黑马程序员™
www.itheima.com

- 黑马程序员

Java 面试宝典

Beta5.0

Copyright© 2018 黑马程序员

Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

http://www.itheima.com Copyright© 2018 黑马程序员

Copyright© 2018 黑马程序员

第一章 内容介绍

该宝典是一份知识点全面又能不断更新，与时俱进的学习手册，不仅收录了作者亲身面试遇到的问题，还收录了近上万名黑马学子面试时遇到的问题。我们会一直不断地更新和充实该宝典，同时也希望读者朋友能够多多提供优质的面试题，也许下一个版本就有你提供的面试题哦。

本人的面试实战记录发布在黑马论坛：http://bbs.itheima.com/thread-196394-1-1.html

大家可以访问上面的网址，通过阳哥的实战记录略微感知一下真实面试的情况，从中学习一些面试技巧以便让自己在未来的面试中能够得心应手，顺利拿到自己喜欢的 offer。

注意：该面试宝典仅供参考，由于作者本人的知识水平有限加之编写时间仓促因此难免有 bug 的存在，希望大家见谅。

该宝典的一个明确目标是能够让 90%以上的 Java 技术面试题都落到该宝典中，如果您有不错的知识或者面试题，您可以发送到 wangzhenyang@itcast.cn，本人将不胜感激。让天下没有难学的知识，希望你我的努力能帮到更多的莘莘学子。

世间事，很多都可投机取巧，但技术却必须靠日积月累的努力来提高。本宝典更加注重的是知识的掌握，而不仅仅是对面试题的应付。在展示常见的面试问题以及回答技巧的同时还详细讲解了每一道题所包含的知识点，让读者不仅知其然，更知其所以然。

http://www.itheima.com Copyright© 2018 黑马程序员

第二章 JavaSE 基础

一、Java 面向对象

1. 面向对象都有哪些特性以及你对这些特性的理解

1）继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段。

2） 封装：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口。

3） 多态性：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当 A 系统访问 B 系统提供的服务时，B 系统有多种提供服务的方式，但一切对 A 系统来说都是透明的。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：1. 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2. 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

4）抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。

http://www.itheima.com Copyright© 2018 黑马程序员

注意：默认情况下面向对象有 3 大特性，封装、继承、多态，如果面试官问让说出 4 大特性，那么我们就把抽象加上去。

2. 访问权限修饰符 public、private、protected，以及不写(默认)时的区别(2017-11-12)

该题目比较简单，不同的权限修饰符的区别见下表。

修饰符	当前类	同 包	子 类	其他包
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

3. 如何理解 clone 对象

3.1 为什么要用 clone？

在实际编程过程中，我们常常要遇到这种情况：有一个对象 A，在某一时刻 A 中已经包含了一些有效值，此时可能会需要一个和 A 完全相同新对象 B，并且此后对 B 任何改动都不会影响到 A 中的值，也就是说，A 与 B 是两个独立的对象，但 B 的初始值是由 A 对象确定的。在 Java 语言中，用简单的赋值语句是不能满足这种需求的。要满足这种需求虽然有很多途径，但实现 clone（）方法是其中最简单，也是最高效的手段。

3.2 new 一个对象的过程和 clone 一个对象的过程区别

new 操作符的本意是分配内存。程序执行到 new 操作符时，首先去看 new 操作符后面的类型，因为知道了类型，才能知道要分配多大的内存空间。分配完内存之后，再调用构造函数，填充对象的各个域，这一步叫做对象的初始化，

http://www.itheima.com Copyright© 2018 黑马程序员

构造方法返回后，一个对象创建完毕，可以把他的引用（地址）发布到外部，在外部就可以使用这个引用操纵这个对象。

clone 在第一步是和 new 相似的，都是分配内存，调用 clone 方法时，分配的内存和原对象（即调用 clone 方法的对象）相同，然后再使用原对象中对应的各个域，填充新对象的域，填充完成之后，clone 方法返回，一个新的相同的对象被创建，同样可以把这个新对象的引用发布到外部。

3.3 clone 对象的使用

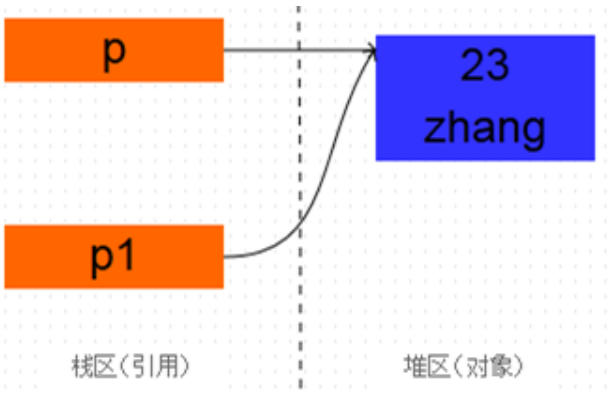
3.3.1 复制对象和复制引用的区别

```
1. Person p = new Person(23, "zhang");
2. Person p1 = p;
3. System.out.println(p);
4. System.out.println(p1);
```

当 Person p1 = p;执行之后， 是创建了一个新的对象吗？ 首先看打印结果：

```
1. com.itheima.Person@2f9ee1ac
2. com.itheima.Person@2f9ee1ac
```

可以看出，打印的地址值是相同的，既然地址都是相同的，那么肯定是同一个对象。p 和 p1 只是引用而已，他们都指向了一个相同的对象 Person(23, “zhang”）。 可以把这种现象叫做引用的复制。上面代码执行完成之后， 内存中的情景如下图所示：



而下面的代码是真真正正的克隆了一个对象。

```
1. Person p = new Person(23, "zhang");
```

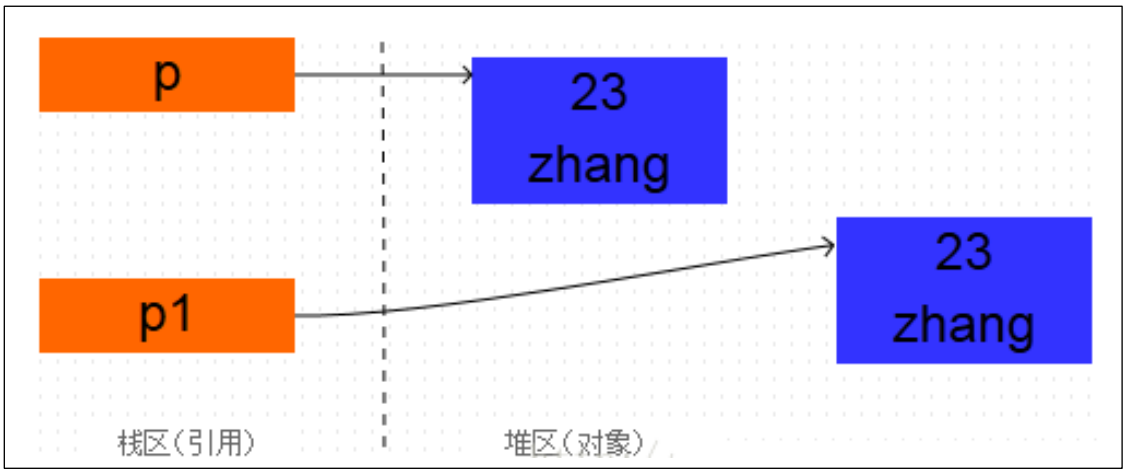

http://www.itheima.com Copyright© 2018 黑马程序员

```
2. Person p1 = (Person) p.clone();
3. System.out.println(p);
4. System.out.println(p1);
```

从打印结果可以看出，两个对象的地址是不同的，也就是说创建了新的对象， 而不是把原对象的地址赋给了一个新的引用变量：

```
1. com.itheima.Person@2f9ee1ac
2. com.itheima.Person@67f1fba0
```

以上代码执行完成后， 内存中的情景如下图所示：



3.3.2 深拷贝和浅拷贝

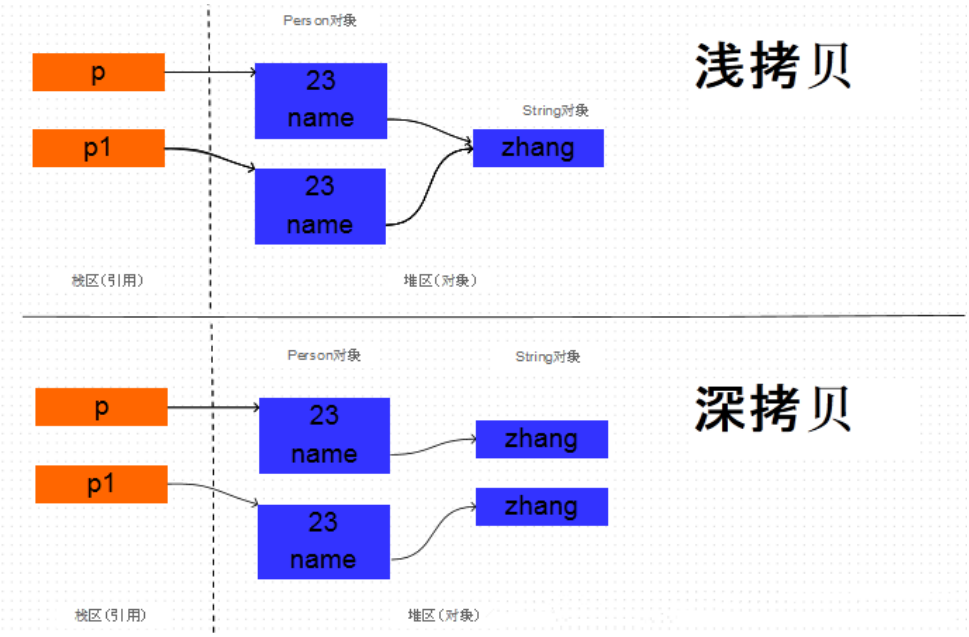
上面的示例代码中，Person 中有两个成员变量，分别是 name 和 age， name 是 String 类型， age 是 int 类型。代码非常简单，如下所示：

```
1. public class Person implements Cloneable{
2. privatint age ;
3.     private String name;
4.     public Person(int age, String name) {
5.         this.age = age;
6.         this.name = name;
7.     }
8.     public Person() {}
9.     public int getAge() {
10.         return age;
11.     }
12.     public String getName() {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
13.         return name;
14.     }
15.     @Override
16.     protected Object clone() throws CloneNotSupportedException {
17.         return (Person)super.clone();
18.     }
19. }
```

由于 age 是基本数据类型，那么对它的拷贝没有什么疑议，直接将一个 4 字节的整数值拷贝过来就行。但是 name 是 String 类型的， 它只是一个引用， 指向一个真正的 String 对象，那么对它的拷贝有两种方式： 直接将原对象中的 name 的引用值拷贝给新对象的 name 字段， 或者是根据原 Person 对象中的 name 指向的字符串对象创建一个新的相同的字符串对象，将这个新字符串对象的引用赋给新拷贝的 Person 对象的 name 字段。这两种拷贝方式分别叫做浅拷贝和深拷贝。深拷贝和浅拷贝的原理如下图所示：



下面通过代码进行验证。如果两个 Person 对象的 name 的地址值相同， 说明两个对象的 name 都指向同一个 String 对象，也就是浅拷贝， 而如果两个对象的 name 的地址值不同， 那么就说明指向不同的 String 对象， 也就是在拷贝 Person 对象的时候， 同时拷贝了 name 引用的 String 对象， 也就是深拷贝。验证代码如下：

```
1. Person p = new Person(23, "zhang");
2. Person p1 = (Person) p.clone();
3. String result = p.getName() == p1.getName()
4.         ? "clone 是浅拷贝的" : "clone 是深拷贝的";
5. System.out.println(result);
```

打印结果为：

```
6. clone 是浅拷贝的
```

所以，clone 方法执行的是浅拷贝， 在编写程序时要注意这个细节。

http://www.itheima.com Copyright© 2018 黑马程序员

如何进行深拷贝：

由上一节的内容可以得出如下结论：如果想要深拷贝一个对象，这个对象必须要实现 Cloneable 接口，实现 clone 方法，并且在 clone 方法内部，把该对象引用的其他对象也要 clone 一份，这就要求这个被引用的对象必须也要实现 Cloneable 接口并且实现 clone 方法。那么，按照上面的结论，实现以下代码 Body 类组合了 Head 类，要想深拷贝 Body 类，必须在 Body 类的 clone 方法中将 Head 类也要拷贝一份。代码如下：

```
1. static class Body implements Cloneable{
2.     public Head head;
3.     public Body() {}
4.     public Body(Head head) {this.head = head;}
5.     @Override
6.     protected Object clone() throws CloneNotSupportedException {
7.         Body newBody =  (Body) super.clone();
8.         newBody.head = (Head) head.clone();
9.         return newBody;
10.     }
11. }
12. static class Head implements Cloneable{
13.     public  Face face;
14.     public Head() {}
15.     @Override
16.     protected Object clone() throws CloneNotSupportedException {
17.         return super.clone();
18.     }  }
19. public static void main(String[] args) throws CloneNotSupportedException {
20.     Body body = new Body(new Head(new Face()));
21.     Body body1 = (Body) body.clone();
22.     System.out.println("body == body1 : " + (body == body1) );
23.     System.out.println("body.head == body1.head : " +  (body.head == body1.head));
24. }
```

打印结果为：

```
1. body == body1 : false
2. body.head == body1.head : false
```


http://www.itheima.com Copyright© 2018 黑马程序员

二、JavaSE 语法（2017-11-12-wl）

1. Java 有没有 goto 语句？ (2017-11-12-wl)

goto 是 Java 中的保留字，在目前版本的 Java 中没有使用。根据 James Gosling（Java 之父）编写的《The Java Programming Language》一书的附录中给出了一个 Java 关键字列表，其中有 goto 和 const，但是这两个是目前无法使用的关键字，因此有些地方将其称之为保留字，其实保留字这个词应该有更广泛的意义，因为熟悉 C 语言的程序员都知道，在系统类库中使用过的有特殊意义的单词或单词的组合都被视为保留字。

2. & 和 && 的区别（2017-11-12-wl)

&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。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是 true 整个表达式的值才是 true。

&&之所以称为短路运算是因为，如果&&左边的表达式的值是 false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&，例如在验证用户登录时判定用户名不是 null 而且不是空字符串，应当写为 username != null &&!username.equals("")，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的 equals 比较，否则会产生 NullPointerException 异常。注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

3. 在 Java 中，如何跳出当前的多重嵌套循环（2017-11-14-wl)

在最外层循环前加一个标记如 A，然后用 break A;可以跳出多重循环。(Java 中支持带标签的 break 和 continue 语句，作用有点类似于 C 和 C++中的 goto 语句，但是就像要避免使用 goto 一样，应该避免使用带标签的 break 和 continue，因为它不会让你的程序变得更优雅，很多时候甚至有相反的作用）。

http://www.itheima.com Copyright© 2018 黑马程序员

4. 两个对象值相同（x.equals(y) == true），但却可有不同的 hashCode，这句话对不对？（2017-11-14-wl）

不对，如果两个对象 x 和 y 满足 x.equals(y) == true，它们的哈希码（hashCode）应当相同。

Java 对于 eqauls 方法和 hashCode 方法是这样规定的：(1)如果两个对象相同（equals 方法返回 true），那么它们的 hashCode 值一定要相同；(2)如果两个对象的 hashCode 相同，它们并不一定相同。当然，你未必要按照要求去做，但是如果你违背了上述原则就会发现在使用容器时，相同的对象可以出现在 Set 集合中，同时增加新元素的效率会大大下降（对于使用哈希存储的系统，如果哈希码频繁的冲突将会造成存取性能急剧下降）。

关于 equals 和 hashCode 方法，很多 Java 程序员都知道，但很多人也就是仅仅知道而已，在 Joshua Bloch 的大作《Effective Java》（很多软件公司，《Effective Java》、《Java 编程思想》以及《重构：改善既有代码质量》是 Java 程序员必看书籍，如果你还没看过，那就赶紧去买一本吧）中是这样介绍 equals 方法的。

首先 equals 方法必须满足自反性（x.equals(x)必须返回 true）、对称性（x.equals(y)返回 true 时，y.equals(x)也必须返回 true）、传递性（x.equals(y)和 y.equals(z)都返回 true 时，x.equals(z)也必须返回 true）和一致性（当 x 和 y 引用的对象信息没有被修改时，多次调用 x.equals(y)应该得到同样的返回值），而且对于任何非 null 值的引用 x，x.equals(null)必须返回 false。实现高质量的 equals 方法的诀窍包括：1. 使用==操作符检查"参数是否为这个对象的引用"；2. 使用 instanceof 操作符检查"参数是否为正确的类型"；3. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配；4. 编写完 equals 方法后，问自己它是否满足对称性、传递性、一致性；5. 重写 equals 时总是要重写 hashCode；6. 不要将 equals 方法参数中的 Object 对象替换为其他的类型，在重写时不要忘掉 @Override 注解。

5. 是否可以继承 String （2017-11-14-wl）

String 类是 final 类，不可以被继承。

http://www.itheima.com Copyright© 2018 黑马程序员

继承 String 本身就是一个错误的行为，对 String 类型最好的重用方式是关联关系（Has-A）和依赖关系（Use-A）而不是继承关系（Is-A）。

6. 当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递？（2017-11-14-wl）

是值传递。Java 语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。C++和 C#中可以通过传引用或传输出参数来改变传入的参数的值。说明：Java 中没有传引用实在是非常的不方便，这一点在 Java 8 中仍然没有得到改进，正是如此在 Java 编写的代码中才会出现大量的 Wrapper 类（将需要通过方法调用修改的引用置于一个 Wrapper 类中，再将 Wrapper 对象传入方法），这样的做法只会让代码变得臃肿，尤其是让从 C 和 C++转型为 Java 程序员的开发者无法容忍。

7. 重载（overload）和重写（override）的区别？重载的方法能否根据返回类型进行区分？（2017-11-15-wl）

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。重载对返回类型没有特殊的要求。

方法重载的规则：

1.方法名一致，参数列表中参数的顺序，类型，个数不同。

2.重载与方法的返回值无关，存在于父类和子类，同类中。

3.可以抛出不同的异常，可以有不同修饰符。

http://www.itheima.com Copyright© 2018 黑马程序员

方法重写的规则：

1.参数列表必须完全与被重写方法的一致，返回类型必须完全与被重写方法的返回类型一致。

2.构造方法不能被重写，声明为 final 的方法不能被重写，声明为 static 的方法不能被重写，但是能够被再次声明。

3.访问权限不能比父类中被重写的方法的访问权限更低。

4.重写的方法能够抛出任何非强制异常（UncheckedException，也叫非运行时异常），无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。

8. 为什么函数不能根据返回类型来区分重载？（2017-11-15-wl）

该道题来自华为面试题。

因为调用时不能指定类型信息，编译器不知道你要调用哪个函数。

例如：

```
1. float max(int a, int b);
2. int max(int a, int b);
```

当调用 max(1, 2);时无法确定调用的是哪个，单从这一点上来说，仅返回值类型不同的重载是不应该允许的。

再比如对下面这两个方法来说，虽然它们有同样的名字和自变量，但其实是很容易区分的：

```
1. void f() {}
2. int f() {}
```

若编译器可根据上下文（语境）明确判断出含义，比如在 int x=f()中，那么这样做完全没有问题。然而，我们也可能调用一个方法，同时忽略返回值；我们通常把这称为“为它的副作用去调用一个方法”，因为我们关心的不是返回值，而是方法调用的其他效果。所以假如我们像下面这样调用方法： f(); Java 怎样判断 f()的具体调用方式呢？而且别人如何识别并理解代码呢？由于存在这一类的问题，所以不能。

http://www.itheima.com Copyright© 2018 黑马程序员

函数的返回值只是作为函数运行之后的一个“状态”，他是保持方法的调用者与被调用者进行通信的关键。并不能作为某个方法的“标识”。

9. char 型变量中能不能存储一个中文汉字，为什么？（2017-11-16-wl）

char 类型可以存储一个中文汉字，因为 Java 中使用的编码是 Unicode（不选择任何特定的编码，直接使用字符在字符集中的编号，这是统一的唯一方法），一个 char 类型占 2 个字节（16 比特），所以放一个中文是没问题的。

补充：使用 Unicode 意味着字符在 JVM 内部和外部有不同的表现形式，在 JVM 内部都是 Unicode，当这个字符被从 JVM 内部转移到外部时（例如存入文件系统中），需要进行编码转换。所以 Java 中有字节流和字符流，以及在字符流和字节流之间进行转换的转换流，如 InputStreamReader 和 OutputStreamReader，这两个类是字节流和字符流之间的适配器类，承担了编码转换的任务；对于 C 程序员来说，要完成这样的编码转换恐怕要依赖于 union（联合体/共用体）共享内存的特征来实现了。

10. 抽象类(abstract class)和接口(interface)有什么异同？（2017-11-16-wl）

不同：

抽象类：

1.抽象类中可以定义构造器

2.可以有抽象方法和具体方法

3.接口中的成员全都是 public 的

4.抽象类中可以定义成员变量

5.有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法

6.抽象类中可以包含静态方法

http://www.itheima.com Copyright© 2018 黑马程序员

7.一个类只能继承一个抽象类

接口：

1.接口中不能定义构造器

2.方法全部都是抽象方法

3.抽象类中的成员可以是 private、默认、protected、public

4.接口中定义的成员变量实际上都是常量

5.接口中不能有静态方法

6.一个类可以实现多个接口

相同：

1.不能够实例化

2.可以将抽象类和接口类型作为引用类型

3.一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类

11. 抽象的(abstract)方法是否可同时是静态的(static)，是否可同时是本地方法(native)，是否可同时被 synchronized（2017-11-16-wl）

都不能。抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的。本地方法是由本地代码（如 C 代码）实现的方法，而抽象方法是没有实现的，也是矛盾的。synchronized 和方法的实现细节有关，抽象方法不涉及实现细节，因此也是相互矛盾的。

12. 阐述静态变量和实例变量的区别？（2017-11-16-wl）

静态变量: 是被 static 修饰符修饰的变量，也称为类变量，它属于类，不属于类的任何一个对象，一个类不

http://www.itheima.com Copyright© 2018 黑马程序员

管创建多少个对象，静态变量在内存中有且仅有一个拷贝；

实例变量: 必须依存于某一实例，需要先创建对象然后通过对象才能访问到它。静态变量可以实现让多个对象共享内存。

13. ==和 equals 的区别？ （2017-11-22-wzz）

equals 和== 最大的区别是一个是方法一个是运算符。

==：如果比较的对象是基本数据类型，则比较的是数值是否相等；如果比较的是引用数据类型，则比较的是对象的地址值是否相等。

equals()：用来比较方法两个对象的内容是否相等。

注意：equals 方法不能用于基本数据类型的变量，如果没有对 equals 方法进行重写，则比较的是引用类型的变量所指向的对象的地址。

14. break 和 continue 的区别？ （2017-11-23-wzz）

break 和 continue 都是用来控制循环的语句。

break 用于完全结束一个循环，跳出循环体执行循环后面的语句。

continue 用于跳过本次循环，执行下次循环。

15. String s = "Hello";s = s + " world!";这两行代码执行后，原始的 String 对象中的内容到底变了没有？ （2017-12-1-lyq）

没有。因为 String 被设计成不可变(immutable)类，所以它的所有对象都是不可变对象。在这段代码中，s 原先指向一个 String 对象，内容是 "Hello"，然后我们对 s 进行了“+”操作，那么 s 所指向的那个对象是否发生了改变呢？答案是没有。这时，s 不指向原来那个对象了，而指向了另一个 String 对象，内容为"Hello world!"，原来那个对象还

http://www.itheima.com Copyright© 2018 黑马程序员

存在于内存之中，只是 s 这个引用变量不再指向它了。

通过上面的说明，我们很容易导出另一个结论，如果经常对字符串进行各种各样的修改，或者说，不可预见的修改，那么使用 String 来代表字符串的话会引起很大的内存开销。因为 String 对象建立之后不能再改变，所以对于每一个不同的字符串，都需要一个 String 对象来表示。这时，应该考虑使用 StringBuffer 类，它允许修改，而不是每个不同的字符串都要生成一个新的对象。并且，这两种类的对象转换十分容易。同时，我们还可以知道，如果要使用内容相同的字符串，不必每次都 new 一个 String。例如我们要在构造器中对一个名叫 s 的 String 引用变量进行初始化，把它设置为初始值，应当这样做：

```
1. public class Demo {
2.    private String s;
3.    ...
4.    s = "Initial Value";
5.    ...
6. }
```

而非

```
1. s = new String("Initial Value");
```

后者每次都会调用构造器，生成新对象，性能低下且内存开销大，并且没有意义，因为 String 对象不可改变，所以对于内容相同的字符串，只要一个 String 对象来表示就可以了。也就说，多次调用上面的构造器创建多个对象，他们的 String 类型属性 s 都指向同一个对象。

上面的结论还基于这样一个事实：对于字符串常量，如果内容相同，Java 认为它们代表同一个 String 对象。而用关键字 new 调用构造器，总是会创建一个新的对象，无论内容是否相同。 至于为什么要把 String 类设计成不可变类，是它的用途决定的。其实不只 String，很多 Java 标准类库中的类都是不可变的。在开发一个系统的时候，我们有时候也需要设计不可变类，来传递一组相关的值，这也是面向对象思想的体现。不可变类有一些优点，比如因为它的对象是只读的，所以多线程并发访问也不会有任何问题。当然也有一些缺点，比如每个不同的状态都要一个对象来代表，可能会造成性能上的问题。所以 Java 标准类库还提供了一个可变版本，即 StringBuffer。

http://www.itheima.com Copyright© 2018 黑马程序员

三、Java 中的多态

1. Java 中实现多态的机制是什么?

靠的是父类或接口定义的引用变量可以指向子类或具体实现类的实例对象，而程序调用的方法在运行期才动态绑定，就是引用变量所指向的具体实例对象的方法，也就是内存里正在运行的那个对象的方法，而不是引用变量的类型中定义的方法。

四、Java 的异常处理

1. Java 中异常分为哪些种类

1）按照异常需要处理的时机分为编译时异常（也叫强制性异常）也叫 CheckedException 和运行时异常（也叫非强制性异常）也叫 RuntimeException。只有 java 语言提供了 Checked 异常，Java 认为 Checked 异常都是可以被处理的异常，所以 Java 程序必须显式处理 Checked 异常。如果程序没有处理 Checked 异常，该程序在编译时就会发生错误无法编译。这体现了 Java 的设计哲学：没有完善错误处理的代码根本没有机会被执行。对 Checked 异常处理方法有两种：

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，则在定义该方法是声明抛出该异常。

运行时异常只有当代码在运行时才发行的异常，编译时不需要 try catch。Runtime 如除数是 0 和数组下标越界等，其产生频繁，处理麻烦，若显示申明或者捕获将会对程序的可读性和运行效率影响很大。所以由系统自动检测并将它们交给缺省的异常处理程序。当然如果你有处理要求也可以显示捕获它们。

2. 调用下面的方法，得到的返回值是什么?

```
1. public int getNum(){
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
2.      try {
3.          int a = 1/0;
4.          return 1;
5.       } catch (Exception e) {
6.          return 2;
7.       }finally{
8.          return 3;
9.       }
```

代码在走到第 3 行的时候遇到了一个 MathException，这时第四行的代码就不会执行了，代码直接跳转到 catch 语句中，走到第 6 行的时候，异常机制有这么一个原则如果在 catch 中遇到了 return 或者异常等能使该函数终止的话那么有 finally 就必须先执行完 finally 代码块里面的代码然后再返回值。因此代码又跳到第 8 行，可惜第 8 行是一个 return 语句，那么这个时候方法就结束了，因此第 6 行的返回结果就无法被真正返回。如果 finally 仅仅是处理了一个释放资源的操作，那么该道题最终返回的结果就是 2。因此上面返回值是 3。

3. error 和 exception 的区别？（2017-2-23）

Error 类和 Exception 类的父类都是 Throwable 类，他们的区别如下。

Error 类一般是指与虚拟机相关的问题，如系统崩溃，虚拟机错误，内存空间不足，方法调用栈溢出等。对于这类错误的导致的应用程序中断，仅靠程序本身无法恢复和和预防，遇到这样的错误，建议让程序终止。

Exception 类表示程序可以处理的异常，可以捕获且可能恢复。遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。

Exception 类又分为运行时异常（Runtime Exception）和受检查的异常(Checked Exception)，运行时异常;ArithmaticException,IllegalArgumentException，编译能通过，但是一运行就终止了，程序不会处理运行时异常，出现这类异常，程序会终止。而受检查的异常，要么用 try。。。catch 捕获，要么用 throws 字句声明抛出，交给它的父类处理，否则编译不会通过。

http://www.itheima.com Copyright© 2018 黑马程序员

4. java 异常处理机制（2017-2-23）

Java 对异常进行了分类，不同类型的异常分别用不同的 Java 类表示，所有异常的根类为 java.lang.Throwable，Throwable 下面又派生了两个子类：Error 和 Exception，Error 表示应用程序本身无法克服和恢复的一种严重问题。Exception 表示程序还能够克服和恢复的问题，其中又分为系统异常和普通异常，系统异常是软件本身缺陷所导致的问题，也就是软件开发人员考虑不周所导致的问题，软件使用者无法克服和恢复这种问题，但在这种问题下还可以让软件系统继续运行或者让软件死掉，例如，数组脚本越界（ArrayIndexOutOfBoundsException），空指针异常（NullPointerException）、类转换异常（ClassCastException）；普通异常是运行环境的变化或异常所导致的问题，是用户能够克服的问题，例如，网络断线，硬盘空间不够，发生这样的异常后，程序不应该死掉。

java 为系统异常和普通异常提供了不同的解决方案，编译器强制普通异常必须 try..catch 处理或用 throws 声明继续抛给上层调用方法处理，所以普通异常也称为 checked 异常，而系统异常可以处理也可以不处理，所以，编译器不强制用 try..catch 处理或用 throws 声明，所以系统异常也称为 unchecked 异常。

5. 请写出你最常见的 5 个 RuntimeException（2017-11-22-wzz）

下面列举几个常见的 RuntimeException。

1）java.lang.NullPointerException 空指针异常；出现原因：调用了未经初始化的对象或者是不存在的对象。

2）java.lang.ClassNotFoundException 指定的类找不到；出现原因：类的名称和路径加载错误；通常都是程序试图通过字符串来加载某个类时可能引发异常。

3）java.lang.NumberFormatException 字符串转换为数字异常；出现原因：字符型数据中包含非数字型字符。

4）java.lang.IndexOutOfBoundsException 数组角标越界异常，常见于操作数组对象时发生。

5）java.lang.IllegalArgumentException 方法传递参数错误。

6）java.lang.ClassCastException 数据类型转换异常。

http://www.itheima.com Copyright© 2018 黑马程序员

7）java.lang.NoClassDefFoundException 未找到类定义错误。

8）SQLException SQL 异常，常见于操作数据库时的 SQL 语句错误。

9）java.lang.InstantiationException 实例化异常。

10）java.lang.NoSuchMethodException 方法不存在异常。 

6. throw 和 throws 的区别（2017-11-22-wzz）

throw:

1）throw 语句用在方法体内，表示抛出异常，由方法体内的语句处理。

2）throw 是具体向外抛出异常的动作，所以它抛出的是一个异常实例，执行 throw 一定是抛出了某种异常。

throws:

1）throws 语句是用在方法声明后面，表示如果抛出异常，由该方法的调用者来进行异常的处理。

2）throws 主要是声明这个方法会抛出某种类型的异常，让它的使用者要知道需要捕获的异常的类型。

3）throws 表示出现异常的一种可能性，并不一定会发生这种异常。

7. final、finally、finalize 的区别？（2017-11-23-wzz）

1）final：用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖，被其修饰的类不可继承。

2）finally：异常处理语句结构的一部分，表示总是执行。

3）finalize：Object 类的一个方法，在垃圾回收器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等。该方法更像是一个对象生命周期的临终方法，当该方法被系统调用则代表该对象即将“死亡”，但是需要注意的是，我们主动行为上去调用该方法并不会导致该对象“死亡”，这是一个被动的方法（其实就是回调方法），不需要我们调用。

http://www.itheima.com Copyright© 2018 黑马程序员

五、JavaSE 常用 API

1. Math.round(11.5)等于多少？Math.round(- 11.5) 又等于多少?(2017-11-14-wl)

Math.round(11.5)的返回值是 12，Math.round(-11.5)的返回值是-11。四舍五入的原理是在参数上加 0.5 然后进行取整。

2. switch 是否能作用在 byte 上，是否能作用在 long 上，是否能作用在 String 上?(2017-11-14-wl)

Java5 以前 switch(expr)中，expr 只能是 byte、short、char、int。从 Java 5 开始，Java 中引入了枚举类型，expr 也可以是 enum 类型。

从 Java 7 开始，expr 还可以是字符串（String），但是长整型（long）在目前所有的版本中都是不可以的。

3. 数组有没有 length() 方法？String 有没有 length() 方法？(2017-11-14-wl)

数组没有 length()方法，而是有 length 的属性。String 有 length()方法。JavaScript 中，获得字符串的长度是通过 length 属性得到的，这一点容易和 Java 混淆。

4. String 、StringBuilder 、StringBuffer 的区别？(2017-11-14-wl)

Java 平台提供了两种类型的字符串：String 和 StringBuffer/StringBuilder，它们都可以储存和操作字符串，区别如下。

1）String 是只读字符串，也就意味着 String 引用的字符串内容是不能被改变的。初学者可能会有这样的误解：

```
1. String str = “abc”;
2. str = “bcd”;
```


http://www.itheima.com Copyright© 2018 黑马程序员

如上，字符串 str 明明是可以改变的呀！其实不然，str 仅仅是一个引用对象，它指向一个字符串对象“abc”。第二行代码的含义是让 str 重新指向了一个新的字符串“bcd”对象，而“abc”对象并没有任何改变，只不过该对象已经成为一个不可及对象罢了。

2）StringBuffer/StringBuilder 表示的字符串对象可以直接进行修改。

3) StringBuilder 是 Java5 中引入的，它和 StringBuffer 的方法完全相同，区别在于它是在单线程环境下使用的，因为它的所有方法都没有被 synchronized 修饰，因此它的效率理论上也比 StringBuffer 要高。

5. 什么情况下用“+”运算符进行字符串连接比调用 StringBuffer/StringBuilder 对象的 append 方法连接字符串性能更好？(2017-11-14-wl)

该题来自华为。

字符串是 Java 程序中最常用的数据结构之一。在 Java 中 String 类已经重载了"+"。也就是说，字符串可以直接使用"+"进行连接，如下面代码所示：

```
1. String s = "abc" + "ddd";
```

但这样做真的好吗？当然，这个问题不能简单地回答 yes or no。要根据具体情况来定。在 Java 中提供了一个 StringBuilder 类（这个类只在 J2SE5 及以上版本提供，以前的版本使用 StringBuffer 类），这个类也可以起到"+"的作用。那么我们应该用哪个呢？

下面让我们先看看如下的代码：

```
1. package string;
2.
3.   public class TestSimplePlus
4.   {
5.       public static void main(String[] args)
6.       {
7.           String s = "abc";
8.           String ss = "ok" + s + "xyz" + 5;
9.           System.out.println(ss);
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
10.       }
11.   }
```

上面的代码将会输出正确的结果。从表面上看，对字符串和整型使用"+"号并没有什么区别，但事实真的如此吗？下面让我们来看看这段代码的本质。

我们首先使用反编译工具（如 jdk 带的 javap、或 jad）将 TestSimplePlus 反编译成 Java Byte Code，其中的奥秘就一目了然了。在本文将使用 jad 来反编译，命令如下：

jad -o -a -s d.java TestSimplePlus.class

反编译后的代码如下：

```
1. package string;
2.
3. import java.io.PrintStream;
4.
5. public class TestSimplePlus
6. {
7.     public TestSimplePlus()
8.     {
9.     //    0    0:aload_0
10.     //    1    1:invokespecial   #8   <Method void Object()>
11.     //    2    4:return
12.     }
13.
14.     public static void main(String args[])
15.     {
16.       String s = "abc";
17.     //    0    0:ldc1            #16  <String "abc">
18.     //    1    2:astore_1
19.       String ss = (new StringBuilder("ok")).append(s).append("xyz").append(5).toString();
20.     //    2    3:new             #18  <Class StringBuilder>
21.     //    3    6:dup
22.     //    4    7:ldc1            #20  <String "ok">
23.     //    5    9:invokespecial   #22  <Method void StringBuilder(String)>
24.     //    6   12:aload_1
25.     //    7   13:invokevirtual   #25  <Method StringBuilder StringBuilder.append(String)>
26.     //    8   16:ldc1            #29  <String "xyz">
27.     //    9   18:invokevirtual   #25  <Method StringBuilder StringBuilder.append(String)>
28.     //   10   21:iconst_5
29.     //   11   22:invokevirtual   #31  <Method StringBuilder StringBuilder.append(int)>
```


 Copyright© 2018 黑马程序员

```
30.     //   12   25:invokevirtual   #34  <Method String StringBuilder.toString()>
31.     //   13   28:astore_2
32.       System.out.println(ss);
33.     //   14   29:getstatic       #38  <Field PrintStream System.out>
34.     //   15   32:aload_2
35.     //   16   33:invokevirtual   #44  <Method void PrintStream.println(String)>
36.     //   17   36:return
37.     }
38. }
```

读者可能看到上面的 Java 字节码感到迷糊，不过大家不必担心。本文的目的并不是讲解 Java Byte Code，因此，并不用了解具体的字节码的含义。

使用 jad 反编译的好处之一就是可以同时生成字节码和源代码。这样可以进行对照研究。从上面的代码很容易看出，虽然在源程序中使用了"+"，但在编译时仍然将"+"转换成 StringBuilder。因此，我们可以得出结论，在 **Java** 中无论使用何种方式进行字符串连接，实际上都使用的是 **StringBuilder**。

那么是不是可以根据这个结论推出使用"+"和 StringBuilder 的效果是一样的呢？这个要从两个方面的解释。如果从运行结果来解释，那么"+"和 StringBuilder 是完全等效的。但如果从运行效率和资源消耗方面看，那它们将存在很大的区别。

当然，如果连接字符串行表达式很简单（如上面的顺序结构），那么"+"和 StringBuilder 基本是一样的，但如果结构比较复杂，如使用循环来连接字符串，那么产生的 Java Byte Code 就会有很大的区别。先让我们看看如下的代码：

```
1. package string;
2.
3.  import java.util.*;
4.
5.  public class TestComplexPlus
6.  {
7.      public static void main(String[] args)
8.      {
9.          String s = "";
10.          Random rand = new Random();
11.          for (int i = 0; i < 10; i++)
12.          {
13.              s = s + rand.nextInt(1000) + " ";
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
    14.           }
    15.           System.out.println(s);
    16.       }
    17.   }
```

上面的代码返编译后的 Java Byte Code 如下：

```
    1. package string;
    2.
    3. import java.io.PrintStream;
    4. import java.util.Random;
    5.
    6. public class TestComplexPlus
    7. {
    8.
    9.     public TestComplexPlus()
    10.     {
    11.     //    0    0:aload_0
    12.     //    1    1:invokespecial   #8   <Method void Object()>
    13.     //    2    4:return
    14.     }
    15.
    16.     public static void main(String args[])
    17.     {
    18.         String s = "";
    19.     //    0    0:ldc1            #16  <String "">
    20.     //    1    2:astore_1
    21.         Random rand = new Random();
    22.     //    2    3:new             #18  <Class Random>
    23.     //    3    6:dup
    24.     //    4    7:invokespecial   #20  <Method void Random()>
    25.     //    5   10:astore_2
    26.         for(int i = 0; i < 10; i++)
    27.     //*   6   11:iconst_0
    28.     //*   7   12:istore_3
    29.     //*   8   13:goto            49
    30.           s = (new StringBuilder(String.valueOf(s))).append(rand.nextInt(1000)).append(" ").t
oString();
    31.     //    9   16:new             #21  <Class StringBuilder>
    32.     //   10   19:dup
    33.     //   11   20:aload_1
    34.     //   12   21:invokestatic    #23  <Method String String.valueOf(Object)>
    35.     //   13   24:invokespecial   #29  <Method void StringBuilder(String)>
```


 Copyright© 2018 黑马程序员

```
36.     //   14   27:aload_2
37.     //   15   28:sipush          1000
38.     //   16   31:invokevirtual   #32  <Method int Random.nextInt(int)>
39.     //   17   34:invokevirtual   #36  <Method StringBuilder StringBuilder.append(int)>
40.     //   18   37:ldc1            #40  <String " ">
41.     //   19   39:invokevirtual   #42  <Method StringBuilder StringBuilder.append(String)>
42.     //   20   42:invokevirtual   #45  <Method String StringBuilder.toString()>
43.     //   21   45:astore_1
44.
45.     //   22   46:iinc            3  1
46.     //   23   49:iload_3
47.     //   24   50:bipush          10
48.     //   25   52:icmplt          16
49.         System.out.println(s);
50.     //   26   55:getstatic       #49  <Field PrintStream System.out>
51.     //   27   58:aload_1
52.     //   28   59:invokevirtual   #55  <Method void PrintStream.println(String)>
53.     //   29   62:return
54.     }
55. }
```

大家可以看到，虽然编译器将"+"转换成了 StringBuilder，但创建 StringBuilder 对象的位置却在 for 语句内部。这就意味着每执行一次循环，就会创建一个 StringBuilder 对象（对于本例来说，是创建了 10 个 StringBuilder 对象），虽然 Java 有垃圾回收器，但这个回收器的工作时间是不定的。如果不断产生这样的垃圾，那么仍然会占用大量的资源。解决这个问题的方法就是在程序中直接使用 StringBuilder 来连接字符串，代码如下：

```
1. package string;
2.
3. import java.util.*;
4.
5. public class TestStringBuilder
6. {
7.     public static void main(String[] args)
8.     {
9.         String s = "";
10.         Random rand = new Random();
11.         StringBuilder result = new StringBuilder();
12.         for (int i = 0; i < 10; i++)
13.         {
14.             result.append(rand.nextInt(1000));
```


 Copyright© 2018 黑马程序员

```
15.             result.append(" ");
16.         }
17.         System.out.println(result.toString());
18.     }
19. }
```

上面代码反编译后的结果如下：

```
1. 20.package string;
2.
3. import java.io.PrintStream;
4. import java.util.Random;
5.
6. public class TestStringBuilder
7. {
8.
9.     public TestStringBuilder()
10.     {
11.     //    0    0:aload_0
12.     //    1    1:invokespecial   #8   <Method void Object()>
13.     //    2    4:return
14.     }
15.
16.     public static void main(String args[])
17.     {
18.         String s = "";
19.     //    0    0:ldc1            #16  <String "">
20.     //    1    2:astore_1
21.         Random rand = new Random();
22.     //    2    3:new             #18  <Class Random>
23.     //    3    6:dup
24.     //    4    7:invokespecial   #20  <Method void Random()>
25.     //    5   10:astore_2
26.         StringBuilder result = new StringBuilder();
27.     //    6   11:new             #21  <Class StringBuilder>
28.     //    7   14:dup
29.     //    8   15:invokespecial   #23  <Method void StringBuilder()>
30.     //    9   18:astore_3
31.         for(int i = 0; i < 10; i++)
32.     //*  10   19:iconst_0
33.     //*  11   20:istore          4
34.     //*  12   22:goto            47
35.         {
36.             result.append(rand.nextInt(1000));
```


 Copyright© 2018 黑马程序员

```
    37.     //   13   25:aload_3
    38.     //   14   26:aload_2
    39.     //   15   27:sipush          1000
    40.     //   16   30:invokevirtual   #24  <Method int Random.nextInt(int)>
    41.     //   17   33:invokevirtual   #28  <Method StringBuilder StringBuilder.append(int)>
    42.     //   18   36:pop
    43.             result.append(" ");
    44.     //   19   37:aload_3
    45.     //   20   38:ldc1            #32  <String " ">
    46.     //   21   40:invokevirtual   #34  <Method StringBuilder StringBuilder.append(String)>
    47.     //   22   43:pop
    48.         }
    49.
    50.     //   23   44:iinc            4  1
    51.     //   24   47:iload           4
    52.     //   25   49:bipush          10
    53.     //   26   51:icmplt          25
    54.         System.out.println(result.toString());
    55.     //   27   54:getstatic       #37  <Field PrintStream System.out>
    56.     //   28   57:aload_3
    57.     //   29   58:invokevirtual   #43  <Method String StringBuilder.toString()>
    58.     //   30   61:invokevirtual   #47  <Method void PrintStream.println(String)>
    59.     //   31   64:return
    60.     }
    61. }
```

从上面的反编译结果可以看出，创建 StringBuilder 的代码被放在了 for 语句外。虽然这样处理在源程序中看起来复杂，但却换来了更高的效率，同时消耗的资源也更少了。

在使用 StringBuilder 时要注意，尽量不要"+"和 StringBuilder 混着用，否则会创建更多的 StringBuilder 对象，如下面代码所：

```
for (int i = 0; i < 10; i++)
    {
        result.append(rand.nextInt(1000));
        result.append(" ");
    }
```

改成如下形式：

```
for (int i = 0; i < 10; i++)
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
{
     result.append(rand.nextInt(1000) + " ");
}
```

则反编译后的结果如下：

```
for(int i = 0; i < 10; i++)
  //*  10    19:iconst_0
  //*  11    20:istore          4
  //*  12    22:goto            65
   {
    result.append((new StringBuilder(String.valueOf(rand.nextInt(1000)))).append(" ").toString());
  //   13    25:aload_3
  //   14    26:new             #21  <Class StringBuilder>
  //   15    29:dup
```

从上面的代码可以看出，Java 编译器将"+"编译成了 StringBuilder，这样 for 语句每循环一次，又创建了一个 StringBuilder 对象。

如果将上面的代码在 JDK1.4 下编译，必须将 StringBuilder 改为 StringBuffer，而 JDK1.4 将"+"转换为 StringBuffer（因为 JDK1.4 并没有提供 StringBuilder 类）。StringBuffer 和 StringBuilder 的功能基本一样，只是 StringBuffer 是线程安全的，而 StringBuilder 不是线程安全的。因此，StringBuilder 的效率会更高。

6. 请说出下面程序的输出(2017-11-14-wl)

```
1. class StringEqualTest {
2.   public static void main(String[] args) {
3.       String s1 = "Programming";
4.       String s2 = new String("Programming");
5.       String s3 = "Program";
6.       String s4 = "ming";
7.       String s5 = "Program" + "ming";
8.       String s6 = s3 + s4;
9.       System.out.println(s1 == s2);          //false
10.      System.out.println(s1 == s5);          //true
11.      System.out.println(s1 == s6);          //false
12.      System.out.println(s1 == s6.intern());  //true
13.     System.out.println(s2 == s2.intern());   //false
14. }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
15. }
```

补充：解答上面的面试题需要知道如下两个知识点：

1. String 对象的 intern（）方法会得到字符串对象在常量池中对应的版本的引用（如果常量池中有一个字符串与 String 对象的 equals 结果是 true），如果常量池中没有对应的字符串，则该字符串将被添加到常量池中，然后返回常量池中字符串的引用；

2. 字符串的+操作其本质是创建了 StringBuilder 对象进行 append 操作，然后将拼接后的 StringBuilder 对象用 toString 方法处理成 String 对象，这一点可以用 javap -c StringEqualTest.class 命令获得 class 文件对应的 JVM 字节码指令就可以看出来。

7. Java 中的日期和时间(2017-11-19-wl)

7.1 如何取得年月日、小时分钟秒？(2017-11-19-wl)

```
1. public class DateTimeTest {
2.  public static void main(String[] args) {
3.    Calendar cal = Calendar.getInstance();
4.    System.out.println(cal.get(Calendar.YEAR));
5.    System.out.println(cal.get(Calendar.MONTH)); // 0 - 11
6.    System.out.println(cal.get(Calendar.DATE));
7.    System.out.println(cal.get(Calendar.HOUR_OF_DAY));
8.    System.out.println(cal.get(Calendar.MINUTE));
9.    System.out.println(cal.get(Calendar.SECOND));
10.    // Java 8
11.    LocalDateTime dt = LocalDateTime.now();
12.    System.out.println(dt.getYear());
13.    System.out.println(dt.getMonthValue()); // 1 - 12
14.    System.out.println(dt.getDayOfMonth());
15.    System.out.println(dt.getHour());
16.    System.out.println(dt.getMinute());
17.    System.out.println(dt.getSecond());
18.  }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
19. }
```

7.2 如何取得从 1970 年 1 月 1 日 0 时 0 分 0 秒到现在的毫秒数？(2017-11-19-wl)

```
1. Calendar.getInstance().getTimeInMillis();  //第一种方式
2. System.currentTimeMillis(); //第二种方式
3. // Java 8
4. Clock.systemDefaultZone().millis();
```

7.3 如何取得某月的最后一天？(2017-11-19-wl)

```
1. //获取当前月第一天：
2. Calendar c = Calendar.getInstance();
3. c.add(Calendar.MONTH, 0);
4. c.set(Calendar.DAY_OF_MONTH,1);//设置为 1 号,当前日期既为本月第一天
5. String first = format.format(c.getTime());
6. System.out.println("===============first:"+first);
7.
8.   //获取当前月最后一天
9.  Calendar ca = Calendar.getInstance();
10. ca.set(Calendar.DAY_OF_MONTH, ca.getActualMaximum(Calendar.DAY_OF_MONTH));
11. String last = format.format(ca.getTime());
12. System.out.println("===============last:"+last);
13.
14. //Java 8
15. LocalDate today = LocalDate.now();
16. //本月的第一天
17. LocalDate firstday = LocalDate.of(today.getYear(),today.getMonth(),1);
18. //本月的最后一天
19. LocalDate lastDay =today.with(TemporalAdjusters.lastDayOfMonth());
20. System.out.println("本月的第一天"+firstday);
21. System.out.println("本月的最后一天"+lastDay);
```


http://www.itheima.com Copyright© 2018 黑马程序员

7.4 如何格式化日期？(2017-11-19-wl)

1）Java.text.DataFormat 的子类（如 SimpleDateFormat 类）中的 format(Date)方法可将日期格式化。

2）Java 8 中可以用 java.time.format.DateTimeFormatter 来格式化时间日期，代码如下所示：

```
1. import java.text.SimpleDateFormat;
2. import java.time.LocalDate;
3. import java.time.format.DateTimeFormatter;
4. import java.util.Date;
5. class DateFormatTest {
6.
7.    public static void main(String[] args) {
8.       SimpleDateFormat oldFormatter = new SimpleDateFormat("yyyy/MM/dd");
9.       Date date1 = new Date();
10.       System.out.println(oldFormatter.format(date1));
11.
12.      // Java 8
13.       DateTimeFormatter newFormatter = DateTimeFormatter.ofPattern("yyyy/MM/dd");
14.       LocalDate date2 = LocalDate.now();
15.       System.out.println(date2.format(newFormatter));
16.    }
17. }
```

补充: Java 的时间日期 API 一直以来都是被诟病的东西，为了解决这一问题，Java 8 中引入了新的时间日期 API，其中包括 LocalDate、LocalTime、LocalDateTime、Clock、Instant 等类，这些的类的设计都使用了不变模式，因此是线程安全的设计。

7.5 打印昨天的当前时刻？ (2017-11-19-wl)

```
1. import java.util.Calendar;
2. class YesterdayCurrent {
3.   public static void main(String[] args){
4.    Calendar cal = Calendar.getInstance();
5.    cal.add(Calendar.DATE, -1);
6.    System.out.println(cal.getTime());
7.   }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
8. }
9.
10.
11. //java-8
12. import java.time.LocalDateTime;
13. class YesterdayCurrent {
14.    public static void main(String[] args) {
15.       LocalDateTime today = LocalDateTime.now();
16.       LocalDateTime yesterday = today.minusDays(1);
17.       System.out.println(yesterday);
18.    }
19. }
```

7.6 Java8 的日期特性？ (2017-12-3-wl)

Java 8 日期/时间特性

Java 8 日期/时间 API 是 JSR-310 的实现，它的实现目标是克服旧的日期时间实现中所有的缺陷，新的日期/时间 API 的一些设计原则是：

- 不变性：新的日期/时间 API 中，所有的类都是不可变的，这对多线程环境有好处。
- 关注点分离：新的 API 将人可读的日期时间和机器时间（unix timestamp）明确分离，它为日期（Date）、时间（Time）、日期时间（DateTime）、时间戳（unix timestamp）以及时区定义了不同的类。
- 清晰：在所有的类中，方法都被明确定义用以完成相同的行为。举个例子，要拿到当前实例我们可以使用 now()方法，在所有的类中都定义了 format()和 parse()方法，而不是像以前那样专门有一个独立的类。为了更好的处理问题，所有的类都使用了工厂模式和策略模式，一旦你使用了其中某个类的方法，与其他类协同工作并不困难。
- 实用操作：所有新的日期/时间 API 类都实现了一系列方法用以完成通用的任务，如：加、减、格式化、解析、从日期/时间中提取单独部分，等等。
- 可扩展性：新的日期/时间 API 是工作在 ISO-8601 日历系统上的，但我们也可以将其应用在非 ISO 的日历上。

http://www.itheima.com Copyright© 2018 黑马程序员

Java 8 日期/时间 API 包解释

- java.time 包：这是新的 Java 日期/时间 API 的基础包，所有的主要基础类都是这个包的一部分，如：LocalDate, LocalTime, LocalDateTime, Instant, Period, Duration 等等。所有这些类都是不可变的和线程安全的，在绝大多数情况下，这些类能够有效地处理一些公共的需求。
- java.time.chrono 包：这个包为非 ISO 的日历系统定义了一些泛化的 API，我们可以扩展 AbstractChronology 类来创建自己的日历系统。
- java.time.format 包：这个包包含能够格式化和解析日期时间对象的类，在绝大多数情况下，我们不应该直接使用它们，因为 java.time 包中相应的类已经提供了格式化和解析的方法。
- java.time.temporal 包：这个包包含一些时态对象，我们可以用其找出关于日期/时间对象的某个特定日期或时间，比如说，可以找到某月的第一天或最后一天。你可以非常容易地认出这些方法，因为它们都具有“withXXX”的格式。
- java.time.zone 包：这个包包含支持不同时区以及相关规则的类。

Java 8 日期/时间常用 API

1.java.time.LocalDate

LocalDate 是一个不可变的类，它表示默认格式(yyyy-MM-dd)的日期，我们可以使用 now()方法得到当前时间，也可以提供输入年份、月份和日期的输入参数来创建一个 LocalDate 实例。该类为 now()方法提供了重载方法，我们可以传入 ZoneId 来获得指定时区的日期。该类提供与 java.sql.Date 相同的功能，对于如何使用该类，我们来看一个简单的例子。

```
package com.journaldev.java8.time;

import java.time.LocalDate;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
import java.time.Month;
import java.time.ZoneId;

/**
 * LocalDate Examples
 * @author pankaj
 *
 */
public class LocalDateExample {

    public static void main(String[] args) {

        //Current Date
        LocalDate today = LocalDate.now();
        System.out.println("Current Date="+today);

        //Creating LocalDate by providing input arguments
        LocalDate firstDay_2014 = LocalDate.of(2014, Month.JANUARY, 1);
        System.out.println("Specific Date="+firstDay_2014);

        //Try creating date by providing invalid inputs
        //LocalDate feb29_2014 = LocalDate.of(2014, Month.FEBRUARY, 29);
        //Exception in thread "main" java.time.DateTimeException:
        //Invalid date 'February 29' as '2014' is not a leap year

        //Current date in "Asia/Kolkata", you can get it from ZoneId javadoc
        LocalDate todayKolkata = LocalDate.now(ZoneId.of("Asia/Kolkata"));
        System.out.println("Current Date in IST="+todayKolkata);

        //java.time.zone.ZoneRulesException: Unknown time-zone ID: IST
        //LocalDate todayIST = LocalDate.now(ZoneId.of("IST"));

        //Getting date from the base date i.e 01/01/1970
        LocalDate dateFromBase = LocalDate.ofEpochDay(365);
        System.out.println("365th day from base date= "+dateFromBase);

        LocalDate hundredDay2014 = LocalDate.ofYearDay(2014, 100);
        System.out.println("100th day of 2014="+hundredDay2014);
    }
}
```

输出：

```
Current Date=2014-04-28
```


 Copyright© 2018 黑马程序员

```
Specific Date=2014-01-01
Current Date in IST=2014-04-29
365th day from base date= 1971-01-01
100th day of 2014=2014-04-10
```

2.java.time.LocalTime

LocalTime 是一个不可变的类，它的实例代表一个符合人类可读格式的时间，默认格式是 hh:mm:ss.zzz。像 LocalDate 一样，该类也提供了时区支持，同时也可以传入小时、分钟和秒等输入参数创建实例，我们来看一个简单的程序，演示该类的使用方法。

```
package com.journaldev.java8.time;

import java.time.LocalTime;
import java.time.ZoneId;

/**
 * LocalTime Examples
 */
public class LocalTimeExample {

    public static void main(String[] args) {

        //Current Time
        LocalTime time = LocalTime.now();
        System.out.println("Current Time="+time);

        //Creating LocalTime by providing input arguments
        LocalTime specificTime = LocalTime.of(12,20,25,40);
        System.out.println("Specific Time of Day="+specificTime);

        //Try creating time by providing invalid inputs
        //LocalTime invalidTime = LocalTime.of(25,20);
        //Exception in thread "main" java.time.DateTimeException:
        //Invalid value for HourOfDay (valid values 0 - 23): 25

        //Current date in "Asia/Kolkata", you can get it from ZoneId javadoc
        LocalTime timeKolkata = LocalTime.now(ZoneId.of("Asia/Kolkata"));
        System.out.println("Current Time in IST="+timeKolkata);

        //java.time.zone.ZoneRulesException: Unknown time-zone ID: IST
        //LocalTime todayIST = LocalTime.now(ZoneId.of("IST"));

        //Getting date from the base date i.e 01/01/1970
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        LocalTime specificSecondTime = LocalTime.ofSecondOfDay(10000);
        System.out.println("10000th second time= "+specificSecondTime);

    }
}
输出：
Current Time=15:51:45.240
Specific Time of Day=12:20:25.000000040
Current Time in IST=04:21:45.276
10000th second time= 02:46:40
```

3. java.time.LocalDateTime

LocalDateTime 是一个不可变的日期-时间对象，它表示一组日期-时间，默认格式是 yyyy-MM-dd-HH-mm-ss.zzz。它提供了一个工厂方法，接收 LocalDate 和 LocalTime 输入参数，创建 LocalDateTime 实例。我们来看一个简单的例子。

```
package com.journaldev.java8.time;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZoneOffset;

public class LocalDateTimeExample {

    public static void main(String[] args) {

        //Current Date
        LocalDateTime today = LocalDateTime.now();
        System.out.println("Current DateTime="+today);

        //Current Date using LocalDate and LocalTime
        today = LocalDateTime.of(LocalDate.now(), LocalTime.now());
        System.out.println("Current DateTime="+today);

        //Creating LocalDateTime by providing input arguments
        LocalDateTime specificDate = LocalDateTime.of(2014, Month.JANUARY, 1, 10, 10, 30);
```


 Copyright© 2018 黑马程序员

```
        System.out.println("Specific Date="+specificDate);

        //Try creating date by providing invalid inputs
        //LocalDateTime feb29_2014 = LocalDateTime.of(2014, Month.FEBRUARY, 28, 25,1,1);
        //Exception in thread "main" java.time.DateTimeException:
        //Invalid value for HourOfDay (valid values 0 - 23): 25

        //Current date in "Asia/Kolkata", you can get it from ZoneId javadoc
        LocalDateTime todayKolkata = LocalDateTime.now(ZoneId.of("Asia/Kolkata"));
        System.out.println("Current Date in IST="+todayKolkata);

        //java.time.zone.ZoneRulesException: Unknown time-zone ID: IST
        //LocalDateTime todayIST = LocalDateTime.now(ZoneId.of("IST"));

        //Getting date from the base date i.e 01/01/1970
        LocalDateTime dateFromBase = LocalDateTime.ofEpochSecond(10000, 0, ZoneOffset.UTC);
        System.out.println("10000th second time from 01/01/1970= "+dateFromBase);
    }
}
输出：
Current DateTime=2014-04-28T16:00:49.455
Current DateTime=2014-04-28T16:00:49.493
Specific Date=2014-01-01T10:10:30
Current Date in IST=2014-04-29T04:30:49.493
10000th second time from 01/01/1970= 1970-01-01T02:46:40
```

在所有这三个例子中，我们已经看到如果我们提供了无效的参数去创建日期/时间，那么系统会抛出java.time.DateTimeException，这是一种运行时异常，我们并不需要显式地捕获它。

同时我们也看到，能够通过传入 ZoneId 得到日期/时间数据，你可以从它的 Javadoc 中得到支持的 Zoneid 的列表，当运行以上类时，可以得到以上输出。

4. java.time.Instant

Instant 类是用在机器可读的时间格式上的，它以 Unix 时间戳的形式存储日期时间，我们来看一个简单的程序

```
package com.journaldev.java8.time;

import java.time.Duration;
import java.time.Instant;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
public class InstantExample {

    public static void main(String[] args) {
        //Current timestamp
        Instant timestamp = Instant.now();
        System.out.println("Current Timestamp = "+timestamp);

        //Instant from timestamp
        Instant specificTime = Instant.ofEpochMilli(timestamp.toEpochMilli());
        System.out.println("Specific Time = "+specificTime);

        //Duration example
        Duration thirtyDay = Duration.ofDays(30);
        System.out.println(thirtyDay);
    }
}
输出:
Current Timestamp = 2014-04-28T23:20:08.489Z
Specific Time = 2014-04-28T23:20:08.489Z
PT720H
```

5. 日期 API 工具

我们早些时候提到过，大多数日期/时间 API 类都实现了一系列工具方法，如：加/减天数、周数、月份数，等等。还有其他的工具方法能够使用 TemporalAdjuster 调整日期，并计算两个日期间的周期。

```
package com.journaldev.java8.time;

import java.time.LocalDate;
import java.time.LocalTime;
import java.time.Period;
import java.time.temporal.TemporalAdjusters;

public class DateAPIUtilities {

    public static void main(String[] args) {

        LocalDate today = LocalDate.now();

        //Get the Year, check if it's leap year
        System.out.println("Year "+today.getYear()+" is Leap Year? "+today.isLeapYear());
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        //Compare two LocalDate for before and after
        System.out.println("Today is before 01/01/2015? "+today.isBefore(LocalDate.of(2015,1,1)));

        //Create LocalDateTime from LocalDate
        System.out.println("Current Time="+today.atTime(LocalTime.now()));

        //plus and minus operations
        System.out.println("10 days after today will be "+today.plusDays(10));
        System.out.println("3 weeks after today will be "+today.plusWeeks(3));
        System.out.println("20 months after today will be "+today.plusMonths(20));

        System.out.println("10 days before today will be "+today.minusDays(10));
        System.out.println("3 weeks before today will be "+today.minusWeeks(3));
        System.out.println("20 months before today will be "+today.minusMonths(20));

        //Temporal adjusters for adjusting the dates
         System.out.println("First date of this month= "+today.
         with(TemporalAdjusters.firstDayOfMonth()));
        LocalDate lastDayOfYear = today.with(TemporalAdjusters.lastDayOfYear());
        System.out.println("Last date of this year= "+lastDayOfYear);

        Period period = today.until(lastDayOfYear);
        System.out.println("Period Format= "+period);
        System.out.println("Months remaining in the year= "+period.getMonths());
    }
}
```

输出：

```
Year 2014 is Leap Year? false
Today is before 01/01/2015? true
Current Time=2014-04-28T16:23:53.154
10 days after today will be 2014-05-08
3 weeks after today will be 2014-05-19
20 months after today will be 2015-12-28
10 days before today will be 2014-04-18
3 weeks before today will be 2014-04-07
20 months before today will be 2012-08-28
First date of this month= 2014-04-01
Last date of this year= 2014-12-31
Period Format= P8M3D
Months remaining in the year= 8
```


http://www.itheima.com Copyright© 2018 黑马程序员

6. 解析和格式化

将一个日期格式转换为不同的格式，之后再解析一个字符串，得到日期时间对象，这些都是很常见的。我们来看一下简单的例子。

```
package com.journaldev.java8.time;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class DateParseFormatExample {

    public static void main(String[] args) {

        //Format examples
        LocalDate date = LocalDate.now();
        //default format
        System.out.println("Default format of LocalDate="+date);
        //specific format
        System.out.println(date.format(DateTimeFormatter.ofPattern("d::MMM::uuuu")));
        System.out.println(date.format(DateTimeFormatter.BASIC_ISO_DATE));

        LocalDateTime dateTime = LocalDateTime.now();
        //default format
        System.out.println("Default format of LocalDateTime="+dateTime);
        //specific format
    System.out.println(dateTime.format(DateTimeFormatter.ofPattern("d::MMM::uuuu HH::mm::ss")));
        System.out.println(dateTime.format(DateTimeFormatter.BASIC_ISO_DATE));

        Instant timestamp = Instant.now();
        //default format
        System.out.println("Default format of Instant="+timestamp);

        //Parse examples
        LocalDateTime dt = LocalDateTime.parse("27::Apr::2014 21::39::48",
                DateTimeFormatter.ofPattern("d::MMM::uuuu HH::mm::ss"));
        System.out.println("Default format after parsing = "+dt);
    }
}
输出：
Default format of LocalDate=2014-04-28
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
28::Apr::2014
20140428
Default format of LocalDateTime=2014-04-28T16:25:49.341
28::Apr::2014 16::25::49
20140428
Default format of Instant=2014-04-28T23:25:49.342Z
Default format after parsing = 2014-04-27T21:39:48
```

7. 旧的日期时间支持

旧的日期/时间类已经在几乎所有的应用程序中使用，因此做到向下兼容是必须的。这也是为什么会有若干工具方法帮助我们将旧的类转换为新的类，反之亦然。我们来看一下简单的例子。

```
package com.journaldev.java8.time;

import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.TimeZone;

public class DateAPILegacySupport {

    public static void main(String[] args) {

        //Date to Instant
        Instant timestamp = new Date().toInstant();
        //Now we can convert Instant to LocalDateTime or other similar classes
        LocalDateTime date = LocalDateTime.ofInstant(timestamp,
                        ZoneId.of(ZoneId.SHORT_IDS.get("PST")));
        System.out.println("Date = "+date);

        //Calendar to Instant
        Instant time = Calendar.getInstance().toInstant();
        System.out.println(time);
        //TimeZone to ZoneId
        ZoneId defaultZone = TimeZone.getDefault().toZoneId();
```


 Copyright© 2018 黑马程序员

```
            System.out.println(defaultZone);

            //ZonedDateTime from specific Calendar
            ZonedDateTime gregorianCalendarDateTime = new GregorianCalendar().toZonedDateTime();
            System.out.println(gregorianCalendarDateTime);

            //Date API to Legacy classes
            Date dt = Date.from(Instant.now());
            System.out.println(dt);

            TimeZone tz = TimeZone.getTimeZone(defaultZone);
            System.out.println(tz);

            GregorianCalendar gc = GregorianCalendar.from(gregorianCalendarDateTime);
            System.out.println(gc);

        }
    }
    输出：
    Date = 2014-04-28T16:28:54.340
    2014-04-28T23:28:54.395Z
    America/Los_Angeles
    2014-04-28T16:28:54.404-07:00[America/Los_Angeles]
    Mon Apr 28 16:28:54 PDT 2014
    sun.util.calendar.ZoneInfo[id="America/Los_Angeles",offset=-
28800000,dstSavings=3600000,useDaylight=true,transitions=185,lastRule=java.util.SimpleTimeZone[id=A
merica/Los_Angeles,offset=-
28800000,dstSavings=3600000,useDaylight=true,startYear=0,startMode=3,startMonth=2,startDay=8,startD
ayOfWeek=1,startTime=7200000,startTimeMode=0,endMode=3,endMonth=10,endDay=1,endDayOfWeek=1,endTime=
7200000,endTimeMode=0]]
    java.util.GregorianCalendar[time=1398727734404,areFieldsSet=true,areAllFieldsSet=true,lenient=t
rue,zone=sun.util.calendar.ZoneInfo[id="America/Los_Angeles",offset=-
28800000,dstSavings=3600000,useDaylight=true,transitions=185,lastRule=java.util.SimpleTimeZone[id=A
merica/Los_Angeles,offset=-
28800000,dstSavings=3600000,useDaylight=true,startYear=0,startMode=3,startMonth=2,startDay=8,startD
ayOfWeek=1,startTime=7200000,startTimeMode=0,endMode=3,endMonth=10,endDay=1,endDayOfWeek=1,endTime=
7200000,endTimeMode=0]],firstDayOfWeek=2,minimalDaysInFirstWeek=4,ERA=1,YEAR=2014,MONTH=3,WEEK_OF_Y
EAR=18,WEEK_OF_MONTH=5,DAY_OF_MONTH=28,DAY_OF_YEAR=118,DAY_OF_WEEK=2,DAY_OF_WEEK_IN_MONTH=4,AM_PM=1
,HOUR=4,HOUR_OF_DAY=16,MINUTE=28,SECOND=54,MILLISECOND=404,ZONE_OFFSET=-28800000,DST_OFFSET=3600000]
```

补充：我们可以看到，旧的 TimeZone 和 GregorianCalendar 类的 toString()方法太啰嗦了，一点都不友好。

http://www.itheima.com Copyright© 2018 黑马程序员

7.7 Java8 之前的日期和时间使用的槽点（2017-11-19-wl）

Tiago Fernandez 做过一次投票，选举最烂的 JAVA API，排第一的 EJB2.X，第二的就是日期 API（**Date** 和 **Calender**）

1. 槽点一

最开始的时候，Date 既要承载日期信息，又要做日期之间的转换，还要做不同日期格式的显示，职责较繁杂（不懂单一职责，你妈妈知道吗？纯属恶搞~哈哈）

后来从 JDK 1.1 开始，这三项职责分开了：

1）使用 Calendar 类实现日期和时间字段之间转换；

2）使用 DateFormat 类来格式化和分析日期字符串；

3）而 Date 只用来承载日期和时间信息。

原有 Date 中的相应方法已废弃。不过，无论是 Date，还是 Calendar，都用着太不方便了，这是 API 没有设计好的地方。

2. 槽点二

坑爹的 **year** 和 **month**。

我们看下面的代码：

```
1. Date date = new Date(2012,1,1);
2. System.out.println(date);
```

输出 Thu Feb 01 00:00:00 CST 3912

观察输出结果，year 是 2012+1900，而 month，月份参数我不是给了 1 吗？怎么输出二月（Feb）了？

应该曾有人告诉你，如果你要设置日期，应该使用 java.util.Calendar，像这样...

```
1. Calendar calendar = Calendar.getInstance();
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
2. calendar.set(2013, 8, 2);
```

这样写又不对了，calendar 的 month 也是从 0 开始的，表达 8 月份应该用 7 这个数字，要么就干脆用枚举

```
1. calendar.set(2013, Calendar.AUGUST, 2);
```

注意上面的代码，Calendar 年份的传值不需要减去 1900（当然月份的定义和 Date 还是一样），这种不一致真是让人抓狂！有些人可能知道，**Calendar 相关的 API 是 IBM 捐出去的**，所以才导致不一致。

3. 槽点三

java.util.Date 与 java.util.Calendar 中的所有属性都是可变的

下面的代码，计算两个日期之间的天数....

```
1. public static void main(String[] args) {
2.    Calendar birth = Calendar.getInstance();
3.    birth.set(1975, Calendar.MAY, 26);
4.    Calendar now = Calendar.getInstance();
5.    System.out.println(daysBetween(birth, now));
6.    System.out.println(daysBetween(birth, now)); // 显示 0?
7. }
8.
9. public static long daysBetween(Calendar begin, Calendar end) {
10.    long daysBetween = 0;
11.   while(begin.before(end)) {
12.      begin.add(Calendar.DAY_OF_MONTH, 1);
13.      daysBetween++;
14.   }
15.   return daysBetween;
16. }
```

daysBetween 有点问题，如果连续计算两个 Date 实例的话，第二次会取得 0，因为 Calendar 状态是可变的，考虑到重复计算的场合，最好复制一个新的 Calendar

```
1. public static long daysBetween(Calendar begin, Calendar end) {
2.    Calendar calendar = (Calendar) begin.clone(); // 复制
3.    long daysBetween = 0;
4.    while(calendar.before(end)) {
5.       calendar.add(Calendar.DAY_OF_MONTH, 1);
6.       daysBetween++;
7.    }
8.    return daysBetween;
```


 Copyright© 2018 黑马程序员

```
9. }
```

以上种种，导致目前有些第三方的 java 日期库诞生，比如广泛使用的 JODA-TIME，还有 Date4j 等，虽然第三方库已经足 3 / 8 够强大，好用，但还是有兼容问题的，比如标准的 JSF 日期转换器与 joda-time API 就不兼容，你需要编写自己的转换器，所以标准的 API 还是必须的，**于是就有了 JSR310。**

7.8 Java8 日期实现 JSR310 规范（2017-11-23-wl)

1. JSR310 介绍

JSR 310 实际上有两个日期概念。第一个是 Instant，它大致对应于 java.util.Date 类，因为它代表了一个确定的时间点，即相对于标准 Java 纪元（1970 年 1 月 1 日）的偏移量；但与 java.util.Date 类不同的是其精确到了纳秒级别。

第二个对应于人类自身的观念，比如 LocalDate 和 LocalTime。他们代表了一般的时区概念，要么是日期（不包含时间），要么是时间（不包含日期），类似于 java.sql 的表示方式。此外，还有一个 MonthDay，它可以存储某人的生日（不包含年份）。每个类都在内部存储正确的数据而不是像 java.util.Date 那样利用午夜 12 点来区分日期，利用 1970-01-01 来表示时间。

目前 Java8 已经实现了 JSR310 的全部内容。新增了 java.time 包定义的类表示了日期-时间概念的规则，包括 instants,durations, dates, times, time-zones and periods。这些都是基于 ISO 日历系统，它又是遵循 Gregorian 规则的。最重要的一点是值不可变，且线程安全，通过下面一张图，我们快速看下 java.time 包下的一些主要的类的值的格式，方便理解。

http://www.itheima.com Copyright© 2018 黑马程序员

```
• LocalDate        2010-12-03
• LocalTime                11:05:30
• LocalDateTime    2010-12-03T11:05:30
• OffsetTime               11:05:30+01:00
• OffsetDateTime   2010-12-03T11:05:30+01:00
• ZonedDateTime    2010-12-03T11:05:30+01:00 Europe/Paris
• Year             2010
• YearMonth        2010-12
• MonthDay             -12-03
• Instant          2576458258.266 seconds after 1970-01-01
```

2. Java8 方法概览

java.time 包下的方法概览

方法名	说明
Of	静态工厂方法
parse	静态工厂方法，关注于解析
get	获取某些东西的值
is	检查某些东西的是否是 true
with	不可变的 setter 等价物
plus	加一些量到某个对象
minus	从某个对象减去一些量
to	转换到另一个类型
at	把这个对象与另一个对象组合起来

http://www.itheima.com Copyright© 2018 黑马程序员

与旧的 API 相比

Java.time ISO Calendar	Java.util Calendar
Instant	Date
LocalDate, LocalTime, LocalDateTime	Calendar
ZonedDateTime	Calendar
OffsetDateTime, OffsetTime,	Calendar
ZoneId, ZoneOffset, ZoneRules	TimeZone
Week Starts on Monday (1 .. 7) enum MONDAY, TUESDAY, ... SUNDAY	Week Starts on Sunday (1 .. 7) int values SUNDAY, MONDAY, ... SATURDAY
12 Months (1 .. 12) enum JANUARY, FEBRUARY, ..., DECEMBER	12 Months (0 .. 11) int values JANUARY, FEBRUARY, ... DECEMBER

3. 简单实用 java.time 的 API 实用

```
1. public class TimeIntroduction {
2.   public static void testClock() throws InterruptedException {
3.     //时钟提供给我们用于访问某个特定 时区的 瞬时时间、日期 和 时间的。
4.     Clock c1 = Clock.systemUTC(); //系统默认 UTC 时钟（当前瞬时时间 System.currentTimeMillis()）
5.     System.out.println(c1.millis()); //每次调用将返回当前瞬时时间（UTC）
6.     Clock c2 = Clock.systemDefaultZone(); //系统默认时区时钟（当前瞬时时间）
7.     Clock c31 = Clock.system(ZoneId.of("Europe/Paris")); //巴黎时区
8.     System.out.println(c31.millis()); //每次调用将返回当前瞬时时间（UTC）
9.     Clock c32 = Clock.system(ZoneId.of("Asia/Shanghai"));//上海时区
10.    System.out.println(c32.millis());//每次调用将返回当前瞬时时间（UTC）
11.    Clock c4 = Clock.fixed(Instant.now(), ZoneId.of("Asia/Shanghai"));//固定上海时区时钟
12.    System.out.println(c4.millis());
13.    Thread.sleep(1000);
14.
```


 Copyright© 2018 黑马程序员

```
15.    System.out.println(c4.millis()); //不变 即时钟时钟在那一个点不动
16.    Clock c5 = Clock.offset(c1, Duration.ofSeconds(2)); //相对于系统默认时钟两秒的时钟
17.    System.out.println(c1.millis());
18.    System.out.println(c5.millis());
19. }
20. public static void testInstant() {
21.      //瞬时时间 相当于以前的 System.currentTimeMillis()
22.      Instant instant1 = Instant.now();
23.      System.out.println(instant1.getEpochSecond());//精确到秒 得到相对于 1970-01-01 00:00:00
24. UTC 的一个时间
25.      System.out.println(instant1.toEpochMilli()); //精确到毫秒
26.      Clock clock1 = Clock.systemUTC(); //获取系统 UTC 默认时钟
27.      Instant instant2 = Instant.now(clock1);//得到时钟的瞬时时间
28.      System.out.println(instant2.toEpochMilli());
29.      Clock clock2 = Clock.fixed(instant1, ZoneId.systemDefault()); //固定瞬时时间时钟
30.      Instant instant3 = Instant.now(clock2);//得到时钟的瞬时时间
31.      System.out.println(instant3.toEpochMilli());//equals instant1
32.  }
33.   public static void testLocalDateTime() {
34.   //使用默认时区时钟瞬时时间创建 Clock.systemDefaultZone() -->即相对于 ZoneId.systemDefault()
35.默认时区
36.   LocalDateTime now = LocalDateTime.now();
37.   System.out.println(now);
38.   //自定义时区
39.   LocalDateTime now2 = LocalDateTime.now(ZoneId.of("Europe/Paris"));
40.   System.out.println(now2);//会以相应的时区显示日期
41.    //自定义时钟
42.   Clock clock = Clock.system(ZoneId.of("Asia/Dhaka"));
43.   LocalDateTime now3 = LocalDateTime.now(clock);
44.   System.out.println(now3);//会以相应的时区显示日期
45.   //不需要写什么相对时间 如 java.util.Date 年是相对于 1900 月是从 0 开始
46.   //2013-12-31 23:59
47.   LocalDateTime d1 = LocalDateTime.of(2013, 12, 31, 23, 59);
48.   //年月日 时分秒 纳秒
49.   LocalDateTime d2 = LocalDateTime.of(2013, 12, 31, 23, 59, 59, 11);
50.   //使用瞬时时间 + 时区
51.   Instant instant = Instant.now();
52.   LocalDateTime d3 = LocalDateTime.ofInstant(Instant.now(), ZoneId.systemDefault());
53.   System.out.println(d3);
54.   //解析 String--->LocalDateTime
55.   LocalDateTime d4 = LocalDateTime.parse("2013-12-31T23:59");
56.   System.out.println(d4);
57.   LocalDateTime d5 = LocalDateTime.parse("2013-12-31T23:59:59.999");//999 毫秒 等价于
```


 Copyright© 2018 黑马程序员

```
58. 999000000 纳秒
59.
60.    System.out.println(d5);
61.    //使用 DateTimeFormatter API 解析 和 格式化
62.    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
63.    LocalDateTime d6 = LocalDateTime.parse("2013/12/31 23:59:59", formatter);
64.    System.out.println(formatter.format(d6));
65.    //时间获取
66.    System.out.println(d6.getYear());
67.    System.out.println(d6.getMonth());
68.    System.out.println(d6.getDayOfYear());
69.    System.out.println(d6.getDayOfMonth());
70.    System.out.println(d6.getDayOfWeek());
71.    System.out.println(d6.getHour());
72.    System.out.println(d6.getMinute());
73.    System.out.println(d6.getSecond());
74.    System.out.println(d6.getNano());
75.    //时间增减
76.    LocalDateTime d7 = d6.minusDays(1);
77.    LocalDateTime d8 = d7.plus(1, IsoFields.QUARTER_YEARS);
78.    //LocalDate 即年月日 无时分秒
79.    //LocalTime 即时分秒 无年月日
80.    //API 和 LocalDateTime 类似就不演示了
81.    }
82.    public static void testZonedDateTime() {
83.    //即带有时区的 date-time 存储纳秒、时区和时差（避免与本地 date-time 歧义）。
84.    //API 和 LocalDateTime 类似，只是多了时差(如 2013-12-20T10:35:50.711+08:00[Asia/Shanghai])
85.    ZonedDateTime now = ZonedDateTime.now();
86.    System.out.println(now);
87.    ZonedDateTime now2 = ZonedDateTime.now(ZoneId.of("Europe/Paris"));
88.    System.out.println(now2);
89.    //其他的用法也是类似的 就不介绍了
90.    ZonedDateTime z1 = ZonedDateTime.parse("2013-12-31T23:59:59Z[Europe/Paris]");
91.    System.out.println(z1);
92. }
93.    public static void testDuration() {
94.    //表示两个瞬时时间的时间段
95.    Duration d1 = Duration.between(Instant.ofEpochMilli(System.currentTimeMillis() - 12323123),
96.  Instant.now())
97. ;
98.    //得到相应的时差
99.    System.out.println(d1.toDays());
100.    System.out.println(d1.toHours());
```


 Copyright© 2018 黑马程序员

```
101.    System.out.println(d1.toMinutes());
102.
103.    System.out.println(d1.toMillis());
104.   System.out.println(d1.toNanos());
105.   //1 天时差 类似的还有如 ofHours()
106.   Duration d2 = Duration.ofDays(1);
107.   System.out.println(d2.toDays());
108. }
109.   public static void testChronology() {
110.    //提供对 java.util.Calendar 的替换，提供对年历系统的支持
111.     Chronology c = HijrahChronology.INSTANCE;
112.     ChronoLocalDateTime d = c.localDateTime(LocalDateTime.now());
113.     System.out.println(d);
114.   }
115. /**
116. * 新旧日期转换
117. */
118.  public static void testNewOldDateConversion(){
119.   Instant instant=new Date().toInstant();
120.   Date date=Date.from(instant);
121.   System.out.println(instant);
122.   System.out.println(date);
123.  }
124.  public static void main(String[] args) throws InterruptedException {
125.   testClock();
126.   testInstant();
127.   testLocalDateTime();
128.   testZonedDateTime();
129.   testDuration();
130.   testChronology();
131.   testNewOldDateConversion();
132.  }
133. }
```

7.9 JSR310 规范 Joda-Time 的区别（2017-11-23-wl）

其实 JSR310 的规范领导者 Stephen Colebourne，同时也是 Joda-Time 的创建者，JSR310 是在 Joda-Time 的基础上建立的，参考了绝大部分的 API，但并不是说 JSR310=JODA-Time，下面几个比较明显的区别是：

http://www.itheima.com Copyright© 2018 黑马程序员

1. 最明显的变化就是包名（从 org.joda.time 以及 java.time）

2. JSR310 不接受 NULL 值，Joda-Time 视 NULL 值为 0

3. JSR310 的计算机相关的时间（Instant）和与人类相关的时间（DateTime）之间的差别变得更明显

4. JSR310 所有抛出的异常都是 DateTimeException 的子类。虽然 DateTimeException 是一个 RuntimeException

7.10 总结 (2017-11-23-wl)

Java.time	java.util.Calendar 以及 Date
流畅的 API	不流畅的 API
实例不可变	实例可变
线程安全	非线程安全

六、Java 的数据类型

1. Java 的基本数据类型都有哪些各占几个字节

如下表所示:

四类	八种	字节数	数据表示范围
整型	byte	1	-128～127
	short	2	-32768～32767
	int	4	-2147483648～2147483647
	long	8	-2^{63}～$2^{63}-1$
浮点型	float	4	-3.403E38～3.403E38
	double	8	-1.798E308～1.798E308
字符型	char	2	表示一个字符，如('a'，'A'，'0'，'家')

http://www.itheima.com Copyright© 2018 黑马程序员

布尔型	boolean	1	只有两个值 true 与 false

2. String 是基本数据类型吗？ (2017-11-12-wl)

String 是引用类型，底层用 char 数组实现的。

3. short s1 = 1; s1 = s1 + 1; 有错吗?short s1 = 1; s1 += 1 有错吗; (2017-11-12-wl)

前者不正确，后者正确。对于 short s1 = 1; s1 = s1 + 1;由于 1 是 int 类型，因此 s1+1 运算结果也是 int 型，需要强制转换类型才能赋值给 short 型。而 short s1 = 1; s1 += 1;可以正确编译，因为 s1+= 1;相当于 s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

4. int 和 和 Integer 有什么区别? (2017-11-12-wl)

Java 是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，为了能够将这些基本数据类型当成对象操作，Java 为每一个基本数据类型都引入了对应的包装类型（wrapper class），int 的包装类就是 Integer，从 Java 5 开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 为每个原始类型提供了包装类型：

- 原始类型: boolean，char，byte，short，int，long，float，double

- 包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

```
class AutoUnboxingTest {

    public static void main(String[] args) {
        Integer a = new Integer(3);
        Integer b = 3;                    // 将 3 自动装箱成 Integer 类型
        int c = 3;
        System.out.println(a == b);       // false 两个引用没有引用同一对象
        System.out.println(a == c);       // true a 自动拆箱成 int 类型再和 c 比较
    }
}
```

ight© 2018 黑马程序员

5. 下面 Integer 类型的数值比较输出的结果为？(2017-11-12-wl)

```
public class Test03 {

    public static void main(String[] args) {
        Integer f1 = 100, f2 = 100, f3 = 150, f4 = 150;

        System.out.println(f1 == f2);
        System.out.println(f3 == f4);
    }
}
```

如果不明就里很容易认为两个输出要么都是 true 要么都是 false。首先需要注意的是 f1、f2、f3、f4 四个变量都是 Integer 对象引用，所以下面的==运算比较的不是值而是引用。装箱的本质是什么呢？当我们给一个 Integer 对象赋一个 int 值的时候，会调用 Integer 类的静态方法 valueOf，如果看看 valueOf 的源代码就知道发生了什么。

源码：

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

IntegerCache 是 Integer 的内部类，其代码如下所示：

```
/**
     * Cache to support the object identity semantics of autoboxing for values between
     * -128 and 127 (inclusive) as required by JLS.
     *
     * The cache is initialized on first usage.  The size of the cache
     * may be controlled by the {@code -XX:AutoBoxCacheMax=<size>} option.
     * During VM initialization, java.lang.Integer.IntegerCache.high property
     * may be set and saved in the private system properties in the
     * sun.misc.VM class.
     */
```

```
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
            } catch( NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}
```

简单的说，如果整型字面量的值在-128 到 127 之间，那么不会 new 新的 Integer 对象，而是直接引用常量池中的 Integer 对象，所以上面的面试题中 f1==f2 的结果是 true，而 f3==f4 的结果是 false。

提醒：越是貌似简单的面试题其中的玄机就越多，需要面试者有相当深厚的功力。

http://www.itheima.com Copyright© 2018 黑马程序员

6. String 类常用方法（2017-11-15-lyq）

方法	说明
int length()	返回当前字符串的长度
int indexOf(int ch)	查找ch字符在该字符串中第一次出现的位置
int indexOf(String str)	查找str子字符串在该字符串中第一次出现的位置
int lastIndexOf(int ch)	查找ch字符在该字符串中最后一次出现的位置
int lastIndexOf(String str)	查找str子字符串在该字符串中最后一次出现的位置
String substring(int beginIndex)	获取从beginIndex位置开始到结束的子字符串
String substring(int beginIndex, int endIndex)	获取从beginIndex位置开始到endIndex位置的子字符串
String trim()	返回去除了前后空格的字符串
boolean equals(Object obj)	将该字符串与指定对象比较，返回true或false
String toLowerCase()	将字符串转换为小写
String toUpperCase()	将字符串转换为大写
char charAt(int index)	获取字符串中指定位置的字符
String[] split(String regex,int limit)	将字符串分割为子字符串，返回字符串数组
byte[] getBytes()	将该字符串转换为byte数组

7. String、StringBuffer、StringBuilder 的区别？（2017-11-23-wzz）

（1）、可变不可变

String：字符串常量，在修改时不会改变自身；若修改，等于重新生成新的字符串对象。

StringBuffer：在修改时会改变对象自身，每次操作都是对 StringBuffer 对象本身进行修改，不是生成新的对象；使用场景：对字符串经常改变情况下，主要方法：append（），insert（）等。

（2）、线程是否安全

String：对象定义后不可变，线程安全。

StringBuffer：是线程安全的（对调用方法加入同步锁），执行效率较慢，适用于多线程下操作字符串缓冲区大量数据。

StringBuilder：是线程不安全的，适用于单线程下操作字符串缓冲区大量数据。

（3）、共同点

StringBuilder 与 StringBuffer 有公共父类 AbstractStringBuilder(抽象类)。

http://www.itheima.com Copyright© 2018 黑马程序员

StringBuilder、StringBuffer 的方法都会调用 AbstractStringBuilder 中的公共方法，如 super.append(...)。只是 StringBuffer 会在方法上加 synchronized 关键字，进行同步。最后，如果程序不是多线程的，那么使用 StringBuilder 效率高于 StringBuffer。

8. 数据类型之间的转换（2017-11-23-wzz）

（1）、字符串如何转基本数据类型?

调用基本数据类型对应的包装类中的方法 parseXXX(String)或 valueOf(String)即可返回相应基本类型。

（2）、基本数据类型如何转字符串?

一种方法是将基本数据类型与空字符串（“”）连接（+）即可获得其所对应的字符串；另一种方法是调用 String 类中的 valueOf()方法返回相应字符串。

七、Java 的 IO

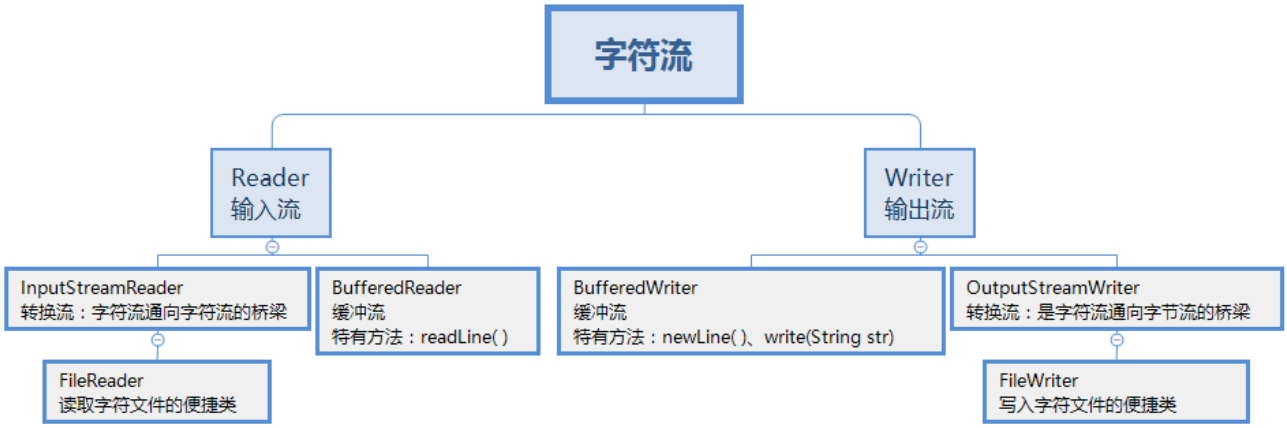
1. Java 中有几种类型的流（2017-11-23-wzz）

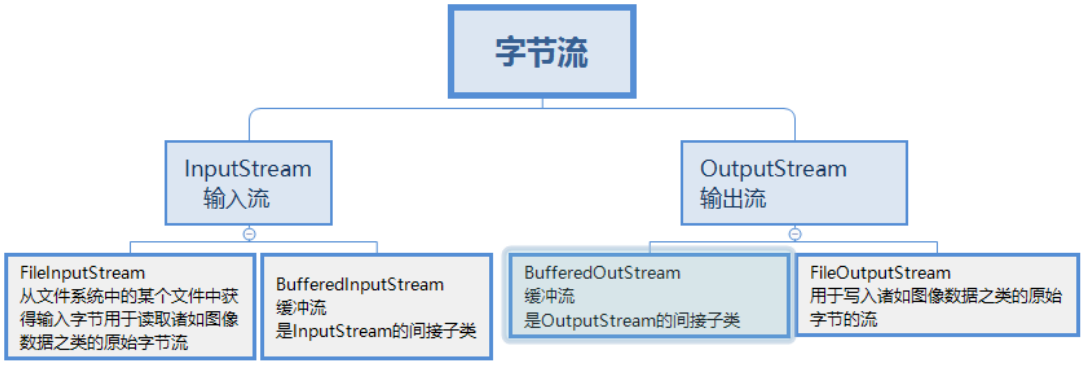
按照流的方向：输入流（inputStream）和输出流（outputStream）。

按照实现功能分：节点流（可以从或向一个特定的地方（节点）读写数据。如 FileReader）和处理流（是对一个已存在的流的连接和封装，通过所封装的流的功能调用实现数据读写。如 BufferedReader。处理流的构造方法总是要带一个其他的流对象做参数。一个流对象经过其他流的多次包装，称为流的链接。）

按照处理数据的单位：字节流和字符流。字节流继承于 InputStream 和 OutputStream，字符流继承于 InputStreamReader 和 OutputStreamWriter。

http://www.itheima.com Copyright© 2018 黑马程序员





2. 字节流如何转为字符流

字节输入流转字符输入流通过 InputStreamReader 实现，该类的构造函数可以传入 InputStream 对象。

字节输出流转字符输出流通过 OutputStreamWriter 实现，该类的构造函数可以传入 OutputStream 对象。

3. 如何将一个 java 对象序列化到文件里

在 java 中能够被序列化的类必须先实现 Serializable 接口，该接口没有任何抽象方法只是起到一个标记作用。

```
1. //对象输出流
2.       ObjectOutputStream objectOutputStream =
3. new ObjectOutputStream(new FileOutputStream(new File("D://obj")));
4.       objectOutputStream.writeObject(new User("zhangsan", 100));
5.       objectOutputStream.close();
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
6.      //对象输入流
7.      ObjectInputStream objectInputStream =
8. new ObjectInputStream(new FileInputStream(new File("D://obj")));
9.      User user = (User)objectInputStream.readObject();
10.     System.out.println(user);
11.     objectInputStream.close();
```

4. 字节流和字符流的区别（2017-11-23-wzz）

字节流读取的时候，读到一个字节就返回一个字节； 字符流使用了字节流读到一个或多个字节（中文对应的字节数是两个，在 UTF-8 码表中是 3 个字节）时。先去查指定的编码表，将查到的字符返回。 字节流可以处理所有类型数据，如：图片，MP3，AVI 视频文件，而字符流只能处理字符数据。只要是处理纯文本数据，就要优先考虑使用字符流，除此之外都用字节流。字节流主要是操作 byte 类型数据，以 byte 数组为准，主要操作类就是 OutputStream、InputStream

字符流处理的单元为 2 个字节的 Unicode 字符，分别操作字符、字符数组或字符串，而字节流处理单元为 1 个字节，操作字节和字节数组。所以字符流是由 Java 虚拟机将字节转化为 2 个字节的 Unicode 字符为单位的字符而成的，所以它对多国语言支持性比较好！如果是音频文件、图片、歌曲，就用字节流好点，如果是关系到中文（文本）的，用字符流好点。在程序中一个字符等于两个字节，java 提供了 Reader、Writer 两个专门操作字符流的类。

5. 如何实现对象克隆？(2017-11-12-wl)

有两种方式。

1). 实现 Cloneable 接口并重写 Object 类中的 clone()方法；

2). 实现 Serializable 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆，代码如下。

```
12. import java.io.ByteArrayInputStream;
13. import java.io.ByteArrayOutputStream;
14. import java.io.ObjectInputStream;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
15. import java.io.ObjectOutputStream;
16. import java.io.Serializable;
17. public class MyUtil {
18. private MyUtil() {
19. throw new AssertionError();
20. }
21. @SuppressWarnings("unchecked")
22. public static <T extends Serializable> T clone(T obj) throws Exception {
23. ByteArrayOutputStream bout = new ByteArrayOutputStream();
24. ObjectOutputStream oos = new ObjectOutputStream(bout);
25. oos.writeObject(obj);
26. ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());
27. ObjectInputStream ois = new ObjectInputStream(bin);
28. return (T) ois.readObject();
29. // 说明：调用 ByteArrayInputStream 或 ByteArrayOutputStream 对象的 close 方法没有任何意义
30. // 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外部资源（如文件流）的释放
31. }
32. }
```

测试代码：

```
1. import java.io.Serializable;
2. /**
3. * 人类
4. */
5. class Person implements Serializable {
6. private static final long serialVersionUID = -9102017020286042305L;
7. private String name; // 姓名
8. private int age; // 年龄
9. private Car car; // 座驾
10. public Person(String name, int age, Car car) {
11. this.name = name;
12. this.age = age;
13. this.car = car;
14. }
15. public String getName() {
16. return name;
17. }
18. public void setName(String name) {
19. this.name = name;
20. }
21. public int getAge() {
22. return age;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
23. }
24. public void setAge(int age) {
25. this.age = age;
26. }
27. public Car getCar() {
28. return car;
29. }
30. public void setCar(Car car) {
31. this.car = car;
32. }
33. @Override
34. public String toString() {
35. return "Person [name=" + name + ", age=" + age + ", car=" + car + "]";
36. }
37. }
```

```
1. /**
2. * 小汽车类
3. */
4. class Car implements Serializable {
5. private static final long serialVersionUID = -5713945027627603702L;
6. private String brand; // 品牌
7. private int maxSpeed; // 最高时速
8. public Car(String brand, int maxSpeed) {
9. this.brand = brand;
10. this.maxSpeed = maxSpeed;
11. }
12. public String getBrand() {
13. return brand;
14. }
15. public void setBrand(String brand) {
16. this.brand = brand;
17. }
18. public int getMaxSpeed() {
19. return maxSpeed;
20. }
21. public void setMaxSpeed(int maxSpeed) {
22. this.maxSpeed = maxSpeed;
23. }
24. @Override
25. public String toString() {
26. return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
27. }
28. }
```

```
1. class CloneTest {
2. public static void main(String[] args) {
3. try {
4. Person p1 = new Person("Hao LUO", 33, new Car("Benz", 300));
5. Person p2 = MyUtil.clone(p1); // 深度克隆
6. p2.getCar().setBrand("BYD");
7. // 修改克隆的 Person 对象 p2 关联的汽车对象的品牌属性
8. // 原来的 Person 对象 p1 关联的汽车不会受到任何影响
9. // 因为在克隆 Person 对象时其关联的汽车对象也被克隆了
10. System.out.println(p1);
11. } catch (Exception e) {
12. e.printStackTrace();
13. }
14. }
15. }
```

注意：基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种是方案明显优于使用 Object 类的 clone 方法克隆对象。让问题在编译的时候暴露出来总是好过把问题留到运行时。

6. 什么是 java 序列化，如何实现 java 序列化？(2017-12-7-lyq)

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题。

序列化的实现：将需要被序列化的类实现 Serializable 接口，该接口没有需要实现的方法，implements Serializable 只是为了标注该对象是可被序列化的，然后使用一个输出流(如：FileOutputStream)来构造一个 ObjectOutputStream(对象流)对象，接着，使用 ObjectOutputStream 对象的 writeObject(Object obj)方法就可以将参数为 obj 的对象写出(即保存其状态)，要恢复的话则用输入流。

原文链接：https://www.cnblogs.com/yangchunze/p/6728086.html

http://www.itheima.com Copyright© 2018 黑马程序员

八、Java 的集合

1. HashMap 排序题，上机题。(本人主要靠这道题入职的第一家公司)

已知一个 HashMap<Integer，User>集合， User 有 name（String）和 age（int）属性。请写一个方法实现对 HashMap 的排序功能，该方法接收 HashMap<Integer，User>为形参，返回类型为 HashMap<Integer，User>，要求对 HashMap 中的 User 的 age 倒序进行排序。排序时 key=value 键值对不得拆散。

注意：要做出这道题必须对集合的体系结构非常的熟悉。HashMap 本身就是不可排序的，但是该道题偏偏让给 HashMap 排序，那我们就得想在 API 中有没有这样的 Map 结构是有序的，LinkedHashMap，对的，就是他，他是 Map 结构，也是链表结构，有序的，更可喜的是他是 HashMap 的子类，我们返回 LinkedHashMap<Integer,User>

http://www.itheima.com Copyright© 2018 黑马程序员

即可，还符合面向接口（父类编程的思想）。

但凡是对集合的操作，我们应该保持一个原则就是能用 JDK 中的 API 就有 JDK 中的 API，比如排序算法我们不应该去用冒泡或者选择，而是首先想到用 Collections 集合工具类。

```
1. public class HashMapTest {
2.  public static void main(String[] args) {
3.      HashMap<Integer, User> users = new HashMap<>();
4.      users.put(1, new User("张三", 25));
5.      users.put(3, new User("李四", 22));
6.      users.put(2, new User("王五", 28));
7.      System.out.println(users);
8.      HashMap<Integer,User> sortHashMap = sortHashMap(users);
9.      System.out.println(sortHashMap);
10.     /**
11.      * 控制台输出内容
12.      * {1=User [name=张三, age=25], 2=User [name=王五, age=28], 3=User [name=李四, age=22]}
13.         {2=User [name=王五, age=28], 1=User [name=张三, age=25], 3=User [name=李四, age=22]}
14.      */
15. }
16.
17. public static HashMap<Integer, User> sortHashMap(HashMap<Integer, User> map) {
18.     // 首先拿到 map 的键值对集合
19.     Set<Entry<Integer, User>> entrySet = map.entrySet();
20.
21.     // 将 set 集合转为 List 集合，为什么，为了使用工具类的排序方法
22.     List<Entry<Integer, User>> list = new ArrayList<Entry<Integer, User>>(entrySet);
23.     // 使用 Collections 集合工具类对 list 进行排序，排序规则使用匿名内部类来实现
24.     Collections.sort(list, new Comparator<Entry<Integer, User>>() {
25.
26.         @Override
27.         public int compare(Entry<Integer, User> o1, Entry<Integer, User> o2) {
28.             //按照要求根据 User 的 age 的倒序进行排
29.             return o2.getValue().getAge()-o1.getValue().getAge();
30.         }
31.     });
32.     //创建一个新的有序的 HashMap 子类的集合
33.     LinkedHashMap<Integer, User> linkedHashMap = new LinkedHashMap<Integer, User>();
34.     //将 List 中的数据存储在 LinkedHashMap 中
35.     for(Entry<Integer, User> entry : list){
36.         linkedHashMap.put(entry.getKey(), entry.getValue());
37.     }
```


 Copyright© 2018 黑马程序员

```
38.     //返回结果
39.     return linkedHashMap;
40. }
41. }
42.
```

2. 集合的安全性问题

请问 ArrayList、HashSet、HashMap 是线程安全的吗？如果不是我想要线程安全的集合怎么办？

我们都看过上面那些集合的源码（如果没有那就看看吧），每个方法都没有加锁，显然都是线程不安全的。话又说过来如果他们安全了也就没第二问了。

在集合中 Vector 和 HashTable 倒是线程安全的。你打开源码会发现其实就是把各自核心方法添加上了 **synchronized** 关键字。

Collections 工具类提供了相关的 API，可以让上面那 3 个不安全的集合变为安全的。

```
1. //        Collections.synchronizedCollection(c)
2. //        Collections.synchronizedList(list)
3. //        Collections.synchronizedMap(m)
4. //        Collections.synchronizedSet(s)
```

上面几个函数都有对应的返回值类型，传入什么类型返回什么类型。打开源码其实实现原理非常简单，就是将集合的核心方法添加上了 **synchronized** 关键字。

3. ArrayList 内部用什么实现的？（2015-11-24）

（回答这样的问题，不要只回答个皮毛，可以再介绍一下 ArrayList 内部是如何实现数组的增加和删除的，因为数组在创建的时候长度是固定的，那么就有个问题我们往 ArrayList 中不断的添加对象，它是如何管理这些数组呢？）

ArrayList 内部是用 Object[]实现的。接下来我们分别分析 ArrayList 的构造、add、remove、clear 方法的实现原理。

http://www.itheima.com Copyright© 2018 黑马程序员

一、构造函数

1）空参构造

```
/**
    * Constructs a new {@code ArrayList} instance with zero initial capacity.
    */
   public ArrayList() {
       array = EmptyArray.OBJECT;
   }
```

array 是一个 Object[]类型。当我们 new 一个空参构造时系统调用了 EmptyArray.OBJECT 属性，EmptyArray 仅仅是一个系统的类库，该类源码如下：

```
   public final class EmptyArray {
       private EmptyArray() {}

       public static final boolean[] BOOLEAN = new boolean[0];
       public static final byte[] BYTE = new byte[0];
       public static final char[] CHAR = new char[0];
       public static final double[] DOUBLE = new double[0];
       public static final int[] INT = new int[0];

       public static final Class<?>[] CLASS = new Class[0];
       public static final Object[] OBJECT = new Object[0];
       public static final String[] STRING = new String[0];
       public static final Throwable[] THROWABLE = new Throwable[0];
       public static final StackTraceElement[] STACK_TRACE_ELEMENT = new StackTraceElement[0];
   }
```

也就是说当我们 new 一个空参 ArrayList 的时候，系统内部使用了一个 new Object[0]数组。

2）带参构造 1

```
/**
    * Constructs a new instance of {@code ArrayList} with the specified
    * initial capacity.
    *
    * @param capacity
    *          the initial capacity of this {@code ArrayList}.
    */
   public ArrayList(int capacity) {
       if (capacity < 0) {
           throw new IllegalArgumentException("capacity < 0: " + capacity);
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        }
        array = (capacity == 0 ? EmptyArray.OBJECT : new Object[capacity]);
    }
```

该构造函数传入一个 int 值，该值作为数组的长度值。如果该值小于 0，则抛出一个运行时异常。如果等于 0，则使用一个空数组，如果大于 0，则创建一个长度为该值的新数组。

3）带参构造 2

```
/**
     * Constructs a new instance of {@code ArrayList} containing the elements of
     * the specified collection.
     *
     * @param collection
     *          the collection of elements to add.
     */
    public ArrayList(Collection<? extends E> collection) {
        if (collection == null) {
            throw new NullPointerException("collection == null");
        }

        Object[] a = collection.toArray();
        if (a.getClass() != Object[].class) {
            Object[] newArray = new Object[a.length];
            System.arraycopy(a, 0, newArray, 0, a.length);
            a = newArray;
        }
        array = a;
        size = a.length;
    }
```

如果调用构造函数的时候传入了一个 Collection 的子类，那么先判断该集合是否为 null，为 null 则抛出空指针异常。如果不是则将该集合转换为数组 a，然后将该数组赋值为成员变量 array，将该数组的长度作为成员变量 size。这里面它先判断 a.getClass 是否等于 Object[].class，其实一般都是相等的，我也暂时没想明白为什么多加了这个判断，toArray 方法是 Collection 接口定义的，因此其所有的子类都有这样的方法，list 集合的 toArray 和 Set 集合的 toArray 返回的都是 Object[]数组。

这里讲些题外话，其实在看 Java 源码的时候，作者的很多意图都很费人心思，我能知道他的目标是啥，但是不知

http://www.itheima.com Copyright© 2018 黑马程序员

道他为何这样写。比如对于 ArrayList， array 是他的成员变量，但是每次在方法中使用该成员变量的时候作者都会重新在方法中开辟一个局部变量，然后给局部变量赋值为 array，然后再使用，有人可能说这是为了防止并发修改 array，毕竟 array 是成员变量，大家都可以使用因此需要将 array 变为局部变量，然后再使用，这样的说法并不是都成立的，也许有时候就是老外们写代码的一个习惯而已。

二、add 方法

add 方法有两个重载，这里只研究最简单的那个。

```
 /**
 * Adds the specified object at the end of this {@code ArrayList}.
 *
 * @param object
 *          the object to add.
 * @return always true
 */
@Override public boolean add(E object) {
    Object[] a = array;
    int s = size;
    if (s == a.length) {
        Object[] newArray = new Object[s +
                (s < (MIN_CAPACITY_INCREMENT / 2) ?
                 MIN_CAPACITY_INCREMENT : s >> 1)];
        System.arraycopy(a, 0, newArray, 0, s);
        array = a = newArray;
    }
    a[s] = object;
    size = s + 1;
    modCount++;
    return true;
}
```

1、首先将成员变量 array 赋值给局部变量 a，将成员变量 size 赋值给局部变量 s。

2、判断集合的长度 s 是否等于数组的长度（如果集合的长度已经等于数组的长度了，说明数组已经满了，该重新分配新数组了），重新分配数组的时候需要计算新分配内存的空间大小，如果当前的长度小于 MIN_CAPACITY_INCREMENT/2（这个常量值是 12，除以 2 就是 6，也就是如果当前集合长度小于 6）则分配 12 个

http://www.itheima.com Copyright© 2018 黑马程序员

长度，如果集合长度大于 6 则分配当前长度 s 的一半长度。这里面用到了三元运算符和位运算，s >> 1，意思就是将 s 往右移 1 位，相当于 s=s/2，只不过位运算是效率最高的运算。

3、将新添加的 object 对象作为数组的 a[s]个元素。

4、修改集合长度 size 为 s+1

5、modCotun++,该变量是父类中声明的，用于记录集合修改的次数，记录集合修改的次数是为了防止在用迭代器迭代集合时避免并发修改异常，或者说用于判断是否出现并发修改异常的。

6、return true，这个返回值意义不大，因为一直返回 true，除非报了一个运行时异常。

三、remove 方法

remove 方法有两个重载，我们只研究 remove（int index）方法。

```
 /**
 * Removes the object at the specified location from this list.
 *
 * @param index
 *          the index of the object to remove.
 * @return the removed object.
 * @throws IndexOutOfBoundsException
 *           when {@code location < 0 || location >= size()}
 */
@Override public E remove(int index) {
    Object[] a = array;
    int s = size;
    if (index >= s) {
        throwIndexOutOfBoundsException(index, s);
    }
    @SuppressWarnings("unchecked")
    E result = (E) a[index];
    System.arraycopy(a, index + 1, a, index, --s - index);
    a[s] = null;  // Prevent memory leak
    size = s;
    modCount++;
    return result;
}
```

1、先将成员变量 array 和 size 赋值给局部变量 a 和 s。

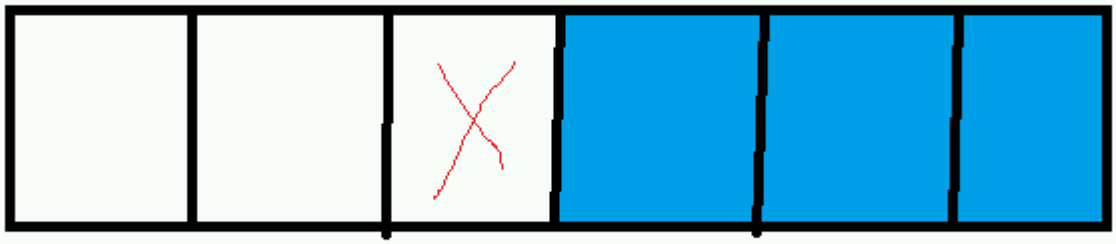
http://www.itheima.com Copyright© 2018 黑马程序员

2、判断形参 index 是否大于等于集合的长度，如果成了则抛出运行时异常

3、获取数组中脚标为 index 的对象 result，该对象作为方法的返回值

4、调用 System 的 arraycopy 函数，拷贝原理如下图所示。

```
System.arraycopy(a, index + 1, a, index, --s - index);
```



假设index=2，那么要删除的对象如图打叉部分，我们只需要将该数组后面蓝色区域整体往前移动一位位置即可。上面的代码完成就是这个工作。

5、接下来就是很重要的一个工作，因为删除了一个元素，而且集合整体向前移动了一位，因此需要将集合最后一个元素设置为 null，否则就可能内存泄露。

6、重新给成员变量 array 和 size 赋值

7、记录修改次数

8、返回删除的元素（让用户再看最后一眼）

四、clear 方法

```
    /**
     * Removes all elements from this {@code ArrayList}, leaving it empty.
     *
     * @see #isEmpty
     * @see #size
     */
    @Override public void clear() {
        if (size != 0) {
            Arrays.fill(array, 0, size, null);
            size = 0;
            modCount++;
        }
    }
```


http://www.itheima.com Copyright© 2018 黑马程序员

如果集合长度不等于 0，则将所有数组的值都设置为 null，然后将成员变量 size 设置为 0 即可，最后让修改记录加 1。

4. 并发集合和普通集合如何区别？（2015-11-24）

并发集合常见的有 ConcurrentHashMap、ConcurrentLinkedQueue、ConcurrentLinkedDeque 等。并发集合位于 java.util.concurrent 包下，是 jdk1.5 之后才有的，主要作者是 Doug Lea（http://baike.baidu.com/view/3141057.htm）完成的。

在 java 中有普通集合、同步（线程安全）的集合、并发集合。普通集合通常性能最高，但是不保证多线程的安全性和并发的可靠性。线程安全集合仅仅是给集合添加了 synchronized 同步锁，严重牺牲了性能，而且对并发的效率就更低了，并发集合则通过复杂的策略不仅保证了多线程的安全又提高的并发时的效率。

参考阅读：

ConcurrentHashMap 是线程安全的 HashMap 的实现，默认构造同样有 initialCapacity 和 loadFactor 属性，不过还多了一个 concurrencyLevel 属性，三属性默认值分别为 16、0.75 及 16。其内部使用锁分段技术，维持这锁 Segment 的数组，在 Segment 数组中又存放着 Entity[]数组，内部 hash 算法将数据较均匀分布在不同锁中。

put 操作：并没有在此方法上加上 synchronized，首先对 key.hashcode 进行 hash 操作，得到 key 的 hash 值。hash 操作的算法和 map 也不同，根据此 hash 值计算并获取其对应的数组中的 Segment 对象(继承自 ReentrantLock)，接着调用此 Segment 对象的 put 方法来完成当前操作。

ConcurrentHashMap 基于 concurrencyLevel 划分出了多个 Segment 来对 key-value 进行存储，从而避免每次 put 操作都得锁住整个数组。在默认的情况下，最佳情况下可允许 16 个线程并发无阻塞的操作集合对象，尽可能地减少并发时的阻塞现象。

get(key)

http://www.itheima.com Copyright© 2018 黑马程序员

首先对 key.hashCode 进行 hash 操作，基于其值找到对应的 Segment 对象，调用其 get 方法完成当前操作。而 Segment 的 get 操作首先通过 hash 值和对象数组大小减 1 的值进行按位与操作来获取数组上对应位置的 HashEntry。在这个步骤中，可能会因为对象数组大小的改变，以及数组上对应位置的 HashEntry 产生不一致性，那么 ConcurrentHashMap 是如何保证的?

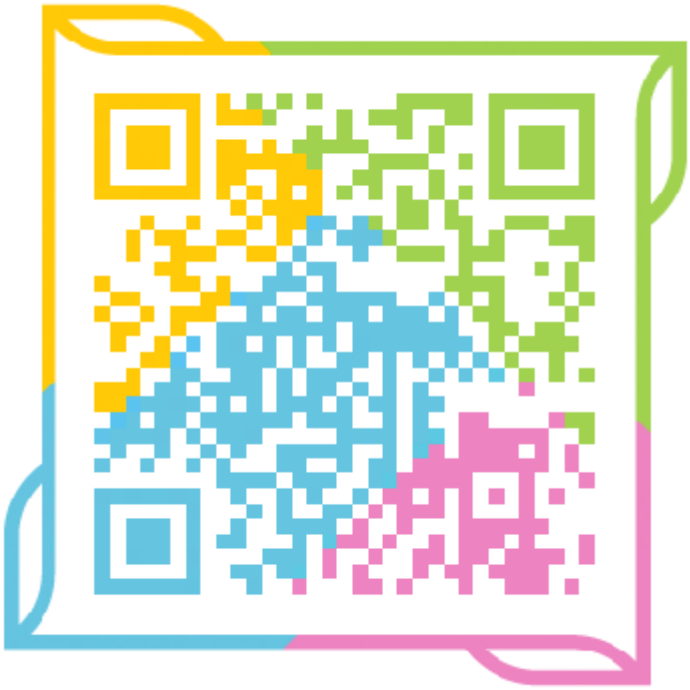
对象数组大小的改变只有在 put 操作时有可能发生，由于 HashEntry 对象数组对应的变量是 volatile 类型的，因此可以保证如 HashEntry 对象数组大小发生改变，读操作可看到最新的对象数组大小。

在获取到了 HashEntry 对象后，怎么能保证它及其 next 属性构成的链表上的对象不会改变呢？这点 ConcurrentHashMap 采用了一个简单的方式，即 HashEntry 对象中的 hash、key、next 属性都是 final 的，这也就意味着没办法插入一个 HashEntry 对象到基于 next 属性构成的链表中间或末尾。这样就可以保证当获取到 HashEntry 对象后，其基于 next 属性构建的链表是不会发生变化的。

ConcurrentHashMap 默认情况下采用将数据分为 16 个段进行存储，并且 16 个段分别持有各自不同的锁 Segment，锁仅用于 put 和 remove 等改变集合对象的操作，基于 volatile 及 HashEntry 链表的不变性实现了读取的不加锁。这些方式使得 ConcurrentHashMap 能够保持极好的并发支持，尤其是对于读远比插入和删除频繁的 Map 而言，而它采用的这些方法也可谓是对于 Java 内存模型、并发机制深刻掌握的体现。

推荐博客地址：http://m.oschina.net/blog/269037

http://www.itheima.com Copyright© 2018 黑马程序员



5. List 的三个子类的特点（2017-2-23）

ArrayList 底层结构是数组,底层查询快,增删慢。

LinkedList 底层结构是链表型的,增删快,查询慢。

voctor 底层结构是数组 线程安全的,增删慢,查询慢。

6. List 和 Map、Set 的区别（2017-11-22-wzz）

6.1 结构特点

List 和 Set 是存储单列数据的集合，Map 是存储键和值这样的双列数据的集合；List 中存储的数据是有顺序，并且允许重复；Map 中存储的数据是没有顺序的，其键是不能重复的，它的值是可以有重复的，Set 中存储的数据是无序的，且不允许有重复，但元素在集合中的位置由元素的 hashcode 决定，位置是固定的（Set 集合根据 hashcode 来

http://www.itheima.com Copyright© 2018 黑马程序员

进行数据的存储，所以位置是固定的，但是位置不是用户可以控制的，所以对于用户来说 set 中的元素还是无序的）；

6.2 实现类

List 接口有三个实现类（**LinkedList**：基于链表实现，链表内存是散乱的，每一个元素存储本身内存地址的同时还存储下一个元素的地址。链表增删快，查找慢；**ArrayList**：基于数组实现，非线程安全的，效率高，便于索引，但不便于插入删除；**Vector**：基于数组实现，线程安全的，效率低）。

Map 接口有三个实现类（**HashMap**：基于 hash 表的 Map 接口实现，非线程安全，高效，支持 null 值和 null 键；**HashTable**：线程安全，低效，不支持 null 值和 null 键；**LinkedHashMap**：是 HashMap 的一个子类，保存了记录的插入顺序；SortMap 接口：TreeMap，能够把它保存的记录根据键排序，默认是键值的升序排序）。

Set 接口有两个实现类（**HashSet**：底层是由 HashMap 实现，不允许集合中有重复的值，使用该方式时需要重写 equals()和 hashCode()方法；LinkedHashSet：继承与 HashSet，同时又基于 LinkedHashMap 来进行实现，底层使用的是 LinkedHashMp）。

6.3 区别

List 集合中对象按照索引位置排序，可以有重复对象，允许按照对象在集合中的索引位置检索对象，例如通过 list.get(i)方法来获取集合中的元素；Map 中的每一个元素包含一个键和一个值，成对出现，键对象不可以重复，值对象可以重复；Set 集合中的对象不按照特定的方式排序，并且没有重复对象，但它的实现类能对集合中的对象按照特定的方式排序，例如 TreeSet 类，可以按照默认顺序，也可以通过实现 Java.util.Comparator<Type>接口来自定义排序方式。

7. HashMap 和 HashTable 有什么区别？（2017-2-23）

HashMap 是线程不安全的,HashMap 是一个接口,是 Map 的一个子接口,是将键映射到值得对象,不允许键值重复,

http://www.itheima.com Copyright© 2018 黑马程序员

允许空键和空值;由于非线程安全,HashMap 的效率要较 HashTable 的效率高一些.

HashTable 是线程安全的一个集合,不允许 null 值作为一个 key 值或者 Value 值;

HashTable 是 sychronize,多个线程访问时不需要自己为它的方法实现同步,而 HashMap 在被多个线程访问的时候需要自己为它的方法实现同步;

8. 数组和链表分别比较适合用于什么场景，为什么？（2017-2-23）

8.1 数组和链表简介

在计算机中要对给定的数据集进行若干处理，首要任务是把数据集的一部分（当数据量非常大时，可能只能一部分一部分地读取数据到内存中来处理）或全部存储到内存中，然后再对内存中的数据进行各种处理。

例如，对于数据集 S{1，2，3，4，5，6}，要求 S 中元素的和，首先要把数据存储到内存中，然后再将内存中的数据相加。

当内存空间中有足够大的连续空间时，可以把数据连续的存放在内存中，各种编程语言中的数组一般都是按这种方式存储的（也可能有例外），如图 1（b）；当内存中只有一些离散的可用空间时，想连续存储数据就非常困难了，这时能想到的一种解决方式是移动内存中的数据，把离散的空间聚集成连续的一块大空间，如图 1（c）所示，这样做当然也可以，但是这种情况因为可能要移动别人的数据，所以会存在一些困难，移动的过程中也有可能会把一些别人的重要数据给丢失。另外一种，不影响别人的数据存储方式是把数据集中的数据分开离散地存储到这些不连续空间中，如图（d）。这时为了能把数据集中的所有数据联系起来，需要在前一块数据的存储空间中记录下一块数据的地址，这样只要知道第一块内存空间的地址就能环环相扣地把数据集整体联系在一起了。C/C++中用指针实现的链表就是这种存储形式。

http://www.itheima.com Copyright© 2018 黑马程序员

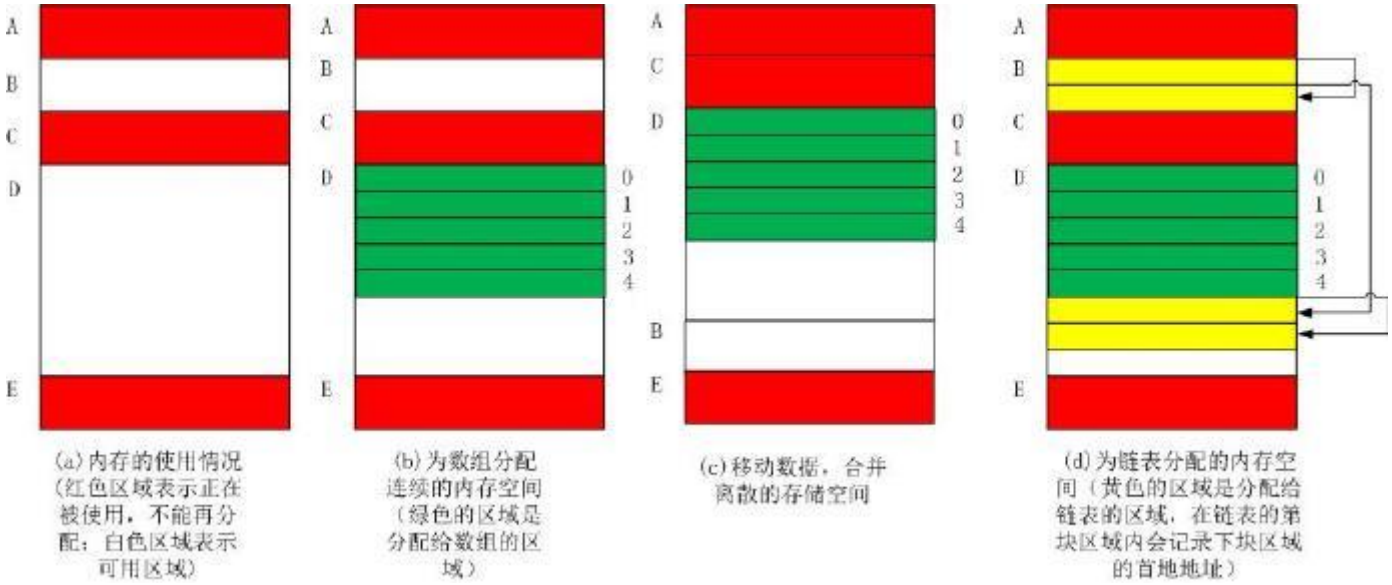


图　内存分配

由上可知，内存中的存储形式可以分为连续存储和离散存储两种。因此，数据的物理存储结构就有连续存储和离散存储两种，它们对应了我们通常所说的数组和链表，

8.2 数组和链表的区别

数组是将元素在内存中连续存储的；它的优点：因为数据是连续存储的，内存地址连续，所以在查找数据的时候效率比较高；它的缺点：在存储之前，我们需要申请一块连续的内存空间，并且在编译的时候就必须确定好它的空间的大小。在运行的时候空间的大小是无法随着你的需要进行增加和减少而改变的，当数据两比较大的时候，有可能会出现越界的情况，数据比较小的时候，又有可能会浪费掉内存空间。在改变数据个数时，增加、插入、删除数据效率比较低

链表是动态申请内存空间，不需要像数组需要提前申请好内存的大小，链表只需在用的时候申请就可以，根据需要来动态申请或者删除内存空间，对于数据增加和删除以及插入比数组灵活。还有就是链表中数据在内存中可以在任意的位置，通过应用来关联数据（就是通过存在元素的指针来联系）

http://www.itheima.com Copyright© 2018 黑马程序员

8.3 链表和数组使用场景

数组应用场景：数据比较少；经常做的运算是按序号访问数据元素；数组更容易实现，任何高级语言都支持；构建的线性表较稳定。

链表应用场景：对线性表的长度或者规模难以估计；频繁做插入删除操作；构建动态性比较强的线性表。

参考博客：http://blog.csdn.net/u011277123/article/details/53908387

8.4 跟数组相关的面试题

用面向对象的方法求出数组中重复 value 的个数，按如下个数输出：

1 出现：1 次

3 出现：2 次

8 出现：3 次

2 出现：4 次

```
int[] arr = {1,4,1,4,2,5,4,5,8,7,8,77,88,5,4,9,6,2,4,1,5};
```


http://www.itheima.com Copyright© 2018 黑马程序员

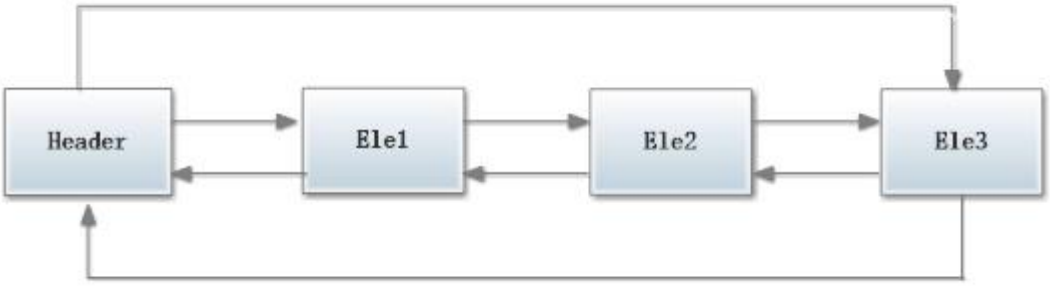
9. Java 中 ArrayList 和 Linkedlist 区别？（2017-2-23）

ArrayList 和 Vector 使用了数组的实现，可以认为 ArrayList 或者 Vector 封装了对内部数组的操作，比如向数组中添加，删除，插入新的元素或者数据的扩展和重定向。

LinkedList 使用了循环双向链表数据结构。与基于数组的 ArrayList 相比，这是两种截然不同的实现技术，这也决定了它们将适用于完全不同的工作场景。

LinkedList 链表由一系列表项连接而成。一个表项总是包含 3 个部分：元素内容，前驱表和后驱表，如图所示：

在下图展示了一个包含 3 个元素的 LinkedList 的各个表项间的连接关系。在 JDK 的实现中，无论 LikedList 是否为空，链表内部都有一个 header 表项，它既表示链表的开始，也表示链表的结尾。表项 header 的后驱表项便是链表中第一个元素，表项 header 的前驱表项便是链表中最后一个元素。



http://www.itheima.com Copyright© 2018 黑马程序员

10. List a=new ArrayList()和 ArrayList a =new ArrayList()的区别？ (2017-2-24)

List list = new ArrayList();这句创建了一个 ArrayList 的对象后把上溯到了 List。此时它是一个 List 对象了，有些 ArrayList 有但是 List 没有的属性和方法，它就不能再用了。而 ArrayList list=new ArrayList();创建一对象则保留了 ArrayList 的所有属性。 所以需要用到 ArrayList 独有的方法的时候不能用前者。实例代码如下：

```
1. List list = new ArrayList();
2. ArrayList arrayList = new ArrayList();
3. list.trimToSize(); //错误，没有该方法。
4. arrayList.trimToSize();   //ArrayList 里有该方法。
```

11. 要对集合更新操作时，ArrayList 和 LinkedList 哪个更适合？(2017-2-24)

1.ArrayList 是实现了基于动态数组的数据结构，LinkedList 基于链表的数据结构。

2.如果集合数据是对于集合随机访问 get 和 set，ArrayList 绝对优于 LinkedList，因为 LinkedList 要移动指针。

3.如果集合数据是对于集合新增和删除操作 add 和 remove，LinedList 比较占优势，因为 ArrayList 要移动数据。

ArrayList 和 LinkedList 是两个集合类，用于存储一系列的对象引用(references)。例如我们可以用 ArrayList 来存储一系列的 String 或者 Integer。那 么 ArrayList 和 LinkedList 在性能上有什么差别呢？什么时候应该用 ArrayList 什么时候又该用 LinkedList 呢?

一．时间复杂度

首先一点关键的是，ArrayList 的内部实现是基于基础的对象数组的，因此，它使用 get 方法访问列表中的任意一个元素时(random access)，它的速度要比 LinkedList 快。LinkedList 中的 get 方法是按照顺序从列表的一端开始检查，直到另外一端。对 LinkedList 而言，访问列表中的某个指定元素没有更快的方法了。

http://www.itheima.com Copyright© 2018 黑马程序员

假设我们有一个很大的列表，它里面的元素已经排好序了，这个列表可能是 ArrayList 类型的也可能是 LinkedList 类型的，现在我们对这个列表来进行二分查找(binary search)，比较列表是 ArrayList 和 LinkedList 时的查询速度，看下面的程序：

```
1. public class TestList{
2.       public static final int N=50000;     //50000 个数
3.       public static List values;           //要查找的集合
4.       //放入 50000 个数给 value;
5.       static{
6.           Integer vals[]=new Integer[N];
7.           Random r=new Random();
8.           for(int i=0,currval=0;i<N;i++)...{
9.               vals=new Integer(currval);
10.               currval+=r.nextInt(100)+1;
11.           }
12.           values=Arrays.asList(vals);
13.       }
14.      //通过二分查找法查找
15.       static long timeList(List lst){
16.           long start=System.currentTimeMillis();
17.           for(int i=0;i<N;i++)...{
18.               int index=Collections.binarySearch(lst, values.get(i));
19.               if(index!=i)
20.                   System.out.println("***错误***");
21.           }
22.           return System.currentTimeMillis()-start;
23.       }
24.       public static void main(String args[])...{
25.           System.out.println("ArrayList 消耗时间："+timeList(new ArrayList(values)));
26.           System.out.println("LinkedList 消耗时间："+timeList(new LinkedList(values)));
27.       }
28. }
```

得到的输出是：

```
1. ArrayList 消耗时间：15
2. LinkedList 消耗时间：2596
```

这个结果不是固定的，但是基本上 ArrayList 的时间要明显小于 LinkedList 的时间。因此在这种情况下不宜用 LinkedList。二分查找法使用的随机访问(random access)策略，而 LinkedList 是不支持快速的随机访问的。对一个

http://www.itheima.com Copyright© 2018 黑马程序员

LinkedList 做随机访问所消耗的时间与这个 list 的大小是成比例的。而相应的，在 ArrayList 中进行随机访问所消耗的时间是固定的。

这是否表明 ArrayList 总是比 LinkedList 性能要好呢？这并不一定，在某些情况下 LinkedList 的表现要优于 ArrayList，有些算法在 LinkedList 中实现 时效率更高。比方说，利用 Collections.reverse 方法对列表进行反转时，其性能就要好些。看这样一个例子，加入我们有一个列表，要对其进行大量的插入和删除操作，在这种情况下 LinkedList 就是一个较好的选择。请看如下一个极端的例子，我们重复的在一个列表的开端插入一个元素：

```
1. import java.util.*;
2. public class ListDemo {
3.       static final int N=50000;
4.       static long timeList(List list){
5.            long start=System.currentTimeMillis();
6.            Object o = new Object();
7.                for(int i=0;i<N;i++)
8.                    list.add(0, o);
9.            return System.currentTimeMillis()-start;
10.       }
11.       public static void main(String[] args) {
12.            System.out.println("ArrayList 耗时："+timeList(new ArrayList()));
13.            System.out.println("LinkedList 耗时："+timeList(new LinkedList()));
14.       }
15. }
```

这时我的输出结果是

```
1. ArrayList 耗时：2463
2. LinkedList 耗时：15
```

二．空间复杂度

在 LinkedList 中有一个私有的内部类，定义如下：

```
1. private static class Entry {
2.           Object element;
3.           Entry next;
4.           Entry previous;
5.       }
```


http://www.itheima.com Copyright© 2018 黑马程序员

每个 Entry 对象 reference 列表 中的一个元素，同时还有在 LinkedList 中它的上一个元素和下一个元素。一个有 1000 个元素的 LinkedList 对象将有 1000 个链接在一起 的 Entry 对象，每个对象都对应于列表中的一个元素。这样的话，在一个 LinkedList 结构中将有一个很大的空间开销，因为它要存储这 1000 个 Entity 对象的相关信息。

ArrayList 使用一个内置的数组来存 储元素，这个数组的起始容量是 10.当数组需要增长时，新的容量按如下公式获得：新容量=(旧容量*3)/2+1，也就是说每一次容量大概会增长 50%。 这就意味着，如果你有一个包含大量元素的 ArrayList 对象，那么最终将有很大的空间会被浪费掉，这个浪费是由 ArrayList 的工作方式本身造成 的。如果没有足够的空间来存放新的元素，数组将不得不被重新进行分配以便能够增加新的元素。对数组进行重新分配，将会导致性能急剧下降。如果我们知道一个 ArrayList 将会有多少个元素，我们可以通过构造方法来指定容量。我们还可以通过 trimToSize 方法在 ArrayList 分配完毕之后去掉浪 费掉的空间。

三．总结

ArrayList 和 LinkedList 在性能上各有优缺点，都有各自所适用的地方，总的说来可以描述如下：

1．对 ArrayList 和 LinkedList 而言，在列表末尾增加一个元素所花的开销都是固定的。对 ArrayList 而言，主要是在内部数组中增加一项，指向所添加的元素，偶 尔可能会导致对数组重新进行分配；而对 LinkedList 而言，这个开销是统一的，分配一个内部 Entry 对象。

2．在 ArrayList 的中间插入或删除一个元素意味着这个列表中剩余的元素都会被移动；而在 LinkedList 的中间插入或删除一个元素的开销是固定的。

3．LinkedList 不支持高效的随机元素访问。

4．ArrayList 的空间浪费主要体现在在 list 列表的结尾预留一定的容量空间，而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗相当的空间

可以这样说：当操作是在一列数据的后面添加数据而不是在前面或中间,并且需要随机地访问其中的元素时,使用

http://www.itheima.com Copyright© 2018 黑马程序员

ArrayList 会提供比较好的性能；当你的操作是在一列数据的前面或中间添加或删除数据,并且按照顺序访问其中的元素时,就应该使用 LinkedList 了。

12. 请用两个队列模拟堆栈结构（2017-2-24）

两个队列模拟一个堆栈，队列是先进先出，而堆栈是先进后出。模拟如下

队列 a 和 b

（1）入栈：a 队列为空，b 为空。例：则将”a,b,c,d,e”需要入栈的元素先放 a 中，a 进栈为”a,b,c,d,e”

（2）出栈：a 队列目前的元素为”a,b,c,,d,e”。将 a 队列依次加入 Arraylist 集合 a 中。以倒序的方法，将 a 中的集合取出，放入 b 队列中，再将 b 队列出列。代码如下：

```
1.  public static void main(String[] args) {
2.          Queue<String> queue = new LinkedList<String>();  //a 队列
3.          Queue<String> queue2=new LinkedList<String>();   //b 队列
4.          ArrayList<String> a=new ArrayList<String>();     //arrylist 集合是中间参数
5.          //往 a 队列添加元素
6.          queue.offer("a");
7.          queue.offer("b");
8.          queue.offer("c");
9.          queue.offer("d");
10.         queue.offer("e");
11.         System.out.print("进栈：");
12.        //a 队列依次加入 list 集合之中
13.         for(String q : queue){
14.          a.add(q);
15.           System.out.print(q);
16.         }
17.        //以倒序的方法取出（a 队列依次加入 list 集合）之中的值，加入 b 对列
18.         for(int i=a.size()-1;i>=0;i--){
19.          queue2.offer(a.get(i));
20.         }
21.        //打印出栈队列
22.         System.out.println("");
23.         System.out.print("出栈：");
```


http://www.itheima.com Copyright© 2018 黑马程序员

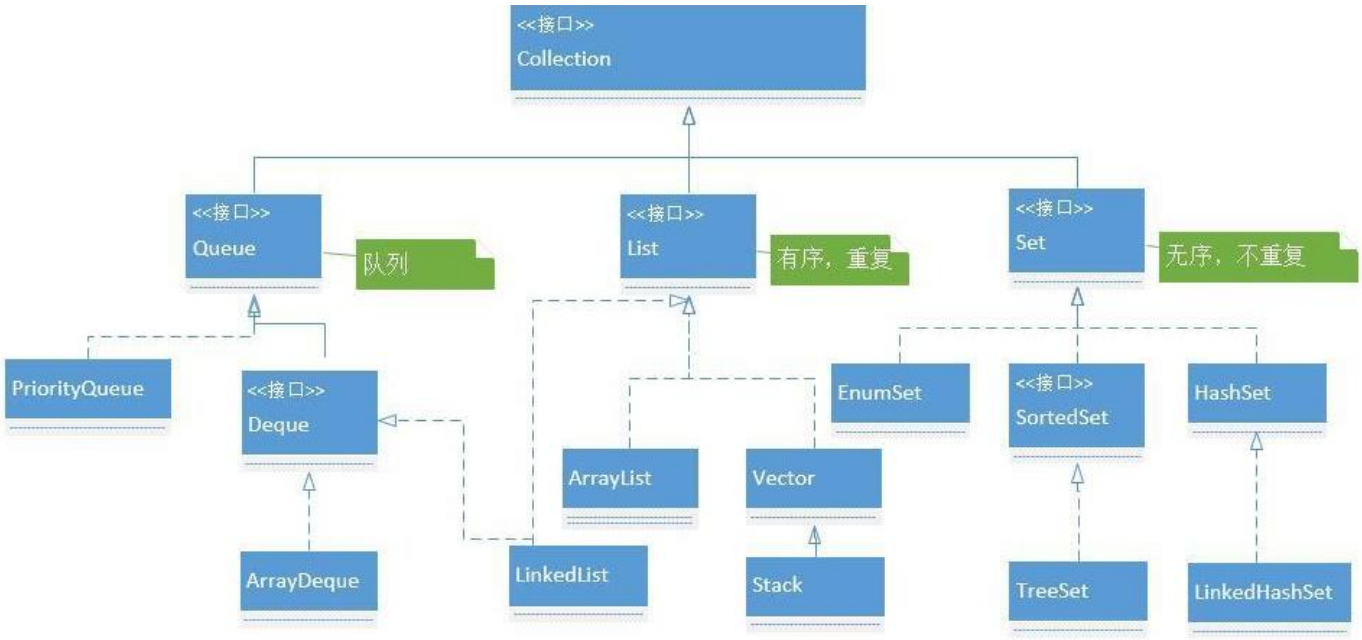
```
24.         for(String q : queue2){
25.             System.out.print(q);
26.         }
27.     }
```

打印结果为（遵循栈模式先进后出）：

```
进栈：a b c d e
出栈：e d c b a
```

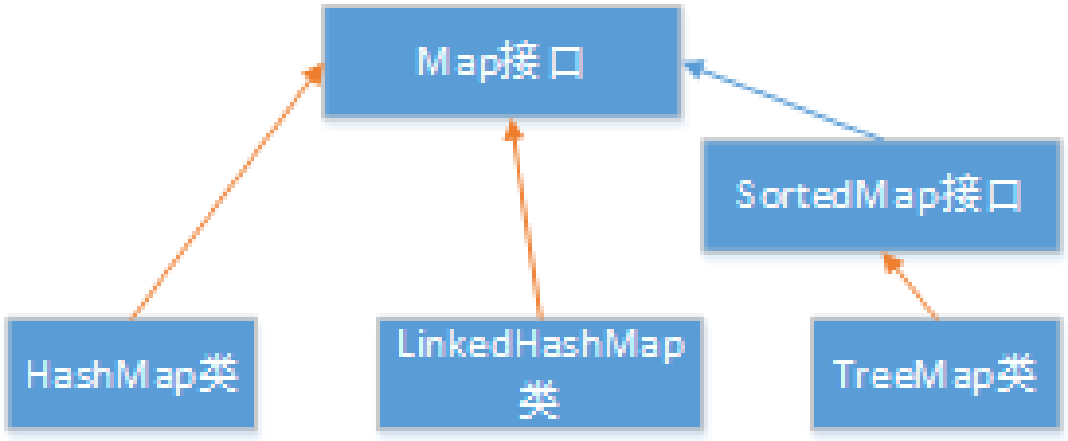
13. Collection 和 Map 的集成体系（2017-11-14-lyq）

Collection：



Map：

http://www.itheima.com Copyright© 2018 黑马程序员



14. Map 中的 key 和 value 可以为 null 么？ （2017-11-21-gxb）

HashMap 对象的 key、value 值均可为 null。

HahTable 对象的 key、value 值均不可为 null。

且两者的的 key 值均不能重复，若添加 key 相同的键值对，后面的 value 会自动覆盖前面的 value，但不会报错。

测试代码如下：

```
1. public class Test {
2.
3.     public static void main(String[] args) {
4.         Map<String, String> map = new HashMap<String, String>();//HashMap 对象
5.         Map<String, String> tableMap = new Hashtable<String, String>();//HashTable 对象
6.
7.         map.put(null, null);
8.         System.out.println("hashMap 的[key]和[value]均可以为 null:" + map.get(null));
9.
10.         try {
11.             tableMap.put(null, "3");
12.             System.out.println(tableMap.get(null));
13.         } catch (Exception e) {
14.             System.out.println("【ERROR】: hashTable 的[key]不能为 null");
15.         }
16.
17.         try {
18.             tableMap.put("3", null);
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            System.out.println(tableMap.get("3"));
        } catch (Exception e) {
            System.out.println("【ERROR】: hashTable 的[value]不能为 null");
        }
    }

}
```

运行结果:

hashMap 的[key]和[value]均可以为 null:null

【ERROR】: hashTable 的[key]不能为 null

【ERROR】: hashTable 的[value]不能为 null

九、Java 的多线程和并发库

对于 Java 程序员来说，多线程在工作中的使用场景还是比较常见的，而仅仅掌握了 Java 中的传统多线程机制，还是不够的。在 JDK5.0 之后，Java 增加的并发库中提供了很多优秀的 API，在实际开发中用的比较多。因此在看具体的面试题之前我们有必要对这部分知识做一个全面的了解。

(一) 多线程基础知识--传统线程机制的回顾 (2017-12-11-wl)

(1) 传统使用类 Thread 和接口 Runnable 实现

1.在 Thread 子类覆盖的 run 方法中编写运行代码

方式一

```
new Thread(){
    @Override
    public void run(){
        while(true){
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        }
    }.start();
```

2.在传递给 Thread 对象的 Runnable 对象的 run 方法中编写代码

```
        new Thread(new Runnable(){
        public void run(){
            while(true){
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName());
            }
        }
    }).start();
```

3.总结

查看 Thread 类的 run()方法的源代码，可以看到其实这两种方式都是在调用 Thread 对象的 run 方法，如果 Thread 类的 run 方法没有被覆盖，并且为该 Thread 对象设置了一个 Runnable 对象，该 run 方法会调用 Runnable 对象的 run 方法

```
    /**
     * If this thread was constructed using a separate
     * <code>Runnable</code> run object, then that
     * <code>Runnable</code> object's <code>run</code> method is called;
     * otherwise, this method does nothing and returns.
     * <p>
     * Subclasses of <code>Thread</code> should override this method.
     *
     * @see     #start()
     * @see     #stop()
     * @see     #Thread(ThreadGroup, Runnable, String)
     */
    @Override
    public void run() {
        if (target != null) {
            target.run();
        }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
    }
```

（2） 定实现时器 Timer 和 TimerTask

Timer 在实际开发中应用场景不多，一般来说都会用其他第三方库来实现。但有时会在一些面试题中出现。下面我们就针对一道面试题来使用 Timer 定时类。

1.请模拟写出双重定时器（面试题）

```
要求：使用定时器,间隔 4 秒执行一次，再间隔 2 秒执行一次，以此类推执行。

class TimerTastCus extends TimerTask{
    @Override
    public void run() {
        count = (count +1)%2;
        System.err.println("Boob boom ");
        new Timer().schedule(new TimerTastCus(), 2000+2000*count);
    }
}

Timer  timer = new Timer();
timer.schedule(new TimerTastCus(), 2000+2000*count);

while (true) {
    System.out.println(new  Date().getSeconds());
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
//PS：下面的代码中的 count 变量中
//此参数要使用在你匿名内部类中，使用 final 修饰就无法对其值进行修改，
//只能改为静态变量
private   static volatile int count  = 0;
```

（3） 线程互斥与同步

在引入多线程后，由于线程执行的异步性，会给系统造成混乱，特别是在急用临界资源时，如多个线程急用同一台打印机，会使打印结果交织在一起，难于区分。当多个线程急用共享变量，表格，链表时，可能会导致数据处理出错，

 Copyright© 2018 黑马程序员

因此线程同步的主要任务是使并发执行的各线程之间能够有效的共享资源和相互合作，从而使程序的执行具有可再现性。

当线程并发执行时，由于资源共享和线程协作，使用线程之间会存在以下两种制约关系。

1. 间接相互制约。一个系统中的多个线程必然要共享某种系统资源，如共享 CPU，共享 I/O 设备，所谓间接相互制约即源于这种资源共享，打印机就是最好的例子，线程 A 在使用打印机时，其它线程都要等待。

2. 直接相互制约。这种制约主要是因为线程之间的合作，如有线程 A 将计算结果提供给线程 B 作进一步处理，那么线程 B 在线程 A 将数据送达之前都将处于阻塞状态。

间接相互制约可以称为**互斥**，直接相互制约可以称为**同步**，对于互斥可以这样理解，线程 A 和线程 B 互斥访问某个资源则它们之间就会产个顺序问题——要么线程 A 等待线程 B 操作完毕，要么线程 B 等待线程操作完毕，这其实就是线程的同步了。因此同步包括互斥，互斥其实是一种特殊的同步。

下面我们通过一道面试题来体会线程的交互。

要求：子线程运行执行 10 次后，主线程再运行 5 次。这样交替执行三遍

```
public static void main(String[] args) {
        final Bussiness bussiness = new Bussiness();
        //子线程
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 3; i++) {
                    bussiness.subMethod();
                }
            }
        }).start();
        //主线程
        for (int i = 0; i < 3; i++) {
            bussiness.mainMethod();
        }
    }
}


class Bussiness {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
    private boolean subFlag = true;

    public synchronized void mainMethod() {
        while (subFlag) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName()
                    + " : main thread running loop count -- " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        subFlag = true;
        notify();
    }

    public synchronized void subMethod() {
        while (!subFlag) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        for (int i = 0; i < 10; i++) {
            System.err.println(Thread.currentThread().getName()
                    + " : sub thread running loop count -- " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        subFlag = false;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            notify();
        }
    }
```

（4） 线程局部变量 ThreadLocal

- ThreadLocal 的作用和目的：用于实现线程内的数据共享，即对于相同的程序代码，多个模块在同一个线程中运行时要共享一份数据，而在另外线程中运行时又共享另外一份数据。
- 每个线程调用全局 ThreadLocal 对象的 set 方法，在 set 方法中，首先根据当前线程获取当前线程的 ThreadLocalMap 对象，然后往这个 map 中插入一条记录，key 其实是 ThreadLocal 对象，value 是各自的 set 方法传进去的值。也就是每个线程其实都有一份自己独享的 ThreadLocalMap 对象，该对象的 Key 是 ThreadLocal 对象，值是用户设置的具体值。在线程结束时可以调用 ThreadLocal.remove()方法，这样会更快释放内存，不调用也可以，因为线程结束后也可以自动释放相关的 ThreadLocal 变量。
- ThreadLocal 的应用场景：
  - 订单处理包含一系列操作：减少库存量、增加一条流水台账、修改总账，这几个操作要在同一个事务中完成，通常也即同一个线程中进行处理，如果累加公司应收款的操作失败了，则应该把前面的操作回滚，否则，提交所有操作，这要求这些操作使用相同的数据库连接对象，而这些操作的代码分别位于不同的模块类中。
  - 银行转账包含一系列操作： 把转出帐户的余额减少，把转入帐户的余额增加，这两个操作要在同一个事务中完成，它们必须使用相同的数据库连接对象，转入和转出操作的代码分别是两个不同的帐户对象的方法。
  - 例如 Strut2 的 ActionContext，同一段代码被不同的线程调用运行时，该代码操作的数据是每个线程各自的状态和数据，对于不同的线程来说，getContext 方法拿到的对象都不相同，对同一个线程来说，不管调用 getContext 方法多少次和在哪个模块中 getContext 方法，拿到的都是同一个。

http://www.itheima.com Copyright© 2018 黑马程序员

1. ThreadLocal 的使用方式

(1) 在关联数据类中创建 private static ThreadLocal

在下面的类中，私有静态 ThreadLocal 实例（serialNum）为调用该类的静态 SerialNum.get() 方法的每个线程维护了一个“序列号”，该方法将返回当前线程的序列号。（线程的序列号是在第一次调用 SerialNum.get() 时分配的，并在后续调用中不会更改。）

```
public class SerialNum {
    // The next serial number to be assigned
    private static int nextSerialNum = 0;

    private static ThreadLocal serialNum = new ThreadLocal() {
        protected synchronized Object initialValue() {
            return new Integer(nextSerialNum++);
        }
    };

    public static int get() {
        return ((Integer) (serialNum.get())).intValue();
    }
}
```

另一个例子，也是私有静态 ThreadLocal 实例：

```
public class ThreadContext {

private String userId;
private Long transactionId;

private static ThreadLocal threadLocal = new ThreadLocal(){
    @Override
    protected ThreadContext initialValue() {
        return new ThreadContext();
    }
};
public static ThreadContext get() {
    return threadLocal.get();
}

public String getUserId() {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        return userId;
    }
    public void setUserId(String userId) {
        this.userId = userId;
    }
    public Long getTransactionId() {
        return transactionId;
    }
    public void setTransactionId(Long transactionId) {
        this.transactionId = transactionId;
    }
}
```

补充：在 JDK 的 API 对 ThreadLocal 私有化的说明。并举例 ‘线程唯一标识符’

UniqueThreadIdGenerator ，大家学习是可以结合官方 API 来学习。

2. 在 Util 类中创建 ThreadLocal

这是上面用法的扩展，即把 ThreadLocal 的创建放到工具类中。

```
    public class HibernateUtil {
    private static Log log = LogFactory.getLog(HibernateUtil.class);
    private static final SessionFactory sessionFactory;     //定义 SessionFactory

    static {
        try {
            // 通过默认配置文件 hibernate.cfg.xml 创建 SessionFactory
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            log.error("初始化 SessionFactory 失败！", ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    //创建线程局部变量 session，用来保存 Hibernate 的 Session
    public static final ThreadLocal session = new ThreadLocal();

    /**
     * 获取当前线程中的 Session
     * @return Session
     * @throws HibernateException
     */
    public static Session currentSession() throws HibernateException {
```


 Copyright© 2018 黑马程序员

```
        Session s = (Session) session.get();
        // 如果 Session 还没有打开，则新开一个 Session
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);          //将新开的 Session 保存到线程局部变量中
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        //获取线程局部变量，并强制转换为 Session 类型
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
    }
}
```

3. 在 Runnable 中创建 ThreadLocal

在线程类内部创建 ThreadLocal，基本步骤如下：

①、在多线程的类（如 ThreadDemo 类）中，创建一个 ThreadLocal 对象 threadXxx，用来保存线程间需要隔离处理的对象 xxx。

②、在 ThreadDemo 类中，创建一个获取要隔离访问的数据的方法 getXxx()，在方法中判断，若 ThreadLocal 对象为 null 时候，应该 new()一个隔离访问类型的对象，并强制转换为要应用的类型

③、在 ThreadDemo 类的 run()方法中，通过调用 getXxx()方法获取要操作的数据，这样可以保证每个线程对应一个数据对象，在任何时刻都操作的是这个对象。

```
    public class ThreadLocalTest implements Runnable{

    ThreadLocal<Studen> studenThreadLocal = new ThreadLocal<Studen>();

    @Override
    public void run() {
        String currentThreadName = Thread.currentThread().getName();
        System.out.println(currentThreadName + " is running...");
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        Random random = new Random();
        int age = random.nextInt(100);
        System.out.println(currentThreadName + " is set age: "  + age);
        Studen  studen = getStudent(); //通过这个方法，为每个线程都独立的 new 一个 student 对象，每个线程的的student 对象都可以设置不同的值
        studen.setAge(age);
        System.out.println(currentThreadName + " is first get age: " + studen.getAge());
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println( currentThreadName + " is second get age: " + studen.getAge());

    }

    private Studen getStudent() {
        Studen studen = studenThreadLocal.get();
        if (null == studen) {
            studen = new Studen();
            studenThreadLocal.set(studen);
        }
        return studen;
    }

    public static void main(String[] args) {
        ThreadLocalTest t = new ThreadLocalTest();
        Thread t1 = new Thread(t,"Thread A");
        Thread t2 = new Thread(t,"Thread B");
        t1.start();
        t2.start();
    }

}

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


http://www.itheima.com Copyright© 2018 黑马程序员

```
}
```

（5） 多线程共享数据

在 Java 传统线程机制中的共享数据方式，大致可以简单分两种情况：

- **多个线程行为一致，共同操作一个数据源。**也就是每个线程执行的代码相同，可以使用同一个 Runnable 对象，这个 Runnable 对象中有那个共享数据，例如，卖票系统就可以这么做。
- **多个线程行为不一致，共同操作一个数据源。**也就是每个线程执行的代码不同，这时候需要用不同的 Runnable 对象。例如，银行存取款。

下面我们通过两个示例代码来分别说明这两种方式。

1. 多个线程行为一致共同操作一个数据

如果每个线程执行的代码相同，可以使用同一个 Runnable 对象，这个 Runnable 对象中有那个共享数据，例如，买票系统就可以这么做。

```
/**
*共享数据类
**/
class ShareData{
    private int num = 10 ;
    public  synchronized void inc() {
        num++;
        System.out.println(Thread.currentThread().getName()+": invoke  inc method num =" + num);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
/**
*多线程类
**/
class RunnableCusToInc implements Runnable{
    private ShareData  shareData;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        public RunnableCusToInc(ShareData data) {
            this.shareData = data;
        }
        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                shareData.inc();
            }
        }
    }

    /**
    *测试方法
    **/
    public static void main(String[] args) {

    ShareData    shareData = new ShareData();

    for (int i = 0; i < 4; i++) {
        new Thread(new    RunnableCusToInc(shareData),"Thread "+ i).start();
        }
    }
    }
```

2. 多个线程行为不一致共同操作一个数据

如果每个线程执行的代码不同，这时候需要用不同的 Runnable 对象，有如下两种方式来实现这些 Runnable 对象之间的数据共享：

1) 将共享数据封装在另外一个对象中，然后将这个对象逐一传递给各个 Runnable 对象。每个线程对共享数据的操作方法也分配到那个对象身上去完成，这样容易实现针对该数据进行的各个操作的互斥和通信。

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

       ShareData    shareData = new ShareData();

       for (int i = 0; i < 4; i++) {
            if(i%2 == 0){
                new Thread(new    RunnableCusToInc(shareData),"Thread "+ i).start();
            }else{
                new Thread(new    RunnableCusToDec(shareData),"Thread "+ i).start();
            }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        }
    }
    //封装共享数据类
    class RunnableCusToInc implements Runnable{

        //封装共享数据
        private ShareData    shareData;
        public RunnableCusToInc(ShareData data) {
            this.shareData = data;
        }
        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                shareData.inc();
            }
        }
    }

    //封装共享数据类
    class RunnableCusToDec implements Runnable{

        //封装共享数据
        private ShareData    shareData;

        public RunnableCusToDec(ShareData data) {
            this.shareData = data;
        }
        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                shareData.dec();
            }
        }
    }

    /**
    *共享数据类
    **/
    class ShareData{
        private int num = 10 ;
        public  synchronized void inc() {
            num++;
            System.out.println(Thread.currentThread().getName()+": invoke  inc method num =" + num);
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

2) 将这些 Runnable 对象作为某一个类中的内部类，共享数据作为这个外部类中的成员变量，每个线程对共享数据的操作方法也分配给外部类，以便实现对共享数据进行的各个操作的互斥和通信，作为内部类的各个 Runnable 对象调用外部类的这些方法。

```
public static void main(String[] args) {
    //公共数据
    final ShareData  shareData = new ShareData();
    for (int i = 0; i < 4; i++) {

        if(i%2 == 0){
            new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 5; i++) {
                        shareData.inc();
                    }
                }

            },"Thread "+ i).start();
        }else{
            new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 5; i++) {
                        shareData.dec();
                    }
                }
            },"Thread "+ i).start();
        }
    }
}
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
class ShareData{
    private int num = 10 ;
    public  synchronized void inc() {
        num++;
        System.out.println(Thread.currentThread().getName()+": invoke  inc method num =" + num);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public synchronized void dec() {
        num--;
        System.err.println(Thread.currentThread().getName()+": invoke  dec  method num =" + num);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

补充：上面两种方式的组合：将共享数据封装在另外一个对象中，每个线程对共享数据的操作方法也分配到那个对象身上去完成，对象作为这个外部类中的成员变量或方法中的局部变量，每个线程的 Runnable 对象作为外部类中的成员内部类或局部内部类。

总之，要同步互斥的几段代码最好是分别放在几个独立的方法中，这些方法再放在同一个类中，这样比较容易实现它们之间的同步互斥和通信。

（二）多线程基础知识--线程并发库（2017-12-11-wl）

Java 5 添加了一个新的包到 Java 平台，java.util.concurrent 包。这个包包含有一系列能够让 Java 的并发编程变得更加简单轻松的类。在这个包被添加以前，你需要自己去动手实现自己的相关工具类。下面带你认识下 java.util.concurrent 包里的这些类，然后你可以尝试着如何在项目中使用它们。本文中将使用 Java 6 版本，我不确

http://www.itheima.com Copyright© 2018 黑马程序员

定这和 Java 5 版本里的是否有一些差异。我不会去解释关于 Java 并发的核心问题 – 其背后的原理，也就是说，如果你对那些东西感兴趣，参考《Java 并发指南》。

（1） Java 的线程并发库介绍

Java5 的多线程并有两个大发库在 java.util.concurrent 包及子包中，子包主要的包有一下两个

1) java.util.concurrent 包 (**多线程并发库**)

➢ java.util.concurrent 包含许多线程安全、测试良好、高性能的并发构建块。不客气地说，创建 java.util.concurrent 的目的就是要实现 Collection 框架对数据结构所执行的并发操作。通过提供一组可靠的、高性能并发构建块，开发人员可以提高并发类的线程安全、可伸缩性、性能、可读性和可靠性，后面、我们会做介绍。

➢如果一些类名看起来相似，可能是因为 java.util.concurrent 中的许多概念源自 Doug Lea 的 util.concurrent 库。

2) java.util.concurrent.atomic 包 (**多线程的原子性操作提供的工具类**)

➢查看 atomic 包文档页下面的介绍，它可以对多线程的基本数据、数组中的基本数据和对象中的基本数据进行多线程的操作（AtomicInteger、AtomicIntegerArray、AtomicIntegerFieldUpDater...）

➢通过如下两个方法快速理解 atomic 包的意义：

■ AtomicInteger 类的 boolean compareAndSet(expectedValue, updateValue);

■ AtomicIntegerArray 类的 int addAndGet(int i, int delta);

➢顺带解释 volatile 类型的作用，需要查看 java 语言规范。

■ volatile 修饰的变量，线程在每次使用变量的时候，都会读取变量修改后的最的值。（具有可见性）

■ volatile 没有原子性。

3) java.util.concurrent.lock 包 (**多线程的锁机制**)

http://www.itheima.com Copyright© 2018 黑马程序员

为锁和等待条件提供一个框架的接口和类，它不同于内置同步和监视器。该框架允许更灵活地使用锁和条件。本包下有三大接口，下面简单介绍下：

- Lock 接口：支持那些语义不同（重入、公平等）的锁规则，可以在非阻塞式结构的上下文（包括 hand-over-hand 和锁重排算法）中使用这些规则。主要的实现是 ReentrantLock。
- ReadWriteLock 接口：以类似方式定义了一些读取者可以共享而写入者独占的锁。此包只提供了一个实现，即 ReentrantReadWriteLock，因为它适用于大部分的标准用法上下文。但程序员可以创建自己的、适用于非标准要求的实现。
- Condition 接口：描述了可能会与锁有关联的条件变量。这些变量在用法上与使用 Object.wait 访问的隐式监视器类似，但提供了更强大的功能。需要特别指出的是，单个 Lock 可能与多个 Condition 对象关联。为了避免兼容性问题，Condition 方法的名称与对应的 Object 版本中的不同。

（2） Java 的并发库入门

下面我们将分别介绍 java.util.concurrent 包下的常用类的使用。

1) java.util.concurrent 包

java.util.concurrent 包描述：

在并发编程中很常用的实用工具类。此包包括了几个小的、已标准化的可扩展框架，以及一些提供有用功能的类。此包下有一些组件，其中包括：

- 执行程序（线程池）
- 并发队列
- 同步器
- 并发 Collocation

http://www.itheima.com Copyright© 2018 黑马程序员

下面我们将 java.util.concurrent 包下的组件逐一简单介绍：

A. 执行程序

➢Executors 线程池工厂类

首次我们来说下线程池的作用:

线程池作用就是限制系统中执行线程的数量。

根据系统的环境情况，可以自动或手动设置线程数量，达到运行的最佳效果；少了浪费了系统资源，多了造成系统拥挤效率不高。用线程池控制线程数量，其他线程 排队等候。一个任务执行完毕，再从队列的中取最前面的任务开始执行。若队列中没有等待进程，线程池的这一资源处于等待。当一个新任务需要运行时，如果线程 池中有等待的工作线程，就可以开始运行了；否则进入等待队列。

为什么要用线程池:

- 减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务
- 可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为因为消耗过多的内存，而把服务器累趴下(每个线程需要大约 1MB 内存，线程开的越多，消耗的内存也就越大，最后死机)

Executors 详解:

Java 里面线程池的顶级接口是 Executor，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 ExecutorService。ThreadPoolExecutor 是 Executors 类的底层实现。我们先介绍下 Executors。

线程池的基本思想还是一种对象池的思想，开辟一块内存空间，里面存放了众多(未死亡)的线程，池中线

http://www.itheima.com Copyright© 2018 黑马程序员

程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。

Java5 中并发库中，线程池创建线程大致可以分为下面三种：

```
//创建固定大小的线程池
ExecutorService fPool = Executors.newFixedThreadPool(3);
//创建缓存大小的线程池
ExecutorService cPool = Executors.newCachedThreadPool();
//创建单一的线程池
ExecutorService sPool = Executors.newSingleThreadExecutor();
```

下面我们通过简单示例来分别说明：

- **固定大小连接池**

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
/**
* Java 线程：线程池-
*
* @author Administrator 2009-11-4 23:30:44
*/
public class Test {
   public static void main(String[] args) {
    //创建一个可重用固定线程数的线程池
    ExecutorService pool = Executors.newFixedThreadPool(2);
    //创建实现了 Runnable 接口对象，Thread 对象当然也实现了 Runnable 接口
    Thread t1 = new MyThread();
    Thread t2 = new MyThread();
    Thread t3 = new MyThread();
    Thread t4 = new MyThread();
   Thread t5 = new MyThread();
   //将线程放入池中进行执行
   pool.execute(t1);
   pool.execute(t2);
   pool.execute(t3);
   pool.execute(t4);
   pool.execute(t5);
   //关闭线程池
  pool.shutdown();
  }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
    }
    class MyThread extends Thread{
      @Override
      public void run() {
        System.out.println(Thread.currentThread().getName()+"正在执行。。。");
      }
    }
```

运行结果：

```
pool-1-thread-1 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-2 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-2 正在执行。。。
```

从上面的运行来看，我们 Thread 类都是在线程池中运行的，线程池在执行 execute 方法来执行 Thread 类中的 run 方法。不管 execute 执行几次，线程池始终都会使用 2 个线程来处理。不会再去创建出其他线程来处理 run 方法执行。这就是固定大小线程池。

- **单任务连接池**

我们将上面的代码

```
//创建一个可重用固定线程数的线程池
ExecutorService pool = Executors.newFixedThreadPool(2);
```

改为：

```
//创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。
ExecutorService pool = Executors.newSingleThreadExecutor();
```

运行结果：

```
pool-1-thread-1 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-1 正在执行。。。
```

运行结果看出，单任务线程池在执行 execute 方法来执行 Thread 类中的 run 方法。不管 execute 执行几次，线程池始终都会使用单个线程来处理。

补充：在 java 的多线程中，一但线程关闭，就会成为死线程。关闭后死线程就没有办法在启动了。再次启动就会出现

http://www.itheima.com Copyright© 2018 黑马程序员

异常信息：Exception in thread "main" java.lang.IllegalThreadStateException。那么如何解决这个问题呢?

我们这里就可以使用 Executors.newSingleThreadExecutor()来再次启动一个线程。（面试）

- **可变连接池**

```
//创建一个可重用固定线程数的线程池
ExecutorService pool = Executors.newFixedThreadPool(2);
改为：
//创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。
ExecutorService pool = Executors.newCachedThreadPool();


运行结果：
pool-1-thread-5 正在执行。。。
pool-1-thread-1 正在执行。。。
pool-1-thread-4 正在执行。。。
pool-1-thread-3 正在执行。。。
pool-1-thread-2 正在执行。。。
```

运行结果看出，可变任务线程池在执行 execute 方法来执行 Thread 类中的 run 方法。这里 execute 执行多次，线程池就会创建出多个线程来处理 Thread 类中 run 方法。所有我们看到连接池会根据执行的情况，在程序运行时创建多个线程来处理，这里就是可变连接池的特点。

那么在上面的三种创建方式，Executors 还可以在执行某个线程时，定时操作。那么下面我们通过代码简单演示下。

- **延迟连接池**

```
import java.util.concurrent.Executors;
    import java.util.concurrent.ScheduledExecutorService;
    import java.util.concurrent.TimeUnit;
    /**
    * Java 线程：线程池-
    *
    * @author Administrator 2009-11-4 23:30:44
    */
    public class Test {
    public static void main(String[] args) {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        //创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。
        ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
        //创建实现了 Runnable 接口对象，Thread 对象当然也实现了 Runnable 接口
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();
        //将线程放入池中进行执行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        //使用定时执行风格的方法
        pool.schedule(t4, 10, TimeUnit.MILLISECONDS);  //t4 和 t5 在 10 秒后执行
        pool.schedule(t5, 10, TimeUnit.MILLISECONDS);
        //关闭线程池
        pool.shutdown();
        }
        }
        class MyThread extends Thread {
         @Override
         public void run() {
           System.out.println(Thread.currentThread().getName() + "正在执行。。。");
         }
        }

    运行结果：
    pool-1-thread-1 正在执行。。。
    pool-1-thread-2 正在执行。。。
    pool-1-thread-1 正在执行。。。
    pool-1-thread-1 正在执行。。。
    pool-1-thread-2 正在执行。。。
```

➢ExecutorService 执行器服务

java.util.concurrent.ExecutorService 接口表示一个异步执行机制，使我们能够在后台执行任务。因此一个 ExecutorService 很类似于一个线程池。实际上，存在于 java.util.concurrent 包里的 ExecutorService 实现就是一个线程池实现。

ExecutorService 例子:

http://www.itheima.com Copyright© 2018 黑马程序员

以下是一个简单的 ExecutorService 例子：

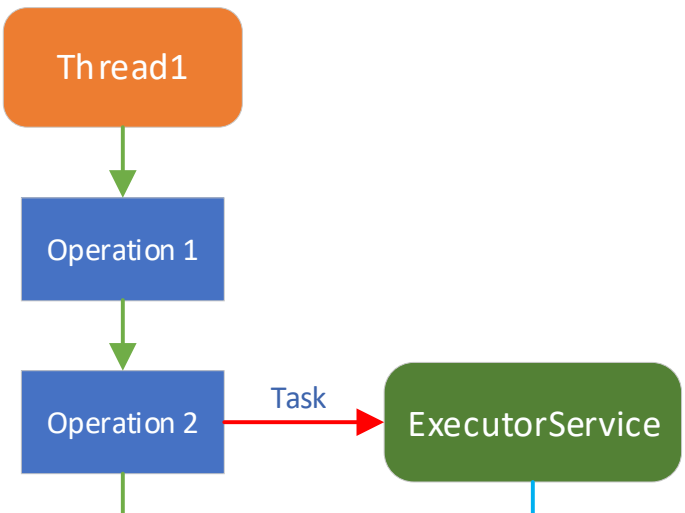
```
//线程工厂类创建出线程池
ExecutorService executorService = Executors.newFixedThreadPool(10);

//执行一个线程任务
executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

//线程池关闭
executorService.shutdown();
```

上面代码首先使用 newFixedThreadPool() 工厂方法创建一个 ExecutorService。这里创建了一个十个线程执行任务的线程池。然后，将一个 Runnable 接口的匿名实现类传递给 execute() 方法。这将导致 ExecutorService 中的某个线程执行该 Runnable。这里可以看成一个**任务分派**，示例代码中的任务分派我们可以理解为：

一个线程将一个任务委派给一个 ExecutorService 去异步执行。

一旦该线程将任务委派给 ExecutorService，该线程将继续它自己的执行，独立于该任务的执行。

如下图：



Copyright© 2018 黑马程序员

ExecutorService 实现:

既然 ExecutorService 是个接口，如果你想用它的话就得去使用它的实现类之一。

java.util.concurrent 包提供了 ExecutorService 接口的以下实现类:

- ThreadPoolExecutor
- ScheduledThreadPoolExecutor

ExecutorService 创建:

ExecutorService 的创建依赖于你使用的具体实现。但是你也可以使用 Executors 工厂类来创建 ExecutorService 实例。代码示例:

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();  //之前 Executors 已介绍
ExecutorService executorService2 = Executors.newFixedThreadPool(10);
ExecutorService executorService3 = Executors.newScheduledThreadPool(10);
```

ExecutorService 使用:

有几种不同的方式来将任务委托给 ExecutorService 去执行:

- execute(Runnable)
- submit(Runnable)

http://www.itheima.com Copyright© 2018 黑马程序员

- submit(Callable)
- invokeAny(...)
- invokeAll(...)

接下来我们挨个看一下这些方法。

✓ **execute(Runnable)**

execute(Runnable) 方法要求一个 java.lang.Runnable 对象，然后对它进行异步执行。以下是使用 ExecutorService 执行一个 Runnable 的示例：

```
//从 Executors 中获得 ExecutorService
ExecutorService executorService = Executors.newSingleThreadExecutor();
//执行 ExecutorService 中的方法
executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});
//线程池关闭
executorService.shutdown();
```

特点：没有办法得知被执行的 Runnable 的执行结果。如果有需要的话你得使用一个 Callable(以下将做介绍)。

✓ **submit(Runnable)**

submit(Runnable) 方法也要求一个 Runnable 实现类，但它返回一个 Future 对象。这个 Future 对象可以用来检查 Runnable 是否已经执行完毕。以下是 ExecutorService submit() 示例：

```
//从 Executors 中获得 ExecutorService
ExecutorService executorService = Executors.newSingleThreadExecutor();

Future future = executorService.submit(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
});

future.get();  //获得执行完 run 方法后的返回值，这里使用的 Runnable，所以这里没有返回值，返回的是 null。
executorService.shutdown();
```

✓ **submit(Runnable)**

submit(Callable) 方法类似于 submit(Runnable) 方法，除了它所要求的参数类型之外。Callable 实例除了它的 call() 方法能够返回一个结果之外和一个 Runnable 很相像。Runnable.run() 不能够返回一个结果。Callable 的结果可以通过 submit(Callable) 方法返回的 Future 对象进行获取。

以下是一个 ExecutorService Callable 示例：

```
//从 Executors 中获得 ExecutorService
ExecutorService executorService = Executors.newSingleThreadExecutor();

Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

System.out.println("future.get() = " + future.get());
executorService.shutdown();

输出：
Asynchronous Callable
future.get() = Callable Result
```

✓ **invokeAny()**

invokeAny() 方法要求一系列的 Callable 或者其子接口的实例对象。调用这个方法并不会返回一个 Future，但它返回其中一个 Callable 对象的结果。无法保证返回的是哪个 Callable 的结果 – 只能表明其中一个已执行结束。

如果其中一个任务执行结束(或者抛了一个异常)，其他 Callable 将被取消。以下是示例代码：

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 2";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 3";
    }
});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();
```

上述代码将会打印出给定 Callable 集合中的一个的执行结果。我自己试着执行了它几次，结果始终在变。有时是 “Task 1”，有时是 “Task 2”等等。

✓ **invokeAll()**

invokeAll() 方法将调用你在集合中传给 ExecutorService 的所有 Callable 对象。invokeAll() 返回一系列的 Future 对象，通过它们你可以获取每个 Callable 的执行结果。

记住，一个任务可能会由于一个异常而结束，因此它可能没有 "成功"。无法通过一个 Future 对象来告知我们是两种结束中的哪一种。

以下是一个代码示例：

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 2";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 3";
    }
});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){
    System.out.println("future.get = " + future.get());
}
executorService.shutdown();

输出结果：
future.get = Task 3
future.get = Task 1
future.get = Task 2
```

Executors 关闭:

使用 shutdown 和 shutdownNow 可以关闭线程池

两者的区别：

shutdown 只是将空闲的线程 interrupt() 了，shutdown （）之前提交的任务可以继续执行直到结束。

shutdownNow 是 interrupt 所有线程， 因此大部分线程将立刻被中断。之所以是大部分，而不是全部 ，是因为 interrupt()方法能力有限。

http://www.itheima.com Copyright© 2018 黑马程序员

➢ThreadPoolExecutor 线程池执行者

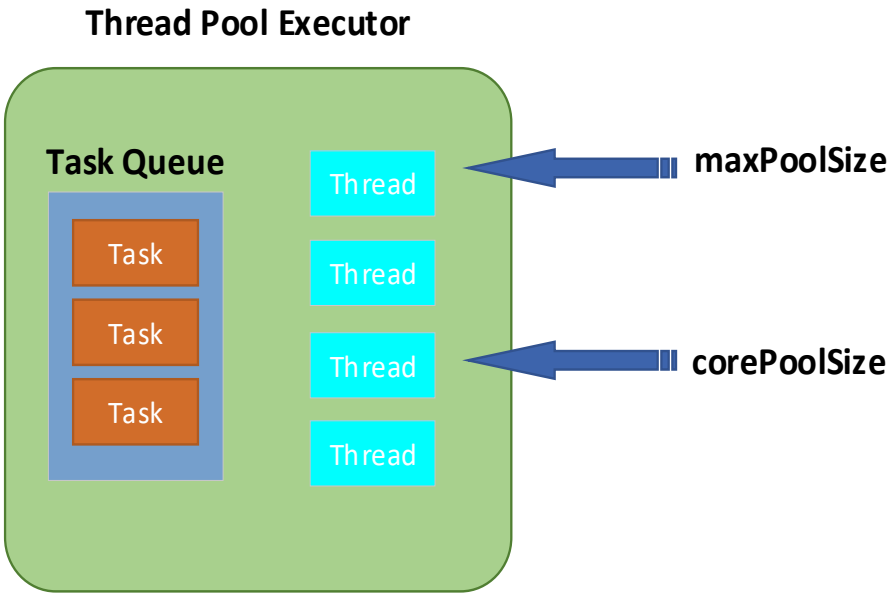
java.util.concurrent.ThreadPoolExecutor 是 ExecutorService 接口的一个实现。ThreadPoolExecutor 使用其内部池中的线程执行给定任务(Callable 或者 Runnable)。

ThreadPoolExecutor 包含的线程池能够包含不同数量的线程。池中线程的数量由以下变量决定:

- corePoolSize
- maximumPoolSize

当一个任务委托给线程池时，如果池中线程数量低于 corePoolSize，一个新的线程将被创建，即使池中可能尚有空闲线程。如果内部任务队列已满，而且有至少 corePoolSize 正在运行，但是运行线程的数量低于 maximumPoolSize，一个新的线程将被创建去执行该任务。

ThreadPoolExecutor 图解:



创建 ThreadPoolExecutor:

```
int corePoolSize = 5;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
int  maxPoolSize   =   10;
long keepAliveTime = 5000;

ExecutorService threadPoolExecutor =
        new ThreadPoolExecutor(
                corePoolSize,
                maxPoolSize,
                keepAliveTime,
                TimeUnit.MILLISECONDS,
                new LinkedBlockingQueue<Runnable>()
                );
```

构造方法参数列表解释：

corePoolSize - 池中所保存的线程数，包括空闲线程。

maximumPoolSize - 池中允许的最大线程数。

keepAliveTime - 当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。

unit - keepAliveTime 参数的时间单位。

workQueue - 执行前用于保持任务的队列。此队列仅保持由 execute 方法提交的 Runnable 任务。

➢ScheduledPoolExecutor 定时线程池执行者

java.util.concurrent.ScheduledExecutorService 是一个 ExecutorService，它能够将任务延后执行，或者间隔固定时间多次执行。 任务由一个工作者线程异步执行，而不是由提交任务给 ScheduledExecutorService 的那个线程执行。

ScheduledPoolExecutor 例子:

```
ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =
    scheduledExecutorService.schedule(new Callable() {
        public Object call() throws Exception {
            System.out.println("Executed!");
            return "Called!";
        }
    },
    5,
    TimeUnit.SECONDS);//5 秒后执行
```


http://www.itheima.com Copyright© 2018 黑马程序员

首先一个内置 5 个线程的 ScheduledExecutorService 被创建。之后一个 Callable 接口的匿名类示例被创建然后传递给 schedule() 方法。后边的俩参数定义了 Callable 将在 5 秒钟之后被执行。

ScheduledExecutorService 的实现:

ScheduledExecutorService 是一个接口，你要用它的话就得使用 java.util.concurrent 包里对它的某个实现类。ScheduledExecutorService 具有以下实现类：**ScheduledThreadPoolExecutor**

创建一个 ScheduledExecutorService:

如何创建一个 ScheduledExecutorService 取决于你采用的它的实现类。但是你也可以使用 Executors 工厂类来创建一个 ScheduledExecutorService 实例。比如：

```
ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(5);
```

ScheduledExecutorService 的使用:

一旦你创建了一个 ScheduledExecutorService，你可以通过调用它的以下方法：

- schedule (Callable task, long delay, TimeUnit timeunit)
- schedule (Runnable task, long delay, TimeUnit timeunit)
- scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)
- scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

下面我们就简单看一下这些方法。

✓ schedule (Callable task, long delay, TimeUnit timeunit)

```
ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =
    scheduledExecutorService.schedule(new Callable() {
```


 Copyright© 2018 黑马程序员

```
            public Object call() throws Exception {
                System.out.println("Executed!");
                return "Called!";
            }
        },
        5,
        TimeUnit.SECONDS);

    System.out.println("result = " + scheduledFuture.get());

    scheduledExecutorService.shutdown();

    输出结果:
    Executed!
    result = Called!
```

- ✓ schedule (Runnable task, long delay, TimeUnit timeunit)

这一方法规划一个任务将被定期执行。该任务将会在首个 initialDelay 之后得到执行，然后每个 period 时间之后重复执行。

如果给定任务的执行抛出了异常，该任务将不再执行。如果没有任何异常的话，这个任务将会持续循环执行到 ScheduledExecutorService 被关闭。

如果一个任务占用了比计划的时间间隔更长的时候，下一次执行将在当前执行结束执行才开始。计划任务在同一时间不会有多个线程同时执行。

- ✓ scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)

这一方法规划一个任务将被定期执行。该任务将会在首个 initialDelay 之后得到执行，然后每个 period 时间之后重复执行。

如果给定任务的执行抛出了异常，该任务将不再执行。如果没有任何异常的话，这个任务将会持续循环执行到 ScheduledExecutorService 被关闭。

如果一个任务占用了比计划的时间间隔更长的时候，下一次执行将在当前执行结束执行才开始。计划任务在同一时间不会有多个线程同时执行。

http://www.itheima.com Copyright© 2018 黑马程序员

✓ scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

除了 period 有不同的解释之外这个方法和 scheduleAtFixedRate() 非常像。scheduleAtFixedRate() 方法中，period 被解释为前一个执行的开始和下一个执行的开始之间的间隔时间。而在本方法中，period 则被解释为前一个执行的结束和下一个执行的结束之间的间隔。因此这个延迟是执行结束之间的间隔，而不是执行开始之间的间隔。

ScheduledExecutorService 的关闭:

正如 ExecutorService，在你使用结束之后你需要把 ScheduledExecutorService 关闭掉。否则他将导致 JVM 继续运行，即使所有其他线程已经全被关闭。

你可以使用从 ExecutorService 接口继承来的 shutdown() 或 shutdownNow() 方法将 ScheduledExecutorService 关闭。参见 ExecutorService 关闭部分以获取更多信息。

➢ForkJoinPool 合并和分叉（线程池）

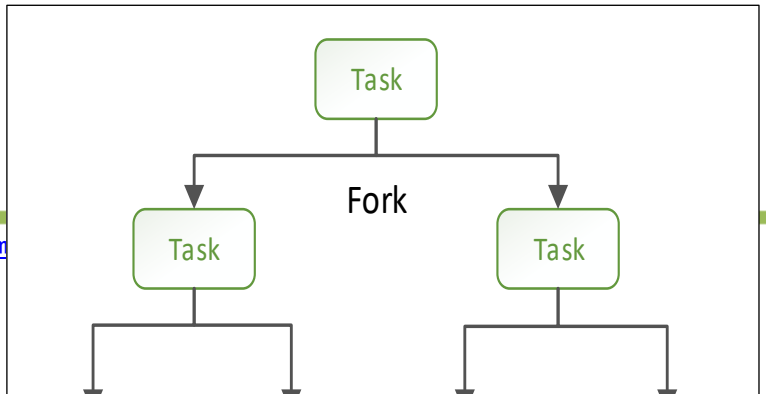
ForkJoinPool 在 Java 7 中被引入。它和 ExecutorService 很相似，除了一点不同。ForkJoinPool 让我们可以很方便地把任务分裂成几个更小的任务，这些分裂出来的任务也将会提交给 ForkJoinPool。任务可以继续分割成更小的子任务，只要它还能分割。可能听起来有些抽象，因此本节中我们将会解释 ForkJoinPool 是如何工作的，还有任务分割是如何进行的。

合并和分叉的解释:

在我们开始看 ForkJoinPool 之前我们先来简要解释一下分叉和合并的原理。分叉和合并原理包含两个递归进行的步骤。两个步骤分别是分叉步骤和合并步骤。

分叉:

一个使用了分叉和合并原理的任务可以将自己分叉(分割)为更小的子任务，这些子任务可以被并发执行。如下图所示:



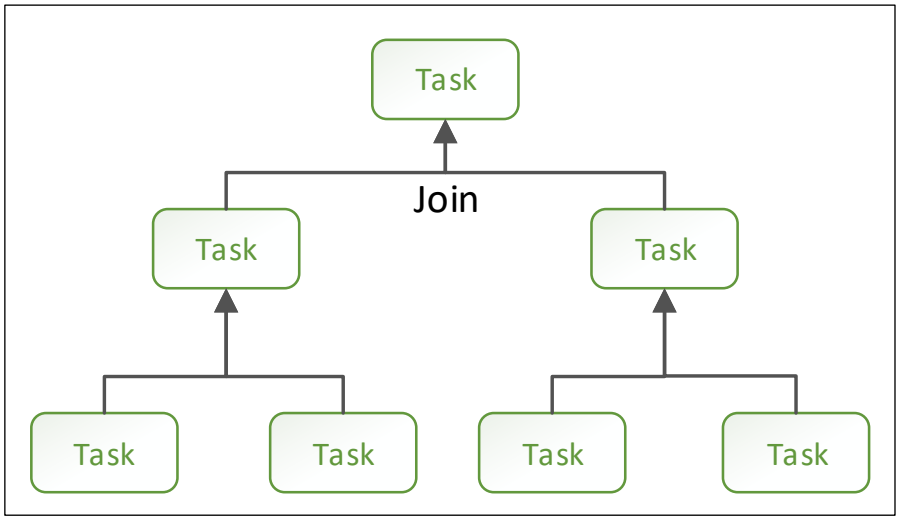
Copyright© 2018 黑马程序员

通过把自己分割成多个子任务，每个子任务可以由不同的 CPU 并行执行，或者被同一个 CPU 上的不同线程执行。只有当给的任务过大，把它分割成几个子任务才有意义。把任务分割成子任务有一定开销，因此对于小型任务，这个分割的消耗可能比每个子任务并发执行的消耗还要大。

什么时候把一个任务分割成子任务是有意义的，这个界限也称作一个阀值。这要看每个任务对有意义阀值的决定。很大程度上取决于它要做的工作的种类。

合并:

当一个任务将自己分割成若干子任务之后，该任务将进入等待所有子任务的结束之中。一旦子任务执行结束，该任务可以把所有结果合并到同一个结果。图示如下:



当然，并非所有类型的任务都会返回一个结果。如果这个任务并不返回一个结果，它只需等待所有子任务执行完毕。也就不需要结果的合并啦。

http://www.itheima.com Copyright© 2018 黑马程序员

所以我们可以将 ForkJoinPool 是一个特殊的线程池，它的设计是为了更好的配合 分叉-和-合并 任务分割的工作。ForkJoinPool 也在 java.util.concurrent 包中，其完整类名为 java.util.concurrent.ForkJoinPool。

创建一个 ForkJoinPool:

你可以通过其构造子创建一个 ForkJoinPool。作为传递给 ForkJoinPool 构造子的一个参数，你可以定义你期望的并行级别。并行级别表示你想要传递给 ForkJoinPool 的任务所需的线程或 CPU 数量。以下是一个 ForkJoinPool 示例：

```
//创建了一个并行级别为 4 的 ForkJoinPool
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
```

提交任务到 ForkJoinPool:

就像提交任务到 ExecutorService 那样，把任务提交到 ForkJoinPool。你可以提交两种类型的任务。一种是没有任何返回值的(一个 “行动”)，另一种是有返回值的(一个”任务”)。这两种类型分别由 RecursiveAction 和 RecursiveTask 表示。接下来介绍如何使用这两种类型的任务，以及如何对它们进行提交。

RecursiveAction:

RecursiveAction 是一种没有任何返回值的任务。它只是做一些工作，比如写数据到磁盘，然后就退出了。一个 RecursiveAction 可以把自己的工作分割成更小的几块，这样它们可以由独立的线程或者 CPU 执行。你可以通过继承来实现一个 RecursiveAction。示例如下：

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveAction;

public class MyRecursiveAction extends RecursiveAction {

    private long workLoad = 0;

    public MyRecursiveAction(long workLoad) {
        this.workLoad = workLoad;
    }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
    @Override
    protected void compute() {

        //if work is above threshold, break tasks up into smaller tasks
        //翻译：如果工作超过门槛，把任务分解成更小的任务
        if(this.workLoad > 16) {
            System.out.println("Splitting workLoad : " + this.workLoad);

            List<MyRecursiveAction> subtasks =
                new ArrayList<MyRecursiveAction>();

            subtasks.addAll(createSubtasks());

            for(RecursiveAction subtask : subtasks){
                subtask.fork();
            }

        } else {
            System.out.println("Doing workLoad myself: " + this.workLoad);
        }
    }

    private List<MyRecursiveAction> createSubtasks() {
        List<MyRecursiveAction> subtasks =
            new ArrayList<MyRecursiveAction>();

        MyRecursiveAction subtask1 = new MyRecursiveAction(this.workLoad / 2);
        MyRecursiveAction subtask2 = new MyRecursiveAction(this.workLoad / 2);

        subtasks.add(subtask1);
        subtasks.add(subtask2);

        return subtasks;
    }

}
```

例子很简单。MyRecursiveAction 将一个虚构的 workLoad 作为参数传给自己的构造子。如果 workLoad 高于一个特定阀值，该工作将被分割为几个子工作，子工作继续分割。如果 workLoad 低于特定阀值，该工作将由 MyRecursiveAction 自己执行。你可以这样规划一个 MyRecursiveAction 的执行：

http://www.itheima.com Copyright© 2018 黑马程序员

```
//创建了一个并行级别为 4 的 ForkJoinPool
ForkJoinPool forkJoinPool = new ForkJoinPool(4);

//创建一个没有返回值的任务
MyRecursiveAction myRecursiveAction = new MyRecursiveAction(24);

//ForkJoinPool 执行任务
forkJoinPool.invoke(myRecursiveAction);

运行结果：
Splitting workLoad : 24
Doing workLoad myself: 12
```

RecursiveTask:

RecursiveTask 是一种会返回结果的任务。它可以将自己的工作分割为若干更小任务，并将这些子任务的执行结果合并到一个集体结果。可以有几个水平的分割和合并。以下是一个 RecursiveTask 示例：

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveTask;


public class MyRecursiveTask extends RecursiveTask<Long> {

    private long workLoad = 0;

    public MyRecursiveTask(long workLoad) {
        this.workLoad = workLoad;
    }

    protected Long compute() {

        //if work is above threshold, break tasks up into smaller tasks
        if(this.workLoad > 16) {
            System.out.println("Splitting workLoad : " + this.workLoad);

            List<MyRecursiveTask> subtasks =
                new ArrayList<MyRecursiveTask>();
            subtasks.addAll(createSubtasks());

            for(MyRecursiveTask subtask : subtasks){
                subtask.fork();
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            }

            long result = 0;
            for(MyRecursiveTask subtask : subtasks) {
                result += subtask.join();
            }
            return result;

        } else {
            System.out.println("Doing workLoad myself: " + this.workLoad);
            return workLoad * 3;
        }
    }

    private List<MyRecursiveTask> createSubtasks() {
        List<MyRecursiveTask> subtasks =
        new ArrayList<MyRecursiveTask>();

        MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad / 2);
        MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad / 2);

        subtasks.add(subtask1);
        subtasks.add(subtask2);

        return subtasks;
    }
}
```

除了有一个结果返回之外，这个示例和 RecursiveAction 的例子很像。MyRecursiveTask 类继承自 RecursiveTask<Long>，这也就意味着它将返回一个 Long 类型的结果。

MyRecursiveTask 示例也会将工作分割为子任务，并通过 fork() 方法对这些子任务计划执行。此外，本示例还通过调用每个子任务的 join() 方法收集它们返回的结果。子任务的结果随后被合并到一个更大的结果，并最终将其返回。对于不同级别的递归，这种子任务的结果合并可能会发生递归。

你可以这样规划一个 RecursiveTask：

```
//创建了一个并行级别为 4 的 ForkJoinPool
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
//创建一个有返回值的任务
MyRecursiveTask myRecursiveTask = new MyRecursiveTask(128);
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
//线程池执行并返回结果
long mergedResult = forkJoinPool.invoke(myRecursiveTask);

System.out.println("mergedResult = " + mergedResult);
```

注意: ForkJoinPool.invoke() 方法的调用来获取最终执行结果的。

B. 并发队列-阻塞队列

常用的并发队列有阻塞队列和非阻塞队列，前者使用锁实现，后者则使用 CAS 非阻塞算法实现。

PS：至于非阻塞队列是靠 CAS 非阻塞算法，在这里不再介绍，大家只用知道，Java 非阻塞队列是使用 CAS 算法来实现的就可以。感兴趣的童鞋可以维基网上自行学习.

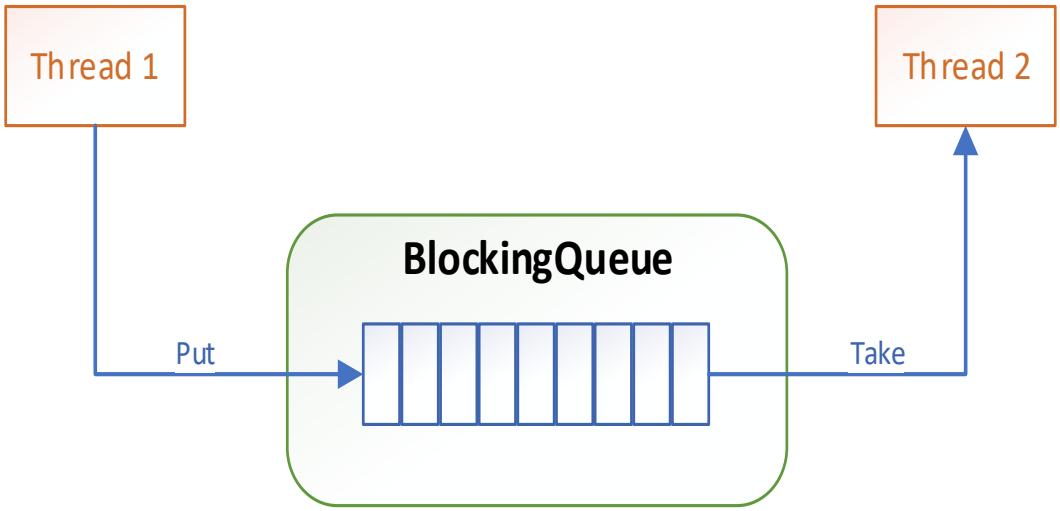
下面我们先介绍阻塞队列。

阻塞队列:

阻塞队列（BlockingQueue)是 Java util.concurrent 包下重要的数据结构，BlockingQueue 提供了线程安全的队列访问方式：当阻塞队列进行插入数据时，如果队列已满，线程将会阻塞等待直到队列非满；从阻塞队列取数据时，如果队列已空，线程将会阻塞等待直到队列非空。并发包下很多高级同步类的实现都是基于 BlockingQueue 实现的。

➢BlockingQueue 阻塞队列

BlockingQueue 通常用于一个线程生产对象，而另外一个线程消费这些对象的场景。下图是对这个原理的阐述：



Copyright© 2018 黑马程序员

一个线程往里边放，另外一个线程从里边取的一个 BlockingQueue。

一个线程将会持续生产新对象并将其插入到队列之中，直到队列达到它所能容纳的临界点。也就是说，它是有限的。如果该阻塞队列到达了其临界点，负责生产的线程将会在往里边插入新对象时发生阻塞。它会一直处于阻塞之中，直到负责消费的线程从队列中拿走一个对象。负责消费的线程将会一直从该阻塞队列中拿出对象。如果消费线程尝试去从一个空的队列中提取对象的话，这个消费线程将会处于阻塞之中，直到一个生产线程把一个对象丢进队列。

BlockingQueue 的方法:

BlockingQueue 具有 4 组不同的方法用于插入、移除以及对队列中的元素进行检查。如果请求的操作不能得到立即执行的话，每个方法的表现也不同。这些方法如下：

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

四组不同的行为方式解释:

抛异常：如果试图的操作无法立即执行，抛一个异常。

特定值：如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。

阻塞：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。

超时：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是 true / false)。

http://www.itheima.com Copyright© 2018 黑马程序员

无法向一个 BlockingQueue 中插入 null。如果你试图插入 null，BlockingQueue 将会抛出一个 NullPointerException.

BlockingQueue 的实现类:

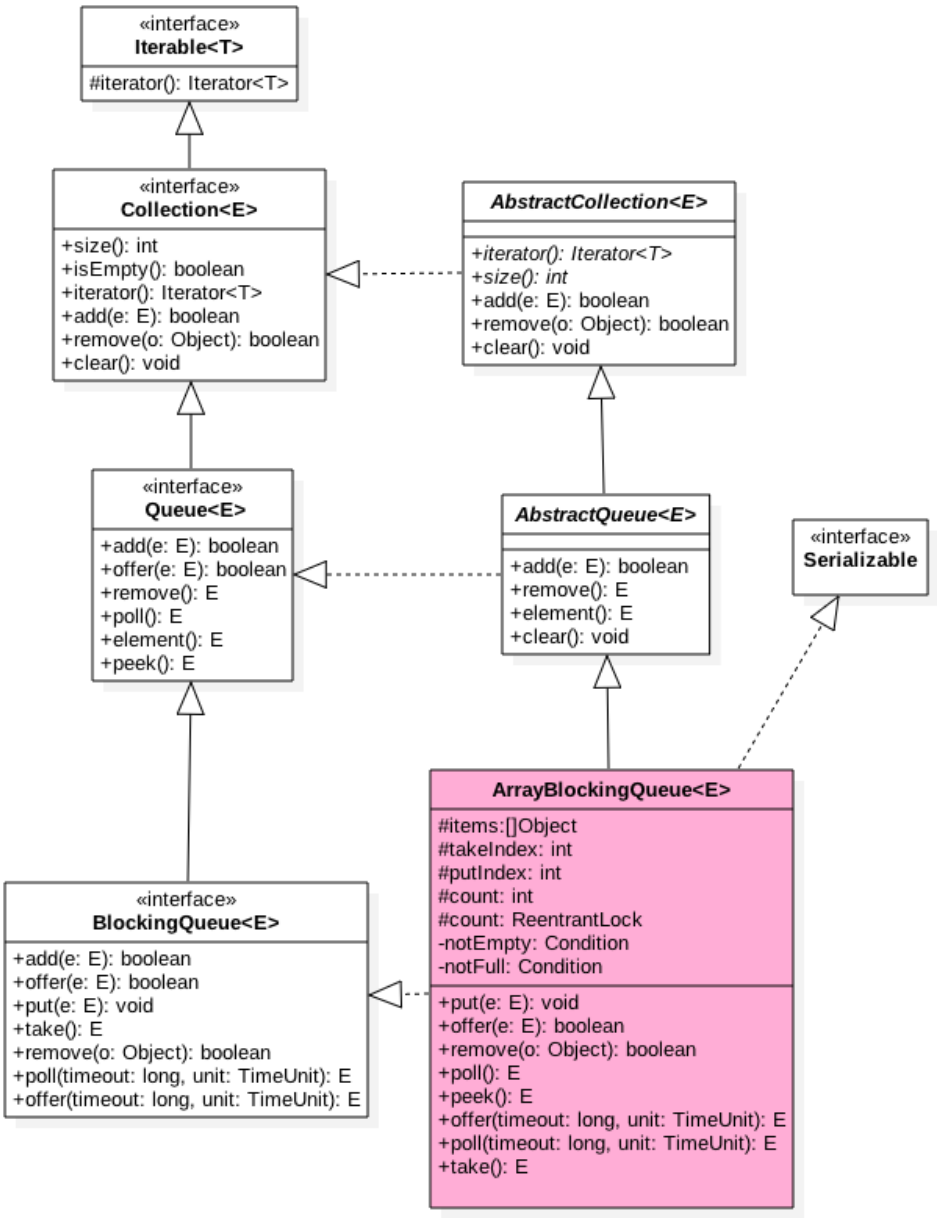
BlockingQueue 是个接口，你需要使用它的实现之一来使用 BlockingQueue，Java.util.concurrent 包下具有以下 BlockingQueue 接口的实现类：

- ArrayBlockingQueue：ArrayBlockingQueue 是一个有界的阻塞队列，其内部实现是将对象放到一个数组里。有界也就意味着，它不能够存储无限多数量的元素。它有一个同一时间能够存储元素数量的上限。你可以在对其初始化的时候设定这个上限，但之后就无法对这个上限进行修改了(译者注：因为它是基于数组实现的，也就具有数组的特性：一旦初始化，大小就无法修改)。
- DelayQueue：DelayQueue 对元素进行持有直到一个特定的延迟到期。注入其中的元素必须实现 java.util.concurrent.Delayed 接口。
- LinkedBlockingQueue：LinkedBlockingQueue 内部以一个链式结构(链接节点)对其元素进行存储。如果需要的话，这一链式结构可以选择一个上限。如果没有定义上限，将使用 Integer.MAX_VALUE 作为上限。
- PriorityBlockingQueue：PriorityBlockingQueue 是一个无界的并发队列。它使用了和类 java.util.PriorityQueue 一样的排序规则。你无法向这个队列中插入 null 值。所有插入到 PriorityBlockingQueue 的元素必须实现 java.lang.Comparable 接口。因此该队列中元素的排序就取决于你自己的 Comparable 实现。
- SynchronousQueue：SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。据此，把这个类称作一个队列显然是夸大其词了。它更多像是一个汇合点。

http://www.itheima.com Copyright© 2018 黑马程序员

➢ArrayBlockingQueue 阻塞队列

ArrayBlockingQueue 类图



如上图 ArrayBlockingQueue 内部有个数组 items 用来存放队列元素，putindex 下标标示入队元素下标，takeIndex 是出队下标，count 统计队列元素个数，从定义可知道并没有使用 volatile 修饰，这是因为访问这些变量使

 Copyright© 2018 黑马程序员

用都是在锁块内，并不存在可见性问题。另外有个独占锁 lock 用来对出入队操作加锁，这导致同时只有一个线程可以访问入队出队，另外 notEmpty，notFull 条件变量用来进行出入队的同步。

另外构造函数必须传入队列大小参数，所以为有界队列，默认是 Lock 为非公平锁。

```
public ArrayBlockingQueue(int capacity) {
        this(capacity, false);
  }

public ArrayBlockingQueue(int capacity, boolean fair) {
        if (capacity <= 0)
            throw new IllegalArgumentException();
        this.items = new Object[capacity];
        lock = new ReentrantLock(fair);
        notEmpty = lock.newCondition();
        notFull =  lock.newCondition();
}
```

ps:

所谓公平锁: 就是在并发环境中，每个线程在获取锁时会先查看此锁维护的等待队列，如果为空，或者当前线程线程是等待队列的第一个，就占有锁，否则就会加入到等待队列中，以后会按照 FIFO 的规则从队列中取到自己。

非公平锁: 比较粗鲁，上来就直接尝试占有锁，如果尝试失败，就再采用类似公平锁那种方式

ArrayBlockingQueue 方法

- ✓ **offer 方法**

在队尾插入元素，如果队列满则返回 false，否者入队返回 true。

```
public boolean offer(E e) {

    //e 为 null，则抛出 NullPointerException 异常
    checkNotNull(e);

    //获取独占锁
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            //如果队列满则返回 false
            if (count == items.length)
                return false;
            else {
                //否者插入元素
                insert(e);
                return true;
            }
        } finally {
            //释放锁
            lock.unlock();
        }
    }


    private void insert(E x) {

        //元素入队
        items[putIndex] = x;

        //计算下一个元素应该存放的下标
        putIndex = inc(putIndex);
        ++count;
        notEmpty.signal();
    }

    //循环队列，计算下标
    final int inc(int i) {
        return (++i == items.length) ? 0 : i;
    }
```

这里由于在操作共享变量前加了锁，所以不存在内存不可见问题，加过锁后获取的共享变量都是从主内存获取的，而不是在 CPU 缓存或者寄存器里面的值，释放锁后修改的共享变量值会刷新会主内存中。

另外这个队列是使用循环数组实现，所以计算下一个元素存放下标时候有些特殊。另外 insert 后调用 notEmpty.signal();是为了激活调用 notEmpty.await()阻塞后放入 notEmpty 条件队列中的线程。

- ✓ **Put 操作**

在队列尾部添加元素，如果队列满则等待队列有空位置插入后返回。

```
    public void put(E e) throws InterruptedException {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        checkNotNull(e);
        final ReentrantLock lock = this.lock;

        //获取可被中断锁
        lock.lockInterruptibly();
        try {

            //如果队列满，则把当前线程放入 notFull 管理的条件队列
            while (count == items.length)
                notFull.await();

            //插入元素
            insert(e);
        } finally {
            lock.unlock();
        }
    }
```

需要注意的是如果队列满了那么当前线程会阻塞，知道出队操作调用了 notFull.signal 方法激活该线程。代码逻辑很简单，但是这里需要思考一个问题为啥调用 lockInterruptibly 方法而不是 Lock 方法。我的理解是因为调用了条件变量的 await()方法，而 await()方法会在中断标志设置后抛出 InterruptedException 异常后退出，所以还不如在加锁时候先看中断标志是不是被设置了，如果设置了直接抛出 InterruptedException 异常，就不用再去获取锁了。然后看了其他并发类里面凡是调用了 await 的方法获取锁时候都是使用的 lockInterruptibly 方法而不是 Lock 也验证了这个想法。

- ✓ **Poll 操作**

从队头获取并移除元素，队列为空，则返回 null。

```
    public E poll() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            //当前队列为空则返回 null,否者
            return (count == 0) ? null : extract();
        } finally {
            lock.unlock();
        }
    }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
private E extract() {
    final Object[] items = this.items;

    //获取元素值
    E x = this.<E>cast(items[takeIndex]);

    //数组中值值为 null;
    items[takeIndex] = null;

    //队头指针计算，队列元素个数减一
    takeIndex = inc(takeIndex);
    --count;

    //发送信号激活 notFull 条件队列里面的线程
    notFull.signal();
    return x;
}
```

✓ **Take 操作**

从队头获取元素，如果队列为空则阻塞直到队列有元素。

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {

        //队列为空，则等待，直到队列有元素
        while (count == 0)
            notEmpty.await();
        return extract();
    } finally {
        lock.unlock();
    }
}
```

需要注意的是如果队列为空，当前线程会被挂起放到 notEmpty 的条件队列里面，直到入队操作执行调用 notEmpty.signal 后当前线程才会被激活，await 才会返回。

✓ **Peek 操作**

返回队列头元素但不移除该元素，队列为空，返回 null。

http://www.itheima.com Copyright© 2018 黑马程序员

```
public E peek() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        //队列为空返回 null,否者返回头元素
        return (count == 0) ? null : itemAt(takeIndex);
    } finally {
        lock.unlock();
    }
}

final E itemAt(int i) {
    return this.<E>cast(items[i]);
}
```

✓ **Size 操作**

获取队列元素个数，非常精确因为计算 size 时候加了独占锁，其他线程不能入队或者出队或者删除元素。

```
public int size() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return count;
    } finally {
        lock.unlock();
    }
}
```

✓ **ArrayBlockingQueue 小结**

ArrayBlockingQueue 通过使用全局独占锁实现同时只能有一个线程进行入队或者出队操作，这个锁的粒度比较大，有点类似在方法上添加 synchronized 的意味。其中 offer,poll 操作通过简单的加锁进行入队出队操作，而 put,take 则使用了条件变量实现如果队列满则等待，如果队列空则等待，然后分别在出队和入队操作中发送信号激活等待线程实现同步。另外相比 LinkedBlockingQueue，ArrayBlockingQueue 的 size 操作的结果是精确的，因为计算前加了全局锁。

ArrayBlockingQueue 示例

需求：在多线程操作下，一个数组中最多只能存入 3 个元素。多放入不可以存入数组，或等待某线程对数组中某

 Copyright© 2018 黑马程序员

个元素取走才能放入，要求使用 java 的多线程来实现。（面试）

代码实现：

```
public class BlockingQueueTest {
    public static void main(String[] args) {
        final BlockingQueue queue = new ArrayBlockingQueue(3);
        for(int i=0;i<2;i++){
            new Thread(){
                public void run(){
                    while(true){
                        try {
                            Thread.sleep((long)(Math.random()*1000));
                            System.out.println(Thread.currentThread().getName() + "准备放数据!");

                            queue.put(1);
                            System.out.println(Thread.currentThread().getName() + "已经放了数据，" +
                                    "队列目前有" + queue.size() + "个数据");
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }

                    }
                }

            }.start();
        }

        new Thread(){
            public void run(){
                while(true){
                    try {
                        //将此处的睡眠时间分别改为 100 和 1000，观察运行结果
                        Thread.sleep(100);
                        System.out.println(Thread.currentThread().getName() + "准备取数据!");
                        System.err.println(queue.take());
                        System.out.println(Thread.currentThread().getName() + "已经取走数据，" +

                                "队列目前有" + queue.size() + "个数据");
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
```


http://www.itheima.com Copyright© 2018 黑马程序员

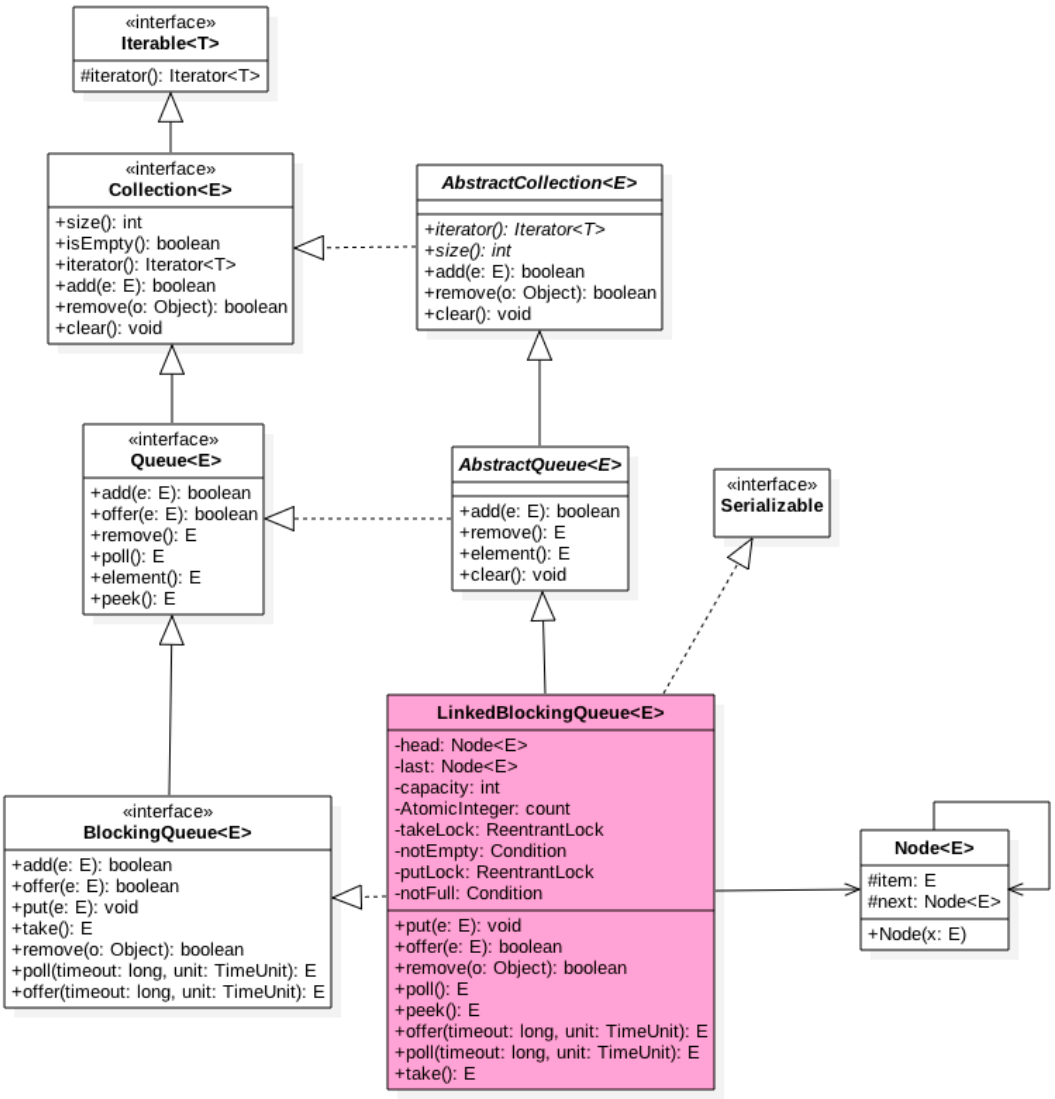
```
                }

            }.start();
        }
    }

    输出结果：
    Thread-0 准备放数据！
    Thread-0 已经放了数据，队列目前有 1 个数据
    Thread-0 准备放数据！
    Thread-0 已经放了数据，队列目前有 2 个数据
    Thread-1 准备放数据！
    Thread-1 已经放了数据，队列目前有 3 个数据
    Thread-2 准备取数据！
    Thread-2 已经取走数据，队列目前有 3 个数据
    Thread-0 准备放数据！
    Thread-1 准备放数据！
    Thread-2 准备取数据！
    Thread-2 已经取走数据，队列目前有 3 个数据
    ……
```

➢LinkedBlockingQueue 阻塞队列

LinkedBlockingQueue 类图

LinkedBlockingQueue 中也有两个 Node 分别用来存放首尾节点，并且里面有个初始值为 0 的原子变量 count 用来记录队列元素个数，另外里面有两个 ReentrantLock 的独占锁，分别用来控制元素入队和出队加锁，其中 takeLock 用来控制同时只有一个线程可以从队列获取元素，其他线程必须等待，putLock 控制同时只能有一个线程可以获取锁去添加元素，其他线程必须等待。另外 notEmpty 和 notFull 用来实现入队和出队的同步。 另外由于出入队是两个非公平独占锁，所以可以同时又一个线程入队和一个线程出队，其实这个是个生产者-消费者模型，如下类图：

http://www.itheima.com Copyright© 2018 黑马程序员




```
/** 通过 take 取出进行加锁、取出 */
private final ReentrantLock takeLock = new ReentrantLock();

/** 等待中的队列等待取出 */
private final Condition notEmpty = takeLock.newCondition();

/*通过 put 放置进行加锁、放置*/
private final ReentrantLock putLock = new ReentrantLock();

/** 等待中的队列等待放置 */
private final Condition notFull = putLock.newCondition();

/* 记录集合中的个数（计数器） */
private final AtomicInteger count = new AtomicInteger(0);
```


 Copyright© 2018 黑马程序员

队列的容量：

```
//队列初始容量，Integer 最大值
public static final int   MAX_VALUE = 0x7fffffff;

public LinkedBlockingQueue() {
    this(Integer.MAX_VALUE);
}

  public LinkedBlockingQueue(int capacity) {
    if (capacity <= 0) throw new IllegalArgumentException();
    this.capacity = capacity;
    //初始化首尾节点
    last = head = new Node<E>(null);
}
```

如图默认队列容量为 0x7fffffff;用户也可以自己指定容量。

LinkedBlockingQueue 方法

ps：下面介绍 LinkedBlockingQueue 用到很多 Lock 对象。详细可以查找 Lock 对象的介绍

✓ **带时间的 Offer 操作-生产者**

在 ArrayBlockingQueue 中已经简单介绍了 Offer()方法，LinkedBlocking 的 Offer 方法类似，在此就不过多去介绍。这次我们从介绍下带时间的 Offer 方法

```
public boolean offer(E e, long timeout, TimeUnit unit)
    throws InterruptedException {

    //空元素抛空指针异常
    if (e == null) throw new NullPointerException();
    long nanos = unit.toNanos(timeout);
    int c = -1;
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;

    //获取可被中断锁，只有一个线程克获取
    putLock.lockInterruptibly();
    try {

        //如果队列满则进入循环
        while (count.get() == capacity) {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
                //nanos<=0 直接返回
                if (nanos <= 0)
                    return false;
                //否者调用 await 进行等待，超时则返回<=0（1）
                nanos = notFull.awaitNanos(nanos);
            }
            //await 在超时时间内返回则添加元素（2）
            enqueue(new Node<E>(e));
            c = count.getAndIncrement();

            //队列不满则激活其他等待入队线程（3）
            if (c + 1 < capacity)
                notFull.signal();
        } finally {
            //释放锁
            putLock.unlock();
        }

        //c==0 说明队列里面有一个元素，这时候唤醒出队线程（4）
        if (c == 0)
            signalNotEmpty();
        return true;
    }

    private void enqueue(Node<E> node) {
        last = last.next = node;
    }

        private void signalNotEmpty() {
            final ReentrantLock takeLock = this.takeLock;
            takeLock.lock();
            try {
                notEmpty.signal();
            } finally {
                takeLock.unlock();
            }
        }
```

- ✓ **带时间的 poll 操作-消费者**

获取并移除队首元素，在指定的时间内去轮询队列看有没有首元素有则返回，否者超时后返回 null。

```
    public E poll(long timeout, TimeUnit unit) throws InterruptedException {
        E x = null;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        int c = -1;
        long nanos = unit.toNanos(timeout);
        final AtomicInteger count = this.count;
        final ReentrantLock takeLock = this.takeLock;

        //出队线程获取独占锁
        takeLock.lockInterruptibly();
        try {

            //循环直到队列不为空
            while (count.get() == 0) {

                //超时直接返回 null
                if (nanos <= 0)
                    return null;
                nanos = notEmpty.awaitNanos(nanos);
            }

            //出队，计数器减一
            x = dequeue();
            c = count.getAndDecrement();

            //如果出队前队列不为空则发送信号，激活其他阻塞的出队线程
            if (c > 1)
                notEmpty.signal();
        } finally {
            //释放锁
            takeLock.unlock();
        }

        //当前队列容量为最大值-1 则激活入队线程。
        if (c == capacity)
            signalNotFull();
        return x;
    }
```

首先获取独占锁，然后进入循环当当前队列有元素才会退出循环，或者超时了，直接返回 null。

超时前退出循环后，就从队列移除元素，然后计数器减去一，如果减去 1 前队列元素大于 1 则说明当前移除后队列还有元素，那么就发信号激活其他可能阻塞到当前条件信号的线程。

最后如果减去 1 前队列元素个数=最大值，那么移除一个后会腾出一个空间来，这时候可以激活可能存在的入队

http://www.itheima.com Copyright© 2018 黑马程序员

阻塞线程。

- ✓ **put 操作-生产者**

与带超时时间的 poll 类似不同在于 put 时候如果当前队列满了它会一直等待其他线程调用 notFull.signal 才会被唤醒。

- ✓ **take 操作-消费者**

与带超时时间的 poll 类似不同在于 take 时候如果当前队列空了它会一直等待其他线程调用 notEmpty.signal()才会被唤醒。

- ✓ **size 操作-消费者**

当前队列元素个数，如代码直接使用原子变量 count 获取。

```
public int size() {
    return count.get();
}
```

- ✓ **peek 操作**

获取但是不移除当前队列的头元素，没有则返回 null。

```
public E peek() {
        //队列空，则返回 null
        if (count.get() == 0)
            return null;
        final ReentrantLock takeLock = this.takeLock;
        takeLock.lock();
        try {
            Node<E> first = head.next;
            if (first == null)
                return null;
            else
                return first.item;
        } finally {
            takeLock.unlock();
        }
}
```

- ✓ **remove 操作**

http://www.itheima.com Copyright© 2018 黑马程序员

删除队列里面的一个元素，有则删除返回 true，没有则返回 false，在删除操作时候由于要遍历队列所以加了双重锁，也就是在删除过程中不允许入队也不允许出队操作。

```
public boolean remove(Object o) {
    if (o == null) return false;

    //双重加锁
    fullyLock();
    try {

        //遍历队列找则删除返回 true
        for (Node<E> trail = head, p = trail.next;
             p != null;
             trail = p, p = p.next) {
            if (o.equals(p.item)) {
                unlink(p, trail);
                return true;
            }
        }
        //找不到返回 false
        return false;
    } finally {
        //解锁
        fullyUnlock();
    }
}

void fullyLock() {
    putLock.lock();
    takeLock.lock();
}

void fullyUnlock() {
    takeLock.unlock();
    putLock.unlock();
}

void unlink(Node<E> p, Node<E> trail) {

    p.item = null;
    trail.next = p.next;
    if (last == p)
```

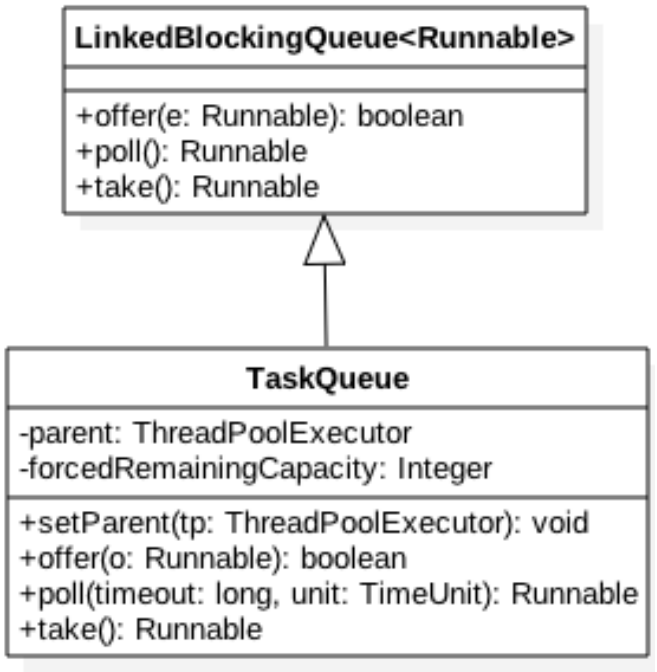

http://www.itheima.com Copyright© 2018 黑马程序员

```
            last = trail;
        //如果当前队列满，删除后，也不忘记最快的唤醒等待的线程
        if (count.getAndDecrement() == capacity)
            notFull.signal();
    }
```

✓ **开源框架的使用**

tomcat 中任务队列 TaskQueue。

类结构图：



可知 TaskQueue 继承了 LinkedBlockingQueue 并且泛化类型固定了为 Runnalbe.重写了 offer,poll,take 方法。tomcat 中有个线程池 ThreadPoolExecutor，在 NIOEndPoint 中当 acceptor 线程接受到请求后，会把任务放入队列，然后 poller 线程从队列里面获取任务，然后就把任务放入线程池执行。这个 ThreadPoolExecutor 中的的一个参数就是 TaskQueue。

先看看 ThreadPoolExecutor 的参数如果是普通 LinkedBlockingQueue 是怎么样的执行逻辑：

当调用线程池方法 execute() 方法添加一个任务时：

http://www.itheima.com Copyright© 2018 黑马程序员

- 如果当前运行的线程数量小于 corePoolSize，则创建新线程运行该任务
- 如果当前运行的线程数量大于或等于 corePoolSize，则将这个任务放入阻塞队列。
- 如果当前队列满了，并且当前运行的线程数量小于 maximumPoolSize，则创建新线程运行该任务；
- 如果当前队列满了，并且当前运行的线程数量大于或等于 maximumPoolSize，那么线程池将会抛出 RejectedExecutionException 异常。

如果线程执行完了当前任务，那么会去队列里面获取一个任务来执行，如果任务执行完了，并且当前线程数大于 corePoolSize，那么会根据线程空闲时间 keepAliveTime 回收一些线程保持线程池 corePoolSize 个线程。

首先看下线程池中 exectue 添加任务时候的逻辑：

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();

    //当前工作线程个数小于 core 个数则开新线程执行（1）
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    //放入队列（2）
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }

    //如果队列满了则开新线程，但是个数要不超过最大值，超过则返回 false
    //然后执行 reject handler（3）
    else if (!addWorker(command, false))
        reject(command);
}
```

可知当当前工作线程个数为 corePoolSize 后，如果在来任务会把任务添加到队列，队列满了或者入队失败了则开

http://www.itheima.com Copyright© 2018 黑马程序员

启新线程。

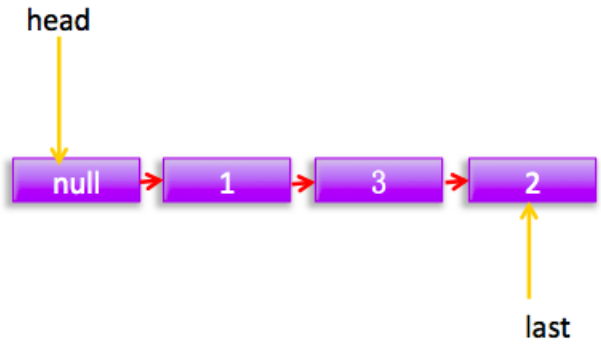
然后看看 TaskQueue 中重写的 offer 方法的逻辑：

```
public boolean offer(Runnable o) {
    // 如果 parent 为 null 则直接调用父类方法
    if (parent==null) return super.offer(o);
    //如果当前线程池中线程个数达到最大，则无条件调用父类方法
    if (parent.getPoolSize() == parent.getMaximumPoolSize()) return super.offer(o);
    //如果当前提交的任务小于当前线程池线程数，说明线程用不完，没必要重新开线程
    if (parent.getSubmittedCount()<(parent.getPoolSize())) return super.offer(o);
    //如果当前线程池线程个数>core 个数但是小于最大个数，则开新线程代替放入队列
    if (parent.getPoolSize()<parent.getMaximumPoolSize()) return false;
    //到了这里，无条件调用父类
    return super.offer(o);
}
```

可知 parent.getPoolSize()<parent.getMaximumPoolSize()普通队列会把当前任务放入队列，TaskQueue 则是返回 false，因为这会开启新线程执行任务，当然前提是当前线程个数没有达到最大值。

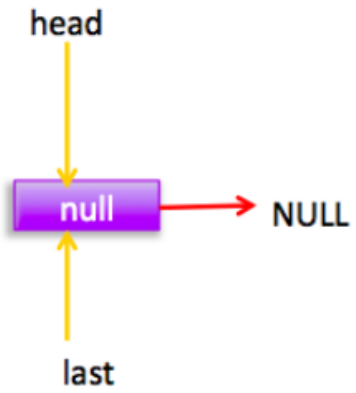
LinkedBlockingQueue 安全分析总结

仔细思考下阻塞队列是如何实现并发安全的维护队列链表的，先分析下简单的情况就是当队列里面有多个元素时候，由于同时只有一个线程（通过独占锁 putLock 实现）入队元素并且是操作 last 节点（，而同时只有一个出队线程(通过独占锁 takeLock 实现）操作 head 节点，所以不存在并发安全问题。



考虑当队列为空的时候队列状态为：

http://www.itheima.com Copyright© 2018 黑马程序员



这时候假如一个线程调用了 take 方法,由于队列为空，所以 count.get()==0 所以当前线程会调用 notEmpty.await()把自己挂起，并且放入 notEmpty 的条件队列，并且释放当前条件变量关联的通过 takeLock.lockInterruptibly()获取的独占锁。由于释放了锁，所以这时候其他线程调用 take 时候就会通过 takeLock.lockInterruptibly()获取独占锁,然后同样阻塞到 notEmpty.await(),同样会被放入 notEmpty 的条件队列，也就说在队列为空的情况下可能会有多个线程因为调用 take 被放入了 notEmpty 的条件队列。

这时候如果有一个线程调用了 put 方法，那么就会调用 enqueue 操作，该操作会在 last 节点后面添加新元素并且设置 last 为新节点。然后 count.getAndIncrement()先获取当前队列元个数为 0 保存到 c，然后自增 count 为 1，由于 c==0 所以调用 signalNotEmpty 激活 notEmpty 的条件队列里面的阻塞时间最长的线程，这时候 take 中调用 notEmpty.await()的线程会被激活 await 内部会重新去获取独占锁获取成功则返回，否者被放入 AQS 的阻塞队列，如果获取成功，那么 count.get() >0 因为可能多个线程 put 了，所以调用 dequeue 从队列获取元素（这时候一定可以获取到），然后调用 c = count.getAndDecrement() 把当前计数返回后并减去 1，如果 c>1 说明当前队列还有其他元素，那么就调用 notEmpty.signal()去激活 notEmpty 的条件队列里面的其他阻塞线程。

考虑当队列满的时候:

http://www.itheima.com Copyright© 2018 黑马程序员

当队列满的时候调用 put 方法时候，会由于 notFull.await()当前线程被阻塞放入 notFull 管理的条件队列里面，同理可能会有多个调用 put 方法的线程都放到了 notFull 的条件队列里面。

这时候如果有一个线程调用了 take 方法,调用 dequeue()出队一个元素，c = count.getAndDecrement()；count 值减一；c==capacity;现在队列有一个空的位置，所以调用 signalNotFull()激活 notFull 条件队列里面等待最久的一个线程。

LinkedBlockingQueue 简单示例

并发库中的 BlockingQueue 是一个比较好玩的类，顾名思义，就是阻塞队列。该类主要提供了两个方法 put()和 take()，前者将一个对象放到队列中，如果队列已经满了，就等待直到有空闲节点；后者从 head 取一个对象，如果没有对象，就等待直到有可取的对象。

下面的例子比较简单，一个读线程，用于将要处理的文件对象添加到阻塞队列中， 另外四个写线程用于取出文件对象，为了模拟写操作耗时长的特点，特让线程睡眠一段随机长度的时间。另外，该 Demo 也使用到了线程池和原子整型 （AtomicInteger），AtomicInteger 可以在并发情况下达到原子化更新，避免使用了 synchronized，而且性能非常高。由 于阻塞队列的 put 和 take 操作会阻塞，为了使线程退出，特在队列中添加了一个“标识”，算法中也叫“哨兵”，当发现这个哨兵后，写线程就退出。

当然线程池也要显式退出了。

```
package concurrent;
import java.io.File;
import java.io.FileFilter;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

public class TestBlockingQueue {
  static long randomTime() {
    return (long) (Math.random() * 1000);
  }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
    public static void main(String[] args) {
      // 能容纳100个文件
      final BlockingQueue<File> queue = new LinkedBlockingQueue<File>(100);
      // 线程池
      final ExecutorService exec = Executors.newFixedThreadPool(5);
      final File root = new File("F:\\JavaLib");
      // 完成标志
      final File exitFile = new File("");
      // 读个数
      final AtomicInteger rc = new AtomicInteger();
      // 写个数
      final AtomicInteger wc = new AtomicInteger();
      // 读线程
      Runnable read = new Runnable() {
        public void run() {
          scanFile(root);
          scanFile(exitFile);
        }

      public void scanFile(File file) {
        if (file.isDirectory()) {
          File[] files = file.listFiles(new FileFilter() {
            public boolean accept(File pathname) {
              return pathname.isDirectory()
                  || pathname.getPath().endsWith(".java");
            }
          });
          for (File one : files)
            scanFile(one);
        } else {
          try {
            int index = rc.incrementAndGet();
            System.out.println("Read0: " + index + " "
                + file.getPath());
            queue.put(file);
          } catch (InterruptedException e) {
          }
        }
      }
    };
    exec.submit(read);
    // 四个写线程
```


http://www.itheima.com Copyright© 2018 黑马程序员

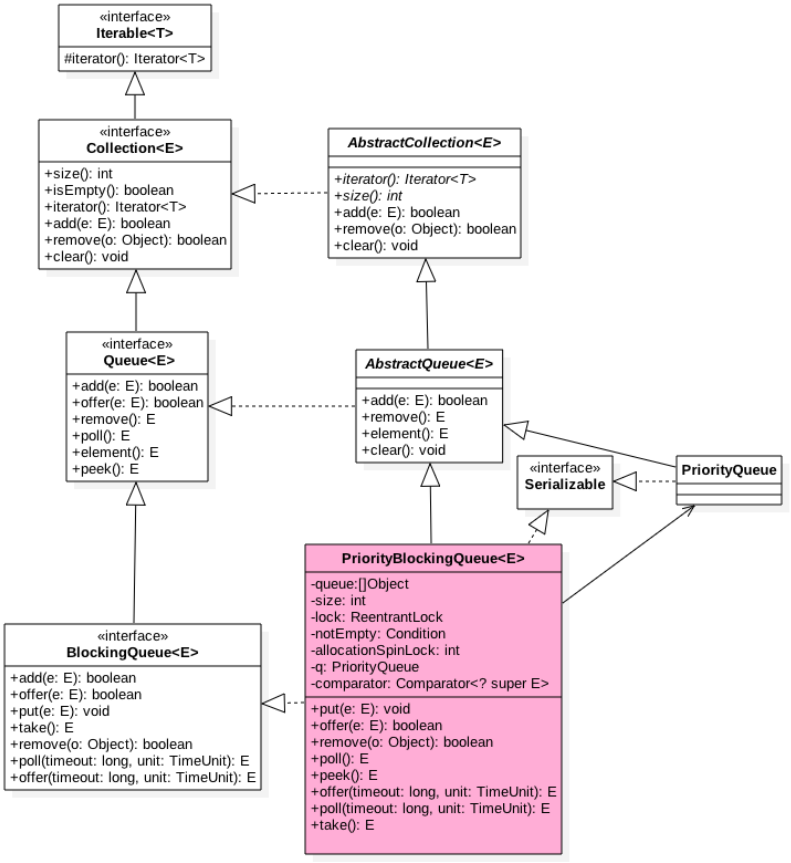
```
    for (int index = 0; index < 4; index++) {
      // write thread
      final int NO = index;
      Runnable write = new Runnable() {
        String threadName = "Write" + NO;
        public void run() {
          while (true) {
            try {
              Thread.sleep(randomTime());
              int index = wc.incrementAndGet();
              File file = queue.take();
              // 队列已经无对象
              if (file == exitFile) {
                // 再次添加"标志"，以让其他线程正常退出
                queue.put(exitFile);
                break;
              }
              System.out.println(threadName + ": " + index + " "
                  + file.getPath());
            } catch (InterruptedException e) {
            }
          }
        }
      };
      exec.submit(write);
    }
    exec.shutdown();
  }
}
```

➢PriorityBlockingQueue 无界阻塞优先级队列

PriorityBlockingQueue 是带优先级的无界阻塞队列，每次出队都返回优先级最高的元素，是二叉树最小堆的实现，研究过数组方式存放最小堆节点的都知道，直接遍历队列元素是无序的。

PriorityBlockingQueue 类图结构

http://www.itheima.com Copyright© 2018 黑马程序员



如图 PriorityBlockingQueue 内部有个数组 queue 用来存放队列元素，size 用来存放队列元素个数，allocationSpinLockOffset 是用来在扩容队列时候做 cas 的，目的是保证只有一个线程可以进行扩容。

由于这是一个优先级队列所以有个比较器 comparator 用来比较元素大小。lock 独占锁对象用来控制同时只能有一个线程可以进行入队出队操作。notEmpty 条件变量用来实现 take 方法阻塞模式。这里没有 notFull 条件变量是因为这里的 put 操作是非阻塞的，为啥要设计为非阻塞的是因为这是无界队列。

最后 PriorityQueue q 用来搞序列化的。

如下构造函数，默认队列容量为 11，默认比较器为 null;

```
private static final int DEFAULT_INITIAL_CAPACITY = 11;

public PriorityBlockingQueue() {
    this(DEFAULT_INITIAL_CAPACITY, null);
}
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
public PriorityBlockingQueue(int initialCapacity) {
    this(initialCapacity, null);
}

public PriorityBlockingQueue(int initialCapacity,
                             Comparator<? super E> comparator) {
    if (initialCapacity < 1)
        throw new IllegalArgumentException();
    this.lock = new ReentrantLock();
    this.notEmpty = lock.newCondition();
    this.comparator = comparator;
    this.queue = new Object[initialCapacity];
}
```

PriorityBlockingQueue 方法

✓ **Offer 操作**

在队列插入一个元素，由于是无界队列，所以一直为成功返回 true;

```
public boolean offer(E e) {

    if (e == null)
        throw new NullPointerException();
    final ReentrantLock lock = this.lock;
    lock.lock();
    int n, cap;
    Object[] array;

    //如果当前元素个数>=队列容量，则扩容(1)
    while ((n = size) >= (cap = (array = queue).length))
        tryGrow(array, cap);


    try {
        Comparator<? super E> cmp = comparator;

        //默认比较器为null
        if (cmp == null)(2)
            siftUpComparable(n, e, array);
        else
            //自定义比较器(3)
```


 Copyright© 2018 黑马程序员

```
                siftUpUsingComparator(n, e, array, cmp);

            //队列元素增加 1，并且激活 notEmpty 的条件队列里面的一个阻塞线程
            size = n + 1;（9）
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
        return true;
    }
```

主流程比较简单，下面看看两个主要函数

```
    private void tryGrow(Object[] array, int oldCap) {
        lock.unlock(); //must release and then re-acquire main lock
        Object[] newArray = null;

        //cas 成功则扩容(4)
        if (allocationSpinLock == 0 &&
            UNSAFE.compareAndSwapInt(this, allocationSpinLockOffset,
                                     0, 1)) {
            try {
                //oldGap<64 则扩容新增 oldcap+2,否者扩容 50%，并且最大为 MAX_ARRAY_SIZE
                int newCap = oldCap + ((oldCap < 64) ?
                                       (oldCap + 2) : // grow faster if small
                                       (oldCap >> 1));
                if (newCap - MAX_ARRAY_SIZE > 0) {    // possible overflow
                    int minCap = oldCap + 1;
                    if (minCap < 0 || minCap > MAX_ARRAY_SIZE)
                        throw new OutOfMemoryError();
                    newCap = MAX_ARRAY_SIZE;
                }
                if (newCap > oldCap && queue == array)
                    newArray = new Object[newCap];
            } finally {
                allocationSpinLock = 0;
            }
        }

        //第一个线程 cas 成功后，第二个线程会进入这个地方，然后第二个线程让出 cpu，尽量让第一个线程执行下面点获取锁，但是这得不到肯定的保证。(5)
        if (newArray == null) // back off if another thread is allocating
            Thread.yield();
        lock.lock();(6)
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        if (newArray != null && queue == array) {
            queue = newArray;
            System.arraycopy(array, 0, newArray, 0, oldCap);
        }
    }
```

tryGrow 目的是扩容，这里要思考下为啥在扩容前要先释放锁，然后使用 cas 控制只有一个线程可以扩容成功。我的理解是为了性能，因为扩容时候是需要花时间的，如果这些操作时候还占用锁那么其他线程在这个时候是不能进行出队操作的，也不能进行入队操作，这大大降低了并发性。

所以在扩容前释放锁，这允许其他出队线程可以进行出队操作，但是由于释放了锁，所以也允许在扩容时候进行入队操作，这就会导致多个线程进行扩容会出现问题，所以这里使用了一个 spinlock 用 cas 控制只有一个线程可以进行扩容，失败的线程调用 Thread.yield()让出 cpu，目的意在让扩容线程扩容后优先调用 lock.lock 重新获取锁，但是这得不到一定的保证，有可能调用 Thread.yield()的线程先获取了锁。

那 copy 元素数据到新数组为啥放到获取锁后面那?原因应该是因为可见性问题,因为 queue 并没有被 volatile 修饰。另外有可能在扩容时候进行了出队操作，如果直接拷贝可能看到的数组元素不是最新的。而通过调用 Lock 后，获取的数组则是最新的，并且在释放锁前 数组内容不会变化。

具体建堆算法：

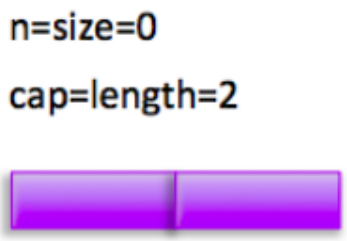
```
    private static <T> void siftUpComparable(int k, T x, Object[] array) {
        Comparable<? super T> key = (Comparable<? super T>) x;

        //队列元素个数>0 则判断插入位置，否者直接入队(7)
        while (k > 0) {
            int parent = (k - 1) >>> 1;
            Object e = array[parent];
            if (key.compareTo((T) e) >= 0)
                break;
            array[k] = e;
            k = parent;
        }
        array[k] = key;(8)
    }
```

下面用图说话模拟下过程：

http://www.itheima.com Copyright© 2018 黑马程序员

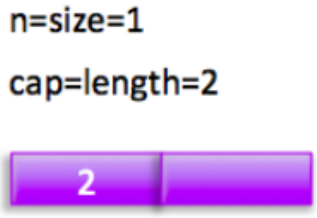
假设队列容量为 2

- 第一次 offer(2)时候



执行（1)为 false 所以执行（2），由于 k=n=size=0;所以执行（8）元素入队，然执行（9）size+1;

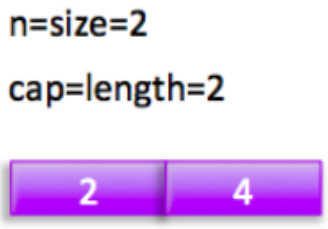
现在队列状态:



- 第二次 offer(4)时候

执行（1)为 false，所以执行（2）由于 k=1,所以进入 while 循环，parent=0;e=2;key=4;key>e 所以 break;

然后把 4 存到数据下标为 1 的地方，这时候队列状态为:



- 第三次 offer(4)时候

执行（1)为 true,所以调用 tryGrow,由于 2<64 所以 newCap=2 + (2+2)=6;然后创建新数组并拷贝，然后调用 siftUpComparable；k=2>0 进入循环 parent=0;e=2;key=6;key>e 所以 break;然后把 6 放入下标为 2 的地方，现在队列状态:

http://www.itheima.com Copyright© 2018 黑马程序员

n=size=3

cap=length=6

- 第四次 offer(1)时候

执行（1)为 false，所以执行（2）由于 k=3,所以进入 while 循环，parent=0;e=2;key=1; key<e；所以把 2 复制到数组下标为 3 的地方，然后 k=0 退出循环；然后把 2 存放到下标为 0 地方，现在状态：

n=size=4

cap=length=6

✓ **Poll 操作**

在队列头部获取并移除一个元素，如果队列为空，则返回 null

```
public E poll() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return dequeue();
    } finally {
        lock.unlock();
    }
}
```

主要看 dequeue

```
private E dequeue() {

    //队列为空，则返回 null
    int n = size - 1;
    if (n < 0)
        return null;
    else {



        //获取队头元素(1)
```

```
            Object[] array = queue;
            E result = (E) array[0];

            //获取对尾元素，并值 null(2)
            E x = (E) array[n];
            array[n] = null;

            Comparator<? super E> cmp = comparator;
            if (cmp == null)//cmp=null 则调用这个，把对尾元素位置插入到 0 位置，并且调整堆为最小堆(3)
                siftDownComparable(0, x, array, n);
            else
                siftDownUsingComparator(0, x, array, n, cmp);
            size = n;（4）
            return result;
        }
    }

    private static <T> void siftDownComparable(int k, T x, Object[] array,
                                               int n) {
         if (n > 0) {
             Comparable<? super T> key = (Comparable<? super T>)x;
             int half = n >>> 1;           // loop while a non-leaf
             while (k < half) {
                 int child = (k << 1) + 1; // assume left child is least
                 Object c = array[child];（5）
                 int right = child + 1;（6）
                 if (right < n &&
                     ((Comparable<? super T>) c).compareTo((T) array[right]) > 0)(7)
                     c = array[child = right];
                 if (key.compareTo((T) c) <= 0)(8)
                     break;
                 array[k] = c;
                 k = child;
             }
             array[k] = key;(9)
         }
     }
```

下面用图说话模拟下过程：

- 第一次调用 poll()

首先执行（1） result=1；然后执行（2）x=2;这时候队列状态

t© 2018 黑马程序员

n=size=4

cap=length=6

然后执行（3）后状态为：

n=size=4

cap=length=6

执行（4）后的结果：

n=size=3

cap=length=6

下面重点说说 siftDownComparable 这个屌屌的建立最小堆的算法：

首先说下思想，其中 k 一开始为 0，x 为数组里面最后一个元素，由于第 0 个元素为树根，被出队时候要被搞掉，所以建堆要从它的左右孩子节点找一个最小的值来当树根，子树根被搞掉后，会找子树的左右孩子最小的元素来代替，直到树节点为止，还不明白，没关系，看图说话：

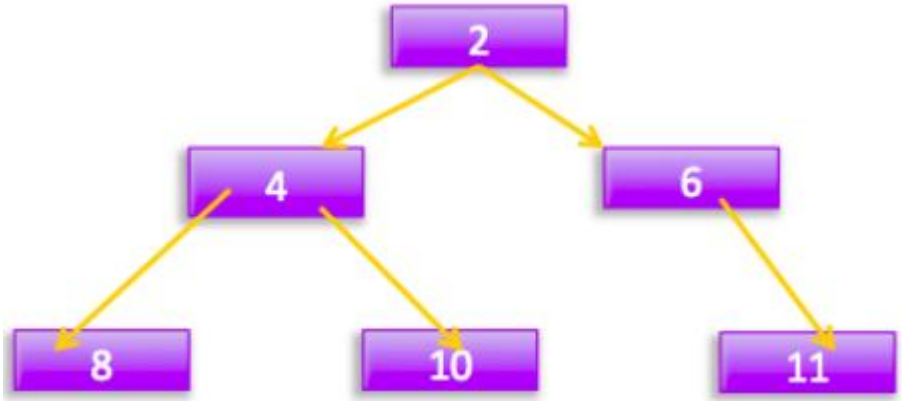
假如当前队列元素：

http://www.itheima.com Copyright© 2018 黑马程序员

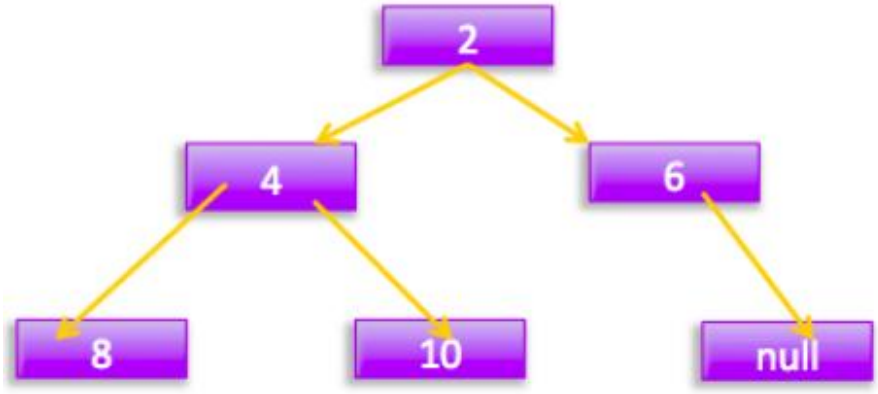
size=6

length=6

那么对于树为：

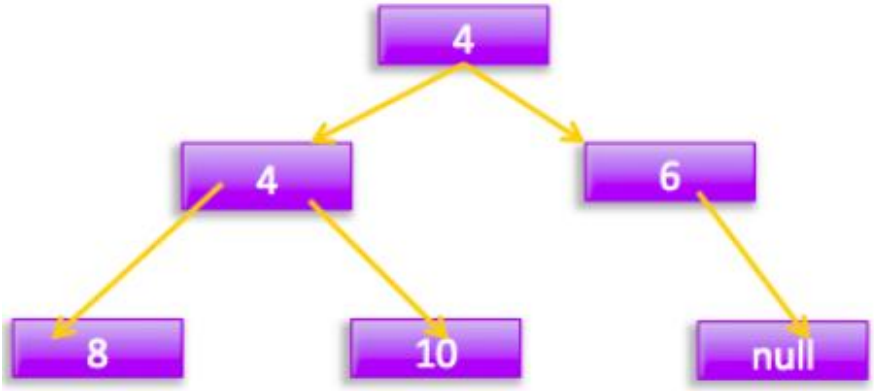


这时候如果调用了 poll();那么 result=2;x=11;现在树为：



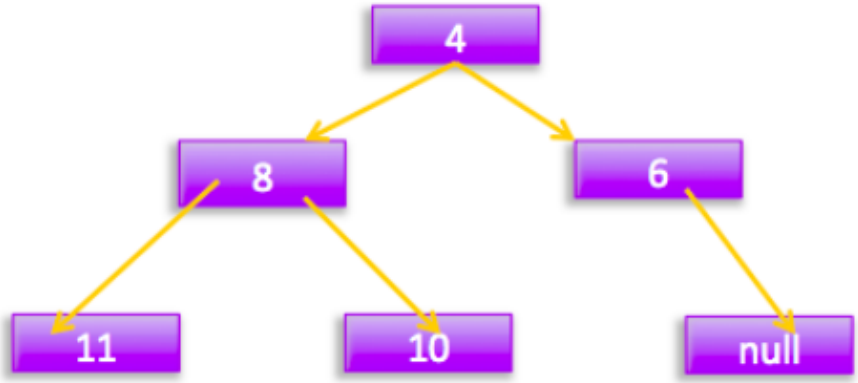
然后看 leftChildVal = 4;rightChildVal = 6; 4<6;所以 c=4;也就是获取根节点的左右孩子值小的那一个； 然后看 11>4 也就是 key>c；然后把 c 放入树根，现在树为：

http://www.itheima.com Copyright© 2018 黑马程序员



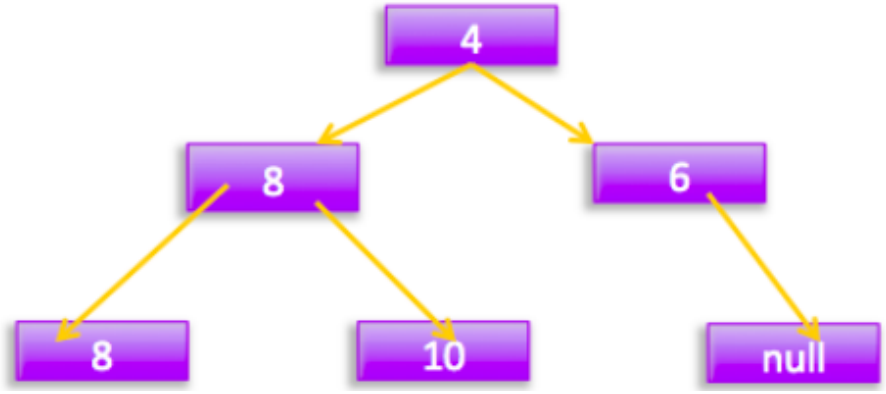
然后看根的左边孩子 4 为根的子树我们要为这个字树找一个根节点。

看 leftChildVal = 8;rightChildVal = 10; 8<10;所以 c=8;也就是获取根节点的左右孩子值小的那一个； 然后看 11>8 也就是 key>c；然后把 c 放入树根，现在树为：



这时候 k=3;half=3 所以推出循环，执行（9）后结果为：

http://www.itheima.com Copyright© 2018 黑马程序员



这时候队列为：

Size=5

length=6

✓ **Put 操作**

内部调用的 offer,由于是无界队列，所以不需要阻塞

```
public void put(E e) {
    offer(e); // never need to block
}
```

✓ **Take 操作**

获取队列头元素，如果队列为空则阻塞。

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    E result;
    try {

        //如果队列为空，则阻塞，把当前线程放入 notEmpty 的条件队列
        while ( (result = dequeue()) == null)
            notEmpty.await();
    } finally {
        lock.unlock();
    }
```


Copyright© 2018 黑马程序员

```
        return result;
    }
```

这里是阻塞实现，阻塞后直到入队操作调用 notEmpty.signal 才会返回。

✓ **Size 操作**

获取队列元个数，由于加了独占锁所以返回结果是精确的

```
    public int size() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            return size;
        } finally {
            lock.unlock();
        }
    }
```

PriorityBlockingQueue 小结

PriorityBlockingQueue 类似于 ArrayBlockingQueue 内部使用一个独占锁来控制同时只有一个线程可以进行入队和出队，另外前者只使用了一个 notEmpty 条件变量而没有 notFull 这是因为前者是无界队列，当 put 时候永远不会处于 await 所以也不需要被唤醒。

PriorityBlockingQueue 始终保证出队的元素是优先级最高的元素，并且可以定制优先级的规则，内部通过使用一个二叉树最小堆算法来维护内部数组，这个数组是可扩容的，当当前元素个数>=最大容量时候会通过算法扩容。

值得注意的是为了避免在扩容操作时候其他线程不能进行出队操作，实现上使用了先释放锁，然后通过 cas 保证同时只有一个线程可以扩容成功。

PriorityBlockingQueue 示例

PriorityBlockingQueue 类是 JDK 提供的优先级队列 本身是线程安全的 内部使用显示锁 保证线程安全。

PriorityBlockingQueue 存储的对象必须是实现 Comparable 接口的 因为 PriorityBlockingQueue 队列会根据内部存储的每一个元素的 compareTo 方法比较每个元素的大小。这样在 take 出来的时候会根据优先级 将优先级最小的最先取出 。

http://www.itheima.com Copyright© 2018 黑马程序员

下面是示例代码

```
public static PriorityBlockingQueue<User> queue = new PriorityBlockingQueue<User>();

public static void main(String[] args) {
    queue.add(new User(1,"wu"));
    queue.add(new User(5,"wu5"));
    queue.add(new User(23,"wu23"));
    queue.add(new User(55,"wu55"));
    queue.add(new User(9,"wu9"));
    queue.add(new User(3,"wu3"));
    for (User user : queue) {
        try {
            System.out.println(queue.take().name);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

//静态内部类
static class User implements Comparable<User>{

    public User(int age,String name) {
        this.age = age;
        this.name = name;
    }

    int age;
    String name;

    @Override
    public int compareTo(User o) {
        return this.age > o.age ? -1 : 1;
    }
}
```

➢SychronousQueue 同步队列

SynchronousQueue 是一个比较特别的队列，由于在线程池方面有所应用，为了更好的理解线程池的实现原理，此队列源码中充斥着大量的 CAS 语句，理解起来是有些难度的，为了方便日后回顾，本篇文章会以简洁的图形化方式

http://www.itheima.com Copyright© 2018 黑马程序员

展示该队列底层的实现原理。

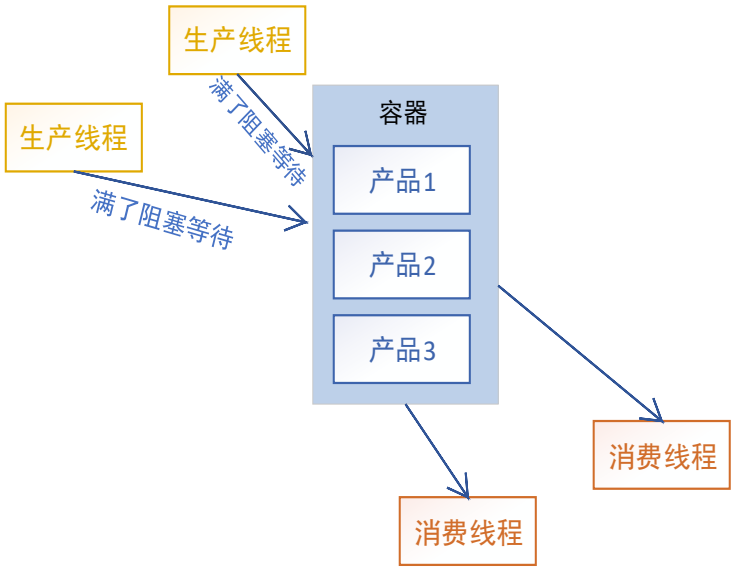
SychronousQueue 简单实用

经典的生产者-消费者模式，操作流程是这样的：

有多个生产者，可以并发生产产品，把产品置入队列中，如果队列满了，生产者就会阻塞；

有多个消费者，并发从队列中获取产品，如果队列空了，消费者就会阻塞；

如下面的示意图所示：



SynchronousQueue 也是一个队列来的，**但它的特别之处在于它内部没有容器**，一个生产线程，当它生产产品（即put 的时候），如果当前没有人想要消费产品(即当前没有线程执行 take)，此生产线程必须阻塞，等待一个消费线程调用 take 操作，take 操作将会唤醒该生产线程，同时消费线程会获取生产线程的产品（即数据传递），这样的一个过程称为一次配对过程(当然也可以先 take 后 put,原理是一样的)。

我们用一个简单的代码来验证一下，如下所示：

```
package com.concurrent;
import java.util.concurrent.SynchronousQueue;
public class SynchronousQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        final SynchronousQueue<Integer> queue = new SynchronousQueue<Integer>();
```

```
        Thread putThread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("put thread start");
                try {
                    queue.put(1);
                } catch (InterruptedException e) {
                }
                System.out.println("put thread end");
            }
        });

        Thread takeThread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("take thread start");
                try {
                    System.out.println("take from putThread: " + queue.take());
                } catch (InterruptedException e) {
                }
                System.out.println("take thread end");
            }
        });

        putThread.start();
        Thread.sleep(1000);
        takeThread.start();
    }
}
```

一种输出结果如下：

```
put thread start
take thread start
take from putThread: 1
put thread end
take thread end
```

从结果可以看出，put 线程执行 queue.put(1) 后就被阻塞了，只有 take 线程进行了消费，put 线程才可以返回。

可以认为这是一种线程与线程间一对一传递消息的模型。

SychronousQueue 实现原理

http://www.itheima.com Copyright© 2018 黑马程序员

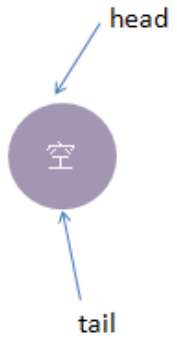
不像 ArrayBlockingQueue、LinkedBlockingDeque 之类的阻塞队列依赖 AQS 实现并发操作，SynchronousQueue 直接使用 CAS 实现线程的安全访问。

队列的实现策略通常分为公平模式和非公平模式，接下来将分别进行说明。

公平模式下的模型：

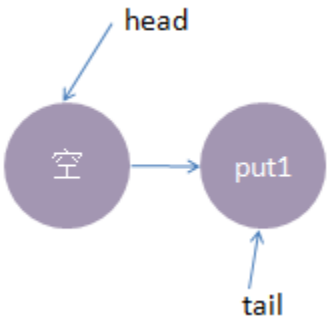
公平模式下，底层实现使用的是 TransferQueue 这个内部队列，它有一个 head 和 tail 指针，用于指向当前正在等待匹配的线程节点。

初始化时，TransferQueue 的状态如下：



接着我们进行一些操作：

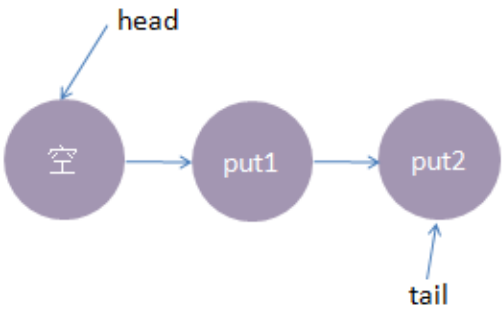
1、线程 put1 执行 put(1)操作，由于当前没有配对的消费线程，所以 put1 线程入队列，自旋一小会后睡眠等待，这时队列状态如下：



2、接着，线程 put2 执行了 put(2)操作，跟前面一样，put2 线程入队列，自旋一小会后睡眠等待，这时队列

http://www.itheima.com Copyright© 2018 黑马程序员

状态如下：

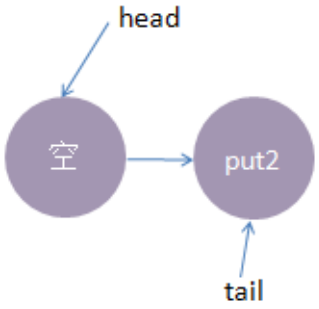


3、这时候，来了一个线程 take1，执行了 take 操作，由于 tail 指向 put2 线程，put2 线程跟 take1 线程配对了(一 put 一 take)，这时 take1 线程不需要入队，但是请注意了，这时候，要唤醒的线程并不是 put2，而是 put1。

为何？ 大家应该知道我们现在讲的是公平策略，所谓公平就是谁先入队了，谁就优先被唤醒，我们的例子明显是 put1 应该优先被唤醒。至于读者可能会有一个疑问，明明是 take1 线程跟 put2 线程匹配上了，结果是 put1 线程被唤醒消费，怎么确保 take1 线程一定可以和次首节点(head.next)也是匹配的呢？其实大家可以拿个纸画一画，就会发现真的就是这样的。

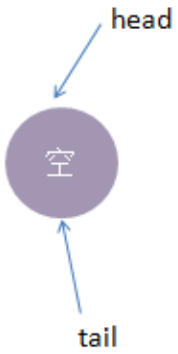
公平策略总结下来就是：队尾匹配队头出队。

执行后 put1 线程被唤醒，take1 线程的 take()方法返回了 1(put1 线程的数据)，这样就实现了线程间的一对一通信，这时候内部状态如下：



http://www.itheima.com Copyright© 2018 黑马程序员

4、最后，再来一个线程 take2，执行 take 操作，这时候只有 put2 线程在等候，而且两个线程匹配上了，线程 put2 被唤醒， take2 线程 take 操作返回了 2(线程 put2 的数据)，这时候队列又回到了起点，如下所示：

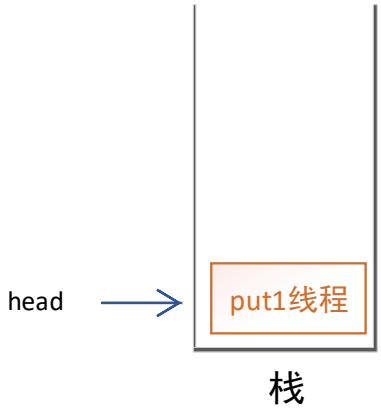


以上便是公平模式下，SynchronousQueue 的实现模型。总结下来就是：队尾匹配队头出队，先进先出，体现公平原则。

非公平模式下的模型：

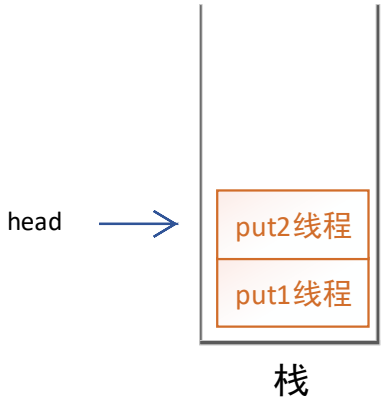
我们还是使用跟公平模式下一样的操作流程，对比两种策略下有何不同。非公平模式底层的实现使用的是 TransferStack，一个栈，实现中用 head 指针指向栈顶，接着我们看看它的实现模型：

1、线程 put1 执行 put(1)操作，由于当前没有配对的消费线程，所以 put1 线程入栈，自旋一小会后睡眠等待，这时栈状态如下：



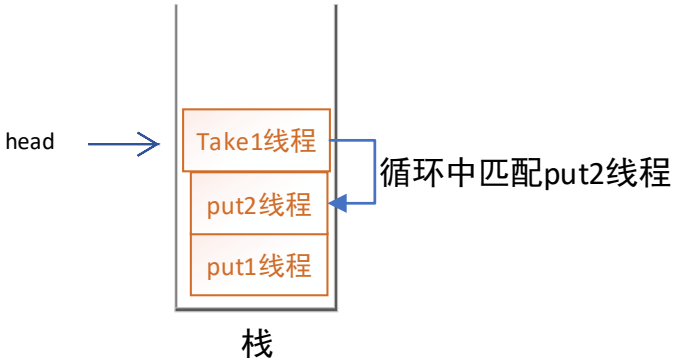
http://www.itheima.com Copyright© 2018 黑马程序员

2、接着，线程 put2 再次执行了 put(2)操作，跟前面一样，put2 线程入栈，自旋一小会后睡眠等待，这时栈状态如下：



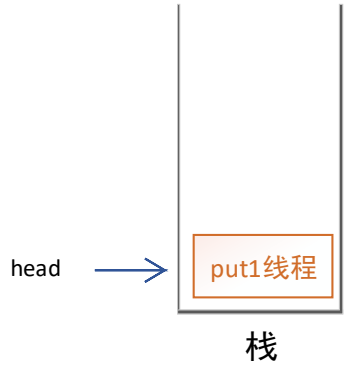
3、这时候，来了一个线程 take1，执行了 take 操作，这时候发现栈顶为 put2 线程，匹配成功，但是实现会先把 take1 线程入栈，然后 take1 线程循环执行匹配 put2 线程逻辑，一旦发现没有并发冲突，就会把栈顶指针直接指向 put1 线程

步骤一：



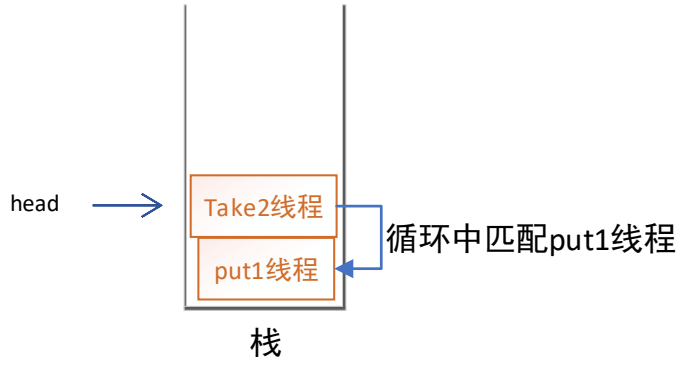
http://www.itheima.com Copyright© 2018 黑马程序员

步骤二：



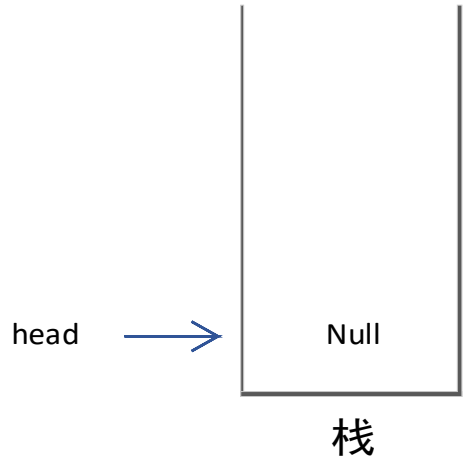
4、最后，再来一个线程 take2，执行 take 操作，这跟步骤 3 的逻辑基本是一致的，take2 线程入栈，然后在循环中匹配 put1 线程，最终全部匹配完毕，栈变为空，恢复初始状态，如下图所示：

步骤一：



步骤二：

http://www.itheima.com Copyright© 2018 黑马程序员



可以从上面流程看出，虽然 put1 线程先入栈了，但是却是后匹配，这就是非公平的由来。

SychronousQueue 总结

SynchronousQueue 由于其独有的线程一一配对通信机制，在大部分平常开发中，可能都不太会用到，但线程池技术中会有所使用，由于内部没有使用 AQS，而是直接使用 CAS，所以代码理解起来会比较困难，但这并不妨碍我们理解底层的实现模型，在理解了模型的基础上，有兴趣的话再查阅源码，就会有方向感，看起来也会比较容易，希望本文有所借鉴意义。

➢DeplayQueue 延时无界阻塞队列

在谈到 DelayQueue 的使用和原理的时候，我们首先介绍一下 DelayQueue，DelayQueue 是一个无界阻塞队列，只有在延迟期满时才能从中提取元素。该队列的头部是延迟期满后保存时间最长的 Delayed 元素。

DelayQueue 阻塞队列在我们系统开发中也常常会用到，例如：缓存系统的设计，缓存中的对象，超过了空闲时间，需要从缓存中移出；任务调度系统，能够准确的把握任务的执行时间。我们可能需要通过线程处理很多时间上要求很严格的数据，如果使用普通的线程，我们就需要遍历所有的对象，一个一个的检 查看数据是否过期等，首先这样在执行上的效率不会太高，其次就是这种设计的风格也大大的影响了数据的精度。一个需要 12:00 点执行的任务可能 12:01 才执行,这样对数据要求很高的系统有更大的弊端。由此我们可以使用 DelayQueue。

http://www.itheima.com Copyright© 2018 黑马程序员

下面将会对 DelayQueue 做一个介绍，然后举个例子。并且提供一个 Delayed 接口的实现和 Sample 代码。DelayQueue 是一个 BlockingQueue，其特化的参数是 Delayed。（不了解 BlockingQueue 的同学，先去了解 BlockingQueue 再看本文）

Delayed 扩展了 Comparable 接口，比较的基准为延时的时间值，Delayed 接口的实现类 getDelay 的返回值应为固定值（final）。DelayQueue 内部是使用 PriorityQueue 实现的。

DelayQueue = BlockingQueue +PriorityQueue + Delayed

DelayQueue 定义和原理

DelayQueue 的关键元素 BlockingQueue、PriorityQueue、Delayed。可以这么说，DelayQueue 是一个使用优先队列（PriorityQueue）实现的 BlockingQueue，**优先队列的比较基准值是时间。**

他们的基本定义如下

```
public interface Comparable<T> {
    public int compareTo(T o);
}
public interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
public class DelayQueue<E extends Delayed> implements BlockingQueue<E> {
    private final PriorityQueue<E> q = new PriorityQueue<E>();
}
```

DelayQueue 内部的实现使用了一个优先队列。当调用 DelayQueue 的 offer 方法时，把 Delayed 对象加入到优先队列 q 中。如下：

```
public boolean offer(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        E first = q.peek();
        q.offer(e);
        if (first == null || e.compareTo(first) < 0)
            available.signalAll();
        return true;
    } finally {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            lock.unlock();
        }
    }
```

DelayQueue 的 take 方法，把优先队列 q 的 first 拿出来（peek），如果没有达到延时阀值，则进行 await 处理。如下：

```
    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            for (;;) {
                E first = q.peek();
                if (first == null) {
                    available.await();
                } else {
                    long delay =  first.getDelay(TimeUnit.NANOSECONDS);
                    if (delay > 0) {
                        long tl = available.awaitNanos(delay);
                    } else {
                        E x = q.poll();
                        assert x != null;
                        if (q.size() != 0)
                            available.signalAll(); // wake up other takers
                        return x;

                    }
                }
            }
        } finally {
            lock.unlock();
        }
    }
```

DelayQueue 实例应用

Ps：为了具有调用行为，存放到 DelayDeque 的元素必须继承 Delayed 接口。Delayed 接口使对象成为延迟对象，它使存放在 DelayQueue 类中的对象具有了激活日期。该接口强制执行下列两个方法。

一下将使用 Delay 做一个缓存的实现。其中共包括三个类

- Pair

http://www.itheima.com Copyright© 2018 黑马程序员

- DelayItem
- Cache

Pair 类:

```
public class Pair<K, V> {
    public K first;

    public V second;

    public Pair() {}

    public Pair(K first, V second) {
        this.first = first;
        this.second = second;
    }
}
```

一下是对 Delay 接口的实现:

```
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;

public class DelayItem<T> implements Delayed {
    /** Base of nanosecond timings, to avoid wrapping */
    private static final long NANO_ORIGIN = System.nanoTime();

    /**
     * Returns nanosecond time offset by origin
     */
    final static long now() {
        return System.nanoTime() - NANO_ORIGIN;
    }

    /**
     * Sequence number to break scheduling ties, and in turn to guarantee FIFO order among tied
     * entries.
     */
    private static final AtomicLong sequencer = new AtomicLong(0);

    /** Sequence number to break ties FIFO */
    private final long sequenceNumber;
```


 Copyright© 2018 黑马程序员

```
    /** The time the task is enabled to execute in nanoTime units */
    private final long time;

    private final T item;

    public DelayItem(T submit, long timeout) {
        this.time = now() + timeout;
        this.item = submit;
        this.sequenceNumber = sequencer.getAndIncrement();
    }

    public T getItem() {
        return this.item;
    }

    public long getDelay(TimeUnit unit) {
        long d = unit.convert(time - now(), TimeUnit.NANOSECONDS);
        return d;
    }

    public int compareTo(Delayed other) {
        if (other == this) // compare zero ONLY if same object
            return 0;
        if (other instanceof DelayItem) {
            DelayItem x = (DelayItem) other;
            long diff = time - x.time;
            if (diff < 0)
                return -1;
            else if (diff > 0)
                return 1;
            else if (sequenceNumber < x.sequenceNumber)
                return -1;
            else
                return 1;
        }
        long d = (getDelay(TimeUnit.NANOSECONDS) - other.getDelay(TimeUnit.NANOSECONDS));
        return (d == 0) ? 0 : ((d < 0) ? -1 : 1);
    }
}
```

以下是 Cache 的实现，包括了 put 和 get 方法

```
import java.util.concurrent.ConcurrentHashMap;
```


 Copyright© 2018 黑马程序员

```
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Cache<K, V> {
    private static final Logger LOG = Logger.getLogger(Cache.class.getName());

    private ConcurrentMap<K, V> cacheObjMap = new ConcurrentHashMap<K, V>();

    private DelayQueue<DelayItem<Pair<K, V>>> q = new DelayQueue<DelayItem<Pair<K, V>>>();

    private Thread daemonThread;

    public Cache() {

        Runnable daemonTask = new Runnable() {
            public void run() {
                daemonCheck();
            }
        };

        daemonThread = new Thread(daemonTask);
        daemonThread.setDaemon(true);
        daemonThread.setName("Cache Daemon");
        daemonThread.start();
    }

    private void daemonCheck() {

        if (LOG.isLoggable(Level.INFO))
            LOG.info("cache service started.");

        for (;;) {
            try {
                DelayItem<Pair<K, V>> delayItem = q.take();
                if (delayItem != null) {
                    // 超时对象处理
                    Pair<K, V> pair = delayItem.getItem();
                    cacheObjMap.remove(pair.first, pair.second); // compare and remove
                }
            } catch (InterruptedException e) {
```


 Copyright© 2018 黑马程序员

```
                if (LOG.isLoggable(Level.SEVERE))
                    LOG.log(Level.SEVERE, e.getMessage(), e);
                break;
            }
        }

        if (LOG.isLoggable(Level.INFO))
            LOG.info("cache service stopped.");
    }

    // 添加缓存对象
    public void put(K key, V value, long time, TimeUnit unit) {
        V oldValue = cacheObjMap.put(key, value);
        if (oldValue != null)
            q.remove(key);

        long nanoTime = TimeUnit.NANOSECONDS.convert(time, unit);
        q.put(new DelayItem<Pair<K, V>>(new Pair<K, V>(key, value), nanoTime));
    }

    public V get(K key) {
        return cacheObjMap.get(key);
    }
```

测试 main 方法：

```
    // 测试入口函数
    public static void main(String[] args) throws Exception {
        Cache<Integer, String> cache = new Cache<Integer, String>();
        cache.put(1, "aaaa", 3, TimeUnit.SECONDS);

        Thread.sleep(1000 * 2);
        {
            String str = cache.get(1);
            System.out.println(str);
        }

        Thread.sleep(1000 * 2);
        {
            String str = cache.get(1);
            System.out.println(str);
        }
    }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
输出结果为：
aaaa
null
```

我们看到上面的结果，如果超过延时的时间，那么缓存中数据就会自动丢失，获得就为 null。

c. 并发（Collection）队列-非阻塞队列

➢非阻塞队列

首先我们要简单的理解下什么是非阻塞队列：

与阻塞队列相反，非阻塞队列的执行并不会被阻塞，无论是消费者的出队，还是生产者的入队。

在底层，非阻塞队列使用的是 CAS(compare and swap)来实现线程执行的非阻塞。

非阻塞队列简单操作

与阻塞队列相同，非阻塞队列中的常用方法，也是出队和入队。

入队方法：

- add()：底层调用 offer();
- offer()：Queue 接口继承下来的方法，实现队列的入队操作，不会阻碍线程的执行，插入成功返回 true；

出队方法：

- poll()：移动头结点指针，返回头结点元素，并将头结点元素出队；队列为空，则返回 null；
- peek()：移动头结点指针，返回头结点元素，并不会将头结点元素出队；队列为空，则返回 null；

➢非阻塞算法 CAS

首先我们需要了解**悲观锁**和**乐观锁**

悲观锁：假定并发环境是悲观的，如果发生并发冲突，就会破坏一致性，所以要通过独占锁彻底禁止冲突发生。有一个经典比喻，“如果你不锁门，那么捣蛋鬼就回闯入并搞得一团糟”，所以“你只能一次打开门放进

http://www.itheima.com Copyright© 2018 黑马程序员

一个人，才能时刻盯紧他”。

乐观锁：假定并发环境是乐观的，即，虽然会有并发冲突，但冲突可发现且不会造成损害，所以，可以不加任何保护，等发现并发冲突后再决定放弃操作还是重试。可类比的比喻为，“如果你不锁门，那么虽然捣蛋鬼会闯入，但他们一旦打算破坏你就能知道”，所以“你大可以放进所有人，等发现他们想破坏的时候再做决定”。通常认为乐观锁的性能比悲观所更高，特别是在某些复杂的场景。这主要由于悲观锁在加锁的同时，也会把某些不会造成破坏的操作保护起来；而乐观锁的竞争则只发生在最小的并发冲突处，如果用悲观锁来理解，就是“锁的粒度最小”。但乐观锁的设计往往比较复杂，因此，复杂场景下还是多用悲观锁。

首先保证正确性，有必要的话，再去追求性能。

CAS

乐观锁的实现往往需要硬件的支持，多数处理器都都实现了一个 CAS 指令，实现“Compare And Swap”的语义（这里的 swap 是“换入”，也就是 set），构成了基本的乐观锁。

CAS 包含 3 个操作数：

- 需要读写的内存位置 V
- 进行比较的值 A
- 拟写入的新值 B

当且仅当位置 V 的值等于 A 时，CAS 才会通过原子方式用新值 B 来更新位置 V 的值；否则不会执行任何操作。无论位置 V 的值是否等于 A，都将返回 V 原有的值。

一个有意思的事实是，“使用 CAS 控制并发”与“使用乐观锁”并不等价。CAS 只是一种手段，既可以实现乐观锁，也可以实现悲观锁。乐观、悲观只是一种并发控制的策略。下文将分别用 CAS 实现悲观锁和乐观锁?

➢ ConcurrentLinkedQueue 非阻塞无界链表队列

ConcurrentLinkedQueue 是一个线程安全的队列，基于链表结构实现，是一个无界队列，理论上来说队列的

http://www.itheima.com Copyright© 2018 黑马程序员

长度可以无限扩大。

与其他队列相同，ConcurrentLinkedQueue 也采用的是先进先出（FIFO）入队规则，对元素进行排序。当我们向队列中添加元素时，新插入的元素会插入到队列的尾部；而当我们获取一个元素时，它会从队列的头部中取出。

因为 ConcurrentLinkedQueue 是链表结构，所以当入队时，插入的元素依次向后延伸，形成链表；而出队时，则从链表的第一个元素开始获取，依次递增；

不知道，我这样形容能否让你对链表的入队、出队产生一个大概的思路！

ConcurrentLinkedQuere 简单示例

值得注意的是，在使用 ConcurrentLinkedQueue 时，如果涉及到队列是否为空的判断，切记不可使用 size()==0 的做法，因为在 size()方法中，是通过遍历整个链表来实现的，在队列元素很多的时候，size()方法十分消耗性能和时间，只是单纯的判断队列为空使用 isEmpty()即可！！！

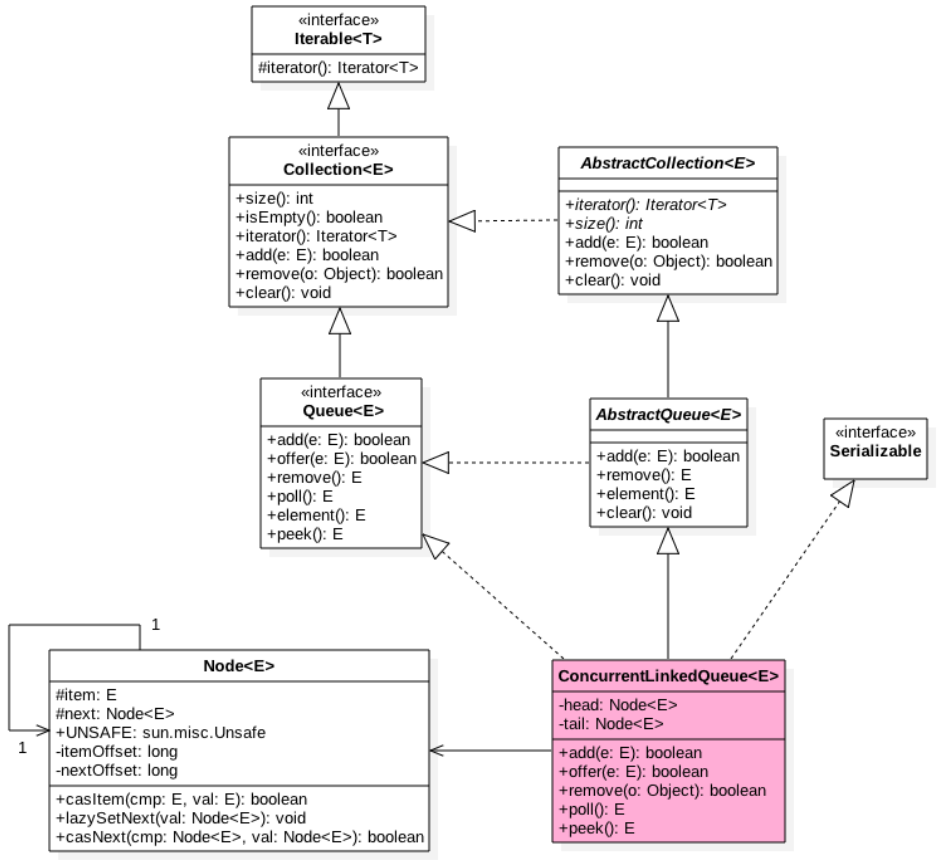
```
public class ConcurrentLinkedQueueTest {

    public static int threadCount = 10;

    public static ConcurrentLinkedQueue<String> queue = new ConcurrentLinkedQueue<String>();

    static class Offer implements Runnable {
        public void run() {
            //不建议使用 queue.size()==0，影响效率。可以使用!queue.isEmpty()
            if(queue.size()==0){
                String ele = new Random().nextInt(Integer.MAX_VALUE)+"";
                queue.offer(ele);
                System.out.println("入队元素为"+ele);
            }
        }
    }

    static class Poll implements Runnable {
        public void run() {
            if(!queue.isEmpty()){
                String ele = queue.poll();
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
                System.out.println("出队元素为"+ele);
            }
        }
    }

    public static void main(String[] agrs) {
        ExecutorService executorService = Executors.newFixedThreadPool(4);
        for(int x=0;x<threadCount;x++){
            executorService.submit(new Offer());
            executorService.submit(new Poll());
        }
        executorService.shutdown();
    }
}

一种输出：
入队元素为 313732926
出队元素为 313732926
入队元素为 812655435
出队元素为 812655435
入队元素为 1893079357
出队元素为 1893079357
入队元素为 1137820958
出队元素为 1137820958
入队元素为 1965962048
出队元素为 1965962048
出队元素为 685567162
入队元素为 685567162
出队元素为 1441081163
入队元素为 1441081163
出队元素为 1627184732
入队元素为 1627184732
```

ConcurrentLinkedQuere 类图

http://www.itheima.com Copyright© 2018 黑马程序员



如图 ConcurrentLinkedQueue 中有两个 volatile 类型的 Node 节点分别用来存在列表的首尾节点，其中 head 节点存放链表第一个 item 为 null 的节点，tail 则并不是总指向最后一个节点。Node 节点内部则维护一个变量 item 用来存放节点的值，next 用来存放下一个节点，从而链接为一个单向无界列表。

```
public ConcurrentLinkedQueue() {
    head = tail = new Node<E>(null);
}
```

如上代码初始化时候会构建一个 item 为 NULL 的空节点作为链表的首尾节点。

ConcurrentLinkedQuere 方法

✓ Offer 操作

offer 操作是在链表末尾添加一个元素，下面看看实现原理。

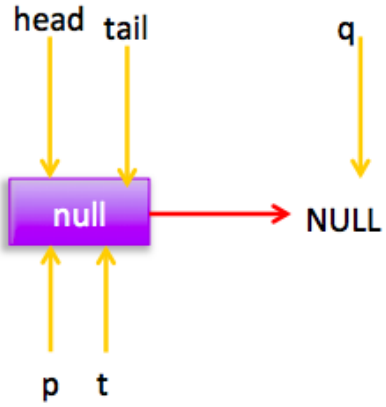
```
public boolean offer(E e) {
    //e 为 null 则抛出空指针异常
```


http://www.itheima.com Copyright© 2018 黑马程序员

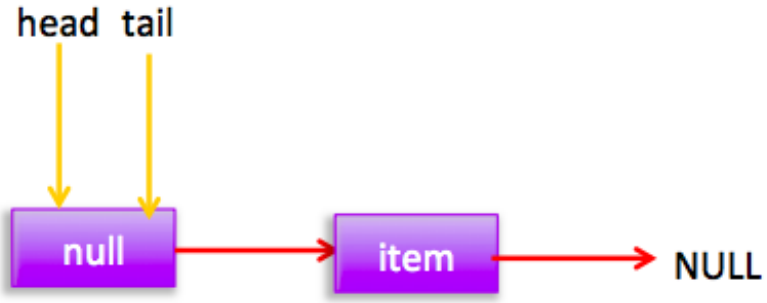
```
        checkNotNull(e);

       //构造 Node 节点构造函数内部调用 unsafe.putObject，后面统一讲
        final Node<E> newNode = new Node<E>(e);



        //从尾节点插入
        for (Node<E> t = tail, p = t;;) {

            Node<E> q = p.next;

            //如果 q=null 说明 p 是尾节点则插入
            if (q == null) {

                //cas 插入（1）
                if (p.casNext(null, newNode)) {
                    //cas 成功说明新增节点已经被放入链表，然后设置当前尾节点（包含 head，1，3，5.。。个节点为尾节点）
                    if (p != t) // hop two nodes at a time
                        casTail(t, newNode);  // Failure is OK.
                    return true;
                }
                // Lost CAS race to another thread; re-read next
            }
            else if (p == q)//(2)
                //多线程操作时候，由于 poll 时候会把老的 head 变为自引用，然后 head 的 next 变为新 head，所以这里需要
                //重新找新的 head，因为新的 head 后面的节点才是激活的节点
                p = (t != (t = tail)) ? t : head;
            else
                // 寻找尾节点(3)
                p = (p != t && t != (t = tail)) ? t : q;
        }
    }
```

从构造函数知道一开始有个 item 为 null 的哨兵节点，并且 head 和 tail 都是指向这个节点，然后当一个线程调用 offer 时候首先

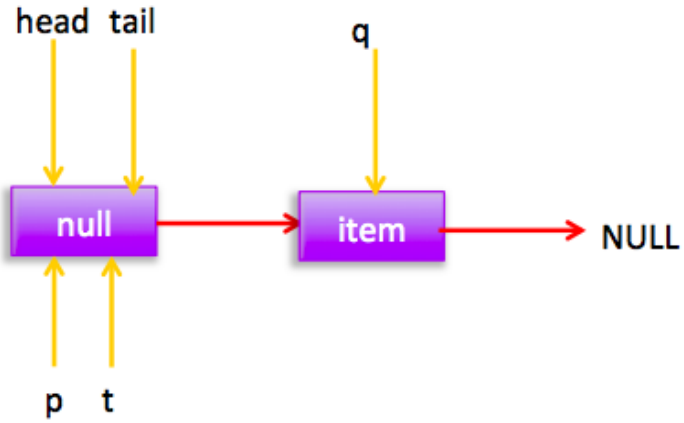
http://www.itheima.com Copyright© 2018 黑马程序员



如图首先查找尾节点，q==null,p 就是尾节点，所以执行 p.casNext 通过 cas 设置 p 的 next 为新增节点，这时候 p==t 所以不重新设置尾节点为当前新节点。由于多线程可以调用 offer 方法，所以可能两个线程同时执行到了（1）进行 cas，那么只有一个会成功（假如线程 1 成功了），成功后的链表为：

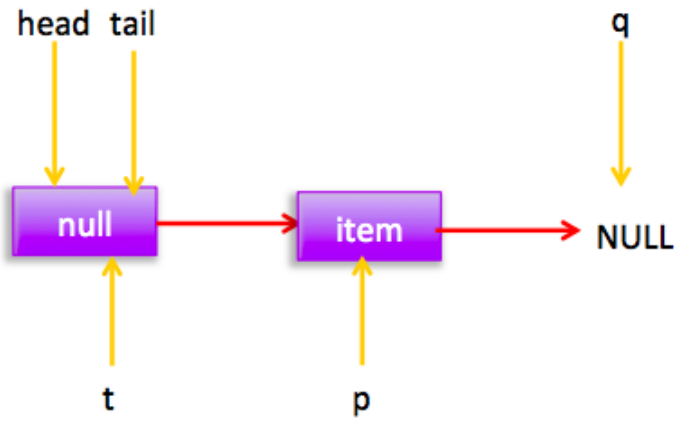


失败的线程会循环一次这时候指针为：

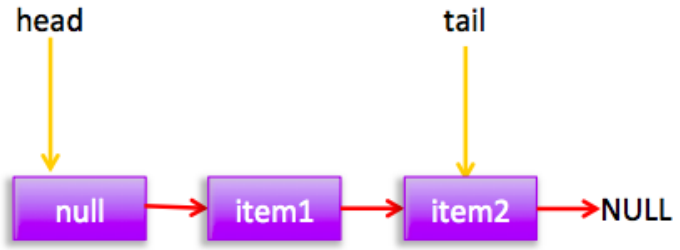


http://www.itheima.com Copyright© 2018 黑马程序员

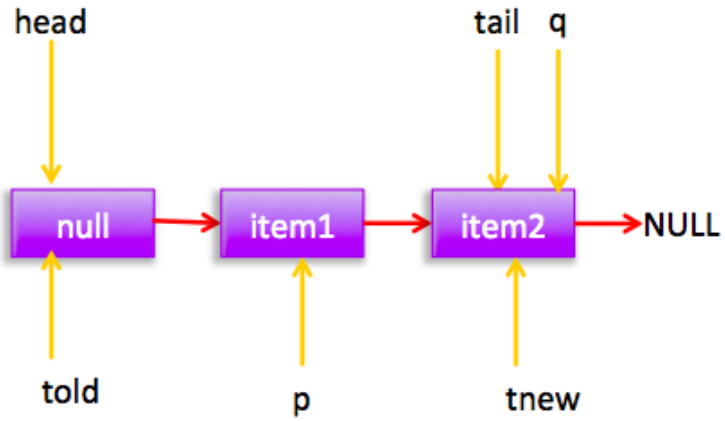
这时候会执行（3）所以 p=q,然后在循环后指针位置为：



所以没有其他线程干扰的情况下会执行（1）执行 cas 把新增节点插入到尾部，没有干扰的情况下线程 2 cas 会成功，然后去更新尾节点 tail,由于 p!=t 所以更新。这时候链表和指针为：



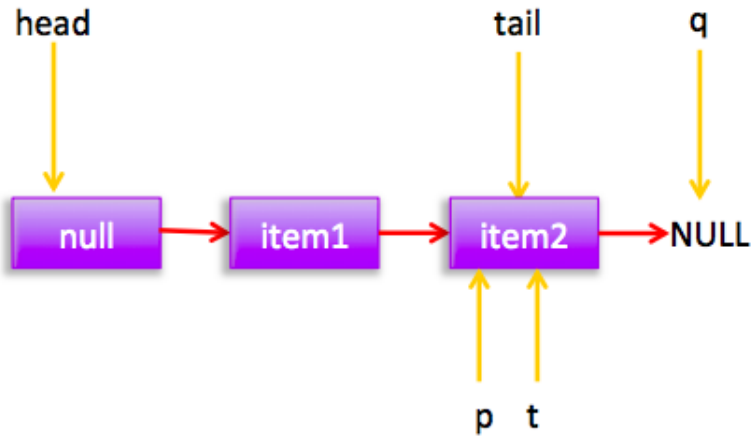
假如线程 2cas 时候线程 3 也在执行，那么线程 3 会失败，循环一次后，线程 3 的节点状态为：



这时候 p!=t ；并且 t 的原始值为 told，t 的新值为 tnew ,所以 told!=tnew，所以 p=tnew=tail;

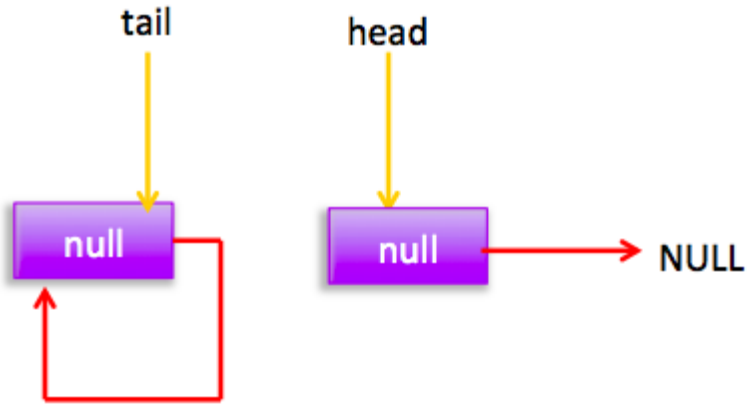
http://www.itheima.com Copyright© 2018 黑马程序员

然后在循环一下后节点状态:



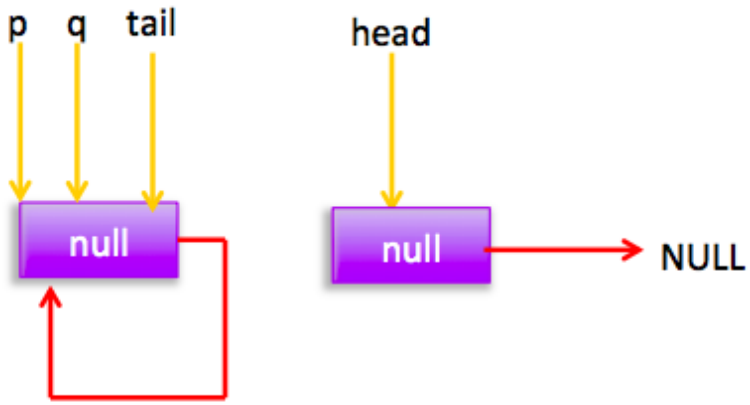
q==null 所以执行（1）。

现在就差 p==q 这个分支还没有走，这个要在执行 poll 操作后才会出现这个情况。poll 后会存在下面的状态



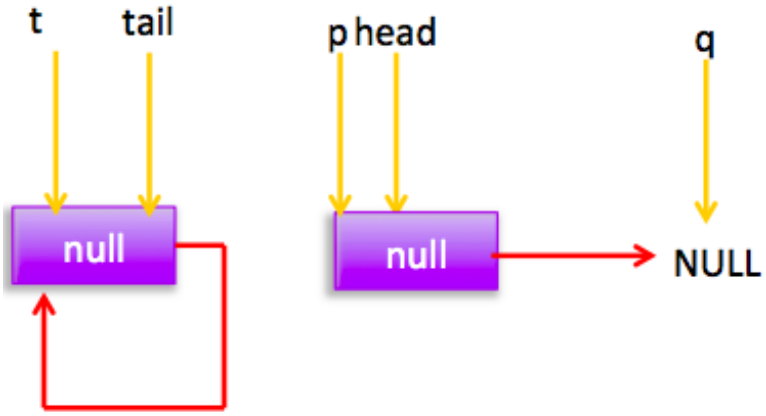
这个时候添加元素时候指针分布为:

http://www.itheima.com Copyright© 2018 黑马程序员

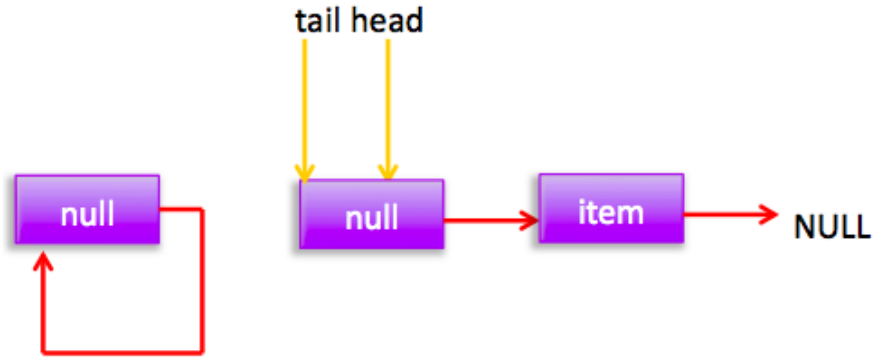


所以会执行（2）分支 结果 p=head

然后循环，循环后指针分布：



所以执行(1),然后 p!=t 所以设置 tail 节点。现在分布图：



自引用的节点会被垃圾回收掉。

http://www.itheima.com Copyright© 2018 黑马程序员

✓ **add 操作**

add 操作是在链表末尾添加一个元素，下面看看实现原理。

其实内部调用的还是 offer

```
public boolean add(E e) {
    return offer(e);
}
```

✓ **poll 操作**

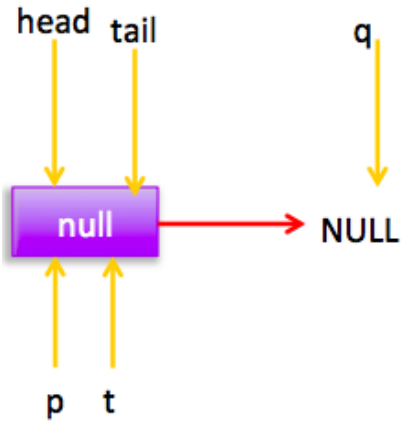
poll 操作是在链表头部获取并且移除一个元素，下面看看实现原理。

```
public E poll() {
    restartFromHead:

    //死循环
    for (;;) {

        //死循环
        for (Node<E> h = head, p = h, q;;) {

            //保存当前节点值
            E item = p.item;

            //当前节点有值则 cas 变为 null（1）
            if (item != null && p.casItem(item, null)) {
                //cas 成功标志当前节点以及从链表中移除
                if (p != h) // 类似 tail 间隔 2 设置一次头节点（2）
                    updateHead(h, ((q = p.next) != null) ? q : p);
                return item;
            }
            //当前队列为空则返回 null（3）
            else if ((q = p.next) == null) {
                updateHead(h, p);
                return null;
            }
            //自引用了，则重新找新的队列头节点（4）
            else if (p == q)
                continue restartFromHead;
            else//(5)
                p = q;
        }
```

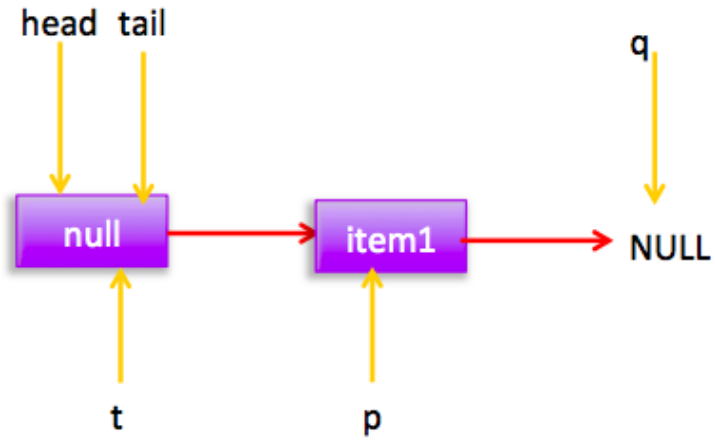

http://www.itheima.com Copyright© 2018 黑马程序员

```
        }
    }
    final void updateHead(Node<E> h, Node<E> p) {
        if (h != p && casHead(h, p))
            h.lazySetNext(h);
    }
```

当队列为空时候：

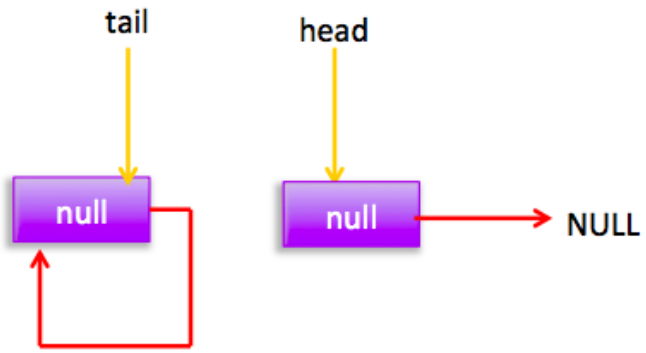


可知执行（3）这时候有两种情况，第一没有其他线程添加元素时候(3)结果为 true 然后因为 h!=p 为 false 所以直接返回 null。第二在执行 q=p.next 前，其他线程已经添加了一个元素到队列，这时候（3）返回 false，然后执行（5）p=q,然后循环后节点分布：

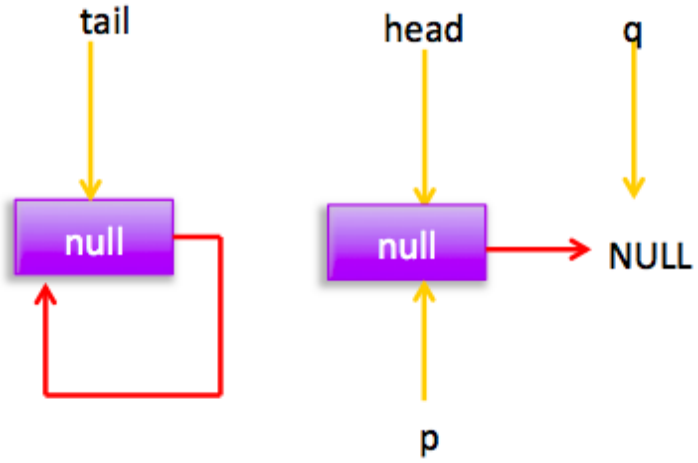


http://www.itheima.com Copyright© 2018 黑马程序员

这时候执行（1）分支，进行 cas 把当前节点值值为 null，同时只有一个线程会成功，cas 成功 标示该节点从队列中移除了，然后 p!=h,调用 updateHead 方法，参数为 h,p;h!=p 所以把 p 变为当前链表 head 节点，然后 h 节点的 next 指向自己。现在状态为：



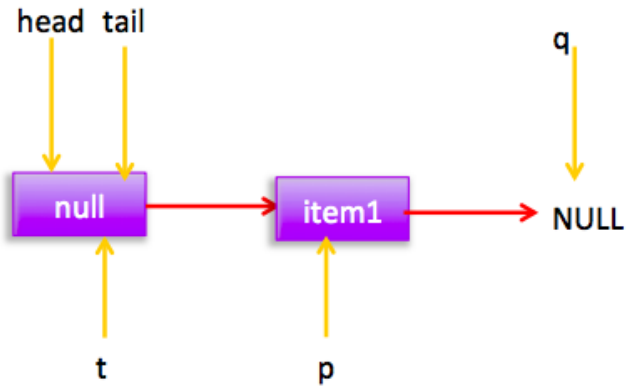
cas 失败 后 会再次循环，这时候分布图为：



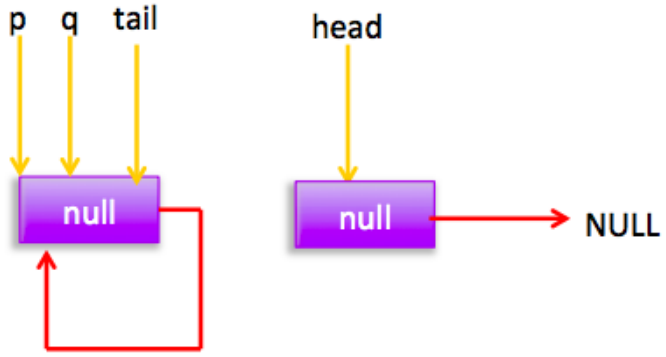
这时候执行（3）返回 null.

现在还有个分支（4）没有执行过，那么什么时候会执行那?

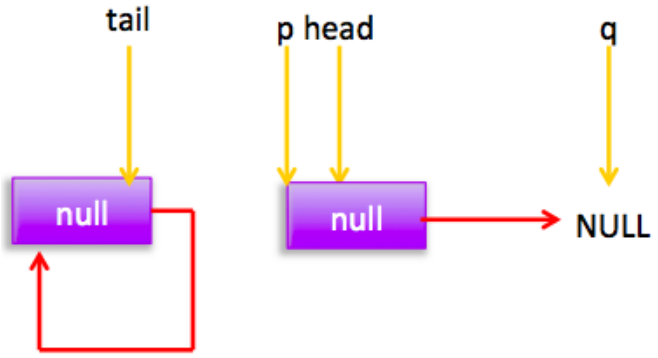
http://www.itheima.com Copyright© 2018 黑马程序员



这时候执行（1）分支，进行 cas 把当前节点值值为 null，同时只有一个线程 A 会成功，cas 成功 标示该节点从队列中移除了，然后 p!=h,调用 updateHead 方法，假如执行 updateHead 前另外一个线程 B 开始 poll 这时候它 p 指向为原来的 head 节点，然后当前线程 A 执行 updateHead 这时候 B 线程链表状态为：



所以会执行（4）重新跳到外层循环，获取当前 head,现在状态为：



- ✓ **peek 操作**

http://www.itheima.com Copyright© 2018 黑马程序员

peek 操作是获取链表头部一个元素（只读取不移除），下面看看实现原理。

代码与 poll 类似，只是少了 castItem.并且 peek 操作会改变 head 指向，offer 后 head 指向哨兵节点，第一次 peek 后 head 会指向第一个真的节点元素。

```
public E peek() {
    restartFromHead:
    for (;;) {
        for (Node<E> h = head, p = h, q;;) {
            E item = p.item;
            if (item != null || (q = p.next) == null) {
                updateHead(h, p);
                return item;
            }
            else if (p == q)
                continue restartFromHead;
            else
                p = q;
        }
    }
}
```

✓ **size 操作**

获取当前队列元素个数，在并发环境下不是很有用，因为使用 CAS 没有加锁所以从调用 size 函数到返回结果期间有可能增删元素，导致统计的元素个数不精确。

```
public int size() {
    int count = 0;
    for (Node<E> p = first(); p != null; p = succ(p))
        if (p.item != null)
            // 最大返回 Integer.MAX_VALUE
            if (++count == Integer.MAX_VALUE)
                break;
    return count;
}

//获取第一个队列元素（哨兵元素不算），没有则为 null
Node<E> first() {
    restartFromHead:
    for (;;) {
        for (Node<E> h = head, p = h, q;;) {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
                boolean hasItem = (p.item != null);
                if (hasItem || (q = p.next) == null) {
                    updateHead(h, p);
                    return hasItem ? p : null;
                }
                else if (p == q)
                    continue restartFromHead;
                else
                    p = q;
            }
        }
    }

    //获取当前节点的 next 元素，如果是自引入节点则返回真正头节点
    final Node<E> succ(Node<E> p) {
        Node<E> next = p.next;
        return (p == next) ? head : next;
    }
```

✓ **remove 操作**

如果队列里面存在该元素则删除给元素，如果存在多个则删除第一个，并返回 true，否者返回 false

```
    public boolean remove(Object o) {

        //查找元素为空，直接返回 false
        if (o == null) return false;
        Node<E> pred = null;
        for (Node<E> p = first(); p != null; p = succ(p)) {
            E item = p.item;

            //相等则使用 cas 值 null,同时一个线程成功，失败的线程循环查找队列中其他元素是否有匹配的。
            if (item != null &&
                o.equals(item) &&
                p.casItem(item, null)) {

                //获取 next 元素
                Node<E> next = succ(p);

                //如果有前驱节点，并且 next 不为空则链接前驱节点到 next,
                if (pred != null && next != null)
                    pred.casNext(p, next);
                return true;
            }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            pred = p;
        }
        return false;
    }
```

✓ **contains 操作**

判断队列里面是否含有指定对象，由于是遍历整个队列，所以类似 size 不是那么精确，有可能调用该方法时候元素还在队列里面，但是遍历过程中才把该元素删除了，那么就会返回 false.

```
    public boolean contains(Object o) {
        if (o == null) return false;
        for (Node<E> p = first(); p != null; p = succ(p)) {
            E item = p.item;
            if (item != null && o.equals(item))
                return true;
        }
        return false;
    }
```

ConcurrentLinkedQuere 的 offer 方法有意思的问题

offer 中有个 判断 t != (t = tail) 假如 t=node1;tail=node2;并且 node1!=node2 那么这个判断是 true 还是 false 那，答案是 true，这个判断是看当前 t 是不是和 tail 相等，相等则返回 true 否者为 false，但是无论结果是啥执行后 t 的值都是 tail。

下面从字节码来分析下为啥。

- 一个例子

```
    public static void main(String[] args)  {
        int t = 2;
        int tail = 3;
        System.out.println(t != (t = tail));
    }
    结果为：true
```

- 字节码文件

```
    C:\Users\Simple\Desktop\TeacherCode\Crm_Test\build\classes\com\itheima\crm\util>javap -c Test001
    警告：二进制文件 Test001 包含 com.itheima.crm.util.Test001
    Compiled from "Test001.java"
    public class com.itheima.crm.util.Test001 {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
  public com.itheima.crm.util.Test001();
    Code:
      0: aload_0
      1: invokespecial #8                  // Method java/lang/Object."<init>":()V
      4: return

  public static void main(java.lang.String[]);
    Code:
      0: iconst_2
      1: istore_1
      2: iconst_3
      3: istore_2
      4: getstatic     #16                 // Field java/lang/System.out:Ljava/io/PrintStream;
      7: iload_1
      8: iload_2
      9: dup
     10: istore_1
     11: if_icmpeq     18
     14: iconst_1
     15: goto          19
     18: iconst_0
     19: invokevirtual #22                 // Method java/io/PrintStream.println:(Z)V
     22: return
}
```

我们从上面标黄的字节码文件中分析

一开始栈为空：

栈

- 第 0 行指令作用是把值 2 入栈栈顶元素为 2

http://www.itheima.com Copyright© 2018 黑马程序员

栈

- 第 1 行指令作用是将栈顶 int 类型值保存到局部变量 t 中

栈

- 第 2 行指令作用是把值 3 入栈栈顶元素为 3

栈

- 第 3 行指令作用是将栈顶 int 类型值保存到局部变量 tail 中。

栈

http://www.itheima.com Copyright© 2018 黑马程序员

- 第 4 调用打印命令
- 第 7 行指令作用是把变量 t 中的值入栈

栈

- 第 8 行指令作用是把变量 tail 中的值入栈

栈

- 现在栈里面的元素为 3、2，并且 3 位于栈顶
- 第 9 行指令作用是当前栈顶元素入栈，所以现在栈内容 3，3，2

栈

http://www.itheima.com Copyright© 2018 黑马程序员

- 第 10 行指令作用是把栈顶元素存放到 t，现在栈内容 3，2

栈

- 第 11 行指令作用是判断栈顶两个元素值，相等则跳转 18。由于现在栈顶严肃为 3，2 不相等所以返回 true.
- 第 14 行指令作用是把 1 入栈
- 然后回头分析下!=是双目运算符，应该是首先把左边的操作数入栈，然后在去计算了右侧操作数。

ConcurrentLinkedQuere 总结

ConcurrentLinkedQueue 使用 CAS 非阻塞算法实现使用 CAS 解决了当前节点与 next 节点之间的安全链接和对当前节点值的赋值。由于使用 CAS 没有使用锁，所以获取 size 的时候有可能进行 offer，poll 或者 remove 操作，导致获取的元素个数不精确，所以在并发情况下 size 函数不是很有用。另外第一次 peek 或者 first 时候会把 head 指向第一个真正的队列元素。

下面总结下如何实现线程安全的，可知入队出队函数都是操作 volatile 变量：head，tail。所以要保证队列线程安全只需要保证对这两个 Node 操作的可见性和原子性，由于 volatile 本身保证可见性，所以只需要看下多线程下如果保证对着两个变量操作的原子性。

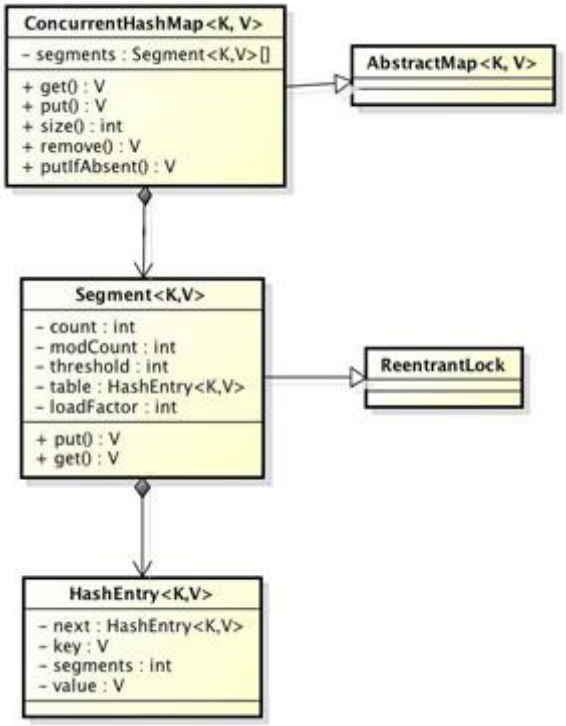
对于 offer 操作是在 tail 后面添加元素，也就是调用 tail.casNext 方法，而这个方法是使用的 CAS 操作，只有一个线程会成功，然后失败的线程会循环一下，重新获取 tail，然后执行 casNext 方法。对于 poll 也是这样的。

http://www.itheima.com Copyright© 2018 黑马程序员

➢ConcurrentHashMap 非阻塞 Hash 集合

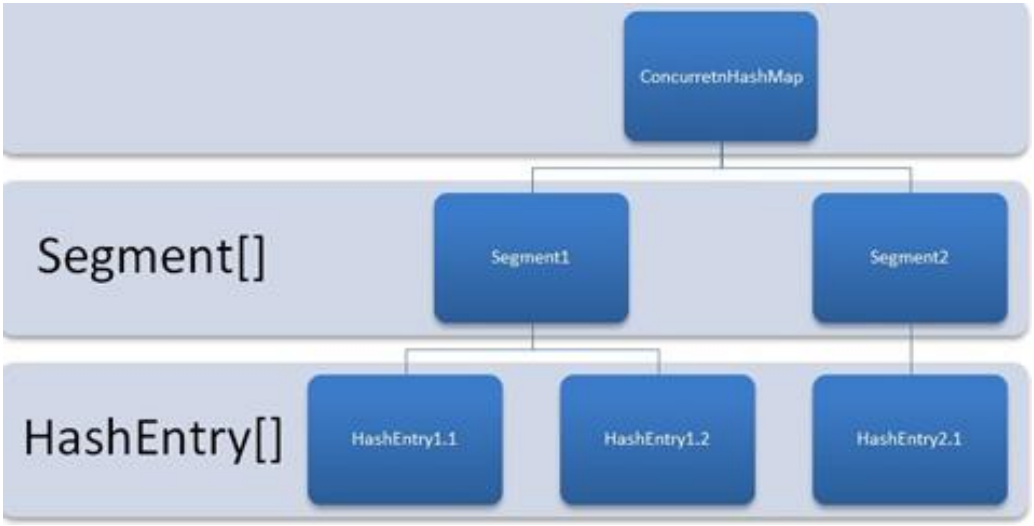
ConcurrentHashMap 是 Java 并发包中提供的一个线程安全且高效的 HashMap 实现，ConcurrentHashMap 在并发编程的场景中使用频率非常之高，本文就来分析下 ConcurrentHashMap 的实现原理，并对其实现原理进行分析。

ConcurrentLinkedQuere 类图



ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。Segment 是一种可重入锁 ReentrantLock，在 ConcurrentHashMap 里扮演锁的角色，HashEntry 则用于存储键值对数据。一个 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，每个 Segment 守护者一个 HashEntry 数组里的元素,当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁。

http://www.itheima.com Copyright© 2018 黑马程序员



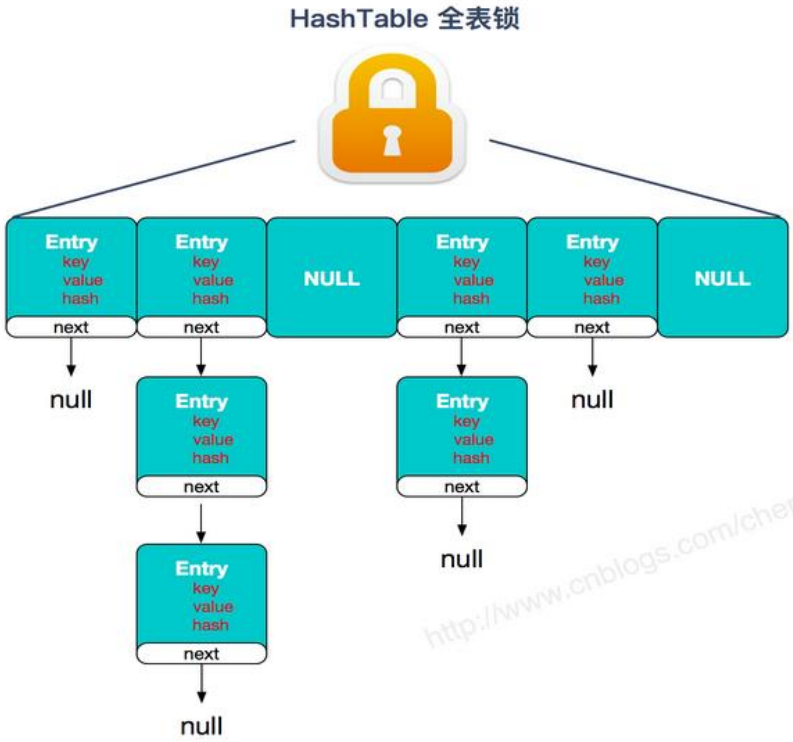
ConcurrentLinkedQuere 实现原理

众所周知，哈希表是中非常高效，复杂度为 O(1)的数据结构，在 Java 开发中，我们最常见到最频繁使用的就是 HashMap 和 HashTable，但是在线程竞争激烈的并发场景中使用都不够合理。

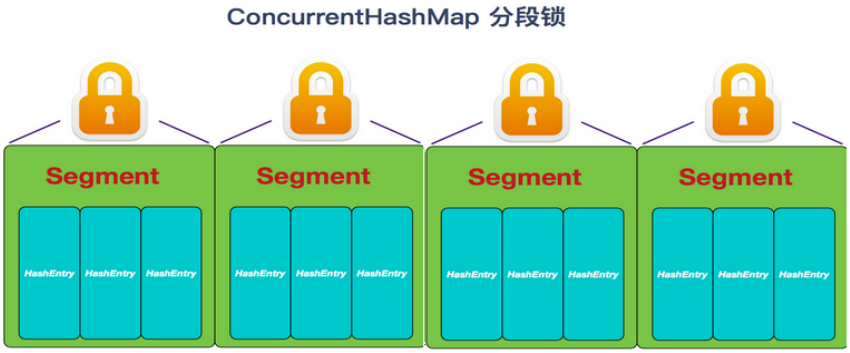
HashMap ：先说 HashMap，HashMap 是线程不安全的，在并发环境下，可能会形成环状链表（扩容时可能造成，具体原因自行百度 google 或查看源码分析），导致 get 操作时，cpu 空转，所以，在并发环境中使用 HashMap 是非常危险的。

HashTable ： HashTable 和 HashMap 的实现原理几乎一样，差别无非是 1.HashTable 不允许 key 和 value 为 null；2.HashTable 是线程安全的。但是 HashTable 线程安全的策略实现代价却太大了，简单粗暴，get/put 所有相关操作都是 synchronized 的，这相当于给整个哈希表加了一把大锁，多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞，相当于将所有的操作串行化，在竞争激烈的并发场景中性能就会非常差。如下图

http://www.itheima.com Copyright© 2018 黑马程序员



HashTable 性能差主要是由于所有操作需要竞争同一把锁，而如果容器中有多把锁，每一把锁锁一段数据，这样在多线程访问时不同段的数据时，就不会存在锁竞争了，这样便可以有效地提高并发效率。这就是ConcurrentHashMap 所采用的"分段锁"思想



ConcurrentLinkedQuere 源码解析

ConcurrentHashMap 采用了非常精妙的"分段锁"策略，ConcurrentHashMap 的主干是个 Segment 数组。

```
final Segment<K,V>[] segments;
```

Segment 继承了 ReentrantLock，所以它就是一种可重入锁（ReentrantLock）。在 ConcurrentHashMap，

 Copyright© 2018 黑马程序员

一个 Segment 就是一个子哈希表，Segment 里维护了一个 HashEntry 数组，并发环境下，对于不同 Segment 的数据进行操作是不用考虑锁竞争的。（就按默认的 ConcurrentLeve 为 16 来讲，理论上就允许 16 个线程并发执行，有木有很酷）

所以，对于同一个 Segment 的操作才需考虑线程同步，不同的 Segment 则无需考虑。Segment 类似于 HashMap，一个 Segment 维护着一个 HashEntry 数组

```
transient volatile HashEntry<K,V>[] table;
```

HashEntry 是目前我们提到的最小的逻辑处理单元了。一个 ConcurrentHashMap 维护一个 Segment 数组，一个 Segment 维护一个 HashEntry 数组。

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

我们说 Segment 类似哈希表，那么一些属性就跟我们之前提到的 HashMap 差不离，比如负载因子 loadFactor，比如阈值 threshold 等等，看下 Segment 的构造方法

```
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
        }
```

我们来看下 ConcurrentHashMap 的构造方法

```
public ConcurrentHashMap(int initialCapacity,
                         float loadFactor, int concurrencyLevel) {
        if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
            throw new IllegalArgumentException();
        //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;
        //ssize 为 segments 数组长度，根据 concurrentLevel 计算得出
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            int ssize = 1;
            while (ssize < concurrencyLevel) {
                ++sshift;
                ssize <<= 1;
            }
            //segmentShift 和 segmentMask 这两个变量在定位 segment 时会用到，后面会详细讲
            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;
            //创建 segments 数组并初始化第一个 Segment，其余的 Segment 延迟初始化
            Segment<K,V> s0 =
                new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
                                 (HashEntry<K,V>[])new HashEntry[cap]);
            Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
            UNSAFE.putOrderedObject(ss, SBASE, s0);
            this.segments = ss;
        }
```

初始化方法有三个参数，如果用户不指定则会使用默认值，initialCapacity 为 16，loadFactor 为 0.75（负载因子，扩容时需要参考），concurrentLevel 为 16。

从上面的代码可以看出来,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。

其实，put 方法对 segment 也会有所体现

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


http://www.itheima.com Copyright© 2018 黑马程序员

```
        //concurrentHashMap 不允许 key/value 为空
        if (value == null)
            throw new NullPointerException();
        //hash 函数对 key 的 hashCode 重新散列，避免差劲的不合理的 hashcode，保证散列均匀
        int hash = hash(key);
        //返回的 hash 值无符号右移 segmentShift 位与段掩码进行位运算，定位 segment
        int j = (hash >>> segmentShift) & segmentMask;
        if ((s = (Segment<K,V>)UNSAFE.getObject          // nonvolatile; recheck
             (segments, (j << SSHIFT) + SBASE)) == null) //  in ensureSegment
            s = ensureSegment(j);
        return s.put(key, hash, value, false);
    }
```

从源码看出，put 的主要逻辑也就两步：

1.定位 segment 并确保定位的 Segment 已初始化

2.调用 Segment 的 put 方法。

Ps：关于 segmentShift 和 segmentMask

segmentShift 和 segmentMask 这两个全局变量的主要作用是用来定位 Segment，int j =(hash >>> segmentShift) & segmentMask。

segmentMask：段掩码，假如 segments 数组长度为 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;若 segments 长度为 32，segmentShift=32-5=27。而计算得出的 hash 值最大为 32 位，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。

ConcurrentLinkedQuere 方法

✓ **Get 操作**

```
    public V get(Object key) {
        Segment<K,V> s;
        HashEntry<K,V>[] tab;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        int h = hash(key);
        long u = (((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;
        //先定位 Segment，再定位 HashEntry
        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;
    }
```

get 方法无需加锁，由于其中涉及到的共享变量都使用 volatile 修饰，volatile 可以保证内存可见性，所以不会读取到过期数据。

来看下 concurrentHashMap 代理到 Segment 上的 put 方法，Segment 中的 put 方法是要加锁的。只不过是锁粒度细了而已。

✓ **Put 操作**

```
    final V put(K key, int hash, V value, boolean onlyIfAbsent) {
            HashEntry<K,V> node = tryLock() ? null :
                scanAndLockForPut(key, hash, value);//tryLock 不成功时会遍历定位到的 HashEnry 位置的链表（遍历主要是为了使 CPU 缓存链表），若找不到，则创建 HashEntry。tryLock 一定次数后（MAX_SCAN_RETRIES 变量决定），则 lock。若遍历过程中，由于其他线程的操作导致链表头结点变化，则需要重新遍历。
            V oldValue;
            try {
                HashEntry<K,V>[] tab = table;
                int index = (tab.length - 1) & hash;//定位 HashEntry，可以看到，这个 hash 值在定位 Segment 时和在 Segment 中定位 HashEntry 都会用到，只不过定位 Segment 时只用到高几位。
                HashEntry<K,V> first = entryAt(tab, index);
                for (HashEntry<K,V> e = first;;) {
                    if (e != null) {
                        K k;
                        if ((k = e.key) == key ||
                            (e.hash == hash && key.equals(k))) {
                            oldValue = e.value;
                            if (!onlyIfAbsent) {
```


 Copyright© 2018 黑马程序员

```
                            e.value = value;
                            ++modCount;
                        }
                        break;
                    }
                    e = e.next;
                }
                else {
                    if (node != null)
                        node.setNext(first);
                    else
                        node = new HashEntry<K,V>(hash, key, value, first);
                    int c = count + 1;
                    //若 c 超出阈值 threshold，需要扩容并 rehash。扩容后的容量是当前容量的 2 倍。这样可以最大程度避免之前散列好的 entry 重新散列，具体在另一篇文章中有详细分析，不赘述。扩容并 rehash 的这个过程是比较消耗资源的。
                    if (c > threshold && tab.length < MAXIMUM_CAPACITY)
                        rehash(node);
                    else
                        setEntryAt(tab, index, node);
                    ++modCount;
                    count = c;
                    oldValue = null;
                    break;
                }
            }
        } finally {
            unlock();
        }
        return oldValue;
    }
```

ConcurrentLinkedQuere 总结

ConcurrentHashMap 作为一种线程安全且高效的哈希表的解决方案，尤其其中的"分段锁"的方案，相比 HashTable 的全表锁在性能上的提升非常之大。本文对 ConcurrentHashMap 的实现原理进行了详细分析，并解读了部分源码，希望能帮助到有需要的童鞋。

➢ConcurrentSkipListMap 非阻塞 Hash 跳表集合

大家都是知道 TreeMap，它是使用树形结构的方式进行存储数据的线程不安全的 Map 集合（有序的哈希表），

http://www.itheima.com Copyright© 2018 黑马程序员

并且可以对 Map 中的 Key 进行排序，Key 中存储的数据需要实现 Comparator 接口或使用 CompareAble 接口的子类来实现排序。

ConcurrentSkipListMap 也是和 TreeMap，它们都是有序的哈希表。但是，它们是有区别的：

第一，它们的线程安全机制不同，TreeMap 是非线程安全的，而 ConcurrentSkipListMap 是线程安全的。

第二，ConcurrentSkipListMap 是通过跳表实现的，而 TreeMap 是通过红黑树实现的。

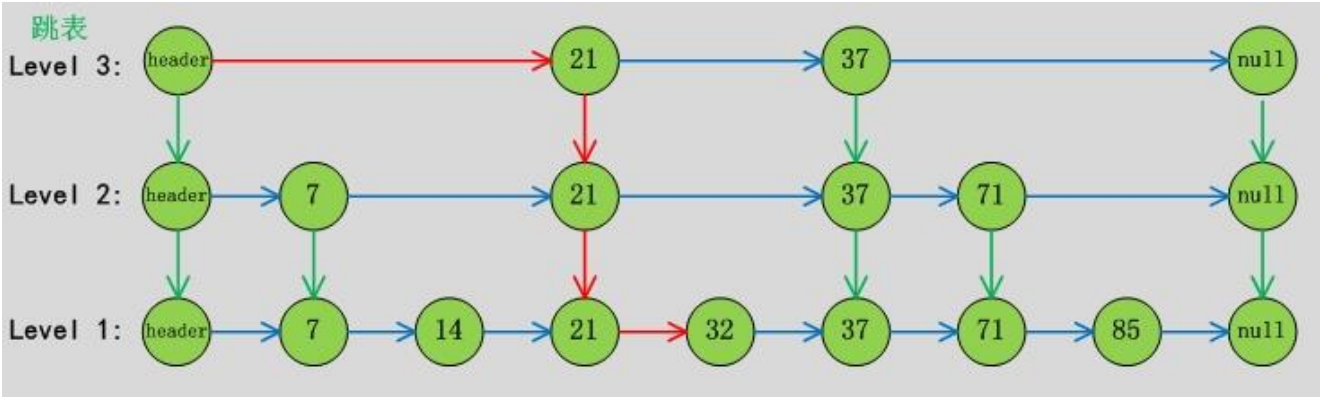
那现在我们需要知道说明是跳表。

什么是 SkipList

Skip list(跳表）是一种可以代替平衡树的数据结构，默认是按照 Key 值升序的。Skip list 让已排序的数据分布在多层链表中，以 0-1 随机数决定一个数据的向上攀升与否，通过“空间来换取时间”的一个算法，在每个节点中增加了向前的指针，在插入、删除、查找时可以忽略一些不可能涉及到的结点，从而提高了效率。

从概率上保持数据结构的平衡比显示的保持数据结构平衡要简单的多。对于大多数应用，用 Skip list 要比用树算法相对简单。由于 Skip list 比较简单，实现起来会比较容易，虽然和平衡树有着相同的时间复杂度(O(logn)),但是 skip list 的常数项会相对小很多。Skip list 在空间上也比较节省。一个节点平均只需要 1.333 个指针（甚至更少）。

下图为 Skip list 结构图（以 7,14,21,32,37,71,85 序列为例）



SkipList 性质

(1) 由很多层结构组成，level 是通过一定的概率随机产生的。

http://www.itheima.com Copyright© 2018 黑马程序员

(2) 每一层都是一个有序的链表，默认是升序，也可以根据创建映射时所提供的 Comparator 进行排序，具体取决于使用的构造方法。

(3) 最底层(Level 1)的链表包含所有元素。

(4) 如果一个元素出现在 Level i 的链表中，则它在 Level i 之下的链表也都会出现。

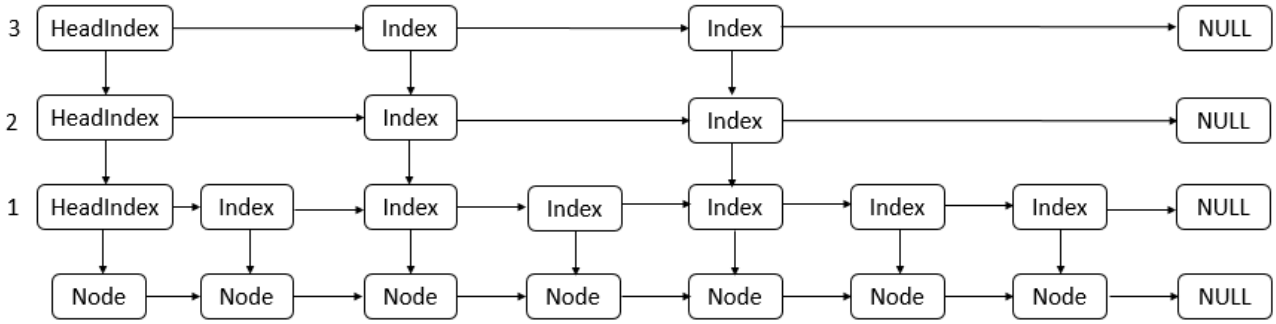
(5) 每个节点包含两个指针，一个指向同一链表中的下一个元素，一个指向下面一层的元素。

什么是 ConcurrentSkipListMap

ConcurrentSkipListMap 提供了一种线程安全的并发访问的排序映射表。内部是 SkipList (跳表) 结构实现，在理论上能够在 O(log(n))时间内完成查找、插入、删除操作。 注意，调用 ConcurrentSkipListMap 的 size 时，由于多个线程可以同时对映射表进行操作，所以映射表需要遍历整个链表才能返回元素个数，这个操作是个 O(log(n))的操作。

ConcurrentSkipListMap 数据结构

ConcurrentSkipListMap 的数据结构，如下图所示：



说明：可以看到 ConcurrentSkipListMap 的数据结构使用的是跳表，每一个 HeadIndex、Index 结点都会包含一个对 Node 的引用，同一垂直方向上的 Index、HeadIndex 结点都包含了最底层的 Node 结点的引用。并且层级越高，该层级的结点（HeadIndex 和 Index）数越少。Node 结点之间使用单链表结构。

ConcurrentSkipListMap 源码分析

ConcurrentSkipListMap 主要用到了 Node 和 Index 两种节点的存储方式，通过 volatile 关键字实现了并发的操

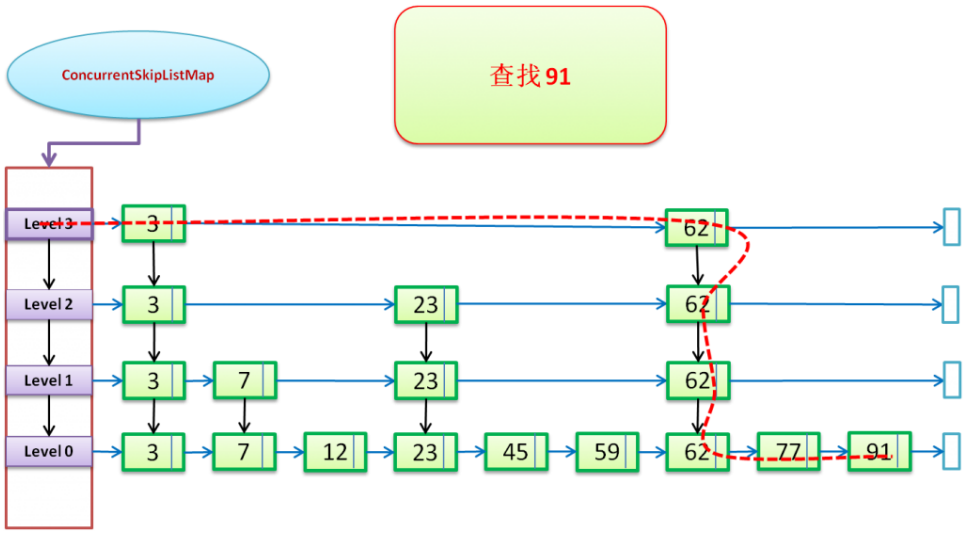
http://www.itheima.com Copyright© 2018 黑马程序员

作

```
static final class Node<K,V> {
        final K key;
        volatile Object value;//value 值
        volatile Node<K,V> next;//next 引用
        ……
}
static class Index<K,V> {
        final Node<K,V> node;
        final Index<K,V> down;//downy 引用
       volatile Index<K,V> right;//右边引用
       ……
}
```

- ✓ **ConcurrentSkipListMap 查找操作**

通过 SkipList 的方式进行查找操作：（下图以“查找 91”进行说明：）



红色虚线，表示查找的路径，蓝色向右箭头表示 right 引用；黑色向下箭头表示 down 引用；

下面就是源码中的实现（get 方法是通过 doGet 方法来时实现的）

```
public V get(Object key) {
      return doGet(key);
 }
 //doGet 的实现
```


 Copyright© 2018 黑马程序员

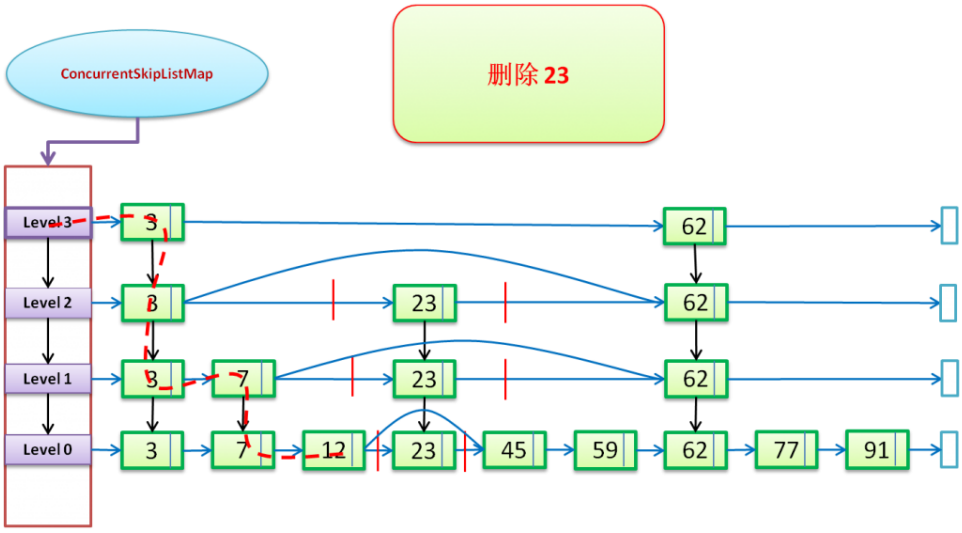
```
private V doGet(Object okey) {
        Comparable<? super K> key = comparable(okey);
        Node<K,V> bound = null;
        Index<K,V> q = head;//把头结点作为当前节点的前驱节点
        Index<K,V> r = q.right;//前驱节点的右节点作为当前节点
        Node<K,V> n;
        K k;
        int c;
        for (;;) {//遍历
            Index<K,V> d;
            // 依次遍历 right 节点
            if (r != null && (n = r.node) != bound && (k = n.key) != null) {
                if ((c = key.compareTo(k)) > 0) {//由于 key 都是升序排列的，所有当前关键字大于所要
查找的 key 时继续向右遍历
                    q = r;
                    r = r.right;
                    continue;
                } else if (c == 0) {
                    //如果找到了相等的 key 节点，则返回该 Node 的 value 如果 value 为空可能是其他并发 delete
导致的，于是通过另一种
                    //遍历 findNode 的方式再查找
                    Object v = n.value;
                    return (v != null)? (V)v : getUsingFindNode(key);
                } else
                    bound = n;
            }
            //如果一个链表中 right 没能找到 key 对应的 value，则调整到其 down 的引用处继续查找
            if ((d = q.down) != null) {
                q = d;
                r = d.right;
            } else
                break;
        }
        // 如果通过上面的遍历方式，还没能找到 key 对应的 value，再通过 Node.next 的方式进行查找
        for (n = q.node.next;  n != null; n = n.next) {
            if ((k = n.key) != null) {
                if ((c = key.compareTo(k)) == 0) {
                    Object v = n.value;
                    return (v != null)? (V)v : getUsingFindNode(key);
                } else if (c < 0)
                    break;
            }
        }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            return null;
        }
```

✓ **ConcurrentSkipListMap 删除操作**

通过 SkipList 的方式进行删除操作：（下图以“删除 23”进行说明：）



红色虚线，表示查找的路径，蓝色向右箭头表示 right 引用；黑色向下箭头表示 down 引用；

下面就是源码中的实现（remove 方法是通过 doRemove 方法来时实现的）

```
    //remove 操作，通过 doRemove 实现，把所有 level 中出现关键字 key 的地方都 delete 掉
    public V remove(Object key) {
            return doRemove(key, null);
     }
     final V doRemove(Object okey, Object value) {
            Comparable<? super K> key = comparable(okey);
            for (;;) {
                Node<K,V> b = findPredecessor(key);//得到 key 的前驱（就是比 key 小的最大节点）
                Node<K,V> n = b.next;//前驱节点的 next 引用
                for (;;) {//遍历
                    if (n == null)//如果 next 引用为空，直接返回
                        return null;
                    Node<K,V> f = n.next;
                    if (n != b.next)                    // 如果两次获得的 b.next 不是相同的 Node，就跳转
    到第一层循环重新获得 b 和 n
                        break;
                    Object v = n.value;
                    if (v == null) {                    // 当 n 被其他线程 delete 的时候，其 value==null，
此时做辅助处理，并重新获取 b 和 n
                        n.helpDelete(b, f);
```

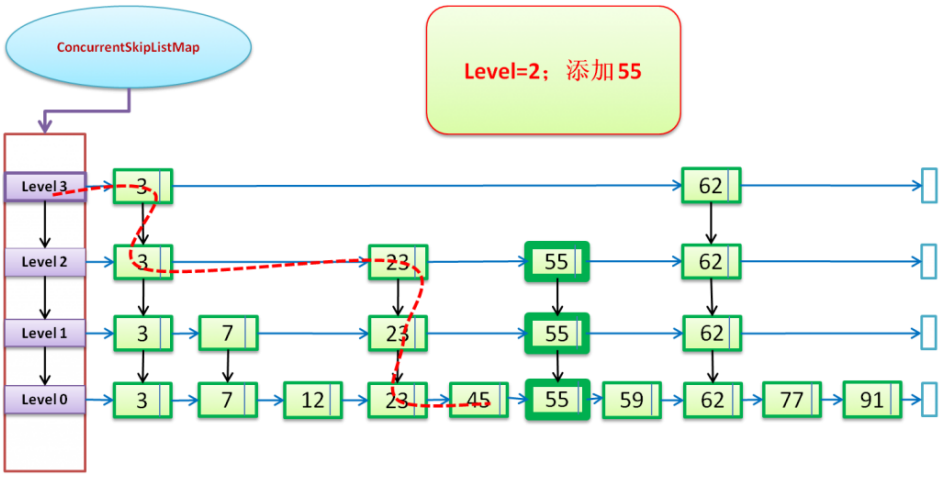

 Copyright© 2018 黑马程序员

```
                    break;
                }
                if (v == n || b.value == null)       // 当其前驱被 delet 的时候直接跳出，重新获取 b 和 n
                    break;
                int c = key.compareTo(n.key);
                if (c < 0)
                    return null;
                if (c > 0) {//当 key 较大时就继续遍历
                    b = n;
                    n = f;
                    continue;
                }
                if (value != null && !value.equals(v))
                    return null;
                if (!n.casValue(v, null))
                    break;
                if (!n.appendMarker(f) || !b.casNext(n, f))//casNext 方法就是通过比较和设置 b（前驱）
的 next 节点的方式来实现删除操作
                    findNode(key);                   // 通过尝试 findNode 的方式继续 find
                else {
                    findPredecessor(key);            // Clean index
                    if (head.right == null)    //如果 head 的 right 引用为空，则表示不存在该 level
                        tryReduceLevel();
                }
                return (V)v;
            }
        }
    }
```

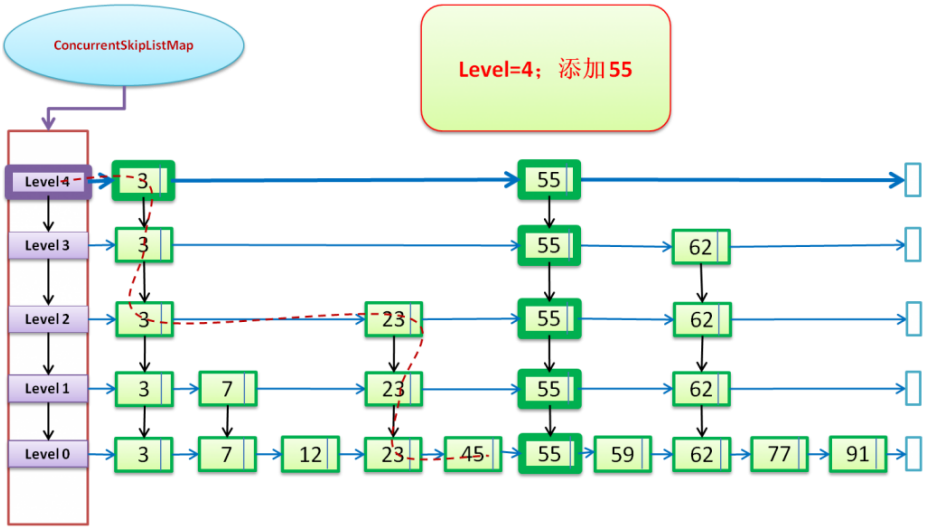
✓ **ConcurrentSkipListMap 插入操作**

通过 SkipList 的方式进行插入操作：（下图以“添加 55”的两种情况，进行说明：）

http://www.itheima.com Copyright© 2018 黑马程序员



在 level=2（该 level 存在）的情况下添加 55 的图示:只需在 level<=2 的合适位置插入 55 即可



在 level=4(该 level 不存在，图示 level4 是新建的)的情况下添加 55 的情况：首先新建 level4,然后在 level<=4 的合适位置插入 55。

下面就是源码中的实现（put 方法是通过 doPut 方法来时实现的）

```
//put 操作，通过 doPut 实现
 public V put(K key, V value) {
        if (value == null)
            throw new NullPointerException();
        return doPut(key, value, false);
 }
private V doPut(K kkey, V value, boolean onlyIfAbsent) {
        Comparable<? super K> key = comparable(kkey);
        for (;;) {
            Node<K,V> b = findPredecessor(key);//前驱
```


 Copyright© 2018 黑马程序员

```
            Node<K,V> n = b.next;
            //定位的过程就是和 get 操作相似
            for (;;) {
                if (n != null) {
                    Node<K,V> f = n.next;
                    if (n != b.next) // 前后值不一致的情况下，跳转到第一层循环重新获得 b 和 n
                        break;;
                    Object v = n.value;
                    if (v == null) {                // n 被 delete 的情况下
                        n.helpDelete(b, f);
                        break;
                    }
                    if (v == n || b.value == null) // b 被 delete 的情况，重新获取 b 和 n
                        break;
                    int c = key.compareTo(n.key);
                    if (c > 0) {
                        b = n;
                        n = f;
                        continue;
                    }
                    if (c == 0) {
                        if (onlyIfAbsent || n.casValue(v, value))
                            return (V)v;
                        else
                            break; // restart if lost race to replace value
                    }
                    // else c < 0; fall through
                }
                Node<K,V> z = new Node<K,V>(kkey, value, n);
                if (!b.casNext(n, z))
                    break;          // restart if lost race to append to b
                int level = randomLevel();//得到一个随机的 level 作为该 key-value 插入的最高 level
                if (level > 0)
                    insertIndex(z, level);//进行插入操作
                return null;
            }
        }
    }

    /**
     * 获得一个随机的 level 值
     */
    private int randomLevel() {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            int x = randomSeed;
            x ^= x << 13;
            x ^= x >>> 17;
            randomSeed = x ^= x << 5;
            if ((x & 0x8001) != 0) // test highest and lowest bits
                return 0;
            int level = 1;
            while (((x >>>= 1) & 1) != 0) ++level;
            return level;
        }
    //执行插入操作：如上图所示，有两种可能的情况：
    //1.当 level 存在时，对 level<=n 都执行 insert 操作
    //2.当 level 不存在（大于目前的最大 level）时，首先添加新的 level，然后在执行操作 1
    private void insertIndex(Node<K,V> z, int level) {
            HeadIndex<K,V> h = head;
            int max = h.level;
            if (level <= max) {//情况 1
                Index<K,V> idx = null;
                for (int i = 1; i <= level; ++i)//首先得到一个包含 1~level 个级别的 down 关系的链表，
    最后的 inx 为最高 level
                    idx = new Index<K,V>(z, idx, null);
                addIndex(idx, h, level);//把最高 level 的 idx 传给 addIndex 方法
            } else { // 情况 2 增加一个新的级别
                level = max + 1;
                Index<K,V>[] idxs = (Index<K,V>[])new Index[level+1];
                Index<K,V> idx = null;
                for (int i = 1; i <= level; ++i)//该步骤和情况 1 类似
                    idxs[i] = idx = new Index<K,V>(z, idx, null);
                HeadIndex<K,V> oldh;
                int k;
                for (;;) {
                    oldh = head;
                    int oldLevel = oldh.level;
                    if (level <= oldLevel) { // lost race to add level
                        k = level;
                        break;
                    }
                    HeadIndex<K,V> newh = oldh;
                    Node<K,V> oldbase = oldh.node;
                    for (int j = oldLevel+1; j <= level; ++j)
                        newh = new HeadIndex<K,V>(oldbase, newh, idxs[j], j);//创建新的
                    if (casHead(oldh, newh)) {
                        k = oldLevel;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
                    break;
                }
            }
            addIndex(idxs[k], oldh, k);
        }
    }
/**
     *在 1~indexlevel 层中插入数据
     */
    private void addIndex(Index<K,V> idx, HeadIndex<K,V> h, int indexLevel) {
        //  insertionLevel 代表要插入的 level，该值会在 indexLevel~1 间遍历一遍
        int insertionLevel = indexLevel;
        Comparable<? super K> key = comparable(idx.node.key);
        if (key == null) throw new NullPointerException();
        // 和 get 操作类似，不同的就是查找的同时在各个 level 上加入了对应的 key
        for (;;) {
            int j = h.level;
            Index<K,V> q = h;
            Index<K,V> r = q.right;
            Index<K,V> t = idx;
            for (;;) {
                if (r != null) {
                    Node<K,V> n = r.node;
                    // compare before deletion check avoids needing recheck
                    int c = key.compareTo(n.key);
                    if (n.value == null) {
                        if (!q.unlink(r))
                            break;
                        r = q.right;
                        continue;
                    }
                    if (c > 0) {
                        q = r;
                        r = r.right;
                        continue;
                    }
                }
                if (j == insertionLevel) {//在该层 level 中执行插入操作
                    // Don't insert index if node already deleted
                    if (t.indexesDeletedNode()) {
                        findNode(key); // cleans up
                        return;
                    }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
                    if (!q.link(r, t))//执行 link 操作，其实就是 inset 的实现部分
                        break; // restart
                    if (--insertionLevel == 0) {
                        // need final deletion check before return
                        if (t.indexesDeletedNode())
                            findNode(key);
                        return;
                    }
                }
                if (--j >= insertionLevel && j < indexLevel)//key 移动到下一层 level
                    t = t.down;
                q = q.down;
                r = q.right;
            }
        }
    }
```

ConcurrentLinkedQuere 示例

下面我们看下面示例输出的结果

```
import java.util.*;import java.util.concurrent.*;
/*
 *   ConcurrentSkipListMap 是"线程安全"的哈希表，而 TreeMap 是非线程安全的。
 *
 *   下面是"多个线程同时操作并且遍历 map"的示例
 *   (01) 当 map 是 ConcurrentSkipListMap 对象时，程序能正常运行。
 *   (02) 当 map 是 TreeMap 对象时，程序会产生 ConcurrentModificationException 异常。
 *
 */public class ConcurrentSkipListMapDemo1 {

    // TODO: map 是 TreeMap 对象时，程序会出错。
    //private static Map<String, String> map = new TreeMap<String, String>();
    private static Map<String, String> map = new ConcurrentSkipListMap<String, String>();
    public static void main(String[] args) {

        // 同时启动两个线程对 map 进行操作!
        new MyThread("a").start();
        new MyThread("b").start();
    }

    private static void printAll() {
        String key, value;
        Iterator iter = map.entrySet().iterator();
```


 Copyright© 2018 黑马程序员

```
            while(iter.hasNext()) {
                Map.Entry entry = (Map.Entry)iter.next();
                key = (String)entry.getKey();
                value = (String)entry.getValue();
                System.out.print("("+key+", "+value+"), ");
            }
            System.out.println();
        }

        private static class MyThread extends Thread {
            MyThread(String name) {
                super(name);
            }
            @Override
            public void run() {
                    int i = 0;
                while (i++ < 6) {
                    // “线程名” + "序号"
                    String val = Thread.currentThread().getName()+i;
                    map.put(val, "0");
                    // 通过“Iterator”遍历 map。
                    printAll();
                }
            }
        }
    }
```

某一次的运行结果：

```
(a1, 0), (a1, 0), (b1, 0), (b1, 0),
(a1, 0), (b1, 0), (b2, 0),
(a1, 0), (a1, 0), (a2, 0), (a2, 0), (b1, 0), (b1, 0), (b2, 0), (b2, 0), (b3, 0),
(b3, 0), (a1, 0),
(a2, 0), (a3, 0), (a1, 0), (b1, 0), (a2, 0), (b2, 0), (a3, 0), (b3, 0), (b1, 0), (b4, 0),
(b2, 0), (a1, 0), (b3, 0), (a2, 0), (b4, 0),
(a3, 0), (a1, 0), (a4, 0), (a2, 0), (b1, 0), (a3, 0), (b2, 0), (a4, 0), (b3, 0), (b1, 0), (b4,
0), (b2, 0), (b5, 0),
(b3, 0), (a1, 0), (b4, 0), (a2, 0), (b5, 0),
(a3, 0), (a1, 0), (a4, 0), (a2, 0), (a5, 0), (a3, 0), (b1, 0), (a4, 0), (b2, 0), (a5, 0), (b3,
0), (b1, 0), (b4, 0), (b2, 0), (b5, 0), (b3, 0), (b6, 0),
(b4, 0), (a1, 0), (b5, 0), (a2, 0), (b6, 0),
(a3, 0), (a4, 0), (a5, 0), (a6, 0), (b1, 0), (b2, 0), (b3, 0), (b4, 0), (b5, 0), (b6, 0),
```


http://www.itheima.com Copyright© 2018 黑马程序员

结果说明：

示例程序中，启动两个线程(线程 a 和线程 b)分别对 ConcurrentSkipListMap 进行操作。以线程 a 而言，它会先获取“线程名”+“序号”，然后将该字符串作为 key，将“0”作为 value，插入到 ConcurrentSkipListMap 中；接着，遍历并输出 ConcurrentSkipListMap 中的全部元素。 线程 b 的操作和线程 a 一样，只不过线程 b 的名字和线程 a 的名字不同。

当 map 是 ConcurrentSkipListMap 对象时，程序能正常运行。如果将 map 改为 TreeMap 时，程序会产生 ConcurrentModificationException 异常。

2) java.util.concurrent.atomic 包

➢AtomicBoolean 原子性布尔

AtomicBoolean 是 java.util.concurrent.atomic 包下的原子变量，这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择一个另一个线程进入，这只是一种逻辑上的理解。实际上是借助硬件的相关指令来实现的，不会阻塞线程(或者说只是在硬件级别上阻塞了)。

AtomicBoolean，在这个 Boolean 值的变化的时候不允许在之间插入，保持操作的原子性。

下面将解释重点方法并举例：

boolean compareAndSet(expectedValue, updateValue)，这个方法主要两个作用:

1. 比较 AtomicBoolean 和 expect 的值，如果一致，执行方法内的语句。其实就是一个 if 语句

2. 把 AtomicBoolean 的值设成 update,比较最要的是这两件事是一气呵成的，这连个动作之间不会被打断，任何内部或者外部的语句都不可能在两个动作之间运行。为多线程的控制提供了解决的方案

下面我们从代码上解释：

 Copyright© 2018 黑马程序员

首先我们看下在不使用 AtomicBoolean 情况下，代码的运行情况：

```
package zmx.atomic.test;

import java.util.concurrent.TimeUnit;

public class BarWorker implements Runnable {
    //静态变量
    private static boolean exists = false;

    private String name;

    public BarWorker(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        if (!exists) {
                try {
                 TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e1) {
                 // do nothing
                }
                exists = true;
                System.out.println(name + " enter");
                try {
                 System.out.println(name + " working");
                 TimeUnit.SECONDS.sleep(2);
                } catch (InterruptedException e) {
                 // do nothing
                }
                System.out.println(name + " leave");
                exists = false;
       } else {
        System.out.println(name + " give up");
       }

    }


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


http://www.itheima.com Copyright© 2018 黑马程序员

```
            BarWorker bar1 = new BarWorker("bar1");
            BarWorker bar2 = new BarWorker("bar2");
            new Thread(bar1).start();
            new Thread(bar2).start();
        }
    }

    运行结果：
    bar1 enter
    bar2 enter
    bar1 working
    bar2 working
    bar1 leave
    bar2 leave
```

从上面的运行结果我们可看到，两个线程运行时，都对静态变量 exists 同时做操作，并没有保证 exists 静态变量的原子性，也就是一个线程在对静态变量 exists 进行操作到时候，其他线程必须等待或不作为。等待一个线程操作完后，才能对其进行操作。

下面我们将静态变量使用 AtomicBoolean 来进行操作

```
package zmx.atomic.test;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;

public class BarWorker2 implements Runnable {
    //静态变量使用 AtomicBoolean 进行操作
    private static AtomicBoolean exists = new AtomicBoolean(false);


     private String name;

     public BarWorker2(String name) {
          this.name = name;
     }

     @Override
     public void run() {
         if (exists.compareAndSet(false, true)) {

             System.out.println(name + " enter");
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
                try {
                     System.out.println(name + " working");
                     TimeUnit.SECONDS.sleep(2);
                } catch (InterruptedException e) {
                     // do nothing
                }
                System.out.println(name + " leave");
                exists.set(false);
           } else {
                System.out.println(name + " give up");
           }

      }
       public static void main(String[] args) {
           BarWorker2 bar1 = new BarWorker2("bar1");
           BarWorker2 bar2 = new BarWorker2("bar2");
           new Thread(bar1).start();
           new Thread(bar2).start();
      }
}

运行结果：
bar1 enter
bar1 working
bar2 give up
bar1 leave
```

可以从上面的运行结果看出仅仅一个线程进行工作，因为 exists.compareAndSet(false, true)提供了原子性操作，比较和赋值操作组成了一个原子操作，中间不会提供可乘之机。使得一个线程操作，其他线程等待或不作为。

下面我们简单介绍下 AtomicBoolean 的 API

创建一个 AtomicBoolean

你可以这样创建一个 AtomicBoolean：

```
AtomicBoolean atomicBoolean = new AtomicBoolean();
```

以上示例新建了一个默认值为 false 的 AtomicBoolean。如果你想要为 AtomicBoolean 实例设置一个显式的初始值，那么你可以将初始值传给 AtomicBoolean 的构造子：

```
AtomicBoolean atomicBoolean = new AtomicBoolean(true);
```


http://www.itheima.com Copyright© 2018 黑马程序员

获得 AtomicBoolean 的值

你可以通过使用 get() 方法来获取一个 AtomicBoolean 的值。示例如下：

```
AtomicBoolean atomicBoolean = new AtomicBoolean(true);
boolean value = atomicBoolean.get();
```

设置 AtomicBoolean 的值

你可以通过使用 set() 方法来设置一个 AtomicBoolean 的值。

示例如下：

```
AtomicBoolean atomicBoolean = new AtomicBoolean(true);
atomicBoolean.set(false);
```

以上代码执行后 AtomicBoolean 的值为 false。

交换 AtomicBoolean 的值

你可以通过 getAndSet() 方法来交换一个 AtomicBoolean 实例的值。getAndSet() 方法将返回 AtomicBoolean 当前的值，并将为 AtomicBoolean 设置一个新值。示例如下：

```
AtomicBoolean atomicBoolean = new AtomicBoolean(true);
boolean oldValue = atomicBoolean.getAndSet(false);
```

以上代码执行后 oldValue 变量的值为 true，atomicBoolean 实例将持有 false 值。代码成功将 AtomicBoolean 当前值 ture 交换为 false。

比较并设置 AtomicBoolean 的值

compareAndSet() 方法允许你对 AtomicBoolean 的当前值与一个期望值进行比较，如果当前值等于期望值的话，将会对 AtomicBoolean 设定一个新值。compareAndSet() 方法是原子性的，因此在同一时间之内有单个线程执行它。因此 compareAndSet() 方法可被用于一些类似于锁的同步的简单实现。以下是一个 compareAndSet() 示例：

```
AtomicBoolean atomicBoolean = new AtomicBoolean(true);

boolean expectedValue = true;
boolean newValue      = false;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
boolean wasNewValueSet = atomicBoolean.compareAndSet(
    expectedValue, newValue);
```

本示例对 AtomicBoolean 的当前值与 true 值进行比较，如果相等，将 AtomicBoolean 的值更新为 false

➢AtomicInteger 原子性整型

AtomicInteger，一个提供原子操作的 Integer 的类。在 Java 语言中，++i 和 i++操作并不是线程安全的，在使用的时候，不可避免的会用到 synchronized 关键字。而 AtomicInteger 则通过一种线程安全的加减操作接口。

我们先来看看 AtomicInteger 给我们提供了什么方法：

```
public final int get() //获取当前的值
```

```
public final int getAndSet(int newValue)//获取当前的值，并设置新的值

public final int getAndIncrement()//获取当前的值，并自增

public final int getAndDecrement() //获取当前的值，并自减

public final int getAndAdd(int delta)  //获取当前的值，并加上预期的值
```

下面通过两个简单的例子来看一下 AtomicInteger 的优势在哪:

普通线程同步:

```
class Test2 {
       private volatile int count = 0;

       public synchronized void increment() {
                count++; //若要线程安全执行执行 count++，需要加锁
       }

       public int getCount() {
                return count;
       }
}
```


http://www.itheima.com Copyright© 2018 黑马程序员

使用 AtomicInteger:

```
class Test2 {
       private AtomicInteger count = new AtomicInteger();

       public void increment() {
                count.incrementAndGet();
       }
      //使用 AtomicInteger 之后，不需要加锁，也可以实现线程安全。
      public int getCount() {
               return count.get();
       }
}
```

从上面的例子中我们可以看出：使用 AtomicInteger 是非常的安全的.而且因为 AtomicInteger 由硬件提供原子操作指令实现的。在非激烈竞争的情况下，开销更小，速度更快。AtomicInteger 是使用非阻塞算法来实现并发控制的。AtomicInteger 的关键域只有一下 3 个：

```
// setup to use Unsafe.compareAndSwapInt for updates
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;
static {
        try {
           valueOffset=
              unsafe.objectFieldOffset(AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) {
              throw new Error(ex);
        }
    }
private volatile int value;
```

这里， unsafe 是 java 提供的获得对对象内存地址访问的类，注释已经清楚的写出了，它的作用就是在更新操作时提供“比较并替换”的作用。实际上就是 AtomicInteger 中的一个工具。valueOffset 是用来记录 value 本身在内存的便宜地址的，这个记录，也主要是为了在更新操作在内存中找到 value 的位置，方便比较。

注意：value 是用来存储整数的时间变量，这里被声明为 volatile，就是为了保证在更新操作时，当前线程可以拿到 value 最新的值（并发环境下，value 可能已经被其他线程更新了）。

优点:最大的好处就是可以避免多线程的优先级倒置和死锁情况的发生，提升在高并发处理下的性能。

 Copyright© 2018 黑马程序员

下面我们简单介绍下 AtomicInteger 的 API

创建一个 AtomicInteger

创建一个 AtomicInteger 示例如下：

```
AtomicInteger atomicInteger = new AtomicInteger();
```

本示例将创建一个初始值为 0 的 AtomicInteger。如果你想要创建一个给定初始值的 AtomicInteger，你可以这样：

```
AtomicInteger atomicInteger = new AtomicInteger(123);
```

本示例将 123 作为参数传给 AtomicInteger 的构造子，它将设置 AtomicInteger 实例的初始值为 123。

获得 AtomicInteger 的值

你可以使用 get() 方法获取 AtomicInteger 实例的值。示例如下：

```
AtomicInteger atomicInteger = new AtomicInteger(123);
int theValue = atomicInteger.get();
```

设置 AtomicInteger 的值

你可以通过 set() 方法对 AtomicInteger 的值进行重新设置。以下是 AtomicInteger.set() 示例：

```
AtomicInteger atomicInteger = new AtomicInteger(123);
atomicInteger.set(234);
```

以上示例创建了一个初始值为 123 的 AtomicInteger，而在第二行将其值更新为 234。

比较并设置 AtomicInteger 的值

AtomicInteger 类也通过了一个原子性的 compareAndSet() 方法。这一方法将 AtomicInteger 实例的当前值与期望值进行比较，如果二者相等，为 AtomicInteger 实例设置一个新值。AtomicInteger.compareAndSet() 代码示例：

```
AtomicInteger atomicInteger = new AtomicInteger(123);
int expectedValue = 123;
int newValue      = 234;
atomicInteger.compareAndSet(expectedValue, newValue);
```


http://www.itheima.com Copyright© 2018 黑马程序员

本示例首先新建一个初始值为 123 的 AtomicInteger 实例。然后将 AtomicInteger 与期望值 123 进行比较，如果相等，将 AtomicInteger 的值更新为 234。

增加 AtomicInteger 的值

AtomicInteger 类包含有一些方法，通过它们你可以增加 AtomicInteger 的值，并获取其值。这些方法如下：

```
public final int addAndGet(int addValue)//在原来的数值上增加新的值，并返回新值

public final int getAndIncrement()//获取当前的值，并自增

public final int incrementAndget() //自减，并获得自减后的值

public final int getAndAdd(int delta)  //获取当前的值，并加上预期的值
```

第一个 addAndGet() 方法给 AtomicInteger 增加了一个值，然后返回增加后的值。getAndAdd() 方法为 AtomicInteger 增加了一个值，但返回的是增加以前的 AtomicInteger 的值。具体使用哪一个取决于你的应用场景。以下是这两种方法的示例：

```
AtomicInteger atomicInteger = new AtomicInteger();
System.out.println(atomicInteger.getAndAdd(10));
System.out.println(atomicInteger.addAndGet(10));
```

本示例将打印出 0 和 20。例子中，第二行拿到的是加 10 之前的 AtomicInteger 的值。加 10 之前的值是 0。第三行将 AtomicInteger 的值再加 10，并返回加操作之后的值。该值现在是为 20。你当然也可以使用这俩方法为 AtomicInteger 添加负值。结果实际是一个减法操作。getAndIncrement() 和 incrementAndGet() 方法类似于 getAndAdd() 和 addAndGet()，但每次只将 AtomicInteger 的值加 1。

减小 AtomicInteger 的值

AtomicInteger 类还提供了一些减小 AtomicInteger 的值的原子性方法。这些方法是：

```
public final int decrementAndGet()
public final int getAndDecrement()
```

decrementAndGet() 将 AtomicInteger 的值减一，并返回减一后的值。getAndDecrement() 也将 AtomicInteger 的值减一，但它返回的是减一之前的值。

http://www.itheima.com Copyright© 2018 黑马程序员

➢AtomicIntegerArray 原子性整型数组

java.util.concurrent.atomic.AtomicIntegerArray 类提供了可以以原子方式读取和写入的底层 int 数组的操作，还包含高级原子操作。 AtomicIntegerArray 支持对底层 int 数组变量的原子操作。 它具有获取和设置方法，如在变量上的读取和写入。 也就是说，一个集合与同一变量上的任何后续 get 相关联。 原子 compareAndSet 方法也具有这些内存一致性功能。

AtomicIntegerArray 本质上是对 int[]类型的封装。使用 Unsafe 类通过 CAS 的方式控制 int[]在多线程下的安全性。它提供了以下几个核心 API：

```
//获得数组第 i 个下标的元素
public final int get(int i)
//获得数组的长度
public final int length()
//将数组第 i 个下标设置为 newValue，并返回旧的值
public final int getAndSet(int i, int newValue)
//进行 CAS 操作，如果第 i 个下标的元素等于 expect，则设置为 update，设置成功返回 true
public final boolean compareAndSet(int i, int expect, int update)
//将第 i 个下标的元素加 1
public final int getAndIncrement(int i)
//将第 i 个下标的元素减 1
public final int getAndDecrement(int i)
//将第 i 个下标的元素增加 delta（delta 可以是负数）
public final int getAndAdd(int i, int delta)
```

下面给出一个简单的示例，展示 AtomicIntegerArray 使用：

```
public class AtomicIntegerArrayDemo {
    static AtomicIntegerArray arr = new AtomicIntegerArray(10);
   public static class AddThread implements Runnable{
    public void run(){
          for(int k=0;k<10000;k++)
             arr.getAndIncrement(k%arr.length());
       }
    }
    public static void main(String[] args) throws InterruptedException {
       Thread[] ts=new Thread[10];
       for(int k=0;k<10;k++){
          ts[k]=new Thread(new AddThread());
       }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            for(int k=0;k<10;k++){ts[k].start();}
            for(int k=0;k<10;k++){ts[k].join();}
            System.out.println(arr);
        }
    }

输出结果：
[10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000]
```

上述代码第 2 行，申明了一个内含 10 个元素的数组。第 3 行定义的线程对数组内 10 个元素进行累加操作，每个元素各加 1000 次。第 11 行，开启 10 个这样的线程。因此，可以预测，如果线程安全，数组内 10 个元素的值必然都是 10000。反之，如果线程不安全，则部分或者全部数值会小于 10000。

➢AtomicLong、AtomicLongArray 原子性整型数组

AtomicLong、AtomicLongArray 的 API 跟 AtomicInteger、AtomicIntegerArray 在使用方法都是差不多的。区别在于用前者是使用原子方式更新的 long 值和 long 数组，后者是使用原子方式更新的 Integer 值和 Integer 数组。两者的相同处在于它们此类确实扩展了 Number，允许那些处理基于数字类的工具和实用工具进行统一访问。在实际开发中，它们分别用于不同的场景。这个就具体情况具体分析了，下面将举例说明 AtomicLong 的使用场景（使用 AtomicLong 生成自增长 ID），其他就不在过多介绍。

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.concurrent.atomic.AtomicLong;

public class AtomicLongTest {

    /**
     * @param args
     */
    public static void main(String[] args) {
        final AtomicLong orderIdGenerator = new AtomicLong(0);
        final List<Item> orders = Collections
                .synchronizedList(new ArrayList<Item>());
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        for (int i = 0; i < 10; i++) {
            Thread orderCreationThread = new Thread(new Runnable() {
                public void run() {
                    for (int i = 0; i < 10; i++) {
                        long orderId = orderIdGenerator.incrementAndGet();
                        Item order = new Item(Thread.currentThread().getName(),
                                orderId);
                        orders.add(order);
                    }
                }
            });
            orderCreationThread.setName("Order Creation Thread " + i);
            orderCreationThread.start();
        }
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Set<Long> orderIds = new HashSet<Long>();
        for (Item order : orders) {
            orderIds.add(order.getID());
            System.out.println("Order name:" + order.getItemName()
             +"----"+"Order id:" + order.getID());
        }
    }
}

class Item {
    String itemName;
    long id;

    Item(String n, long id) {
        this.itemName = n;
        this.id = id;
    }

    public String getItemName() {
        return itemName;
    }

    public long getID() {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
            return id;
        }
    }

    输出：
    Order name:Order Creation Thread 0----Order id:1
    Order name:Order Creation Thread 1----Order id:2
    Order name:Order Creation Thread 0----Order id:4
    Order name:Order Creation Thread 1----Order id:5
    Order name:Order Creation Thread 3----Order id:3
    Order name:Order Creation Thread 0----Order id:7
    Order name:Order Creation Thread 1----Order id:6
    ........
    Order name:Order Creation Thread 2----Order id:100
```

从运行结果我们看到，不管是哪个线程。它们获得的 ID 是不会重复的，保证的 ID 生成的原子性，避免了线程安全上的问题。

3) java.util.concurrent.lock 包

待续...

（三）多线程面试题

1. 多线程的创建方式（2017-11-23-wzz）

（1）、继承 Thread 类：但 Thread 本质上也是实现了 Runnable 接口的一个实例，它代表一个线程的实例，并且，启动线程的唯一方法就是通过 Thread 类的 start()实例方法。start()方法是一个 native 方法，它将启动一个新线程，并执行 run()方法。这种方式实现多线程很简单，通过自己的类直接 extend Thread，并复写 run()方法，就可以启动新线程并执行自己定义的 run()方法。例如：继承 Thread 类实现多线程，并在合适的地方启动线程

```
1. public class MyThread extends Thread {
2.     public void run() {
3.      System.out.println("MyThread.run()");
4.     }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
5. }
```

```
6. MyThread myThread1 = new MyThread();
7. MyThread myThread2 = new MyThread();
8. myThread1.start();
9. myThread2.start();
```

（2）、实现 Runnable 接口的方式实现多线程，并且实例化 Thread，传入自己的 Thread 实例，调用 run()方法

```
1. public class MyThread implements Runnable {
2.     public void run() {
3.      System.out.println("MyThread.run()");
4.     }
5. }
6. MyThread myThread = new MyThread();
7. Thread thread = new Thread(myThread);
8. thread.start();
```

（3）、使用 ExecutorService、Callable、Future 实现有返回结果的多线程：ExecutorService、Callable、Future 这个对象实际上都是属于 Executor 框架中的功能类。想要详细了解 Executor 框架的可以访问 http://www.javaeye.com/topic/366591 ，这里面对该框架做了很详细的解释。返回结果的线程是在 JDK1.5 中引入的新特征，确实很实用，有了这种特征我就不需要再为了得到返回值而大费周折了，而且即便实现了也可能漏洞百出。可返回值的任务必须实现 Callable 接口，类似的，无返回值的任务必须 Runnable 接口。执行 Callable 任务后，可以获取一个 Future 的对象，在该对象上调用 get 就可以获取到 Callable 任务返回的 Object 了，再结合线程池接口 ExecutorService 就可以实现传说中有返回结果的多线程了。下面提供了一个完整的有返回结果的多线程测试例子，在 JDK1.5 下验证过没问题可以直接使用。代码如下：

```
1. import java.util.concurrent.*;
2. import java.util.Date;
3. import java.util.List;
4. import java.util.ArrayList;
5.
6. /**
7. * 有返回值的线程
8. */
```


 Copyright© 2018 黑马程序员

```
@SuppressWarnings("unchecked")
public class Test {
public static void main(String[] args) throws ExecutionException,
    InterruptedException {
   System.out.println("----程序开始运行----");
   Date date1 = new Date();

   int taskSize = 5;
   // 创建一个线程池
   ExecutorService pool = Executors.newFixedThreadPool(taskSize);
   // 创建多个有返回值的任务
   List<Future> list = new ArrayList<Future>();
   for (int i = 0; i < taskSize; i++) {
    Callable c = new MyCallable(i + " ");
    // 执行任务并获取 Future 对象
    Future f = pool.submit(c);
    // System.out.println(">>>" + f.get().toString());
    list.add(f);
   }
   // 关闭线程池
   pool.shutdown();

   // 获取所有并发任务的运行结果
   for (Future f : list) {
    // 从 Future 对象上获取任务的返回值，并输出到控制台
    System.out.println(">>>" + f.get().toString());
   }

   Date date2 = new Date();
   System.out.println("----程序结束运行----，程序运行时间【"
     + (date2.getTime() - date1.getTime()) + "毫秒】");
}
}

class MyCallable implements Callable<Object> {
private String taskNum;

MyCallable(String taskNum) {
   this.taskNum = taskNum;
}

public Object call() throws Exception {
   System.out.println(">>>" + taskNum + "任务启动");
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
52.    Date dateTmp1 = new Date();
53.    Thread.sleep(1000);
54.    Date dateTmp2 = new Date();
55.    long time = dateTmp2.getTime() - dateTmp1.getTime();
56.    System.out.println(">>>" + taskNum + "任务终止");
57.    return taskNum + "任务返回运行结果,当前任务时间【" + time + "毫秒】";
58. }
59. }
```

2. 在 java 中 wait 和 sleep 方法的不同？

最大的不同是在等待时 wait 会释放锁，而 sleep 一直持有锁。wait 通常被用于线程间交互，sleep 通常被用于暂停执行。 

3. synchronized 和 volatile 关键字的作用

一旦一个共享变量（类的成员变量、类的静态成员变量）被 volatile 修饰之后，那么就具备了两层语义：

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2）禁止进行指令重排序。

volatile 本质是在告诉 jvm 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；

synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

1.volatile 仅能使用在变量级别；

synchronized 则可以使用在变量、方法、和类级别的

2.volatile 仅能实现变量的修改可见性，并不能保证原子性；

synchronized 则可以保证变量的修改可见性和原子性

3.volatile 不会造成线程的阻塞；

http://www.itheima.com Copyright© 2018 黑马程序员

synchronized 可能会造成线程的阻塞。

4.volatile 标记的变量不会被编译器优化；

synchronized 标记的变量可以被编译器优化

4. 分析线程并发访问代码解释原因

```
1.  public class Counter {
2.      private volatile int count = 0;
3.      public void inc(){
4.          try {
5.              Thread.sleep(3);
6.          } catch (InterruptedException e) {
7.              e.printStackTrace();
8.          }
9.          count++;
10.     }
11.     @Override
12.     public String toString() {
13.         return "[count=" + count + "]";
14.     }
15. }
16. //---------------------------------华丽的分割线-----------------------------
17. public class VolatileTest {
18.     public static void main(String[] args) {
19.         final Counter counter = new Counter();
20.         for(int i=0;i<1000;i++){
21.             new Thread(new Runnable() {
22.                 @Override
23.                 public void run() {
24.                     counter.inc();
25.                 }
26.             }).start();
27.         }
28.         System.out.println(counter);
29.     }
30. }
```

上面的代码执行完后输出的结果确定为 1000 吗?

http://www.itheima.com Copyright© 2018 黑马程序员

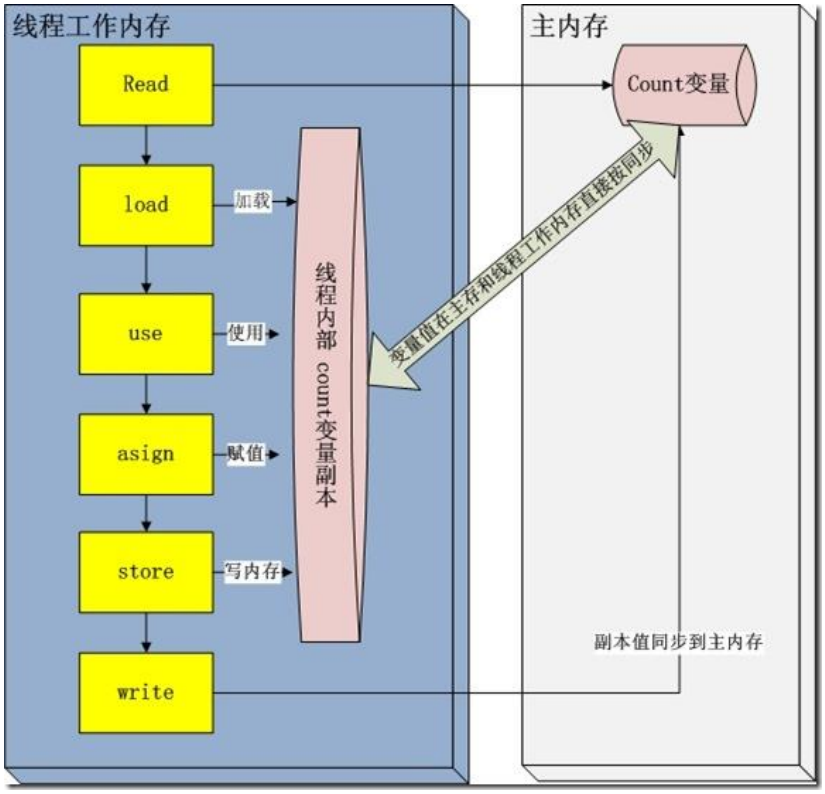
答案是不一定，或者不等于 1000。这是为什么吗?

在 java 的内存模型中每一个线程运行时都有一个线程栈，线程栈保存了线程运行时候变量值信息。当线程访问某一个对象时候值的时候，首先通过对象的引用找到对应在堆内存的变量的值，然后把堆内存变量的具体值 load 到线程本地内存中，建立一个变量副本，之后线程就不再和对象在堆内存变量值有任何关系，而是直接修改副本变量的值，在修改完之后的某一个时刻（线程退出之前），自动把线程变量副本的值回写到对象在堆中变量。这样在堆中的对象的值就产生变化了。

也就是说上面主函数中开启了 1000 个子线程，每个线程都有一个变量副本，每个线程修改变量只是临时修改了自己的副本，当线程结束时再将修改的值写入在主内存中，这样就出现了线程安全问题。因此结果就不可能等于 1000 了，一般都会小于 1000。

上面的解释用一张图表示如下:

(图片来自网络，非本人所绘)



http://www.itheima.com Copyright© 2018 黑马程序员

5. 什么是线程池，如何使用？

线程池就是事先将多个线程对象放到一个容器中，当使用的时候就不用 new 线程而是直接去池中拿线程即可，节省了开辟子线程的时间，提高的代码执行效率。

在 JDK 的 java.util.concurrent.Executors 中提供了生成多种线程池的静态方法。

```
1. ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();
2. ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(4);
3. ScheduledExecutorService newScheduledThreadPool = Executors.newScheduledThreadPool(4);
4. ExecutorService newSingleThreadExecutor = Executors.newSingleThreadExecutor();
```

然后调用他们的 execute 方法即可。

6. 常用的线程池有哪些？（2017-11-23-wzz）

newSingleThreadExecutor：创建一个单线程的线程池，此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

newFixedThreadPool：创建固定大小的线程池，每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。

newCachedThreadPool：创建一个可缓存的线程池，此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。

newScheduledThreadPool：创建一个大小无限的线程池，此线程池支持定时以及周期性执行任务的需求。

newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

7. 请叙述一下您对线程池的理解？（2015-11-25）

（如果问到了这样的问题，可以展开的说一下线程池如何用、线程池的好处、线程池的启动策略）

合理利用线程池能够带来三个好处。

http://www.itheima.com Copyright© 2018 黑马程序员

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

8. 线程池的启动策略？（2015-11-25）

官方对线程池的执行过程描述如下：

```
26. /*
27.         * Proceed in 3 steps:
28.         *
29.         * 1. If fewer than corePoolSize threads are running, try to
30.         * start a new thread with the given command as its first
31.         * task.  The call to addWorker atomically checks runState and
32.         * workerCount, and so prevents false alarms that would add
33.         * threads when it shouldn't, by returning false.
34.         *
35.         * 2. If a task can be successfully queued, then we still need
36.         * to double-check whether we should have added a thread
37.         * (because existing ones died since last checking) or that
38.         * the pool shut down since entry into this method. So we
39.         * recheck state and if necessary roll back the enqueuing if
40.         * stopped, or start a new thread if there are none.
41.         *
42.         * 3. If we cannot queue task, then we try to add a new
43.         * thread.  If it fails, we know we are shut down or saturated
44.         * and so reject the task.
45.         */
```

1、线程池刚创建时，里面没有一个线程。任务队列是作为参数传进来的。不过，就算队列里面有任务，线程池也不会马上执行它们。

2、当调用 execute() 方法添加一个任务时，线程池会做如下判断：

a. 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务；

http://www.itheima.com Copyright© 2018 黑马程序员

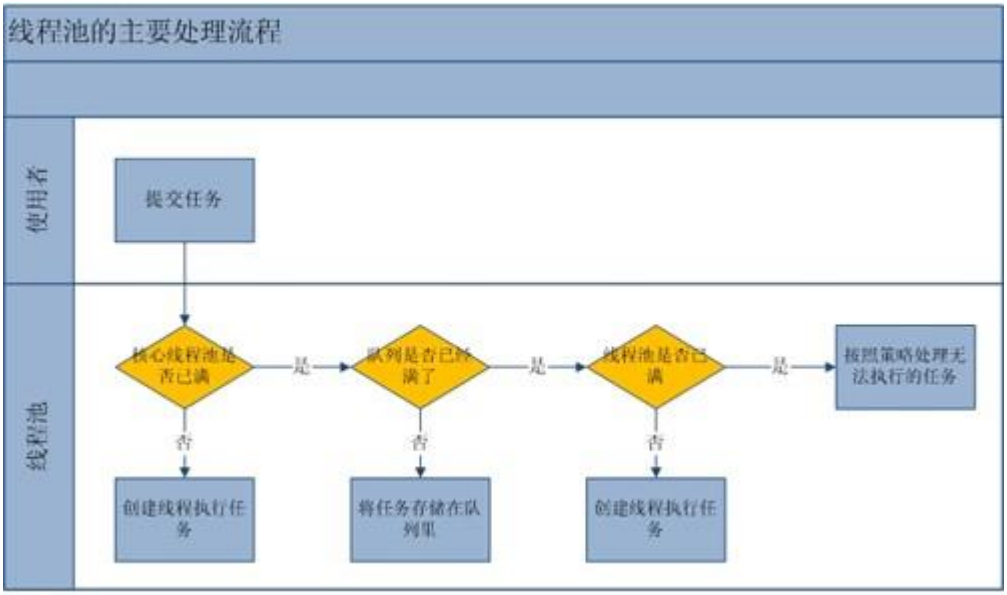
b. 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列。

c. 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要创建线程运行这个任务；

d. 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会抛出异常，告诉调用者“我不能再接受任务了”。

3、当一个线程完成任务时，它会从队列中取下一个任务来执行。

4、当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断，如果当前运行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。



9. 如何控制某个方法允许并发访问线程的个数？（2015-11-30）

```
1. package com.yange;
2.
3. import java.util.concurrent.Semaphore;
4. /**
5.  *
6.  * @author wzy 2015-11-30
7.  *
8.  */
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
9. public class SemaphoreTest {
10. /*
11. * permits the initial number of permits available. This value may be negative,
12. in which case releases must occur before any acquires will be granted.
13. fair true if this semaphore will guarantee first-in first-out granting of
14. permits under contention, else false
15. */
16. static Semaphore semaphore = new Semaphore(5,true);
17. public static void main(String[] args) {
18.     for(int i=0;i<100;i++){
19.         new Thread(new Runnable() {
20.
21.             @Override
22.             public void run() {
23.                 test();
24.             }
25.         }).start();
26.     }
27.
28. }
29.
30. public static void test(){
31.     try {
32.       //申请一个请求
33.         semaphore.acquire();
34.     } catch (InterruptedException e1) {
35.         e1.printStackTrace();
36.     }
37.     System.out.println(Thread.currentThread().getName()+"进来了");
38.     try {
39.         Thread.sleep(1000);
40.     } catch (InterruptedException e) {
41.         e.printStackTrace();
42.     }
43.     System.out.println(Thread.currentThread().getName()+"走了");
44.   //释放一个请求
45.     semaphore.release();
46. }
47. }
```

可以使用 Semaphore 控制，第 16 行的构造函数创建了一个 Semaphore 对象，并且初始化了 5 个信号。这样的效果是控件 test 方法最多只能有 5 个线程并发访问，对于 5 个线程时就排队等待，走一个来一下。第 33 行，请求一

http://www.itheima.com Copyright© 2018 黑马程序员

个信号（消费一个信号），如果信号被用完了则等待，第 45 行释放一个信号，释放的信号新的线程就可以使用了。

10. 三个线程 a、b、c 并发运行，b,c 需要 a 线程的数据怎么实现（上海 3 期学员提供）

根据问题的描述，我将问题用以下代码演示，ThreadA、ThreadB、ThreadC，ThreadA 用于初始化数据 num，只有当 num 初始化完成之后再让 ThreadB 和 ThreadC 获取到初始化后的变量 num。

分析过程如下：

考虑到多线程的不确定性，因此我们不能确保 ThreadA 就一定先于 ThreadB 和 ThreadC 前执行，就算 ThreadA 先执行了，我们也无法保证 ThreadA 什么时候才能将变量 num 给初始化完成。因此我们必须让 ThreadB 和 ThreadC 去等待 ThreadA 完成任何后发出的消息。

现在需要解决两个难题，一是让 ThreadB 和 ThreadC 等待 ThreadA 先执行完，二是 ThreadA 执行完之后给 ThreadB 和 ThreadC 发送消息。

解决上面的难题我能想到的两种方案，一是使用纯 Java API 的 Semaphore 类来控制线程的等待和释放，二是使用 Android 提供的 Handler 消息机制。

```
1. package com.example;
2. /**
3.  * 三个线程 a、b、c 并发运行，b,c 需要 a 线程的数据怎么实现（上海 3 期学员提供）
4.  *
5.  */
6. public class ThreadCommunication {
7.  private static int num;//定义一个变量作为数据
8.
9.  public static void main(String[] args) {
10.
11.     Thread threadA = new Thread(new Runnable() {
12.
13.         @Override
14.         public void run() {
15.             try {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
16.                 //模拟耗时操作之后初始化变量 num
17.                 Thread.sleep(1000);
18.                 num = 1;
19.
20.             } catch (InterruptedException e) {
21.                 e.printStackTrace();
22.             }
23.         }
24.     });
25.     Thread threadB = new Thread(new Runnable() {
26.
27.         @Override
28.         public void run() {
29.             System.out.println(Thread.currentThread().getName()+"获取到 num 的值为："+num);
30.         }
31.     });
32.     Thread threadC = new Thread(new Runnable() {
33.
34.         @Override
35.         public void run() {
36.             System.out.println(Thread.currentThread().getName()+"获取到 num 的值为："+num);
37.         }
38.     });
39.     //同时开启 3 个线程
40.     threadA.start();
41.     threadB.start();
42.     threadC.start();
43.
44. }
45. }
46.
```

解决方案一:

```
1. public class ThreadCommunication {
2.  private static int num;
3.  /**
4.   * 定义一个信号量，该类内部维持了多个线程锁，可以阻塞多个线程，释放多个线程，
5. 线程的阻塞和释放是通过 permit 概念来实现的
6.   * 线程通过 semaphore.acquire()方法获取 permit，如果当前 semaphore 有 permit 则分配给该线程，
7. 如果没有则阻塞该线程直到 semaphore
8.   * 调用 release（）方法释放 permit。
9.   * 构造函数中参数：permit（允许） 个数，
10.  */
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
11. private static Semaphore semaphore = new Semaphore(0);
12. public static void main(String[] args) {
13.
14.     Thread threadA = new Thread(new Runnable() {
15.
16.         @Override
17.         public void run() {
18.             try {
19.                 //模拟耗时操作之后初始化变量 num
20.                 Thread.sleep(1000);
21.                 num = 1;
22.                 //初始化完参数后释放两个 permit
23.                 semaphore.release(2);
24.
25.             } catch (InterruptedException e) {
26.                 e.printStackTrace();
27.             }
28.         }
29.     });
30.     Thread threadB = new Thread(new Runnable() {
31.
32.         @Override
33.         public void run() {
34.             try {
35.                 //获取 permit，如果 semaphore 没有可用的 permit 则等待，如果有则消耗一个
36.                 semaphore.acquire();
37.             } catch (InterruptedException e) {
38.                 e.printStackTrace();
39.             }
40.             System.out.println(Thread.currentThread().getName()+"获取到 num 的值为："+num);
41.         }
42.     });
43.     Thread threadC = new Thread(new Runnable() {
44.
45.         @Override
46.         public void run() {
47.             try {
48.                 //获取 permit，如果 semaphore 没有可用的 permit 则等待，如果有则消耗一个
49.                 semaphore.acquire();
50.             } catch (InterruptedException e) {
51.                 e.printStackTrace();
52.             }
53.             System.out.println(Thread.currentThread().getName()+"获取到 num 的值为："+num);
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
54.         }
55.     });
56.     //同时开启 3 个线程
57.     threadA.start();
58.     threadB.start();
59.     threadC.start();
60.
61. }
62. }
```

11. 同一个类中的 2 个方法都加了同步锁，多个线程能同时访问同一个类中的这两个方法吗？（2017-2-24）

这个问题需要考虑到 Lock 与 synchronized 两种实现锁的不同情形。因为这种情况下使用 Lock 和 synchronized 会有截然不同的结果。Lock 可以让等待锁的线程响应中断，Lock 获取锁，之后需要释放锁。如下代码，多个线程不可访问同一个类中的 2 个加了 Lock 锁的方法。

```
63. package com;
64. import java.util.concurrent.locks.Lock;
65. import java.util.concurrent.locks.ReentrantLock;
66. public class qq {
67.
68.     private int count = 0;
69.     private Lock lock = new ReentrantLock();//设置 lock 锁
70.     //方法 1
71.     public Runnable run1 = new Runnable(){
72.         public void run() {
73.             lock.lock();  //加锁
74.                 while(count < 1000) {
75.                     try {
76.                         //打印是否执行该方法
77.                         System.out.println(Thread.currentThread().getName() + " run1: "+count++);
78.                     } catch (Exception e) {
79.                         e.printStackTrace();
80.                     }
81.                 }
82.             }
83.             lock.unlock();
84.         }};
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
85.    //方法 2
86.    public Runnable run2 = new Runnable(){
87.           public void run() {
88.              lock.lock();
89.                  while(count < 1000) {
90.                     try {
91.                         System.out.println(Thread.currentThread().getName() +
92.                                 " run2: "+count++);
93.                     } catch (Exception e) {
94.                         e.printStackTrace();
95.                     }
96.                  }
97.               lock.unlock();
98.           }};
99.
100.
101.
102.    public static void main(String[] args) throws InterruptedException {
103.        qq t = new qq();   //创建一个对象
104.        new Thread(t.run1).start();//获取该对象的方法 1
105.
106.        new Thread(t.run2).start();//获取该对象的方法 2
107.    }
108. }
```

结果是：

```
Thread-0 run1: 0
Thread-0 run1: 1
Thread-0 run1: 2
Thread-0 run1: 3
Thread-0 run1: 4
Thread-0 run1: 5
Thread-0 run1: 6
........
```

而 synchronized 却不行，使用 synchronized 时，当我们访问同一个类对象的时候，是同一把锁，所以可以访问该对象的其他 synchronized 方法。代码如下：

```
1. package com;
2. import java.util.concurrent.locks.Lock;
3. import java.util.concurrent.locks.ReentrantLock;
4. public class qq {
5.   private int count = 0;
6.    private Lock lock = new ReentrantLock();
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
7.    public Runnable run1 = new Runnable(){
8.        public void run() {
9.          synchronized(this) {  //设置关键字 synchronized，以当前类为锁
10.               while(count < 1000) {
11.                  try {
12.                     //打印是否执行该方法
13.                      System.out.println(Thread.currentThread().getName() + " run1: "+count++);
14.                  } catch (Exception e) {
15.                     e.printStackTrace();
16.                  }
17.              }
18.          }
19.       }};
20.    public Runnable run2 = new Runnable(){
21.           public void run() {
22.              synchronized(this) {
23.                  while(count < 1000) {
24.                     try {
25.                        System.out.println(Thread.currentThread().getName()
26.                                + " run2: "+count++);
27.                     } catch (Exception e) {
28.                        e.printStackTrace();
29.                     }
30.                  }
31.              }
32.          }};
33.    public static void main(String[] args) throws InterruptedException {
34.       qq t = new qq();  //创建一个对象
35.        new Thread(t.run1).start(); //获取该对象的方法 1
36.        new Thread(t.run2).start(); //获取该对象的方法 2
37.    }
38. }
```

结果为：

```
Thread-1 run2: 0
Thread-1 run2: 1
Thread-1 run2: 2
Thread-0 run1: 0
Thread-0 run1: 4 Thread-0 run1: 5 Thread-0 run1: 6
 ......
```


http://www.itheima.com Copyright© 2018 黑马程序员

12. 什么情况下导致线程死锁，遇到线程死锁该怎么解决？（2017-2-24）

11.1 死锁的定义：所谓死锁是指多个线程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。

11.2 死锁产生的必要条件：

互斥条件：线程要求对所分配的资源（如打印机）进行排他性控制，即在一段时间内某 资源仅为一个线程所占有。此时若有其他线程请求该资源，则请求线程只能等待。

不剥夺条件：线程所获得的资源在未使用完毕之前，不能被其他线程强行夺走，即只能由获得该资源的线程自己来释放（只能是主动释放)。

请求和保持条件：线程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其他线程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。

循环等待条件：存在一种线程资源的循环等待链，链中每一个线程已获得的资源同时被链中下一个线程所请求。即存在一个处于等待状态的线程集合{Pl, P2, ..., pn}，其中 Pi 等待的资源被 P(i+1)占有（i=0, 1, ..., n-1)，Pn 等待的资源被 P0 占有，如图 2-15 所示。

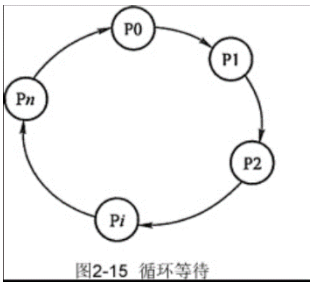


图2-15 循环等待

11.3 产生死锁的一个例子

```
1. package itheima.com;
2. /**
3. * 一个简单的死锁类
4. * 当 DeadLock 类的对象 flag==1 时（td1），先锁定 o1,睡眠 500 毫秒
5. * 而 td1 在睡眠的时候另一个 flag==0 的对象（td2）线程启动，先锁定 o2,睡眠 500 毫秒
```


 Copyright© 2018 黑马程序员

```
6. * td1 睡眠结束后需要锁定 o2 才能继续执行，而此时 o2 已被 td2 锁定；
7. * td2 睡眠结束后需要锁定 o1 才能继续执行，而此时 o1 已被 td1 锁定；
8. * td1、td2 相互等待，都需要得到对方锁定的资源才能继续执行，从而死锁。
9. */
10. public class DeadLock implements Runnable {
11.     public int flag = 1;
12.     //静态对象是类的所有对象共享的
13.     private static Object o1 = new Object(), o2 = new Object();
14.     public void run() {
15.         System.out.println("flag=" + flag);
16.         if (flag == 1) {
17.             synchronized (o1) {
18.                 try {
19.                     Thread.sleep(500);
20.                 } catch (Exception e) {
21.                     e.printStackTrace();
22.                 }
23.                 synchronized (o2) {
24.                     System.out.println("1");
25.                 }
26.             }
27.         }
28.         if (flag == 0) {
29.             synchronized (o2) {
30.                 try {
31.                     Thread.sleep(500);
32.                 } catch (Exception e) {
33.                     e.printStackTrace();
34.                 }
35.                 synchronized (o1) {
36.                     System.out.println("0");
37.                 }
38.             }
39.         }
40.     }
41.     public static void main(String[] args) {
42.         DeadLock td1 = new DeadLock();
43.         DeadLock td2 = new DeadLock();
44.         td1.flag = 1;
45.         td2.flag = 0;
46.         //td1,td2 都处于可执行状态，但 JVM 线程调度先执行哪个线程是不确定的。
47.         //td2 的 run()可能在 td1 的 run()之前运行
48.         new Thread(td1).start();
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
49.         new Thread(td2).start();
50.     }
51. }
```

11.4 如何避免死锁

在有些情况下死锁是可以避免的。两种用于避免死锁的技术：

1）加锁顺序（线程按照一定的顺序加锁）

```
1. package itheima.com;
2. public class DeadLock {
3.     public int flag = 1;
4.     //静态对象是类的所有对象共享的
5.     private static Object o1 = new Object(), o2 = new Object();
6.     public void money(int flag) {
7.       this.flag=flag;
8.       if( flag ==1){
9.           synchronized (o1) {
10.               try {
11.                   Thread.sleep(500);
12.               } catch (Exception e) {
13.                   e.printStackTrace();
14.               }
15.               synchronized (o2) {
16.                   System.out.println("当前的线程是"+
17.                           Thread.currentThread().getName()+" "+"flag 的值"+"1");
18.               }
19.           }
20.       }
21.       if(flag ==0){
22.         synchronized (o2) {
23.               try {
24.                   Thread.sleep(500);
25.               } catch (Exception e) {
26.                   e.printStackTrace();
27.               }
28.               synchronized (o1) {
29.                   System.out.println("当前的线程是"+
30.                           Thread.currentThread().getName()+" "+"flag 的值"+"0");
31.               }
32.           }
33.       }
34.     }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
35.
36.     public static void main(String[] args) {
37.         final DeadLock td1 = new DeadLock();
38.         final DeadLock td2 = new DeadLock();
39.         td1.flag = 1;
40.         td2.flag = 0;
41.         //td1,td2 都处于可执行状态，但 JVM 线程调度先执行哪个线程是不确定的。
42.         //td2 的 run()可能在 td1 的 run()之前运行
43.         final Thread t1=new Thread(new Runnable(){
44.           public void run() {
45.               td1.flag = 1;
46.               td1.money(1);
47.           }
48.         });
49.         t1.start();
50.         Thread t2= new Thread(new Runnable(){
51.           public void run() {
52.               // TODO Auto-generated method stub
53.               try {
54.                   //让 t2 等待 t1 执行完
55.                   t1.join();//核心代码，让 t1 执行完后 t2 才会执行
56.               } catch (InterruptedException e) {
57.                   // TODO Auto-generated catch block
58.                   e.printStackTrace();
59.               }
60.               td2.flag = 0;
61.               td1.money(0);
62.           }
63.         });
64.         t2.start();
65.     }
66. }
```

结果：

```
当前的线程是 Thread-0 flag 的值 1
当前的线程是 Thread-1 flag 的值 0
```

2）加锁时限（线程尝试获取锁的时候加上一定的时限，超过时限则放弃对该锁的请求，并释放自己占有的锁）

```
1.  package itheima.com;
2. import java.util.concurrent.TimeUnit;
3. import java.util.concurrent.locks.Lock;
4. import java.util.concurrent.locks.ReentrantLock;
5. public class DeadLock {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
6.    public int flag = 1;
7.    //静态对象是类的所有对象共享的
8.    private static Object o1 = new Object(), o2 = new Object();
9.    public void money(int flag) throws InterruptedException {
10.     this.flag=flag;
11.     if( flag ==1){
12.         synchronized (o1) {
13.                 Thread.sleep(500);
14.              synchronized (o2) {
15.                 System.out.println("当前的线程是"+
16.                         Thread.currentThread().getName()+" "+"flag 的值"+"1");
17.              }
18.           }
19.       }
20.       if(flag ==0){
21.         synchronized (o2) {
22.                 Thread.sleep(500);
23.              synchronized (o1) {
24.                 System.out.println("当前的线程是"+
25.                         Thread.currentThread().getName()+" "+"flag 的值"+"0");
26.              }
27.           }
28.       }
29.   }
30.
31.   public static void main(String[] args) {
32.     final Lock lock = new ReentrantLock();
33.       final DeadLock td1 = new DeadLock();
34.       final DeadLock td2 = new DeadLock();
35.       td1.flag = 1;
36.       td2.flag = 0;
37.       //td1,td2 都处于可执行状态，但 JVM 线程调度先执行哪个线程是不确定的。
38.       //td2 的 run()可能在 td1 的 run()之前运行
39.
40.       final Thread t1=new Thread(new Runnable(){
41.         public void run() {
42.             // TODO Auto-generated method stub
43.             String tName = Thread.currentThread().getName();
44.
45.             td1.flag = 1;
46.             try {
47.                 //获取不到锁，就等 5 秒，如果 5 秒后还是获取不到就返回 false
48.                 if (lock.tryLock(5000, TimeUnit.MILLISECONDS)) {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
49.                     System.out.println(tName + "获取到锁！");
50.                 } else {
51.                     System.out.println(tName + "获取不到锁！");
52.                     return;
53.                 }
54.             } catch (Exception e) {
55.                 e.printStackTrace();
56.             }
57.
58.             try {
59.                 td1.money(1);
60.             } catch (Exception e) {
61.                 System.out.println(tName + "出错了！！！");
62.             } finally {
63.                 System.out.println("当前的线程是"+Thread.currentThread().getName()+"释放锁！！
");
64.                 lock.unlock();
65.             }
66.         }
67.       });
68.       t1.start();
69.       Thread t2= new Thread(new Runnable(){
70.         public void run() {
71.             String tName = Thread.currentThread().getName();
72.             // TODO Auto-generated method stub
73.             td1.flag = 1;
74.             try {
75.                 //获取不到锁，就等 5 秒，如果 5 秒后还是获取不到就返回 false
76.                 if (lock.tryLock(5000, TimeUnit.MILLISECONDS)) {
77.                     System.out.println(tName + "获取到锁！");
78.                 } else {
79.                     System.out.println(tName + "获取不到锁！");
80.                     return;
81.                 }
82.             } catch (Exception e) {
83.                 e.printStackTrace();
84.             }
85.             try {
86.                 td2.money(0);
87.             } catch (Exception e) {
88.                 System.out.println(tName + "出错了！！！");
89.             } finally {
90.                 System.out.println("当前的线程是"+Thread.currentThread().getName()+"释放锁！！
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
");
    91.                     lock.unlock();
    92.                 }
    93.             }
    94.         });
    95.         t2.start();
    96.     }
    97. }
```

打印结果：

```
Thread-0 获取到锁!
当前的线程是 Thread-0  flag 的值 1
当前的线程是 Thread-0 释放锁！！
Thread-1 获取到锁!
当前的线程是 Thread-1  flag 的值 0
当前的线程是 Thread-1 释放锁！！
```

13. Java 中多线程间的通信怎么实现?(2017-2-24)

线程通信的方式：

1.共享变量

线程间通信可以通过发送信号，发送信号的一个简单方式是在共享对象的变量里设置信号值。线程 A 在一个同步块里设置 boolean 型成员变量 hasDataToProcess 为 true，线程 B 也在同步块里读取 hasDataToProcess 这个成员变量。这个简单的例子使用了一个持有信号的对象，并提供了 set 和 get 方法:

```
1. package itheima.com;
2. public class MySignal{
3.   //共享的变量
4.   private boolean hasDataToProcess=false;
5.   //取值
6.   public boolean getHasDataToProcess() {
7.       return hasDataToProcess;
8.   }
9.   //存值
10.  public void setHasDataToProcess(boolean hasDataToProcess) {
11.      this.hasDataToProcess = hasDataToProcess;
```


 Copyright© 2018 黑马程序员

```
12.  }
13.  public static void main(String[] args){
14.      //同一个对象
15.      final MySignal my=new MySignal();
16.      //线程 1 设置 hasDataToProcess 值为 true
17.      final Thread t1=new Thread(new Runnable(){
18.          public void run() {
19.              my.setHasDataToProcess(true);
20.          }
21.      });
22.      t1.start();
23.     //线程 2 取这个值 hasDataToProcess
24.      Thread t2=new Thread(new Runnable(){
25.          public void run() {
26.              try {
27.                 //等待线程 1 完成然后取值
28.                  t1.join();
29.              } catch (InterruptedException e) {
30.                  e.printStackTrace();
31.              }
32.              my.getHasDataToProcess();
33.              System.out.println("t1 改变以后的值：" + my.isHasDataToProcess());
34.          }
35.      });
36.      t2.start();
37. }
38. }
```

结果：

```
t1 改变以后的值：true
```

2.wait/notify 机制

以资源为例，生产者生产一个资源，通知消费者就消费掉一个资源，生产者继续生产资源，消费者消费资源，以此循环。代码如下：

```
1. package itheima.com;
2. //资源类
3.  class Resource{
4.     private String name;
5.     private int count=1;
6.     private boolean flag=false;
7.     public synchronized void set(String name){
8.        //生产资源
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
9.        while(flag) {
10.           try{
11.              //线程等待。消费者消费资源
12.              wait();
13.              }catch(Exception e){}
14.        }
15.        this.name=name+"---"+count++;
16.        System.out.println(Thread.currentThread().getName()+"...生产者..."+this.name);
17.        flag=true;
18.        //唤醒等待中的消费者
19.        this.notifyAll();
20.        }
21.    public synchronized void out(){
22.      //消费资源
23.        while(!flag) {
24.          //线程等待，生产者生产资源
25.            try{wait();}catch(Exception e){}
26.        }
27.        System.out.println(Thread.currentThread().getName()+"...消费者..."+this.name);
28.        flag=false;
29.        //唤醒生产者，生产资源
30.        this.notifyAll();
31.        }
32. }
33.  //生产者
34.  class Producer implements Runnable{
35.     private Resource res;
36.     Producer(Resource res){
37.         this.res=res;
38.     }
39.     //生产者生产资源
40.     public void run(){
41.         while(true){
42.            res.set("商品");
43.         }
44.     }
45.  }
46.  //消费者消费资源
47.  class Consumer implements Runnable{
48.     private Resource res;
49.     Consumer(Resource res){
50.         this.res=res;
51.     }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
52.     public void run(){
53.         while(true){
54.             res.out();
55.         }
56.     }
57. }
58. public class ProducerConsumerDemo{
59.     public static void main(String[] args){
60.         Resource r=new Resource();
61.         Producer pro=new Producer(r);
62.         Consumer con=new Consumer(r);
63.         Thread t1=new Thread(pro);
64.         Thread t2=new Thread(con);
65.         t1.start();
66.         t2.start();
67.     }
68. }
```

14. 线程和进程的区别（2017-11-23-wzz）

进程：具有一定独立功能的程序关于某个数据集合上的一次运行活动，是操作系统进行资源分配和调度的一个独立单位。

线程：是进程的一个实体，是 cpu 调度和分派的基本单位，是比进程更小的可以独立运行的基本单位。

特点：线程的划分尺度小于进程，这使多线程程序拥有高并发性，进程在运行时各自内存单元相互独立，线程之间内存共享，这使多线程编程可以拥有更好的性能和用户体验

注意：多线程编程对于其它程序是不友好的，占据大量 cpu 资源。

15. 请说出同步线程及线程调度相关的方法？（2017-11-23-wzz）

wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；

sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理 InterruptedException 异常；

notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，

http://www.itheima.com Copyright© 2018 黑马程序员

而是由 JVM 确定唤醒哪个线程，而且与优先级无关；

notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

注意：java 5 通过 Lock 接口提供了显示的锁机制，Lock 接口中定义了加锁（lock（）方法）和解锁（unLock（）方法），增强了多线程编程的灵活性及对线程的协调

16. 启动一个线程是调用 run()方法还是 start()方法？（2017-11-23-wzz）

启动一个线程是调用 start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由 JVM 调度并执行，这并不意味着线程就会立即运行。

run()方法是线程启动后要进行回调（callback）的方法。

十、Java 内部类

1. 静态嵌套类（Static Nested Class）和内部类(Inner Class)的不同？（2017-11-16-wl）

静态嵌套类：Static Nested Class 是被声明为静态（static）的内部类，它可以不依赖于外部类实例被实例化。

内部类：需要在外部类实例化后才能实例化，其语法看起来挺诡异的。

2. 下面的代码哪些地方会产生编译错误？（2017-11-16-wl）

```
1. class Outer {
2.
3. class Inner {}
4.
5. public static void foo() { new Inner(); }
6.
7. public void bar() { new Inner(); }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
8.
9. public static void main(String[] args) {
10. new Inner();
11. }
12. }
```

注意：Java 中非静态内部类对象的创建要依赖其外部类对象，上面的面试题中 foo 和 main 方法都是静态方法，静态方法中没有 this，也就是说没有所谓的外部类对象，因此无法创建内部类对象，如果要在静态方法中创建内部类对象，可以这样做

```
1. new Outer().new Inner();
```

第三章 JavaSE 高级

一、Java 中的反射

1. 说说你对 Java 中反射的理解

Java 中的反射首先是能够获取到 Java 中要反射类的字节码，获取字节码有三种方法，1.Class.forName(className) 2.类名.class 3.this.getClass()。然后将字节码中的方法，变量，构造函数等映射成相应的 Method、Filed、Constructor 等类，这些类提供了丰富的方法可以被我们所使用。

二、Java 中的动态代理

1. 写一个 ArrayList 的动态代理类（笔试题）

```
1. final List<String> list = new ArrayList<String>();
2.
3.      List<String> proxyInstance =
4. (List<String>)Proxy.newProxyInstance(list.getClass().getClassLoader(),
5. list.getClass().getInterfaces(),
6. new InvocationHandler() {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
7.
8.          @Override
9.          public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
10.             return method.invoke(list, args);
11.         }
12.     });
13.     proxyInstance.add("你好");
14.     System.out.println(list);
```

2. 动静态代理的区别，什么场景使用？（2015-11-25）

静态代理通常只代理一个类，动态代理是代理一个接口下的多个实现类。

静态代理事先知道要代理的是什么，而动态代理不知道要代理什么东西，只有在运行时才知道。

动态代理是实现 JDK 里的 InvocationHandler 接口的 invoke 方法，但注意的是代理的是接口，也就是你的业务类必须要实现接口，通过 Proxy 里的 newProxyInstance 得到代理对象。

还有一种动态代理 CGLIB，代理的是类，不需要业务类继承接口，通过派生的子类来实现代理。通过在运行时，动态修改字节码达到修改类的目的。

AOP 编程就是基于动态代理实现的，比如著名的 Spring 框架、Hibernate 框架等等都是动态代理的使用例子。

三、Java 中的设计模式&回收机制

1. 你所知道的设计模式有哪些

Java 中一般认为有 23 种设计模式，我们不需要所有的都会，但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式我单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

http://www.itheima.com Copyright© 2018 黑马程序员

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

2. 单例设计模式

最好理解的一种设计模式，分为懒汉式和饿汉式。

饿汉式：

```
1. public class Singleton {
2.  // 直接创建对象
3.  public static Singleton instance = new Singleton();
4.
5.  // 私有化构造函数
6.  private Singleton() {
7.  }
8.
9.  // 返回对象实例
10. public static Singleton getInstance() {
11.     return instance;
12. }
13.}
```

懒汉式：

```
1. public class Singleton {
2.  // 声明变量
3.  private static volatile Singleton singleton = null;
4.
5.  // 私有构造函数
6.  private Singleton() {
7.  }
8.
9.  // 提供对外方法
10. public static Singleton getInstance() {
11.     if (singleton == null) {
12.         synchronized (Singleton.class) {
13.             if (singleton == null) {
14.                 singleton = new Singleton();
```


 Copyright© 2018 黑马程序员

```
15.             }
16.         }
17.     }
18.     return singleton;
19. }
20. }
```

3. 工厂设计模式

工厂模式分为工厂方法模式和抽象工厂模式。

工厂方法模式

工厂方法模式分为三种：普通工厂模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。

多个工厂方法模式，是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

普通工厂模式

```
1. public interface Sender {
2.  public void Send();
3. }
4. public class MailSender implements Sender {
5.
6.  @Override
7.  public void Send() {
8.      System.out.println("this is mail sender!");
9.  }
10. }
11. public class SmsSender implements Sender {
12.
13. @Override
14. public void Send() {
15.     System.out.println("this is sms sender!");
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
16. }
17. }
18. public class SendFactory {
19.  public Sender produce(String type) {
20.      if ("mail".equals(type)) {
21.          return new MailSender();
22.      } else if ("sms".equals(type)) {
23.          return new SmsSender();
24.      } else {
25.          System.out.println("请输入正确的类型!");
26.          return null;
27.      }
28. }
29. }
```

- 多个工厂方法模式

该模式是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

```
1. public class SendFactory {
2.   public Sender produceMail(){
3.         return new MailSender();
4.     }
5.
6.     public Sender produceSms(){
7.         return new SmsSender();
8.     }
9. }
10.
11. public class FactoryTest {
12.  public static void main(String[] args) {
13.      SendFactory factory = new SendFactory();
14.      Sender sender = factory.produceMail();
15.      sender.send();
16.  }
17. }
```

- 静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

```
1. public class SendFactory {
2.   public static Sender produceMail(){
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
3.         return new MailSender();
4.     }
5.
6.     public static Sender produceSms(){
7.         return new SmsSender();
8.     }
9. }
10.
11.
12. public class FactoryTest {
13.  public static void main(String[] args) {
14.      Sender sender = SendFactory.produceMail();
15.      sender.send();
16.  }
17. }
```

抽象工厂模式

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

```
1. public interface Provider {
2.   public Sender produce();
3. }
4. ---------------------------------------------------------------------------------------
5. public interface Sender {
6.   public void send();
7. }
8. ---------------------------------------------------------------------------------------
9. public class MailSender implements Sender {
10.
11. @Override
12. public void send() {
13.      System.out.println("this is mail sender!");
14. }
15. }
16. ---------------------------------------------------------------------------------------
17. public class SmsSender implements Sender {
18.
```


 Copyright© 2018 黑马程序员

```
19. @Override
20. public void send() {
21.     System.out.println("this is sms sender!");
22. }
23. }
24. ----------------------------------------------------------------------------------
25. public class SendSmsFactory implements Provider {
26.
27. @Override
28. public Sender produce() {
29.     return new SmsSender();
30. }
31. }
```

```
1. public class SendMailFactory implements Provider {
2.
3.  @Override
4.  public Sender produce() {
5.      return new MailSender();
6.  }
7. }
8. ----------------------------------------------------------------------------------
9. public class Test {
10. public static void main(String[] args) {
11.     Provider provider = new SendMailFactory();
12.     Sender sender = provider.produce();
13.     sender.send();
14. }
15. }
```

4. 建造者模式（Builder）

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的 Test 结合起来得到的。

```
1. public class Builder {
2.  private List<Sender> list = new ArrayList<Sender>();
3.
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
4.  public void produceMailSender(int count) {
5.      for (int i = 0; i < count; i++) {
6.          list.add(new MailSender());
7.      }
8.  }
9.
10. public void produceSmsSender(int count) {
11.     for (int i = 0; i < count; i++) {
12.         list.add(new SmsSender());
13.     }
14. }
15.}
```

```
1. public class Builder {
2.  private List<Sender> list = new ArrayList<Sender>();
3.
4.  public void produceMailSender(int count) {
5.      for (int i = 0; i < count; i++) {
6.          list.add(new MailSender());
7.      }
8.  }
9.
10. public void produceSmsSender(int count) {
11.     for (int i = 0; i < count; i++) {
12.         list.add(new SmsSender());
13.     }
14. }
15.}
```

```
1. public class TestBuilder {
2.  public static void main(String[] args) {
3.      Builder builder = new Builder();
4.      builder.produceMailSender(10);
5.  }
6. }
```

5. 适配器设计模式

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容

http://www.itheima.com Copyright© 2018 黑马程序员

性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

类的适配器模式

```
1. public class Source {
2.  public void method1() {
3.      System.out.println("this is original method!");
4.  }
5. }
6. ----------------------------------------------------------
7. public interface Targetable {
8.  /* 与原类中的方法相同 */
9.  public void method1();
10. /* 新类的方法 */
11. public void method2();
12. }
13. public class Adapter extends Source implements Targetable {
14. @Override
15. public void method2() {
16.     System.out.println("this is the targetable method!");
17. }
18. }
19. public class AdapterTest {
20. public static void main(String[] args) {
21.     Targetable target = new Adapter();
22.     target.method1();
23.     target.method2();
24. }
25. }
```

对象的适配器模式

基本思路和类的适配器模式相同，只是将 Adapter 类作修改，这次不继承 Source 类，而是持有 Source 类的实例，以达到解决兼容性的问题。

```
1. public class Wrapper implements Targetable {
2.  private Source source;
3.
4.  public Wrapper(Source source) {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
5.      super();
6.      this.source = source;
7.  }
8.
9.  @Override
10. public void method2() {
11.     System.out.println("this is the targetable method!");
12. }
13.
14. @Override
15. public void method1() {
16.     source.method1();
17. }
18. }
19. ----------------------------------------------------------------
20. public class AdapterTest {
21.
22.     public static void main(String[] args) {
23.         Source source = new Source();
24.         Targetable target = new Wrapper(source);
25.         target.method1();
26.         target.method2();
27.     }
28. }
```

接口的适配器模式

接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

6. 装饰模式（Decorator）

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个

http://www.itheima.com Copyright© 2018 黑马程序员

接口，装饰对象持有被装饰对象的实例。

```
1. public interface Sourceable {
2.   public void method();
3. }
4. ---------------------------------------------------
5. public class Source implements Sourceable {
6.   @Override
7.   public void method() {
8.       System.out.println("the original method!");
9.   }
10. }
11. ---------------------------------------------------
12. public class Decorator implements Sourceable {
13.  private Sourceable source;
14.  public Decorator(Sourceable source) {
15.      super();
16.      this.source = source;
17.  }
18.
19.  @Override
20.  public void method() {
21.      System.out.println("before decorator!");
22.      source.method();
23.      System.out.println("after decorator!");
24.  }
25. }
26. ---------------------------------------------------
27. public class DecoratorTest {
28.  public static void main(String[] args) {
29.      Sourceable source = new Source();
30.      Sourceable obj = new Decorator(source);
31.      obj.method();
32.  }
33. }
```

7. 策略模式（strategy）

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法

http://www.itheima.com Copyright© 2018 黑马程序员

的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数。策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

```
1. public interface ICalculator {
2.  public int calculate(String exp);
3. }
4. ----------------------------------------------------------
5. public class Minus extends AbstractCalculator implements ICalculator {
6.
7.  @Override
8.  public int calculate(String exp) {
9.      int arrayInt[] = split(exp, "-");
10.     return arrayInt[0] - arrayInt[1];
11. }
12. }
13. ----------------------------------------------------------
14. public class Plus extends AbstractCalculator implements ICalculator {
15.
16. @Override
17. public int calculate(String exp) {
18.     int arrayInt[] = split(exp, "\\+");
19.     return arrayInt[0] + arrayInt[1];
20. }
21. }
22. ----------------------------------------------------------
23. public class AbstractCalculator {
24. public int[] split(String exp, String opt) {
25.     String array[] = exp.split(opt);
26.     int arrayInt[] = new int[2];
27.     arrayInt[0] = Integer.parseInt(array[0]);
28.     arrayInt[1] = Integer.parseInt(array[1]);
29.     return arrayInt;
30. }
31. }
```

```
1. public class StrategyTest {
2.  public static void main(String[] args) {
3.      String exp = "2+8";
4.      ICalculator cal = new Plus();
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
5.      int result = cal.calculate(exp);
6.      System.out.println(result);
7.  }
8. }
```

8. 观察者模式（Observer）

观察者模式很好理解，类似于邮件订阅和 RSS 订阅，当我们浏览一些博客或 wiki 时，经常会看到 RSS 图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

```
1. public interface Observer {
2.  public void update();
3. }
4.
5. public class Observer1 implements Observer {
6.  @Override
7.  public void update() {
8.       System.out.println("observer1 has received!");
9.  }
10. }
11.
12. public class Observer2 implements Observer {
13. @Override
14. public void update() {
15.      System.out.println("observer2 has received!");
16. }
17. }
18.
19. public interface Subject {
20.  /*增加观察者*/
21.    public void add(Observer observer);
22.
23.    /*删除观察者*/
24.    public void del(Observer observer);
25.    /*通知所有的观察者*/
1.    public void notifyObservers();
2.
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
3.    /*自身的操作*/
4.    public void operation();
5. }
6.
7. public abstract class AbstractSubject implements Subject {
8.
9.  private Vector<Observer> vector = new Vector<Observer>();
10.
11. @Override
12. public void add(Observer observer) {
13.     vector.add(observer);
14. }
15.
16. @Override
17. public void del(Observer observer) {
18.     vector.remove(observer);
19. }
20.
21. @Override
22. public void notifyObservers() {
23.     Enumeration<Observer> enumo = vector.elements();
24.     while (enumo.hasMoreElements()) {
25.         enumo.nextElement().update();
26.     }
27. }
28. }
29.
30. public class MySubject extends AbstractSubject {
31.
32. @Override
33. public void operation() {
34.     System.out.println("update self!");
35.       notifyObservers();
36. }
37. }
38.
39. public class ObserverTest {
40. public static void main(String[] args) {
41.     Subject sub = new MySubject();
42.     sub.add(new Observer1());
43. sub.add(new Observer2());
44.     sub.operation();
45. }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
46. }
```

9. JVM 垃圾回收机制和常见算法

理论上来讲 Sun 公司只定义了垃圾回收机制规则而不局限于其实现算法，因此不同厂商生产的虚拟机采用的算法也不尽相同。

GC（Garbage Collector）在回收对象前首先必须发现那些无用的对象，如何去发现定位这些无用的对象？常用的**搜索算法**如下：

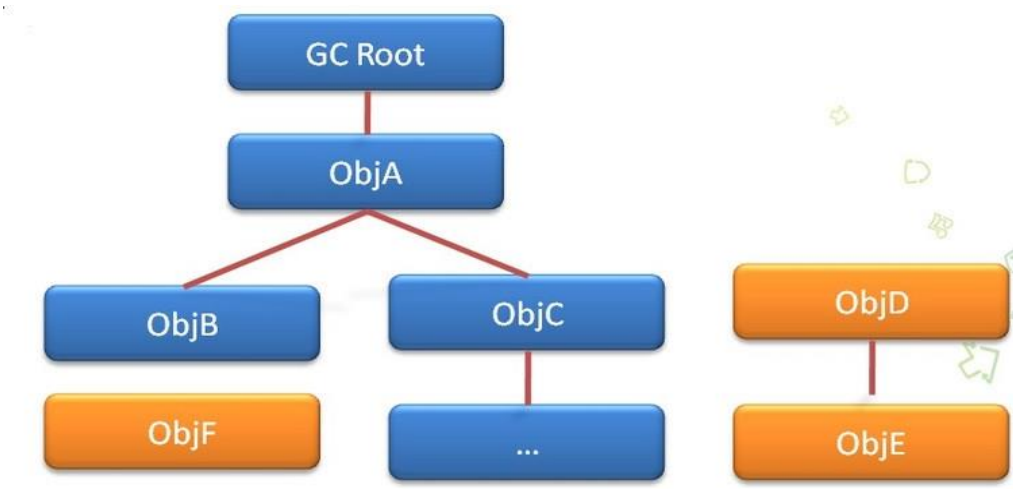
1）引用计数器算法（废弃）

引用计数器算法是给每个对象设置一个计数器，当有地方引用这个对象的时候，计数器+1，当引用失效的时候，计数器-1，当计数器为 0 的时候，JVM 就认为对象不再被使用，是“垃圾”了。

引用计数器实现简单，效率高；但是不能解决循环引用问问题（A 对象引用 B 对象，B 对象又引用 A 对象，但是 A,B 对象已不被任何其他对象引用），同时每次计数器的增加和减少都带来了很多额外的开销，所以在 JDK1.1 之后，这个算法已经不再使用了。

2）根搜索算法（使用）

根搜索算法是通过一些“GC Roots”对象作为起点，从这些节点开始往下搜索，搜索通过的路径成为引用链（Reference Chain），当一个对象没有被 GC Roots 的引用链连接的时候，说明这个对象是不可用的。



Copyright© 2018 黑马程序员

GC Roots 对象包括：

a) 虚拟机栈（栈帧中的本地变量表）中的引用的对象。

b) 方法区域中的类静态属性引用的对象。

c) 方法区域中常量引用的对象。

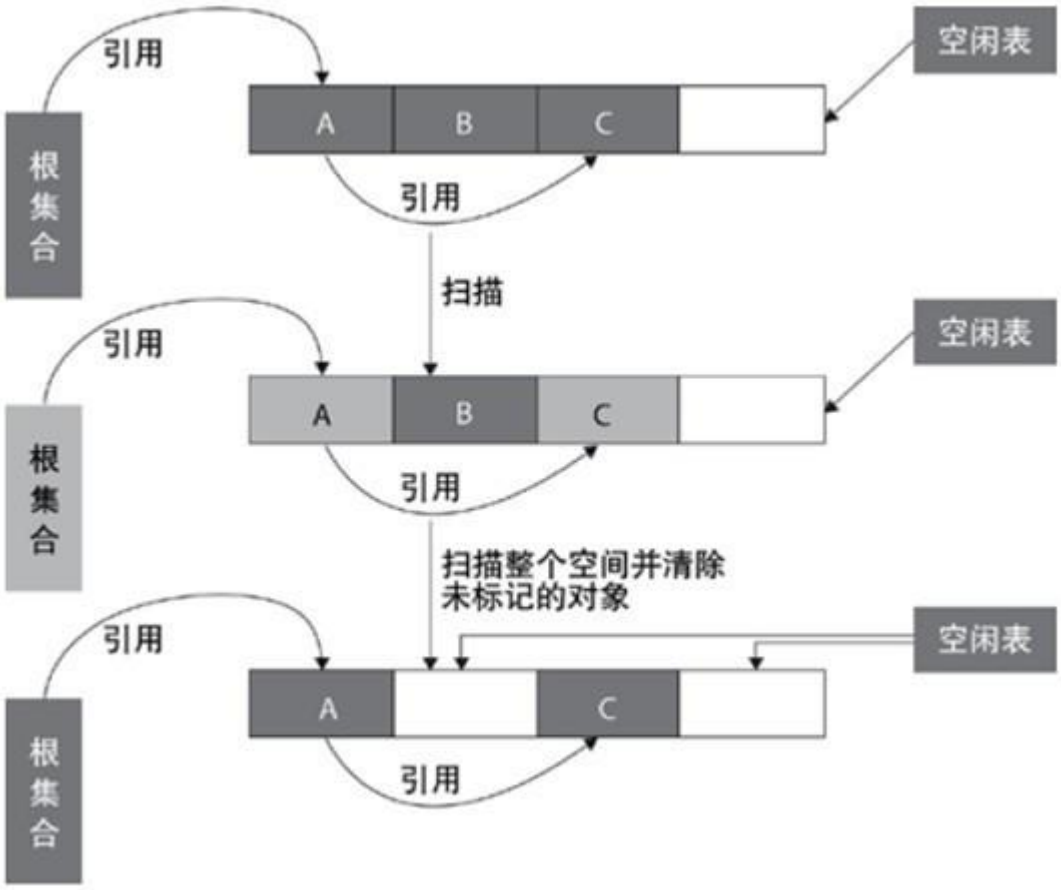
d) 本地方法栈中 JNI（Native 方法）的引用的对象。

通过上面的算法搜索到无用对象之后，就是回收过程，**回收算法**如下：

1）标记—清除算法（Mark-Sweep）（DVM 使用的算法）

标记—清除算法包括两个阶段：“标记”和“清除”。在标记阶段，确定所有要回收的对象，并做标记。清除阶段紧随标记阶段，将标记阶段确定不可用的对象清除。标记—清除算法是基础的收集算法，标记和清除阶段的效率不高，而且清除后回产生大量的不连续空间，这样当程序需要分配大内存对象时，可能无法找到足够的连续空间。

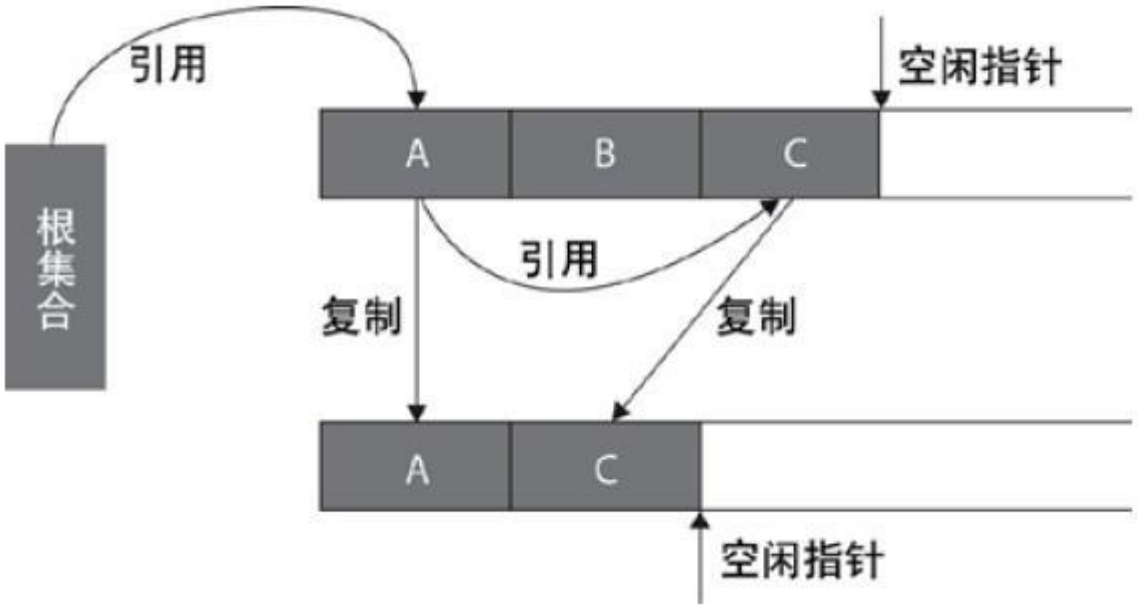
http://www.itheima.com Copyright© 2018 黑马程序员



2）复制算法（Copying）

复制算法是把内存分成大小相等的两块，每次使用其中一块，当垃圾回收的时候，把存活的对象复制到另一块上，然后把这块内存整个清理掉。复制算法实现简单，运行效率高，但是由于每次只能使用其中的一半，造成内存的利用率不高。现在的 JVM 用复制方法收集新生代，由于新生代中大部分对象（98%）都是朝生夕死的，所以两块内存的比例不是 1:1(大概是 8:1)。

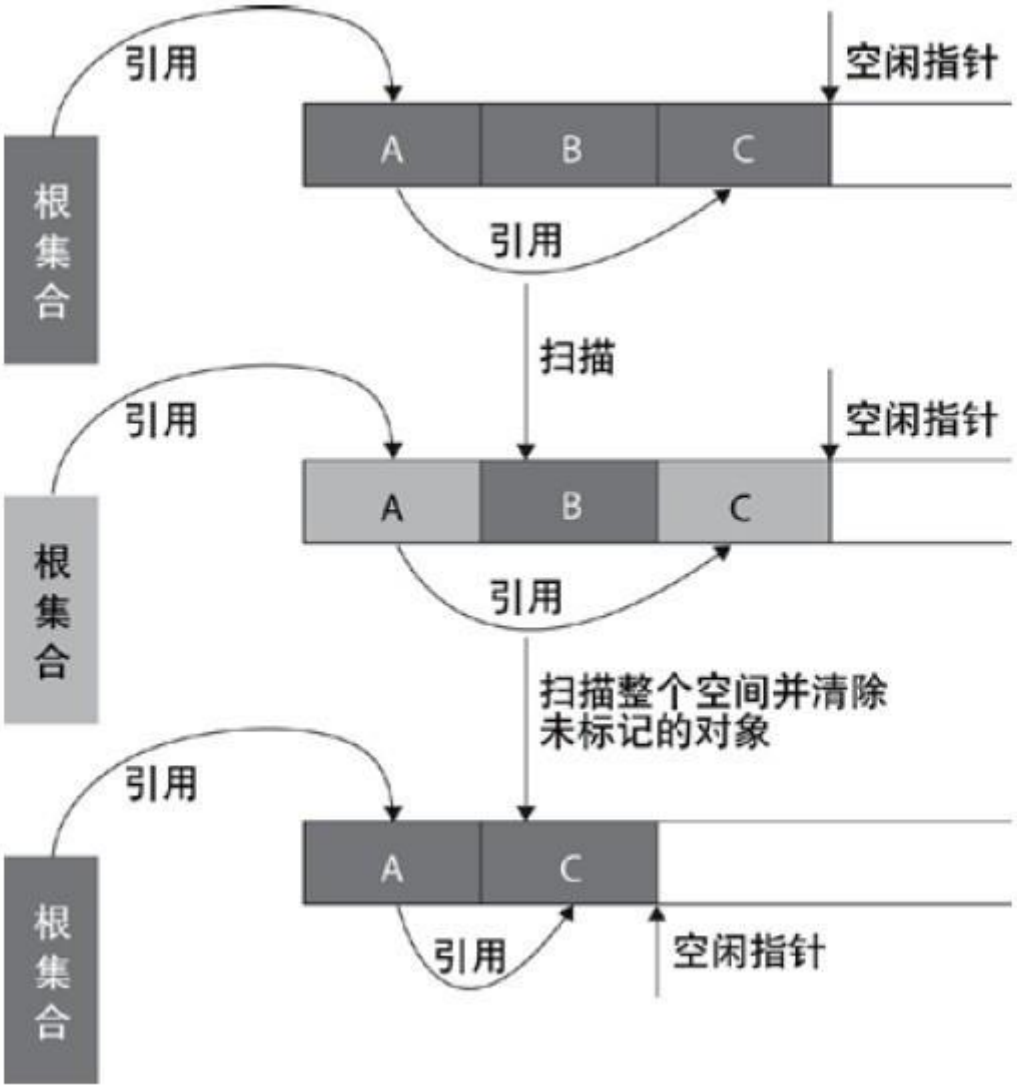
http://www.itheima.com Copyright© 2018 黑马程序员



3）标记—整理算法（Mark-Compact）

标记—整理算法和标记—清除算法一样，但是标记—整理算法不是把存活对象复制到另一块内存，而是把存活对象往内存的一端移动，然后直接回收边界以外的内存。标记—整理算法提高了内存的利用率，并且它适合在收集对象存活时间较长的老年代。

http://www.itheima.com Copyright© 2018 黑马程序员



4）分代收集（Generational Collection）

分代收集是根据对象的存活时间把内存分为新生代和老年代，根据各个代对象的存活特点，每个代采用不同的垃圾回收算法。新生代采用复制算法，老年代采用标记—整理算法。垃圾算法的实现涉及大量的程序细节，而且不同的虚拟机平台实现的方法也各不相同。

10. 谈谈 JVM 的内存结构和内存分配

a) Java 内存模型

http://www.itheima.com Copyright© 2018 黑马程序员

Java 虚拟机将其管辖的内存大致分三个逻辑部分：方法区(Method Area)、Java 栈和 Java 堆。

1、方法区是静态分配的，编译器将变量绑定在某个存储位置上，而且这些绑定不会在运行时改变。

常数池，源代码中的命名常量、String 常量和 static 变量保存在方法区。

2、Java Stack 是一个逻辑概念，特点是后进先出。一个栈的空间可能是连续的，也可能是不连续的。

最典型的 Stack 应用是方法的调用，Java 虚拟机每调用一次方法就创建一个方法帧（frame），退出该方法则对应的 方法帧被弹出(pop)。栈中存储的数据也是运行时确定的。

3、Java 堆分配(heap allocation)意味着以随意的顺序，在运行时进行存储空间分配和收回的内存管理模型。

堆中存储的数据常常是大小、数量和生命期在编译时无法确定的。Java 对象的内存总是在 heap 中分配。

我们每天都在写代码，每天都在使用 JVM 的内存。

b) java 内存分配

1、基础数据类型直接在栈空间分配;

2、方法的形式参数，直接在栈空间分配，当方法调用完成后从栈空间回收;

3、引用数据类型，需要用 new 来创建，既在栈空间分配一个地址空间，又在堆空间分配对象的类变量;

4、方法的引用参数，在栈空间分配一个地址空间，并指向堆空间的对象区，当方法调用完后从栈空间回收;

5、局部变量 new 出来时，在栈空间和堆空间中分配空间，当局部变量生命周期结束后，栈空间立刻被回收，堆空间区域等待 GC 回收;

6、方法调用时传入的实际参数，先在栈空间分配，在方法调用完成后从栈空间释放;

7、字符串常量在 DATA 区域分配 ，this 在堆空间分配;

8、数组既在栈空间分配数组名称， 又在堆空间分配数组实际的大小!

http://www.itheima.com Copyright© 2018 黑马程序员

11. Java 中引用类型都有哪些？（重要）

Java 中对象的引用分为四种级别，这四种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

- 强引用（StrongReference）

这个就不多说，我们写代码天天在用的就是强引用。如果一个对象被被人拥有强引用，那么垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

Java 的对象是位于 heap 中的，heap 中对象有强可及对象、软可及对象、弱可及对象、虚可及对象和不可到达对象。应用的强弱顺序是强、软、弱、和虚。对于对象是属于哪种可及的对象，由他的最强的引用决定。如下代码：

```
1.      String abc=new String("abc");  //1
2.      SoftReference<String> softRef=new SoftReference<String>(abc);  //2
3.      WeakReference<String> weakRef = new WeakReference<String>(abc); //3
4.      abc=null; //4
5.      softRef.clear();//5
```

第一行在 heap 堆中创建内容为“abc”的对象，并建立 abc 到该对象的强引用，该对象是强可及的。

第二行和第三行分别建立对 heap 中对象的软引用和弱引用，此时 heap 中的 abc 对象已经有 3 个引用，显然此时 abc 对象仍是强可及的。

第四行之后 heap 中对象不再是强可及的，变成软可及的。

第五行执行之后变成弱可及的。

- 软引用（SoftReference）

如果一个对象只具有软引用，那么如果内存空间足够，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。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高

http://www.itheima.com Copyright© 2018 黑马程序员

速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，Java 虚拟机就会把这个软引用加入到与之关联的引用队列中。

软引用是主要用于内存敏感的高速缓存。在 jvm 报告内存不足之前会清除所有的软引用，这样以来 gc 就有可能收集软可及的对象，可能解决内存吃紧问题，避免内存溢出。什么时候会被收集取决于 gc 的算法和 gc 运行时可用内存的大小。当 gc 决定要收集软引用时执行以下过程,以上面的 softRef 为例：

1 首先将 softRef 的 referent（abc）设置为 null，不再引用 heap 中的 new String("abc")对象。

2 将 heap 中的 new String("abc")对象设置为可结束的(finalizable)。

3 当 heap 中的 new String("abc")对象的 finalize()方法被运行而且该对象占用的内存被释放， softRef 被添加到它的 ReferenceQueue(如果有的话)中。

注意:对 ReferenceQueue 软引用和弱引用可以有可无，但是虚引用必须有。

被 Soft Reference 指到的对象，即使没有任何 Direct Reference，也不会被清除。一直要到 JVM 内存不足且没有 Direct Reference 时才会清除，SoftReference 是用来设计 object-cache 之用的。如此一来 SoftReference 不但可以把对象 cache 起来，也不会造成内存不足的错误 （OutOfMemoryError）。


弱引用（WeakReference）

如果一个对象只具有弱引用，那该类就是可有可无的对象，因为只要该对象被 gc 扫描到了随时都会把它干掉。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

http://www.itheima.com Copyright© 2018 黑马程序员

虚引用（PhantomReference）

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。虚引用主要用来跟踪对象被垃圾回收的活动。

虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

建立虚引用之后通过 get 方法返回结果始终为 null,通过源代码你会发现,虚引用通向会把引用的对象写进 referent,只是 get 方法返回结果为 null。先看一下和 gc 交互的过程再说一下他的作用。

1 不把 referent 设置为 null，直接把 heap 中的 new String("abc")对象设置为可结束的(finalizable)。

2 与软引用和弱引用不同，先把 PhantomRefrence 对象添加到它的 ReferenceQueue 中.然后在释放虚可及的对象。

12. heap 和 stack 有什么区别（2017-2-23）

从以下几个方面阐述堆（heap）和栈（stack）的区别。

1. 申请方式

stack:由系统自动分配。例如，声明在函数中一个局部变量 int b; 系统自动在栈中为 b 开辟空间

heap:需要程序员自己申请，并指明大小，在 c 中 malloc 函数，对于 Java 需要手动 new Object()的形式开辟

2. 申请后系统的响应

stack：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

http://www.itheima.com Copyright© 2018 黑马程序员

heap：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

3. 申请大小的限制

stack：栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS 下，栈的大小是 2M（也有的说是 1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示 overflow。因此，能从栈获得的空间较小。

heap：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

4. 申请效率的比较：

stack：由系统自动分配，速度较快。但程序员是无法控制的。

heap：由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。

5. heap 和 stack 中的存储内容

stack： 在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的 C 编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。

当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

heap：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

http://www.itheima.com Copyright© 2018 黑马程序员

6. 数据结构层面的区别

还有就是数据结构方面的堆和栈，这些都是不同的概念。这里的堆实际上指的就是（满足堆性质的）优先队列的一种数据结构，第 1 个元素有最高的优先权；栈实际上就是满足先进后出的性质的数学或数据结构。

虽然堆栈，堆栈的说法是连起来叫，但是他们还是有很大区别的，连着叫只是由于历史的原因。

7. 拓展知识（Java 中堆栈的应用）

1). 栈(stack)与堆(heap)都是 Java 用来在 Ram 中存放数据的地方。与 C++不同，Java 自动管理栈和堆，程序员不能直接地设置栈或堆。

2). 栈的优势是，存取速度比堆要快，仅次于直接位于 CPU 中的寄存器。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。另外，栈数据可以共享，详见第 3 点。堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，Java 的垃圾回收器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。

3). Java 中的数据类型有两种。

一种是基本类型(primitive types)，共有 8 种，即 int, short, long, byte, float, double, boolean, char(注意，并没有 string 的基本类型)。这种类型的定义是通过诸如 int a = 3; long b = 255L;的形式来定义的，称为自动变量（自动变量：只在定义它们的时候才创建，在定义它们的函数返回时系统回收变量所占存储空间。对这些变量存储空间的分配和回收是由系统自动完成的。）。值得注意的是，自动变量存的是字面值，不是类的实例，即不是类的引用，这里并没有类的存在。如 int a = 3; 这里的 a 是一个指向 int 类型的引用，指向 3 这个字面值。这些字面值的数据，由于大小可知，生存期可知(这些字面值固定定义在某个程序块里面，程序块退出后，字段值就消失了)，出于追求速度的原因，就存在于栈中。

另外，栈有一个很重要的特殊性，就是存在栈中的数据可以共享。假设我们同时定义

int a = 3;

http://www.itheima.com Copyright© 2018 黑马程序员

int b = 3;

编译器先处理 int a = 3；首先它会在栈中创建一个变量为 a 的引用，然后查找有没有字面值为 3 的地址，没找到，就开辟一个存放 3 这个字面值的地址，然后将 a 指向 3 的地址。接着处理 int b = 3；在创建完 b 的引用变量后，由于在栈中已经有 3 这个字面值，便将 b 直接指向 3 的地址。这样，就出现了 a 与 b 同时均指向 3 的情况。

特别注意的是，这种字面值的引用与类对象的引用不同。假定两个类对象的引用同时指向一个对象，如果一个对象引用变量修改了这个对象的内部状态，那么另一个对象引用变量也即刻反映出这个变化。相反，通过字面值的引用来修改其值，不会导致另一个指向此字面值的引用的值也跟着改变的情况。如上例，我们定义完 a 与 b 的值后，再令 a=4；那么，b 不会等于 4，还是等于 3。在编译器内部，遇到 a=4；时，它就会重新搜索栈中是否有 4 的字面值，如果没有，重新开辟地址存放 4 的值；如果已经有了，则直接将 a 指向这个地址。因此 a 值的改变不会影响到 b 的值。

另一种是包装类数据，如 Integer, String, Double 等将相应的基本数据类型包装起来的类。这些类数据全部存在于堆中，Java 用 new()语句来显示地告诉编译器，在运行时才根据需要动态创建，因此比较灵活，但缺点是要占用更多的时间。

4).每个 JVM 的线程都有自己的私有的栈空间，随线程创建而创建，java 的 stack 存放的是 frames，java 的 stack 和 c 的不同，只是存放本地变量，返回值和调用方法，不允许直接 push 和 pop frames ，因为 frames 可能是有 heap 分配的，所以 java 的 stack 分配的内存不需要是连续的。java 的 heap 是所有线程共享的，堆存放所有 runtime data ，里面是所有的对象实例和数组，heap 是 JVM 启动时创建。

5). String 是一个特殊的包装类数据。即可以用 String str = new String("abc");的形式来创建，也可以用 String str = "abc"；的形式来创建(作为对比，在 JDK 5.0 之前，你从未见过 Integer i = 3;的表达式，因为类与字面值是不能通用的，除了 String。而在 JDK 5.0 中，这种表达式是可以的！因为编译器在后台进行 Integer i = new Integer(3)的转换)。前者是规范的类的创建过程，即在 Java 中，一切都是对象，而对象是类的实例，全部通过 new()

http://www.itheima.com Copyright© 2018 黑马程序员

的形式来创建。那为什么在 String str = "abc"；中，并没有通过 new()来创建实例，是不是违反了上述原则？其实没有。

5.1). 关于 String str = "abc"的内部工作。Java 内部将此语句转化为以下几个步骤：

(1)先定义一个名为 str 的对 String 类的对象引用变量：String str；

(2)在栈中查找有没有存放值为"abc"的地址，如果没有，则开辟一个存放字面值为"abc"的地址，接着创建一个新的 String 类的对象 o，并将 o 的字符串值指向这个地址，而且在栈中这个地址旁边记下这个引用的对象 o。如果已经有了值为"abc"的地址，则查找对象 o，并返回 o 的地址。

(3)将 str 指向对象 o 的地址。

值得注意的是，一般 String 类中字符串值都是直接存值的。但像 String str = "abc"；这种场合下，其字符串值却是保存了一个指向存在栈中数据的引用！

为了更好地说明这个问题，我们可以通过以下的几个代码进行验证。

```
String str1 = "abc";

String str2 = "abc";

System.out.println(str1==str2);    //true
```

注意，我们这里并不用 str1.equals(str2)；的方式，因为这将比较两个字符串的值是否相等。==号，根据 JDK 的说明，只有在两个引用都指向了同一个对象时才返回真值。而我们在这里要看的是，str1 与 str2 是否都指向了同一个对象。

结果说明，JVM 创建了两个引用 str1 和 str2，但只创建了一个对象，而且两个引用都指向了这个对象。

我们再来更进一步，将以上代码改成：

```
String str1 = "abc";

String str2 = "abc";
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
str1 = "bcd";
System.out.println(str1 + "," + str2);     //bcd, abc
System.out.println(str1==str2);     //false
```

这就是说，赋值的变化导致了类对象引用的变化，str1 指向了另外一个新对象！而 str2 仍旧指向原来的对象。上例中，当我们将 str1 的值改为"bcd"时，JVM 发现在栈中没有存放该值的地址，便开辟了这个地址，并创建了一个新的对象，其字符串的值指向这个地址。

事实上，String 类被设计成为不可改变(immutable)的类。如果你要改变其值，可以，但 JVM 在运行时根据新值悄悄创建了一个新对象，然后将这个对象的地址返回给原来类的引用。这个创建过程虽说是完全自动进行的，但它毕竟占用了更多的时间。在对时间要求比较敏感的环境中，会带有一定的不良影响。

再修改原来代码：

```
String str1 = "abc";
String str2 = "abc";
str1 = "bcd";
String str3 = str1;
System.out.println(str3);     //bcd
String str4 = "bcd";
System.out.println(str1 == str4);     //true
```

str3 这个对象的引用直接指向 str1 所指向的对象(注意，str3 并没有创建新对象)。当 str1 改完其值后，再创建一个 String 的引用 str4，并指向因 str1 修改值而创建的新的对象。可以发现，这回 str4 也没有创建新的对象，从而再次实现栈中数据的共享。

我们再接着看以下的代码。

http://www.itheima.com Copyright© 2018 黑马程序员

String str1 = new String("abc");

String str2 = "abc";

System.out.println(str1==str2); //false

创建了两个引用。创建了两个对象。两个引用分别指向不同的两个对象。

以上两段代码说明，只要是用 new()来新建对象的，都会在堆中创建，而且其字符串是单独存值的，即使与栈中的数据相同，也不会与栈中的数据共享。

6). 数据类型包装类的值不可修改。不仅仅是 String 类的值不可修改，所有的数据类型包装类都不能更改其内部的值。

7). 结论与建议：

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

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

(3)当比较包装类里面的数值是否相等时,用 equals()方法;当测试两个包装类的引用是否指向同一个对象时，用==。

(4)由于 String 类的 immutable 性质，当 String 变量需要经常变换其值时，应该考虑使用 StringBuffer 类，

http://www.itheima.com Copyright© 2018 黑马程序员

以提高程序效率。

如果 java 不能成功分配 heap 的空间，将抛出 OutOfMemoryError。

13. 解释内存中的栈（stack）、堆（heap）和方法区（method area）的用法（2017-11-12-wl）

通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用 JVM 中的栈空间；而通过 new 关键字和构造器创建的对象则放在堆空间，堆是垃圾收集器管理的主要区域，由于现在的垃圾收集器都采用分代收集算法，所以堆空间还可以细分为新生代和老生代，再具体一点可以分为 Eden、Survivor（又可分为 From Survivor 和 To Survivor）、Tenured；方法区和堆都是各个线程共享的内存区域，用于存储已经被 JVM 加载的类信息、常量、静态变量、JIT 编译器编译后的代码等数据；程序中的字面量（literal）如直接书写的 100、"hello"和常量都是放在常量池中，常量池是方法区的一部分。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，栈和堆的大小都可以通过 JVM 的启动参数来进行调整，栈空间用光了会引发 StackOverflowError，而堆和常量池空间不足则会引发 OutOfMemoryError。

String str = new String("hello");

上面的语句中变量 str 放在栈上，用 new 创建出来的字符串对象放在堆上，而"hello"这个字面量是放在方法区的。

四、Java 的类加载器（2015-12-2）

1. Java 的类加载器的种类都有哪些?

1、根类加载器(Bootstrap) --C++写的 ，看不到源码

2、扩展类加载器（Extension） --加载位置 ： jre\lib\ext 中

http://www.itheima.com Copyright© 2018 黑马程序员

3、系统(应用)类加载器(System\App) --加载位置 ： classpath 中

4、自定义加载器(必须继承 ClassLoader)

2. 类什么时候被初始化?

1）创建类的实例，也就是 new 一个对象

2）访问某个类或接口的静态变量，或者对该静态变量赋值

3）调用类的静态方法

4）反射（Class.forName("com.lyj.load")）

5）初始化一个类的子类（会首先初始化子类的父类）

6）JVM 启动时标明的启动类，即文件名和类名相同的那个类

只有这 6 中情况才会导致类的类的初始化。

类的初始化步骤:

1）如果这个类还没有被加载和链接，那先进行加载和链接

2）假如这个类存在直接父类，并且这个类还没有被初始化（注意：在一个类加载器中，类只能初始化一次），那就初始化直接的父类（不适用于接口）

3)加入类中存在初始化语句（如 static 变量和 static 块），那就依次执行这些初始化语句。

3. Java 类加载体系之 ClassLoader 双亲委托机制 （2017-2-24）

java 是一种类型安全的语言，它有四类称为安全沙箱机制的安全机制来保证语言的安全性，这四类安全沙箱分别是:

1） 类加载体系

2） .class 文件检验器

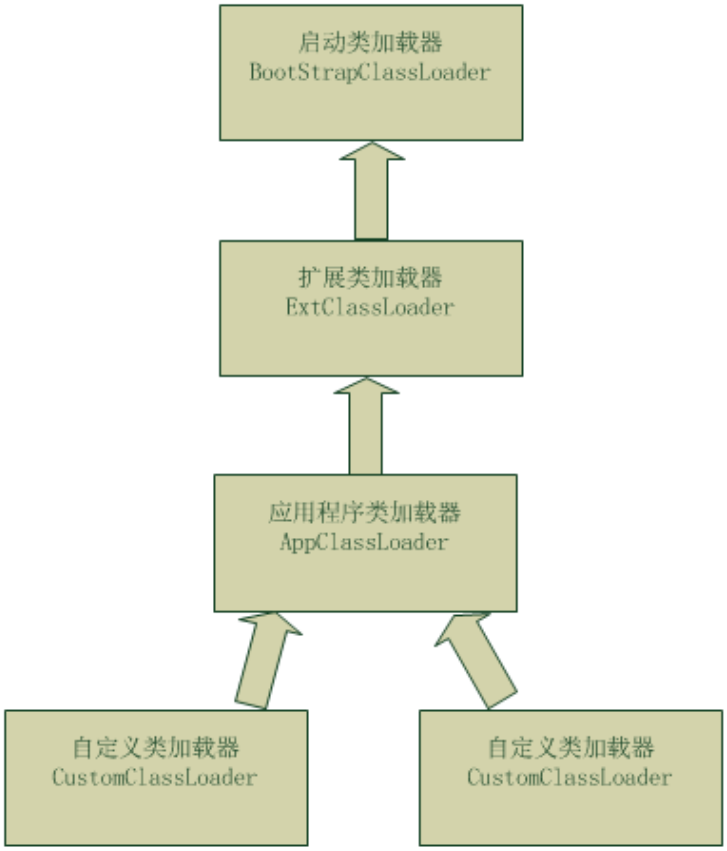
http://www.itheima.com Copyright© 2018 黑马程序员

3） 内置于 Java 虚拟机（及语言）的安全特性

4） 安全管理器及 Java API

主要讲解类的加载体系：

java 程序中的 .java 文件编译完会生成 .class 文件，而 .class 文件就是通过被称为类加载器的 ClassLoader 加载的，而 ClassLoder 在加载过程中会使用“双亲委派机制”来加载 .class 文件，先上图：



BootStrapClassLoader：启动类加载器，该 ClassLoader 是 jvm 在启动时创建的，用于加载 $JAVA_HOME$/jre/lib 下面的类库（或者通过参数-Xbootclasspath 指定）。由于启动类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不能直接通过引用进行操作。

ExtClassLoader:扩展类加载器,该ClassLoader是在sun.misc.Launcher里作为一个内部类ExtClassLoader定义的（即 sun.misc.Launcher$ExtClassLoader），ExtClassLoader 会加载 $JAVA_HOME/jre/lib/ext 下的类

http://www.itheima.com Copyright© 2018 黑马程序员

库（或者通过参数-Djava.ext.dirs 指定）。

AppClassLoader：应用程序类加载器，该 ClassLoader 同样是在 sun.misc.Launcher 里作为一个内部类 AppClassLoader 定义的（即 sun.misc.Launcher$AppClassLoader），AppClassLoader 会加载 java 环境变量 CLASSPATH 所指定的路径下的类库，而 CLASSPATH 所指定的路径可以通过 System.getProperty("java.class.path")获取；当然，该变量也可以覆盖，可以使用参数-cp，例如：java -cp 路径 （可以指定要执行的 class 目录）。

CustomClassLoader：自定义类加载器，该 ClassLoader 是指我们自定义的 ClassLoader，比如 tomcat 的 StandardClassLoader 属于这一类；当然，大部分情况下使用 AppClassLoader 就足够了。

前面谈到了 ClassLoader 的几类加载器，而 ClassLoader 使用双亲委派机制来加载 class 文件的。ClassLoader 的双亲委派机制是这样的（这里先忽略掉自定义类加载器 CustomClassLoader）：

1）当 AppClassLoader 加载一个 class 时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器 ExtClassLoader 去完成。

2）当 ExtClassLoader 加载一个 class 时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给 BootStrapClassLoader 去完成。

3）如果 BootStrapClassLoader 加载失败（例如在$JAVA_HOME$/jre/lib 里未查找到该 class），会使用 ExtClassLoader 来尝试加载；

4）若 ExtClassLoader 也加载失败，则会使用 AppClassLoader 来加载，如果 AppClassLoader 也加载失败，则会报出异常 ClassNotFoundException。

下面贴下 ClassLoader 的 loadClass(String name, boolean resolve)的源码：

```
1. protected synchronized Class<?> loadClass(String name, boolean resolve)
2.  throws ClassNotFoundException {
3.       // 首先找缓存是否有 class
4.       Class c = findLoadedClass(name);
5.       if (c == null) {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
6.          //没有判断有没有父类
7.          try {
8.              if (parent != null) {
9.                  //有的话，用父类递归获取 class
10.                 c = parent.loadClass(name, false);
11.             } else {
12.                 //没有父类。通过这个方法来加载
13.                 c = findBootstrapClassOrNull(name);
14.             }
15.         } catch (ClassNotFoundException e) {
16.             // ClassNotFoundException thrown if class not found
17.             // from the non-null parent class loader
18.         }
19.         if (c == null) {
20.             // 如果还是没有找到，调用 findClass(name)去找这个类
21.             c = findClass(name);
22.         }
23.     }
24.     if (resolve) {
25.         resolveClass(c);
26.     }
27.     return c;
28. }
```

代码很明朗：首先找缓存（findLoadedClass），没有的话就判断有没有 parent，有的话就用 parent 来递归的 loadClass，然而 ExtClassLoader 并没有设置 parent，则会通过 findBootstrapClassOrNull 来加载 class，而 findBootstrapClassOrNull 则会通过 JNI 方法”private native Class findBootstrapClass(String name)“来使用 BootStrapClassLoader 来加载 class。

然后如果 parent 未找到 class，则会调用 findClass 来加载 class，findClass 是一个 protected 的空方法，可以覆盖它以便自定义 class 加载过程。

另外，虽然 ClassLoader 加载类是使用 loadClass 方法，但是鼓励用 ClassLoader 的子类重写 findClass(String)，而不是重写 loadClass，这样就不会覆盖了类加载默认的双亲委派机制。

双亲委派托机制为什么安全

举个例子，ClassLoader 加载的 class 文件来源很多，比如编译器编译生成的 class、或者网络下载的字节码。

http://www.itheima.com Copyright© 2018 黑马程序员

而一些来源的 class 文件是不可靠的，比如我可以自定义一个 java.lang.Integer 类来覆盖 jdk 中默认的 Integer 类，例如下面这样：

```
1. package java.lang;
2. public class Integer {
3.  public Integer(int value) {
4.      System.exit(0);
5.  }
6. }
```

初始化这个 Integer 的构造器是会退出 JVM，破坏应用程序的正常进行，如果使用双亲委派机制的话该 Integer 类永远不会被调用，以为委托 BootStrapClassLoader 加载后会加载 JDK 中的 Integer 类而不会加载自定义的这个，可以看下下面这测试个用例：

```
1. public static void main(String... args) {
2.       Integer i = new Integer(1);
3.       System.err.println(i);
4.   }
```

执行时 JVM 并未在 new Integer(1)时退出，说明未使用自定义的 Integer，于是就保证了安全性。

4. 描述一下 JVM 加载 class （2017-11-15-wl）

JVM 中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java 中的类加载器是一个重要的 Java 运行时系统组件，它负责在运行时查找和装入类文件中的类。

由于 Java 的跨平台性，经过编译的 Java 源程序并不是一个可执行程序，而是一个或多个类文件。当 Java 程序需要使用某个类时,JVM 会确保这个类已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class 文件中的数据读入到内存中，通常是创建一个字节数组读入.class 文件，然后产生与所加载类对应的 Class 对象。加载完成后，Class 对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后 JVM 对类进行初始化，包括：

http://www.itheima.com Copyright© 2018 黑马程序员

如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；

如果类中存在初始化语句，就依次执行这些初始化语句。类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader的子类）。

从 Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM 更好的保证了 Java 平台的安全性，在该机制中，JVM 自带的 Bootstrap 是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM 不会向 Java 程序提供对 Bootstrap 的引用。

下面是关于几个类加载器的说明：

- Bootstrap：一般用本地代码实现，负责加载 JVM 基础核心类库（rt.jar）；
- Extension：从 java.ext.dirs 系统属性所指定的目录中加载类库，它的父加载器是 Bootstrap；
- System：又叫应用类加载器，其父类是 Extension。它是应用最广泛的类加载器。它从环境变量 classpath 或者系统属性 java.class.path 所指定的目录中记载类，是用户自定义加载器的默认父加载器。

5. 获得一个类对象有哪些方式？（2017-11-23-wzz）

类型.class，例如：String.class

对象.getClass()，例如：”hello”.getClass()

Class.forName()，例如：Class.forName(“java.lang.String”)

http://www.itheima.com Copyright© 2018 黑马程序员

五、JVM 基础知识（2017-11-16-wl）

1. 既然有 GC 机制，为什么还会有内存泄露的情况 （2017-11-16-wl）

理论上 Java 因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是 Java 被广泛使用于服务器端编程的一个重要原因）。然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被 GC 回收，因此也会导致内存泄露的发生。

例如 hibernate 的 Session（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象，如果不及时关闭（close）或清空（flush）一级缓存就可能导致内存泄露。

下面例子中的代码也会导致内存泄露。

```
1. import java.util.Arrays;
2. import java.util.EmptyStackException;
3. public class MyStack<T> {
4.  private T[] elements;
5.  private int size = 0;
6.  private static final int INIT_CAPACITY = 16;
7.  public MyStack() {
8.      elements = (T[]) new Object[INIT_CAPACITY];
9.  }
10.  public void push(T elem) {
11.  ensureCapacity();
12.  elements[size++] = elem;
13.  }
14.  public T pop() {
15.  if(size == 0)throw new EmptyStackException();
16.   return elements[--size];
17. }
18.  private void ensureCapacity() {
19.  if(elements.length == size) {
20.    elements = Arrays.copyOf(elements, 2 * size + 1);
21.  }
22. }
23. }
```


http://www.itheima.com Copyright© 2018 黑马程序员

上面的代码实现了一个栈（先进后出（FILO））结构，乍看之下似乎没有什么明显的问题，它甚至可以通过你编写的各种单元测试。然而其中的 pop 方法却存在内存泄露的问题，当我们用 pop 方法弹出栈中的对象时，该对象不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，因为栈内部维护着对这些对象的过期引用（obsolete reference）。在支持垃圾回收的语言中，内存泄露是很隐蔽的，这种内存泄露其实就是无意识的对象保持。如果一个对象引用被无意识的保留起来了，那么垃圾回收器不会处理这个对象，也不会处理该对象引用的其他对象，即使这样的对象只有少数几个，也可能会导致很多的对象被排除在垃圾回收之外，从而对性能造成重大影响，极端情况下会引发 Disk Paging （物理内存与硬盘的虚拟内存交换数据），甚至造成 OutOfMemoryError。

六、GC 基础知识（2017-11-16-wl）

1. Java 中为什么会有 GC 机制呢？（2017-11-16-wl）

Java 中为什么会有 GC 机制呢?

- 安全性考虑；-- for security.
- 减少内存泄露；-- erase memory leak in some degree.
- 减少程序员工作量。-- Programmers don't worry about memory releasing.

2. 对于 Java 的 GC 哪些内存需要回收（2017-11-16-wl）

内存运行时 JVM 会有一个运行时数据区来管理内存。它主要包括 5 大部分：程序计数器(Program Counter Register)、虚拟机栈(VM Stack)、本地方法栈(Native Method Stack)、方法区(Method Area)、堆(Heap).

而其中程序计数器、虚拟机栈、本地方法栈是每个线程私有的内存空间，随线程而生，随线程而亡。例如栈中每一个栈帧中分配多少内存基本上在类结构确定是哪个时就已知了，因此这 3 个区域的内存分配和回收都是确定的，无需考虑内存回收的问题。

但方法区和堆就不同了，一个接口的多个实现类需要的内存可能不一样，我们只有在程序运行期间才会知道会创

http://www.itheima.com Copyright© 2018 黑马程序员

建哪些对象，这部分内存的分配和回收都是动态的，GC 主要关注的是这部分内存。

总而言之，GC 主要进行回收的内存是 JVM 中的方法区和堆；

3. Java 的 GC 什么时候回收垃圾（2017-11-16-wl）

在面试中经常会碰到这样一个问题（事实上笔者也碰到过）：如何判断一个对象已经死去?

很容易想到的一个答案是：对一个对象添加引用计数器。每当有地方引用它时，计数器值加 1；当引用失效时，计数器值减 1.而当计数器的值为 0 时这个对象就不会再被使用，判断为已死。是不是简单又直观。然而，很遗憾。这种做法是错误的！为什么是错的呢？事实上，用引用计数法确实在大部分情况下是一个不错的解决方案，而在实际的应用中也有不少案例，但它却无法解决对象之间的循环引用问题。比如对象 A 中有一个字段指向了对象 B，而对象 B 中也有一个字段指向了对象 A，而事实上他们俩都不再使用，但计数器的值永远都不可能为 0，也就不会被回收，然后就发生了内存泄露。

所以，正确的做法应该是怎样呢?

在 Java，C#等语言中，比较主流的判定一个对象已死的方法是：可达性分析(Reachability Analysis).所有生成的对象都是一个称为"GC Roots"的根的子树。从 GC Roots 开始向下搜索，搜索所经过的路径称为引用链(Reference Chain)，当一个对象到 GC Roots 没有任何引用链可以到达时，就称这个对象是不可达的（不可引用的），也就是可以被 GC 回收了。

无论是引用计数器还是可达性分析，判定对象是否存活都与引用有关！那么，如何定义对象的引用呢?

我们希望给出这样一类描述：当内存空间还够时，能够保存在内存中；如果进行了垃圾回收之后内存空间仍旧非常紧张，则可以抛弃这些对象。所以根据不同的需求，给出如下四种引用，根据引用类型的不同，GC 回收时也会有不同的操作：

1)强引用(Strong Reference):Object obj = new Object();只要强引用还存在，GC 永远不会回收掉被引用的对象。

http://www.itheima.com Copyright© 2018 黑马程序员

2)软引用(Soft Reference)：描述一些还有用但非必需的对象。在系统将会发生内存溢出之前，会把这些对象列入回收范围进行二次回收（即系统将会发生内存溢出了，才会对他们进行回收。）

弱引用(Weak Reference):程度比软引用还要弱一些。这些对象只能生存到下次 GC 之前。当 GC 工作时，无论内存是否足够都会将其回收（即只要进行 GC，就会对他们进行回收。）

虚引用(Phantom Reference):一个对象是否存在虚引用，完全不会对其生存时间构成影响。

关于方法区中需要回收的是一些废弃的常量和无用的类。

1.废弃的常量的回收。这里看引用计数就可以了。没有对象引用该常量就可以放心的回收了。

2.无用的类的回收。什么是无用的类呢?

A.该类所有的实例都已经被回收。也就是 Java 堆中不存在该类的任何实例;

B.加载该类的 ClassLoader 已经被回收;

C.该类对应的 java.lang.Class 对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法。

总而言之:

对于堆中的对象，主要用可达性分析判断一个对象是否还存在引用，如果该对象没有任何引用就应该被回收。而根据我们实际对引用的不同需求，又分成了 4 中引用，每种引用的回收机制也是不同的。

对于方法区中的常量和类，当一个常量没有任何对象引用它，它就可以被回收了。而对于类，如果可以判定它为无用类，就可以被回收了。

七、Java8 的新特性以及使用（2017-12-02-wl）

1. 通过 10 个示例来初步认识 Java8 中的 lambda 表达式（2017-12-02-wl)

我个人对 Java 8 发布非常激动，尤其是 lambda 表达式和流 API。越来越多的了解它们，我能写出更干净的代码。虽然一开始并不是这样。第一次看到用 lambda 表达式写出来的 Java 代码时，我对这种神秘的语法感到非常失望，认

http://www.itheima.com Copyright© 2018 黑马程序员

为它们把 Java 搞得不可读，但我错了。花了一天时间做了一些 lambda 表达式和流 API 示例的练习后，我开心的看到了更清晰的 Java 代码。这有点像学习泛型，第一次见的时候我很讨厌它。我甚至继续使用老版 Java 1.4 来处理集合，直到有一天，朋友跟我介绍了使用泛型的好处（才意识到它的好处）。所以基本立场就是，不要畏惧 lambda 表达式以及方法引用的神秘语法，做几次练习，从集合类中提取、过滤数据之后，你就会喜欢上它。下面让我们开启学习 Java 8 lambda 表达式的学习之旅吧~

本小节中先不说 lambda 表达的含义和繁琐的概念。我们先从最简单的示例来介绍 java8 中的 lambda 表达式

例 1、用 lambda 表达式实现 Runnable

```
// Java 8 之前：
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Before Java8, too much code for too little to do");
    }
}).start();

//Java 8 方式：
new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start();

输出：
too much code, for too little to do
Lambda expression rocks !!
```

这个例子向我们展示了 Java 8 lambda 表达式的语法。你可以使用 lambda 写出如下代码：

```
(params) -> expression
(params) -> statement
(params) -> { statements }
```

例如，如果你的方法不对参数进行修改、重写，只是在控制台打印点东西的话，那么可以这样写：

```
() -> System.out.println("Hello Lambda Expressions");
```

如果你的方法接收两个参数，那么可以写成如下这样：

```
(int even, int odd) -> even + odd
```

顺便提一句，通常都会把 lambda 表达式内部变量的名字起得短一些。这样能使代码更简短，放在同一行。所以，

http://www.itheima.com Copyright© 2018 黑马程序员

在上述代码中，变量名选用 a、b 或者 x、y 会比 even、odd 要好。

例 2、使用 Java 8 lambda 表达式进行事件处理

如果你用过 Swing API 编程，你就会记得怎样写事件监听代码。这又是一个旧版本简单匿名类的经典用例，但现在可以不这样了。你可以用 lambda 表达式写出更好的事件监听代码，如下所示：

```
// Java 8 之前：
JButton show =  new JButton("Show");
show.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
    System.out.println("Event handling without lambda expression is boring");
    }
});

// Java 8 方式：
show.addActionListener((e) -> {
    System.out.println("Light, Camera, Action !! Lambda expressions Rocks");
});
```

Java 开发者经常使用匿名类的另一个地方是为 Collections.sort() 定制 Comparator。在 Java 8 中，你可以用更可读的 lambda 表达式换掉丑陋的匿名类。我把这个留做练习，应该不难，可以按照我在使用 lambda 表达式实现 Runnable 和 ActionListener 的过程中的套路来做。

例 3、使用 Java 8 lambda 表达式进行事件处理　使用 lambda 表达式对列表进行迭代

如果你使过几年 Java，你就知道针对集合类，最常见的操作就是进行迭代，并将业务逻辑应用于各个元素，例如处理订单、交易和事件的列表。由于 Java 是命令式语言，Java 8 之前的所有循环代码都是顺序的，即可以对其元素进行并行化处理。如果你想做并行过滤，就需要自己写代码，这并不是那么容易。通过引入 lambda 表达式和默认方法，将做什么和怎么做的问题分开了，这意味着 Java 集合现在知道怎样做迭代，并可以在 API 层面对集合元素进行并行处理。下面的例子里，我将介绍如何在使用 lambda 或不使用 lambda 表达式的情况下迭代列表。你可以看到列表现在有了一个 forEach() 方法，它可以迭代所有对象，并将你的 lambda 代码应用在其中。

http://www.itheima.com Copyright© 2018 黑马程序员

```
// Java 8 之前：
List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");
for (String feature : features) {
    System.out.println(feature);
}

// Java 8 之后：
List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");
features.forEach(n -> System.out.println(n));

// 使用 Java 8 的方法引用更方便，方法引用由::双冒号操作符标示，
// 看起来像 C++的作用域解析运算符
features.forEach(System.out::println);

输出：
Lambdas
Default Method
Stream API
Date and Time API
```

列表循环的最后一个例子展示了如何在 Java 8 中使用方法引用（method reference）。你可以看到 C++里面的双冒号、范围解析操作符现在在 Java 8 中用来表示方法引用。

例 4、使用 lambda 表达式和函数式接口 Predicate

除了在语言层面支持函数式编程风格，Java 8 也添加了一个包，叫做 java.util.function。它包含了很多类，用来支持 Java 的函数式编程。其中一个便是 Predicate，使用 java.util.function.Predicate 函数式接口以及 lambda 表达式，可以向 API 方法添加逻辑，用更少的代码支持更多的动态行为。下面是 Java 8 Predicate 的例子，展示了过滤集合数据的多种常用方法。Predicate 接口非常适用于做过滤。

```
public static void main(args[]){
    List languages = Arrays.asList("Java", "Scala", "C++", "Haskell", "Lisp");

    System.out.println("Languages which starts with J :");
    filter(languages, (str)->str.startsWith("J"));

    System.out.println("Languages which ends with a ");
    filter(languages, (str)->str.endsWith("a"));
```


 Copyright© 2018 黑马程序员

```
        System.out.println("Print all languages :");
        filter(languages, (str)->true);

        System.out.println("Print no language : ");
        filter(languages, (str)->false);

        System.out.println("Print language whose length greater than 4:");
        filter(languages, (str)->str.length() > 4);
    }

    public static void filter(List names, Predicate condition) {
        for(String name: names)  {
            if(condition.test(name)) {
                System.out.println(name + " ");
            }
        }
    }


    // filter 更好的办法--filter 方法改进
    public static void filter(List names, Predicate condition) {
        names.stream().filter((name) -> (condition.test(name))).forEach((name) -> {
            System.out.println(name + " ");
        });
    }
```

可以看到，Stream API 的过滤方法也接受一个 Predicate，这意味着可以将我们定制的 filter() 方法替换成写在里面的内联代码，这就是 lambda 表达式的魔力。另外，Predicate 接口也允许进行多重条件的测试，下个例子将要讲到。

例 5、如何在 lambda 表达式中加入 Predicate

上个例子说到，java.util.function.Predicate 允许将两个或更多的 Predicate 合成一个。它提供类似于逻辑操作符 AND 和 OR 的方法，名字叫做 and()、or()和 xor()，用于将传入 filter() 方法的条件合并起来。例如，要得到所有以J开始,长度为四个字母的语言,可以定义两个独立的 Predicate 示例分别表示每一个条件,然后用 Predicate.and() 方法将它们合并起来，如下所示:

http://www.itheima.com Copyright© 2018 黑马程序员

```
    // 甚至可以用 and()、or()和 xor()逻辑函数来合并 Predicate，
    // 例如要找到所有以 J 开始，长度为四个字母的名字，你可以合并两个 Predicate 并传入
    Predicate<String> startsWithJ = (n) -> n.startsWith("J");
    Predicate<String> fourLetterLong = (n) -> n.length() == 4;
    names.stream()
        .filter(startsWithJ.and(fourLetterLong))
        .forEach((n) -> System.out.print("nName, which starts with 'J' and four letter long is : " +
n));
```

类似地，也可以使用 or() 和 xor() 方法。本例着重介绍了如下要点：可按需要将 Predicate 作为单独条件然后将其合并起来使用。简而言之，你可以以传统 Java 命令方式使用 Predicate 接口，也可以充分利用 lambda 表达式达到事半功倍的效果。

例 6、Java 8 中使用 lambda 表达式的 Map 和 Reduce 示例

本例介绍最广为人知的函数式编程概念 map。它允许你将对象进行转换。例如在本例中，我们将 costBeforeTax 列表的每个元素转换成为税后的值。我们将 x -> x*x lambda 表达式传到 map() 方法，后者将其应用到流中的每一个元素。然后用 forEach() 将列表元素打印出来。使用流 API 的收集器类，可以得到所有含税的开销。有 toList() 这样的方法将 map 或任何其他操作的结果合并起来。由于收集器在流上做终端操作，因此之后便不能重用流了。你甚至可以用流 API 的 reduce() 方法将所有数字合成一个，下一个例子将会讲到。

```
    // 不使用 lambda 表达式为每个订单加上 12%的税
    List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
    for (Integer cost : costBeforeTax) {
        double price = cost + .12*cost;
        System.out.println(price);
    }

    // 使用 lambda 表达式
    List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
    costBeforeTax.stream().map((cost) -> cost + .12*cost).forEach(System.out::println);
```

在上面例子中，可以看到 map 将集合类（例如列表）元素进行转换的。还有一个 reduce() 函数可以将所有值合并成一个。Map 和 Reduce 操作是函数式编程的核心操作，因为其功能，reduce 又被称为折叠操作。另外，reduce 并不是一个新的操作，你有可能已经在使用它。SQL 中类似 sum()、avg() 或者 count() 的聚集函数，实际上就是

http://www.itheima.com Copyright© 2018 黑马程序员

reduce 操作，因为它们接收多个值并返回一个值。流 API 定义的 reduceh() 函数可以接受 lambda 表达式，并对所有值进行合并。IntStream 这样的类有类似 average()、count()、sum() 的内建方法来做 reduce 操作，也有 mapToLong()、mapToDouble() 方法来做转换。这并不会限制你，你可以用内建方法，也可以自己定义。在这个 Java 8 的 Map Reduce 示例里，我们首先对所有价格应用 12% 的 VAT，然后用 reduce() 方法计算总和。

```
    // 为每个订单加上 12%的税
    // 老方法:
    List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
    double total = 0;
    for (Integer cost : costBeforeTax) {
        double price = cost + .12*cost;
        total = total + price;
    }
    System.out.println("Total : " + total);

    // 新方法:
    List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
    double bill = costBeforeTax.stream().map((cost) -> cost + .12*cost).reduce((sum, cost) -> sum +
cost).get();
    System.out.println("Total : " + bill);
```

例 7、通过过滤创建一个 String 列表

过滤是 Java 开发者在大规模集合上的一个常用操作，而现在使用 lambda 表达式和流 API 过滤大规模数据集合是惊人的简单。流提供了一个 filter() 方法，接受一个 Predicate 对象，即可以传入一个 lambda 表达式作为过滤逻辑。下面的例子是用 lambda 表达式过滤 Java 集合，将帮助理解。

```
    // 创建一个字符串列表，每个字符串长度大于 2
    List costBeforeTax = Arrays.asList("abc","bcd","defg","jk");
    List<String> filtered = strList.stream().filter(x -> x.length()> 2).collect(Collectors.toList());
    System.out.printf("Original List : %s, filtered list : %s %n", strList, filtered);


    输出:
    Original List : [abc, , bcd, , defg, jk], filtered list : [abc, bcd, defg]
```

另外，关于 filter() 方法有个常见误解。在现实生活中，做过滤的时候，通常会丢弃部分，但使用 filter()方法则是

http://www.itheima.com Copyright© 2018 黑马程序员

获得一个新的列表，且其每个元素符合过滤原则。

例 8、对列表的每个元素应用函数

我们通常需要对列表的每个元素使用某个函数，例如逐一乘以某个数、除以某个数或者做其它操作。这些操作都很适合用 map() 方法，可以将转换逻辑以 lambda 表达式的形式放在 map() 方法里，就可以对集合的各个元素进行转换了，如下所示。

```
// 将字符串换成大写并用逗号链接起来
List<String> G7 = Arrays.asList("USA", "Japan", "France", "Germany", "Italy", "U.K.","Canada");
String G7Countries = G7.stream().map(x -> x.toUpperCase()).collect(Collectors.joining(", "));
System.out.println(G7Countries);

输出：
USA, JAPAN, FRANCE, GERMANY, ITALY, U.K., CANADA
```

例 9、复制不同的值，创建一个子列表

本例展示了如何利用流的 distinct() 方法来对集合进行去重

```
// 用所有不同的数字创建一个正方形列表
List<Integer> numbers = Arrays.asList(9, 10, 3, 4, 7, 3, 4);
List<Integer> distinct = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
System.out.printf("Original List : %s,  Square Without duplicates : %s %n", numbers, distinct);

输出：
Original List : [9, 10, 3, 4, 7, 3, 4],  Square Without duplicates : [81, 100, 9, 16, 49]
```

例 10、计算集合元素的最大值、最小值、总和以及平均值

IntStream、LongStream 和 DoubleStream 等流的类中，有个非常有用的方法叫做 summaryStatistics() 。可以返回 IntSummaryStatistics、LongSummaryStatistics 或者 DoubleSummaryStatistic s，描述流中元素的各种摘要数据。在本例中，我们用这个方法来计算列表的最大值和最小值。它也有 getSum() 和 getAverage() 方法来获得列表的所有元素的总和及平均值。

```
//获取数字的个数、最小值、最大值、总和以及平均值
List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29);
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
IntSummaryStatistics stats = primes.stream().mapToInt((x) -> x).summaryStatistics();
System.out.println("Highest prime number in List : " + stats.getMax());
System.out.println("Lowest prime number in List : " + stats.getMin());
System.out.println("Sum of all prime numbers : " + stats.getSum());
System.out.println("Average of all prime numbers : " + stats.getAverage());

输出：
Highest prime number in List : 29
Lowest prime number in List : 2
Sum of all prime numbers : 129
Average of all prime numbers : 12.9
```

Java 8 的 10 个 lambda 表达式，这对于新手来说是个合适的任务量，你可能需要亲自运行示例程序以便掌握。试着修改要求创建自己的例子，达到快速学习的目的。

补充：从例子中我们可以可以看到，以前写的匿名内部类都用了 lambda 表达式代替了。那么，我们简单谈谈“lambda 表达式&匿名内部类”

两者不用：

1. 关键字 this

   (1) 匿名内部类中的 this 代表匿名类

   (2) Lambda 表达式中的 this 代表 lambda 表达式的类

2. 编译方式不同

   (1) 匿名内部类中会编译成一个.class 文件，文件命名方式为：主类+$+(1,2,3.....)

   (2) Java 编译器将 lambda 表达式编译成类的私有方法。使用了 Java 7 的 invokedynamic 字节码指令来动态绑定这个方法

2. Java8 中的 lambda 表达式要点（2017-12-02-wl）

通过面 10 个小示例中学习，我们下面说下 lambda 表达式的 6 个要点

要点 1：lambda 表达式的使用位置

http://www.itheima.com Copyright© 2018 黑马程序员

预定义使用了 @Functional 注释的函数式接口，自带一个抽象函数的方法，或者 SAM（Single Abstract Method 单个抽象方法）类型。这些称为 lambda 表达式的目标类型，可以用作返回类型，或 lambda 目标代码的参数。例如，若一个方法接收 Runnable、Comparable 或者 Callable 接口，都有单个抽象方法，可以传入 lambda 表达式。类似的，如果一个方法接受声明于 java.util.function 包内的接口，例如 Predicate、Function、Consumer 或 Supplier，那么可以向其传 lambda 表达式。

要点 2：lambda 表达式和方法引用

lambda 表达式内可以使用方法引用，仅当该方法不修改 lambda 表达式提供的参数。本例中的 lambda 表达式可以换为方法引用，因为这仅是一个参数相同的简单方法调用。

```
list.forEach(n -> System.out.println(n));
list.forEach(System.out::println);  // 使用方法引用
```

然而，若对参数有任何修改，则不能使用方法引用，而需键入完整地 lambda 表达式，如下所示：

```
list.forEach((String s) -> System.out.println("*" + s + "*"));
```

事实上，可以省略这里的 lambda 参数的类型声明，编译器可以从列表的类属性推测出来。

要点 3：lambda 表达式内部引用资源

lambda 内部可以使用静态、非静态和局部变量，这称为 lambda 内的变量捕获。

要点 4：lambda 表达式也成闭包

Lambda 表达式在 Java 中又称为闭包或匿名函数，所以如果有同事把它叫闭包的时候，不用惊讶。

要点 5：lambda 表达式的编译方式

Lambda 方法在编译器内部被翻译成私有方法，并派发 invokedynamic 字节码指令来进行调用。可以使用 JDK 中的 javap 工具来反编译 class 文件。使用 javap -p 或 javap -c -v 命令来看一看 lambda 表达式生成的字节码。大致应该长这样：

http://www.itheima.com Copyright© 2018 黑马程序员

```
    private static java.lang.Object lambda$0(java.lang.String);
```

要点 6：lambda 表达式的限制

lambda 表达式有个限制，那就是只能引用 final 或 final 局部变量，这就是说不能在 lambda 内部修改定义在域外的变量。

```
    List<Integer> primes = Arrays.asList(new Integer[]{2, 3,5,7});
    int factor = 2;
    primes.forEach(element -> { factor++; });


    Error:
    Compile time error : "local variables referenced from a lambda expression must be final or effectively final"
```

另外，只是访问它而不作修改是可以的，如下所示：

```
    List<Integer> primes = Arrays.asList(new Integer[]{2, 3,5,7});
    int factor = 2;
    primes.forEach(element -> { System.out.println(factor*element); });

    输出：
    4
    6
    10
    14
```

因此，它看起来更像不可变闭包，类似于 Python。

3. Java8 中的 Optional 类的解析（2017-12-02-wl）

身为一名 Java 程序员，大家可能都有这样的经历：调用一个方法得到了返回值却不能直接将返回值作为参数去调用别的方法。我们首先要判断这个返回值是否为 null，只有在非空的前提下才能将其作为其他方法的参数。这正是一些类似 Guava 的外部 API 试图解决的问题。一些 JVM 编程语言比如 Scala、Ceylon 等已经将对在核心 API 中解决了这个问题。在我的前一篇文章中，介绍了 Scala 是如何解决了这个问题。

新版本的 Java，比如 Java 8 引入了一个新的 Optional 类。Optional 类的 Javadoc 描述如下：

 Copyright© 2018 黑马程序员

这是一个可以为 null 的容器对象。如果值存在则 isPresent()方法会返回 true，调用 get()方法会返回该对象。

下面会逐个探讨 Optional 类包含的方法，并通过一两个示例展示如何使用。

方法 1：Optional.of()

作用：为非 null 的值创建一个 Optional。

说明：of 方法通过工厂方法创建 Optional 类。需要注意的是，创建对象时传入的参数不能为 null。如果传入参数为 null，则抛出 NullPointerException 。

```
//调用工厂方法创建 Optional 实例
Optional<String> name = Optional.of("Sanaulla");
//传入参数为 null，抛出 NullPointerException.
Optional<String> someNull = Optional.of(null);
```

方法 2：Optional.ofNullable()

作用：为指定的值创建一个 Optional，如果指定的值为 null，则返回一个空的 Optional。

说明：ofNullable 与 of 方法相似，唯一的区别是可以接受参数为 null 的情况。

```
//下面创建了一个不包含任何值的 Optional 实例
//例如，值为'null'
Optional empty = Optional.ofNullable(null);
```

方法 3：Optional.isPresent()

作用：判断预期值是否存在

说明：如果值存在返回 true，否则返回 false。

```
//isPresent 方法用来检查 Optional 实例中是否包含值
Optional<String> name = Optional.of("Sanaulla");
if (name.isPresent()) {
  //在 Optional 实例内调用 get()返回已存在的值
  System.out.println(name.get());//输出 Sanaulla
}
```

方法 4：Optional.get()

http://www.itheima.com Copyright© 2018 黑马程序员

作用：如果 Optional 有值则将其返回，否则抛出 NoSuchElementException。

说明：上面的示例中，get 方法用来得到 Optional 实例中的值。下面我们看一个抛出 NoSuchElementException 的例子

```
//执行下面的代码会输出：No value present
try {
  Optional empty = Optional.ofNullable(null);
  //在空的 Optional 实例上调用 get()，抛出 NoSuchElementException
  System.out.println(empty.get());
} catch (NoSuchElementException ex) {
  System.out.println(ex.getMessage());
}
```

方法 5：Optional.ifPresent()

作用：如果 Optional 实例有值则为其调用 consumer，否则不做处理

说明：要理解 ifPresent 方法，首先需要了解 Consumer 类。简答地说，Consumer 类包含一个抽象方法。该抽象方法对传入的值进行处理，但没有返回值。Java8 支持不用接口直接通过 lambda 表达式传入参数，如果 Optional 实例有值，调用 ifPresent()可以接受接口段或 lambda 表达式

```
//ifPresent 方法接受 lambda 表达式作为参数。
//lambda 表达式对 Optional 的值调用 consumer 进行处理。
Optional<String> name = Optional.of("Sanaulla");
name.ifPresent((value) -> {
  System.out.println("The length of the value is: " + value.length());
});
```

方法 7：Optional.orElse()

作用：如果有值则将其返回，否则返回指定的其它值。

说明：如果 Optional 实例有值则将其返回，否则返回 orElse 方法传入的参数。示例如下：

```
Optional<String> name = Optional.of("Sanaulla");
Optional<String> someNull = Optional.of(null);
//如果值不为 null，orElse 方法返回 Optional 实例的值。
//如果为 null，返回传入的消息。
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
//输出：There is no value present!
System.out.println(empty.orElse("There is no value present!"));
//输出：Sanaulla
System.out.println(name.orElse("There is some value!"));
```

方法 8：Optional.orElseGet()

作用：如果有值则将其返回，否则返回指定的其它值。

说明：orElseGet 与 orElse 方法类似，区别在于得到的默认值。orElse 方法将传入的字符串作为默认值，orElseGet 方法可以接受 Supplier 接口的实现用来生成默认值

```
Optional<String> name = Optional.of("Sanaulla");
Optional<String> someNull = Optional.of(null);

//orElseGet 与 orElse 方法类似，区别在于 orElse 传入的是默认值，
//orElseGet 可以接受一个 lambda 表达式生成默认值。
//输出：Default Value
System.out.println(empty.orElseGet(() -> "Default Value"));
//输出：Sanaulla
System.out.println(name.orElseGet(() -> "Default Value"));
```

方法 9：Optional.orElseThrow()

作用：如果有值则将其返回，否则抛出 supplier 接口创建的异常。

说明：在 orElseGet 方法中，我们传入一个 Supplier 接口。然而，在 orElseThrow 中我们可以传入一个 lambda 表达式或方法，如果值不存在来抛出异常

```
try {
   Optional<String> empty= Optional.of(null);

  //orElseThrow 与 orElse 方法类似。与返回默认值不同，
  //orElseThrow 会抛出 lambda 表达式或方法生成的异常
  empty.orElseThrow(ValueAbsentException::new);
} catch (Throwable ex) {
  //输出：No value present in the Optional instance
  System.out.println(ex.getMessage());
}

ValueAbsentException 定义如下：
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
class ValueAbsentException extends Throwable {

  public ValueAbsentException() {
    super();
  }

  public ValueAbsentException(String msg) {
    super(msg);
  }

  @Override
  public String getMessage() {
    return "No value present in the Optional instance";
  }
}
```

方法 10：Optional.map()

作用：如果有值，则对其执行调用 mapping 函数得到返回值。如果返回值不为 null，则创建包含 mapping 返回值的 Optional 作为 map 方法返回值，否则返回空 Optional。

说明：map 方法用来对 Optional 实例的值执行一系列操作。通过一组实现了 Function 接口的 lambda 表达式传入操作。

```
Optional<String> name = Optional.of("Sanaulla");
//map 方法执行传入的 lambda 表达式参数对 Optional 实例的值进行修改。
//为 lambda 表达式的返回值创建新的 Optional 实例作为 map 方法的返回值。
Optional<String> upperName = name.map((value) -> value.toUpperCase());
System.out.println(upperName.orElse("No value found"));
```

方法 11：Optional.flatMap()

作用：如果有值，为其执行 mapping 函数返回 Optional 类型返回值，否则返回空 Optional。flatMap 与 map（Funtion）方法类似，区别在于 flatMap 中的 mapper 返回值必须是 Optional。调用结束时，flatMap 不会对结果用 Optional 封装。

说明：flatMap 方法与 map 方法类似，区别在于 mapping 函数的返回值不同。map 方法的 mapping 函数返回

http://www.itheima.com Copyright© 2018 黑马程序员

值可以是任何类型 T，而 flatMap 方法的 mapping 函数必须是 Optional。

```
Optional<String> name = Optional.of("Sanaulla");
//flatMap 与 map（Function）非常类似，区别在于传入方法的 lambda 表达式的返回类型。
//map 方法中的 lambda 表达式返回值可以是任意类型，在 map 函数返回之前会包装为 Optional。
//但 flatMap 方法中的 lambda 表达式返回值必须是 Optionl 实例。
upperName = name.flatMap((value) -> Optional.of(value.toUpperCase()));
System.out.println(upperName.orElse("No value found"));//输出 SANAULLA
```

方法 12：Optional.filter()

作用：如果有值并且满足断言条件返回包含该值的 Optional，否则返回空 Optional。

说明：filter 个方法通过传入限定条件对 Optional 实例的值进行过滤。这里可以传入一个 lambda 表达式。对于 filter 函数我们应该传入实现了 Predicate 接口的 lambda 表达式。

```
Optional<String> name = Optional.of("Sanaulla");
//filter 方法检查给定的 Option 值是否满足某些条件。
//如果满足则返回同一个 Option 实例，否则返回空 Optional。
Optional<String> longName = name.filter((value) -> value.length() > 6);
System.out.println(longName.orElse("The name is less than 6 characters"));//输出 Sanaulla

//另一个例子是 Optional 值不满足 filter 指定的条件。
Optional<String> anotherName = Optional.of("Sana");
Optional<String> shortName = anotherName.filter((value) -> value.length() > 6);
//输出：name 长度不足 6 字符
System.out.println(shortName.orElse("The name is less than 6 characters"));
```

总结：Optional 方法

以上，我们介绍了 Optional 类的各个方法。下面通过一个完整的示例对用法集中展示。

```
1    public class OptionalDemo {
2
3      public static void main(String[] args) {
4        //创建 Optional 实例，也可以通过方法返回值得到。
5        Optional<String> name = Optional.of("Sanaulla");
6
7        //创建没有值的 Optional 实例，例如值为'null'
8        Optional empty = Optional.ofNullable(null);
9
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
10        //isPresent 方法用来检查 Optional 实例是否有值。
11        if (name.isPresent()) {
12          //调用 get()返回 Optional 值。
13          System.out.println(name.get());
14        }
15
16        try {
17          //在 Optional 实例上调用 get()抛出 NoSuchElementException。
18          System.out.println(empty.get());
19        } catch (NoSuchElementException ex) {
20          System.out.println(ex.getMessage());
21        }
22
23        //ifPresent 方法接受 lambda 表达式参数。
24        //如果 Optional 值不为空，lambda 表达式会处理并在其上执行操作。
25        name.ifPresent((value) -> {
26          System.out.println("The length of the value is: " + value.length());
27        });
28
29        //如果有值 orElse 方法会返回 Optional 实例，否则返回传入的错误信息。
30        System.out.println(empty.orElse("There is no value present!"));
31        System.out.println(name.orElse("There is some value!"));
32
33        //orElseGet 与 orElse 类似，区别在于传入的默认值。
34        //orElseGet 接受 lambda 表达式生成默认值。
35        System.out.println(empty.orElseGet(() -> "Default Value"));
36        System.out.println(name.orElseGet(() -> "Default Value"));
37
38        try {
39          //orElseThrow 与 orElse 方法类似，区别在于返回值。
40          //orElseThrow 抛出由传入的 lambda 表达式/方法生成异常。
41          empty.orElseThrow(ValueAbsentException::new);
42        } catch (Throwable ex) {
43          System.out.println(ex.getMessage());
44        }
45
46        //map 方法通过传入的 lambda 表达式修改 Optonal 实例默认值。
47        //lambda 表达式返回值会包装为 Optional 实例。
48        Optional<String> upperName = name.map((value) -> value.toUpperCase());
49        System.out.println(upperName.orElse("No value found"));
50
51        //flatMap 与 map（Funtion）非常相似，区别在于 lambda 表达式的返回值。
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
52          //map 方法的 lambda 表达式返回值可以是任何类型，但是返回值会包装成 Optional 实例。
53          //但是 flatMap 方法的 lambda 返回值总是 Optional 类型。
54          upperName = name.flatMap((value) -> Optional.of(value.toUpperCase()));
55          System.out.println(upperName.orElse("No value found"));
56
57          //filter 方法检查 Optiona 值是否满足给定条件。
58          //如果满足返回 Optional 实例值，否则返回空 Optional。
59          Optional<String> longName = name.filter((value) -> value.length() > 6);
60          System.out.println(longName.orElse("The name is less than 6 characters"));
61
62          //另一个示例，Optional 值不满足给定条件。
63          Optional<String> anotherName = Optional.of("Sana");
64          Optional<String> shortName = anotherName.filter((value) -> value.length() > 6);
65          System.out.println(shortName.orElse("The name is less than 6 characters"));
66
67      }
68
69  }
```

八、在开发中遇到过内存溢出么？原因有哪些？解决方法有哪些？（2017-11-23-gxb）

引起内存溢出的原因有很多种，常见的有以下几种：

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。)

http://www.itheima.com Copyright© 2018 黑马程序员

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。

重点排查以下几点：

1.检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

2.检查代码中是否有死循环或递归调用。

3.检查是否有大循环重复产生新对象实体。

4.检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中 数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

5.检查 List、MAP 等集合对象是否有使用完后，未清除的问题。List、MAP 等集合对象会始终存有对对象的引用，使得这些对象不能被 GC 回收。

第四步，使用内存查看工具动态查看内存使用情况。

第四章 JavaWEB 基础

一、JDBC 技术

1. 说下原生 jdbc 操作数据库流程？（2017-11-25-wzz）

第一步：Class.forName()加载数据库连接驱动；

第二步：DriverManager.getConnection()获取数据连接对象；

http://www.itheima.com Copyright© 2018 黑马程序员

第三步：根据 SQL 获取 sql 会话对象，有 2 种方式 Statement、PreparedStatement ;

第四步：执行 SQL 处理结果集，执行 SQL 前如果有参数值就设置参数值 setXXX();

第五步：关闭结果集、关闭会话、关闭连接。

详细代码请看（封装）：http://blog.csdn.net/qq_29542611/article/details/52426006

2. 什么要使用 PreparedStatement? （2017-11-25-wzz）

1、 PreparedStatement 接口继承 Statement， PreparedStatement 实例包含已编译的 SQL 语句，所以其执行速度要快于 Statement 对象。

2、作为 Statement 的子类， PreparedStatement 继承了 Statement 的所有功能。三种方法 execute、 executeQuery 和 executeUpdate 已被更改以使之不再需要参数

3、在 JDBC 应用中,在任何时候都不要使用 Statement，原因如下:

一、代码的可读性和可维护性.Statement 需要不断地拼接，而 PreparedStatement 不会。

二、PreparedStatement 尽最大可能提高性能.DB 有缓存机制，相同的预编译语句再次被调用不会再次需要编译。

三、最重要的一点是极大地提高了安全性.Statement 容易被 SQL 注入，而 PreparedStatementc 传入的内

http://www.itheima.com Copyright© 2018 黑马程序员

容不会和 sql 语句发生任何匹配关系。

3. 关系数据库中连接池的机制是什么？（2017-12-6-lyq）

前提：为数据库连接建立一个缓冲池。

1：从连接池获取或创建可用连接

2：使用完毕之后，把连接返回给连接池

3：在系统关闭前，断开所有连接并释放连接占用的系统资源

4：能够处理无效连接，限制连接池中的连接总数不低于或者不超过某个限定值。

其中有几个概念需要大家理解：

最小连接数是连接池一直保持的数据连接。如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费掉。

最大连接数是连接池能申请的最大连接数。如果数据连接请求超过此数，后面的数据连接请求将被加入到等待队列中，这会影响之后的数据库操作。

如果最小连接数与最大连接数相差太大，那么，最先的连接请求将会获利，之后超过最小连接数量的连接请求等价于建立一个新的数据库连接。不过，这些大于最小连接数的数据库连接在使用完不会马上被释放，它将被放到连接池中等待重复使用或是空闲超时后被释放。

上面的解释，可以这样理解：数据库池连接数量一直保持一个不少于最小连接数的数量，当数量不够时，数据库会创建一些连接，直到一个最大连接数，之后连接数据库就会等待。

http://www.itheima.com Copyright© 2018 黑马程序员

三、Http 协议

1. http 的长连接和短连接（2017-11-14-lyq）

HTTP 协议有 HTTP/1.0 版本和 HTTP/1.1 版本。HTTP1.1 默认保持长连接（HTTP persistent connection，也翻译为持久连接），数据传输完成了保持 TCP 连接不断开（不发 RST 包、不四次握手），等待在同域名下继续用这个通道传输数据；相反的就是短连接。

在 HTTP/1.0 中，默认使用的是短连接。也就是说，浏览器和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接。从 HTTP/1.1 起，默认使用的是长连接，用以保持连接特性。

2. HTTP/1.1 与 HTTP/1.0 的区别（2017-11-21-wzy）

参考原文：http://blog.csdn.net/forgotaboutgirl/article/details/6936982

1 可扩展性

a) HTTP/1.1 在消息中增加版本号，用于兼容性判断。

b) HTTP/1.1 增加了 OPTIONS 方法，它允许客户端获取一个服务器支持的方法列表。

c) 为了与未来的协议规范兼容，HTTP/1.1 在请求消息中包含了 Upgrade 头域，通过该头域，客户端可以让服

http://www.itheima.com Copyright© 2018 黑马程序员

务器知道它能够支持的其它备用通信协议，服务器可以据此进行协议切换，使用备用协议与客户端进行通信。

2 缓存

在 HTTP/1.0 中，使用 Expire 头域来判断资源的 fresh 或 stale，并使用条件请求（conditional request）来判断资源是否仍有效。HTTP/1.1 在 1.0 的基础上加入了一些 cache 的新特性，当缓存对象的 Age 超过 Expire 时变为 stale 对象，cache 不需要直接抛弃 stale 对象，而是与源服务器进行重新激活（revalidation）。

3 带宽优化

HTTP/1.0 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了。例如，客户端只需要显示一个文档的部分内容，又比如下载大文件时需要支持断点续传功能，而不是在发生断连后不得不重新下载完整的包。

HTTP/1.1 中在请求消息中引入了 range 头域，它允许只请求资源的某个部分。在响应消息中 Content-Range 头域声明了返回的这部分对象的偏移值和长度。如果服务器相应地返回了对象所请求范围的内容，则响应码为 206（Partial Content），它可以防止 Cache 将响应误以为是完整的一个对象。

另外一种情况是请求消息中如果包含比较大的实体内容，但不确定服务器是否能够接收该请求（如是否有权限），此时若贸然发出带实体的请求，如果被拒绝也会浪费带宽。

HTTP/1.1 加入了一个新的状态码 100（Continue）。客户端事先发送一个只带头域的请求，如果服务器因为权限拒绝了请求，就回送响应码 401（Unauthorized）；如果服务器接收此请求就回送响应码 100，客户端就可以继续发送带实体的完整请求了。注意，HTTP/1.0 的客户端不支持 100 响应码。但可以让客户端在请求消息中加入 Expect 头域，并将它的值设置为 100-continue。

节省带宽资源的一个非常有效的做法就是压缩要传送的数据。Content-Encoding 是对消息进行端到端（end-to-end）的编码，它可能是资源在服务器上保存的固有格式（如 jpeg 图片格式）；在请求消息中加入 Accept-Encoding 头域，它可以告诉服务器客户端能够解码的编码方式。

http://www.itheima.com Copyright© 2018 黑马程序员

4 长连接

HTTP/1.0 规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个 TCP 连接，服务器完成请求处理后立即断开 TCP 连接，服务器不跟踪每个客户也不记录过去的请求。此外，由于大多数网页的流量都比较小，一次 TCP 连接很少能通过 slow-start 区，不利于提高带宽利用率。

HTTP 1.1 支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个 TCP 连接上可以传送多个 HTTP 请求和响应，减少了建立和关闭连接的消耗和延迟。例如：一个包含有许多图像的网页文件的多个请求和应答可以在一个连接中传输，但每个单独的网页文件的请求和应答仍然需要使用各自的连接。

HTTP 1.1 还允许客户端不用等待上一次请求结果返回，就可以发出下一次请求，但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样也显著地减少了整个下载过程所需要的时间。

5 消息传递

HTTP 消息中可以包含任意长度的实体，通常它们使用 Content-Length 来给出消息结束标志。但是，对于很多动态产生的响应，只能通过缓冲完整的消息来判断消息的大小，但这样做会加大延迟。如果不使用长连接，还可以通过连接关闭的信号来判定一个消息的结束。

HTTP/1.1 中引入了 Chunkedtransfer-coding 来解决上面这个问题，发送方将消息分割成若干个任意大小的数据块，每个数据块在发送时都会附上块的长度，最后用一个零长度的块作为消息结束的标志。这种方法允许发送方只缓冲消息的一个片段，避免缓冲整个消息带来的过载。

在 HTTP/1.0 中，有一个 Content-MD5 的头域，要计算这个头域需要发送方缓冲完整个消息后才能进行。而 HTTP/1.1 中，采用 chunked 分块传递的消息在最后一个块（零长度）结束之后会再传递一个拖尾（trailer），它包含一个或多个头域，这些头域是发送方在传递完所有块之后再计算出值的。发送方会在消息中包含一个 Trailer 头域告诉接收方这个拖尾的存在。

http://www.itheima.com Copyright© 2018 黑马程序员

6 Host 头域

在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址,因此,请求消息中的URL并没有传递主机名(hostname)。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个 IP 地址。

HTTP1.1 的请求消息和响应消息都应支持 Host 头域，且请求消息中如果没有 Host 头域会报告一个错误（400 Bad Request）。此外，服务器应该接受以绝对路径标记的资源请求。

7 错误提示

HTTP/1.0 中只定义了 16 个状态响应码，对错误或警告的提示不够具体。HTTP/1.1 引入了一个 Warning 头域，增加对错误或警告信息的描述。

此外，在 HTTP/1.1 中新增了 24 个状态响应码，如 409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。

3. http 常见的状态码有哪些？（2017-11-23-wzz）

200 OK //客户端请求成功

301 Moved Permanently（永久移除），请求的 URL 已移走。Response 中应该包含一个 Location URL，说明资源现在所处的位置

302 found 重定向

400 Bad Request //客户端请求有语法错误，不能被服务器所理解

401 Unauthorized //请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用

403 Forbidden //服务器收到请求，但是拒绝提供服务

404 Not Found //请求资源不存在，eg：输入了错误的 URL

http://www.itheima.com Copyright© 2018 黑马程序员

500 Internal Server Error //服务器发生不可预期的错误

503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

4. GET 和 POST 的区别？（2017-11-23-wzz）

从表面现像上面看 GET 和 POST 的区别：

1. GET 请求的数据会附在 URL 之后（就是把数据放置在 HTTP 协议头中），以?分割 URL 和传输数据，参数之间以&相连，如：login.action?name=zhagnsan&password=123456。POST 把提交的数据则放置在是 HTTP 包的包体中。

2. GET 方式提交的数据最多只能是 1024 字节，理论上 POST 没有限制，可传较大量的数据。其实这样说是错误的，不准确的：

“GET 方式提交的数据最多只能是 1024 字节"，因为 GET 是通过 URL 提交数据，那么 GET 可提交的数据量就跟 URL 的长度有直接关系了。而实际上，URL 不存在参数上限的问题，HTTP 协议规范没有对 URL 长度进行限制。这个限制是特定的浏览器及服务器对它的限制。IE 对 URL 长度的限制是 2083 字节(2K+35)。对于其他浏览器，如 Netscape、FireFox 等，理论上没有长度限制，其限制取决于操作系统的支持。

3.POST 的安全性要比 GET 的安全性高。注意：这里所说的安全性和上面 GET 提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而这里安全的含义是真正的 Security 的含义，比如：通过 GET 提交数据，用户名和密码将明文出现在 URL 上，因为(1)登录页面有可能被浏览器缓存，(2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了，除此之外，使用 GET 提交数据还可能会造成 Cross-site request forgery 攻击。

Get 是向服务器发索取数据的一种请求，而 Post 是向服务器提交数据的一种请求，在 FORM（表单）中，Method 默认为"GET"，实质上，GET 和 POST 只是发送机制不同，并不是一个取一个发！

参考原文：https://www.cnblogs.com/hyddd/archive/2009/03/31/1426026.html

http://www.itheima.com Copyright© 2018 黑马程序员

5. http 中重定向和请求转发的区别？（2017-11-23-wzz）

本质区别：转发是服务器行为，重定向是客户端行为。

重定向特点：两次请求，浏览器地址发生变化，可以访问自己 web 之外的资源，传输的数据会丢失。

请求转发特点：一次强求，浏览器地址不变，访问的是自己本身的 web 资源，传输的数据不会丢失。

四、Cookie 和 Session

1. Cookie 和 Session 的区别（2017-11-15-lyq）

Cookie 是 web 服务器发送给浏览器的一块信息，浏览器会在本地一个文件中给每个 web 服务器存储 cookie。以后浏览器再给特定的 web 服务器发送请求时，同时会发送所有为该服务器存储的 cookie。

Session 是存储在 web 服务器端的一块信息。session 对象存储特定用户会话所需的属性及配置信息。当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。

Cookie 和 session 的不同点：

1、无论客户端做怎样的设置，session 都能够正常工作。当客户端禁用 cookie 时将无法使用 cookie。

http://www.itheima.com Copyright© 2018 黑马程序员

2、在存储的数据量方面：session 能够存储任意的 java 对象，cookie 只能存储 String 类型的对象。

2. session 共享怎么做的（分布式如何实现 session 共享）？

参考原文：http://blog.csdn.net/sxiaobei/article/details/57086489

问题描述：一个用户在登录成功以后会把用户信息存储在 session 当中，这时 session 所在服务器为 server1，那么用户在 session 失效之前如果再次使用 app，那么可能会被路由到 server2，这时问题来了，server 没有该用户的 session，所以需要用户重新登录，这时的用户体验会非常不好，所以我们想如何实现多台 server 之间共享 session，让用户状态得以保存。

1、服务器实现的 session 复制或 session 共享，这类型的共享 session 是和服务器紧密相关的，比如 webSphere 或 JBOSS 在搭建集群时候可以配置实现 session 复制或 session 共享，但是这种方式有一个致命的缺点，就是不好扩展和移植，比如我们更换服务器，那么就要修改服务器配置。

2、利用成熟的技术做 session 复制，比如 12306 使用的 gemfire，比如常见的内存数据库如 redis 或 memorycache，这类方案虽然比较普适，但是严重依赖于第三方，这样当第三方服务器出现问题的时候，那么将是应用的灾难。

3、将 session 维护在客户端，很容易想到就是利用 cookie，但是客户端存在风险，数据不安全，而且可以存放的

http://www.itheima.com Copyright© 2018 黑马程序员

数据量比较小，所以将 session 维护在客户端还要对 session 中的信息加密。

我们实现的方案可以说是第二种方案和第三种方案的合体，可以利用 gemfire 实现 session 复制共享，还可以将 session 维护在 redis 中实现 session 共享，同时可以将 session 维护在客户端的 cookie 中，但是前提是数据要加密。这三种方式可以迅速切换，而不影响应用正常执行。我们在实践中，首选 gemfire 或者 redis 作为 session 共享的载体，一旦 session 不稳定出现问题的时候，可以紧急切换 cookie 维护 session 作为备用，不影响应用提供服务。

这里主要讲解 redis 和 cookie 方案，gemfire 比较复杂大家可以自行查看 gemfire 工作原理。利用 redis 做 session 共享，首先需要与业务逻辑代码解耦，不然 session 共享将没有意义，其次支持动态切换到客户端 cookie 模式。redis 的方案是，重写服务器中的 HttpSession 和 HttpServletRequest，首先实现 HttpSession 接口，重写 session 的所有方法，将 session 以 hash 值的方式存在 redis 中，一个 session 的 key 就是 sessionID，setAtrribute 重写之后就是更新 redis 中的数据，getAttribute 重写之后就是获取 redis 中的数据，等等需要将 HttpSession 的接口一一实现。

实现了 HttpSesson，那么我们先将该 session 类叫做 MySession（当然实践中不是这么命名的），当 MySession 出现之后问题才开始，怎么能在不影响业务逻辑代码的情况下，还能让原本的 request.getSession（）获取到的是 MySession，而不是服务器原生的 session。这里，我决定重写服务器的 HttpServletRequet，这里先称为 MyRequest，但是这可不是单纯的重写，我需要在原生的 request 基础上重写，于是我决定在 filter 中，实现 request 的偷梁换柱，我的思路是这样的，MyRequest 的构建器，必须以 request 作为参数，于是我在 filter 中将服务器原生的 request（也有可能是框架封装过的 request），当做参数 new 出来一个 MyRequest，并且 MyRequest 也实现了 HttpServletRequest 接口，其实就是对原生 request 的一个增强，这里主要重写了几个 request 的方法，但是最重要的是重写了 request.getSession()，写到这里大家应该都明白为什么重写这个方法了吧，当然是为了获取 MySession，于是这样就在 filter 中，偷偷的将原生的 request 换成 MyRequest 了，然后再将替换过的 request 传入 chan.doFilter()，这样 filter 时候的代码都使用的是 MyRequest 了，同时对业务代码是透明的，业务代码获取 session 的方法仍然是

http://www.itheima.com Copyright© 2018 黑马程序员

request.getSession ()，但其实获取到的已经是 MySession 了，这样对 session 的操作已经变成了对 redis 的操作。这样实现的好处有两个，第一开发人员不需要对 session 共享做任何关注，session 共享对用户是透明的；第二，filter 是可配置的，通过 filter 的方式可以将 session 共享做成一项可插拔的功能，没有任何侵入性。

这个时候已经实现了一套可插拔的 session 共享的框架了，但是我们想到如果 redis 服务出了问题，这时我们该怎么办呢，于是我们延续 redis 的想法，想到可以将 session 维护在客户端内（加密的 cookie），当然实现方法还是一样的，我们重写 HttpSession 接口，实现其所有方法，比如 setAttribute 就是写入 cookie，getAttribute 就是读取 cookie，我们可以将重写的 session 称作 MySession2，这时怎么让开发人员透明的获取到 MySession2 呢，实现方法还是在 filter 内偷梁换柱，在 MyRequest 加一个判断，读取 sessionType 配置，如果 sessionType 是 redis 的，那么 getSession 的时候获取到的是 MySession，如果 sessionType 是 coolie 的，那么 getSession 的时候获取到的是 MySession2，以此类推，用同样的方法就可以获取到 MySession 3,4,5,6 等等。

这样两种方式都有了，那么我们怎实现两种 session 共享方式的快速切换呢，刚刚我提到一个 sessionType，这是用来决定获取到 session 的类型的，只要变换 sessionType 就能实现两种 session 共享方式的切换，但是 sessionType 必须对所有的服务器都是一致的，如果不一致那将会出现比较严重的问题，我们目前是将 sessionType 维护在环境变量里，如果要切换 sessionType 就要重启每一台服务器，完成 session 共享的转换，但是当服务器太多的时候将是一种灾难。而且重启服务意味着服务的中断，所以这样的方式只适合服务器规模比较小，而且用户量比较少的情况，当服务器太多的时候，务必需要一种协调技术，能够让服务器能够及时获取切换的通知。基于这样的原因，我们选用 zookeeper 作为配置平台，每一台服务器都会订阅 zookeeper 上的配置，当我们切换 sessionType 之后，所有服务器都会订阅到修改之后的配置，那么切换就会立即生效，当然可能会有短暂的时间延迟，但这是可以接受的。

http://www.itheima.com Copyright© 2018 黑马程序员

3. 在单点登录中，如果 cookie 被禁用了怎么办？（2017-11-23-gxb）

单点登录的原理是后端生成一个 session ID，然后设置到 cookie，后面的所有请求浏览器都会带上 cookie，然后服务端从 cookie 里获取 session ID，再查询到用户信息。所以，保持登录的关键不是 cookie，而是通过 cookie 保存和传输的 session ID，其本质是能获取用户信息的数据。除了 cookie，还通常使用 HTTP 请求头来传输。但是这个请求头浏览器不会像 cookie 一样自动携带，需要手工处理。

五、jsp 技术

1. 什么是jsp,什么是Servlet？jsp和Servlet有什么区别？（2017-11-23-wzz）

jsp 本质上就是一个 Servlet，它是 Servlet 的一种特殊形式（由 SUN 公司推出），每个 jsp 页面都是一个 servlet 实例。

Servlet 是由 Java 提供用于开发 web 服务器应用程序的一个组件，运行在服务端，由 servlet 容器管理，用来生成动态内容。一个 servlet 实例是实现了特殊接口 Servlet 的 Java 类，所有自定义的 servlet 均必须实现 Servlet 接口。

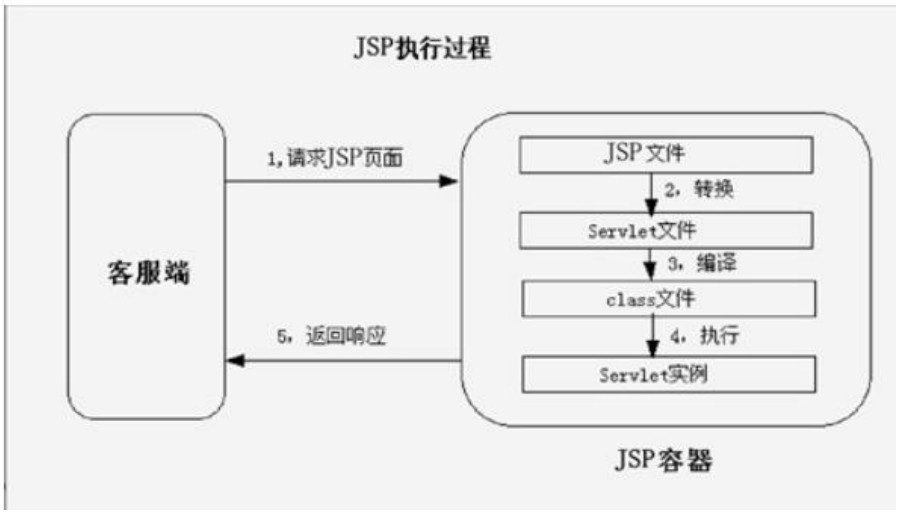
区别：

jsp 是 html 页面中内嵌的 Java 代码，侧重页面显示；

Servlet 是 html 代码和 Java 代码分离，侧重逻辑控制，mvc 设计思想中 jsp 位于视图层，servlet 位于控制层

Jsp 运行机制：如下图

http://www.itheima.com Copyright© 2018 黑马程序员



JVM 只能识别 Java 类，并不能识别 jsp 代码！web 容器收到以.jsp 为扩展名的 url 请求时，会将访问请求交给 tomcat 中 jsp 引擎处理，每个 jsp 页面第一次被访问时，jsp 引擎将 jsp 代码解释为一个 servlet 源程序，接着编译 servlet 源程序生成.class 文件，再有 web 容器 servlet 引擎去装载执行 servlet 程序，实现页面交互。

2. jsp 有哪些域对象和内置对象及他们的作用？（2017-11-25-wzz）

四大域对象：

（1）pageContext page 域-指当前页面，在当前 jsp 页面有效，跳到其它页面失效

（2）request request 域-指一次请求范围内有效，从 http 请求到服务器处理结束，返回响应的整个过程。在这个过程中使用 forward（请求转发）方式跳转多个 jsp，在这些页面里你都可以使用这个变量

（3）session session 域-指当前会话有效范围，浏览器从打开到关闭过程中，转发、重定向均可以使用

（4）application context 域-指只能在同一个 web 中使用，服务器未关闭或者重启，数据就有效

九大内置对象：

	生命周期	作用域	使用情况
Request	一次请求	只在 Jsp 页面内有效	用于接受通过 HTTP 协议传送到服务器的数据（包括头信息、系统信息、请求方式以及请求参数等）。
Reponse	一次响应	只在 Jsp 页面内有效	表示服务器端对客户端的回应。主要用于设置头信息、跳转、Cookie 等

http://www.itheima.com Copyright© 2018 黑马程序员

Session	从存入数据开始，默认闲置 30 分钟后失效	会话内有效	用于存储特定的用户会话所需的信息
Out	http://www.cnblogs.com/leirenyuan/p/6016063.html		用于在 Web 浏览器内输出信息，并且管理应用服务器上的输出缓冲区
PageContext	详细了解：http://www.cnblogs.com/leirenyuan/p/6016063.html		用于存取其他隐含对象，如 request、reponse、session、application 等对象。（实际上，pageContext 对象提供了对 JSP 页面所有的对象及命名空间的访问。
Page	http://www.cnblogs.com/leirenyuan/p/6016063.html		page 对象代表 JSP 本身（对应 this），只有在 JSP 页面内才是合法的
Exception	http://www.cnblogs.com/leirenyuan/p/6016063.html		显示异常信息，必须在 page 指令中设定< %@ page isErrorPage="true" %>才能使用，在一般的 JSP 页面中使用该对象将无法编译 JSP 文件
Application	服务器启动发送第一个请求时就产生了 Application 对象，直到服务器关闭。		用于存储和访问来自任何页面的变量所有的用户分享一个 Application 对象
Config	http://www.cnblogs.com/leirenyuan/p/6016063.html		取得服务器的配置信息

六、XML 技术

1. 什么是 xml，使用 xml 的优缺点，xml 的解析器有哪几种，分别有什么区别？(2017-11-25-wzz)

xml 是一种可扩展性标记语言，支持自定义标签（使用前必须预定义）使用 DTD 和 XML Schema 标准化 XML 结构。

具体了解 xml 详见：http://www.importnew.com/10839.html

http://www.itheima.com Copyright© 2018 黑马程序员

优点：用于配置文件，格式统一，符合标准；用于在互不兼容的系统间交互数据，共享数据方便；

缺点：xml 文件格式复杂，数据传输占流量，服务端和客户端解析 xml 文件占用大量资源且不易维护

Xml 常用解析器有 2 种，分别是：DOM 和 SAX;

主要区别在于它们解析 xml 文档的方式不同。使用 DOM 解析，xml 文档以 DOM

树形结构加载入内存，而 SAX 采用的是事件模型，

详细区别：https://wenku.baidu.com/view/fc3fb5610b1c59eef8c7b410.html

第五章 JavaWEB 高级

一、Filter 和 Listener

未完待续.....

二、AJAX

1. 谈谈你对 ajax 的认识？ （2017-11-23-wzz）

Ajax 是一种创建交互式网页应用的的网页开发技术；Asynchronous JavaScript and XML”的缩写。

Ajax 的优势：

通过异步模式，提升了用户体验。

优化了浏览器和服务器之间的传输，减少不必要的数据往返，减少了带宽占用。

Ajax 引擎在客户端运行，承担了一部分本来由服务器承担的工作，从而减少了大用户量下的服务器负载。

Ajax 的最大特点：

可以实现局部刷新，在不更新整个页面的前提下维护数据，提升用户体验度。

注意：

ajax 在实际项目开发中使用率非常高（牢固掌握），针对 ajax 的详细描述：http://www.jb51.net/article/93258.htm

http://www.itheima.com Copyright© 2018 黑马程序员

2. jsonp 原理（2017-11-21-gxb）

JavaScript 是一种在 Web 开发中经常使用的前端动态脚本技术。在 JavaScript 中，有一个很重要的安全性限制，被称为“Same-Origin Policy”（同源策略）。这一策略对于 JavaScript 代码能够访问的页面内容做了很重要的限制，即 JavaScript 只能访问与包含它的文档在同一域下的内容。

JavaScript 这个安全策略在进行多 iframe 或多窗口编程、以及 Ajax 编程时显得尤为重要。根据这个策略，在 baidu.com 下的页面中包含的 JavaScript 代码，不能访问在 google.com 域名下的页面内容；甚至不同的子域名之间的页面也不能通过 JavaScript 代码互相访问。对于 Ajax 的影响在于，通过 XMLHttpRequest 实现的 Ajax 请求，不能向不同的域提交请求，例如，在 abc.example.com 下的页面，不能向 def.example.com 提交 Ajax 请求，等等。

然而，当进行一些比较深入的前端编程的时候，不可避免地需要进行跨域操作，这时候“同源策略”就显得过于苛刻。JSONP 跨域 GET 请求是一个常用的解决方案，下面我们来看一下 JSONP 跨域是如何实现的，并且探讨下 JSONP 跨域的原理。

jsonp 的最基本的原理是：动态添加一个<script>标签，使用 script 标签的 src 属性没有跨域的限制的特点

http://www.itheima.com Copyright© 2018 黑马程序员

实现跨域。首先在客户端注册一个 callback，然后把 callback 的名字传给服务器。此时，服务器先生成 json 数据。 然后以 javascript 语法的方式，生成一个 function，function 名字就是传递上来的参数 jsonp。最后将 json 数据直接以入参的方式，放置到 function 中，这样就生成了一段 js 语法的文档，返回给客户端。

客户端浏览器，解析 script 标签，并执行返回的 javascript 文档，此时数据作为参数，传入到了客户端预先定义好的 callback 函数里。

参考资料：http://www.nowamagic.net/librarys/veda/detail/224

三、Linux

1. 说一下常用的 Linux 命令

列出文件列表：ls 【参数 -a -l】

创建目录和移除目录：mkdir rmdir

用于显示文件后几行内容：tail

打包：tar -xvf

打包并压缩：tar -zcvf

http://www.itheima.com Copyright© 2018 黑马程序员

查找字符串：grep

显示当前所在目录：pwd

创建空文件：touch

编辑器：vim　vi

列出文件列表：ls 【参数 -a -l】

创建目录和移除目录：mkdir　rmdir

用于显示文件后几行内容：tail

打包：tar -xvf

打包并压缩：tar -zcvf

查找字符串：grep

显示当前所在目录：pwd

创建空文件：touch

编辑器：vim　vi

2. Linux 中如何查看日志？（2017-11-21-gxb）

动态打印日志信息：tail –f 日志文件

参考资料：https://www.cnblogs.com/zdz8207/p/linux-log-tail-cat-tac.html

http://www.itheima.com Copyright© 2018 黑马程序员

3. Linux 怎么关闭进程（2017-11-21-gxb）

通常用 ps 查看进程 PID ，用 kill 命令终止进程。

ps 命令用于查看当前正在运行的进程。

grep 是搜索

例如： ps -ef | grep java

表示查看所有进程里 CMD 是 java 的进程信息。

ps -aux | grep java

-aux 显示所有状态

kill 命令用于终止进程。

例如： kill -9 [PID]

-9 表示强迫进程立即停止。

http://www.itheima.com Copyright© 2018 黑马程序员

四、常见的前端框架有哪些

1. EasyUI（2017-11-23-lyq）

EasyUI 是一种基于 jQuery 的用户界面插件集合。easyui 为创建现代化，互动，JavaScript 应用程序，提供必要的功能。使用 easyui 你不需要写很多代码，你只需要通过编写一些简单 HTML 标记，就可以定义用户界面。

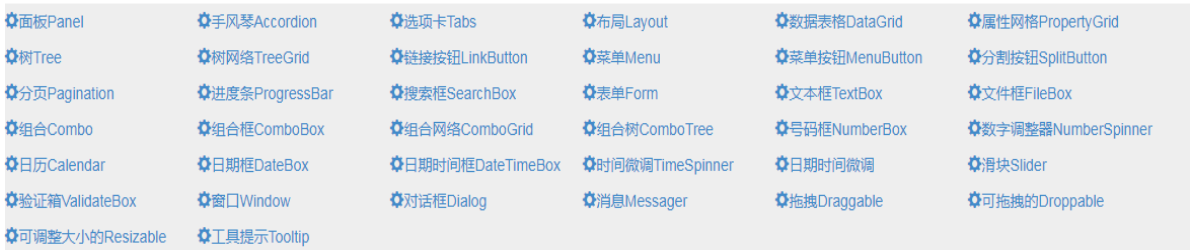
优势：开源免费，页面也还说的过去。

easyUI 入门：

页面引入必要的 js 和 css 样式文件，文件引入顺序为：

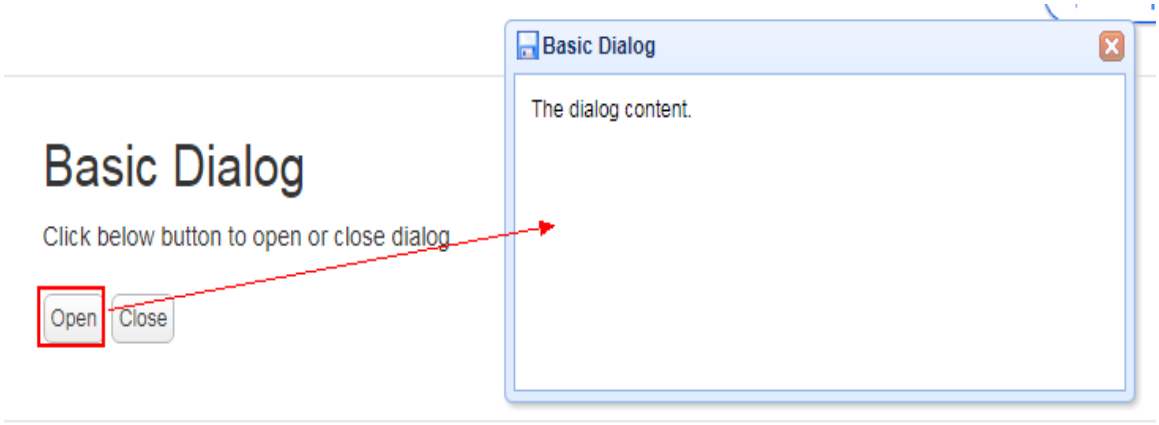
```
1. <!-- 引入 JQuery -->
2. <script type="text/javascript" src="../jquery-easyui-1.4.1/jquery.min.js"></script>
3.    <!-- 引入 EasyUI -->
4. <script type="text/javascript" src="../jquery-easyui-1.4.1/jquery.easyui.min.js"></script>
5.   <!-- 引入 EasyUI 的中文国际化 js，让 EasyUI 支持中文 -->
6. <script type="text/javascript" src="../jquery-easyui-1.4.1/locale/easyui-lang-zh_CN.js"></script>
7.    <!-- 引入 EasyUI 的样式文件-->
8. <link rel="stylesheet" href="../jquery-easyui-1.4.1/themes/default/easyui.css" type="text/css"/>
9.    <!-- 引入 EasyUI 的图标样式文件-->
10. <link rel="stylesheet" href="../jquery-easyui-1.4.1/themes/icon.css" type="text/css"/>
```

然后在页面写 easyUI 代码就行，easyUI 提供了很多样式：



示例如下：

http://www.itheima.com Copyright© 2018 黑马程序员



实现代码如下：

```
1. <!DOCTYPE html>
2. <html>
3. <head>
4.  <meta charset="UTF-8">
5.  <title>Basic Dialog - jQuery EasyUI Demo</title>
6.  <link rel="stylesheet" type="text/css" href="../../themes/default/easyui.css">
7.  <link rel="stylesheet" type="text/css" href="../../themes/icon.css">
8.  <link rel="stylesheet" type="text/css" href="../demo.css">
9.  <script type="text/javascript" src="../../jquery.min.js"></script>
10. <script type="text/javascript" src="../../jquery.easyui.min.js"></script>
11. </head>
12. <body>
13. <h2>Basic Dialog</h2>
14. <p>Click below button to open or close dialog.</p>
15. <div style="margin:20px 0;">
16.     <a href="javascript:void(0)" class="easyui-linkbutton"
onclick="$('#dlg').dialog('open')">Open</a>
17.     <a href="javascript:void(0)" class="easyui-linkbutton"
onclick="$('#dlg').dialog('close')">Close</a>
18. </div>
19. <div id="dlg" class="easyui-dialog" title="Basic Dialog" data-options="iconCls:'icon-save'"
style="width:400px;height:200px;padding:10px">
20.     The dialog content.
21. </div>
22. </body>
23. </html>
24.
```


http://www.itheima.com Copyright© 2018 黑马程序员

2. MiniUI（2017-11-23-lyq）

基于 jquery 的框架，开发的界面功能都很丰富。jQuery MiniUI - 快速开发 WebUI。它能缩短开发时间，减少代码量，使开发者更专注于业务和服务端，轻松实现界面开发，带来绝佳的用户体验。使用 MiniUI，开发者可以快速创建 Ajax 无刷新、 B/S 快速录入数据、 CRUD、 Master-Detail、菜单工具栏、弹出面板、布局导航、数据验证、分页表格、树、树形表格等典型 WEB 应用系统界面。**缺点：收费，没有源码，基于这个开发如果想对功能做扩展就需要找他们的团队进行升级！**

提供以下几大类控件：

表格控件

树形控件

布局控件：标题面板、弹出面板、折叠分割器、布局器、表单布局器等

导航控件：分页导航器、导航菜单、选项卡、菜单、工具栏等。

表单控件：多选输入框、弹出选择框、文本输入框、数字输入框、日期选择框、下拉选择框、下拉树形选择框、下拉表格选择框、文件上传控件、多选框、列表框、多选框组、单选框组、按钮等

富文本编辑器

图表控件：柱状图、饼图、线形图、双轴图等。

技术亮点：

快速开发：使用 Html 配置界面，减少 80%界面代码量。

易学易用：简单的 API 设计，可以独立、组合使用控件。

性能优化：内置数据懒加载、低内存开销、快速界面布局等机制。

丰富控件：包含表格、树、数据验证、布局导航等超过 50 个控件。

超强表格：提供锁定列、多表头、分页排序、行过滤、数据汇总、单元格编辑、详细行、Excel 导出等功

http://www.itheima.com Copyright© 2018 黑马程序员

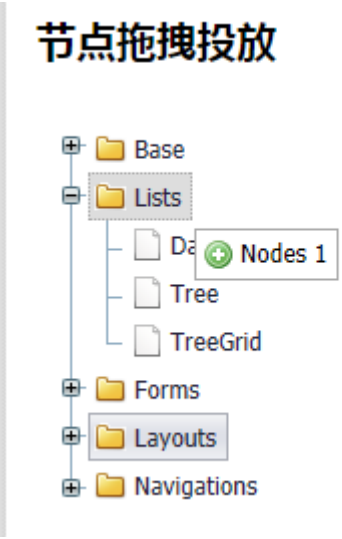
能。

第三方兼容：与 ExtJS、jQuery、YUI、Dojo 等任意第三方控件无缝集成。

浏览器兼容：支持 IE6+、FireFox、Chrome 等。

跨平台支持：支持 Java、.NET、PHP 等。

示例如下：



实现代码如下：

```
<ul id="tree1" class="mini-tree" url="../data/tree.txt" style="width:200px;padding:5px;"
    showTreeIcon="true" textField="text" idField="id"
    allowDrag="true" allowDrop="true"
    >
</ul>
```

1. jQueryUI（2017-11-23-lyq）

jQuery UI 是一套 jQuery 的页面 UI 插件，包含很多种常用的页面空间，例如 Tabs（如本站首页右上角部分）、拉帘效果（本站首页左上角）、对话框、拖放效果、日期选择、颜色选择、数据排序、窗体大小调整等等非常多的内容。

技术亮点：

简单易用：继承 jQuery 简易使用特性，提供高度抽象接口，短期改善网站易用性。

http://www.itheima.com Copyright© 2018 黑马程序员

开源免费：采用 MIT & GPL 双协议授权，轻松满足自由产品至企业产品各种授权需求。

广泛兼容：兼容各主流桌面浏览器。包括 IE 6+、Firefox 2+、Safari 3+、Opera 9+、Chrome 1+。

轻便快捷：组件间相对独立，可按需加载，避免浪费带宽拖慢网页打开速度。

标准先进：支持 WAI-ARIA，通过标准 XHTML 代码提供渐进增强，保证低端环境可访问性。

美观多变：提供近 20 种预设主题，并可自定义多达 60 项可配置样式规则，提供 24 种背景纹理选择。

度娘上搜 jQueryUI 的 api，其用法与 easyUI、MiniUI 都大同小异，此处将不再举例。

2. Vue.js（2017-11-23-lyq）

参考原文：https://cn.vuejs.org/v2/guide/

Vue.js (读音 /vju:/，类似于 view) 是一套构建用户界面的渐进式框架。与其他重量级框架不同的是，Vue 采用自底向上增量开发的设计。Vue 的核心库只关注视图层，它不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与单文件组件和 Vue 生态系统支持的库结合使用时，Vue 也完全能够为复杂的单页应用程序提供驱动。

Vue.js 起步：

引入相应文件：

```
1.  <script src="https://unpkg.com/vue"></script>
```

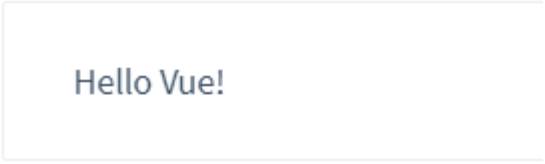

http://www.itheima.com Copyright© 2018 黑马程序员

声明式渲染：

Vue.js 的核心是一个允许采用简洁的模板语法来声明式的将数据渲染进 DOM 的系统：

```
<!-- html 文件中 -->
<div id="app">
  {{ message }}
</div>

<!-- js 文件中 -->
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

通过浏览器查看效果图为：



创建 vue 实例：

每个 Vue 应用都是通过 Vue 函数创建一个新的 Vue 实例开始的，当创建一个 Vue 实例时，你可以传入一个选项对象。可以使用这些选项来创建你想要的行为。

```
<!-- js 文件中 -->
var vm = new Vue({
  // 选项
})
```

实例生命周期：

每个 Vue 实例在被创建之前都要经过一系列的初始化过程。例如需要设置数据监听、编译模板、挂载实例到 DOM、在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做生命周期钩子的函数，给予用户机会在一些特定的场景下添加他们自己的代码。比如 created 钩子可以用来在一个实例被创建之后执行代码：

```
<!-- js 文件中 -->
new Vue({
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
3.    data: {
4.      a: 1
5.    },
6.    created: function () {
7.      // `this` 指向 vm 实例
8.      console.log('a is: ' + this.a)
9.    }
10. })
11. // => "a is: 1"
```

3. AngularJS （2017-11-23-lyq）

参考原文：http://www.angularjs.net.cn/api/

AngularJS 是 google 开发者设计的一个前端开发框架，它是由是由 JavaScript 编写的一个 JS 框架。通常它是用来在静态网页构建动态应用不足而设计的。

AngularJS 特点如下：

1、 数据绑定： AngularJS 是数据双向绑定。

2、 MVVM (Model-View-ViewModel) 模式： Model 简单数据对象，View 视图 (如 HTML,JSP 等)，ViewModel 是用来提供数据和方法，和 View 进行交互。这种设计模式使得代码解耦合。

http://www.itheima.com Copyright© 2018 黑马程序员

3、依赖注入：AngularJS 支持注入方式把需要的对象，方法等注入到指定的对象中。

4、 指令： AngularJS 内部自带各种常用指令，同时也支持开发者自定义指令。

5、HTML 模板和扩展 HTML： AngularJS 可以定义与 HTML 兼容的自定义模板。

AngularJS 的 Api：

AngularJS 提供了很多功能丰富的组件，处理核心的 ng 组件外，还扩展了很多常用的功能组件，如 ngRoute(路由)，ngAnimate(动画)，ngTouch(移动端操作)等，只需要引入相应的头文件，并依赖注入你的工作模块，则可使用。

ng (core module)：AngularJS 的默认模块，包含 AngularJS 的所有核心组件。

指令(directive)	这是指令的核心集合，你可以在你的模板代码中使用它们来构建一个 AngularJS 应用。 一些例子包括:ngClick,ngInclude, ngRepeat,等等
服务(service)	这是服务的核心集合，依赖注入后可在你的应用中使用。 一些例子包括: $compile,$http, $location,等等
过滤器(filter)	在组件模块中的核心过滤器是用来转换模板数据。 一些例子包括: filter, date, currency,lowercase, uppercase 等等
全局函数 (function)	核心的全局函数作为 angularjs 对象。这些核心功能在您的应用程序的原生的 JavaScript 操作有用。 一些例子包括: angular.copy(), angular.equals(), angular.element()等等

ngRoute：AgularJS 的路由模块，你能使用 ngRoute 结合"#"定义你的地址访问。引入 angular-route.js 文件，然后在你当前的工作模块依赖注入 ngRoute 模块。

服务(service)	下面这些服务用作 AngularJS 的路由管理。 $routeParams - 解析返回路由中带有的参数 $route - 用于构建各个路由的 url、view、controller 这三者的关系 $routeProvider - 提供路由配置

http://www.itheima.com Copyright© 2018 黑马程序员

指令(Directive)	指令 ngView 提供不同路由模板插入的视图层

ngAnimate：AngularJS 的动画模块，使用 ngAnimate 各种核心指令能为你的应用程序提供动画效果。动画可使用 css 或者 JavaScript 回调函数。引入 angular-animate.js 文件，然后在你当前的工作模块依赖注入 ngAnimate 模块。

服务(service)	使用$animate 来在你的指令代码中触发动画操作。
CSS-based animations	按照 nganimate 的 CSS 命名结构参考 CSS 转换/关键帧动画在 AngularJS。 一旦定义，动画可以通过引用 CSS 在 HTML 模板代码触发。
JS-based animations	使用 module.animation() 来注册一个 JavaScript 动画。 一旦注册，动画可以通过引用 CSS 在 HTML 模板代码触发。

ngAria：帮助制作 AngularJS 自定义组件的新模块。引入 angular-aria.js 文件，然后在你当前的工作模块依赖注入 ngAria 模块。

服务(service)	这是服务的核心集合，依赖注入后可在你的应用中使用。 一些例子包括: ngClick, ngInclude, ngRepeat 等等

ngResource：AngularJS 的动画模块，使用 ngAnimate 各种核心指令能为你的应用程序提供动画效果。动画可使用 css 或者 JavaScript 回调函数。引入 angular-resource.js 文件，然后在你当前的工作模块依赖注入 ngResource 模块。

服务(service)	这是服务的核心集合，依赖注入后可在你的应用中使用。 一些例子包括: ngClick, ngInclude, ngRepeat 等等

ngCookies：ngCookies 模块提供了一个方便的包用于读取和写入浏览器的 cookies。引入 angular-cookies.js 文件，然后在你当前的工作模块依赖注入 ngCookies 模块。

http://www.itheima.com Copyright© 2018 黑马程序员

服务(service)	这是服务的核心集合，依赖注入后可在你的应用中使用。 一些例子包括: ngClick, ngInclude, ngRepeat 等等

ngTouch：AngularJS 的动画模块，使用 ngAnimate 各种核心指令能为你的应用程序提供动画效果。动画可使用 css 或者 JavaScript 回调函数。引入 angular-touch.js 文件，然后在你当前的工作模块依赖注入 ngTouch 模块。

服务(service)	这是服务的核心集合，依赖注入后可在你的应用中使用。 一些例子包括: ngClick, ngInclude, ngRepeat 等等

ngSanitize：使用 ngSanitize 可安全地解析和在你的应用程序中操作 HTML 数据。

引入 angular-sanitize.js 文件，然后在你当前的工作模块依赖注入 ngSanitize 模块。

服务(service)	这是服务的核心集合，依赖注入后可在你的应用中使用。 一些例子包括: ngClick, ngInclude, ngRepeat 等等

---

ngMock：AngularJS 的动画模块，使用 ngAnimate 各种核心指令能为你的应用程序提供动画效果。动画可使用 css 或者 JavaScript 回调函数。引入 angular-animate.js 文件，然后在你当前的工作模块依赖注入 ngMock 模块。

服务(service)	这是服务的核心集合，依赖注入后可在你的应用中使用。 一些例子包括: ngClick, ngInclude, ngRepeat 等等

http://www.itheima.com Copyright© 2018 黑马程序员

第六章 数据库

一、Mysql

1. SQL 的 select 语句完整的执行顺序（2017-11-15-lyq）

SQL Select 语句完整的执行顺序：

1、from 子句组装来自不同数据源的数据；

2、where 子句基于指定的条件对记录行进行筛选；

3、group by 子句将数据划分为多个分组；

4、使用聚集函数进行计算；

5、使用 having 子句筛选分组；

6、计算所有的表达式；

7、select 的字段；

8、使用 order by 对结果集进行排序。

SQL 语言不同于其他编程语言的最明显特征是处理代码的顺序。在大多数据库语言中，代码按编码顺序被处理。但在 SQL 语句中，第一个被处理的子句式 FROM，而不是第一出现的 SELECT。SQL 查询处理的步骤序号：

(1) FROM <left_table>

(2) <join_type> JOIN <right_table>

(3) ON <join_condition>

(4) WHERE <where_condition>

(5) GROUP BY <group_by_list>

http://www.itheima.com Copyright© 2018 黑马程序员

(6) WITH {CUBE | ROLLUP}

(7) HAVING <having_condition>

(8) SELECT

(9) DISTINCT

(9) ORDER BY <order_by_list>

(10) <TOP_specification> <select_list>

以上每个步骤都会产生一个虚拟表，该虚拟表被用作下一个步骤的输入。这些虚拟表对调用者(客户端应用程序或者外部查询)不可用。只有最后一步生成的表才会会给调用者。如果没有在查询中指定某一个子句，将跳过相应的步骤。

逻辑查询处理阶段简介：

1、 FROM：对 FROM 子句中的前两个表执行笛卡尔积(交叉联接)，生成虚拟表 VT1。

2、 ON：对 VT1 应用 ON 筛选器，只有那些使为真才被插入到 TV2。

3、 OUTER (JOIN):如果指定了 OUTER JOIN(相对于 CROSS JOIN 或 INNER JOIN)，保留表中未找到匹配的行将作为外部行添加到 VT2，生成 TV3。如果 FROM 子句包含两个以上的表，则对上一个联接生成的结果表和下一个表重复执行步骤 1 到步骤 3，直到处理完所有的表位置。

4、 WHERE：对 TV3 应用 WHERE 筛选器，只有使为 true 的行才插入 TV4。

5、 GROUP BY：按 GROUP BY 子句中的列列表对 TV4 中的行进行分组，生成 TV5。

6、 CUTE|ROLLUP：把超组插入 VT5，生成 VT6。

7、 HAVING：对 VT6 应用 HAVING 筛选器，只有使为 true 的组插入到 VT7。

8、 SELECT：处理 SELECT 列表，产生 VT8。

9、 DISTINCT：将重复的行从 VT8 中删除，产品 VT9。

http://www.itheima.com Copyright© 2018 黑马程序员

10、 ORDER BY：将 VT9 中的行按 ORDER BY 子句中的列列表顺序，生成一个游标(VC10)。

11、 TOP：从 VC10 的开始处选择指定数量或比例的行，生成表 TV11，并返回给调用者。

where 子句中的条件书写顺序

2. SQL 之聚合函数（2017-11-15-lyq）

聚合函数是对一组值进行计算并返回单一的值的函数，它经常与 select 语句中的 group by 子句一同使用。

a. avg()：返回的是指定组中的平均值，空值被忽略。

b. count()：返回的是指定组中的项目个数。

c. max()：返回指定数据中的最大值。

d. min()：返回指定数据中的最小值。

e. sum()：返回指定数据的和，只能用于数字列，空值忽略。

f. group by()：对数据进行分组，对执行完 group by 之后的组进行聚合函数的运算，计算每一组的值。最后用 having 去掉不符合条件的组，having 子句中的每一个元素必须出现在 select 列表中(只针对于 mysql)。

3. SQL 之连接查询（左连接和右连接的区别）（2017-11-15-lyq）

外连接：

左连接（左外连接）：以左表作为基准进行查询，左表数据会全部显示出来，右表如果和左表匹配的数据则显示相应字段的数据，如果不匹配则显示为 null。

右连接（右外连接）：以右表作为基准进行查询，右表数据会全部显示出来，左表如果和右表匹配的数据则显示相应字段的数据，如果不匹配则显示为 null。

全连接：先以左表进行左外连接，再以右表进行右外连接。

http://www.itheima.com Copyright© 2018 黑马程序员

内连接：

显示表之间有连接匹配的所有行。

4. SQL 之 sql 注入（2017-11-15-lyq）

通过在 Web 表单中输入（恶意）SQL 语句得到一个存在安全漏洞的网站上的数据库，而不是按照设计者意图去执行 SQL 语句。举例：当执行的 sql 为 select * from user where username = “admin” or “a” = “a” 时，sql 语句恒成立，参数 admin 毫无意义。

防止 sql 注入的方式：

1. 预编译语句：如，select * from user where username = ? ，sql 语句语义不会发生改变，sql 语句中变量用? 表示，即使传递参数时为“admin or ‘a’ = ‘a’ ”，也会把这整体当做一个字符创去查询。

2. Mybatis 框架中的 mapper 方式中的 # 也能很大程度的防止 sql 注入（$无法防止 sql 注入）。

5. Mysql 性能优化（2017-11-15-lyq）

1、当只要一行数据时使用 limit 1

查询时如果已知会得到一条数据，这种情况下加上 limit 1 会增加性能。因为 mysql 数据库引擎会在找到一条结果停止搜索，而不是继续查询下一条是否符合标准直到所有记录查询完毕。

2、选择正确的数据库引擎

Mysql 中有两个引擎 MyISAM 和 InnoDB，每个引擎有利有弊。

MyISAM 适用于一些大量查询的应用，但对于有大量写功能的应用不是很好。甚至你只需要 update 一个字段整个表都会被锁起来。而别的进程就算是读操作也不行要等到当前 update 操作完

http://www.itheima.com Copyright© 2018 黑马程序员

成之后才能继续进行。另外，MyISAM 对于 select count(*)这类操作是超级快的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用会比 MyISAM 还慢，但是支持“行锁”，所以在写操作比较多的时候会比较优秀。并且，它支持很多的高级应用，例如：事物。

3. 用 not exists 代替 not in

   Not exists 用到了连接能够发挥已经建立好的索引的作用，not in 不能使用索引。Not in 是最慢的方式要同每条记录比较，在数据量比较大的操作红不建议使用这种方式。

4. 对操作符的优化，尽量不采用不利于索引的操作符

   如：in　　not in　　is null　　is not null　　<>　等

   某个字段总要拿来搜索，为其建立索引：

   Mysql 中可以利用 alter table 语句来为表中的字段添加索引，语法为：alter table 表明 add index (字段名);

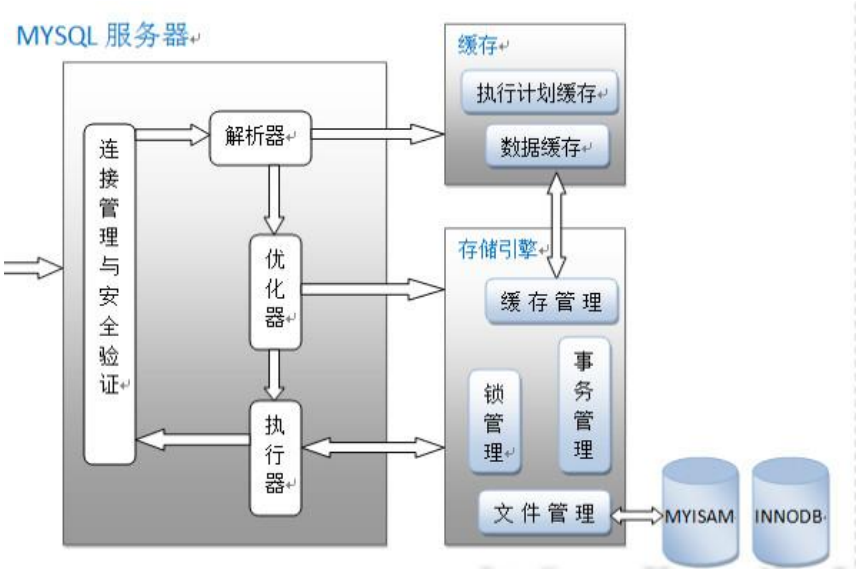
6. 必看 sql 面试题（学生表_课程表_成绩表_教师表）（2017-11-25-wzz）

给大家推荐一篇非常好的博客，该博客中收集了最常见的 Mysql 常见面试题和笔试题。

博客链接：http://www.cnblogs.com/qixuejia/p/3637735.html

http://www.itheima.com Copyright© 2018 黑马程序员

7. Mysql 数据库架构图（2017-11-25-wzz）



MyISAM 和 InnoDB 是最常见的两种存储引擎，特点如下。

MyISAM 存储引擎

MyISAM 是 MySQL 官方提供默认的存储引擎，其特点是不支持事务、表锁和全文索引，对于一些 OLAP（联机分析处理）系统，操作速度快。

每个 MyISAM 在磁盘上存储成三个文件。文件名都和表名相同，扩展名分别是.frm（存储表定义）、.MYD (MYData，存储数据)、.MYI (MYIndex，存储索引)。这里特别要注意的是 MyISAM 不缓存数据文件，只缓存索引文件。

InnoDB 存储引擎

InnoDB 存储引擎支持事务，主要面向 OLTP（联机事务处理过程）方面的应用，其特点是行锁设置、支持外键，并支持类似于 Oracle 的非锁定读，即默认情况下读不产生锁。InnoDB 将数据放在一个逻辑表空间中（类似 Oracle）。InnoDB 通过多版本并发控制来获得高并发性，实现了 ANSI 标准的 4 种隔离级别，默认为 Repeatable，使用一种被称为 next-key locking 的策略避免幻读。

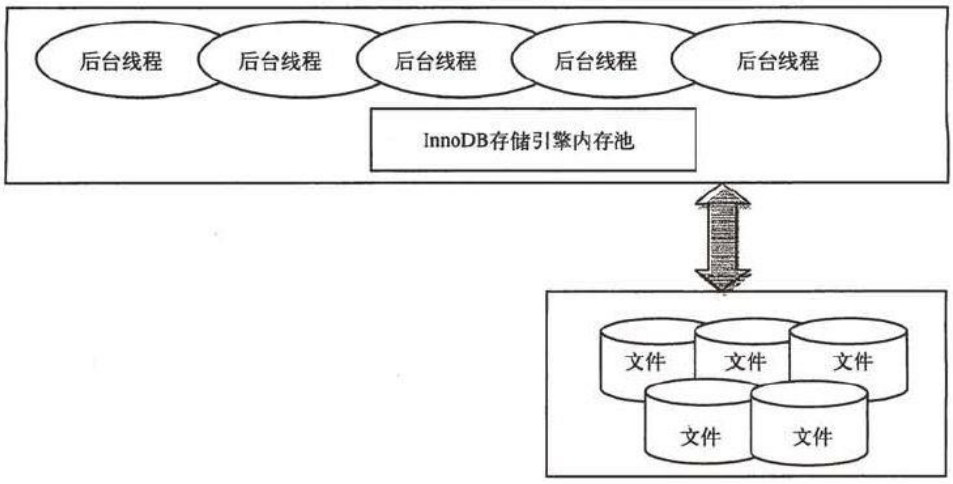
对于表中数据的存储，InnoDB 采用类似 Oracle 索引组织表 Clustered 的方式进行存储。

InnoDB 存储引擎提供了具有提交、回滚和崩溃恢复能力的事务安全。但是对比 Myisam 的存储引擎，InnoDB 写的

http://www.itheima.com Copyright© 2018 黑马程序员

处理效率差一些并且会占用更多的磁盘空间以保留数据和索引。

InnoDB 体系架构



8. Mysql 架构器中各个模块都是什么？（2017-11-25-wzz）

（1）、连接管理与安全验证是什么?

每个客户端都会建立一个与服务器连接的线程，服务器会有一个线程池来管理这些 连接；如果客户端需要连接到 MYSQL 数据库还需要进行验证，包括用户名、密码、主机信息等。

（2）、解析器是什么?

解析器的作用主要是分析查询语句，最终生成解析树；首先解析器会对查询语句的语法进行分析，分析语法是否有问题。还有解析器会查询缓存，如果在缓存中有对应的语句，就返回查询结果不进行接下来的优化执行操作。前提是缓存中的数据没有被修改，当然如果被修改了也会被清出缓存。

（3）、优化器怎么用?

优化器的作用主要是对查询语句进行优化操作，包括选择合适的索引，数据的读取方式，包括获取查询的开销信息，统计信息等，这也是为什么图中会有优化器指向存储引擎的箭头。之前在别的文章没有看到优化器跟存储引擎之

http://www.itheima.com Copyright© 2018 黑马程序员

间的关系，在这里我个人的理解是因为优化器需要通过存储引擎获取查询的大致数据和统计信息。

（4）、执行器是什么?

执行器包括执行查询语句，返回查询结果，生成执行计划包括与存储引擎的一些处理操作。

9. Mysql 存储引擎有哪些？（2017-11-25-wzz）

（1）、InnoDB 存储引擎

InnoDB 是事务型数据库的首选引擎，支持事务安全表（ACID），支持行锁定和外键，InnoDB 是默认的 MySQL 引擎。

（2）、MyISAM 存储引擎

MyISAM 基于 ISAM 存储引擎，并对其进行扩展。它是在 Web、数据仓储和其他应用环境下最常使用的存储引擎之一。MyISAM 拥有较高的插入、查询速度，但不支持事物。

（3）、MEMORY 存储引擎

MEMORY 存储引擎将表中的数据存储到内存中，未查询和引用其他表数据提供快速访问。

（4）、NDB 存储引擎

DB 存储引擎是一个集群存储引擎，类似于 Oracle 的 RAC，但它是 Share Nothing 的架构，因此能提供更高级别的高可用性和可扩展性。NDB 的特点是数据全部放在内存中，因此通过主键查找非常快。

关于 NDB，有一个问题需要注意，它的连接(join)操作是在 MySQL 数据库层完成，不是在存储引擎层完成，这意味着，复杂的 join 操作需要巨大的网络开销，查询速度会很慢。

（5）、Memory (Heap) 存储引擎

Memory 存储引擎（之前称为 Heap）将表中数据存放在内存中，如果数据库重启或崩溃，数据丢失，因此它非常适合存储临时数据。

http://www.itheima.com Copyright© 2018 黑马程序员

（6）、Archive 存储引擎

正如其名称所示，Archive 非常适合存储归档数据，如日志信息。它只支持 INSERT 和 SELECT 操作，其设计的主要目的是提供高速的插入和压缩功能。

（7）、Federated 存储引擎

Federated 存储引擎不存放数据，它至少指向一台远程 MySQL 数据库服务器上的表，非常类似于 Oracle 的透明网关。

（8）、Maria 存储引擎

Maria 存储引擎是新开发的引擎，其设计目标是用来取代原有的 MyISAM 存储引擎，从而成为 MySQL 默认的存储引擎。

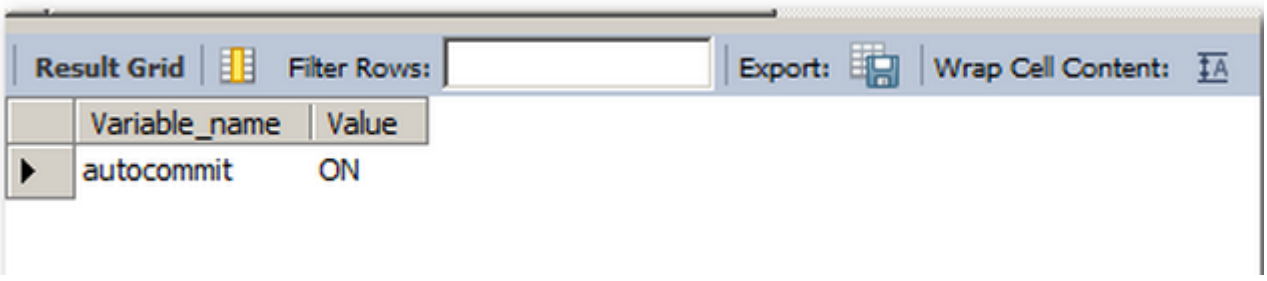
上述引擎中，InnoDB 是事务安全的存储引擎，设计上借鉴了很多 Oracle 的架构思想，一般而言，在 OLTP 应用中，InnoDB 应该作为核心应用表的首先存储引擎。InnoDB 是由第三方的 Innobase Oy 公司开发，现已被 Oracle 收购，创始人是 Heikki Tuuri，芬兰赫尔辛基人，和著名的 Linux 创始人 Linus 是校友。

10. MySQL 事务介绍（2017-11-25-wzz）

MySQL 和其它的数据库产品有一个很大的不同就是事务由存储引擎所决定，例如 MYISAM,MEMORY,ARCHIVE 都不支持事务，事务就是为了解决一组查询要么全部执行成功，要么全部执行失败。

MySQL 事务默认是采取自动提交的模式，除非显示开始一个事务。

```
SHOW VARIABLES LIKE 'AUTOCOMMIT';
```



http://www.itheima.com Copyright© 2018 黑马程序员

修改自动提交模式，0=OFF,1=ON

注意：修改自动提交对非事务类型的表是无效的，因为它们本身就没有提交和回滚的概念，还有一些命令是会强制自动提交的，比如 DLL 命令、lock tables 等。

```
SET AUTOCOMMIT=OFF 或 SET AUTOCOMMIT=0
```

10.1 事务的四大特征是什么？

数据库事务 transanction 正确执行的四个基本要素。ACID,原子性(Atomicity)、一致性(Correspondence)、隔离性(Isolation)、持久性(Durability)。

（1）原子性：整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

（2）一致性：在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。

（3）隔离性：隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行 相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆， 必须串行化或序列化请 求，使得在同一时间仅有一个请求用于同一数据。

（4）持久性：在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

10.2 Mysql 中四种隔离级别分别是什么？

隔离级别	脏读	不可重复读	幻读
Read uncommitted(读未提交)	是	是	是
Read committed（读已提交）	否	是	是
Repeatable read（可重复读）	否	否	是
Serializable（串行读）	否	否	否

http://www.itheima.com Copyright© 2018 黑马程序员

读未提交（READ UNCOMMITTED）：未提交读隔离级别也叫读脏，就是事务可以读取其它事务未提交的数据。

读已提交（READ COMMITTED）：在其它数据库系统比如 SQL Server 默认的隔离级别就是提交读，已提交读隔离级别就是在事务未提交之前所做的修改其它事务是不可见的。

可重复读（REPEATABLE READ）：保证同一个事务中的多次相同的查询的结果是一致的，比如一个事务一开始查询了一条记录然后过了几秒钟又执行了相同的查询，保证两次查询的结果是相同的，可重复读也是 mysql 的默认隔离级别。

可串行化（SERIALIZABLE）：可串行化就是保证读取的范围内没有新的数据插入，比如事务第一次查询得到某个范围的数据，第二次查询也同样得到了相同范围的数据，中间没有新的数据插入到该范围中。

11. MySQL 怎么创建存储过程（2017-11-25-wzz）

MySQL 存储过程是从 MySQL5.0 开始增加的新功能。存储过程的优点有一箩筐。不过最主要的还是执行效率和 SQL 代码封装。特别是 SQL 代码封装功能，如果没有存储过程，在外部程序访问数据库时，要组织很多 SQL 语句。特别是业务逻辑复杂的时候，一大堆的 SQL 和条件夹杂在代码中，让人不寒而栗。现在有了 MySQL 存储过程，业务逻辑可以封装存储过程中，这样不仅容易维护，而且执行效率也高。

一、创建 MySQL 存储过程

下面代码创建了一个叫 pr_add 的 MySQL 存储过程，这个 MySQL 存储过程有两个 int 类型的输入参数“a”、“b”，返回这两个参数的和。

1）drop procedure if exists pr_add; （备注：如果存在 pr_add 的存储过程，则先删掉）

2）计算两个数之和（备注：实现计算两个整数之和的功能）

```
create procedure pr_add ( a int, b int ) begin declare c int;
if a is null then set a = 0;
end if;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
if b is null then   set b = 0;
end if;
set c = a + b;
select c as sum;
```

二、调用 MySQL 存储过程

call pr_add(10, 20);

12. MySQL 触发器怎么写？（2017-11-25-wzz）

MySQL 包含对触发器的支持。触发器是一种与表操作有关的数据库对象，当触发器所在表上出现指定事件时，将调用该对象，即表的操作事件触发表上的触发器的执行。

在 MySQL 中，创建触发器语法如下：

```
CREATE TRIGGER trigger_name
trigger_time
trigger_event ON tbl_name
FOR EACH ROW
trigger_stmt
```

其中：

trigger_name：标识触发器名称，用户自行指定；

trigger_time：标识触发时机，取值为 BEFORE 或 AFTER；

trigger_event：标识触发事件，取值为 INSERT、UPDATE 或 DELETE；

tbl_name：标识建立触发器的表名，即在哪张表上建立触发器；

trigger_stmt：触发器程序体，可以是一句 SQL 语句，或者用 BEGIN 和 END 包含的多条语句。

由此可见，可以建立 6 种触发器，即：BEFORE INSERT、BEFORE UPDATE、BEFORE DELETE、AFTER INSERT、AFTER UPDATE、AFTER DELETE。

另外有一个限制是不能同时在一个表上建立 2 个相同类型的触发器，因此在一个表上最多建立 6 个触发器。

假设系统中有两个表：

http://www.itheima.com Copyright© 2018 黑马程序员

1)班级表 class(班级号 classID, 班内学生数 stuCount)

2)学生表 student(学号 stuID, 所属班级号 classID)

要创建触发器来使班级表中的班内学生数随着学生的添加自动更新，代码如下：

```
create trigger tri_stuInsert after insert
on student for each row
begin
declare c int;
set c = (select stuCount from class where classID=new.classID);
update class set stuCount = c + 1 where classID = new.classID;
```

查看触发器：

和查看数据库（show databases;）查看表格（show tables;）一样，查看触发器的语法如下：

```
SHOW TRIGGERS [FROM schema_name];
```

其中，schema_name 即 Schema 的名称，在 MySQL 中 Schema 和 Database 是一样的，也就是说，可以指定数据库名，这样就不必先“USE database_name;”了。

删除触发器：

和删除数据库、删除表格一样，删除触发器的语法如下：

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name
```

13. MySQL 语句优化（2017-11-26-wzz）

13.1 where 子句中可以对字段进行 null 值判断吗？

可以，比如 select id from t where num is null 这样的 sql 也是可以的。但是最好不要给数据库留 NULL，尽可能的使用 NOT NULL 填充数据库。不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了，不管是否插入值（NULL 也包含在内），都是占用 100 个字符的空间的，如果是 varchar 这样的变长字段，null 不占用空间。可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：select id from t where num = 0。

http://www.itheima.com Copyright© 2018 黑马程序员

13.2 select * from admin left join log on admin.admin_id = log.admin_id where log.admin_id>10 如何优化?

优化为： select * from (select * from admin where admin_id>10) T1 lef join log on T1.admin_id = log.admin_id。

使用 JOIN 时候，应该用小的结果驱动大的结果（left join 左边表结果尽量小如果有条件应该放到左边先处理，right join 同理反向），同时尽量把牵涉到多表联合的查询拆分多个 query（多个连表查询效率低，容易到之后锁表和阻塞）。

13.3 limit 的基数比较大时使用 between

例如：select * from admin order by admin_id limit 100000,10

优化为：select * from admin where admin_id between 100000 and 100010 order by admin_id。

13.4 尽量避免在列上做运算，这样导致索引失效

例如：select * from admin where year(admin_time)>2014

优化为： select * from admin where admin_time> '2014-01-01'

14. MySQL 中文乱码问题完美解决方案（2017-12-07-lwl）

解决乱码的核心思想是统一编码。我们在使用 MySQL 建数据库和建表时应尽量使用统一的编码，强烈推荐的是 utf8 编码，因为该编码几乎可以兼容世界上所有的字符。

数据库在安装的时候可以设置默认编码，在安装时就一定要设置为 utf8 编码。设置之后再创建的数据库和表如果不指定编码，默认都会使用 utf8 编码，省去了很多麻烦。

http://www.itheima.com Copyright© 2018 黑马程序员

数据库软件安装好之后可以通过如下命令查看默认编码：

1、查询数据库软件使用的默认编码格式

```
show variables like “%colla%”;
show varables like “%char%”
```

```
mysql> show variables like "%colla%";
+----------------------+-----------------+
| Variable_name        | Value           |
+----------------------+-----------------+
| collation_connection | utf8_general_ci |
| collation_database   | utf8_general_ci |
| collation_server     | utf8_general_ci |
+----------------------+-----------------+
3 rows in set (0.03 sec)

mysql> show variables like "%char%";
+--------------------------+----------------------------
| Variable_name            | Value
+--------------------------+----------------------------
| character_set_client     | utf8
| character_set_connection | utf8
| character_set_database   | utf8
| character_set_filesystem | binary
| character_set_results    | utf8
| character_set_server     | utf8
| character_set_system     | utf8
| character_sets_dir       | D:\Program Files\MySQL\MySQ
+--------------------------+----------------------------
```

其中 collation，代表了字符串排序（比较）的规则，如果值是 utf8_general_ci,代表使用 utf8 字符集大小写不敏感的自然方式比较。

如果 character_set 的值不为 utf8，那么可以使用如下命令修改为 utf8。

2、修改数据库默认编码为 utf8

```
SET character_set_client='utf8';
SET character_set_connection='utf8';
SET character_set_results='utf8';
```

如果不想设置数据库软件的全局默认编码，也可以单独修改或者设置某个具体数据库的编码也可以单独修改或设置某个数据库中某个表的编码。

3、创建数据库的时候指定使用 utf8 编码

```
CREATE DATABASE `test`
CHARACTER SET 'utf8'
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
COLLATE 'utf8_general_ci';
```

4、创建表的时候指定使用 utf8 编码

```
CREATE TABLE `database_user` (
`ID` varchar(40) NOT NULL default '',
`UserID` varchar(40) NOT NULL default '',
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

如果数据库已经创建好了，可以使用 show database 数据库名；和 show create table 表名；查看一下数据库和表的字符集是否为 utf8 ，如果不是则在命令行下面可以用如下命令，将数据库和表编码修改为 utf8.

5、修改具体某数据库或表的编码

```
ALTER DATABASE `db_name` DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
ALTER TABLE `tb_name` DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
```

15. 如何提高 MySQL 的安全性（2017-12-8-lwl）

1.如果 MySQL 客户端和服务器端的连接需要跨越并通过不可信任的网络，那么需要使用 ssh 隧道来加密该连接的通信。

2.使用 set password 语句来修改用户的密码，先“mysql -u root”登陆数据库系统，然后“mysql> update mysql.user set password=password(‘ newpwd’)”，最后执行“flush privileges”。

3.MySQL 需要提防的攻击有，防偷听、篡改、回放、拒绝服务等，不涉及可用性和容错方面。对所有的连接、查询、其他操作使用基于 ACL（ACL（访问控制列表）是一种路由器配置和控制网络访问的一种有力的工具，它可控制路由器应该允许或拒绝数据包通过，可监控流量，可自上向下检查网络的安全性，可检查和过滤数据和限制不必要的路由更新，因此让网络资源节约成本的 ACL 配置技术在生活中越来越广泛应用。）即访问控制列表的安全措施来完成。

4.设置除了 root 用户外的其他任何用户不允许访问 mysql 主数据库中的 user 表；

5.使用 grant 和 revoke 语句来进行用户访问控制的工作;

6.不要使用明文密码，而是使用 md5()和 sha1()等单向的哈系函数来设置密码;

http://www.itheima.com Copyright© 2018 黑马程序员

7.不要选用字典中的字来做密码;

8.采用防火墙可以去掉 50%的外部危险，让数据库系统躲在防火墙后面工作，或放置在 DMZ（DMZ 是英文“demilitarized zone”的缩写，隔离区，它是为了解决安装防火墙后外部网络的访问用户不能访问内部网络服务器的问题，而设立的一个非安全系统与安全系统之间的缓冲区。）区域中;

9.从因特网上用 nmap 来扫描 3306 端口，也可用 telnet server_host 3306 的方法测试，不允许从非信任网络中访问数据库服务器的 3306 号 tcp 端口，需要在防火墙或路由器上做设定;

10.服务端要对 SQL 进行预编译，避免 SQL 注入攻击，例如 where id=234，别人却输入 where id=234 or 1=1。

11.在传递数据给 mysql 时检查一下大小;

12.应用程序连接到数据库时应该使用一般的用户帐号，开放少数必要的权限给该用户;

13.学会使用 tcpdump 和 strings 工具来查看传输数据的安全性，例如 tcpdump -l -i eth0 -w -src or dst port 3306 strings。以普通用户来启动 mysql 数据库服务;

14.确信在 mysql 目录中只有启动数据库服务的用户才可以对文件有读和写的权限;

15.不许将 process 或 super 权限付给非管理用户，该 mysqladmin processlist 可以列举出当前执行的查询文本;super 权限可用于切断客户端连接、改变服务器运行参数状态、控制拷贝复制数据库的服务器;

16.如果不相信 dns 服务公司的服务，可以在主机名称允许表中只设置 ip 数字地址;

17.使用 max_user_connections 变量来使 mysqld 服务进程，对一个指定帐户限定连接数;

18.grant 语句也支持资源控制选项;

19.启动 mysqld 服务进程的安全选项开关，–local-infile=0 或 1，若是 0 则客户端程序就无法使用 local load data 了，赋权的一个例子 grant insert(user) on mysql.user to ‘user_name’@'host_name’;若使用–skip-grant-tables 系统将对任何用户的访问不做任何访问控制，但可以用 mysqladmin flush-privileges 或 mysqladmin reload 来开启访问控制;默认情况是 show databases 语句对所有用户开放，可以用–skip-show-databases 来关闭掉。

http://www.itheima.com Copyright© 2018 黑马程序员

23.碰到 error 1045(28000) access denied for user ‘root’@'localhost’ (using password:no)错误时，你需要重新设置密码，具体方法是:先用–skip-grant-tables 参数启动 mysqld，然后执行 mysql -u root mysql,mysql>update user set password=password(’ newpassword’) where user=’ root’ ;mysql>flush privileges;，最后重新启动 mysql 就可以了。

二、Oracle

1. 什么是存储过程，使用存储过程的好处？（2017-11-25-wzz)

存储过程（Stored Procedure ）是一组为了完成特定功能的 SQL 语句集，经编译后存储在数据库中。用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。存储过程是数据库中的一个重要对象，任何一个设计良好的数据库应用程序都应该用到存储过程。

优点:

(1)允许模块化程序设计，就是说只需要创建一次过程，以后在程序中就可以调用该过程任意次。

(2)允许更快执行，如果某操作需要执行大量 SQL 语句或重复执行，存储过程比 SQL 语句执行的要快。

(3)减少网络流量，例如一个需要数百行的 SQL 代码的操作有一条执行语句完成，不需要在网络中发送数百行代码。

(4) 更好的安全机制，对于没有权限执行存储过程的用户，也可授权他们执行存储过程。

存储过程的具体使用详见：http://www.cnblogs.com/yank/p/4235609.html

http://www.itheima.com Copyright© 2018 黑马程序员

2. Oracle 存储过程怎么创建？（2017-11-25-wzz）

存储过程创建语法：

```
create or replace procedure 存储过程名（param1 in type，param2 out type）
as
变量 1 类型（值范围）；
变量 2 类型（值范围）；
Begin
    Select count(*) into 变量 1 from 表 A where 列名=param1;
    If （判断条件） then
       Select 列名 into 变量 2 from 表 A where 列名=param1;
       Dbms_output。Put_line(‘打印信息’);
    Elsif （判断条件） then
       Dbms_output。Put_line(‘打印信息’);
    Else
       Raise 异常名（NO_DATA_FOUND）；
    End if;
Exception
    When others then
       Rollback;
End;
```

注意事项：

1. 存储过程参数不带取值范围，in 表示传入，out 表示输出

2. 变量带取值范围，后面接分号

http://www.itheima.com Copyright© 2018 黑马程序员

3. 在判断语句前最好先用 count（*）函数判断是否存在该条操作记录

4. 用 select 。。。into。。。给变量赋值

5. 在代码中抛异常用 raise+异常名

3. 如何使用 Oracle 的游标？（2017-11-25-wzz）

参考博客：https://www.cnblogs.com/sc-xx/archive/2011/12/03/2275084.html

（1）、Oracle 中的游标分为显示游标和隐式游标

（2）、显示游标是用 cursor...is 命令定义的游标，它可以对查询语句(select)返回的多条记录进行处理；

（3）、隐式游标是在执行插入（insert)、删除(delete)、修改(update) 和返回单条记录的查询(select)语句时由 PL/SQL 自动定义的。

（4）、显式游标的操作：打开游标、操作游标、关闭游标；PL/SQL 隐式地打开 SQL 游标，并在它内部处理 SQL 语句，然后关闭它。

4. Oracle 中字符串用什么连接？（2017-11-25-wzz）

Oracle 中使用 || 这个符号连接字符串 如 ‘abc’ || ‘d’ 的结果是 abcd。

http://www.itheima.com Copyright© 2018 黑马程序员

5. Oracle 中是如何进行分页查询的？（2017-11-25-wzz）

Oracle 中使用 rownum 来进行分页，这个是效率最好的分页方法，hibernate 也是使用 rownum 来进行 Oralce 分页的

```
select * from
  ( select rownum r,a from tabName where rownum <= 20 )
where r > 10
```

6. 存储过程和存储函数的特点和区别？（2017-11-25-wzz）

特点:

（1）、一般来说，存储过程实现的功能要复杂一点，而函数的实现的功能针对性比较强。

（2）、对于存储过程来说可以返回参数，而函数只能返回值或者表对象。

（3）、存储过程一般是作为一个独立的部分来执行，而函数可以作为查询语句的一个部分来调用，由于函数可以返回一个表对象，因此它可以在查询语句中位于 FROM 关键字的后面。

区别:

（1）、函数必须有返回值,而过程没有.

（2）、函数可以单独执行.而过程必须通过 execute 执行.

（3）、函数可以嵌入到 SQL 语句中执行.而过程不行.

其实我们可以将比较复杂的查询写成函数.然后到存储过程中去调用这些函数.

7. 存储过程与 SQL 的对比？（2017-11-21-gxb）

优势:

1、提高性能

http://www.itheima.com Copyright© 2018 黑马程序员

SQL 语句在创建过程时进行分析和编译。 存储过程是预编译的，在首次运行一个存储过程时，查询优化器对其进行分析、优化，并给出最终被存在系统表中的存储计划，这样，在执行过程时便可节省此开销。

2、降低网络开销

存储过程调用时只需用提供存储过程名和必要的参数信息，从而可降低网络的流量。

3、便于进行代码移植

数据库专业人员可以随时对存储过程进行修改，但对应用程序源代码却毫无影响，从而极大的提高了程序的可移植性。

4、更强的安全性

1）系统管理员可以对执行的某一个存储过程进行权限限制，避免非授权用户对数据的访问

2）在通过网络调用过程时，只有对执行过程的调用是可见的。 因此，恶意用户无法看到表和数据库对象名称、嵌入自己的 Transact-SQL 语句或搜索关键数据。

3）使用过程参数有助于避免 SQL 注入攻击。 因为参数输入被视作文字值而非可执行代码，所以，攻击者将命令插入过程内的 Transact-SQL 语句并损害安全性将更为困难。

4）可以对过程进行加密，这有助于对源代码进行模糊处理。

劣势：

1、存储过程需要专门的数据库开发人员进行维护，但实际情况是，往往由程序开发员人员兼职

2、设计逻辑变更，修改存储过程没有 SQL 灵活

8. 你觉得存储过程和 SQL 语句该使用哪个？（2017-11-21-gxb）

1、在一些高效率或者规范性要求比较高的项目，建议采用存储过程

2、对于一般项目建议采用参数化命令方式，是存储过程与 SQL 语句一种折中的方式

http://www.itheima.com Copyright© 2018 黑马程序员

3、对于一些算法要求比较高，涉及多条数据逻辑，建议采用存储过程

9. 触发器的作用有哪些？（2017-11-21-gxb）

1）触发器可通过数据库中的相关表实现级联更改；通过级联引用完整性约束可以更有效地执行这些更改。

2）触发器可以强制比用 CHECK 约束定义的约束更为复杂的约束。与 CHECK 约束不同，触发器可以引用其它表中的列。例如，触发器可以使用另一个表中的 SELECT 比较插入或更新的数据，以及执行其它操作，如修改数据或显示用户定义错误信息。

3）触发器还可以强制执行业务规则

4）触发器也可以评估数据修改前后的表状态，并根据其差异采取对策。

参考资料：http://www.cnblogs.com/yank/p/4193820.html

10. 在千万级的数据库查询中，如何提高效率？（2017-11-23-gxb）

1）数据库设计方面

a. 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

b. 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，

http://www.itheima.com Copyright© 2018 黑马程序员

如： select id from t where num is null 可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询： select id from t where num=0

c. 并不是所有索引对查询都有效，SQL 是根据表中数据来进行查询优化的，当索引列有大量数据重复时,查询可能不会去利用索引，如一表中有字段 sex，male、female 几乎各一半，那么即使在 sex 上建了索引也对查询效率起不了作用。

d. 索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过 6 个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

e. 应尽可能的避免更新索引数据列，因为索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新索引数据列，那么需要考虑是否应将该索引建为索引。

f. 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

g. 尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

h. 尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

i. 避免频繁创建和删除临时表，以减少系统表资源的消耗。

j. 临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

k. 在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成

http://www.itheima.com Copyright© 2018 黑马程序员

大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先 create table，然后 insert。

l. 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

2)SQL 语句方面

a. 应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

b. 应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：select id from t where num=10 or num=20 可以这样查询： select id from t where num=10 union all select id from t where num=20

c. in 和 not in 也要慎用，否则会导致全表扫描，如： select id from t where num in(1,2,3) 对于连续的数值，能用 between 就不要用 in 了： select id from t where num between 1 and 3

d. 下面的查询也将导致全表扫描： select id from t where name like ‘%abc%’

e. 如果在 where 子句中使用参数，也会导致全表扫描。因为 SQL 只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描： select id from t where num=@num 可以改为强制查询使用索引： select id from t with(index(索引名)) where num=@num

f. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：select id from t where num/2=100 应改为: select id from t where num=100*2

g. 应尽量避免在where子句中对字段进行函数操作,这将导致引擎放弃使用索引而进行全表扫描。如：select id from t where substring(name,1,3)= ‘abc’ –name 以 abc 开头的 id select id from t where datediff(day,createdate,’2005-11-30’)=0–‘2005-11-30’生成的 id 应改为: select id from t where name like ‘abc%’ select id from t where createdate>=’2005-11-30’and createdate<’2005-12-1’

http://www.itheima.com Copyright© 2018 黑马程序员

h. 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

i. 不要写一些没有意义的查询，如需要生成一个空表结构： select col1,col2 into #t from t where 1=0 这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样： create table #t(...)

j. 很多时候用 exists 代替 in 是一个好的选择： select num from a where num in(select num from b) 用下面的语句替换： select num from a where exists(select 1 from b where num=a.num)

k. 任何地方都不要使用 select * from t ，用具体的字段列表代替“*”，不要返回用不到的任何字段。

l. 尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过 1 万行，那么就应该考虑改写。

m. 尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

n. 尽量避免大事务操作，提高系统并发能力。

3)java 方面：重点内容

a.尽可能的少造对象。

b.合理摆正系统设计的位置。大量数据操作，和少量数据操作一定是分开的。大量的数据操作，肯定不是 ORM 框架搞定的。，

c.使用 jDBC 链接数据库操作数据

d.控制好内存，让数据流起来，而不是全部读到内存再处理，而是边读取边处理；

e.合理利用内存，有的数据要缓存

http://www.itheima.com Copyright© 2018 黑马程序员

第七章 框架

一、SpringMVC

1. SpringMVC 的工作原理（2017-11-13-lyq）

a. 用户向服务器发送请求，请求被 springMVC 前端控制器 DispatchServlet 捕获；

b. DispatcherServle 对请求 URL 进行解析，得到请求资源标识符(URL)，然后根据该 URL 调用 HandlerMapping 将请求映射到处理器 HandlerExcutionChain；

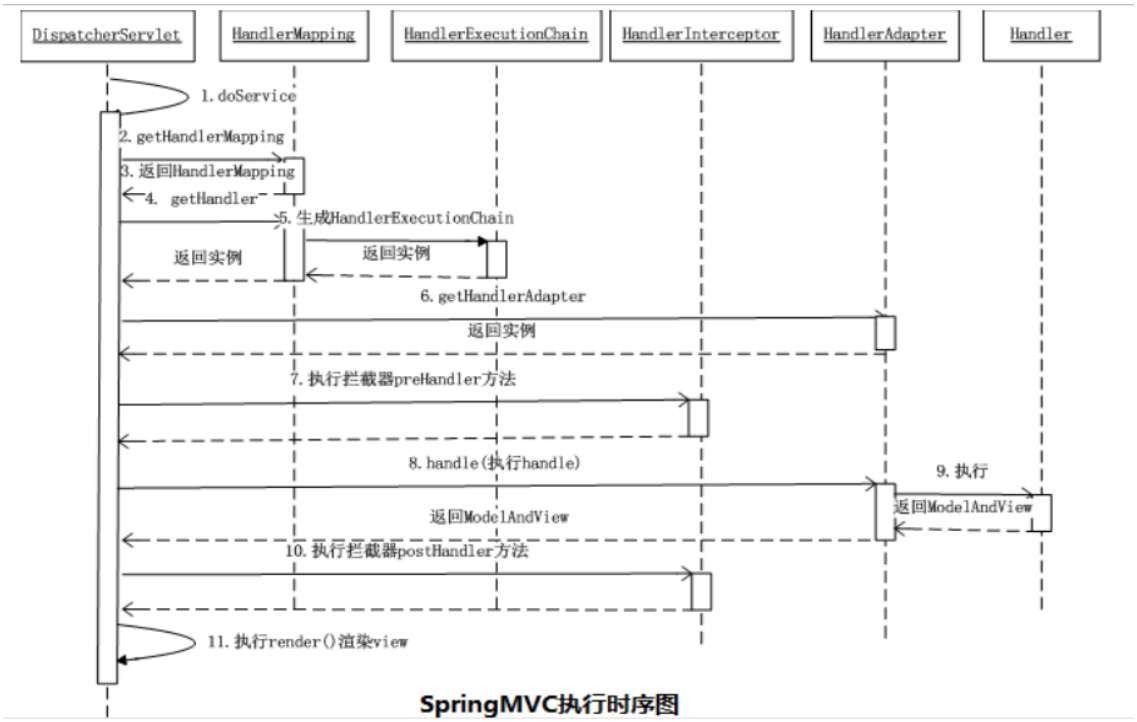
c. DispatchServlet 根据获得 Handler 选择一个合适的 HandlerAdapter 适配器处理；

d. Handler 对数据处理完成以后将返回一个 ModelAndView（）对象给 DisPatchServlet;

e. Handler 返回的 ModelAndView()只是一个逻辑视图并不是一个正式的视图，DispatcherSevlet 通过 ViewResolver 试图解析器将逻辑视图转化为真正的视图 View;

h. DispatcherServle 通过 model 解析出 ModelAndView()中的参数进行解析最终展现出完整的 view 并返回给客户端;

http://www.itheima.com Copyright© 2018 黑马程序员



SpringMVC执行时序图

2. SpringMVC 常用注解都有哪些？（2017-11-24-gxb）

@requestMapping 用于请求 url 映射。

@RequestBody 注解实现接收 http 请求的 json 数据，将 json 数据转换为 java 对象。

@ResponseBody 注解实现将 controller 方法返回对象转化为 json 响应给客户。

3. 如何开启注解处理器和适配器？（2017-11-24-gxb）

我们在项目中一般会在 springmvc.xml 中通过开启 <mvc:annotation-driven>来实现注解处理器和适配器的开启。

4. 如何解决 get 和 post 乱码问题？（2017-11-24-gxb）

解决 post 请求乱码:我们可以在 web.xml 里边配置一个 CharacterEncodingFilter 过滤器。 设置为 utf-8.

解决 get 请求的乱码:有两种方法。对于 get 请求中文参数出现乱码解决方法有两个:

http://www.itheima.com Copyright© 2018 黑马程序员

1. 修改 tomcat 配置文件添加编码与工程编码一致。

2. 另外一种方法对参数进行重新编码 String userName = New String(Request.getParameter(“userName”).getBytes(“ISO8859-1”), “utf-8”);

二、Spring

1. 谈谈你对 Spring 的理解（2017-11-13-lyq）

Spring 是一个开源框架，为简化企业级应用开发而生。Spring 可以是使简单的 JavaBean 实现以前只有 EJB 才能实现的功能。Spring 是一个 IOC 和 AOP 容器框架。

Spring 容器的主要核心是：

控制反转（IOC），传统的 java 开发模式中，当需要一个对象时，我们会自己使用 new 或者 getInstance 等直接或者间接调用构造方法创建一个对象。而在 spring 开发模式中，spring 容器使用了工厂模式为我们创建了所需要的对象，不需要我们自己创建了，直接调用 spring 提供的对象就可以了，这是控制反转的思想。

依赖注入（DI），spring 使用 javaBean 对象的 set 方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程，就是依赖注入的思想。

面向切面编程（AOP），在面向对象编程（oop）思想中，我们将事物纵向抽成一个个的对象。而在面向切面编程中，我们将一个个的对象某些类似的方面横向抽成一个切面，对这个切面进行一些如权限控制、事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。AOP 底层是动态代理，如果是接口采用 JDK 动态代理，如果是类采用 CGLIB 方式实现动态代理。

2. Spring 中的设计模式（2017-11-13-lyq）

a. 单例模式——spring 中两种代理方式，若目标对象实现了若干接口，spring 使用 jdk 的 java.lang.reflect.Proxy

http://www.itheima.com Copyright© 2018 黑马程序员

类代理。若目标兑现没有实现任何接口，spring 使用 CGLIB 库生成目标类的子类。

单例模式——在 spring 的配置文件中设置 bean 默认为单例模式。

b. 模板方式模式——用来解决代码重复的问题。

比如：RestTemplate、JmsTemplate、JpaTemplate

d. 前端控制器模式——spring 提供了前端控制器 DispatherServlet 来对请求进行分发。

e. 试图帮助（view helper）——spring 提供了一系列的 JSP 标签，高效宏来帮助将分散的代码整合在试图中。

f. 依赖注入——贯穿于 BeanFactory/ApplacationContext 接口的核心理念。

g. 工厂模式——在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用同一个接口来指向新创建的对象。Spring 中使用 beanFactory 来创建对象的实例。

3. Spring 的常用注解（2017-11-13-lyq）

Spring 在 2.5 版本以后开始支持注解的方式来配置依赖注入。可以用注解的方式来代替 xml 中 bean 的描述。注解注入将会被容器在 XML 注入之前被处理，所以后者会覆盖掉前者对于同一个属性的处理结果。

注解装配在 spring 中默认是关闭的。所以需要在 spring 的核心配置文件中配置一下才能使用基于注解的装配模式。配置方式如下：

<context:annotation-config />

常用的注解：

@Required:该注解应用于设值方法

@Autowired：该注解应用于有值设值方法、非设值方法、构造方法和变量。

@Qualifier：该注解和@Autowired 搭配使用，用于消除特定 bean 自动装配的歧义。

http://www.itheima.com Copyright© 2018 黑马程序员

4. 简单介绍一下 Spring bean 的生命周期（2017-11-21-gxb）

bean 定义：在配置文件里面用<bean></bean>来进行定义。

bean 初始化：有两种方式初始化:

1.在配置文件中通过指定 init-method 属性来完成

2.实现 org.springframwork.beans.factory.InitializingBean 接口

bean 调用：有三种方式可以得到 bean 实例，并进行调用

bean 销毁：销毁有两种方式

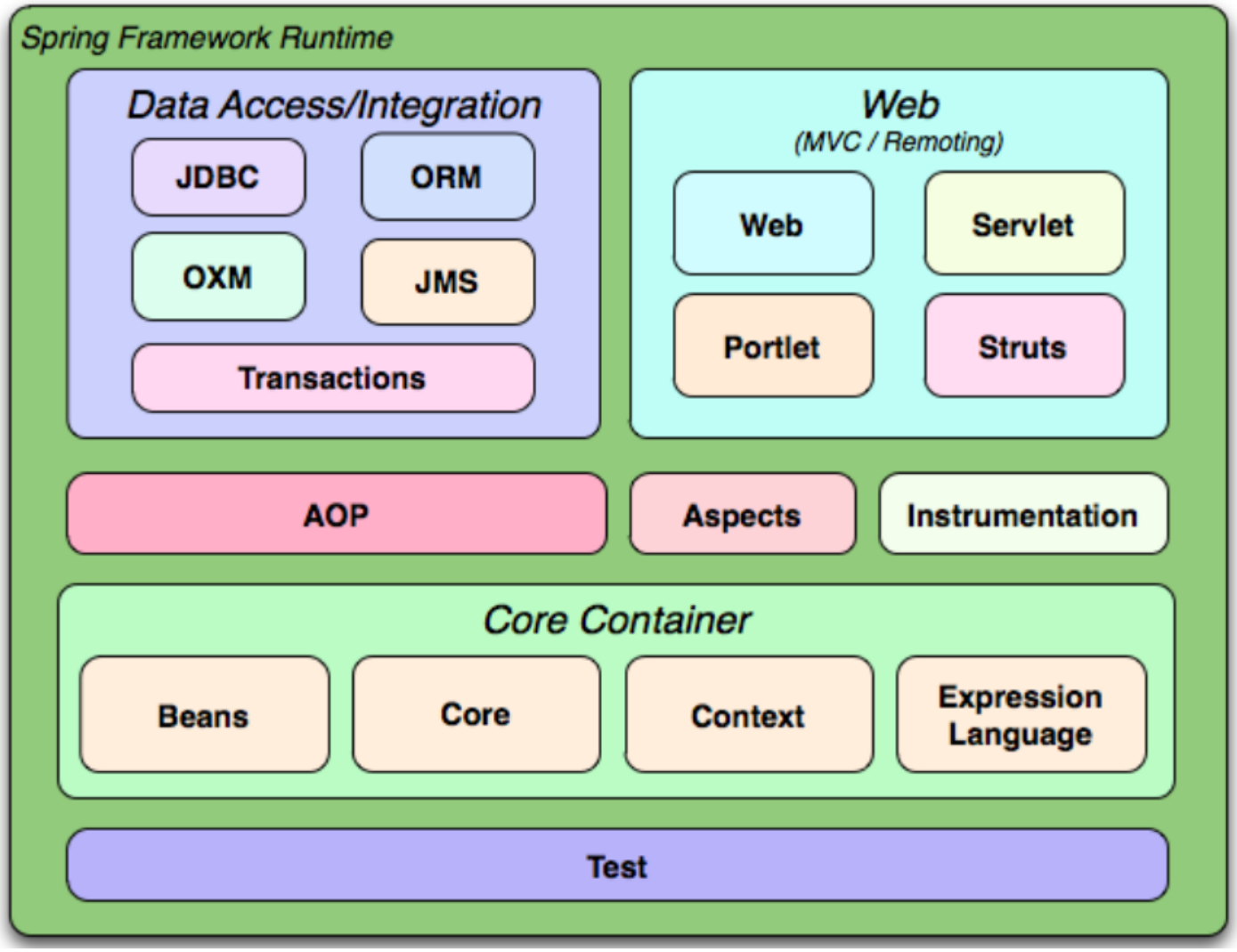
1.使用配置文件指定的 destroy-method 属性

2.实现 org.springframwork.bean.factory.DisposeableBean 接口

参考资料：https://www.cnblogs.com/zrtqsk/p/3735273.html

http://www.itheima.com Copyright© 2018 黑马程序员

5. Spring 结构图（2017-11-22-lyq）



（1）核心容器：包括 Core、Beans、Context、EL 模块。

Core 模块：封装了框架依赖的最底层部分，包括资源访问、类型转换及一些常用工具类。

Beans 模块：提供了框架的基础部分，包括反转控制和依赖注入。其中 Bean Factory 是容器核心，本质是“工厂设计模式”的实现，而且无需编程实现“单例设计模式”，单例完全由容器控制，而且提倡面向接口编程，而非面向实现编程；所有应用程序对象及对象间关系由框架管理，从而真正把你从程序逻辑中把维护对象之间的依赖关系提取出来，所有这些依赖关系都由 BeanFactory 来维护。

Context 模块：以 Core 和 Beans 为基础，集成 Beans 模块功能并添加资源绑定、数据验证、国际化、Java EE 支

http://www.itheima.com Copyright© 2018 黑马程序员

持、容器生命周期、事件传播等；核心接口是 ApplicationContext。

EL 模块：提供强大的表达式语言支持，支持访问和修改属性值，方法调用，支持访问及修改数组、容器和索引器，命名变量，支持算数和逻辑运算，支持从 Spring 容器获取 Bean，它也支持列表投影、选择和一般的列表聚合等。

（2）AOP、Aspects 模块：

AOP 模块：Spring AOP 模块提供了符合 AOP Alliance 规范的面向方面的编程（aspect-oriented programming）实现，提供比如日志记录、权限控制、性能统计等通用功能和业务逻辑分离的技术，并且能动态的把这些功能添加到需要的代码中；这样各专其职，降低业务逻辑和通用功能的耦合。

Aspects 模块：提供了对 AspectJ 的集成，AspectJ 提供了比 Spring ASP 更强大的功能。

数据访问/集成模块：该模块包括了 JDBC、ORM、OXM、JMS 和事务管理。

事务模块：该模块用于 Spring 管理事务，只要是 Spring 管理对象都能得到 Spring 管理事务的好处，无需在代码中进行事务控制了，而且支持编程和声明性的事务管理。

JDBC 模块：提供了一个 JBDC 的样例模板，使用这些模板能消除传统冗长的 JDBC 编码还有必须的事务控制，而且能享受到 Spring 管理事务的好处。

ORM 模块：提供与流行的“对象-关系”映射框架的无缝集成，包括 Hibernate、JPA、MyBatis 等。而且可以使用 Spring 事务管理，无需额外控制事务。

OXM 模块：提供了一个对 Object/XML 映射实现，将 java 对象映射成 XML 数据，或者将 XML 数据映射成 java 对象，Object/XML 映射实现包括 JAXB、Castor、XMLBeans 和 XStream。

JMS 模块：用于 JMS(Java Messaging Service)，提供一套 “消息生产者、消息消费者”模板用于更加简单的使用 JMS，JMS 用于用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

Web/Remoting 模块：Web/Remoting 模块包含了 Web、Web-Servlet、Web-Struts、Web-Porlet 模块。

Web 模块：提供了基础的 web 功能。例如多文件上传、集成 IoC 容器、远程过程访问（RMI、Hessian、Burlap）

http://www.itheima.com Copyright© 2018 黑马程序员

以及 Web Service 支持，并提供一个 RestTemplate 类来提供方便的 Restful services 访问。

Web-Servlet 模块：提供了一个 Spring MVC Web 框架实现。Spring MVC 框架提供了基于注解的请求资源注入、更简单的数据绑定、数据验证等及一套非常易用的 JSP 标签，完全无缝与 Spring 其他技术协作。

Web-Struts 模块：提供了与 Struts 无缝集成，Struts1.x 和 Struts2.x 都支持

Test 模块： Spring 支持 Junit 和 TestNG 测试框架，而且还额外提供了一些基于 Spring 的测试功能，比如在测试 Web 框架时，模拟 Http 请求的功能。

6. Spring 能帮我们做什么？（2017-11-22-lyq）

a. Spring 能帮我们根据配置文件创建及组装对象之间的依赖关系。

Spring 根据配置文件来进行创建及组装对象间依赖关系，只需要改配置文件即可

b. Spring 面向切面编程能帮助我们无耦合的实现日志记录，性能统计，安全控制。

Spring 面向切面编程能提供一种更好的方式来完成，一般通过配置方式，而且不需要在现有代码中添加任何额外代码，现有代码专注业务逻辑。

c. Spring 能非常简单的帮我们管理数据库事务。

采用 Spring，我们只需获取连接，执行 SQL，其他事物相关的都交给 Spring 来管理了。

d. Spring 还能与第三方数据库访问框架 (如 Hibernate、JPA) 无缝集成, 而且自己也提供了一套 JDBC 访问模板，来方便数据库访问。

e. Spring 还能与第三方 Web（如 Struts、JSF）框架无缝集成，而且自己也提供了一套 Spring MVC 框架，来方便 web 层搭建。

f. Spring 能方便的与 Java EE（如 Java Mail、任务调度）整合，与更多技术整合（比如缓存框架）。

http://www.itheima.com Copyright© 2018 黑马程序员

7. 请描述一下 Spring 的事务（2017-11-22-lyq）

声明式事务管理的定义：用在 Spring 配置文件中声明式的处理事务来代替代码式的处理事务。这样的好处是，事务管理不侵入开发的组件，具体来说，业务逻辑对象就不会意识到正在事务管理之中，事实上也应该如此，因为事务管理是属于系统层面的服务，而不是业务逻辑的一部分，如果想要改变事务管理策划的话，也只需要在定义文件中重新配置即可，这样维护起来极其方便。

基于 TransactionInterceptor 的声明式事务管理：两个次要的属性： transactionManager，用来指定一个事务治理器，并将具体事务相关的操作请托给它；其他一个是 Properties 类型的 transactionAttributes 属性，该属性的每一个键值对中，键指定的是方法名，方法名可以行使通配符，而值就是表现呼应方法的所运用的事务属性。

```
<beans...>
.....
<bean id="transactionInterceptor"
      class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributes">
      <props>
          <prop key="transfer">PROPAGATION_REQUIRED</prop>
      </props>
  </property>
</bean>
<bean id="bankServiceTarget"
       class="footmark.spring.core.tx.declare.origin.BankServiceImpl">
  <property name="bankDao" ref="bankDao"/>
</bean>
<bean id="bankService"
       class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="bankServiceTarget"/>
  <property name="interceptorNames">
    <list>
       <idref bean="transactionInterceptor"/>
    </list>
  </property>
</bean>
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
25. ......
26. </beans>
```

基于 TransactionProxyFactoryBean 的声明式事务管理：设置配置文件与先前比照简化了许多。我们把这类设置配置文件格式称为 Spring 经典的声明式事务治理

```
1. <beans......>
2. ......
3. <bean id="bankServiceTarget"
4.       class="footmark.spring.core.tx.declare.classic.BankServiceImpl">
5.   <property name="bankDao" ref="bankDao"/>
6. </bean>
7. <bean id="bankService"
8.       class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
9.    <property name="target" ref="bankServiceTarget"/>
10.   <property name="transactionManager" ref="transactionManager"/>
11.   <property name="transactionAttributes">
12.     <props>
13.        <prop key="transfer">PROPAGATION_REQUIRED</prop>
14.     </props>
15.    </property>
16. </bean>
17. ......
18. </beans></beans>
```

基于 <tx> 命名空间的声明式事务治理：在前两种方法的基础上，Spring 2.x 引入了 <tx> 命名空间，连络行使 <aop> 命名空间，带给开发人员设置配备声明式事务的全新体验。

```
1. <beans......>
2. ......
3. <bean id="bankService"
4.       class="footmark.spring.core.tx.declare.namespace.BankServiceImpl">
5.    <property name="bankDao" ref="bankDao"/>
6. </bean>
7. <tx:advice id="bankAdvice" transaction-manager="transactionManager">
8.    <tx:attributes>
9.       <tx:method name="transfer" propagation="REQUIRED"/>
10.   </tx:attributes>
11. </tx:advice>
12.
13. <aop:config>
14.     <aop:pointcut id="bankPointcut" expression="execution(* *.transfer(..))"/>
15.     <aop:advisor advice-ref="bankAdvice" pointcut-ref="bankPointcut"/>
```


 Copyright© 2018 黑马程序员

```
16. </aop:config>
17. ......
18. </beans>
```

基于 @Transactional 的声明式事务管理：Spring 2.x 还引入了基于 Annotation 的体式格式，具体次要触及 @Transactional 标注。@Transactional 可以浸染于接口、接口方法、类和类方法上。算作用于类上时，该类的一切 public 方法将都具有该类型的事务属性。

```
1. @Transactional(propagation = Propagation.REQUIRED)
2. public boolean transfer(Long fromId, Long toId, double amount) {
3.         return bankDao.transfer(fromId, toId, amount);
4. }
```

编程式事物管理的定义：在代码中显式挪用 beginTransaction()、commit()、rollback()等事务治理相关的方法，这就是编程式事务管理。Spring 对事物的编程式管理有基于底层 API 的编程式管理和基于 TransactionTemplate 的编程式事务管理两种方式。

基于底层 API 的编程式管理：凭证 PlatformTransactionManager、TransactionDefinition 和 TransactionStatus 三个焦点接口，来实现编程式事务管理。

```
5. Public class BankServiceImpl implements BancService{
6. Private BanckDao bankDao;
7. private TransactionDefinition txDefinition;
8. private PlatformTransactionManager txManager;
9. ......
10. public boolean transfer(Long fromId, Long toId, double amount) {
11. TransactionStatus txStatus = txManager.getTransaction(txDefinition);
12. boolean result = false;
13. try {
14. result = bankDao.transfer(fromId, toId, amount);
15. txManager.commit(txStatus);
16. } catch (Exception e) {
17. result = false;
18. txManager.rollback(txStatus);
19. System.out.println("Transfer Error!");
20. }
21. return result;
22. }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
23. }
```

基于 TransactionTemplate 的编程式事务管理:为了不损坏代码原有的条理性，避免出现每一个方法中都包括相同的启动事物、提交、回滚事物样板代码的现象，spring 提供了 transactionTemplate 模板来实现编程式事务管理。

```
1. public class BankServiceImpl implements BankService {
2. private BankDao bankDao;
3. private TransactionTemplate transactionTemplate;
4. ......
5. public boolean transfer(final Long fromId, final Long toId, final double amount) {
6. return (Boolean) transactionTemplate.execute(new TransactionCallback(){
7. public Object doInTransaction(TransactionStatus status) {
8. Object result;
9. try {
10. result = bankDao.transfer(fromId, toId, amount);
11. } catch (Exception e) {
12. status.setRollbackOnly();
13. result = false;
14. System.out.println("Transfer Error!");
15. }
16. return result;
17. }
18. });
19. }
20. }
```

编程式事务与声明式事务的区别：

1）编程式事务是自己写事务处理的类，然后调用

2）声明式事务是在配置文件中配置，一般搭配在框架里面使用！

8. BeanFactory 常用的实现类有哪些？ （2017-12-03-gxb）

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和依赖从正真的应用代码中分离。常用的 BeanFactory 实现有 DefaultListableBeanFactory 、 XmlBeanFactory 、 ApplicationContext 等。

XMLBeanFactory，最常用的就是 org.springframework.beans.factory.xml.XmlBeanFactory ，它根据 XML 文件中

http://www.itheima.com Copyright© 2018 黑马程序员

的定义加载 beans。该容器从 XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

9. 解释 Spring JDBC、Spring DAO 和 Spring ORM（2017-12-03-gxb）

Spring-DAO 并非 Spring 的一个模块，它实际上是指示你写 DAO 操作、写好 DAO 操作的一些规范。因此，对于访问你的数据它既没有提供接口也没有提供实现更没有提供模板。在写一个 DAO 的时候，你应该使用 @Repository 对其进行注解，这样底层技术(JDBC，Hibernate，JPA，等等)的相关异常才能一致性地翻译为相应的 DataAccessException 子类。

Spring-JDBC 提供了 Jdbc 模板类，它移除了连接代码以帮你专注于 SQL 查询和相关参数。Spring-JDBC 还提供了一个 JdbcDaoSupport，这样你可以对你的 DAO 进行扩展开发。它主要定义了两个属性：一个 DataSource 和一个 JdbcTemplate，它们都可以用来实现 DAO 方法。JdbcDaoSupport 还提供了一个将 SQL 异常转换为 Spring DataAccessExceptions 的异常翻译器。

Spring-ORM 是一个囊括了很多持久层技术(JPA，JDO，Hibernate，iBatis)的总括模块。对于这些技术中的每一个，Spring 都提供了集成类，这样每一种技术都能够在遵循 Spring 的配置原则下进行使用，并平稳地和 Spring 事务管理进行集成。

对于每一种技术，配置主要在于将一个 DataSource bean 注入到某种 SessionFactory 或者 EntityManagerFactory 等 bean 中。纯 JDBC 不需要这样的一个集成类(JdbcTemplate 除外)，因为 JDBC 仅依赖于一个 DataSource。

如果你计划使用一种 ORM 技术，比如 JPA 或者 Hibernate，那么你就不需要 Spring-JDBC 模块了，你需要的是这个 Spring-ORM 模块。

10. 简单介绍一下 Spring WEB 模块。（2017-12-03-gxb）

Spring 的 WEB 模块是构建在 application context 模块基础之上，提供一个适合 web 应用的上下文。这个模块

http://www.itheima.com Copyright© 2018 黑马程序员

也包括支持多种面向 web 的任务，如透明地处理多个文件上传请求和程序级请求参数的绑定到你的业务对象。它也有对 Jakarta Struts 的支持。

11. Spring 配置文件有什么作用？（2017-12-03-gxb）

Spring 配置文件是个 XML 文件，这个文件包含了类信息，描述了如何配置它们，以及如何相互调用。

12. 什么是 Spring IOC 容器？（2017-12-03-gxb）

IOC 控制反转：Spring IOC 负责创建对象，管理对象。通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。

13. IOC 的优点是什么?

IOC 或 依赖注入把应用的代码量降到最低。它使应用容易测试，单元测试不再需要单例和 JNDI 查找机制。最小的代价和最小的侵入性使松散耦合得以实现。IOC 容器支持加载服务时的饿汉式初始化和懒加载。

14. ApplicationContext 的实现类有哪些？（2017-12-03-gxb）

FileSystemXmlApplicationContext：此容器从一个 XML 文件中加载 beans 的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。

ClassPathXmlApplicationContext：此容器也从一个 XML 文件中加载 beans 的定义，这里，你需要正确设置 classpath 因为这个容器将在 classpath 里找 bean 配置。

WebXmlApplicationContext：此容器加载一个 XML 文件，此文件定义了一个 WEB 应用的所有 bean。

http://www.itheima.com Copyright© 2018 黑马程序员

15. BeanFactory 与 AppliacationContext 有什么区别（2017-12-03-gxb）

1. BeanFactory

基础类型的 IOC 容器，提供完成的 IOC 服务支持。如果没有特殊指定，默认采用延迟初始化策略。相对来说，容器启动初期速度较快，所需资源有限。

2.ApplicationContext

ApplicationContext 是在 BeanFactory 的基础上构建，是相对比较高级的容器实现，除了 BeanFactory 的所有支持外，ApplicationContext 还提供了事件发布、国际化支持等功能。ApplicationContext 管理的对象，在容器启动后默认全部初始化并且绑定完成。

16. 什么是 Spring 的依赖注入？（2017-12-04-gxb）

平常的 java 开发中，程序员在某个类中需要依赖其它类的方法，则通常是 new 一个依赖类再调用类实例的方法，这种开发存在的问题是 new 的类实例不好统一管理，spring 提出了依赖注入的思想，即依赖类不由程序员实例化，而是通过 spring 容器帮我们 new 指定实例并且将实例注入到需要该对象的类中。依赖注入的另一种说法是“控制反转”，通俗的理解是：平常我们 new 一个实例，这个实例的控制权是我们程序员，而控制反转是指 new 实例工作不由我们程序员来做而是交给 spring 容器来做。

17. 有哪些不同类型的 IOC（依赖注入）方式？（2017-12-04-gxb）

Spring 提供了多种依赖注入的方式。

1.Set 注入

2.构造器注入

3.静态工厂的方法注入

http://www.itheima.com Copyright© 2018 黑马程序员

4.实例工厂的方法注入

参考资料：https://www.cnblogs.com/java-class/p/4727775.html

18. 什么是 Spring beans？ （2017-12-04-gxb）

Spring beans 是那些形成 Spring 应用的主干的 java 对象。它们被 Spring IOC 容器初始化，装配，和管理。这些 beans 通过容器中配置的元数据创建。比如，以 XML 文件中<bean/> 的形式定义。

Spring 框架定义的 beans 都是单例 beans。

19. 一个 Spring Beans 的定义需要包含什么？ （2017-12-04-gxb）

一个 Spring Bean 的定义包含容器必知的所有配置元数据，包括如何创建一个 bean，它的生命周期详情及它的依赖。

http://www.itheima.com Copyright© 2018 黑马程序员

20. 你怎样定义类的作用域？（2017-12-04-gxb）

当定义一个<bean> 在 Spring 里，我们还能给这个 bean 声明一个作用域。它可以通过 bean 定义中的 scope 属性来定义。如，当 Spring 要在需要的时候每次生产一个新的 bean 实例，bean 的 scope 属性被指定为 prototype。另一方面，一个 bean 每次使用的时候必须返回同一个实例，这个 bean 的 scope 属性必须设为 singleton。

21. Spring 支持的几种 bean 的作用域。（2017-12-04-gxb）

Spring 框架支持以下五种 bean 的作用域：

singleton : bean 在每个 Spring ioc 容器中只有一个实例。

prototype：一个 bean 的定义可以有多个实例。

request：每次 http 请求都会创建一个 bean，该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。

session：在一个 HTTP Session 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。

global-session：在一个全局的 HTTP Session 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。

缺省的 Spring bean 的作用域是 Singleton。

22. Spring 框架中的单例 bean 是线程安全的吗？（2017-12-04-gxb）

Spring 框架中的单例 bean 不是线程安全的。

http://www.itheima.com Copyright© 2018 黑马程序员

23. 什么是 Spring 的内部 bean？（2017-12-04-gxb）

当一个 bean 仅被用作另一个 bean 的属性时，它能被声明为一个内部 bean，为了定义 inner bean，在 Spring 的 基于 XML 的 配置元数据中，可以在 <property/>或 <constructor-arg/> 元素内使用<bean/> 元素，内部 bean 通常是匿名的，它们的 Scope 一般是 prototype。

24. 在 Spring 中如何注入一个 java 集合？（2017-12-04-gxb）

Spring 提供以下几种集合的配置元素：

<list>类型用于注入一列值，允许有相同的值。

<set> 类型用于注入一组值，不允许有相同的值。

<map> 类型用于注入一组键值对，键和值都可以为任意类型。

<props>类型用于注入一组键值对，键和值都只能为 String 类型。

25. 什么是 bean 的自动装配？（2017-12-04-gxb）

无须在 Spring 配置文件中描述 javaBean 之间的依赖关系（如配置<property>、<constructor-arg>）。IOC 容器会自动建立 javabean 之间的关联关系。

26. 解释不同方式的自动装配 。（2017-12-04-gxb）

有五种自动装配的方式，可以用来指导 Spring 容器用自动装配方式来进行依赖注入。

1）no：默认的方式是不进行自动装配，通过显式设置 ref 属性来进行装配。

2）byName：通过参数名自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成 byname，之后容器试图匹配、装配和该 bean 的属性具有相同名字的 bean。

http://www.itheima.com Copyright© 2018 黑马程序员

3）byType:: 通过参数类型自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成 byType，之后容器试图匹配、装配和该 bean 的属性具有相同类型的 bean。如果有多个 bean 符合条件，则抛出错误。

4）constructor：这个方式类似于 byType， 但是要提供给构造器参数，如果没有确定的带参数的构造器参数类型，将会抛出异常。

5）autodetect：首先尝试使用 constructor 来自动装配，如果无法工作，则使用 byType 方式。

27. 什么是基于 Java 的 Spring 注解配置? 给一些注解的例子(2017-12-05-gxb)

基于 Java 的配置，允许你在少量的 Java 注解的帮助下，进行你的大部分 Spring 配置而非通过 XML 文件。以@Configuration 注解为例，它用来标记类可以当做一个 bean 的定义，被 Spring IOC 容器使用。另一个例子是@Bean 注解，它表示此方法将要返回一个对象，作为一个 bean 注册进 Spring 应用上下文。

28. 什么是基于注解的容器配置? （2017-12-05-gxb)

相对于 XML 文件，注解型的配置依赖于通过字节码元数据装配组件，而非尖括号的声明。开发者通过在相应的类，方法或属性上使用注解的方式，直接组件类中进行配置，而不是使用 xml 表述 bean 的装配关系。

29. 怎样开启注解装配? （2017-12-05-gxb)

注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在 Spring 配置文件中配置 <context:annotation-config/>元素。

30. 在 Spring 框架中如何更有效地使用 JDBC? （2017-12-05-gxb)

使用 SpringJDBC 框架，资源管理和错误处理的代价都会被减轻。所以开发者只需写 statements 和 queries 从数据存取数据，JDBC 也可以在 Spring 框架提供的模板类的帮助下更有效地被使用，这个模板叫 JdbcTemplate 。

http://www.itheima.com Copyright© 2018 黑马程序员

JdbcTemplate 类提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据错误处理。

31. 使用 Spring 通过什么方式访问 Hibernate？（2017-12-05-gxb）

在 Spring 中有两种方式访问 Hibernate：

1）控制反转 HibernateTemplate 和 Callback。

2）继承 HibernateDAOSupport 提供一个 AOP 拦截器。

32. Spring 支持的 ORM 框架有哪些？（2017-12-05-gxb）

Spring 支持以下 ORM：

Hibernate、iBatis、JPA (Java Persistence API)、TopLink、JDO (Java Data Objects)、OJB

33. 简单解释一下 spring 的 AOP（2017-12-05-gxb）

AOP（Aspect Oriented Programming），即面向切面编程，可以说是 OOP（Object Oriented Programming，面向对象编程）的补充和完善。OOP 引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过 OOP 允许开发者定义纵向的关系，但并不适合定义横向的关系，例如日志功能。日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为横切（cross cutting），在 OOP 设计中，它导致了大量代码的重复，而不利于各个模块的重用。

AOP 技术恰恰相反，它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可

http://www.itheima.com Copyright© 2018 黑马程序员

维护性。

使用"横切"技术，AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事物。AOP 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。AOP 核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组 API 提供横切功能。比如，一个日志模块可以被称作日志的 AOP 切面。根据需求的不同，一个应用程序可以有若干切面。在 Spring AOP 中，切面通过带有@Aspect 注解的类实现。

34. 在 Spring AOP 中，关注点和横切关注的区别是什么？（2017-12-05-gxb）

关注点是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。横切关注点是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

35. 什么是连接点？（2017-12-05-gxb）

被拦截到的点，因为 Spring 只支持方法类型的连接点，所以在 Spring 中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器。

36. Spring 的通知是什么？有哪几种类型？（2017-12-05-gxb）

通知是个在方法执行前或执行后要做的动作，实际上是程序执行时要通过 SpringAOP 框架触发的代码段。

Spring 切面可以应用五种类型的通知：

1）before：前置通知，在一个方法执行前被调用。

2）after: 在方法执行之后调用的通知，无论方法执行是否成功。

http://www.itheima.com Copyright© 2018 黑马程序员

3）after-returning: 仅当方法成功完成后执行的通知。

4）after-throwing: 在方法抛出异常退出时执行的通知。

5）around: 在方法执行之前和之后调用的通知。

37. 什么是切点？（2017-12-05-gxb）

切入点是一个或一组连接点，通知将在这些位置执行。可以通过表达式或匹配的方式指明切入点。

38. 什么是目标对象？（2017-12-05-gxb）

被一个或者多个切面所通知的对象。它通常是一个代理对象。也指被通知（advised）对象。

39. 什么是代理？（2017-12-05-gxb）

代理是通知目标对象后创建的对象。从客户端的角度看，代理对象和目标对象是一样的。

40. 什么是织入？什么是织入应用的不同点？（2017-12-05-gxb）

织入是将切面和到其他应用类型或对象连接或创建一个被通知对象的过程。织入可以在编译时，加载时，或运行时完成。

三、Shiro

1. 简单介绍一下 Shiro 框架（2017-11-23-gxb）

Apache Shiro 是 Java 的一个安全框架。使用 shiro 可以非常容易的开发出足够好的应用，其不仅可以用在 JavaSE 环境，也可以用在 JavaEE 环境。Shiro 可以帮助我们完成：认证、授权、加密、会话管理、与 Web 集成、缓存等。

http://www.itheima.com Copyright© 2018 黑马程序员

三个核心组件：Subject, SecurityManager 和 Realms.

Subject：即“当前操作用户”。但是，在 Shiro 中，Subject 这一概念并不仅仅指人，也可以是第三方进程、后台帐户（Daemon Account）或其他类似事物。它仅仅意味着“当前跟软件交互的东西”。但考虑到大多数目的和用途，你可以把它认为是 Shiro 的“用户”概念。

Subject 代表了当前用户的安全操作，SecurityManager 则管理所有用户的安全操作。

SecurityManager：它是 Shiro 框架的核心，典型的 Facade 模式，Shiro 通过 SecurityManager 来管理内部组件实例，并通过它来提供安全管理的各种服务。

Realm： Realm 充当了 Shiro 与应用安全数据间的“桥梁”或者“连接器”。也就是说，当对用户执行认证（登录）和授权（访问控制）验证时，Shiro 会从应用配置的 Realm 中查找用户及其权限信息。

2. Shiro 主要的四个组件（2017-12-2-wzz）

1）SecurityManager

典型的 Facade，Shiro 通过它对外提供安全管理的各种服务。

2）Authenticator

对“Who are you ？”进行核实。通常涉及用户名和密码。 这个组件负责收集 principals 和 credentials，并将它们提交给应用系统。如果提交的 credentials 跟应用系统中提供的 credentials 吻合，就能够继续访问，否则需要重新提交 principals 和 credentials， 或者直接终止访问。

3）Authorizer

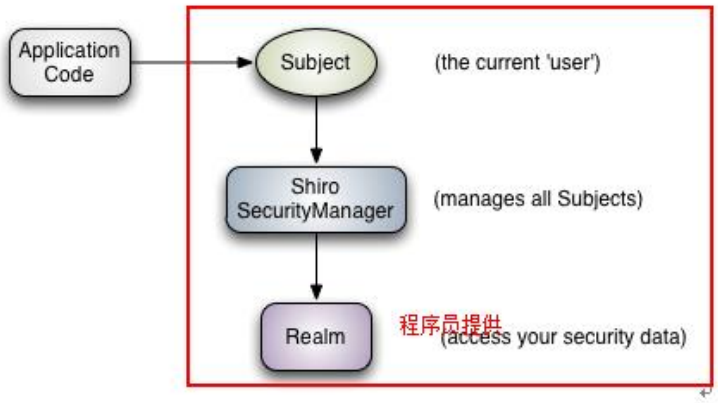
身份份验证通过后，由这个组件对登录人员进行访问控制的筛查，比如“who can do what”， 或者“who can do which actions”。 Shiro 采用“基于 Realm”的方法，即用户（又称 Subject）、 用户组、角 色和 permission 的聚合体。

http://www.itheima.com Copyright© 2018 黑马程序员

4）Session Manager

这个组件保证了异构客户端的访问，配置简单。它是基于 POJO/J2SE 的，不跟任何的客户 端或者协议绑定。

3. Shiro 运行原理（2017-12-2-wzz）



1、Application Code:应用程序代码，就是我们自己的编码，如果在程序中需要进 行权限控制，需要调用 Subject 的 API。

2、Subject:主体，代表的了当前用户。所有的 Subject 都绑定到 SecurityManager， 与 Subject 的所有交互都会委托给 SecurityManager,可以将 Subject 当成一个 门面，而真正执行者是 SecurityManager 。

3、SecurityManage:安全管理器，所有与安全有关的操作都会与 SecurityManager 交互，并且它管理所有的 Subject 。

4、Realm:域 shiro 是从 Realm 来获取安全数据（用户，角色，权限）。就是说 SecurityManager 要验证用户身份， 那么它需要从 Realm 获取相应的用户进行比较以确定用户 身份是否合法；也需要从 Realm 得到用户相应的角色/权限进行验证用户是否 能进行操作； 可以把 Realm 看成 DataSource，即安全数据源 。

http://www.itheima.com Copyright© 2018 黑马程序员

4. Shiro 的四种权限控制方式（2017-12-2-wzz）

1）url 级别权限控制

2）方法注解权限控制

3）代码级别权限控制

页面标签权限控制

详见网址：https://www.cnblogs.com/cocosili/p/7103025.html

5. 授权实现的流程（2017-12-2-wzz）

（1）、什么是粗颗粒和细颗粒权限?

对资源类型的管理称为粗颗粒度权限控制，即只控制到菜单、按钮、方法，粗粒度的例子比如：用户具有用户管理的权限，具有导出订单明细的权限。对资源实例的控制称为细颗粒度权限管理，即控制到数据级别的权限，比如：用户只允许修改本部门的员工信息，用户只允许导出自己创建的订单明细。

总结:

粗颗粒权限：针对 url 链接的控制。

细颗粒权限：针对数据级别的控制。

http://www.itheima.com Copyright© 2018 黑马程序员

比如：查询用户权限。

卫生局可以查询所有用户。

卫生室可以查询本单位的用户。

通常在 service 中编程实现。

（2）、粗颗粒和细颗粒如何授权?

对于粗颗粒度的授权可以很容易做系统架构级别的功能，即系统功能操作使用统一的粗颗粒度的权限管理。

对于细颗粒度的授权不建议做成系统架构级别的功能，因为对数据级别的控制是系统的业务需求，随着业务需求的变更业务功能变化的可能性很大，建议对数据级别的权限控制在业务层个性化开发，比如：用户只允许修改自己创建的商品信息可以在 service 接口添加校验实现，service 接口需要传入当前操作人的标识，与商品信息创建人标识对比，不一致则不允许修改商品信息。

粗颗粒权限：可以使用过虑器统一拦截 url。

细颗粒权限：在 service 中控制，在程序级别来控制，个性化 编程。

四、Mybatis

1. Mybatis 中#和$的区别？（2017-11-23-gxb）

#相当于对数据 加上 双引号，$相当于直接显示数据

1. #将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。如：order by #user_id#，如果传入的值是 111,那么解析成 sql 时的值为 order by "111"，如果传入的值是 id，则解析成的 sql 为 order by "id".

2. $将传入的数据直接显示生成在 sql 中。如：order by $user_id$，如果传入的值是 111,那么解析成 sql 时的值为 order by user_id，如果传入的值是 id，则解析成的 sql 为 order by id.

http://www.itheima.com Copyright© 2018 黑马程序员

3. #方式能够很大程度防止 sql 注入。

4.$方式无法防止 Sql 注入。

5.$方式一般用于传入数据库对象，例如传入表名.

6.一般能用#的就别用$.

2. Mybatis 的编程步骤是什么样的？（2017-12-2-wzz）

1、创建 SqlSessionFactory

2、通过 SqlSessionFactory 创建 SqlSession

3、通过 sqlsession 执行数据库操作

4、调用 session.commit()提交事务

5、调用 session.close()关闭会话

3. JDBC 编程有哪些不足之处，MyBatis 是如何解决这些问题的？（2017-12-2-wzz）

1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

   解决：在 SqlMapConfig.xml 中配置数据链接池，使用连接池管理数据库链接。

2. Sql 语句写在代码中造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变 java 代码。

   解决：将 Sql 语句配置在 XXXXmapper.xml 文件中与 java 代码分离。

3. 向 sql 语句传参数麻烦，因为 sql 语句的 where 条件不一定，可能多也可能少，占位符需要和参数一一对应。

   解决： Mybatis 自动将 java 对象映射至 sql 语句。

4. 对结果集解析麻烦，sql 变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成 pojo 对象解析比较方便。

http://www.itheima.com Copyright© 2018 黑马程序员

解决：Mybatis 自动将 sql 执行结果映射至 java 对象。

4. 使用 MyBatis 的 mapper 接口调用时有哪些要求？（2017-12-2-wzz）

1. Mapper 接口方法名和 mapper.xml 中定义的每个 sql 的 id 相同

2. Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同

3. Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同

4. Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

5. Mybatis 中一级缓存与二级缓存？（2017-12-4-lyq）

1. 一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空。

2. 二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。作用域为 namespance 是指对该 namespance 对应的配置文件中所有的 select 操作结果都缓存，这样不同线程之间就可以共用二级缓存。启动二级缓存：在 mapper 配置文件中：<cache />。

二级缓存可以设置返回的缓存对象策略：<cache readOnly="true">。当 readOnly="true"时，表示二级缓存返回给所有调用者同一个缓存对象实例，调用者可以 update 获取的缓存实例，但是这样可能会造成其他调用者出现数据不一致的情况（因为所有调用者调用的是同一个实例）。当 readOnly="false"时，返回给调用者的是二级缓存总缓存对象的拷贝，即不同调用者获取的是缓存对象不同的实例，这样调用者对各自的缓存对象的修改不会影响到其他的调用者，即是安全的，所以默认是 readOnly="false";

3. 对于缓存数据更新机制,当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

http://www.itheima.com Copyright© 2018 黑马程序员

6. MyBatis 在 insert 插入操作时返回主键 ID（2017-12-4-lyq）

数据库为 MySql 时：

```
1. <insert id="insert" parameterType="com.test.User" keyProperty="userId"
useGeneratedKeys="true" >
```

“keyProperty” 表示返回的 id 要保存到对象的那个属性中，“useGeneratedKeys” 表示主键 id 为自增长模式。MySQL 中做以上配置就 OK 了

数据库为 Oracle 时：

```
1. <insert id="insert" parameterType="com.test.User">
2.   <selectKey resultType="INTEGER" order="BEFORE" keyProperty="userId">
3.      SELECT SEQ_USER.NEXTVAL as userId from DUAL
4.   </selectKey>
5.    insert into user (user_id, user_name, modified, state)
6.    values (#{userId,jdbcType=INTEGER}, #{userName,jdbcType=VARCHAR},
#{modified,jdbcType=TIMESTAMP}, #{state,jdbcType=INTEGER})
7. </insert>
```

由于 Oracle 没有自增长一说法，只有序列这种模仿自增的形式，所以不能再使用 “useGeneratedKeys” 属性。而是使用<selectKey>将 ID 获取并赋值到对象的属性中，insert 插入操作时正常插入 id。

五、Struts2

1. 简单介绍一下 Struts2（2017-11-24-gxb）

Struts2 框架是一个按照 MVC 设计模式设计的 WEB 层框架，是在 struts 1 和 WebWork 的技术基础上进行了合并的全新的框架。其全新的 Struts 2 的体系结构与 Struts 1 的体 系结构差别巨大。Struts 2 以 WebWork 为核心，采用拦截器的机制来处理用户的请求， 这样的设计也使得业务逻辑控制器能够与 ServletAPI 完全脱离开。我们可以把 struts2 理解为一个大大的 servlet，而这个 servlet 就是 ActionServlet。struts2 在处理客户端请求时，会先读取 web.xml 配置文件，根据前端控制器将符合条件的请求 分给各个不同的 Action 处理。 在此之前，会把

http://www.itheima.com Copyright© 2018 黑马程序员

ActionServlet 会把数据封装成一个 javaBean。

Struts2 框架提供了许多的拦截器，在封装数据的过程中，我们可以对数据进行一些操 作，例如:数据校验等等。

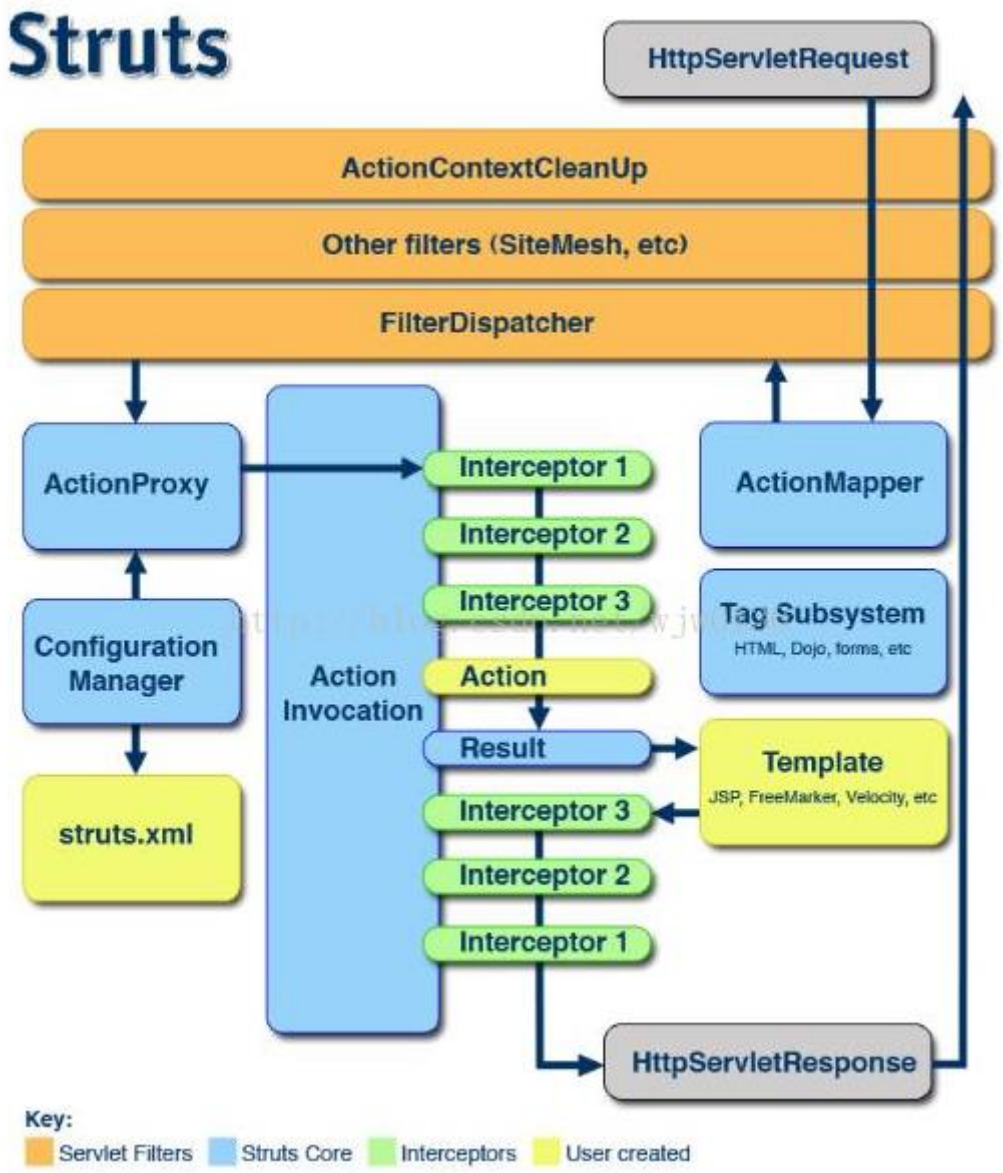
当 Action 执行完后要返回一个结果视图，这个结果视图可以跟据 struts2 的配置文件中 配置，选择转发或者重定向。

2. Struts2 的执行流程了解么？ （2017-11-24-gxb）

参考资料：http://blog.csdn.net/wjw0130/article/details/46371847

Struts2 的官方文档附带了 Struts2 的架构图。从这张图可以很好的去理解 Struts2

http://www.itheima.com Copyright© 2018 黑马程序员



关于图中的 Key：

Servlet Filters：过滤器链，客户端的所有请求都要经过 Filter 链的处理。

Struts Core：Struts2 的核心部分，但是 Struts2 已经帮我们做好了，我们不需要去做这个

Interceptors，Struts2 的拦截器。Struts2 提供了很多默认的拦截器，可以完成日常开发的绝大部分工作；而我们自定义的拦截器，用来实现实际的客户业务需要的功能。

http://www.itheima.com Copyright© 2018 黑马程序员

User Created，由开发人员创建的，包括 struts.xml、Action、Template，这些是每个使用 Struts2 来进行开发的人员都必须会的。

1. FilterDispatcher 是整个 Struts2 的调度中心，也就是 MVC 中的 C（控制中心），根据 ActionMapper 的结果来决定是否处理请求，如果 ActionMapper 指出该 URL 应该被 Struts2 处理，那么它将会执行 Action 处理，并停止过滤器链上还没有执行的过滤器。

2. ActionMapper 会判断这个请求是否应该被 Struts2 处理，如果需要 Struts2 处理，ActionMapper 会返回一个对象来描述请求对应的 ActionInvocation 的信息。

3. ActionProxy，它会创建一个 ActionInvocation 实例，位于 Action 和 xwork 之间，使得我们在将来有机会引入更多的实现方式，比如通过 WebService 来实现等。

4. ConfigurationManager 是 xwork 配置的管理中心，可以把它看做 struts.xml 这个配置文件在内存中的对应。

5. struts.xml，是开发人员必须光顾的地方。是 Stuts2 的应用配置文件，负责诸如 URL 与 Action 之间映射关系的配置、以及执行后页面跳转的 Result 配置等。

6. ActionInvocation：真正调用并执行 Action，它拥有一个 Action 实例和这个 Action 所依赖的拦截器实例。ActionInvocation 会按照指定的顺序去执行这些拦截器、Action 以及相应的 Result。

Interceptor(拦截器)：是 Struts2 的基石，类似于 JavaWeb 的 Filter，拦截器是一些无状态的类，拦截器可以自动拦截 Action，它们给开发者提供了在 Action 运行之前或 Result 运行之后来执行一些功能代码的机会。

7. Action：用来处理请求，封装数据。

3. Struts2 中 Action 配置的注意事项有哪些？（2017-11-24-gxb）

1. name 包名称，在 struts2 的配置文件中，包名不能重复，name 并不是真正包名，只是为了管理 Action

http://www.itheima.com Copyright© 2018 黑马程序员

2. namespace 和 <action>的 name 属性，决定 Action 的访问路径（以/开始 ）

3. extends 继承哪个包，通常开发中继承 struts-default 包（struts-default 包 在 struts-default.xml 中定义 ）【可以使用包中默认的拦截器和结果集】

4. 拦截器和过滤器有哪些区别？（2017-11-24-gxb）

* 拦截器是基于 java 的反射机制的，而过滤器是基于函数回调

* 拦截器不依赖与 servlet 容器，而过滤器依赖与 servlet 容器

* 拦截器只能对 action 请求起作用，而过滤器则可以对几乎所有的请求起作用

* 拦截器可以访问 action 上下文、值栈里的对象，而过滤器不能

* 在 action 的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化时被调用一 次

5. Struts2 的封装方式有哪些？（2017-11-24-gxb）

一、属性封装

1. 在 action 中设置成员变量，变量名与表单中的 name 属性值相同

2. 生成变量的 set 方法

实例：获取用户输入的用户名和密码

jsp 页面如下：

```
<form action="hello" method="post">
userName:<input type="text" name="userName"><br>
userPwd:<input type="text" name="userPwd"><br>
<input type="submit" value="提交">
</form>
```

java 代码如下：

http://www.itheima.com Copyright© 2018 黑马程序员

```
    private String userName;
    private String userPwd;
    public String execute(){
        System.out.println("userName"+"  "+userPwd);
        return NONE;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public void setUserPwd(String userPwd) {
        this.userPwd = userPwd;
    }
```

二、模型驱动（常用）

1. action 实现 ModeDriven 接口

2. 在 action 里创建实体类对象

3. 实现接口的 getModel 方法并返回所创建的对象

示例：获取用户输入的用户名和密码

jsp 页面如下：

```
<form action="hello" method="post">
userName:<input type="text" name="userName"><br>
userPwd:<input type="text" name="userPwd"><br>
<input type="submit" value="提交">
</form>
```

java 代码如下：

```
    private User user = new User();

    public User getModel() {

        return user;
    }

    public String execute(){
        System.out.println(user.getUserName()+" "+user.getUserPwd());
        return NONE;
    }
```


http://www.itheima.com Copyright© 2018 黑马程序员

需注意的是表单 name 的值应与类的属性名相同。

三、表达式封装

1. 在 action 中声明实体类

2. 生成实体类的 set 和 get 方法

3. 在表单输入项的 name 属性值里面写表达式

jsp 页面如下：

```
<form action="hello" method="post">
userName:<input type="text" name="user.userName"><br>
userPwd:<input type="text" name="user.userPwd"><br>
<input type="submit" value="提交">
</form>
```

java 代码如下：

```
    private User user ;
    public String execute(){
        System.out.println(user.getUserName()+" "+user.getUserPwd());
        return NONE;
    }
    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
```

6. 简单介绍一下 Struts2 的值栈。（2017-11-24-gxb）

值栈是对应每一个请求对象的数据存储中心。Struts2 的一个很重要的特点就是引入了值栈。之前我们通过缓存或者模型驱动在 action 和页面之间传递数据，数据混乱，并且

难以管理，缓存还有时间和数量限制，使用起来非常的困难。值栈的引入解决了这个问题，它可以统一管理页面和 action 之间的数据，供 action、result、interceptor 等使用。我们大多数情

http://www.itheima.com Copyright© 2018 黑马程序员

况下不需要考虑值栈在哪里，里面有什么，只需要去获取自己需要的数据就可以了，大大的降低了开发人员的工作量和逻辑复杂性。

参考资料：https://www.cnblogs.com/hlhdidi/p/6185836.html

7. SpringMVC 和 Struts2 的区别？（2017-11-23-gxb）

1、Struts2 是类级别的拦截， 一个类对应一个 request 上下文，SpringMVC 是方法级别的拦截，一个方法对应一个 request 上下文，而方法同时又跟一个 url 对应,所以说从架构本身上 SpringMVC 就容易实现 restful url,而 struts2 的架构实现起来要费劲，因为 Struts2 中 Action 的一个方法可以对应一个 url，而其类属性却被所有方法共享，这也就无法用注解或其他方式标识其所属方法了。

2、由上边原因，SpringMVC 的方法之间基本上独立的，独享 request response 数据，请求数据通过参数获取，处理结果通过 ModelMap 交回给框架，方法之间不共享变量，而 Struts2 搞的就比较乱，虽然方法之间也是独立的，但其所有 Action 变量是共享的，这不会影响程序运行，却给我们编码 读程序时带来麻烦，每次来了请求就创建一个 Action，一个 Action 对象对应一个 request 上下文。

3、由于 Struts2 需要针对每个 request 进行封装，把 request，session 等 servlet 生命周期的变量封装成一个一

http://www.itheima.com Copyright© 2018 黑马程序员

个 Map，供给每个 Action 使用，并保证线程安全，所以在原则上，是比较耗费内存的。

4、 拦截器实现机制上，Struts2 有以自己的 interceptor 机制，SpringMVC 用的是独立的 AOP 方式，这样导致 Struts2 的配置文件量还是比 SpringMVC 大。

5、SpringMVC 的入口是 servlet，而 Struts2 是 filter（这里要指出，filter 和 servlet 是不同的。），这就导致了二者的机制不同，这里就牵涉到 servlet 和 filter 的区别了。

6、SpringMVC 集成了 Ajax，使用非常方便，只需一个注解@ResponseBody 就可以实现，然后直接返回响应文本即可，而 Struts2 拦截器集成了 Ajax，在 Action 中处理时一般必须安装插件或者自己写代码集成进去，使用起来也相对不方便。

7、SpringMVC 验证支持 JSR303，处理起来相对更加灵活方便，而 Struts2 验证比较繁琐，感觉太烦乱。

8、Spring MVC 和 Spring 是无缝的。从这个项目的管理和安全上也比 Struts2 高（当然 Struts2 也可以通过不同的目录结构和相关配置做到 SpringMVC 一样的效果，但是需要 xml 配置的地方不少）。

9、 设计思想上，Struts2 更加符合 OOP 的编程思想， SpringMVC 就比较谨慎，在 servlet 上扩展。

10、SpringMVC 开发效率和性能高于 Struts2。

11、SpringMVC 可以认为已经 100%零配置。

8. Struts2 中的 # 和 % 分别是做什么的？（2017-11-30-wzz）

（1）使用#获取 context 里面数据

```
<s:iterator value = “list” var=”user”>
<s:property value = “#user.username”>
</s:iterator>
```

（2）向 request 域放值(获取 context 里面数据，写 ognl 时候，首先添加符号#context 的 key 名称.域对象名称)

（3）在页面中使用 ognl 获取

```
<s:property value = “#request.req”>
```


http://www.itheima.com Copyright© 2018 黑马程序员

（4）%在 struts2 标签中表单标签

在 struts2 标签里面使用 ognl 表达式，如果直接在 struts2 表单标签里面使用 ognl 表达式不识别，只有%之后才会识别。

```
<s:textfield name=”username” value=”%{#request.req}”>
```

9. Struts2 中有哪些常用结果类型？（2017-12-1-lyq）

1）dispatcher ： 默认的请求转发的结果类型，Action 转发给 JSP

2） chain ： Action 转发到另一个 Action （同一次请求）

3） redirect ： 重定向，重定向到一个路径信息，路径信息没有限制（不在一个请求中），Action 重定向到 JSP

4） redirectAction ： Action 重定向到另一个 Action

5）stream ： 将原始数据作为流传递回浏览器端，该结果类型对下载的内容和图片非常有用。

6）freemarker ： 呈现 freemarker 模板。

7）plaintext ： 返回普通文本内容。

六、Hibernate

1. 简述一下 hibernate 的开发流程（2017-11-24-gxb）

第一步:加载 hibernate 的配置文件，读取配置文件的参数(jdbc 连接参数，数据 库方言，hbm 表与对象关系映射文件)

第二步:创建 SessionFactory 会话工厂(内部有连接池)

第三步:打开 session 获取连接，构造 session 对象(一次会话维持一个数据连接， 也是一级缓存)

第四步:开启事务

第五步:进行操作

http://www.itheima.com Copyright© 2018 黑马程序员

第六步:提交事务

第七步:关闭 session(会话)将连接释放

第八步:关闭连接池

2. hibernate 中对象的三种状态（2017-11-24-gxb）

瞬时态(临时态、自由态):不存在持久化标识 OID，尚未与 Hibernate Session 关联对象， 被认为处于瞬时态，失去引用将被 JVM 回收

持久态:存在持久化标识 OID，与当前 session 有关联，并且相关联的 session 没有关闭 ，并且事务未提交

脱管态(离线态、游离态):存在持久化标识 OID，但没有与当前 session 关联，脱管状态 改变 hibernate 不能检测到

3. hibernate 的缓存机制。（2017-11-24-gxb）

Hibernate 缓存分为两层：Hibernate 的一级缓存和 Hibernate 二级缓存。

1.Hibernate 一级缓存（Session 的缓存）：

（1）Session 实现了第一级 Cache，属于事务级数据缓冲。一旦事务结束，缓存随之失效。一个 Session 的生命周期对应一个数据库事务或一个程序事务。

（2）Session-Cache 总是被打开并且不能被关闭的。

（3）Session-Cache 保证一个 Session 中两次请求同一个对象时，取得的对象是同一个 Java 实例，有时它可以避免不必要的数据冲突。

a.在对于同一个对象进行循环引用时，不至于产生堆栈溢出。

b.当数据库事务结束时，对于同一数据表行，不会产生数据冲突。因为对于数据库中的一行，最多有一个对象来表

http://www.itheima.com Copyright© 2018 黑马程序员

示它。

c.一个事务中可能会有很多个处理单元，在每一个处理单元中做的操作都会立即被其他的数据单元得知。

2.Hibernate 二级缓存（SessionFactory 的缓存）：

（1）二级缓存是 SessionFactory 范围内的缓存，所有的 Session 共享同一个二级缓存。在二级缓存中保存持久化实例的散装形式的数据。

（2）持久化不同的数据需要不同的 Cache 策略，比如一些因素将影响 Cache 策略的选择：数据的读/写比例、数据表是否能被其他的应用程序所访问等。

（3）设置 Hibernate 二级缓存需要分两步：首先，确认使用什么数据并发策略。然后，配置缓存过期时间并设置 Cache 提供器。

4. Hibernate 的查询方式有哪些？（2017-11-24-gxb）

Hibernate 的查询方式常见的主要分为三种: HQL, QBC（命名查询），以及使用原生 SQL 查询（SqlQuery）

参考资料：http://blog.csdn.net/u010963948/article/details/16818043

Copyright© 2018 黑马程序员

5. Hibernate 和 Mybatis 的区别？（2017-11-23-gxb）

两者相同点：

1）Hibernate 与 MyBatis 都可以是通过 SessionFactoryBuider 由 XML 配置文件生成 SessionFactory，然后由 SessionFactory 生成 Session，最后由 Session 来开启执行事务和 SQL 语句。其中 SessionFactoryBuider，SessionFactory，Session 的生命周期都是差不多的。

2）Hibernate 和 MyBatis 都支持 JDBC 和 JTA 事务处理。

Mybatis 优势：

1）MyBatis 可以进行更为细致的 SQL 优化，可以减少查询字段。

2）MyBatis 容易掌握，而 Hibernate 门槛较高。

Hibernate 优势：

1）Hibernate 的 DAO 层开发比 MyBatis 简单，Mybatis 需要维护 SQL 和结果映射。

2）Hibernate 对对象的维护和缓存要比 MyBatis 好，对增删改查的对象的维护要方便。

3）Hibernate 数据库移植性很好，MyBatis 的数据库移植性不好，不同的数据库需要写不同 SQL。

4）Hibernate 有更好的二级缓存机制，可以使用第三方缓存。MyBatis 本身提供的缓存机制不佳。

6. Hibernate 和 JDBC 优缺点对比（2017-11-29-wzz）

相同点：

1）两者都是 java 数据库操作的中间件、

2）两者对数据库进行直接操作的对象都是线程不安全的，都需要及时关闭。

3）两者都可对数据库的更新操作进行显式的事务处理。

不同点：

http://www.itheima.com Copyright© 2018 黑马程序员

JDBC 是 SUN 公司提供一套操作数据库的规范，使用 java 代码操作数据库。Hibernate 是一个基于 jdbc 的主流持久化框架，对 JDBC 访问数据库的代码做了封装。

使用的 SQL 语言不同：JDBC 使用的是基于关系型数据库的标准 SQL 语言，Hibernate 使用的是 HQL(Hibernate query language)语言。

操作的对象不同：JDBC 操作的是数据，将数据通过 SQL 语句直接传送到数据库中执行，Hibernate 操作的是持久化对象，由底层持久化对象的数据更新到数据库中。

数据状态不同：JDBC 操作的数据是“瞬时”的，变量的值无法与数据库中的值保持一致，而 Hibernate 操作的数据是可持久的，即持久化对象的数据属性的值是可以跟数据库中的值保持一致的。

7. 关于 Hibernate 的 orm 思想你了解多少？ (2017-11-29-wzz)

ORM 指的是对象关系型映射(Object RelationShip Mapping)，指的就是我们通过创建实体类对象和数据库中的表关系进行一一对应，来实现通过操作实体类对象来更改数据库里边的数据信息。这里边起到关键作用的是通过 Hibernate 的映射文件+Hibernate 的核心配置文件。

详细内容请见：http://blog.csdn.net/wanghuan203/article/details/7566518

 Copyright© 2018 黑马程序员

8. get 和 load 的区别？（2017-11-30-wzz）

（1）get 是立即加载，load 是延时加载。

（2）get 会先查一级缓存，再查二级缓存，然后查数据库;load 会先查一级缓存，如果没有找到，就创建代理对象，等需要的时候去查询二级缓存和数据库。(这里就体现 load 的延迟加载的特性。)

（3）get 如果没有找到会返回 null，load 如果没有找到会抛出异常。

（4）当我们使用 session.load()方法来加载一个对象时，此时并不会发出 sql 语句，当前得到的这个对象其实是一个代理对象，这个代理对象只保存了实体对象的 id 值，只有当我们要使用这个对象，得到其它属性时，这个时候才会发出 sql 语句，从数据库中去查询我们的对象；相对于 load 的延迟加载方式，get 就直接的多，当我们使用 session.get()方法来得到一个对象时，不管我们使不使用这个对象，此时都会发出 sql 语句去从数据库中查询出来。

9. 如何进行 Hibernate 的优化？（2017-11-30-wzz）

（1）数据库设计调整。

（2）HQL 优化。

（3）API 的正确使用(如根据不同的业务类型选用不同的集合及查询 API)。

（4）主配置参数(日志，查询缓存，fetch_size, batch_size 等)。

（5）映射文件优化(ID 生成策略，二级缓存，延迟加载，关联优化)。

（6）一级缓存的管理。

（7）针对二级缓存，还有许多特有的策略。

（8）事务控制策略。

详情解释请见：https://www.cnblogs.com/xhj123/p/6106088.html

http://www.itheima.com Copyright© 2018 黑马程序员

10. 什么是 Hibernate 延迟加载?（2017-12-1-lyq）

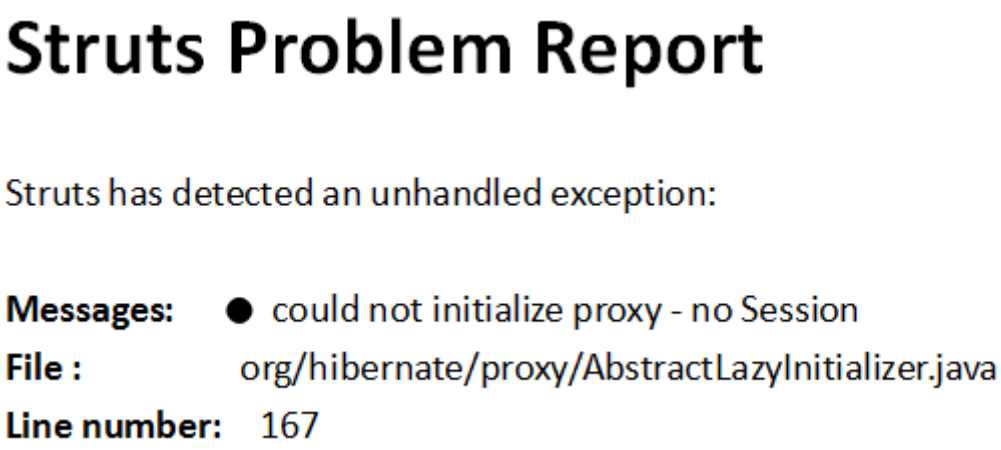
延迟加载机制是为了避免一些无谓的性能开销而提出来的，所谓延迟加载就是当在真正需要数据的时候，才真正执行数据加载操作。在 Hibernate 中提供了对实体对象的延迟加载以及对集合的延迟加载，另外在 Hibernate3 中还提供了对属性的延迟加载。

延迟加载的过程：通过代理（Proxy）机制来实现延迟加载。Hibernate 从数据库获取某一个对象数据时、获取某一个对象的集合属性值时，或获取某一个对象所关联的另一个对象时，由于没有使用该对象的数据（除标识符外），Hibernate 并不从数据库加载真正的数据，而只是为该对象创建一个代理对象来代表这个对象，这个对象上的所有属性都为默认值；只有在真正需要使用该对象的数据时才创建这个真正的对象，真正从数据库中加载它的数据。

11. No Session 问题原理及解决方法? （2017-12-4-lyq）

Nosession 问题报错如下：

http://www.itheima.com Copyright© 2018 黑马程序员



根据字面上的意思，是指代理不能被初始化，session 已经关闭。

Nosession 问题产生的原因：

当执行 Session 的 load()方法时，Hibernate 不会立即执行查询所查询对象关联的对象（在此我们统称被关联的对象类为 A 类），仅仅返回 A 类的代理类的实例，这个代理类具由以下特征：

（1）由 Hibernate 在运行时动态生成，它扩展了 A 类，因此它继承了 A 类的所有属性和方法，但它的实现对于应用程序是透明的。

（2）当 Hibernate 创建 A 类代理类实例时，仅仅初始化了它的 OID 属性，其他属性都为 null，因此这个代理类实例占用的内存很少。

（3）当应用程序第一次访问 A 代理类实例时（例如调用 a..getXXX()或 a.setXXX()方法），Hibernate 会初始化代理类实例，在初始化过程中执行 select 语句，真正从数据库中加载 A 对象的所有数据。但有个例外，那就是当应用程序访问 A 代理类实例的 getId()方法时，Hibernate 不会初始化代理类实例，因为在创建代理类实例时 OID 就存在了，不必到数据库中去查询。

提示：Hibernate 采用 CGLIB 工具来生成持久化类的代理类。CGLIB 是一个功能强大的 Java 字节码生成工具，它能够在程序运行时动态生成扩展 Java 类或者实现 Java 接口的代理类。

因为 Hibernate 中如果采用 load 加载的话(默认的是延迟加载)，也就是 lazy=true 操作，因此，当调用完 load

http://www.itheima.com Copyright© 2018 黑马程序员

后，session 即可关闭。因为我们的 session 只是放置到了 Dao 层，表现层根本获取不到，所以在表现层调用的时候，session 已经关闭，报错。

12. Spring 的两种代理 JDK 和 CGLIB 的区别浅谈（2017-12-4-lyq）

Java 动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用 InvokeHandler 来处理。而 cglib 动态代理是利用 asm 开源包，对代理对象类的 class 文件加载进来，通过修改其字节码生成子类来处理。

1、如果目标对象实现了接口，默认情况下会采用 JDK 的动态代理实现 AOP

2、如果目标对象实现了接口，可以强制使用 CGLIB 实现 AOP

3、如果目标对象没有实现了接口，必须采用 CGLIB 库，spring 会自动在 JDK 动态代理和 CGLIB 之间转换

参考资料：http://blog.csdn.net/tanga842428/article/details/52716875

13.叙述 Session 的缓存的作用（2017-12-9-lwl）

（1）减少访问数据库的频率。应用程序从内存中读取持久化对象的速度显然比到数据库中查询数据的速度快多了，因此 Session 的缓存可以提高数据访问的性能。

http://www.itheima.com Copyright© 2018 黑马程序员

（2）保证缓存中的对象与数据库中的相关记录保持同步。当缓存中持久化对象的状态发生了变换，Session 并不会立即执行相关的 SQL 语句，这使得 Session 能够把几条相关的 SQL 语句合并为一条 SQL 语句，以便减少访问数据库的次数，从而提高应用程序的性能。

14.Session 的清理和清空有什么区别？ （2017-12-10-lwl）

Session 清理缓存是指按照缓存中对象的状态的变化来同步更新数据库；清空是 Session 的关闭；

15.请简述 Session 的特点有哪些？ （2017-12-10-lwl）

(1)不是线程安全的，因此在设计软件架构时，应该避免多个线程共享同一个 Session 实例。

(2)Session 实例是轻量级的，所谓轻量级是指它的创建和销毁不需要消耗太多的资源。这意味着在程序中可以经常创建或销毁 Session 对象，例如为每个客户请求分配单独的 Session 实例，或者为每个工作单元分配单独的 Session 实例。

(3)在 Session 中，每个数据库操作都是在一个事务(transaction)中进行的，这样就可以隔离开不同的操作（甚至包括只读操作）。

16.比较 Hibernate 三种检索策略的优缺点（2017-12-10-lwl）

1、立即检索

优点：对应用程序完全透明，不管对象处于持久化状态，还是游离状态，应用程序都可以方便的从一个对象导航到与它关联的对象；

缺点：1.select 语句太多；2.可能会加载应用程序不需要访问的对象白白浪费许多内存空间；

2、延迟检索

优点：由应用程序决定需要加载哪些对象，可以避免可执行多余的 select 语句，以及避免加载应用程序不需要访

http://www.itheima.com Copyright© 2018 黑马程序员

问的对象。因此能提高检索性能，并且能节省内存空间；

缺点：应用程序如果希望访问游离状态代理类实例，必须保证他在持久化状态时已经被初始化；

3、 迫切左外连接检索

优点：1、对应用程序完全透明，不管对象处于持久化状态，还是游离状态，应用程序都可以方便地冲一个对象导航到与它关联的对象。2、使用了外连接，select 语句数目少；

缺点：1、可能会加载应用程序不需要访问的对象，白白浪费许多内存空间；2、复杂的数据库表连接也会影响检索性能；

七、Quartz 定时任务

1.什么是 Quartz 框架（2017-12-2-wzz）

Quartz 是一个开源的作业调度框架，它完全由 Java 写成，并设计用于 J2SE 和 J2EE 应用中。它提供了巨大的灵活性而不牺牲简单性。你能够用它来为执行一个作业而创建简单的或复杂的调度。

2.配置文件 applicationContext_job.xml 各个属性作用（2017-12-2-wzz）

（1）、Job：表示一个任务（工作），要执行的具体内容。

（2）、JobDetail：表示一个具体的可执行的调度程序，Job 是这个可执行程调度程序所要执行的内容，另外 JobDetail 还包含了这个任务调度的方案和策略。

（3）、Trigger：代表一个调度参数的配置，什么时候去调。

（4）、Scheduler：代表一个调度容器，一个调度容器中可以注册多个 JobDetail 和 Trigger。当 Trigger 与 JobDetail 组合，就可以被 Scheduler 容器调度了。

http://www.itheima.com Copyright© 2018 黑马程序员

3.Cron 表达式详解（2017-12-2-wzz)

Cron 表达式是一个字符串，字符串以 5 或 6 个空格隔开，分为 6 或 7 个域，每一个域代表一个含义。

域:

Seconds（秒）：可出现", - * /"四个字符，有效范围为 0-59 的整数。

Minutes（分钟）：可出现", - * /"四个字符，有效范围为 0-59 的整数 。

Hours（小时）:可出现", - * /"四个字符，有效范围为 0-23 的整数 。

DayofMonth（日 of 月）:可出现", - * / ? L W C"八个字符，有效范围为 0-31 的整数。

Month（月）:可出现", - * /"四个字符，有效范围为 1-12 的整数。

DayofWeek（日 of 星期）:可出现", - * / ? L C #"四个字符，有效范围为 1-7 的整数 1 表示星期天，2 表示星期一， 依次类推。

Year（年）:可出现", - * /"四个字符，有效范围为 1970-2099 年。

4.如何监控 Quartz 的 job 执行状态: 运行中，暂停中，等待中? （2017-12-2-wzz)

通过往表（新建一个操作日志表）里插入日志的形式:

1）运行中：通过 JobListener 监听器来实现运行时更改表信息。

2）暂停中：调用 scheduler.pauseTrigger()方法时，更改表中 job 信息。

3）等待中：新添加的 job 默认给其等待中的状态，也是更改表中的 job 信息 但是上面这种形式的麻烦之处是得频繁的往表里插入数据。

http://www.itheima.com Copyright© 2018 黑马程序员

第八章 最新技术

一、Redis

1. Redis 的特点？（2017-11-25-wzz）

Redis 是由意大利人 Salvatore Sanfilippo（网名：antirez）开发的一款内存高速缓存数据库。Redis 全称为：Remote Dictionary Server（远程数据服务），该软件使用 C 语言编写，典型的 NoSQL 数据库服务器，Redis 是一个 key-value 存储系统，它支持丰富的数据类型，如：string、list、set、zset(sorted set)、hash。

Redis 本质上是一个 Key-Value 类型的内存数据库，很像 memcached，整个 数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据 flush 到硬盘上进行保存。因为是纯内存操作，Redis 的性能非常出色，每秒可以处理超过 10 万次读写操作，是已知性能最快的 Key-Value DB。

Redis 的出色之处不仅仅是性能，Redis 最大的魅力是支持保存多种数据结构，此外单 个 value 的最大限制是 1GB，不像 memcached 只能保存 1MB 的数据，另外 Redis 也可以对存入的 Key-Value 设置 expire 时间。

Redis 的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。

2. 为什么 redis 需要把所有数据放到内存中？（2017-11-25-wzz）

Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以 redis 具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘 I/O 速度为严重影响 redis 的性能。在内存越来越便宜的今天，redis 将会越来越受欢迎。如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

http://www.itheima.com Copyright© 2018 黑马程序员

3. Redis 常见的性能问题都有哪些？如何解决？（2017-11-25-wzz）

（1）、Master 写内存快照，save 命令调度 rdbSave 函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以 Master 最好不要写内存快照。

（2）、Master AOF 持久化，如果不重写 AOF 文件，这个持久化方式对性能的影响是最小的，但是 AOF 文件会不断增大，AOF 文件过大会影响 Master 重启的恢复速度。Master 最好不要做任何持久化工作，包括内存快照和 AOF 日志文件，特别是不要启用内存快照做持久化,如果数据比较关键，某个 Slave 开启 AOF 备份数据，策略为每秒同步一次。

（3）、Master 调用 BGREWRITEAOF 重写 AOF 文件，AOF 在重写的时候会占大量的 CPU 和内存资源，导致服务 load 过高，出现短暂服务暂停现象。

（4）、Redis 主从复制的性能问题，为了主从复制的速度和连接的稳定性，Slave 和 Master 最好在同一个局域网内

4. Redis 最适合的场景有哪些？（2017-11-25-wzz）

（1）、会话缓存（Session Cache）

（2）、全页缓存（FPC）

（3）、队列

（4）、排行榜/计数器

（5）、发布/订阅

5. Memcache 与 Redis 的区别都有哪些？（2017-11-25-wzz）

（1）、存储方式不同，Memcache 是把数据全部存在内存中，数据不能超过内存的大小，断电后数据库会挂掉。

http://www.itheima.com Copyright© 2018 黑马程序员

Redis 有部分存在硬盘上，这样能保证数据的持久性。

（2）、数据支持的类型不同 memcahe 对数据类型支持相对简单，redis 有复杂的数据类型。

（3）、使用底层模型不同 它们之间底层实现方式 以及与客户端之间通信的应用协议不一样。Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

（4）、支持的 value 大小不一样 redis 最大可以达到 1GB，而 memcache 只有 1MB。

6. Redis 用过 RedisNX 吗？Redis 有哪几种数据结构？（2017-11-14-lyq）

反正我是不知道 redisnx 是什么，度娘也不清楚，如果面试中问道自己没有接触过或者没有听过的技术可以直接大胆的告诉他，没有接触过，或者没有听过。

Redis 的数据结构有五种，分别是：

String——字符串

String 数据结构是简单的 key-value 类型，value 不仅可以是 String，也可以是数字（当数字类型用 Long 可以表示的时候 encoding 就是整型，其他都存储在 sdshdr 当做字符串）。

Hash——字典

在 Memcached 中，我们经常将一些结构化的信息打包成 hashmap，在客户端序列化后存储为一个字符串的值（一般是 JSON 格式），比如用户的昵称、年龄、性别、积分等。

List——列表

List 说白了就是链表（redis 使用双端链表实现的 List），相信学过数据结构知识的人都应该能理解其结构。

Set——集合

Set 就是一个集合，集合的概念就是一堆不重复值的组合。利用 Redis 提供的 Set 数据结构，可以存储一些集合性的数据。

http://www.itheima.com Copyright© 2018 黑马程序员

Sorted Set——有序集合

和 Sets 相比，Sorted Sets 是将 Set 中的元素增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列，

1. 带有权重的元素，比如一个游戏的用户得分排行榜

2.比较复杂的数据结构，一般用到的场景不算太多

7. Redis 的优缺点（2017-11-22-lyq）

优点：

a) 性能极高 – Redis 能支持超过 100K+ 每秒的读写频率。

b) 丰富的数据类型 – Redis 支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

c) 原子 – Redis 的所有操作都是原子性的，同时 Redis 还支持对几个操作全并后的原子性执行。

attention 原子性定义：例如，A 想要从自己的帐户中转 1000 块钱到 B 的帐户里。那个从 A 开始转帐，到转帐结束的这一个过程，称之为一个事务。如果在 A 的帐户已经减去了 1000 块钱的时候，忽然发生了意外，比如停电什么的，导致转帐事务意外终止了，而此时 B 的帐户里还没有增加 1000 块钱。那么，我们称这个操作失败了，要进行回滚。回滚就是回到事务开始之前的状态，也就是回到 A 的帐户还没减 1000 块的状态，B 的帐户的原来的状态。此时 A 的帐户仍然有 3000 块，B 的帐户仍然有 2000 块。我们把这种要么一起成功（A 帐户成功减少 1000，同时 B 帐户成功增加 1000），要么一起失败（A 帐户回到原来状态，B 帐户也回到原来状态）的操作叫原子性操作。如果把一个事务可看作是一个程序,它要么完整的被执行,要么完全不执行，这种特性就叫原子性。

·d）丰富的特性 – Redis 还支持 publish/subscribe, 通知, key 过期等等特性。

缺点：

a）. 由于是内存数据库，所以，单台机器，存储的数据量，跟机器本身的内存大小。虽然 redis 本身有 key 过

http://www.itheima.com Copyright© 2018 黑马程序员

期策略，但是还是需要提前预估和节约内存。如果内存增长过快，需要定期删除数据。

b). 如果进行完整重同步，由于需要生成 rdb 文件，并进行传输，会占用主机的 CPU，并会消耗现网的带宽。不过 redis2.8 版本，已经有部分重同步的功能，但是还是有可能有完整重同步的。比如，新上线的备机。

c). 修改配置文件，进行重启，将硬盘中的数据加载进内存，时间比较久。在这个过程中，redis 不能提供服务。

8. Redis 的持久化（2017-11-23-lyq）

RDB 持久化：该机制可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）。

AOF 持久化：记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。 Redis 还可以在后台对 AOF 文件进行重写（rewrite），使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小

无持久化：让数据只在服务器运行时存在。

同时应用 AOF 和 RDB：当 Redis 重启时， 它会优先使用 AOF 文件来还原数据集， 因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整。

RDB 的优缺点：

优点：RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。RDB 非常适用于灾难恢复（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心，或者亚马逊 S3 中。RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。RDB 在

http://www.itheima.com Copyright© 2018 黑马程序员

恢复大数据集时的速度比 AOF 的恢复速度要快。

缺点：如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为 RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端；如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。

AOF 的优缺点。

优点：

1、使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync ，每秒钟一次 fsync ，或者每次执行写入命令时 fsync 。 AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（ fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 seek ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， redis-check-aof 工具也可以轻易地修复这种问题。

2、Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 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。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。

缺点：

对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。根据所使用的 fsync 策略，AOF 的速

http://www.itheima.com Copyright© 2018 黑马程序员

度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。

AOF 在过去曾经发生过这样的 bug ： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。 （举个例子，阻塞命令 BRPOPLPUSH 就曾经引起过这样的 bug 。） 测试套件里为这种情况添加了测试： 它们会自动生成随机的、复杂的数据集， 并通过重新载入这些数据来确保一切正常。 虽然这种 bug 在 AOF 文件中并不常见， 但是对比来说， RDB 几乎是不可能出现这种 bug 的。

二、消息队列 ActiveMQ

1. 如何使用 ActiveMQ 解决分布式事务？（2017-11-21-gxb）

在互联网应用中，基本都会有用户注册的功能。在注册的同时，我们会做出如下操作：

1. 收集用户录入信息，保存到数据库

2. 向用户的手机或邮箱发送验证码

等等...

如果是传统的集中式架构，实现这个功能非常简单：开启一个本地事务，往本地数据库中插入一条用户数据，发送验证码，提交事物。

但是在分布式架构中，用户和发送验证码是两个独立的服务，它们都有各自的数据库，那么就不能通过本地事物保证操作的原子性。这时我们就需要用到 ActiveMQ（消息队列）来为我们实现这个需求。

在用户进行注册操作的时候，我们为该操作创建一条消息，当用户信息保存成功时，把这条消息发送到消息队列。验证码系统会监听消息，一旦接受到消息，就会给该用户发送验证码。

http://www.itheima.com Copyright© 2018 黑马程序员

问题：

1.如何防止消息重复发送?

解决方法很简单：增加消息状态表。通俗来说就是一个账本，用来记录消息的处理状态，每次处理消息之前，都去状态表中查询一次，如果已经有相同的消息存在，那么不处理，可以防止重复发送。

2. 了解哪些消息队列？（2017-11-24-gxb）

ActiveMQ、RabbitMQ、kafka。

RabbitMQ：

RabbitMQ 是使用 Erlang 编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正因如此，它非常重量级，更适合于企业级的开发。同时实现了 Broker 构架，这意味着消息在发送给客户端时先在中心队列排队。对路由，负载均衡或者数据持久化都有很好的支持。

ActiveMQ：

ActiveMQ 是 Apache 下的一个子项目。 类似于 ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于 RabbitMQ，它少量代码就可以高效地实现高级应用场景。

Kafka/Jafka：

Kafka 是 Apache 下的一个子项目，是一个高性能跨语言分布式发布/订阅消息队列系统，而 Jafka 是在 Kafka 之上孵化而来的，即 Kafka 的一个升级版。具有以下特性：快速持久化，可以在 O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到 10W/s 的吞吐速率；完全的分布式系统，Broker、Producer、Consumer 都原生自动支持分布式，自动实现负载均衡；支持 Hadoop 数据并行加载，对于像 Hadoop 的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka 通过 Hadoop 的并行加载机制统一了在线和离线的消息处理。Apache Kafka 相对于 ActiveMQ 是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工

http://www.itheima.com Copyright© 2018 黑马程序员

作良好的分布式系统。

MQ 选型对比图

A	B	C	D	E
特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
PRODUCER-COMSUMER	支持	支持	支持	支持
PUBLISH-SUBSCRIBE	支持	支持	支持	支持
REQUEST-REPLY	支持	支持		
API完备性	高	高	高	高
多语言支持	支持，JAVA优先	语言无关	只支持JAVA	支持，java优先
单机吞吐量	万级	万级	万级	十万级
消息延迟		微秒级	毫秒级	毫秒级
可用性	高（主从）	高（主从）	非常高（分布式）	非常高（分布式）
消息丢失	低	低	理论上不会丢失	理论上不会丢失
消息重复		可控制		理论上会有重复
文档的完备性	高	高	高	高
提供快速入门	有	有	有	有
首次部署难度		低		中
社区活跃度	高	高	中	高
商业支持	无	无	阿里云	无
成熟度	成熟	成熟	比较成熟	成熟日志领域
特点	功能齐全，被大 量开源项目使用	由于Erlang 语言的并发能 力，性能很好	各个环节分 布式扩展设计，主从 HA；支持上万个队列；多 种消费模式；性能很好	
支持协议	OpenWire、 STOMP、 REST、XMPP、AMQP	AMQP	自己定义的一 套(社区提供 JMS--不成熟)	
持久化	内存、文件、数据库	内存、文件	磁盘文件	
事务	支持	支持	支持	
负载均衡	支持	支持	支持	
管理界面	一般	好	有web console实现	
部署方式	独立、嵌入	独立	独立	
评价	优点：成熟的产品，已经在很多公 司得到应用（非大规模场景）。有 较多的文档。各种协议支持较好， 有多重语言的成熟的客户端； 缺点：根据其他 用户反馈，会出莫 名其妙的问题，切会丢 失消息。其 重心放到 activemq6.0 产品—apollo 上去了，目前社区不活跃，且对5.x 维护较少； Activemq 不适合用于上 千个队列的应用场景。	优点：由于erlang语言的特 性，mq性能较好；管理界面 较丰富，在互联网公司也有 较大规模的应用；支持amqp 系诶，有多中语言且支持 amqp的客户端可用； 缺点：erlang语言难度较 大。集群不支持动态扩展。	优点：模型简单，接口易用（JMS的接口很多 场合并不太实用）。在阿里大规模应用。目前 支付宝中的余额宝等新兴产品均使用 rocketmq 。集群规模大概在50台左右，单日处理消息上 百亿；性能非常好，可以大量堆积消息在 broker中；支持多种消费，包括集群消费、广 播消费等。开发度较活跃，版本更新很快。 缺点：产品较新文档比较缺乏。没有在mq核心 中去实现JMS等接口，对已有系统而言不能兼 容。阿里内部还有一套未开源的MQAPI，这一 层API可以将上层应用和下层MQ的实现解耦 （阿里内部有多个mq的实现，如notify、 metaq1.x，metaq2.x，rocketmq等），使得下 面mq可以很方便的进行切换和升级而对应用无 任何影响，目前这一套东西未开源。	

http://blog.csdn.net/oMaverick1

3. ActiveMQ 如果消息发送失败怎么办？ （2017-11-24-gxb）

Activemq 有两种通信方式，点到点形式和发布订阅模式。

如果是点到点模式的话，如果消息发送不成功此 消息默认会保存到 activemq 服务端知道有消费者将其消费，所

以此时消息是不会丢失的。

如果是发布订阅模式的通信方式，默认情况下只通知一次，如果接收不到此消息就没有了。这种场景只适 用于对消息送达率要求不高的情况。如果要求消息必须送达不可以丢失的话，需要配置持久订阅。每个订阅端定义一个 id，在订阅是向 activemq 注册。发布消息和接收消息时需要配置发送模式为持久化。此时 如果客户端接收不到消息，消息会持久化到服务端，直到客户端正常接收后为止。

三、Dubbo

1. Dubbo 的容错机制有哪些。（2017-11-23-gxb）

Dubbo 官网提出总共有六种容错策略

1）Failover Cluster 模式

失败自动切换，当出现失败，重试其它服务器。(默认)

2）Failfast Cluster

快速失败，只发起一次调用，失败立即报错。 通常用于非幂等性的写操作，比如新增记录。

3）Failsafe Cluster

失败安全，出现异常时，直接忽略。 通常用于写入审计日志等操作。

4）Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。 通常用于消息通知操作。

5）Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks=”2”来设置最大并行数。

http://www.itheima.com Copyright© 2018 黑马程序员

6）Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。(2.1.0 开始支持）通常用于通知所有提供者更新缓存或日志等本地资源信息。

总结： 在实际应用中查询语句容错策略建议使用默认 Failover Cluster，而增删改建议使用 Failfast Cluster 或者 使用 Failover Cluster（retries=”0”） 策略 防止出现数据 重复添加等等其它问题！建议在设计接口时候把查询接口方法单独做一个接口提供查询。

2. 使用 dubbo 遇到过哪些问题？（2017-11-23-gxb）

1. 增加提供服务版本号和消费服务版本号

这个具体来说不算是一个问题,而是一种问题的解决方案,在我们的实际工作中会面临各种环境资源短缺的问题,也是很实际的问题,刚开始我们还可以提供一个服务进行相关的开发和测试,但是当有多个环境多个版本,多个任务的时候就不满足我们的需求,这时候我们可以通过给提供方增加版本的方式来区分.这样能够剩下很多的物理资源,同时为今后更换接口定义发布在线时，可不停机发布，使用版本号.

引用只会找相应版本的服务,例如:

<dubbo:serviceinterface=“com.xxx.XxxService” ref=“xxxService” version=“1.0” />

<dubbo:referenceid=“xxxService” interface=“com.xxx.XxxService” version=“1.0”/>

2. dubbo reference 注解问题

@Reference 只能在 springbean 实例对应的当前类中使用，暂时无法在父类使用；如果确实要在父类声明一个引用，可通过配置文件配置 dubbo:reference，然后在需要引用的地方跟引用 springbean 一样就可以了.

3.出现 RpcException: No provider available for remote service 异常，表示没有可用的服务提供者

1). 检查连接的注册中心是否正确

http://www.itheima.com Copyright© 2018 黑马程序员

2). 到注册中心查看相应的服务提供者是否存在

3). 检查服务提供者是否正常运行

4. 服务提供者没挂，但在注册中心里看不到

首先，确认服务提供者是否连接了正确的注册中心，不只是检查配置中的注册中心地址，而且要检查实际的网络连接。

其次，看服务提供者是否非常繁忙，比如压力测试，以至于没有 CPU 片段向注册中心发送心跳，这种情况，减小压力，将自动恢复。

3. Dubbo 的连接方式有哪些？（2017-12-1-lyq）

Dubbo 的客户端和服务端有三种连接方式，分别是：广播，直连和使用 zookeeper 注册中心。

3.1、Dubbo 广播

这种方式是 dubbo 官方入门程序所使用的连接方式，但是这种方式有很多问题。在企业开发中，不使用广播的方式。

taotao-manager 服务端配置：

```
1.     <!-- applicationContext-service.xml 文件中 -->
2.     <!-- 提供方应用信息，用于计算机依赖关系  -->
3.   <dubbo:application name="taotao-manager-service" />
4.    <!-- 使用 multicast 广播暴露服务地址  -->
5.   <dubbo:registry address="multicast://224.5.6.7:1234" />
6.    <!--  使用 dubbo 协议在 20880 协议暴露服务 -->
7.   <dubbo:protocol name="dubbo" port="20880" />
8.    <!-- 声明需要暴露的服务接口 -->
9.   <dubbo:service interface="com.taotao.manager.service.TestService" ref="testServiceImpl" />
10. </bean>
```

客户端配置 taotao-manager-web 的配置如下：

```
1.     <!--springMVC.xml 文件中 -->
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
    <!-- 配置 dubbo 服务 -->
    <dubbp:application name="taotao-manager-web" />
    <!-- 使用 multicast 广播暴露服务地址 -->
    <dubbo:registry address="multicast://224.5.6.7:1234" />
    <!-- 声明要调用的服务 timeout 是设置连接超时最长时间，如果不设置，默认超时时间为 3 秒 -->
    <dubbo:service interface="com.taotao.manager.service.TestService" id="testService"
timeout="10000000" />
</bean>
```

3.2、Dubbo 直连

这种方式在企业中一般在开发中环境中使用，但是生产环境很少使用，因为服务是直接调用，没有使用注册中心，很难对服务进行管理。Dubbo 直连，首先要取消广播，然后客户端直接到指定需要的服务的 url 获取服务即可。

服务端配置：taotao-manager 的修改如下，取消广播，注册中心地址为 N/A

```
    <!-- applicationContext-service.xml 文件中 -->
    <!-- 提供方应用信息，用于计算机依赖关系 -->
    <dubbo:application name="taotao-manager-service" />
    <!-- <dubbo:registry address="multicast://224.5.6.7:1234" /> -->
    <!-- 使用 multicast 广播暴露服务地址 -->
    <dubbo:registry address="N/A" />
    <!-- 使用 dubbo 协议在 20880 协议暴露服务 -->
    <dubbo:protocol name="dubbo" port="20880" />
    <!-- 声明需要暴露的服务接口 -->
    <dubbo:service interface="com.taotao.manager.service.TestService" ref="testServiceImpl" />
</bean>
```

客户端配置：taotao-manager-web 配置如下，取消广播，从指定的 url 中获取服务

```
    <!--springMVC.xml 文件中 -->
    <!-- 配置 dubbo 服务 -->
    <dubbp:application name="taotao-manager-web" />
    <!-- 使用 multicast 广播暴露服务地址 -->
    <!-- <dubbo:registry address="multicast://224.5.6.7:1234" /> -->
    <!-- 声明要调用的服务 timeout 是设置连接超时最长时间，如果不设置，默认超时时间为 3 秒 -->
    <dubbo:service interface="com.taotao.manager.service.TestService" id="testService"
timeout="10000000" />
</bean>
```


http://www.itheima.com Copyright© 2018 黑马程序员

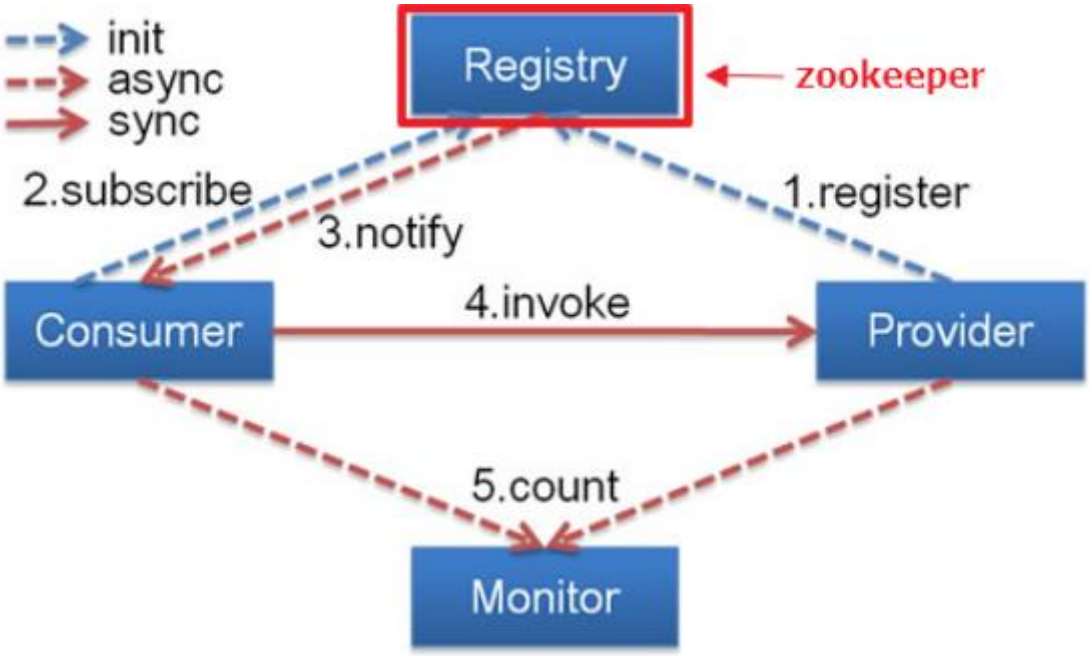
3.3、zookeeper 注册中心

Dubbo 注册中心和广播注册中心配置类似，不过需要指定注册中心类型和注册中心地址，这个时候就不是把服务信息进行广播了，而是告诉给注册中心进行管理，这个时候我们就需要有一个注册中心。

官方推荐使用 zookeeper 作为注册中心。

3.3.1、Zookeeper 介绍

zookeeper 在 dubbo 所处的位置：



1）Provider：暴露服务的服务提供方。

2）Consumer：调用远程服务的服务消费方。

3）Registry：服务注册与发现的注册中心。

4) Monitor：统计服务的调用次调和调用时间的监控中心。

5)Container：服务运行容器。

http://www.itheima.com Copyright© 2018 黑马程序员

调用关系说明：

1)服务容器负责启动，加载，运行服务提供者。

2)服务提供者在启动时，向注册中心注册自己提供的服务。

3)服务消费者在启动时，向注册中心订阅自己所需的服务。

4)注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

5)服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

6)服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

四、并发相关

1. 如何测试并发量？（2017-11-23-gxb）

可以使用 apache 提供的 ab 工具测试。

参考资料：http://blog.csdn.net/whynottrythis/article/details/46495309

http://www.itheima.com Copyright© 2018 黑马程序员

五、Nginx

1. Nginx 反向代理为什么能够提升服务器性能？ （2017-11-24-gxb）

对于后端是动态服务来说，比如 Java 和 PHP。这类服务器(如 JBoss 和 PHP-FPM)的 IO 处理能力往往不高。Nginx 有个好处是它会把 Request 在读取完整之前 buffer 住，这样交给后端的就是一个完整的 HTTP 请求，从而提高后端的效率，而不是断断续续的传递(互联网上连接速度一般比较慢)。 同样，Nginx 也可以把 response 给 buffer 住，同样也是减轻后端的压力。

2. Nginx 和 Apache 各有什么优缺点？ （2017-11-24-gxb）

nginx 相对 apache 的优点:

1)轻量级，同样起 web 服务，比 apache 占用更少的内存及资源

2)抗并发，nginx 处理请求是异步非阻塞的，而 apache 则是阻塞型的，在高并发下 nginx 能保持

3)低资源低消耗高性能

4)高度模块化的设计，编写模块相对简单

http://www.itheima.com Copyright© 2018 黑马程序员

5)社区活跃，各种高性能模块出品迅速啊

apache 相对 nginx 的优点:

1)rewrite，比 nginx 的 rewrite 强大

2)模块超多，基本想到的都可以找到

3)少 bug，nginx 的 bug 相对较多

4)超稳定,一般来说，需要性能的 web 服务，用 nginx 。 如果不需要性能只求稳定，那就 apache 吧。

3. Nginx 多进程模型是如何实现高并发的？（2017-12-5-lyq）

进程数与并发数不存在很直接的关系。这取决取 server 采用的工作方式。如果一个 server 采用一个进程负责一个 request 的方式，那么进程数就是并发数。那么显而易见的，就是会有很多进程在等待中。等什么？最多的应该是等待网络传输。

Nginx 的异步非阻塞工作方式正是利用了这点等待的时间。在需要等待的时候，这些进程就空闲出来待命了。因此表现为少数几个进程就解决了大量的并发问题。apache 是如何利用的呢，简单来说：同样的 4 个进程，如果采用一个进程负责一个 request 的方式，那么，同时进来 4 个 request 之后，每个进程就负责其中一个，直至会话关闭。期间，如果有第 5 个 request 进来了。就无法及时反应了，因为 4 个进程都没干完活呢，因此，一般有个调度进程，每当新进来了一个 request，就新开个进程来处理。nginx 不这样，每进来一个 request，会有一个 worker 进程去处理。但不是全程的处理，处理到什么程度呢？处理到可能发生阻塞的地方，比如向上游（后端）服务器转发 request，并等待请求返回。那么，这个处理的 worker 不会这么傻等着，他会在发送完请求后，注册一个事件：“如果 upstream 返回了，告诉我一声，我再接着干”。于是他就休息去了。此时，如果再有 request 进来，他就可以很快再按这种方式处理。而一旦上游服务器返回了，就会触发这个事件， worker 才会来接手，这个 request 才会接着往下走。

由于 web server 的工作性质决定了每个 request 的大部份生命都是在网络传输中，实际上花费在 server 机器

http://www.itheima.com Copyright© 2018 黑马程序员

上的时间片不多。这是几个进程就解决高并发的秘密所在。webserver 刚好属于网络 io 密集型应用，不算是计算密集型。异步，非阻塞，使用 epoll，和大量细节处的优化。也正是 nginx 之所以然的技术基石。

六、Zookeeper

1. 简单介绍一下 zookeeper 以及 zookeeper 的原理。（2017-11-24-gxb）

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务，是 Google 的 Chubby 一个开源的实现，是 Hadoop 和 Hbase 的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括:配置维护、域名服务、分布式同步、组服务等。

ZooKeeper 的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、 功能稳定的系统提供给用户。

ZooKeeper 包含一个简单的原语集， 提供 Java 和 C 的接口。

ZooKeeper 代码版本中，提供了分布式独享锁、选举、队列的接口，代码在 zookeeper-3.4.3\src\recipes。其中分布锁和队列有 Java 和 C 两个版本，选举只有 Java 版本。

原理:

ZooKeeper 是以 Fast Paxos 算法为基础的，Paxos 算法存在活锁的问题，即当有多个 proposer 交错提交时，有可能互相排斥导致没有一个 proposer 能提交成功，而 Fast Paxos 作了一些优化，通过选举产生一个 leader (领导者)，只有 leader 才能提交 proposer，具体 算法可见 Fast Paxos。因此，要想弄懂 ZooKeeper 首先得对 Fast Paxos 有所了解。

ZooKeeper 的基本运转流程:1、选举 Leader。2、同步数据。3、选举 Leader 过程中算法有很多，但要达到的选举标准是一致的。 4、Leader 要具有最高的执行 ID，类似 root 权限。 5、集群中大多数的机器得到响应并 follow

http://www.itheima.com Copyright© 2018 黑马程序员

选出的 Leader。

七、solr

1. 简单介绍一下 solr（2017-11-24-gxb）

Solr 是一个独立的企业级搜索应用服务器，它对外提供类似于 Web-service 的 API 接口。用户可以通过 http 请求，向搜索引擎服务器提交一定格式的 XML 文件，生成索引;也可以 通过 Http Get 操作提出查找请求，并得到 XML 格式的返回结果。

特点:

Solr 是一个高性能，采用 Java5 开发，基于 Lucene 的全文搜索服务器。同时对其进行 了扩展，提供了比 Lucene 更为丰富的查询语言，同时实现了可配置、可扩展并对查询性能 进行了优化，并且提供了一个完善的功能管理界面，是一款非常优秀的全文搜索引擎。

工作方式:

文档通过 Http 利用 XML 加到一个搜索集合中。查询该集合也是通过 http 收到一个 XML/JSON 响应来实现。它的主要特性包括:高效、灵活的缓存功能，垂直搜索功能，高亮 显示搜索结果，通过索引复制来提高可用性，提供一套强大 Data Schema 来定义字段，类 型和设置文本分析，提供基于 Web 的管理界面等。

2. solr 怎么设置搜索结果排名靠前？（2017-11-24-gxb）

可以设置文档中域的 boost 值，boost 值越高，计算出来的相关度得分就越高，排名也就越靠前。此方法可以把热点商品或者推广商品的排名提高。

http://www.itheima.com Copyright© 2018 黑马程序员

3. solr 中 IK 分词器原理是什么？（2017-11-24-gxb）

Ik 分词器的分词原理本质上是词典分词。先在内存中初始化一个词典，然后在分词过程中挨个读取字符，和字典中的字符相匹配，把文档中的所有的词语拆分出来的过程。

八、webService

1. 什么是 webService？（2017-11-24-lyq）

WebService 是一种跨编程语言和跨操作系统平台的远程调用技术。所谓跨编程语言和跨操作平台，就是说服务端程序采用 java 编写，客户端程序则可以采用其他编程语言编写，反之亦然！跨操作系统平台则是指服务端程序和客户端程序可以在不同的操作系统上。

2. 常见的远程调用技术（2017-11-24-lyq）

RMI 是 java 语言本身提供的远程通讯协议，稳定高效，是 EJB 的基础。但它只能用于 JAVA 程序之间的通讯。

Hessian 和 Burlap 是 caucho 公司提供的开源协议，基于 HTTP 传输，服务端不用开防火墙端口。协议的规范公开，可以用于任意语言。跨平台有点小问题。

Httpinvoker 是 SpringFramework 提供的远程通讯协议，只能用于 JAVA 程序间的通讯，且服务端和客户端必须使用 SpringFramework。

Web service 是连接异构系统或异构语言的首选协议，它使用 SOAP 形式通讯，可以用于任何语言，目前的许多开发工具对其的支持也很好。

效率相比：RMI > Httpinvoker >= Hessian >> Burlap >> web service。

http://www.itheima.com Copyright© 2018 黑马程序员

九、Restful

1. 谈谈你对 restful 的理解以及在项目中的使用？ （2017-11-30-wzz）

注意：下面回答内容来自百度百科。

一种软件架构风格、设计风格，而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。REST 指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful。它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用。

给大家推荐如下一篇博客，该博客从多个维度讲解了什么是 Restful 并且给了 Restful 风格样式的 API 接口。

http://blog.csdn.net/liuwenbiao1203/article/details/52351129

http://www.itheima.com Copyright© 2018 黑马程序员

第九章 企业实战面试题

一、智慧星（2017-11-25-wmm）

1. 选择题

1.1 在 Java 中,负责对字节代码解释执行的是(B)

A. 应用服务器

B. 虚拟机

C. 垃圾回收器

D. 编译器

1.2 一个栈的输入序列为 1 2 3 4 5, 则下列序列中不可能是栈输出的序列的是(A)

A. 5 4 1 3 2

B. 2 3 4 1 5

C. 1 5 4 3 2

D. 2 3 1 4 5

1.3 下列那一个选项按照顺序包括了 OSI 模型的 7 个层次(C)

A. 物理层 数据链路层 传输层 网络层 会话层 表示层 应用层

B. 物理层 数据链路层 会话层 网络层 传输层 表示层 应用层

C. 物理层 数据链路层 网络层 传输层 会话层 表示层 应用层

http://www.itheima.com Copyright© 2018 黑马程序员

D. 网络层 传输层 物理层 数据链路层 会话层 表示层 应用层

1.4 当客户度关闭一个从连接池中获取的连接，会发生下面哪一种情况?(A)

A. 连接不会关闭，只是简单地归还给连接池

B. 连接被关闭 ，但又被重新打开并归还给连接池

C. 连接永久性关闭

1.5 以下哪些不是 javaScript 的全局函数(C)

A. eval

B. escape

C. setTimeout

D. parseFloat

1.6 你使用 mkdir 命令创建一个临时的文件夹/tmp/aaa，并将一些文件复制其中，使用完后要删除/mnt/tmp 文件夹及其中的所有文件，应该使用命令（B）

A. rm /tmp/aaa

B. rm –r /tmp/aaa

C. rmdir –r /tmp/aaa

D. rmdir /tmp/aaa

1.7 在 UML 提供的图中,(C) 用于按数据顺序描述对象间的交互

A. 协作图

http://www.itheima.com Copyright© 2018 黑马程序员

B. 网络图

C. 序列图

D. 状态图

1.8 下面有关系统并发访问数估算数据哪个最有效: （B）

A. 高峰时段日处理业务量 100000

B. 高峰时段平均每秒请求数 80

C. 同时在线用户 100

D. 平均每秒用户请求 50

1.9 不同级别的用户对同一对象拥有不同的访问权利或某个客户端不能直接操作到某个对象,但有必须和那个对象有所互动，这种情况最好使用什么设计模式.(D)

A．Bridge 模式

B. Fa?ade 模式

C. Adapter 模式

D. Proxy 模式

1.10 下面哪个 Set 是排序的? (C)

A. LinkedHashSet

B. HashSet

C. TreeSet

http://www.itheima.com Copyright© 2018 黑马程序员

D. AbstractSet

2. 编程题

2.1 用 1,2 , 2 ,3, 4 ,5 这 6 个数字，用 Java 写一个 main 函数，打印出所有不同的排列，如: 512234, 412345 等，要求: “4”不能在第三位, “3”与”5”不能相连。

```
import java.util.Iterator;
import java.util.TreeSet;
public class numberRandom {
String[] stra = {"1","2","2","3","4","5"};
int n = stra.length;
 boolean[] visited = new boolean[n];
   String result = "";
   TreeSet<String> ts = new TreeSet<String>();
   int[][] a = new int[n][n];
   private  void searchMap()
   {
      for(int i=0;i<n;i++)
      {
         for(int j=0;j<n;j++)
         {
            if(i==j)
            {
               a[i][j]=0;
               }else{
                  a[i][j]=1;
               }
         }
      }
      //3 和 5 不能相连
      a[3][5]=0;
      a[5][3]=0;
      //开始遍历
      for(int i=0;i<n;i++)
      {
         search(i);
      }
      Iterator<String> it = ts.iterator();
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        while(it.hasNext())
        {
            String str =it.next();
            //4 不能在第三位
            if(str.indexOf("4")!=2){
                System.out.println(str);
            }
        }
      }

    private void search(int startIndex){
        visited[startIndex] = true;
        result = result + stra[startIndex];
        if(result.length() ==n)
        {
            ts.add(result);
        }
        for(int j=0;j<n;j++)
        {
            if(a[startIndex][j]==1&&visited[j]==false)
            {
                search(j);
            }else
            {
                continue;
            }
        }
        //一个 result 结束后踢掉最后一个，寻找别的可能性，若没有的话，则继续向前踢掉当前最后一个
        result = result.substring(0,result.length()-1);
        visited[startIndex] = false;

    }
    public static void main(String[] args){
        new numberRandom().searchMap();
    }

}
```


http://www.itheima.com Copyright© 2018 黑马程序员

2.2 一个数如果恰好等于它的因子之和，这个数就称为”完数”. 例如 6 = 1+2+3。编程找出1000 以内的所有完数。

```
    public class wsTest {
        public static void main(String[] args) {
            for(int m=2;m<1000;m++){
                int s=0;
                for(int i=1;i<m;i++){
                    if((m%i)==0)
                    s+=i;
                }
                if(s==m){
                    System.out.print(m+" its factors are:");
                    for(int j=1;j<m;j++)
                {
                if((m%j)==0){
                    System.out.print(j);
                    System.out.print(" ");
                }
            }
            System.out.println();
            }
        }
    }
}

结果:
6 its factors are:1 2 3
28 its factors are:1 2 4 7 14
496 its factors are:1 2 4 8 16 31 62 124 248
```


http://www.itheima.com Copyright© 2018 黑马程序员

二、中讯志远科技(2017-11-26-wmm)

1. 问答题

1.1 下面程序的运行结果是？为什么？

```
String str1 = "hello";
String str2 = "he"+new String("llo");
String str3 = "he"+"llo";
System.err.println(str1== str2);
System.err.println(str1 == str3);
答案： false true
```

1.2 HashSet 里的元素是不能重复的，那用什么方法来区分重复与否呢？

答案：当往集合在添加元素时，调用 add（Object）方法时候，首先会调用 Object 的 hashCode()方法判 hashCode 是否已经存在，如不存在则直接插入元素；

如果已存在则调用 Object 对象的 equals()方法判断是否返回 true，如果为 true 则说明元素已经 存在，如为 false 则插入元素。

1.3. List ,Set, Map 是否继承来自 Collection 接口？存取元素时，有何差异？

答案：List,Set 是继承 Collection 接口； Map 不是。

List：元素有放入顺序，元素可重复 ，通过下标来存取 和值来存取

Map：元素按键值对存取，无放入顺序

Set：元素无存取顺序，元素不可重复（注意：元素虽然无放入顺序，但是元素在 set 中的位置是有该元素的 HashCode 决定的，其位置其实是固定的）

http://www.itheima.com Copyright© 2018 黑马程序员

1.4. 简述 Java 中的值传递和引用传递?

参考网址：https://www.cnblogs.com/zhangshiwen/p/5830062.html

按值传递是指的是在方法调用时，传递的参数是按值的拷贝传递。

按值传递重要特点：传递的是值的拷贝，也就是说传递后就互不相关了

示例如下：

```
public class TempTest {
private void test1(int a){
a = 5;
System.out.println("test1 方法中的 a="+a);
}
public static void main(String[] args) {
TempTest t = new TempTest();
int a = 3;
t.test1(a);//传递后，test1 方法对变量值的改变不影响这里的 a
System.out.println(”main 方法中的 a=”+a);
}
}
运行结果是：
test1 方法中的 a=5
main 方法中的 a=3
```

按引用传递是指的是在方法调用时，传递的参数是按引用进行传递，其实传递的引用的地址，也就是变量所

 Copyright© 2018 黑马程序员

对应的内存空间的地址。传递的是值的引用，也就是说传递前和传递后都指向同一个引用（也就是同一个内存空间）。

示例如下：

```
public class TempTest {
private void test1(A a){
a.age = 20;
System.out.println("test1 方法中的 age="+a.age);
}
public static void main(String[] args) {
TempTest t = new TempTest();
A a = new A();
a.age = 10;
t.test1(a);
System.out.println(”main 方法中的 age=” +a.age);
}
}
class A{
public int age = 0;
运行结果如下：

[java] view plain copy
test1 方法中的 age=20
main 方法中的 age=20
```

1.5 switch 是否作用在 byte 上，是否能作用在 long 上，是否能作用在 String 上？

答案：switch 可作用于 char byte short int；

switch 可作用于 char byte short int 对应的包装类；

switch 不可作用于 long double float boolean，包括他们的包装类；

1.6 Java 语言如何进行异常处理？请写出几个常见的运行时异常的编译时异常

答案： java 语言进行异常处理的方式有：

http://www.itheima.com Copyright© 2018 黑马程序员

throws: throws 是方法可能抛出异常的声明。(用在声明方法时，表示该方法可能要抛出异常)

throw : throw 是语句抛出一个异常。

常见的运行时异常的编译时异常:

NullPointerException - 空指针引用异常

ClassCastException - 类型强制转换异常。

IllegalArgumentException - 传递非法参数异常。

ClassNotFoundException - 类找不到异常

ArrayStoreException - 向数组中存放与声明类型不兼容对象异常

IndexOutOfBoundsException - 下标越界异常

NegativeArraySizeException - 创建一个大小为负数的数组错误异常

NumberFormatException - 数字格式异常

SecurityException - 安全异常

UnsupportedOperationException - 不支持的操作异常

1.7 简述数据库事务和实际工作中的作用

所谓事务是用户定义的一个数据库操作序列,这些操作要么全做要么全不做,是一个不可分割的工作单位。例如,在关系数据库中,一个事务可以是一条 SQL 语句、一组 SQL 语句或整个程序。

简单举个例子就是你要同时修改数据库中两个不同表的时候，如果它们不是一个事务的话，当第一个表修改完，可是第二表改修出现了异常而没能修改的情况下，就只有第二个表回到未修改之前的状态，而第一个表已经被修改完毕。而当你把它们设定为一个事务的时候，当第一个表修改完，可是第二表改修出现了异常而没能修改

http://www.itheima.com Copyright© 2018 黑马程序员

的情况下，第一个表和第二个表都要回到未修改的状态！这就是所谓的事务回滚。

例如，在将资金从一个帐户转移到另一个帐户的银行应用中，一个帐户将一定的金额贷记到一个数据库表中，同时另一个帐户将相同的金额借记到另一个数据库表中。由于计算机可能会因停电、网络中断等而出现故障，因此有可能更新了一个表中的行，但没有更新另一个表中的行。如果数据库支持事务，则可以将数据库操作组成一个事务，以防止因这些事件而使数据库出现不一致。如果事务中的某个点发生故障，则所有更新都可以回滚到事务开始之前的状态。如果没有发生故障，则通过以完成状态提交事务来完成更新。

三、腾讯（2016 年校招面试题 2017-11-29-wzy）

1. 选择题

1.1 已知一棵二叉树，如果先序遍历的节点顺序是：ADCEFGHB ，中序遍历是：CDFEGHAB ，则后序遍历结果为：（ D ）

A. CFHGEBDA

B. CDFEGHBA

C. FGHCDEBA

D. CFHGEDBA

知识点

对于二叉树的遍历方式一般分为三种先序、中序、后序三种方式：

先序遍历（根左右）

若二叉树为空，则不进行任何操作：否则

http://www.itheima.com Copyright© 2018 黑马程序员

1、访问根结点。

2、先序方式遍历左子树。

3、先序遍历右子树。

中序遍历 （左根右）

若二叉树为空，则不进行任何操作：否则

1、中序遍历左子树。

2、访问根结点。

3、中序遍历右子树。

后序遍历 （左右根）

若二叉树为空，则不进行任何操作：否则

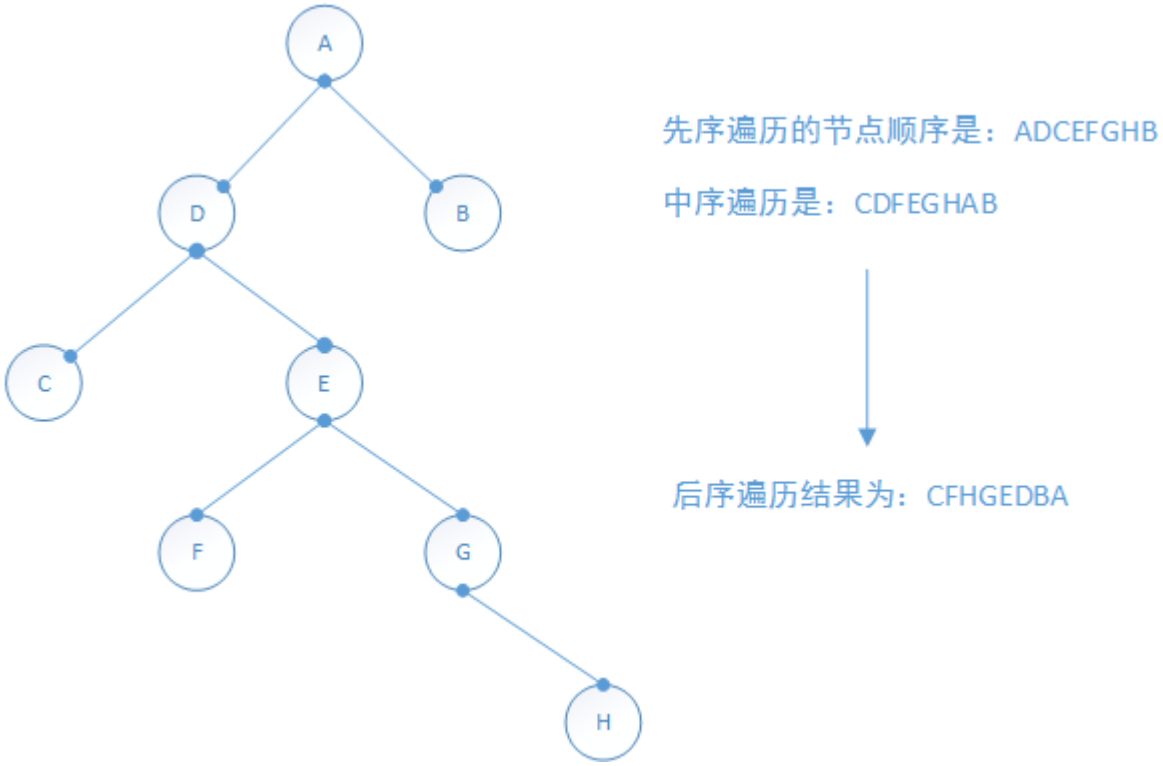
1、后序遍历左子树。

2、后序遍历右子树。

3、放问根结点。

因此，根据题目给出的先序遍历和中序遍历，可以画出二叉树：

http://www.itheima.com Copyright© 2018 黑马程序员



1.2 下列哪两个数据结构，同时具有较高的查找和删除性能？（CD）

A. 有序数组

B. 有序链表

C. AVL 树

D. Hash 表

知识点:

http://www.itheima.com Copyright© 2018 黑马程序员

数据结构	查找	插入	删除	遍历
数组	O(N)	O(1)	O(N)	—
有序数组	O(logN)	O(N)	O(N)	O(N)
链表	O(N)	O(1)	O(N)	—
有序链表	O(N)	O(N)	O(N)	O(N)
二叉树（一般情况）	O(logN)	O(logN)	O(logN)	O(N)
二叉树（最坏情况）	O(N)	O(N)	O(N)	O(N)
平衡树（一般情况和最坏情况）	O(logN)	O(logN)	O(logN)	O(N)
哈希表	O(1)	O(1)	O(1)	—

http://blog.csdn.net/zs634134578

平衡二叉树的查找，插入和删除性能都是 O(logN) ，其中查找和删除性能较好；哈希表的查找、插入和删除性能都是 O(1) ，都是最好的。所以最后的结果选择： CD

1.3 下列排序算法中，哪些时间复杂度不会超过 nlogn？（BC）

A. 快速排序

B. 堆排序

C. 归并排序

D. 冒泡排序

知识点

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定
注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数						

根据上图，观察平均情况，最好最差情况的时间复杂度基本可以知道答案了，最后结果选择： BC。

1.4 初始序列为 1 8 6 2 5 4 7 3 一组数采用堆排序，当建堆（小根堆）完毕时，堆所对应的二叉树中序遍历序列为：（ A ）

A. 8 3 2 5 1 6 4 7

B. 3 2 8 5 1 4 6 7

C. 3 8 2 5 1 6 7 4

D. 8 2 3 5 1 4 7 6

初始化序列：1 8 6 2 5 4 7 3,，小根堆就是要求结点的值小于其左右孩子结点的值，左右孩子的大小没有关系，那么小根堆排序之后为：1 2 4 3 5 6 7 8;

中序遍历：左根右，故遍历结果为：8 3 2 5 1 6 4 7

故最后选择的结果： A

http://www.itheima.com Copyright© 2018 黑马程序员

1. 当 n = 5 时，下列函数的返回值是：（A）

```
int foo(int n)
{
    if(n<2)return n;
    return foo(n-1)+foo(n-2);
}
```

A．5

B．7

C．8

D．1

1.5 S 市 A ，B 共有两个区，人口比例为 3：5 ，据历史统计 A 区的犯罪率为 0.01% ，B 区为 0.015% ，现有一起新案件发生在 S 市，那么案件发生在 A 区的可能性有多大？（C）

A．37.5%

B．32.5%

C．28.6%

D．26.1%

这道题首先得了解犯罪率是什么？犯罪率就是犯罪人数与总人口数的比。因此可以直接得出公式：（ 3 0.01% ）/（ 3 0.01% + 5 * 0.015% ）= 28.6%

当然如果不好理解的话，我们可以实例化，比如 B 区假设 5000 人，A 区 3000 人，A 区的犯罪率为 0.01%，那

http://www.itheima.com Copyright© 2018 黑马程序员

么 A 区犯罪人数为 30 人，B 区的犯罪率为 0.015% ，那么 B 区的犯罪人数为 75 人 ，求发生在 A 区的可能性，就是说 A 区的犯罪人数在总犯罪人数的多少，也就是 30/(30+75)=0.2857

当然，也可以回归到我们高中遗忘的知识：

假设 C 表示犯案属性

在 A 区犯案概率：P(C|A)=0.01%

在 B 区犯案概率：P(C|B)=0.015%

在 A 区概率：P(A)=3/8

在 B 区概率：P(B)=5/8

犯案概率：P(C)=（3/80.01%+5/80.015%)

根据贝叶斯公式：P(A|C) = P(A,C) / P(C) = [P(C|A) P(A)] / [P(C|A) P(A)+ P(C|B) P(B)] 也可以算出答案来

故，最后结果选择为： C

1.6 Unix 系统中，哪些可以用于进程间的通信？（ABCD ）

A．Socket

B．共享内存

C．消息队列

D．信号量

知识点

管道（Pipe）及有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信；

信号（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程

http://www.itheima.com Copyright© 2018 黑马程序员

还可以发送信号给进程本身；linux 除了支持 Unix 早期信号语义函数 sigal 外，还支持语义符合 Posix.1 标准的信号函数 sigaction（实际上，该函数是基于 BSD 的，BSD 为了实现可靠信号机制，又能够统一对外接口，用 sigaction 函数重新实现了 signal 函数）；

报文（Message）队列（消息队列）：消息队列是消息的链接表，包括 Posix 消息队列 system V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

共享内存：使得多个进程可以访问同一块内存空间，是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。

套接口（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由 Unix 系统的 BSD 分支开发出来的，但现在一般可以移植到其它类 Unix 系统上：Linux 和 System V 的变种都支持套接字。

故最后选择的结果为： ABCD

1.7 静态变量通常存储在进程哪个区？（ C ）

A．栈区

B．堆区

C．全局区

D．代码区

静态变量的修饰关键字：static，又称静态全局变量。故最后选择的结果为： C

http://www.itheima.com Copyright© 2018 黑马程序员

1.8 如何提供查询 Name 字段的性能（ B ）

A． 在 Name 字段上添加主键

B． 在 Name 字段上添加索引

C． 在 Age 字段上添加主键

D． 在 Age 字段上添加索引

1.9 IP 地址 131.153.12.71 是一个（B）类 IP 地址

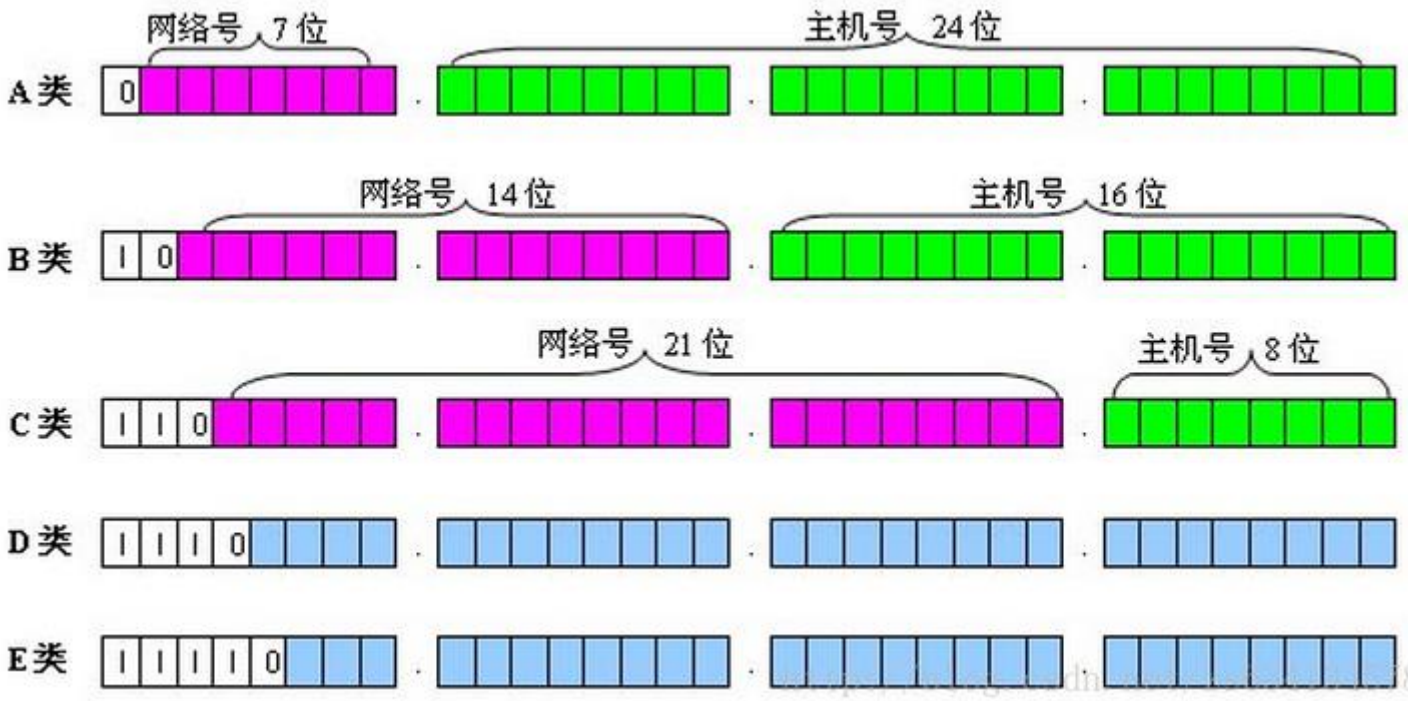
A．A

B．B

C．C

D．D

知识点



http://www.itheima.com Copyright© 2018 黑马程序员

1.10 浏览器访问某页面，HTTP 协议返回状态码为 403 时表示：（ B ）

A. 找不到该页面

B. 禁止访问

C. 内部服务器访问

D. 服务器繁忙

1.11 如果某系统 15*4=112 成立，则系统采用的是（ A ）进制

A．6

B．7

C．8

D．9

这题因为是选择题，我们可以直接从 A 的选项开始，假设是 6 进制的，我们把等式 15 4 = 112 转为十进制，就是 11 4 = 44，最后验证等式是否成立，明显等式是成立的，因此答案已经出来了，选择 A 。

当然我们也可以假设是 X 进制，且我们知道 X 大于 5， 则：（x+5）4 = xx +x +2，所以最后计算的结果也为 6。

1.12 TCP 和 IP 分别对应了 OSI 中的哪几层？（CD）

A. Application layer

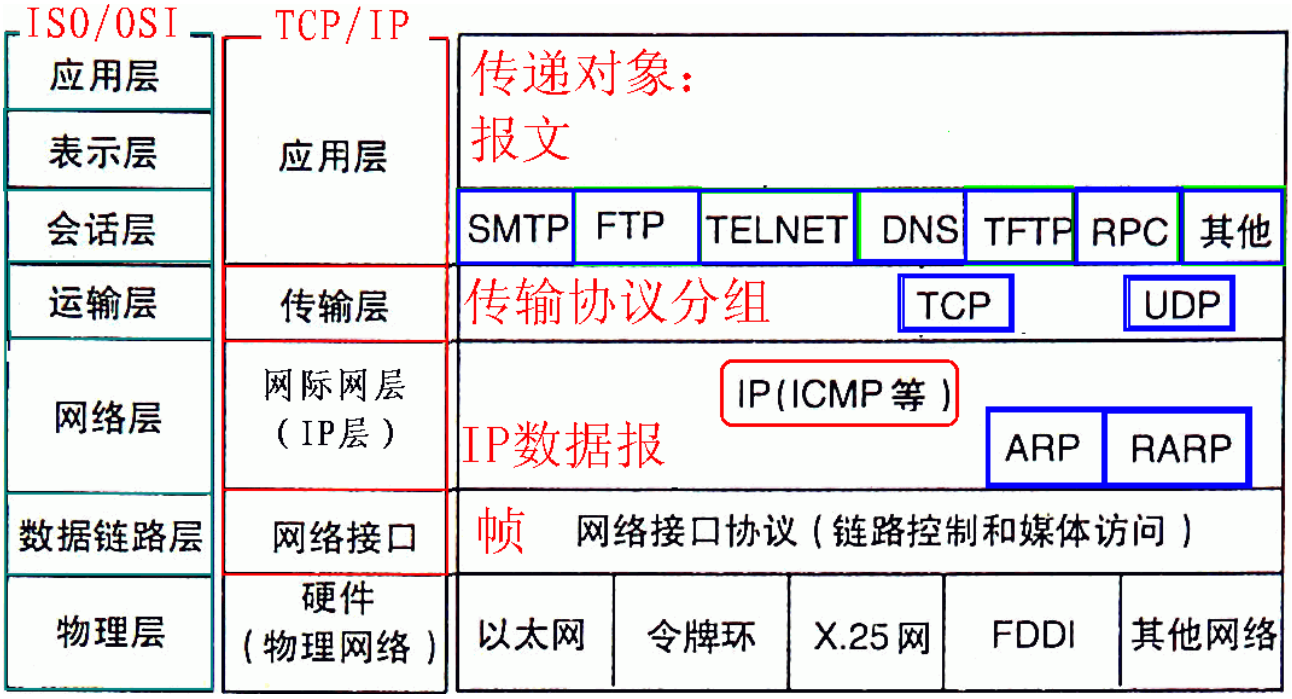
B. Presentation layer

C. Transport layer

D. Network layer

http://www.itheima.com Copyright© 2018 黑马程序员

知识点



1.13 一个栈的入栈序列是 A，B，C，D，E，则栈的不可能的输出序列是？（C）

A．EDCBA

B．DECBA

C．DCEAB

D．ABCDE

堆栈分别是先进后出,后进先出，

选项 a 是 abcde 先入栈,然后依次出栈,正好是 edcba

选项 b 是 abcd 先依次入栈,然后 d 出栈, e 再入栈, e 出栈

选项 c 是错误的,不可能 a 先出栈

选项 d 是 a 入栈,然后 a 出栈；b 再入栈, b 出栈.依此类推

http://www.itheima.com Copyright© 2018 黑马程序员

最后的结果选择 C。

1.14 同一进程下的线程可以共享以下？（BD）

A．stack

B．data section

C．register set

D．file fd

知识点

线程共享的内容包括：

1.进程代码段

2.进程的公有数据(利用这些共享的数据，线程很容易的实现相互之间的通讯)

3.进程打开的文件描述符、

4.信号的处理器、

5.进程的当前目录和

6.进程用户 ID 与进程组 ID

线程独有的内容包括：

1.线程 ID

2.寄存器组的值

3.线程的堆栈

4.错误返回码

5.线程的信号屏蔽码

http://www.itheima.com Copyright© 2018 黑马程序员

所以选择为 BD。

1.15 对于派生类的构造函数，在定义对象时构造函数的执行顺序为？（D）

1：成员对象的构造函数

2：基类的构造函数

3：派生类本身的构造函数

A．123

B．231

C．321

D．213

1.16 递归函数最终会结束，那么这个函数一定？（B）

A. 使用了局部变量

B. 有一个分支不调用自身

C. 使用了全局变量或者使用了一个或多个参数

D. 没有循环调用

1.17 编译过程中，语法分析器的任务是（BCD）

A. 分析单词是怎样构成的

B. 分析单词串是如何构成语言和说明的

C. 分析语句和说明是如何构成程序的

http://www.itheima.com Copyright© 2018 黑马程序员

D. 分析程序的结构

知识点

1.词法分析（lexical analysis）

词法分析是编译过程的第一个阶段。这个阶段的任务是从左到右的读取每个字符，然后根据构词规则识别单词。词法分析可以用 lex 等工具自动生成。

2.语法分析（syntax analysis）

语法分析是编译过程的一个逻辑阶段。语法分析在词法分析的基础上，将单词序列组合成各类语法短语，如“程序”，“语句”，“表达式”等等。语法分析程序判断程序在结构上是否正确。

3.语义分析（semantic analysis）

属于逻辑阶段。对源程序进行上下文有关性质的审查，类型检查。如赋值语句左右端类型匹配问题。

所以 BCD 都属于词法分析，选择结果为 BCD。

1.18 同步机制应该遵循哪些基本准则？（ABCD）

A．空闲让进

B．忙则等待

C．有限等待

D．让权等待

1.19 进程进入等待状态有哪几种方式？（D）

A. CPU 调度给优先级更高的线程

B. 阻塞的线程获得资源或者信号

http://www.itheima.com Copyright© 2018 黑马程序员

C. 在时间片轮转的情况下，如果时间片到了

D. 获得 spinlock 未果

1.20 设计模式中，属于结构型模式的有哪些？（BC）

A. 状态模式

B. 装饰模式

C. 代理模式

D. 观察者模式

四、北京宝蓝德股份科技有限公司（2017-12-03-wmm）

1.选择题

1.1 下面代码的运行结果是（C）

```
public class Test{
    public static void main (String[] args){
       List<String> a = null;
       test(a);
       System.out.println(a.size());
     }
   public static void test(List<String> a){
     a=new arrayList<String>();
     a.add(“abc”);
  }
}
```

A．0

B．1

C．Java.lang.NullPointerException

http://www.itheima.com Copyright© 2018 黑马程序员

D．以上都不正确

1.2 Linux 下查看进程占用的 CPU 的百分比， 使用工具（A）

A. Ps

B. Cat

C. More

D. Sep

1.3 JVM 内存里哪个区域不可能发生 OutOfMerncyError(A)

A. 程序计数器

B. 堆

C. 方法区

D. 本地方法栈

1.4 下面关于阻塞队列（java.util.concurrent.BlockingQueue）的说法不正确的是(C)

A. 阻塞队列是线程安全的

B. 阻塞队列的主要应用场景是“生产者-消费者”模型

C. 阻塞队列里的元素不能为 null

D. 阻塞队列的实现必须显示地设置容量

http://www.itheima.com Copyright© 2018 黑马程序员

1.5 如果现在需要创建一组任务，他们并行的执行工作，然后进行下一个步骤之前等待，直至所有的任务都完成，而去这种控制可以重用多次， 这种情形使用 java.util.concurrent 包中引入哪种同步工具最适合（B）

A. CountDownLatch

B. CyclicBarrier

C. Semaphore

D. FutureTask

2.问答题

2.1 java 中， 为什么基类不能做为 HashMap 的键值， 而只能是引用类型，把引用类型作为 HashMap 的键值， 需要注意哪些地方？

答案：引用类型和原始类型的行为完全不同，并且它们具有不同的语义。引用类型和原始类型具有不同的特征和用法，它们包括：大小和速度问题，这种类型以哪种类型的数据结构存储，当引用类型和原始类型用作某个类的实例数据时所指定的缺省值。对象引用实例变量的缺省值为 null，而原始类型实例变量的缺省值与它们的类型有关。

2.2 编写一个工具类 StringUtil，提供方法 int compare(char[] v1 ,char[] v2)方法，比较字符串 v1,v2 ,如果按照字符顺序 v1>v2 则 return 1 ,v1=v2 则 return 0， v1<v2 则 return-1。

```
public class StringUtil{
int compare(char[] v1,char[] v2) {
   String str1 = new String(v1);
   String str2 = new String(v2);
   int result = str1.compareTo(str2);
   return result == 0 ? 0 : (result > 0 ? 1 : -1);
}
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
    }
```

2.3 Java 出现 OutOfMemoryError(OOM)的原因有那些？出现 OOM 错误后，怎么解决？

参考播客：http://www.jianshu.com/p/2fdee831ed03

触发 java.lang.OutOfMemoryError:最常见的原因就是应用程序需要的堆空间是大的，但是 JVM 提供的却小。这个的解决方法就是提供大的堆空间即可。除此之外还有复杂的原因：，

内存泄露：特定的编程错误会导致你的应用程序不停的消耗更多的内存，每次使用有内存泄漏风险的功能就会留下一些不能被回收的对象到堆空间中，随着时间的推移，泄漏的对象会消耗所有的堆空间，最终触发 java.lang.OutOfMemoryError: Java heap space 错误。

解决方案：

第一个解决方案是显而易见的，你应该确保有足够的堆空间来正常运行你的应用程序，在 JVM 的启动配置中增加如下配置：

```
-Xmx1024m
```

流量/数据量峰值：应用程序在设计之初均有用户量和数据量的限制，某一时刻，当用户数量或数据量突然达到一个峰值，并且这个峰值已经超过了设计之初预期的阈值，那么以前正常的功能将会停止，并触发

http://www.itheima.com Copyright© 2018 黑马程序员

java.lang.OutOfMemoryError: Java heap space 异常

解决方案，如果你的应用程序确实内存不足，增加堆内存会解决 GC overhead limit 问题，就如下面这样，给你的应用程序 1G 的堆内存：

```
java -Xmx1024m com.yourcompany.YourClass
```

五、智慧流（2017-12-04-wmm）

1.选择题

1.1 下列关于栈的描述错误的是（B）

A. 栈是先进后出的线性表

B. 栈只能顺序存储

C. 栈具有记忆功能

D.对栈的插入和删除操作中，不需要改变栈底指针

1.2 对于长度为 n 的线性表，在最坏的情况下，下列个排序法所对应的比较次数中正确的是（D）

A. 冒泡排序为 n/2

B. 冒泡排序为 n

C. 快速排序为 n

http://www.itheima.com Copyright© 2018 黑马程序员

D. 快速排序为 n(n-1)/2

1.3 阅读下列代码后， 下列正确的说法是（A）

```
public class Person{
     int arr[] = new int[10];
     public static void main(String args[ ]){
       System.out.println(arr[1]);
    }
  }
```

A 编译时将产生错误

B 编译时正确，运行时将产生错误

C 输出空

D 输出 0

1.4 执行以下程序后输出的结果是（D）

```
public class Test {
       public static void main(String[] args) {
          StringBuffer a = new StringBuffer("A");
          StringBuffer b = new StringBuffer("B");
          operator(a,b);
          System.out.println(a+","+b);
       }
      public static void operator(StringBuffer x,StringBuffer y){
          x.append(y);
          y=x;
      }
}
```

A . A,A

B. A,B

C. B,B

http://www.itheima.com Copyright© 2018 黑马程序员

D. AB,B

1.5 下列不属于持久化的是（A）

A. 把对象转换成为字符串的形式通过网络传输，在另一端接收到字符串把对象还原出来

B. 把程序数据从数据库中读出来

C. 从 XML 配置文件中读取程序的配置信息

D. 把程序数据保存为文件

1.6 下列代码输出的结果是（C）

```
int x= 0;
    int y=10;
    do{
    y--;
    ++x;
    }while(x<6);
    System.out.println();
}
```

A. 5,6

B. 5,5

C. 6,5

D. 6,6

1.7 下列程序段输出的结果是（B）

```
Void complicatedexpression_f(){
int x=20,y=30;
boolean j;
j=x>50&&y>60|| x>50&& y<-60 || x<-50&&y>60 || x<-50&& y<-60;
System.out.println(j);
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
    }
```

A. true

B. false

C. 1

D. 001

1.8 一个栈的输入序列为 123，则下列序列中不可能是栈输出的序列的是（C）

A. 2 3 1

B. 3 2 1

C. 3 1 2

D. 1 2 3

8、当 n = 5 时， 下列函数的返回值是（D）

```
int foo(int n){
   if(n<2) return n;
   return foo(n-1)+foo(n-2);
}
```

A. 1

B. 8

C. 7

D. 5

http://www.itheima.com Copyright© 2018 黑马程序员

1.9 设有一个二维数组 A[m][n],假设 A[0][0]存放的位置在 644（10），A[2][2]存放的文职在 676（10）每个元素占一个空间，问 A[3][3]（10）存放在什么位置？脚注（10）表示用 10 进制表示（C）

A. 688

B. 678

C. 692

D. 699

1.10 下列代码执行结果是（B）

```
public static void main (String args[]){
   Thread t = new Thread(){
   public void run(){
    pong();
  }
  };
   t.run();
   System.out.print("ping");
 }
static void pong(){
   System.out.print("pong");
}
```

A. pingpong

B. pongping

C. pingpong 和 pongping 都有可能

D.都有可能

http://www.itheima.com Copyright© 2018 黑马程序员

1.11 下面程序能正常运行吗（可以）

```
Public class NULL{
    Public static void haha(){
        System.out.println(“haha”);
    }
    Public static void main(String[] args){
        ((NULL)null).haha();
    }
}
```

2. 问答题

2.1 解释一下什么是 Servlet，说一说 Servlet 的生命周期

Servlet 是一种服务器端的 Java 应用程序，具有独立于平台和协议的特性,可以生成动态的 Web 页面。 它担当客户请求 (Web 浏览器或其他 HTTP 客户程序) 与服务器响应 (HTTP 服务器上的数据库或应用程序) 的中间层。 Servlet 是位于 Web 服务器内部的服务器端的 Java 应用程序，与传统的从命令行启动的 Java 应用程序不同，Servlet 由 Web 服务器进行加载，该 Web 服务器必须包含支持 Servlet 的 Java 虚拟机

Servlet 生命周期可以分成四个阶段：加载和实例化、初始化、服务、销毁。

当客户第一次请求时，首先判断是否存在 Servlet 对象，若不存在，则由 Web 容器创建对象，而后调用 init()方法对其初始化，此初始化方法在整个 Servlet 生命周期中只调用一次。

完成 Servlet 对象的创建和实例化之后，Web 容器会调用 Servlet 对象的 service()方法来处理请求。

当 Web 容器关闭或者 Servlet 对象要从容器中被删除时，会自动调用 destory()方法。

2.2 过滤器有哪些作用和用法？

对于一个 web 应用程序来说，过滤器是处于 web 容器内的一个组件，它会过滤特定请求资源请求信息和响应信

http://www.itheima.com Copyright© 2018 黑马程序员

息。一个请求来到时，web 容器会判断是否有过滤器与该信息资源相关联，如果有则交给过滤器处理，然后再交给目标资源，响应的时候则以相反的顺序交给过滤器处理，最后再返回给用户浏览器。

常见的过滤器用途主要包括：对用户请求进行统一认证、对用户的访问请求进行记录和审核、对用户发送的数据进行过滤或替换、转换图象格式、对响应内容进行压缩以减少传输量、对请求或响应进行加解密处理、触发资源访问事件等。

2.3 写出一个冒泡排序

从大到小：

```
public void BigAndSmall(){
        int arr[]={-5,29,7,10,5,16};
        for(int i=1;i<arr.length;i++){
            for(int j=0;j<arr.length-i;j++){
                if(arr[j]<arr[j+1]){
                    int temp;
                    temp=arr[j];
                    arr[j]=arr[j+1];
                    arr[j+1]=temp;
                }
            }
        }
        for(int i =0;i<arr.length;i++){
            System.out.print(" "+arr[i]+" ");
        }
}
```

1. 写出一个单例的实现（懒加载方式）

```
public class LazySingleton {
 private LazySingleton(){
 }
 private static class SingletonHolder{
  private static LazySingleton instance = new LazySingleton();
 }
 public static LazySingleton getInstance(){
    return SingletonHolder.instance;
 }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
}
```

3. 逻辑思维题

3.1 4 2 12 28 80 （C）

A. 124

B. 96

C. 216

D. 348

3.2 2006 年某人连续打工 24 天，共赚了 190 元（日工资 10 元，星期日工资 5 元，星期日休息无工资）。已知他打工是从 1 月下旬的某一天开始的，这个月的 1 日恰好是星期日，这人打工结束的那一天是 2 月（C）日

A. 2 月 6 日

B. 2 月 14 日

C. 2 月 18 日

D. 2 月 21 日

2.3 由甲地到乙地有一天线路的巴士，全程行驶时间 42 分钟，到达总站后，司机至少休息 10 分钟，巴士就掉头行驶，如果这条线路甲，乙两边总站每隔 8 分钟都发一辆（不必是同一时间），则这条线路至少需要是多少俩巴士（C）

A. 15

B. 14

http://www.itheima.com Copyright© 2018 黑马程序员

C. 13

D.12

2.4 编号为 1 至 10 的 10 个果盘中，每盘都盛有水果，共盛放 100 个。其中第一盘里有 16 个，并且编号相邻的三个果盘中水果是的和都相等，求第 8 盘中水果最多可能有几个（A）

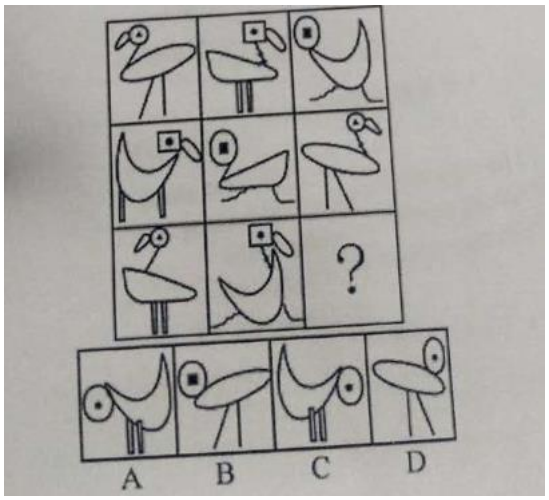
A. 11

B. 12

C. 13

14. 14

2.5 根据下面图片选出正确的答案（B）



2.6 一只蜗牛掉进 20 米深的井中，白天往上爬 3 米，晚上有掉下去 2 米，请问要几天才能爬出来？

第一天爬了 3 米,然后掉了 2 米,实际上爬了 1 米;

http://www.itheima.com Copyright© 2018 黑马程序员

第二天从 1 米处开始向上爬了 3 米,然后掉了 2 米,实际上爬了 2 米;

第三天从 2 米处开始向上爬了 3 米,然后掉了 2 米,实际上爬了 3 米;

. .....

第十八天从 17 米处开始向上爬了 3 米,嘿刚好是 20 米.到了

正解:18

2.7 按规律填数字 1， 1， 2， 3， ？。

答案：5。

2.8 假设一个池塘，里面有无穷多的水，现在有 2 个空水壶容积分别是 5 升和 6 升，问如何用这两只水壶取得 3 升水。

答案：5L 桶打满水，全部倒入 6L 桶；

5L 桶再次打满，往 6L 桶倒水至其满。此时 5L 桶留下 4L 水；

6L 桶清空，将 5L 桶中的 4L 水倒入 6L 桶；

5L 桶打满水，往 6L 桶倒水至其满，则 5L 桶中得 3L 水。

2.9 在房里有三盏灯，房外有三个开关，在房外看不见房内的情况，你只能进门一次，你用什么方法来区分那个开关控制哪一盏灯。

答案：先打开第一个开关，开一会再关上，然后打开第二个开关进入房间

再摸一下每个灯，发热的那盏是第一个开关的，

亮的那盏是第二个开关的，

http://www.itheima.com Copyright© 2018 黑马程序员

没变化的那盏是第三个开关的 。

2.10 两个盲人，他们各自买个两双黑袜和白袜，8 双袜子的布质，大小完全相同，每双袜子都有 1 张商标纸连着，两位盲人不小心把 8 双袜子混在的一起，问他们怎样才能取回黑袜和白袜各两双。

答案：把每双袜子分成两只。

每人各拿一只。

这样，每人手中就有四只黑袜，四只白袜。

每人也就有两双黑袜，两双白袜了。

2.11 一楼到十楼的每层电梯门口都方和一颗钻石，钻石大小不一，你乘坐电梯从一楼到十楼，每层楼电梯门都会打开一次，手里只能拿一颗钻石，问怎样才能拿到最大的钻石。

答案：电梯每层都会开一下的，所以，在第一层就拿，到第二层，看到更大就换一下，更小就不换，一直这样上去，到最上层后，拿到的就是最大的

六、某公司 (2017-12-05-wmm)

1. 选择题

1.1 ArrayList list = new ArrayList(20);语句中的 list 集合大小扩充了几次（A）

A.0

B.1

C.2

http://www.itheima.com Copyright© 2018 黑马程序员

D.3

1.2 如果去掉了 main 方法的 static 修饰符会怎样（B）

A.程序无法翻译

B.程序能正常编译，运行时或抛出 NoSuchMethodError 异常

C.程序能正常编译，正常运行

D.程序能正常编译，正常运行一会会立刻退出

1.3 启动 java 程序进程时，输入一下哪个参数可以实现年轻代的堆大小为 50M(C)

A.-Xms50M

B.-Xmx50M

C.-Xmn50M

D.-Xss50M

1.4 下面程序输出的结果是（）

```
    static boolean foo(char c) {
    System.out.print(c);
    return true;
}

public static void main(String[] args) {
    int i = 0;
    for (foo('A'); foo('B') && (i < 2); foo('C')) {
        i++;
        foo('D');
    }
}
```

输出结果为：(A)

http://www.itheima.com Copyright© 2018 黑马程序员

```
A. ABDCBDCB
B. ABDCDBCB
C. ABDBCDCB
D. ABDBCDCB
```

1.5 下面哪些是 Thread 类的方法（A，B）

A.start()

B.run()

C.exit()

D.getPriority()

1.6 以下语句输出的结果是什么（C）

System.out.print(Integer.MAX_VALUE*2);

System.out.print(Integer.MIN_VALUE*2);

A. -2-1

B. -1-2

C. -20

D. -1-1

1.7log4j 的优先级从高到低的排序为（A）

A. error>warn>info>debug

B. warn>info>debug>error

http://www.itheima.com Copyright© 2018 黑马程序员

C. warn >debug>error>info

D. error>warn>debug>info

1.8 下列哪些方法可以使线程从运行状态进入到阻塞状态（BCD）

A.notify

B.wait

C.sleep

D.yield

1.9 下列关于 Thread 类提供的线程控制的方法中，错误的一项是（A）

A. 在线程 A 中执行线程 B 的 join()方法，则线程 A 等待直到 B 执行完成

B. 线程 A 通过调用 interrupt()方法来中断其阻塞状态。

C. currentThread()方法返回当前线程的引用

D. 若线程 A 调用方法 isAlive（）返回为 true,则说明 A 正在执行中

1.10 设 String s1 =”Topwalk”;String s2 =”Company”; 以下方法可以得到字符串“TopwalkCompany”有：（ABD）

A. s2+s1;

B. s1.concat(s2)

C. s1.append(s2);

D.StringBuffer buf = new StringBuffer(s1);buf.append(s2);

http://www.itheima.com Copyright© 2018 黑马程序员

1.11 String a = new String(“1”+”2”)最终创建了几个对象（B）

A.1

B.2

C.3

D.4

1.12 int 类型占用（C）个字节？

A.2

B.4

C.8

D.16

1.13.下列那一条语句可以实现快速的复制一张数据库表（C）

A. select * into b from a where 1<>1;

B. creat table b as select * from a where 0=1;

C. insert into b as select * from a where 1<>1;

D. insert into b select * from a where 1<>1;

1.14 属于单利模式的特点的是（ACD）

A. 提供了对唯一实现的受控访问

B. 允许可变数目的实例

http://www.itheima.com Copyright© 2018 黑马程序员

C. 单利模式的抽象层会导致单例类扩展有和那的困难

D. 单利模式很容易导致数据库的连接池溢出

1.15 选择 Oracle 的分页语句的关键字（A）

A. rownum

B. limit

C.TOP

D. pagenum

1.16 选出可以查询出所有的表和视图的方法：（B）

A.preparedStatement.getMetaData().getTables(***);

B.connection.getMetaData().getTables(***);

C.result.getMetaData().getTables(***);

D..DiverManager.getMeta().getTables(***);

1.17 可以监控到数据库变化的机制有哪些（AB）

A. 存储过程

B. 数据库日志

C. 触发器

D. 物化视图

http://www.itheima.com Copyright© 2018 黑马程序员

1.18 清空表所有数据的性能最优的语句是哪一个(B)

A. delete from tsuer;

B. truncate table tuser;ss

C. drop table tuser;

D. delete tuser;

1.19 文件对外共享的协议有哪几个（AB）

A. FTP

B. Windows 共享

C. TCP

D.SSH

1.20 关于 Java 中国特殊符号的用法正确的是（AD）

A. 判断一个字符串 str 中是否含有“.”,可以根据 str.indexOf(“.”)是否等于-1 判断。

B. 判断一个字符串 str 是否含有“.”,可以根据 str.indexOf(“\\.”)是否等于-1 判断。

C. 根据“.”分隔字符串 str 的写法可以是 str.split(“\\.”)

D. 根据“.”分隔字符串 str 的写法可以是 str.split(“.”)

1.21 根据以下代码回答问题，放置什么方法在地 6 行，会引起编译错误的是（B）

```
1  class Super{
2      public float getNum(){

3          }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
4       }
5   public class Sub extends Super{
6
7       }
```

A. public float getNum{return 4,0f;}

B.public void getNum(){};

C.public void getNum(double d()){}

D.public double getNum（float d）{return 4,0d;}

1.22 根据以下代码回答问题：输出结果是什么？(B)

```
public class Foo{
    public static void main(String args[]){
        try{return;}
        finally{System.out.println("Finally");}
    }
}
```

A. print out nothing;

B. print out "Finally"

C. 编译错误

D. 以上都不对

1.23 根据以下代码回答问题，请问输出 i 和 j 的值是多少（D）

```
int i=1,j=10;
do{
    if(i++>--j) continue;
}while(i<5)
```

A. i=6 j=5

http://www.itheima.com Copyright© 2018 黑马程序员

B i=5 j=5

C i=6 j=4

D i=5 j=6

1.24 请问 java 关键字？（CD）

A. run

B. low

C. import

D. implement

1.25 以下哪些不属于约束（CD）

A.主键

B.外键

C.索引

D.唯一索引

E.not null

1.26 下列关于数据库连接池的说法中哪个是错误的（D）

A. 服务器启动时会初始建立一定数量的池连接，并一直维持不少于此数目的池连接

B.客户端程序需要连接时，池驱动程序会返回一个使用的池连接并将其使用计数加 1；

C. 如果当前没有空闲连接，驱动程序就会再新建一定数量的连接，新建连接的数量可以由配置参数决定。

http://www.itheima.com Copyright© 2018 黑马程序员

D. 当使用池连接调用完成后，池驱动程序将此连接标记为空间，其他调用就可以使用这个连接

1.27 以下哪句是对索引的错误描述（C）

A. 选择性差的索引只会降低 DML 语句的执行速度

B. 选择性强的索引只有被 Access Path 使用到才是有用的索引

C. 过多的索引只会阻碍性能的提升，而不是加速性能

D.在适当的时候将最常用的列放在复合索引的最前面

E. 索引和表的数据都存储在同一个 Segment 中

1.28 关于锁 locks，描述正确的是（A）

A. 当一个事务在表上防止了共享锁（shared lock）,其他事务，能阅读表里的数据

B. 当一个事务在表上防止了共享锁（shared lock）,其他事务，能更新表里的数据

C. 当一个事务在表上防止了排他锁（exclusive lock）,其他事务，能阅读表里的数据

D. 当一个事务在表上防止了排他锁（exclusive lock）,其他事务，能更新表里的数据

1.29 如下那种情况下，Oracle 不会使用 Full Table Scean(D)

A.缺乏索引，特别是在列上使用了函数，如果要利用索引，则需要使用函数索引。

B.当访问的数据占整个表中的大部分数据时

C.如果时一个表的 high water mark 数据块数少于初始化参数 DB_FILE_MULTIBLOCK_READ_COUNT

D.本次查询可以用到该张表的一个引用，但是该表具有多个索引包含用于过滤的字段

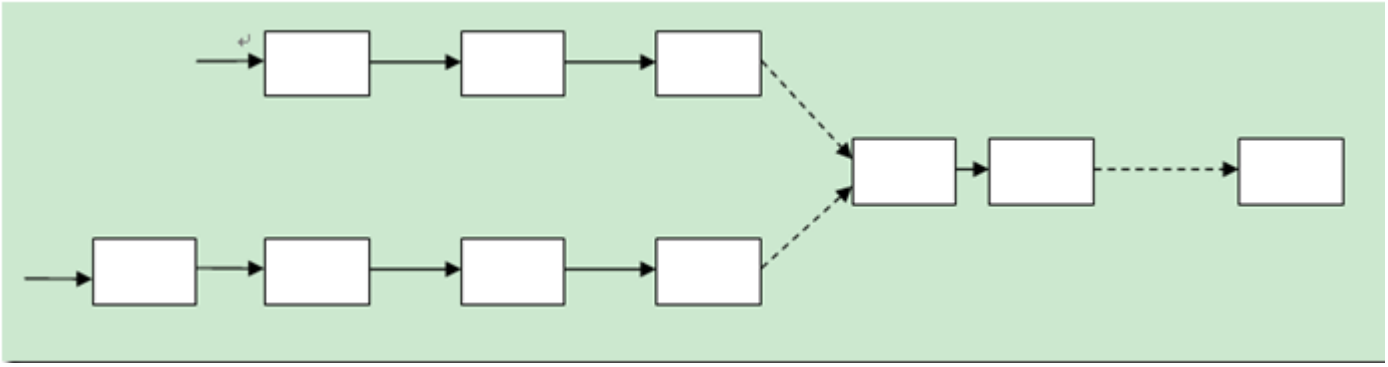
http://www.itheima.com Copyright© 2018 黑马程序员

2. 问答题

2.1 如何判断两个单向链表相交，如何找相交点？

答案参考：http://blog.csdn.net/jiary5201314/article/details/50990349

第一种情况：两个链表均不含有环



思路：

1)、直接法

采用暴力的方法，遍历两个链表，判断第一个链表的每个结点是否在第二个链表中，时间复杂度为O(len1*len2)，耗时很大。

2)、hash 计数法

http://www.itheima.com Copyright© 2018 黑马程序员

如 果 两个链表相交，则两个链表就会有共同的结点；而结点地址又是结点唯一标识。因而判断两个链表中是否存在地址一致的节点，就可以知道是否相交了。可以对第一 个链表的节点地址进行 hash 排序，建立 hash 表，然后针对第二个链表的每个节点的地址查询 hash 表，如果它在 hash 表中出现，则说明两个链表有共 同的结点。这个方法的时间复杂度为：O(max(len1+len2)；但同时还得增加 O(len1)的存储空间存储哈希表。这样减少了时间复杂度，增加 了存储空间。

以链表节点地址为值，遍历第一个链表，使用 Hash 保存所有节点地址值，结束条件为到最后一个节点（无环）或 Hash 中该地址值已经存在（有环）。

再遍历第二个链表，判断节点地址值是否已经存在于上面创建的 Hash 表中。

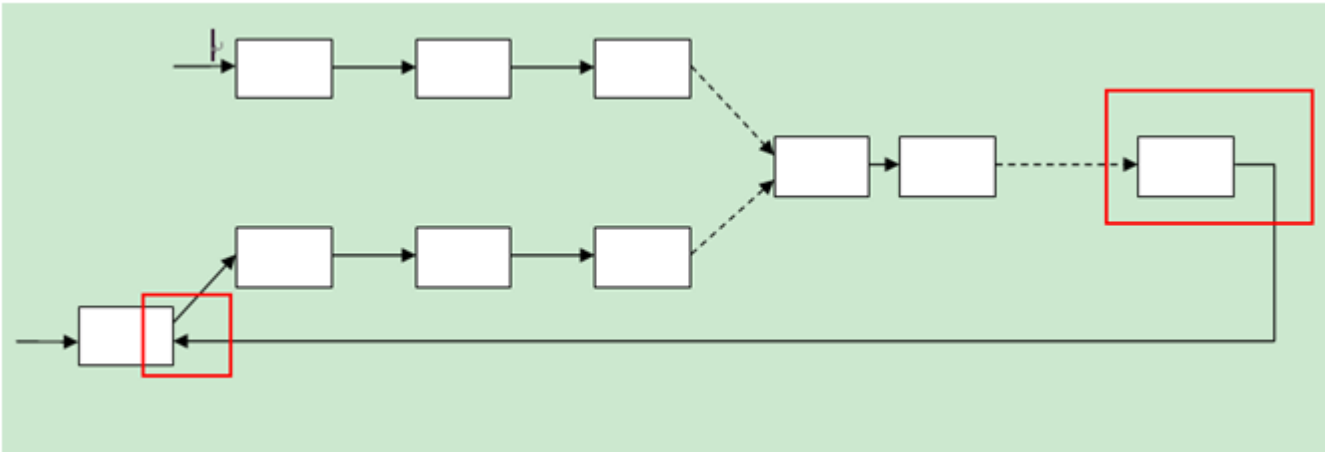
这个方面可以解决题目中的所有情况，时间复杂度为 O(m+n)，m 和 n 分别是两个链表中节点数量。由于节点地址指针就是一个整型，假设链表都是在堆中动态创建的，可以使用堆的起始地址作为偏移量，以地址减去这个偏移量作为 Hash 函数

3)、第三种思路是比较奇特的，在编程之美上看到的。先遍历第一个链表到他的尾部，然后将尾部的 next 指针指向第二个链表(尾部指针的 next 本来指向的是 null)。这样两个链表就合成了一个链表，判断原来的两个链表是否相交也就转变成了判断新的链表是否有环的问题了：即判断单链表是否有环?

这样进行转换后就可以从链表头部进行判断了，其实并不用。通过简单的了解我们就很容易知道，如果新链表是有环的，那么原来第二个链表的头部一定在环上。因此我们就可以从第二个链表的头部进行遍历的，从而减少了时间复杂度(减少的时间复杂度是第一个链表的长度)。

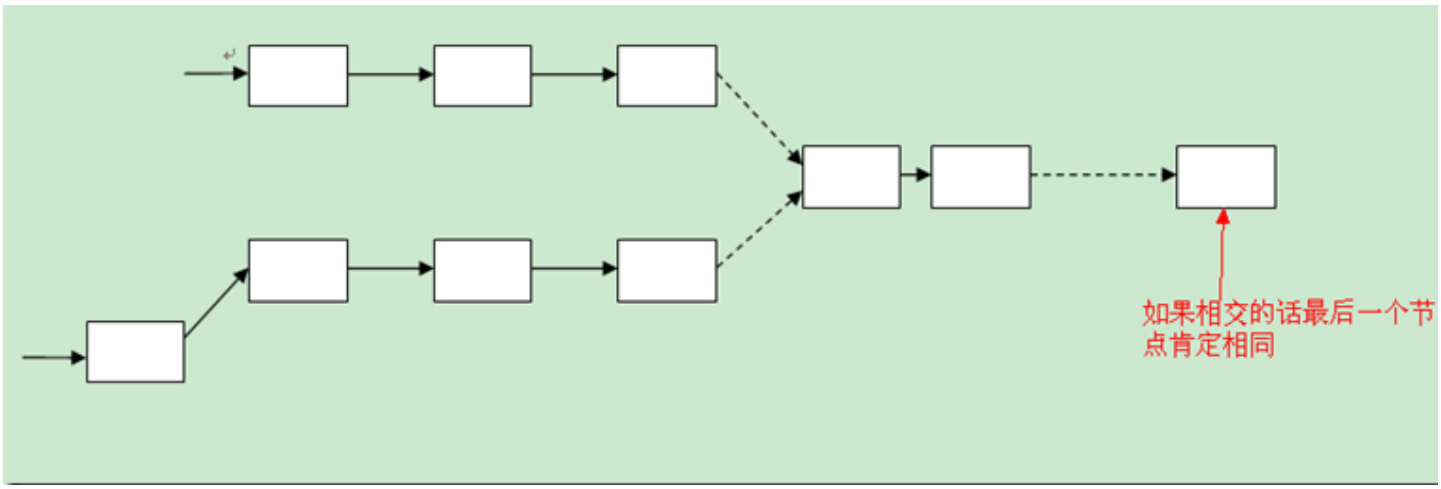
下图是一个简单的演示：

http://www.itheima.com Copyright© 2018 黑马程序员



这种方法可以判断两个链表是否相交，但不太容易找出他们的交点。

4)、仔细研究两个链表，如果他们相交的话，那么他们最后的一个节点一定是相同的，否则是不相交的。因此判断两个链表是否相交就很简单了，分别遍历到两个链表的尾部，然后判断他们是否相同，如果相同，则相交；否则不相交。示意图如下：



判断出两个链表相交后就是判断他们的交点了。假设第一个链表长度为 len1，第二个问 len2，然后找出长度较长的，让长度较长的链表指针向后移动|len1 - len2| (len1-len2 的绝对值)，然后在开始遍历两个链表，判断节点是否相同即可。

2.2 如何从 cookie 中拿到 session?

答案：session 在服务器端，cookie 在客户端（浏览器），session 默认被存在在服务器的一个文件里（不

http://www.itheima.com Copyright© 2018 黑马程序员

是内存），session 的运行依赖 session id，而 session id 是存在 cookie 中的，也就是说，如果浏览器禁用了 cookie ，同时 session 也会失效（但是可以通过其它方式实现，比如在 url 中传递 session_id）；session 可以放在 文件、数据库、或内存中都可以。

用户验证这种场合一般会用 session ，因此，维持一个会话的核心就是客户端的唯一标识，即 session id

2.3. System.out.println(3/2); System.out.println(3.0/2); System.out.println(3.0/2.0); 分别会打印什么结果？

答案：1, 1.5，1.5

2.4 打印出下面两个数组的交集，结果不能重复

String[] arr1 = {"112","wqw","2121"};

String[] arr2 = {"112","aad","ewqw"};

打印出两个数组的交集，结果不能重复

答案：

```
public class test {
    @Test//测试程序
   public void test()
        String[] arr1 = {"112","wqw","2121"};
      String[] arr2 = {"112","aad","ewqw"};
       String[] result=StringIntersection(arr1,arr2);
        for (String str:result){
           System.out.printf(str);
        }
    }
   //取两个 string 数组的交集
   public  String[] StringIntersection(String[] arr1,String[] arr2){
        Map<String,Boolean> map = new HashMap<String,Boolean>();
       List<String> list = new LinkedList<String>();
        //取出 str1 数组的值存放到 map 集合中，将值作为 key，所以的 value 都设置为 false
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
        for (String str1:arr1){
            if (!map.containsKey(str1)){
                map.put(str1,Boolean.FALSE);
            }
        }
        //取出 str2 数组的值循环判断是否有重复的 key，如果有就将 value 设置为 true
        for (String str2:arr2){
            if (map.containsKey(str2)){
                map.put(str2,Boolean.TRUE);
            }
        }
        //取出 map 中所有 value 为 true 的 key 值，存放到 list 中
        for (Map.Entry<String,Boolean> entry:map.entrySet()){
            if (entry.getValue().equals(Boolean.TRUE)){
                list.add(entry.getKey());
            }
        }
        //声明 String 数组存储交集
        String[] result={};
        return list.toArray(result);
    }
}
```

2.5 SpringMvc 拦截器用过吗？什么场景会用到，过滤器，拦截器，监听器有什么区别？

拦截器是指通过统一拦截从浏览器发往服务器的请求来完成功能的增强。

使用场景：解决请求的共性问题（乱码问题、权限验证问题）

过滤器

Servlet 中的过滤器 Filter 是实现了 javax.servlet.Filter 接口的服务器端程序，主要的用途是过滤字符编码、做一些业务逻辑判断等。其工作原理是，只要你在 web.xml 文件配置好要拦截的客户端请求，它都会帮你拦截到请求，此时你就可以对请求或响应(Request、Response)统一设置编码，简化操作；同时还可进行逻辑判断，如用户是否已经登陆、有没有权限访问该页面等等工作。它是随你的 web 应用启动而启动的，只初始化一次，以后就可以拦截相关请求，只有当你的 web 应用停止或重新部署的时候才销毁。

http://www.itheima.com Copyright© 2018 黑马程序员

监听器

现在来说说 Servlet 的监听器 Listener，它是实现了 javax.servlet.ServletContextListener 接口的服务器端程序，它也是随 web 应用的启动而启动，只初始化一次，随 web 应用的停止而销毁。主要作用是： 做一些初始化的内容添加工作、设置一些基本的内容、比如一些参数或者是一些固定的对象等等

拦截器

拦截器是在面向切面编程中应用的，就是在你的 service 或者一个方法前调用一个方法，或者在方法后调用一个方法。是基于 JAVA 的反射机制。拦截器不是在 web.xml

1).过滤器：所谓过滤器顾名思义是用来过滤的，在 java web 中，你传入的 request,response 提前过滤掉一些信息，或者提前设置一些参数，然后再传入 servlet 或者 struts 的 action 进行业务逻辑，比如过滤掉非法 url（不是 login.do 的地址请求，如果用户没有登陆都过滤掉）,或者在传入 servlet 或者 struts 的 action 前统一设置字符集，或者去除掉一些非法字符（聊天室经常用到的，一些骂人的话）。filter 流程是线性的， url 传来之后，检查之后，可保持原来的流程继续向下执行，被下一个 filter, servlet 接收等.

2).监听器：这个东西在 c/s 模式里面经常用到，他会对特定的事件产生产生一个处理。监听在很多模式下用到。比如说观察者模式，就是一个监听来的。又比如 struts 可以用监听来启动。Servlet 监听器用于监听一些重要事件的发生，监听器对象可以在事情发生前、发生后可以做一些必要的处理。

3).java 的拦截器 主要是用在插件上，扩展件上比如 hivernate spring struts2 等 有点类似面向切片的技术，在用之前先要在配置文件即 xml 文件里声明一段的那个东西。

2.6 ThreadLocal 的原理和应用场景

参考文档：http://blog.csdn.net/sonny543/article/details/51336457

http://www.itheima.com Copyright© 2018 黑马程序员

每一个 ThreadLocal 能够放一个线程级别的变量，可是它本身能够被多个线程共享使用，并且又能够达到线程安全的目的，且绝对线程安全。

ThreadLocal 的应用场景：

最常见的 ThreadLocal 使用场景为 用来解决数据库连接、Session 管理等

2.7 简述 TCP 的三次握手

在 TCP/IP 协议中,TCP 协议提供可靠的连接服务,采用三次握手建立一个连接。

1).第一次握手：建立连接时,客户端发送 syn 包(syn=j)到服务器,并进入 SYN_SEND 状态,

等待服务器确认； SYN：同步序列编号(Synchronize Sequence Numbers)

2).第二次握手：服务器收到 syn 包,必须确认客户的 SYN（ack=j+1）,同时自己也发送一个 SYN 包（syn=k）,即 SYN+ACK 包,此时服务器进入 SYN_RECV 状态；

3)第三次握手：客户端收到服务器的 SYN + ACK 包,向服务器发送确认包 ACK(ack=k+1)， 此包发送完毕,客户端和服务器进入 ESTABLISHED 状态,完成三次握手。

http://www.itheima.com Copyright© 2018 黑马程序员

完成三次握手,客户端与服务器开始传送数据。

2.8 SpringMVC request 接收设置是线程安全的吗？

参考文档：http://blog.csdn.net/q1512451239/article/details/53122512

答案：是线程安全的，request、response 以及 requestcontext 在使用时不需要进行同步。而根据 spring 的默认规则，controller 对于 beanfactory 而言是单例的。即 controller 只有一个， controller 中的 request 等实例对象也只有一个

2.9 列举 Maven 常见的六种依赖范围

答案：

1.compile:编译依赖范围(默认),对其三种都有效

2.test:测试依赖范围,只对测试 classpath 有效

3.runtime:运行依赖范围,只对测试和运行有效,编译主代码无效,例如 JDBC

4.provided:已提供依赖范围,只对编译和测试有效,运行时无效,例如 selvet-api

http://www.itheima.com Copyright© 2018 黑马程序员

5.system:系统依赖范围.谨慎使用.例如本地的,maven 仓库之外的类库文件

6.import(maven2.0.9 以上):导入依赖范围,不会对其他三种有影响

2.10 Mybatis 如何防止 sql 注入？mybatis 拦截器了解过吗，应用场景是什么？

答案：Mybatis 使用#{}经过预编译的，是安全的，防止 sql 注入。

Mybatis 拦截器只能拦截四种类型的接口：Executor、StatementHandler、

ParameterHandler 和 ResultSetHandler。这是在 Mybatis 的 Configuration 中写死了的，如果要支持拦截其他接口就需要我们重写 Mybatis 的 Configuration。Mybatis 可以对这四个接口中所有的方法进行拦截。Mybatis 拦截器常常会被用来进行分页处理。

2.11 简单解释自动装配的各种模式，或者叫装配方式。

在 Spring 框架中共有 5 种自动装配:

no：这是 Spring 框架的默认设置，在该设置下自动装配是关闭的，开发者需要自行在 bean 定义中用标签明确的设置依赖关系。

byName：该选项可以根据 bean 名称设置依赖关系。当向一个 bean 中自动装配一个属 性时，容器将根据 bean 的名称自动在在配置文件中查询一个匹配的 bean。如果找到的 话，就装配这个属性，如果没找到的话就报错。

byType：该选项可以根据 bean 类型设置依赖关系。当向一个 bean 中自动装配一个属性时，容器将根据 bean 的类型自动在在配置文件中查询一个匹配的 bean。如果找到的话，就装配这个属性，如果没找到的话就报错。

constructor：造器的自动装配和 byType 模式类似，但是仅仅适用于与有构造器相同参数 的 bean，如果在

http://www.itheima.com Copyright© 2018 黑马程序员

容器中没有找到与构造器参数类型一致的 bean，那么将会抛出异常。

autodetect：该模式自动探测使用构造器自动装配或者 byType 自动装配。首先，首先会 尝试找合适的带参数的构造器，如果找到的话就是用构造器自动装配，如果在 bean 内部没有找到相应的构造器或者是无参构造器，容器就会自动选择 byTpe 的自动装配方式。

2.12 mvc 的各个部分都有哪些技术来实现？如何实现的？

MVC 是 Model - View - Controller 的简写。 Model 代表的是应用的业务逻辑（ 通过 JavaBean， EJB 组件实现）， View 是应用的表示面（ 由 JSP 页面产生）， Controller 是提供应用的处理过程控制（ 一般是一个 Servlet）， 通过这种设计模型把应用逻辑， 处 理过程和显示逻辑分成不同的组件实现。 这些组件可以进行交互和重用。

2.13 反射机制一般应用在什么场景？

答案： 反射机制的应用场景：

1)逆向代码 ，例如反编译

3)与注解相结合的框架 例如 Retrofit

4)单纯的反射机制应用框架 例如 EventBus 2.x

5)动态生成类框架 例如 Gson

2.14 设计 Java 程序，假设有 50 瓶饮料，喝完三个空瓶可以换一瓶饮料，依次类推，请问总共喝了多少饮料。

答案：

```
public class Buy {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
public static void main(String[] args) {
int n = 50;          //初始饮料总数
int i=0;            //兑换次数
while(true){
   n -= 3;           //喝 3 瓶
   n++;             //兑换 1 瓶
   i++;             //兑换次数+1
  if(n<3)
  {
  System.out.println ("共喝了"+(50+i)+"瓶");
  break;
  }
}
}
}
```

2.15 根据某年某月某日，输出这是一年中第几天。

答案:

```
/定义存储年月日的变量
    int year = 0, month = 0, day = 0;
    //提示用户输入
    printf("请输入年月日(比如 1990-1-1):");
    //接受用户输入, 切记,scanf 中""内的格式是什么, 输入的格式必须一致
    scanf("%d-%d-%d", &year,&month, &day);
    //定义一个数组存放每个月的天数
    int dayOfMonth[12] = {31, 28, 31, 30, 31,30, 31, 31, 30, 31, 30, 31};
    //判断是否不是闰年
    if (year % 400 == 0 || (year % 4 == 0&& year % 100 != 0)) {
    //闰年二月 29 天
    dayOfMonth[1] = 29;
    }
//定义变量记录是第几天
    int whichDay = 0;
    //例如 3 月 20 日是一年的第几天, 计算方法 1 月的天数 + 2 月的天数 + 20
    for (int i = 0; i < month - 1; i++) {
    whichDay += dayOfMonth[i];
    }
    whichDay += day;
    printf("是一年的%d 天\n", whichDay);
```


http://www.itheima.com Copyright© 2018 黑马程序员

2.16 利润与奖金，某公司销售 10 万元到 20 万元的奖金 10%，在 20 万元的奖金 10 万元以上的奖金 7.5%，到 40 万元超出 20 万元的部分奖金为 5%,到 60 万元的超出 40 万元的部分奖金 3%，到 100 万元的超出 60 万元部分奖金 1%，请输出说的奖金。

答案:

```
public class Test1 {


    public static void main(String[] args) {
        float jiangjin=0;
        Scanner scan=new Scanner(System.in);
        System.out.print("请输入利润：");
        float num=scan.nextInt();
        if(num<=100000){
            jiangjin=(float) (num*0.1);
        }
        else if(num<=200000){
            jiangjin=(float) ((num-100000)*0.075+100000*0.1);
        }
        else if(num<=400000){

            jiangjin=(float) ((num-200000)*0.5 +100000*0.175);
        }
        else if(num<=600000){
            jiangjin=(float) ((num-400000)*0.3 +100000*0.175+200000*0.5);
        }
        else if(num<=1000000){
            jiangjin=(float) ((num-600000)*0.015 +100000*0.175+200000*0.5+200000*0.3);
        }
        else{
            jiangjin=(float) ((num-1000000)*0.01 +100000*0.175+200000*0.5+200000*0.3+400000*0.015);
        }
        System.out.println("奖金："+jiangjin);
    }

}
```


http://www.itheima.com Copyright© 2018 黑马程序员

2.17 请描述工作流机制（JBPM/BPEL）等

参考：http://blog.csdn.net/leroy008/article/details/8058187

2.18 除了懒汉式和饿汉式你还了解那些单利模式？

答案：

```
 第一种： 双重查锁模式
public class DoubleCheckLock {
private static DoubleCheckLock instance = null;


private DoubleCheckLock() {
}


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


Copyright© 2018 黑马程序员

```
}
}
第二种：枚举单例
public enum SingleEnum {

INSTANCE;
public void doSomething() {
ToastUtils.showLongToast("do something...");

}
}
第三种：静态内部类方式
public class StaticInner {

private static StaticInner instance;
public static StaticInner getInstance() {
return SingletonHolder.STATIC_INNER;
}
private static class SingletonHolder {
private static final StaticInner STATIC_INNER = new StaticInner();
}
}
```

2.19 简述 SSH 的概念以及中主要的设计思想？

SSH 是 struts+spring+hibernate 的一个集成框架，是目前比较流行的一种 Web 应用程序开源框架。

集成 SSH 框架的系统从职责上分为四层：表示层、业务逻辑层、数据持久层和域模块层，以帮助开发人员在短期内搭建结构清晰、可复用性好、维护方便的 Web 应用程序。其中使用 Struts 作为系统的整体基础架构，负责 MVC 的分离，在 Struts 框架的模型部分，控制业务跳转，利用 Hibernate 框架对持久层提供支持，Spring 做管理，管理 struts 和 hibernate。具体做法是：用面向对象的分析方法根据需求提出一些模型，将这些模型实现为基本的 Java 对象，然后编写基本的 DAO(Data Access Objects)接口，并给出 Hibernate 的 DAO 实现，采用 Hibernate 架构实现的 DAO 类来实现 Java 类与数据库之间的转换和访问，最后由 Spring 做管理

http://www.itheima.com Copyright© 2018 黑马程序员

20..unix 下如何让命令在后台执行？

要让程序在后台执行，只需在命令行的最后加上“&”符号。

例如：$ find . -name abc -print&;

2.21 rm-i 与 rm-r 个实现什么功能？

rm -i --interactive 交互模式删除文件，删除文件前给出提示。

rm -r -r 或-R：递归处理，将指定目录下的所有文件与子目录一并处理。

2.22 什么是乐观锁，什么是悲观锁，两者的区别是什么？

悲观锁(Pessimistic Lock)，顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会 block 直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

乐观锁(Optimistic Lock)，顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于 write_condition 机制的其实都是提供的乐观锁。

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行 retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

http://www.itheima.com Copyright© 2018 黑马程序员

2.23 日志打印的 log4j 的配置中%t 表示什么？

答案：%t 输出产生该日志事件的线程名

扩展：%M 是输出方法的名字、%m 是输出代码指定的日志信息。

指定的打印信息的具体格式 ConversionPattern，具体参数：

%m 输出代码中指定的消息

%p 输出优先级，即 DEBUG，INFO，WARN，ERROR，FATAL

%r 输出自应用启动到输出该 log 信息耗费的毫秒数

%c 输出所属的类目，通常就是所在类的全名

%t 输出产生该日志事件的线程名

%n 输出一个回车换行符，Windows 平台为"rn”，Unix 平台为"n”

%d 输出日志时间点的日期或时间，默认格式为 ISO8601，也可以在其后指定格式，比如：%d{yyyy MM dd HH:mm:ss,SSS}，输出类似：2002 年 10 月 18 日 22：10：28，921 %l 输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数。

%x: 输出和当前线程相关联的 NDC(嵌套诊断环境),尤其用到像 java servlets 这样的多客户多线程的应用中。

%%: 输出一个”%”字符

%F: 输出日志消息产生时所在的文件名称

%M: 输出执行方法

%L: 输出代码中的行号

http://www.itheima.com Copyright© 2018 黑马程序员

2.24 Spring 中什么时候引起 NotWritablePropertyException 和 Could not open calss path resource[ApplicationContext.xml]

出现 NotWritablePropertyException 异常的原因一般是在 ApplicationContext.xml 中 property name 的错误等相关错误。

七、华胜天成（2017-12-11-wzy）

1. 不定项选择题

1.1 关于 Web 应用程序，下列说法错误的是（b）。

a) WEB-INF 目录存在于 web 应用的根目录下

b) WEB-INF 目录与 classes 目录平行

c) web.xml 在 WEB-INF 目录下

d) Web 应用程序可以打包为 war 文件

解释：classes 目录位于 WEB-INF 目录之下，而不是平行的关系。

1.2 有关 Servlet 的生命周期说法正确的有（ cd）。

a) Servlet 的生命周期由 Servlet 实例控制

b) init()方法在创建完 Servlet 实例后对其进行初始化，传递的参数为实现 ServletContext 接口的对象

c) service()方法响应客户端发出的请求

d) destroy()方法释放 Servlet 实例

解释：Servlet 的生命周期是由 Servlet 容器（Tomcat 就是常见的 Servlet 容器）管理的，因此 a 不对。

http://www.itheima.com Copyright© 2018 黑马程序员

Servlet 中的 init（）方法有两个重载，一个是空参的，另外一个是带 ServletConfig 形参的，而不是 ServletContext，因此 b 不对。

关于 d 选项，说法其实并不好，正确的表达应该是因为 Servlet 实例要释放（销毁）了，才会先调用 destroy（）方法。

1.3 有关会话跟踪技术描述正确的是（ abc）。

a) Cookie 是 Web 服务器发送给客户端的一小段信息，客户端请求时，可以读取该信息发送到服务器端

b) 关闭浏览器意味着会话 ID 丢失，但所有与原会话关联的会话数据仍保留在服务器上，直至会话过期

c) 在禁用 Cookie 时可以使用 URL 重写技术跟踪会话

d) 隐藏表单域将字段添加到 HTML 表单并在客户端浏览器中显示

1.4 以下 web.xml 片断（ d ）正确地声明 servlet 上下文参数。

a)
```
<init-param>
    <param-name>MAX</param-name>
    <param-value>100</param-value>
</init-param>
```

b)
```
<context-param>
    <param name="MAX" value="100" />
<context-param>
```

c)
```
<context>
```


http://www.itheima.com Copyright© 2018 黑马程序员

<param name="MAX" value="100" />

<context>

d) <context-param>

<param-name>MAX</param-name>

<param-value>100</param-value>

<context-param>

补充：关于 context-param 和 init-param 的作用，请参考下文。

https://www.cnblogs.com/hzj-/articles/1689836.html

1.5 以下（a）可用于检索 session 属性 userid 的值。

a) session. getAttribute（“userid”）;

b) session. setAttribute（“userid”）;

c) request. getParameter（“userid”）;

Copyright© 2018 黑马程序员

d) request. getAttribute (“userid”);

1.6 下列 JSP 代码，以下（cd）可放置在//1 处，不会发生编译错误。

```
<html>
    <body>
    <%
        for(int i = 0; i < 10; i++) {
            //1
        }
    %>
    </body>
</html>
```

a) <%= i %>

b) i

c) %><%= i %><%

d) 不写任何内容

1.7 考虑下面两个 JSP 文件代码片断：

test1.jsp:

http://www.itheima.com Copyright© 2018 黑马程序员

```
<HTML>
    <BODY>
    <% pageContext.setAttribute(“ten”,new Integer(10));%>
    //1
    </BODY>
</HTML>
```

test2.jsp:

数字为：<%= pageContext.getAttribute(“ten”)%>

以下（ c ）放置在 test1.jsp 中的//1 处，当请求 test1.jsp 时正确输出 test2.jsp 中的内容。

a) <jsp:include page=”test2.jsp” />

b) <jsp:forword page=”test2.jsp” />

c) <%@ include file=”test2.jsp” %>

d) 由于 pageContext 对象的 scope 属性为 page,所以 test2.jsp 不能访问 test1.jsp 定义的属性

解释：JSP 中 include 的两种方法，1、<jsp:include page=”test2.jsp” />属于动态引入/2、<%@ include file=”test2.jsp” %>属于静态引入。两者的区别如下：

1.执行时间上的区别

<%@ include file=”test2.jsp”%> 是在翻译阶段执行（将 JSP 页面转换成 servlet 的阶段）。通俗的话讲就是先合并在一起，然后再编译成一个 Servlet 文件。

<jsp:include page=”test2.jsp”/> 在请求处理阶段执行。通俗的话讲就是先各自编译，然后在处理请求的时候结果再合并到一起。

2.引入内容的方式区别

http://www.itheima.com Copyright© 2018 黑马程序员

<%@ include file=”test2.jsp”%>是纯粹的把部分代码写到了另一页面（或者说是共享），而那另一页面中不能有相同的变量名，但可以借用主页面的内容。

<jsp:include page=”test2.jsp”/>引入执行页面或 servlet 所生成的应答文本。

<jsp:forword page=”test2.jsp” />属于请求转发，由于 pageContext 对象的 scope 属性为 page,所以如果请求转发了，那么 test2.jsp 就不能访问 test1.jsp 定义的属性了。所以该题选 C。

1.8 有关 JSP 隐式对象，以下（acd ）描述正确。

a) 隐式对象是 WEB 容器加载的一组类的实例，可以直接在 JSP 页面使用

b) 不能通过 config 对象获取 ServletContext 对象

c) response 对象通过 sendRedirect 方法实现重定向

d) 只有在出错处理页面才有 exception 对象

解释: jsp 的九大内置对象分别是: config、request、response、out, page、pageContext、session、exception、application。其中 exception 是特殊的内置对象，只有当在 jsp 中添加 isErrorPage="true"属性时如下配置时才可以使用。该属性一般出现在设定的错误界面。

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" isErrorPage="true" %>
```

1.9 考虑下面 JSP 文件代码片断：

<HTML>

<BODY>

http://www.itheima.com Copyright© 2018 黑马程序员

```
<jsp:include page=" test2.jsp" >
    <jsp:param name=" username"  value=" zhangsan" />
</jsp:include>
</BODY>
</HTML>
```

以下（ c）代码片断放置在 test2.jsp 中不会导致错误。

a) <jsp:getParam name=" username" />

b) <jsp:include param =" username" />

c) <%=request.getParameter("username")%>

d) <%=request.getAttribute("username")%>

解释：<jsp:include page=" test2.jsp" >属于动态调用 test2.jsp 界面，相当于动态去请求 test2.jsp 所生成的 Servlet，在请求的同时携带了请求参数 "username" ，我们知道在 Servlet 中获取请求携带的参数就是通过 request.getParameter(key)来获取的，因此 C 正确。

1.10 以下是 login.jsp 文件的代码片断：

```
<%@ page isELIgnored="false"%>
<html>
    <body>
        <FORM action="login.jsp" method="GET">
          <input type="text" name="name" value="${param['name']}">
          <input type="submit" value="提交">
```


Copyright© 2018 黑马程序员

```
        </FORM>
    <P>
        用户名为: ${param.name}
    </body>
</html>
```

以下（C ）描述正确。

a) 发生运行错误

b) 页面会出现一文本框，并且文本框中内容为${param['name']}

c) 当用户输入名字并单击“提交”按钮时，在同一页面中的“用户名为：”字样后面会显示用户输入的内容

d) 当用户输入名字并单击“提交”按钮时，在同一页面中的“用户名为：”字样后面会显示${param.name}

1.11 doAfterBody()方法是在（ b ）接口中定义的。

a) Tag

b) IterationTag

c) BodyTag

d) TagSupport

解释：该知识点属于 jsp 自定义标签中的知识，比较生僻。如果想了解如何在 jsp 中使用自定义标签的可以参考下文。

https://www.cnblogs.com/flying607/p/5063207.html

http://www.itheima.com Copyright© 2018 黑马程序员

1.12 编写一个 Filter，需要（ b）。

a) 继承 Filter 类

b) 实现 Filter 接口

c) 继承 HttpFilter 类

d) 实现 HttpFilter 接口

1.13 有关 MVC 设计模式（b ）描述不正确。

a) 使用 Servlet 作为控制器

b) MVC 设计模式增大了维护难度

c) MVC 设计模式属于 Model 2

d) 模型对象向客户端显示应用程序界面

补充知识：

http://www.itheima.com Copyright© 2018 黑马程序员

Model 1

Model 1 的基础是 JSP 文件，它由一些相互独立的 JSP 文件，和其他一些 Java Class 组成（不是必须的）。这些 JSP 从 HTTP Request 中获得所需要的数据，处理业务逻辑，然后将结果通过 Response 返回前端浏览器。

Model 2

采用面向对象技术实现 MVC 模式从而扩展 JSP/Servlet 的模式被成为是 Model 2 模式。Apache Jakarta 项目中 Struts 是一个实现 Model 2 的很好的框架，它通过一些 Custom Tag Lib 处理表现层，用 ActionFrom Bean 表示数据，用自己提供的一个 ActionServlet 作为控制器实现页面的流转的控制功能。

说的直白一些，model1 即为单纯的 jsp+java，没有框架参与，通过 response 和 request 对象传送值域，而 model2,则使用较为流行的 struts2 框架。

1.14 在 Linux 中，可以使用命令（c）加挂计算机上的非 Linux 文件系统。

a) cat /proc/filesystems

b) ln

c) mount

d) df

1.15 下面关于 Linux 中 shell 的说法错误的是（d）。

a) shell 是解释用户在终端键入的命令的一种中间程序

b) shell 可以读取并执行脚本文件中的命令

c) 用户可以使用参数将命令行的参数传递给 shell 脚本，从而实现在 Linux 中的交互式编程

d) 默认情况下，Linux 中创建的所有文件都具有执行权限

http://www.itheima.com Copyright© 2018 黑马程序员

1.16 在 Oracle 中，当需要使用显式游标更新或删除游标中的行时，UPDATE 或 DELETE 语句必须使用（c）子句。

a) WHERE CURRENT OF

b) WHERE CURSOR OF

c) FOR UPDATE

d) FOR CURSOR OF

1.17 在 Oracle 中，使用下列的语句 CREATE PUBLIC SYNONYM parts FOR Scott.inventory;完成的任务是（d）。

a) 将 Scott.inventory 对象的访问权限赋予所有用户

b) 指定了新的对象权限

c) 指定了新的系统权限

d) 给 Scott.inventory 对象创建一个公用同义词 parts

1.18 在 Oracle 中，执行如下 PL/SQL 语句后

```
CREATE TYPE car AS OBJECT ( id NUMBER, model VARCHAR2(25), color VARCHAR2(15) );
...
DECLARE
  myvar car.model%TYPE;
BEGIN
...
```


http://www.itheima.com Copyright© 2018 黑马程序员

END;

变量 myvar 的数据类型为（c）。

a) NUMBER

b) car 类型

c) VARCHAR2

d) OBJECT

解释：该知识点属于 PL/SQL 高级编程，中的对象类型（OBJECT TYPE）

2. 简答题

2.1 List、Map、Set 三个接口存储元素时各有什么特点？

2.2 在项目中用过 Spring 的哪些方面？及用过哪些 Ajax 框架？

八、诚迈（2017-12-7-lyq）

1. 选择题

1.1、下列说法正确的有（C）

A、class 中的 constructor 不可忽略

B、constructor 可以作为普通方法被调用

http://www.itheima.com Copyright© 2018 黑马程序员

C、constructor 在一个对象被 new 时被调用

D、一个 class 只能定义一个 constructor

1.2、下列运算符合法的是（D）

A、&&

B、<>

C、If

D、:=

1.3、下列哪种说法不正确（ABC）

A、实例方法可以直接调用超类的实例方法

B、实例方法可以直接调用超类的类方法

C、实例方法可以直接调用其他类的实例方法

D、实例方法可以直接调用本类的类方法

1.4、执行如下程序代码后，c 的值是（C）

```
a=0;
c=0;
do{
    --c;
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
    a=a-1;
}while(a>0);
```

A、0 B、1 C、-1 D、死循环

2. 判断题

2.1、（×）constructor 必须与 class 同名，但是方法不能与 class 同名

2.2、（×）constructor 可以被继承，因此可以重写 Overriding，也可以被重载 Overloading

2.3、（√）String 类是 final 类 故不可以被继承

2.4、（×）数组的大小可以任意改变，又称动态数组

2.5、（×）try{}里有一个 return 语句，则紧跟在 try 后面的 finally{}里的 code 将在 return 后执行

3. 简答题

3.1、抽象类和接口有什么不同点？

接口和抽象类的概念不一样。接口是对动作的抽象，抽象类是对根源的抽象。

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。比如，男人，女人，这两个类（如果是类的话……），他们的抽象类是人。说明，他们都是人。

人可以吃东西，狗也可以吃东西，你可以把“吃东西”定义成一个接口，然后让这些类去实现它.

不同点:

http://www.itheima.com Copyright© 2018 黑马程序员

参数	抽象类	接口
默认的方法实现	它可以有默认的方法实现	接口完全是抽象的。它根本不存在方法的实现
实现	子类使用 **extends** 关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。	子类使用关键字 **implements** 来实现接口。它需要提供接口中所有声明的方法的实现
构造器	抽象类可以有构造器	接口不能有构造器
与正常 Java 类的区别	除了你不能实例化抽象类之外，它和普通 Java 类没有任何区别	接口是完全不同的类型
访问修饰符	抽象方法可以有 **public**、**protected** 和 **default** 这些修饰符	接口方法默认修饰符是 **public**。你不可以使用其它修饰符。
main 方法	抽象方法可以有 main 方法并且我们可以运行它	接口没有 main 方法，因此我们不能运行它。
多继承	抽象方法可以继承一个类和实现多个接口	接口只可以继承一个或多个其它接口
速度	它比接口速度要快	接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。
添加新方法	如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。	如果你往接口中添加方法，那么你必须改变实现该接口的类。

http://www.itheima.com Copyright© 2018 黑马程序员

3.1.1 什么时候使用抽象类和接口

如果你拥有一些方法并且想让它们中的一些有默认实现，那么使用抽象类吧。

如果你想实现多重继承，那么你必须使用接口。由于 Java 不支持多继承，子类不能够继承多个类，但可以实现多个接口。因此你就可以使用接口来解决它。

如果基本功能在不断改变，那么就需要使用抽象类。如果不断改变基本功能并且使用接口，那么就需要改变所有实现了该接口的类。

3.2、sleep（）和 wait（）有什么不同点？

主要区别在于：

sleep 表示睡眠，wait 表示等待 sleep 需要手动唤醒线程，而 wait 不需要手动唤醒等待结束后自动执行。

原文链接：https://www.zhihu.com/question/23328075

3.3、TreeMap 和 HashMap 有什么不同点？

第一点：HashMap：基于哈希表实现。使用 HashMap 要求添加的键类明确定义了 hashCode()和 equals()[可以重写

Copyright© 2018 黑马程序员

hashCode()和 equals()]，为了优化 HashMap 空间的使用，您可以调优初始容量和负载因子。

TreeMap：基于红黑树实现。TreeMap 没有调优选项，因为该树总处于平衡状态。

第二点：HashMap 通过 hashcode 对其内容进行快速查找。

TreeMap 中所有的元素都保持着某种固定的顺序，如果你需要得到一个有序的结果你就应该使用 TreeMap（HashMap 中元素的排列顺序是不固定的）。

第三点：HashMap 非线程安全

TreeMap 线程安全

第四点：HashMap：适用于在 Map 中插入、删除和定位元素。

Treemap：适用于按自然顺序或自定义顺序遍历键(key)。

总结：HashMap 通常比 TreeMap 快一点(树和哈希表的数据结构使然)，建议多使用 HashMap，在需要排序的 Map 时候才用 TreeMap

原文链接：https://www.cnblogs.com/ouwenkgwpf/p/4547509.html

Copyright© 2018 黑马程序员

3.4、throws 和 throw 有什么不同点？

throw 是语句抛出一个异常。

语法：throw (异常对象);

throw e;

throws 是方法可能抛出异常的声明。(用在声明方法时，表示该方法可能要抛出异常)

语法：[(修饰符)](返回值类型)(方法名)([参数列表])[throws(异常类)]{.....}

public void doA(int a) throws Exception1,Exception3{.....}

3.5、length（）和 length 有什么不同点？

java 中的 length 属性是针对数组说的,比如说你声明了一个数组,想知道这个数组的长度则用到了 length 这个属性。

java 中的 length()方法是针对字符串 String 说的,如果想看这个字符串的长度则用到 length()这个方法。

4. 编程题

4.1、请编写一个 jdbc 查询任意一个数据库（如 mysql、Oracle 等），数据库名为 test，表明为 user，只有一个字段。

```
1.    public class MysqlDemo {
2.        public static void main(String[] args) throws Exception {
3.          Connection conn = null;
4.          String sql;
5.          // MySQL 的 JDBC URL 编写方式：jdbc:mysql://主机名称：连接端口/数据库的名称?参数=值
6.          // 避免中文乱码要指定 useUnicode 和 characterEncoding
7.          String url = "jdbc:mysql://localhost:3306/test?"
8.                  + "user=root&password=root&useUnicode=true&characterEncoding=UTF8";
9.
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
10.         try {
11.             // 之所以要使用下面这条语句，是因为要使用 MySQL 的驱动，所以我们要把它驱动起来，
12.             // 可以通过 Class.forName 把它加载进去，也可以通过初始化来驱动起来，下面三种形式都可以
13.             Class.forName("com.mysql.jdbc.Driver");// 动态加载 mysql 驱动
14.             // or:
15.             // com.mysql.jdbc.Driver driver = new com.mysql.jdbc.Driver();
16.             // or:
17.             // new com.mysql.jdbc.Driver();
18.
19.             System.out.println("成功加载 MySQL 驱动程序");
20.             // 一个 Connection 代表一个数据库连接
21.             conn = DriverManager.getConnection(url);
22.             // Statement 里面带有很多方法，比如 executeUpdate 可以实现插入，更新和删除等
23.             Statement stmt = conn.createStatement();
24.             sql = "select * from user";
25.             ResultSet rs = stmt.executeQuery(sql);
26.             while (rs.next()) {//遍历 user 表中所有数据
27.                     String name = rs.getString("name") ;
28.                     System.out.println("姓名是："+name);// 假如 user 表中的字段为 name
29.                 }
30.         }catch (Exception e) {
31.             e.printStackTrace();
32.         } finally {
33.             try {
34.                 if (rs != null)
35.                     rs.close(); // 关闭结果数据集
36.                 if (stmt != null)
37.                     stmt.close(); // 关闭执行环境
38.                 if (conn != null)
39.                     conn.close(); // 关闭数据库连接
40.             } catch (SQLException e) {
41.                 e.printStackTrace();
42.             }
43.         }
44.
45.     }
46.
47. }
```

注意：连接 oracle 与 mysql 不同点为，url=jdbc:Oracle:thin:@localhost:1521:orcl

driver = oracle.jdbc.driver.OracleDriver

http://www.itheima.com Copyright© 2018 黑马程序员

4.2、请编写一 socket 的程序，客户端向服务器端发送字符串，服务器端在返回相同的字符串。（类似 echo 功能）

```
1.    public class Client { //客户端
2.      public static void main(String args[])throws Exception
3.      {
4.          String clientMessage;//来自用户输入的的信息
5.          String serverMessage; //服务器端的信息
6.          Socket ClientSocket=new Socket("127.0.0.0",5557);//参数是本机地址和端口,客户端套接字，发起 TCP 连接
7.          BufferedReader fromUser=new BufferedReader(new InputStreamReader(System.in));//字符读取流，获取从键盘输入的字符
8.          BufferedReader fromServer=new BufferedReader(new InputStreamReader(ClientSocket.getInputStream()));//获取从服务器端的流，建立套接字输入流
9.          DataOutputStream toServer=new DataOutputStream(ClientSocket.getOutputStream());//建立套接字输出流
10.         clientMessage=fromUser.readLine();//读取从用户的输入
11.         toServer.writeBytes(clientMessage);//写到服务器端
12.         serverMessage=fromServer.readLine();//从服务器端读取
13.                 ClientSocket.close();//关闭套接字连接
14.     }
15. }


1.  public class Server {//服务端
2.      public static void main(String args[])throws Exception
3.      {
4.          String ClientMessage;
5.          String ServerMessage;
6.          ServerSocket serversocket=new ServerSocket(5557);//端口要和客户端对应
7.          System.out.println("正在监听 5555 端口");//
8.          while(true)
9.          {
10.             Socket collection=serversocket.accept();//调用 accept()函数，建立 TCP 连接
11.             DataInputStream fromClient=new DataInputStream(collection.getInputStream());
12.             DataOutputStream toClient=new DataOutputStream(collection.getOutputStream());
13.             ClientMessage=fromClient.readUTF();//接收来自客户端的信息
14.             toClient.writeBytes(ServerMessage);//写到服务器端
15.             System.out.println("成功建立 TCP 连接");
16.         }
17.     }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
18. }
```

4.3、写一个多线程程序，四个线程对一个 int 变量，2 个加 1，2 个减 1，输出。

```
1.  public class TMain {
2.           int j=1;
3.           public synchronized void inc(){
4.                   j++;
5.                    System.out.println(Thread.currentThread().getName()+"-inc:"+j);
6.                }
7.       class T1 implements Runnable{
8.         public void run(){
9.            inc();
10.        }
11.      }
12.      public synchronized void dec(){
13.           j--;
14.           System.out.println(Thread.currentThread().getName()+"-dec:"+j);
15.        }
16.
17.
18.    class T11 implements Runnable{
19.        public void run(){
20.           dec();
21.        }
22.    }
23.    public static void main(String[] args) {
24.        TMain t = new TMain();
25.        T1 t1 =t.new T1();
26.        T11 t11 =t.new T11();
27.        for(int i=0;i<2;i++){
28.         Thread thread=new Thread(t1);
29.         thread.start();
30.
31.
32.         Thread thread1=new Thread(t11);
33.         thread1.start();
34.        }
35.
36.
37.    }
38.    }
```


http://www.itheima.com Copyright© 2018 黑马程序员

5. linux 试题

5.1、说出常用的 10 个 linux 操作命令，至少 5 个，并简述命令的作用。

LS 命令

- 作用：显示目录内容，类似 DOS 下的 DIR

- 格式：LS【options】【filename】

- 常用参数：

  >-a:all，不隐藏任何以"."字符开始的文件

  >-l：使用较长的格式列出信息

  >-r:按照文件名的逆序打印输出

  >-F:加上文件类型的指示符

  ls -lF | grep /　过滤

  man ls　　查询 ls 的帮助文件

cat 命令

- 作用：显示文件内容，concatenate 的缩写，类似 dos 的 type 命令。

- 格式：cat【options】【fielname】

- 常用参数：

  >-n：显示文件内容的行号。

  >-b：类似-n，但是不对空白行进行编号。

  >-s：当遇到有连续两行以上的空白行时，就代换为一行的空白行。

http://www.itheima.com Copyright© 2018 黑马程序员

mv 命令

- 作用：更改文件或者目录的名字。

- 格式：mv[options]source destination

- 常用参数：

  >-f：强制模式，覆盖文件不提示。

  >-i：交互模式，当要覆盖文件的时候给提示。

rm 命令

- 作用：删除文件命令，类似 dos 的 del 命令

- 格式：rm【options】filenames

- 常用参数：

 >-f：强制模式，不给提示。

 >-r,-R：删除目录，recursive

原文链接：http://blog.csdn.net/sinat_21903855/article/details/48936175

Copyright© 2018 黑马程序员

5.2、说出常见的 5 个 linux 系统日志，至少 3 个并做简述日志的用途。

access-log	纪录 HTTP/web 的传输
acct/pacct	纪录用户命令
aculog	纪录 MODEM 的活动
btmp	纪录失败的纪录
lastlog	纪录最近几次成功登录的事件和最后一次不成功的登录
messages	从 syslog 中记录信息（有的链接到 syslog 文件）
sudolog	纪录使用 sudo 发出的命令
sulog	纪录使用 su 命令的使用
syslog	从 syslog 中记录信息（通常链接到 messages 文件）
utmp	纪录当前登录的每个用户
wtmp	一个用户每次登录进入和退出时间的永久纪录
xferlog	纪录 FTP 会话

原文链接：http://os.51cto.com/art/200711/60313.htm

Copyright© 2018 黑马程序员

6. 数据库试题

创建一张员工表，表明 EMPLOYEES,有四个字段，EMPLOYEE_ID:员工表（主键）、DEPT_ID:部门号、EMPLOYEE_NAME:员工姓名、EMPLOYEE_SALARY:员工工资。

问题 1、写出建表语句

```
CREATE TABLE EMPLOYEES(
    EMPLOYEE_ID int not null primary key,
    DEPT_IDint,
    EMPLOYEE_NAME char(40),
    EMPLOYEE_SALARY double
);
```

问题 2、检索出员工工资最高的员工姓名和工资

```
select * from user where employee_salary= (select max(employee_salary) from user)
```

问题 3、检索出部门中员工最多的部门号和此部门员工数量

```
select dept_id,count(*) cno from user GROUP BY dept_id desc limit 1
```

7. 应用服务器试题

7.1、j2ee 中的应用服务器有哪些？

1）Weblogic

http://www.itheima.com Copyright© 2018 黑马程序员

2）Tomcat

3）JBoss

4）WebSphere

5）IIS

各个服务器之间的区别请参考链接：http://blog.csdn.net/qq1175421841/article/details/52280652

7.2、EJB 程序与普通的 java 程序区别有哪些？

EJB 是 sun 的服务器端组件模型，最大的用处是部署分布式应用程序当然,还有许多方式可以实现分布式应用，类似微软的.net 技术。凭借 java 跨平台的优势，用 EJB 技术部署的分布式系统可以不限于特定的平台。EJB (EnterpriseJavaBean)是 J2EE 的一部分，定义了一个用于开发基于组件的企业多重应用程序的标准。其特点包括网络服务支持和核心开发工具(SDK)。 在 J2EE 里，Enterprise Java Beans(EJB)称为 Java 企业 Bean，是 Java 的核心代码，分别是会话 Bean（Session Bean），实体 Bean（Entity Bean）和消息驱动 Bean（MessageDriven Bean）。

简单来讲：比如做一个工程就和盖房子，如果，你会 java，那么你就拥有了基本的技能，一步一步累砖，总能把房子盖好但是 EJB 就是一个框架，盖房子的时候，先有这个框架，然后你根据这个框架去累砖，房子就会盖的又快又

Copyright© 2018 黑马程序员

好。java 是基础，EJB 是在 java 上发展出来的模型，框架。

原文链接：http://blog.csdn.net/cs_fei/article/details/9824639

3、请简述什么是集群？---了解就可以

（服务器集群就是指将很多服务器集中起来一起进行同一种服务，在客户端看来就象是只有一个服务器。集群可以利用多个计算机进行并行计算从而获得很高的计算速度，也可以用多个计算机做备份，从而使得任何一个机器坏了整个系统还是能正常运行。一旦在服务器上安装并运行了群集服务，该服务器即可加入群集。群集化操作可以减少单点故障数量，并且实现了群集化资源的高可用性。下述各节简要介绍了群集创建和群集操作中的节点行为。）

九、科大讯飞（2017-12-11-lyq）

1. 字符串中有重复的内容去重 例如：abbccccaaddaggb-->abvadagb

```
1.  public class Test {
2.      public static void main(String[] args) {
3.          List<String> list = new ArrayList<String>();
4.          list.add("测试 1");
5.          list.add("测试 2");
6.          list.add("测试 3");
7.          list.add("测试 4");
8.          list.add("测试 4");
```


 Copyright© 2018 黑马程序员

```
9.        list.add("测试 2");
10.       list.add("测试 5");
11.       System.out.println("没有去重前的数据为>>>"+list.toString());
12.       for(int i = 0;i<list.size()-1;i++){
13.          for(int j = list.size()-1;j>i;j--)  {
14.              if(list.get(j).equals(list.get(i))){
15.                   list.remove(j);
16.              }
17.          }
18.       }
19.   System.out.println("去重后的数据为>>>"+list.toString());
```

2. javaScript 字符串去重的方法

a) for 遍历

```
1.    function quchong1(str){
2.     var newStr="";
3.     var flag;
4.     for(var i=0;i<str.length;i++){
5.         flag=1;
6.         for(var j=0;j<newStr.length;j++){
7.             if(str[i]==newStr[j]){
8.                 flag=0;
9.                 break;
10.            }
11.        }
12.        if(flag)  newStr+=str[i];
13.    }
14.    return newStr;
15. }
```

b) indexOf(无兼容问题)

```
1.    function quchong2(str){
2.     var newStr="";
3.     for(var i=0;i<str.length;i++){
4.         if(newStr.indexOf(str[i])==-1){
5.             newStr+=str[i];
6.         }
7.     }
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
8.      return newStr;
9.  }
```

c) search()方法

```
1.      function quchong3(str){
2.      var newStr="";
3.      for(var i=0;i<str.length;i++){
4.          if(newStr.search(str[i])==-1)
5.          newStr+=str[i];
6.
7.      }
8.      return newStr;
9.  }
```

d) 对象属性

```
1.    function quchong4(str){
2.      var obj={};
3.      var newStr="";
4.      for(var i=0;i<str.length;i++){
5.          if(!obj[str[i]]){
6.              newStr+=str[i];
7.              obj[str[i]]=1;
8.          }
9.      }
10.     return newStr;
11. }
```

3. 利用 java 面向对象的思路设计正方形.长方形.和圆的计算面积的类

圆：

```
12.     Public class MianJi {
13.        float r;
14.        float pai = (float) 3.14;
15.        //圆的面积
16.        void gongShi(){
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
17.             Float s = pai*r*r;
18.             System.out.println(“圆的面积为”+s);
19.       }
20.       //正方形的面积
21.       void zhengFangXing(float bianChang){
22.          System.out.println(“正方形的面积为”+bianChang*bianChang);
23.       }
24.       //长方形的面积
25.       void zhengFangXing(float chang,float kuan){
26.          System.out.println(“长方形的面积为”+chang*kuan);
27.       }
28.
29.       Public static void main(String[] arg){
30.             MianJi c = new  MianJi ();
31.             System.out.println(“请输入圆的半径：”)
32.             Scanner sc = new Scanner(System,in);
33.              c.r = sc.nextFloat();
34.              c.gongShi();
35.
36.             System.out.println(“请输入正方形的边长：”)
37.             Scanner sc = new Scanner(System,in);
38.              float bian = sc.nextFloat();
39.              c.zhengFangXing(bian)
40.
41.             System.out.println(“请输入长方形的长和宽：”)
42.             Scanner sc = new Scanner(System,in);
43.              float chang= sc.nextFloat();
44.              float kuan= sc.nextFloat();
45.              c.changFangXing(chang,kuan);
46.
47.       }
48.    }
```

4. 任何>=6 的偶数都可以分解为两个质数之和，从键盘输入一个偶数，输出其分解的质数

```
1.     public class D3hw8 {
2.     //将偶数拆分成俩质数的和
3.     public static void main(String[] args) {
```


http://www.itheima.com Copyright© 2018 黑马程序员

```
4.          int num=inPut();
5.          outPut(num);
6.
7.      }
8.
9.      public static int inPut(){
10.         //输入数字并通过检验确定一个符合要求的数
11.         Scanner sc=new Scanner(System.in);
12.         System.out.println("请输入大于 6 的偶数：");
13.         int num=sc.nextInt();
14.         if(num%2!=0||num<=6){
15.             System.out.println("输入错误，请重新输入大于 6 的偶数：");
16.             return inPut();//错误则返回 inPut()继续输入
17.         }
18.         return num;//正确则返回 num 值
19.     }
20.
21.     public static boolean isPrim(int num){
22.         //判断是否是质数
23.         for(int i=2;i<=Math.sqrt((double)num);i++){
24.             if(num%i==0){
25.                 return false;
26.             }
27.         }
28.         return true;
29.     }
30.
31.     public static void outPut(int num){
32.         //输出所有符合条件的质数对
33.         for(int i=2;i<=num/2;i++){
34.             if(isPrim(i)==true&&isPrim(num-i)==true){
35.                 System.out.println(i+" "+(num-i));
36.             }
37.         }
38.     }
39.
40.
41. }
```


http://www.itheima.com Copyright© 2018 黑马程序员

十、泰瑞（2017-12-16-wmm）

1. 笔试题

1.1 面向对象的特征有哪些方面？

答案：封装，多态，继承。详细内容见本文第二章内容:JavaSE 基础，第一节 Java 面向对象

1.2 int 和 Integer 有什么区别？

答案：Int 是整型,Integer 是 int 的封装类型。详细内容：见本文第二章内容：JavaSE 基础，第六节：Java 的数据类型

1.3、JAVA 语言如何进行异常处理，关键字：throws,throw,try,catch,finally 分别代表什么意义？在 try 块中可以抛出异常吗?

答案：详细内容：见本文第二章内容：JavaSE 基础，第四节： Java 的异常处理。

1.4、java 中实现多态的机制是什么?

答案：详细内容：见本文第二章内容：JavaSE 基础，第三节： Java 中的多态。

1.5、说出一些常用的类，包，接口，请各举 5 个。

答案：包：lang，util，awt，swing，io

类：System, Math,Date,RunTime,Test

http://www.itheima.com Copyright© 2018 黑马程序员

2. 上机题

2.1 题目：雇员查询系统

2.1.1. 语言和环境

A、实现语言：

Java

B、实现技术：

HTML

JavaScript

JSP

Servlet

C、环境要求：

Eclipse

Oracle 9i

JDK 1.6

Tomcat 6

火狐

2.1.2. 要求

XXX 公司有一个人事管理系统，其中一个功能模块是根据员工职位及员工姓名对员工明细进行查询。

功能和页面设计要求：

查询页面可以显示雇员的职位名称，职位名称的显示内容来源于数据库表 POST。职位名称不能重复。

查询页面能够完成客户端校验工作，能够对未选择职位类别进行校验提示。

结果显示页面能够根据输入的雇员名称和选择的雇员职位查询结果，并正确的使用表格显示结果，如果查询内容为空，应正确显示提示信息。

数据库设计要求：

数据库名称是 handson，数据块下的用户名是 EMPSYS。

数据库表 POST 的所有字段必须按（表 1）内容设置。内容按照（表 3）内容填充数据表

数据库表 EMPLOYEE 的所有字段必须按（表 2）内容设置。内容按照（表 4）内容填充数据表。

2.1.3. 数据库的设计

数据库的名称: handson

用户：EMPSYS

http://www.itheima.com Copyright© 2018 黑马程序员

表 1：

表名	POST					
主键	POST_ID					
序号	字段名称	字段说明	类型	位数	属性	备注
1	POST_ID	职位编号	int	4	必填，非空	
2	POST_NAME	职位名称	nvarchar	50	必填，非空	
3	POST_DESC	职位描述	nvarchar	100		

表 2：

表名	EMPLOYEE					
主键	EMP_ID					
序号	字段名称	字段说明	类型	位数	属性	备注
1	EMP_ID	雇员编号	int	4	必填，非空	
	POST_ID	职位编号	int	4	必填，非空	
2	EMP_NAME	雇员姓名	nvarchar	100		
3	EMP_SEX	雇员性别	int	4		1:男 2:女
	EMP_AGE	雇员年龄	int	4		
	EMP_DEPART	所属部门	nvarchar	50		
4	EMP_YEAR	雇员工龄	int	4		

2.1.4. 推荐实现步骤

1. 建立数据库

A、建立用户名 EMPSYS 和密码 EMPSYS：（5 分）

B、建立数据库表，表的结构参见上述表的结构：（5 分）

C、数据库完成以后，录入下面记录测试数据，如下表：（5 分）

表 3:

POST_ID	POST_NAME	POST_DESC
1000	行政助理	行政辅助人员
1001	行政主管	行政负责人
1002	业务经理	业务处主管
1003	总经理	公司负责人

表 4:

EMP_ID	POST_ID	EMP_NAME	EMP_SEX	EMP_AGE	EMP_DEPART	EMP_YEAR
00001	1000	李晓明	1	25	行政部	2
00002	1000	杨伟林	1	29	行政部	5
00003	1002	尤志苗	2	33	业务部	9
00004	1003	牛晓飞	1	40	集团	10

2.1.4 设计 WEB 页面

创建项目：

A、在 Eclipse 中建立 JAVA WEB 项目。（3 分）

制作首页：

http://www.itheima.com Copyright© 2018 黑马程序员

A、设计雇员查询的主页面，命名为 default.jsp，页面风格可以参看图 1 所示。（5 分）

B、编写程序获取数据表 POST 中的职位编号和名称，填充在<select>页面控件中，要求职位名称不能出现重复记录。（10 分）

C、单击“查询”按钮时，要验证客户是否选择职位名称，如果没有选择职位名称提示错误信息。（10 分）

D、当首页的雇员名项为空时，显示所有记录。（15 分）

E、当“雇员名称”项和“职位名称”项同时作为条件时，按要求找出查询记录。（15 分）

图 1 查询主界面

设计制作结果页面：

A、客户选填写了雇员名称或选择雇员职位后，单击“查询”按钮，开始根据要求查询，显示查询结果页面，如图 2 所示：（10 分）

图 2 雇员明细信息显示页面

http://www.itheima.com Copyright© 2018 黑马程序员

2.1.5. 注意事项：

请注意代码的软件书写，实体的命名规范（12 分）。

十一、文思创新(2017-12-17-wmm)

1. 什么叫对象？什么叫类？什么面向对象（OOP）？

答案：类的概念：类是具有相同属性和服务的一组对象的集合。它为属于该类的所有对象提供了统一的抽象描述，其内部包括属性和服务两个主要部分。在面向对象的编程语言中，类是一个独立的程序单位，它应该有一个类名并包括属性说明和服务说明两个主要部分。

对象的概念：对象是系统中用来描述客观事物的一个实体，它是构成系统的一个基本单位。一个对象由一组属性和对这组属性进行操作的一组服务组成。从更抽象 的角度来说，对象是问题域或实现域中某些事物的一个抽象，它反映该事物在系统中需要保存的信息和发挥的作用；它是一组属性和有权对这些属性进行操作的一组 服务的封装体。客观世界是由对象和对象之间的联系组成的。

 Copyright© 2018 黑马程序员

类与对象的关系就如模具和铸件的关系，类的实例化结果就是对象，而对一类对象的抽象就是类.类描述了一组有相同特性（属性）和相同行为（方法）的对象。上面大概就是它们的定义吧， 也许你是刚接触面象对象的朋友， 不要被概念的东西搞晕了， 给你举个列子吧，如果你去中关村想买几台组装的 PC 机，到了那里你第一步要干什么， 是不是装机的工程师和你坐在一起，按你提供的信息和你一起完成一个装机的配置单呀， 这个配置单就可以想像成是类，它就是一张纸，但是它上面记录了你要买的 PC 机的信息，如果用这个配置单买 10 台机器，那么这 10 台机子，都是按这个配置单 组成的，所以说这 10 台机子是一个类型的，也可以说是一类的。那么什么是对象呢，类的实例化结果就是对象，用这个配置单配置出来（实例化出来）的机子就是对象， 是我们可以操作的实体， 10 台机子， 10 个对象。每台机子都是独立的，只能说明他们是同一类的，对其中一个机做任何动作都不会影响其它 9 台机器，但是我对类修改， 也就是在这个配置单上加一个或少一个配件， 那么装出来的 9 个机子都改变了， 这是类和对象的关系(类的实例化结果就是对象）。

2. 相对于 JDK1.4，JDK1.5 有哪些新特性?

答案：泛型（Generics）

增强的“for”循环（Enhanced For loop）

自动装箱/ 自动拆箱（Autoboxing/unboxing）

类型安全的枚举（Type safe enums）

静态导入（Static import）

可变参数(Var args)

3. JAVA 中使用 final 修饰符，对程序有哪些影响?

答案:

1、修饰类

http://www.itheima.com Copyright© 2018 黑马程序员

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

在使用 final 修饰类的时候，要注意谨慎选择，除非这个类真的在以后不会用来继承或者出于安全的考虑，尽量不要将类设计为 final 类。

2、修饰方法

被 final 修饰的方法将不能被子类覆盖，主要用于 1，把方法锁定，以防任何继承类修改它的含。2，在早期的 Java 实现版本中，会将 final 方法转为内嵌调用，所以效率能够提升

3、修饰变量

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

当用 final 作用于类的成员变量时，成员变量（注意是类的成员变量，局部变量只需要保证在使用之前被初始化赋值即可）必须在定义时或者构造器中进行初始化赋值，而且 final 变量一旦被初始化赋值之后，就不能再被赋值了。

4. Java 环境变量 Unix/Linux 下如何配置?

答案：修改/etc/profile 文件当本机仅仅作为开发使用时推荐使用这种方法，因为此种配置时所有用户的 shell 都有权使用这些环境变量，可能会给系统带来安全性问题。用文本编辑器打开/etc/profile，在 profile 文件末尾加入：

```
JAVA_HOME=/usr/share/jdk1.5.0_05
PATH=$JAVA_HOME/bin:$PATH
CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
export JAVA_HOME
```


http://www.itheima.com Copyright© 2018 黑马程序员

export PATH

export CLASSPATH 重新登录即可

5. 写出 5 个你在 JAVA 开发中常用的包含（全名），并简述其作用。

常用的五个：

1）java.lang.*

提供利用 Java 编程语言进行程序设计的基础类。最重要的类是 Object（它是类层次结构的根）和 Class（它的实例表示正在运行的应用程序中的类）。

2）java.util.*

包含集合框架、遗留的 collection 类、事件模型、日期和时间设施、国际化和各种实用工具类（字符串标记生成器、随机数生成器和位数组、日期 Date 类、堆栈 Stack 类、向量 Vector 类等）。集合类、时间处理模式、日期时间工具等各类常用工具包

3）java.io.*

Java 的核心库 java.io 提供了全面的 IO 接口。包括：文件读写、标准设备输出等。Java 中 IO 是以流为基础进行输入输出的，所有数据被串行化写入输出流，或者从输入流读入。

4）java.net.*

并非所有系统都支持 IPv6 协议，而当 Java 网络连接堆栈尝试检测它并在可用时透明地使用它时，还可以利用系统属性禁用它。在 IPv6 不可用或被显式禁用的情况下，Inet6Address 对大多数网络连接操作都不再是有效参数。虽然可以保证在查找主机名时 java.net.InetAddress.getByName 之类的方法不返回 Inet6Address，但仍然可能通过传递字面值来创建此类对象。在此情况下，大多数方法在使用 Inet6Address 调用时都将抛出异常。

5）java.sql.*

http://www.itheima.com Copyright© 2018 黑马程序员

提供使用 JavaTM 编程语言访问并处理存储在数据源（通常是一个关系数据库）中的数据的 API。此 API 包括一个框架，凭借此框架可以动态地安装不同驱动程序来访问不同数据源。

6. 写出 5 个常见的运行时异常（RuntimeException）。

1）ClassCastException(类转换异常)

2）IndexOutOfBoundsException(数组越界)

3）NullPointerException(空指针)

4）ArrayStoreException(数据存储异常，操作数组时类型不一致)

5）IO 操作的 BufferOverflowException 异常

7. 方法重载（overload）需要满足什么条件，方法覆盖/方法重写（override）需要满足什么条件？（二选一）

答案：重载需要满足的条件：在同一类中定义的方法， 方法名必须相同 ，返回类型必须相同，参数一定不同。

发生覆盖的条件：

“三同一不低”，子类和父类的方法名称，参数列表，返回类型必须完全相同，而且子类方法的访问修饰符的权限不能比父类低。

子类方法不能抛出比父类方法更多的异常。即子类方法所抛出的异常必须和父类方法所抛出的异常一致，或者是其子类，或者什么也不抛出；

被覆盖的方法不能是 final 类型的。因为 final 修饰的方法是无法覆盖的。

被覆盖的方法不能为 private。否则在其子类中只是新定义了一个方法，并没有对其进行覆盖。

被覆盖的方法不能为 static。所以如果父类中的方法为静态的，而子类中的方法不是静态的，但是两个方法除了这一点外其他都满足覆盖条件，那么会发生编译错误。反之亦然。即使父类和子类中的方法都是静态的，并且满足覆盖条

http://www.itheima.com Copyright© 2018 黑马程序员

件，但是仍然不会发生覆盖，因为静态方法是在编译的时候把静态方法和类的引用类型进行匹配。

重写规则：重写方法不能比被重写方法限制有更严格的访问级别。（但是可以更广泛，比如父类方法是包访问权限，子类的重写方法是 public 访问权限。）比如：Object 类有个 toString()方法，开始重写这个方法的时候我们总容易忘记 public 修饰符，编译器当然不会放过任何教训我们的机会。出错的原因就是：没有加任何访问修饰符的方法具有包访问权限，包访问权限比 public 当然要严格了，所以编译器会报错的。参数列表必须与被重写方法的相同。重写有个孪生的弟弟叫重载，也就是后面要出场的。如果子类方法的参数与父类对应的方法不同，那么就是你认错人了，那是重载，不是重写。返回类型必须与被重写方法的返回类型相同。

父类方法 A：void eat(){} 子类方法 B：int eat(){} 两者虽然参数相同，可是返回类型不同，所以不是重写。

父类方法 A：int eat(){} 子类方法 B：long eat(){} 返回类型虽然兼容父类，但是不同就是不同，所以不是重写。

8. 继承（inheritance）的优缺点是什么?

优点:

新的实现很容易，因为大部分是继承而来的 。很容易修改和扩展已有的实现

缺点:

打破了封装，因为基类向子类暴露了实现细节 ，白盒重用，因为基类的内部细节通常对子类是可见的 ，当父类的实现改变时可能要相应的对子类做出改变 ，不能在运行时改变由父类继承来的实现。由此可见，组合比继承具有更大的灵活性和更稳定的结构，一般情况下应该优先考虑组合。只有当下列条件满足时才考虑使用继承： 子类是一种特殊的类型，而不只是父类的一个角色 ，子类的实例不需要变成另一个类的对象子类扩展，而不是覆盖或者使父类的功能失效。

http://www.itheima.com Copyright© 2018 黑马程序员

9. 为什么要使用接口和抽象类?

Java 接口和 Java 抽象类代表的就是抽象类型，就是我们需要提出的抽象层的具体表现。OOP 面向对象的编程，如果要提高程序的复用率，增加程序的可维护性，可扩展性，就必须是面向接口的编程，面向抽象的编程，正确地使用接口、抽象类这些太有用的抽象类型做为你结构层次上的顶层。

1、Java 接口和 Java 抽象类最大的一个区别，就在于 Java 抽象类可以提供某些方法的部分实现，而 Java 接口不可以，这大概就是 Java 抽象类唯一的优点吧，但这个优点非常有用。 如果向一个抽象类里加入一个新的具体方法时，那么它所有的子类都一下子都得到了这个新方法，而 Java 接口做不到这一点，如果向一个 Java 接口里加入一个新方法，所有实现这个接口的类就无法成功通过编译了，因为你必须让每一个类都再实现这个方法才行.

2、一个抽象类的实现只能由这个抽象类的子类给出，也就是说，这个实现处在抽象类所定义出的继承的等级结构中，而由于 Java 语言的单继承性，所以抽象类作为类型定义工具的效能大打折扣。 在这一点上，Java 接口的优势就出来了，任何一个实现了一个 Java 接口所规定的方法的类都可以具有这个接口的类型，而一个类可以实现任意多个 Java 接口，从而这个类就有了多种类型。

3、从第 2 点不难看出，Java 接口是定义混合类型的理想工具，混合类表明一个类不仅仅具有某个主类型的行为，而且具有其他的次要行为。

4、结合 1、2 点中抽象类和 Java 接口的各自优势，具精典的设计模式就出来了：声明类型的工作仍然由 Java 接口承担，但是同时给出一个 Java 抽象类，且实现了这个接口，而其他同属于这个抽象类型的具体类可以选择实现这个 Java 接口，也可以选择继承这个抽象类，也就是说在层次结构中，Java 接口在最上面，然后紧跟着抽象类，哈，这下两个的最大优点都能发挥到极至了。这个模式就是“缺省适配模式”。 在 Java 语言 API 中用了这种模式，而且全都遵循一定的命名规范：Abstract +接口名。

Java 接口和 Java 抽象类的存在就是为了用于具体类的实现和继承的，如果你准备写一个具体类去继承另一个具体类的话，那你的设计就有很大问题了。Java 抽象类就是为了继承而存在的，它的抽象方法就是为了强制子类必须去

http://www.itheima.com Copyright© 2018 黑马程序员

实现的。

使用 Java 接口和抽象 Java 类进行变量的类型声明、参数是类型声明、方法的返还类型说明，以及数据类型的转换等。而不要用具体 Java 类进行变量的类型声明、参数是类型声明、方法的返还类型说明，以及数据类型的转换等。我想，如果你编的代码里面连一个接口和抽象类都没有的话，也许我可以说你根本没有用到任何设计模式，任何一个设计模式都是和抽象分不开的，而抽象与 Java 接口和抽象 Java 类又是分不开的。

接口的作用，一言以蔽之，就是标志类的类别。把不同类型的类归于不同的接口，可以更好的管理他们。把一组看如不相关的类归为一个接口去调用.可以用一个接口型的变量来引用一个对象,这是接口我认为最大的作用.

10. 什么是自定义异常？如何自定义异常？

参考：https://www.cnblogs.com/AlanLee/p/6104492.html

11. Set，List，Map 有什么区别？

答案：见本文第二章，第八节：Java 的集合

http://www.itheima.com Copyright© 2018 黑马程序员

12. 什么叫对象持久化（OBJect PERSIstence），为什么要进行对象持久化?

持久化的对象，是已经存储到数据库或保存到本地硬盘中的对象，我们称之为持久化对象。为了保存在内存中的各种对象的状态（也就是实例变量，不是方法），并且可以把保存的对象状态再读出来。虽然你可以用你自己的各种各样的方法来保存 object states，但是 Java 给你提供一种应该比你自己好的保存对象状态的机制，那就是序列化。简单说就是对象序列化是将对象状态转换为可保持或传输的格式的过程。

什么情况下需要序列化 ：

a）当你想把的内存中的对象状态保存到一个文件中或者数据库中时候；

b）当你想用套接字在网络上传送对象的时候；

c）当你想通过 RMI 传输对象的时候；

对象要实现序列化，是非常简单的，只需要实现 Serializable 接口就可以了。

public class Test implements Serializable

13. JavaScript 有哪些优缺点?

(1).javascript 的优点:

javascript 减少网络传输。

在 javascript 这样的用户端脚本语言出现之前，传统的数据提交和验证工作均由用户端浏览器通过网络传输到服务器开发上进行。如果数据量很大，这对于网络和服务器开发的资源来说实在是一种无形的浪费。而使用 javascript 就可以在客户端进行数据验证。

javascript 方便操纵 html 对象。

javascript 可以方便地操纵各种页面中的对象，用户可以使用 javascript 来控制页面中各个元素的外观、状态甚至运行方式，javascript 可以根据用户的需要“定制”浏览器，从而使网页更加友好。

http://www.itheima.com Copyright© 2018 黑马程序员

javascript 支持分布式应用运算。

javascript 可以使多种任务仅在用户端就可以完成，而不需要网络和服务器开发的参与，从而支持分布式应用 的运算和处理。

（2）javascript 的局限性：

各浏览器厂商对 javascript 支持程度不同。

目前在互联网 上有很多浏览器，如 firefox、internet explorer、opera 等，但每种浏览器支持 javascript 的程度是不一样的，不同的浏览器在浏览一个带有 javascript 脚本的主页时，由于对 javascript 的支持稍有不同，其效果会有一定的差距，有时甚至会显示不出来。

“web 安全性”对 javascript 一些功能牺牲。

当把 javascript 的一个设计目标设定为“web 安全性”时，就需要牺牲 javascript 的一些功能。因此，纯粹的 javascript 将不能打开、读写和保存用户计算机上的文件。其有权访问的唯一信息就是该 javascript 所嵌入开发 的那个 web 主页中的信息，简言之，javascript 将只存在于它自己的小小世界—web 主页里。

14. Jsp 有什么特点?

JSP(Java Server Pages)是由 Sun Microsystems 公司倡导、许多公司参与一起建立的一种动态网页技术标准。JSP 技术是用 JAVA 语言作为脚本语言的，JSP 网页为整个服务器端的 JAVA 库单元提供了一个接口来服务于 HTTP 的应用程序。

在传统的网页 HTML 文件(*.htm,*.html)中加入 Java 程序片段(Scriptlet)和 JSP 标记(tag)，就构成了 JSP 网页(*.jsp)。Web 服务器在遇到访问 JSP 网页的请求时，首先执行其中的程序片段，然后将执行结果以 HTML 格式返回给客户。程序片段可以操作数据库、重新定向网页以及发送 email 等等，这就是建立动态网站所需要的功能。所有程序操作都在服务器端执行，网络上传送给客户端的仅是得到的结果，对客户浏览器的要求最低，可以实现无 Plugin，无

http://www.itheima.com Copyright© 2018 黑马程序员

ActiveX，无 Java Applet，甚至无 Frame。

JSP 的优点：

1)对于用户界面的更新，其实就是由 Web Server 进行的，所以给人的感觉更新很快。

2)所有的应用都是基于服务器的，所以它们可以时刻保持最新版本。

3)客户端的接口不是很繁琐，对于各种应用易于部署、维护和修改。

15. 什么叫脏数据，什么叫脏读（Dirty Read）

脏数据在临时更新（脏读）中产生。事务 A 更新了某个数据项 X，但是由于某种原因，事务 A 出现了问题，于是要把 A 回滚。但是在回滚之前，另一个事务 B 读取了数据项 X 的值(A 更新后)，A 回滚了事务，数据项恢复了原值。事务 B 读取的就是数据项 X 的就是一个“临时”的值，就是脏数据。

脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是脏数据，依据脏数据所做的操作可能是不正确的。

第十章 项目业务逻辑问题

一、 传统项目（2017-12-5-lyq）

1. 什么是 BOS?

ERP 系统是企业资源计划(Enterprise Resource Planning)的简称。

http://www.itheima.com Copyright© 2018 黑马程序员

BOSS(Business & Operation Support)指的是业务运营支撑系统。

BOS 是 ERP 的集成与应用平台。 BOS 遵循面向服务的架构体系，是一个面向业务的可视化开发平台；是一个 ERP 和第三方应用集成的技术平台。它有效的解决了 ERP 应用的最主要矛盾 - - - 用户需求个性化和传统 ERP 软件标准化之间的矛盾。

BOS 与 ERP 是什么关系?

ERP 是企业管理信息化的全面解决方案， ERP 是基于 BOS 构建的。 ERP 满足企业全面业务的标准应用；BOS 确保了企业 ERP 应用中的个性化需求完美实现。基于 BOS 的 ERP，可以为不同行业不同发展阶段的企业构建灵活的、可扩展的、全面集成的整体解决方案。

2. Activity 工作流

2.1 什么是工作流？

（举个栗子）现在大多数公司的请假流程是这样的：员工打电话（或网聊）向上级提出请假申请——上级口头同意——上级将请假记录下来——月底将请假记录上交公司——公司将请假录入电脑。采用工作流技术的公司的请假流程是这样的：员工使用账户登录系统——点击请假——上级登录系统点击允许。就这样，一个请假流程就结束了。有人会问，那上级不用向公司提交请假记录？公司不用将记录录入电脑？答案是，用的。但是这一切的工作都会在上级点击允许后自动运行！这就是工作流技术。

Georgakopoulos 给出的工作流定义是： 工作流是将一组任务组织起来以完成某个经营过程：定义了任务的触发顺序和触发条件，每个任务可以由一个或多个软件系统完成，也可以由一个或一组人完成，还可以由一个或多个人与软件系统协作完。

http://www.itheima.com Copyright© 2018 黑马程序员

2.2 工作流技术的优点

从上面的例子，很容易看出，工作流系统实现了工作流程的自动化，提高了企业运营效率、改善企业资源利用、提高企业运作的灵活性和适应性、提高量化考核业务处理的效率、减少浪费（时间就是金钱）。而手工处理工作流程，一方面无法对整个流程状况进行有效跟踪、了解，另一方面难免会出现人为的失误和时间上的延时导致效率低下，特别是无法进行量化统计，不利于查询、报表及绩效评估。

2.3 工作流生命周期

除了我们自行启动（start）或者结束（finish）一个 Activity，我们并不能直接控制一个 Activity 的生命状态，我们只能通过实现 Activity 生命状态的表现——即回调方法来达到管理 Activity 生命周期的变化。

http://www.itheima.com Copyright© 2018 黑马程序员