容操作自然也会有并发的出现，这种情况下，第二步可以支持节点的并发复制，这样性能自然提升不少，但实现的复杂度也上升了一个台阶。先看第一步，构建 nextTable，毫无疑问，这个过程只能只有单个线程进行 nextTable 的初始化，通过 Unsafe.compareAndSwapInt 修改 sizeCtl 值，保证只有一个线程能够初始化 nextTable，扩容后的数组长度为原来的两倍，但是容量是原来的 1.5。
节点从 table 移动到 nextTable，大体思想是遍历、复制的过程。
遍历过所有的节点以后就完成了复制工作，把 table 指向 nextTable，并更新 sizeCtl 为新数组大小的 0.75 倍 ，扩容完成。

7. 红黑树构造

注意：如果链表结构中元素超过 **TREEIFY_THRESHOLD 阈值，默认为 8 个**，则把链表转化为红黑树，提高遍历查询效率。

可以看出，生成树节点的代码块是同步的（synchronized），进入同步代码块之后，再次验证 table 中 index 位置元素是否被修改过。主要根据 Node 节点的 hash 值大小构建二叉树。

8. get 操作

判断 table 是否为空，如果为空，直接返回 null。
计算 key 的 hash 值，并获取指定 table 中指定位置的 Node 节点，通过遍历链表或则树结构找到对应的节点，返回 value 值。

9. 总结

ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于 HashTable 和同步包装器包装的 HashMap，使用一个全局的锁来同步不同线程间的并发访问，同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器，这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成串行化的了。

JVM

1 详细 JVM 内存分布

答： Java 虚拟机所管理的内存将会包括以下几个运行是数据区域：

线程私有区
- ①程序计数器，记录正在执行的虚拟机字节码的地址；
- ②虚拟机栈：方法执行的内存区，每个方法执行时会在虚拟机栈中创建栈帧；
- ③本地方法栈：虚拟机的 Native 方法执行的内存区；

线程共享区
- ④Java 堆：对象分配内存的区域；
- ⑤方法区：存放类信息、常量、静态变量、编译器编译后的代码等数据；
- 常量池：存放编译器生成的各种字面量和符号引用，是方法区的一部分。

①程序计数器：当前线程所执行的字节码行号指示器。每一条线程都有一个独立的程序计数器，即为线程私有的。注意，此内存区域是唯一一个没有规定任何 OutOfMemoryError 情况的区域。

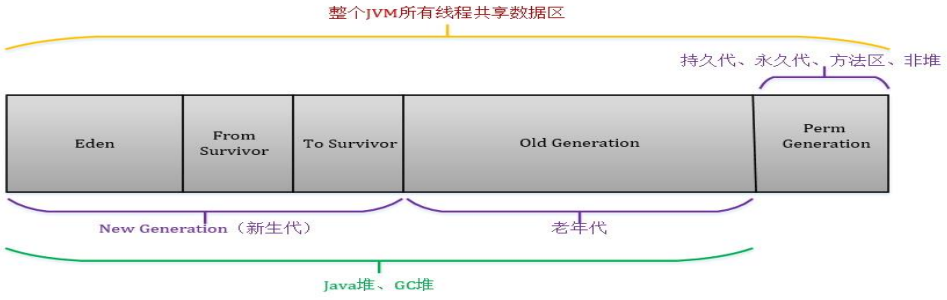
②Java 虚拟机栈(Stack)：生命周期与线程相同，虚拟机栈用于存放局部变量表，操作数栈，动态链接，方法出口等信息。两种异常状况：①如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常；②如果虚拟机栈在动态扩展时无法申请到足够的内存，将抛出 OutOfMemoryError 异常。

③本地方法栈：本地方法栈则为虚拟机使用到的 Native 方法服务，两种异常。

④Java 堆(Heap)：Java 堆（Java Heap）是被所有线程共享的内存区域，在虚拟机启动时创建。这个区域是用来存放对象实例的，几乎所有对象实例都会在这里分配内存。堆是 Java 垃圾收集器管理的主要区域（GC 堆）。Java 堆可以细分为：新生代和老年代；再细致一点的有 Eden 空间，From Survivor 空间，To Survivor 空间等。空间比例为 Eden:S0:S1==8:1:1。Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。可以通过-Xmx 和-Xms 控制。

⑤方法区：方法区（Method Area）也叫永久代，永久代也是被所有线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量（JDK7 中被移到 Java 堆）、即使编译器编译后的代码等数据。

运行时常量池，它是方法区的一部分，用于存放编译期生成的各种字面量和符号引用（其实就是八大基本类型的包装类型和 String 类型数据**（JDK7 中被移到 Java 堆）**）。从 JDK7 开始永久代的移除工作，贮存在永久代的一部分数据已经转移到了 Java Heap 或者是 Native Heap。但**永久代仍然存在于 JDK7，并没有完全的移除**，随着 **JDK8 的到来，JVM 不再有 PermGen。**



- **JVM 中共享数据空间可以分成三个大区**，新生代（Young Generation）、老年代（Old Generation）、永久代（Permanent Generation），其中 JVM 堆分为新生代和老年代
- 新生代可以划分为三个区，Eden 区（存放新生对象），两个幸存区（From Survivor 和 To Survivor）（存放每次垃圾回收后存活的对象）。

2 JVM 内存模型

Java 内存模型是 JVM 的抽象模型（主内存，本地内存）；

(1)主内存与工作内存

Java 内存模型的主要目标是**定义程序中各个变量的访问规则**，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量包括：实例字段、静态字段和构成数组对象的元素，但不包括局部变量与方法参数。

Java 内存模型规定了所有的变量都存储在主内存中。每条线程还有自己的工作内存，线程的工作内存中保存了该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。

(2) 内存间的交互操作

Java 内存模型定义了以下八种操作来完成，每一种操作都是原子的、不可再分的：

①**lock（锁定）**：作用于主内存的变量，把一个变量标识为一条线程独占状态；

②**unlock（解锁）**：作用于主内存变量，把锁定状态的变量释放出来，释放后才可以被其他线程锁定；

③**read（读取）**：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中；

④**load（载入）**：作用于工作内存的变量，把 read 操作从主内存中得到的变量值放入工作内存的变量副本中。

⑤**use（使用）**：作用于工作内存的变量；

⑥**assign（赋值）**：作用于工作内存的变量；

⑦**store（存储）**：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中；

⑧**write（写入）**：作用于主内存的变量，它把 store 操作从工作内存中一个变量的值传送到主内存的变量中。

如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行 **read 和 load** 操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行 **store 和 write** 操作。Java 内存模型只要求上述操作**必须按顺序执行，而没有保证必须是连续执行**。

(3) Java 内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

①不允许 read 和 load、store 和 write 操作之一单独出现，既不允许一个变量从主内存读取了但工作内存不接受，或者从工作内存发起回写了但主内存不接受的情况出现。

②不允许一个线程丢弃它的最近 assign 的操作，即变量在工作内存中改变了之后必须同步到主内存中。

③不允许一个线程无原因地（没有发生过任何 assign 操作）把数据从工作内存同步回主内存中。

④一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化的变量。即就是对一个变量实施 use 和 store 操作之前，必须先执行过了 assign 和 load 操作。

⑤一个变量在同一时刻只允许一条线程对其进行 lock 操作，但 lock 操作可以被同一条线程重复执行多次，多次执行 lock 后，只有执行相同次数的 unlock 操作，变量才会被解锁。lock 和 unlock 必须成对出现;

⑥如果对一个变量执行 lock，将会清空工作内存中此变量的值，使用这个变量前需要重新执行 load 或 assign 操作初始化变量的值;

⑦如果一个变量事先没有被 lock 操作锁定，则不允许对它执行 unlock 操作；也不允许去 unlock 一个被其他线程锁定的变量;

⑧对一个变量执行 unlock 操作之前，必须先把此变量同步到主内存中（执行 store 和 write 操作）。

(4)对于 volatile 型变量的特殊规则

当一个变量定义为 volatile 之后，它将具备两种特性：第一：保证此变量对所有线程的可见性。第二：禁止指令重排序 (在 JDK1.5 中才被完全修复) 。

(5)对于 long 和 double 型变量的特殊规则

Java 模型对于 64 为的数据类型（long 和 double），在模型中特别定义了一条相对宽松的规定：允许虚拟机将没有被 volatile 修饰的 64 位数据的读写操作分为两次 32 为的操作来进行，即允许虚拟机实现选择可以不保证 64 位数据类型的 load、store、read 和 write 这 4 个操作的原子性。目前虚拟机都把 64 位数据的读写操作作为原子操作来对待。

(6)原子性、可见性和有序性

JAVA 内存模型是围绕着并发过程中如何处理原子性、可见性和有序性来建立的：

①原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。基本数据类型的访问读写是具备原子性的（例外是 long 和 double），在 synchronized 块之间的操作也具备原子性。

②可见性：可见性是指当一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。除了 volatile 之外，Java 还有两个关键字能实现可见性：synchronized 和 final。

③有序性：即程序执行的顺序按照代码的先后顺序执行。Java 提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性。

(7)先行发生原则 happen-before

这些先行发生关系无须任何同步就已经存在，如果不再此列就不能保障顺序性，虚拟机就可以对它们任意地进行重排序：

1.程序次序规则：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。

2.管程锁定规则：同一个锁，一个 unlock 操作先发生于后面对同一个锁的 lock 操作。

3.Volatile 变量规则：对一个 volatile 变量的写操作先行发生于后面对这个变量的读操作。

4.线程启动规则：Thread 对象的 start()方法先行发生于此线程的每一个动作

5.线程终止规则：线程中的所有操作都先发生于对此线程的终止；

6.线程中断规则：对线程 interrupt()方法的调用先行发生于被中断线程的代码检测到中断时间的发生；

7.对象终结规则：一个对象的初始化完成(构造函数执行结束)先行发生于它的 finalize()方法的开始

8.传递性：如果操作 A 先发生于操作 B，操作 B 先发生于操作 C，那就可以得出操作 A 先发生于操作 C。

3 讲讲什么情况下会出现内存溢出，内存泄漏？

内存溢出(out of memory)：指程序在申请内存时，没有足够的内存空间供其使用，出现 out of memory。

内存泄露(memory leak)：指程序在申请内存后，无法释放已申请的内存空间。

内存溢出的原因：

1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；

2.集合类中有对对象的引用，使用完后未清空，使得 JVM 不能回收；

3.代码中存在死循环或循环产生过多重复的对象实体；

4.使用的第三方软件中的 BUG；

5.启动参数内存值设定的过小

内存溢出的解决方案：

第一步，修改 JVM 启动参数，直接增加内存。(-Xms，-Xmx 参数一定不要忘记加。)

第二步，检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误。

第三步，对代码进行走查和分析，找出可能发生内存溢出的位置。

StackOverflowError：线程请求的栈深度大于虚拟机所允许的深度时抛出该异常。要么是方法调用层次过多（比如存在无限递归调用），要么是线程栈太小。减少方法调用层次；调整-Xss 参数增加线程栈大小。

OutOfMemoryError: PermGen space：在 JDK8 之前的 HotSpot 实现中，类的元数据如方法数据、方法信息（字节码，栈和变量大小）、运行时常量池、已确定的符号引用和虚方法表等被保存在永久代中，32 位默认永久代的大小为 64M，64 位默认为 85M，可以通过参数-XX:MaxPermSize 进行设置。JDK8 的 HotSpot 中，把永久代从 Java 堆中移除了，并把类的元数据直接保存在本地内存区域。

4 说说 Java 线程栈

jstack pid 可以查看线程栈：JVM 线程堆栈是一个给定时间的快照，提供所有被创建出来的 Java 线程的完整清单。每一个被发现的 Java 线程都会给你如下信息： - 线程的名称，- 线程类型 & 优先级， - Java 线程 ID， - 状态 以及 - Java 线程栈跟踪。

(1) found one java-level deadlock ，线程 1 waiting to lock A，locked B；线程 2 waiting to lock B，locked A；

5 JVM 垃圾回收机制--新生代到年老代

堆大小=新生代+老年代；（新生代占堆空间的 1/3、老年代占堆空间 2/3）。新生代又分为 eden、from survivor、to survivor(8:1:1)。

当新创建对象时一般在新生代中分配内存空间，当新生代垃圾收集器回收几次之后仍然存活的对象会被移动到年老代内存中，当大对象在新生代中无法找到足够的连续内存时也直接在年老代中创。当 Eden 区内存不够的时候就会触发 MinorGC，对新生代区进行一次垃圾回收。

MinorGC 的过程：MinorGC 采用**复制算法**。首先，把 Eden 和 Servivor From 中还存活的对象一次性复制到 Servicor To 区域（如果有对象的年龄以及达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1（如果 Servicor To 不够位置了就放到老年区，当年龄达到某个值时（默认 15，通过设置参数 -xx:maxtenuringThreshold 来设置），这些对象就会成为老年代）；然后，清空 Eden 和 ServicorFrom 中的对象；最后，ServicorTo 和 Servicor From 互换，原 Servicor To 成为下一次 GC 时的 Servicor From 区。

注意：**(1) 对象进入老年代的 4 种情况：**

①假如进行 Minor GC 时发现，存活的对象在 ToSpace 区中存不下，那么把存活的对象存入老年代;②大对象直接进入老年代；③长期存活的对象将进入老年代；④动态对象年龄判定：如果在 From 空间中，相同年龄所有对象的大小总和大于 From 和 To 空间总和的一半，那么年龄大于等于该年龄的对象就会被移动到老年代，而不用等到 15 岁(默认)：。

(2) 触发 Full GC：①调用 System.gc 时，系统建议执行 Full GC，但是不必然执行；②老年代空间不足；③perm gen 空间不足；④通过 Minor GC 后进入老年代的平均大小大于老年代的可用内存；⑤由 Eden 区、From Space 区向 To Space 区复制时，对象大小大于 To Space 可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小。

(3) 内存分配合回收策略：

- 对象优先在 Eden 分配；
- 大对象直接进入老年代；
- 长期存活的对象将进入老年代；
- 动态对象年龄判定；
- 空间分配担保：

①在发生 Minor GC 之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么 Minor GC 可以确保是安全的。②如果不成立，则虚拟机会查看 HandlerPromotionFailure 这个参数设置的值(true 或 flase)是否允许担保失败，③如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试进行一次 Minor GC；④如果小于，或者 HandlerPromotionFailure 为 false，那么这次 Minor GC 将升级为 Full GC。

6 JVM 垃圾回收机制-- 频繁 full GC

开启了 -XX:+HeapDumpBeforeFullGC。用 jvisualvm 查看，查看占用内存多的数据：

①程序里有内存缓存，缓存的是字符串，内存缓存逐渐增多，逐步移步老年代，最终导致爆满。

②有大量拼接字符串的地方，

③static 的变量，存储大量的字符串，排名第六的是 hashMap，猜想可能是有 static 的 hashMap？？

解决方案：①Xms 值与 Xmx 相等，这样就不会因为所使用的 Java 堆不够用而进行调节；

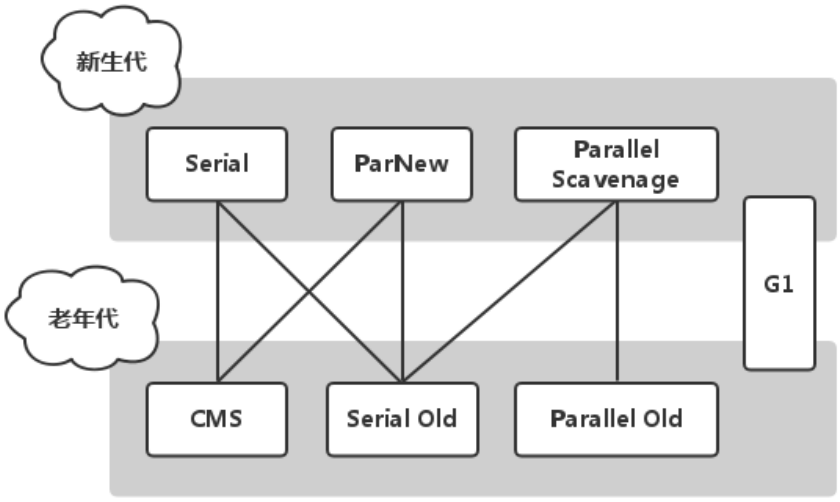
②-XX:**CMSInitiatingOccupancyFraction**=80 -XX:+UseCMSInitiatingOccupancyOnly：在 old 区达到 80%时 FGC；

③-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -**Xloggc**:gc.log:打印 gc 信息。

④开启 XX:+CMSScavengeBeforeRemark：在 FGC 前执行 YGC，防止老年代对象的 GC ROOT 在新生代的话；

④程序优化：尽早释放无用对象的引用，特别注意一些复杂对象，如数组，队列等，将不用的引用对象赋为 null；尽量少用 finalize 函数

7 各种回收器，各自优缺点，重点 CMS、G1



(1) Serial 收集器（串行 GC）

Serial 是一个采用**单个线程**并基于**复制算法**工作在**新生代**的收集器，进行垃圾收集时，必须暂停其他所有的工作线程。对于单 CPU 环境来说，Serial 由于没有线程交互的开销，可以很高效的进行垃圾收集动作，是 Client 模式下新生代默认的收集器。

(2) ParNew 收集器（并行 GC）

ParNew 是 serial 的多线程版本，许多运行在 Server 模式下的虚拟机中首选的新生代收集器，除 Serial 外，只有它能与 CMS 收集器配合工作。除了使用多条线程进行垃圾收集之外，其余与 Serial 一样。

(3) Parallel Scavenge 收集器（并行回收 GC）

Parallel Scavenge 是一个采用**多线程**基于**复制算法**并工作在新生代的收集器，其关注点在于达到一个可控的吞吐量，经常被称为“吞吐量优先”的收集器。(吞吐量 = 用户代码运行时间 /(用户代码运行时间 + 垃圾收集时间))。

(4) Serial Old 收集器（串行 GC）

Serial Old 是 Serial 收集器的老年代版本，同样是单线程收集器，使用“标记-整理算法”。

(5) Parallel Old 收集器（并行 GC）

Parallel Old 是 Parallel Scavenge 收集器的老年代版本，使用多线程和“标记-整理算法”算法。

(6) CMS 收集器（并发 GC）

CMS(Concurrent Mark Sweep) 是一种以**获得最短回收停顿时间为目标**的收集器，基于“标记-整理算法”算法。整个过程分为以下 4 步：

①**初始标记**：这个过程只是标记一下 GC Roots 能够直接关联的对象，但是仍然会 Stop The World；

②**并发标记**：进行 GC Roots Tracing 的过程，可以和用户线程一起工作。

③**重新标记**：用于修正并发标记期间由于用户程序继续运行而导致标记产生变动的那部分记录，这个过程会暂停所有线程，但其停顿时间远比并发标记的时间短；

④**并发清理**：可以和用户线程一起工作。

优点： **并发收集、低停顿**；

缺点：① **CMS 对 CUP 资源非常敏感** 。在并发阶段，占用部分线程而导致应用程序变慢， **总吞吐量降低**。

②**CMS 无法处理浮动垃圾**。可能出现 Concurrent Mode Failure 失败而导致另一次 fullGC 的产生。CMS 不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在 JDK1.6 中，启动阈值为 92%。要是 CMS 运行期间预留的内存无法满足程序需要就会出现 Concurrent Mode Failure 失败，这是就会临时启用 Serial old 收集器来重新进行老年代的垃圾收集，这样停顿时间就更长了。

③CMS 是基于“标记-清除”算法实现的收集器，**在收集结束时会有大量空间碎片产生**，可能导致出现老年代剩余空间很大，却无法找到足够大的连续空间分配当前对象，不得不提前触发一次 Full GC。

(7) G1 收集器

G1（Garbage First）是 JDK1.7 提供的一个工作**在新生代和老年代**的收集器，基于**“标记-整理”**算法实现，在收集结束后可以避免内存碎片问题。

G1 优点：①并行与并发：充分利用多 CPU 来缩短 Stop The World 的停顿时间；
②分代收集：采用不同的方式处理新建的对象、已经存活一段时间和经历过多次 GC 的对象；
③空间整合：整体来看是基于“标记 -整理”算法实现的，从局部看来是基于“复制”算法实现的;
④停顿预测：G1 中可以建立可预测的停顿时间模型，能让使用者明确指定在 M 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒。

使用 G1 收集器时，整个 Java 堆会被划分为多个大小相等的独立区域 Region，新生代和老年代不再是物理隔离了，都是一部分 Region（不需要连续）的集合。G1 会跟踪各个 Region 的垃圾收集情况（回收空间大小和回收消耗的时间），维护一个优先列表，根据允许的收集时间，优先回收价值最大的 Region。为了可达性判定确定对象是否存活的时候保证准确性，使用 Remembered Set 来避免全堆扫描。G1 的每个 region 都有一个与之对应的 Remembered Set，在 GC 根节点的枚举范围中加入 Remembered Set 及可保证不对全堆扫描也不会有遗漏。

8 各种回收算法

答：垃圾收集算法主要有：标记-清除、复制和标记-整理。

(1)标记-清除算法：

算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

缺点：①效率问题：标记和清除两个过程的效率都不高；②空间问题：收集之后会产生大量的内存碎片，不利于大对象的分配。

(2)复制算法

复制算法将可用内存划分成大小相等的两块 A 和 B，每次只使用其中一块，当 A 的内存用完了，就把存活的对象复制到 B，并清空 A 的内存。

不仅提高了标记的效率，因为只需要标记存活的对象，同时也避免了内存碎片的问题，代价是可用内存缩小为原来的一半。

(3) 标记-整理算法

在老年代中，对象存活率较高，复制算法的效率很低。在标记-整理算法中，标记出所有存活的对象，并移动到一端，然后直接清理边界以外的内存。

(4) 分代收集算法

新生代中，只有少量对象存活，所以选择复制算法。Java 虚拟机对新生代的垃圾回收称为 Minor GC，次数比较频繁，每次回收时间也比较短。使用 java 虚拟机**-Xmn** 参数可以指定新生代内存大小。

老年代中因为对象存活率高，使用“标记-清理”或者“标记-整理”算法。Java 虚拟机对年老代的垃圾回收称为 MajorGC/Full GC。

永久代也使用标记-整理算法进行垃圾回收，java 虚拟机参数-XX:PermSize 和-XX:MaxPermSize 可以设置永久代的初始大小和最大容量。

9 Java 中的引用

在 Java 层面，一共有四种引用：强引用、软引用、弱引用、虚引用，从名字也可以发现，这几种引用的生命周期由强到弱。

(1)强引用（Strong Reference）：使用最普遍的引用，99%的代码可能都是强引用，是指创建一个对象并把这个对象赋给一个引用变量。类似“Object obj = new Object()”这类的应用。如果一个对象，和 GC Root 有强

引用的关系，当内存不足发生 GC 时，宁可抛出 OOM 异常，终止程序，也不会回收这些对象。相反，当一个对象，和 GC Root 没有强引用关系时，可能会被回收（因为可能还有其它引用），如果没有任何引用关系，GC 之后，该对象就被回收了。

(2)软引用（Soft Reference）：主要用来描述一些不那么重要的对象，内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存，这种特别适合用来实现缓存。比如：

```
Object reference = new MyObject();
Reference root = new SoftReference(reference);     //在 JDK1.2 之后，  SoftReference 实现软应用。
reference = null;                                  // MyObject 对象只有软引用
```

(3) 弱引用（Weak Reference）：相对于软引用，它的生命周期更短，当发生 GC 时，如果扫描到一个对象只有弱引用，不管当前内存是否足够，都会对它进行回收，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。在 java 中，用 java.lang.ref.WeakReference 类来表示。

(4) 虚引用（Phantom Reference）：和之前两种引用的最大不同是：它的 get 方法一直返回 null。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。虚引用的使用场景很窄，在 JDK 中，目前只知道在申请堆外内存时有它的身影。提供了 PhantomReference 类来实现弱应用。

10 类加载为什么要使用双亲委派模式，有没有什么场景是打破了这个模式？

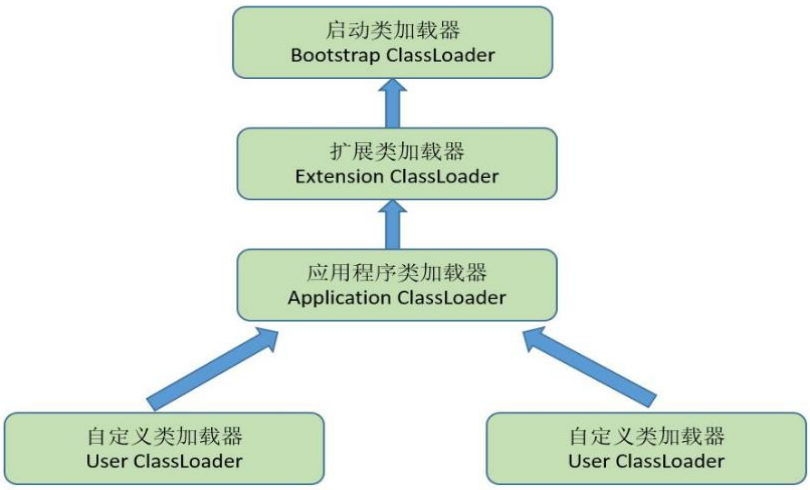
JVM 提供了 3 种类加载器：

(1)启动类加载器（Bootstrap ClassLoader）：是用本地代码实现的类装入器，负责加载 JAVA_HOME\lib 目录中的，或通过-Xbootclasspath 参数指定路径中的，且被虚拟机认可（按文件名识别，如 rt.jar）的类。

(2)扩展类加载器（Extension ClassLoader）：负责加载 JAVA_HOME\lib\ext 目录中的，或通过 java.ext.dirs 系统变量指定路径中的类库。

(3)应用程序类加载器（Application ClassLoader）：负责加载用户路径（classpath）上的类库,开发者可以直接使用这个类加载器，如果没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

JVM 基于上述类加载器，通过双亲委派模型进行类的加载，当然我们也可以通过继承 java.lang.ClassLoader 实现自定义的类加载器。这张图表示**类加载器的双亲委派模型**。



双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都是应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

使用双亲委派模型的好处就是：保证某个范围的类一定是被某个类加载器所加载的，这就保证在程序中同一个类不会被不同的类加载器加载。例如类 java.lang.Object,它存放在 rt.jar 中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 Object 类在程序的各种类加

载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为 java.lang.object 的类，并放在程序的 ClassPath 中，那系统中将会出现多个不同的 Object 类，Java 类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱。

双亲委派模型的实现：实现双亲委派模型代码都集中在 java.lang.ClassLoader 的 loadClass()方法之中，首先检查是否已经被加载过，若没有加载则调用父加载器的 loadClass()方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载失败，抛出 ClassNotFoundException 异常后，再调用自己的 findClass()方法进行加载。

打破双亲委派机制则不仅要继承 ClassLoader 类，还要重写 loadClass 和 findClass 方法。

11 JAVA 命令

Jps

Jstat

Jinfo

Jmap

Jstack

12 JVM 常见启动参数

-Xms / -Xmx — 堆的初始大小 / 堆的最大大小

-Xmn — 堆中年轻代的大小

-XX:-DisableExplicitGC — 让 System.gc()不产生任何作用

-XX:+PrintGCDetails — 打印 GC 的细节

-XX:+PrintGCDateStamps — 打印 GC 操作的时间戳

-XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小

-XX:NewRatio — 可以设置老生代和新生代的比例

-XX:PrintTenuringDistribution — 设置每次新生代 GC 后输出幸存者乐园中对象年龄的分布

-XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值

-XX:TargetSurvivorRatio：设置幸存区的目标使用率

12 怎么定位 JVM 出现的问题

缓存

1 Redis 用过哪些数据数据，以及 Redis 底层怎么实现

Redis 是一种基于键值对（key-value）的 NoSQL 数据库，redis 中的值可以是由 **String**（字符串）、**hash**（哈希）、**list**（列表）、**set**（集合）、**zset**（有序集合）、Bitmaps（位图）等多种数据结构和算法组成。Redis 还提供了键过期、发布订阅、事务、Lua 脚本等附加功能。Redis 使用了**单线程架构**和 I/O 多路复用模型来实现高性能的内存数据库服务。

查看所有键	keys *	键总数	dbsize
键是否存在	exists key	删除键	del key
键过期	expire key seconds	键的剩余过期时间	ttl key

(1) 字符串 String

字符串对象的编码可以是 int、raw 或者 embstr。embstr 应该是 Redis 3.0 新增的数据结构。如果字符串对象的长度小于 39 字节，就用 embstr 对象。否则用传统的 raw 对象。

①设置值　set key value [ex seconds] [px milliseconds] [nx|xx]

ex seconds：为键设置秒级过期时间　　px seconds：为键设置毫秒级过期时间

nx：键必须不存在，才能设置成功　　xx：键必须存在，才能设置成功，用于更新。

获取值	get key	计数	incr key
批量设置/获取值	mset / mget key	设置并返回原值：	getset key value
setex key seconds value		setnx key value	失败返回 0，成功返回 OK

(2) 哈希 hash

key= {{field1,value1},...{ fieldN,valueN}}，底层实现类似 Java 里面的 Map<String,Object>

hset key field value		hget key field	
hmget key field [field ...]		Hmset key field value [field value ...]	
hkeys key	获取所有 field	hvals key	获取所有 value
hexists key field	field 是否存在	hgetall key	获取所有的 field-value

(3) **列表 list**

列表用来存储多个有序的字符串。列表中的元素 **有序 可重复**。list 的实现为一个双向链表，即可以支持反向查找和遍历。

增加 rpush	lpush	linsert	查 lrange lindex llen
删除 lpop rpop lrem ltrim		修改 lset	
阻塞操作 blpop/brpop key [key …] timeout		列表为空，阻塞 timeout 后返回	

(4) 集合 set

不允许重复 无序， set 的内部实现是一个 value 永远为 null 的 HashMap，即为 hashtable。

添加	sad key element [element ...]	个数	scard key
是否存在	sismenmber key element	获取所有元素	smembers key
删除	srem key element [element ...]	随机弹出元素	spop key
交集/并集/差集:	sinter / suinon / sdiff key [key ...]		

(5) 有序集合 zset

sorted set 可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。**zset 的成员是唯一的,但分数(score)却可以重复**。

sorted set 的内部使用 HashMap 和跳跃表(SkipList)来保证数据的存储和有序。skiplist 是一种跳跃表，它实现了有序集合中的快速查找，在大多数情况下它的速度都可以和平衡树差不多。但它的实现比较简单，

可以作为平衡树的替代品。

添加	zadd key score member [score member …]	范围的成员	zrange key start end
删除	zrem key member [member …]	成员个数	zcard key

(6) 注意

persist 命令可以删除任意类型键的过期时间，但是 set 命令也会删除字符串类型键的过期时间，这在开发时容易被忽视。

2 Redis 缓存穿透，缓存雪崩

(1) 缓存穿透

什么是缓存穿透？指查询一个根本不存在的数据，缓存层和存储层都不会命中。缓存穿透将导致不存在的数据每次请求都要到存储层去查询，失去了缓存保护后端存储的意义。

解决办法：

①缓存空对象，将 null 变成一个值

如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

缓存空对象会有两个问题：

第一，空值做了缓存，意味着缓存层中存了更多的键，需要更多的内存空间，针对这类数据设置一个较短的过期时间，让其自动剔除。

第二，缓存层和存储层的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如过期时间设置为 5 分钟，如果此时存储层添加了这个数据，那此段时间就会出现缓存层和存储层数据的不一致，此时可以利用消息系统或者其他方式清除掉缓存层中的空对象。

②布隆过滤器拦截

将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。这种方式适合数据命中不高，数据性对固定、实时性低（数据集较大）的应用场景。

在访问缓存层和存储层之前，将存在的 key 用布隆过滤器提前保存起来，做第一层拦截。如果布隆过滤器认为该 key 不存在，那么就不会访问存储层，在一定程度保护了存储层。

2) 缓存雪崩

什么是缓存雪崩：当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如 DB)带来很大压力。

如何避免？ ①在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待。②不同的 key，设置不同的过期时间，让缓存失效的时间点尽量均匀。③保证缓存层服务高可用性。

3) 缓存击穿/热点 key 重建优化

什么是缓存击穿？对于一些设置了过期时间的 key，如果这些 key 可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里**针对某一 key 缓存**，前者则是很多 key。

如何解决？①使用互斥锁(mutex key)。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去 load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如 Redis 的 SETNX 或者 Memcache 的 ADD）去 set 一个 mutex key，当操作返回成功时，再进行 load db 的操作并回设缓存；否则，就重试整个 get 缓存的方法。此方法只允许一个线程重建缓存，其他线程待重建缓存的线程执行完，重新从缓存获取数据即可。②"提前"使用互斥锁(mutex key)；③永远不过期，这种方法对于性能非常友好，唯一不足的就是构建缓存时候，其余线程(非构建缓存的线程)可能访问的是老数据。

```
public String get(key) {
    String value = redis.get(key);
```

```
    if (value == null) { //代表缓存值过期
            //设置 3min 的超时，防止 del 操作失败的时候，下次缓存过期一直不能 load db
            if (redis.setnx(key_mutex, 1, 3 * 60) == 1) {   //代表设置成功
                value = db.get(key);
                redis.set(key, value, expire_secs);
                redis.del(key_mutex);
            } else {   //这个时候代表同时候的其他线程已经 load db 并回设到缓存了
                sleep(50);
                get(key);   //重试
            }
     } else {
            return value;
     }
 }
```

3 如何使用 Redis 来实现分布式锁

当前没有锁（key 不存在），那么就进行加锁操作，并对锁设置个有效期，同时 value 表示加锁的客户端；已有锁存在，不做任何操作。

```
public class RedisTool {
    private static final Long RELEASE_SUCCESS = 1L;
    private static final String LOCK_SUCCESS = "OK";
    private static final String SET_IF_NOT_EXIST = "NX";
    private static final String SET_WITH_EXPIRE_TIME = "PX";
    /**
     * 尝试获取分布式锁
     * @param jedis Redis 客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @param expireTime 超期时间
     * @return 是否获取成功
     */
    public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {
        String result = jedis.set(lockKey, requestId, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME, expireTime);
        if (LOCK_SUCCESS.equals(result)) {
            return true;
        }
        return false;
    }
    /**
     * 释放分布式锁
     * @param jedis Redis 客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @return 是否释放成功
     */
```

```
 public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {
        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del',
KEYS[1]) else return 0 end";
        Object result = jedis.eval(script, Collections.singletonList(lockKey),
Collections.singletonList(requestId));
        if (RELEASE_SUCCESS.equals(result)) {
            return true;
        }
        return false;
    }
}
```

加锁：第一个为 key，我们使用 key 来当锁，因为 key 是唯一的。第二个为 value，我们传的是 requestId，通过给 value 赋值为 requestId，就知道这把锁是哪个请求加的了。requestId 可以使用 UUID.randomUUID().toString()方法生成。第三个为 nxxx，这个参数我们填的是 NX，意思是 SET IF NOT EXIST，即当 key 不存在时，我们进行 set 操作；若 key 已经存在，则不做任何操作；第四个为 expx，这个参数我们传的是 PX，意思是我们要给这个 key 加一个过期的设置，第五个为 time，代表 key 的过期时间。

首先，set()加入了 NX 参数，可以保证如果已有 key 存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即 key 被删除），不会发生死锁。最后，因为我们将 value 赋值为 requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。

错误示范：setnx() + expire()方法：由于这是两条 Redis 命令，不具有原子性，如果程序在执行完 setnx()之后突然崩溃，导致锁没有设置过期时间。那么将会发生死锁。

解锁：第一行代码，是一个简单的 Lua 脚本代码。第二行代码，将 Lua 代码传到 jedis.eval()方法里，并使参数 KEYS[1]赋值为 lockKey，ARGV[1]赋值为 requestId。eval()方法是将 Lua 代码交给 Redis 服务端执行，eval()方法可以确保原子性。

4 Redis 的并发竞争问题

方案一：可以使用独占锁的方式，类似操作系统的 mutex 机制。（网上有例子，http://blog.csdn.net/black_ox/article/details/48972085 不过实现相对复杂，成本较高）。

方案二：使用乐观锁的方式进行解决（成本较低，非阻塞，性能较高）。本质上是假设不会进行冲突，使用 redis 的命令 watch 进行构造条件。本质上是假设不会进行冲突，使用 redis 的命令 watch 进行构造条件。伪代码如下：

```
watch price
get price $price
$price = $price + 10
multi
set price $price
exec
```

在 WATCH 执行之后，EXEC 执行之前，有其他客户端修改了 key 的值，那么当前客户端的事务就会失败。

5 Redis 持久化

几种实现方式？Redis 支持两种方式的持久化：①RDB：RDB 持久化是把当前进程数据生成快照保存到硬盘的过程。可以**在指定的时间间隔内生成内存中整个数据集的持久化快照**。快照文件默认被存储在当前文件夹中，名称为 dump.rdb，可以通过 dir 和 dbfilename 参数来修改默认值。②AOF，AOP（append only file）

将 Reids 的操作日志以追加的方式写入文件，解决了数据持久化的实时性。开启 AOF 功能需要设置配置：appendonly yes，默认是不开启。AOF 文件名通过 appendfilename 配置设置，默认文件名是：appendonly.aof。

RDB 存在哪些优势呢？①RDB 是一个紧凑压缩的二进制文件（使用 LZF 算法压缩），代表 Redis 在某个时间点上的数据快照，非常适用于备份；②Redis 加载 RDB 恢复数据远远快于 AOF 的方式。

RDB 缺点？①RDB 是间隔一段时间进行持久化，没法做到实时持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候。②如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是 1 秒钟。

AOF 存在哪些优势呢？①可以带来更高的数据安全性，可以设置不同的 fsync 策略，Redis 中提供了 3 中同步策略，即每秒同步、每修改同步和不同步。②由于该机制对日志文件的写入操作采用的是 append 模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。③如果日志过大，Redis 可以自动启用重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。

AOF 缺点？①对于相同数量的数据集而言，AOF 文件通常要大于 RDB 文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。②根据同步策略的不同，AOF 在运行效率上往往会慢于 RDB。

怎么选择？做备份：当数据量大，且对恢复速度有要求，并且数据的一致性要求不高的话，可以只使用 RDB；②只做缓存：不用开启任何的持久化方式；③两者都开启的建议。

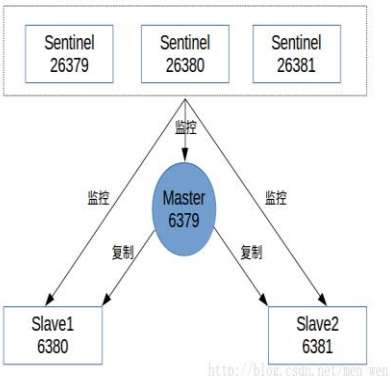
怎么实现的？在配置文件中修改：

save 900 1	#900 秒内如果有超过 1 个 key 被修改则发起保存快照
save 300 10	#300 秒内如果有超过 10 个 key 被修改则发起保存快照
save 60 10000	#60 秒内如果有超过 1000 个 key 被修改则发起保存快照
appendonly yes	#启用 AOF 方式
appendfsync always	#每次有新命令追加到 AOF 文件时就执行一次 fsync
appendfsync everysec	#每秒 fsync 一次：足够快，并且在故障时只会丢失 1 秒钟的数据
appendfsync no	#从不 fsync ：将数据交给操作系统来处理。更快，也更不安全

6 Redis 集群，高可用，缓存分片

(1) Redis Sentinel(['sentɪnl],哨兵)的高可用性：Redis Sentinel 是一个分布式架构，包含若干个 Sentinel 节点和 Redis 数据节点，每个 sentinel 节点会对数据节点和其余 Sentinel 节点进行监控，，当发现节点不可达时，会对节点做下线标识。当大多数 sentinel 节点都认为主节点不可达时，会选举出一个 Sentinel 节点来完成自动故障转移的工作。

Sentinel 用于管理多个 redis 服务器，主要执行三个任务：监控、提醒、自动故障转移。



(2) Redis Cluster：Redis 集群是一个提供在多个 Redis 间节点间共享数据的程序集，(节点数量为 6 个才能保证组成完整高可用的集群)集群使用了主从复制模型。

创建集群： redis-trib.rb create –replicas 1 127.0.0.1:7031 127.0.0.1:7032 127.0.0.1:7033 127.0.0.1:7034 127.0.0.1:7035 127.0.0.1:7036

其中：replicas 参数指定集群中每个主节点配备几个从节点，这里配置为 1.

连接集群：redis-cli -c -h 127.0.0.1 -p 7031

新增节点：添加节点；分配槽；指定从节点；

删除节点：将这个节点的数据重新分片到其他主节点上；删除

(3) Redis cluster 采用虚拟槽分区，所有的键根据哈希函数映射到 0~16383 整数槽内，

计算公式：**CRC16(key)&16383**，每个节点负责维护一部分槽以及槽所映射的键值数据。

集群功能限制：①key 批量操作支持有限，如 mset、mget，目前只支持具有相同 slot 值的 key 执行批量操作；②key 事务操作支持有限；③key 作为数据分区的最小粒度，因此不能将一个大的键值对象如 hash、list 等映射到不同的节点；④不支持多数据库空间；⑤复制结构只支持一层，从节点只能复制主节点。

7 Redis 的缓存失效策略

(1) 影响生存时间的一些操作

①DEL 命令来删除整个 key 来移除，或者被 SET 和 GETSET 命令覆盖原来的数据；

②执行 EXPIRE 命令，新指定的生存时间会取代旧的生存时间。

(2) redis 内存数据集大小上升到一定大小的时候，就会实行数据淘汰策略。

①volatile-lru：从已设置过期时间的数据集中挑选最近最少使用的数据淘汰；

②volatile-ttl：从已设置过期时间的数据集中挑选将要过期的数据淘汰；

③volatile-random：从已设置过期时间的数据集中任意选择数据淘汰

④allkeys-lru：从数据集中挑选最近最少使用的数据淘汰

⑤allkeys-random：从数据集中任意选择数据淘汰

⑥no-enviction（驱逐）：禁止驱逐数据

数据库

1 分页优化

第一种简单粗暴，就是不允许查看这么靠后的数据；

第二种方法，在查询下一页时把上一页的行 id 作为参数传递给客户端程序，然后 sql 就改成了：select * from table where id>3000000 limit 10;

如果主键 id 是自增的，并且中间没有删除和断点，那么还有一种方式，比如 100 页的 10 条数据：select * from table where id>100*10 limit 10;

第三种：select table.* from table inner join (select id from table limit 3000000,10) as tmp on tmp.id=table.id;

2 悲观锁&乐观锁

悲观锁：排它锁，当事务在操作数据时把这部分数据进行锁定，直到操作完毕后再解锁，其他事务操作才可操作该部分数据。这将防止其他进程读取或修改表中的数据。

一般使用 select ...for update 操作来实现悲观锁。当数据库执行 select for update 时会获取被 select 中的数据行的行锁，获取的行锁会在当前事务结束时自动释放。

如果加锁的时间过长，其他用户长时间无法访问，影响了程序的并发访问性，同时这样对数据库性能开销影响也很大

乐观锁：先进行业务操作，不到万不得已不去拿锁。在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，更新被拒绝的，可以让用户重新操作。

大多数基于**数据版本（Version）**记录机制实现(或者时间戳)：一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将 version 字段的值一同读出，数据每更新一次，version 值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的 version 值进行比对，如果数据库表当前版本号与第一次取出来的 version 值相等，则予以更新，否则认为是过期数据。

update t_goods set status=2,version=version+1 where id=#{id} and version=#{version};

3 表锁&行锁

	行锁	表锁
MyISAM		√
InnoDB	√	√

(1) MyISAM 表锁

表锁：MyISAM 存储引擎只支持表锁。MySQL 表级锁有两种模式：表共享锁（Table Read Lock）和表独占写锁（Table Write Lock）。**对 MyISAM 的读操作，不阻塞读，阻塞写；对 MyISAM 的写操作，则会阻塞读和写。**

如何加表锁：MyISAM 在执行查询语句（SELECT）前，会自动加读锁，在执行更新操作（UPDATE、DELETE、INSERT 等）前，会自动加写锁。

(2) InnoDB 的行锁模式及加锁方法

$$InnoDB\text{锁}\begin{cases}\text{行锁}\begin{cases}\text{共享锁(S)：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁}\\\text{排他锁(X)：允许获取锁的事务更新数据，阻止其他事务取得共享读锁和排他写锁}\end{cases}\\\text{表锁}\begin{cases}\text{意向共享锁(IS)：事务在给一个数据行加共享锁前必须先取得该表的 IS 锁}\\\text{意向排他锁(IX)：事务在给一个数据行加排他锁前必须先取得该表的 IX 锁}\end{cases}\end{cases}$$

InnoDB 实现了两种行锁：共享锁(S)和排它锁(X)；另外，为了允许行锁和表锁共存，实现多粒度锁机制，还有两种内部使用的意向锁（Intention Locks），是 InnoDB 自动加的，不需用户干预。意向锁都是表锁：意向共享锁(IS)和意向排它锁(IX)。

当前/请求锁模式	X	IX	S	IS
X	冲突	冲突	冲突	冲突
IX	冲突	兼容	冲突	兼容
S	冲突	冲突	兼容	兼容
IS	冲突	兼容	兼容	兼容

(3)间隙锁（Next-Key 锁）

当用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB 会给符合条件的已有数据的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙(GAP)”，InnoDB 也会对这个“间隙”加锁，这种锁机制是所谓的间隙锁（Next-Key 锁）。间隙锁的主要作用是为了防止出现幻读，但是它会把锁定范围扩大。

间隙锁的出现主要集中在同一个事务中先 delete 后 insert 的情况下，当我们通过一个参数去删除一条记录的时候，如果参数在数据库中存在， 那么这个时候产生的是普通行锁， 锁住这个记录，然后删除，然后释放锁。如果这条记录不存在，数据库会扫描索引，发现这个记录不存在，这个时候的 delete 语句获取到的就是一个间隙锁，然后数据库会向左扫描扫到第一个比给定参数小的值，向右扫描到第一个比给定参数大的值，然后以此为界，构建一个区间，锁住整个区间内的数据。

4 组合索引，最左原则

需要创建索引场景？

①主键自动建立唯一索引
②频繁作为查询条件的字段，查询中排序或者分组的字段，应该创建索引
③查询中与其他表关联的字段，外键关系建立索引
④频繁更新的字段不适合建立索引
⑤单键/组合索引的选择问题，who?(在高并发下倾向创建组合索引)

最左原则？如果想使用索引，必须保证按索引的最左边前缀来进行查询。如果 where 条件中只用到索引项，则加的是行锁；否则加的是表锁。

①匹配全值：对索引中的所有列进行精确匹配，索引可以被用到。
②匹配最左前缀：查询条件精确匹配索引的左边连续一个或几个列，索引可以被用到。

失效情况：

①如果条件中有 or，索引失效；
②like 查询是以%开头，索引失效；
③如果列类型是字符串，查询条件中没有使用引号引起来，索引失效；
④一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配
⑤查询条件中使用函数，索引失效；
⑥对于多列索引，不是使用的第一部分，索引失效；

5 事务的特性和隔离级别

事务具有 4 属性，通常称为事务的 ACID 属性：

①原性性（Actomicity）：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。
②一致性（Consistent）：在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以操持完整性；事务结束时，所有的内部数据结构（如 B 树索引或双向链表）也都必须是正确的。
③隔离性（Isolation）：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。
④持久性（Durable）：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

事务隔离级别

①Read uncommitted (未提交读)：一个事务可以读取另一个未提交事务的数据。这也被称为脏读(dirty read)。

②Read Committed(提交读)：一个事务只能读取数据库中已经提交过的数据，解决了脏读问题，但不能重复读，即一个事务内的两次查询返回的数据是不一样的。

③Repeatable Read(可重复读)：Mysql 默认的隔离级别。保证同一事务中多次读取同样记录的结果是一致的。但别的事务新增数据也能读取到，即但不能避免幻读的问题

④Serializable(可串行化)：读取的每一行数据上都加上共享锁和排他锁。

隔离级别	脏读	不可重读的	幻读
Read uncommitted	√	√	√
Read Committed	×	√	√
Repeatable Read	×	×	√
Serializable	×	×	×

6 myisam & innodb 区别

①InnoDB 支持事物，而 MyISAM 不支持事物

②InnoDB 支持行级锁，而 MyISAM 支持表级锁

③InnoDB 支持外键，而 MyISAM 不支持

④InnoDB 不支持全文索引，而 MyISAM 支持

⑤InnoDB 中不保存表的具体行数，要扫描一遍整个表来计算；MyISAM 保存行数。

⑥MyISAM 的索引和数据是分开的，并且索引是有压缩的，而 Innodb 是索引和数据是紧密捆绑的，没有使用压缩，从而会造成 Innodb 比 MyISAM 体积庞大不小。

7 分表分库

8 性能优化

9 索引分类

10 b 树，b+树，红黑树

有什么不同，MySQL 为什么用 b+树索引

：B+，hash；什么情况用什么索引