

史上最简单的java面试

java基础

[Java基础：Java面向对象的特征](#)

[Java基础：面向对象六大原则](#)

[Java基础：谈谈final、finally、finalize的区别](#)

[Java基础：Java抽象类与接口的区别](#)

[Java基础：Java 中的 ==, equals 与 hashCode 的区别与联系](#)

[Java基础：Java 内部类和静态内部类的区别](#)

[Java基础：Java中重载与重写的区别](#)

[Java基础：java 泛型详解-绝对是对泛型方法讲解最详细的，没有之一](#)

[Java基础：int与Integer区别](#)

[Java基础：Java基础：Java的反射机制](#)

[Java基础：Java finally语句到底是在return之前还是之后执行？](#)

[Java基础：Java对象初始化过程](#)

[Java基础：面向接口编程详解](#)

java集合

[Java基础：Java容器之LinkedList](#)

[Java基础：JAVA Hashmap的死循环及Java8的修复](#)

[Java基础：JAVA中BitSet使用详解](#)

[Java基础：java中HashSet详解](#)

[Java基础：Java容器之ArrayList](#)

[Java基础：Java容器之HashMap](#)

java io

[Java基础：Java IO流学习总结](#)

[Java基础：攻破JAVA NIO技术壁垒1](#)

[Java基础：攻破JAVA NIO技术壁垒2](#)

java线程

[Java基础：Java线程基础](#)

[Java基础：java线程状态](#)

java并发

[Java并发： AtomicInteger源码分析——基于CAS的乐观锁实现](#)

[Java并发： BlockingQueue解读](#)

[Java并发： ConcurrentHashMap解读](#)

[Java并发： CopyOnWriteArrayList实现原理及源码分析](#)

[Java并发： Java中CAS详解](#)

[Java并发： Java中锁的分类](#)

[Java并发： Java并发编程： CountDownLatch、 CyclicBarrier和Semaphore](#)

[Java并发： java线程池详解](#)

[Java并发： Synchronized原理和优化](#)

[Java并发： 彻底理解ThreadLocal](#)

[Java并发： 单例模式的双检查](#)

java虚拟机

[Java虚拟机： JVM内存模型](#)

[Java虚拟机： JVM内存模型和volatile详解](#)

[Java虚拟机： 垃圾收集算法](#)

[Java虚拟机： 垃圾收集器和内存分配策略](#)

[Java虚拟机： 怎么确定对象已经死了？](#)

[Java虚拟机： JVM类加载机制](#)

[Java虚拟机： 虚拟机类加载机制](#)

[Java虚拟机： JVM性能调优监控工具jps、 jstack、 jmap、 jhat、 jstat、 hprof使用详解](#)

Mysql数据库

[数据库： 真正理解Mysql的四种隔离级别](#)

[数据库： Mysql综合练习题](#)

[数据库： Mysql数据类型](#)

[数据库： MySQL几种常用的存储引擎区别](#)

[数据库： 数据库索引优化](#)

[数据库： 数据库索引的类型](#)

[数据库： 数据库连接池原理详解与自定义连接池实现](#)

springboot面试

[Spring Boot最核心的27个干货注解，你了解多少？](#)

[面试必问—— Spring Boot 是如何实现自动配置的？](#)

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

spring boot面试问题集锦

ssm面试

SSM:面试被问烂的 Spring IOC

SSM:Spring AOP是什么？你都拿它做什么？

SSM:SpringMVC工作原理详解

SSM:谈谈你对SpringAOP的了解

SSM:必须掌握的20种Spring常用注解

SSM:Mybatis架构与原理

SSM:Mybatis常见面试题总结及答案

SSM:MyBatis中的\$和#，用不好，准备走人！

算法

算法：5亿整数的大文件，怎么排？

网络

网络：TCP、IP协议族(一) HTTP简介、请求方法与响应状态码

网络：TCP、IP协议族(二) HTTP报文头解析

网络：TCP、IP协议族(三) 数字签名与HTTPS详解

公众号：方志朋

Java基础：Java面向对象的特征

面向对象的三个基本特征是：封装、继承、多态。

封装

封装最好理解了。封装是面向对象的特征之一，是对对象和类概念的主要特性。

封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

封装的优点

- 将变化隔离
- 便于使用
- 提高重用性
- 提高安全性

封装的缺点：

将变量等使用private修饰，或者封装进方法内，使其不能直接被访问，增加了访问步骤与难度！

封装的实现形式

- A、使用访问权限修饰符private
在定义JavaBean时对于成员变量使用private进行修饰，同时对外提供set、get方法
使用了private修饰的成员在其他类中不能直接访问，此时需要使用set、get方法进行。
- B、定义一个Java类与Java的方法就是最简单最常见的面向对象的封装操作，这些操作符合隐藏实现细节，提供访问方式的思路。

继承

面向对象编程(OOP)语言的一个主要功能就是“继承”。继承是指这样一种能力：它可以用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

通过继承创建的新类称为“子类”或“派生类”。

被继承的类称为“基类”、“父类”或“超类”。

继承的过程，就是从一般到特殊的过程。

要实现继承，可以通过“继承”(Inheritance)和“组合”(Composition)来实现。

在Java语言中，一个类只能单继承，可以实现多个接口。继承就是子类继承父类的特征和行为，使得子类对象具有父类的非private属性和方法。

类的继承格式：

通过extends关键字申明一个类继承另一个类，如

```
class父类{}  
class子类extends父类{}
```

为什么需要继承？

- 减少代码重复、臃肿，提高代码可维护性。

继承的特性：

- 子类拥有父类非private的属性和方法；
- 子类可以拥有完全属于自己的属性和方法（对父类扩展）；
- Java是单继承(每个子类只能继承一个父类)；但是Java可以是多重继承（如A继承B，B继承C）。

Super和this关键字：

Super关键字：我们可以通过super关键字来实现子类对父类成员的访问，引用当前实例对象的父类。

This关键字：指向实例对象自己的引用。

多态

多态就是同一个接口，使用不同的实现，而执行不同的操作。

多态的三个必要条件：

- 继承 (extends)
- 重写（子类重写父类的同名方法）
- 父类引用指向子类的对象，如：

子类继承父类，重写父类的方法，当子类对象调用重写的方法时，调用的是子类的方法，而不是父类的方法，当想要调用父类中被重写的方法时，则需使用关键字super。

参考资料

https://blog.csdn.net/Wei_HHH/article/details/74864628

<https://blog.csdn.net/u011159417/article/details/73500054>

<https://blog.csdn.net/cancan8538/article/details/8057095>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

公众号：方志朋

Java基础：面向对象六大原则

本文主要介绍了面向对象六大原则。

单一职责原则(Single-Responsibility Principle)。

"对一个类而言，应该仅有一个引起它变化的原因。"本原则是我们非常熟悉地"高内聚性原则"的引申，但是通过将"职责"极具创意地定义为"变化的原因"，使得本原则极具操作性，尽显大师风范。同时，本原则还揭示了内聚性和耦合性，基本途径就是提高内聚性；如果一个类承担的职责过多，那么这些职责就会相互依赖，一个职责的变化可能会影响另一个职责的履行。其实OOD的实质，就是合理地进行类的职责分配。

开放封闭原则(Open-Closed principle)。

"软件实体应该是可以扩展的，但是不可修改。"本原则紧紧围绕变化展开，变化来临时，如果不改动软件实体的源代码，就能扩充它的行为，那么这个软件实体设计就是满足开放封闭原则的。如果说我们预测到某种变化，或者某种变化发生了，我们应当创建抽象类来隔离以后发生的同类变化。在Java中，这种抽象是指抽象基类或接口；在C++中，这各抽象是指抽象基类或纯抽象基类。当然，没有对所有情况都贴切的模型，我们必须对软件实体应该面对的变化做出选择。

Liskov替换原则(Liskov-Substitution Principle)。

"子类型必须能够替换掉它们的基类型。"本原则和开放封闭原则关系密切，正是子类型的可替换性，才使得使用基类型模块无需修改就可扩充。Liskov替换原则从基于契约的设计演化而来，契约通过为每个方法声明"先验条件"和"后验条件"；定义子类时，必须遵守这些"先验条件"和"后验条件"。当前基于契约的设计发展势头正劲，对实现"软件工厂"的"组装生产"梦想是一个有力的支持。

依赖倒置原则(Dependency-Inversion Principle)。

"抽象不应依赖于细节，细节应该依赖于抽象。"本原则几乎就是软件设计的正本清源之道。因为人解决问题的思考过程是先抽象后具体，从笼统到细节，所以我们先生产出的势必是抽象程度比较高的实体，而后才是更加细节化的实体。于是，"细节依赖于抽象"就意味着后来的依赖于先前的，这是自然而然的重用之道。而且，抽象的实体代表着笼而统之的认识，人们总是比较容易正确认识它们，而且本身也是不易变的，依赖于它们是安全的。依赖倒置原则适应了人类认识过程的规律，是面向对象设计的标志所在。

接口隔离原则(Interface–Segregation Principle)。

"多个专用接口优于一个单一的通用接口。"本原则是单一职责原则用于接口设计的自然结果。一个接口应该保证，实现该接口的实例对象可以只呈现为单一的角色；这样，当某个客户程序的要求发生变化，而迫使接口发生改变时，影响到其他客户程序的可能性小。

良性依赖原则。

"不会在实际中造成危害的依赖关系，都是良性依赖。"通过分析不难发现，本原则的核心思想是"务实"，很好地揭示了极限编程(Extreme Programming)中"简单设计"各"重构"的理论基础。本原则可以帮助我们抵御"面向对象设计五大原则"以及设计模式的诱惑，以免陷入过度设计(Over-engineering)的尴尬境地，带来不必要的复杂性。

参考资料

<https://blog.csdn.net/cancan8538/article/details/8057095>

<https://www.jianshu.com/p/f2c2250cc33b>

公众号：方志朋



程序员理财，请关注：



Java基础：谈谈final、finally、finalize的区别

谈谈final、finally、finalize的区别？

final

根据程序上下文环境，Java关键字final有“这是无法改变的”或者“终态的”含义，它可以修饰非抽象类、非抽象类成员方法和变量。你可能出于两种理解而需要阻止改变：设计或效率。

- final类不能被继承，没有子类，final类中的方法默认是final的。
- final方法不能被子类的方法覆盖，但可以被继承。
- final成员变量表示常量，只能被赋值一次，赋值后值不再改变。
- final不能用于修饰构造方法。

1.final类

final类不能被继承，因此final类的成员方法没有机会被覆盖，默认都是final的。在设计类时候，如果这个类不需要有子类，类的实现细节不允许改变，并且确信这个类不会被扩展，那么就设计为final类。

2.final方法

如果一个类不允许其子类覆盖某个方法，则可以把这个方法声明为final方法。

使用final方法的原因有二：

- 把方法锁定，防止任何继承类修改它的意义和实现。
- 高效。编译器在遇到调用final方法时候会转入内嵌机制，大大提高执行效率。

例如：

```
1
2 public class Test1 {
3 public static void main(String[] args) {
4     // TODO 自动生成方法存根
5 }
6 public void f1() {
```

```
7     System.out.println("f1");
8 }
9 //无法被子类覆盖的方法
10 public final void f2() {
11     System.out.println("f2");
12 }
13 public void f3() {
14     System.out.println("f3");
15 }
16 private void f4() {
17     System.out.println("f4");
18 }
19 }
20 public class Test2 extends Test1 {
21
22 public void f1(){
23     System.out.println("Test1父类方法f1被覆盖!");
24 }
25 public static void main(String[] args) {
26     Test2 t=new Test2();
27     t.f1();
28     t.f2(); //调用从父类继承过来的final方法
29     t.f3(); //调用从父类继承过来的方法
30     //t.f4(); //调用失败，无法从父类继承获得
31 }
32 }
```

公众号：方志朋

3. final变量（常量）

- 用final修饰的成员变量表示常量，值一旦给定就无法改变！
- final修饰的变量有三种：静态变量、实例变量和局部变量，分别表示三种类型的常量。
- 从下面的例子中可以看出，一旦给final变量初值后，值就不能再改变了。

另外，final变量定义的时候，可以先声明，而不给初值，这中变量也称为final空白，无论什么情况，编译器都确保空白final在使用之前必须被初始化。但是，final空白在final关键字final的使用上提供了更大的灵活性，为此，一个类中的final数据成员就可以实现依对象而有所不同，却有保持其恒定不变的特征。

```
1
2 public class Test3 {
3     private final String S = "final实例变量S";
4     private final int A = 100;
5     public final int B = 90;
6
7     public static final int C = 80;
8     private static final int D = 70;
9
10    public final int E; //final空白,必须在初始化对象的时候赋初值
11
12    public Test3(int x) {
13        E = x;
14    }
15
16    /**
17     * @param args
18     */
19    public static void main(String[] args) {
20        Test3 t = new Test3(2);
21        //t.A=101;      //出错,final变量的值一旦给定就无法改变
22        //t.B=91; //出错,final变量的值一旦给定就无法改变
23        //t.C=81; //出错,final变量的值一旦给定就无法改变
24        //t.D=71; //出错,final变量的值一旦给定就无法改变
25
26        System.out.println(t.A);
27        System.out.println(t.B);
28        System.out.println(t.C); //不推荐用对象方式访问静态字
段
29        System.out.println(t.D); //不推荐用对象方式访问静态字
段
30        System.out.println(Test3.C);
31        System.out.println(Test3.D);
32        //System.out.println(Test3.E); //出错,因为E为final
空白,依据不同对象值有所不同.
33        System.out.println(t.E);
34
35        Test3 t1 = new Test3(3);
36        System.out.println(t1.E); //final空白变量E依据对象的
不同而不同
```

公众号: 方志朋

```
37     }
38
39     private void test() {
40         System.out.println(new Test3(1).A);
41         System.out.println(Test3.C);
42         System.out.println(Test3.D);
43     }
44
45     public void test2() {
46         final int a;      //final空白，在需要的时候才赋值
47         final int b = 4;  //局部常量--final用于局部变量的情
形
48         final int c;      //final空白，一直没有给赋值。
49         a = 3;
50         //a=4;    出错，已经给赋过值了。
51         //b=2;    出错，已经给赋过值了。
52     }
53 }
```

公众号：方志朋

4.final参数

当函数参数为final类型时，你可以读取使用该参数，但是无法改变该参数的值。

```
1 public class Test4 {
2     public static void main(String[] args) {
3         new Test4().f1(2);
4     }
5
6     public void f1(final int i) {
7         //i++;    //i是final类型的，值不允许改变的。
8         System.out.print(i);
9     }
10 }
```

2.finally

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

finally是关键字，在异常处理中，try子句中执行需要运行的内容，catch子句用于捕获异常，finally子句表示不管是否发生异常，都会执行。finally可有可无。但是try...catch必须成对出现。

3.finalize()

finalize() 方法名，Object类的方法，Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象进行调用。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的子类覆盖 finalize() 方法以整理系统资源或者执行其他清理操作。

代码实例：

```
1 class Person
2 {
3     private String name;
4     private int age;
5
6     public Person(String name, int age) {
7         this.name = name;
8         this.age = age;
9     }
10
11    public String toString()
12    {
13        return "姓名: "+this.name+", 年龄: "+this.age;
14    }
15
16    public void finalize() throws Throwable{//对象释放空间是默认调用此
方法
17        System.out.println("对象被释放-->"+this); //直接输出次对象，调用
toString()方法
18    }
19
20 }
21
22 public class SystemDemo {
23
24     /**
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
25     * @param args
26     */
27     public static void main(String[] args) {
28         // TODO Auto-generated method stub
29         Person per=new Person("zhangsan",30);
30         per=null;//断开引用，释放空间
31         //方法1：
32         System.gc();//强制性释放空间
33         //方法2：
34 //         Runtime run=Runtime.getRuntime();
35 //         run.gc();
36     }
37
38 }
```

参考资料

<https://www.cnblogs.com/xwdreamer/archive/2012/04/17/2454178.html>

<https://www.jianshu.com/p/ac5e3f0d6ad5>

公众号：方志朋



程序员理财，请关注：



Java基础：Java抽象类与接口的区别

谈谈Java抽象类与接口的区别？

很多常见的面试题都会出诸如抽象类和接口有什么区别，什么情况下会使用抽象类和什么情况你会使用接口这样的问题。本文我们将仔细讨论这些话题。

抽象类与接口是java语言中对抽象概念进行定义的两种机制，正是由于他们的存在才赋予java强大的面向对象的能力。

在讨论它们之间的不同点之前，我们先看看抽象类、接口各自的特性。

抽象类

我们都应该知道在面向对象的领域一切都是对象，同时所有的对象都是通过类来描述的，但是并不是所有的类都是来描述对象的。如果一个类没有足够的信息来描述一个具体的对象，而需要其他具体的类来支撑它，那么这样的类我们称它为抽象类。比如new Animal()，我们都知道这个是产生一个动物Animal对象，但是这个Animal具体长成什么样子我们并不知道，它没有一个具体动物的概念，所以他就是一个抽象类，需要一个具体的动物，如狗、猫来对它进行特定的描述，我们才知道它长成啥样。

在面向对象领域由于抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是不能实例化的。

抽象类是用来捕捉子类的通用特性的。它不能被实例化，只能被用作子类的超类。抽象类是被用来创建继承层级里子类的模板。以JDK中的GenericServlet为例：

```
1 public abstract class GenericServlet implements Servlet, ServletCo
  nfig, Serializable {
2     // abstract method
3     abstract void service(ServletRequest req, ServletResponse res)
4     ;
5     void init() {
6         // Its implementation
7     }
8     // other method related to Servlet
9 }
```

公众号：方志朋

当HttpServlet类继承GenericServlet时，它提供了service方法的实现：

```
1
2 public class HttpServlet extends GenericServlet {
3     void service(ServletRequest req, ServletResponse res) {
4         // implementation
5     }
6
7     protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
8         // Implementation
9     }
10
11    protected void doPost(HttpServletRequest req, HttpServletResponse resp) {
12        // Implementation
13    }
14
15    // some other methods related to HttpServlet
16 }
```

创建抽象类和抽象方法非常有用,因为他们可以使类的抽象性明确起来,并告诉用户和编译器打算怎样使用他们.抽象类还是有用的重构器,因为它们使我们可以很容易地将公共方法沿着继承层次结构向上移动。

在使用抽象类时需要注意几点：

- 1、抽象类不能被实例化，实例化的工作应该交由它的子类来完成，它只需要有一个引用即可。
- 2、抽象方法必须由子类来进行重写。
- 3、只要包含一个抽象方法的抽象类，该方法必须要定义成抽象类，不管是否还包含有其他方法。
- 4、抽象类中可以包含具体的方法，当然也可以不包含抽象方法。
- 5、子类中的抽象方法不能与父类的抽象方法同名。
- 6、abstract不能与final并列修饰同一个类。
- 7、abstract 不能与private、static、final或native并列修饰同一个方法。

接口

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

接口是抽象方法的集合。如果一个类实现了某个接口，那么它就继承了这个接口的抽象方法。这就像契约模式，如果实现了这个接口，那么就必须确保使用这些方法。接口只是一种形式，接口自身不能做任何事情。以Externalizable接口为例：

```
1 public interface Externalizable extends Serializable {  
2  
3     void writeExternal(ObjectOutput out) throws IOException;  
4  
5     void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;  
6 }
```

当你实现这个接口时，你就需要实现上面的两个方法：

```
1 public class Employee implements Externalizable {  
2  
3     int employeeId;  
4     String employeeName;  
5  
6     @Override  
7     public void readExternal(ObjectInput in) throws IOException,  
ClassNotFoundException {  
8         employeeId = in.readInt();  
9         employeeName = (String) in.readObject();  
10    }  
11  
12    @Override  
13    public void writeExternal(ObjectOutput out) throws IOException {  
14        out.writeInt(employeeId);  
15        out.writeObject(employeeName);  
16    }  
17  
18 }  
19 }
```

在使用接口过程中需要注意如下几个问题：

- 1、一个Interface的所有方法访问权限自动被声明为public。确切的说只能为public，当然你可以显示的声明为protected、private，但是编译会出错！
- 2、接口中可以定义“成员变量”，或者说是不可变的常量，因为接口中的“成员变量”会自动变为为public static final。可以通过类命名直接访问：ImplementClass.name。
- 3、接口中不存在实现的方法。
- 4、实现接口的非抽象类必须要实现该接口的所有方法。抽象类可以不用实现。
- 5、不能使用new操作符实例化一个接口，但可以声明一个接口变量，该变量必须引用（refer to）一个实现该接口的类的对象。可以使用 instanceof 检查一个对象是否实现了某个特定的接口。例如：
if(anObject instanceof Comparable){}。
- 6、在实现多接口的时候一定要避免方法名的重复。

二者区别

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	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。

现。因此你不需要改变你现在的代码。

总结

- 1、抽象类在java语言中所表示的是一种继承关系，一个子类只能存在一个父类，但是可以存在多个接口。
- 2、在抽象类中可以拥有自己的成员变量和非抽象类方法，但是接口中只能存在静态的不可变的成员数据（不过一般都不在接口中定义成员数据），而且它的所有方法都是抽象的。
- 3、抽象类和接口所反映的设计理念是不同的，抽象类所代表的是“is-a”的关系，而接口所代表的是“like-a”的关系。

抽象类和接口是java语言中两种不同的抽象概念，他们的存在对多态提供了非常好的支持，虽然他们之间存在很大的相似性。但是对于他们的选择往往反应了您对问题域的理解。只有对问题域的本质有良好的理解，才能做出正确、合理的设计。

参考资料

<https://blog.csdn.net/chenssy/article/details/12858267>

<http://www.importnew.com/12399.html>

作者：方志朋，一线大厂架构师，

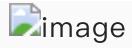
来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



公众号：方志朋

Java基础：Java 中的 ==, equals 与 hashCode 的区别与联系

Java 中的 ==, equals 与 hashCode 的区别与联系

概述

概念：

- == : 该操作符生成的是一个boolean结果，它计算的是操作数的值之间的关系
- equals : Object 的实例方法，比较两个对象的content是否相同
- hashCode : Object 的 native方法，获取对象的哈希值，用于确定该对象在哈希表中的索引位置，它实际上是一个int型整数

二、关系操作符 ==

1、操作数的值

基本数据类型变量

在Java中有八种基本数据类型：

- 浮点型：float(4 byte), double(8 byte)
- 整型：byte(1 byte), short(2 byte), int(4 byte) , long(8 byte)
- 字符型: char(2 byte)
- 布尔型: boolean(JVM规范没有明确规定其所占的空间大小，仅规定其只能取字面值”true”和”false”)

对于这八种基本数据类型的变量，变量直接存储的是“值”。因此，在使用关系操作符 == 来进行比较时，比较的就是“值”本身。要注意的是，浮点型和整型都是有符号类型的（最高位仅用于表示正负，不参与计算【以 byte 为例，其范围为 $-2^7 \sim 2^7 - 1$ ，-0即-128】），而char是无符号类型的（所有位均参与计算，所以char类型取值范围为 $0 \sim 2^{16} - 1$ ）。

公众号：方志朋

引用类型变量

在Java中，引用类型的变量存储的并不是“值”本身，而是与其关联的对象在内存中的地址。比如下面这行代码：

```
1 String str1;
```

这句话声明了一个引用类型的变量，此时它并没有和任何对象关联。

而通过 new 来产生一个对象，并将这个对象和str1进行绑定：

```
1 str1= new String("hello");
```

那么 str1 就指向了这个对象，此时引用变量str1中存储的是它指向的对象在内存中的存储地址，并不是“值”本身，也就是说并不是直接存储的字符串”hello”。这里面的引用和 C/C++ 中的指针很类似。

小结

因此，对于关系操作符 ==：

- 若操作数的类型是基本数据类型，则该关系操作符判断的是左右两边操作数的值是否相等
- 若操作数的类型是引用数据类型，则该关系操作符判断的是左右两边操作数的内存地址是否相同。也就是说，若此时返回true，则该操作符作用的一定是同一个对象。

equals

在初学Java的时候，很多人会说在比较对象的时候，==是比较地址，equals()是比较对象的内容，谁说的？

看看equals()方法在Object类中的定义：

```
1 public boolean equals(Object obj){  
2     return (this == obj);  
3 }
```

这是比较内容么？明显是比较指针(地址)么...

但是为什么会有equals是比较内容的这种说法呢？

因为在String、Double等封装类中，已经重载(overriding)了Object类的equals()方法，于是有了另一种计算公式，是进行内容的比较。

比如在String类中：

```
1
2 public int hashCode() {
3     int h = hash;
4     if (h == 0) {
5         char val[] = value;
6         int len = count;
7         for (int i = 0; i < len; i++) {
8             h = 31*h + val[i];
9         }
10        hash = h;
11    }
12    return h;
13 }
```

hashCode

在Object类中的定义为：

```
1 public native int hashCode();
```

是一个本地方法，返回的对象的地址值。

但是，同样的思路，在String等封装类中对此方法进行了重写。方法调用得到一个计算公式得到的 int值。

hashCode和equals的关系

- 1、如果两个对象相同（即用equals比较返回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（即用equals比较返回false）
- 原因：从散列的角度考虑，不同的对象计算哈希码的时候，可能引起冲突，大家一定还记得数据结构中。

自我的理解：由于为了提高程序的效率才实现了hashCode方法，先进行hashCode的比较，如果不同，那就没必要在进行equals的比较了，这样就大大减少了equals比较的次数，这对比需要比较的数量很大的效率提高是很明显的，一个很好的例子就是在集合中的使用；

我们都知道java中的List集合是有序的，因此是可以重复的，而set集合是无序的，因此是不能重复的，那么怎么能保证不能被放入重复的元素呢，但靠equals方法一样比较的话，如果原来集合中以后又10000个元素了，那么放入10001个元素，难道要将前面的所有元素都进行比较，看看是否有重复，欧码噶的，这个效率可想而知，因此hashCode就应运而生了，java就采用了hash表，利用哈希算法（也叫散列算法），就是将对象数据根据该对象的特征使用特定的算法将其定义到一个地址上，那么在后面定义进来的数据只要看对应的hashCode地址上是否有值，那么就用equals比较，如果没有则直接插入，只要就大大减少了equals的使用次数，执行效率就大大提高了。

继续上面的话题，为什么必须要重写hashCode方法，其实简单的说就是为了保证同一个对象，保证在equals相同的情况下hashCode值必定相同，如果重写了equals而未重写hashCode方法，可能就会出现两个没有关系的对象equals相同的（因为equal都是根据对象的特征进行重写的），但hashCode确实不相同的。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

公众号：方志朋

Java基础：Java 内部类和静态内部类的区别

Java 内部类和静态内部类的区别

java 内部类和静态内部类的区别

下面说一说内部类（Inner Class）和静态内部类（Static Nested Class）的区别：

定义在一个类内部的类叫内部类，包含内部类的类称为外部类。内部类可以声明public、protected、private等访问限制，可以声明为abstract的供其他内部类或外部类继承与扩展，或者声明为static、final的，也可以实现特定的接口。外部类按常规的类访问方式使用内部类，唯一的差别是外部类可以访问内部类的所有方法与属性，**包括私有方法与属性。**

创建实例

- 内部类创建实例

```
OutClass.InnerClass obj = outClassInstance.new InnerClass(); //注意是外部类实例.new，内部类
```

- 静态内部类创建实例

```
AAA.StaticInner in = new AAA.StaticInner(); //注意是外部类本身，静态内部类
```

内部类访问外部类

内部类中的this与其他类一样是指的本身。创建内部类对象时，它会与创造它的外围对象有了某种联系，于是能访问外围类的所有成员，不需任何特殊条件，可理解为内部类链接到外部类。用外部类创建内部类对象时，此内部类对象会秘密的捕获一个指向外部类的引用，于是，可以通过这个引用来访问外围类的成员。

外部类访问内部类

内部类类似外部类的属性，因此访问内部类对象时总是需要一个创建好的外部类对象。内部类对象通过‘外部类名.this.xxx’的形式访问外部类的属性与方法。如：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
System.out.println("Print in inner Outer.index=" + pouter.this.index);
```

```
System.out.println("Print in inner Inner.index=" + this.index);
```

内部类向上转型

内部类也可以和普通类一样拥有向上转型的特性。将内部类向上转型为基类型，尤其是接口时，内部类就有了用武之地。如果内部类是private的，只可以被它的外部类访问，从而完全隐藏实现的细节。

方法内的类

方法内创建的类（注意方法中也能定义类），不能加访问修饰符。另外，方法内部的类也不是在调用方法时才会创建的，它们一样也被事先编译了。

静态内部类

定义静态内部类：在定义内部类的时候，可以在其前面加上一个权限修饰符static。此时这个内部类就变成了静态内部类。

通常称为嵌套类，当内部类是static时，意味着：

- [1]要创建嵌套类的对象，并不需要其外围类的对象；
- [2]不能从嵌套类的对象中访问非静态的外围类对象（不能够从静态内部类的对象中访问外部类的非静态成员）；

嵌套类与普通的内部类还有一个区别：普通内部类的字段与方法，只能放在类的外部层次上，所以普通的内部类不能有static数据和static字段，也不能包含嵌套类。但是在嵌套类里可以包含所有这些东西。也就是说，在非静态内部类中不可以声明静态成员，只有将某个内部类修饰为静态类，然后才能够在这个类中定义静态的成员变量与成员方法。

另外，在创建静态内部类时不需要将静态内部类的实例绑定在外部类的实例上。普通非静态内部类的对象是依附在外部类对象之中的，要在一个外部类中定义一个静态的内部类，不需要利用关键字new来创建内部类的实例。静态类和方法只属于类本身，并不属于该类的对象，更不属于其他外部类的对象。

内部类标识符

每个类会产生一个.class文件，文件名即为类名。同样，内部类也会产生这么一个.class文件，但是它的名称却不是内部类的类名，而是有着严格的限制：外围类的名字，加上\$，再加上内部类名字。

为何要用内部类？

- - a. 内部类一般只为其外部类使用；
 - b. 内部类提供了某种进入外部类的窗户；
 - c. 也是最吸引人的原因，每个内部类都能独立地继承一个类，而无论外部类是否已经继承了某个类。因此，内部类使多重继承的解决方案变得更加完整。

加深印象，参考一下：

```
1
2 public class OutClassTest {
3     static int a;
4
5     int b;
6
7     public static void test() {
8         System.out.println("outer class static function");
9     }
10
11    public static void main(String[] args) {
12        OutClassTest oc = new OutClassTest();
13        // new一个外部类
14        OutClassTest oc1 = new OutClassTest();
15        // 通过外部类的对象new一个非静态的内部类
16        OutClassTest.InnerClass no_static_inner = oc1.new InnerCl
17        ass();
18        // 调用非静态内部类的方法
19        System.out.println(no_static_inner.getKey());
20
21        // 调用静态内部类的静态变量
22        System.out.println(OutClassTest.InnerStaticClass.static_v
23        alue);
24        // 不依赖于外部类实例，直接实例化内部静态类
25        OutClassTest.InnerStaticClass inner = new OutClassTest.In
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
    innerStaticClass();

24        // 调用静态内部类的非静态方法
25        System.out.println(inner.getValue());
26        // 调用内部静态类的静态方法
27        System.out.println(OutClassTest.InnerStaticClass.getMessage());
28    }
29

30    private class InnerClass {
31        // 只有在静态内部类中才能够声明或定义静态成员
32        // private static String tt = "0";
33        private int flag = 0;
34

35        public InnerClass() {
36            // 三. 非静态内部类的非静态成员可以访问外部类的非静态变量和静态变
量
37            System.out.println("InnerClass create a:" + a);
38            System.out.println("InnerClass create b:" + b);
39            System.out.println("InnerClass create flag:" + flag);
40            //
41            System.out.println("InnerClass call outer static func
tion");
42            // 调用外部类的静态方法
43            test();
44        }
45

46        public String getKey() {
47            return "no-static-inner";
48        }
49    }

50

51    private static class InnerStaticClass {
52        // 静态内部类可以有静态成员，而非静态内部类则不能有静态成员。
53        private static String static_value = "0";
54

55        private int flag = 0;
56

57        public InnerStaticClass() {
58            System.out.println("InnerClass create a:" + a);
59            // 静态内部类不能够访问外部类的非静态成员
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
60         // System.out.println("InnerClass create b:" + b);
61         System.out.println("InnerStaticClass flag is " + fla
g);
62         System.out.println("InnerStaticClass tt is " + static
_value);
63     }
64
65     public int getValue() {
66         // 静态内部类访问外部类的静态方法
67         test();
68         return 1;
69     }
70
71     public static String getMessage() {
72         return "static-inner";
73     }
74 }
75
76     public OutClassTest() {
77         // new一个非静态的内部类
78         InnerClass ic = new InnerClass();
79         System.out.println("OuterClass create");
80     }
81
82 }
```

公众号：方志朋

总结

- 1.静态内部类可以有静态成员(方法，属性)，而非静态内部类则不能有静态成员(方法，属性)。
- 2.静态内部类只能够访问外部类的静态成员,而非静态内部类则可以访问外部类的所有成员(方法，属性)。
- 3.实例化一个非静态的内部类的方法：
 - a.先生成一个外部类对象实例
OutClassTest oc1 = new OutClassTest();
 - b.通过外部类的对象实例生成内部类对象
OutClassTest.InnerClass no_static_inner = oc1.new InnerClass();
- 4.实例化一个静态内部类的方法：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- a. 不依赖于外部类的实例,直接实例化内部类对象

```
OutClassTest.InnerStaticClass inner = new OutClassTest.InnerStaticClass();
```

- b. 调用内部静态类的方法或静态变量,通过类名直接调用

```
OutClassTest.InnerStaticClass.static_value
```

```
OutClassTest.InnerStaticClass.getMessage()
```

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

Java基础：Java中重载与重写的区别

Java中重载与重写区别的区别

重载(Overloading)

- 方法重载是让类以统一的方式处理不同类型数据的一种手段。多个同名函数同时存在，具有不同的参数个数/类型。重载Overloading是一个类中多态性的一种表现。
- Java的方法重载，就是在类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义。调用方法时通过传递给它们的不同参数个数和参数类型来决定具体使用哪个方法，这就是多态性。
- 重载的时候，方法名要一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准。

下面是重载的例子：

这是这个程序的第一种编程方法，在main方法中先创建一个Dog类实例，然后在Dog类的构造方法中利用this关键字调用不同的bark方法。

不同的重载方法bark是根据其参数类型的不同而区分的。

```
1 public class Dog {  
2     Dog()  
3     {  
4         this.bark();  
5     }  
6     void bark()//bark()方法是重载方法  
7     {  
8         System.out.println(\"no barking!\");  
9         this.bark(\"female\", 3.4);  
10    }  
11    void bark(String m,double l)//注意：重载的方法的返回值都是一样的,  
12    {  
13        System.out.println(\"a barking dog!\");  
14        this.bark(5, \"China\");  
15    }  
16    void bark(int a,String n)//不能以返回值区分重载方法，而只能  
以“参数类型”和“类名”来区分
```

公众号：方志朋

```
17     {
18         System.out.println("a howling dog");
19     }
20
21     public static void main(String[] args)
22     {
23         Dog dog = new Dog();
24         //dog.bark(); [Page]
25         //dog.bark(\"male\", \"yellow\");
26         //dog.bark(5, \"China\");
27     }
```

重写 (Overriding)

- 父类与子类之间的多态性，对父类的函数进行重新定义。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写 (Overriding)。在Java中，子类可继承父类中的方法，而不需要重新编写相同的方法。

但有时子类并不想原封不动地继承父类的方法，而是想作一定的修改，这就需要采用方法的重写。

方法重写又称方法覆盖。

- 若子类中的方法与父类中的某一方法具有相同的方法名、返回类型和参数表，则新方法将覆盖原有的方法。

如需父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类。

- 子类函数的访问修饰权限不能少于父类的；

重写方法的规则：

- 1、参数列表必须完全与被重写的方法相同，否则不能称其为重写而是重载。
- 2、返回的类型必须一直与被重写的方法的返回类型相同，否则不能称其为重写而是重载。
- 3、访问修饰符的限制一定要大于被重写方法的访问修饰符 (public>protected>default>private)
- 4、重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常。例如：
父类的一个方法申明了一个检查异常IOException，在重写这个方法时就不能抛出Exception,只能抛出IOException的子类异常，可以抛出非检查异常。

重载的规则：

- 1、必须具有不同的参数列表；
- 2、可以有不责骂的返回类型，只要参数列表不同就可以了；
- 3、可以有不同的访问修饰符；
- 4、可以抛出不同的异常；

重写与重载的区别在于：

- 重写多态性起作用，对调用被重载过的方法可以大大减少代码的输入量，同一个方法名只要往里面传递不同的参数就可以拥有不同的功能或返回值。
- 用好重写和重载可以设计一个结构清晰而简洁的类，可以说重写和重载在编写代码过程中的作用非同一般。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



Java基础：java 泛型详解-绝对是对泛型方法讲解最详细的，没有之一

对java的泛型特性的了解仅限于表面的浅浅一层，直到在学习设计模式时发现有不了解的用法，才想起详细的记录一下。本文参考java 泛型详解、Java中的泛型方法、 java泛型详解

概述

泛型在java中有很重要的地位，在面向对象编程及各种设计模式中有非常广泛的应用。什么是泛型？为什么要使用泛型？

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

一个栗子

一个被举了无数次的例子：

```
1 List arrayList = new ArrayList();
2 arrayList.add("aaaa");
3 arrayList.add(100);
4
5 for(int i = 0; i< arrayList.size(); i++){
6     String item = (String)arrayList.get(i);
7     Log.d("泛型测试", "item = " + item);
8 }
```

毫无疑问，程序的运行结果会以崩溃结束：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
1 java.lang.ClassCastException: java.lang.Integer cannot be cast to  
java.lang.String
```

ArrayList可以存放任意类型，例子中添加了一个String类型，添加了一个Integer类型，再使用时都以String的方式使用，因此程序崩溃了。为了解决类似这样的问题（在编译阶段就可以解决），泛型应运而生。

我们将第一行声明初始化list的代码更改一下，编译器会在编译阶段就能够帮我们发现类似这样的问题。

```
1 List<String> arrayList = new ArrayList<String>();  
2 ...  
3 //arrayList.add(100); 在编译阶段，编译器就会报错
```

特性

泛型只在编译阶段有效。看下面的代码：

```
1 List<String> stringArrayList = new ArrayList<String>();  
2 List<Integer> integerArrayList = new ArrayList<Integer>();  
3  
4 Class classStringArrayList = stringArrayList.getClass();  
5 Class classIntegerArrayList = integerArrayList.getClass();  
6  
7 if(classStringArrayList.equals(classIntegerArrayList)){  
8     Log.d("泛型测试","类型相同");  
9 }
```

输出结果：D/泛型测试: 类型相同。

通过上面的例子可以证明，在编译之后程序会采取去泛型化的措施。也就是说Java中的泛型，只在编译阶段有效。在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，泛型信息不会进入到运行时阶段。

对此总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。

泛型的使用

泛型有三种使用方式，分别为：泛型类、泛型接口、泛型方法

泛型类

泛型类型用于类的定义中，被称为泛型类。通过泛型可以完成对一组类的操作对外开放相同的接口。最典型的就是各种容器类，如：List、Set、Map。

泛型类的最基本写法（这么看可能会有点晕，会在下面的例子中详解）：

```
1 class 类名称 <泛型标识：可以随便写任意标识号，标识指定的泛型的类型>{  
2     private 泛型标识 /*（成员变量类型）*/ var;  
3     ....  
4 }  
5 }  
6 }
```

一个最普通的泛型类：

```
//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型  
//在实例化泛型类时，必须指定T的具体类型
```

```
1 public class Generic<T>{  
2     //key这个成员变量的类型为T，T的类型由外部指定  
3     private T key;  
4  
5     public Generic(T key) { //泛型构造方法形参key的类型也为T，T的类型由  
       外部指定  
6         this.key = key;  
7     }  
8  
9     public T getKey(){ //泛型方法getKey的返回值类型为T，T的类型由外部指定  
10        return key;  
11    }  
12 }
```

```
1 //泛型的类型参数只能是类类型（包括自定义类），不能是简单类型
2 //传入的实参类型需与泛型的类型参数类型相同，即为Integer.
3 Generic<Integer> genericInteger = new Generic<Integer>(123456);
4
5 //传入的实参类型需与泛型的类型参数类型相同，即为String.
6 Generic<String> genericString = new Generic<String>("key_vlaue");
7 Log.d("泛型测试","key is " + genericInteger.getKey());
8 Log.d("泛型测试","key is " + genericString.getKey());
```

```
1 12-27 09:20:04.432 13063-13063/? D/泛型测试: key is 123456
2 12-27 09:20:04.432 13063-13063/? D/泛型测试: key is key_vlaue
```

定义的泛型类，就一定要传入泛型类型实参么？并不是这样，在使用泛型的时候如果传入泛型实参，则会根据传入的泛型实参做相应的限制，此时泛型才会起到本应起到的限制作用。如果不传入泛型类型实参的话，在泛型类中使用泛型的方法或成员变量定义的类型可以为任何的类型。

看一个例子：

```
1 Generic generic = new Generic("111111");
2 Generic generic1 = new Generic(4444);
3 Generic generic2 = new Generic(55.55);
4 Generic generic3 = new Generic(false);
5
6 Log.d("泛型测试","key is " + generic.getKey());
7 Log.d("泛型测试","key is " + generic1.getKey());
8 Log.d("泛型测试","key is " + generic2.getKey());
9 Log.d("泛型测试","key is " + generic3.getKey());
```

```
1 D/泛型测试: key is 111111
2 D/泛型测试: key is 4444
3 D/泛型测试: key is 55.55
4 D/泛型测试: key is false
```

公众号：方志朋

注意：

- 泛型的类型参数只能是类型，不能是简单类型。
- 不能对确切的泛型类型使用instanceof操作。如下面的操作是非法的，编译时会出错。

```
1 if(ex_num instanceof Generic<Number>){  
2 }
```

泛型接口

泛型接口与泛型类的定义及使用基本相同。泛型接口常被用在各种类的生产器中，可以看一个例子：

```
1 //定义一个泛型接口  
2 public interface Generator<T> {  
3     public T next();  
4 }
```

当实现泛型接口的类，未传入泛型实参时：

```
1 /**  
2  * 未传入泛型实参时，与泛型类的定义相同，在声明类的时候，需将泛型的声明也一起加  
3  * 到类中  
4  * 即： class FruitGenerator<T> implements Generator<T>{  
5  * 如果不声明泛型，如： class FruitGenerator implements Generator<T>，  
6  * 编译器会报错："Unknown class"  
7  */  
8 class FruitGenerator<T> implements Generator<T>{  
9     @Override  
10    public T next() {  
11        return null;  
12    }  
13 }
```

当实现泛型接口的类，传入泛型实参时：

```
1
2 /**
3  * 传入泛型实参时：
4  * 定义一个生产器实现这个接口，虽然我们只创建了一个泛型接口Generator<T>
5  * 但是我们可以为T传入无数个实参，形成无数种类型的Generator接口。
6  * 在实现类实现泛型接口时，如已将泛型类型传入实参类型，则所有使用泛型的地方都要
7  * 替换成传入的实参类型
8  * 即：Generator<T>, public T next();中的的T都要替换成传入的String类型。
9
10
11    private String[] fruits = new String[]{"Apple", "Banana", "Pe
12    ar"};
13
14    @Override
15    public String next() {
16        Random rand = new Random();
17        return fruits[rand.nextInt(3)];
18    }

```

公众号：方志朋

泛型通配符

我们知道Integer是Number的一个子类，同时在特性章节中我们也验证过Generic与Generic实际上是相同的一种基本类型。那么问题来了，在使用Generic作为形参的方法中，能否使用Generic的实例传入呢？在逻辑上类似于Generic和Generic是否可以看成具有父子关系的泛型类型呢？

为了弄清楚这个问题，我们使用Generic这个泛型类继续看下面的例子：

```
1 public void showKeyValue1(Generic<Number> obj){
2     Log.d("泛型测试","key value is " + obj.getKey());
3 }
```

```
1 Generic<Integer> gInteger = new Generic<Integer>(123);
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
2 Generic<Number> gNumber = new Generic<Number>(456);
3
4 showKeyValue(gNumber);
5
6 // showKeyValue这个方法编译器会为我们报错：Generic<java.lang.Integer>
7 // cannot be applied to Generic<java.lang.Number>
8 // showKeyValue(gInteger);
```

通过提示信息我们可以看到Generic不能被看作为`Generic的子类。由此可以看出:同一种泛型可以对应多个版本（因为参数类型是不确定的），不同版本的泛型类实例是不兼容的。

回到上面的例子，如何解决上面的问题？总不能为了定义一个新的方法来处理Generic类型的类，这显然与java中的多态理念相违背。因此我们需要一个在逻辑上可以表示同时是Generic和Generic父类的引用类型。由此类型通配符应运而生。

我们可以将上面的方法改一下：

```
1 public void showKeyValue1(Generic<?> obj){
2     Log.d("泛型测试","key value is " + obj.getKey());
3 }
```

类型通配符一般是使用？代替具体的类型实参，注意了，此处？是类型实参，而不是类型形参。重要说三遍！此处？是类型实参，而不是类型形参！此处？是类型实参，而不是类型形参！再直白点的意思就是，此处的？和Number、String、Integer一样都是一种实际的类型，可以把？看成所有类型的父类。是一种真实的类型。

可以解决当具体类型不确定的时候，这个通配符就是？；当操作类型时，不需要使用类型的具体功能时，只使用Object类中的功能。那么可以用？通配符来表未知类型。

泛型方法

在java中，泛型类的定义非常简单，但是泛型方法就比较复杂了。

尤其是我们见到的大多数泛型类中的成员方法也都使用了泛型，有的甚至泛型类中也包含着泛型方法，这样在初学者中非常容易将泛型方法理解错了。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

泛型类，是在实例化类的时候指明泛型的具体类型；泛型方法，是在调用方法的时候指明泛型的具体类型

```
1 /**
2  * 泛型方法的基本介绍
3  * @param tClass 传入的泛型实参
4  * @return T 返回值为T类型
5  * 说明：
6  *      1) public 与 返回值中间<T>非常重要，可以理解为声明此方法为泛型方法。
7  *      2) 只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型方法。
8  *      3) <T>表明该方法将使用泛型类型T，此时才可以在方法中使用泛型类型T。
9  *      4) 与泛型类的定义一样，此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型。
10 */
11 public <T> T genericMethod(Class<T> tClass) throws InstantiationException,
12     IllegalAccessException{
13     T instance = tClass.newInstance();
14     return instance;
15 }
```

公众号：方志朋

```
1 Object obj = genericMethod(Class.forName("com.test.test"));
```

泛型方法的基本用法

光看上面的例子有的同学可能依然会非常迷糊，我们再通过一个例子，把我泛型方法再总结一下。

```
1 public class GenericTest {
2     //这个类是个泛型类，在上面已经介绍过
3     public class Generic<T>{
4         private T key;
5
6         public Generic(T key) {
7             this.key = key;
```

```
8     }
9
10    //我想说的其实是这个，虽然在方法中使用了泛型，但是这并不是一个泛型方
11    //法。
12    //这只是类中一个普通的成员方法，只不过他的返回值是在声明泛型类已经声明
13    //过的泛型。
14    //所以在这个方法中才可以继续使用 T 这个泛型。
15    public T getKey(){
16        return key;
17    }
18    /**
19     * 这个方法显然是有问题的，在编译器会给我们提示这样的错误信息"cannot
20     reslove symbol E"
21     * 因为在类的声明中并未声明泛型E，所以在使用E做形参和返回值类型时，编
22     译器会无法识别。
23     public E setKey(E key){
24         this.key = key
25     }
26     /**
27     * 这才是一个真正的泛型方法。
28     * 首先在public与返回值之间的<T>必不可少，这表明这是一个泛型方法，并且声
29     明了一个泛型T
30     * 这个T可以出现在这个泛型方法的任意位置。
31     * 泛型的数量也可以为任意多个
32     * 如：public <T,K> K showKeyName(Generic<T> container){
33     *     ...
34     * }
35     public <T> T showKeyName(Generic<T> container){
36         System.out.println("container key :" + container.getKey()
37     );
38     //当然这个例子举的不太合适，只是为了说明泛型方法的特性。
39     T test = container.getKey();
40     return test;
41 }
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
42     //这也不是一个泛型方法，这就是一个普通的方法，只是使用了Generic<Number>
43     //这个泛型类做形参而已。
44
45     public void showKeyValue1(Generic<Number> obj){
46         Log.d("泛型测试","key value is " + obj.getKey());
47     }
48
49     //这也不是一个泛型方法，这也是一个普通的方法，只不过使用了泛型通配符？
50     //同时这也印证了泛型通配符章节所描述的，?是一种类型实参，可以看做为Number
51     //等所有类的父类
52
53     public void showKeyValue2(Generic<?> obj){
54         Log.d("泛型测试","key value is " + obj.getKey());
55     }
56
57     /**
58      * 这个方法是有问题的，编译器会为我们提示错误信息："UnKnown class 'E'
59      *
60      * 虽然我们声明了<T>，也表明了这是一个可以处理泛型的类型的泛型方法。
61      * 但是只声明了泛型类型T，并未声明泛型类型E，因此编译器并不知道该如何处理E
62      * 这个类型。
63
64      * 对于编译器来说T这个类型并未项目中声明过，因此编译也不知道该如何编译这个
65      * 类。
66      * 所以这也不是一个正确的泛型方法声明。
67
68      public void showkey(T genericObj){
69
70
71      public static void main(String[] args) {
72
73
74  }
```

公众号：方志朋

类中的泛型方法

当然这并不是泛型方法的全部，泛型方法可以出现在任何地方和任何场景中使用。但是有一种情况是非常特殊的，当泛型方法出现在泛型类中时，我们再通过一个例子看一下

```
1 public class GenericFruit {  
2     class Fruit{  
3         @Override  
4         public String toString() {  
5             return "fruit";  
6         }  
7     }  
8  
9     class Apple extends Fruit{  
10        @Override  
11        public String toString() {  
12            return "apple";  
13        }  
14    }  
15  
16    class Person{  
17        @Override  
18        public String toString() {  
19            return "Person";  
20        }  
21    }  
22  
23    class GenerateTest<T>{  
24        public void show_1(T t){  
25            System.out.println(t.toString());  
26        }  
27  
28        //在泛型类中声明了一个泛型方法，使用泛型E，这种泛型E可以为任意类型。可以类型与T相同，也可以不同。  
29        //由于泛型方法在声明的时候会声明泛型<E>，因此即使在泛型类中并未声明泛型，编译器也能够正确识别泛型方法中识别的泛型。  
30    }
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
30     public <E> void show_3(E t){  
31         System.out.println(t.toString());  
32     }  
33  
34     //在泛型类中声明了一个泛型方法，使用泛型T，注意这个T是一种全新的类型，  
可以与泛型类中声明的T不是同一种类型。  
35     public <T> void show_2(T t){  
36         System.out.println(t.toString());  
37     }  
38 }  
39  
40     public static void main(String[] args) {  
41         Apple apple = new Apple();  
42         Person person = new Person();  
43  
44         GenerateTest<Fruit> generateTest = new GenerateTest<Fruit  
>();  
45         //apple是Fruit的子类，所以这里可以  
46         generateTest.show_1(apple);  
47         //编译器会报错，因为泛型类型实参指定的是Fruit，而传入的实参类是Perso  
n  
48         //generateTest.show_1(person);  
49  
50         //使用这两个方法都可以成功  
51         generateTest.show_2(apple);  
52         generateTest.show_2(person);  
53  
54         //使用这两个方法也都可以成功  
55         generateTest.show_3(apple);  
56         generateTest.show_3(person);  
57     }  
58 }
```

公众号：方志朋

泛型方法与可变参数

再看一个泛型方法和可变参数的例子：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
1 public <T> void printMsg( T... args){  
2     for(T t : args){  
3         Log.d("泛型测试","t is " + t);  
4     }  
5 }
```

```
1 printMsg("111",222,"aaaa","2323.4",55.55);
```

静态方法与泛型

静态方法有一种情况需要注意一下，那就是在类中的静态方法使用泛型：静态方法无法访问类上定义的泛型；如果静态方法操作的引用数据类型不确定的时候，必须要将泛型定义在方法上。

即：如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法。

```
1 public class StaticGenerator<T> {  
2     ....  
3     ....  
4     /**  
5      * 如果在类中定义使用泛型的静态方法，需要添加额外的泛型声明（将这个方法定义  
6      * 成泛型方法）  
7      * 即使静态方法要使用泛型类中已经声明过的泛型也不可以。  
8      * 如：public static void show(T t){..}，此时编译器会提示错误信息：  
9      * "StaticGenerator cannot be referenced from static context  
10     *"  
11  
12 }  
13 }
```

泛型方法总结

泛型方法能使方法独立于类而产生变化，以下是一个基本的指导原则：

无论何时，如果你能做到，你就该尽量使用泛型方法。也就是说，如果使用泛型方法将整个类泛型化，那么就应该使用泛型方法。另外对于一个static的方法而已，无法访问泛型类型的参数。所以如果static方法要使用泛型能力，就必须使其成为泛型方法。

泛型上下边界

在使用泛型的时候，我们还可以为传入的泛型类型实参进行上下边界的限制，如：类型实参只准传入某种类型的父类或某种类型的子类。

为泛型添加上边界，即传入的类型实参必须是指定类型的子类型

```
1 public void showKeyValue1(Generic<? extends Number> obj){  
2     Log.d("泛型测试","key value is " + obj.getKey());  
3 }
```

```
1 Generic<String> generic1 = new Generic<String>("11111");  
2 Generic<Integer> generic2 = new Generic<Integer>(2222);  
3 Generic<Float> generic3 = new Generic<Float>(2.4f);  
4 Generic<Double> generic4 = new Generic<Double>(2.56);  
5  
6 //这一行代码编译器会提示错误，因为String类型并不是Number类型的子类  
7 //showKeyValue1(generic1);  
8  
9 showKeyValue1(generic2);  
10 showKeyValue1(generic3);  
11 showKeyValue1(generic4);
```

如果我们把泛型类的定义也改一下：

```
1 public class Generic<T extends Number>{  
2     private T key;  
3  
4     public Generic(T key) {  
5         this.key = key;
```

```
6     }
7
8     public T getKey(){
9         return key;
10    }
11 }
```

//这一行代码也会报错，因为String不是Number的子类

```
1 Generic<String> generic1 = new Generic<String>("11111");
```

再来一个泛型方法的例子：

```
1 //在泛型方法中添加上下边界限制的时候，必须在权限声明与返回值之间的<T>上添加上下边界，即在泛型声明的时候添加
2 //public <T> T showKeyName(Generic<T extends Number> container)，编译器会报错："Unexpected bound"
3 public <T extends Number> T showKeyName(Generic<T> container){
4     System.out.println("container key :" + container.getKey());
5     T test = container.getKey();
6     return test;
7 }
```

通过上面的两个例子可以看出：泛型的上下边界添加，必须与泛型的声明在一起。

关于泛型数组要提一下

看到了很多文章中都会提起泛型数组，经过查看sun的说明文档，在java中是”不能创建一个确切的泛型类型的数组”的。

也就是说下面的这个例子是不可以的：

```
1 List<String>[] ls = new ArrayList<String>[10];
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

而使用通配符创建泛型数组是可以的，如下面这个例子：

```
1 List<?>[] ls = new ArrayList<?>[10];
```

这样也是可以的：

```
1 List<String>[] ls = new ArrayList[10];
```

下面使用Sun的一篇文档的一个例子来说明这个问题：

```
1 List<String>[] lsa = new List<String>[10]; // Not really allowed.
2 Object o = lsa;
3 Object[] oa = (Object[]) o;
4 List<Integer> li = new ArrayList<Integer>();
5 li.add(new Integer(3));
6 oa[1] = li; // Unsound, but passes run time store check
7 String s = lsa[1].get(0); // Run-time error: ClassCastException.
```

这种情况下，由于JVM泛型的擦除机制，在运行时JVM是不知道泛型信息的，所以可以给oa[1]赋上一个ArrayList而不会出现异常，但是在取出数据的时候却要做一次类型转换，所以就会出现ClassCastException，如果可以进行泛型数组的声明，上面说的这种情况在编译期将不会出现任何的警告和错误，只有在运行时才会出错。

而对泛型数组的声明进行限制，对于这样的情况，可以在编译期提示代码有类型安全问题，比没有任何提示要强很多。

下面采用通配符的方式是被允许的：数组的类型不可以是类型变量，除非是采用通配符的方式，因为对于通配符的方式，最后取出数据是要做显式的类型转换的。

```
1 List<?>[] lsa = new List<?>[10]; // OK, array of unbounded wildcard type.
2 Object o = lsa;
3 Object[] oa = (Object[]) o;
4 List<Integer> li = new ArrayList<Integer>();
5 li.add(new Integer(3));
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
6 oa[1] = li; // Correct.  
7 Integer i = (Integer) lsa[1].get(0); // OK
```

最后

本文中的例子主要是为了阐述泛型中的一些思想而简单举出的，并不一定有着实际的可用性。另外，一提到泛型，相信大家用到最多的就是在集合中，其实，在实际的编程过程中，自己可以使用泛型去简化开发，且能很好的保证代码质量。

原文链接

<https://blog.csdn.net/s10461/article/details/53941091>

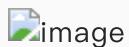
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号: 方志朋

程序员理财，请关注：



Java基础：int与Integer区别

int与Integer区别

int与Integer的基本使用对比

- Integer是int的包装类； int是基本数据类型；
- Integer变量必须实例化后才能使用； int变量不需要；
- Integer实际是对象的引用，指向此new的Integer对象； int是直接存储数据值；
- Integer的默认值是null； int的默认值是0。

int与Integer的深入对比

(1) 由于Integer变量实际上是对一个Integer对象的引用，所以两个通过new生成的Integer变量永远是不相等的（因为new生成的是两个对象，其内存地址不同）。

```
1 Integer i = new Integer(100);
2 Integer j = new Integer(100);
3 System.out.print(i == j); //false
```

(2) Integer变量和int变量比较时，只要两个变量的值是相等的，则结果为true（因为包装类Integer和基本数据类型int比较时，java会自动拆包装为int，然后进行比较，实际上就变为两个int变量的比较）

```
1 Integer i = new Integer(100);
2 int j = 100;
3 System.out.print(i == j); //true
```

(3) 非new生成的Integer变量和new Integer()生成的变量比较时，结果为false。（因为非new生成的Integer变量指向的是java常量池中的对象，而new Integer()生成的变量指向堆中新建的对象，两者在内存中的地址不同）

```
1 Integer i = new Integer(100);
2 Integer j = 100;
```

```
3 System.out.print(i == j); //false
```

(4) 对于两个非new生成的Integer对象，进行比较时，如果两个变量的值在区间-128到127之间，则比较结果为true，如果两个变量的值不在此区间，则比较结果为false

```
1 Integer i = 100;
2 Integer j = 100;
3 System.out.print(i == j); //true
4
5 Integer i = 128;
6 Integer j = 128;
7 System.out.print(i == j); //false
```

对于第4条的原因： java在编译Integer i = 100 ;时，会翻译成为Integer i = Integer.valueOf(100)。而java API中对Integer类型的valueOf的定义如下，对于-128到127之间的数，会进行缓存，Integer i = 127时，会将127进行缓存，下次再写Integer j = 127时，就会直接从缓存中取，就不会new了。

```
1 public static Integer valueOf(int i){
2     assert IntegerCache.high >= 127;
3     if (i >= IntegerCache.low && i <= IntegerCache.high){
4         return IntegerCache.cache[i + (-IntegerCache.low)];
5     }
6     return new Integer(i);
7 }
```

Java两种数据类型

Java两种数据类型分类

- 基本数据类型，分为boolean、byte、int、char、long、short、double、float；
- 引用数据类型，分为数组、类、接口。

Java为每个原始类型提供了封装类

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

为了编程的方便还是引入了基本数据类型，但是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。

基本数据类型: boolean, char, byte, short, int, long, float, double

封装类类型: Boolean, Character, Byte, Short, Integer, Long, Float, Double

基本解析

自动装箱：将基本数据类型重新转化为对象

```
1 public class Test {  
2     public static void main(String[] args) {  
3         //声明一个Integer对象  
4         Integer num = 9;  
5  
6         //以上的声明就是用到了自动的装箱：解析为：Integer num = new In  
7         teger(9);  
8     }  
9 }
```

9是属于基本数据类型的，原则上它是不能直接赋值给一个对象Integer的，但jdk1.5后你就可以进行这样的声明。自动将基本数据类型转化为对应的封装类型，成为一个对象以后就可以调用对象所声明的所有方法。

自动拆箱：将对象重新转化为基本数据类型

```
1 public class Test {  
2     public static void main(String[] args) {  
3         //声明一个Integer对象  
4         Integer num = 9;  
5  
6         //进行计算时隐含的有自动拆箱  
7         System.out.print(num--);  
8     }  
9 }
```

9 }

因为对象时不能直接进行运算的，而是要转化为基本数据类型后才能进行加减乘除。对比：

```
/装箱  
Integer num = 10;  
//拆箱  
int num1 = num;
```

深入解析

情况描述

```
1  
2 public class Test {  
3     public static void main(String[] args) {  
4         //在-128~127 之外的数  
5         Integer num1 = 128;    Integer num2 = 128;  
6         System.out.println(num1==num2);    //false  
7  
8         // 在-128~127 之内的数  
9         Integer num3 = 9;    Integer num4 = 9;  
10        System.out.println(num3==num4);    //true  
11    }  
12 }
```

解析原因：归结于java对于Integer与int的自动装箱与拆箱的设计，是一种模式：叫享元模式

（flyweight）。加大对简单数字的重利用，Java定义在自动装箱时对于值从-128到127之间的值，它们被装箱为Integer对象后，会存在内存中被重用，始终只存在一个对象。而如果超过了从-128到127之间的值，被装箱后的Integer对象并不会被重用，即相当于每次装箱时都新建一个 Integer对象。

Integer源码解析

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，源码如下：

```
1 public static Integer valueOf(String s, int radix) throws NumberFormatException {
2     return Integer.valueOf(parseInt(s, radix));
3 }
4
5 public static Integer valueOf(int i) {
6     assert IntegerCache.high >= 127;
7     if (i >= IntegerCache.low && i <= IntegerCache.high)
8         return IntegerCache.cache[i + (-IntegerCache.low)];
9     return new Integer(i);
10}
```

IntegerCache是Integer的内部类，源码如下：

```
1 /**
2  * 缓存支持自动装箱的对象标识语义
3  * -128和127（含）。
4  *
5  * 缓存在第一次使用时初始化。缓存的大小
6  * 可以由-XX: AutoBoxCacheMax = <size>选项控制。
7  * 在VM初始化期间，java.lang.Integer.IntegerCache.high属性
8  * 可以设置并保存在私有系统属性中
9  */
10
11 private static class IntegerCache {
12     static final int low = -128;
13     static final int high;
14     static final Integer cache[];
15
16     static {
17         // high value may be configured by property
18         int h = 127;
19         String integerCacheHighPropValue =
20             sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
21         if (integerCacheHighPropValue != null) {
22             int i = parseInt(integerCacheHighPropValue);
23             i = Math.max(i, 127);
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
24          // Maximum array size is Integer.MAX_VALUE
25          h = Math.min(i, Integer.MAX_VALUE - (-low) - 1);
26      }
27      high = h;
28
29      cache = new Integer[(high - low) + 1];
30      int j = low;
31      for(int k = 0; k < cache.length; k++)
32          cache[k] = new Integer(j++);
33  }
34
35  private IntegerCache() {}
36 }
```

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众账号：方志朋

程序员理财，请关注：



Java基础： Java基础： Java的反射机制

反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。【翻译于 官方文档】

本篇将从以下几个方面讲述反射的知识：

- class的使用
- 方法的反射
- 构造函数的反射
- 成员变量的反射

一、什么是class类

在面向对象的世界里，万物皆对象。类是对象，类是java.lang.Class类的实例对象。另外class类只有java虚拟机才能new出来。任何一个类都是Class类的实例对象。这实例对象有三种表达方式：

```
1 public class User{  
2 }  
3  
4 public class ClassTest{  
5 User u=new User();  
6 //方式1:  
7 Class c1=User.class;  
8 //方式2:  
9 Class c2=u.getClass();  
10 //方式3:  
11 Class c3=Class.forName("com.forezp.User");  
12  
13 //可以通过类的类型创建该类的实例对象  
14 User user=(User)c1.newInstance();  
15 }
```

二、class类的动态加载

Class.forName(类的全称);该方法不仅表示了类的类型，还代表了动态加载类。编译时刻加载类是静态加载、运行时刻加载类是动态加载类。

三、获取方法信息

基本的数据类型，void关键字都Class类的实例;可以通过getName();get SimpleName();获取类的名称。

```
1 Class c1=String.class;
2 Class c2=int.class;
3 Class c3=void.class;
4 System.out.println(c1.getName());
5 System.out.println(c2.getSimple Name());
```

获取类的所有方法，并打印出来：

```
1 public static void printClassInfo(Object object){
2     Class c=object.getClass();
3     System.out.println("类的名称：" +c.getName());
4
5     /**
6      * 一个成员方法就是一个method对象
7      * getMethod()所有的 public方法，包括父类继承的 public
8      * getDeclaredMethods()获取该类所有的方法，包括private，但不包
9      * 括继承的方法。
10     */
11     Method[] methods=c.getMethod(); //获取方法
12     //获取所以的方法，包括private，c.getDeclaredMethods();
13
14     for(int i=0;i<methods.length;i++){
15         //得到方法的返回类型
16         Class returnType=methods[i].getReturnType();
17         System.out.print(returnType.getName());
18         //得到方法名：
19         System.out.print(methods[i].getName()+"(");
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
19
20         Class[] parameterTypes=methods[i].getParameterTypes
21     () ;
22         for(Class class1:parameterTypes){
23             System.out.print(class1.getName()+" ,");
24         }
25         System.out.println(" )");
26     }
```

```
1 public class ReflectTest {
2
3     public static void main(String[] args){
4         String s="ss";
5         ClassUtil.printClassInfo(s);
6     }
7 }
```

运行：

```
类的名称：java.lang.String
booleanequals(java.lang.Object,
```

```
java.lang.StringtoString()
```

```
inhashCode()
```

```
...
```

四、获取成员变量的信息

也可以获取类的成员变量信息

```
1
2 public static void printFiledInfo(Object o){
3 }
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
4     Class c=o.getClass();
5
6     /**
7      * getFileds()获取public
8      * getDeclaredFields()获取所有
9      */
10
11    for(Field f:fileds){
12        //获取成员变量的类型
13        Class filedType=f.getType();
14        System.out.println(filedType.getName()+" "+f.getName
15    );
16
17 }
```

```
1 public static void main(String[] args){
2             String s="ss";
3             //ClassUtil.printClassInfo(s);
4             ClassUtil.printFiledInfo(s);
5 }
```

公众号：方志朋

运行：

```
[C value
int hash
long serialVersionUID
[Ljava.io.ObjectStreamField; serialPersistentFields
java.util.Comparator CASE_INSENSITIVE_ORDER
int HASHING_SEED
int hash32
```

五、获取构造函数的信息

```
1 public static void printConstructInfo(Object o){
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
2     Class c=o.getClass();
3
4     Constructor[] constructors=c.getDeclaredConstructors();
5     for (Constructor con:constructors){
6         System.out.print(con.getName()+"(");
7
8         Class[] typeParas=con.getParameterTypes();
9         for (Class class1:typeParas){
10             System.out.print(class1.getName()+" ,");
11         }
12         System.out.println(")");
13     }
14 }
```

```
1 public static void main(String[] args){
2     String s="ss";
3     //ClassUtil.printClassInfo(s);
4     //ClassUtil.printFiledInfo(s);
5     ClassUtil.printConstructInfo(s);
6 }
```

公众号：方志朋

运行：

```
java.lang.String([B ,)
java.lang.String([B ,int ,int ,)
java.lang.String([B ,java.nio.charset.Charset ,)
java.lang.String([B ,java.lang.String ,)
java.lang.String([B ,int ,int ,java.nio.charset.Charset ,)
java.lang.String(int ,int ,[C ,)
java.lang.String([C ,boolean ,)
java.lang.String(java.lang.StringBuilder ,)
java.lang.String(java.lang.StringBuffer ,)
```

...

六、方法反射的操作

获取一个方法：需要获取方法的名称和方法的参数才能决定一个方法。

方法的反射操作：

```
1 method.invoke(对象, 参数列表);
```

举个例子：

```
1 class A{  
2  
3     public void add(int a,int b){  
4         System.out.print(a+b);  
5     }  
6  
7     public void toUpper(String a){  
8         System.out.print(a.toUpperCase());  
9     }  
10 }
```

公众号：方志朋

```
1 public static void main(String[] args) {  
2     A a=new A();  
3     Class c=a.getClass();  
4     try {  
5         Method method=c.getMethod("add",new Class[]{int.class,  
6             int.class});  
7         //也可以 Method method=c.getMethod("add",int.class,in  
8             t.class);  
9         //方法的反射操作  
10        method.invoke(a,10,10);  
11    }catch (Exception e){  
12        e.printStackTrace();  
13    }  
14 }
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

运行：

20

本篇文章已经讲解了java反射的基本用法， 它可以在运行时判断任意一个对象所属的类；在运行时构造任意一个类的对象；在运行时判断任意一个类所具有的成员变量和方法；在运行时调用任意一个对象的方法；生成动态代理。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

Java基础：Java finally语句到底是在return之前还是之后执行？

网上有很多人探讨Java中异常捕获机制try...catch...finally块中的finally语句是不是一定会被执行？很多人都说不是，当然他们的回答是正确的，经过我试验，至少有两种情况下finally语句是不会被

- try语句没有被执行到，如在try语句之前就返回了，这样finally语句就不会执行，这也说明了finally语句被执行的必要而非充分条件是：相应的try语句一定被执行到。
- 在try块中有System.exit(0);这样的语句，System.exit(0);是终止Java虚拟机JVM的，连JVM都停止了，所有都结束了，当然finally语句也不会被执行到。

当然还有很多人探讨Finally语句的执行与return的关系，颇为让人迷惑，不知道finally语句是在try的return之前执行还是之后执行？我也是一头雾水，我觉得他们的说法都不正确，我觉得应该是：finally语句是在try的return语句执行之后，return返回之前执行。这样的说法有点矛盾，也许是我表述不太清楚，下面我给出自己试验的一些结果和示例进行佐证，有什么问题欢迎大家提出来。

finally语句在return语句执行之后return返回之前执行的。

```
1
2 public class FinallyTest1 {
3
4     public static void main(String[] args) {
5
6         System.out.println(test1());
7     }
8
9     public static int test1() {
10        int b = 20;
11
12        try {
13            System.out.println("try block");
14
15            return b += 80;
16        }
17        catch (Exception e) {
```

```
18
19         System.out.println("catch block");
20     }
21     finally {
22
23         System.out.println("finally block");
24
25         if (b > 25) {
26             System.out.println("b>25, b = " + b);
27         }
28     }
29
30     return b;
31 }
32
33 }
```

运行结果是：

```
1 try block
2 finally block
3 b>25, b = 100
4 100
```

说明return语句已经执行了再去执行finally语句，不过并没有直接返回，而是等finally语句执行完了再返回结果。

如果觉得这个例子还不足以说明这个情况的话，下面再加个例子加强证明结论：

```
1 public class FinallyTest1 {
2
3     public static void main(String[] args) {
4
5         System.out.println(test11());
6     }
7
8     public static String test11() {
```

```
9     try {
10         System.out.println("try block");
11
12         return test12();
13     } finally {
14         System.out.println("finally block");
15     }
16 }
17
18 public static String test12() {
19     System.out.println("return statement");
20
21     return "after return";
22 }
23
24 }
```

运行结果为：

```
1 try block
2 return statement
3 finally block
4 after return
```

说明try中的return语句先执行了但并没有立即返回，等到finally执行结束后再

这里大家可能会想：如果finally里也有return语句，那么是不是就直接返回了，try中的return就不能返回了？看下面。

finally块中的return语句会覆盖try块中的return返回。

```
1
2 public class FinallyTest2 {
3
4     public static void main(String[] args) {
5 }
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
6         System.out.println(test2());
7     }
8
9     public static int test2() {
10         int b = 20;
11
12         try {
13             System.out.println("try block");
14
15             return b += 80;
16         } catch (Exception e) {
17
18             System.out.println("catch block");
19         } finally {
20
21             System.out.println("finally block");
22
23             if (b > 25) {
24                 System.out.println("b>25, b = " + b);
25             }
26
27             return 200;
28         }
29
30         // return b;
31     }
32
33 }
```

公众号：方志朋

运行结果是：

```
1 try block
2 finally block
3 b>25, b = 100
4 200
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

这说明finally里的return直接返回了，就不管try中是否还有返回语句，这里还有个小细节需要注意，finally里加上return过后，finally外面的return b就变成不可到达语句了，也就是永远不能被执行到，所以需要注释掉否则编译器报错。

这里大家可能又想：如果finally里没有return语句，但修改了b的值，那么try中return返回的是修改后的值还是原值？看下面。

如果finally语句中没有return语句覆盖返回值，那么原来的返回值可能因为finally里的修改而改变也可能不变。

测试用例1：

```
1
2 public class FinallyTest3 {
3
4     public static void main(String[] args) {
5
6         System.out.println(test3());
7     }
8
9     public static int test3() {
10        int b = 20;
11
12        try {
13            System.out.println("try block");
14
15            return b += 80;
16        } catch (Exception e) {
17
18            System.out.println("catch block");
19        } finally {
20
21            System.out.println("finally block");
22
23            if (b > 25) {
24                System.out.println("b>25, b = " + b);
25            }
26        }
27    }
28}
```

公众号：方志朋

```
27         b = 150;
28     }
29
30     return 2000;
31 }
32 }
```

运行结果是：

```
1 try block
2 finally block
3 b>25, b = 100
4 100
```

测试用例2：

```
1
2 import java.util.*;
3
4 public class FinallyTest6
5 {
6     public static void main(String[] args) {
7         System.out.println(getMap().get("KEY").toString());
8     }
9
10    public static Map<String, String> getMap() {
11        Map<String, String> map = new HashMap<String, String>();
12        map.put("KEY", "INIT");
13
14        try {
15            map.put("KEY", "TRY");
16            return map;
17        }
18        catch (Exception e) {
19            map.put("KEY", "CATCH");
20        }
21        finally {
```

公众号：方志朋

```
22         map.put("KEY", "FINALLY");
23         map = null;
24     }
25
26     return map;
27 }
28 }
```

运行结果是：

FINALLY

为什么测试用例1中finally里的b = 150;并没有起到作用而测试用例2中finally的map.put("KEY", "FINALLY");起了作用而map = null;却没起作用呢？这就是Java到底是传值还是传址的问题了，具体请看精选30道Java笔试题解答，里面有详细的解答，简单来说就是：Java中只有传值没有传址，这也是为什么map = null这句不起作用。这同时也说明了返回语句是try中的return语句而不是 finally外面的return b;这句，不相信的话可以试下，将return b;改为return 294，对原来的结果没有一点影响。

这里大家可能又要想：是不是每次返回的一定是try中的return语句呢？那么finally外的return b不是一点作用没吗？请看下面。

try块里的return语句在异常的情况下不会被执行，这样具体返回哪个看情况。

```
1 public class FinallyTest4 {
2
3     public static void main(String[] args) {
4
5         System.out.println(test4());
6     }
7
8     public static int test4() {
9         int b = 20;
10
11     try {
12         System.out.println("try block");
13
14         b = b / 0;
15     } catch (Exception e) {
16         System.out.println("catch block");
17     }
18
19     }
20
21     public static void main(String[] args) {
22         System.out.println(test4());
23     }
24 }
```

```
15
16         return b += 80;
17     } catch (Exception e) {
18
19         b += 15;
20         System.out.println("catch block");
21     } finally {
22
23         System.out.println("finally block");
24
25         if (b > 25) {
26             System.out.println("b>25, b = " + b);
27         }
28
29         b += 50;
30     }
31
32     return 204;
33 }
34
35 }
```

公众号：方志朋

运行结果是：

```
1
2 try block
3 catch block
4 finally block
5 b>25, b = 35
6 85
```

这里因为在return之前发生了除0异常，所以try中的return不会被执行到，而是接着执行捕获异常的catch语句和最终的finally语句，此时两者对b的修改都影响了最终的返回值，这时return b;就起到作用了。当然如果你这里将return b改为return 300什么的，最后返回的就是300，这毋庸置疑。

这里大家可能又有疑问：如果catch中有return语句呢？当然只有在异常的情况下才有可能会执行，那么是在finally之前就返回吗？看下面。

当发生异常后， catch中的return执行情况与未发生异常时try中return的执行情况完全一样。

```
1 public class FinallyTest5 {  
2  
3     public static void main(String[] args) {  
4  
5         System.out.println(test5());  
6     }  
7  
8     public static int test5() {  
9         int b = 20;  
10  
11     try {  
12         System.out.println("try block");  
13  
14         b = b /0;  
15  
16         return b += 80;  
17     } catch (Exception e) {  
18  
19         System.out.println("catch block");  
20         return b += 15;  
21     } finally {  
22  
23         System.out.println("finally block");  
24  
25         if (b > 25) {  
26             System.out.println("b>25, b = " + b);  
27         }  
28  
29         b += 50;  
30     }  
31  
32     //return b;  
33 }  
34 }
```

公众号：方志朋

运行结果如下：

```
1 try block
2 catch block
3 finally block
4 b>25, b = 35
5 35
```

说明了发生异常后，catch中的return语句先执行，确定了返回值再去执行finally块，执行完了catch再返回，finally里对b的改变对返回值无影响，原因同前面一样，也就是说情况与try中的return语句执行完全一样。

最后总结：finally块的语句在try或catch中的return语句执行之后返回之前执行且finally里的修改语句可能影响也可能不影响try或catch中 return已经确定的返回值，若finally里也有return语句则覆盖try或catch中的return语句直接返回。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



Java基础： Java对象初始化过程

我们都知道， 创建对象是由 new关键字调用构造方法 返回类实例。

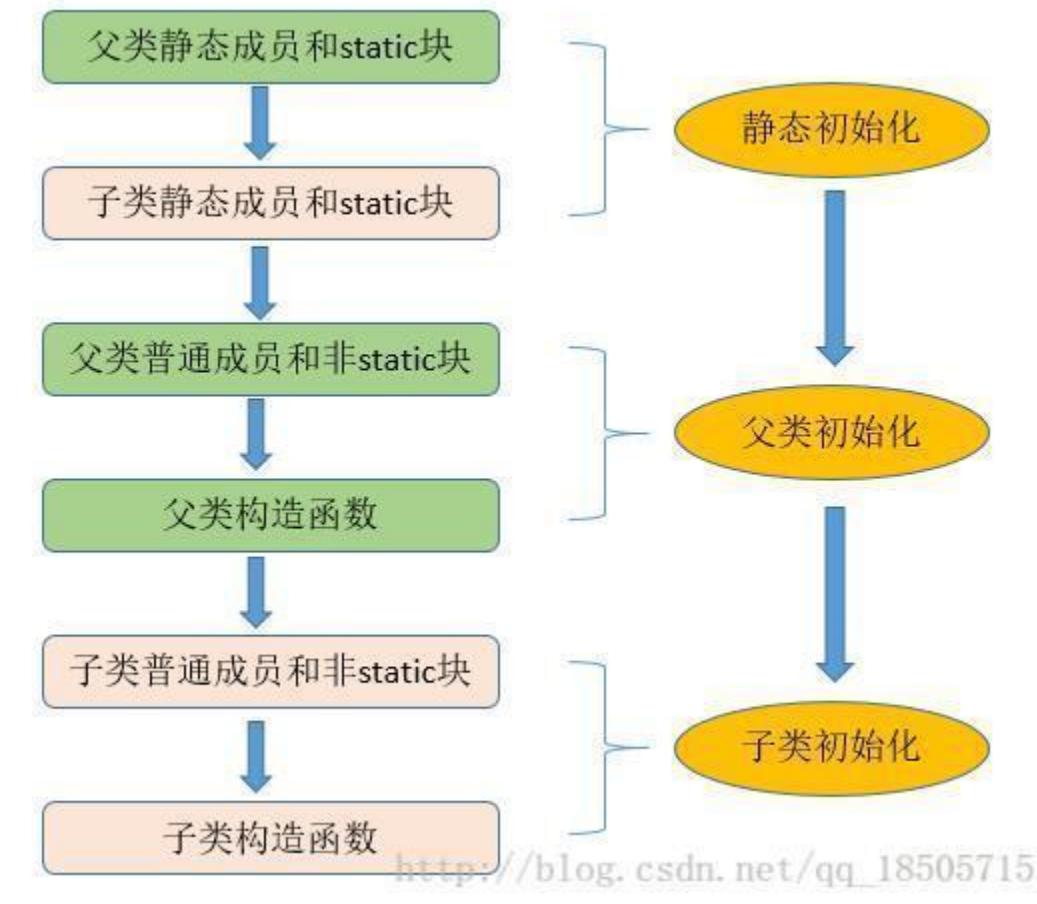
例如：Person jack = new Person();

这句话到底做了什么事情呢？ 其实就是讲对象的初始化过程。

- 1、new 用到了Person.class,所以会先找到Person.class文件，并加载到内存中(用到类中的内容类就会被加载)
- 2、执行该对象的static代码块(静态初始块)。(如果有的话，给Person.class类进行初始化)
- 3、在堆内存中开辟空间，分配内存地址
- 4、在堆内存中建立对象特有属性，并进行默认初始化
- 5、对属性进行显示初始化(声明成员属性并赋值)
- 6、执行普通初始块
- 7、执行构造函数
- 8、将内存地址赋值给栈内存中的jack变量

如下图：

公众号：方志朋



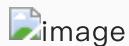
http://blog.csdn.net/qq_18505715

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



Java基础：面向接口编程详解

我想，对于各位使用面向对象编程语言的程序员来说，“接口”这个名词一定不陌生，但是不知各位有没有这样的疑惑：接口有什么用途？它和抽象类有什么区别？能不能用抽象类代替接口呢？而且，作为程序员，一定经常听到“面向接口编程”这个短语，那么它是什么意思？有什么思想内涵？和面向对象编程是什么关系？本文将一一解答这些疑问。

面向接口编程和面向对象编程是什么关系

首先，面向接口编程和面向对象编程并不是平级的，它并不是比面向对象编程更先进的一种独立的编程思想，而是附属于面向对象思想体系，属于其一部分。或者说，它是面向对象编程体系中的思想精髓之一。

接口的本质

接口，在表面上是由几个没有主体代码的方法定义组成的集合体，有唯一的名称，可以被类或其他接口所实现（或者也可以说继承）。它在形式上可能是如下的样子：

```
1 interface InterfaceName
2 {
3     void Method1();
4     void Method2(int para1);
5     void Method3(string para2,string para3);
6 }
```

那么，接口的本质是什么呢？或者说接口存在的意义是什么。我认为可以从以下两个视角考虑：

1) 接口是一组规则的集合，它规定了实现本接口的类或接口必须拥有一组规则。体现了自然界“如果你是……则必须能……”的理念。

例如，在自然界中，人都能吃饭，即“如果你是人，则必须能吃饭”。那么模拟到计算机程序中，就应该有一个IPerson（习惯上，接口名由“I”开头）接口，并有一个方法叫Eat()，然后我们规定，每一个表示“人”的类，必须实现IPerson接口，这就模拟了自然界“如果你是人，则必须能吃饭”这条规则。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

从这里，我想各位也能看到些许面向对象思想的东西。面向对象思想的核心之一，就是模拟真实世界，把真实世界中的事物抽象成类，整个程序靠各个类的实例互相通信、互相协作完成系统功能，这非常符合真实世界的运行状况，也是面向对象思想的精髓。

接口是在一定粒度视图上同类事物的抽象表示。注意这里我强调了在一定粒度视图上，因为“同类事物”这个概念是相对的，它因为粒度视图不同而不同。

例如，在我的眼里，我是一个人，和一头猪有本质区别，我可以接受我和我同学是同类这个说法，但绝不能接受我和一头猪是同类。但是，如果在一个动物学家眼里，我和猪应该是同类，因为我们都是动物，他可以认为“人”和“猪”都实现了IAnimal这个接口，而他在研究动物行为时，不会把我和猪分开对待，而会从“动物”这个较大的粒度上研究，但他会认为我和一棵树有本质区别。

现在换了一个遗传学家，情况又不同了，因为生物都能遗传，所以他眼里，我不仅和猪没区别，和一只蚊子、一个细菌、一颗树、一个蘑菇乃至一个SARS病毒都没什么区别，因为他会认为我们都实现了IDescendable这个接口（注：descend vi. 遗传），即我们都是可遗传的东西，他不会分别研究我们，而会将所有生物作为同类进行研究，在他眼里没有人和病毒之分，只有可遗传的物质和不可遗传的物质。但至少，我和一块石头还是有区别的。

可不幸的事情发生了，某日，地球上出现了一位伟大的人，他叫列宁，他在熟读马克思、恩格斯的辩证唯物主义思想巨著后，颇有心得，于是他下了一个著名的定义：所谓物质，就是能被意识所反映的客观实在。至此，我和一块石头、一丝空气、一条成语和传输手机信号的电磁场已经没什么区别了，因为在列宁的眼里，我们都是可以被意识所反映的客观实在。如果列宁是一名程序员，他会这么说：所谓物质，就是所有同时实现了“IReflectable”和“IEsse”两个接口的类所生成的实例。（注：reflect v. 反映 esse n. 客观实在）

也许你会觉得我上面的例子像在瞎掰，但是，这正是接口得以存在的意义。面向对象思想和核心之一叫做多态性，什么叫多态性？说白了就是在某个粒度视图层面上对同类事物不加区别的对待而统一处理。之所以敢这样做，就是因为有接口的存在。像那个遗传学家，他明白所有生物都实现了IDescendable接口，那只要是生物，一定有Descend () 这个方法，于是他就可以统一研究，而不至于分别研究每一种生物而最终累死。

可能这里还不能给你一个关于接口本质和作用的直观印象。那么在后文的例子和对几个设计模式的解析中，你将会更直观体验到接口的内涵。

面向接口编程综述

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

通过上文，我想大家对接口和接口的思想内涵有了一个了解，那么什么是面向接口编程呢？我个人的定义是：在系统分析和架构中，分清层次和依赖关系，每个层次不是直接向其上层提供服务（即不是直接实例化在上层中），而是通过定义一组接口，仅向上层暴露其接口功能，上层对于下层仅仅是接口依赖，而不依赖具体类。

这样做的好处是显而易见的，首先对系统灵活性大有好处。当下层需要改变时，只要接口及接口功能不变，则上层不用做任何修改。甚至可以在不改动上层代码时将下层整个替换掉，就像我们将一个WD的60G硬盘换成一个希捷的160G的硬盘，计算机其他地方不用做任何改动，而是把原硬盘拔下来、新硬盘插上就行了，因为计算机其他部分不依赖具体硬盘，而只依赖一个IDE接口，只要硬盘实现了这个接口，就可以替换上去。从这里看，程序中的接口和现实中的接口极为相似，所以我一直认为，接口（interface）这个词用的真是神似！

使用接口的另一个好处就是不同部件或层次的开发人员可以并行开工，就像造硬盘的不用等造CPU的，也不用等造显示器的，只要接口一致，设计合理，完全可以并行进行开发，从而提高效率。

本篇文章先到此。最后我想再啰嗦一句：面向对象的精髓是模拟现实，这也可以说是我这篇文章的灵魂。所以，多从现实中思考面向对象的东西，对提高系统分析设计能力大有裨益。

下篇文章，我将用一个实例来展示接口编程的基本方法。

而第三篇，我将解析经典设计模式中的一些面向接口编程思想，并解析一下.NET分层架构中的面向接口思想。

对本文的补充：

仔细看了各位的回复，非常高兴能和大家一起讨论技术问题。感谢给出肯定的朋友，也要感谢提出意见和质疑的朋友，这促使我更深入思考一些东西，希望能借此进步。在这里我想补充一些东西，以讨论一些回复中比较集中的问题。

1.关于“面向接口编程”中的“接口”与具体面向对象语言中“接口”两个词

看到有朋友提出“面向接口编程”中的“接口”二字应该比单纯编程语言中的interface范围更大。我经过思考，觉得很有道理。这里我写得确实不太合理。我想，面向对象语言中的“接口”是指具体的一种代码结构，例如C#中用interface关键字定义的接口。而“面向接口编程”中的“接口”可以说是一种从软件架构的角度、从一个更抽象的层面上指那种用于隐藏具体底层类和实现多态性的结构部件。从这个意义上说，如果定义一个抽象类，并且目的是为了实现多态，那么我认为把这个抽象类也称为“接口”是合理的。但是用抽象类实现多态合理不合理？在下面第二条讨论。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

概括来说，我觉得两个“接口”的概念既相互区别又相互联系。“面向接口编程”中的接口是一种思想层面的用于实现多态性、提高软件灵活性和可维护性的架构部件，而具体语言中的“接口”是将这种思想中的部件具体实施到代码里的手段。

2.关于抽象类与接口

看到回复中这是讨论的比较激烈的一个问题。很抱歉我考虑不周没有在文章中讨论这个问题。我个人对这个问题的理解如下：

如果单从具体代码来看，对这两个概念很容易模糊，甚至觉得接口就是多余的，因为单从具体功能来看，除多重继承外（C#，Java中），抽象类似乎完全能取代接口。但是，难道接口的存在是为了实现多重继承？当然不是。我认为，抽象类和接口的区别在于使用动机。使用抽象类是为了代码的复用，而使用接口的动机是为了实现多态性。所以，如果你在为某个地方该使用接口还是抽象类而犹豫不决时，那么可以想想你的动机是什么。

看到有朋友对IPerson这个接口的质疑，我个人的理解是，IPerson这个接口该不该定义，关键看具体应用中是怎么个情况。如果我们的项目中有Women和Man，都继承Person，而且Women和Man绝大多数方法都相同，只有一个方法DoSomethingInWC () 不同（例子比较粗俗，各位见谅），那么当然定义一个AbstractPerson抽象类比较合理，因为它可以把其他所有方法都包含进去，子类只定义DoSomethingInWC ()，大大减少了重复代码量。

但是，如果我们程序中的Women和Man两个类基本没有共同代码，而且有一个PersonHandle类需要实例化他们，并且不希望知道他们是男是女，而只需把他们当作人看待，并实现多态，那么定义成接口就有必要了。

总而言之，接口与抽象类的区别主要在于使用的动机，而不在于其本身。而一个东西该定义成抽象类还是接口，要根据具体环境的上下文决定。

再者，我认为接口和抽象类的另一个区别在于，抽象类和它的子类之间应该是一般和特殊的关系，而接口仅仅是它的子类应该实现的一组规则。（当然，有时也可能存在一般与特殊的关系，但我们使用接口的目的不在这里）如，交通工具定义成抽象类，汽车、飞机、轮船定义成子类，是可以接受的，因为汽车、飞机、轮船都是一种特殊的交通工具。再譬如Icomparable接口，它只是说，实现这个接口的类必须要可以进行比较，这是一条规则。如果Car这个类实现了Icomparable，只是说，我们的Car中有一个方法可以对两个Car的实例进行比较，可能是比哪辆车更贵，也可能比哪辆车更大，这都无所谓，但我们不能说“汽车是一种特殊的可以比较”，这在文法上都不通。

原文链接

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

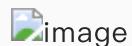
<http://www.cnblogs.com/leoo2sk/archive/2008/04/10/1146447.html>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

Java基础： Java容器之LinkedList

Java容器之LinkedList

定义

实现List接口与Deque接口双向链表，实现了列表的所有操作，并且允许包括null值的所有元素，对于LinkedList定义我产生了如下疑问：

- 1.Deque接口是什么，定义了一个怎样的规范？
- 2.LinkedList是双向链表，其底层实现是怎样的，具体包含哪些操作？

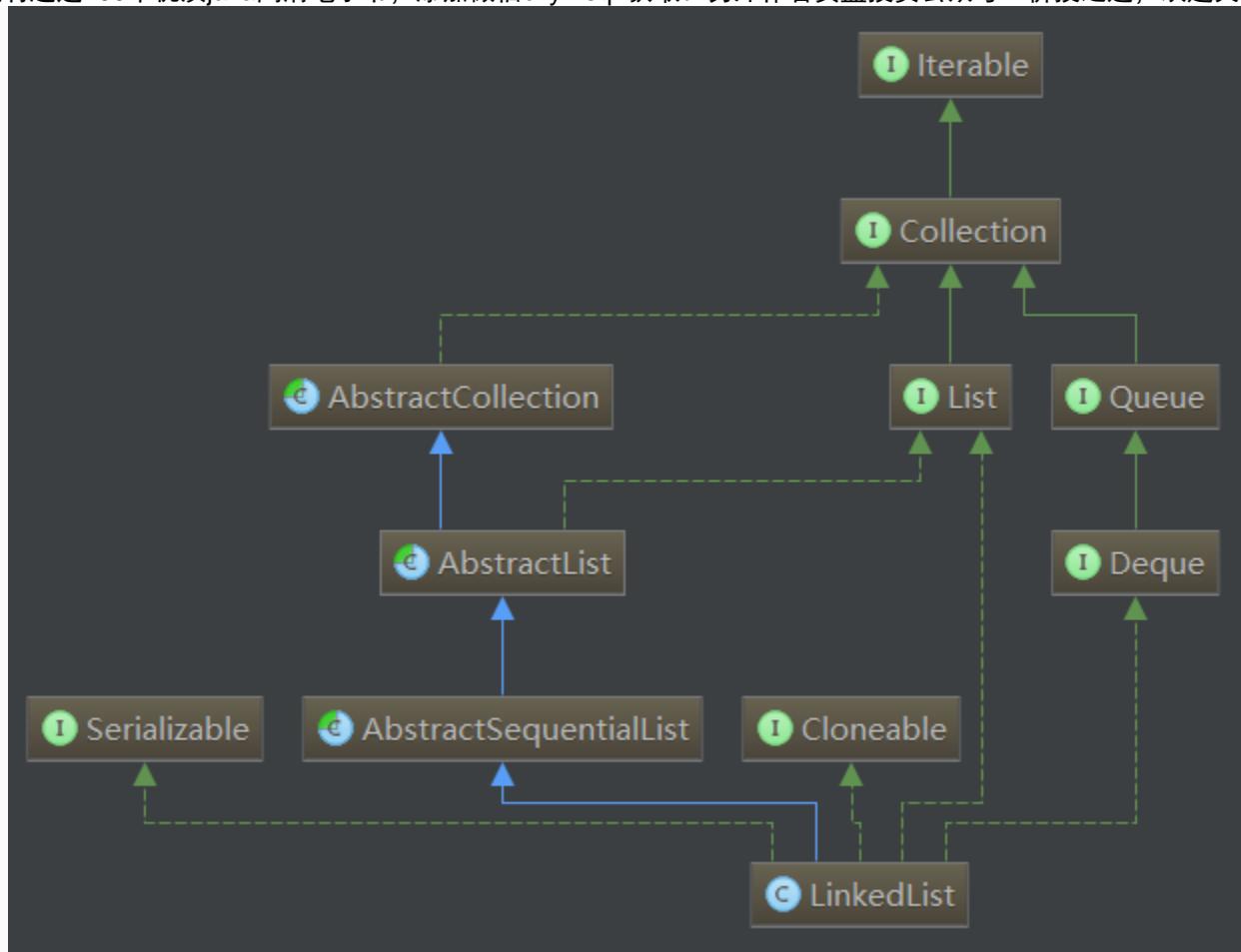
下文将围绕这两个问题进行，去探寻LinkedList内部的奥秘，以下源码是基于JDK1.7.0_79。

结构

类结构

LinkedList的类的结构如下图所示：

公众号：方志朋



通过上图可以看出，`LinkedList`继承的类与实现的接口如下：

- `1. Collection 接口`: Collection接口是所有集合类的根节点，Collection表示一种规则，所有实现了Collection接口的类遵循这种规则
- `2. List 接口`: List是Collection的子接口，它是一个元素有序(按照插入的顺序维护元素顺序)、可重复、可以为null的集合
- `3. AbstractCollection 类`: Collection接口的骨架实现类，最小化实现了Collection接口所需要实现的工作量
- `4. AbstractList 类`: List接口的骨架实现类，最小化实现了List接口所需要实现的工作量
- `5. Cloneable 接口`: 实现了该接口的类可以显示的调用Object.clone()方法，合法的对该类实例进行字段复制，如果没有实现Cloneable接口的实例上调用Object.clone()方法，会抛出CloneNotSupportedException异常。正常情况下，实现了Cloneable接口的类会以公共方法重写Object.clone()
- `6. Serializable 接口`: 实现了该接口标示了类可以被序列化和反序列化，具体的查询序列化详解
- `7. Deque 接口`: Deque定义了一个线性Collection，支持在两端插入和删除元素，Deque实际是“double ended queue(双端队列)”的简称，大多数Deque接口的实现都不会限制元素的数量，但是

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

这个队列既支持有容量限制的实现，也支持没有容量限制的实现，比如LinkedList就是有容量限制的实现，其最大的容量为Integer.MAX_VALUE

- 8. AbstractSequentialList 类：提供了List接口的骨干实现，最大限度地减少了实现受“连续访问”数据存储(如链表)支持的此接口所需的工作，对于随机访问数据(如数组)，应该优先使用AbstractList，而不是使用AbstractSequentialList类

基础属性及构造方法

基础属性

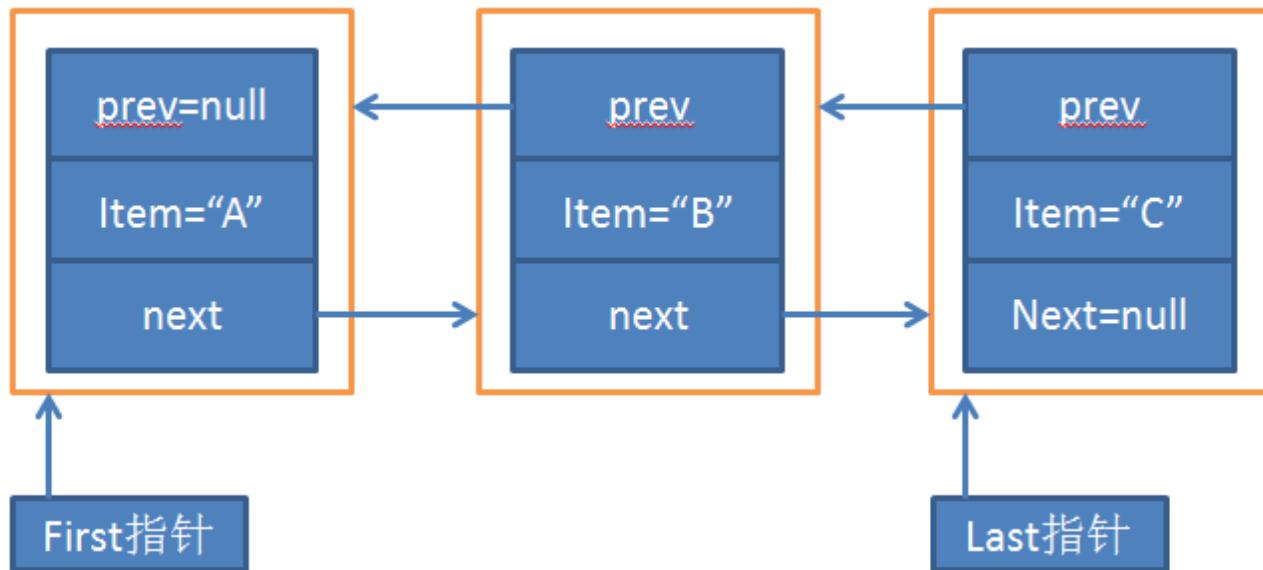
```
1 public class LinkedList<E>
2     extends AbstractSequentialList<E>
3     implements List<E>, Deque<E>, Cloneable, java.io.Serializable
4 {
5     //长度
6     transient int size = 0;
7     //指向头结点
8     transient Node<E> first;
9     //指向尾结点
10    transient Node<E> last;
11 }
```

如上源码中为LinkedList中的基本属性，其中size为LinkedList的长度，first为指向头结点，last指向尾结点，Node为LinkedList的一个私有内部类，其定义如下，即定义了item(元素)，next(指向后一个元素的指针)，prev(指向前一个元素的指针)

```
1 private static class Node<E> {
2     //元素
3     E item;
4     //指向后一个元素的指针
5     Node<E> next;
6     //指向前一个元素的指针
7     Node<E> prev;
8
9     Node(Node<E> prev, E element, Node<E> next) {
10         this.item = element;
```

```
11     this.next = next;
12     this.prev = prev;
13 }
14 }
```

那么假如LinkedList中的元素为["A","B","C"],其内部的结构如下图所示



可以看出一个节点中包含三个属性，也就是上面源码中定义的属性，可以清晰的看出LinkedList底层是双向链表的实现

构造方法

在源码中，LinkedList主要提供了两个构造方法，

- 1. `public LinkedList()` : 空的构造方法，啥事情都没有做
- 2. `public LinkedList(Collection<? extends E> c)` : 将一个元素集合添加到LinkedList中

底层实现

在2.2.1中的LinkedList内部结构图，可以清晰的看出LinkedList双向链表的实现，下面将通过源码分析如何在双向链表中添加和删除节点的。

添加节点

通常我们会使用add(E e)方法添加元素，通过源码我们发现add(E e)内部主要调用了以下方法

//在链表的最后添加元素

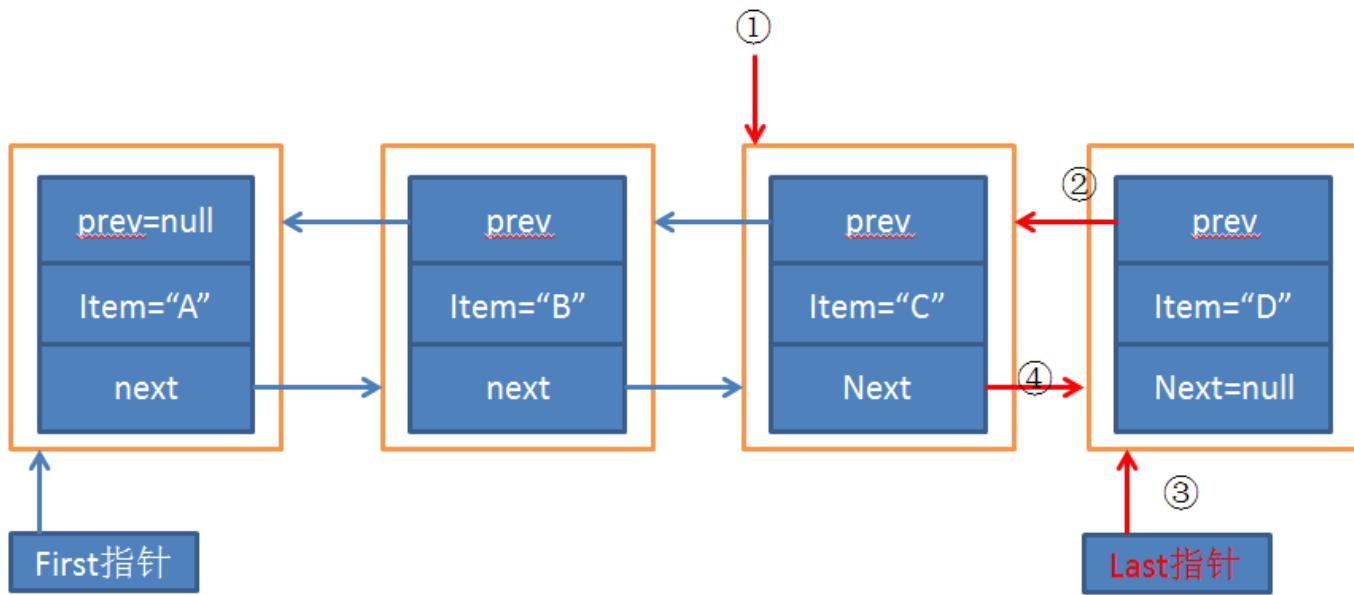
```
1 void linkLast(E e) {  
2     final Node<E> l = last;  
3     final Node<E> newNode = new Node<>(l, e, null);  
4     last = newNode;  
5     if (l == null)  
6         first = newNode;  
7     else  
8         l.next = newNode;  
9     size++;  
10    modCount++;  
11 }
```

其实通过源码可以看出添加的过程如下

- 1.记录当前末尾节点，通过构造另外一个指向末尾节点的指针l
- 2.产生新的节点：注意的是由于是添加在链表的末尾，next是为null的
- 3.last指向新的节点
- 4.这里有个判断，我的理解是判断是否为第一个元素(当l==null时，表示链表中是没有节点的)，那么就很好理解这个判断了，如果是第一节点，则使用first指向这个节点，若不是则当前节点的next指向新增的节点
- 5.size增加

例如，在上面提到的LinkedList["A","B","C"]中添加元素“D”，过程大致如图所示

公众号：方志朋



LinkedList中还提供如下的方法，进行添加元素，具体逻辑与linkLast方法大同小异，就不在这里一一介绍了。

删除节点

LinkedList中提供了两个方法删除节点，如下源码所示

```
1 //方法1.删除指定索引上的节点
2 public E remove(int index) {
3     //检查索引是否正确
4     checkElementIndex(index);
5     //这里分为两步，第一通过索引定位到节点，第二删除节点
6     return unlink(node(index));
7 }
8
9 //方法2.删除指定值的节点
10 public boolean remove(Object o) {
11     //判断删除的元素是否为null
12     if (o == null) {
13         //若是null遍历删除
14         for (Node<E> x = first; x != null; x = x.next) {
15             if (x.item == null) {
16                 unlink(x);
17                 return true;
18             }
19         }
20     }
21 }
```

公众号: 方志朋

```
18     }
19 }
20 } else {
21     //若不是遍历删除
22     for (Node<E> x = first; x != null; x = x.next) {
23         if (o.equals(x.item)) {
24             unlink(x);
25             return true;
26         }
27     }
28 }
29 return false;
30 }
```

通过源码可以看出两个方法都是通过unlink()删除，在方法一种有个方法要介绍下，就是node(index)该方法的作用就是根据下标找到对应的节点，要是本人去写这个方法肯定是遍历到index找到对应的节点，而JDK提供的方法如下所示

- 1.首先确定index的位置，是靠近first还是靠近last
- 2.若靠近first则从头开始查询，否则从尾部开始查询，可以看出这样避免极端情况的发生，也更好的利用了LinkedList双向链表的特征

```
1 Node<E> node(int index) {
2     // assert isElementIndex(index);
3     if (index < (size >> 1)) {
4         Node<E> x = first;
5         for (int i = 0; i < index; i++)
6             x = x.next;
7         return x;
8     } else {
9         Node<E> x = last;
10        for (int i = size - 1; i > index; i--)
11            x = x.prev;
12        return x;
13    }
14 }
```

公众号：方志朋

下面会详细介绍unlink()方法的源码，这是删除节点最核心的方法

```
1 E unlink(Node<E> x) {  
2     // assert x != null;  
3     final E element = x.item;  
4     final Node<E> next = x.next;  
5     final Node<E> prev = x.prev;  
6  
7     //删除的是第一个节点，first向后移动  
8     if (prev == null) {  
9         first = next;  
10    } else {  
11        prev.next = next;  
12        x.prev = null;  
13    }  
14  
15    //删除的是最后一个节点，last向前移  
16    if (next == null) {  
17        last = prev;  
18    } else {  
19        next.prev = prev;  
20        x.next = null;  
21    }  
22  
23    x.item = null;  
24    size--;  
25    modCount++;  
26    return element;  
27 }
```

- 1. 获取到需要删除元素当前的值，指向它前一个节点的引用，以及指向它后一个节点的引用。
- 2. 判断删除的是否为第一个节点，若是则first向后移动，若不是则将当前节点的前一个节点next指向当前节点的后一个节点
- 3. 判断删除的是否为最后一个节点，若是则last向前移动，若不是则将当前节点的后一个节点的prev指向当前节点的前一个节点
- 4. 将当前节点的值置为null
- 5. size减少并返回删除节点的值

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

至此介绍了LinkedList添加、删除元素的内部实现。

4.对比

在ArrayList详解中讲解了ArrayList的相关的内容，下面将对ArrayList与LinkedList进行对比，主要从以下方面进行

4.1 相同点

- 1.接口实现：都实现了List接口，都是线性列表的实现
- 2.线程安全：都是线程不安全的

4.2 区别

- 1.底层实现：ArrayList内部是数组实现，而LinkedList内部实现是双向链表结构
- 2.接口实现：ArrayList实现了RandomAccess可以支持随机元素访问，而LinkedList实现了Deque可以当做队列使用
- 3.性能：新增、删除元素时ArrayList需要使用到拷贝原数组，而LinkedList只需移动指针，查找元素ArrayList支持随机元素访问，而LinkedList只能一个节点的去遍历

4.3 性能比较

下面通过代码去比较下ArrayList与LinkedList在性能方面的差别，代码如下

```
1 public class ListPerformance {  
2  
3     private static ArrayList<String> arrayList= new ArrayList<Str  
ing>();  
4  
5     private static LinkedList<String> linkedList = new LinkedList  
<String>();  
6  
7     /**  
8      * 插入数据  
9      * @param list  
10     * @param count
```

```
11     */
12     public static void insertElements(List<String> list, int count) {
13         Long startTime = System.currentTimeMillis();
14         for (int i = 0; i < count; i++) {
15             list.add(String.valueOf(i));
16         }
17         Long endTime = System.currentTimeMillis();
18         System.out.println("insert elements use time: " +(endTime - startTime) + " ms");
19     }
20
21     /**
22      * 删除元素
23      * @param list
24      * @param count
25      */
26     public static void removeElements(List<String> list, int count) {
27         Long startTime = System.currentTimeMillis();
28         for (int i = 0; i < count; i++) {
29             list.remove(0);
30         }
31         Long endTime = System.currentTimeMillis();
32         System.out.println("remove elements use time: " +(endTime - startTime) + " ms");
33     }
34
35     /**
36      * 获取元素
37      * @param list
38      * @param count
39      */
40     public static void getElements(List<String> list, int count){
41         Long startTime = System.currentTimeMillis();
42         for (int i = 0; i < count; i++) {
43             list.get(i);
44         }
45         Long endTime = System.currentTimeMillis();
46         System.out.println("get elements use time: " +(endTime - st
```

```
        artTime) + " ms");

47    }
48    /**
49     * 删除元素第二种实现
50     * @param list
51     * @param count
52     */
53    public static void removeElements2(List<String> list, int count){
54        Long startTime = System.currentTimeMillis();
55        for (int i = count-1; i > 0; i--) {
56            list.remove(i);
57        }
58        Long endTime = System.currentTimeMillis();
59        System.out.println("remove elements use time: " +(endTime - startTime) + " ms");
60    }
61    public static void main(String[] args){
62        System.out.println("arrayList test");
63        insertElements(arrayList,100000);
64        getElements(arrayList,100000);
65        removeElements(arrayList,100000);

66
67        System.out.println("linkedList test");
68        insertElements(linkedList,100000);
69        getElements(linkedList,100000);
70        removeElements(linkedList,100000);

71
72
73        System.out.println("arrayList test2");
74        insertElements(arrayList,100000);
75        getElements(arrayList,100000);
76        removeElements2(arrayList,100000);

77
78        System.out.println("linkedList test2");
79        insertElements(linkedList,100000);
80        getElements(linkedList,100000);
81        removeElements2(linkedList,100000);
82    }
```

结果如下图所示，可以看出

~

- 1.LinkedList下插入、删除是性能优于ArrayList，这是由于插入、删除元素时ArrayList中需要额外的开销去移动、拷贝元素(但是使用removeElements2方法所示去遍历删除是速度异常的快，这种方式的做法是从末尾开始删除，不存在移动、拷贝元素，从而速度非常快)
- 2.ArrayList在查询元素的性能上要由于LinkedList

自己动手写个LinkedList

自己动手实现一个这样的LinkedList：

```
1 public class MyLinkList<T> implements Iterable<T> {  
2  
3     private int size;  
4     private int modeCount = 0;  
5     private Node<T> beginMarker;  
6     private Node<T> endMarker;  
7  
8     public MyLinkList() {  
9         doClear();  
10    }  
11  
12    private void doClear() {  
13        size = 0;  
14        modeCount++;  
15        beginMarker = new Node<T>(null, null, null);  
16        endMarker = new Node<T>(null, beginMarker, null);  
17        beginMarker.next = endMarker;  
18    }  
19  
20    public boolean add(T data) {  
21        Node<T> node = new Node<>(data, endMarker.prev, endMarker)  
22        ;  
23        endMarker.prev.next = node;  
24        endMarker.prev = node;
```

公众号：方志朋

```
24         modeCount++;
25         size++;
26         return true;
27     }
28
29     public T remove(int index) {
30         Node<T> node = getNode(index);
31         node.prev.next = node.next;
32         node.next.prev = node.prev;
33         size--;
34         modeCount--;
35         return node.data;
36     }
37
38     public Node<T> getNode(int index) {
39         Node<T> p;
40         if (index < 0 || index >= size) {
41             throw new IndexOutOfBoundsException("out of index");
42         }
43         if (index < size / 2) {//从左边开始找
44             p = beginMarker.next;
45             for (int i = 0; i < index; i++) {
46                 p = p.next;
47             }
48
49         } else {//从右边开始找
50             p = endMarker;
51             for (int i = size; i > index; i--) {
52                 p = p.prev;
53             }
54
55         }
56         return p;
57     }
58
59     public T get(int index) {
60
61         return getNode(index).data;
62     }
63 }
```

```
64
65
66     @Override
67     public Iterator<T> iterator() {
68         return null;
69     }
70
71     @Override
72     public void forEach(Consumer<? super T> action) {
73
74     }
75
76     @Override
77     public Spliterator<T> spliterator() {
78         return null;
79     }
80
81     static class Node<T> {
82         private T data;
83         private Node<T> prev;
84         private Node<T> next;
85
86         public Node(T data, Node<T> prev, Node<T> next) {
87             this.data = data;
88             this.prev = prev;
89             this.next = next;
90         }
91
92     }
93 }
94 }
```

公众号：方志朋

参考文章

<https://www.jianshu.com/p/732b5294a985>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

Java基础： JAVA Hashmap的死循环及Java8的修复

在淘宝内网里看到同事发了贴说了一个CPU被100%的线上故障，并且这个事发生了很多次，原因是在Java语言在并发情况下使用HashMap造成Race Condition，从而导致死循环。

这个事情我4、5年前也经历过，本来觉得没什么好写的，因为Java的HashMap是非线程安全的，所以在并发下必然出现问题。但是，我发现近几年，很多人都经历过这个事（在网上查“HashMap Infinite Loop”可以看到很多人都在说这个事）所以，觉得这个是个普遍问题，需要写篇疫苗文章说一下这个事，并且给大家看看一个完美的“Race Condition”是怎么形成的。

问题的症状

从前我们的Java代码因为一些原因使用了HashMap这个东西，但是当时的程序是单线程的，一切都没有问题。后来，我们的程序性能有问题，所以需要变成多线程的，于是，变成多线程后到了线上，发现程序经常占了100%的CPU，查看堆栈，你会发现程序都Hang在了HashMap.get()这个方法上了，重启程序后问题消失。但是过段时间又会来。而且，这个问题在测试环境里可能很难重现。

我们简单的看一下我们自己的代码，我们就知道HashMap被多个线程操作。而Java的文档说HashMap是非线程安全的，应该用ConcurrentHashMap。

但是在这里我们可以研究一下原因。

Hash表数据结构

我需要简单地说一下HashMap这个经典的数据结构。

HashMap通常会用一个指针数组（假设为table[]）来做分散所有的key，当一个key被加入时，会通过Hash算法通过key算出这个数组的下标i，然后把这个<key, value>插到table[i]中，如果有两个不同的key被算在了同一个i，那么就叫冲突，又叫碰撞，这样会在table[i]上形成一个链表。

我们知道，如果table[]的尺寸很小，比如只有2个，如果要放进10个keys的话，那么碰撞非常频繁，于是一个O(1)的查找算法，就变成了链表遍历，性能变成了O(n)，这是Hash表的缺陷（可参看《Hash Collision DoS 问题》）。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

所以，Hash表的尺寸和容量非常的重要。一般来说，Hash表这个容器当有数据要插入时，都会检查容量有没有超过设定的threshold，如果超过，需要增大Hash表的尺寸，但是这样一来，整个Hash表里的元素都需要被重算一遍。这叫rehash，这个成本相当的大。

相信大家对这个基础知识已经很熟悉了。

HashMap的rehash源代码

下面，我们来看一下Java的HashMap的源代码。

Put一个Key,Value对到Hash表中：

```
1
2 public V put(K key, V value)
3 {
4     .....
5     //算Hash值
6     int hash = hash(key.hashCode());
7     int i = indexFor(hash, table.length);
8     //如果该key已被插入，则替换掉旧的value（链接操作）
9     for (Entry<K,V> e = table[i]; e != null; e = e.next) {
10         Object k;
11         if (e.hash == hash && ((k = e.key) == key || key.equals(k
12 )))) {
13             V oldValue = e.value;
14             e.value = value;
15             e.recordAccess(this);
16             return oldValue;
17         }
18     modCount++;
19     //该key不存在，需要增加一个结点
20     addEntry(hash, key, value, i);
21     return null;
22 }
```

检查容量是否超标：

```
1
2 void addEntry(int hash, K key, V value, int bucketIndex)
3 {
4     Entry<K,V> e = table[bucketIndex];
5     table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
6     //查看当前的size是否超过了我们设定的阈值threshold, 如果超过, 需要resize
7     if (size++ >= threshold)
8         resize(2 * table.length);
9 }
```

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

```
1 void resize(int newCapacity)
2 {
3     Entry[] oldTable = table;
4     int oldCapacity = oldTable.length;
5     .....
6     //创建一个新的Hash Table
7     Entry[] newTable = new Entry[newCapacity];
8     //将Old Hash Table上的数据迁移到New Hash Table上
9     transfer(newTable);
10    table = newTable;
11    threshold = (int)(newCapacity * loadFactor);
12 }
```

迁移的源代码，注意高亮处：

```
1
2 void transfer(Entry[] newTable)
3 {
4     Entry[] src = table;
5     int newCapacity = newTable.length;
6     //下面这段代码的意思是：
7     // 从OldTable里摘一个元素出来，然后放到NewTable中
8     for (int j = 0; j < src.length; j++) {
```

```
9      Entry<K,V> e = src[j];
10     if (e != null) {
11         src[j] = null;
12         do {
13             Entry<K,V> next = e.next;
14             int i = indexFor(e.hash, newCapacity);
15             e.next = newTable[i];
16             newTable[i] = e;
17             e = next;
18         } while (e != null);
19     }
20 }
21 }
```

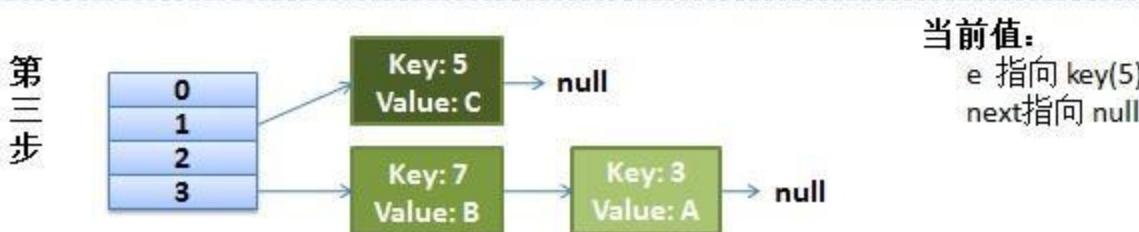
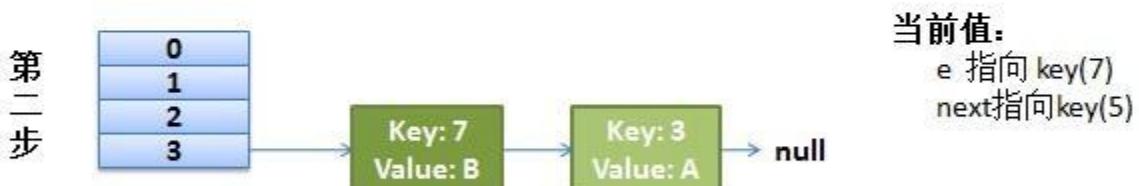
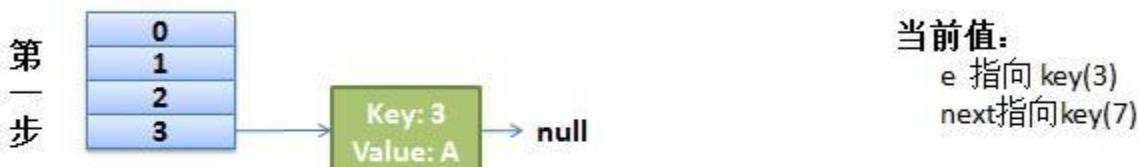
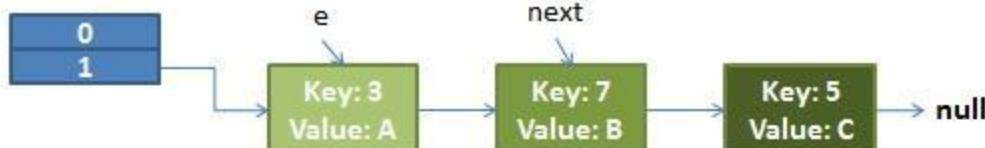
好了，这个代码算是比较正常的。而且没有什么问题。

正常的ReHash的过程

画了个图做了个演示。

- 我假设了我们的hash算法就是简单的用key mod 一下表的大小（也就是数组的长度）。
- 最上面的是old hash 表，其中的Hash表的size=2，所以key = 3, 7, 5，在mod 2以后都冲突在table[1]这里了。
- 接下来的三个步骤是Hash表 resize成4，然后所有的<key,value>

公众号：方志朋



并发下的Rehash

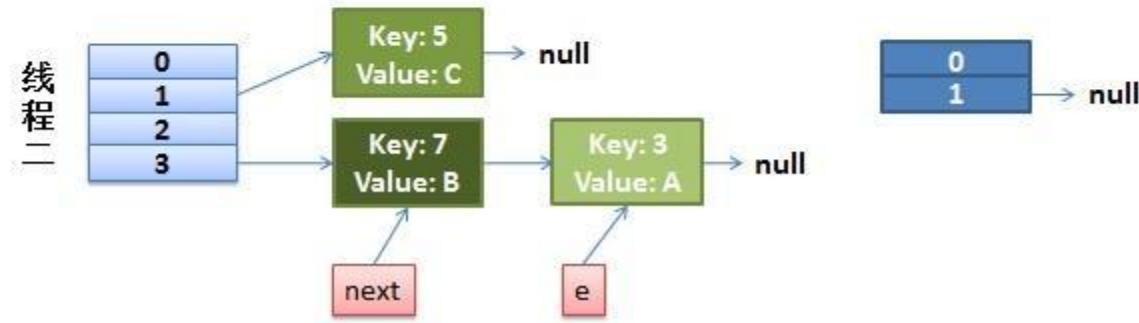
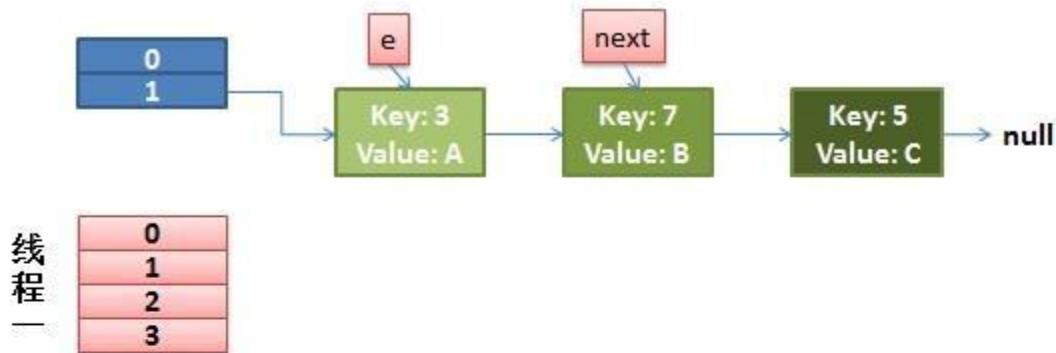
1) 假设我们有两个线程。我用红色和浅蓝色标注了一下。

我们再回头看一下我们的 transfer 代码中的这个细节：

```
1 do {  
2     Entry<K,V> next = e.next; // <--假设线程一执行到这里就被调度挂起了  
3     int i = indexFor(e.hash, newCapacity);  
4     e.next = newTable[i];  
5     newTable[i] = e;  
6     e = next;  
7 } while (e != null);
```

而我们的线程二执行完成了。于是我们有下面的样子。

公众号：方志朋

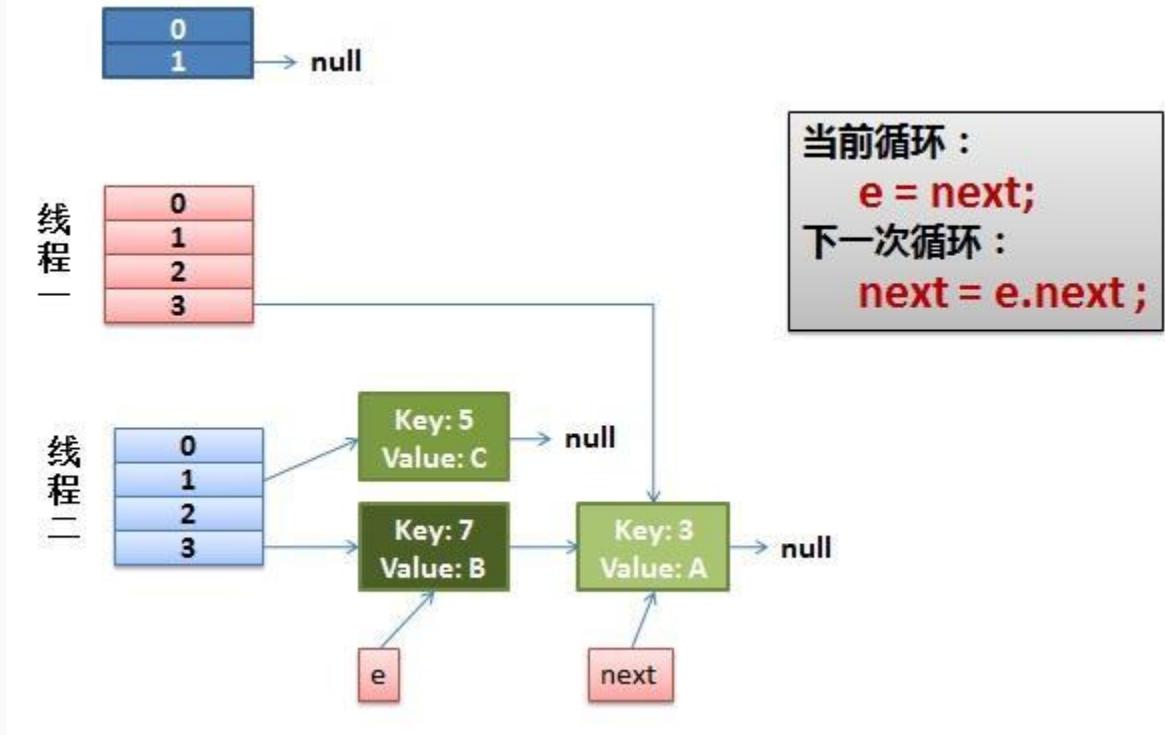


注意，因为Thread1的 e 指向了key(3)，而next指向了key(7)，其在线程二rehash后，指向了线程二重组后的链表。我们可以看到链表的顺序被反转后。

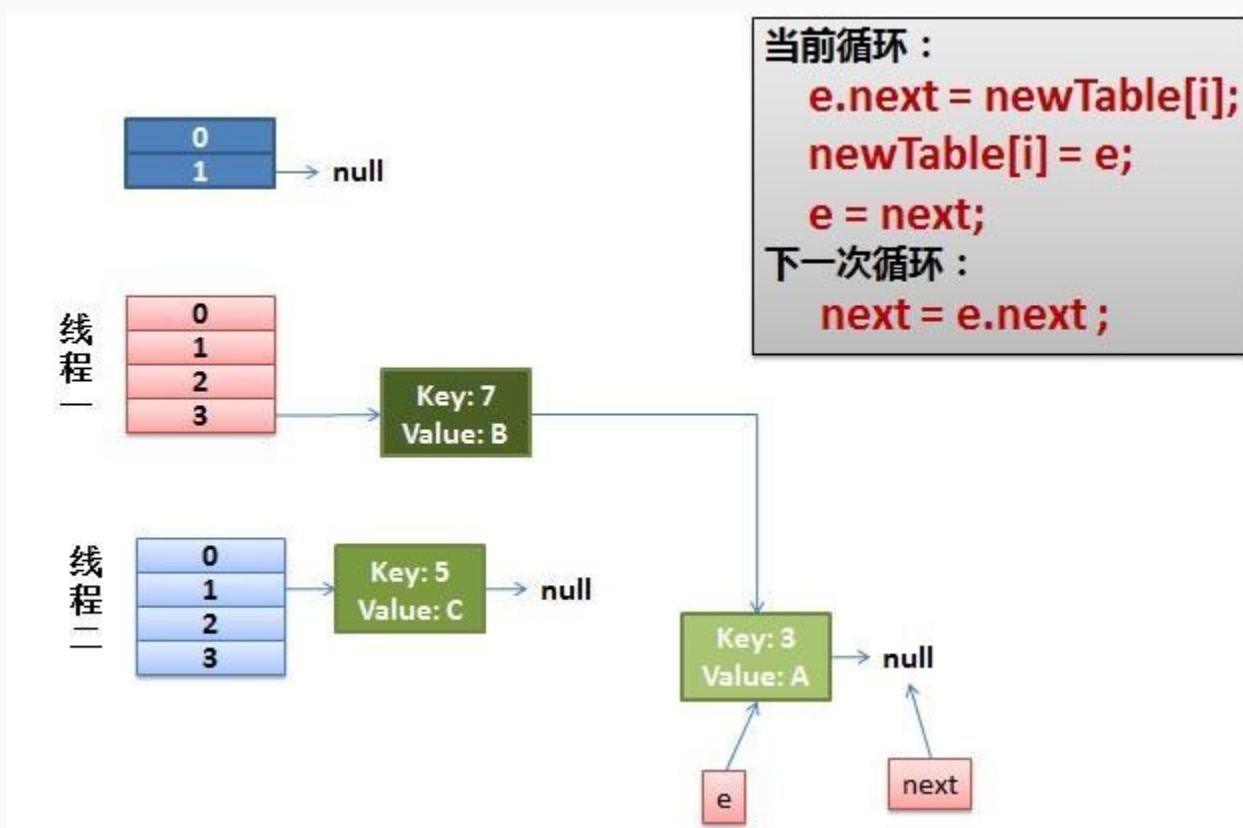
2) 线程一被调度回来执行。

- 先是执行 `newTable[i] = e;`
- 然后是 `e = next`, 导致了 `e` 指向了 `key(7)`,
- 而下一次循环的 `next = e.next` 导致了 `next` 指向了 `key(3)`

公众号：方志朋



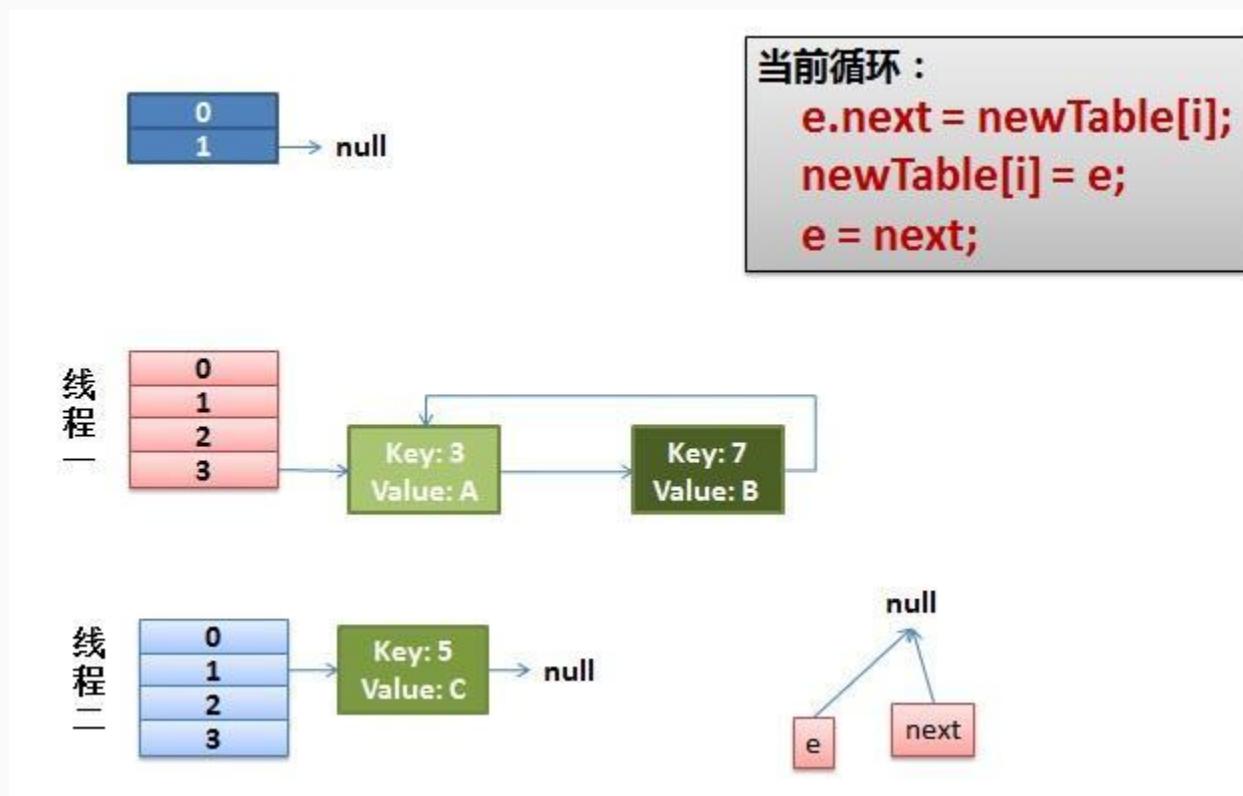
3) 一切安好。



4) 环形链接出现。

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

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



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

Java8在死循环的修复

相较于JDK1.7，在1.8中resize()方法不再调用transfer()方法，而是直接将原来transfer()方法中的代码写在自己方法体内；当然表面上我们能看到方法的减少，其实还有一个重大改变，那就是：**扩容后，新数组中的链表顺序依然与旧数组中的链表顺序保持一致！**

数组长度的技巧

在分析源码之前，我们还需要知道HashMap底层数组的一些优化： 数组长度总是2的倍数，扩容则是直接在原有数组长度基础上乘以2。

为什么要这么做？有两个优点：

- 通过与元素的hash值进行与操作，能够快速定位到数组下标 相对于取模运算，直接进行与操作能提高计算效率。在CPU中，所有的加减乘除都是通过加法实现的，而与操作时CPU直接支持的。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 扩容时简化计算数组下标的计算量 因为数组每次扩容都是原来的两倍，所以每一个元素在新数组中的位置要么是原来的index，要么 $index = index + oldCap$ ，假设

数组长度：2，key: 3 0 0 0 1 0 0 1 1 结果为 $0 \oplus 0 \oplus 1 = 1$ 所以数组下标为1；

扩容后，数组长度：4，key: 3 0 0 1 1 0 0 1 1 结果为 $0 \oplus 0 \oplus 1 \oplus 1 = 3 =$ 原来的 $index + oldCap = 1 + 2$

即确定元素在新数组的下标时，我们只需要检测元素的hash值与oldCap作与操作的结果是否为0：为0，那么下标还是原来的下标；为1，那么下标等于原来下标加上旧数组长度。

我们来看关键代码（resize()方法完整代码附后）：

```
1 //如果扩容后，元素的index依然与原来一样，那么使用这个head和tail指针
2 Node<K,V> loHead = null, loTail = null
3 //如果扩容后，元素的index=index+oldCap，那么使用这个head和tail指针
4 Node<K,V> hiHead = null, hiTail = null
5 Node<K,V> next;
6 do {
7     next = e.next;
8     //这个地方直接通过hash值与oldCap进行与操作得出元素在新数组的index
9     if ((e.hash & oldCap) == 0) {
10         if (loTail == null)
11             loHead = e;
12         else
13             loTail.next = e;
14         //tail指针往后移动一位，维持顺序
15         loTail = e;
16     }
17     else {
18         if (hiTail == null)
19             hiHead = e;
20         else
21             hiTail.next = e;
22         //tail指针往后移动一位，维持顺序
23         hiTail = e;
24     }
25 } while ((e = next) != null);
26 if (loTail != null) {
27     loTail.next = null;
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
28     //还是原来的index
29     newTab[j] = loHead;
30 }
31 if (hiTail != null) {
32     hiTail.next = null;
33     //index = index + oldCap
34     newTab[j + oldCap] = hiHead;
35 }
```

总结

Java8虽然修复了死循环的BUG，但是HashMap 还是非线程安全类，仍然会产生数据丢失等问题。

参考资料

<https://my.oschina.net/alexqdyj/blog/1377268>

<https://coolshell.cn/articles/9606.html>

<https://my.oschina.net/liuxiaomian/blog/848996>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



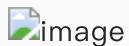
还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

公众号：方志朋

Java基础： JAVA中BitSet使用详解

JAVA中BitSet使用详解

适用场景：整数，无重复；



Bitset 基础

Bitset，也就是位图，由于可以用非常紧凑的格式来表示给定范围的连续数据而经常出现在各种算法设计中。上面的图来自c++库中bitset的一张图。

基本原理是，用1位来表示一个数据是否出现过，0为没有出现过，1表示出现过。使用的时候既可根据某一个是否为0表示此数是否出现过。

一个1G的空间，有 $8102410241024=8.5810^9$ bit，也就是可以表示85亿个不同的数。

常见的应用是那些需要对海量数据进行一些统计工作的时候，比如日志分析等。

面试题中也常出现，比如：统计40亿个数据中没有出现的数据，将40亿个不同数据进行排序等。

又如：现在有1千万个随机数，随机数的范围在1到1亿之间。现在要求写出一种算法，将1到1亿之间没有在随机数中的数求出来(百度)。

programming pearls上也有一个关于使用bitset来查找电话号码的题目。

Bitmap的常见扩展，是用2位或者更多位来表示此数字的更多信息，比如出现了多少次等。

Java中Bitset的实现

Bitset这种结构虽然简单，实现的时候也有一些细节需要主要。其中的关键是一些位操作，比如如何将指定位进行反转、设置、查询指定位的状态（0或者1）等。

本文，分析一下java中bitset的实现，抛砖引玉，希望给那些需要自己设计位图结构的需要的程序员有所启发。

公众号：方志朋

Bitmap的基本操作有：

- 初始化一个bitset，指定大小。
- 清空bitset。
- 反转某一指定位。
- 设置某一指定位。
- 获取某一位的状态。
- 当前bitset的bit总位数。

使用场景

常见的应用是那些需要对海量数据进行一些统计工作的时候，比如日志分析、用户数统计等等

如统计40亿个数据中没有出现的数据，将40亿个不同数据进行排序等。

现在有1千万个随机数，随机数的范围在1到1亿之间。现在要求写出一种算法，将1到1亿之间没有在随机数中的数求出来

代码示例

```
1 package util;
2
3 import java.util.Arrays;
4 import java.util.BitSet;
5
6 public class BitSetDemo {
7
8     /**
9      * 求一个字符串包含的char
10     *
11     */
12     public static void containChars(String str) {
13         BitSet used = new BitSet();
14         for (int i = 0; i < str.length(); i++)
15             used.set(str.charAt(i)); // set bit for char
16
17         StringBuilder sb = new StringBuilder();
18
19         for (int i = 0; i < 10000000; i++)
20             if (!used.get(i))
21                 sb.append((char) i);
22
23         System.out.println(sb.toString());
24     }
25 }
```

公众号：方志朋

```
18     sb.append("[");
19     int size = used.size();
20     System.out.println(size);
21     for (int i = 0; i < size; i++) {
22         if (used.get(i)) {
23             sb.append((char) i);
24         }
25     }
26     sb.append("]");
27     System.out.println(sb.toString());
28 }
29
30 /**
31 * 求素数 有无限个。一个大于1的自然数，如果除了1和它本身外，不能被其他自
32 * 然数整除(除0以外) 的数称之为素数(质数) 否则称为合数
33 */
34 public static void computePrime() {
35     BitSet sieve = new BitSet(1024);
36     int size = sieve.size();
37     for (int i = 2; i < size; i++)
38         sieve.set(i);
39     int finalBit = (int) Math.sqrt(sieve.size());
40
41     for (int i = 2; i < finalBit; i++)
42         if (sieve.get(i))
43             for (int j = 2 * i; j < size; j += i)
44                 sieve.clear(j);
45
46     int counter = 0;
47     for (int i = 1; i < size; i++) {
48         if (sieve.get(i)) {
49             System.out.printf("%5d", i);
50             if (++counter % 15 == 0)
51                 System.out.println();
52         }
53     }
54 }
55
56 /**
```

```
57     * 进行数字排序
58     */
59     public static void sortArray() {
60         int[] array = new int[] { 423, 700, 9999, 2323, 356, 640
61         0, 1, 2, 3, 2, 2, 2, 2 };
62         BitSet bitSet = new BitSet(2 << 13);
63         // 虽然可以自动扩容，但尽量在构造时指定估算大小，默认为64
64         System.out.println("BitSet size: " + bitSet.size());
65
66         for (int i = 0; i < array.length; i++) {
67             bitSet.set(array[i]);
68         }
69         //剔除重复数字后的元素个数
70         int bitLen=bitSet.cardinality();
71
72         //进行排序，即把bit为true的元素复制到另一个数组
73         int[] orderedArray = new int[bitLen];
74         int k = 0;
75         for (int i = bitSet.nextSetBit(0); i >= 0; i = bitSet.ne
76         xtSetBit(i + 1)) {
77             orderedArray[k++] = i;
78
79         System.out.println("After ordering: ");
80         for (int i = 0; i < bitLen; i++) {
81             System.out.print(orderedArray[i] + "\t");
82         }
83
84         System.out.println("iterate over the true bits in a Bits
85         et");
86         //或直接迭代BitSet中bit为true的元素iterate over the true bit
87         s in a BitSet
88         for (int i = bitSet.nextSetBit(0); i >= 0; i = bitSet.ne
89         xtSetBit(i + 1)) {
90             System.out.print(i+"\t");
91         }
92     }  
/*
```

```
92     * 将BitSet对象转化为ByteArray
93     * @param bitSet
94     * @return
95     */
96     public static byte[] bitSet2ByteArray(BitSet bitSet) {
97         byte[] bytes = new byte[bitSet.size() / 8];
98         for (int i = 0; i < bitSet.size(); i++) {
99             int index = i / 8;
100            int offset = 7 - i % 8;
101            bytes[index] |= (bitSet.get(i) ? 1 : 0) << offset;
102        }
103        return bytes;
104    }
105
106    /**
107     * 将ByteArray对象转化为BitSet
108     * @param bytes
109     * @return
110     */
111    public static BitSet byteArray2BitSet(byte[] bytes) {
112        BitSet bitSet = new BitSet(bytes.length * 8);
113        int index = 0;
114        for (int i = 0; i < bytes.length; i++) {
115            for (int j = 7; j >= 0; j--) {
116                bitSet.set(index++, (bytes[i] & (1 << j)) >> j =
117 = 1 ? true
118                               : false);
119            }
120        }
121    }
122
123    /**
124     * 简单使用示例
125     */
126    public static void simpleExample() {
127        String names[] = { "Java", "Source", "and", "Support" };
128        BitSet bits = new BitSet();
129        for (int i = 0, n = names.length; i < n; i++) {
130            if ((names[i].length() % 2) == 0) {
```

```
131             bits.set(i);
132         }
133     }
134
135     System.out.println(bits);
136     System.out.println("Size : " + bits.size());
137     System.out.println("Length: " + bits.length());
138     for (int i = 0, n = names.length; i < n; i++) {
139         if (!bits.get(i)) {
140             System.out.println(names[i] + " is odd");
141         }
142     }
143     BitSet bites = new BitSet();
144     bites.set(0);
145     bites.set(1);
146     bites.set(2);
147     bites.set(3);
148     bites.andNot(bits);
149     System.out.println(bites);
150 }
151
152 public static void main(String args[]) {
153     //BitSet使用示例
154     BitSetDemo.containChars("How do you do? 你好呀");
155     BitSetDemo.computePrime();
156     BitSetDemo.sortArray();
157     BitSetDemo.simpleExample();
158
159
160     //BitSet与Byte数组互转示例
161     BitSet bitSet = new BitSet();
162     bitSet.set(3, true);
163     bitSet.set(98, true);
164     System.out.println(bitSet.size() + ", " + bitSet.cardinality());
165
166     //将BitSet对象转成byte数组
167     byte[] bytes = BitSetDemo.bitSet2ByteArray(bitSet);
168     System.out.println(Arrays.toString(bytes));
169
//在将byte数组转回来
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
170     bitSet = BitSetDemo.byteArray2BitSet(bytes);
171     System.out.println(bitSet.size()+"+"+bitSet.cardinality());
172     System.out.println(bitSet.get(3));
173     System.out.println(bitSet.get(98));
174     for (int i = bitSet.nextSetBit(0); i >= 0; i = bitSet.ne
175         xtSetBit(i + 1)) {
176         System.out.print(i+"\t");
177     }
178 }
```

原文链接

<https://blog.csdn.net/jiangnan2014/article/details/53735429>



程序员理财，请关注：



公众号：方志朋

Java基础：java中HashSet详解

java中HashSet详解

HashSet 的实现

对于 HashSet 而言，它是基于 HashMap 实现的，HashSet 底层采用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，查看 HashSet 的源代码，可以看到如下代码：

```
1 public class HashSet<E>
2     extends AbstractSet<E>
3     implements Set<E>, Cloneable, java.io.Serializable
4 {
5     // 使用 HashMap 的 key 保存 HashSet 中所有元素
6     private transient HashMap<E, Object> map;
7     // 定义一个虚拟的 Object 对象作为 HashMap 的 value
8     private static final Object PRESENT = new Object();
9     ...
10    // 初始化 HashSet，底层会初始化一个 HashMap
11    public HashSet()
12    {
13        map = new HashMap<E, Object>();
14    }
15    // 以指定的 initialCapacity、loadFactor 创建 HashSet
16    // 其实就是以相应的参数创建 HashMap
17    public HashSet(int initialCapacity, float loadFactor)
18    {
19        map = new HashMap<E, Object>(initialCapacity, loadFactor);
20    }
21    public HashSet(int initialCapacity)
22    {
23        map = new HashMap<E, Object>(initialCapacity);
24    }
25    HashSet(int initialCapacity, float loadFactor, boolean dummy)
26    {
27        map = new LinkedHashMap<E, Object>(initialCapacity
28                , loadFactor);
```

公众号：方志朋

```
29 }
30 // 调用 map 的 keySet 来返回所有的 key
31 public Iterator<E> iterator()
32 {
33     return map.keySet().iterator();
34 }
35 // 调用 HashMap 的 size() 方法返回 Entry 的数量，就得到该 Set 里元素的个数
36 public int size()
37 {
38     return map.size();
39 }
40 // 调用 HashMap 的 isEmpty() 判断该 HashSet 是否为空,
41 // 当 HashMap 为空时，对应的 HashSet 也为空
42 public boolean isEmpty()
43 {
44     return map.isEmpty();
45 }
46 // 调用 HashMap 的 containsKey 判断是否包含指定 key
47 // HashSet 的所有元素就是通过 HashMap 的 key 来保存的
48 public boolean contains(Object o)
49 {
50     return map.containsKey(o);
51 }
52 // 将指定元素放入 HashSet 中，也就是将该元素作为 key 放入 HashMap
53 public boolean add(E e)
54 {
55     return map.put(e, PRESENT) == null;
56 }
57 // 调用 HashMap 的 remove 方法删除指定 Entry，也就删除了 HashSet 中对应的元素
58 public boolean remove(Object o)
59 {
60     return map.remove(o)==PRESENT;
61 }
62 // 调用 Map 的 clear 方法清空所有 Entry，也就清空了 HashSet 中所有元素
63 public void clear()
64 {
65     map.clear();
66 }
```

由上面源程序可以看出，`HashSet` 的实现其实非常简单，它只是封装了一个`HashMap` 对象来存储所有的集合元素，所有放入`HashSet` 中的集合元素实际上由`HashMap` 的 key 来保存，而`HashMap` 的 value 则存储了一个`PRESENT`，它是一个静态的`Object` 对象。

`HashSet` 的绝大部分方法都是通过调用`HashMap` 的方法来实现的，因此`HashSet` 和`HashMap` 两个集合在实现本质上是相同的。

掌握上面理论知识之后，接下来看一个示例程序，测试一下自己是否真正掌握了`HashMap` 和`HashSet` 集合的功能。

```

1 class Name
2 {
3     private String first;
4     private String last;
5
6     public Name(String first, String last)
7     {
8         this.first = first;
9         this.last = last;
10    }
11
12    public boolean equals(Object o)
13    {
14        if (this == o)
15        {
16            return true;
17        }
18
19        if (o.getClass() == Name.class)
20        {
21            Name n = (Name)o;
22            return n.first.equals(first)
23                  && n.last.equals(last);
24        }
25        return false;
26    }
27 }
```

公众号：方志朋

```
28
29 public class HashSetTest
30 {
31     public static void main(String[] args)
32     {
33         Set<Name> s = new HashSet<Name>();
34         s.add(new Name("abc", "123"));
35         System.out.println(
36             s.contains(new Name("abc", "123")));
37     }
38 }
```

上面程序中向 HashSet 里添加了一个 new Name("abc", "123") 对象之后，立即通过程序判断该 HashSet 是否包含一个 new Name("abc", "123") 对象。粗看上去，很容易以为该程序会输出 true。

实际运行上面程序将看到程序输出 false，这是因为 HashSet 判断两个对象相等的标准除了要求通过 equals() 方法比较返回 true 之外，还要求两个对象的 hashCode() 返回值相等。因为在判断key是否存在 Hashmap 中，首先判断的是 hashCode，而上面程序没有重写 Name 类的 hashCode() 方法，两个 Name 对象的 hashCode() 返回值并不相同，因此 HashSet 会把它们当成 2 个对象处理，因此程序返回 false。

由此可见，当我们试图把某个类的对象当成 HashMap 的 key，或试图将这个类的对象放入 HashSet 中保存时，重写该类的 equals(Object obj) 方法和 hashCode() 方法很重要，而且这两个方法的返回值必须保持一致：当该类的两个的 hashCode() 返回值相同时，它们通过 equals() 方法比较也应该返回 true。通常来说，所有参与计算 hashCode() 返回值的关键属性，都应该用于作为 equals() 比较的标准。如下程序就正确重写了 Name 类的 hashCode() 和 equals() 方法，程序如下：

```
1
2 class Name
3 {
4     private String first;
5     private String last;
6     public Name(String first, String last)
7     {
8         this.first = first;
9         this.last = last;
10    }
11    // 根据 first 判断两个 Name 是否相等
```

java架构师公众号：方志朋

```
12     public boolean equals(Object o)
13     {
14         if (this == o)
15         {
16             return true;
17         }
18         if (o.getClass() == Name.class)
19         {
20             Name n = (Name)o;
21             return n.first.equals(first);
22         }
23         return false;
24     }
25
26     // 根据 first 计算 Name 对象的 hashCode() 返回值
27     public int hashCode()
28     {
29         return first.hashCode();
30     }
31
32     public String toString()
33     {
34         return "Name[first=" + first + ", last=" + last + "]";
35     }
36 }
37
38 public class HashSetTest2
39 {
40     public static void main(String[] args)
41     {
42         HashSet<Name> set = new HashSet<Name>();
43         set.add(new Name("abc" , "123"));
44         set.add(new Name("abc" , "456"));
45         System.out.println(set);
46     }
47 }
```

公众号：方志朋

上面程序中提供了一个 Name 类，该 Name 类重写了 equals() 和 toString() 两个方法，这两个方法都是根据 Name 类的 first 实例变量来判断的，当两个 Name 对象的 first 实例变量相等时，这两个 Name

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

对象的 hashCode() 返回值也相同，通过 equals() 比较也会返回 true。

程序主方法先将第一个 Name 对象添加到 HashSet 中，该 Name 对象的 first 实例变量值为"abc"，接着程序再次试图将一个 first 为"abc"的 Name 对象添加到 HashSet 中，很明显，此时没法将新的 Name 对象添加到该 HashSet 中，因为此处试图添加的 Name 对象的 first 也是" abc"， HashSet 会判断此处新增的 Name 对象与原有的 Name 对象相同，因此无法添加进入，程序在①号代码处输出 set 集合时将看到该集合里只包含一个 Name 对象，就是第一个、last 为"123"的 Name 对象。

原文链接

<https://alex09.iteye.com/blog/539549>

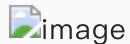
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：

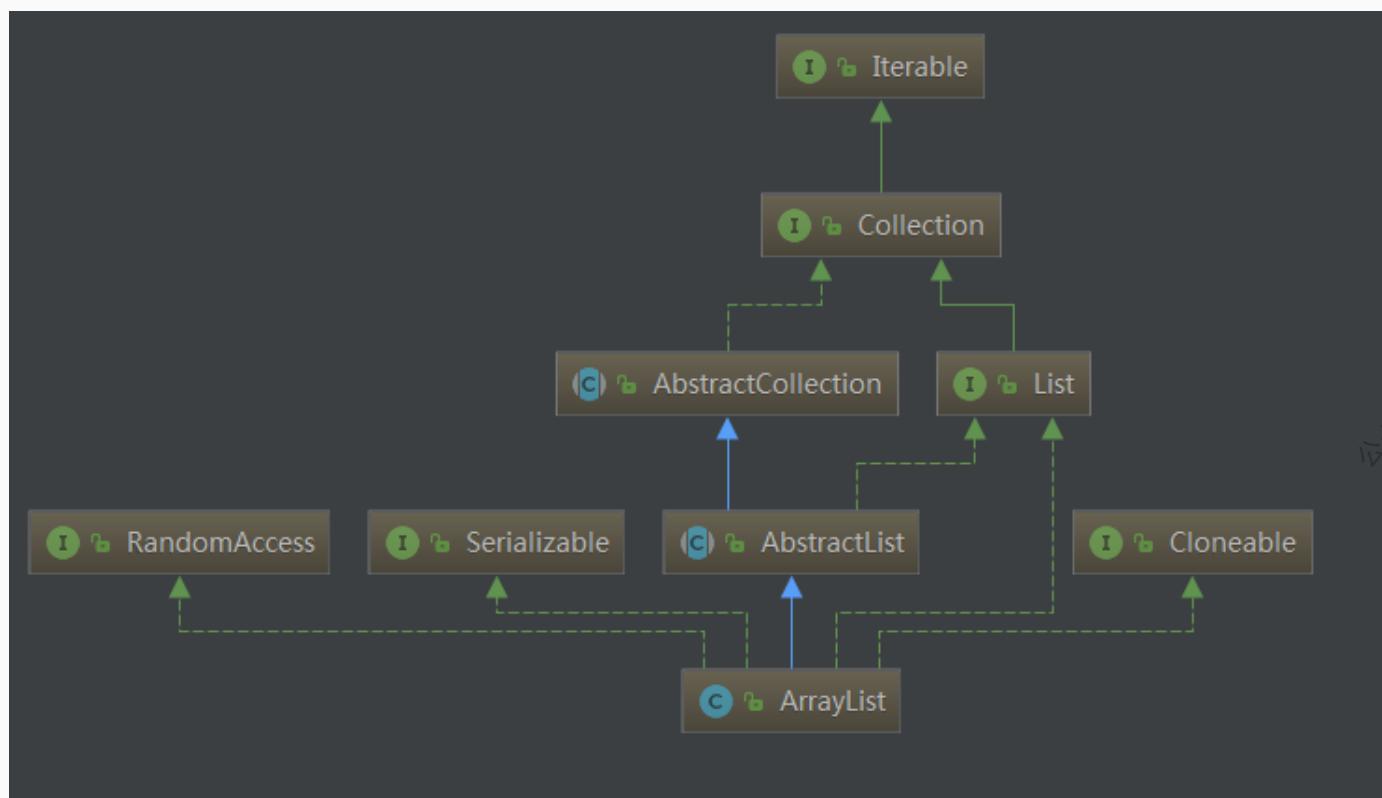


Java基础：Java容器之ArrayList

Java容器之ArrayList

ArrayList结构图

ArrayList 是 java 集合框架中比较常用的数据结构了。继承自 AbstractList，实现了 List 接口。底层基于数组实现容量大小动态变化。允许 null 的存在。同时还实现了 RandomAccess、Cloneable、Serializable 接口，所以ArrayList 是支持快速访问、复制、序列化的。



ArrayList类简介

- 1、ArrayList是内部是以动态数组的形式来存储数据的、知道数组的可能会疑惑：数组不是定长的吗？这里的动态数组不是意味着去改变原有内部生成的数组的长度、而是保留原有数组的引用、将其指向新生成的数组对象、这样会造成数组的长度可变的假象。
- 2、ArrayList具有数组所具有的特性、通过索引支持随机访问、所以通过随机访问ArrayList中的元素效率非常高、但是执行插入、删除时效率比较低下、具体原因后面有分析。
- 3、ArrayList实现了AbstractList抽象类、List接口、所以其更具有了AbstractList和List的功能、前面我们知道AbstractList内部已经实现了获取Iterator和ListIterator的方法、所以ArrayList只需关心

对数组操作的方法的实现、

- 4、ArrayList实现了RandomAccess接口、此接口只有声明、没有方法体、表示ArrayList支持随机访问。
- 5、ArrayList实现了Cloneable接口、此接口只有声明、没有方法体、表示ArrayList支持克隆。
- 6、ArrayList实现了Serializable接口、此接口只有声明、没有方法体、表示ArrayList支持序列化、即可以将ArrayList以流的形式通过ObjectInputStream/ObjectOutputStream来写/读。

基础属性

ArrayList部分源码如下：

```
1 public class ArrayList<E> extends AbstractList<E>
2     implements List<E>, RandomAccess, Cloneable, java.io.Seri
3     alizable
4 {
5     private static final int DEFAULT_CAPACITY = 10;
6
7     private static final Object[] EMPTY_ELEMENTDATA = {};
8
9     private transient Object[] elementData;
10
11
12     //...省略部分代码
13 }
```

如上代码中为ArrayList的主要属性：

- DEFAULT_CAPACITY：默认容量，即为初始值大小
- EMPTY_ELEMENTDATA：共享的空数组，用于初始化空实例
- elementData：ArrayList内部结构，是一个Object[]类型的数组
- size：数组长度大小

构造方法

如下为ArrayList的构造方法：

```
1 public ArrayList(int initialCapacity)
2
3 public ArrayList()
4
5 public ArrayList(Collection<? extends E> c){
6     elementData = c.toArray();
7     size = elementData.length;
8     // c.toArray might (incorrectly) not return Object[] (see 626
9     // 0652)
10    if (elementData.getClass() != Object[].class)
11        elementData = Arrays.copyOf(elementData, size, Object[].c
lass);
11 }
```

- 1.构造方法1，表示接受指定地容量值，初始化创建数组，建议在可估算数组大小时,创建ArrayList可指定
- 2.构造方法2，是默认的构造方法，它将创建一个空数组
- 3.构造方法3，接收一个Collection的实体，将该Collection实体转换为ArrayList对象

主干流程

1.添加指定元素代码如下

```
1 public boolean add(E e) {
2     ensureCapacityInternal(size + 1); // Increments modCount!!
3     elementData[size++] = e;
4     return true;
5 }
```

可以看到实际上只有3行代码，其流程主要如下：

1.扩容（这里便解释了，在介绍时提出的问题）：

主要源码如下

```
1
2 private void ensureCapacityInternal(int minCapacity) {
3     if (elementData == EMPTY_ELEMENTDATA) {
4         minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
5     }
6
7     ensureExplicitCapacity(minCapacity);
8 }
9
10 private void ensureExplicitCapacity(int minCapacity) {
11     modCount++;
12
13     // overflow-conscious code
14     if (minCapacity - elementData.length > 0)
15         grow(minCapacity);
16 }
17
18 //最大数组容量
19 private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
20
21 private void grow(int minCapacity) {
22     // overflow-conscious code
23     int oldCapacity = elementData.length;
24     int newCapacity = oldCapacity + (oldCapacity >> 1);
25     if (newCapacity - minCapacity < 0)
26         newCapacity = minCapacity;
27     if (newCapacity - MAX_ARRAY_SIZE > 0)
28         newCapacity = hugeCapacity(minCapacity);
29     // minCapacity is usually close to size, so this is a win:
30     elementData = Arrays.copyOf(elementData, newCapacity);
31 }
```

公众号：方志朋

- 第一个方法的逻辑为：判断是不是第一次添加元素，若为第一次，则设置初始化大小为默认的值10，否则使用传入的参数
- 第二个方法的逻辑为：若长度大于数组长度，则扩容
- 第三个方法的逻辑为：

1·扩容的大小为3/2倍原数组长度

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 2.若值newCapacity比传入值minCapacity还要小，则使用传入minCapacity，若newCapacity比设定的最大数组容量大，则使用最大整数值
- 3.实际扩容，使用了Arrays.copyOf(elementData, newCapacity)
(此处有两个问题
 - 1.为啥扩容是原来的3/2倍原数组的长度？
 - 2.调用Arrays.copyOf(elementData, newCapacity)方法具体做了什么操作？)

2.赋值：将添加的值放置到size++的位置上

3.返回：返回true

2.添加指定元素到指定的位置上代码如下：

```
1
2 public void add(int index, E element) {
3     rangeCheckForAdd(index);
4
5     ensureCapacityInternal(size + 1); // Increments modCount!!
6     System.arraycopy(elementData, index, elementData, index + 1,
7                       size - index);
8     elementData[index] = element;
9     size++;
10 }
```

其流程为：

- 1.校验下标：调用rangeCheckForAdd方法进行下标校验，不正确则会抛出IndexOutOfBoundsException异常
- 2.扩容：详见上部分中做的介绍
- 3.移动数据：将数据index后面的数据，都向后移动
- 4.赋值：将加入的值放置到index位置中
- 5.长度增加：长度增加

公众号：方志朋

常见问题

1.问题描述

在使用ArrayList比较常见的一个问题就是在遍历ArrayList的时候调用remove()方法进行元素的删除操作，从而得到意想不到的结果，本人在开发过程中也遇到过这样的问题，所以在这里提出了，希望能够帮助到大家。

2.实例及分析

如下代码中，在遍历List时，调用了remove方法，删除元素a

```
1 //arrayList中的值为 [a,a,c,a,a]
2 for (int i = 0; i < arrayList.size(); i++) {
3     if (arrayList.get(i) == "a") {
4         arrayList.remove(i);
5     }
6 }
7 System.out.println(arrayList);
```

公众号：方志朋

- 这段代码看似解决了删除列表中所有的a元素，但是删除后得出List的结果为[a, c, a]，为什么这种方式没有达到想要的效果，其实仔细分析后会发现，在调用remove()方法时List的长度会发生变化而且元素的位置会发生移动，从而在遍历时list实际上是变化的，例如
- 当i=0时，此时list中的元素为[a,a,c,a,a]，
- 但当i=1时，此时List中的元素为[a,c,a,a]，元素的位置发生了移动，从而导致在遍历的过程中不能达到删除的效果

3.解决方案

通过上述的分析可以看出，出现问题的原因是元素的位置发生了移动，从而导致异常的结果

方案一、逆向遍历List删除,代码如下，这种做法可行主要是因为remove()方法删除index处的元素时，是将index+1到size-1索引处的元素前移，而逆向遍历可以避免元素位置的移动

```
1 for (int i = arrayList.size()-1; i >=0 ; i--) {
2     if (arrayList.get(i) == "a") {
3         arrayList.remove(i);
```

```
4     }
5 }
6 System.out.println(arrayList);
```

方案二、使用迭代器中的remove方法，迭代器具体参考Iterator详解，主要代码如下(这种方式比较推荐)

```
1 Iterator<String> ite = arrayList.listIterator();
2 while (ite.hasNext()){
3     if(ite.next() == "a")
4         ite.remove();
5 }
6 System.out.println(arrayList);
```

手写一个ArrayList

自己手写一个ArrayList,代码如下：

```
1
2 public class MyArrayList<T> implements Iterable<T> {
3     private T[] theItems;
4     private int theSize;
5     private static final int DEFAULT_CAPACITY=10;
6
7     public MyArrayList(){
8         theSize=0;
9         ensureCapacity(DEFAULT_CAPACITY);
10
11    }
12
13     public void add(T data){
14         if(size()==theItems.length){
15             ensureCapacity(size()*2+1);
16         }
17         theItems[size()]=data;
18         theSize++;
19     }
```

公众号：方志朋

```
20
21     public void add(int index,T data){
22         if(size()==theItems.length){
23             ensureCapacity(size()*2+1);
24         }
25         for(int i=theSize;i>index;i--){
26             theItems[i]=theItems[i-1];
27         }
28         theItems[index]=data;
29         theSize++;
30     }
31
32     public T get(int index){
33         if(index<0 | index>=size())){
34             throw new IndexOutOfBoundsException("index error");
35         }
36         return theItems[index];
37     }
38
39     public T remove(int index){
40         T removeData=get(index);
41         for(int i=index;i<size()-1;i++){
42             theItems[i]=theItems[i+1];
43         }
44         theSize--;
45         return removeData;
46     }
47
48     public int size(){
49         return theSize;
50     }
51
52     private void ensureCapacity(int newCapacity){
53         if(theSize>newCapacity){
54             return;
55         }
56
57         T[] old=theItems;
58         theItems= (T[]) new Object[newCapacity];
59         for(int i=0;i<size();i++){
```

```
60         theItems[i] = old[i];
61     }
62 }
63
64
65
66     @Override
67     public Iterator<T> iterator() {
68         return null;
69     }
70
71     @Override
72     public void forEach(Consumer<? super T> action) {
73
74     }
75
76     @Override
77     public Spliterator<T> spliterator() {
78         return null;
79     }
80 }
```

公众号：方志朋

总结

- 1.ArrayList是基于数组实现的，它的内存储元素的数组为 elementData;elementData的声明为：
 transient Object[] elementData;
- 2.ArrayList中EMPTY_ELEMENTDATA和DEFAULTCAPACITY_EMPTY_ELEMENTDATA的使用；
 这两个常量，使用场景不同。前者是用在用户通过ArrayList(int initialCapacity)该构造方法直接指定
 初始容量为0时，后者是用户直接使用无参构造创建ArrayList时。
- 3.ArrayList默认容量为10。调用无参构造新建一个ArrayList时，它的elementData =
 DEFAULTCAPACITY_EMPTY_ELEMENTDATA, 当第一次使用 add() 添加元素时，ArrayList的容量
 会为 10。
- 4.ArrayList的扩容计算为 newCapacity = oldCapacity + (oldCapacity >> 1);且扩容并非是无限制
 的，有内存限制，虚拟机限制。
- 5.ArrayList的toArray()方法和subList()方法，在源数据和子数据之间的区别；
- 6.注意扩容方法ensureCapacityInternal()。ArrayList在每次增加元素（可能是1个，也可能是一组）
 时，都要调用该方法来确保足够的容量。当容量不足以容纳当前的元素个数时，就设置新的容量为旧的
 容量的1.5倍加1，如果设置后的新容量还不够，则直接新容量设置为传入的参数（也就是所需的容

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！量），而后用Arrays.copyOf()方法将元素拷贝到新的数组。从中可以看出，当容量不够时，每次增加元素，都要将原来的元素拷贝到一个新的数组中，非常耗时，也因此建议在事先能确定元素数量的情况下，才使用ArrayList，否则不建议使用。

参考资料

https://blog.csdn.net/crave_shy/article/details/17436773

<https://www.jianshu.com/p/92373a603d42>

<https://juejin.im/post/5a1bc1006fb9a045030fce0e>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：

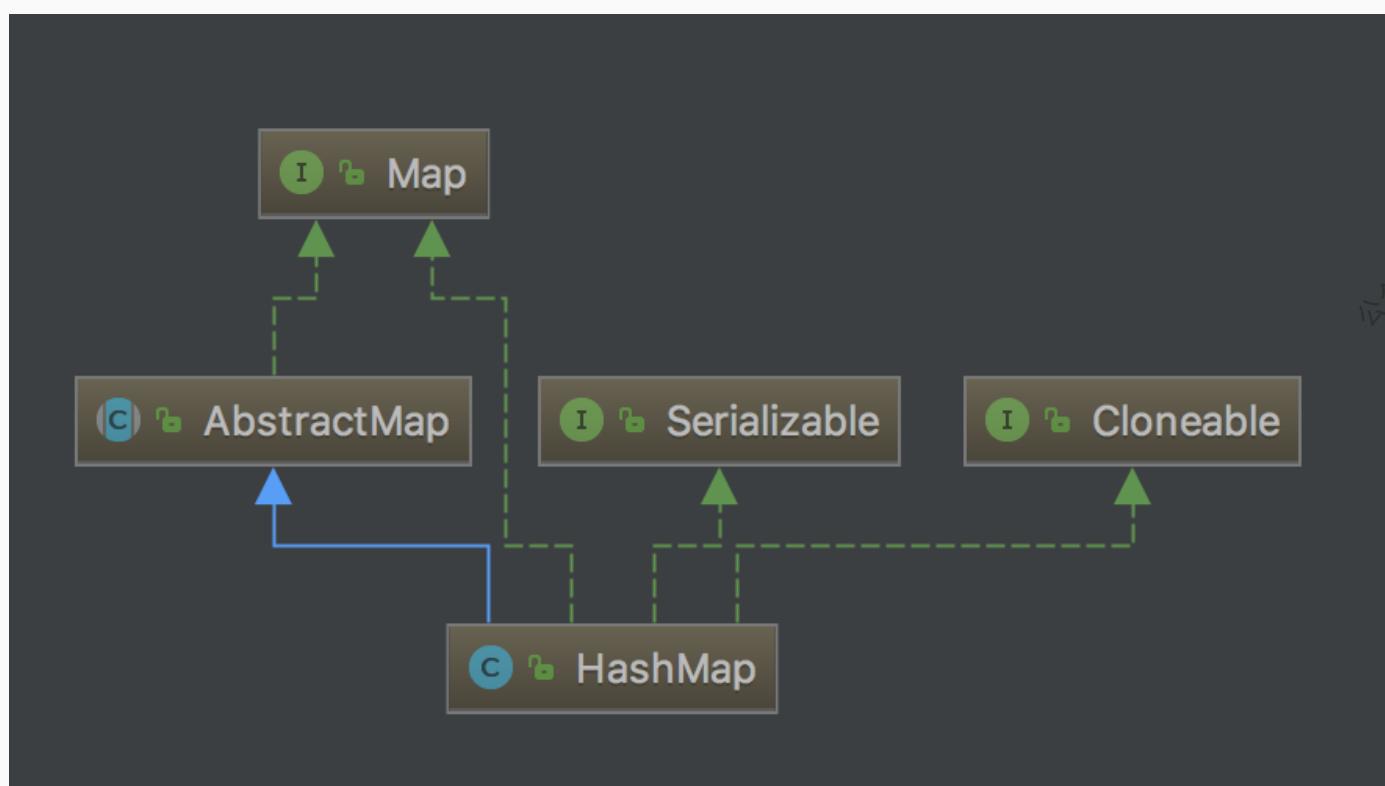


Java基础：Java容器之HashMap

Java容器之HashMap

HashMap 概述

Map 是 Key–Value 对映射的抽象接口，该映射不包括重复的键，即一个键对应一个值。HashMap 是 Java Collection Framework 的重要成员，也是 Map 族(如下图所示)中我们最为常用的一种。简单地说，HashMap 是基于哈希表的 Map 接口的实现，以 Key–Value 的形式存在，即存储的对象是 Entry (同时包含了 Key 和 Value)。在HashMap中，其会根据hash算法来计算key–value的存储位置并进行快速存取。特别地，HashMap最多只允许一条Entry的键为Null(多条会覆盖)，但允许多条Entry的值为Null。此外，HashMap 是 Map 的一个非同步的实现。



同样地，HashSet 也是 Java Collection Framework 的重要成员，是 Set 接口的常用实现类，但其与 HashMap 有很多相似之处。对于 HashSet 而言，其采用 Hash 算法决定元素在Set中的存储位置，这样可以保证元素的快速存取；对于 HashMap 而言，其将 key–value 当成一个整体(Entry 对象)来处理，其也采用同样的 Hash 算法去决定 key–value 的存储位置从而保证键值对的快速存取。虽然 HashMap 和 HashSet 实现的接口规范不同，但是它们底层的 Hash 存储机制完全相同。实际上，HashSet 本身就是在 HashMap 的基础上实现的。因此，通过对 HashMap 的数据结构、实现原理、源码实现三个方面了解，我们不但可以进一步掌握其底层的 Hash 存储机制，也有助于对 HashSet 的了解。

必须指出的是，虽然容器号称存储的是 Java 对象，但实际上并不会真正将 Java 对象放入容器中，只是在容器中保留这些对象的引用。也就是说，Java 容器实际上包含的是引用变量，而这些引用变量指向了我们要实际保存的 Java 对象。

当我们执行下面的操作时：

```
1 HashMap<String, Integer> map = new HashMap<String, Integer>();
2 map.put("语文", 1);
3 map.put("数学", 2);
4 map.put("英语", 3);
5 map.put("历史", 4);
6 map.put("政治", 5);
7 map.put("地理", 6);
8 map.put("生物", 7);
9 map.put("化学", 8);
10 for(Entry<String, Integer> entry : map.entrySet()) {
11     System.out.println(entry.getKey() + ":" + entry.getValue());
12 }
```

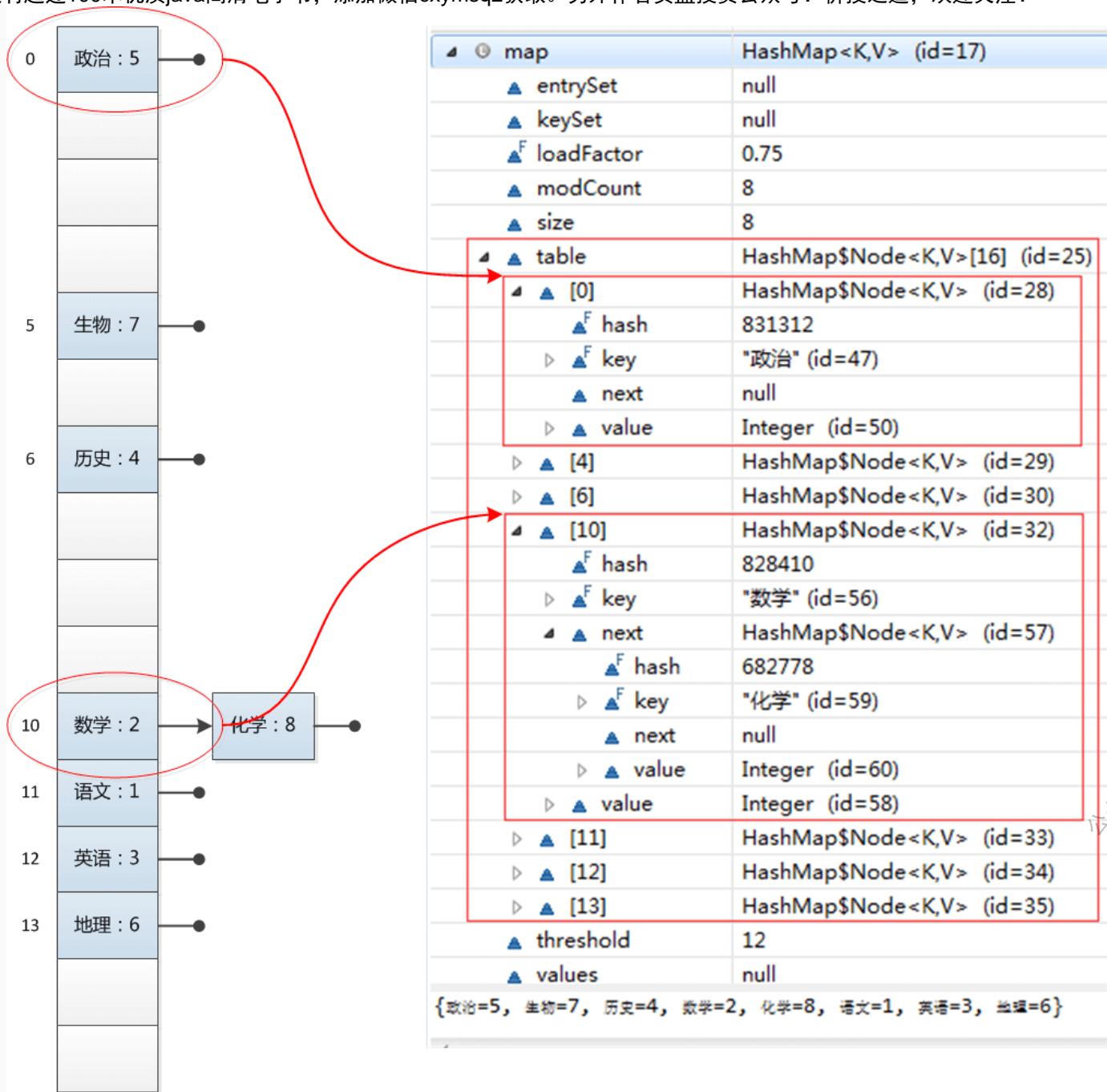
运行结果是

```
政治: 5
生物: 7
历史: 4
```

```
数学: 2
化学: 8
语文: 1
英语: 3
地理: 6
```

发生了什么呢？下面是一个大致的结构，希望我们对HashMap的结构有一个感性的认识：

公众号：方志朋



在官方文档中是这样描述HashMap的：

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

几个关键的信息：基于Map接口实现、允许null键/值、非同步、不保证有序(比如插入的顺序)、也不保证序不随时间变化。

两个重要的参数

在HashMap中有两个很重要的参数，容量(Capacity)和负载因子(Load factor)

- Initial capacity The capacity is the number of buckets in the hash table, The initial capacity is simply the capacity at the time the hash table is created.
- Load factor The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.

简单的说，Capacity就是buckets的数目，Load factor就是buckets填满程度的最大比例。如果对迭代性能要求很高的话不要把capacity设置过大，也不要把load factor设置过小。当bucket填充的数目（即 hashmap中元素的个数）大于capacity*load factor时就需要调整buckets的数目为当前的2倍。

put函数的实现

put函数大致的思路为：

- 对key的hashCode()做hash，然后再计算index;
- 如果没碰撞直接放到bucket里；
- 如果碰撞了，以链表的形式存在buckets后；
- 如果碰撞导致链表过长(大于等于TREEIFY_THRESHOLD)，就把链表转换成红黑树；
- 如果节点已经存在就替换old value(保证key的唯一性)
- 如果bucket满了(超过load factor*current capacity)，就要resize。

具体代码的实现如下：

```
1 public V put(K key, V value) {  
2     // 对key的hashCode()做hash  
3     return putVal(hash(key), key, value, false, true);  
4 }  
5 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  
6                 boolean evict) {  
7     Node<K,V>[] tab; Node<K,V> p; int n, i;  
8     // tab为空则创建  
9     if ((tab = table) == null || (n = tab.length) == 0)  
10        n = (tab = resize()).length;  
11    // 计算index，并对null做处理
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
12     if ((p = tab[i = (n - 1) & hash]) == null)
13         tab[i] = newNode(hash, key, value, null);
14     else {
15         Node<K,V> e; K k;
16         // 节点存在
17         if (p.hash == hash &&
18             ((k = p.key) == key || (key != null && key.equals(k)))
19         )
20             e = p;
21         // 该链为树
22         else if (p instanceof TreeNode)
23             e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, ke
24                 y, value);
25         // 该链为链表
26         else {
27             for (int binCount = 0; ; ++binCount) {
28                 if ((e = p.next) == null) {
29                     p.next = newNode(hash, key, value, null);
30                     if (binCount >= TREEIFY_THRESHOLD - 1) // -1
31                         treeifyBin(tab, hash);
32                     break;
33                 }
34                 if (e.hash == hash &&
35                     ((k = e.key) == key || (key != null && key.eq
36                     uals(k))))
37                     break;
38                 p = e;
39             }
40             // 写入
41             if (e != null) { // existing mapping for key
42                 V oldValue = e.value;
43                 if (!onlyIfAbsent || oldValue == null)
44                     e.value = value;
45                 afterNodeAccess(e);
46             }
47             ++modCount;
}
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
48     // 超过load factor*current capacity, resize
49     if (++size > threshold)
50         resize();
51     afterNodeInsertion(evict);
52     return null;
53 }
```

get函数的实现

在理解了put之后，get就很简单了。大致思路如下：

- bucket里的第一个节点，直接命中；
- 如果有冲突，则通过key.equals(k)去查找对应的entry
- 若为树，则在树中通过key.equals(k)查找，O(logn)；
- 若为链表，则在链表中通过key.equals(k)查找，O(n)。

具体代码的实现如下：

```
1
2 public V get(Object key) {
3     Node<K,V> e;
4     return (e = getNode(hash(key), key)) == null ? null : e.value
5 ;
6 }
7 final Node<K,V> getNode(int hash, Object key) {
8     Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
9     if ((tab = table) != null && (n = tab.length) > 0 &&
10        (first = tab[(n - 1) & hash]) != null) {
11         // 直接命中
12         if (first.hash == hash && // always check first node
13            ((k = first.key) == key || (key != null && key.equals
14            (k))))
15             return first;
16         // 未命中
17         if ((e = first.next) != null) {
18             // 在树中get
19             if (first instanceof TreeNode)
20                 return ((TreeNode<K,V>)first).getTreeNode(hash, k
21
22 }
```

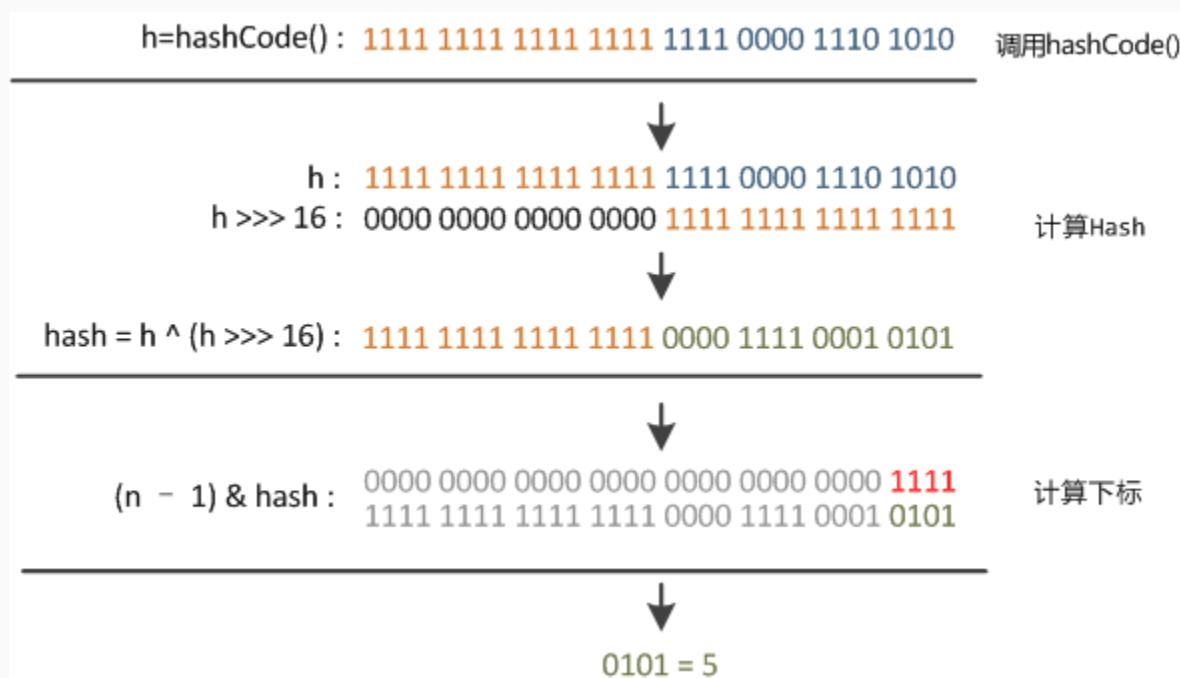
公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
ey);  
19         // 在链表中get  
20     do {  
21         if (e.hash == hash &&  
22             ((k = e.key) == key || (key != null && key.eq  
uals(k))))  
23             return e;  
24     } while ((e = e.next) != null);  
25 }  
26 }  
27 return null;  
28 }
```

hash函数的实现

在get和put的过程中，计算下标时，先对hashCode进行hash操作，然后再通过hash值进一步计算下标，如下图所示：



在对hashCode()计算hash时具体实现是这样的：

```
1 static final int hash(Object key) {  
2     int h;
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
3     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
4 }
```

在设计hash函数时，因为目前的table长度n为2的幂，而计算下标的时候，是这样实现的(使用&位操作，而非%求余)：

```
1  
2 (n - 1) & hash
```

设计者认为这方法很容易发生碰撞。为什么这么说呢？不妨思考一下，在 $n - 1$ 为15(0x1111)时，其实散列真正生效的只是低4bit的有效位，当然容易碰撞了。

因此，设计者想了一个顾全大局的方法(综合考虑了速度、作用、质量)，就是把高16bit和低16bit异或了一下。设计者还解释到因为现在大多数的hashCode的分布已经很不错了，就算是发生了碰撞也用 $O(\log n)$ 的tree去做了。仅仅异或一下，既减少了系统的开销，也不会造成的因为高位没有参与下标的计算(table长度比较小时)，从而引起的碰撞。

如果还是产生了频繁的碰撞，会发生什么问题呢？作者注释说，他们使用树来处理频繁的碰撞(we use trees to handle large sets of collisions in bins)，在JEP-180中，描述了这个问题：

Improve the performance of java.util.HashMap under high hash-collision conditions by using balanced trees rather than linked lists to store map entries. Implement the same improvement in the LinkedHashMap class.

之前已经提过，在获取HashMap的元素时，基本分两步：

- 首先根据hashCode()做hash，然后确定bucket的index；
- 如果bucket的节点的key不是我们需要的，则通过keys.equals()在链中找。

在Java 8之前的实现中是用链表解决冲突的，在产生碰撞的情况下，进行get时，两步的时间复杂度是 $O(1)+O(n)$ 。因此，当碰撞很厉害的时候n很大， $O(n)$ 的速度显然是影响速度的。

因此在Java 8中，利用红黑树替换链表，这样复杂度就变成了 $O(1)+O(\log n)$ 了，这样在n很大的时候，能够比较理想的解决这个问题，在Java 8：HashMap的性能提升一文中有性能测试的结果。

RESIZE的实现

当put时，如果发现目前的bucket占用程度已经超过了Load Factor所希望的比例，那么就会发生resize。在resize的过程，简单的说就是把bucket扩充为2倍，之后重新计算index，把节点再放到新的bucket中。resize的注释是这样描述的：

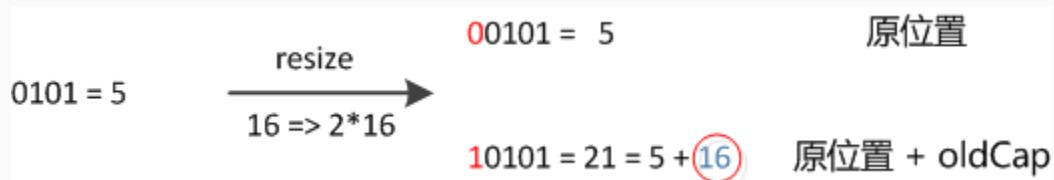
Initializes or doubles table size. If null, allocates in accord with initial capacity target held in field threshold. Otherwise, because we are using power-of-two expansion, the elements from each bin must either stay at same index, or move with a power of two offset in the new table.

大致意思就是说，当超过限制的时候会resize，然而又因为我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。

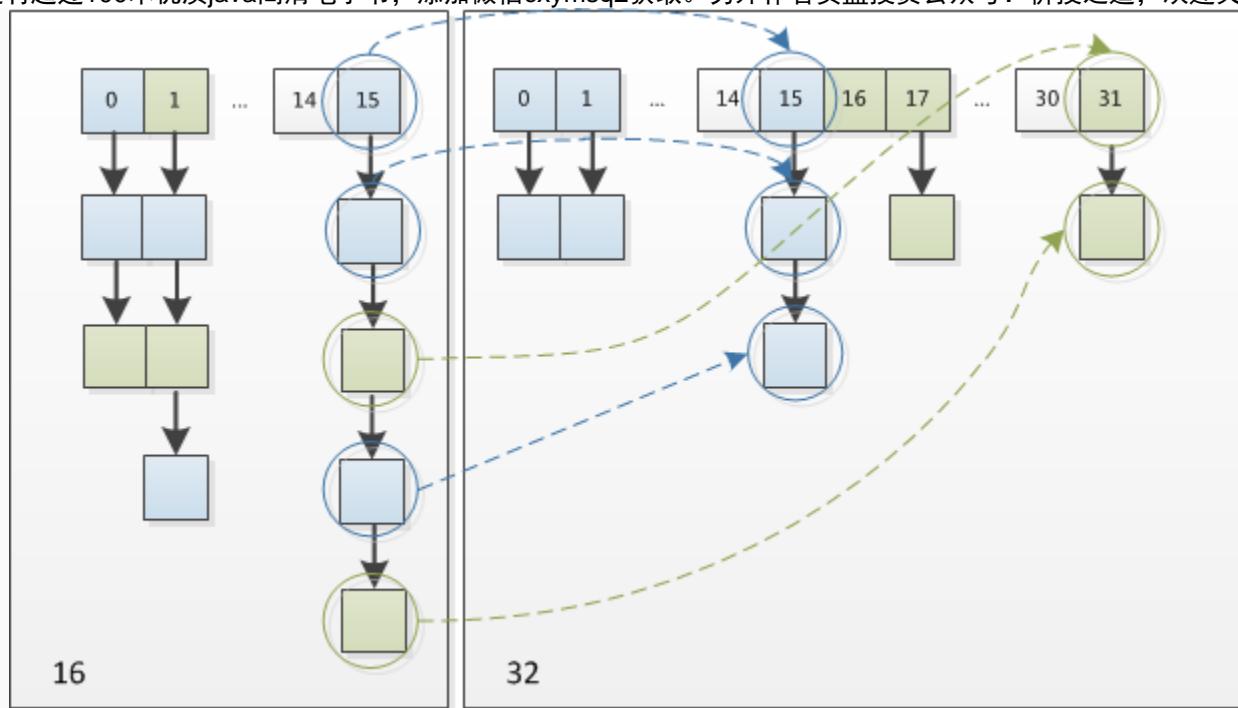
怎么理解呢？例如我们从16扩展为32时，具体的变化如下所示：

<code>n - 1 0000 0000 0000 0000 0000 0000 1111</code>	<code>1111 1111 1111 1111 0000 1111 0001 1111</code>
<code>hash1 1111 1111 1111 1111 0000 1111 0000 0101</code>	<code>1111 1111 1111 1111 0000 1111 0000 0 0101</code>
<code>hash2 1111 1111 1111 1111 0000 1111 0001 0101</code>	<code>1111 1111 1111 1111 0000 1111 0001 0101</code>

因此元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



因此，我们在扩充HashMap的时候，不需要重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”。可以看看下图为16扩充为32的resize示意图：



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。

下面是代码的具体实现：

```
1 final Node<K,V>[] resize() {
2     Node<K,V>[] oldTab = table;
3     int oldCap = (oldTab == null) ? 0 : oldTab.length;
4     int oldThr = threshold;
5     int newCap, newThr = 0;
6     if (oldCap > 0) {
7         // 超过最大值就不再扩充了，就只好随你碰撞去吧
8         if (oldCap >= MAXIMUM_CAPACITY) {
9             threshold = Integer.MAX_VALUE;
10            return oldTab;
11        }
12        // 没超过最大值，就扩充为原来的2倍
13        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
14                  oldCap >= DEFAULT_INITIAL_CAPACITY)
15            newThr = oldThr << 1; // double threshold
16    }
17    else if (oldThr > 0) // initial capacity was placed in threshold
old
```

```
18     newCap = oldThr;
19     else { // zero initial threshold signifies using defaults
20         newCap = DEFAULT_INITIAL_CAPACITY;
21         newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
22     }
23     // 计算新的resize上限
24     if (newThr == 0) {
25         float ft = (float)newCap * loadFactor;
26         newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
27             (int)ft : Integer.MAX_VALUE);
28     }
29     threshold = newThr;
30     @SuppressWarnings({"rawtypes","unchecked"})
31     Node<K,V>[] newTab = (Node<K,V>[] )new Node[newCap];
32     table = newTab;
33     if (oldTab != null) {
34         // 把每个bucket都移动到新的buckets中
35         for (int j = 0; j < oldCap; ++j) {
36             Node<K,V> e;
37             if ((e = oldTab[j]) != null) {
38                 oldTab[j] = null;
39                 if (e.next == null)
40                     newTab[e.hash & (newCap - 1)] = e;
41                 else if (e instanceof TreeNode)
42                     ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
43             else { // preserve order
44                 Node<K,V> loHead = null, loTail = null;
45                 Node<K,V> hiHead = null, hiTail = null;
46                 Node<K,V> next;
47                 do {
48                     next = e.next;
49                     // 原索引
50                     if ((e.hash & oldCap) == 0) {
51                         if (loTail == null)
52                             loHead = e;
53                         else
```

```
54                     loTail.next = e;
55                     loTail = e;
56                 }
57                 // 原索引+oldCap
58             else {
59                 if (hiTail == null)
60                     hiHead = e;
61                 else
62                     hiTail.next = e;
63                 hiTail = e;
64             }
65         } while ((e = next) != null);
66         // 原索引放到bucket里
67         if (loTail != null) {
68             loTail.next = null;
69             newTab[j] = loHead;
70         }
71         // 原索引+oldCap放到bucket里
72         if (hiTail != null) {
73             hiTail.next = null;
74             newTab[j + oldCap] = hiHead;
75         }
76     }
77 }
78 }
79 }
80 return newTab;
81 }
```

公众号：方志朋

总结

我们现在可以回答开始的几个问题，加深对HashMap的理解：

1. 什么时候会使用HashMap？他有什么特点？

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

是基于Map接口的实现，存储键值对时，它可以接收null的键值，是非同步的，HashMap存储着Entry(hash, key, value, next)对象。

2. 你知道HashMap的工作原理吗？

通过hash的方法，通过put和get存储和获取对象。存储对象时，我们将K/V传给put方法时，它调用hashCode计算hash从而得到bucket位置，进一步存储，HashMap会根据当前bucket的占用情况自动调整容量(超过Load Factor则resize为原来的2倍)。获取对象时，我们将K传给get，它调用hashCode计算hash从而得到bucket位置，并进一步调用equals()方法确定键值对。如果发生碰撞的时候，HashMap通过链表将产生碰撞冲突的元素组织起来，在Java 8中，如果一个bucket中碰撞冲突的元素超过某个限制(默认是8)，则使用红黑树来替换链表，从而提高速度。

3. 你知道get和put的原理吗？equals()和hashCode()的都有什么作用？

通过对key的hashCode()进行hashing，并计算下标($n - 1 \& \text{hash}$)，从而获得buckets的位置。如果产生碰撞，则利用key.equals()方法去链表或树中去查找对应的节点

4. 你知道hash的实现吗？为什么要这样实现？

在Java 1.8的实现中，是通过hashCode()的高16位异或低16位实现的： $(h = k.hashCode()) ^ (h >>> 16)$ ，主要是从速度、功效、质量来考虑的，这么做可以在bucket的n比较小的时候，也能保证考虑到高低bit都参与到hash的计算中，同时不会有太大的开销。

5. 如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？

如果超过了负载因子(默认0.75)，则会重新resize一个原来长度两倍的HashMap，并且重新调用hash方法。

自己动手写一个HashMap

实现的过程如下：

```
1 package com.forezp.datastruct.lqbz;  
2
```

```
3 /**
4  * Created by forezp on 2018/3/20.
5 */
6 public class MyHashMap<K, V> {
7
8
9     private Node[] theNodes;
10    private int size;
11    private int THE_NODES_SIZE = 16;
12
13    public MyHashMap() {
14        theNodes = new Node[THE_NODES_SIZE];
15        size = 0;
16    }
17
18    public void put(K key, V value) {
19        Node node = findNode(key);
20        if (node == null) {
21            node = new Node(key.hashCode(), key, value, null);
22            theNodes[findItemIndex(key)] = node;
23        } else {
24            while (node.next != null) {
25                node = node.next;
26            }
27            node.next = new Node(key.hashCode(), key, value, null
28        );
29        size++;
30    }
31
32    public V get(K key) {
33        Node<K, V> node = theNodes[findItemIndex(key)];
34        if (node == null) {
35            return null;
36        }
37        while (node.next != null) {
38            if (node != null && node.hash != key.hashCode()) {
39                node = node.next;
40            } else {
41                if (node != null) {
```

```
42             return node.value;
43         }
44     }
45 }
46 return null;
47
48 }
49
50 private Node findNode(K key) {
51     return theNodes[findItemIndex(key)];
52 }
53
54 private int findItemIndex(K key) {
55     int hashCode = key.hashCode();
56     int nodeIndex = hashCode % THE_NODES_SIZE;
57     return nodeIndex;
58 }
59
60 class Node<K, V> {
61     public int hash;
62     public K key;
63     public V value;
64     public Node next;
65
66     public Node(int hash, K key, V value, Node next) {
67         this.hash = hash;
68         this.key = key;
69         this.value = value;
70         this.next = next;
71     }
72 }
73
74
75 }
```

公众号：方志朋

参考资料

https://blog.csdn.net/justloveyou_/article/details/62893086

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

Java基础： Java IO流学习总结

IO是指对数据流的输入和输出，也称为IO流，IO流主要分为两大类，字节流和字符流。字节流可以处理任何类型的数据，如图片，视频等，字符流只能处理字符类型的数据。

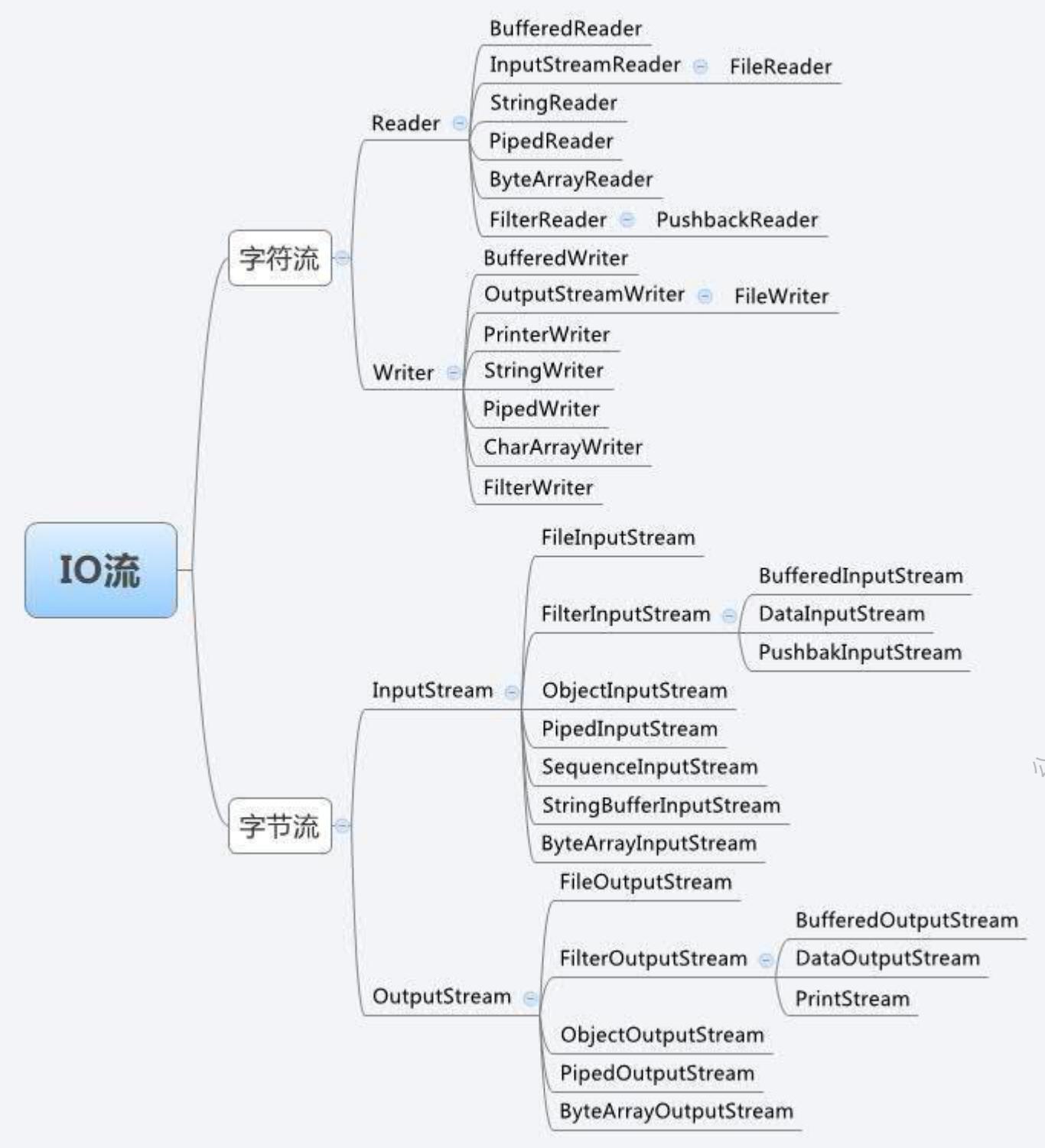
IO流的本质是数据传输，并且流是单向的。

Java流操作有关的类或接口：

类	说明
File	文件类
RandomAccessFile	随机存取文件类
InputStream	字节输入流
OutputStream	字节输出流
Reader	字符输入流
Writer	字符输出流

Java流类图结构

公众号：方志朋



公众号：方志朋

IO流的分类

字符流和字节流

字符流的由来： 因为数据编码的不同，而有了对字符进行高效操作的流对象。本质其实也就是基于字节流读取时，去查了指定的码表。 字节流和字符流的区别：

- 读写单位不同：字节流以字节（8bit）为单位，字符流以字符为单位，根据码表映射字符，一次可能读多个字节。
- 处理对象不同：字节流能处理所有类型的数据（如图片、avi等），而字符流只能处理字符类型的数据。

结论：只要是处理纯文本数据，就优先考虑使用字符流。除此之外都使用字节流。

Java IO流对象

1. 输入字节流InputStream

- IO 中输入字节流的继承图可见上图，可以看出：
- InputStream 是所有的输入字节流的父类，它是一个抽象类。
 - ByteArrayInputStream、StringBufferInputStream、FileInputStream 是三种基本的介质流，它们分别从Byte 数组、StringBuffer、和本地文件中读取数据。PipedInputStream 是从与其它线程共用的管道中读取数据，与Piped 相关的知识后续单独介绍。
 - ObjectInputStream 和所有FilterInputStream 的子类都是装饰流（装饰器模式的主角）。

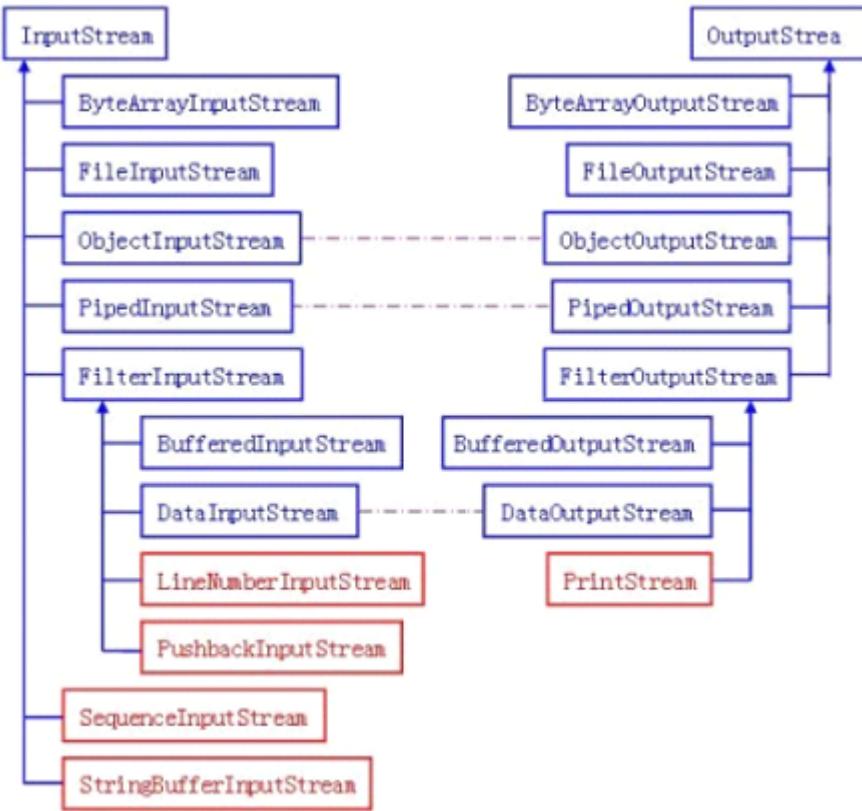
2. 输出字节流OutputStream

IO 中输出字节流的继承图可见上图，可以看出：

- OutputStream 是所有的输出字节流的父类，它是一个抽象类。
- ByteArrayOutputStream、FileOutputStream 是两种基本的介质流，它们分别向Byte 数组、和本地文件中写入数据。PipedOutputStream 是向与其它线程共用的管道中写入数据，
- ObjectOutputStream 和所有FilterOutputStream 的子类都是装饰流。

3. 字节流的输入与输出的对应

公众号：方志朋



图中蓝色的为最主要的对应部分，红色的部分就是不对应部分。紫色的虚线部分代表这些流一般要搭配使用。从上面的图中可以看出Java IO 中的字节流是极其对称的。“存在及合理”我们看看这些字节流中不太对称的几个类吧！

- LineNumberInputStream 主要完成从流中读取数据时，会得到相应的行号，至于什么时候分行、在哪里分行是由改类主动确定的，并不是在原始中有这样一个行号。在输出部分没有对应的部分，我们完全可以自己建立一个LineNumberOutputStream，在最初写入时会有一个基准的行号，以后每次遇到换行时会在下一行添加一个行号，看起来也是可以的。好像更不入流了。
- PushbackInputStream 的功能是查看最后一个字节，不满意就放入缓冲区。主要用在编译器的语法、词法分析部分。输出部分的BufferedOutputStream 几乎实现相近的功能。
- StringBufferInputStream 已经被Deprecated，本身就不应该出现在InputStream 部分，主要因为 String 应该属于字符流的范围。已经被废弃了，当然输出部分也没有必要需要它了！还允许它存在只是为了保持版本的向下兼容而已。
- SequenceInputStream 可以认为是一个工具类，将两个或者多个输入流当成一个输入流依次读取。完全可以从IO 包中去除，还完全不影响IO 包的结构，却让其更“纯洁”——纯洁的Decorator 模式。
- PrintStream 也可以认为是一个辅助工具。主要可以向其他输出流，或者FileInputStream 写入数据，本身内部实现还是带缓冲的。本质上是对其它流的综合运用的一个工具而已。一样可以踢出IO 包！System.out 和System.out 就是PrintStream 的实例！

4.字符输入流Reader

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

在上面的继承关系图中可以看出：

- Reader 是所有的输入字符流的父类，它是一个抽象类。
- CharReader、StringReader 是两种基本的介质流，它们分别将Char 数组、String 中读取数据。PipedReader 是从与其它线程共用的管道中读取数据。
- BufferedReader 很明显就是一个装饰器，它和其子类负责装饰其它Reader 对象。
- FilterReader 是所有自定义具体装饰流的父类，其子类PushbackReader 对Reader 对象进行装饰，会增加一个行号。
- InputStreamReader 是一个连接字节流和字符流的桥梁，它将字节流转变为字符流。FileReader 可以说是一个达到此功能、常用的工具类，在其源代码中明显使用了将FileInputStream 转变为Reader 的方法。我们可以从这个类中得到一定的技巧。Reader 中各个类的用途和使用方法基本和InputStream 中的类使用一致。后面会有Reader 与InputStream 的对应关系。

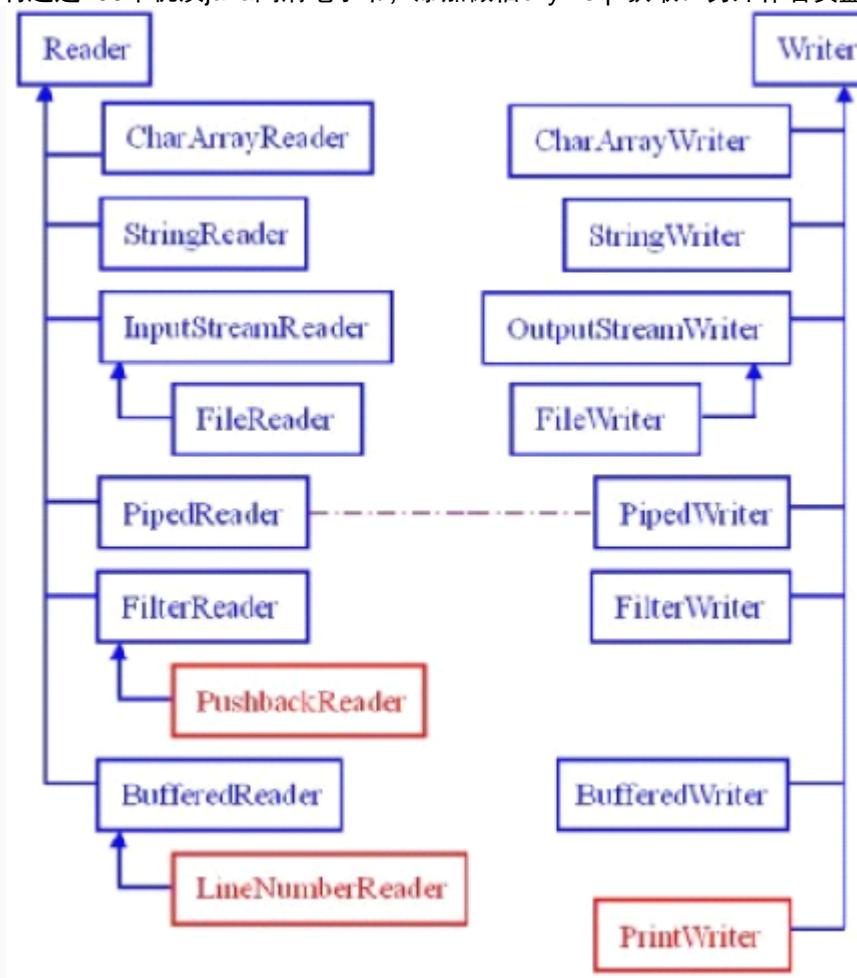
5.字符输出流Writer

在上面的关系图中可以看出：

- Writer 是所有的输出字符流的父类，它是一个抽象类。
- CharArrayWriter、StringWriter 是两种基本的介质流，它们分别向Char 数组、String 中写入数据。PipedWriter 是向与其它线程共用的管道中写入数据，
- BufferedWriter 是一个装饰器为Writer 提供缓冲功能。
- PrintWriter 和PrintStream 极其类似，功能和使用也非常相似。
- OutputStreamWriter 是OutputStream 到Writer 转换的桥梁，它的子类FileWriter 其实就是一个实现此功能的具体类（具体可以研究一SourceCode）。功能和使用和OutputStream 极其类似，后面会有它们的对应图。

6.字符流的输入与输出的对应

公众号：方志朋



公众号：方志朋

7.字符流与字节流转换

转换流的特点：

- 其是字符流和字节流之间的桥梁
- 可对读取到的字节数据经过指定编码转换成字符
- 可对读取到的字符数据经过指定编码转换成字节

何时使用转换流？

- 当字节和字符之间有转换动作时；
- 流操作的数据需要编码或解码时。

具体的对象体现：

- InputStreamReader:字节到字符的桥梁
- OutputStreamWriter:字符到字节的桥梁

这两个流对象是字符体系中的成员，它们有转换作用，本身又是字符流，所以在构造的时候需要传入字节流对象进来。

8.File类

File类是对文件系统中文件以及文件夹进行封装的对象，可以通过对象的思想来操作文件和文件夹。 File类保存文件或目录的各种元数据信息，包括文件名、文件长度、最后修改时间、是否可读、获取当前文件的路径名，判断指定文件是否存在、获得当前目录中的文件列表，创建、删除文件和目录等方法。

9.RandomAccessFile类

该对象并不是流体系中的一员，其封装了字节流，同时还封装了一个缓冲区（字符数组），通过内部的指针来操作字符数组中的数据。该对象特点：

- 该对象只能操作文件，所以构造函数接收两种类型的参数：a.字符串文件路径；b.File对象。
- 该对象既可以对文件进行读操作，也能进行写操作，在进行对象实例化时可指定操作模式(r,rw)

注意：该对象在实例化时，如果要操作的文件不存在，会自动创建；如果文件存在，写数据未指定位置，会从头开始写，即覆盖原有的内容。可以用于多线程下载或多个线程同时写数据到文件。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

Java基础：攻破JAVA NIO技术壁垒1

现在使用NIO的场景越来越多，很多网上的技术框架或多或少的使用NIO技术，譬如Tomcat，Jetty。学习和掌握NIO技术已经不是一个JAVA攻城狮的加分技能，而是一个必备技能。再者，现在互联网的面试中上点level的都会涉及一下NIO或者AIO的问题（AIO下次再讲述，本篇主要讲述NIO），掌握好NIO也能帮助你获得一份较好的offer。驱使博主写这篇文章的关键是网上关于NIO的文章并不是很多，而且案例较少，针对这个特性，本文主要通过实际案例主要讲述NIO的用法，每个案例都经过实际检验。博主通过自己的理解以及一些案例希望能给各位在学习NIO之时多一份参考。博主能力有限，文中有不足之处欢迎之处。

概述

NIO主要有三大核心部分：Channel(通道)，Buffer(缓冲区)，Selector。传统IO基于字节流和字符流进行操作，而NIO基于Channel和Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择区)用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。

NIO和传统IO（一下简称IO）之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

Channel

首先说一下Channel，国内大多翻译成“通道”。Channel和IO中的Stream(流)是差不多一个等级的。只不过Stream是单向的，譬如：InputStream, OutputStream.而Channel是双向的，既可以用来进行读操

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！
作，又可以用来进行写操作。

NIO中的Channel的主要实现有：

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

这里看名字就可以猜出个所以然来：分别可以对应文件IO、UDP和TCP（Server和Client）。下面演示的案例基本上就是围绕这4个类型的Channel进行陈述的。

Buffer

NIO中的关键Buffer实现有：ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer，分别对应基本数据类型：byte, char, double, float, int, long, short。当然NIO中还有MappedByteBuffer, HeapByteBuffer, DirectByteBuffer等这里先不进行陈述。

Selector

Selector运行单线程处理多个Channel，如果你的应用打开了多个通道，但每个连接的流量都很低，使用Selector就会很方便。例如在一个聊天服务器中。要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新的连接进来、数据接收等。

FileChannel

看完上面的陈述，对于第一次接触NIO的同学来说云里雾里，只说了一些概念，也没记住什么，更别说怎么用了。这里开始通过传统IO以及更改后的NIO来做对比，以更形象的突出NIO的用法，进而使你对NIO有一点点的了解。

传统IO vs NIO

首先，案例1是采用FileInputStream读取文件内容的：

```
1  public static void method2(){
2      InputStream in = null;
3      try{
4          in = new BufferedInputStream(new FileInputStream("sr
java架构师公众号：方志朋
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
    c/nomal_io.txt"));

5
6         byte [] buf = new byte[1024];
7         int bytesRead = in.read(buf);
8         while(bytesRead != -1)
9         {
10             for(int i=0;i<bytesRead;i++)
11                 System.out.print((char)buf[i]);
12             bytesRead = in.read(buf);
13         }
14     }catch (IOException e)
15     {
16         e.printStackTrace();
17     }finally{
18     try{
19         if(in != null){
20             in.close();
21         }
22     }catch (IOException e){
23         e.printStackTrace();
24     }
25 }
```

公众号：方志朋

输出结果：（略）

案例是对应的NIO（这里通过RandomAccessFile进行操作，当然也可以通过FileInputStream.getChannel()进行操作）：

```
1  public static void method1(){
2      RandomAccessFile aFile = null;
3      try{
4          aFile = new RandomAccessFile("src/nio.txt","rw");
5          FileChannel fileChannel = aFile.getChannel();
6          ByteBuffer buf = ByteBuffer.allocate(1024);
7
8          int bytesRead = fileChannel.read(buf);
9          System.out.println(bytesRead);
10 }
```

```
11     while(bytesRead != -1)
12     {
13         buf.flip();
14         while(buf.hasRemaining())
15         {
16             System.out.print((char)buf.get());
17         }
18
19         buf.compact();
20         bytesRead = fileChannel.read(buf);
21     }
22 }catch (IOException e){
23     e.printStackTrace();
24 }finally{
25     try{
26         if(aFile != null){
27             aFile.close();
28         }
29     }catch (IOException e){
30         e.printStackTrace();
31     }
32 }
33 }
```

公众号：方志朋

输出结果：（略）

通过仔细对比案例1和案例2，应该能看出个大概，最起码能发现NIO的实现方式比较复杂。有了一个大概的印象可以进入下一步了。

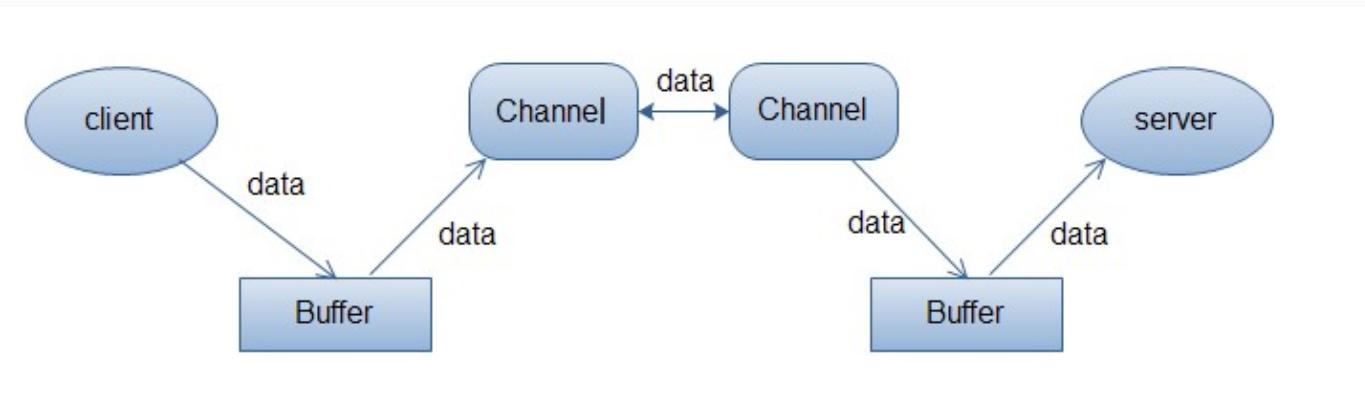
Buffer的使用

从案例2中可以总结出使用Buffer一般遵循下面几个步骤：

- 分配空间 (ByteBuffer buf = ByteBuffer.allocate(1024); 还有一种allocateDirect后面再陈述)
- 写入数据到Buffer(int bytesRead = fileChannel.read(buf);)
- 调用flip()方法 (buf.flip();)
- 从Buffer中读取数据 (System.out.print((char)buf.get());)
- 调用clear()方法或者compact()方法

Buffer顾名思义：缓冲区，实际上是一个容器，一个连续数组。Channel提供从文件、网络读取数据的

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！
渠道，但是读写的数据都必须经过Buffer。如下图：



向Buffer中写数据：

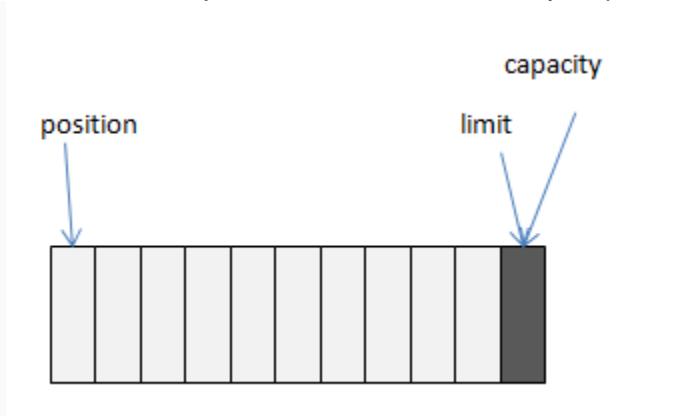
从Channel写到Buffer (`fileChannel.read(buf)`)

- 通过Buffer的put()方法 (`buf.put(...)`)
- 从Buffer中读取数据：

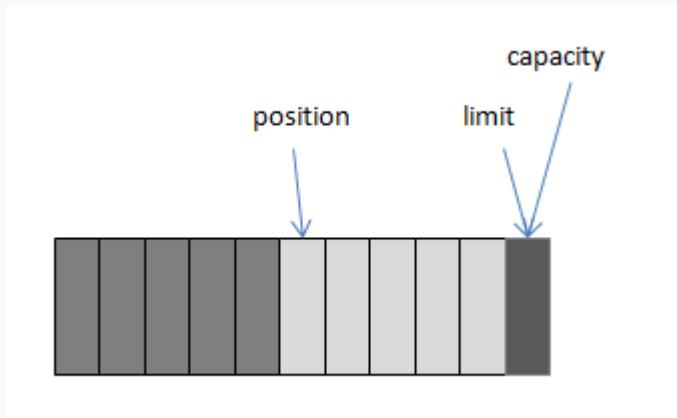
从Buffer读取到Channel (`channel.write(buf)`)

- 使用get()方法从Buffer中读取数据 (`buf.get()`)
- 可以把Buffer简单地理解为一组基本数据类型的元素列表，它通过几个变量来保存这个数据的当前位置状态：`capacity, position, limit, mark`：

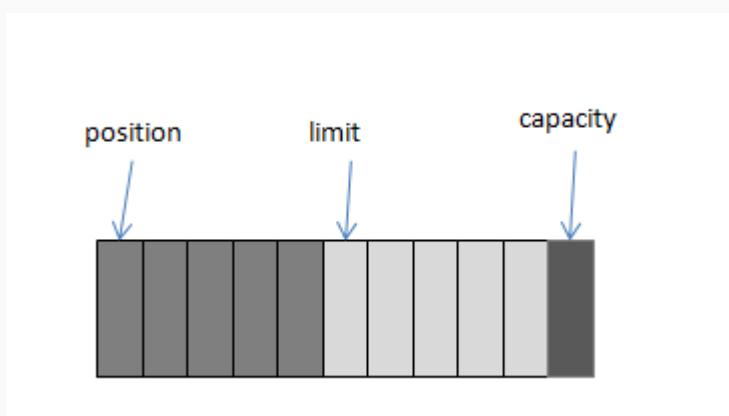
索引	说明
capacity	缓冲区数组的总长度
position	下一个要操作的数据元素的位置
limit	缓冲区数组中不可操作的下一个元素的位置： $limit \leq capacity$
mark	用于记录当前position的前一个位置或者默认是-1



图无真相，举例：我们通过ByteBuffer.allocate(11)方法创建了一个11个byte的数组的缓冲区，初始状态如上图，position的位置为0，capacity和limit默认都是数组长度。当我们写入5个字节时，变化如下图：



这时我们需要将缓冲区中的5个字节数据写入Channel的通信信道，所以我们调用ByteBuffer.flip()方法，变化如下图所示(position设回0，并将limit设成之前的position的值)：



这时底层操作系统就可以从缓冲区中正确读取这个5个字节数据并发送出去了。在下一次写数据之前我们再调用clear()方法，缓冲区的索引位置又回到了初始位置。

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

调用clear()方法：position将被设回0，limit设置成capacity，换句话说，Buffer被清空了，其实Buffer中的数据并未被清楚，只是这些标记告诉我们可以从哪里开始往Buffer里写数据。如果Buffer中有一些未读的数据，调用clear()方法，数据将“被遗忘”，意味着不再有任何标记会告诉你哪些数据被读过，哪些还没有。如果Buffer中仍有未读的数据，且后续还需要这些数据，但是此时想要先写些数据，那么使用compact()方法。compact()方法将所有未读的数据拷贝到Buffer起始处。然后将position设到最后一个未读元素正后面。limit属性依然像clear()方法一样，设置成capacity。现在Buffer准备好写数据了，但是不会覆盖未读的数据。

通过调用Buffer.mark()方法，可以标记Buffer中的一个特定的position，之后可以通过调用Buffer.reset()方法恢复到这个position。Buffer.rewind()方法将position设回0，所以你可以重读Buffer中的所有数据。limit保持不变，仍然表示能从Buffer中读取多少个元素。

原文链接

<https://blog.csdn.net/u013256816/article/details/51457215#comments>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



Java基础：攻破JAVA NIO技术壁垒2

攻破JAVA NIO技术壁垒2

SocketChannel

说完了FileChannel和Buffer，大家应该对Buffer的用法比较了解了，这里使用SocketChannel来继续探讨NIO。NIO的强大功能部分来自于Channel的非阻塞特性，套接字的某些操作可能会无限期地阻塞。例如，对accept()方法的调用可能会因为等待一个客户端连接而阻塞；对read()方法的调用可能会因为没有数据可读而阻塞，直到连接的另一端传来新的数据。总的来说，创建/接收连接或读写数据等I/O调用，都可能无限期地阻塞等待，到底层的网络实现发生了什么。慢速的，有损耗的网络，或仅仅是简单的网络故障都可能导致任意时间的延迟。然而不幸的是，在调用一个方法之前无法知道其是否阻塞。NIO的channel抽象的一个重要特征就是可以通过配置它的阻塞行为，以实现非阻塞式的信道。

```
1  
2 channel.configureBlocking(false)
```

在非阻塞式信道上调用一个方法总是会立即返回。这种调用的返回值指示了所请求的操作完成的程度。例如，在一个非阻塞式ServerSocketChannel上调用accept()方法，如果有连接请求来了，则返回客户端SocketChannel，否则返回null。

这里先举一个TCP应用案例，客户端采用NIO实现，而服务端依旧使用IO实现。

客户端代码（案例3）：

```
1 public static void client(){  
2     ByteBuffer buffer = ByteBuffer.allocate(1024);  
3     SocketChannel socketChannel = null;  
4     try  
5     {  
6         socketChannel = SocketChannel.open();  
7         socketChannel.configureBlocking(false);  
8         socketChannel.connect(new InetSocketAddress("10.10.19  
5.115", 8080));  
9     }  
10    if(socketChannel.finishConnect())
```

```
1      {
2          int i=0;
3          while(true)
4          {
5              TimeUnit.SECONDS.sleep(1);
6              String info = "I'm "+i+++"-th information fro
7              m client";
8              buffer.clear();
9              buffer.put(info.getBytes());
10             buffer.flip();
11             while(buffer.hasRemaining()){
12                 System.out.println(buffer);
13                 socketChannel.write(buffer);
14             }
15         }
16     }
17     catch (IOException | InterruptedException e)
18     {
19         e.printStackTrace();
20     }
21     finally{
22         try{
23             if(socketChannel!=null){
24                 socketChannel.close();
25             }
26         }catch(IOException e){
27             e.printStackTrace();
28         }
29     }
30 }
```

公众号：方志朋

服务端代码（案例4）：

```
1  public static void server(){
2      ServerSocket serverSocket = null;
3      InputStream in = null;
4      try
```

```
5
6         serverSocket = new ServerSocket(8080);
7         int recvMsgSize = 0;
8         byte[] recvBuf = new byte[1024];
9         while(true){
10             Socket clntSocket = serverSocket.accept();
11             SocketAddress clientAddress = clntSocket.getRemoteSocketAddress();
12             System.out.println("Handling client at "+clientAddress);
13             in = clntSocket.getInputStream();
14             while((recvMsgSize=in.read(recvBuf)) != -1){
15                 byte[] temp = new byte[recvMsgSize];
16                 System.arraycopy(recvBuf, 0, temp, 0, recvMsgSize);
17                 System.out.println(new String(temp));
18             }
19         }
20     }
21     catch (IOException e)
22     {
23         e.printStackTrace();
24     }
25     https://github.com/Netflix/Hystrix/wiki/images/soa-5-isolation-focused-640.png
26     finally{
27         try{
28             if(serverSocket!=null){
29                 serverSocket.close();
30             }
31             if(in!=null){
32                 in.close();
33             }
34         }catch(IOException e){
35             e.printStackTrace();
36         }
37     }
}
```

公众号：方志朋

输出结果：（略）

根据案例分析，总结一下SocketChannel的用法。

打开SocketChannel：

```
1 socketChannel = SocketChannel.open();
2 socketChannel.connect(new InetSocketAddress("10.10.195.115", 8080)
);
```

关闭：

```
1
2 socketChannel.close();
```

读取数据：

```
1     String info = "I'm "+i+++"-th information from client";
2     buffer.clear();
3     buffer.put(info.getBytes());
4     buffer.flip();
5     while(buffer.hasRemaining()){
6         System.out.println(buffer);
7         socketChannel.write(buffer);
8 }
```

注意SocketChannel.write()方法的调用是在一个while循环中的。Write()方法无法保证能写多少字节到SocketChannel。所以，我们重复调用write()直到Buffer没有要写的字节为止。

非阻塞模式下,read()方法在尚未读取到任何数据时可能就返回了。所以需要关注它的int返回值，它会告诉你读取了多少字节。

TCP服务端的NIO写法

到目前为止，所举的案例中都没有涉及Selector。不要急，好东西要慢慢来。Selector类可以用于避免使用阻塞式客户端中很浪费资源的“忙等”方法。例如，考虑一个IM服务器。像QQ或者旺旺这样的，可能有几万甚至几千万个客户端同时连接到了服务器，但在任何时刻都只是非常少量的消息。

还有超过100本优质java高清电子书，添加微信cxymysq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

需要读取和分发。这就需要一种方法阻塞等待，直到至少有一个信道可以进行I/O操作，并指出是哪个信道。NIO的选择器就实现了这样的功能。一个Selector实例可以同时检查一组信道的I/O状态。用专业术语来说，选择器就是一个多路开关选择器，因为一个选择器能够管理多个信道上的I/O操作。然而如果用传统的方式来处理这么多客户端，使用的方法是循环地一个一个地去检查所有的客户端是否有I/O操作，如果当前客户端有I/O操作，则可能把当前客户端扔给一个线程池去处理，如果没有I/O操作则进行下一个轮询，当所有的客户端都轮询过了又接着从头开始轮询；这种方法是非常笨而且也非常浪费资源，因为大部分客户端是没有I/O操作，我们也要去检查；而Selector就不一样了，它在内部可以同时管理多个I/O，当一个信道有I/O操作的时候，他会通知Selector，Selector就是记住这个信道有I/O操作，并且知道是何种I/O操作，是读呢？是写呢？还是接受新的连接；所以如果使用Selector，它返回的结果只有两种结果，一种是0，即在你调用的时刻没有任何客户端需要I/O操作，另一种结果是一组需要I/O操作的客户端，这时你就根本不需要再检查了，因为它返回给你的肯定是你想要的。这样一种通知的方式比那种主动轮询的方式要高效得多！

要使用选择器（Selector），需要创建一个Selector实例（使用静态工厂方法open()）并将其注册（register）到想要监控的信道上（注意，这要通过channel的方法实现，而不是使用selector的方法）。最后，调用选择器的select()方法。该方法会阻塞等待，直到有一个或更多的信道准备好了I/O操作或等待超时。select()方法将返回可进行I/O操作的信道数量。现在，在一个单独的线程中，通过调用select()方法就能检查多个信道是否准备好进行I/O操作。如果经过一段时间后仍然没有信道准备好，select()方法就会返回0，并允许程序继续执行其他任务。

下面将上面的TCP服务端代码改写成NIO的方式（案例5）：

```
1 public class ServerConnect
2 {
3     private static final int BUF_SIZE=1024;
4     private static final int PORT = 8080;
5     private static final int TIMEOUT = 3000;
6
7     public static void main(String[] args)
8     {
9         selector();
10    }
11
12     public static void handleAccept(SelectionKey key) throws IOException{
13         ServerSocketChannel ssChannel = (ServerSocketChannel)key.channel();
14         SocketChannel sc = ssChannel.accept();
15         sc.configureBlocking(false);
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
16         sc.register(key.selector(), SelectionKey.OP_READ, ByteBuff
er.allocateDirect(BUF_SIZE));
17     }
18
19     public static void handleRead(SelectionKey key) throws IOException{
20         SocketChannel sc = (SocketChannel)key.channel();
21         ByteBuffer buf = (ByteBuffer)key.attachment();
22         long bytesRead = sc.read(buf);
23         while(bytesRead>0){
24             buf.flip();
25             while(buf.hasRemaining()){
26                 System.out.print((char)buf.get());
27             }
28             System.out.println();
29             buf.clear();
30             bytesRead = sc.read(buf);
31         }
32         if(bytesRead == -1){
33             sc.close();
34         }
35     }
36
37     public static void handleWrite(SelectionKey key) throws IOException{
38         ByteBuffer buf = (ByteBuffer)key.attachment();
39         buf.flip();
40         SocketChannel sc = (SocketChannel) key.channel();
41         while(buf.hasRemaining()){
42             sc.write(buf);
43         }
44         buf.compact();
45     }
46
47     public static void selector() {
48         Selector selector = null;
49         ServerSocketChannel ssc = null;
50         try{
51             selector = Selector.open();
52             ssc= ServerSocketChannel.open();
```

```
53         ssc.socket().bind(new InetSocketAddress(PORT));
54         ssc.configureBlocking(false);
55         ssc.register(selector, SelectionKey.OP_ACCEPT);
56
57     while(true){
58         if(selector.select(TIMEOUT) == 0){
59             System.out.println("==");
60             continue;
61         }
62         Iterator<SelectionKey> iter = selector.selectedKeys().iterator();
63         while(iter.hasNext()){
64             SelectionKey key = iter.next();
65             if(key.isAcceptable()){
66                 handleAccept(key);
67             }
68             if(key.isReadable()){
69                 handleRead(key);
70             }
71             if(key.isWritable() && key.isValid()){
72                 handleWrite(key);
73             }
74             if(key.isConnectable()){
75                 System.out.println("isConnectable = true");
76             }
77             iter.remove();
78         }
79     }
80
81 }catch(IOException e){
82     e.printStackTrace();
83 }finally{
84     try{
85         if(selector!=null){
86             selector.close();
87         }
88         if(ssc!=null){
89             ssc.close();
90         }
91     }
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
91         }catch(IOException e){  
92             e.printStackTrace();  
93         }  
94     }  
95 }  
96 }
```

下面来慢慢讲解这段代码。

ServerSocketChannel

打开ServerSocketChannel：

```
1 ServerSocketChannel serverSocketChannel = ServerSocketChannel.open()  
();
```

关闭ServerSocketChannel：

```
1 serverSocketChannel.close();
```

监听新进来的连接：

```
1 while(true){  
2     SocketChannel socketChannel = serverSocketChannel.accept();  
3 }
```

ServerSocketChannel可以设置成非阻塞模式。在非阻塞模式下，accept()方法会立刻返回，如果还没有新进来的连接，返回的将是null。因此，需要检查返回的SocketChannel是否是null.如：

```
1     ServerSocketChannel serverSocketChannel = ServerSocketCha  
nnel.open();  
2     serverSocketChannel.socket().bind(new InetSocketAddress(9  
999));
```

```
3     serverSocketChannel.configureBlocking(false);
4     while (true)
5     {
6         SocketChannel socketChannel = serverSocketChannel.accept();
7         if (socketChannel != null)
8         {
9             // do something with socketChannel...
10        }
11    }
```

Selector

Selector的创建： Selector selector = Selector.open();

为了将Channel和Selector配合使用，必须将Channel注册到Selector上，通过 SelectableChannel.register()方法来实现，沿用案例5中的部分代码：

```
1     ssc= ServerSocketChannel.open();
2     ssc.socket().bind(new InetSocketAddress(PORT));
3     ssc.configureBlocking(false);
4     ssc.register(selector, SelectionKey.OP_ACCEPT);
```

与Selector一起使用时， Channel必须处于非阻塞模式下。这意味着不能将FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式。而套接字通道都可以。

注意register()方法的第二个参数。这是一个“interest集合”，意思是在通过Selector监听Channel时对什么事件感兴趣。可以监听四种不同类型的事件：

- Connect
- Accept
- Read
- Write

通道触发了一个事件意思是该事件已经就绪。所以，某个channel成功连接到另一个服务器称为“连接就绪”。一个server socket channel准备好接收新进入的连接称为“接收就绪”。一个有数据可读的通道可以说是“读就绪”。等待写数据的通道可以说是“写就绪”。

这四种事件用SelectionKey的四个常量来表示：

- SelectionKey.OP_CONNECT
- SelectionKey.OP_ACCEPT
- SelectionKey.OP_READ
- SelectionKey.OP_WRITE

SelectionKey

当向Selector注册Channel时，register()方法会返回一个SelectionKey对象。这个对象包含了一些你感兴趣的属性：

- interest集合
- ready集合
- Channel
- Selector
- 附加的对象（可选）

interest集合：就像向Selector注册通道一节中所描述的，interest集合是你所选择的感兴趣的事件集合。可以通过SelectionKey读写interest集合。

ready 集合是通道已经准备就绪的操作的集合。在一次选择(Selection)之后，你会首先访问这个ready set。Selection将在下一小节进行解释。可以这样访问ready集合：

```
1 int readySet = selectionKey.readyOps();
```

可以用像检测interest那样的方法，来检测channel中什么事件或操作已经就绪。但是，也可以使用以下四个方法，它们都会返回一个布尔类型：

```
1 selectionKey.isAcceptable();
2 selectionKey.isConnectable();
3 selectionKey.isReadable();
4 selectionKey.isWritable();
```

从SelectionKey访问Channel和Selector很简单。如下：

```
1 Channel channel = selectionKey.channel();
2 Selector selector = selectionKey.selector();
```

可以将一个对象或者更多信息附着到SelectionKey上，这样就能方便的识别某个给定的通道。例如，可以附加与通道一起使用的Buffer，或是包含聚集数据的某个对象。使用方法如下：

```
1 selectionKey.attach(theObject);
2 Object attachedObj = selectionKey.attachment();
```

还可以在用register()方法向Selector注册Channel的时候附加对象。如：

```
1 SelectionKey key = channel.register(selector, SelectionKey.OP_READ
, theObject);
```

通过Selector选择通道

一旦向Selector注册了一或多个通道，就可以调用几个重载的select()方法。这些方法返回你所感兴趣的事情（如连接、接受、读或写）已经准备就绪的那些通道。换句话说，如果你对“读就绪”的通道感兴趣，select()方法会返回读事件已经就绪的那些通道。

下面是select()方法：

- int select()
- int select(long timeout)
- int selectNow()
- select()阻塞到至少有一个通道在你注册的事件上就绪了。
- select(long timeout)和select()一样，除了最长会阻塞timeout毫秒(参数)。
- selectNow()不会阻塞，不管什么通道就绪都立刻返回（译者注：此方法执行非阻塞的选择操作。如果自从前一次选择操作后，没有通道变成可选择的，则此方法直接返回零。）。

select()方法返回的int值表示有多少通道已经就绪。亦即，自上次调用select()方法后有多少通道变成就绪状态。如果调用select()方法，因为有一个通道变成就绪状态，返回了1，若再次调用select()方法，如果另一个通道就绪了，它会再次返回1。如果对第一个就绪的channel没有做任何操作，现在就有两个就绪的通道，但在每次select()方法调用之间，只有一个通道就绪了。

一旦调用了select()方法，并且返回值表明有一个或更多个通道就绪了，然后可以通过调用selector的selectedKeys()方法，访问“已选择键集 (selected key set) ”中的就绪通道。如下所示：

```
1 Set selectedKeys = selector.selectedKeys();
```

当向Selector注册Channel时，Channel.register()方法会返回一个SelectionKey 对象。这个对象代表了注册到该Selector的通道。

注意每次迭代末尾的keyIterator.remove()调用。Selector不会自己从已选择键集中移除SelectionKey实例。必须在处理完通道时自己移除。下次该通道变成就绪时，Selector会再次将其放入已选择键集中。

SelectionKey.channel()方法返回的通道需要转型成你要处理的类型，如ServerSocketChannel或SocketChannel等。

一个完整的使用Selector和ServerSocketChannel的案例可以参考案例5的selector()方法。

原文链接

<https://blog.csdn.net/u013256816/article/details/51457215#comments>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

公众号：方志朋

Java基础： Java线程基础

Java线程基础

操作系统中线程和进程的概念

现在的操作系统是多任务操作系统。多线程是实现多任务的一种方式。

进程是指一个内存中运行的应用程序，每个进程都有自己独立的一块内存空间，一个进程中可以启动多个线程。比如在Windows系统中，一个运行的exe就是一个进程。

线程是指进程中的一个执行流程，一个进程中可以运行多个线程。比如java.exe进程中可以运行很多线程。线程总是属于某个进程，进程中的多个线程共享进程的内存。

在java中要想实现多线程，有两种手段，一种是继承Thread类，另外一种是实现Runnable接口。(其实准确来讲，应该有三种，还有一种是实现Callable接口，并与Future、线程池结合使用，此文这里不讲这个。

Java线程的实现形式

这里继承Thread类的方法是比较常用的一种，如果说你只是想起一条线程。没有什么其它特殊的要求，那么可以使用Thread。

继承Thread类

扩展Thread类实现的多线程例子：

```
1
2 /**
3 * 测试扩展Thread类实现的多线程程序
4 */
5 public class TestThread extends Thread {
6     public TestThread(String name){
7         super(name);
8     }
9     @Override
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
10    public void run() {
11        for(int i=0;i<5;i++){
12            for(long k=0;k<1000000000;k++);
13            System.out.println(this.getName()+"："+i);
14        }
15    }
16    public static void main(String[] args){
17        Thread t1=new TestThread("李白");
18        Thread t2=new TestThread("屈原");
19        t1.start();
20        t2.start();
21    }
22 }
```

执行结果：

```
屈原:0
李白:0
屈原:1
李白:1
屈原:2
李白:2
屈原:3
屈原:4
李白:3
李白:4
```

公众号：方志朋

实现Runnable接口

```
1
2 public class RunnableImpl implements Runnable{
3     private Stringname;
4     public RunnableImpl(String name) {
5         this.name = name;
6     }
7     @Override
8     public void run() {
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
9      for (int i = 0; i < 5; i++) {  
10         for(long k=0;k<1000000000;k++);  
11         System.out.println(name+":"+i);  
12     }  
13 }  
14 }  
15  
16 /**  
17 * 测试Runnable类实现的多线程程序  
18 */  
19 public class TestRunnable {  
20  
21     public static void main(String[] args) {  
22         RunnableImpl ri1=new RunnableImpl("李白");  
23         RunnableImpl ri2=new RunnableImpl("屈原");  
24         Thread t1=new Thread(ri1);  
25         Thread t2=new Thread(ri2);  
26         t1.start();  
27         t2.start();  
28     }  
29 }
```

公众号：方志朋

输出：

```
1 C运行 : 0  
2 D运行 : 0  
3 D运行 : 1  
4 C运行 : 1  
5 D运行 : 2  
6 C运行 : 2  
7 D运行 : 3  
8 C运行 : 3  
9 D运行 : 4  
10 C运行 : 4
```

说明：

Thread2类通过实现Runnable接口，使得该类有了多线程类的特征。run () 方法是多线程程序的一个约

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

定。所有的多线程代码都在run方法里面。Thread类实际上也是实现了Runnable接口的类。

在启动的多线程的时候，需要先通过Thread类的构造方法Thread(Runnable target) 构造出对象，然后调用Thread对象的start()方法来运行多线程代码。

实际上所有的多线程代码都是通过运行Thread的start()方法来运行的。因此，不管是扩展Thread类还是实现Runnable接口来实现多线程，最终还是通过Thread的对象的API来控制线程的，熟悉Thread类的API是进行多线程编程的基础。

使用Callable和Future接口创建线程。

具体是创建Callable接口的实现类，并实现call()方法。并使用FutureTask类来包装Callable实现类的对象，且以此FutureTask对象作为Thread对象的target来创建线程。

```
1 1 public class ThreadTest {  
2 2  
3 3     public static void main(String[] args) {  
4 4  
5 5         Callable<Integer> myCallable = new MyCallable();      //  
   创建MyCallable对象  
6 6         FutureTask<Integer> ft = new FutureTask<Integer>(myCal  
   lable); //使用FutureTask来包装MyCallable对象  
7 7  
8 8         for (int i = 0; i < 100; i++) {  
9 9             System.out.println(Thread.currentThread().getName()  
 ) + " " + i);  
10 10        if (i == 30) {  
11 11            Thread thread = new Thread(ft);    //FutureTask  
   对象作为Thread对象的target创建新的线程  
12 12            thread.start();                  //线程进入  
   到就绪状态  
13 13        }  
14 14    }  
15 15  
16 16    System.out.println("主线程for循环执行完毕..");  
17 17  
18 18    try {  
19 19        int sum = ft.get();                //取得新创建的新线程中  
   的call()方法返回的结果  
20 20    System.out.println("sum = " + sum);
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
21 21         } catch (InterruptedException e) {
22 22             e.printStackTrace();
23 23         } catch (ExecutionException e) {
24 24             e.printStackTrace();
25 25     }
26 26
27 27 }
28 28 }
29 29
30 30
31 31 class MyCallable implements Callable<Integer> {
32 32     private int i = 0;
33 33
34 34     // 与run()方法不同的是，call()方法具有返回值
35 35     @Override
36 36     public Integer call() {
37 37         int sum = 0;
38 38         for (; i < 100; i++) {
39 39             System.out.println(Thread.currentThread().getName(
40 40                 ) + " " + i);
41 41             sum += i;
42 42         }
43 43         return sum;
44 44     }
```

公众号：方志朋

首先，我们发现，在实现Callable接口中，此时不再是run()方法了，而是call()方法，此call()方法作为线程执行体，同时还具有返回值！在创建新的线程时，是通过FutureTask来包装MyCallable对象，同时作为为了Thread对象的target。那么看下FutureTask类的定义：

```
1 1 public class FutureTask<V> implements RunnableFuture<V> {
2 2
3 3     //....
4 4
5 5 }
```

```
1 1 public interface RunnableFuture<V> extends Runnable, Future<V> {
```

```
2 2
3 3     void run();
4 4
5 5 }
```

于是，我们发现FutureTask类实际上是同时实现了Runnable和Future接口，由此才使得其具有Future和Runnable双重特性。通过Runnable特性，可以作为Thread对象的target，而Future特性，使得其可以取得新创建线程中的call()方法的返回值。

执行下此程序，我们发现sum = 4950永远都是最后输出的。而“主线程for循环执行完毕..”则很可能是在子线程循环中间输出。由CPU的线程调度机制，我们知道，“主线程for循环执行完毕..”的输出时机是没有任何问题的，那么为什么sum =4950会永远最后输出呢？

原因在于通过ft.get()方法获取子线程call()方法的返回值时，当子线程此方法还未执行完毕，ft.get()方法会一直阻塞，直到call()方法执行完毕才能取到返回值。

main方法其实也是一个线程。在java中所有的线程都是同时启动的，至于什么时候，哪个先执行，完全看谁先得到CPU的资源。

在java中，每次程序运行至少启动2个线程。一个是main线程，一个是垃圾收集线程。因为每当使用java命令执行一个类的时候，实际上都会启动一个JVM，每一个JVM实习在就是在操作系统中启动了一个进程。

链接

<https://blog.csdn.net/kwame211/article/details/78963044>

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

<http://www.cnblogs.com/lwbqqyumi/p/3804883.html>

<https://www.cnblogs.com/lwbqqyumi/p/3817517.html>

<https://blog.csdn.net/Evankaka/article/details/44153709>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：

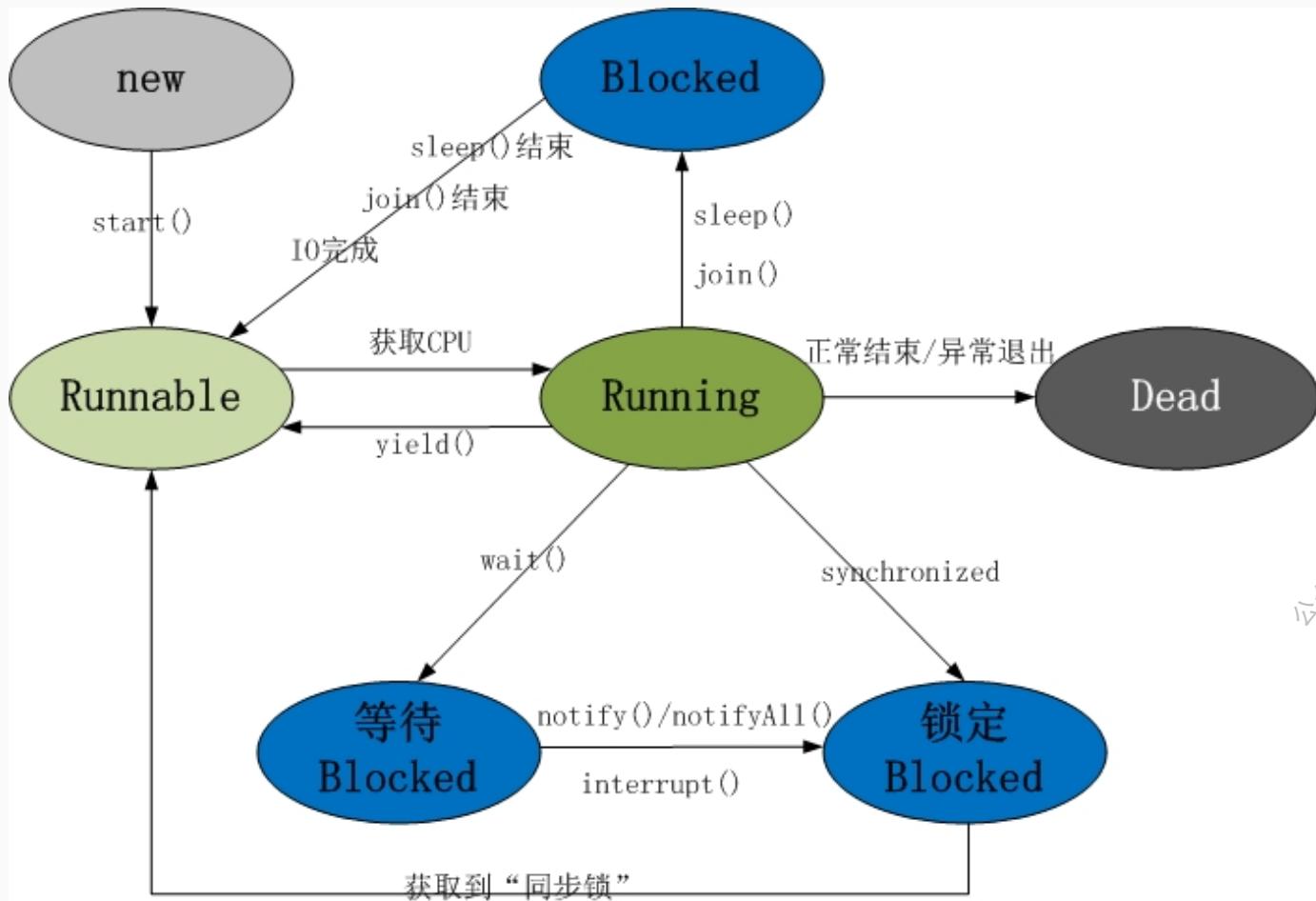


公众号：方志朋

Java基础：java线程状态

java线程状态

线程的生命周期及五种基本状态



上图中基本上囊括了Java中多线程各重要知识点。掌握了上图中的各知识点，Java中的多线程也就基本上掌握了。主要包括：

Java线程具有五中基本状态

新建状态 (New)：当线程对象对创建后，即进入了新建状态，如：`Thread t = new MyThread();`

就绪状态 (Runnable)：当调用线程对象的`start()`方法 (`t.start();`)，线程即进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待CPU调度执行，并不是说执行了`t.start()`此线程立即就会执行；

运行状态 (Running)：当CPU开始调度处于就绪状态的线程时，此时线程才得以真正执行，即进入到运行状态。注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

阻塞状态 (Blocked)：处于运行状态中的线程由于某种原因，暂时放弃对CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才有机会再次被CPU调用以进入到运行状态。根据阻塞产生的原因不同，阻塞状态又可以分为三种：

- 1.等待阻塞：运行状态中的线程执行wait()方法，使本线程进入到等待阻塞状态；
- 2.同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态；
- 3.其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

死亡状态 (Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

就绪状态转换为运行状态：当此线程得到处理器资源；

运行状态转换为就绪状态：当此线程主动调用yield()方法或在运行过程中失去处理器资源。

运行状态转换为死亡状态：当此线程线程执行体执行完毕或发生了异常。

此处需要特别注意的是：当调用线程的yield()方法时，线程从运行状态转换为就绪状态，但接下来CPU调度就绪状态中的哪个线程具有一定的随机性，因此，可能会出现A线程调用了yield()方法后，接下来CPU仍然调度了A线程的情况。

常用的线程函数

sleep(long millis)

在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）

join():指等待t线程终止

join是Thread类的一个方法，启动线程后直接调用，即join()的作用是：“等待该线程终止”，这里需要理解的就是该线程是指的主线程等待子线程的终止。也就是在子线程调用了join()方法后面的代码，只有等到子

线程结束了才能执行。

在很多情况下，主线程生成并启动了子线程，如果子线程里要进行大量的耗时的运算，主线程往往将于子线程之前结束，但是如果主线程处理完其他的事务后，需要用到子线程的处理结果，也就是主线程需要等待子线程执行完成之后再结束，这个时候就要用到join()方法了。

不加join:

```
1 class Thread1 extends Thread{  
2     private String name;  
3     public Thread1(String name) {  
4         super(name);  
5         this.name=name;  
6     }  
7     public void run() {  
8         System.out.println(Thread.currentThread().getName() + "  
线程运行开始!");  
9         for (int i = 0; i < 5; i++) {  
10             System.out.println("子线程"+name + "运行 : " + i);  
11             try {  
12                 sleep((int) Math.random() * 10);  
13             } catch (InterruptedException e) {  
14                 e.printStackTrace();  
15             }  
16         }  
17         System.out.println(Thread.currentThread().getName() + "  
线程运行结束!");  
18     }  
19 }  
20  
21 public class Main {  
22  
23     public static void main(String[] args) {  
24         System.out.println(Thread.currentThread().getName()+"主线  
程运行开始!");  
25         Thread1 mTh1=new Thread1("A");  
26         Thread1 mTh2=new Thread1("B");  
27         mTh1.start();  
28         mTh2.start();
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
29         System.out.println(Thread.currentThread().getName()+"主线  
30             程运行结束!");  
31     }
```

输出结果：

```
1 main主线程运行开始!  
2 main主线程运行结束!  
3 B 线程运行开始!  
4 子线程B运行 : 0  
5 A 线程运行开始!  
6 子线程A运行 : 0  
7 子线程B运行 : 1  
8 子线程A运行 : 1  
9 子线程A运行 : 2  
10 子线程A运行 : 3  
11 子线程A运行 : 4  
12 A 线程运行结束!  
13 子线程B运行 : 2  
14 子线程B运行 : 3  
15 子线程B运行 : 4  
16 B 线程运行结束!  
17 发现主线程比子线程早结束
```

加join:

```
1  
2 public class Main {  
3  
4     public static void main(String[] args) {  
5         System.out.println(Thread.currentThread().getName()+"主线  
程运行开始!");  
6         Thread1 mTh1=new Thread1("A");  
7         Thread1 mTh2=new Thread1("B");  
8         mTh1.start();  
9         mTh2.start();
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
10     try {
11         mTh1.join();
12     } catch (InterruptedException e) {
13         e.printStackTrace();
14     }
15     try {
16         mTh2.join();
17     } catch (InterruptedException e) {
18         e.printStackTrace();
19     }
20     System.out.println(Thread.currentThread().getName() + "主线
程运行结束!");
21
22 }
23 }
```

运行结果：

```
1 main主线程运行开始!
2 A 线程运行开始!
3 子线程A运行 : 0
4 B 线程运行开始!
5 子线程B运行 : 0
6 子线程A运行 : 1
7 子线程B运行 : 1
8 子线程A运行 : 2
9 子线程B运行 : 2
10 子线程A运行 : 3
11 子线程B运行 : 3
12 子线程A运行 : 4
13 子线程B运行 : 4
14 A 线程运行结束!
15 主线程一定会等子线程都结束了才结束
```

yield():暂停当前正在执行的线程对象，并执行其他线程。

hread.yield()方法作用是：暂停当前正在执行的线程对象，并执行其他线程。

yield()应该做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用yield()的目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。

结论：yield()从未导致线程转到等待/睡眠/阻塞状态。在大多数情况下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。可看上面的图。

```
1
2
3 class ThreadYield extends Thread{
4     public ThreadYield(String name) {
5         super(name);
6     }
7
8     @Override
9     public void run() {
10        for (int i = 1; i <= 50; i++) {
11            System.out.println(">" + this.getName() + "-----" + i)
12        ;
13        // 当i为30时，该线程就会把CPU时间让掉，让其他或者自己的线程执行
14        // (也就是谁先抢到谁执行)
15        if (i ==30) {
16            this.yield();
17        }
18    }
19 }
20
21 public class Main {
22
23     public static void main(String[] args) {
24
25         ThreadYield yt1 = new ThreadYield("张三");
26         ThreadYield yt2 = new ThreadYield("李四");
27         yt1.start();
28         yt2.start();
29     }
}
```

公众号：方志朋

```
30  
31 }
```

运行结果：

第一种情况：李四（线程）当执行到30时会CPU时间让掉，这时张三（线程）抢到CPU时间并执行。

第二种情况：李四（线程）当执行到30时会CPU时间让掉，这时李四（线程）抢到CPU时间并执行。

sleep()和yield()的区别

sleep()和yield()的区别: sleep()使当前线程进入停滞状态，所以执行sleep()的线程在指定的时间内肯定不会被执行；yield()只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行。

sleep方法使当前运行中的线程睡眠一段时间，进入不可运行状态，这段时间的长短是由程序设定的，yield方法使当前线程让出CPU占有权，但让出的时间是不可设定的。实际上，yield()方法对应了如下操作：先检测当前是否有相同优先级的线程处于同可运行状态，如有，则把CPU的占有权交给此线程，否则，继续运行原来的线程。所以yield()方法称为“退让”，它把运行机会让给了同等优先级的其他线程

另外，sleep方法允许较低优先级的线程获得运行机会，但yield()方法执行时，当前线程仍处在可运行状态，所以，不可能让出较低优先级的线程些时获得CPU占有权。在一个运行系统中，如果较高优先级的线程没有调用sleep方法，又没有受到I/O阻塞，那么，较低优先级线程只能等待所有较高优先级的线程运行结束，才有机会运行。

setPriority(): 更改线程的优先级。

- MIN_PRIORITY = 1
- NORM_PRIORITY = 5
- MAX_PRIORITY = 10

用法：

```
1 Thread4 t1 = new Thread4("t1");  
2 Thread4 t2 = new Thread4("t2");  
3 t1.setPriority(Thread.MAX_PRIORITY);  
4 t2.setPriority(Thread.MIN_PRIORITY);
```

interrupt():

不要以为它是中断某个线程！它只是线程发送一个中断信号，让线程在无限等待时（如死锁时）能抛出抛出，从而结束线程，但是如果你吃掉了这个异常，那么这个线程还是不会中断的！

wait()

Obj.wait(), 与Obj.notify()必须要与synchronized(Obj)一起使用，也就是wait,与notify是针对已经获取了Obj锁进行操作，从语法角度来说就是Obj.wait(),Obj.notify必须在synchronized(Obj){...}语句块内。从功能上来说wait就是说线程在获取对象锁后，主动释放对象锁，同时本线程休眠。直到有其它线程调用对象的notify()唤醒该线程，才能继续获取对象锁，并继续执行。相应的notify()就是对对象锁的唤醒操作。但有一点需要注意的是notify()调用后，并不是马上就释放对象锁的，而是在相应的synchronized(){...}语句块执行结束，自动释放锁后，JVM会在wait()对象锁的线程中随机选取一线程，赋予其对象锁，唤醒线程，继续执行。这样就提供了在线程间同步、唤醒的操作。Thread.sleep()与Object.wait()二者都可以暂停当前线程，释放CPU控制权，主要的区别在于Object.wait()在释放CPU同时，释放了对象锁的控制。

单单在概念上理解清楚了还不够，需要在实际的例子中进行测试才能更好的理解。对Object.wait(), Object.notify()的应用最经典的例子，应该是三线程打印ABC的问题了吧，这是一道比较经典的面试题，题目要求如下：

建立三个线程，A线程打印10次A，B线程打印10次B,C线程打印10次C，要求线程同时运行，交替打印10次ABC。这个问题用Object的wait(), notify()就可以很方便的解决。代码如下：

```
1
2 public class MyThreadPrinter2 implements Runnable {
3
4     private String name;
5     private Object prev;
6     private Object self;
7
8     private MyThreadPrinter2(String name, Object prev, Object sel
9         f) {
10         this.name = name;
11         this.prev = prev;
12         this.self = self;
13     }
14 }
```

```
13
14     @Override
15     public void run() {
16         int count = 10;
17         while (count > 0) {
18             synchronized (prev) {
19                 synchronized (self) {
20                     System.out.print(name);
21                     count--;
22
23                     self.notify();
24                 }
25             try {
26                 prev.wait();
27             } catch (InterruptedException e) {
28                 e.printStackTrace();
29             }
30         }
31     }
32 }
33 }
34
35     public static void main(String[] args) throws Exception {
36         Object a = new Object();
37         Object b = new Object();
38         Object c = new Object();
39         MyThreadPrinter2 pa = new MyThreadPrinter2("A", c, a);
40         MyThreadPrinter2 pb = new MyThreadPrinter2("B", a, b);
41         MyThreadPrinter2 pc = new MyThreadPrinter2("C", b, c);
42
43
44         new Thread(pa).start();
45         Thread.sleep(100); //确保按顺序A、B、C执行
46         new Thread(pb).start();
47         Thread.sleep(100);
48         new Thread(pc).start();
49         Thread.sleep(100);
50     }
51 }
```

输出结果：

ABCABCABCABCABCABCABCABCABC

先来解释一下其整体思路，从大的方向上来讲，该问题为三线程间的同步唤醒操作，主要的目的就是 ThreadA->ThreadB->ThreadC->ThreadA循环执行三个线程。为了控制线程执行的顺序，那么就必须要确定唤醒、等待的顺序，所以每一个线程必须同时持有两个对象锁，才能继续执行。一个对象锁是 prev，就是前一个线程所持有的对象锁。还有一个就是自身对象锁。主要的思想就是，为了控制执行的顺序，必须要先持有prev锁，也就前一个线程要释放自身对象锁，再去申请自身对象锁，两者兼备时打印，之后首先调用self.notify()释放自身对象锁，唤醒下一个等待线程，再调用prev.wait()释放prev对象锁，终止当前线程，等待循环结束后再次被唤醒。运行上述代码，可以发现三个线程循环打印ABC，共10次。程序运行的主要过程就是A线程最先运行，持有C,A对象锁，后释放A,C锁，唤醒B。线程B等待A锁，再申请B锁，后打印B，再释放B，A锁，唤醒C，线程C等待B锁，再申请C锁，后打印C，再释放C,B锁，唤醒A。看起来似乎没什么问题，但如果你仔细想一下，就会发现有问题，就是初始条件，三个线程按照A,B,C的顺序来启动，按照前面的思考，A唤醒B，B唤醒C，C再唤醒A。但是这种假设依赖于JVM中线程调度、执行的顺序。

wait和sleep区别

共同点：

- - a. 他们都是在多线程的环境下，都可以在程序的调用处阻塞指定的毫秒数，并返回。
- - b. wait()和sleep()都可以通过interrupt()方法 打断线程的暂停状态，从而使线程立刻抛出 InterruptedException。

如果线程A希望立即结束线程B，则可以对线程B对应的Thread实例调用interrupt方法。如果此刻线程B正在wait/sleep /join，则线程B会立刻抛出InterruptedException，在catch() {} 中直接return即可安全地结束线程。

需要注意的是，InterruptedException是线程自己从内部抛出的，并不是interrupt()方法抛出的。对某一线程调用 interrupt()时，如果该线程正在执行普通的代码，那么该线程根本就不会抛出 InterruptedException。但是，一旦该线程进入到 wait()/sleep()/join()后，就会立刻抛出 InterruptedException。

不同点：

- - a. Thread类的方法：sleep(),yield()等
 - Object的方法：wait()和notify()等
-

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- b. 每个对象都有一个锁来控制同步访问。Synchronized关键字可以和对象的锁交互，来实现线程的同步。

sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

- c. wait, notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用

所以sleep()和wait()方法的最大区别是：

- msleep()睡眠时，保持对象锁，仍然占有该锁；
- 而wait()睡眠时，释放对象锁。
- 但是wait()和sleep()都可以通过interrupt()方法打断线程的暂停状态，从而使线程立刻抛出 InterruptedException（但不建议使用该方法）。

sleep () 方法

- sleep()使当前线程进入停滞状态（阻塞当前线程），让出CPU的使用、目的是不让当前线程独自霸占该进程所获的CPU资源，以留一定时间给其他线程执行的机会；
- sleep()是Thread类的Static(静态)的方法；因此他不能改变对象的机锁，所以当在一个Synchronized块中调用Sleep()方法时，线程虽然休眠了，但是对象的机锁并没有被释放，其他线程无法访问这个对象（即使睡着也持有对象锁）。
- 在sleep()休眠时间期满后，该线程不一定会立即执行，这是因为其它线程可能正在运行而且没有被调度为放弃执行，除非此线程具有更高的优先级。

wait () 方法

- wait()方法是Object类里的方法；当一个线程执行到wait()方法时，它就进入到一个和该对象相关的等待池中，同时失去（释放）了对象的机锁（暂时失去机锁，wait(long timeout)超时时间到后还需要返还对象锁）；其他线程可以访问；
- wait()使用notify或者notifyAll或者指定睡眠时间来唤醒当前等待池中的线程。
- wait()必须放在synchronized block中，否则会在program runtime时扔出”java.lang.IllegalMonitorStateException“异常。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

Java并发： AtomicInteger源码分析——基于CAS的乐观锁实现

AtomicInteger源码分析——基于CAS的乐观锁实现

悲观锁与乐观锁

我们都知道，cpu是时分复用的，也就是把cpu的时间片，分配给不同的thread/process轮流执行，时间片与时间片之间，需要进行cpu切换，也就是会发生进程的切换。切换涉及到清空寄存器，缓存数据。然后重新加载新的thread所需数据。当一个线程被挂起时，加入到阻塞队列，在一定的时间或条件下，在通过notify()，notifyAll()唤醒回来。在某个资源不可用的时候，就将cpu让出，把当前等待线程切换为阻塞状态。等到资源(比如一个共享数据) 可用了，那么就将线程唤醒，让他进入Runnable状态等待cpu调度。这就是典型的悲观锁的实现。独占锁是一种悲观锁，synchronized就是一种独占锁，它假设最坏的情况，并且只有在确保其它线程不会造成干扰的情况下执行，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。

但是，由于在进程挂起和恢复执行过程中存在着很大的开销。当一个线程正在等待锁时，它不能做任何事，所以悲观锁有很大的缺点。举个例子，如果一个线程需要某个资源，但是这个资源的占用时间很短，当线程第一次抢占这个资源时，可能这个资源被占用，如果此时挂起这个线程，可能立刻就发现资源可用，然后又需要花费很长的时间重新抢占锁，时间代价就会非常的高。

所以就有了乐观锁的概念，他的核心思路就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。在上面的例子中，某个线程可以不让出cpu,而是一直while循环，如果失败就重试，直到成功为止。所以，当数据争用不严重时，乐观锁效果更好。比如CAS就是一种乐观锁思想的应用。

java中CAS的实现

CAS就是Compare and Swap的意思，比较并操作。很多的cpu直接支持CAS指令。CAS是一项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

JDK1.5中引入了底层的支持，在int、long和对象的引用等类型上都公开了CAS的操作，并且JVM把它们编译为底层硬件提供的最有效的方法，在运行CAS的平台上，运行时把它们编译为相应的机器指令。在java.util.concurrent.atomic包下面的所有原子变量类型中，比如AtomicInteger，都使用了这些底层的JVM支持为数字类型的引用类型提供一种高效的CAS操作。

在CAS操作中，会出现ABA问题。就是如果V的值先由A变成B，再由B变成A，那么仍然认为是发生了变化，并需要重新执行算法中的步骤。有简单的解决方案：不是更新某个引用的值，而是更新两个值，包括一个引用和一个版本号，即使这个值由A变为B，然后变为A，版本号也是不同的。

AtomicStampedReference和AtomicMarkableReference支持在两个变量上执行原子的条件更新。

AtomicStampedReference更新一个“对象-引用”二元组，通过在引用上加上“版本号”，从而避免ABA问题，AtomicMarkableReference将更新一个“对象引用-布尔值”的二元组。

AtomicInteger的实现。

AtomicInteger是一个支持原子操作的Integer类，就是保证对AtomicInteger类型变量的增加和减少操作是原子性的，不会出现多个线程下的数据不一致问题。如果不使用AtomicInteger，要实现一个按顺序获取的ID，就必须在每次获取时进行加锁操作，以避免出现并发时获取到同样的ID的现象。

接下来通过源代码来看AtomicInteger具体是如何实现的原子操作。

首先看incrementAndGet()方法，下面是具体的代码。

```
1
2 public final int incrementAndGet() {
3     for (;;) {
4         int current = get();
5         int next = current + 1;
6         if (compareAndSet(current, next))
7             return next;
8     }
9 }
```

通过源码，可以知道，这个方法的做法为先获取到当前的value属性值，然后将value加1，赋值给一个局部的next变量，然而，这两步都是非线程安全的，但是内部有一个死循环，不断去做compareAndSet操作，直到成功为止，也就是修改的根本在compareAndSet方法里面，compareAndSet()方法的代码如下：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
1 public final boolean compareAndSet(int expect, int update) {  
2     return unsafe.compareAndSwapInt(this, valueOffset, expect,  
3                                     update);  
4 }
```

compareAndSet()方法调用的compareAndSwapInt()方法的声明如下，是一个native方法。

```
1 public final native boolean compareAndSwapInt(Object var1, long var  
2 , int var4, int var5);
```

compareAndSet 传入的为执行方法时获取到的 value 属性值，next 为加 1 后的值，compareAndSet 所做的为调用 Sun 的 UnSafe 的 compareAndSwapInt 方法来完成，此方法为 native 方法，compareAndSwapInt 基于的是CPU 的 CAS 指令来实现的。所以基于 CAS 的操作可认为是无阻塞的，一个线程的失败或挂起不会引起其它线程也失败或挂起。并且由于 CAS 操作是 CPU 原语，所以性能比较好。

类似的，还有decrementAndGet()方法。它和incrementAndGet()的区别是将 value 减 1，赋值给next 变量。

AtomicInteger中还有getAndIncrement() 和getAndDecrement() 方法，他们的实现原理和上面的两个方法完全相同，区别是返回值不同，前两个方法返回的是改变之后的值，即next。而这两个方法返回的是改变之前的值，即current。还有很多其他的其他方法，就不列举了。

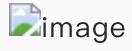
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

程序员理财，请关注：



公众号：方志朋

Java并发：BlockingQueue解读

BlockingQueue解读

在新增的Concurrent包中，BlockingQueue很好的解决了多线程中，如何高效安全“传输”数据的问题。通过这些高效并且线程安全的队列类，为我们快速搭建高质量的多线程程序带来极大的便利。本文详细介绍了BlockingQueue家庭中的所有成员，包括他们各自的功能以及常见使用场景。

认识BlockingQueue

首先，最基本的来说，BlockingQueue是一个先进先出的队列（Queue），为什么说是阻塞（Blocking）的呢？是因为 BlockingQueue支持当获取队列元素但是队列为空时，会阻塞等待队列中有元素再返回；也支持添加元素时，如果队列已满，那么等到队列可以放入新元素时再放入。

BlockingQueue是一个接口，继承自 Queue，所以其实现类也可以作为 Queue 的实现来使用，而 Queue 又继承自 Collection 接口。

BlockingQueue对插入操作、移除操作、获取元素操作提供了四种不同的方法用于不同的场景中使用：
1、抛出异常；2、返回特殊值（null 或 true/false，取决于具体的操作）；3、阻塞等待此操作，直到这个操作成功；4、阻塞等待此操作，直到成功或者超时指定时间。总结如下：

action	Throws exception	Special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	not applicable	not applicable

lockingQueue的各个实现都遵循了这些规则，当然我们也不用死记这个表格，知道有这么回事，然后写代码的时候根据自己的需要去看方法的注释来选取合适的方法即可。

对于 BlockingQueue，我们的关注点应该在 put(e) 和 take() 这两个方法，因为这两个方法是带阻塞的。

BlockingQueue不接受 null 值的插入，相应的方法在碰到 null 的插入时会抛出 NullPointerException 异常。null 值在这里通常用于作为特殊值返回（表格中的第三列），代表 poll 失败。所以，如果允许插入 null 值的话，那获取的时候，就不能很好地用 null 来判断到底是代表失败，还是获取的值就是 null 值。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

一个 BlockingQueue 可能是有界的，如果在插入的时候，发现队列满了，那么 put 操作将会阻塞。通常，在这里我们说的无界队列也不是说真正的无界，而是它的容量是 Integer.MAX_VALUE（21亿多）。

BlockingQueue 是设计用来实现生产者-消费者队列的，当然，你也可以将它当做普通的 Collection 来用，前面说了，它实现了 java.util.Collection 接口。例如，我们可以用 remove(x) 来删除任意一个元素，但是，这类操作通常并不高效，所以尽量只在少数的场合使用，比如一条消息已经入队，但是需要做取消操作的时候。

BlockingQueue 的实现都是线程安全的，但是批量的集合操作如 addAll, containsAll, retainAll 和 removeAll 不一定是原子操作。如 addAll(c) 有可能在添加了一些元素后中途抛出异常，此时 BlockingQueue 中已经添加了部分元素，这个是允许的，取决于具体的实现。

BlockingQueue 不支持 close 或 shutdown 等关闭操作，因为开发者可能希望不会有新的元素添加进去，此特性取决于具体的实现，不做强制约束。

最后，BlockingQueue 在生产者-消费者的场景中，是支持多消费者和多生产者的，说的其实就是线程安全问题。

相信上面说的每一句都很清楚了，BlockingQueue 是一个比较简单的线程安全容器，下面我会分析其具体的在 JDK 中的实现，这里又到了 Doug Lea 表演时间了。

公众号：方志朋

BlockingQueue成员详细介绍

ArrayBlockingQueue

基于数组的阻塞队列实现，在ArrayBlockingQueue内部，维护了一个定长数组，以便缓存队列中的数据对象，这是一个常用的阻塞队列，除了一个定长数组外，ArrayBlockingQueue内部还保存着两个整形变量，分别标识着队列的头部和尾部在数组中的位置。

ArrayBlockingQueue在生产者放入数据和消费者获取数据，都是共用同一个锁对象，由此也意味着两者无法真正并行运行，这点尤其不同于LinkedBlockingQueue；按照实现原理来分析，ArrayBlockingQueue完全可以采用分离锁，从而实现生产者和消费者操作的完全并行运行。Doug Lea之所以没这样去做，也许是因为ArrayBlockingQueue的数据写入和获取操作已经足够轻巧，以至于引入独立的锁机制，除了给代码带来额外的复杂性外，其在性能上完全占不到任何便宜。ArrayBlockingQueue和LinkedBlockingQueue间还有一个明显的不同之处在于，前者在插入或删除元素时不会产生或销毁任何额外的对象实例，而后者则会生成一个额外的Node对象。这在长时间内需要高效并发地处理大批量数据的系统中，其对于GC的影响还是存在一定的区别。而在创建ArrayBlockingQueue时，我们还可以控制对象的内部锁是否采用公平锁，默认采用非公平锁。

LinkedBlockingQueue

基于链表的阻塞队列，同ArrayListBlockingQueue类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成），当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到最大值缓存容量时（LinkedBlockingQueue可以通过构造函数指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒，反之对于消费者这端的处理也基于同样的原理。而LinkedBlockingQueue之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

作为开发者，我们需要注意的是，如果构造一个LinkedBlockingQueue对象，而没有指定其容量大小，LinkedBlockingQueue会默认一个类似无限大小的容量（Integer.MAX_VALUE），这样的话，如果生产者的速度一旦大于消费者的 speed，也许还没有等到队列满阻塞产生，系统内存就有可能已被消耗殆尽了。

ArrayBlockingQueue和LinkedBlockingQueue是两个最普通也是最常用的阻塞队列，一般情况下，在处理多线程间的生产者消费者问题，使用这两个类足以。

下面的代码演示了如何使用BlockingQueue：

```
1 public class BlockingQueueTest {  
2  
3     public static void main(String[] args) throws InterruptedException {  
4         // 声明一个容量为10的缓存队列  
5         BlockingQueue<String> queue = new LinkedBlockingQueue<String>(10);  
6  
7         Producer producer1 = new Producer(queue);  
8         Producer producer2 = new Producer(queue);  
9         Producer producer3 = new Producer(queue);  
10        Consumer consumer = new Consumer(queue);  
11  
12        // 借助Executors  
13        ExecutorService service = Executors.newCachedThreadPool()  
14        ;  
15        // 启动线程  
16        service.execute(producer1);
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
16         service.execute(producer2);
17         service.execute(producer3);
18         service.execute(consumer);
19
20         // 执行10s
21         Thread.sleep(10 * 1000);
22         producer1.stop();
23         producer2.stop();
24         producer3.stop();
25
26         Thread.sleep(2000);
27         // 退出Executor
28         service.shutdown();
29     }
30 }
```

DelayQueue

DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。

使用场景：

DelayQueue使用场景较少，但都相当巧妙，常见的例子比如使用一个DelayQueue来管理一个超时未响应的连接队列。

PriorityBlockingQueue

基于优先级的阻塞队列（优先级的判断通过构造函数传入的Compator对象来决定），但需要注意的是PriorityBlockingQueue并不会阻塞数据生产者，而只会在没有可消费的数据时，阻塞数据的消费者。因此使用的时候要特别注意，生产者生产数据的速度绝对不能快于消费者消费数据的速度，否则时间一长，会最终耗尽所有的可用堆内存空间。在实现PriorityBlockingQueue时，内部控制线程同步的锁采用的是公平锁。

SynchronousQueue

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

一种无缓冲的等待队列，类似于无中介的直接交易，有点像原始社会中的生产者和消费者，生产者拿着产品去集市销售给产品的最终消费者，而消费者必须亲自去集市找到所要商品的直接生产者，如果一方没有找到合适的目标，那么对不起，大家都在集市等待。相对于有缓冲的BlockingQueue来说，少了一个中间经销商的环节（缓冲区），如果有经销商，生产者直接把产品批发给经销商，而无需在意经销商最终会将这些产品卖给那些消费者，由于经销商可以库存一部分商品，因此相对于直接交易模式，总体来说采用中间经销商的模式会吞吐量高一些（可以批量买卖）；但另一方面，又因为经销商的引入，使得产品从生产者到消费者中间增加了额外的交易环节，单个产品的及时响应性能可能会降低。

声明一个SynchronousQueue有两种不同的方式，它们之间有着不太一样的行为。公平模式和非公平模式的区别：

如果采用公平模式：SynchronousQueue会采用公平锁，并配合一个FIFO队列来阻塞多余的生产者和消费者，从而体系整体的公平策略；

但如果采用非公平模式（SynchronousQueue默认）：SynchronousQueue采用非公平锁，同时配合一个LIFO队列来管理多余的生产者和消费者，而后一种模式，如果生产者和消费者的处理速度有差距，则很容易出现饥渴的情况，即可能有某些生产者或者是消费者的数据永远都得不到处理。

小结

BlockingQueue不光实现了一个完整队列所具有的基本功能，同时在多线程环境下，他还自动管理了多线间的自动等待于唤醒功能，从而使得程序员可以忽略这些细节，关注更高级的功能。

参考资料

<http://www.importnew.com/28053.html>

<https://www.cnblogs.com/jackyuj/archive/2010/11/24/1886553.html>

作者：方志朋，一线大厂架构师，

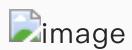
来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

Java并发： ConcurrentHashMap解读

ConcurrentHashMap解读

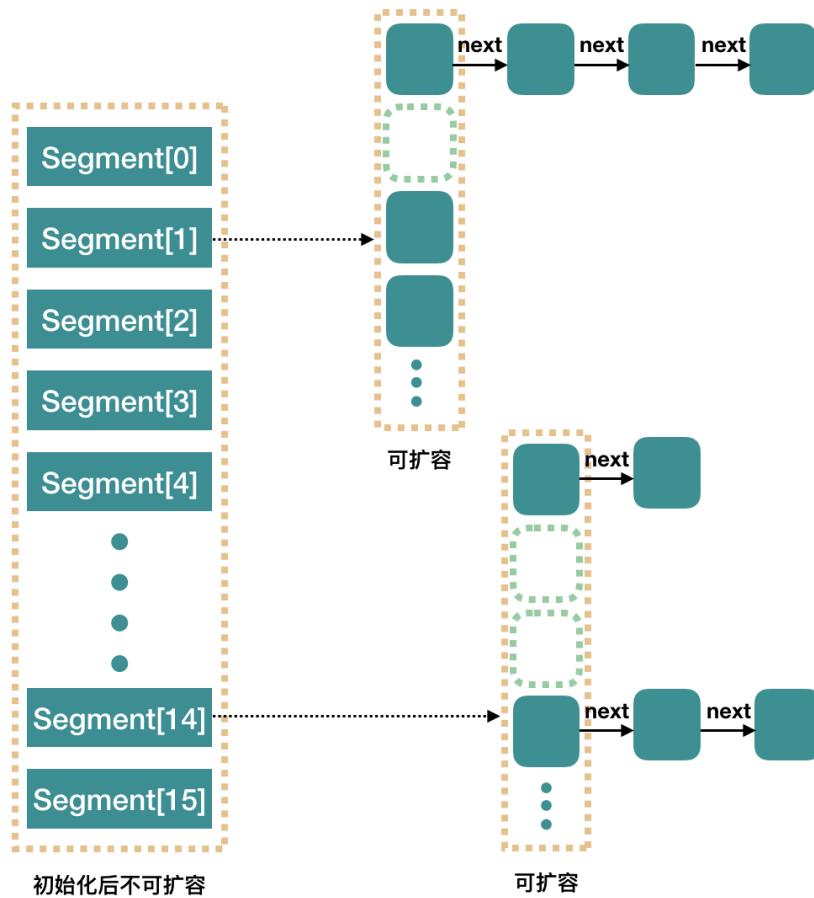
Java7 基于分段锁的ConcurrentHashMap

ConcurrentHashMap 和 HashMap 思路是差不多的，但是因为它支持并发操作，所以要复杂一些。

整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表“部分”或“一段”的意思，所以很多地方都会将其描述为分段锁。注意，行文中，我很多地方用了“槽”来代表一个 segment。

简单理解就是，ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。

Java7 ConcurrentHashMap 结构



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

concurrencyLevel：并行级别、并发数、Segment 数，怎么翻译不重要，理解它。默认是 16，也就是说 ConcurrentHashMap 有 16 个 Segments，所以理论上，这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。

再具体到每个 Segment 内部，其实每个 Segment 很像之前介绍的 HashMap，不过它要保证线程安全，所以处理起来要麻烦些。

get 过程中是没有加锁的，那自然我们就需要去考虑并发问题。

添加节点的操作 put 和删除节点的操作 remove 都是要加 segment 上的独占锁的，所以它们之间自然不会有问题是，我们需要考虑的问题就是 get 的时候在同一个 segment 中发生了 put 或 remove 操作。

put 操作的线程安全性。

- 初始化槽，这个我们之前就说过了，使用了 CAS 来初始化 Segment 中的数组。
- 添加节点到链表的操作是插入到表头的，所以，如果这个时候 get 操作在链表遍历的过程已经到了中间，是不会影响的。当然，另一个并发问题就是 get 操作在 put 之后，需要保证刚刚插入表头的节点被读取，这个依赖于 setEntryAt 方法中使用的 UNSAFE.putOrderedObject。
- 扩容。扩容是新创建了数组，然后进行迁移数据，最后将 newTable 设置给属性 table。所以，如果 get 操作此时也在进行，那么也没关系，如果 get 先行，那么就是在旧的 table 上做查询操作；而 put 先行，那么 put 操作的可见性保证就是 table 使用了 volatile 关键字。

remove 操作的线程安全性。

- remove 操作我们没有分析源码，所以这里说的读者感兴趣的话还是需要到源码中去求实一下的。
- get 操作需要遍历链表，但是 remove 操作会“破坏”链表。
- 如果 remove 破坏的节点 get 操作已经过去了，那么这里不存在任何问题。
- 如果 remove 先破坏了一个节点，分两种情况考虑。1、如果此节点是头结点，那么需要将头结点的 next 设置为数组该位置的元素，table 虽然使用了 volatile 修饰，但是 volatile 并不能提供数组内部操作的可见性保证，所以源码中使用了 UNSAFE 来操作数组，请看方法 setEntryAt。2、如果要删除的节点不是头结点，它会将要删除节点的后继节点接到前驱节点中，这里的并发保证就是 next 属性是 volatile 的。

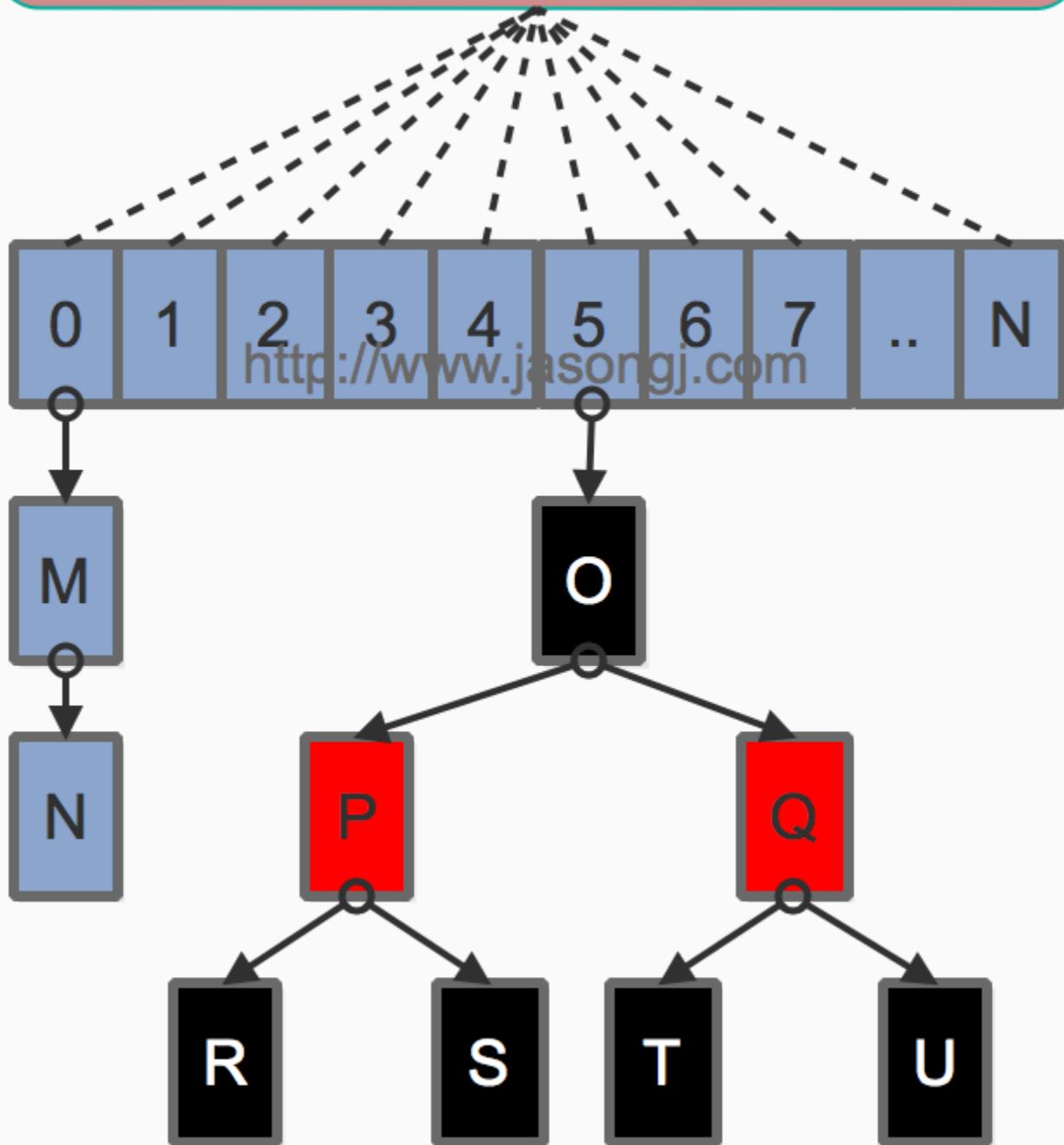
Java 8 基于CAS的ConcurrentHashMap

数据结构

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

Java 7为实现并行访问，引入了Segment这一结构，实现了分段锁，理论上最大并发度与Segment个数相等。Java 8为进一步提高并发性，摒弃了分段锁的方案，而是直接使用一个大的数组。同时为了提高哈希碰撞下的寻址性能，Java 8在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为O(N)）转换为红黑树（寻址时间复杂度为O(log(N)))）。其数据结构如下图所示：

ConcurrentHashMap



寻址方式

Java 8的ConcurrentHashMap同样是通过Key的哈希值与数组长度取模确定该Key在数组中的索引。同样为了避免不太好的Key的hashCode设计，它通过如下方法计算得到Key的最终哈希值。不同的是，Java 8的ConcurrentHashMap作者认为引入红黑树后，即使哈希冲突比较严重，寻址效率也足够高，所以作者并未在哈希值的计算上做过多设计，只是将Key的hashCode值与其高16位作异或并保证最高位为0（从而保证最终结果为正整数）。

```
1
2 static final int spread(int h) {
3     return (h ^ (h >>> 16)) & HASH_BITS;
4 }
```

同步方式

对于put操作，如果Key对应的数组元素为null，则通过CAS操作将其设置为当前值。如果Key对应的数组元素（也即链表表头或者树的根元素）不为null，则对该元素使用synchronized关键字申请锁，然后进行操作。如果该put操作使得当前链表长度超过一定阈值，则将该链表转换为树，从而提高寻址效率。

对于读操作，由于数组被volatile关键字修饰，因此不用担心数组的可见性问题。同时每个元素是一个Node实例（Java 7中每个元素是一个HashEntry），它的Key值和hash值都由final修饰，不可变更，无须关心它们被修改后的可见性问题。而其Value及对下一个元素的引用由volatile修饰，可见性也有保障。

```
1 static class Node<K,V> implements Map.Entry<K,V> {
2     final int hash;
3     final K key;
4     volatile V val;
5     volatile Node<K,V> next;
6 }
```

对于Key对应的数组元素的可见性，由Unsafe的getObjectVolatile方法保证。

```
1 static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
2     return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) +
ABASE);
3 }
```

size操作

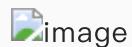
put方法和remove方法都会通过addCount方法维护Map的size。size方法通过sumCount获取由addCount方法维护的Map的size。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

Java并发：CopyOnWriteArrayList实现原理及源码分析

CopyOnWriteArrayList实现原理及源码分析

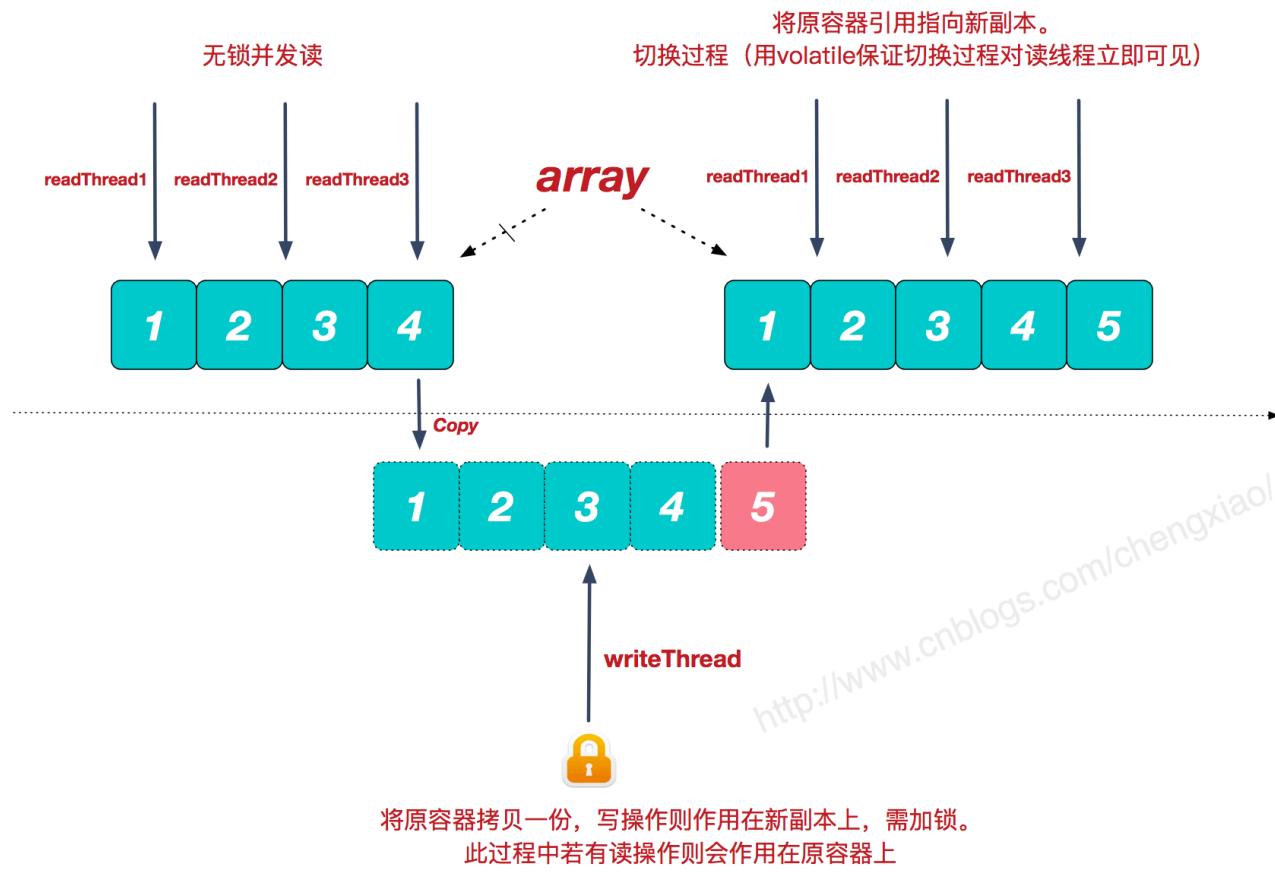
CopyOnWriteArrayList是Java并发包中提供的一个并发容器，它是个线程安全且读操作无锁的ArrayList，写操作则通过创建底层数组的新副本实现，是一种读写分离的并发策略，我们也可以称这种容器为“写时复制器”，Java并发包中类似的容器还有CopyOnWriteSet。本文会对CopyOnWriteArrayList的实现原理及源码进行分析。

实现原理

我们都知道，集合框架中的ArrayList是非线程安全的，Vector虽是线程安全的，但由于简单粗暴的锁同步机制，性能较差。而CopyOnWriteArrayList则提供了另一种不同的并发处理策略（当然是针对特定的并发场景）。

很多时候，我们的系统应对的都是读多写少的并发场景。CopyOnWriteArrayList容器允许并发读，读操作是无锁的，性能较高。至于写操作，比如向容器中添加一个元素，则首先将当前容器复制一份，然后在新副本上执行写操作，结束之后再将原容器的引用指向新容器。

公众号：方志朋



优缺点分析

了解了CopyOnWriteArrayList的实现原理，分析它的优缺点及使用场景就很容易了。

优点

读操作性能很高，因为无需任何同步措施，比较适用于读多写少的并发场景。Java的list在遍历时，若中途有别的线程对list容器进行修改，则会抛出ConcurrentModificationException异常。而CopyOnWriteArrayList由于其“读写分离”的思想，遍历和修改操作分别作用在不同的list容器，所以在使用迭代器进行遍历时候，也就不会抛出ConcurrentModificationException异常了

缺点

缺点也很明显，一是内存占用问题，毕竟每次执行写操作都要将原容器拷贝一份，数据量大时，对内存压力较大，可能会引起频繁GC；二是无法保证实时性，Vector对于读写操作均加锁同步，可以保证读和写的强一致性。而CopyOnWriteArrayList由于其实现策略的原因，写和读分别作用在新老不同容器上，在写操作执行过程中，读不会阻塞但读取到的却是老容器的数据。

源码分析

基本原理了解了，CopyOnWriteArrayList的代码实现看起来就很容易理解了。

```
1
2 public boolean add(E e) {
3     //ReentrantLock加锁，保证线程安全
4     final ReentrantLock lock = this.lock;
5     lock.lock();
6     try {
7         Object[] elements = getArray();
8         int len = elements.length;
9         //拷贝原容器，长度为原容器长度加一
10        Object[] newElements = Arrays.copyOf(elements, len +
11            1);
12        //在新副本上执行添加操作
13        newElements[len] = e;
14        //将原容器引用指向新副本
15        setArray(newElements);
16        return true;
17    } finally {
18        //解锁
19        lock.unlock();
20    }
}
```

添加的逻辑很简单，先将原容器copy一份，然后在新副本上执行写操作，之后再切换引用。当然此过程是要加锁的。

删除操作

```
1 public E remove(int index) {
2     //加锁
3     final ReentrantLock lock = this.lock;
4     lock.lock();
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
5   try {
6       Object[] elements = getArray();
7       int len = elements.length;
8       E oldValue = get(elements, index);
9       int numMoved = len - index - 1;
10      if (numMoved == 0)
11          //如果要删除的是列表末端数据，拷贝前len-1个数据到新副本上,
12          //再切换引用
13          setArray((Arrays.copyOf(elements, len - 1));
14      else {
15          //否则，将除要删除元素之外的其他元素拷贝到新副本中，并切换引
16          //用
17          Object[] newElements = new Object[len - 1];
18          System.arraycopy(elements, 0, newElements, 0, ind
19          ex);
20          System.arraycopy(elements, index + 1, newElements
21          , index,
22          numMoved);
23          setArray(newElements);
24      }
25      return oldValue;
26  } finally {
27      //解锁
28      lock.unlock();
29  }
30 }
```

公众号：方志朋

删除操作同理，将除要删除元素之外的其他元素拷贝到新副本中，然后切换引用，将原容器引用指向新副本。同属写操作，需要加锁。

我们再来看看读操作，CopyOnWriteArrayList的读操作是不用加锁的，性能很高。

```
1 public E get(int index) {
2     return get(getArray(), index);
3 }
```

直接读取即可，无需加锁

```
1 private E get(Object[] a, int index) {  
2     return (E) a[index];  
3 }
```

总结

本文对CopyOnWriteArrayList的实现原理和源码进行了分析，并对CopyOnWriteArrayList的优缺点也进行了分析（Java并发包中还提供了CopyOnWriteSet，原理类似）。其实所谓并发容器的优缺点，无非是取决于我们在面对特定并发场景时，是否能做出相对合理的选择和应用。也希望本文能帮助到有需要的童鞋，共勉。

原文链接

<https://www.cnblogs.com/chengxiao/p/6881974.html>

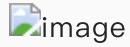
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



Java并发：Java中CAS详解

Java中CAS详解

在JDK 5之前Java语言是靠synchronized关键字保证同步的，这会导致有锁。

锁机制存在以下问题：

- (1) 在多线程竞争下，加锁、释放锁会导致比较多的上下文切换和调度延时，引起性能问题。
- (2) 一个线程持有锁会导致其它所有需要此锁的线程挂起。
- (3) 如果一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置，引起性能风险。

volatile是不错的机制，但是volatile不能保证原子性。因此对于同步最终还是要回到锁机制上来。

独占锁是一种悲观锁，synchronized就是一种独占锁，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。而另一个更加有效的锁就是乐观锁。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁用到的机制就是CAS，Compare and Swap。

什么是CAS

CAS,compare and swap的缩写，中文翻译成比较并交换。

我们都知道，在java语言之前，并发就已经广泛存在并在服务器领域得到了大量的应用。所以硬件厂商老早就在芯片中加入了大量直至并发操作的原语，从而在硬件层面提升效率。在intel的CPU中，使用cmpxchg指令。

在Java发展初期，java语言是不能够利用硬件提供的这些便利来提升系统的性能的。而随着java不断的发展，Java本地方法(JNI)的出现，使得java程序越过JVM直接调用本地方法提供了一种便捷的方式，因而java在并发的手段上也多了起来。而在Doug Lea提供的cucurenct包中，CAS理论是它实现整个java包的基石。

CAS 操作包含三个操作数——内存位置 (V) 、预期原值 (A) 和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。

通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行多步计算来获得新值 B，然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改，则 CAS 操作成功。

类似于 CAS 的指令允许算法执行读-修改-写操作，而无需害怕其他线程同时修改变量，因为如果其他线程修改变量，那么 CAS 会检测它（并失败），算法可以对该操作重新计算。

CAS的目的

利用CPU的CAS指令，同时借助JNI来完成Java的非阻塞算法。其它原子操作都是利用类似的特性完成的。而整个J.U.C都是建立在CAS之上的，因此对于synchronized阻塞算法，J.U.C在性能上有了很大的提升。

CAS存在的问题

CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。ABA问题，循环时间长开销大和只能保证一个共享变量的原子操作

1. ABA问题。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A-B-A 就会变成1A-2B-3A。

从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

关于ABA问题参考文档: <http://blog.hesey.net/2011/09/resolve-aba-by-atomicstampedeference.html>

2. 循环时间长开销大。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令(de-pipeline)，使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突(memory order violation)而引起CPU流水线被清空(CPU pipeline flush)，从而提高CPU的执行效率。

3. 只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！
者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i=2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

concurrent包的实现

由于java的CAS同时具有 volatile 读和volatile写的内存语义，因此Java线程之间的通信现在有了下面四种方式：

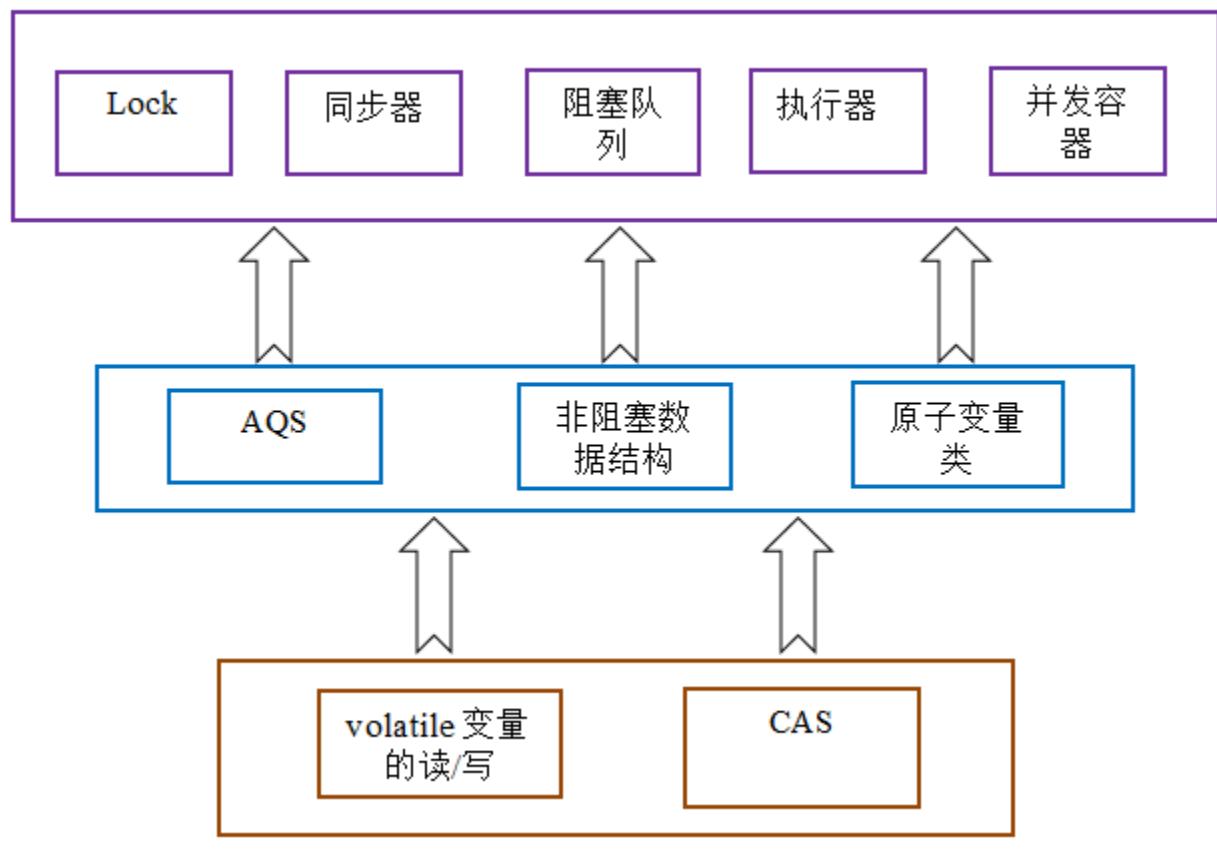
- A线程写volatile变量，随后B线程读这个volatile变量。
- A线程写volatile变量，随后B线程用CAS更新这个volatile变量。
- A线程用CAS更新一个volatile变量，随后B线程用CAS更新这个volatile变量。
- A线程用CAS更新一个volatile变量，随后B线程读这个volatile变量。

Java的CAS会使用现代处理器上提供的高效机器级别原子指令，这些原子指令以原子方式对内存执行读-改-写操作，这是在多处理器中实现同步的关键（从本质上来说，能够支持原子性读-改-写指令的计算机，是顺序计算图灵机的异步等价机器，因此任何现代的多处理器都会去支持某种能对内存执行原子性读-改-写操作的原子指令）。同时，volatile变量的读/写和CAS可以实现线程之间的通信。把这些特性整合在一起，就形成了整个concurrent包得以实现的基石。如果我们仔细分析concurrent包的源代码实现，会发现一个通用化的实现模式：

- 首先，声明共享变量为volatile；
- 然后，使用CAS的原子条件更新来实现线程之间的同步；
- 同时，配合以volatile的读/写和CAS所具有的volatile读和写的内存语义来实现线程之间的通信。

AQS，非阻塞数据结构和原子变量类（java.util.concurrent.atomic包中的类），这些concurrent包中的基础类都是使用这种模式来实现的，而concurrent包中的高层类又是依赖于这些基础类来实现的。从整体来看，concurrent包的实现示意图如下：

公众号：方志朋



原文地址

<https://blog.csdn.net/ls5718/article/details/52563959>

作者：方志朋，一线大厂架构师，

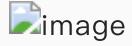
来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：

java架构师公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



公众号：方志朋

Java并发：Java中锁的分类

Java中锁的分类

在读很多并发文章中，会提及各种各样锁如公平锁，乐观锁等等，这篇文章介绍各种锁的分类。介绍的内容如下：

- 公平锁/非公平锁
- 可重入锁
- 独享锁/共享锁
- 互斥锁/读写锁
- 乐观锁/悲观锁
- 分段锁
- 偏向锁/轻量级锁/重量级锁
- 自旋锁

上面是很多锁的名词，这些分类并不是全是指锁的状态，有的指锁的特性，有的指锁的设计，下面总结的内容是对每个锁的名词进行一定的解释。

公平锁/非公平锁

公平锁是指多个线程按照申请锁的顺序来获取锁。

非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿现象。

对于Java ReentrantLock而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。

对于Synchronized而言，也是一种非公平锁。由于其并不像ReentrantLock是通过AQS(AbstractQueuedSynchronizer)的来实现线程调度，所以并没有任何办法使其变成公平锁。

可重入锁

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。说的有点抽象，下面会有一个代码的示例。

对于Java ReentrantLock而言，他的名字就可以看出是一个可重入锁，其名字是Reentrant Lock重新进入锁。

对于Synchronized而言，也是一个可重入锁。可重入锁的一个好处是可一定程度避免死锁。

```
1 synchronized void setA() throws Exception{  
2     Thread.sleep(1000);  
3     setB();  
4 }  
5  
6 synchronized void setB() throws Exception{  
7     Thread.sleep(1000);  
8 }
```

上面的代码就是一个可重入锁的一个特点，如果不是可重入锁的话，setB可能不会被当前线程执行，可能造成死锁。

独享锁/共享锁

- 独享锁是指该锁一次只能被一个线程所持有。
- 共享锁是指该锁可被多个线程所持有。

对于Java ReentrantLock而言，其是独享锁。但是对于Lock的另一个实现类ReadWriteLock，其读锁是共享锁，其写锁是独享锁。

读锁的共享锁可保证并发读是非常高效的，读写，写读，写写的过程是互斥的。

独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。

对于Synchronized而言，当然是独享锁。

互斥锁/读写锁

上面讲的独享锁/共享锁就是一种广义的说法，互斥锁/读写锁就是具体的实现。

- 互斥锁在Java中的具体实现就是ReentrantLock
- 读写锁在Java中的具体实现就是ReadWriteLock

乐观锁/悲观锁

乐观锁与悲观锁不是指具体的什么类型的锁，而是指看待并发同步的角度。

- 悲观锁认为对于同一个数据的并发操作，一定是会发生修改的，哪怕没有修改，也会认为修改。因此对于同一个数据的并发操作，悲观锁采取加锁的形式。悲观的认为，不加锁的并发操作一定会出问题。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 乐观锁则认为对于同一个数据的并发操作，是不会发生修改的。在更新数据的时候，会采用尝试更新，不断重新的方式更新数据。乐观的认为，不加锁的并发操作是没有事情的。

从上面的描述我们可以看出，悲观锁适合写操作非常多的场景，乐观锁适合读操作非常多的场景，不加锁会带来大量的性能提升。

悲观锁在Java中的使用，就是利用各种锁。

乐观锁在Java中的使用，是无锁编程，常常采用的是CAS算法，典型的例子就是原子类，通过CAS自旋实现原子操作的更新。

分段锁

分段锁其实是一种锁的设计，并不是具体的一种锁，对于ConcurrentHashMap而言，其并发的实现就是通过分段锁的形式来实现高效的并发操作。

我们以ConcurrentHashMap来说一下分段锁的含义以及设计思想，ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap（JDK7与JDK8中HashMap的实现）的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock）。

当需要put元素的时候，并不是对整个hashmap进行加锁，而是先通过hashcode来知道他要放在那一个分段中，然后对这个分段进行加锁，所以当多线程put的时候，只要不是放在一个分段中，就实现了真正的并行的插入。

但是，在统计size的时候，可就是获取hashmap全局信息的时候，就需要获取所有的分段锁才能统计。

分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。

偏向锁/轻量级锁/重量级锁

这三种锁是指锁的状态，并且是针对Synchronized。在Java 5通过引入锁升级的机制来实现高效Synchronized。这三种锁的状态是通过对对象监视器在对象头中的字段来表明的。

- 偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价。
- 轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。
- 重量级锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。重量级锁会让其他申请的线程进入阻塞，性能降低。

自旋锁

在Java中，自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。

典型的自旋锁实现的例子，可以参考自旋锁的实现

原文链接

<https://www.cnblogs.com/qifengshi/p/6831055.html>

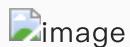
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



Java并发：Java并发编程：CountDownLatch、CyclicBarrier和Semaphore

Java并发编程：CountDownLatch、CyclicBarrier和Semaphore

CountDownLatch

A synchronization aid that allows one or more threads to wait until

- a set of operations being performed in other threads completes.

翻译：CountDownLatch是一个异步辅助类，它能让一个或多个线程处于等待状态，直到其他线程完成了一些列操作。

比如某个线程需要其他线程执行完毕才能执行其他的：

```
1 public void await() throws InterruptedException { }; //调用await  
()方法的线程会被挂起，它会等待直到count值为0才继续执行  
2 public boolean await(long timeout, TimeUnit unit) throws InterruptedException { }; //和await()类似，只不过等待一定的时间后count值还没变为0  
的话就会继续执行  
3 public void countDown() { }; //将count值减1
```

例子如下：

```
1 public class CountDownLatchTest {  
2  
3  
4     public static void main(String[] args) {  
5         final CountDownLatch latch = new CountDownLatch(2);  
6         new Thread(() -> {  
7             System.out.println("子线程1执行开始");  
8             try {  
9                 latch.await();  
10            } catch (InterruptedException e) {  
11                e.printStackTrace();  
12            }  
13            System.out.println("子线程1执行结束");  
14        });  
15        new Thread(() -> {  
16            System.out.println("子线程2执行开始");  
17            try {  
18                latch.await();  
19            } catch (InterruptedException e) {  
20                e.printStackTrace();  
21            }  
22            System.out.println("子线程2执行结束");  
23        }).start();  
24    }  
25}
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
10         Thread.sleep(3000);
11     } catch (InterruptedException e) {
12         e.printStackTrace();
13     }
14     System.out.println("子线程1执行结束");
15     latch.countDown();
16
17 }).start();
18
19
20     new Thread(() -> {
21
22         System.out.println("子线程2执行开始");
23         try {
24             Thread.sleep(3000);
25         } catch (InterruptedException e) {
26             e.printStackTrace();
27         }
28         System.out.println("子线程2执行结束");
29         latch.countDown();
30
31 }).start();
32
33     try {
34         latch.await();
35         System.out.println("所有子线程执行完毕了。。。");
36     } catch (InterruptedException e) {
37         e.printStackTrace();
38     }
39 }
40 }
```

公众号：方志朋

执行结果：

```
子线程1执行开始
子线程2执行开始
子线程1执行结束
子线程2执行结束
所有子线程执行完毕了。。。
```

CyclicBarrier用法

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called *cyclic* because it can be re-used after the waiting threads are released.

翻译：回环栏栅这个类，让所有的现场都去相互等待，知道它们都到达了一个栏栅的点。

字面意思回环栅栏，通过它可以实现让一组线程等待至某个状态之后再全部同时执行。叫做回环是因为当所有等待线程都被释放以后，CyclicBarrier可以被重用。我们暂且把这个状态就叫做barrier，当调用await()方法之后，线程就处于barrier了。

```
1 public CyclicBarrier(int parties, Runnable barrierAction) {  
2 }  
3  
4 public CyclicBarrier(int parties) {  
5 }
```

参数parties指让多少个线程或者任务等待至barrier状态；参数barrierAction为当这些线程都达到barrier状态时会执行的内容。

CyclicBarrier最重要的是中最重要的方法就是await方法，

```
1  
2 public int await() throws InterruptedException, BrokenBarrierException { };
```

用来挂起当前线程，直至所有线程都到达barrier状态再同时执行后续任务；

```
1  
2 public class CyclicBarrierTest1 {
```

```
3
4     public static void main(String[] args) {
5         final int N = 4;
6         CyclicBarrier barrier = new CyclicBarrier(N);
7         for (int i=0;i<N;i++){
8             new Thread(new Writer(barrier)).start();
9         }
10    }
11
12    static class Writer implements Runnable {
13        CyclicBarrier barrier;
14
15        public Writer(CyclicBarrier barrier) {
16            this.barrier = barrier;
17        }
18
19        @Override
20        public void run() {
21
22            String name = Thread.currentThread().getName();
23            System.out.println(name + "开始读写数据");
24            try {
25                Thread.sleep(4000);
26                System.out.println(name + "结束读写数据");
27                barrier.await();
28            } catch (InterruptedException e) {
29                e.printStackTrace();
30            } catch (BrokenBarrierException e) {
31                e.printStackTrace();
32            }
33            System.out.println(name + "所以线程都读取完毕，开始其他的操作");
34        }
35    }
36}
37
38 }
```

公众号：方志朋

控制台打印：

```
Thread-0开始读写数据  
Thread-1开始读写数据  
Thread-2开始读写数据  
Thread-3开始读写数据  
Thread-0结束读写数据  
Thread-1结束读写数据  
Thread-3结束读写数据  
Thread-2结束读写数据  
Thread-2所以线程都读取完毕，开始其他的操作  
Thread-1所以线程都读取完毕，开始其他的操作  
Thread-0所以线程都读取完毕，开始其他的操作  
Thread-3所以线程都读取完毕，开始其他的操作
```

从上面输出结果可以看出，每个写入线程执行完写数据操作之后，就在等待其他线程写入操作完毕。

当所有线程写入操作完毕之后，所有线程就继续进行后续的操作了。

如果说想在所有线程写入操作完之后，进行额外的其他操作可以为CyclicBarrier提供Runnable参数：

```
1  
2 public class CyclicBarrierTest2 {  
3  
4  
5     public static void main(String[] args) {  
6         final int N = 4;  
7         CyclicBarrier barrier = new CyclicBarrier(N, new Runnable  
8             () {  
9                 @Override  
10                public void run() {  
11                    String name=Thread.currentThread().getName();  
12                    System.out.println(name +"barrier Runnable");  
13                }  
14            });  
15            for (int i=0;i<N;i++){  
16                new Thread(new CyclicBarrierTest1.Writer(barrier)).st  
art();  
17        }
```

公众号：方志朋

```
17    }
18
19    static class Writer implements Runnable {
20        CyclicBarrier barrier;
21
22        public Writer(CyclicBarrier barrier) {
23            this.barrier = barrier;
24        }
25
26        @Override
27        public void run() {
28
29            String name = Thread.currentThread().getName();
30            System.out.println(name + "开始读写数据");
31            try {
32                Thread.sleep(4000);
33                System.out.println(name + "结束读写数据");
34                barrier.await();
35            } catch (InterruptedException e) {
36                e.printStackTrace();
37            } catch (BrokenBarrierException e) {
38                e.printStackTrace();
39            }
40            System.out.println(name + "所以线程都读取完毕，开始其他的操作");
41
42        }
43    }
44 }
```

公众号：方志朋

控制台打印：

```
Thread-0开始读写数据
Thread-1开始读写数据
Thread-2开始读写数据
Thread-3开始读写数据
Thread-0结束读写数据
Thread-1结束读写数据
Thread-3结束读写数据
```

Thread-2结束读写数据

Thread-2barrier Runnable

Thread-2所以线程都读取完毕，开始其他的操作

Thread-1所以线程都读取完毕，开始其他的操作

Thread-0所以线程都读取完毕，开始其他的操作

Thread-3所以线程都读取完毕，开始其他的操作

Semaphore

A counting semaphore. Conceptually, a semaphore maintains a set of

- permits. Each {@link #acquire} blocks if necessary until a permit is available, and then takes it. Each {@link #release} adds a permit, potentially releasing a blocking acquirer.
- However, no actual permit objects are used; the {@code Semaphore} just keeps a count of the number available and acts accordingly.

翻译：计数信号量从概念上讲，信号量维护着一组信号许可证。每个{@link #acquire}都会根据需要进行阻止，直到获得许可可用，然后把它。每个{@link #release}都会添加一个许可证，潜在地释放阻止的收购方。但是，没有使用实际的许可证对象；只是信号量而已保持可用数量的计数，并采取相应的行动。

Semaphore翻译成字面意思为 信号量，Semaphore可以控同时访问的线程个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。

```
1 public Semaphore(int permits) {          //参数permits表示许可数目，即
    同时可以允许多少线程进行访问
2     sync = new NonfairSync(permits);
3 }
4
5 public Semaphore(int permits, boolean fair) { //这个多了一个参数fa
    ir表示是否是公平的，即等待时间越久的越先获取许可
6     sync = (fair)? new FairSync(permits) : new NonfairSync(permits);
7 }
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

下面说一下Semaphore类中比较重要的几个方法，首先是acquire()、release()方法：

```
1 public void acquire() throws InterruptedException {  
2  
3 }      //获取一个许可  
4 public void acquire(int permits) throws InterruptedException {  
5  
6 }      //获取permits个许可  
7 public void release() {  
8  
9 }      //释放一个许可  
10 public void release(int permits) {  
11  
12 }      //释放permits个许可
```

acquire()用来获取一个许可，若无许可能够获得，则会一直等待，直到获得许可。

release()用来释放许可。注意，在释放许可之前，必须先获获得许可。

这4个方法都会被阻塞，如果想立即得到执行结果，可以使用下面几个方法：

```
1 public boolean tryAcquire() { };      //尝试获取一个许可，若获取成功，则立  
即返回true，若获取失败，则立即返回false  
2 public boolean tryAcquire(long timeout, TimeUnit unit) throws Inte  
rruptedException { }; //尝试获取一个许可，若在指定的时间内获取成功，则立即  
返回true，否则则立即返回false  
3 public boolean tryAcquire(int permits) { }; //尝试获取permits个许可，  
若获取成功，则立即返回true，若获取失败，则立即返回false  
4 public boolean tryAcquire(int permits, long timeout, TimeUnit uni  
t) throws InterruptedException { }; //尝试获取permits个许可，若在
```

另外还可以通过availablePermits()方法得到可用的许可数目。

```
1 public class SemaphoreTest {  
2  
3     public static void main(String[] args) {
```

```
4     Semaphore semaphore=new Semaphore(3);
5
6     for (int i=0;i<10;i++){
7         new Thread(new Worker(semaphore,i)).start();
8     }
9 }
10
11 static class Worker implements Runnable{
12     Semaphore semaphore;
13     int num;
14
15     public Worker(Semaphore semaphore,int i){
16         this.semaphore=semaphore;
17         this.num=i;
18     }
19
20     @Override
21     public void run() {
22         try {
23             semaphore.acquire();
24             System.out.println("工人"+this.num+"占用一个机器在生
产...");
25             Thread.sleep(2000);
26             System.out.println("工人"+this.num+"释放出机器");
27             semaphore.release();
28         } catch (InterruptedException e) {
29             e.printStackTrace();
30         }
31     }
32 }
33 }
```

公众号：方志朋

下面对上面说的三个辅助类进行一个总结：

- CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：

CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；

而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；

另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

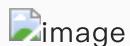
- Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

Java并发：java线程池详解

java线程池详解

随着cpu核数越来越多，不可避免的利用多线程技术以充分利用其计算能力。所以，多线程技术是服务端开发人员必须掌握的技术。

线程的创建和销毁，都涉及到系统调用，比较消耗系统资源，那么有没有一种办法使得线程可以复用，就是执行完一个任务，并不被销毁，而是可以继续执行其他的任务？所以就引入了线程池技术，避免频繁的线程创建和销毁。

在Java用有一个Executors工具类，可以为我们创建一个线程池，其本质就是new了一个ThreadPoolExecutor对象。线程池几乎也是面试必考问题。本文结合源代码，说说ThreadExecutor的工作原理。

ThreadPoolExecutor类

java.util.concurrent.ThreadPoolExecutor类是线程池中最核心的一个类，因此如果要透彻地了解Java中的线程池，必须先了解这个类。下面我们来看一下ThreadPoolExecutor类的具体实现源码。

在ThreadPoolExecutor类中提供了四个构造方法：

```
1 public class ThreadPoolExecutor extends AbstractExecutorService {  
2     ....  
3     public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,  
4                                long keepAliveTime, TimeUnit unit,  
5                                BlockingQueue<Runnable> workQueue);  
6  
7     public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,  
8                                long keepAliveTime, TimeUnit unit,  
9                                BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory);  
10    public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,  
11                                long keepAliveTime, TimeUnit unit,  
12                                BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler);
```

公众号：方志朋

```
11  
12     public ThreadPoolExecutor(int corePoolSize,int maximumPoolSiz  
e,long keepAliveTime,TimeUnit unit,  
13         BlockingQueue<Runnable> workQueue,ThreadFactory threadFac  
tory,RejectedExecutionHandler handler);  
14     ...
```

从上面的代码可以得知，ThreadPoolExecutor继承了AbstractExecutorService类，并提供了四个构造器，事实上，通过观察每个构造器的源码具体实现，发现前面三个构造器都是调用的第四个构造器进行的初始化工作。

下面解释下一下构造器中各个参数的含义：

- corePoolSize：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；
- maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；
- keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；
- unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

```
1 TimeUnit.DAYS;           //天  
2 TimeUnit.HOURS;          //小时  
3 TimeUnit.MINUTES;        //分钟  
4 TimeUnit.SECONDS;         //秒  
5 TimeUnit.MILLISECONDS;    //毫秒  
6 TimeUnit.MICROSECONDS;   //微妙  
7 TimeUnit.NANOSECONDS;    //纳秒
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

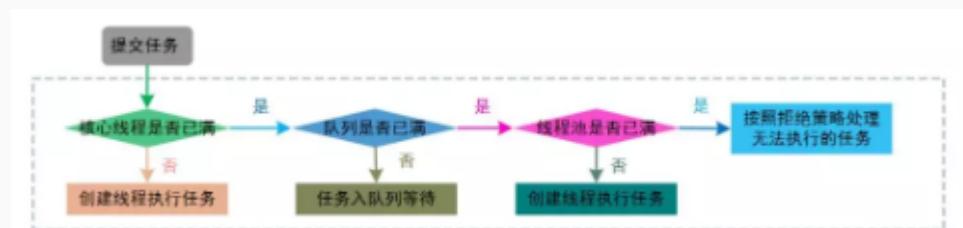
```
1 ArrayBlockingQueue;  
2 LinkedBlockingQueue;  
3 SynchronousQueue;
```

ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。

- threadFactory：线程工厂，主要用来创建线程；
- handler：表示当拒绝处理任务时的策略，有以下四种取值：

```
1 ThreadPoolExecutor.AbortPolicy: 丢弃任务并抛出RejectedExecutionException异常。  
2 ThreadPoolExecutor.DiscardPolicy: 也是丢弃任务，但是不抛出异常。  
3 ThreadPoolExecutor.DiscardOldestPolicy: 丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）  
4 ThreadPoolExecutor.CallerRunsPolicy: 由调用线程处理该任务
```

这里用一个图来说明线程池的执行流程



任务被提交到线程池，会先判断当前线程数量是否小于corePoolSize，如果小于则创建线程来执行提交的任务，否则将任务放入workQueue队列，如果workQueue满了，则判断当前线程数量是否小于maximumPoolSize,如果小于则创建线程执行任务，否则就会调用handler，以表示线程池拒绝接收任务。

这里以jdk1.8.0_111的源代码为例，看一下具体实现

1、先看一下线程池的executor方法

```
int c = ctl.get();
if (workerCountOf(c) < corePoolSize) {
    if (addWorker(command, true))
        return;
    c = ctl.get();
}
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);
}
else if (! addWorker(command, false))
    reject(command);
```

- 判断当前活跃线程数是否小于corePoolSize,如果小于，则调用addWorker创建线程执行任务
- 如果不小于corePoolSize，则将任务添加到workQueue队列。
- 如果放入workQueue失败，则创建线程执行任务，如果这时创建线程失败(当前线程数不小于maximumPoolSize时)，就会调用reject(内部调用handler)拒绝接受任务。

2、再看下addWorker的方法实现

```
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
                firstTask == null &&
                ! workQueue.isEmpty())))
            return false;

        for (;;) {
            int wc = workerCountOf(c);
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get(); // Re-read ctl
            if (runStateOf(c) != rs)
                continue retry;
        }
    }
}
```

这块代码是在创建非核心线程时，即core等于false。判断当前线程数是否大于等于maximumPoolSize，如果大于等于则返回false，即上边说到的③中创建线程失败的情况。

addWorker方法的下半部分：

公众号: 方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
w = new Worker(firstTask);
final Thread t = w.thread;
if (t != null) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // Recheck while holding lock.
        // Back out on ThreadFactory failure or if
        // shut down before lock acquired.
        int rs = runStateOf(ctl.get());
        if (rs < SHUTDOWN ||
            (rs == SHUTDOWN && firstTask == null)) {
            if (t.isAlive()) // precheck that t is startable
                throw new IllegalThreadStateException();
            workers.add(w);
            int s = workers.size();
            if (s > largestPoolSize)
                largestPoolSize = s;
            workerAdded = true;
        }
    } finally {
        mainLock.unlock();
    }
    if (workerAdded) {
        t.start(); ②
        workerStarted = true;
    }
}
```

- 创建Worker对象，同时也会实例化一个Thread对象。
- 启动启动这个线程

3、再到Worker里看看其实现

```
/*
Worker(Runnable firstTask) {
    setState(-1); // inhibit interrupts until runWorker
    this.firstTask = firstTask;
    this.thread = getThreadFactory().newThread(this);
}

/** Delegates main run loop to outer runWorker */
public void run() { runWorker(this); }
```

可以看到在创建Worker时会调用threadFactory来创建一个线程。上边的②中启动一个线程就会触发Worker的run方法被线程调用。

4、接下来咱们看看runWorker方法的逻辑

公众号：方志朋

```

final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        while (task != null || (task = getTask()) != null) {
            wt.lock();
            // If pool is stopping, ensure thread is interrupted;
            // if not, ensure thread is not interrupted. This
            // requires a recheck in second case to deal with
            // shutdownNow race while clearing interrupt
            if ((runStateAtLeast(ctl.get(), STOP) ||
                (Thread.interrupted() &&
                 runStateAtLeast(ctl.get(), STOP))) ||
                !wt.isInterrupted())
                wt.interrupt();
            try {
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
                    task.run();
                } catch (RuntimeException x) {
                    thrown = x; throw x;
                } catch (Error x) {
                    thrown = x; throw x;
                } catch (Throwable x) {
                    thrown = x; throw new Error(x);
                } finally {
                    afterExecute(task, thrown);
                }
            } finally {

```

线程调用runWoker，会while循环调用getTask方法从workerQueue里读取任务，然后执行任务。只要getTask方法不返回null，此线程就不会退出。

5、最后在看看getTask方法实现

```

private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount();
            return null;
        }

        int wc = workerCountOf(c);

        // Are workers subject to culling?
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) {
            if (compareAndDecrementWorkerCount(c))
                return null;
            continue;
        }

        try {
            Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take();
            if (r != null)
                return r;
            timedOut = true;
        } catch (InterruptedException retry) {
            timedOut = false;
        }
    }
}

```

- 咱们先不管allowCoreThreadTimeOut，这个变量默认值是false。wc>corePoolSize则是判断当前线程数是否大于corePoolSize。
- 如果当前线程数大于corePoolSize，则会调用workQueue的poll方法获取任务，超时时间是keepAliveTime。如果超过keepAliveTime时长，poll返回了null，上边提到的while循序就会退出，线程也就执行完了。

公众号：方志朋

如果当前线程数小于corePoolSize，则会调用workQueue的take方法阻塞在当前。

参考资料

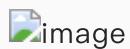
<https://www.cnblogs.com/dolphin0520/p/3932921.html>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

Java并发：Synchronized原理和优化

Synchronized原理和优化

Synchronized是Java中解决并发问题的一种最常用的方法，也是最简单的一种方法。Synchronized的作用主要有三个：（1）确保线程互斥的访问同步代码（2）保证共享变量的修改能够及时可见（3）有效解决重排序问题。

Synchronized 原理

我们先来了解Synchronized的原理，我们先通过反编译下面的代码来看看Synchronized是如何实现对代码块进行同步的

```
1  
2 package com.padx.test.concurrent;  
3  
4 public class SynchronizedDemo {  
5     public void method() {  
6         synchronized (this) {  
7             System.out.println("Method 1 start");  
8         }  
9     }  
10 }
```

反编译结果：



关于这两条指令的作用，我们直接参考JVM规范中描述：

monitorenter :

这段话的大概意思为：

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

- 1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
- 2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1。
- 3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

monitorexit:

这段话的大概意思为：

执行monitorexit的线程必须是objectref所对应的monitor的所有者。

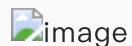
指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

通过这两段描述，我们应该能很清楚的看出Synchronized的实现原理，Synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

我们再来看一下同步方法的反编译结果：

```
1  
2 package com.padx.test.concurrent;  
3  
4 public class SynchronizedMethod {  
5     public synchronized void method() {  
6         System.out.println("Hello World!");  
7     }  
8 }
```

反编译结果：



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

从反编译的结果来看，方法的同步并没有通过指令monitorenter和monitorexit来完成（理论上其实也可以通过这两条指令来实现），不过相对于普通方法，其常量池中多了ACC_SYNCHRONIZED标示符。JVM就是根据该标示符来实现方法的同步的：当方法调用时，调用指令将会检查方法的ACC_SYNCHRONIZED访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。

重量级锁

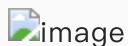
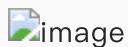
Synchronized是通过对对象内部的一个叫做监视器锁（monitor）来实现的。但是监视器锁本质又是依赖于底层的操作系统的Mutex Lock来实现的。而操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间，这就是为什么Synchronized效率低的原因。因此，这种依赖于操作系统Mutex Lock所实现的锁我们称之为“重量级锁”。JDK中对Synchronized做的种种优化，其核心都是为了减少这种重量级锁的使用。JDK1.6以后，为了减少获得锁和释放锁所带来的性能消耗，提高性能，引入了“轻量级锁”和“偏向锁”。

锁状态	25 bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC标记	空				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01
无锁	对象的hashCode		对象分代年龄	0	01

“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

轻量级锁的加锁过程

- (1) 在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 Displaced Mark Word。这时候线程堆栈与对象头的状态如图2.1所示。
- (2) 拷贝对象头中的Mark Word复制到锁记录中。
- (3) 拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock record里的owner指针指向object mark word。如果更新成功，则执行步骤 (3)，否则执行步骤 (4)。
- (4) 如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态，这时候线程堆栈与对象头的状态如图2.2所示。
- (5) 如果这个更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。而当前线程便尝试使用自旋来获取锁，自旋就是为了不让线程阻塞，而采用循环去获取锁的过程。



公众号：方志朋

轻量级锁的解锁过程：

- (1) 通过CAS操作尝试把线程中复制的Displaced Mark Word对象替换当前的Mark Word。
- (2) 如果替换成功，整个同步过程就完成了。
- (3) 如果替换失败，说明有其他线程尝试过获取该锁（此时锁已膨胀），那就要在释放锁的同时，唤醒被挂起的线程。

偏向锁

引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗）。上面说过，轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能。

1、偏向锁获取过程：

- (1) 访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01——确认为可偏向状态。
- (2) 如果为可偏向状态，则测试线程ID是否指向当前线程，如果是，进入步骤（5），否则进入步骤（3）。
- (3) 如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行（5）；如果竞争失败，执行（4）。
- (4) 如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。
- (5) 执行同步代码。

2、偏向锁的释放：

偏向锁的撤销在上述第四步骤中有提到。偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动去释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

3、重量级锁、轻量级锁和偏向锁之间转换



该图主要是对上述内容的总结，如果对上述内容有较好的了解的话，该图应该很容易看懂。

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。	如果线程间存在锁竞争，会带来额外的锁撤销的消耗。	适用于只有一个线程访问同步块场景。
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度。	如果始终得不到锁竞争的线程使用自旋会消耗CPU。	追求响应时间。
重量级锁	线程竞争不使用自旋，不会消耗CPU。	线程阻塞，响应时间缓慢。	追求吞吐量。

其他优化

1、**适应性自旋（Adaptive Spinning）**：从轻量级锁获取的流程中我们知道，当线程在获取轻量级锁的过程中执行CAS操作失败时，是要通过自旋来获取重量级锁的。问题在于，自旋是需要消耗CPU的，如果一

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

直获取不到锁的话，那该线程就一直处在自旋状态，白白浪费CPU资源。解决这个问题最简单的办法就是指定自旋的次数，例如让其循环10次，如果还没获取到锁就进入阻塞状态。但是JDK采用了更聪明的方式——适应性自旋，简单来说就是线程如果自旋成功了，则下次自旋的次数会更多，如果自旋失败了，则自旋的次数就会减少。

2、锁粗化（Lock Coarsening）：锁粗化的概念应该比较好理解，就是将多次连接在一起的加锁、解锁操作合并为一次，将多个连续的锁扩展成一个范围更大的锁。举个例子：

```
1 public class StringBufferTest {  
2     StringBuffer stringBuffer = new StringBuffer();  
3  
4     public void append(){  
5         stringBuffer.append("a");  
6         stringBuffer.append("b");  
7         stringBuffer.append("c");  
8     }  
9 }
```

这里每次调用stringBuffer.append方法都需要加锁和解锁，如果虚拟机检测到有一系列连串的对同一个对象加锁和解锁操作，就会将其合并成一次范围更大的加锁和解锁操作，即在第一次append方法时进行加锁，最后一次append方法结束后进行解锁。

3、锁消除（Lock Elimination）：锁消除即删除不必要的加锁操作。根据代码逃逸技术，如果判断到一段代码中，堆上的数据不会逃逸出当前线程，那么可以认为这段代码是线程安全的，不必要加锁。看下面这段程序：

```
1 public class SynchronizedTest02 {  
2  
3     public static void main(String[] args) {  
4         SynchronizedTest02 test02 = new SynchronizedTest02();  
5         //启动预热  
6         for (int i = 0; i < 10000; i++) {  
7             i++;  
8         }  
9         long start = System.currentTimeMillis();  
10        for (int i = 0; i < 100000000; i++) {  
11            test02.append("abc", "def");  
12        }  
13    }  
14 }
```

```
12     }
13     System.out.println("Time=" + (System.currentTimeMillis()
14         - start));
15
16     public void append(String str1, String str2) {
17         StringBuffer sb = new StringBuffer();
18         sb.append(str1).append(str2);
19     }
20 }
```

虽然StringBuffer的append是一个同步方法，但是这段程序中的StringBuffer属于一个局部变量，并且不会从该方法中逃逸出去，所以其实这过程是线程安全的，可以将锁消除。下面是我本地执行的结果：



为了尽量减少其他因素的影响，这里禁用了偏向锁（-XX:-UseBiasedLocking）。通过上面程序，可以看出消除锁以后性能还是有比较大提升的。

注：可能JDK各个版本之间执行的结果不尽相同，我这里采用的JDK版本为1.6。

参考资料

<https://www.cnblogs.com/paddix/p/5405678.html>

<http://www.cnblogs.com/paddix/p/5367116.html>

<https://blog.csdn.net/chenssy/article/details/54883355>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

Java并发：彻底理解ThreadLocal

彻底理解ThreadLocal

深挖过threadLocal之后，一句话概括：Synchronized用于线程间的数据共享，而ThreadLocal则用于线程间的数据隔离。所以ThreadLocal的应用场合，最适合的是按线程多实例（每个线程对应一个实例）的对象的访问，并且这个对象很多地方都要用到。

数据隔离的秘诀其实是这样的，Thread有个ThreadLocalMap类型的属性，叫做threadLocals，该属性用来保存该线程本地变量。这样每个线程都有自己的数据，就做到了不同线程间数据的隔离，保证了数据安全。

接下来采用jdk1.8源码进行深挖一下ThreadLocal和ThreadLocalMap。

ThreadLocal是什么

早在JDK 1.2的版本中就提供java.lang.ThreadLocal，ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序。

当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

从线程的角度看，目标变量就象是线程的本地变量，这也是类名中“Local”所要表达的意思。

所以，在Java中编写线程局部变量的代码相对来说要笨拙一些，因此造成线程局部变量没有在Java开发者中得到很好的普及。

原理

ThreadLocal，连接ThreadLocalMap和Thread。来处理Thread的ThreadLocalMap属性，包括init初始化属性赋值、get对应的变量，set设置变量等。通过当前线程，获取线程上的ThreadLocalMap属性，对数据进行get、set等操作。

ThreadLocalMap，用来存储数据，采用类似hashmap机制，存储了以threadLocal为key，需要隔离的数据为value的Entry键值对数组结构。

ThreadLocal，有个ThreadLocalMap类型的属性，存储的数据就放在这儿。

ThreadLocal、ThreadLocal、Thread之间的关系

ThreadLocalMap是ThreadLocal内部类，由ThreadLocal创建，Thread有 ThreadLocal.ThreadLocalMap类型的属性

ThreadLocalMap简介

看名字就知道是个map，没错，这就是个hashMap机制实现的map，用Entry数组来存储键值对，key是ThreadLocal对象，value则是具体的值。值得一提的是，为了方便GC，Entry继承了WeakReference，也就是弱引用。里面有一些具体关于如何清理过期的数据、扩容等机制，思路基本和hashmap差不多，有兴趣的可以自行阅读了解，这边只需知道大概的数据存储结构即可。

Thread同步机制的比较

ThreadLocal和线程同步机制相比有什么优势呢？

Synchronized用于线程间的数据共享，而ThreadLocal则用于线程间的数据隔离。

在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

而ThreadLocal则从另一个角度来解决多线程的并发访问。ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

Spring使用ThreadLocal解决线程安全问题我们知道在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域。就是因为Spring对一些Bean（如RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder等）中非线程安全状态采用ThreadLocal进行处理，让它们也成为线程安全的状态，因为有状态的Bean就可以在多线程中共享了。

一般的Web应用划分为展现层、服务层和持久层三个层次，在不同的层中编写对应的逻辑，下层通过接口向上层开放功能调用。在一般情况下，从接收请求到返回响应所经过的所有程序调用都同属于一个线程。

同一线程贯通三层这样你就可以根据需要，将一些非线程安全的变量以ThreadLocal存放，在同一次请求响应的调用线程中，所有关联的对象引用到的都是同一个变量。

下面的实例能够体现Spring对有状态Bean的改造思路：

TestDao：非线程安全

```
1
2 public class TestDao {
3     private Connection conn; // ①一个非线程安全的变量
4
5     public void addTopic() throws SQLException {
6         Statement stat = conn.createStatement(); // ②引用非线程安全变
量
7         // ...
8     }
}
```

由于conn是成员变量，因为addTopic()方法是非线程安全的，必须在使用时创建一个新TopicDao实例（非singleton）。下面使用ThreadLocal对conn这个非线程安全的“状态”进行改造：

TestDao：线程安全

```
1
2 public class TestDaoNew { // ①使用ThreadLocal保存Connection变量
3     private static ThreadLocal<Connection> connThreadLocal = Thread
Local.withInitial(Test::createConnection);
4
5     // 具体创建数据库连接的方法
6     private static Connection createConnection() {
7         Connection result = null;
8         /**
9          * create a real connection...
10         * such as :
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
11     * result = DriverManager.getConnection(dbUrl, dbUser, dbPw
d);
12     */
13     return result;
14 }
15
16 // ③直接返回线程本地变量
17 public static Connection getConnection() {
18     return connThreadLocal.get();
19 }
20
21 // 具体操作
22 public void addTopic() throws SQLException {
23     // ④从ThreadLocal中获取线程对应的Connection
24     Statement stat = getConnection().createStatement();
25     //....any other operation
26 }
27 }
```

不同的线程在使用TopicDao时，根据之前的深挖get具体操作，判断connThreadLocal.get()会去判断是否有map，没有则根据initivalValue创建一个Connection对象并添加到本地线程变量中，initivalValue对应的值也就是上述的lambda表达式对应的创建connection的方法返回的结果，下次get则由于已经有了，则会直接获取已经创建好的Connection，这样，就保证了不同的线程使用线程相关的Connection，而不会使用其它线程的Connection。因此，这个TopicDao就可以做到singleton共享了。

当然，这个例子本身很粗糙，将Connection的ThreadLocal直接放在DAO只能做到本DAO的多个方法共享Connection时不发生线程安全问题，但无法和其它DAO共用同一个Connection，要做到同一事务多DAO共享同一Connection，必须在一个共同的外部类使用ThreadLocal保存Connection。

ConnectionManager.java

```
1 public class ConnectionManager {
2
3     private static ThreadLocal<Connection> connectionHolder = Thr
eadLocal.withInitial(() -> {
4         Connection conn = null;
5         try {
6             conn = DriverManager.getConnection(
7                 "jdbc:mysql://127.0.0.1:3306/test",
8                 "root",
9                 "123456");
10        }
11        catch (SQLException e) {
12            e.printStackTrace();
13        }
14    });
15
16     public static Connection getConnection() {
17         return connectionHolder.get();
18     }
19
20     public static void closeConnection() {
21         connectionHolder.remove();
22     }
23 }
```

```
7         "jdbc:mysql://localhost:3306/test", "username"
8         "password");
9     } catch (SQLException e) {
10         e.printStackTrace();
11     }
12     return conn;
13 });
14
15     public static Connection getConnection() {
16         return connectionHolder.get();
17     }
18 }
```

线程隔离的秘密

秘密就就在于上述叙述的ThreadLocalMap这个类。ThreadLocalMap是ThreadLocal类的一个静态内部类，它实现了键值对的设置和获取（对比Map对象来理解），每个线程中都有一个独立的ThreadLocalMap副本，它所存储的值，只能被当前线程读取和修改。ThreadLocal类通过操作每一个线程特有的ThreadLocalMap副本，从而实现了变量访问在不同线程中的隔离。因为每个线程的变量都是自己特有的，完全不会有并发错误。还有一点就是，ThreadLocalMap存储的键值对中的键是this对象指向的ThreadLocal对象，而值就是你所设置的对象了。

为了加深理解，我们接着看上面代码中出现的getMap和createMap方法的实现：

```
1 /**
2  * Get the map associated with a ThreadLocal. Overridden in
3  * InheritableThreadLocal.
4  *
5  * @param t the current thread
6  * @return the map
7  */
8 ThreadLocalMap getMap(Thread t) {
9     return t.threadLocals;
10}
11
12 /**
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
13     * Create the map associated with a ThreadLocal. Overridden i
n
14     * InheritableThreadLocal.
15     *
16     * @param t the current thread
17     * @param firstValue value for the initial entry of the map
18     * @param map the map to store.
19     */
20 void createMap(Thread t, T firstValue) {
21     t.threadLocals = new ThreadLocalMap(this, firstValue);
22 }
```

小结

ThreadLocal是解决线程安全问题一个很好的思路，它通过为每个线程提供一个独立的变量副本解决了变量并发访问的冲突问题。在很多情况下，ThreadLocal比直接使用synchronized同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

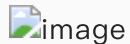
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



Java并发：单例模式的双检查

单例模式的双检查

单例类在Java开发者中非常常用，但是它给初级开发者们造成了很多挑战。他们所面对的其中一个关键挑战是，怎样确保单例类的行为是单例？也就是说，无论任何原因，如何防止单例类有多个实例。在整个应用生命周期中，要保证只有一个单例类的实例被创建，双重检查锁（Double checked locking of Singleton）是一种实现方法。顾名思义，在双重检查锁中，代码会检查两次单例类是否有已存在的实例，一次加锁一次不加锁，一次确保不会有多个实例被创建。顺便提一下，在JDK1.5中，Java修复了其内存模型的问题。在JDK1.5之前，这种方法会有问题。本文中，我们将会看到怎样用Java实现双重检查锁的单例类，为什么Java 5之前的版本双重检查锁会有问题，以及怎么解决这个问题。顺便说一下，这也是重要的面试要点，我曾经在金融业和服务业的公司面试被要求手写双重检查锁实现单例模式、相信我，这很棘手，除非你清楚理解了你在做什么。你也可以阅读我的完整列表“单例模式设计问题”来更好的准备面试。

为什么你需要双重检查锁来实现单例类？

一个常见情景，单例类在多线程环境中违反契约。如果你要一个新手写出单例模式，可能会得到下面的代码：

```
1 private static Singleton _instance;
2
3 public static Singleton getInstance() {
4     if (_instance == null) {
5         _instance = new Singleton();
6     }
7     return _instance;
8 }
```

然后，当你指出这段代码在超过一个线程并行被调用的时候会创建多个实例的问题时，他很可能会把整个getInstance()方法设为同步（synchronized），就像我们展示的第二段示例代码getInstanceTS()方法一样。尽管这样做到了线程安全，并且解决了多实例问题，但并不高效。在任何调用这个方法的时候，你都需要承受同步带来的性能开销，然而同步只在第一次调用的时候才被需要，也就是单例类实例创建的时候。这将促使我们使用双重检查锁模式（double checked locking pattern），一种只在临界区代码加锁的方法。程序员称其为双重检查锁，因为会有两次检查 _instance == null，一次不加锁，另一次在同步块上加锁。这就是使用Java双重检查锁的示例：

```
1 public static Singleton getInstanceDC() {  
2     if (_instance == null) { // Single Checked  
3         synchronized (Singleton.class) {  
4             if (_instance == null) { // Double checked  
5                 _instance = new Singleton();  
6             }  
7         }  
8     }  
9     return _instance;  
10 }
```

这个方法表面上看起来很完美，你只需要付出一次同步块的开销，但它依然有问题。除非你声明`instance`变量时使用了`volatile`关键字。没有`volatile`修饰符，可能出现Java中的另一个线程看到一个初始化了一半的`instance`的情况，但使用了`volatile`变量后，就能保证先行发生关系（happens-before relationship）。对于`volatile instance`，所有的写（write）都将先行发生于读（read），在Java 5之前不是这样，所以在这之前使用双重检查锁有问题。现在，有了先行发生的保障（happens-before guarantee），你可以安全地假设其会工作良好。另外，这不是创建线程安全的单例模式的最好方法，你可以使用枚举实现单例模式，这种方法在实例创建时提供了内置的线程安全。另一种方法是使用静态持有者模式（static holder pattern）。

```
1 /*  
2  * A journey to write double checked locking of Singleton class in Java.  
3 */  
4  
5 class Singleton {  
6  
7     private volatile static Singleton _instance;  
8  
9     private Singleton() {  
10         // preventing Singleton object instantiation from outside  
11     }  
12  
13     /*  
14      * 1st version: creates multiple instance if two thread access  
15      * this method simultaneously  
16      */
```

```
17
18     public static Singleton getInstance() {
19         if (_instance == null) {
20             _instance = new Singleton();
21         }
22         return _instance;
23     }
24
25     /*
26      * 2nd version : this definitely thread-safe and only
27      * creates one instance of Singleton on concurrent environment
28      * but unnecessarily expensive due to cost of synchronization
29      * at every call.
30     */
31
32     public static synchronized Singleton getInstanceTS() {
33         if (_instance == null) {
34             _instance = new Singleton();
35         }
36         return _instance;
37     }
38
39     /*
40      * 3rd version : An implementation of double checked locking o
41      * f Singleton.
42      * Intention is to minimize cost of synchronization and impro
43      * ve performance,
44      * by only locking critical section of code, the code which cr
45      * eates instance of Singleton class.
46      * By the way this is still broken, if we don't make _instance
47      * volatile, as another thread can
48      * see a half initialized instance of Singleton.
49     */
50
51     public static Singleton getInstanceDC() {
52         if (_instance == null) {
53             synchronized (Singleton.class) {
54                 if (_instance == null) {
55                     _instance = new Singleton();
56                 }
57             }
58         }
59         return _instance;
60     }
61
62     private Singleton() {
63     }
64 }
```

```
53     }
54 }
55     return _instance;
56 }
57 }
```

这就是本文的所有内容了。这是个用Java创建线程安全单例模式的有争议的方法，使用枚举实现单例类更简单有效。我并不建议你像这样实现单例模式，因为用Java有许多更好的方式。但是，这个问题有历史意义，也教授了并发是如何引入一些微妙错误的。正如之前所说，这是面试中非常重要的一点。在去参加任何Java面试之前，要练习手写双重检查锁实现单例类。这将增强你发现Java程序员们所犯编码错误的洞察力。另外，在现在的测试驱动开发中，单例模式由于难以被模拟其行为而被视为反模式（anti pattern），所以如果你是测试驱动开发的开发者，最好避免使用单例模式。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



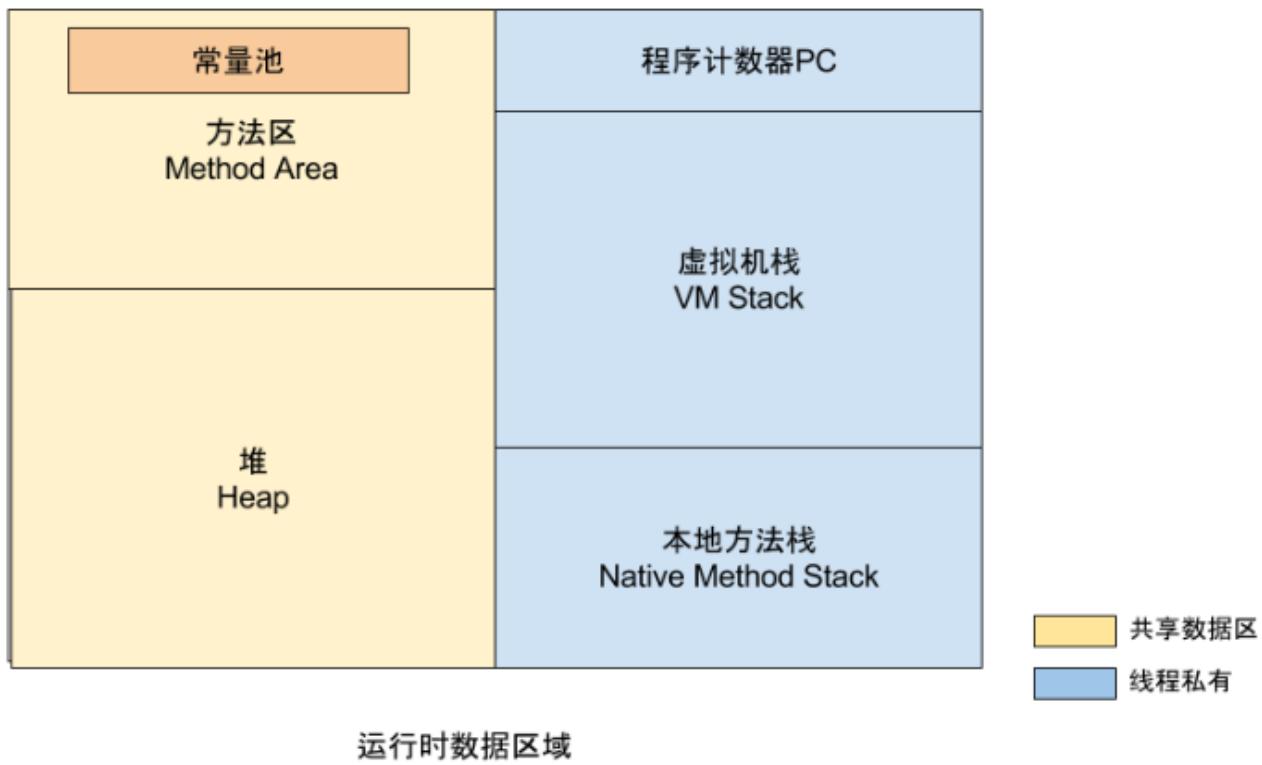
Java虚拟机：JVM内存模型

JVM内存模型

内存模型

Java内存模型，往往是指Java程序在运行时内存的模型，而Java代码是运行在Java虚拟机之上的，由Java虚拟机通过解释执行(解释器)或编译执行(即时编译器)来完成，故Java内存模型，也就是指Java虚拟机的运行时内存模型。

作为Java开发人员来说，并不需要像C/C++开发人员，需要时刻注意内存的释放，而是全权交给虚拟机去管理，那么有必要了解虚拟机的运行时内存是如何构成的。运行时内存模型，分为线程私有和共享数据区两大类，其中线程私有的数据区包含程序计数器、虚拟机栈、本地方法区，所有线程共享的数据区包含Java堆、方法区，在方法区内有一个常量池。



(1) 线程私有区：

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

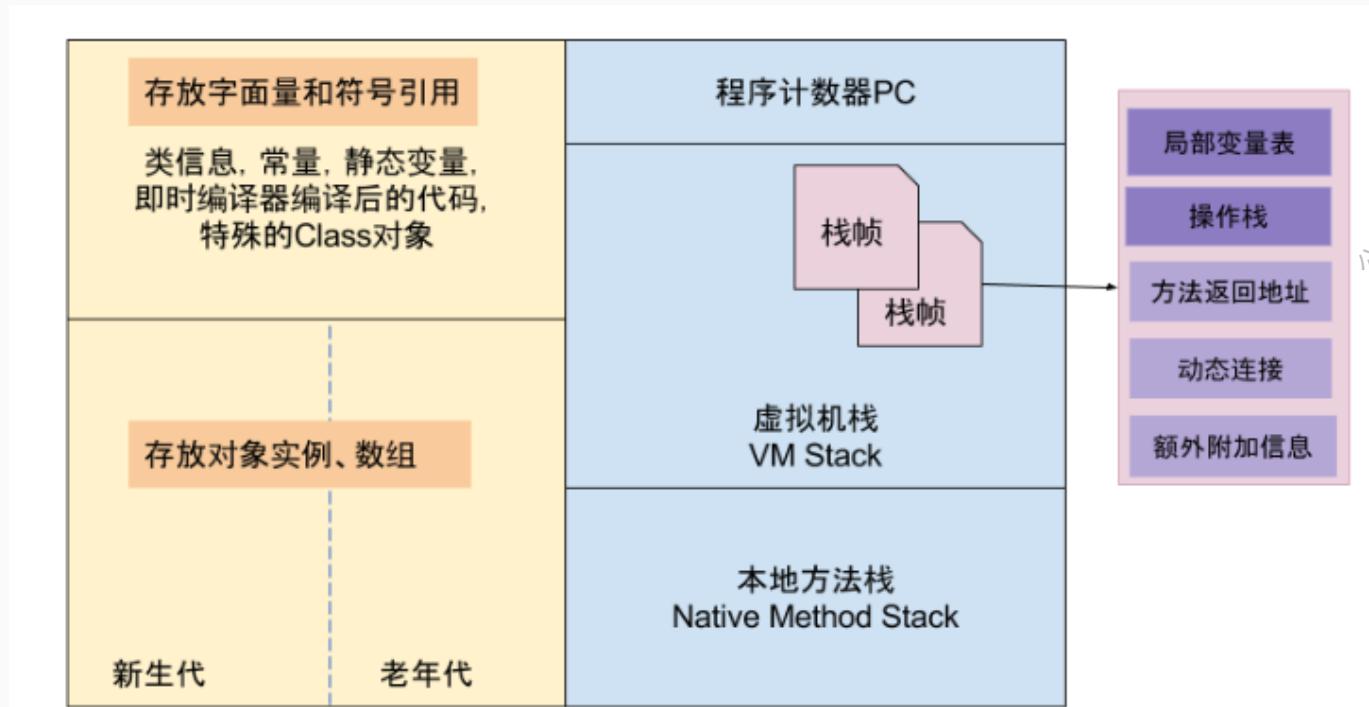
(2) 线程共享区：

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

对于大多数的程序员来说，Java内存比较流行的说法便是堆和栈，这其实是非常粗略的一种划分，这种划分的“堆”对应内存模型的Java堆，“栈”是指虚拟机栈，然而Java内存模型远比这更复杂，想深入了解Java的内存，还是有必要明白整个内存模型。

详细模型

运行时内存分为五大块区域（常量池属于方法区，算作一块区域），前面简要介绍了每个区域的功能，那接下来再详细说明每个区域的内容，Java内存总体结构图如下：



程序计数器PC

程序计数器PC，当前线程所执行的字节码行号指示器。每个线程都有自己计数器，是私有内存空间，该区域是整个内存中较小的一块。

当线程正在执行一个Java方法时，PC计数器记录的是正在执行的虚拟机字节码的地址；当线程正在执行的一个Native方法时，PC计数器则为空（Undefined）。

虚拟机栈

虚拟机栈，生命周期与线程相同，是Java方法执行的内存模型。每个方法(不包含native方法)执行的同时都会创建一个栈帧结构，方法执行过程，对应着虚拟机栈的入栈到出栈的过程。

栈帧(Stack Frame)结构

栈帧是用于支持虚拟机进行方法执行的数据结构，是属性运行时数据区的虚拟机站的栈元素。见上图，栈帧包括：

- 局部变量表 (locals大小，编译期确定)，一组变量存储空间，容量以slot为最小单位。
- 操作栈(stack大小，编译期确定)，操作栈元素的数据类型必须与字节码指令序列严格匹配
- 动态连接，指向运行时常量池中该栈帧所属方法的引用，为了动态连接使用。
 - 前面的解析过程其实是静态解析；
 - 对于运行期转化为直接引用，称为动态解析。
- 方法返回地址
 - 正常退出，执行引擎遇到方法返回的字节码，将返回值传递给调用者
 - 异常退出，遇到Exception，并且方法未捕捉异常，那么不会有任何返回值。
- 额外附加信息，虚拟机规范没有明确规定，由具体虚拟机实现。

异常(Exception)

Java虚拟机规范规定该区域有两种异常：

- StackOverFlowError：当线程请求栈深度超出虚拟机栈所允许的深度时抛出
- OutOfMemoryError：当Java虚拟机动态扩展到无法申请足够内存时抛出

本地方法栈

本地方法栈则为虚拟机使用到的Native方法提供内存空间，而前面讲的虚拟机栈为Java方法提供内存空间。有些虚拟机的实现直接把本地方法栈和虚拟机栈合二为一，比如非常典型的Sun HotSpot虚拟机。

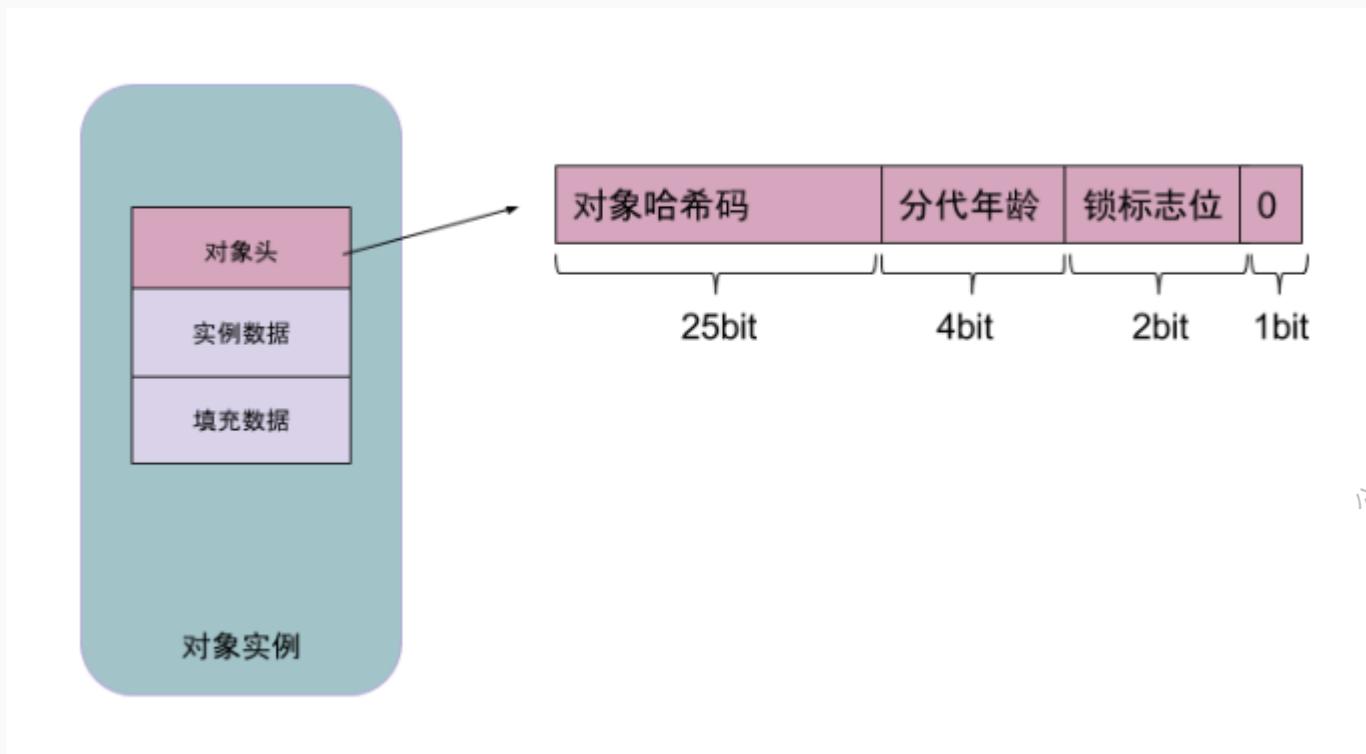
异常(Exception)：Java虚拟机规范规定该区域可抛出StackOverFlowError和OutOfMemoryError。

Java堆

Java堆，是Java虚拟机管理的最大的一块内存，也是GC的主战场，里面存放的是几乎所有的对象实例和数组数据。JIT编译器有栈上分配、标量替换等优化技术的实现导致部分对象实例数据不存在Java堆，而是栈内存。

- 从内存回收角度，Java堆被分为新生代和老年代；这样划分的好处是为了更快的回收内存；
- 从内存分配角度，Java堆可以划分出线程私有的分配缓冲区(Thread Local Allocation Buffer,TLAB)；这样划分的好处是为了更快的分配内存；

对象创建的过程是在堆上分配着实例对象，那么对象实例的具体结构如下：



对于填充数据不是一定存在的，仅仅是为了字节对齐。HotSpot VM的自动内存管理要求对象起始地址必须是8字节的整数倍。对象头本身是8的倍数，当对象的实例数据不是8的倍数，便需要填充数据来保证8字节的对齐。该功能类似于高速缓存行的对齐。

另外，关于在堆上内存分配是并发进行的，虚拟机采用CAS加失败重试保证原子操作，或者是采用每个线程预先分配TLAB内存。

异常(Exception)：Java虚拟机规范规定该区域可抛出OutOfMemoryError。

方法区

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

方法区主要存放的是已被虚拟机加载的类信息、常量、静态变量、编译器编译后的代码等数据。GC在该区域出现的比较少。

异常(Exception)：Java虚拟机规范规定该区域可抛出OutOfMemoryError。

运行时常量池

运行时常量池也是方法区的一部分，用于存放编译器生成的各种字面量和符号引用。运行时常量池除了编译期产生的Class文件的常量池，还可以在运行期间，将新的常量加入常量池，比较常见的是String类的intern()方法。

- 字面量：与Java语言层面的常量概念相近，包含文本字符串、声明为final的常量值等。
 - 符号引用：编译语言层面的概念，包括以下3类：
 - 类和接口的全限定名
 - 字段的名称和描述符
 - 方法的名称和描述符
- 但是该区域不会抛出OutOfMemoryError异常。

原文链接

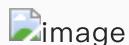
<http://gityuan.com/2016/01/09/java-memory/>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

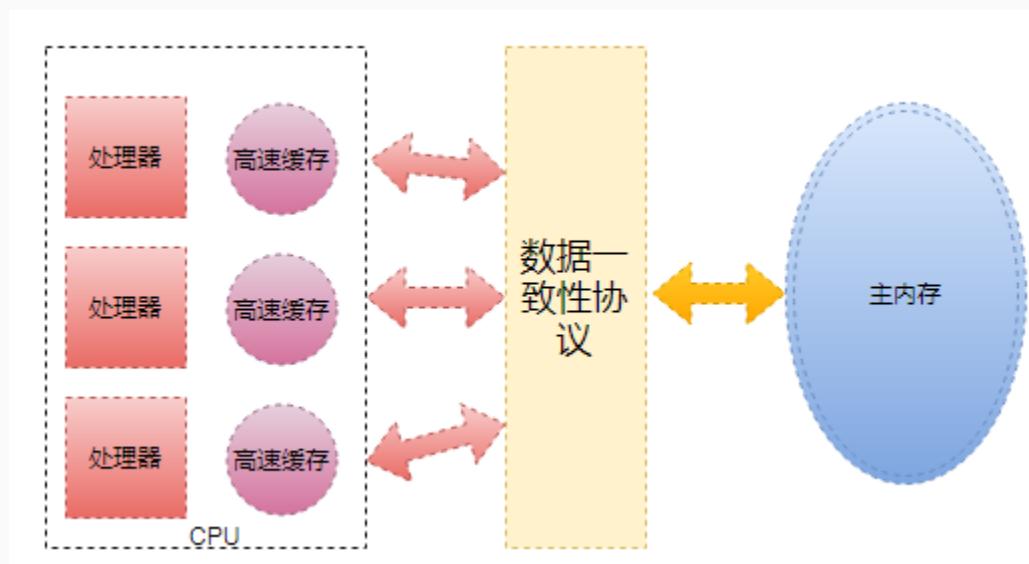
公众号：方志朋

Java虚拟机：JVM内存模型和volatile详解

JVM内存模型和volatile详解

Java内存模型

随着计算机的CPU的飞速发展，CPU的运算能力已经远远超出了从主内存（运行内存）中读取的数据的能力，为了解决这个问题，CPU厂商设计出了CPU内置高速缓存区。高速缓存区的加入使得CPU在运算的过程中直接从高速缓存区读取数据，在一定程度上解决了性能的问题。但也引起了另外一个问题，在CPU多核的情况下，每个处理器都有自己的缓存区，数据如何保持一致性。为了保证多核处理器的数据一致性，引入多处理器的数据一致性的协议，这些协议包括MOSI、Synapse、Firely、DragonProtocol等。



JVM在执行多线程任务时，共享数据保存在主内存中，每一个线程（执行在不同的处理器）有自己的高速缓存，线程对共享数据进行修改的时候，首先是从主内存拷贝到线程的高速缓存，修改之后，然后从高速缓存再拷贝到主内存。当有多个线程执行这样的操作的时候，会导致共享数据出现不可预期的错误。

举个例子：

```
i++;//操作
```

这个i操作，线程首先从主内存读取i的值，比如i=0，然后复制到自己的高速缓存区，进行i操作，最后将操作后的结果从高速缓存区复制到主内存中。如果是两个线程通过操作i++,预期的结果是2。这时结果真的为2吗？答案是否定的。线程1读取主内存的i=0,复制到自己的高速缓存区，这时线程2也读取i=0,复制到自己的高速缓存区，进行i++操作，怎么最终得到的结构为1，而不是2。

为了解决缓存不一致的问题，有两种解决方案：

- 在总线加锁，即同时只有一个线程能执行`i++`操作（包括读取、修改等）。
- 通过缓存一致性协议

第一种方式就没什么好说的，就是同步代码块或者同步方法。也就只能一个线程能进行对共享数据的读取和修改，其他线程处于线程阻塞状态。

第二种方式就是缓存一致性协议，比如Intel 的MESI协议，它的核心思想就是当某个处理器写变量的数据，如果其他处理器也存在这个变量，会发出信号量通知该处理器高速缓存的数据设置为无效状态。当其他处理需要读取该变量的时候，会让其重新从主内存中读，然后再复制到高速缓存区。

并发编程的概念

并发编程的有三个概念，包括原子性、可见性、有序性。

原子性

原子性是指，操作为原子性的，要么成功，要么失败，不存在第三种情况。比如：

```
1 String s="abc";
```

这个复杂操作是原子性的。再比如：

```
1 int i=0;
2 i++;
```

`i=0`这是一个赋值操作，这一步是原子性操作；那么`i++`是原子性操作吗？当然不是，首先它需要读取`i=0`，然后需要执行运算，写入`i`的新值1，它包含了读取和写入两个步骤，所以不是原子性操作。

可见性

可见性是指共享数据的时候，一个线程修改了数据，其他线程知道数据被修改，会重新读取最新的主存的数据。

举个例子：

```
1 i=0;//主内存  
2  
3 i++; //线程1  
4  
5 j=i; //线程2
```

线程1修改了i值，但是没有将i值复制到主内存中，线程2读取i的值，并将i的值赋值给j，我们期望j=1，但是由于线程1修改了，没有来得及复制到主内存中，线程2读取了i，并赋值给j，这时j的值为0。
也就是线程i值被修改，其他线程并不知道。

有序性

是指代码执行的有序性，因为代码有可能发生指令重排序（Instruction Reorder）。

Java语言提供了 volatile 和 synchronized 两个关键字来线程代码操作的有序性，volatile 是因为其本身包含“禁止指令重排序”的语义，synchronized 在单线程中执行代码，无论指令是否重排，最终的执行结果是一致的。

volatile详解

volatile关键字作用

被volatile关键字修饰变量，起到了2个作用：

1. 某个线程修改了被volatile关键字修饰变量是，根据数据一致性的协议，通过信号量，更改其他线程的高速缓存中volatile关键字修饰变量状态为无效状态，其他线程如果需要重写读取该变量会再次从主内存中读取，而不是读取自己的高速缓存中的。
2. 被volatile关键字修饰变量不会指令重排序。

volatile能够保证可见性和防止指令重排

在Java并发编程实战一书中这样

```
1 public class NoVisibility {
2     private static boolean ready;
3     private static int a;
4
5     public static void main(String[] args) throws InterruptedException {
6         new ReadThread().start();
7         Thread.sleep(100);
8         a = 32;
9         ready = true;
10
11    }
12
13
14    private static class ReadThread extends Thread {
15        @Override
16        public void run() {
17            while (!ready) {
18                Thread.yield();
19            }
20            System.out.println(a);
21        }
22    }
23 }
```

公众号：方志朋

在上述代码中，有可能（概率非常小，但是有这种可能性）永远不会打印a的值，因为线程ReadThread读取了主内存的ready为false,主线程虽然更新了ready，但是ReadThread的高速缓存中并没有更新。

另外：

```
a = 32;
ready = true;
```

这两行代码有可能发生指令重排。也就是可以打印出a的值为0。

如果在变量加上volatile关键字，可以防止上述两种不正常的情况的发生。

volatile不能保证原子性

首先用一段代码测试下，开起了10个线程，这10个线程共享一个变量inc（被volatile修饰），并在每个线程循环1000次对inc进行inc++操作。我们预期的结果是10000。

```
1 public class VolatileTest {  
2  
3  
4     public volatile int inc = 0;  
5  
6     public void increase() {  
7         inc++;  
8     }  
9  
10    public static void main(String[] args) throws InterruptedException {  
11        final VolatileTest test = new VolatileTest();  
12        for (int i = 0; i < 10; i++) {  
13            new Thread(() -> {  
14                for (int j = 0; j < 1000; j++)  
15                    test.increase();  
16            }).start();  
17        }  
18        //保证前面的线程都执行完  
19        Thread.sleep(3000);  
20        System.out.println(test.inc);  
21    }  
22  
23 }
```

多次运行main函数，你会发现结果永远都不会为10000，都是小于10000。可能有这样的疑问，volatile保证了共享数据的可见性，线程1修改了inc变量线程2会重新从主内存中重新读，这样就能保证inc++的正确性了啊，可为什么没有得到我们预期的结果呢？

在之前已经讲述过inc++这样的操作不是一个原子性操作，它分为读、加加、写。一种情况，当线程1读取了inc的值，还没有修改，线程2也读取了，线程1修改完了，通知线程2将线程的缓存的inc的值无效需要重读，可这时它不需要读取inc，它仍执行写操作，然后赋值给主线程，这时数据就会出现问题。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

所以volatile不能保证原子性。这时需要用锁来保证，在increase方法加上synchronized，重新运行打印的结果为10000。

```
1 public synchronized void increase() {  
2     inc++;  
3 }
```

volatile的使用场景

状态标记

volatile最常见的使用场景是状态标记，如下：

```
1 private volatile boolean asleep ;  
2  
3 //线程1  
4  
5 while(!asleep){  
6     countSheep();  
7 }  
8  
9 //线程2  
10 asleep=true;
```

防止指令重排

```
1 volatile boolean init = false;  
2 //线程1:  
3 context = loadContext();  
4 init = true;  
5 //上面两行代码如果不使用volatile修饰，可能会发生指令重排，导致报错  
6  
7 //线程2:  
8 while(!init){
```

```
9 sleep()  
10 }  
11 doSomethingWithConfig(context);
```

happens-before

从jdk5开始，java使用新的JSR-133内存模型，基于happens-before的概念来阐述操作之间的内存可见性。

在JMM中，如果一个操作的执行结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系，这个的两个操作既可以在同一个线程，也可以在不同的两个线程中。

与程序员密切相关的happens-before规则如下：

- 程序顺序规则：一个线程中的每个操作，happens-before于该线程中任意的后续操作。
- 监视器锁规则：对一个锁的解锁操作，happens-before于随后对这个锁的加锁操作。
- volatile域规则：对一个volatile域的写操作，happens-before于任意线程后续对这个volatile域的读。
- 传递性规则：如果 A happens-before B，且 B happens-before C，那么A happens-before C。

注意：两个操作之间具有happens-before关系，并不意味前一个操作必须要在后一个操作之前执行！仅仅要求前一个操作的执行结果，对于后一个操作是可见的，且前一个操作按顺序排在后一个操作之前。

参考资料

《Java 并发编程实战》

《深入理解JVM》

海子的博客：<http://www.cnblogs.com/dolphin0520/p/3920373.html>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

Java虚拟机：垃圾收集算法

垃圾收集算法

垃圾回收机制的意义

Java语言中一个显著的特点就是引入了垃圾回收机制，使c++程序员最头疼的内存管理的问题迎刃而解，它使得Java程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java中的对象不再有“作用域”的概念，只有对象的引用才有“作用域”。垃圾回收可以有效的防止内存泄露，有效的使用空闲的内存。

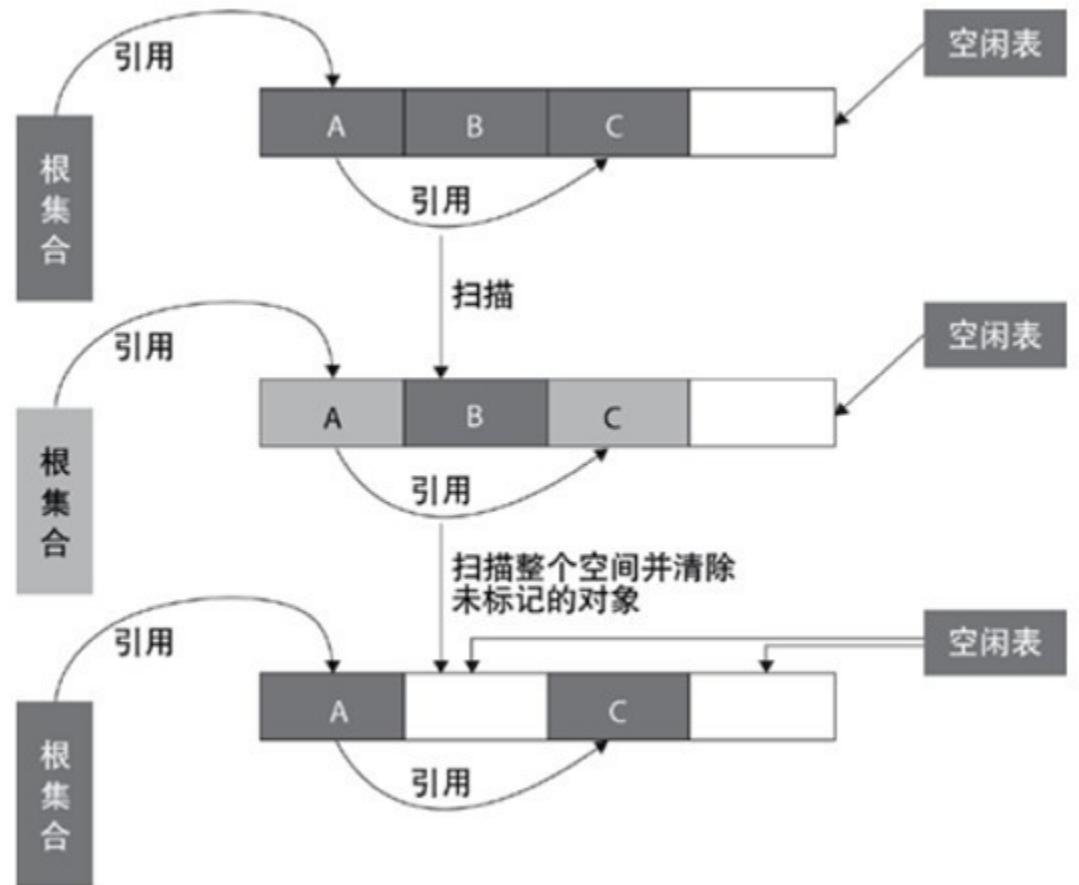
ps:内存泄露是指该内存空间使用完毕之后未回收，在不涉及复杂数据结构的一般情况下，Java 的内存泄露表现为一个内存对象的生命周期超出了程序需要它的时间长度，我们有时也将其称为“对象游离”。

垃圾回收算法

标记清除算法

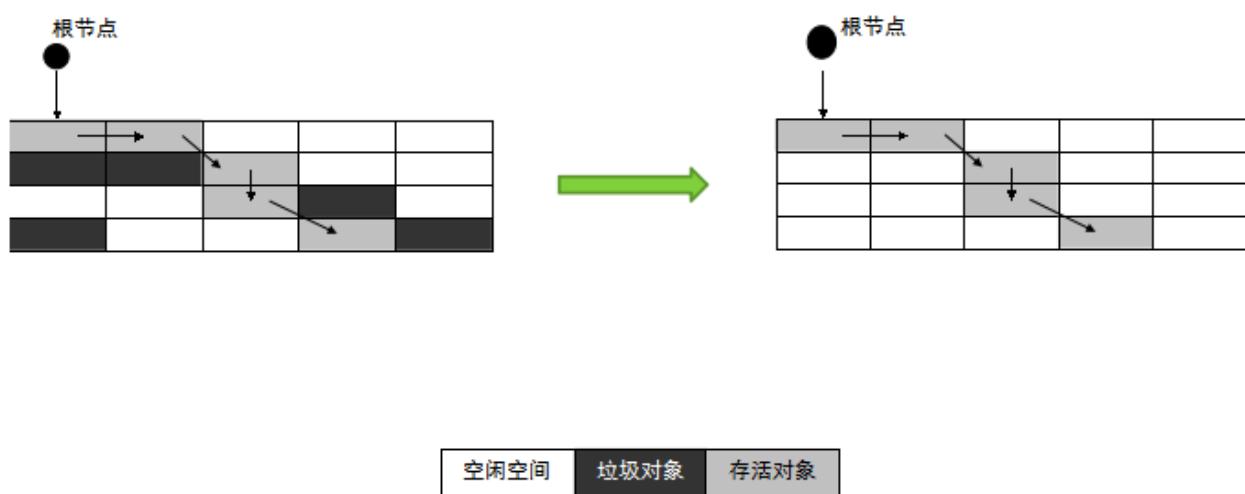
标记-清除算法分为标记和清除两个阶段。该算法首先从根集合进行扫描，对存活的对象对象标记，标记完毕后，再扫描整个空间中未被标记的对象并进行回收，如下图所示。

公众号：方志朋



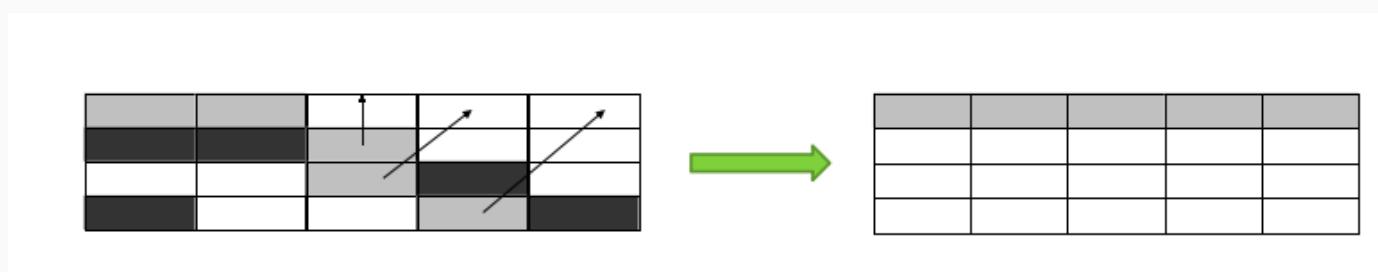
标记–清除算法的主要不足有两个：

- 效率问题：标记和清除两个过程的效率都不高；
- 空间问题：标记–清除算法不需要进行对象的移动，并且仅对不存活的对象进行处理，因此标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

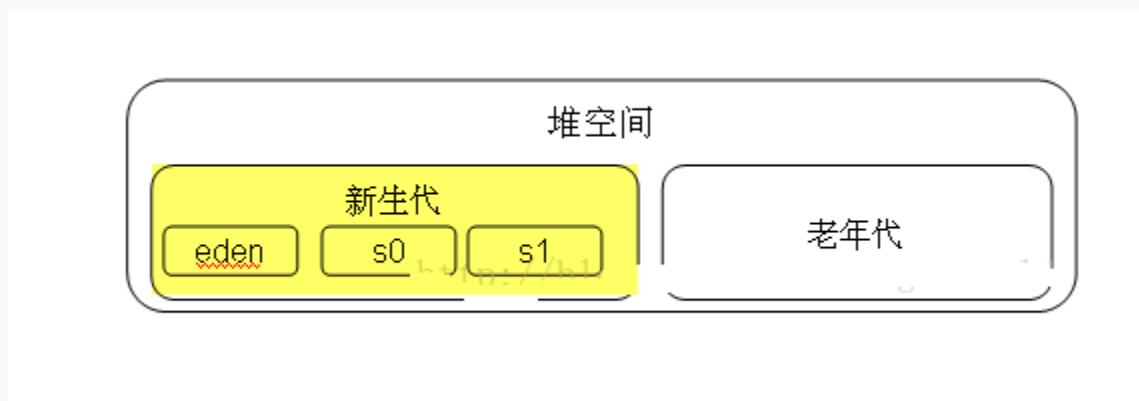


复制算法

复制算法将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这种算法适用于对象存活率低的场景，比如新生代。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。该算法示意图如下所示：



事实上，现在商用的虚拟机都采用这种算法来回收新生代。因为研究发现，新生代中的对象每次回收都基本上只有10%左右的对象存活，所以需要复制的对象很少，效率还不错。正如在博文《JVM 内存模型概述》中介绍的那样，实践中会将新生代内存分为一块较大的Eden空间和两块较小的Survivor空间（如下图所示），每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还活着的对象一次地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是 8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%（ $80\%+10\%$ ），只有10% 的内存会被“浪费”。

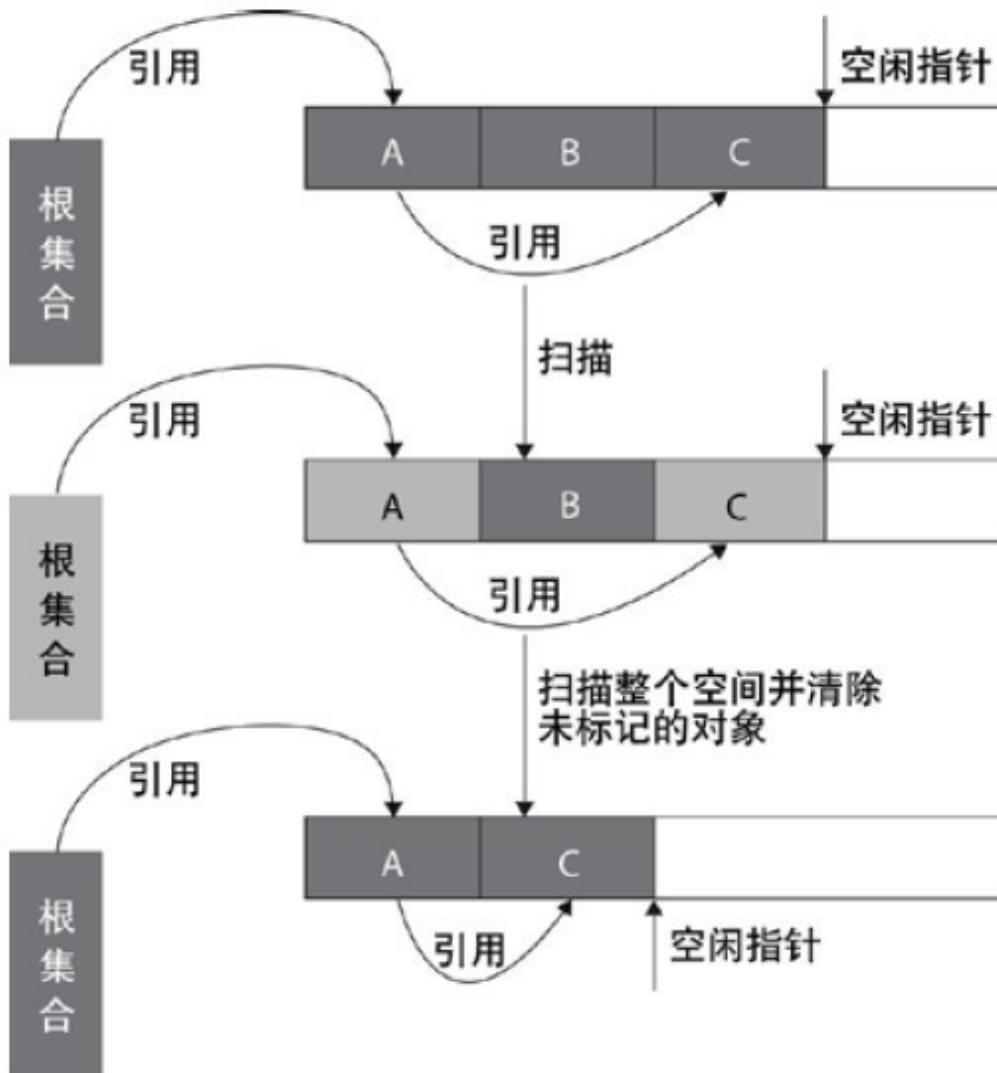


标记整理算法

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想要浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。标记整理算法的标记过程类似标记清除算法，但后续步骤

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存，类似于磁盘整理的过程，该垃圾回收算法适用于对象存活率高的场景（老年代），其作用原理如下图所示。



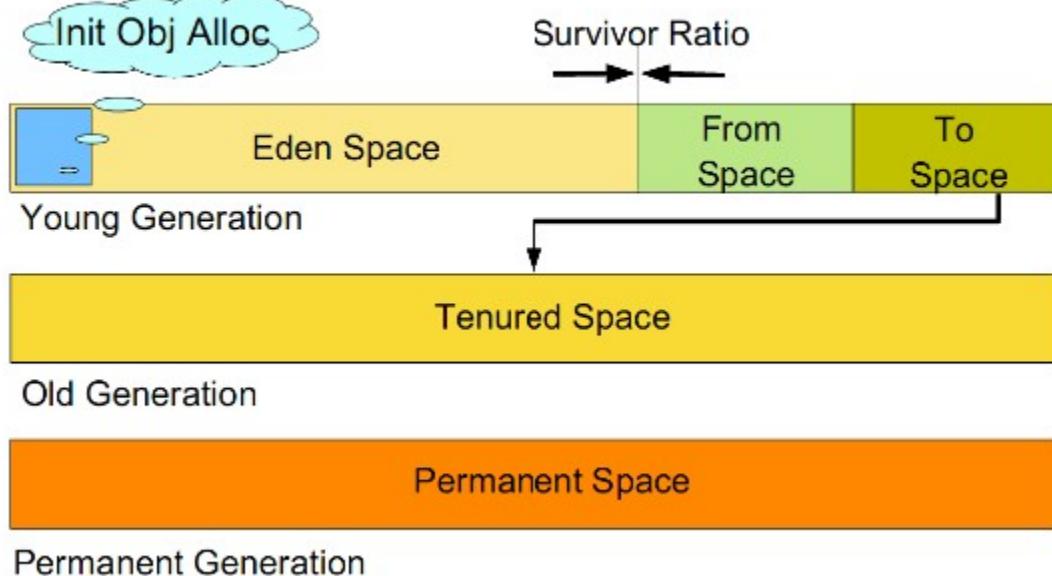
公众号: 方志朋

标记整理算法与标记清除算法最显著的区别是：标记清除算法不进行对象的移动，并且仅对不存活的对象进行处理；而标记整理算法会将所有的存活对象移动到一端，并对不存活对象进行处理，因此其不会产生内存碎片。标记整理算法的作用示意图如下：



分代收集算法

对于一个大型的系统，当创建的对象和方法变量比较多时，堆内存中的对象也会比较多，如果逐一分析对象是否该回收，那么势必造成效率低下。分代收集算法是基于这样一个事实：不同的对象的生命周期(存活情况)是不一样的，而不同生命周期的对象位于堆中不同的区域，因此对堆内存不同区域采用不同的策略进行回收可以提高 JVM 的执行效率。当代商用虚拟机使用的都是分代收集算法：新生代对象存活率低，就采用复制算法；老年代存活率高，就用标记清除算法或者标记整理算法。Java堆内存一般可以分为新生代、老年代和永久代三个模块，如下图所示：



新生代 (Young Generation)

新生代的目标就是尽可能快速的收集掉那些生命周期短的对象，一般情况下，所有新生成的对象首先都是放在新生代的。新生代内存按照 8:1:1 的比例分为一个eden区和两个survivor(survivor0, survivor1)区，大部分对象在Eden区中生成。在进行垃圾回收时，先将eden区存活对象复制到survivor0区，然后清空eden区，当这个survivor0区也满了时，则将eden区和survivor0区存活对象复制到survivor1区，然后清空eden区和这个survivor0区，此时survivor0区是空的，然后交换survivor0区和survivor1区的角色（即下次

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

垃圾回收时会扫描Eden区和survivor1区），即保持survivor0区为空，如此往复。特别地，当survivor1区也不足以存放eden区和survivor0区的存活对象时，就将存活对象直接存放到老年代。如果老年代也满了，就会触发一次FullGC，也就是新生代、老年代都进行回收。注意，新生代发生的GC也叫做MinorGC，MinorGC发生频率比较高，不一定等 Eden区满了才触发。

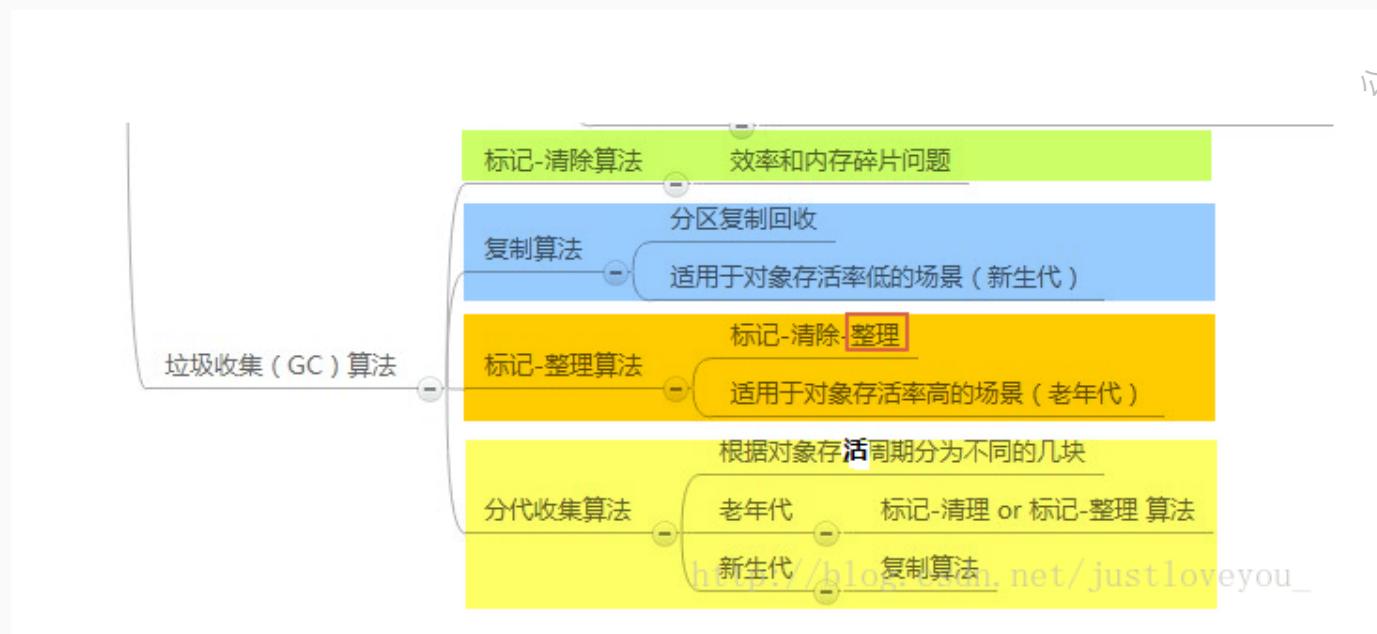
老年代（Old Generation）

老年代存放的都是一些生命周期较长的对象，就像上面所叙述的那样，在新生代中经历了N次垃圾回收后仍然存活的对象就会被放到老年代中。此外，老年代的内存也比新生代大很多(大概比例是1:2)，当老年代满时会触发Major GC(Full GC)，老年代对象存活时间比较长，因此FullGC发生的频率比较低。

永久代（Permanent Generation）

永久代主要用于存放静态文件，如Java类、方法等。永久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如使用反射、动态代理、CGLib等bytecode框架时，在这种时候需要设置一个比较大的永久代空间来存放这些运行过程中新增的类。

小结



由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。垃圾回收有两种类型，Minor GC 和 Full GC。

- Minor GC：对新生代进行回收，不会影响到老年代。因为新生代的 Java 对象大多死亡频繁，所以 Minor GC 非常频繁，一般在这里使用速度快、效率高的算法，使垃圾回收能尽快完成。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

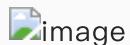
- Full GC：也叫 Major GC，对整个堆进行回收，包括新生代和老年年代。由于Full GC需要对整个堆进行回收，所以比Minor GC要慢，因此应该尽可能减少Full GC的次数，导致Full GC的原因包括：老年年代被写满、永久代（Perm）被写满和System.gc()被显式调用等。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

Java虚拟机：垃圾收集器和内存分配策略

垃圾收集器和内存分配策略

垃圾收集器

说垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。下图展示了7种作用于不同分代的收集器，其中用于回收新生代的收集器包括Serial、PraNew、Parallel Scavenge，回收老年代的收集器包括Serial Old、Parallel Old、CMS，还有用于回收整个Java堆的G1收集器。不同收集器之间的连线表示它们可以搭配使用。

虚拟机包含了所有的收集器如图所示：

公众号：方志朋

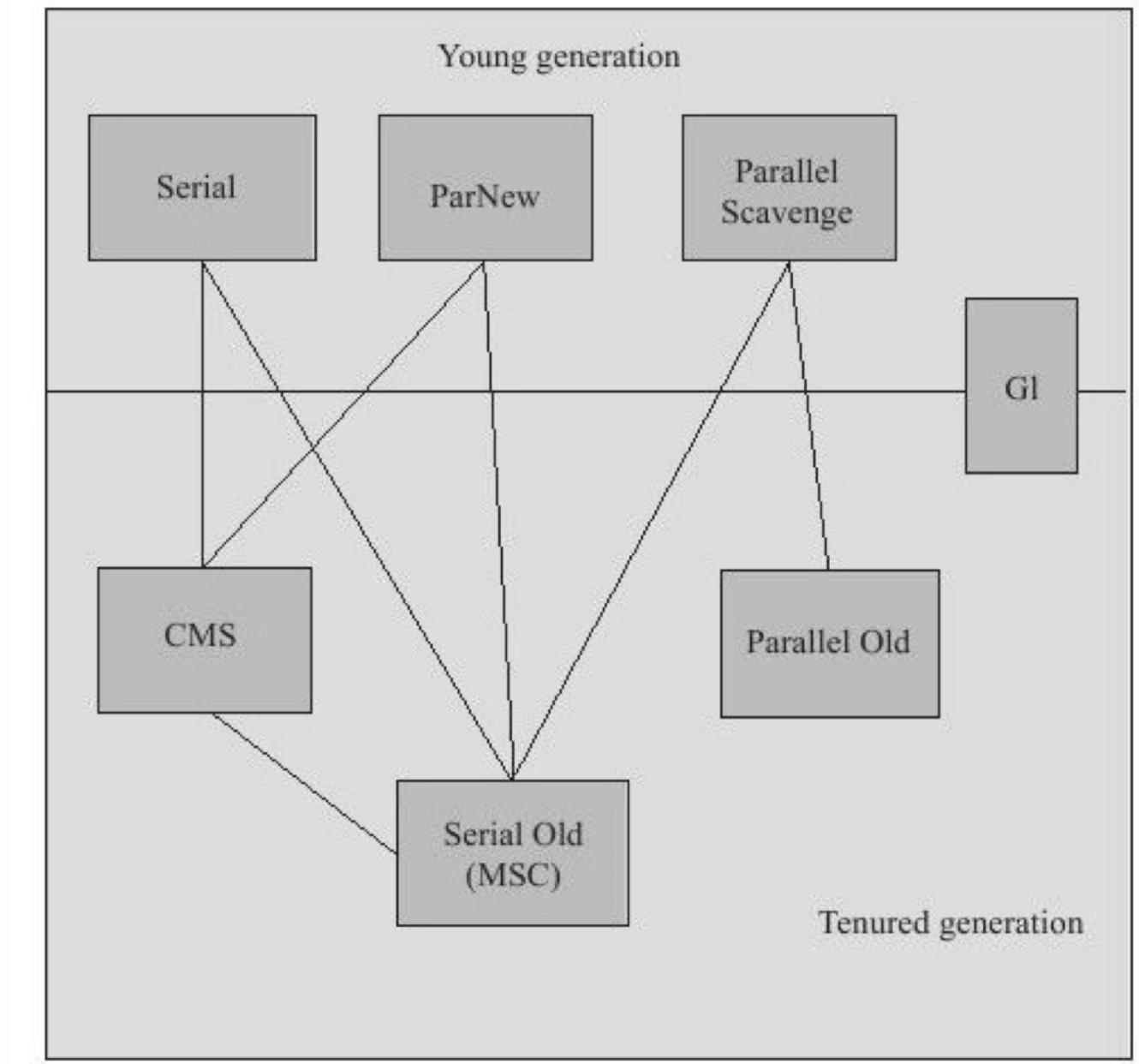


图 3-5 HotSpot虚拟机的垃圾收集器[1]

图中展示了7种作用不同分代的收集器，如果两个收集器之间存在连线，就说明它们可以搭配使用。

Serial收集器

复制算法

Serial收集器曾经是虚拟机新生代的唯一选择。“单线程”收集，并“stop the world”

简单高效

ParNew收集器

复制算法

ParNew收集器是Serial收集器的多线程版本。除了Serial外，只有它能于Cms收集器配合。

ParNew收集器在单CPU的环境中绝对不会比Serial收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个CPU的环境中都不能百分之百地保证可以超越Serial收集器。当然，随着可以使用的CPU的数量的增加，它对于GC时系统资源的有效利用还是很有好处的。它默认开启的收集线程数与CPU的数量相同，在CPU非常多（譬如32个，现在CPU动辄就4核加超线程，服务器超过32个逻辑CPU的情况越来越多了）的环境下，可以使用-XX: ParallelGCThreads参数来限制垃圾收集的线程数。

在谈垃圾收集器上下文的时：

- 并行 (parallel): 指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
- 并发 (concurrent): 指用户线程与垃圾收集线程同时执行（但不一定并行的，可能交替执行），用户程序继续运行，而垃圾收集程序运行在另一个cpu上。

Parallel Scavenge收集器

Parallel Scavenge收集器是一个新生代收集器，它也是采用复制算法，是并行的多线程收集器。

它的关注点在于吞吐量，而其他收集器关注缩短停顿时间。

吞吐量=运行用户代码时间 / (运行用户代码时间+垃圾收集时间)

Serial old收集器

Serial old是Serial收集器的老年代版本，它是一个单线程收集器，使用“标记整理”算法。

两种用途：与Parallel scavenger收集器搭配使用，另外作为CMS收集器的后背预案，在并发收集发生Concurrent Mode Failure时使用。

公众号：方志朋

Parallel Old收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在JDK 1.6中才开始提供的，在此之前，新生代的Parallel Scavenge收集器一直处于比较尴尬的状态。原因是，如果新生代选择了Parallel Scavenge收集器，老年代除了Serial Old (PS MarkSweep) 收集器外别无选择（还记得上面说过Parallel Scavenge收集器无法与CMS收集器配合工作吗？）。由于老年代Serial Old收集器在服务端应用性能上的“拖累”，使用了Parallel Scavenge收集器也未必能在整体应用上获得吞吐量最大化的效果，由于单线程的老年代收集中无法充分利用服务器多CPU的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有ParNew加CMS的组合“给力”。

CMS收集器（标记清除算法）

CMS (Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为为目标的收集器。

- 初始标记
- 并发标记
- 重新标记
- 并发清除

其中，初始标记、重新标记这两个步骤仍然需要“stop the world”。初始标记仅仅只是标记一下GC roots能够关联到的对象，速度很快。并发标记是进行GC roots Tracing过程。重新标记，修正并发标记期间用户程序就行运行而导致标记产生变动的那一部分对象。

三个缺点：

- 对CPU资源非常敏感
- CMS收集器无法处理浮动垃圾 (Floating Garbage)，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在JDK1.5的默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX: CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次数从而获取更好的性能，在JDK 1.6中，CMS收集器的启动阈值已经提升至92%。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- CMS 是标记清除算法，会有大量的空间碎片，但是当无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。

G1收集器

采用标记-整理算法

G1收集器的特短：

- 并行与并发
- 分代收集（与其他收集器一样）
- 空间整理（标记-整理算法）
- 可预测的停顿

如果不计算维护Remembered Set的操作，G1收集器的运作大致可划分为以下几个步骤：

- 初始标记（Initial Marking）
- 并发标记（Concurrent Marking）
- 最终标记（Final Marking）
- 筛选回收（Live Data Counting and Evacuation）

内存分配与回收策略

Java 技术体系中所提倡的自动内存管理最终可以归结为自动化地解决两个问题：给对象分配内存以及回收分配对对象的内存。

对象优先在Eden分配

大多数情况下，对象在新生代Eden区中分配。当Eden区没有足够空间进行分配时，虚拟机将发起一次Minor Gc.

大对象直接进入老年代

所谓大对象是指，需要大量连续内存空间的Java对象，最典型的大对象就是字节数组等。

长期存活的对象将进入老年代

虚拟机给每个对象定义了一个对象年龄的计数器。如果对象在Eden出生并经过Minor Gc仍然存活，并且能够被Survivor容器容纳，将被移到Survivor空间中，并且年龄设置为1.对象在Survivor区中每“熬过”一次Minor Gc，年龄就增加1岁，当它的年龄达到一定程度（默认为15岁），就被晋升到老年期中。

动态对象年龄判断

如果在Survivor 空间中相同年龄的对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年期，无需等到MaxTenuringThreshold中要求的年龄。

空间分配担保

在发生Minor Gc之前，虚拟机会先检查老年期最大可用的连续空间是否大于新生代所有对象的总空间，如果这个条件成立，那么Minor Gc可用确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年期最大空间是否大于历次晋升到老年期对象的平均大小，如果大于，将尝试一次 Minor Gc；如果小于，或者HandlePromotionFailure设置不允许冒险，那么这时要进行一次Full Gc.

下面解释一下“冒险”是冒了什么风险，前面提到过，新生代使用复制收集算法，但为了内存利用率，只使用其中一个Survivor空间来作为轮换备份，因此当出现大量对象在Minor GC后仍然存活的情况（最极端的情况就是内存回收后新生代中所有对象都存活），就需要老年期进行分配担保，把Survivor无法容纳的对象直接进入老年期。与生活中的贷款担保类似，老年期要进行这样的担保，前提是老年期本身还有容纳这些对象的剩余空间，一共有多少对象会活下来在实际完成内存回收之前是无法明确知道的，所以只好取之前每一次回收晋升到老年期对象容量的平均大小值作为经验值，与老年期的剩余空间进行比较，决定是否进行Full Gc来让老年期腾出更多的空间。

取平均值进行比较其实仍然是一种动态概率的手段，也就是说，如果某次Minor GC存活后的对象突增，远远高于平均值的话，依然会导致担保失败（Handle Promotion Failure）。如果出现了HandlePromotionFailure失败，那就只好在失败后重新发起一次Full GC。虽然担保失败时绕的圈子是最大的，但大部分情况下都还是会将HandlePromotionFailure开关打开，避免Full Gc过于频繁。

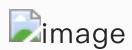
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

Java虚拟机：怎么确定对象已经死了？

怎么确定对象已经死了？

怎么确定对象已经死了？怎么确定一个对象已经死了？

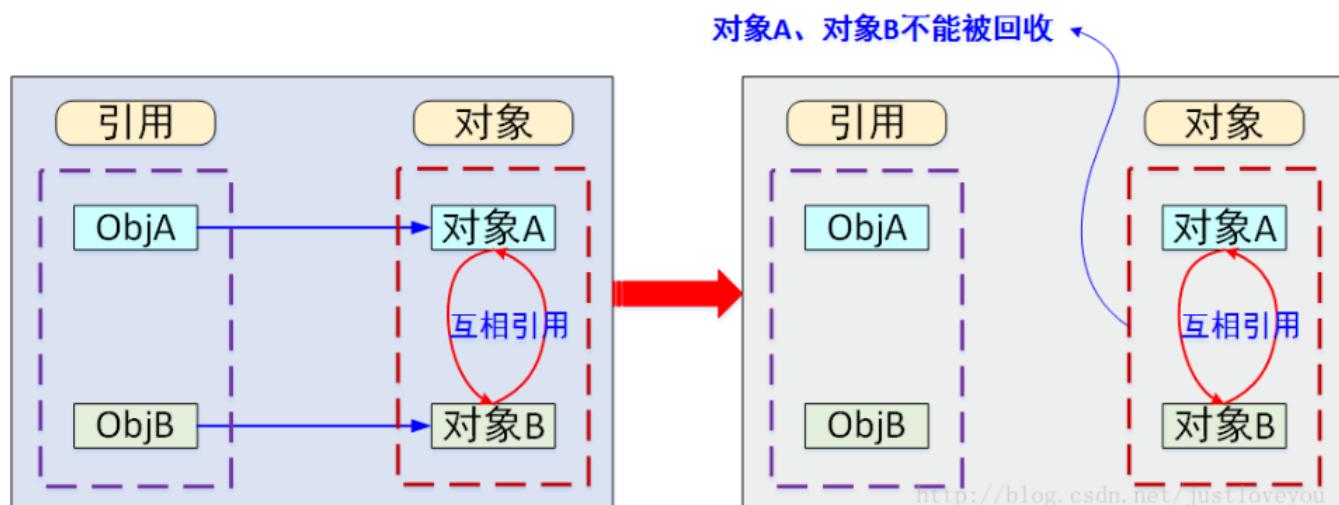
引用计数算法

给对象中添加一个引用计数器，每当有个地方引用它，计数器值就加1，引用失效，计数器减1，任何时刻计数器为0的对象就不能再应用了。

很难解决对象之间的相互循环引用。

引用计数收集器可以很快的执行，并且交织在程序运行中，对程序需要不被长时间打断的实时环境比较有利，但其很难解决对象之间相互循环引用的问题。如下面的程序和示意图所示，对象objA和objB之间的引用计数永远不可能为0，那么这两个对象就永远不能被回收。

引用计数算法很难解决对象之间相互循环引用的问题



```
1 public class ReferenceCountingGC {  
2  
3     public Object instance = null;  
4  
5     public static void testGC(){
```

```
6
7         ReferenceCountingGC objA = new ReferenceCountingGC ()
8
9
10        ReferenceCountingGC objB = new ReferenceCountingGC ()
11
12
13
14        // 对象之间相互循环引用，对象objA和objB之间的引用计数永远不可能
15        // 为 0
16        objB.instance = objA;
17        objA.instance = objB;
18
19
20        objA = null;
21        objB = null;
22
23
24        System.gc();
25    }
```

上述代码最后面两句将objA和objB赋值为null，也就是说objA和objB指向的对象已经不可能再被访问，但是由于它们互相引用对方，导致它们的引用计数器都不为0，那么垃圾收集器就永远不会回收它们。

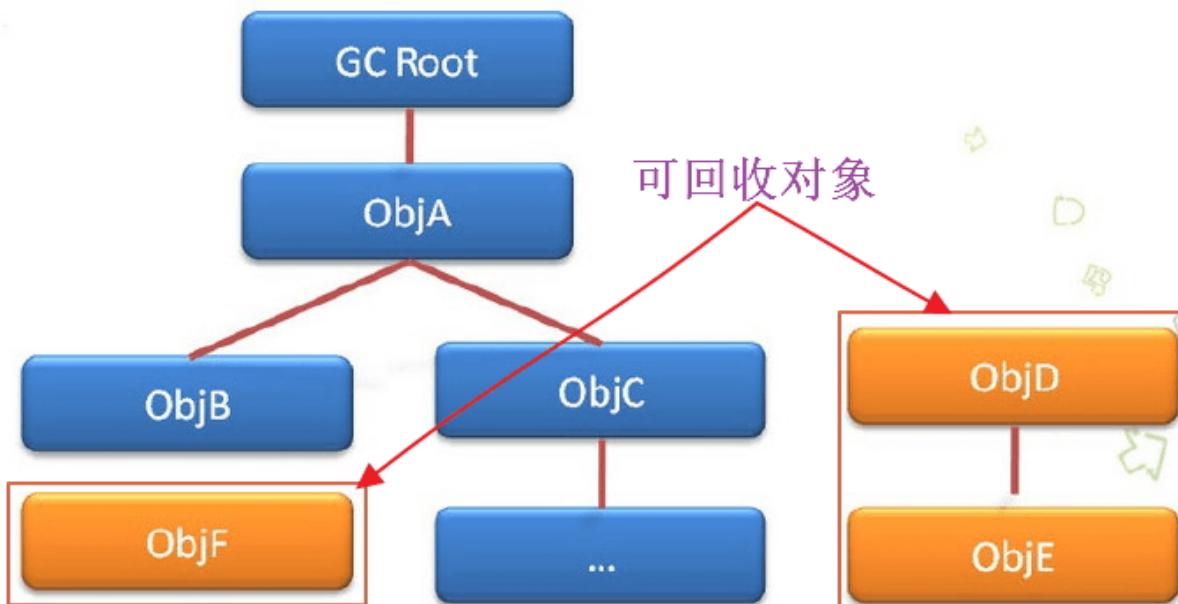
可达性分析算法

通过一系列“GC roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径成为引用链 (reference chain)，当一个对象到GcRoot 没有任何的引用链，则证明此对象不可用。

Gcroot对象包括：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象。
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中JNI（native方法）引用的对象。

公众号：方志朋



生存还是死亡

即时当一个对象不可达，也并非非死不可，这时它们处于一个缓行的阶段要真正判处一个对象死亡，至少要经历2次标记。第一次标记为不可到达。当对象没有覆盖`finalize()`方法，或者`finalize`方法已经被虚拟机调用了，虚拟机都被视为没必要执行（没必要执行是不是直接回收？）

如果一个对象被判定有必要执行，则将这个对象放在一个F-Queue的队列中，并由一个线程去执行它。`finalize`方法是对象逃脱死亡命运的最后机会。当在`finalize`方法中重新将之间赋值给了某个变量，那么第二次标记就会被移除。如果对象第二次还没有逃脱，那么基本就被回收了。

```
1
2 public class FinalizeEscapeGc {
3
4     private static FinalizeEscapeGc instance = null;
5
6     public void alive() {
7         System.out.println("i'm alive");
8     }
9
10
11 /**
12 * 该方法只调用一次
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
13     * @throws Throwable
14     */
15     @Override
16     protected void finalize() throws Throwable {
17         super.finalize();
18         System.out.println("finalize()");
19         FinalizeEscapeGc.instance = this;
20     }
21
22     public static void main(String[] args) throws InterruptedException {
23         instance = new FinalizeEscapeGc();
24         instance = null;
25         System.gc();
26
27         Thread.sleep(500); // finalize() method has a low priority
28         to execute
29         if (instance != null) {
30             instance.alive();
31         } else {
32             System.out.println("ooops,i'm dead");
33         }
34
35         instance = null;
36         System.gc();
37
38         Thread.sleep(500);
39
40         if (instance != null) {
41             instance.alive();
42         } else {
43             System.out.println("ooops,i'm dead");
44         }
45     }
46 }
```

公众号：方志朋

输出：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
finalize()  
i'm alive  
ooops,i'm dead
```

回收方法区

方法区在hotspot虚拟机称为永久代，永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。回收废弃常量与回收堆类似，当一个字符串“abc”，进入了常量池，且没有任何String对象叫“abc”，那么它将被回收。

判断一个类为无用的类：

- 该类的所有实例都被回收
- 加载该类的classloader已经被回收
- 该类对应java.lang.class对象没有在任何地方引用，无法通过反射访问该类。

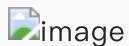
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

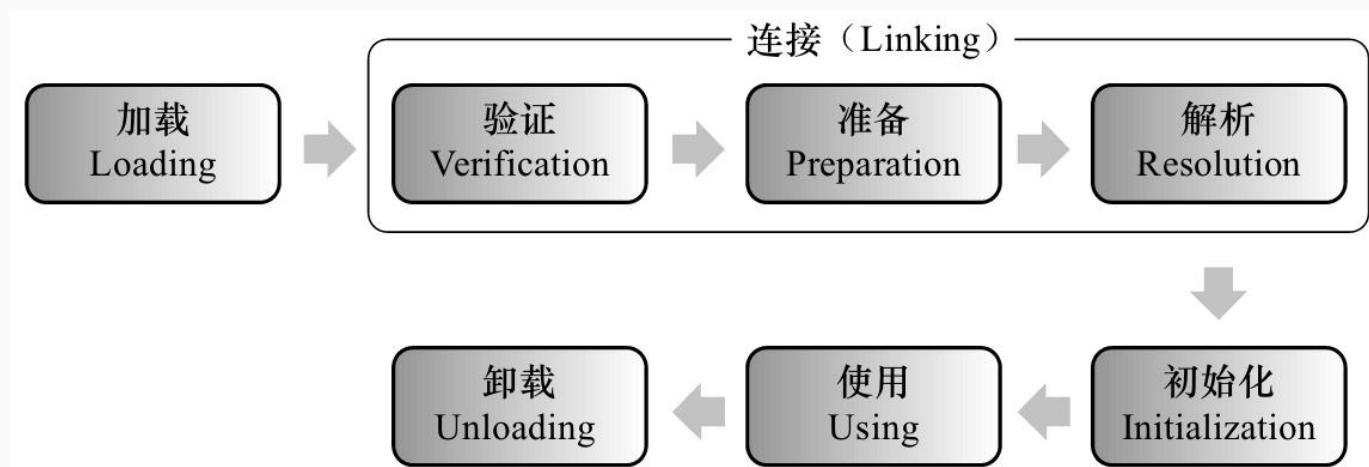
程序员理财，请关注：



Java虚拟机：JVM类加载机制

JVM类加载机制

如下图所示，JVM类加载机制分为五个部分：加载，验证，准备，解析，初始化，下面我们就分别来看一下这五个过程。



加载

加载是类加载过程中的一个阶段，这个阶段会在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的入口。注意这里不一定非得要从一个Class文件获取，这里既可以从ZIP包中读取（比如从jar包和war包中读取），也可以在运行时计算生成（动态代理），也可以由其它文件生成（比如将JSP文件转换成对应的Class类）。

验证

这一阶段的主要目的是为了确保Class文件的字节流中包含的信息是否符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

准备

准备阶段是正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量所使用的内存空间。注意这里所说的初始值概念，比如一个类变量定义为：

```
1 public static int v = 8080;
```

实际上变量v在准备阶段过后的初始值为0而不是8080，将v赋值为8080的putstatic指令是程序被编译后，存放于类构造器方法之中，这里我们后面会解释。

但是注意如果声明为：

```
1 public static final int v = 8080;
```

在编译阶段会为v生成ConstantValue属性，在准备阶段虚拟机会根据ConstantValue属性将v赋值为8080。

解析

解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是class文件中的：

- CONSTANT_Class_info
 - CONSTANT_Field_info
 - CONSTANT_Method_info
- 等类型的常量。

下面我们解释一下符号引用和直接引用的概念：

符号引用与虚拟机实现的布局无关，引用的目标并不一定要已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在。

初始化

初始化阶段是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载器以外，其它操作都由JVM主导。到了初始阶段，才开始真正执行类中定义的Java程序代码。

初始化阶段是执行类构造器方法的过程。方法是由编译器自动收集类中的类变量的赋值操作和静态语句块中的语句合并而成的。虚拟机会保证方法执行之前，父类的方法已经执行完毕。p.s: 如果一个类中没有对静态变量赋值也没有静态语句块，那么编译器可以不为这个类生成()方法。

注意以下几种情况不会执行类初始化：

- 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
- 定义对象数组，不会触发该类的初始化。
- 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。
- 通过类名获取Class对象，不会触发类的初始化。
- 通过Class.forName加载指定类时，如果指定参数initialize为false时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
- 通过ClassLoader默认的loadClass方法，也不会触发初始化动作。

类加载器

虚拟机设计团队把加载动作放到JVM外部实现，以便让应用程序决定如何获取所需的类，JVM提供了3种类加载器：

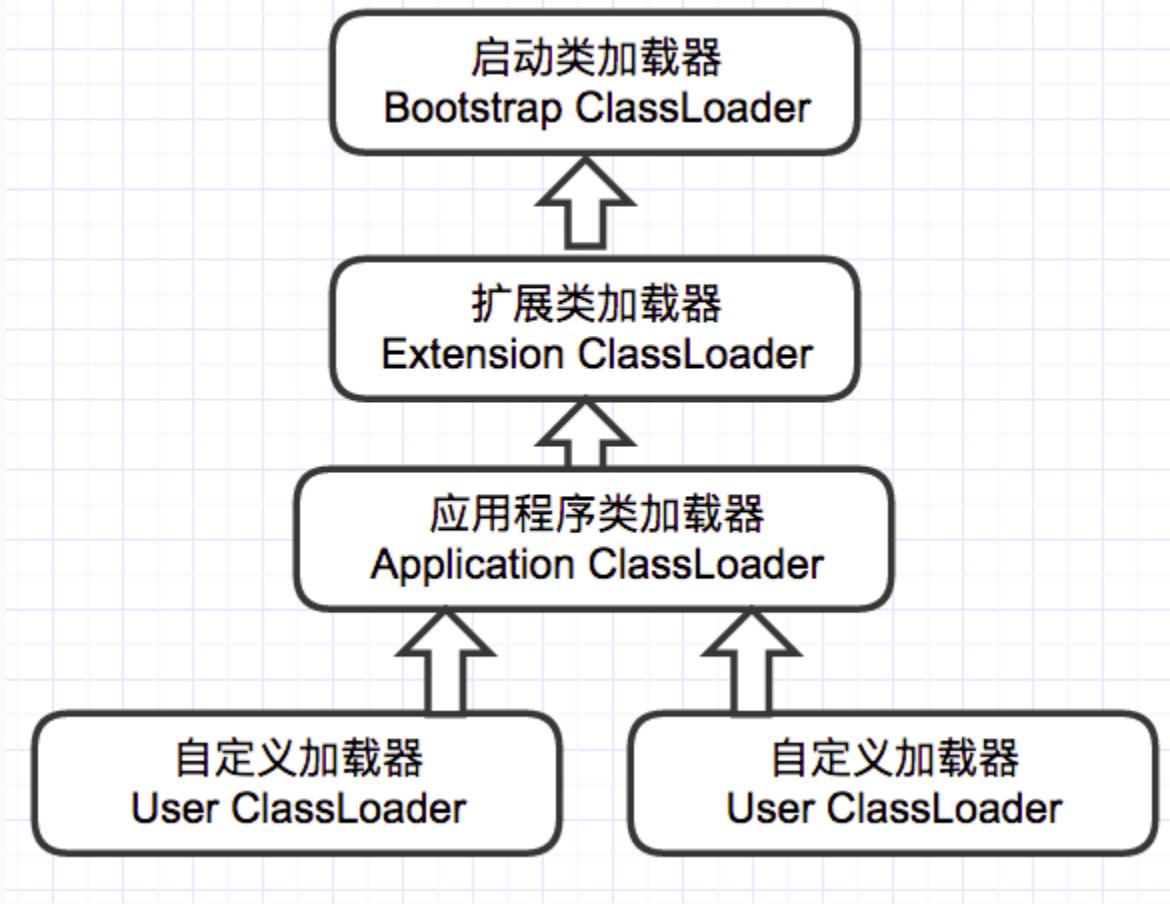
启动类加载器(Bootstrap ClassLoader)：负责加载 JAVA_HOME\lib 目录中的，或通过-Xbootclasspath参数指定路径中的，且被虚拟机认可（按文件名识别，如rt.jar）的类。

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

应用程序类加载器(Application ClassLoader)：负责加载用户路径(classpath)上的类库。

JVM通过双亲委派模型进行类的加载，当然我们也可以通过继承java.lang.ClassLoader实现自定义的类加载器。

公众号：方志朋



当一个类加载器收到类加载任务，会先交给其父类加载器去完成，因此最终加载任务都会传递到顶层的启动类加载器，只有当父类加载器无法完成加载任务时，才会尝试执行加载任务。

采用双亲委派的一个好处是比如加载位于rt.jar包中的类java.lang.Object，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载器最终得到的都是同样一个Object对象。

在有些情境中可能会出现要我们自己来实现一个类加载器的需求，由于这里涉及的内容比较广泛，我想以后单独写一篇文章来讲述，不过这里我们还是稍微来看一下。我们直接看一下jdk中的ClassLoader的源码实现：

```
1  
2 protected synchronized Class<?> loadClass(String name, boolean re  
solve)  
3     throws ClassNotFoundException {  
4     // First, check if the class has already been loaded  
5     Class c = findLoadedClass(name);  
6     if (c == null) {
```

```
7     try {
8         if (parent != null) {
9             c = parent.loadClass(name, false);
10        } else {
11            c = findBootstrapClass0(name);
12        }
13    } catch (ClassNotFoundException e) {
14        // If still not found, then invoke findClass in order
15        // to find the class.
16        c = findClass(name);
17    }
18 }
19 if (resolve) {
20     resolveClass(c);
21 }
22 return c;
23 }
```

- 首先通过Class c = findLoadedClass(name);判断一个类是否已经被加载过。
- 如果没有被加载过执行if (c == null)中的程序，遵循双亲委派的模型，首先会通过递归从父加载器开始找，直到父类加载器是Bootstrap ClassLoader为止。
- 最后根据resolve的值，判断这个class是否需要解析。

而上面的findClass()的实现如下，直接抛出一个异常，并且方法是protected，很明显这是留给我们开发者自己去实现的，这里我们以后我们单独写一篇文章来讲一下如何重写findClass方法来实现我们自己的类加载器。

```
1 protected Class<?> findClass(String name) throws ClassNotFoundException {
2     throw new ClassNotFoundException(name);
3 }
```

References

《UNDERSTANDING THE JVM》

[原文链接](#)

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

<http://www.importnew.com/25295.html>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：

image

公众号：方志朋

Java虚拟机：虚拟机类加载机制

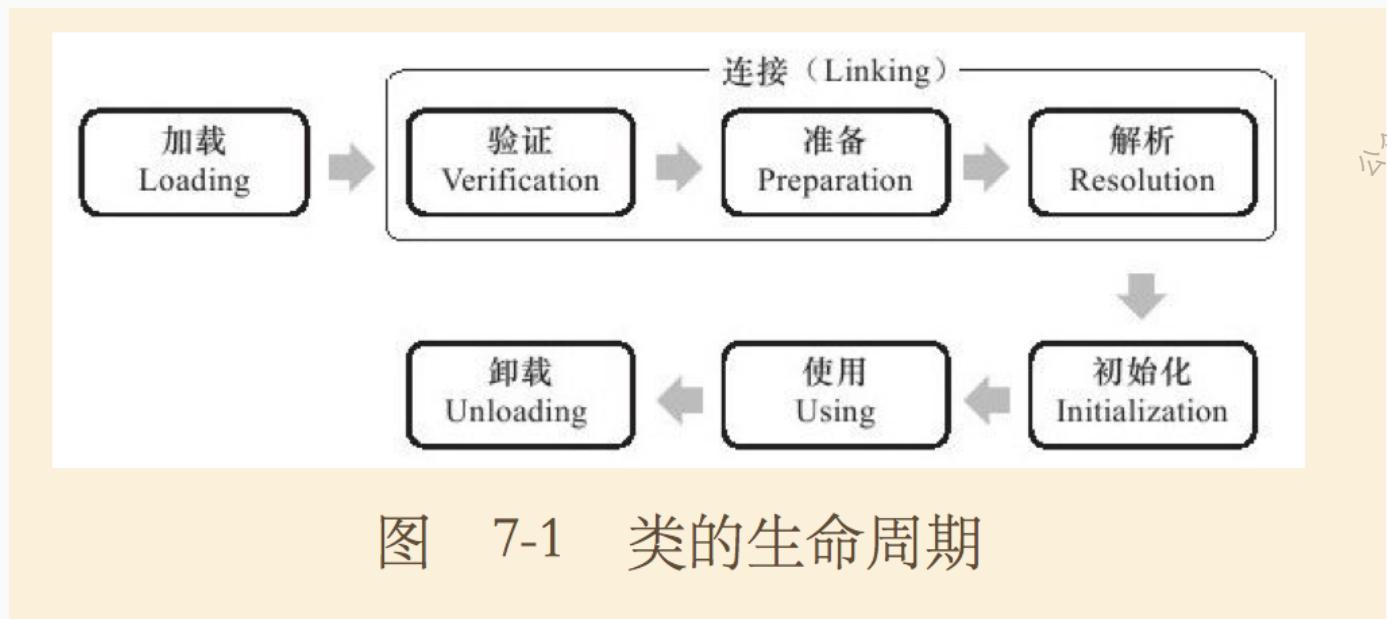
虚拟机类加载机制

代码编译的结果从本地机器码转变成字节码，是存储格式发展的一小步，确是编程语言发展的一大步。

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型，这就是虚拟机的类加载机制。

类加载的时机

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）7个阶段。其中验证、准备、解析3个部分统称为连接（Linking），这7个阶段的发生顺序如图7-1所示。



什么情况下需要开始类加载过程的第一个阶段：加载？Java虚拟机规范中并没有进行强制约束，这点可以交给虚拟机的具体实现来自由把握。但是对于初始化阶段，虚拟机规范则是严格规定了有且只有5种情况必须立即对类进行“初始化”（而加载、验证、准备自然需要在此之前开始）：

- 1) 遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

- 2) 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
- 3) 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
- 4) 当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。
- 5) 当使用JDK 1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF_getStatic、REF_putStatic、REF_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

一些案例：

- 对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。
- 被动使用类字段演示二：通过数组定义来引用类，不会触发此类的初始化

```
1 SuperClass [] sca=new SuperClass [10];
```

- 常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。

接口的加载过程与类加载过程稍有一些不同，针对接口需要做一些特殊说明：接口也有初始化过程，这点与类是一致的，上面的代码都是用静态语句块“static{}”来输出初始化信息的，而接口中不能使用“static{}”语句块，但编译器仍然会为接口生成“<clinit> ()”类构造器[2]，用于初始化接口中所定义的成员变量。接口与类真正有所区别的是前面讲述的5种“有且仅有”需要开始初始化场景中的第3种：当一个类在初始化时，要求其父类全部都已经初始化过了，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量）才会初始化。

类加载过程

Java虚拟机中类加载的全过程，也就是加载、验证、准备、解析和初始化这5个阶段所执行的具体动作。

加载

“加载”是类加载过程的一个阶段。在加载阶段，虚拟机完成了以下三个事情：

- 通过一个类的全限定名来获取定义此类的二进制字节流。
- 将这个字节流所代表的静态存储结构转换成方法区的运行时数据结构。
- 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

加载阶段与连接阶段的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

验证

验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

文件格式验证

元数据验证

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求，这个阶段可能包括的验证点如下：

字节码验证

第三阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在第二阶段对元数据信息中的数据类型做完校验后，这个阶段将对类的方法体进行校验分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的事件，例如：

符号引用验证

最后一个阶段的校验发生在虚拟机将符号引用转换成为直接引用的时候，这个转化动作将在连接的第三阶段-解析阶段中发生。符号引用验证可以看作是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验。通常检验以下的内容：

- 符号引用中通过字符串描述的全限定是否能找到对应的类。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段。
- 符号引用中的类、字段、方法的访问性 (private protected public default) 是否可被当前类访问。

准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的概念需要强调一下，首先，这时候进行内存分配的仅包括类变量（被static修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在Java堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

```
1 public static int value=123;
```

那变量value在准备阶段过后的初始值为0而不是123，因为这时候尚未开始执行任何Java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器<clinit> () 方法之中，所以把value赋值为123的动作将在初始化阶段才会执行。表7-1列出了Java中所有基本数据类型的零值。

上面提到，在“通常情况”下初始值是零值，那相对的会有一些“特殊情况”：如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值，假设上面类变量value的定义变为：

```
1 public final static int value=123;
```

编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为123。

解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，符号引用在前一章讲解class文件格式的时候已经出现过多次，在class文件中它以constant_class_info constant_fieldref_info constant_methodref_info等类型的常量出现，那解析阶段中所说的直接引用与符号引用又有什么关联呢？

符号引用：符号引用以一组符号来描述所引用的目标，符号可以是以任何形式的字面量，只要使用时能无歧义定位到目标即可。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

直接引用（Direct References）：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在内存中存在。

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行，分别对应于常量池的CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT_Methodref_info、CONSTANT_InterfaceMethodref_info、CONSTANT_MethodType_info、CONSTANT_MethodHandle_info和CONSTANT_InvokeDynamic_info

类或接口的解析

假设当前代码所处的类为D，如果要把一个从未解析过的符号引用N解析为一个类或接口C的直接引用，那虚拟机完成整个解析的过程需要以下3个步骤：

- 1) 如果C不是一个数组类型，那虚拟机将会把代表N的全限定名传递给D的类加载器去加载这个类C。在加载过程中，由于元数据验证、字节码验证的需要，又可能触发其他相关类的加载动作，例如加载这个类的父类或实现的接口。一旦这个加载过程出现了任何异常，解析过程就宣告失败。
- 2) 如果C是一个数组类型，并且数组的元素类型为对象，也就是N的描述符会是类似“[Ljava/lang/Integer”的形式，那将会按照第1点的规则加载数组元素类型。如果N的描述符如前面所假设的形式，需要加载的元素类型就是“java.lang.Integer”，接着由虚拟机生成一个代表此数组维度和元素的数组对象。
- 3) 如果上面的步骤没有出现任何异常，那么C在虚拟机中实际上已经成为一个有效的类或接口了，但在解析完成之前还要进行符号引用验证，确认D是否具备对C的访问权限。如果发现不具备访问权限，将抛出java.lang.IllegalAccessError异常。

字段解析

要解析一个未被解析过的字段符号引用，首先将会对字段表内class_index[2]项中索引的CONSTANT_Class_info符号引用进行解析，也就是字段所属的类或接口的符号引用。如果在解析这个类或接口符号引用的过程中出现了任何异常，都会导致字段符号引用解析的失败。如果解析成功完成，那将这个字段所属的类或接口用C表示，虚拟机规范要求按照如下步骤对C进行后续字段的搜索。

- 1) 如果C本身就包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 2) 否则，如果在C中实现了接口，将会按照继承关系从下往上递归搜索各个接口和它的父接口，如果接口中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。
- 3) 否则，如果C不是java.lang.Object的话，将会按照继承关系从下往上递归搜索其父类，如果在父类中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。
- 4) 否则，查找失败，抛出java.lang.NoSuchFieldError异常。

类方法解析

类方法解析的第一个步骤与字段解析一样，也需要先解析出类方法表的class_index[3]项中索引的方法所属的类或接口的符号引用，如果解析成功，我们依然用C表示这个类，接下来虚拟机将会按照如下步骤进行后续的类方法搜索。

- 1) 类方法和接口方法符号引用的常量类型定义是分开的，如果在类方法表中发现class_index中索引的C是个接口，那就直接抛出java.lang.IncompatibleClassChangeError异常。
- 2) 如果通过了第1步，在类C中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。
- 3) 否则，在类C的父类中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。
- 4) 否则，在类C实现的接口列表及它们的父接口之中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果存在匹配的方法，说明类C是一个抽象类，这时查找结束，抛出java.lang.AbstractMethodError异常。
- 5) 否则，宣告方法查找失败，抛出java.lang.NoSuchMethodError。

最后，如果查找过程成功返回了直接引用，将会对这个方法进行权限验证，如果发现不具备对此方法的访问权限，将抛出java.lang.IllegalAccessError异常。

接口方法解析

接口方法也需要先解析出接口方法表的class_index[4]项中索引的方法所属的类或接口的符号引用，如果解析成功，依然用C表示这个接口，接下来虚拟机将会按照如下步骤进行后续的接口方法搜索。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 1) 与类方法解析不同，如果在接口方法表中发现class_index中的索引C是个类而不是接口，那就直接抛出java.lang.IncompatibleClassChangeError异常。
- 2) 否则，在接口C中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。
- 3) 否则，在接口C的父接口中递归查找，直到java.lang.Object类（查找范围会包括Object类）为止，看是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。
- 4) 否则，宣告方法查找失败，抛出java.lang.NoSuchMethodError异常。

初始化

在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器<clinit>()方法的过程。我们在下文会讲解<clinit>()方法是怎么生成的，在这里，我们先看一下<clinit>()方法执行过程中一些可能会影响程序运行行为的特点和细节，这部分相对更贴近于普通的程序开发人员[1]。

<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{}块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问，如代码清单7-5中的例子所示。

<clinit>()方法与类的构造函数（或者说实例构造器<init>()方法）不同，它不需要显式地调用父类构造器，虚拟机会保证在子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此在虚拟机中第一个被执行的<clinit>()方法的类肯定是java.lang.Object。

虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果在一个类的<clinit>()方法中有耗时很长的操作，就可能造成多个进程阻塞[2]，在实际应用中这种阻塞往往是很隐蔽的。代码清单7-7演示了这种场景。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

Java虚拟机：JVM性能调优监控工具jps、jstack、jmap、jhat、jstat、hprof使用详解

JVM性能调优监控工具jps、jstack、jmap、jhat、jstat、hprof使用详解

现实企业级Java应用开发、维护中，有时候我们会碰到下面这些问题：

- OutOfMemoryError，内存不足
- 内存泄露
- 线程死锁
- 锁争用（Lock Contention）
- Java进程消耗CPU过高
-

这些问题在日常开发、维护中可能被很多人忽视（比如有的人遇到上面的问题只是重启服务器或者调大内存，而不会深究问题根源），但能够理解并解决这些问题时Java程序员进阶的必备要求。本文将对一些常用的JVM性能调优监控工具进行介绍，希望能起抛砖引玉之用。

而且这些监控、调优工具的使用，无论你是运维、开发、测试，都是必须掌握的。

jps(Java Virtual Machine Process Status Tool)

jps主要用来输出JVM中运行的进程状态信息。语法格式如下：

```
1 jps [options] [hostid]
```

如果不指定hostid就默认为当前主机或服务器。

命令行参数选项说明如下：

```
1 -q 不输出类名、Jar名和传入main方法的参数  
2  
3 -m 输出传入main方法的参数  
4
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

5 -l 输出main类或Jar的全限名

6

7 -v 输出传入JVM的参数

比如下面：

```
1 root@ubuntu:/# jps -m -l
2 2458 org.artifactory.standalone.main.Main /usr/local/artifactory-
2.2.5/etc/jetty.xml
3 29920 com.sun.tools.hat.Main -port 9998 /tmp/dump.dat
4 3149 org.apache.catalina.startup.Bootstrap start
5 30972 sun.tools.jps.Jps -m -l
6 8247 org.apache.catalina.startup.Bootstrap start
7 25687 com.sun.tools.hat.Main -port 9999 dump.dat
8 21711 mrf-center.jar
```

jstack

jstack主要用来查看某个Java进程内的线程堆栈信息。语法格式如下：

```
1 jstack [option] pid
2 jstack [option] executable core
3 jstack [option] [server-id@]remote-hostname-or-ip
```

命令行参数选项说明如下：

```
1 -l long listings, 会打印出额外的锁信息，在发生死锁时可以用jstack -l pid来观
察锁持有情况-m mixed mode, 不仅会输出Java堆栈信息，还会输出C/C++堆栈信息（比
如Native方法）
```

jstack可以定位到线程堆栈，根据堆栈信息我们可以定位到具体代码，所以它在JVM性能调优中使用得非常多。下面我们来一个实例找出某个Java进程中最耗费CPU的Java线程并定位堆栈信息，用到的命令有ps、top、printf、jstack、grep。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

第一步先找出Java进程ID，我部署在服务器上的Java应用名称为mrf-center：

```
1 root@ubuntu:/# ps -ef | grep mrf-center | grep -v grep
2 root      21711      1  1 14:47 pts/3    00:02:10 java -jar mrf-cent
er.jar
```

得到进程ID为21711，第二步找出该进程中最耗费CPU的线程，可以使用ps -Lf p pid或者ps -mp pid -o THREAD, tid, time或者top -Hp pid，我这里用第三个，输出如下：

```
top - 17:10:22 up 59 days, 1:56, 1 user, load average: 0.05,
Tasks: 36 total, 0 running, 36 sleeping, 0 stopped, 0 zzz
Cpu(s): 0.8%us, 0.3%sy, 0.0%ni, 98.7%id, 0.2%wa, 0.0%hi, (Mem: 8075600k total, 7799876k used, 275724k free, 77260k
Swap: 25061368k total, 15506616k used, 9554752k free, 865340k
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
21742	root	20	0	4185m	95m	10m	S	1	1.2	1:12.24	java
21711	root	20	0	4185m	95m	10m	S	0	1.2	0:00.00	java
21712	root	20	0	4185m	95m	10m	S	0	1.2	0:02.22	java
21713	root	20	0	4185m	95m	10m	S	0	1.2	0:00.18	java
21714	root	20	0	4185m	95m	10m	S	0	1.2	0:00.16	java
21715	root	20	0	4185m	95m	10m	S	0	1.2	0:00.17	java
21716	root	20	0	4185m	95m	10m	S	0	1.2	0:00.16	java
21717	root	20	0	4185m	95m	10m	S	0	1.2	0:00.59	java
21718	root	20	0	4185m	95m	10m	S	0	1.2	0:00.00	java
21719	root	20	0	4185m	95m	10m	S	0	1.2	0:00.00	java
21720	root	20	0	4185m	95m	10m	S	0	1.2	0:00.00	java
21721	root	20	0	4185m	95m	10m	S	0	1.2	0:03.25	java
21722	root	20	0	4185m	95m	10m	S	0	1.2	0:03.42	java
21723	root	20	0	4185m	95m	10m	S	0	1.2	0:00.00	java
21724	root	20	0	4185m	95m	10m	S	0	1.2	0:01.89	java
21727	root	20	0	4185m	95m	10m	S	0	1.2	0:01.19	java

TIME列就是各个Java线程耗费的CPU时间，CPU时间最长的是线程ID为21742的线程，用

```
1 printf "%x\n" 21742
```

得到21742的十六进制值为54ee，下面会用到。

OK，下一步终于轮到jstack上场了，它用来输出进程21711的堆栈信息，然后根据线程ID的十六进制值grep，如下：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
1 root@ubuntu:/# jstack 21711 | grep 54ee
2 "PollIntervalRetrySchedulerThread" prio=10 tid=0x00007f950043e000
  nid=0x54ee in Object.wait() [0x00007f94c6eda000]
```

可以看到CPU消耗在PollIntervalRetrySchedulerThread这个类的Object.wait()，我找了下我的代码，定位到下面的代码：

```
1 // Idle wait
2 getLog().info("Thread [" + getName() + "] is idle waiting...");
3 schedulerThreadState = PollTaskSchedulerThreadState.IdleWaiting;
4 long now = System.currentTimeMillis();
5 long waitTime = now + getIdleWaitTime();
6 long timeUntilContinue = waitTime - now;
7 synchronized(sigLock) { try {
8     if(!halted.get()) {
9         sigLock.wait(timeUntilContinue);
10    }
11 } catch (InterruptedException ignore) {
12 }
13 }
```

它是轮询任务的空闲等待代码，上面的sigLock.wait(timeUntilContinue)就对应了前面的Object.wait()。

jmap (Memory Map) 和jhat (Java Heap Analysis Tool)

jmap用来查看堆内存使用状况，一般结合jhat使用。

jmap语法格式如下：

```
1 jmap [option] pid
2 jmap [option] executable core
3 jmap [option] [server-id@]remote-hostname-or-ip
```

如果运行在64位JVM上，可能需要指定-J-d64命令选项参数。

```
1 jmap -permstat pid
```

打印进程的类加载器和类加载器加载的持久代对象信息，输出：类加载器名称、对象是否存活（不可靠）、对象地址、父类加载器、已加载的类大小等信息，如下图：

```
打印进程的类加载器和类加载器加载的持久代对象信息，输出：类加载器名称、对象是否存活（不可靠）、对象地址、父类加载器、已加载的类大小等信息，如下图： SpringBoot:
root@ubuntu:~# jmap -permstat 21711
Attaching to process ID 21711, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 20.10-b01
8331 intern Strings occupying 761376 bytes.
finding class loader instances ..Finding object size using Printezis bits and skipping over...
Finding object size using Printezis bits and skipping over...
Finding object size using Printezis bits and skipping over...
done.
computing per loader stat ..done.
please wait.. computing liveness.....liveness analysis may be inaccurate ...
class_loader    classes   bytes   parent_loader   alive?   type
<bootstrap>      1418    8882640    null     live   <internal>
0x0000000784d14628  1        1944    null     dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1410  1        3112    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1368  1        3168    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef14b8  1        3152    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef13a0  1        3184    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1560  1        3216    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784d4b500  0        0       0x0000000784c000b0  dead   java/util/ResourceBundle$RBClassLoader@0x000000077fdd35d0
0x0000000784ef1528  1        1944    null     dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef14f0  1        3336    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784d14548  1        1944    null     dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef12c0  1        1944    null     dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1598  1        1944    null     dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784f87b08  1        3200    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784d14580  1        1944    null     dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784c00f8  8        49544   null     live   sun/misc/Launcher$ExtClassLoader@0x000000077fb01ff8
0x0000000784fa5e98  1        3112    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784fa5b28  1        3128    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1480  1        3112    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784c000b0  2362   14145024   0x0000000784c000f8  live   sun/misc/Launcher$AppClassLoader@0x000000077fc40b!
0x0000000784d14698  1        1944    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1260  1        1960    null     dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1330  1        3104    null     dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
0x0000000784ef1448  1        3136    0x0000000784c000b0  dead   sun/reflect/DelegatingClassLoader@0x000000077fa675e8
```

使用jmap -heap pid查看进程堆内存使用情况，包括使用的GC算法、堆配置参数和各代中堆内存使用情况。比如下面的例子：

```
1 root@ubuntu:~# jmap -heap 21711
2 Attaching to process ID 21711, please wait...
3 Debugger attached successfully.
4 Server compiler detected.
5 JVM version is 20.10-b01
6
7 using thread-local object allocation.
8 Parallel GC with 4 thread(s)
9
10 Heap Configuration:
11 MinHeapFreeRatio = 40
12 MaxHeapFreeRatio = 70
13 MaxHeapSize      = 2067791872 (1972.0MB)
14 NewSize          = 1310720 (1.25MB)
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
15 MaxNewSize      = 17592186044415 MB
16 OldSize         = 5439488 (5.1875MB)
17 NewRatio        = 2
18 SurvivorRatio   = 8
19 PermSize        = 21757952 (20.75MB)
20 MaxPermSize    = 85983232 (82.0MB)
21
22 Heap Usage:
23 PS Young Generation
24 Eden Space:
25     capacity = 6422528 (6.125MB)
26     used      = 5445552 (5.1932830810546875MB)
27     free       = 976976 (0.9317169189453125MB)
28     84.78829520089286% used
29 From Space:
30     capacity = 131072 (0.125MB)
31     used      = 98304 (0.09375MB)
32     free       = 32768 (0.03125MB)
33     75.0% used
34 To Space:
35     capacity = 131072 (0.125MB)
36     used      = 0 (0.0MB)
37     free       = 131072 (0.125MB)
38     0.0% used
39 PS Old Generation
40     capacity = 35258368 (33.625MB)
41     used      = 4119544 (3.9287033081054688MB)
42     free       = 31138824 (29.69629669189453MB)
43     11.683876009235595% used
44 PS Perm Generation
45     capacity = 52428800 (50.0MB)
46     used      = 26075168 (24.867218017578125MB)
47     free       = 26353632 (25.132781982421875MB)
48     49.73443603515625% used
49     ....
```

公众号：方志朋

使用jmap –histo[:live] pid查看堆内存中的对象数目、大小统计直方图，如果带上live则只统计活对象，如下：

```
1 root@ubuntu:/# jmap -histo:live 21711 | more
2 num      #instances          #bytes  class name-----
3
4   1:        38445       5597736  <constMethodKlass>
5   2:        38445       5237288  <methodKlass>
6   3:        3500        3749504  <constantPoolKlass>
7   4:       60858       3242600  <symbolKlass>
8   5:        3500        2715264  <instanceKlassKlass>
9   6:        2796        2131424  <constantPoolCacheKlass>
10  7:        5543        1317400  [I
11  8:       13714        1010768  [C
12  9:        4752        1003344  [B
13 10:        1225        639656   <methodDataKlass>
14 11:       14194        454208   java.lang.String
15 12:        3809        396136   java.lang.Class
16 13:        4979        311952   [S
17 14:        5598        287064   [[I
18 15:        3028        266464   java.lang.reflect.Method
19 16:        280         163520   <objArrayKlassKlass>
20 17:        4355        139360   java.util.HashMap$Entry
21 18:       1869        138568   [Ljava.util.HashMap$Entry;
22 19:        2443        97720    java.util.LinkedHashMap$Entry
23 20:        2072        82880    java.lang.ref.SoftReference
24 21:        1807        71528    [Ljava.lang.Object;
25 22:        2206        70592    java.lang.ref.WeakReference
26 23:        934         52304    java.util.LinkedHashMap
27 24:        871         48776    java.beans.MethodDescriptor
28 25:       1442        46144    java.util.concurrent.ConcurrentHashMap$HashEntry
29
30 26:          804        38592    java.util.HashMap
31 27:          948        37920    java.util.concurrent.ConcurrentHashMap$Segment
32
33 28:        1621        35696    [Ljava.lang.Class;
34 29:        1313        34880    [Ljava.lang.String;
35 30:        1396        33504    java.util.LinkedList$Entry
36 31:          462        33264    java.lang.reflect.Field
37 32:        1024        32768    java.util.Hashtable$Entry
38 33:          948        31440    [Ljava.util.concurrent.ConcurrentHashMap$HashEntry;
```

公众号：方志朋

class name是对象类型，说明如下：

```
1 B byte
2 C char
3 D double
4 F float
5 I int
6 J long
7 Z boolean
8 [ 数组，如[I表示int[]]
9 [L+类名 其他对象
```

还有一个很常用的情况是：用jmap把进程内存使用情况dump到文件中，再用jhat分析查看。jmap进行dump命令格式如下：

```
1 jmap -dump:format=b,file=dumpFileName pid
```

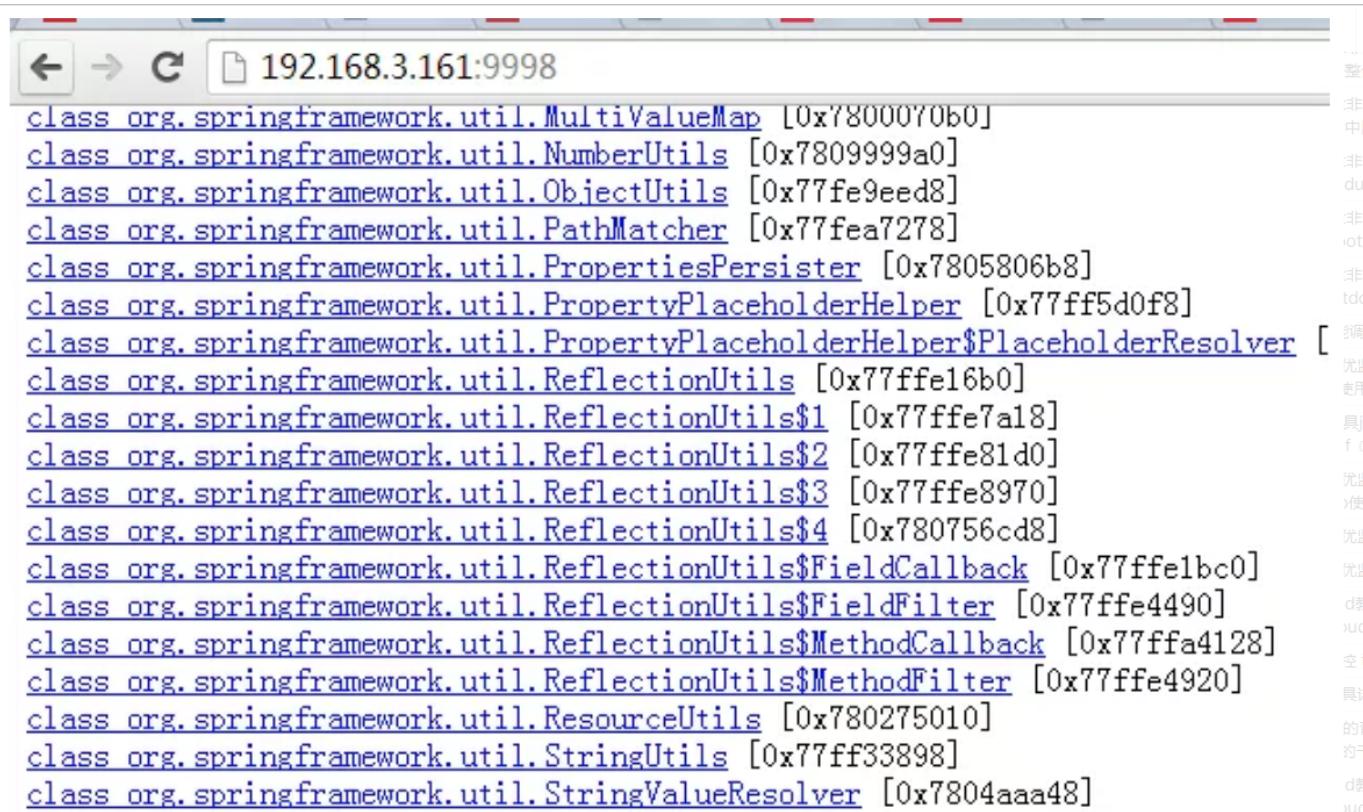
我一样地对上面进程ID为21711进行Dump：

```
1 root@ubuntu:/# jmap -dump:format=b,file=/tmp/dump.dat 21711
2 Dumping heap to /tmp/dump.dat ...
3 Heap dump file created
4     dump出来的文件可以用MAT、VisualVM等工具查看，这里用jhat查看：
5
6 root@ubuntu:/# jhat -port 9998 /tmp/dump.dat
7 Reading from /tmp/dump.dat...
8 Dump file created Tue Jan 28 17:46:14 CST 2014Snapshot read, reso
    lving...
9 Resolving 132207 objects...
10 Chasing references, expect 26 dots.....
11 Eliminating duplicate references.....
12 Snapshot resolved.
13 Started HTTP server on port 9998Server is ready.
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

注意如果Dump文件太大，可能需要加上-J-Xmx512m这种参数指定最大堆内存，即jhat -J-Xmx512m -port 9998 /tmp/dump.dat。然后就可以在浏览器中输入主机地址:9998查看了：



The screenshot shows a browser window with the URL '192.168.3.161:9998'. The page displays a list of Java classes from the 'org.springframework.util' package, each with its memory address. The classes listed include MultiValueMap, NumberUtils, ObjectUtils, PathMatcher, PropertiesPersister, PropertyPlaceholderHelper, PropertyPlaceholderHelper\$PlaceholderResolver, ReflectionUtils, ReflectionUtils\$1, ReflectionUtils\$2, ReflectionUtils\$3, ReflectionUtils\$4, ReflectionUtils\$FieldCallback, ReflectionUtils\$FieldFilter, ReflectionUtils\$MethodCallback, ReflectionUtils\$MethodFilter, ResourceUtils, StringUtils, and StringValueResolver. The right side of the browser window has a vertical toolbar with various icons.

```
class org.springframework.util.MultiValueMap [0x7800070b0]
class org.springframework.util.NumberUtils [0x7809999a0]
class org.springframework.util.ObjectUtils [0x77fe9eed8]
class org.springframework.util.PathMatcher [0x77fea7278]
class org.springframework.util.PropertiesPersister [0x7805806b8]
class org.springframework.util.PropertyPlaceholderHelper [0x77ff5d0f8]
class org.springframework.util.PropertyPlaceholderHelper$PlaceholderResolver [0x77ff5d0f8]
class org.springframework.util.ReflectionUtils [0x77ffe16b0]
class org.springframework.util.ReflectionUtils$1 [0x77ffe7a18]
class org.springframework.util.ReflectionUtils$2 [0x77ffe81d0]
class org.springframework.util.ReflectionUtils$3 [0x77ffe8970]
class org.springframework.util.ReflectionUtils$4 [0x780756cd8]
class org.springframework.util.ReflectionUtils$FieldCallback [0x77ffe1bc0]
class org.springframework.util.ReflectionUtils$FieldFilter [0x77ffe4490]
class org.springframework.util.ReflectionUtils$MethodCallback [0x77ffa4128]
class org.springframework.util.ReflectionUtils$MethodFilter [0x77ffe4920]
class org.springframework.util.ResourceUtils [0x780275010]
class org.springframework.util.StringUtils [0x77ff33898]
class org.springframework.util.StringValueResolver [0x7804aaa48]
```

Package org.springframework.util.xml

```
class org.springframework.util.xml.DomUtils [0x780429528]
class org.springframework.util.xml.SimpleSaxErrorHandler [0x77fff4110]
class org.springframework.util.xml.XmlValidationModeDetector [0x77fff5830]
```

Other Queries

jstat (JVM统计监测工具)

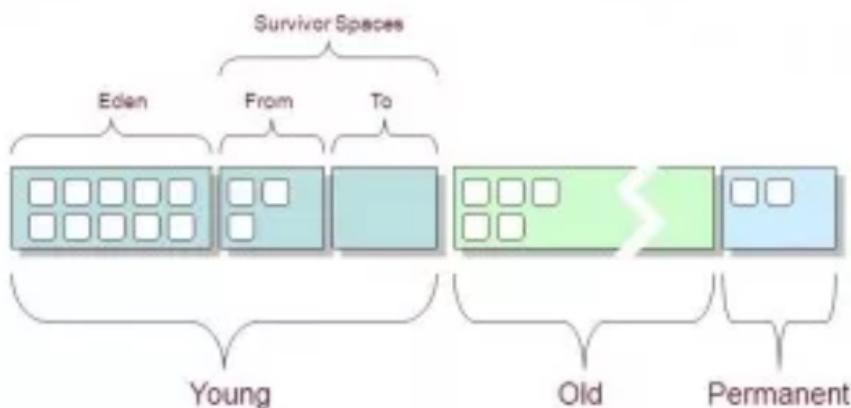
语法格式如下：

```
1 jstat [ generalOption | outputOptions vmid [interval[s|ms] [count]] ]
```

vmid是Java虚拟机ID，在Linux/Unix系统上一般就是进程ID。interval是采样时间间隔。count是采样数目。比如下面输出的是GC信息，采样时间间隔为250ms，采样数为4：

```
1 root@ubuntu:/# jstat -gc 21711 250 4
2 S0C    S1C    S0U    S1U      EC      EU      OC      OU
      PC      PU     YGC     YGCT     FGC     FGCT     GCT
3 192.0  192.0   64.0    0.0    6144.0   1854.9   32000.0   4111.6
   55296.0 25472.7    702    0.431     3       0.218     0.649
4 192.0  192.0   64.0    0.0    6144.0   1972.2   32000.0   4111.6
   55296.0 25472.7    702    0.431     3       0.218     0.649
5 192.0  192.0   64.0    0.0    6144.0   1972.2   32000.0   4111.6
   55296.0 25472.7    702    0.431     3       0.218     0.649
6 192.0  192.0   64.0    0.0    6144.0   2109.7   32000.0   4111.6
   55296.0 25472.7    702    0.431     3       0.218     0.649
```

要明白上面各列的意义，先看JVM堆内存布局：



可以看出：

堆内存 = 年轻代 + 年老代 + 永久代

年轻代 = Eden区 + 两个Survivor区 (From和To)

现在来解释各列含义：

- 1 S0C、S1C、S0U、S1U: Survivor 0/1区容量 (Capacity) 和使用量 (Used)
- 2 EC、EU: Eden区容量和使用量
- 3 OC、OU: 年老代容量和使用量
- 4 PC、PU: 永久代容量和使用量
- 5 YGC、YGT: 年轻代GC次数和GC耗时

6 FGC、FGCT: Full GC次数和Full GC耗时

7 GCT: GC总耗时

hprof (Heap/CPU Profiling Tool)

hprof能够展现CPU使用率，统计堆内存使用情况。

语法格式如下：

```
1 java -agentlib:hprof[=options] ToBeProfiledClass  
2 java -Xrunprof[:options] ToBeProfiledClass  
3 javac -J-agentlib:hprof[=options] ToBeProfiledClass
```

完整的命令选项如下：

1 Option Name and Value	Description	Default
2 -----	-----	-----
3 heap=dump sites all	heap profiling	all
4 cpu=samples times old	CPU usage	off
5 monitor=y n	monitor contention	n
6 format=a b	text(txt) or binary output	a
7 file=<file> [.txt]	write data to file	java.hprof
8 net=<host>:<port>	send data over a socket	off
9 depth=<size>	stack trace depth	4
10 interval=<ms>	sample interval in ms	10
11 cutoff=<value>	output cutoff point	0.0001
12 lineno=y n	line number in traces?	y
13 thread=y n	thread in traces?	n
14 doe=y n	dump on exit?	y
15 msa=y n	Solaris micro state accounting	n
16 force=y n	force output to <file>	y
17 verbose=y n	print messages about dumps	y

来几个官方指南上的实例。

CPU Usage Sampling Profiling(cpu=samples)的例子：

```
1 java -agentlib:hprof=cpu=samples,interval=20,depth=3 Hello
```

上面每隔20毫秒采样CPU消耗信息，堆栈深度为3，生成的profile文件名称是java.hprof.txt，在当前目录。

CPU Usage Times Profiling(cpu=times)的例子，它相对于CPU Usage Sampling Profile能够获得更加细粒度的CPU消耗信息，能够细到每个方法调用的开始和结束，它的实现使用了字节码注入技术(BCI)：

```
1 javac -J-agentlib:hprof=cpu=times Hello.java
```

Heap Allocation Profiling(heap=sites)的例子：

```
1 javac -J-agentlib:hprof=heap=sites Hello.java
```

Heap Dump(heap=dump)的例子，它比上面的Heap Allocation Profiling能生成更详细的Heap Dump信息：

```
1 javac -J-agentlib:hprof=heap=dump Hello.java
```

虽然在JVM启动参数中加入-Xrunprof:heap=sites参数可以生成CPU/Heap Profile文件，但对JVM性能影响非常大，不建议在线上服务器环境使用。

作者：方志朋，一线大厂架构师，

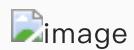
来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

数据库：真正理解Mysql的四种隔离级别

真正理解Mysql的四种隔离级别

什么是事务

事务是应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中所作的所有更改都会被撤消。也就是事务具有原子性，一个事务中的一系列的操作要么全部成功，要么一个都不做。

事务的结束有两种，当事务中的所以步骤全部成功执行时，事务提交。如果其中一个步骤失败，将发生回滚操作，撤消撤消之前到事务开始时的所以操作。

事务的 ACID

事务具有四个特征：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持续性（Durability）。这四个特性简称为 ACID 特性。

- 原子性。事务是数据库的逻辑工作单位，事务中包含的各操作要么都做，要么都不做
- 一致性。事 务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。因此当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态。如果数据库系统 运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态，或者说是 不一致的状态。
- 隔离性。一个事务的执行不能其它事务干扰。即一个事务内部的操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持续性。也称永久性，指一个事务一旦提交，它对数据库中的数据的改变就应该是永久性的。接下来的其它操作或故障不应该对其执行结果有任何影响。

Mysql的四种隔离级别

SQL标准定义了4类隔离级别，包括了一些具体规则，用来限定事务内外的哪些改变是可见的，哪些是不可见的。低级别的隔离级一般支持更高的并发处理，并拥有更低的系统开销。

Read Uncommitted（读取未提交内容）

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

Read Committed（读取提交内容）

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。

Repeatable Read（可重读）

这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读（Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影”行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。

Serializable（可串行化）

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

这四种隔离级别采取不同的锁类型来实现，若读取的是同一个数据的话，就容易发生问题。例如：

- 脏读(Dirty Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。
- 不可重复读(Non-repeatable read)：在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。
- 幻读(Phantom Read)：在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就有几列数据是未查询出来的，如果此时插入和另外一个事务插入的数据，就会报错。

在MySQL中，实现了这四种隔离级别，分别有可能产生问题如下所示：

隔离级别	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	✗	√	√
Repeatable read	✗	✗	√
Serializable	✗	✗	✗

测试Mysql的隔离级别

下面，将利用MySQL的客户端程序，我们分别来测试一下这几种隔离级别。

测试数据库为demo，表为test；表结构：

```
CREATE TABLE `test` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `num` int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
```

两个命令行客户端分别为A，B；不断改变A的隔离级别，在B端修改数据。

将A的隔离级别设置为read uncommitted(未提交读)

```
mysql> set session transaction isolation level read uncommitted;
Query OK, 0 rows affected (0.00 sec)
```

A：启动事务，此时数据为初始状态

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
mysql> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| READ-UNCOMMITTED |
+-----+
1 row in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

B: 启动事务，更新数据，但不提交

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update test set num=10 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 10 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

A: 再次读取数据，发现数据已经被修改了，这就是所谓的“脏读”

```
mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 10 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

B: 回滚事务

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
mysql> rollback;
Query OK, 0 rows affected (0.00 sec)
```

A: 再次读数据，发现数据变回初始状态

```
mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

经过上面的实验可以得出结论，事务B更新了一条记录，但是没有提交，此时事务A可以查询出未提交记录。造成脏读现象。未提交读是最低的隔离级别。

将客户端A的事务隔离级别设置为read committed(已提交读)

```
mysql> set session transaction isolation level read committed;
Query OK, 0 rows affected (0.00 sec)
```

A: 启动事务，此时数据为初始状态

```
mysql> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| READ-COMMITTED |
+-----+
1 row in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

B: 启动事务，更新数据，但不提交

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update test set num=10 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 10 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

A: 再次读数据，发现数据未被修改

```
mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

B: 提交事务

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

A: 再次读取数据，发现数据已发生变化，说明B提交的修改被事务中的A读到了，这就是所谓的“不可重复读”

```
mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 10 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

经过上面的实验可以得出结论，已提交读隔离级别解决了脏读的问题，但是出现了不可重复读的问题，即事务A在两次查询的数据不一致，因为在两次查询之间事务B更新了一条数据。已提交读只允许读取已提交的记录，但不要求可重复读。

将A的隔离级别设置为repeatable read(可重复读)

```
mysql> set session transaction isolation level repeatable read;
Query OK, 0 rows affected (0.00 sec)
```

A: 启动事务，此时数据为初始状态

```
mysql> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| REPEATABLE-READ |
+-----+
1 row in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

B: 启动事务，更新数据，但不提交

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update test set num=10 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 10 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

A: 再次读取数据，发现数据未被修改

java架构师公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

B: 提交事务

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

A: 再次读取数据，发现数据依然未发生变化，这说明这次可以重复读了

```
mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

B: 插入一条新的数据，并提交

```
mysql> insert into test (num) value(4);
Query OK, 1 row affected (0.00 sec)

mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
+----+----+
4 rows in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

A: 再次读取数据，发现数据依然未发生变化，虽然可以重复读了，但是却发现读的不是最新数据，这就是所谓的“幻读”

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+----+----+
3 rows in set (0.00 sec)
```

A: 提交本次事务，再次读取数据，发现读取正常了

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test;
+----+----+
| id | num |
+----+----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
+----+----+
4 rows in set (0.00 sec)
```

由以上的实验可以得出结论，可重复读隔离级别只允许读取已提交记录，而且在一个事务两次读取一个记录期间，其他事务部的更新该记录。但该事务不要求与其他事务可串行化。例如，当一个事务可以找到由一个已提交事务更新的记录，但是可能产生幻读问题(注意是可能，因为数据库对隔离级别的实现有所差别)。像以上的实验，就没有出现数据幻读的问题。

将A的隔离级别设置为可串行化(Serializable)

```
mysql> set session transaction isolation level serializable;
Query OK, 0 rows affected (0.00 sec)
```

A: 启动事务，此时数据为初始状态

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
mysql> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| SERIALIZABLE |
+-----+
1 row in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

B：发现B此时进入了等待状态，原因是A的事务尚未提交，只能等待（此时，B可能会发生等待超时）

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into test (num) value(4);
|
```

A：提交事务

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

B：发现插入成功

```
mysql> insert into test (num) value(4);
Query OK, 1 row affected (3.28 sec)
```

Serializable完全锁定字段，若一个事务来查询同一份数据就必须等待，直到前一个事务完成并解除锁定为止。是完整的隔离级别，会锁定对应的数据表格，因而会有效率的问题。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

数据库：Mysql综合练习题

Mysql综合练习题

10.9 本章实例

在本小节中将在 student 表和 score 表上进行查询。student 表和 score 表的定义如表 10.3 和表 10.4 所示。

表 10.3 student 表的定义

字段名	字段描述	数据类型	主键	外键	非空	唯一	自增
id	学号	INT(10)	是	否	是	是	否
name	姓名	VARCHAR(20)	否	否	是	否	否
sex	性别	VARCHAR(4)	否	否	否	否	否
birth	出生年分	YEAR	否	否	否	否	否
department	院系	VARCHAR(20)	否	否	是	否	否
address	家庭住址	VARCHAR(50)	否	否	否	否	否

公众号：方志朋

表 10.4 score表的定义

字段名	字段描述	数据类型	主键	外键	非空	唯一	自增
id	编号	INT(10)	是	否	是	是	是
stu_id	学号	INT(10)	否	否	是	否	否
c_name	课程名	VARCHAR(20)	否	否	否	否	否
grade	分数	INT(10)	否	否	否	否	否

student 表和 score 表中记录如表 10.5 和表 10.6 所示：

表 10.5 student表的记录

id	name	sex	birth	department	address
901	张老大	男	1985	计算机系	北京市海淀区
902	张老二	男	1986	中文系	北京市昌平区
903	张三	女	1990	中文系	湖南省永州市
904	李四	男	1990	英语系	辽宁省阜新市
905	王五	女	1991	英语系	福建省厦门市
906	王六	男	1988	计算机系	湖南省衡阳市

表 10.6 score表的记录

id	stu_id	c_name	grade
1	901	计算机	98
2	901	英语	80
3	902	计算机	65
4	902	中文	88
5	903	中文	95
6	904	计算机	70
7	904	英语	92
8	905	英语	94
9	906	计算机	90
10	906	英语	85

创建表

创建数据库：

```
1 CREATE DATABASE test DEFAULT CHARSET utf8 COLLATE utf8_general_ci;
```

创建student表

```
1 create table student(id int(10) not null unique primary key ,name
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
varchar(20) not null, sex varchar(4), birth year, department varcha  
r(20) ,address varchar(50));
```

创建score表

```
1 create table score(id int(10) not null unique primary key auto_in  
crement,stu_id int(10) not null,c_name varchar(20),grade int(10));
```

插入数据

插入学生：

```
1 insert into student values(901,'张老大','男',1985,'计算机系','北京市海  
淀区');  
2 insert into student values(902,'张老二','男',1986,'中文系','北京市昌平  
区');  
3 insert into student values(903,'张三','女',1990,'中文系','湖南省永州  
市');  
4 insert into student values(904,'李四','男',1990,'英语系','辽宁省阜新  
市');  
5 insert into student values(905,'王五','女',1991,'英语系','福建省厦门市');  
6 insert into student values(906,'王六','男',1988,'计算机系','湖南省衡阳  
市');
```

插入成绩：

```
1 insert into score values(null,901,'计算机',98);  
2 insert into score values(null,901,'英语',80);  
3 insert into score values(null,902,'计算机',65);  
4 insert into score values(null,902,'中文',88);  
5 insert into score values(null,903,'计算机',95);  
6 insert into score values(null,904,'计算机',70);  
7 insert into score values(null,904,'英语',92);  
8 insert into score values(null,905,'英语',94);
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
9 insert into score values(null,906,'计算机',90);
10 insert into score values(null,906,'英语',85);
```

查询学生表中的所有记录

```
1 select * from student;
2 select id ,name ,sex,birth,department,address from student;
```

查询 student表中2-4条记录

```
1 select * from student limit 1,3;
```

查询student学生的学号、姓名和院校信息

```
1 select id ,name ,department from student;
```

查询计算机系和英语系的学生的信息的两种方法

```
1 select * from student where department in ('计算机系','英语系');
2
3 select * from student where department ='计算机系' or department='英
语系';
```

查询年龄为18-22岁的学生

```
1 select name ,2009-birth as age from student;
2
3 select id ,name ,sex,2009-birth as age ,department,address from st
udent where 2009-birth between 18 and 22;
```

```
4  
5 select id ,name ,sex ,2009-birth as age ,department,address from s  
tudent where 2009-birth>=18 and 2009-birth <=22;
```

student表中查询每个院系有多少人

```
1 select department,count(id) from student group by department;  
2  
3 select department,count(id) as sum_of_department from student gro  
up by department;
```

从score 表中查询每个科目的最高分

```
1  
2 select c_name,max(grade) from score group by c_name;
```

查询李四的考试科目 (c_name)和考试成绩(grade).

```
1 select c_name ,grade from score where stu_id =(select id from stud  
ent where name='李四');
```

用连接查询的方式查询所有学生的信息和考试成绩

```
1  
2 select student.id ,name,sex,birth,department,address,c_name,grade f  
rom student,score where student.id=score.stu_id;  
3  
4 select s1.id ,name ,sex,birth ,department,address,c_name,grade fro  
m student s1,score s2 where s1.id =s2.stu_id;
```

公众号: 方志朋

计算每个学生的总成绩

```
1 select stu_id,sum(grade) from score group by stu_id;
```

如果要显示学生的姓名：

```
1 select student.id ,name ,sum(grade) from student ,score where student.id=score.stu_id group by student.id;
```

计算每个考试科目的平均成绩

```
1 select c_name ,avg(grade) from score group by c_name;
```

查询计算机成绩低于95分的学生成绩

```
1 select * from student where id in(select stu_id from score where c_name='计算机' and grade <95);
```

查询同时参加计算机和英语考试的学生信息

```
1 select * from student where id=any (select stu_id from score where stu_id in (select stu_id from score where c_name='计算机') and c_name ='英语');
```

将计算机成绩按从高到低进行排序

```
1 select stu_id,grade from score where c_name='计算机' order by grade desc;
```

从student表和score 表中查询出学号然后合并查询结果

```
1 select id from student union select stu_id from score;
```

查询姓张和姓王的同学的姓名、院系、考试科目和成绩。

```
1  
2 select student.id ,name,sex,birth,department,address,c_name,grade  
      from student,score where (name like '张%' or name like '王%') and  
      student.id =score.stu_id;
```

查询都是湖南的同学的姓名、年龄、院系、考试科目和成绩

```
1 select student.id ,name ,sex,birth ,department,address,c_name,grad  
e from student,score where address like '湖南%' and student.id=scr  
o.e.stu_id;
```

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

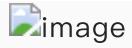


程序员理财，请关注：

java架构师公众号：方志朋

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



公众号：方志朋

数据库：Mysql数据类型

Mysql数据类型

mysql 数据类型

MySQL数据类型	含义 (有符号)
tinyint(m)	1个字节 范围(-128~127)
smallint(m)	2个字节 范围(-32768~32767)
mediumint(m)	3个字节 范围(-8388608~8388607)
int(m)	4个字节 范围(-2147483648~2147483647)
bigint(m)	8个字节 范围(+-9.22*10的18次方)

取值范围如果加了unsigned，则最大值翻倍，如tinyint unsigned的取值范围为(0~256)。

int(m)里的m是表示SELECT查询结果集中的显示宽度，并不影响实际的取值范围，没有影响到显示的宽度，不知道这个m有什么用。

浮点型(float和double)

MySQL数据类型	含义
float(m,d)	单精度浮点型 8位精度(4字节) m总个数, d小数位
double(m,d)	双精度浮点型 16位精度(8字节) m总个数, d小数位

设一个字段定义为float(5,3)，如果插入一个数123.45678，实际数据库里存的是123.457，但总个数还以实际为准，即6位。

定点数

浮点型在数据库中存放的是近似值，而定点类型在数据库中存放的是精确值。

decimal(m,d) 参数m<65 是总个数，d<30且 d<m 是小数位。

字符串(char,varchar,_text)

MySQL数据类型	含义
char(n)	固定长度，最多255个字符
varchar(n)	固定长度，最多65535个字符
tinytext	可变长度，最多255个字符
text	可变长度，最多65535个字符
mediumtext	可变长度，最多2的24次方-1个字符
longtext	可变长度，最多2的32次方-1个字符

char和varchar：

- 1.char(n) 若存入字符数小于n，则以空格补于其后，查询之时再将空格去掉。所以char类型存储的字符串末尾不能有空格，varchar不限于此。
- 2.char(n) 固定长度，char(4)不管是存入几个字符，都将占用4个字节，varchar是存入的实际字符数+1个字节 (n<=255) 或2个字节(n>255)，所以varchar(4),存入3个字符将占用4个字节。
- 3.char类型的字符串检索速度要比varchar类型的快。

varchar和text：

- 1.varchar可指定n，text不能指定，内部存储varchar是存入的实际字符数+1个字节 (n<=255) 或2个字节(n>255)，text是实际字符数+2个字节。
- 2.text类型不能有默认值。
- 3.varchar可直接创建索引，text创建索引要指定前多少个字符。varchar查询速度快于text,在都创建索引的情况下，text的索引似乎不起作用。

二进制数据(_Blob)

- 1.BLOB和text存储方式不同，TEXT以文本方式存储，英文存储区分大小写，而Blob是以二进制方式存储，不分大小写。
- 2._BLOB存储的数据只能整体读出。
- 3.TEXT可以指定字符集，BLO不用指定字符集。

日期时间类型

MySQL数据类型	含义
date	日期 '2008-12-2'

time	时间 '12:25:36'
datetime	日期时间 '2008-12-2 22:06:44'
timestamp	自动存储记录修改时间

若定义一个字段为timestamp，这个字段里的时间数据会随其他字段修改的时候自动刷新，所以这个数据类型的字段可以存放这条记录最后被修改的时间。

数据类型的属性

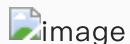
MySQL关键字	含义
NULL	数据列可包含NULL值
NOT NULL	数据列不允许包含NULL值
DEFAULT	默认值
PRIMARY KEY	主键
AUTO_INCREMENT	自动递增，适用于整数类型
UNSIGNED	无符号
CHARACTER SET name	指定一个字符集

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

公众号：方志朋

数据库：MySQL几种常用的存储引擎区别

MySQL是我们经常使用的数据库处理系统（DBMS），不知小伙伴们有没有注意过其中的“存储引擎”（storage_engine）呢？有时候面试题中也会问道MySQL几种常用的存储引擎的区别。这次就简短侃一下存储引擎那些事儿。

先去查一下“引擎”概念。

引擎（Engine）是电子平台上开发程序或系统的核心组件。利用引擎，开发者可迅速建立、铺设程序所需的功能，或利用其辅助程序的运转。一般而言，引擎是一个程序或一套系统>的支持部分。常见的程序引擎有游戏引擎，搜索引擎，杀毒引擎等。

k，我们知道了，引擎就是一个程序的核心组件。

简单来说，存储引擎就是指表的类型以及表在计算机上的存储方式。

存储引擎的概念是MySQL的特点，Oracle中没有专门的存储引擎的概念，Oracle有OLTP和OLAP模式的区分。不同的存储引擎决定了MySQL数据库中的表可以用不同的方式来存储。我们可以根据数据的特点来选择不同的存储引擎。

在MySQL中的存储引擎有很多种，可以通过“SHOW ENGINES”语句来查看。下面重点关注InnoDB、MyISAM、MEMORY这三种。

InnoDB存储引擎

InnoDB给MySQL的表提供了事务处理、回滚、崩溃修复能力和多版本并发控制的事务安全。在MySQL从3.23.34a开始包含InnoDB。它是MySQL上第一个提供外键约束的表引擎。而且InnoDB对事务处理的能力，也是其他存储引擎不能比拟的。靠后版本的MySQL的默认存储引擎就是InnoDB。

InnoDB存储引擎总支持AUTO_INCREMENT。自动增长列的值不能为空，并且值必须唯一。MySQL中规定自增列必须为主键。在插入值的时候，如果自动增长列不输入值，则插入的值为自动增长后的值；如果输入的值为0或空（NULL），则插入的值也是自动增长后的值；如果插入某个确定的值，且该值在前面没有出现过，就可以直接插入。

InnoDB还支持外键（FOREIGN KEY）。外键所在的表叫做子表，外键所依赖（REFERENCES）的表叫做父表。父表中被子表外键关联的字段必须为主键。当删除、更新父表中的某条信息时，子表也必须有相

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！
应的改变，这是数据库的参照完整性规则。

InnoDB中，创建的表的表结构存储在.frm文件中（我觉得是frame的缩写吧）。数据和索引存储在innodb_data_home_dir和innodb_data_file_path定义的表空间中。

InnoDB的优势在于提供了良好的事务处理、崩溃修复能力和并发控制。缺点是读写效率较差，占用的数据空间相对较大。

MyISAM存储引擎

MyISAM是MySQL中常见的存储引擎，曾经是MySQL的默认存储引擎。MyISAM是基于ISAM引擎发展起来的，增加了许多有用的扩展。

MyISAM的表存储成3个文件。文件的名字与表名相同。拓展名为frm、MYD、MYI。其实，frm文件存储表的结构；MYD文件存储数据，是MYData的缩写；MYI文件存储索引，是MYIndex的缩写。

基于MyISAM存储引擎的表支持3种不同的存储格式。包括静态型、动态型和压缩型。其中，静态型是MyISAM的默认存储格式，它的字段是固定长度的；动态型包含变长字段，记录的长度不是固定的；压缩型需要用到myisampack工具，占用的磁盘空间较小。

MyISAM的优势在于占用空间小，处理速度快。缺点是不支持事务的完整性和并发性。

MEMORY存储引擎

MEMORY是MySQL中一类特殊的存储引擎。它使用存储在内存中的内容来创建表，而且数据全部放在内存中。这些特性与前面的两个很不同。

每个基于MEMORY存储引擎的表实际对应一个磁盘文件。该文件的文件名与表名相同，类型为frm类型。该文件中只存储表的结构。而其数据文件，都是存储在内存中，这样有利于数据的快速处理，提高整个表的效率。值得注意的是，服务器需要有足够的内存来维持MEMORY存储引擎的表的使用。如果不需要了，可以释放内存，甚至删除不需要的表。

MEMORY默认使用哈希索引。速度比使用B型树索引快。当然如果你想用B型树索引，可以在创建索引时指定。

注意，MEMORY用到的很少，因为它是把数据存到内存中，如果内存出现异常就会影响数据。如果重启或者关机，所有数据都会消失。因此，基于MEMORY的表的生命周期很短，一般是一次性的。

怎样选择存储引擎

在实际工作中，选择一个合适的存储引擎是一个比较复杂的问题。每种存储引擎都有自己的优缺点，不能笼统地说谁比谁好。



InnoDB：支持事务处理，支持外键，支持崩溃修复能力和并发控制。如果需要对事务的完整性要求比较高（比如银行），要求实现并发控制（比如售票），那选择InnoDB有很大的优势。如果需要频繁的更新、删除操作的数据库，也可以选择InnoDB，因为支持事务的提交（commit）和回滚（rollback）。

MyISAM：插入数据快，空间和内存使用比较低。如果表主要是用于插入新记录和读出记录，那么选择MyISAM能实现处理高效率。如果应用的完整性、并发性要求比较低，也可以使用。

MEMORY：所有的数据都在内存中，数据的处理速度快，但是安全性不高。如果需要很快的读写速度，对数据的安全性要求较低，可以选择MEMOYEY。它对表的大小有要求，不能建立太大的表。所以，这类数据库只使用在相对较小的数据库表。

注意，同一个数据库也可以使用多种存储引擎的表。如果一个表要求比较高的事务处理，可以选择InnoDB。这个数据库中可以将查询要求比较高的表选择MyISAM存储。如果该数据库需要一个用于查询的临时表，可以选择MEMORY存储引擎。

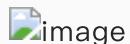
公众号：方志朋

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

公众号：方志朋

数据库：数据库索引优化

数据库索引优化

摘要

本文以MySQL数据库为研究对象，讨论与数据库索引相关的一些话题。特别需要说明的是，MySQL支持诸多存储引擎，而各种存储引擎对索引的支持也各不相同，因此MySQL数据库支持多种索引类型，如BTree索引，哈希索引，全文索引等等。为了避免混乱，本文将只关注于BTree索引，因为这是平常使用MySQL时主要打交道的索引，至于哈希索引和全文索引本文暂不讨论。

常见的查询算法及数据结构

为什么这里要讲查询算法和数据结构呢？因为之所以要建立索引，其实就是为了构建一种数据结构，可以在上面应用一种高效的查询算法，最终提高数据的查询速度。

索引的本质

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。提取句子主干，就可以得到索引的本质：索引是数据结构。

常见的查询算法

我们知道，数据库查询是数据库的最主要功能之一。我们都希望查询数据的速度能尽可能的快，因此数据库系统的设计者会从查询算法的角度进行优化。那么有哪些查询算法可以使查询速度变得更快呢？

顺序查找（linear search）

最基本的查询算法当然是顺序查找（linear search），也就是对比每个元素的方法，不过这种算法在数据量很大时效率是极低的。

- 数据结构：有序或无序队列
- 复杂度： $O(n)$

```
1 //顺序查找
2 int SequenceSearch(int a[], int value, int n)
3 {
4     int i;
5     for(i=0; i<n; i++)
6         if(a[i]==value)
7             return i;
8     return -1;
9 }
```

二分查找 (binary search)

比顺序查找更快的查询方法应该就是二分查找了，二分查找的原理是查找过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组为空，则代表找不到。

- 数据结构：有序数组
- 复杂度：O(logn)

公众号：方志朋

```
1 //二分查找，递归版本
2 int BinarySearch2(int a[], int value, int low, int high)
3 {
4     int mid = low+(high-low)/2;
5     if(a[mid]==value)
6         return mid;
7     if(a[mid]>value)
8         return BinarySearch2(a, value, low, mid-1);
9     if(a[mid]<value)
10        return BinarySearch2(a, value, mid+1, high);
11 }
```

二叉排序树查找

二叉排序树的特点是：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 它的左、右子树也分别为二叉排序树。

搜索的原理：

- 若b是空树，则搜索失败，否则：
- 若x等于b的根节点的数据域之值，则查找成功；否则：
- 若x小于b的根节点的数据域之值，则搜索左子树；否则：查找右子树
- 数据结构：二叉排序树
- 时间复杂度： $O(\log_2 N)$

哈希散列法(哈希表)

其原理是首先根据key值和哈希函数创建一个哈希表（散列表），然后根据键值，通过散列函数，定位数据元素位置。

- 数据结构：哈希表
- 时间复杂度：几乎是 $O(1)$ ，取决于产生冲突的多少。

分块查找

分块查找又称索引顺序查找，它是顺序查找的一种改进方法。其算法思想是将n个数据元素“按块有序”划分为m块 ($m \leq n$)。每一块中的结点不必有序，但块与块之间必须“按块有序”；即第1块中任一元素的关键字都必须小于第2块中任一元素的关键字；而第2块中任一元素又都必须小于第3块中的任一元素，依次类推。

算法流程：

- 先选取各块中的最大关键字构成一个索引表；
- 查找分两个部分：先对索引表进行二分查找或顺序查找，以确定待查记录在哪一块中；然后，在已确定的块中用顺序法进行查找。

这种搜索算法每一次比较都使搜索范围缩小一半。它们的查询速度就有了很大的提升。如果稍微分析一下会发现，每种查找算法都只能应用于特定的数据结构之上，例如二分查找要求被检索数据有序，而二叉树查找只能应用于二叉查找树上，但是数据本身的组织结构不可能完全满足各种数据结构（例如，理论上不可能同时将两列都按顺序进行组织），所以，在数据之外，数据库系统还维护着满足特定查找算法的数据

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

平衡多路搜索树B树 (B-tree)

上面讲到了二叉树，它的搜索时间复杂度为 $O(\log_2 N)$ ，所以它的搜索效率和树的深度有关，如果要提高查询速度，那么就要降低树的深度。要降低树的深度，很自然的方法就是采用多叉树，再结合平衡二叉树的思想，我们可以构建一个平衡多叉树结构，然后就可以在上面构建平衡多路查找算法，提高大数据量下的搜索效率。

B Tree

B树 (Balance Tree) 又叫做B- 树（其实B-是由B-tree翻译过来，所以B-树和B树是一个概念），它就是一种平衡路查找树。下图就是一个典型的B树：



从上图中我们可以大致看到B树的一些特点，为了更好的描述B树，我们定义记录为一个二元组[key, data]，key为记录的键值，data表示其它数据（上图中只有key，没有画出data数据）。下面是对B树的一个详细定

- 有一个根节点，根节点只有一个记录和两个孩子或者根节点为空；
- 每个节点记录中的key和指针相互间隔，指针指向孩子节点；
- d是表示树的宽度，除叶子节点之外，其它每个节点有 $[d/2, d-1]$ 条记录，并且些记录中的key都是从左到右按大小排列的，有 $[d/2+1, d]$ 个孩子；
- 在一个节点中，第n个子树中的所有key，小于这个节点中第n个key，大于第n-1个key，比如上图中B节点的第2个子节点E中的所有key都小于B中的第2个key 9，大于第1个key 3；
- 所有的叶子节点必须在同一层次，也就是它们具有相同的深度；

由于B-Tree的特性，在B-Tree中按key检索数据的算法非常直观：首先从根节点进行二分查找，如果找到则返回对应节点的数据，否则对相应区间的指针指向的节点递归进行查找，直到找到节点或找到null指针，前者查找成功，后者查找失败。B-Tree上查找算法的伪代码如下：

```
1 BTREE_Search(node, key) {  
2     if(node == null) return null;  
3     foreach(node.key){  
4         if(node.key[i] == key) return node.data[i];  
5     }  
6 }
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
5         if(node.key[i] > key) return BTREE_Search(point[i]->node
6     );
7     return BTREE_Search(point[i+1]->node);
8 }
9 data = BTREE_Search(root, my_key);
```

关于B-Tree有一系列有趣的性质，例如一个度为d的B-Tree，设其索引N个key，则其树高h的上限为 $\log d((N+1)/2)$ ，检索一个key，其查找节点个数的渐进复杂度为 $O(\log d N)$ 。从这点可以看出，B-Tree是一个非常有效率的索引数据结构。

另外，由于插入删除新的数据记录会破坏B-Tree的性质，因此在插入删除时，需要对树进行一个分裂、合并、转移等操作以保持B-Tree性质，本文不打算完整讨论B-Tree这些内容，因为已经有许多资料详细说明了B-Tree的数学性质及插入删除算法，有兴趣的朋友可以查阅其它文献进行详细研究。

B+Tree

其实B-Tree有许多变种，其中最常见的是B+Tree，比如MySQL就普遍使用B+Tree实现其索引结构。B-Tree相比，B+Tree有以下不同点：

- 每个节点的指针上限为 $2d$ 而不是 $2d+1$ ；
- 内节点不存储data，只存储key；
- 叶子节点不存储指针；

下面是一个简单的B+Tree示意



由于并不是所有节点都具有相同的域，因此B+Tree中叶节点和内节点一般大小不同。这点与B-Tree不同，虽然B-Tree中不同节点存放的key和指针可能数量不一致，但是每个节点的域和上限是一致的，所以在实现中B-Tree往往对每个节点申请同等大小的空间。一般来说，B+Tree比B-Tree更适合实现外存储索引结构，具体原因与外存储器原理及计算机存取原理有关，将在下面讨论。

带有顺序访问指针的B+Tree

一般在数据库系统或文件系统中使用的B+Tree结构都在经典B+Tree的基础上进行了优化，增加了顺序访问指针。



如图所示，在B+Tree的每个叶子节点增加一个指向相邻叶子节点的指针，就形成了带有顺序访问指针的B+Tree。做这个优化的目的是为了提高区间访问的性能，例如图4中如果要查询key为从18到49的所有数据记录，当找到18后，只需顺着节点和指针顺序遍历就可以一次性访问到所有数据节点，极大提升了区间查询效率。

这一节对B-Tree和B+Tree进行了一个简单的介绍，下一节结合存储器存取原理介绍为什么目前B+Tree是数据库系统实现索引的首选数据结构。

索引数据结构相关的计算机原理

上文说过，二叉树、红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B-/+Tree作为索引结构，这一节将结合计算机组成原理相关知识讨论B-/+Tree作为索引的理论基础。

两种类型的存储

在计算机系统中一般包含两种类型的存储，计算机主存（RAM）和外部存储器（如硬盘、CD、SSD等）。在设计索引算法和存储结构时，我们必须要考虑到这两种类型的存储特点。主存的读取速度快，相对于主存，外部磁盘的数据读取速率要比主存好几个数量级，具体它们之间的差别后面会详细介绍。上面讲的所有查询算法都是假设数据存储在计算机主存中的，计算机主存一般比较小，实际数据库中数据都是存储到外部存储器的。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。下面详细介绍内存和磁盘存取原理，然后再结合这些原理分析B-/+Tree作为索引的效率。

存取原理

目前计算机使用的主存基本都是随机读写存储器（RAM），现代RAM的结构和存取原理比较复杂，这里本文抛却具体差别，抽象出一个十分简单的存取模型来说明RAM的工作原理。



从抽象角度看，主存是一系列的存储单元组成的矩阵，每个存储单元存储固定大小的数据。每个存储单元有唯一的地址，现代主存的编址规则比较复杂，这里将其简化成一个二维地址：通过一个行地址和一个列

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

地址可以唯一定位到一个存储单元。上图展示了一个 4×4 的主存模型。

主存的存取过程如下：

当系统需要读取主存时，则将地址信号放到地址总线上传给主存，主存读到地址信号后，解析信号并定位到指定存储单元，然后将此存储单元数据放到数据总线上，供其它部件读取。写主存的过程类似，系统将要写入单元地址和数据分别放在地址总线和数据总线上，主存读取两个总线的内容，做相应的写操作。

这里可以看出，主存存取的时间仅与存取次数呈线性关系，因为不存在机械操作，两次存取的数据的“距离”不会对时间有任何影响，例如，先取A0再取A1和先取A0再取D3的时间消耗是一样的。

磁盘存取原理

上文说过，索引一般以文件形式存储在磁盘上，索引检索需要磁盘I/O操作。与主存不同，磁盘I/O存在机械运动耗费，因此磁盘I/O的时间消耗是巨大的。

磁盘读取数据靠的是机械运动，当需要从磁盘读取数据时，系统会将数据逻辑地址传给磁盘，磁盘的控制电路按照寻址逻辑将逻辑地址翻译成物理地址，即确定要读的数据在哪个磁道，哪个扇区。为了读取这个扇区的数据，需要将磁头放到这个扇区上方，为了实现这一点，磁头需要移动对准相应磁道，这个过程叫做寻道，所耗费时间叫做寻道时间，然后磁盘旋转将目标扇区旋转到磁头下，这个过程耗费的时间叫做旋转时间，最后便是对读取数据的传输。所以每次读取数据花费的时间可以分为寻道时间、旋转延迟、传输时间三个部分。其中：

- 寻道时间是磁臂移动到指定磁道所需要的时间，主流磁盘一般在5ms以下。
- 旋转延迟就是我们经常听说的磁盘转速，比如一个磁盘7200转，表示每分钟能转7200次，也就是说1秒钟能转120次，旋转延迟就是 $1/120/2 = 4.17\text{ms}$ 。
- 传输时间指的是从磁盘读出或将数据写入磁盘的时间，一般在零点几毫秒，相对于前两个时间可以忽略不计。

那么访问一次磁盘的时间，即一次磁盘IO的时间约等于 $5+4.17 = 9\text{ms}$ 左右，听起来还挺不错的，但要知道一台500 -MIPS的机器每秒可以执行5亿条指令，因为指令依靠的是电的性质，换句话说执行一次IO的时间可以执行40万条指令，数据库动辄十百万乃至千万级数据，每次9毫秒的时间，显然是个灾难。

局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

放入内存。这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

数据库索引所采用的数据结构B-/+Tree及其性能分析

到这里终于可以分析为何数据库索引采用B-/+Tree存储结构了。上文说过数据库索引是存储到磁盘的而我们又一般以使用磁盘I/O次数来评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问 $h-1$ 个节点（根节点常驻内存）。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

B-Tree中一次检索最多需要 $h-1$ 次I/O（根节点常驻内存），渐进复杂度为 $O(h)=O(\log d N)$ 。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

综上所述，如果我们采用B-Tree存储结构，搜索时I/O次数一般不会超过3次，所以用B-Tree作为索引结构效率是非常高的。

B+树性能分析

从上面介绍我们知道，B树的搜索复杂度为 $O(h)=O(\log d N)$ ，所以树的出度d越大，深度h就越小，I/O的次数就越少。B+Tree恰恰可以增加出度d的宽度，因为每个节点大小为一个页大小，所以出度的上限取决于节点内key和data的大小：

1

2 `dmax=floor(pagesize/(keysize+datasize+pointsize))`//floor表示向下取整

由于B+Tree内节点去掉了data域，因此可以拥有更大的出度，从而拥有更好的性能。

B+树查找过程



B-树和B+树查找过程基本一致。如上图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常高。

这一章从理论角度讨论了与索引相关的数据结构与算法问题，下一章将讨论B+Tree是如何具体实现为MySQL中索引，同时将结合MyISAM和InnoDB存储引擎介绍非聚集索引和聚集索引两种不同的索引实现形式。

MySQL索引实现

在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的，本文主要讨论MyISAM和InnoDB两个存储引擎的索引实现方式。

MyISAM索引实现

MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。下图是MyISAM索引的原理图：



这里设表一共有三列，假设我们以Col1为主键，则上图是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

相应数据记录。

MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

InnoDB索引实现

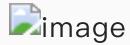
虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

第一个重大区别是InnoDB的数据文件本身就是索引文件。从上文知道，MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



上图是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，下图为定义在Col3上的一个辅助索引：



这里以英文字符的ASCII码作为比较准则。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

下一章将具体讨论这些与索引有关的优化策略。

索引使用策略及优化

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

MySQL的优化主要分为结构优化（Scheme optimization）和查询优化（Query optimization）。本章讨论的高性能索引策略主要属于结构优化范畴。本章的内容完全基于上文的理论基础，实际上一旦理解了索引背后的机制，那么选择高性能的策略就变成了纯粹的推理，并且可以理解这些策略背后的逻辑。

联合索引及最左前缀原理

联合索引（复合索引）

首先介绍一下联合索引。联合索引其实很简单，相对于一般索引只有一个字段，联合索引可以为多个字段创建一个索引。它的原理也很简单，比如，我们在（a,b,c）字段上创建一个联合索引，则索引记录会首先按照A字段排序，然后再按照B字段排序然后再是C字段，因此，联合索引的特点就是：

- 第一个字段一定是有序的
- 当第一个字段值相等的时候，第二个字段又是有序的，比如下表中当A=2时所有B的值是有序排列的，依次类推，当同一个B值得所有C字段是有序排列的

1		A		B		C	
2		1		2		3	
3		1		4		2	
4		1		1		4	
5		2		3		5	
6		2		4		4	
7		2		4		6	
8		2		5		5	

其实联合索引的查找就跟查字典是一样的，先根据第一个字母查，然后再根据第二个字母查，或者只根据第一个字母查，但是不能跳过第一个字母从第二个字母开始查。这就是所谓的最左前缀原理。

最左前缀原理

我们再来详细介绍下联合索引的查询。还是上面例子，我们在（a,b,c）字段上建了一个联合索引，所以这个索引是先按a再按b再按c进行排列的，所以：

以下的查询方式都可以用到索引

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
1 select * from table where a=1;  
2 select * from table where a=1 and b=2;  
3 select * from table where a=1 and b=2 and c=3;
```

上面三个查询按照 (a) , (a, b) , (a, b, c) 的顺序都可以利用到索引，这就是最左前缀匹配。

如果查询语句是：

```
1 select * from table where a=1 and c=3; 那么只会用到索引a。
```

如果查询语句是：

```
1 select * from table where b=2 and c=3; 因为没有用到最左前缀a，所以这个查询是用户到索引的。
```

如果用到了最左前缀，但是顺序颠倒会用到索引码？

```
1 select * from table where b=2 and a=1;  
2 select * from table where b=2 and a=1 and c=3;
```

如果用到了最左前缀而只是颠倒了顺序，也是可以用到索引的，因为mysql查询优化器会判断纠正这条sql语句该以什么样的顺序执行效率最高，最后才生成真正的执行计划。但我们还是最好按照索引顺序来查询，这样查询优化器就不用重新编译了。

前缀索引

除了联合索引之外，对mysql来说其实还有一种前缀索引。前缀索引就是用列的前缀代替整个列作为索引key，当前缀长度合适时，可以做到既使得前缀索引的选择性接近全列索引，同时因为索引key变短而减少了索引文件的大小和维护开销。

一般来说以下情况可以使用前缀索引：

- 字符串列(varchar,char,text等)，需要进行全字段匹配或者前匹配。也就是='xxx' 或者 like 'xxx%'

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 字符串本身可能比较长，而且前几个字符就开始不相同。比如我们对中国人的姓名使用前缀索引就没啥意义，因为中国人名字都很短，另外对收件地址使用前缀索引也不是很实用，因为一方面收件地址一般都是以XX省开头，也就是说前几个字符都是差不多的，而且收件地址进行检索一般都是like '%xxx%'，不会用到前匹配。相反对外国人的姓名可以使用前缀索引，因为其字符较长，而且前几个字符的选择性比较高。同样电子邮件也是一个可以使用前缀索引的字段。
- 前一半字符的索引选择性就已经接近于全字段的索引选择性。如果整个字段的长度为20，索引选择性为0.9，而我们对前10个字符建立前缀索引其选择性也只有0.5，那么我们需要继续加大前缀字符的长度，但是这个时候前缀索引的优势已经不明显，没有太大的建前缀索引的必要了。

一些文章中也提到：

MySQL 前缀索引能有效减小索引文件的大小，提高索引的速度。但是前缀索引也有它的坏处：MySQL 不能在 ORDER BY 或 GROUP BY 中使用前缀索引，也不能把它们用作覆盖索引(Covering Index)。

索引优化策略

- 最左前缀匹配原则，上面讲到了
- 主键外检一定要建索引
- 对 where, on, group by, order by 中出现的列使用索引
- 尽量选择区分度高的列作为索引，区分度的公式是 $\text{count}(\text{distinct col})/\text{count}(\star)$ ，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0
- 对较小的数据列使用索引，这样会使索引文件更小，同时内存中也可以装载更多的索引键
- 索引列不能参与计算，保持列“干净”，比如 `from_unixtime(create_time) = '2014-05-29'` 就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成 `create_time = unix_timestamp('2014-05-29')`；
- 为较长的字符串使用前缀索引
- 尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可
- 不要过多创建索引，权衡索引个数与DML之间关系，DML也就是插入、删除数据操作。这里需要权衡一个问题，建立索引的目的是为了提高查询效率的，但建立的索引过多，会影响插入、删除数据的速度，因为我们修改的表数据，索引也需要进行调整重建
- 对于like查询，“%”不要放在前面。

```
1 SELECT * FROM houdunwang WHERE uname LIKE '后盾%' -- 走索引
2 SELECT * FROM houdunwang WHERE uname LIKE "%后盾%" -- 不走索引
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 查询where条件数据类型不匹配也无法使用索引

字符串与数字比较不使用索引；

```
1 CREATE TABLEa(achar(10));
2 EXPLAIN SELECT * FROMaWHEREa="1" - 走索引
3 EXPLAIN SELECT * FROM a WHERE a=1 - 不走索引
```

- 正则表达式不使用索引,这应该很好理解,所以为什么在SQL中很难看到regexp关键字的原因

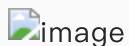
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



数据库：数据库索引的类型

索引是对数据库表中一列或多列的值进行排序的一种结构，例如 employee 表的姓（name）列。如果要按姓查找特定职员，与必须搜索表中的所有行相比，索引会帮助您更快地获得该信息。

索引是一个单独的、物理的数据库结构，它是某个表中一列或若干列值的集合和相应的指向表中物理标识这些值的数据页的逻辑指针清单。 索引提供指向存储在表的指定列中的数据值的指针，然后根据您指定的排序顺序对这些指针排序。数据库使用索引的方式与您使用书籍中的索引的方式很相似：它搜索索引以找到特定值，然后顺指针找到包含该值的行。

在数据库关系图中，您可以在选定表的“索引/键”属性页中创建、编辑或删除每个索引类型。当保存索引所附加到的表，或保存该表所在的关系图时，索引将保存在数据库中。

可以基于数据库表中的单列或多列创建索引。多列索引使您可以区分其中一列可能有相同值的行。

如果经常同时搜索两列或多列或按两列或多列排序时，索引也很有帮助。例如，如果经常在同一查询中为姓和名两列设置判据，那么在这两列上创建多列索引将很有意义。 确定索引的有效性： 检查查询的 WHERE 和 JOIN 子句。在任一子句中包括的每一列都是索引可以选择的对象。 对新索引进行试验以检查它对运行查询性能的影响。 考虑已在表上创建的索引数量。最好避免在单个表上有很多索引。 检查已在表上创建的索引的定义。最好避免包含共享列的重叠索引。 检查某列中唯一数据值的数量，并将该数量与表中的行数进行比较。比较的结果就是该列的可选择性，这有助于确定该列是否适合建立索引，如果适合，确定索引的类型。

建立索引的优点：

- 1.大大加快数据的检索速度；
- 2.创建唯一性索引，保证数据库表中每一行数据的唯一性；
- 3.加速表和表之间的连接；
- 4.在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间。

索引类型：

根据数据库的功能，可以在数据库设计器中创建四种索引：唯一索引、非唯一索引、主键索引和聚集索引。 尽管唯一索引有助于定位信息，但为获得最佳性能结果，建议改用主键或唯一约束。

唯一索引：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

唯一索引是不允许其中任何两行具有相同索引值的索引。当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在 employee 表中职员的姓 (lname) 上创建了唯一索引，则任何两个员工都不能同姓。

非唯一索引：

非唯一索引是相对唯一索引，允许其中任何两行具有相同索引值的索引。当现有数据中存在重复的键值时，数据库是允许将新创建的索引与表一起保存。这时数据库不能防止添加将在表中创建重复键值的新数据。

主键索引：

数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

聚集索引（也叫聚簇索引）：

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

公众号：方志朋

数据库：数据库连接池原理详解与自定义连接池实现

数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，这些数据库连接的数量是由最小数据库连接数制约。无论这些数据库连接是否被使用，连接池都将一直保证至少拥有这么多的连接数量。连接池的最大数据库连接数量限定了这个连接池能占有的最大连接数，当应用程序向连接池请求的连接数超过最大连接数量时，这些请求将被加入到等待队列中。

连接池基本的思想是在系统初始化的时候，将数据库连接作为对象存储在内存中，当用户需要访问数据库时，并非建立一个新的连接，而是从连接池中取出一个已建立的空闲连接对象。使用完毕后，用户也并非将连接关闭，而是将连接放回连接池中，以供下一个请求访问使用。而连接的建立、断开都由连接池自身来管理。同时，还可以通过设置连接池的参数来控制连接池中的初始连接数、连接的上下限数以及每个连接的最大使用次数、最大空闲时间等等。也可以通过其自身的管理机制来监视数据库连接的数量、使用情况等。

注意事项

- 1、数据库连接池的最小连接数是连接池一直保持的数据库连接，所以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。
- 2、数据库连接池的最大连接数是连接池能申请的最大连接数，如果数据库连接请求超过此数，后面的数据库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。
- 3、最大连接数具体值要看系统的访问量.要经过不断测试取一个平衡值
- 4、隔一段时间对连接池进行检测,发现小于最小连接数的则补充相应数量的新连接
- 5、最小连接数与最大连接数差距，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。

数据库连接池配置属性

目前数据库连接池种类繁多，不同种类基本的配置属性大同小异，例如c3p0、Proxool、
DDConnectionBroker、DBPool、XAPOOL、Druid、dbcp，这里我们以dbcp为例说说主要的配置项：

- 1 #最大连接数量：连接池在同一时间能够分配的最大活动连接的数量，如果设置为非正数则表示不限制，默认值8
- 2 maxActive=15

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
3 #最小空闲连接：连接池中容许保持空闲状态的最小连接数量，低于这个数量将创建新的连接，如果设置为0则不创建，默认值0
4 minIdle=5
5 #最大空闲连接：连接池中容许保持空闲状态的最大连接数量，超过的空闲连接将被释放，如果设置为负数表示不限制，默认值8
6 maxIdle=10
7 #初始化连接数：连接池启动时创建的初始化连接数量，默认值0
8 initialSize=5
9 #连接被泄露时是否打印
10 logAbandoned=true
11 #是否自动回收超时连接
12 removeAbandoned=true
13 #超时时间(以秒数为单位)
14 removeAbandonedTimeout=180
15 # 最大等待时间：当没有可用连接时，连接池等待连接被归还的最大时间(以毫秒计数)，超过时间则抛出异常，如果设置为-1表示无限等待，默认值无限
16 maxWait=3000
17 #在空闲连接回收器线程运行期间休眠的时间值(以毫秒为单位)。
18 timeBetweenEvictionRunsMillis=10000
19 #在每次空闲连接回收器线程(如果有)运行时检查的连接数量
20 numTestsPerEvictionRun=8
21 #连接在池中保持空闲而不被空闲连接回收器线程
22 minEvictableIdleTimeMillis=10000
23 #用来验证从连接池取出的连接
24 validationQuery=SELECT 1
25 #指明是否在从池中取出连接前进行检验
26 testOnBorrow=true
27 #testOnReturn false 指明是否在归还到池中前进行检验
28 testOnReturn=true
29 #设置为true后如果要生效，validationQuery参数必须设置为非空字符串
30 testWhileIdle
```

公众号：方志朋

自定义数据库连接池示例

首先看一下连接池的定义。它通过构造函数初始化连接的最大上限，通过一个双向队列来维护连接，调用方需要先调用fetchConnection(long)方法来指定在多少毫秒内超时获取连接，当连接使用完成后，需要调用releaseConnection(Connection)方法将连接放回线程池

```
1 public class ConnectionPool {  
2     private LinkedList<Connection> pool = new LinkedList<Connecti  
on>();  
3  
4     /**  
5      * 初始化连接池的大小  
6      * @param initialSize  
7      */  
8     public ConnectionPool(int initialSize) {  
9         if (initialSize > 0) {  
10             for (int i = 0; i < initialSize; i++) {  
11                 pool.addLast(ConnectionDriver.createConnection  
());  
12             }  
13         }  
14     }  
15  
16     /**  
17      * 释放连接，放回到连接池  
18      * @param connection  
19      */  
20     public void releaseConnection(Connection connection){  
21         if(connection != null){  
22             synchronized (pool) {  
23                 // 连接释放后需要进行通知，这样其他消费者能够感知到连接池中  
已经归还了一个连接  
24                 pool.addLast(connection);  
25                 pool.notifyAll();  
26             }  
27         }  
28     }  
29  
30     /**  
31      * 在mills内无法获取到连接，将会返回null  
32      * @param mills  
33      * @return  
34      * @throws InterruptedException  
35      */  
36     public Connection fetchConnection(long mills) throws Interrup
```

还有超过100本优质java高清电子书，添加微信cxymq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
tedException{
    synchronized (pool) {
        // 无限制等待
        if (mills <= 0) {
            while (pool.isEmpty()) {
                pool.wait();
            }
            return pool.removeFirst();
        }else{
            long future = System.currentTimeMillis() + mills;
            long remaining = mills;
            while (pool.isEmpty() && remaining > 0) {
                // 等待超时
                pool.wait(remaining);
                remaining = future - System.currentTimeMillis();
            };
        }
        Connection result = null;
        if (!pool.isEmpty()) {
            result = pool.removeFirst();
        }
        return result;
    }
}
```

方志朋：公众号

由于java.sql.Connection是一个接口，最终的实现是由数据库驱动提供方来实现的，考虑到只是个示例，我们通过动态代理构造了一个Connection，该Connection的代理实现仅仅是在commit()方法调用时休眠100毫秒

```
1
2 public class ConnectionDriver {
3     static class ConnectionHandler implements InvocationHandler{
4         @Override
5             public Object invoke(Object proxy, Method method, Object[
6                 ] args) throws Throwable {
7                 if(method.equals("commit")){
8                     return null;
9                 }
10            }
11        }
12    }
13}
```

```
7     TimeUnit.MILLISECONDS.sleep(100);
8 }
9     return null;
10}
11}
12
13 /**
14 * 创建一个Connection的代理，在commit时休眠100毫秒
15 * @return
16 */
17 public static final Connection createConnection(){
18     return (Connection) Proxy.newProxyInstance(ConnectionDriver.class.getClassLoader(),
19             new Class[] { Connection.class },new ConnectionHandler());
20 }
21 }
```

下面通过一个示例来测试简易数据库连接池的工作情况，模拟客户端ConnectionRunner获取、使用、最后释放连接的过程，当它使用时连接将会增加获取到连接的数量，反之，将会增加未获取到连接的数量

```
1 public class ConnectionPoolTest {
2     static ConnectionPool pool = new ConnectionPool(10);
3     // 保证所有ConnectionRunner能够同时开始
4     static CountDownLatch start = new CountDownLatch(1);
5     // main线程将会等待所有ConnectionRunner结束后才能继续执行
6     static CountDownLatch end;
7     public static void main(String[] args) {
8         // 线程数量，可以修改线程数量进行观察
9         int threadCount = 10;
10        end = new CountDownLatch(threadCount);
11        int count = 20;
12        AtomicInteger got = new AtomicInteger();
13        AtomicInteger notGot = new AtomicInteger();
14        for (int i = 0; i < threadCount; i++) {
15            Thread thread = new Thread(new ConnectionRunner(count,
16                got, notGot), "ConnectionRunnerThread");
17            thread.start();
```

公众号：方志朋

```
17     }
18     start.countDown();
19     try {
20         end.await();
21     } catch (InterruptedException e) {
22         e.printStackTrace();
23     }
24     System.out.println("total invoke: " + (threadCount * count));
25     System.out.println("got connection: " + got);
26     System.out.println("not got connection " + notGot);
27 }
28
29     static class ConnectionRunner implements Runnable {
30
31         int count;
32         AtomicInteger got;
33         AtomicInteger notGot;
34         public ConnectionRunner(int count, AtomicInteger got, AtomicInteger notGot) {
35             this.count = count;
36             this.got = got;
37             this.notGot = notGot;
38         }
39         @Override
40         public void run() {
41             try {
42                 start.await();
43             } catch (Exception ex) {
44             }
45             while (count > 0) {
46                 try {
47                     // 从线程池中获取连接，如果1000ms内无法获取到，将会返回null
48                     // 分别统计连接获取的数量got和未获取到的数量notGot
49                     Connection connection = pool.fetchConnection(
50                         1);
51                     if (connection != null) {
52                         try {
53                             connection.createStatement();
54                         
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
53                     connection.commit();
54             } finally {
55                 pool.releaseConnection(connection);
56                 got.incrementAndGet();
57             }
58         } else {
59             notGot.incrementAndGet();
60         }
61     } catch (Exception ex) {
62     } finally {
63         count--;
64     }
65 }
66 end.countDown();
67 }
68
69 }
70
71 }
```

公众号: 方志朋

CountDownLatch类是一个同步计数器,构造时传入int参数,该参数就是计数器的初始值,每调用一次countDown()方法,计数器减1,计数器大于0时,await()方法会阻塞程序继续执行CountDownLatch如其所写,是一个倒计数的锁存器,当计数减至0时触发特定的事件。利用这种特性,可以让主线程等待子线程的结束。这这里保证让所有的ConnetionRunner都执行完再执行main进行打印。

运行结果:

20个客户端

```
1 total invoke: 200
2 got connection: 200
3 not got connection 0
```

50个客户端

```
1 total invoke: 1000
2 got connection: 999
```

3 not got connection 1

在资源一定的情况下（连接池中的10个连接），随着客户端线程的逐步增加，客户端出现超时无法获取连接的比率不断升高。虽然客户端线程在这种超时获取的模式下会出现连接无法获取的情况，但是它能够保证客户端线程不会一直挂在连接获取的操作上，而是“按时”返回，并告知客户端连接获取出现问题，是系统的一种自我保护机制。数据库连接池的设计也可以复用到其他的资源获取的场景，针对昂贵资源（比如数据库连接）的获取都应该加以超时限制。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



Spring Boot最核心的27个干货注解，你了解多少？

Spring Boot方式的项目开发已经逐步成为Java应用开发领域的主流框架，它不仅可以方便地创建生产级的Spring应用程序，还能轻松地通过一些注解配置与目前比较火热的微服务框架SpringCloud集成。而Spring Boot之所以能够轻松地实现应用的创建及与其他框架快速集成，最核心的原因就在于它极大地简化了项目的配置，最大化地实现了“约定大于配置”的原则。然而基于Spring Boot虽然极大地方便了开发，但是也很容易让人“云里雾里”，特别是各种注解很容易让人“知其然而不知其所以然”。

所以，要想用好Spring Boot就必须对其提供的各类功能注解有一个全面而清晰地认识和理解。一方面可以提高基于Spring Boot的开发效率，另一方面也是面试中被问及框架原理时所必需要掌握的知识点。在接下来的内容中，小编就带大家一起来探究下Spring Boot的一些常用注解吧！

Spring相关6个注解

Spring Boot的有些注解也需要与Spring的注解搭配使用，这里小编梳理了在项目中与Spring Boot注解配合最为紧密的6个Spring基础框架的注解。如

@Configuration

从Spring3.0，@Configuration用于定义配置类，可替换xml配置文件，被注解的类内部包含有一个或多个被@Bean注解的方法，这些方法将会被AnnotationConfigApplicationContext或AnnotationConfigWebApplicationContext类进行扫描，并用于构建bean定义，初始化Spring容器。

```
1 @Configuration
2 public class TaskAutoConfiguration {
3
4     @Bean
5     @Profile("biz-electrfence-controller")
6     public BizElectrfenceControllerJob bizElectrfenceControllerJo
7         b() {
8             return new BizElectrfenceControllerJob();
9         }
}
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
10     @Bean
11     @Profile("biz-consume-1-datasync")
12     public BizBikeElectrFenceTradeSyncJob bizBikeElectrFenceTrade
13         SyncJob() {
14             return new BizBikeElectrFenceTradeSyncJob();
15 }
```

@ComponentScan

做过web开发的同学一定都有用过@Controller, @Service, @Repository注解，查看其源码你会发现，他们中有一个共同的注解@Component，没错@ComponentScan注解默认就会装配标识了@Controller, @Service, @Repository, @Component注解的类到spring容器中。

```
1 @ComponentScan(value = "com.abacus.check.api")
2 public class CheckApiApplication {
3     public static void main(String[] args) {
4         SpringApplication.run(CheckApiApplication.class, args);
5     }
6 }
```

@SpringBootApplication注解也包含了@ComponentScan注解，所以在使用中我们也可以通过@SpringBootApplication注解的scanBasePackages属性进行配置。

```
1 @SpringBootApplication(scanBasePackages = {"com.abacus.check.api",
2                                         "com.abacus.check.service"})
3 public class CheckApiApplication {
4     public static void main(String[] args) {
5         SpringApplication.run(CheckApiApplication.class, args);
6     }
7 }
```

@Conditional

还有超过100本优质java高清电子书，添加微信cxymysq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

@Conditional是Spring4新提供的注解，通过@Conditional注解可以根据代码中设置的条件装载不同的bean，在设置条件注解之前，先要把装载的bean类去实现Condition接口，然后对该实现接口的类设置是否装载的条件。Spring Boot注解中的@ConditionalOnProperty、@ConditionalOnBean等以@Conditional*开头的注解，都是通过集成了@Conditional来实现相应功能的。

@Import

通过导入的方式实现把实例加入springIOC容器中。可以在需要时将没有被Spring容器管理的类导入至Spring容器中。

```
1  
2 //类定义  
3 public class Square {}  
4  
5 public class Circular {}  
6  
7 //导入  
8 @Import({Square.class,Circular.class})  
9 @Configuration  
10 public class MainConfig{}
```

@ImportResource

和@Import类似，区别就是@ImportResource导入的是配置文件。

```
1 @ImportResource("classpath:spring-redis.xml")          //导入xml配置  
2  
3 public class CheckApiApplication {  
4     public static void main(String[] args) {  
5         SpringApplication.run(CheckApiApplication.class, args);  
6     }  
7 }
```

@Component

@Component是一个元注解，意思是它可以注解其他类注解，如@Controller @Service @Repository。带此注解的类被看作组件，当使用基于注解的配置和类路径扫描的时候，这些类就会被实例化。其他类级别的注解也可以被认定为是一种特殊类型的组件，比如@Controller控制器（注入服务）、@Service服务（注入dao）、@Repositorydao（实现dao访问）。@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注，作用就相当于 XML配置，。

Spring Boot最核心的20个注解

说完与Spring Boot密切相关的几个Spring基础注解后，下面我们就再一起看看Spring Boot提供的核心注解的内容吧！

@SpringBootApplication

这个注解是Spring Boot最核心的注解，用在 Spring Boot的主类上，标识这是一个 Spring Boot 应用，用来开启 Spring Boot 的各项能力。实际上这个注解是

@Configuration,@EnableAutoConfiguration,@ComponentScan三个注解的组合。由于这些注解一般都是一起使用，所以Spring Boot提供了一个统一的注解@SpringBootApplication。

```
1 @SpringBootApplication(exclude = {  
2     MongoAutoConfiguration.class,  
3     MongoDataAutoConfiguration.class,  
4     DataSourceAutoConfiguration.class,  
5     ValidationAutoConfiguration.class,  
6     MybatisAutoConfiguration.class,  
7     MailSenderAutoConfiguration.class,  
8 })  
9 public class API {  
10     public static void main(String[] args) {  
11         SpringApplication.run(API.class, args);  
12     }  
13 }
```

@EnableAutoConfiguration

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

允许 Spring Boot 自动配置注解，开启这个注解之后，Spring Boot 就能根据当前类路径下的包或者类来配置 Spring Bean。

如：当前类路径下有 Mybatis 这个 JAR 包，MybatisAutoConfiguration 注解就能根据相关参数来配置 Mybatis 的各个 Spring Bean。

@EnableAutoConfiguration实现的关键在于引入了AutoConfigurationImportSelector，其核心逻辑为 selectImports方法，逻辑大致如下：

- 从配置文件META-INF/spring.factories加载所有可能用到的自动配置类；
- 去重，并将exclude和excludeName属性携带的类排除；
- 过滤，将满足条件（@Conditional）的自动配置类返回；

```
1 @Target({ElementType.TYPE})
2 @Retention(RetentionPolicy.RUNTIME)
3 @Documented
4 @Inherited
5 @AutoConfigurationPackage
6 //导入AutoConfigurationImportSelector的子类
7 @Import({EnableAutoConfigurationImportSelector.class})
8 public @interface EnableAutoConfiguration {
9     String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautocon
figuration";
10
11     Class<?>[] exclude() default {};
12
13     String[] excludeName() default {};
14 }
```

@SpringBootConfiguration

这个注解就是 `@Configuration` 注解的变体，只是用来修饰是 Spring Boot 配置而已，或者可利于 Spring Boot 后续的扩展。

@ConditionalOnBean

还有超过100本优质java高清电子书，添加微信cxymysq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

@ConditionalOnBean(A.class)仅仅在当前上下文中存在A对象时，才会实例化一个Bean，也就是说只有当A.class 在spring的applicationContext中存在时，这个当前的bean才能够创建。

```
1 @Bean  
2 //当前环境上下文存在DefaultMQProducer实例时，才能创建RocketMQProducerLife  
cycle这个Bean  
3 @ConditionalOnBean(DefaultMQProducer.class)  
4 public RocketMQProducerLifecycle rocketMQLifecycle() {  
5     return new RocketMQProducerLifecycle();  
6 }
```

@ConditionalOnMissingBean

组合@Conditional注解，和@ConditionalOnBean注解相反，仅仅在当前上下文中不存在A对象时，才会实例化一个Bean。

```
1 @Bean  
2 //仅当当前环境上下文缺失RocketMQProducer对象时，才允许创建RocketMQProduc  
er Bean对象  
3 @ConditionalOnMissingBean(RocketMQProducer.class)  
4 public RocketMQProducer mqProducer() {  
5     return new RocketMQProducer();  
6 }
```

@ConditionalOnClass

组合 `@Conditional` 注解，可以仅当某些类存在于classpath上时候才创建某个Bean。

```
1 @Bean  
2 //当classpath中存在类HealthIndicator时，才创建HealthIndicator Bean对  
象  
3 @ConditionalOnClass(HealthIndicator.class)  
4 public HealthIndicator rocketMQProducerHealthIndicator(Map<Strin  
g, DefaultMQProducer> producers) {
```

公众号：方志朋

```
5     if (producers.size() == 1) {  
6         return new RocketMQProducerHealthIndicator(producers.val  
ues().iterator().next());  
7     }  
8 }
```

@ConditionalOnMissingClass

组合@Conditional注解，和@ConditionalOnMissingClass注解相反，当classpath中没有指定的 Class 才开启配置。

@ConditionalOnWebApplication

组合@Conditional 注解，当前项目类型是 WEB 项目才开启配置。当前项目有以下 3 种类型:ANY(任何 Web项目都匹配)、SERVLET (仅但基础的Servelet项目才会匹配) 、REACTIVE (只有基于响应的web 应用程序才匹配) 。

@ConditionalOnNotWebApplication

组合@Conditional注解，和@ConditionalOnWebApplication 注解相反，当前项目类型不是 WEB 项目 才开启配置。

@ConditionalOnProperty

组合 @Conditional 注解，当指定的属性有指定的值时才开启配置。具体操作是通过其两个属性name以及havingValue来实现的，其中name用来从application.properties中读取某个属性值，如果该值为空，则返回false;如果值不为空，则将该值与havingValue指定的值进行比较，如果一样则返回true;否则返回false。如果返回值为false，则该configuration不生效；为true则生效。

```
1 @Bean  
2 //匹配属性rocketmq.producer.enabled值是否为true  
3 @ConditionalOnProperty(value = "rocketmq.producer.enabled", havin  
gValue = "true", matchIfMissing = true)  
4 public RocketMQProducer mqProducer() {  
5     return new RocketMQProducer();
```

```
6 }
```

@ConditionalOnExpression

组合 [@Conditional](#) 注解，当 SpEL 表达式为 true 时才开启配置。

```
1 @Configuration
2 @ConditionalOnExpression("${enabled:false}")
3 public class BigpipeConfiguration {
4     @Bean
5     public OrderMessageMonitor orderMessageMonitor(ConfigContext configContext) {
6         return new OrderMessageMonitor(configContext);
7     }
8 }
```

@ConditionalOnJava

组合 [@Conditional](#) 注解，当运行的 Java JVM 在指定的版本范围时才开启配置。

@ConditionalOnResource

组合 [@Conditional](#) 注解，当类路径下有指定的资源才开启配置。

```
1 @Bean
2 @ConditionalOnResource(resources="classpath:shiro.ini")
3 protected Realm iniClasspathRealm(){
4     return new Realm();
5 }
```

@ConditionalOnJndi

组合 [@Conditional](#) 注解，当指定的 JNDI 存在时才开启配置。

@ConditionalOnCloudPlatform

组合 `@Conditional` 注解，当指定的云平台激活时才开启配置。

@ConditionalOnSingleCandidate

组合 `@Conditional` 注解，当指定的 class 在容器中只有一个 Bean，或者同时有多个但为首选时才开启配置。

@ConfigurationProperties

Spring Boot 可使用注解的方式将自定义的 properties 文件映射到实体 bean 中，比如 config.properties 文件。

```
1 @Data
2 @ConfigurationProperties("rocketmq.consumer")
3 public class RocketMQConsumerProperties extends RocketMQProperties {
4     private boolean enabled = true;
5
6     private String consumerGroup;
7
8     private MessageModel messageModel = MessageModel.CLUSTERING;
9
10    private ConsumeFromWhere consumeFromWhere = ConsumeFromWhere.CONSUME_FROM_LAST_OFFSET;
11
12    private int consumeThreadMin = 20;
13
14    private int consumeThreadMax = 64;
15
16    private int consumeConcurrentlyMaxSpan = 2000;
17
18    private int pullThresholdForQueue = 1000;
19
20    private int pullInterval = 0;
```

```
21  
22     private int consumeMessageBatchMaxSize = 1;  
23  
24     private int pullBatchSize = 32;  
25 }
```

@EnableConfigurationProperties

当@EnableConfigurationProperties注解应用到你的@Configuration时，任何被@ConfigurationProperties注解的beans将自动被Environment属性配置。这种风格的配置特别适合与SpringApplication的外部YAML配置进行配合使用。

```
1 @Configuration  
2 @EnableConfigurationProperties({  
3     RocketMQProducerProperties.class,  
4     RocketMQConsumerProperties.class,  
5 })  
6 @AutoConfigureOrder  
7 public class RocketMQAutoConfiguration {  
8     @Value("${spring.application.name}")  
9     private String applicationName;  
10 }
```

@AutoConfigureAfter

用在自动配置类上面，表示该自动配置类需要在另外指定的自动配置类配置完之后。

如 Mybatis 的自动配置类，需要在数据源自动配置类之后。

```
1 @AutoConfigureAfter(DataSourceAutoConfiguration.class)  
2 public class MybatisAutoConfiguration {  
3 }
```

@AutoConfigureBefore

1 这个和@AutoConfigureAfter注解使用相反，表示该自动配置类需要在另外指定的自动配置类配置之前。

@AutoConfigureOrder

Spring Boot 1.3.0中有一个新的注解@AutoConfigureOrder，用于确定配置加载的优先级顺序。

```
1 @AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE) // 自动配置里面的最
高优先级
2 @Configuration
3 @ConditionalOnWebApplication // 仅限于web应用
4 @Import(BeanPostProcessorsRegistrar.class) // 导入内置容器的设置
5 public class EmbeddedServletContainerAutoConfiguration {
6     @Configuration
7     @ConditionalOnClass({ Servlet.class, Tomcat.class })
8     @ConditionalOnMissingBean(value = EmbeddedServletContainerF
actory.class, search = SearchStrategy.CURRENT)
9     public static class EmbeddedTomcat {
10         // ...
11     }
12
13     @Configuration
14     @ConditionalOnClass({ Servlet.class, Server.class, Loader.c
lass, WebApplicationContext.class })
15     @ConditionalOnMissingBean(value = EmbeddedServletContainerF
actory.class, search = SearchStrategy.CURRENT)
16     public static class EmbeddedJetty {
17         // ...
18     }
19 }
```

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

面试必问——Spring Boot 是如何实现自动配置的？

Spring Boot是Spring旗下众多的子项目之一，其理念是约定优于配置，它通过实现了自动配置（大多数用户平时习惯设置的配置作为默认配置）的功能来为用户快速构建出标准化的应用。Spring Boot的特点可以概述为如下几点：

- 内置了嵌入式的Tomcat、Jetty等Servlet容器，应用可以不用打包成War格式，而是可以直接以Jar格式运行。
- 提供了多个可选择的”starter”以简化Maven的依赖管理（也支持Gradle），让您可以按需加载需要的功能模块。
- 尽可能地进行自动配置，减少了用户需要动手写的各种冗余配置项，Spring Boot提倡无XML配置文件的理念，使用Spring Boot生成的应用完全不会生成任何配置代码与XML配置文件。
- 提供了一整套的对应用状态的监控与管理的功能模块（通过引入spring-boot-starter-actuator），包括应用的线程信息、内存信息、应用是否处于健康状态等，为了满足更多的资源监控需求，Spring Cloud中的很多模块还对其进行扩展。

有关Spring Boot的使用方法就不做多介绍了，如有兴趣请自行阅读官方文档Spring Boot或其他文章。

如今微服务的概念愈来愈热，转型或尝试微服务的团队也在如日渐增，而对于技术选型，Spring Cloud是一个比较好的选择，它提供了一站式的分布式系统解决方案，包含了许多构建分布式系统与微服务需要用到的组件，例如服务治理、API网关、配置中心、消息总线以及容错管理等模块。可以说，Spring Cloud”全家桶”极其适合刚刚接触微服务的团队。似乎有点跑题了，不过说了这么多，我想要强调的是，Spring Cloud中的每个组件都是基于Spring Boot构建的，而理解了Spring Boot的自动配置的原理，显然也是有好处的。

Spring Boot的自动配置看起来神奇，其实原理非常简单，背后全依赖于@Conditional注解来实现的。

什么是@Conditional？

@Conditional是由Spring 4提供的一个新特性，用于根据特定条件来控制Bean的创建行为。而在我们开发基于Spring的应用的时候，难免会需要根据条件来注册Bean。

例如，你想要根据不同的运行环境，来让Spring注册对应环境的数据源Bean，对于这种简单的情况，完全可以使用@Profile注解实现，就像下面代码所示：

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
1 @Configuration
2 public class AppConfig {
3     @Bean
4     @Profile("DEV")
5     public DataSource devDataSource() {
6         ...
7     }
8
9     @Bean
10    @Profile("PROD")
11    public DataSource prodDataSource() {
12        ...
13    }
14 }
```

剩下只需要设置对应的Profile属性即可，设置方法有如下三种：

- 通过context.getEnvironment().setActiveProfiles("PROD")来设置Profile属性。
- 通过设定jvm的spring.profiles.active参数来设置环境（Spring Boot中可以直接在application.properties配置文件中设置该属性）。
- 通过在DispatcherServlet的初始参数中设置。

```
1 <servlet>
2     <servlet-name>dispatcher</servlet-name>
3     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
4     <init-param>
5         <param-name>spring.profiles.active</param-name>
6         <param-value>PROD</param-value>
7     </init-param>
8 </servlet>
```

但这种方法只局限于简单的情况，而且通过源码我们可以发现@Profile自身也使用了@Conditional注解。

```
1 package org.springframework.context.annotation;
2
3 @Target({ElementType.TYPE, ElementType.METHOD})
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
4 @Retention(RetentionPolicy.RUNTIME)
5 @Documented
6 @Conditional({ProfileCondition.class}) // 组合了Conditional注解
7 public @interface Profile {
8     String[] value();
9 }
10
11 package org.springframework.context.annotation;
12
13 class ProfileCondition implements Condition {
14     ProfileCondition() {
15     }
16
17     // 通过提取出@Profile注解中的value值来与profiles配置信息进行匹配
18     public boolean matches(ConditionContext context, AnnotatedType
eMetadata metadata) {
19         if(context.getEnvironment() != null) {
20             MultiValueMap attrs = metadata.getAllAnnotationAttrib
utes(Profile.class.getName());
21             if(attrs != null) {
22                 Iterator var4 = ((List)attrs.get("value")).iterat
or();
23
24                 Object value;
25                 do {
26                     if(!var4.hasNext()) {
27                         return false;
28                     }
29
30                     value = var4.next();
31                 } while(!context.getEnvironment().acceptsProfiles
((String[])((String[])value)));
32
33                 return true;
34             }
35         }
36
37         return true;
38     }
39 }
```

在业务复杂的情况下，显然需要使用到@Conditional注解来提供更加灵活的条件判断，例如以下几个判断条件：

- 在类路径中是否存在这样的一个类。
- 在Spring容器中是否已经注册了某种类型的Bean（如未注册，我们可以让其自动注册到容器中，上一条同理）。
- 一个文件是否在特定的位置上。
- 一个特定的系统属性是否存在。
- 在Spring的配置文件中是否设置了某个特定的值。

举个栗子，假设我们有两个基于不同数据库实现的DAO，它们全都实现了UserDao，其中JdbcUserDAO与MySql进行连接，MongoUserDAO与MongoDB进行连接。现在，我们有了一个需求，需要根据命令行传入的系统参数来注册对应的UserDao，就像java -jar app.jar -DdbType=MySQL会注册JdbcUserDAO，而java -jar app.jar -DdbType=MongoDB则会注册MongoUserDAO。使用@Conditional可以很轻松地实现这个功能，仅仅需要在你自定义的条件类中去实现Condition接口，让我们来看下面的代码。（以下案例来自：<https://dzone.com/articles/how-springboot-autoconfiguration-magic-works>）

```
1 public interface UserDao {  
2     ....  
3 }  
4  
5 public class JdbcUserDAO implements UserDao {  
6     ....  
7 }  
8  
9 public class MongoUserDAO implements UserDao {  
10    ....  
11 }  
12  
13 public class MySQLDatabaseTypeCondition implements Condition {  
14     @Override  
15     public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata metadata) {  
16         String enabledDbType = System.getProperty("dbType"); //  
        获得系统参数 dbType  
17         // 如果该值等于MySQL，则条件成立
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
18         return (enabledDBType != null && enabledDBType.equalsIgn
oreCase("MySql"));
19     }
20 }
21
22 // 与上述逻辑一致
23 public class MongoDBDatabaseTypeCondition implements Condition {
24     @Override
25     public boolean matches(ConditionContext conditionContext, Ann
otatedTypeMetadata metadata) {
26         String enabledDBType = System.getProperty("dbType");
27         return (enabledDBType != null && enabledDBType.equalsIgn
oreCase("MongoDB"));
28     }
29 }
30
31 // 根据条件来注册不同的Bean
32 @Configuration
33 public class AppConfig {
34     @Bean
35     @Conditional(MySQLDatabaseTypeCondition.class)
36     public UserDAO jdbcUserDAO() {
37         return new JdbcUserDAO();
38     }
39
40     @Bean
41     @Conditional(MongoDatabaseTypeCondition.class)
42     public UserDAO mongoUserDAO() {
43         return new MongoUserDAO();
44     }
45 }
```

公众号：方志朋

现在，我们又有了一个新需求，我们想要根据当前工程的类路径中是否存在MongoDB的驱动类来确认是否注册MongoUserDAO。为了实现这个需求，可以创建检查MongoDB驱动是否存在的两个条件类。

```
1 public class MongoDriverPresentsCondition implements Condition {
2     @Override
3     public boolean matches(ConditionContext conditionContext, Ann
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
1     annotatedTypeMetadata metadata) {
2         try {
3             Class.forName("com.mongodb.Server");
4             return true;
5         } catch (ClassNotFoundException e) {
6             return false;
7         }
8     }
9 }
10 }
11 }
12
13 public class MongoDriverNotPresentsCondition implements Condition
14 {
15     @Override
16     public boolean matches(ConditionContext conditionContext, Anno
17     tatedTypeMetadata metadata) {
18         try {
19             Class.forName("com.mongodb.Server");
20             return false;
21         } catch (ClassNotFoundException e) {
22             return true;
23         }
24     }
25 }
```

公众号：方志朋

假如，你想要在UserDAO没有被注册的情况下注册一个UserDAOBean，那么我们可以定义一个条件类来检查某个类是否在容器中已被注册。

```
1 public class UserDAOBeanNotPresentsCondition implements Condition
2 {
3     @Override
4     public boolean matches(ConditionContext conditionContext, Anno
5     tatedTypeMetadata metadata) {
6         UserDAO userDAO = conditionContext.getBeanFactory().getBea
7         n(UserDAO.class);
8         return (userDAO == null);
9     }
10 }
```

如果你想根据配置文件中的某项属性来决定是否注册MongoDAO，例如app.dbType是否等于MongoDB，我们可以实现以下的条件类。

```
1 public class MongoDBTypePropertyCondition implements Condition {  
2     @Override  
3     public boolean matches(ConditionContext conditionContext, Anno  
tatedTypeMetadata metadata) {  
4         String dbType = conditionContext.getEnvironment().getPrope  
rty("app.dbType");  
5         return "MONGO".equalsIgnoreCase(dbType);  
6     }  
7 }
```

我们已经尝试并实现了各种类型的条件判断，接下来，我们可以选择一种更为优雅的方式，就像@Profile一样，以注解的方式来完成条件判断。首先，我们需要定义一个注解类。

```
1 @Target({ ElementType.TYPE, ElementType.METHOD })  
2 @Retention(RetentionPolicy.RUNTIME)  
3 @Documented  
4 @Conditional(DatabaseTypeCondition.class)  
5 public @interface DatabaseType {  
6     String value();  
7 }
```

具体的条件判断逻辑在DatabaseTypeCondition类中，它会根据系统参数dbType来判断注册哪一个Bean。

```
1 public class DatabaseTypeCondition implements Condition {  
2     @Override  
3     public boolean matches(ConditionContext conditionContext, Ann  
otatedTypeMetadata metadata) {  
4         Map<String, Object> attributes = metadata  
                .getAnnotationAttribu  
tes(DatabaseType.class.getName());  
5         String type = (String) attributes.get("value");  
6     }
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
7     // 默认值为MySql
8     String enabledDBType = System.getProperty("dbType", "MySq
l");
9     return (enabledDBType != null && type != null && enabledD
BType.equalsIgnoreCase(type));
10    }
11 }
```

最后，在配置类应用该注解即可。

```
1 @Configuration
2 @ComponentScan
3 public class AppConfig {
4     @Bean
5     @DatabaseType("MySql")
6     public UserDAO jdbcUserDAO() {
7         return new JdbcUserDAO();
8     }
9
10    @Bean
11    @DatabaseType("mongoDB")
12    public UserDAO mongoUserDAO() {
13        return new MongoUserDAO();
14    }
15 }
```

AutoConfigure源码分析

通过了解@Conditional注解的机制其实已经能够猜到自动配置是如何实现的了，接下来我们通过源码来看看它是怎么做的。本文中讲解的源码基于Spring Boot 1.5.9版本（最新的正式版本）。

使用过Spring Boot的童鞋应该都很清楚，它会替我们生成一个入口类，其命名规格为ArtifactNameApplication，通过这个入口类，我们可以发现一些信息。

```
1 @SpringBootApplication
2 public class DemoApplication {
```

```
3
4     public static void main(String[] args) {
5         SpringApplication.run(DemoApplication.class, args);
6     }
7
8 }
```

首先该类被@SpringBootApplication注解修饰，我们可以先从它开始分析，查看源码后可以发现它是一个包含许多注解的组合注解。

```
1 @Target({ElementType.TYPE})
2 @Retention(RetentionPolicy.RUNTIME)
3 @Documented
4 @Inherited
5 @SpringBootConfiguration
6 @EnableAutoConfiguration
7 @ComponentScan(
8     excludeFilters = {@Filter(
9         type = FilterType.CUSTOM,
10        classes = {TypeExcludeFilter.class}
11    ), @Filter(
12        type = FilterType.CUSTOM,
13        classes = {AutoConfigurationExcludeFilter.class}
14    )}
15 )
16 public @interface SpringBootApplication {
17     @AliasFor(
18         annotation = EnableAutoConfiguration.class,
19         attribute = "exclude"
20     )
21     Class<?>[] exclude() default {};
22
23     @AliasFor(
24         annotation = EnableAutoConfiguration.class,
25         attribute = "excludeName"
26     )
27     String[] excludeName() default {};
28 }
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
29     @AliasFor(  
30         annotation = ComponentScan.class,  
31         attribute = "basePackages"  
32     )  
33     String[] scanBasePackages() default {};  
34  
35     @AliasFor(  
36         annotation = ComponentScan.class,  
37         attribute = "basePackageClasses"  
38     )  
39     Class<?>[] scanBasePackageClasses() default {};  
40 }
```

该注解相当于同时声明了@Configuration、@EnableAutoConfiguration与@ComponentScan三个注解（如果我们想定制自定义的自动配置实现，声明这三个注解就够了），而@EnableAutoConfiguration是我们的关注点，从它的名字可以看出来，它是用来开启自动配置的，源码如下：

```
1  
2 @Target({ElementType.TYPE})  
3 @Retention(RetentionPolicy.RUNTIME)  
4 @Documented  
5 @Inherited  
6 @AutoConfigurationPackage  
7 @Import({EnableAutoConfigurationImportSelector.class})  
8 public @interface EnableAutoConfiguration {  
9     String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautocon  
figuration";  
10  
11     Class<?>[] exclude() default {};  
12  
13     String[] excludeName() default {};  
14 }
```

我们发现@Import (Spring 提供的一个注解，可以导入配置类或者Bean到当前类中) 导入了 EnableAutoConfigurationImportSelector类，根据名字来看，它应该就是我们要找到的目标了。不过查看它的源码发现它已经被Deprecated了，而官方API中告知我们去查看它的父类 AutoConfigurationImportSelector。

```
1  /** @deprecated */
2  @Deprecated
3  public class EnableAutoConfigurationImportSelector extends AutoCo
nfigurationImportSelector {
4      public EnableAutoConfigurationImportSelector() {
5      }
6
7      protected boolean isEnabled(AnnotationMetadata metadata) {
8          return this.getClass().equals(EnableAutoConfigurationImpo
rtSelector.class)?((Boolean)this.getEnvironment().getProperty("sp
ring.boot.enableautoconfiguration", Boolean.class, Boolean.valueO
f(true))).booleanValue():true;
9      }
10 }
```

由于AutoConfigurationImportSelector的源码太长了，这里我只截出关键的地方，显然方法selectImports是选择自动配置的主入口，它调用了其他的几个方法来加载元数据等信息，最后返回一个包含许多自动配置类信息的字符串数组。

```
1 public String[] selectImports(AnnotationMetadata annotationMetada
ta) {
2     if(!this.isEnabled(annotationMetadata)) {
3         return NO_IMPORTS;
4     } else {
5         try {
6             AutoConfigurationMetadata ex = AutoConfigurationMetad
ataLoader.loadMetadata(this.beanClassLoader);
7             AnnotationAttributes attributes = this.getAttributes
(annotationMetadata);
8             List configurations = this.getCandidateConfigurations
(annotationMetadata, attributes);
9             configurations = this.removeDuplicates(configuration
s);
10            configurations = this.sort(configurations, ex);
11            Set exclusions = this.getExclusions(annotationMetadat
a, attributes);
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
12         this.checkExcludedClasses(configurations, exclusion
13             s);
14         configurations.removeAll(exclusions);
15         configurations = this.filter(configurations, ex);
16         this.fireAutoConfigurationImportEvents(configuration
17             s, exclusions);
18         return (String[])configurations.toArray(new String[co
19             nfigurations.size()]);
20     } catch (IOException var6) {
21         throw new IllegalStateException(var6);
22     }
23 }
```

重点在于方法getCandidateConfigurations()返回了自动配置类的信息列表，而它通过调用SpringFactoriesLoader.loadFactoryNames()来扫描加载含有META-INF/spring.factories文件的jar包，该文件记录了具有哪些自动配置类。（建议还是用IDE去看源码吧，这些源码单行实在太长了，估计文章中的观看效果很差）

```
1 protected List<String> getCandidateConfigurations(AnnotationMetad
2 ata metadata, AnnotationAttributes attributes) {
3     List configurations = SpringFactoriesLoader
4                     .loadFactoryNames(this.ge
5                     tSpringFactoriesLoaderFactoryClass(), this.getBeanClassLoader());
6     Assert.notEmpty(configurations, "No auto configuration classe
7
8     found in META-INF spring.factories.
9     If you are using a custom packaging, make sure that file is c
10    orrect.");
11    return configurations;
12 }
13
14 public static List<String> loadFactoryNames(Class<?> factoryClas
15 s, ClassLoader classLoader) {
16     String factoryClassName = factoryClass.getName();
17
18     try {
19         Enumeration ex = classLoader != null?classLoader.getResou
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
rces("META-INF/spring.factories"):ClassLoader.getSystemResources
("META-INF/spring.factories");
15     ArrayList result = new ArrayList();
16
17     while(ex.hasMoreElements()) {
18         URL url = (URL)ex.nextElement();
19         Properties properties = PropertiesLoaderUtils.loadPro
perties(new UrlResource(url));
20         String factoryClassNames = properties.getProperty(fac
toryClassName);
21         result.addAll(Arrays.asList(StringUtils.commaDelimite
dListToStringArray(factoryClassNames)));
22     }
23
24     return result;
25 } catch (IOException var8) {
26     throw new IllegalArgumentException("Unable to load [" + f
actoryClass.getName() + "] factories from location [" + "META-IN
F/spring.factories" + "]", var8);
27 }
28 }
```

公众号：方志朋

自动配置类中的条件注解

接下来，我们在spring.factories文件中随便找一个自动配置类，来看看是怎样实现的。我查看了MongoDataAutoConfiguration的源码，发现它声明了@ConditionalOnClass注解，通过看该注解的源码后可以发现，这是一个组合了@Conditional的组合注解，它的条件类是OnClassCondition。

```
1 @Configuration
2 @ConditionalOnClass({Mongo.class, MongoTemplate.class})
3 @EnableConfigurationProperties({MongoProperties.class})
4 @AutoConfigureAfter({MongoAutoConfiguration.class})
5 public class MongoDataAutoConfiguration {
6     ....
7 }
8
9 @Target({ ElementType.TYPE, ElementType.METHOD })
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
10 @Retention(RetentionPolicy.RUNTIME)
11 @Documented
12 @Conditional({OnClassCondition.class})
13 public @interface ConditionalOnClass {
14     Class<?>[] value() default {};
15
16     String[] name() default {};
17 }
```

然后，我们开始看OnClassCondition的源码，发现它并没有直接实现Condition接口，只好往上找，发现它的父类SpringBootCondition实现了Condition接口。

```
1 class OnClassCondition extends SpringBootCondition implements Aut
oConfigurationImportFilter, BeanFactoryAware, BeanClassLoaderAware {
2     .....
3 }
4
5 public abstract class SpringBootCondition implements Condition {
6     private final Log logger = LoggerFactory.getLog(this.getClass()
());
7
8     public SpringBootCondition() {
9     }
10
11     public final boolean matches(ConditionContext context, Annota
tedTypeMetadata metadata) {
12         String classOrMethodName = getClassOrMethodName(metadata);
13
14         try {
15             ConditionOutcome ex = this.getMatchOutcome(context, m
etadata);
16             this.logOutcome(classOrMethodName, ex);
17             this.recordEvaluation(context, classOrMethodName, e
x);
18             return ex.isMatch();
19         } catch (NoClassDefFoundError var5) {
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
20         throw new IllegalStateException("Could not evaluate c
ondition on " + classOrMethodName + " due to " + var5.getMessage()
() + " not found. Make sure your own configuration does not rely
on that class. This can also happen if you are @ComponentScannin
g a springframework package (e.g. if you put a @ComponentScan in
the default package by mistake)", var5);
21     } catch (RuntimeException var6) {
22         throw new IllegalStateException("Error processing con
dition on " + this.getName(metadata), var6);
23     }
24 }
25
26     public abstract ConditionOutcome getMatchOutcome(ConditionCon
text var1, AnnotatedTypeMetadata var2);
27 }
```

SpringBootCondition实现的matches方法依赖于一个抽象方法this.getMatchOutcome(context, metadata)，我们在它的子类OnClassCondition中可以找到这个方法的具体实现。

```
1 public ConditionOutcome getMatchOutcome(ConditionContext context,
AnnotatedTypeMetadata metadata) {
2     ClassLoader classLoader = context.getClassLoader();
3     ConditionMessage matchMessage = ConditionMessage.empty();
4     // 找出所有ConditionalOnClass注解的属性
5     List onClasses = this.getCandidates(metadata, ConditionalOnCl
ass.class);
6     List onMissingClasses;
7     if(onClasses != null) {
8         // 找出不在类路径中的类
9         onMissingClasses = this.getMatches(onClasses, OnClassCond
ition.MatchType.MISSING, classLoader);
10        // 如果存在不在类路径中的类，匹配失败
11        if(!onMissingClasses.isEmpty()) {
12            return ConditionOutcome.noMatch(ConditionMessage.forC
ondition(ConditionalOnClass.class, new Object[0]).didNotFind("req
uired class", "required classes").items(Style.QUOTE, onMissingCla
sses));
13        }
}
```

公众号：方志朋

```
14
15     matchMessage = matchMessage.andCondition(ConditionalOnCla
ss.class, new Object[0]).found("required class", "required classe
s").items(Style.QUOTE, this.getMatches(onClasses, OnClassConditi
on.MatchType.PRESENT, classLoader));
16 }
17
18 // 接着找出所有ConditionalOnMissingClass注解的属性
19 // 它与ConditionalOnClass注解的含义正好相反，所以以下逻辑也与上面相反
20 onMissingClasses = this.getCandidates(metadata, ConditionalOn
MissingClass.class);
21 if(onMissingClasses != null) {
22     List present = this.getMatches(onMissingClasses, OnClassC
ondition.MatchType.PRESENT, classLoader);
23     if(!present.isEmpty()) {
24         return ConditionOutcome.noMatch(ConditionMessage.forC
ondition(ConditionalOnMissingClass.class, new Object[0]).found("u
nwanted class", "unwanted classes").items(Style.QUOTE, present));
25     }
26
27     matchMessage = matchMessage.andCondition(ConditionalOnMis
singClass.class, new Object[0]).didNotFind("unwanted class", "unw
anted classes").items(Style.QUOTE, this.getMatches(onMissingClass
es, OnClassCondition.MatchType.MISSING, classLoader));
28 }
29
30 return ConditionOutcome.match(matchMessage);
31 }
32
33 // 获得所有annotationType注解的属性
34 private List<String> getCandidates(AnnotatedTypeMetadata metadat
a, Class<?> annotationType) {
35     MultiValueMap attributes = metadata.getAllAnnotationAttribute
s(annotationType.getName(), true);
36     ArrayList candidates = new ArrayList();
37     if(attributes == null) {
38         return Collections.emptyList();
39     } else {
40         this.addAll(candidates, (List)attributes.get("value"));
41         this.addAll(candidates, (List)attributes.get("name"));
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
42         return candidates;
43     }
44 }
45
46 private void addAll(List<String> list, List<Object> itemsToAdd) {
47     if(itemsToAdd != null) {
48         Iterator var3 = itemsToAdd.iterator();
49
50         while(var3.hasNext()) {
51             Object item = var3.next();
52             Collections.addAll(list, (String[])((String[])item));
53         }
54     }
55
56 }
57
58 // 根据matchType.matches方法来进行匹配
59 private List<String> getMatches(Collection<String> candidates, On
  ClassCondition.MatchType matchType, ClassLoader classLoader) {
60     ArrayList matches = new ArrayList(candidates.size());
61     Iterator var5 = candidates.iterator();
62
63     while(var5.hasNext()) {
64         String candidate = (String)var5.next();
65         if(matchType.matches(candidate, classLoader)) {
66             matches.add(candidate);
67         }
68     }
69
70     return matches;
71 }
```

公众号：方志朋

关于match的具体实现在MatchType中，它是一个枚举类，提供了PRESENT和MISSING两种实现，前者返回类路径中是否存在该类，后者相反。

```
1 private static enum MatchType {
2     PRESENT {
3         public boolean matches(String className, ClassLoader clas
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
    sLoader) {
4         return OnClassCondition.MatchType.isPresent(className,
5             classLoader);
6     },
7     MISSING {
8         public boolean matches(String className, ClassLoader clas-
9             sLoader) {
10            return !OnClassCondition.MatchType.isPresent(className,
11                classLoader);
12        }
13    };
14
15
16    // 跟我们之前看过的案例一样，都利用了类加载功能来进行判断
17    private static boolean isPresent(String className, ClassLoade-
18        r classLoader) {
19        if(classLoader == null) {
20            classLoader = ClassUtils.getDefaultClassLoader();
21        }
22        try {
23            forName(className, classLoader);
24            return true;
25        } catch (Throwable var3) {
26            return false;
27        }
28    }
29
30    private static Class<?> forName(String className, ClassLoader
31        classLoader) throws ClassNotFoundException {
32        return classLoader != null?classLoader.loadClass(className):
33            Class.forName(className);
34    }
35 }
```

公众号：方志朋

现在终于真相大白，`@ConditionalOnClass`的含义是指定的类必须存在于类路径下，`MongoDataAutoConfiguration`类中声明了类路径下必须含有`Mongo.class`, `MongoTemplate.class`这两个类，否则该自动配置类不会被加载。

在Spring Boot中到处都有类似的注解，像`@ConditionalOnBean`（容器中是否有指定的Bean），`@ConditionalOnWebApplication`（当前工程是否为一个Web工程）等等，它们都只是`@Conditional`注解的扩展。当你揭开神秘的面纱，去探索本质时，发现其实Spring Boot自动配置的原理就是如此简单，在了解这些知识后，你完全可以自己去实现自定义的自动配置类，然后编写出自定义的starter。

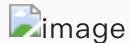
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



spring boot面试问题集锦

Q: 什么是spring boot?

A: 多年来，随着新功能的增加，spring变得越来越复杂。只需访问页面https://spring.io/projects，我们将看到所有在应用程序中使用的所有不同功能的spring项目。如果必须启动一个新的spring项目，我们必须添加构建路径或maven依赖项，配置application server，添加spring配置。因此，启动一个新的spring项目需要大量的工作，因为我们目前必须从头开始做所有事情。Spring Boot是这个问题的解决方案。

Spring boot构建在现有Spring框架之上。使用spring boot，我们可以避免以前必须执行的所有样板代码和配置。因此，Spring boot帮助我们更健壮地使用现有的Spring功能，并且只需最少的工作量。

Q: Spring Boot的优点是什么？

A: Spring Boot的优点是

减少开发、测试的时间和工作量。

使用JavaConfig有助于避免使用XML。

避免大量maven导入和各种版本冲突。

提供可选的开发方法。

通过提供默认开发方式进行快速开发。

不需要单独的Web服务器。这意味着您不再需要启动Tomcat、Glassfish或其他任何东西。

由于没有web.xml文件，所以需要更少的配置。只需添加带@ configuration注释的类，然后可以添加带@ bean注释的方法，Spring将自动加载对象并像往常一样管理它。您甚至可以将@Autowired添加到bean方法中，使Spring autowire成为bean所需的依赖项。

基于环境的配置——使用这些属性，您可以将其传递到您正在使用的应用程序环境中:- dspring .profile .active={enviorenment}。在加载主应用程序属性文件之后，Spring将在(application-{environment}.properties)处加载后续的应用程序属性文件。

Q: 您使用过哪些构建工具来开发Spring引导应用程序？

A: Spring Boot应用程序可以使用Maven和Gradle开发。

Q:什么是JavaConfig?

A:Spring JavaConfig是Spring社区的一个产品，它提供了一种纯java方法来配置Spring IoC容器。因此，它有助于避免使用XML配置。使用JavaConfig的优点是：

面向对象的配置。因为配置在JavaConfig中定义为类，所以用户可以充分利用Java中的面向对象特性。一个配置类可以子类化另一个配置类，覆盖它的@Bean方法，等等。

减少或消除XML配置。已经证明了基于依赖注入原则的外部化配置的好处。然而，许多开发人员不愿意在XML和Java之间来回切换。JavaConfig为开发人员提供了一种纯java方法来配置Spring容器，这种方法在概念上类似于XML配置。从技术上讲，仅使用 JavaConfig配置类来配置容器是可行的，但是在实践中，许多人发现将JavaConfig与XML混合并匹配是理想的。

类型安全的重构能力。JavaConfig提供了一种类型安全的配置Spring容器的方法。由于Java 5.0对泛型的支持，现在可以通过类型而不是名称检索bean，不需要进行任何基于类型转换或字符串的查找。

问:如何在不重启服务器的情况下在Spring引导时重新加载我的更改?

答:这可以通过开发工具来实现。有了这个依赖项，您保存的任何更改都将重新启动嵌入的tomcat。

Spring Boot有一个开发人员工具(DevTools)模块，它有助于提高开发人员的工作效率。Java开发人员面临的关键挑战之一是将文件更改自动部署到服务器并自动重启服务器。开发人员可以在Spring引导时重新加载更改，而不必重新启动服务器。这将消除每次手动部署更改的需要。Spring Boot在发布第一个版本时没有这个特性。这是开发人员最需要的特性。DevTools模块完全满足开发人员的需求。此模块将在生产环境中禁用。它还提供了H2-database控制台，以便更好地测试应用程序。使用以下依赖项

什么是Spring boot actuator?

答:Spring boot actuator是Spring boot framework的重要特性之一。Spring boot actuator帮助您访问生产环境中正在运行的应用程序的当前状态，在生产环境中必须检查和监视几个指标。甚至一些外部应用程序也可能使用这些服务来触发对相关人员的警报消息。actuator模块公开一组REST端点，这些端点可以作为HTTP URL直接访问，以检查状态。

问:如何将Spring Boot应用程序作为war包部署?

答:Spring Boot WAR部署

问:什么是Docker吗?如何将Spring引导应用程序部署到Docker?

A: Docker是什么

将基于Spring的WAR应用程序部署到Docker

将基于Spring的JAR应用程序部署到Docker

问:如何禁用执行器端点安全在Spring启动?

答:默认情况下，所有敏感的HTTP端点都是安全的，只有具有ACTUATOR角色的用户才能访问它们。安全
java架构师公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

性是使用标准HttpServletRequest.isUserInRole方法实现的。

我们可以使用–禁用安全性

```
management.security.enabled = false
```

建议仅当在防火墙后访问ACTUATOR端点时禁用安全性。

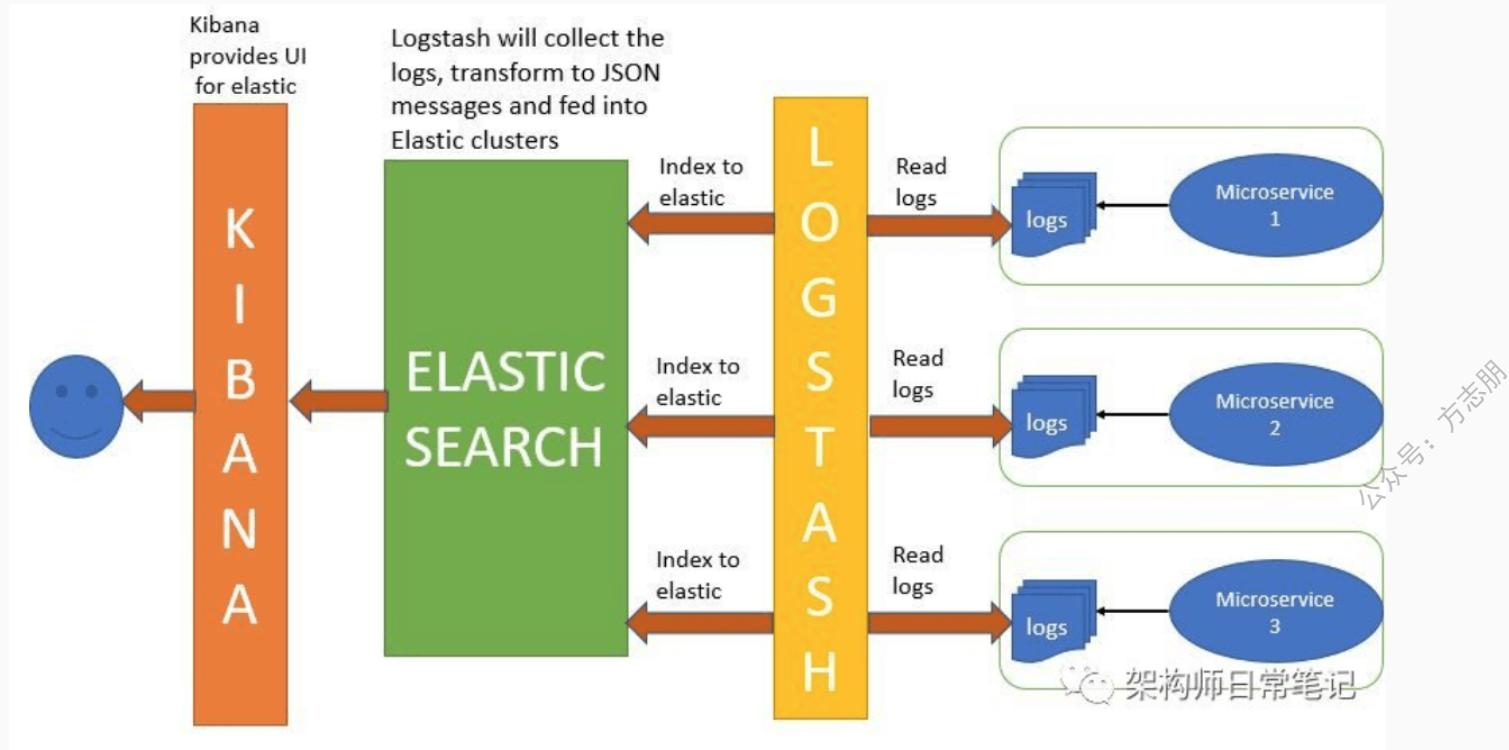
问:如何将Spring引导应用程序运行到自定义端口?

要在自定义端口上运行spring引导应用程序，可以在application.properties中指定端口。

```
server.port = 8090
```

什么是ELK堆栈?如何与Spring Boot一起使用?

答:ELK堆栈由三个开源产品组成——Elasticsearch、Logstash和Kibana from Elastic。



Elasticsearch是一个基于Lucene搜索引擎的NoSQL数据库。

Logstash是一个日志管道工具，它接受来自不同来源的输入，执行不同的转换，并将数据导出到不同的目标。它是一个动态的数据收集管道，具有可扩展的插件生态系统和强大的弹性搜索协同作用。

Kibana是一个可视化UI层，工作在Elasticsearch之上。

这三个项目一起用于各种环境中的日志分析。因此Logstash收集和解析日志、弹性搜索索引并存储这些信息，而Kibana提供了一个UI层，提供可操作的可见性。

Spring Boot + ELK stack

问:您有使用Spring Boot编写测试用例吗?

答:SpringBoot为编写单元测试用例提供了[@SpringBootTest](#)

Spring引导单元测试的简单示例

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

问:YAML是什么？

答:YAML是一种人类可读的数据序列化语言。它通常用于配置文件。

与属性文件相比， YAML文件的结构更加结构化，如果我们希望在配置文件中添加复杂的属性，那么它不会造成太大的混乱。可以看到， YAML具有分层的配置数据。

在Spring引导中使用YAML属性

问:如何为Spring引导应用程序实现安全性？

答:为了实现Spring Boot的安全性，我们使用Spring – Boot –starter–security依赖项，必须添加安全配置。它只需要很少的代码。Config类必须扩展WebSecurityConfigurerAdapter并覆盖它的方法。

Spring引导安全性示例和说明

问:您是否集成了Spring Boot和ActiveMQ ?

为了集成Spring Boot和ActiveMQ，我们使用Spring – Boot –starter– ActiveMQ依赖项，它只需要很少的配置，没有样板代码。

Spring引导ActiveMQ说明

问:您是否集成了Spring Boot和Apache Kafka ?

答:为了集成Spring Boot和Apache Kafka，我们使用Spring – Kafka依赖项。

Spring Boot + Apache Kafka示例

问:如何使用Spring引导实现分页和排序？

答:使用Spring Boot实现分页非常简单。使用Spring Data-JPA，这是通过传递可分页的org.springframework.data.domain来实现的。可分页到存储库方法。

Spring引导分页说明

什么是Swagger?您是否使用Spring Boot实现了它?

答:Swagger被广泛用于可视化api， Swagger UI为前端开发人员提供在线沙箱环境。在本教程中，我们将使用Swagger 2规范的Springfox实现。Swagger是一种工具、规范和完整的框架实现，用于生成RESTful Web服务的可视化表示。它允许文档以与服务器相同的速度更新。当通过Swagger正确定义时，使用者可以用最少的实现逻辑理解远程服务并与之交互。因此Swagger消除了调用服务时的猜测。

Spring Boot + Swagger2

问:什么是Spring Profiles?如何使用Spring Boot实现它?

答:Spring Profiles允许用户根据配置文件(dev, test, prod等)注册bean。因此，当应用程序在开发中运行时，只能加载某些bean，当应用程序在生产中运行时，只能加载某些其他bean。假设我们的需求是Swagger文档只对QA环境启用，对所有其他环境禁用。这可以使用配置文件来完成。Spring Boot使得使用配置文件非常容易。

Spring引导+配置文件

什么是Spring Boot Batch?如何使用Spring Boot实现它?

答:Spring Boot Batch提供了处理大量记录所必需的可重用功能，包括日志/跟踪、事务管理、作业处理统计信息、作业重启、作业跳过和资源管理。它还提供了更高级的技术服务和特性，通过优化和分区技术，这些特性将支持极高容量和高性能的批处理作业。无论是简单的还是复杂的，大容量批处理作业都可以以高度可伸缩的方式利用该框架来处理大量信息。

Spring Boot Batch

问:什么是FreeMarker模板?如何使用Spring Boot实现它?

答:FreeMarker是一个基于java的模板引擎，最初专注于使用MVC软件架构生成动态web页面。使用Freemarker的主要优势是完全分离了表示层和业务层。程序员可以处理应用程序代码，而设计人员可以处理html页面设计。最后，使用freemarker，这些可以组合在一起，给出最终的输出页面。

Spring Boot + FreeMarker的例子

问:如何使用Spring Boot实现异常处理?

答:Spring提供了一种非常有用的方法，可以使用ControllerAdvice处理异常。我们将实现一个ControllerAdvice类，它将处理控制器类抛出的所有异常。

Spring引导异常处理

什么是缓存?您在Spring引导中使用过缓存框架吗?

答:缓存是本地内存的一个区域，它保存了频繁访问的数据的副本，否则获取或计算这些数据将非常昂贵。使用Hazelcast进行缓存。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

Spring Boot + Hazelcast示例

问:您是否使用Spring Boot公开了SOAP web服务端点?

是的。使用Spring Boot公开了要使用的web服务。使用契约优先的方法从wsdl生成类。

Spring引导+ SOAP Web服务示例

问:您如何使用Spring Boot执行数据库操作?

答:Spring引导教程-Spring Data JPA

Spring引导JDBC示例

问:如何使用Spring上传文件?

A: Spring Boot +文件上传的例子

问:如何用Spring Boot实现拦截器?

答:使用Spring MVC HandlerInterceptor与Spring引导

问:如何在Spring Boot下使用schedulers ?

答:Spring引导任务调度程序示例

问:您使用过哪些启动器maven依赖项?

答:使用过不同的starter依赖项，如spring-boot-starter-activemq依赖项、spring-boot-starter-security依赖项、spring-boot-starter-web依赖项。

这有助于减少依赖项的数量，并减少版本组合。

Spring引导安全性示例和说明

什么是CSRF攻击?如何启用CSRF对其进行保护?

CSRF代表跨站请求伪造。它是一种攻击，迫使最终用户在其当前已经过身份验证的web应用程序上执行不需要的操作。CSRF攻击专门针对状态更改请求，而不是数据窃取，因为攻击者无法看到对伪造请求的响应。

Spring引导安全性—启用CSRF保护

问:如何使用Spring引导使用表单登录身份验证?

答:Spring引导表单安全登录Hello World示例

公众号: 方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

什么是OAuth2?如何使用Spring Boot实现它？

答:Spring Boot + OAuth2实现

问:GZIP是什么?如何使用Spring Boot实现它?

答: gzip是一种文件格式，是一种用于文件压缩和解压缩的软件应用程序。

Spring引导+ GZIP压缩

问:您在Spring引导中使用过集成框架吗?

答:已将Apache Camel与Spring引导集成。使用Apache Camel Spring启动启动依赖项。

Spring Boot +Apache Camel

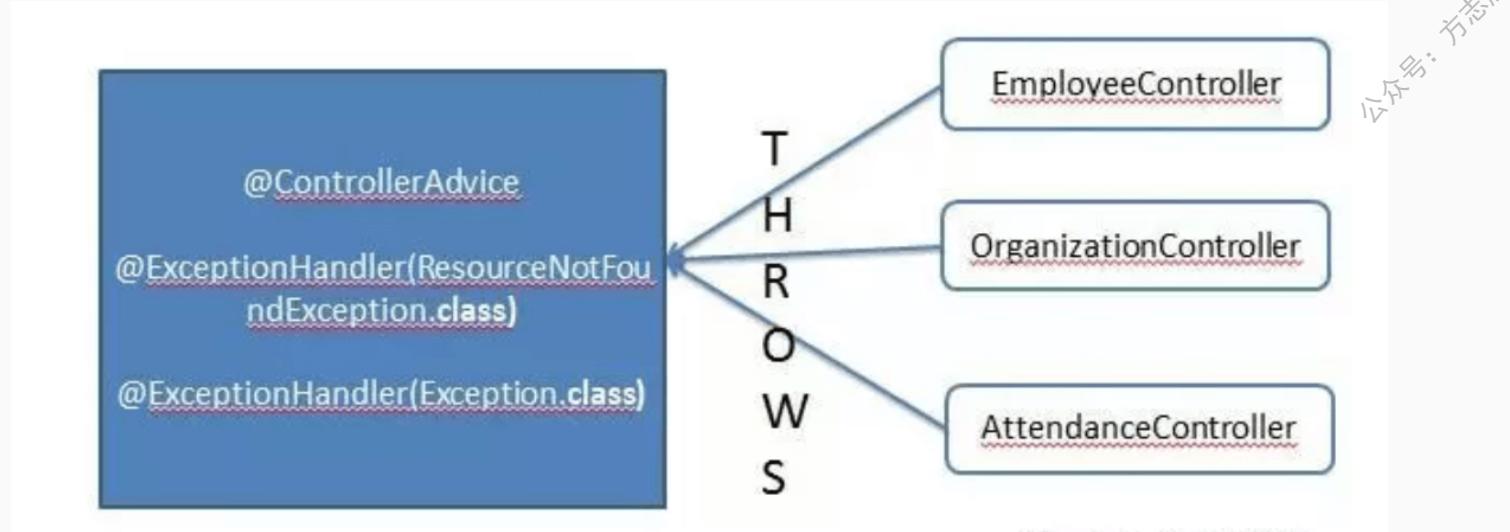
问:什么是Apache Freemarker?什么时候使用它而不是JSP?如何与Spring Boot集成?

答:JSP是为网页量身定做的，Freemarker模板是一种更通用的模板语言——它可以用来生成html、纯文本、电子邮件等。

Spring Boot + FreeMarker的例子

问:你什么时候使用WebSockets?如何使用Spring Boot实现它?

答:WebSocket是一种计算机通信协议，通过单个TCP连接提供全双工通信通道。



WebSocket是双向的——使用WebSocket客户端或服务器都可以发起发送消息。

WebSocket是全双工的——客户端和服务器之间的通信是相互独立的。

单个TCP连接——初始连接使用HTTP，然后将此连接升级为基于套接字的连接。然后，这个单一连接将用于未来的所有通信

轻— WebSocket消息数据交换比http轻得多。

Spring Boot + WebSockets的例子

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

什么是AOP?如何与Spring Boot一起使用？

答:在软件开发过程中，跨越应用程序多个点的功能称为横切关注点。这些横切关注点不同于应用程序的主要业务逻辑。因此，将这些横切关注点从业务逻辑中分离出来是面向方面编程(AOP)的切入点。

Spring Boot + AOP示例

问:什么是Apache Kafka?如何与Spring Boot集成?

答:apache Kafka是一个分布式发布-订阅消息传递系统。它是一个可伸缩的、容错的、发布-订阅消息传递系统，使我们能够构建分布式应用程序。这是一个Apache顶级项目。Kafka适用于离线和在线的消息消费。

Spring Boot + Apache Kafka示例

问:我们如何监视所有Spring Boot微服务?

答:Spring Boot提供了actuator 端点来监控单个微服务的指标。这些端点对于获取关于应用程序的信息非常有帮助，比如应用程序是否启动，它们的组件(如数据库等)是否正常工作。但是，使用actuator 接口的一个主要缺点或困难是，我们必须逐个命中这些接口，以了解应用程序的状态或健康状况。假设微服务涉及50个应用程序，管理员将不得不命中所有50个应用程序的actuator 端点。为了帮助我们处理这种情况，我们将使用位于`https://github.com/codecentric/spring-boot-admin`的开源项目。

它构建在Spring Boot Actuator之上，提供了一个web UI，使我们能够可视化多个应用程序的指标。

Spring Boot Admin

问:您在Spring引导中使用过Spring Cloud组件吗?

答:使用过Netflix Eureka等Spring Cloud组件进行服务注册，Ribbon用于负载平衡。

Spring Boot + Cloud Components

Spring Cloud interview Questions

问:如何将Spring Boot应用程序部署到Pivotal Cloud Foundry(PCF)?

Deploying Spring Boot Application to PCF

问:如何将Spring Boot + MySQL应用部署到Pivotal Cloud Foundry(PCF)?

A: Pivotal Cloud Foundry Tutorial – Deploying Spring Boot + MySQL Application to PCF

问:如何将Spring Boot + RabbitMQ应用部署到Pivotal Cloud Foundry(PCF)?

A: Pivotal Cloud Foundry Tutorial – Deploying Spring Boot + RabbitMQ Application to PCF

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：

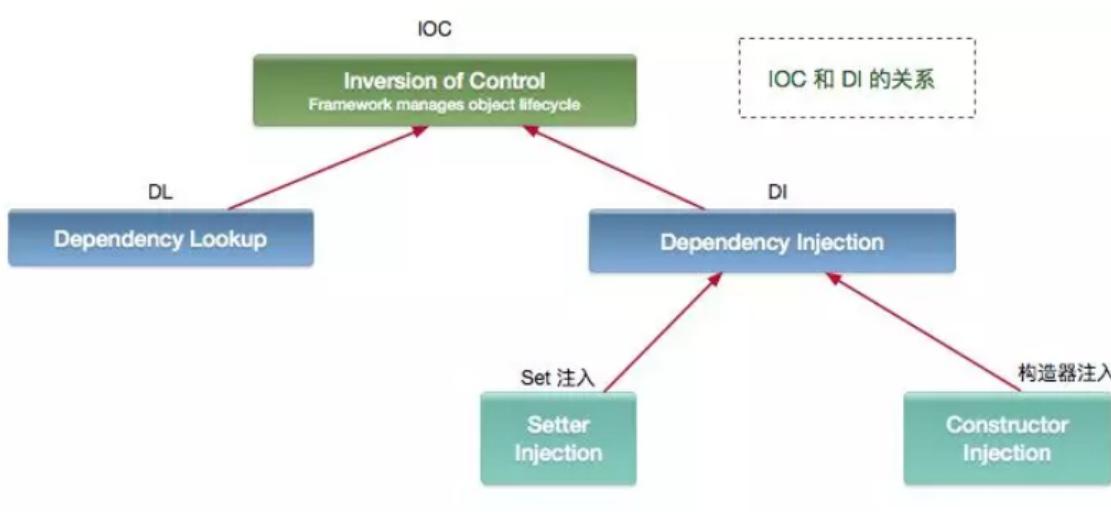


公众号：方志朋

SSM:面试被问烂的 Spring IOC

广义的 IOC

- IoC(Inversion of Control) 控制反转，即“不用打电话过来，我们会打给你”。
 - 两种实现：依赖查找（DL）和依赖注入（DI）。
 - IOC 和 DI 、 DL 的关系（这个 DL, Avalon 和 EJB 就是使用的这种方式实现的 IoC）：



- DL 已经被抛弃，因为他需要用户自己去使用 API 进行查找资源和组装对象。即有侵入性。
- DI 是 Spring 使用的方式，容器负责组件的装配。

注意：Java 使用 DI 方式实现 IoC 的不止 Spring，包括 Google 的 Guice，还有一个冷门的 PicoContainer（极度轻量，但只提供 IoC）。

Spring 的 IoC

Spring 的 IoC 设计支持以下功能：

- 依赖注入
- 依赖检查
- 自动装配
- 支持集合
- 指定初始化方法和销毁方法

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 支持回调某些方法（但是需要实现 Spring 接口，略有侵入）

其中，最重要的就是依赖注入，从 XML 的配置上说，即 ref 标签。对应 Spring RuntimeBeanReference 对象。

对于 IoC 来说，最重要的就是容器。容器管理着 Bean 的生命周期，控制着 Bean 的依赖注入。那么，Spring 如何设计容器的呢？

Spring 作者 Rod Johnson 设计了两个接口用以表示容器。

- BeanFactory
- ApplicationContext

BeanFactory 粗暴简单，可以理解为就是个 HashMap，Key 是 BeanName，Value 是 Bean 实例。通常只提供注册（put），获取（get）这两个功能。我们可以称之为“低级容器”。

ApplicationContext 可以称之为“高级容器”。因为他比 BeanFactory 多了更多的功能。他继承了多个接口。因此具备了更多的功能。

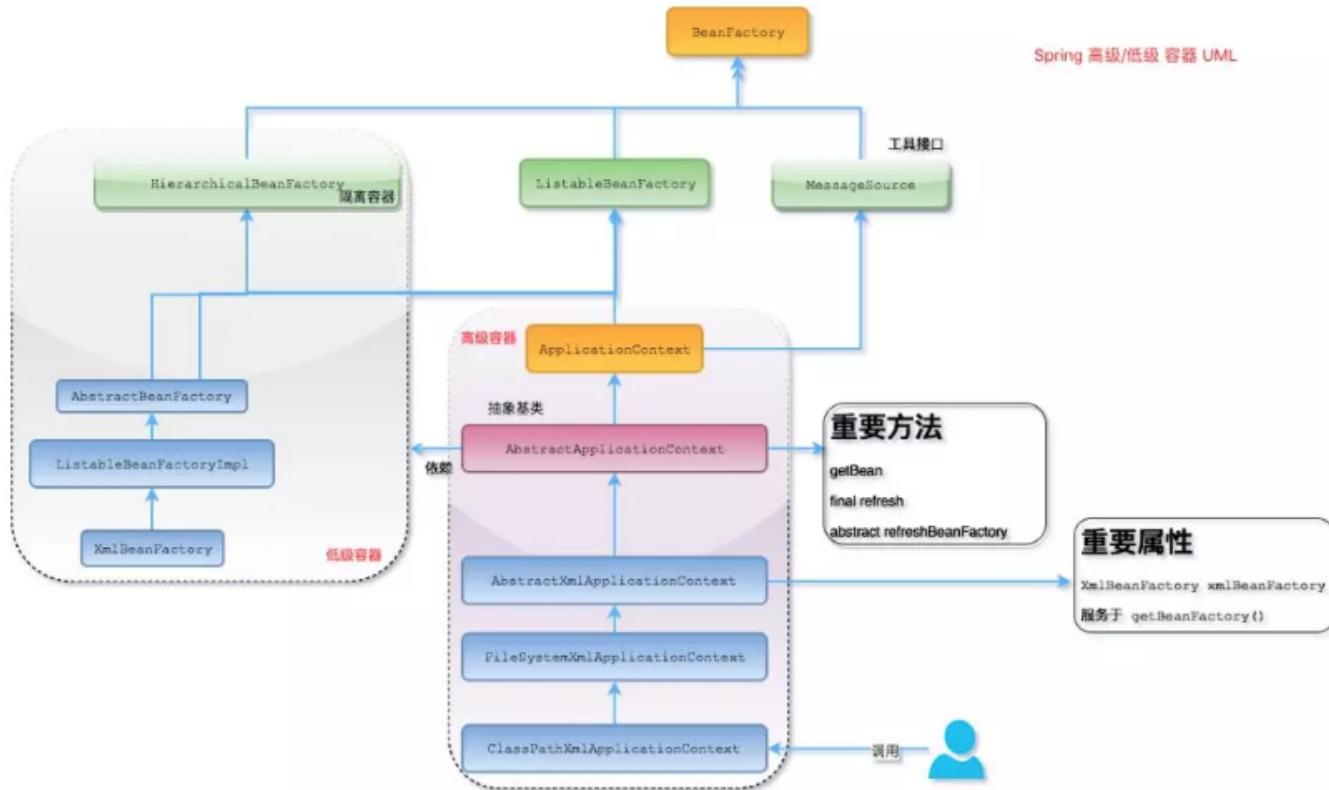
例如资源的获取，支持多种消息（例如 JSP tag 的支持），对 BeanFactory 多了工具级别的支持等待。所以你看他的名字，已经不是 BeanFactory 之类的工厂了，而是“应用上下文”，代表着整个大容器的所有功能。

该接口定义了一个 refresh 方法，此方法是所有阅读 Spring 源码的人的最熟悉的方法，用于刷新整个容器，即重新加载/刷新所有的 bean。

当然，除了这两个大接口，还有其他的辅助接口，但我今天不会花太多篇幅介绍他们。

为了更直观的展示“低级容器”和“高级容器”的关系，我这里通过常用的 ClassPathXmlApplicationContext 类，来展示整个容器的层级 UML 关系。

公众号：方志朋



有点复杂？先不要慌，我来解释一下。

最上面的 BeanFactory 知道吧？我就不讲了。

下面的 3 个绿色的，都是功能扩展接口，这里就不展开讲。

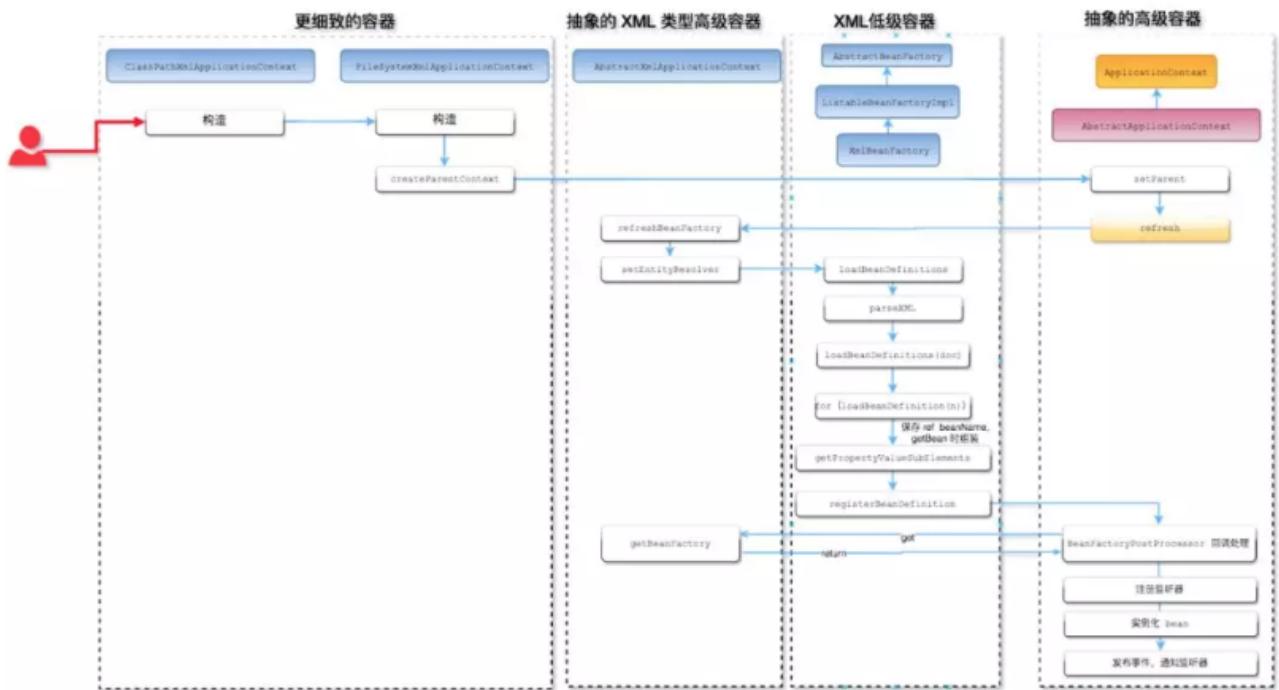
看下面的隶属 ApplicationContext 粉红色的“高级容器”，依赖着“低级容器”，这里说的是依赖，不是继承哦。他依赖着“低级容器”的 getBean 功能。而高级容器有更多的功能：支持不同的信息源头，可以访问文件资源，支持应用事件（Observer 模式）。

通常用户看到的就是“高级容器”。但 BeanFactory 也非常够用啦！

左边灰色区域的是“低级容器”，只负责加载 Bean，获取 Bean。容器其他的高级功能是没有的。例如上图画的 refresh 刷新 Bean 工厂所有配置。生命周期事件回调等。

好，解释了低级容器和高级容器，我们可以看看一个 IoC 启动过程是什么样子的。说白了，就是 ClassPathXmlApplicationContext 这个类，在启动时，都做了啥。（由于我这是 interface21 的代码，肯定和你的 Spring 4.x 系列不同）。

下图是 ClassPathXmlApplicationContext 的构造过程，实际就是 Spring IoC 的初始化过程。



注意，这里为了理解方便，有所简化。

这里再用文字来描述这个过程：

用户构造 ClassPathXmlApplicationContext（简称 CPAC）

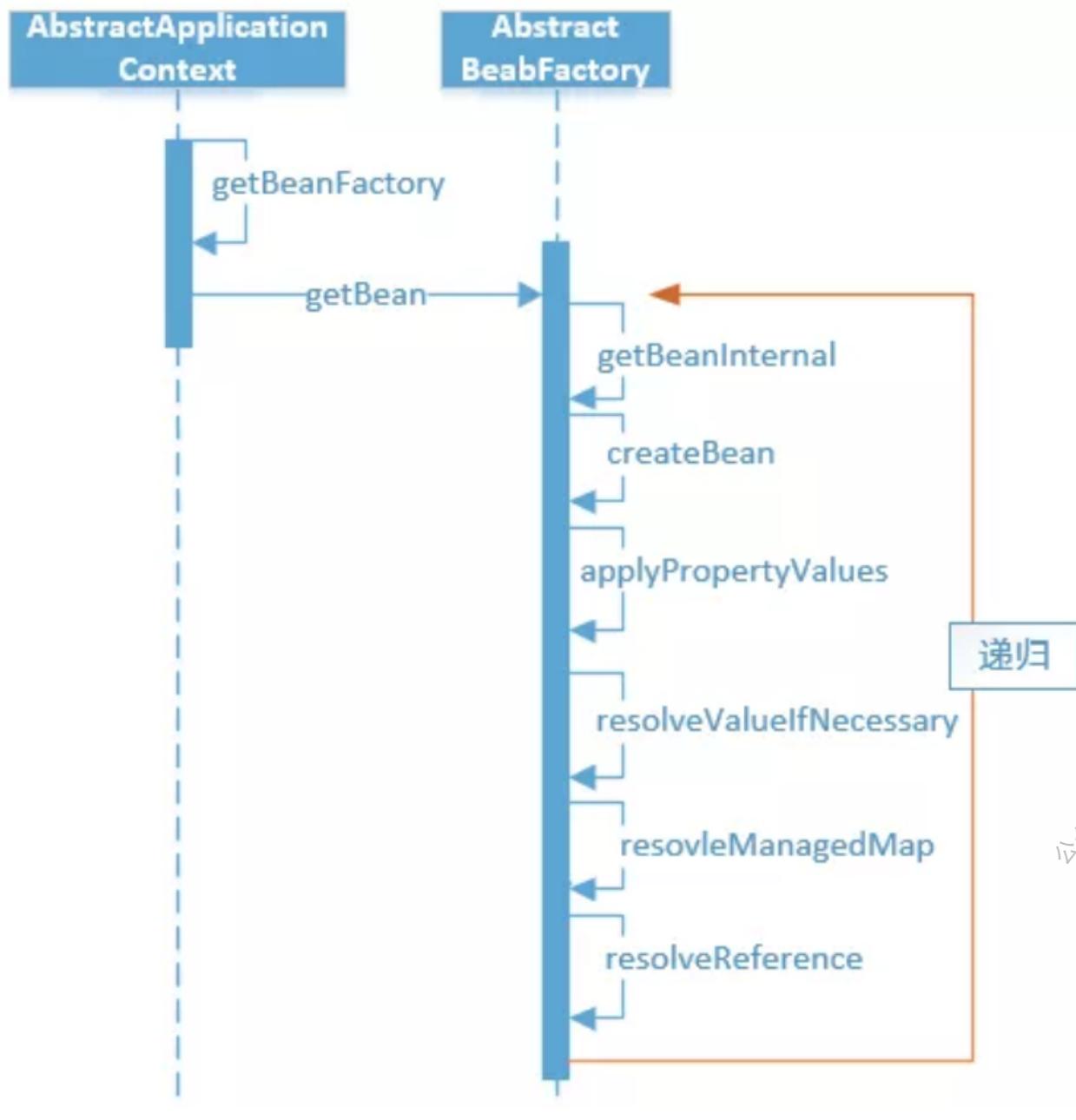
- CPAC 首先访问了“抽象高级容器”的 final 的 refresh 方法，这个方法是模板方法。所以要回调子类（低级容器）的
- refreshBeanFactory 方法，这个方法的作用是使用低级容器加载所有 BeanDefinition 和 Properties 到容器中。
- 低级容器加载成功后，高级容器开始处理一些回调，例如 Bean 后置处理器。回调 setBeanFactory 方法。或者注册监听器等，发布事件，实例化单例 Bean 等等功能，这些功能，随着 Spring 的不断升级，功能越来越多，很多人在这里迷失了方向：）。

简单说就是：

- 低级容器 加载配置文件（从 XML，数据库，Applet），并解析成 BeanDefinition 到低级容器中。
- 加载成功后，高级容器启动高级功能，例如接口回调，监听器，自动实例化单例，发布事件等等功能。

所以，一定要把“低级容器”和“高级容器”的区别弄清楚。不能一叶障目不见泰山。

好，当我们创建好容器，就会使用 getBean 方法，获取 Bean，而 getBean 的流程如下：



从图中可以看出，`getBean` 的操作都是在低级容器里操作的。其中有个递归操作，这个是什么意思呢？

假设：当 Bean_A 依赖着 Bean_B，而这个 Bean_A 在加载的时候，其配置的 `ref = “Bean_B”` 在解析的时候只是一个占位符，被放入了 Bean_A 的属性集合中，当调用 `getBean` 时，需要真正 Bean_B 注入到 Bean_A 内部时，就需要从容器中获取这个 Bean_B，因此产生了递归。

为什么不是在加载的时候，就直接注入呢？因为加载的顺序不同，很可能 Bean_A 依赖的 Bean_B 还没有加载好，也就无法从容器中获取，你不能要求用户把 Bean 的加载顺序排列好，这是不人道的。

所以，Spring 将其分为了 2 个步骤：

- 加载所有的 Bean 配置成 BeanDefinition 到容器中，如果 Bean 有依赖关系，则使用占位符暂时代替。

-然后，在调用 getBean 的时候，进行真正的依赖注入，即如果碰到了属性是 ref 的（占位符），那么就从容器里获取这个 Bean，然后注入到实例中——称之为依赖注入。

可以看到，依赖注入实际上，只需要“低级容器”就可以实现。

这就是 IoC。

所以 ApplicationContext refresh 方法里面的操作不只是 IoC，是高级容器的所有功能（包括 IoC），IoC 的功能在低级容器里就可以实现。

总结

说了这么多，不知道你有没有理解 Spring IoC？这里小结一下：IoC 在 Spring 里，只需要低级容器就可以实现，2 个步骤：

- 加载配置文件，解析成 BeanDefinition 放在 Map 里。
- 调用 getBean 的时候，从 BeanDefinition 所属的 Map 里，拿出 Class 对象进行实例化，同时，如果有依赖关系，将递归调用 getBean 方法——完成依赖注入。

上面就是 Spring 低级容器（BeanFactory）的 IoC。

至于高级容器 ApplicationContext，他包含了低级容器的功能，当他执行 refresh 模板方法的时候，将刷新整个容器的 Bean。同时其作为高级容器，包含了太多的功能。一句话，他不仅仅是 IoC。他支持不同信息源头，支持 BeanFactory 工具类，支持层级容器，支持访问文件资源，支持事件发布通知，支持接口回调等等。

可以预见，随着 Spring 的不断发展，高级容器的功能会越来越多。

诚然，了解 IoC 的过程，实际上为了了解 Spring 初始化时，各个接口的回调时机。例如 InitializingBean，BeanFactoryAware，ApplicationListener 等等接口，这些接口的作用，笔者之前写过一篇文章进行介绍，有兴趣可以看一下，关键字：Spring 必知必会 扩展接口。

但是请注意，实现 Spring 接口代表着你这个应用就绑定死 Spring 了！代表 Spring 具有侵入性！要知道，Spring 发布时，无侵入性就是他最大的宣传点之一——即 IoC 容器可以随便更换，代码无需变动。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

而现如今，Spring 已然成为 J2EE 社区准官方解决方案，也没有了所谓的侵入性这个说法。因为他就是标准，和 Servlet 一样，你能不实现 Servlet 的接口吗？:-)

好了，下次如果再有面试官问 Spring IoC 初始化过程，就再也不会含糊其词、支支吾吾了！！！

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

SSM:Spring AOP是什么？你都拿它做什么？

转自：我叫刘半仙，

<https://my.oschina.net/liughDevelop/blog/1457097>

为什么会有面向切面编程（AOP）？我们知道Java是一个面向对象（OOP）的语言，但它有一些弊端，比如当我们需要为多个不具有继承关系的对象引入一个公共行为，例如日志、权限验证、事务等功能时，只能在每个对象里引用公共行为。这样做不便于维护，而且有大量重复代码。AOP的出现弥补了OOP的这点不足。

为了阐述清楚Spring AOP，我们从将以下方面进行讨论：

- 代理模式
- 静态代理原理及实践
- 动态代理原理及实践
- Spring AOP原理及实战

代理模式

代理模式：为其他对象提供一种代理以控制对这个对象的访问。这段话比较官方，但我更倾向于用自己的语言理解：比如A对象要做一件事情，在没有代理前，自己来做；在对A代理后，由A的代理类B来做。代理其实是在原实例前后加了一层处理，这也是AOP的初级轮廓。

静态代理原理及实践

静态代理模式：静态代理说白了，就是在程序运行前就已经存在代理类的字节码文件、代理类和原始类的关系在运行前就已经确定。废话不多说，我们看一下代码。为了方便阅读，博主把单独的 class 文件合并到接口中，读者可以直接复制代码运行：

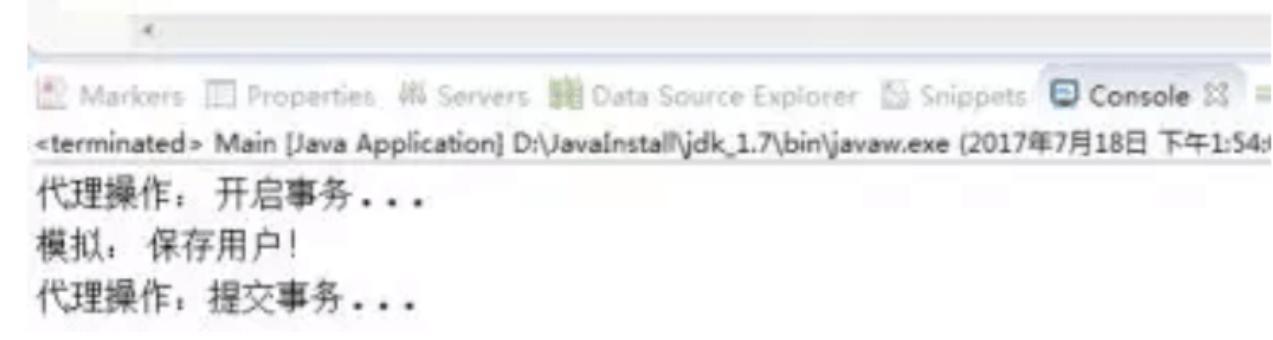
```
1 package test.staticProxy;  
2  
3 // 接口  
4 public interface IUserDao {  
5     void save();  
6     void find();  
7 }
```

```
8  
9 //目标对象  
10 class UserDao implements IUserDao{  
11     @Override  
12     public void save() {  
13         System.out.println("模拟：保存用户！");  
14     }  
15     @Override  
16     public void find() {  
17         System.out.println("模拟：查询用户");  
18     }  
19 }  
20  
21 /**  
22 * 静态代理  
23 * 特点：  
24 * 2. 目标对象必须要实现接口  
25 * 2. 代理对象，要实现与目标对象一样的接口  
26 */  
27 class UserDaoProxy implements IUserDao{  
28  
29     // 代理对象，需要维护一个目标对象  
30     private IUserDao target = new UserDao();  
31  
32     @Override  
33     public void save() {  
34         System.out.println("代理操作：开启事务...");  
35         target.save();    // 执行目标对象的方法  
36         System.out.println("代理操作：提交事务...");  
37     }  
38  
39     @Override  
40     public void find() {  
41         target.find();  
42     }  
43 }
```

公众号：方志朋

测试结果：

```
3 public class Main {  
4  
5     public static void main(String[] args) {  
6         // 代理对象  
7         IUserDao proxy = new UserDaoProxy();  
8         // 执行代理方法  
9         proxy.save();  
10    }  
11 }  
12 }  
13 }
```



静态代理虽然保证了业务类只需关注逻辑本身，代理对象的一个接口只服务于一种类型的对象。如果要代理的方法很多，势必要为每一种方法都进行代理。再者，如果增加一个方法，除了实现类需要实现这个方法外，所有的代理类也要实现此方法。增加了代码的维护成本。那么要如何解决呢？答案是使用动态代理。

动态代理原理及实践

动态代理模式：动态代理类的源码是在程序运行期间，通过JVM反射等机制动态生成。代理类和委托类的关系是运行时才确定的。实例如下：

```
1  
2 package test.dynamicProxy;  
3  
4 import java.lang.reflect.InvocationHandler;  
5 import java.lang.reflect.Method;  
6 import java.lang.reflect.Proxy;
```

```
7  
8 // 接口  
9 public interface IUserDao {  
10    void save();  
11    void find();  
12 }  
13  
14 //目标对象  
15 class UserDao implements IUserDao{  
16  
17     @Override  
18     public void save() {  
19         System.out.println("模拟：保存用户！");  
20     }  
21  
22     @Override  
23     public void find() {  
24         System.out.println("查询");  
25     }  
26 }  
27  
28 /**  
29 * 动态代理：  
30 * 代理工厂，给多个目标对象生成代理对象！  
31 *  
32 */  
33 class ProxyFactory {  
34  
35     // 接收一个目标对象  
36     private Object target;  
37  
38     public ProxyFactory(Object target) {  
39         this.target = target;  
40     }  
41  
42     // 返回对目标对象(target)代理后的对象(proxy)  
43     public Object getProxyInstance() {  
44         Object proxy = Proxy.newProxyInstance(  
45             target.getClass().getClassLoader(), // 目标对象使用的类  
        加载器
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
46         target.getClass().getInterfaces(), // 目标对象实现的所有接口
47         new InvocationHandler() {           // 执行代理对象方法
48            时候触发
49             @Override
50             public Object invoke(Object proxy, Method method,
51             Object[] args)
52                 throws Throwable {
53
54                 // 获取当前执行的方法的方法名
55                 String methodName = method.getName();
56
57                 // 方法返回值
58                 Object result = null;
59
59                 if ("find".equals(methodName)) {
60                     // 直接调用目标对象方法
61                     result = method.invoke(target, args);
62
63                 } else {
64                     System.out.println("开启事务...");
65
66                     // 执行目标对象方法
67                     result = method.invoke(target, args);
68                     System.out.println("提交事务...");
69
70                 }
71             }
72         );
73
74         return proxy;
75     }
76 }
```

公众号：方志朋

测试结果如下：

The screenshot shows the Eclipse IDE interface. On the left is the Java code for a main method that creates a UserDao target, gets a proxy instance from a ProxyFactory, and then calls proxy.save(). The code is numbered from 5 to 16. On the right is the Console view showing the execution output:

```
<terminated> Main (1) [Java Application] D:\JavaInstall\jdk_1.7\bin\javaw.exe (2017年7月19日 上午9:13:16)
目标对象: class test.dynamicProxy.UserDao
代理对象: class com.sun.proxy.$Proxy0
开启事务...
模拟: 保存用户!
提交事务...
```

```
1 IUserDao proxy = (IUserDao)new ProxyFactory(target).getProxyInstance();
```

其实是 JDK 动态生成了一个类去实现接口，隐藏了这个过程：

```
1 class $jdkProxy implements IUserDao{}
```

使用 JDK 生成的动态代理的前提是目标类必须有实现的接口。但这里又引入一个问题，如果某个类没有实现接口，就不能使用 JDK 动态代理。所以 CGLIB 代理就是解决这个问题的。

CGLIB 是以动态生成的子类继承目标的方式实现，在运行期动态的在内存中构建一个子类，如下：

CGLIB 使用的前提是目标类不能为 final 修饰。因为 final 修饰的类不能被继承。

现在，我们可以看看 AOP 的定义：面向切面编程，核心原理是使用动态代理模式在方法执行前后或出现异常时加入相关逻辑。

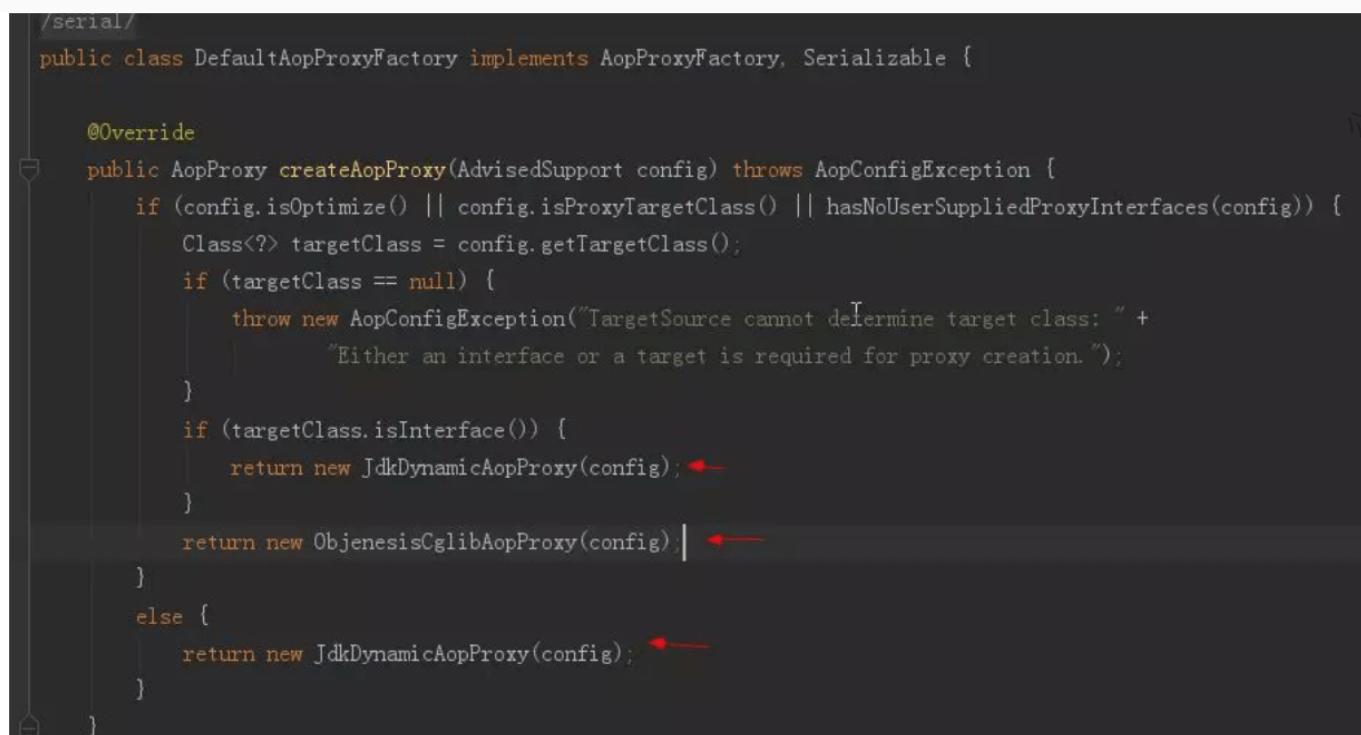
通过定义和前面代码我们可以发现3点：

- AOP 是基于动态代理模式。
- AOP 是方法级别的。
- AOP 可以分离业务代码和关注点代码（重复代码），在执行业务代码时，动态的注入关注点代码。切面就是关注点代码形成的类。

Spring AOP

前文提到 JDK 代理和 CGLIB 代理两种动态代理。优秀的 Spring 框架把两种方式在底层都集成了进去，我们无需担心自己去实现动态生成代理。那么，Spring是如何生成代理对象的？

- 创建容器对象的时候，根据切入点表达式拦截的类，生成代理对象。
- 如果目标对象有实现接口，使用 JDK 代理。如果目标对象没有实现接口，则使用 CGLIB 代理。然后从容器获取代理后的对象，在运行期植入“切面”类的方法。通过查看 Spring 源码，我们在 DefaultAopProxyFactory 类中，找到这样一段话。



The screenshot shows the Java code for the `DefaultAopProxyFactory` class. The code implements the `AopProxyFactory` and `Serializable` interfaces. It overrides the `createAopProxy` method to handle proxy creation based on the target class. If the target class is an interface or the target object itself, it returns a `JdkDynamicAopProxy`. Otherwise, it returns a `ObjenesisCglibAopProxy`. Red arrows point from the text description below to the corresponding return statements in the code.

```
/serial/
public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {

    @Override
    public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
        if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
            Class<?> targetClass = config.getTargetClass();
            if (targetClass == null) {
                throw new AopConfigException("TargetSource cannot determine target class: " +
                    "Either an interface or a target is required for proxy creation.");
            }
            if (targetClass.isInterface()) {
                return new JdkDynamicAopProxy(config); ←
            }
            return new ObjenesisCglibAopProxy(config); ←
        }
        else {
            return new JdkDynamicAopProxy(config); ←
        }
    }
}
```

简单的从字面意思看出：如果有接口，则使用 JDK 代理，反之使用 CGLIB，这刚好印证了前文所阐述的内容。Spring AOP 综合两种代理方式的使用前提有会如下结论：如果目标类没有实现接口，且 class 为 final 修饰的，则不能进行 Spring AOP 编程！

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

知道了原理，现在我们将自己手动实现 Spring 的 AOP：

```
1
2 package test.spring_aop_anno;
3
4 import org.aspectj.lang.ProceedingJoinPoint;
5
6 public interface IUserDao {
7     void save();
8 }
9
10 // 用于测试 CGLIB 动态代理
11 class OrderDao {
12     public void save() {
13         //int i =1/0; 用于测试异常通知
14         System.out.println("保存订单...");
```

```
15     }
16 }
17
18 // 用于测试 JDK 动态代理
19 class UserDao implements IUserDao {
20     public void save() {
21         //int i =1/0; 用于测试异常通知
22         System.out.println("保存用户...");
```

```
23     }
24 }
25
26 // 切面类
27 class TransactionAop {
28
29     public void beginTransaction() {
30         System.out.println("[前置通知] 开启事务..");
31     }
32
33     public void commit() {
34         System.out.println("[后置通知] 提交事务..");
35     }
36
37     public void afterReturing() {
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
38         System.out.println("[返回后通知]");  
39     }  
40  
41     public void afterThrowing() {  
42         System.out.println("[异常通知]");  
43     }  
44  
45     public void arroud(ProceedingJoinPoint pjp) throws Throwable  
{  
46         System.out.println("[环绕前:]");  
47         pjp.proceed(); // 执行目标方法  
48         System.out.println("[环绕后:]");  
49     }  
50 }
```

Spring 的 XML 配置文件：

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2 <beans xmlns="http://www.springframework.org/schema/beans"  
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
4   xmlns:context="http://www.springframework.org/schema/context"  
5   xmlns:aop="http://www.springframework.org/schema/aop"  
6   xsi:schemaLocation="  
7       http://www.springframework.org/schema/beans  
8       http://www.springframework.org/schema/beans/spring-beans.  
xsd  
9       http://www.springframework.org/schema/context  
10      http://www.springframework.org/schema/context/spring-cont  
ext.xsd  
11      http://www.springframework.org/schema/aop  
12      http://www.springframework.org/schema/aop/spring-aop.xs  
d">  
13      <!-- dao实例加入容器 -->  
14      <bean id="userDao" class="test.spring_aop_anno.UserDao"></bea  
n>  
15  
16      <!-- dao实例加入容器 -->  
17      <bean id="orderDao" class="test.spring_aop_anno.OrderDao"></b
```

公众号：方志朋

```
ean>
18
19      <!-- 实例化切面类 -->
20      <bean id="transactionAop" class="test.spring_aop_anno.Transac-
tionAop"></bean>
21
22      <!-- Aop相关配置 -->
23      <aop:config>
24          <!-- 切入点表达式定义 -->
25          <aop:pointcut expression="execution(* test.spring_aop_anno.*Dao.*(..))" id="transactionPointcut"/>
26          <!-- 切面配置 -->
27          <aop:aspect ref="transactionAop">
28              <!-- 【环绕通知】 -->
29              <aop:around method="arroud" pointcut-ref="transaction-
Pointcut"/>
30              <!-- 【前置通知】 在目标方法之前执行 -->
31              <aop:before method="beginTransaction" pointcut-ref="t-
ransactionPointcut" />
32              <!-- 【后置通知】 -->
33              <aop:after method="commit" pointcut-ref="transactionP-
ointcut"/>
34              <!-- 【返回后通知】 -->
35              <aop:after-returning method="afterReturing" pointcut-
ref="transactionPointcut"/>
36              <!-- 异常通知 -->
37              <aop:after-throwing method="afterThrowing" pointcut-r-
ef="transactionPointcut"/>
38          </aop:aspect>
39      </aop:config>
40  </beans>
```

公众号：方志朋

切入点表达式不在这里介绍。参考 Spring AOP 切入点表达式

代码的测试结果如下：

```
8
9     private ApplicationContext ac = new ClassPathXmlApplicationContext(
10         "applicationContext.xml", getClass());
11
12     @Test
13     public void testProxy() throws Exception {
14         // springIOC容器中获取对象,测试spring中Jdk中动态代理方式
15         IUserDao userDao = (IUserDao) ac.getBean("userDao");
16         System.out.println(userDao.getClass());
17         userDao.save();
18     }
19     @Test
20     public void testCglib() throws Exception {
21         // springIOC容器中获取对象,测试spring中Cglib动态代理方式
22         OrderDao orderDao = (OrderDao) ac.getBean("orderDao");
23         System.out.println(orderDao.getClass());
24         orderDao.save();
25     }
26 }
```

问题 控制台 声明 搜索 可达性 工具栏 属性 X 窗口

<已终止> Main.testProxy [JUnit] D:\Java\Install\jdk_1.7\bin\javaw.exe (2017年7月19日上午9:18:35)

```
class com.sun.proxy.$Proxy7
[环绕前: ]
[前置通知] 开启事务...
保存用户...
[环绕后: ]
[后置通知] 提交事务...
[返回后通知]
```

到这里，我们已经全部介绍完Spring AOP。回到开篇的问题，我们拿它做什么？

- Spring声明式事务管理配置：请参考博主的另一篇文章：分布式系统架构实战 demo：SSM+Dubbo
- Controller层的参数校验：参考 Spring AOP拦截Controller做参数校验
- 使用 Spring AOP 实现 MySQL 数据库读写分离案例分析
- 在执行方法前，判断是否具有权限
- 对部分函数的调用进行日志记录：监控部分重要函数，若抛出指定的异常，可以以短信或邮件方式通知相关人员。
- 信息过滤，页面转发等等功能

博主一个人的力量有限，只能列举这么多，欢迎评论区对文章做补充。

Spring AOP还能做什么，实现什么魔幻功能，就在于我们每一个平凡而又睿智的程序猿！

参考资料

Spring AOP 切入点表达式：<http://blog.csdn.net/keda8997110/article/details/50747923>

分布式系统架构实战 demo：SSM+Dubbo：<https://my.oschina.net/liughDevelop/blog/1480061>

Spring AOP 拦截Controller做参数校验：<https://my.oschina.net/liughDevelop/blog/1480061>

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

使用 Spring AOP 实现 MySQL 数据库读写分离案例分析：

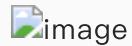
<http://blog.csdn.net/xlgen157387/article/details/53930382>

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



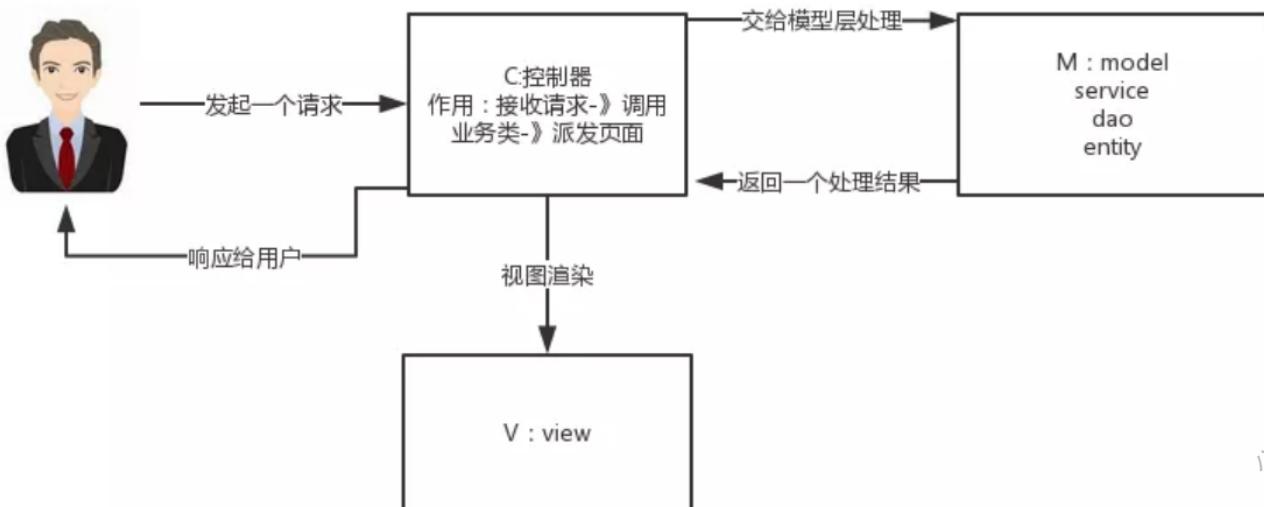
公众号：方志朋

SSM:SpringMVC工作原理详解

先来看一下什么是 MVC 模式

MVC 是一种设计模式。

MVC 的原理图如下：



SpringMVC 简单介绍

SpringMVC 框架是以请求为驱动，围绕 Servlet 设计，将请求发给控制器，然后通过模型对象，分派器来展示请求结果视图。其中核心类是 DispatcherServlet，它是一个 Servlet，顶层是实现的Servlet接口。

SpringMVC 使用

需要在 web.xml 中配置 DispatcherServlet 。并且需要配置 Spring 监听器ContextLoaderListener

```
1 <listener>
2   <listener-class>org.springframework.web.context.ContextLoader
  Listener
```

```
3     </listener-class>
4 </listener>
5 <servlet>
6     <servlet-name>springmvc</servlet-name>
7     <servlet-class>org.springframework.web.servlet.DispatcherServ
let
8     </servlet-class>
9     <!-- 如果不设置init-param标签，则必须在/WEB-INF/下创建xxx-servlet.x
ml文件，其中xxx是servlet-name中配置的名称。 -->
10    <init-param>
11        <param-name>contextConfigLocation</param-name>
12        <param-value>classpath:spring/springmvc-servlet.xml</para
m-value>
13    </init-param>
14    <load-on-startup>1</load-on-startup>
15 </servlet>
16 <servlet-mapping>
17     <servlet-name>springmvc</servlet-name>
18     <url-pattern>/</url-pattern>
19 </servlet-mapping>
```

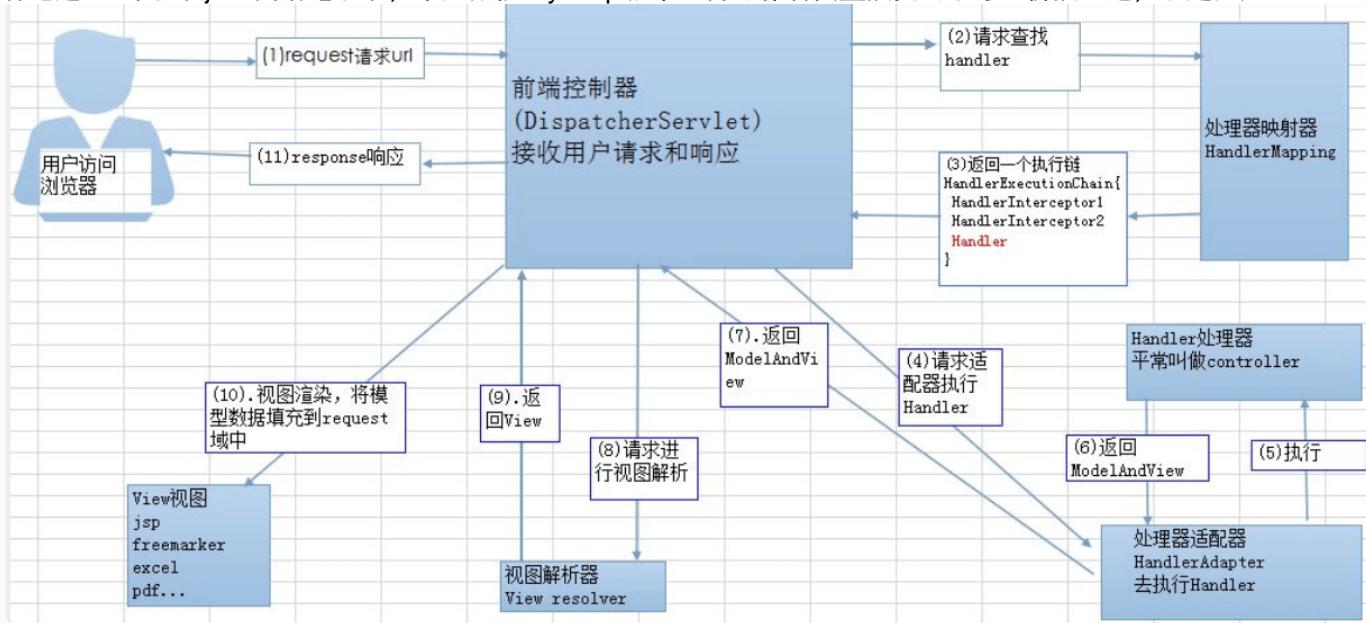
SpringMVC 工作原理（重要）

简单来说：

客户端发送请求-> 前端控制器 DispatcherServlet 接受客户端请求 -> 找到处理器映射 HandlerMapping 解析请求对应的 Handler-> HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求，并处理相应的业务逻辑 -> 处理器返回一个模型视图 ModelAndView -> 视图解析器进行解析 -> 返回一个视图对象->前端控制器 DispatcherServlet 渲染数据（Model）-> 将得到视图对象返回给用户

如下图所示：

公众号：方志朋



上图的一个笔误的小问题：Spring MVC 的入口函数也就是前端控制器 DispatcherServlet 的作用是接收请求，响应结果。

流程说明（重要）：

- (1) 客户端（浏览器）发送请求，直接请求到 DispatcherServlet。
- (2) DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。
- (3) 解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。
- (4) HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求，并处理相应的业务逻辑。
- (5) 处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 View。
- (6) ViewResolver 会根据逻辑 View 查找实际的 View。
- (7) DispatcherServlet 把返回的 Model 传给 View（视图渲染）。
- (8) 把 View 返回给请求者（浏览器）

SpringMVC 重要组件说明

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

1、前端控制器DispatcherServlet（不需要工程师开发），由框架提供（重要）

作用：Spring MVC 的入口函数。接收请求，响应结果，相当于转发器，中央处理器。有了 DispatcherServlet 减少了其它组件之间的耦合度。用户请求到达前端控制器，它就相当于mvc模式中的 c，DispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，DispatcherServlet的存在降低了组件之间的耦合性。

2、处理器映射器HandlerMapping(不需要工程师开发),由框架提供

作用：根据请求的url查找Handler。HandlerMapping负责根据用户请求找到Handler即处理器（Controller），SpringMVC提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

3、处理器适配器HandlerAdapter

作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler
通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

4、处理器Handler(需要工程师开发)

注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler
Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。

由于Handler涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发Handler。

5、视图解析器View resolver(不需要工程师开发),由框架提供

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）
View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。
springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。
一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

6、视图View(需要工程师开发)

View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...）

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

注意：处理器Handler（也就是我们平常说的Controller控制器）以及视图层view都是需要我们自己手动开发的。其他的一些组件比如：前端控制器DispatcherServlet、处理器映射器HandlerMapping、处理器适配器HandlerAdapter等等都是框架提供给我们的，不需要自己手动开发。

DispatcherServlet详细解析

首先看下源码：

```
1 package org.springframework.web.servlet;
2
3 @SuppressWarnings("serial")
4 public class DispatcherServlet extends FrameworkServlet {
5
6     public static final String MULTIPART_RESOLVER_BEAN_NAME = "multipartResolver";
7     public static final String LOCALE_RESOLVER_BEAN_NAME = "localeResolver";
8     public static final String THEME_RESOLVER_BEAN_NAME = "themeResolver";
9     public static final String HANDLER_MAPPING_BEAN_NAME = "handlerMapping";
10    public static final String HANDLER_ADAPTER_BEAN_NAME = "handlerAdapter";
11    public static final String HANDLER_EXCEPTION_RESOLVER_BEAN_NAME = "handlerExceptionResolver";
12    public static final String REQUEST_TO_VIEW_NAME_TRANSLATOR_BEAN_NAME = "viewNameTranslator";
13    public static final String VIEW_RESOLVER_BEAN_NAME = "viewResolver";
14    public static final String FLASH_MAP_MANAGER_BEAN_NAME = "flashMapManager";
15    public static final String WEB_APPLICATION_CONTEXT_ATTRIBUTE =
16        DispatcherServlet.class.getName() + ".CONTEXT";
16    public static final String LOCALE_RESOLVER_ATTRIBUTE =
17        DispatcherServlet.class.getName() + ".LOCALE_RESOLVER";
17    public static final String THEME_RESOLVER_ATTRIBUTE =
18        DispatcherServlet.class.getName() + ".THEME_RESOLVER";
18    public static final String THEME_SOURCE_ATTRIBUTE =
19        DispatcherServlet.class.getName() + ".THEME_SOURCE";
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
erServlet.class.getName() + ".THEME_SOURCE";
19     public static final String INPUT_FLASH_MAP_ATTRIBUTE = DispatcherServlet.class.getName() + ".INPUT_FLASH_MAP";
20     public static final String OUTPUT_FLASH_MAP_ATTRIBUTE = DispatcherServlet.class.getName() + ".OUTPUT_FLASH_MAP";
21     public static final String FLASH_MAP_MANAGER_ATTRIBUTE = DispatcherServlet.class.getName() + ".FLASH_MAP_MANAGER";
22     public static final String EXCEPTION_ATTRIBUTE = DispatcherServlet.class.getName() + ".EXCEPTION";
23     public static final String PAGE_NOT_FOUND_LOG_CATEGORY = "org.springframework.web.servlet.PageNotFound";
24     private static final String DEFAULT_STRATEGIES_PATH = "DispatcherServlet.properties";
25     protected static final Log pageNotFoundLogger = LoggerFactory.getLogger(PAGE_NOT_FOUND_LOG_CATEGORY);
26     private static final Properties defaultStrategies;
27     static {
28         try {
29             ClassPathResource resource = new ClassPathResource(DEFAULT_STRATEGIES_PATH, DispatcherServlet.class);
30             defaultStrategies = PropertiesLoaderUtils.loadProperties(resource);
31         }
32         catch (IOException ex) {
33             throw new IllegalStateException("Could not load 'DispatcherServlet.properties': " + ex.getMessage());
34         }
35     }
36
37     /** Detect all HandlerMappings or just expect "handlerMapping" bean? */
38     private boolean detectAllHandlerMappings = true;
39
40     /** Detect all HandlerAdapters or just expect "handlerAdapter" bean? */
41     private boolean detectAllHandlerAdapters = true;
42
43     /** Detect all HandlerExceptionResolvers or just expect "handlerExceptionResolver" bean? */
44     private boolean detectAllHandlerExceptionResolvers = true;
```

```
45  
46     /** Detect all ViewResolvers or just expect "viewResolver" b  
ean? */  
47     private boolean detectAllViewResolvers = true;  
48  
49     /** Throw a NoHandlerFoundException if no Handler was found  
to process this request? */  
50     private boolean throwExceptionIfNoHandlerFound = false;  
51  
52     /** Perform cleanup of request attributes after include requ  
est? */  
53     private boolean cleanupAfterInclude = true;  
54  
55     /** MultipartResolver used by this servlet */  
56     private MultipartResolver multipartResolver;  
57  
58     /** LocaleResolver used by this servlet */  
59     private LocaleResolver localeResolver;  
60  
61     /** ThemeResolver used by this servlet */  
62     private ThemeResolver themeResolver;  
63  
64     /** List of HandlerMappings used by this servlet */  
65     private List<HandlerMapping> handlerMappings;  
66  
67     /** List of HandlerAdapters used by this servlet */  
68     private List<HandlerAdapter> handlerAdapters;  
69  
70     /** List of HandlerExceptionResolvers used by this servlet  
*/  
71     private List<HandlerExceptionResolver> handlerExceptionResol  
vers;  
72  
73     /** RequestToViewNameTranslator used by this servlet */  
74     private RequestToViewNameTranslator viewNameTranslator;  
75  
76     private FlashMapManager flashMapManager;  
77  
78     /** List of ViewResolvers used by this servlet */  
79     private List<ViewResolver> viewResolvers;
```

```
80
81     public DispatcherServlet() {
82         super();
83     }
84
85     public DispatcherServlet(ApplicationContext webApplication
86         nContext) {
87         super(webApplicationContext);
88     }
89     @Override
90     protected void onRefresh(ApplicationContext context) {
91         initStrategies(context);
92     }
93
94     protected void initStrategies(ApplicationContext context) {
95         initMultipartResolver(context);
96         initLocaleResolver(context);
97         initThemeResolver(context);
98         initHandlerMappings(context);
99         initHandlerAdapters(context);
100        initHandlerExceptionResolvers(context);
101        initRequestToViewNameTranslator(context);
102        initViewResolvers(context);
103        initFlashMapManager(context);
104    }
```

DispatcherServlet类中的属性beans：

- HandlerMapping：用于handlers映射请求和一系列的对于拦截器的前处理和后处理，大部分用@Controller注解。
- HandlerAdapter：帮助DispatcherServlet处理映射请求处理程序的适配器，而不用考虑实际调用的是哪个处理程序。---
- ViewResolver：根据实际配置解析实际的View类型。
- ThemeResolver：解决Web应用程序可以使用的主题，例如提供个性化布局。
- MultipartResolver：解析多部分请求，以支持从HTML表单上传文件。
- FlashMapManager：存储并检索可用于将一个请求属性传递到另一个请求的input和output的FlashMap，通常用于重定向。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

在Web MVC框架中，每个DispatcherServlet都拥自己的WebApplicationContext，它继承了 ApplicationContext。WebApplicationContext包含了其上下文和Servlet实例之间共享的所有基础框架beans。

HandlerMapping

Type hierarchy of 'org.springframework.web.servlet.HandlerMapping':

- ✓ ⓘ HandlerMapping - org.springframework.web.servlet
- ✓ ⓘ AbstractHandlerMapping - org.springframework.web.servlet.handler
 - ⓘ AbstractHandlerMethodMapping<T> - org.springframework.web.servlet.handler
 - ✓ ⓘ AbstractUrlHandlerMapping - org.springframework.web.servlet.handler
 - ✓ ⓘ AbstractDetectingUrlHandlerMapping - org.springframework.web.servlet.handler
 - ✓ ⓘ AbstractControllerHandlerMapping - org.springframework.web.servlet.mvc
 - ⌚ ControllerBeanNameHandlerMapping - org.springframework.web.servlet.m
 - ⌚ ControllerClassNameHandlerMapping - org.springframework.web.servlet.m
 - ⌚ BeanNameUrlHandlerMapping - org.springframework.web.servlet.handler
 - ⌚ DefaultAnnotationHandlerMapping - org.springframework.web.servlet.an
 - ⌚ SimpleUrlHandlerMapping - org.springframework.web.servlet.handler
 - ⌚ EmptyHandlerMapping - org.springframework.web.servlet.config.annotation.WebM

HandlerMapping接口处理请求的映射HandlerMapping接口的实现类：

- SimpleUrlHandlerMapping类通过配置文件把URL映射到Controller类。
- DefaultAnnotationHandlerMapping类通过注解把URL映射到Controller类。

HandlerAdapter

- ✓ ⓘ HandlerAdapter - org.springframework.web.servlet
- ✓ ⓘ AbstractHandlerMethodAdapter - org.springframework.web.servlet.mvc.method
 - ⌚ RequestMappingHandlerAdapter - org.springframework.web.servlet.mvc.method.ar
 - ⌚ AnnotationMethodHandlerAdapter - org.springframework.web.servlet.mvc.annotation
 - ⌚ HttpRequestHandlerAdapter - org.springframework.web.servlet.mvc
 - ⌚ SimpleControllerHandlerAdapter - org.springframework.web.servlet.mvc
 - ⌚ SimpleServletHandlerAdapter - org.springframework.web.servlet.handler

- HandlerAdapter接口–处理请求映射
- AnnotationMethodHandlerAdapter：通过注解，把请求URL映射到Controller类的方法上。

HandlerExceptionResolver

Type hierarchy of 'org.springframework.web.servlet.HandlerExceptionResolver':

- HandlerExceptionResolver - org.springframework.web.servlet
- AbstractHandlerExceptionResolver - org.springframework.web.servlet.handler
- AbstractHandlerMethodExceptionResolver - org.springframework.web.servlet.handler
 - ExceptionHandlerExceptionResolver - org.springframework.web.servlet.mvc.method
 - AnnotationMethodHandlerExceptionResolver - org.springframework.web.servlet.mvc.method
 - DefaultHandlerExceptionResolver - org.springframework.web.servlet.mvc.support
 - ResponseStatusExceptionResolver - org.springframework.web.servlet.mvc.annotation
 - SimpleMappingExceptionResolver - org.springframework.web.servlet.handler
- HandlerExceptionResolverComposite - org.springframework.web.servlet.handler
- MyExceptionHandler - com.flight.manager.merchant.handler
- MyExceptionHandler - com.flight.manager.handler

HandlerExceptionResolver接口–异常处理接口

- SimpleMappingExceptionResolver通过配置文件进行异常处理。
- AnnotationMethodHandlerExceptionResolver：通过注解进行异常处理。

ViewResolver

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- ✓ ① **ViewResolver** - org.springframework.web.servlet
- ✓ ② **AbstractCachingViewResolver** - org.springframework.web.servlet.view
 - ③ **ResourceBundleViewResolver** - org.springframework.web.servlet.view
- ✓ ④ **UrlBasedViewResolver** - org.springframework.web.servlet.view
 - ✓ ⑤ **AbstractTemplateViewResolver** - org.springframework.web.servlet.view
 - ⑥ **FreeMarkerViewResolver** - org.springframework.web.servlet.view.freemarker
 - ⑦ **GroovyMarkupViewResolver** - org.springframework.web.servlet.view.groovy
 - ✓ ⑧ **VelocityViewResolver** - org.springframework.web.servlet.view.velocity
 - ⑨ **VelocityLayoutViewResolver** - org.springframework.web.servlet.view.velocity
 - ⑩ **InternalResourceViewResolver** - org.springframework.web.servlet.view
 - ⑪ **JasperReportsViewResolver** - org.springframework.web.servlet.view.jasperreports
 - ⑫ **ScriptTemplateViewResolver** - org.springframework.web.servlet.view.script
 - ⑬ **TilesViewResolver** - org.springframework.web.servlet.view.tiles3
 - ⑭ **TilesViewResolver** - org.springframework.web.servlet.view.tiles2
 - ⑮ **XsltViewResolver** - org.springframework.web.servlet.view.xslt
- ⑯ **XmlViewResolver** - org.springframework.web.servlet.view
- ⑰ **BeanNameViewResolver** - org.springframework.web.servlet.view
- ⑱ **ContentNegotiatingViewResolver** - org.springframework.web.servlet.view
- ⑲ **StaticViewResolver** - org.springframework.test.web.servlet.setup.StandaloneMockMvcBuilder
- ⑳ **ViewResolverComposite** - org.springframework.web.servlet.view

<https://blog.csdn.net/yanweihpu>

ViewResolver接口解析View视图。

UrlBasedViewResolver类 通过配置文件，把一个视图名交给到一个View来处理。

本文整理自网络，原文出处暂不知，对原文做了较大的改动，在此说明！整理人：java团长

作者：方志朋，一线大厂架构师，

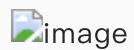
来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

SSM:谈谈你对SpringAOP的了解

一、引言

众所周知，一旦提到AOP，相信大家都是条件反射的想到JDK代理和CGLib代理，没错，这两个代理都是在运行时内存中临时生成代理类，故而又称作运行时增强——动态代理。世间万物都不是绝对的，既然有动态代理，那么，是否有想过：是不是存在静态代理呢？

二、LTW (Load Time Weaving)

其实，除了运行时织入切面的方式外，我们还有一种途径进行切面织入，它可以在类加载期通过字节码转换，进而将目标织入切入点（目标类），这种方式就是LTW，即静态代理（静待代理也被称作编译时增强，后面会有相关代码样例）。

LTW在Java5的时候就被引入了，想要了解其原理，先要了解一个知识——Instrument包。

三、java.lang.instrument包的工作原理

JDK5.0时引入了此包，目的就是为了能对JVM底层组建进行访问。如何访问？其实说来个人觉得还挺麻烦的，就是需要通过JVM的启动参数–javaagent在启动时获取JVM内部组件的引用。参数格式如下：

–javaagent:[=options]

此处先卖个关子，不急着解释参数中的jarpath和options，后面的运行代码及结果的样例中会进行针对使用红框标记说明，效果更好。

那么，它和AOP有和关系呢？

因为它在JVM启动时会装配并应用ClassTransformer，对类字节码进行转换，进而实现AOP的功能。

下面说一下instrument包下的两个重要接口：

ClassFileTransformer

它是Class文件转换器接口，这个接口有且仅有一个方法，如图所示：

```
MANIFEST.MF MyTransformer.java MyAgent.java InstrumentTest.java ClassFileTransformer.class
1 /**
2 package java.lang.instrument;
3
4 import java.security.ProtectionDomain;
5
6 public abstract interface ClassFileTransformer
7 {
8     public abstract byte[] transform(ClassLoader paramClassLoader, String paramString, Class<?> paramClass, ProtectionDomain paramProtectionDomain, byte[] paramArrayOfByte);
9 }
10 }
```

注意，这个参数可以看作是className，具体效果可以参照文章后面的代码及运行示例，此处因截屏窗口原因，挡住了一些参数，固下面还有张图来补全

芋道源码

```
MANIFEST.MF MyTransformer.java MyAgent.java InstrumentTest.java ClassFileTransformer.class
1 /**
2 * 2012 Chao Chen (cnfree2000@hotmail.com) ***
3
4
5
6
7
8     public abstract void addTransformer(ClassFileTransformer paramClassFileTransformer, boolean paramBoolean);
9
10 
```

上图的截图未展示最后一个参数，所以此处分两次截图补全

这个参数正是类文件对应的字节码数组

芋道源码

注意：transform方法会有一个返回值，类型是byte[], 表示转换后的字节码，但是如果返回为空，则表示不进行节码转换处理，千万不要当作是把原先类的字节码清空。

Instrumentation

这个接口提供了很多方法，我们主要注意一个方法即可，即：addTransformer方法，它的作用就是把一些ClassFileTransformer注册到JVM内部，接口如图所示：

```
MANIFEST.MF MyTransformer.java MyAgent.java InstrumentTest.java Instrumentation.class
1 /**
2 package java.lang.instrument;
3
4 import java.util.jar.JarFile;
5
6 public abstract interface Instrumentation
7 {
8     public abstract void addTransformer(ClassFileTransformer paramClassFileTransformer, boolean paramBoolean);
9     public abstract void addTransformer(ClassFileTransformer paramClassFileTransformer);
10 }
```

芋道源码

具体工作原理是这样的：

- ClassFileTransformer实例注册到JVM之后，JVM在加载Class文件时，就会先调用ClassFileTransformer的transform()方法进行字节码转换；
- 若注册了多个ClassFileTransformer实例，则按照注册时的顺序进行一次调用。

这样也就实现了从JVM层面截获字节码，进而织入操作者自己希望添加的逻辑，即实现AOP效果。

代码及演示效果

说了这么多，来点干货，下面用代码给大家演示一下如何向JVM中注册转换器实现AOP的。为了方便大家阅读，重要的说明笔者已经写在代码的注释上或者图片空白处，大家注意查看。

首先，我们实现一个自己的转换器，用于模拟需要切入的功能



```
MANIFEST.MF MyTransformer.java MyAgent.java InstrumentTest.java
1 package demo.instrument;
2
3 import java.lang.instrument.ClassFileTransformer;
4 import java.lang.instrument.IllegalClassFormatException;
5 import java.security.ProtectionDomain;
6
7 /**
8 * 模拟转换器具体功能
9 * author java架构的微慢与偏见
10 *
11 */
12 public class MyTransformer implements ClassFileTransformer{
13
14     /**
15      * 重写了transform方法，以实现我们自身的切面逻辑
16      */
17     @Override
18     public byte[] transform(ClassLoader paramClassLoader, String paramString,
19             Class<?> paramClass, ProtectionDomain paramProtectionDomain,
20             byte[] paramArrayOfByte) throws IllegalClassFormatException {
21         System.out.println("此处顺便给大家看一下当前的ClassLoader:"+paramClassLoader.getClass());
22         System.out.println("模拟AOP织入所需的功能，打印当前类名：" + paramString);
23         return null;
24     }
25
26 }
27
```

这里打印classloader是为了给大家回忆一下，不了解classloader相关知识可以看下笔者之前的文
章。
此处打印一下参数paramString，因为文章开始时说了这个参数可以理解为 className。
于道源码

注意，这里再强调下，代码中的return null;并不是将加载类的字节码置空。

其次，我们再实现一个代理类

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
MANIFEST.MF MyTransformer.java MyAgent.java InstrumentTest.java
1 package demo.instrument;
2
3 import java.lang.instrument.ClassFileTransformer;
4 import java.lang.instrument.Instrumentation;
5
6 /**
7 * 代理类，用于注册转换器
8 * @author java架构的傲慢与偏见
9 *
10 */
11 public class MyAgent {
12
13 /**
14 * 注意，此处的方法名premain不是随意起的，代理类必须按照下面方法进行定义
15 *
16 * @param args 这个参数不要小看，-javaagent启动参数的jarpath的值就会通过这个参数传递进来
17 * @param instrumentation 代表JVM内部组建的实例，用于注册ClassFileTransformer
18 */
19 public static void premain(String args, Instrumentation instrumentation) {
20     ClassFileTransformer transformer = new MyTransformer();
21     instrumentation.addTransformer(transformer);
22 }
23 }
```

这里说的JVM启动参数格式如下：
-javaagent:<jarpath>[<options>
说明下，jarpath是代理类的jar文件，options是传递给当前代理类的一个字符串参数

新出我们自己的转换器，通过调用addTransformer方法进行注册入JVM

为什么要实现代理类内，因为不是动态代理呀。。。

最后，我们写一个主函数，代表程序入口

```
MANIFEST.MF MyTransformer.java MyAgent.java InstrumentTest.java
1 package demo.instrument;
2
3 /**
4 * 程序入口类
5 * @author java架构的傲慢与偏见
6 *
7 */
8 public class InstrumentTest {
9
10 /**
11 * 用main()方法模拟程序入口
12 * @param args
13 */
14 public static void main(String[] args) {
15     System.out.println("This is InstrumentTest main()");
16 }
17 }
18 }
```

用main()方法模拟程序入口

公众号: 方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

到此为止，我们的Demo算是完成了，先来看一下运行的结果：

The screenshot shows a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The title bar also displays 'Microsoft Windows [版本 10.0.17134.765] (c) 2018 Microsoft Corporation. 保留所有权利。'. The window content includes:
1. A red box highlights the command: 'G:\>java -javaagent:myTransformer.jar demo.instrument.InstrumentTest'.
2. A red arrow points from the text '模拟AOF织入所需的功能，打印当前类名: demo/instrument/InstrumentTest' to the highlighted command.
3. Another red arrow points from the text 'This is InstrumentTest main()' to the command.
4. A red box highlights the output: 'This is InstrumentTest main()'.
5. A red box highlights the text: '这里大家可以看到在我们的main方法执行之前，织入了我们自己需要处理的逻辑，以此达到切面效果'.

使用JVM启动参数模拟运行

G:\>java -javaagent:myTransformer.jar demo.instrument.InstrumentTest
模拟AOF织入所需的功能，打印当前类名: demo/instrument/InstrumentTest
This is InstrumentTest main()

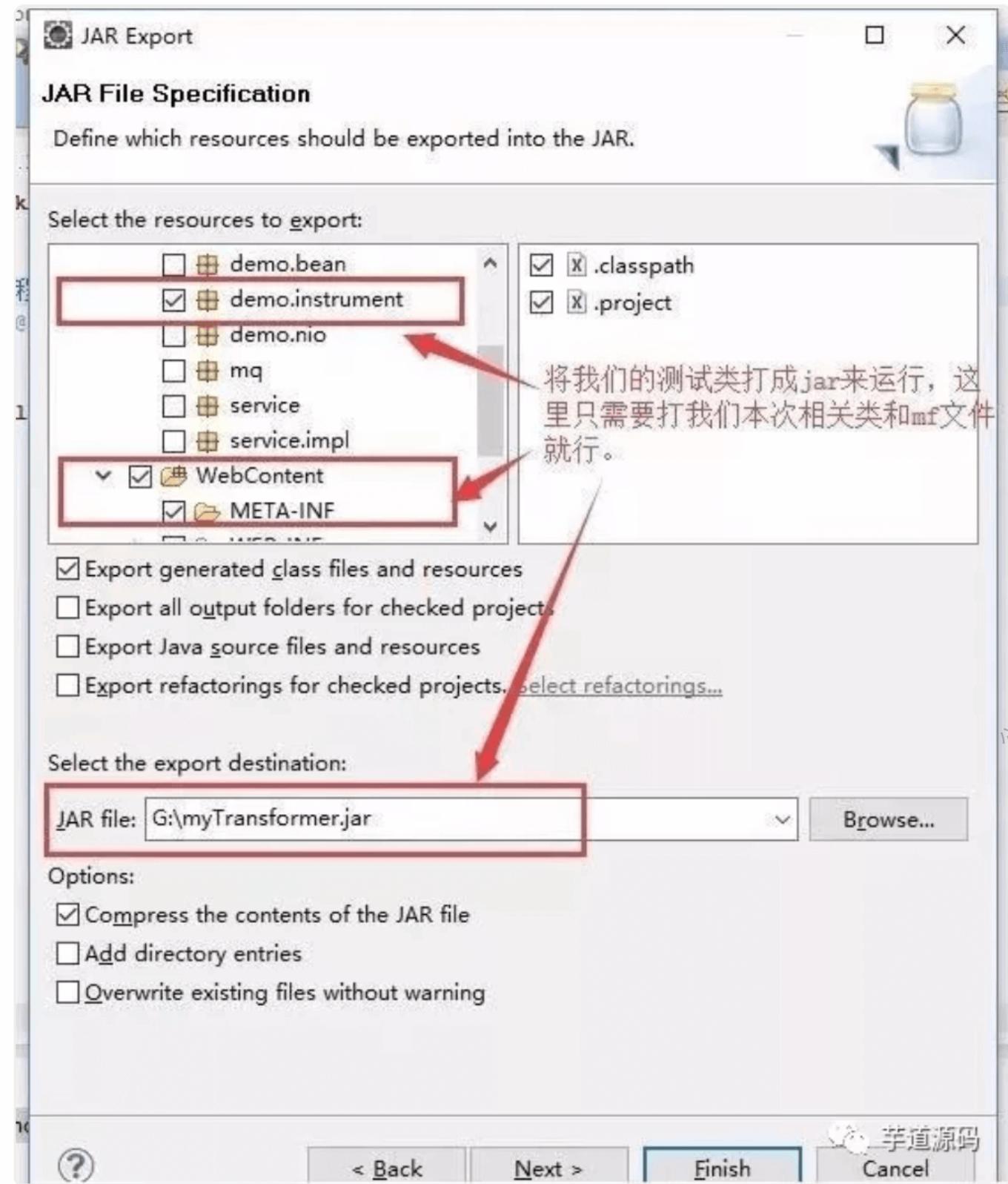
这里大家可以看到在我们的main方法执行之前，织入了我们自己需要处理的逻辑，以此达到切面效果

芋道源码

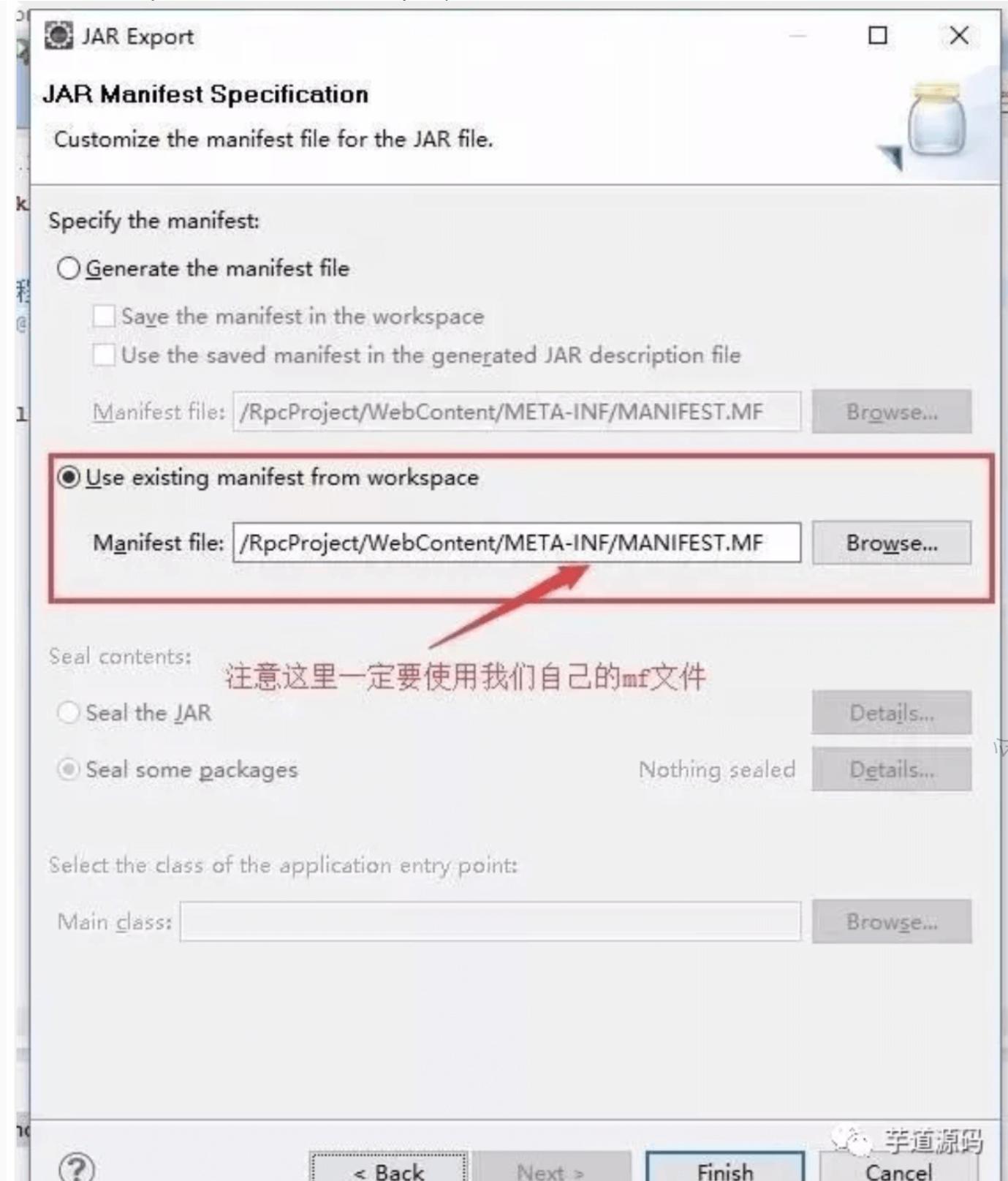
五、打jar的时候需要注意的地方

大家看到执行结果的截图中，cmd界面下运行javaagent参数时指定了一个myTransformer.jar，这个jar是我们自己需要打出来的，可以直接使用eclipse具体步骤如下图所示，注意图中说明：

公众号：方志朋



公众号：方志朋



总结

大家可以看到，其实使用此类代理并没有动态代理方便，甚至转换器可能会对JVM所有类都产生影响，操作起来更新相对麻烦，实际生产部署时会有很多不便。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

但是，写这些是为了让大家更好、更多的去了解AOP，我们所熟知的AOP其实还有很多东西有待我们自身去学习和发现，其实Spring在“操作麻烦”这方面还是做了不少事的，提供了一些xml的配置化管理（此处就不再说了，因为感觉一说又是一大长篇，有兴趣的大家可以自己去看看，多了解写东西总没有坏处），很多情况下已经不需要再配置javaagent参数了。

最后提一句，如果在面试中提到了这些，相信面试官也会有加分吧。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

SSM:必须掌握的20种Spring常用注解

注解本身没有功能的，就和 xml 一样。注解和 xml 都是一种元数据，元数据即解释数据的数据，这就是所谓配置。

本文主要罗列 Spring、Spring MVC相关注解的简介。

Spring部分

1、声明bean的注解

`@Component` 组件，没有明确的角色

`@Service` 在业务逻辑层使用（service层）

`@Repository` 在数据访问层使用（dao层）

`@Controller` 在展现层使用，控制器的声明（C）

2、注入bean的注解

`@Autowired`: 由Spring提供

`@Inject`: 由JSR-330提供

`@Resource`: 由JSR-250提供

都可以注解在set方法和属性上，推荐注解在属性上（一目了然，少写代码）。

3、java配置类相关注解

`@Configuration` 声明当前类为配置类，相当于xml形式的Spring配置（类上）

`@Bean` 注解在方法上，声明当前方法的返回值为一个bean，替代xml中的方式（方法上）

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

@Configuration 声明当前类为配置类，其中内部组合了@Component注解，表明这个类是一个bean（类上）

@ComponentScan 用于对Component进行扫描，相当于xml中的（类上）

@WishlyConfiguration 为@Configuration与@ComponentScan的组合注解，可以替代这两个注解

4、切面（AOP）相关注解

Spring支持AspectJ的注解式切面编程。

@Aspect 声明一个切面（类上）

使用@After、@Before、@Around定义建言（advice），可直接将拦截规则（切点）作为参数。

@After 在方法执行之后执行（方法上）

@Before 在方法执行之前执行（方法上）

@Around 在方法执行之前与之后执行（方法上）

@PointCut 声明切点

在java配置类中使用@EnableAspectJAutoProxy注解开启Spring对AspectJ代理的支持（类上）

5、@Bean的属性支持

@Scope 设置Spring容器如何新建Bean实例（方法上，得有@Bean）

其设置类型包括：

Singleton（单例，一个Spring容器中只有一个bean实例，默认模式），

Protetype（每次调用新建一个bean），

Request（web项目中，给每个http request新建一个bean），

Session（web项目中，给每个http session新建一个bean），

GlobalSession（给每一个global http session新建一个Bean实例）

@StepScope 在Spring Batch中还有涉及

@PostConstruct 由JSR-250提供，在构造函数执行完之后执行，等价于xml配置文件中bean的initMethod

@PreDestory 由JSR-250提供，在Bean销毁之前执行，等价于xml配置文件中bean的destroyMethod

6、@Value注解

@Value 为属性注入值（属性上）

支持如下方式的注入：

》注入普通字符

```
1 @Value("Michel Jackson")
2 String name;
```

》注入操作系统属性

```
1 @Value("#{systemProperties['os.name']}")
2 String osName;
```

》注入表达式结果

```
1 @Value("#{ T(java.lang.Math).random()*10}")
2 String randomNuber;
```

》注入其它bean属性

```
1 @Value("#{domeClass.name}")
2 String name;
```

》注入文件资源

```
1 @Value("classpath:com/hgs/hello/test.txt")
2 String resourceFile;
```

》注入网站资源

```
1 @Value("http://www.fangzhipeng.com")
2 Resource url;
```

》注入配置文件

```
1 @Value ("${book.name}")
2 String bookName;
```

注入配置使用方法：

- 编写配置文件 (test.properties)

book.name=《三体》

- `@PropertySource` 加载配置文件(类上)

```
1 @PropertySource("classpath:com/hgs/hello/test.txt")
```

还需配置一个PropertySourcesPlaceholderConfigurer的bean。

7、环境切换

`@Profile` 通过设定Environment的ActiveProfiles来设定当前context需要使用的配置环境。 (类或方法上)

`@Conditional` Spring4中可以使用此注解定义条件话的bean，通过实现Condition接口，并重写matches方法，从而决定该bean是否被实例化。 (方法上)

8、异步相关

`@EnableAsync` 配置类中，通过此注解开启对异步任务的支持，叙事性AsyncConfigurer接口 (类上)

`@Async` 在实际执行的bean方法使用该注解来申明其是一个异步任务 (方法上或类上所有的方法都将异步，需要`@EnableAsync`开启异步任务)

9、定时任务相关

`@EnableScheduling` 在配置类上使用，开启计划任务的支持（类上）

`@Scheduled` 来申明这是一个任务，包括cron,fixDelay,fixRate等类型（方法上，需先开启计划任务的支持）

10、`@Enable*`注解说明

这些注解主要用来开启对xxx的支持。

`@EnableAspectJAutoProxy` 开启对AspectJ自动代理的支持

`@EnableAsync` 开启异步方法的支持

`@EnableScheduling` 开启计划任务的支持

`@EnableWebMvc` 开启Web MVC的配置支持

`@EnableConfigurationProperties` 开启对@ConfigurationProperties注解配置Bean的支持

`@EnableJpaRepositories` 开启对SpringData JPA Repository的支持

`@EnableTransactionManagement` 开启注解式事务的支持

`@EnableTransactionManagement` 开启注解式事务的支持

`@EnableCaching` 开启注解式的缓存支持

11、测试相关注解

`@RunWith` 运行器，Spring中通常用于对JUnit的支持

```
1 @RunWith(SpringJUnit4ClassRunner.class)
```

`@ContextConfiguration` 用来加载配置ApplicationContext，其中classes属性用来加载配置类

```
1 @ContextConfigration(classes={Testconfig.class})
```

SpringMVC部分

@EnableWebMvc 在配置类中开启Web MVC的配置支持，如一些ViewResolver或者MessageConverter等，若无此句，重写WebMvcConfigurerAdapter方法（用于对SpringMVC的配置）。

@Controller 声明该类为SpringMVC中的Controller

@RequestMapping 用于映射Web请求，包括访问路径和参数（类或方法上）

@ResponseBody 支持将返回值放在response内，而不是一个页面，通常用户返回json数据（返回值旁或方法上）

@RequestBody 允许request的参数在request体中，而不是在直接连接在地址后面。（放在参数前）

@PathVariable 用于接收路径参数，比如@RequestMapping("/hello/{name}")申明的路径，将注解放在参数中前，即可获取该值，通常作为Restful的接口实现方法。

@RestController 该注解为一个组合注解，相当于@Controller和@ResponseBody的组合，注解在类上，意味着，该Controller的所有方法都默认加上了@ResponseBody。

@ControllerAdvice 通过该注解，我们可以将对于控制器的全局配置放置在同一个位置，注解了@Controller的类的方法可使用@ExceptionHandler、@InitBinder、@ModelAttribute注解到方法上，这对所有注解了 @RequestMapping的控制器内的方法有效。

@ExceptionHandler 用于全局处理控制器里的异常

@InitBinder 用来设置WebDataBinder，WebDataBinder用来自动绑定前台请求参数到Model中。

@ModelAttribute 本来的作用是绑定键值对到Model里，在@ControllerAdvice中是让全局的 @RequestMapping都能获得在此处设置的键值对。

原文链接：

<https://blog.csdn.net/yelvgou9995/article/details/83345267>

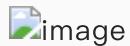
还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



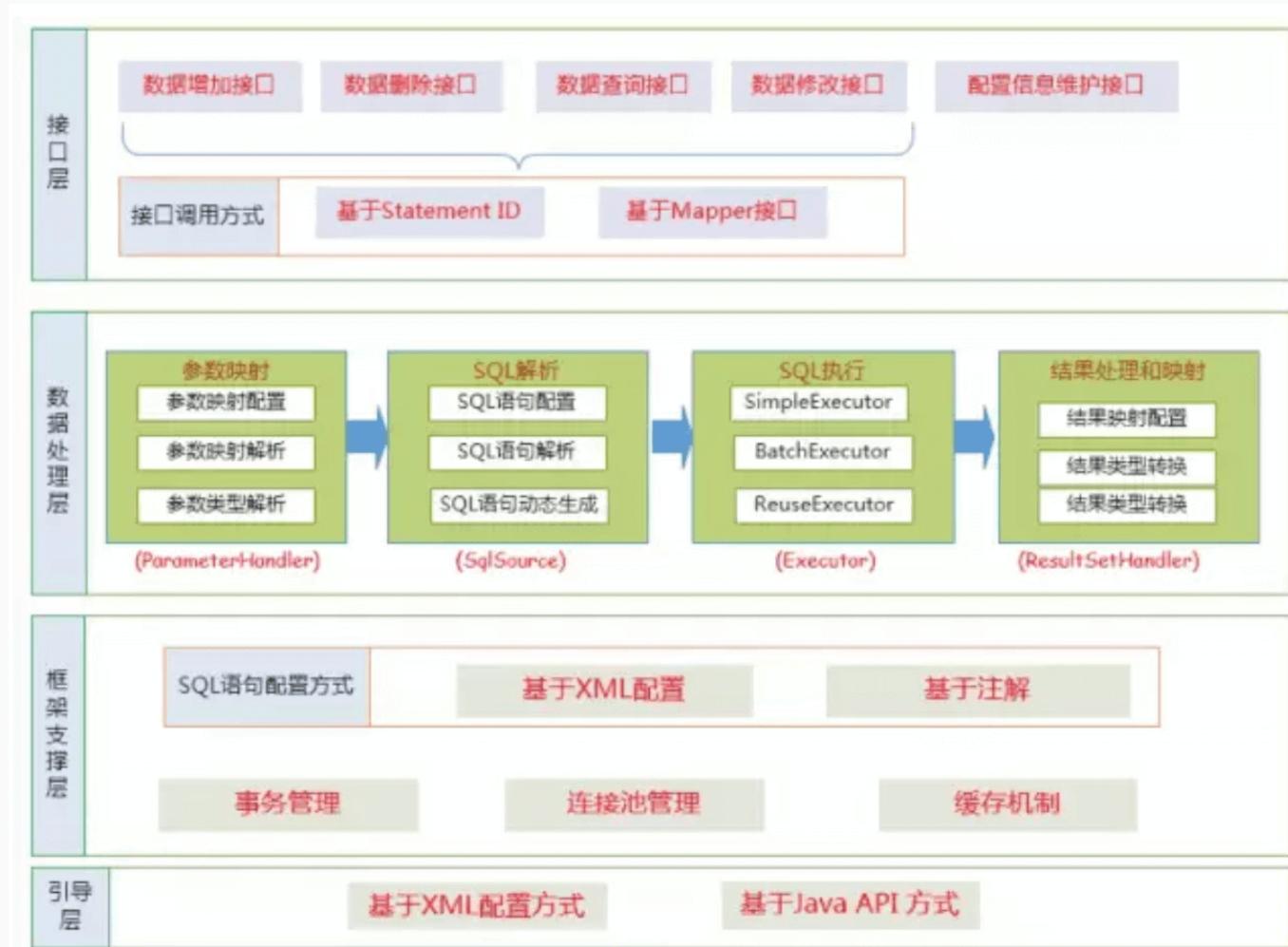
程序员理财，请关注：



公众号：方志朋

SSM:Mybatis架构与原理

MyBatis功能架构设计

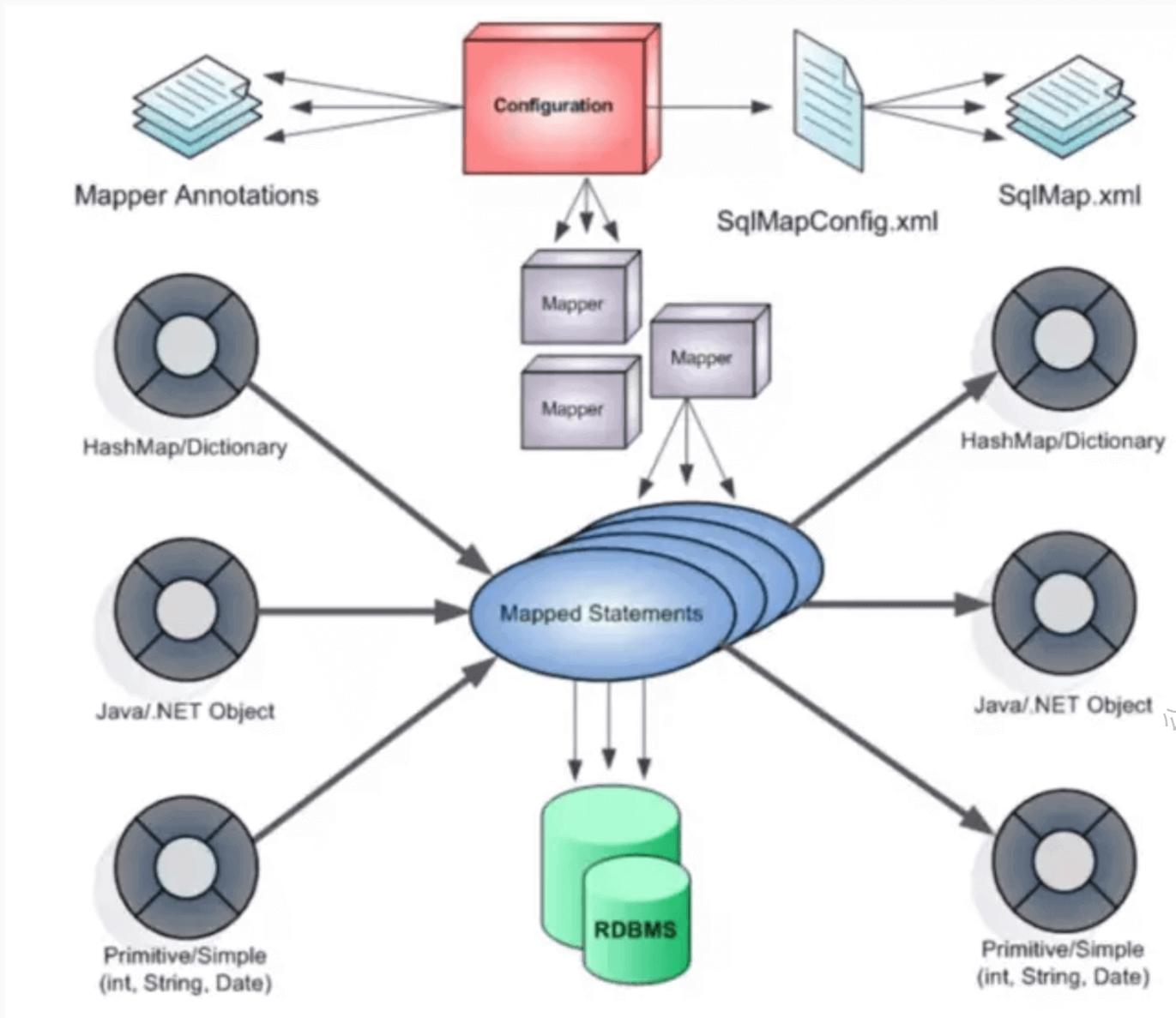


功能架构讲解：

我们把Mybatis的功能架构分为三层：

- API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。
- 数据处理层：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。
- 基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑。

框架架构



框架架构讲解：

这张图从上往下看。MyBatis的初始化，会从mybatis-config.xml配置文件，解析构造成Configuration这个类，就是图中的红框。

- 加载配置：配置来源于两个地方，一处是配置文件，一处是Java代码的注解，将SQL的配置信息加载成为一个个MappedStatement对象（包括了传入参数映射配置、执行的SQL语句、结果映射配置），存储在内存中。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

-SQL解析：当API接口层接收到调用请求时，会接收到传入SQL的ID和传入对象（可以是Map、JavaBean或者基本数据类型），Mybatis会根据SQL的ID找到对应的MappedStatement，然后根据传入参数对象对MappedStatement进行解析，解析后可以得到最终要执行的SQL语句和参数。

- SQL执行：将最终得到的SQL和参数拿到数据库进行执行，得到操作数据库的结果。
- 结果映射：将操作数据库的结果按照映射的配置进行转换，可以转换成HashMap、JavaBean或者基本数据类型，并将最终结果返回。

MyBatis核心类

1、SqlSessionFactoryBuilder

每一个MyBatis的应用程序的入口是SqlSessionFactoryBuilder。

它的作用是通过XML配置文件创建Configuration对象（当然也可以在程序中自行创建），然后通过build方法创建SqlSessionFactory对象。没有必要每次访问Mybatis就创建一次SqlSessionFactoryBuilder，通常的做法是创建一个全局的对象就可以了。

示例程序如下：

```
1
2 private static SqlSessionFactoryBuilder sqlSessionFactoryBuilder;
3 private static SqlSessionFactory sqlSessionFactory;
4
5 private static void init() throws IOException {
6     String resource = "mybatis-config.xml";
7     Reader reader = Resources.getResourceAsReader(resource);
8     sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();
9     sqlSessionFactory = sqlSessionFactoryBuilder.build(reader);
10 }
```

org.apache.ibatis.session.Configuration 是mybatis初始化的核心。

mybatis-config.xml中的配置，最后会解析xml成Configuration这个类。

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymysq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

SqlSessionFactoryBuilder根据传入的数据流(XML)生成Configuration对象，然后根据Configuration对象创建默认的SqlSessionFactory实例。

SqlSessionFactory对象由SqlSessionFactoryBuilder创建：

它的主要功能是创建SqlSession对象，和SqlSessionFactoryBuilder对象一样，没有必要每次访问Mybatis就创建一次SqlSessionFactory，通常的做法是创建一个全局的对象就可以了。

SqlSessionFactory对象一个必要的属性是Configuration对象，它是保存Mybatis全局配置的一个配置对象，通常由SqlSessionFactoryBuilder从XML配置文件创建。

这里给出一个简单的示例：

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE configuration PUBLIC
3   "-//mybatis.org//DTD Config 3.0//EN"
4   "http://mybatis.org/dtd/mybatis-3-config.dtd">
5 <configuration>
6   <!-- 配置别名 -->
7   <typeAliases>
8     <typeAlias type="org.iMybatis.abc.dao.UserDao" alias="User
      Dao" />
9     <typeAlias type="org.iMybatis.abc.dto.UserDto" alias="User
      Dto" />
10  </typeAliases>
11
12  <!-- 配置环境变量 -->
13  <environments default="development">
14    <environment id="development">
15      <transactionManager type="JDBC" />
16      <dataSource type="POOLED">
17        <property name="driver" value="com.mysql.jdbc.Driv
          er" />
18        <property name="url" value="jdbc:mysql://127.0.0.
          1:3306/iMybatis?characterEncoding=GBK" />
19        <property name="username" value="iMybatis" />
20        <property name="password" value="iMybatis" />
21      </dataSource>
22    </environment>
```

公众号：方志朋

```
23    </environments>
24
25    <!-- 配置mappers -->
26    <mappers>
27        <mapper resource="org/iMybatis/abc/dao/UserDao.xml" />
28    </mappers>
29
30 </configuration>
```

3、SqlSession

SqlSession对象的主要功能是完成一次数据库的访问和结果的映射，它类似于数据库的session概念，由于不是线程安全的，所以SqlSession对象的作用域需限制方法内。

SqlSession的默认实现类是DefaultSqlSession，它有两个必须配置的属性：Configuration和Executor。Configuration前文已经描述这里不再多说。SqlSession对数据库的操作都是通过Executor来完成的。

SqlSession：默认创建DefaultSqlSession 并且开启一级缓存，创建执行器、赋值。

SqlSession有一个重要的方法getMapper，顾名思义，这个方式是用来获取Mapper对象的。什么是Mapper对象？

根据Mybatis的官方手册，应用程除了要初始并启动Mybatis之外，还需要定义一些接口，接口里定义访问数据库的方法，存放接口的包路径下需要放置同名的XML配置文件。

SqlSession的getMapper方法是联系应用程序和Mybatis纽带，应用程序访问getMapper时，Mybatis会根据传入的接口类型和对应的XML配置文件生成一个代理对象，这个代理对象就叫Mapper对象。应用程序获得Mapper对象后，就应该通过这个Mapper对象来访问Mybatis的SqlSession对象，这样就达到里插入到Mybatis流程的目的。

```
1 SqlSession session= sqlSessionFactory.openSession();
2 UserDao userDao = session.getMapper(UserDao.class);
3 UserDto user = new UserDto();
4 user.setUsername("iMybatis");
5 List<UserDto> users = userDao.queryUsers(user);
6
7 public interface UserDao {
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
8     public List<UserDto> queryUsers(UserDto user) throws Exception
9 {
10
11    <?xml version="1.0" encoding="UTF-8" ?>
12    <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "htt
13    p://mybatis.org/dtd/mybatis-3-mapper.dtd">
14    <mapper namespace="org.iMybatis.abc.dao.UserDao">
15        <select id="queryUsers" parameterType="UserDto" resultType="U
16        serDto"
17            useCache="false">
18            <![CDATA[
19                select * from t_user t where t.username = #{username}
20            ]]>
21        </select>
22    </mapper>
```

4、Executor

Executor对象在创建Configuration对象的时候创建，并且缓存在Configuration对象里。Executor对象的主要功能是调用StatementHandler访问数据库，并将查询结果存入缓存中（如果配置了缓存的话）。

5、StatementHandler

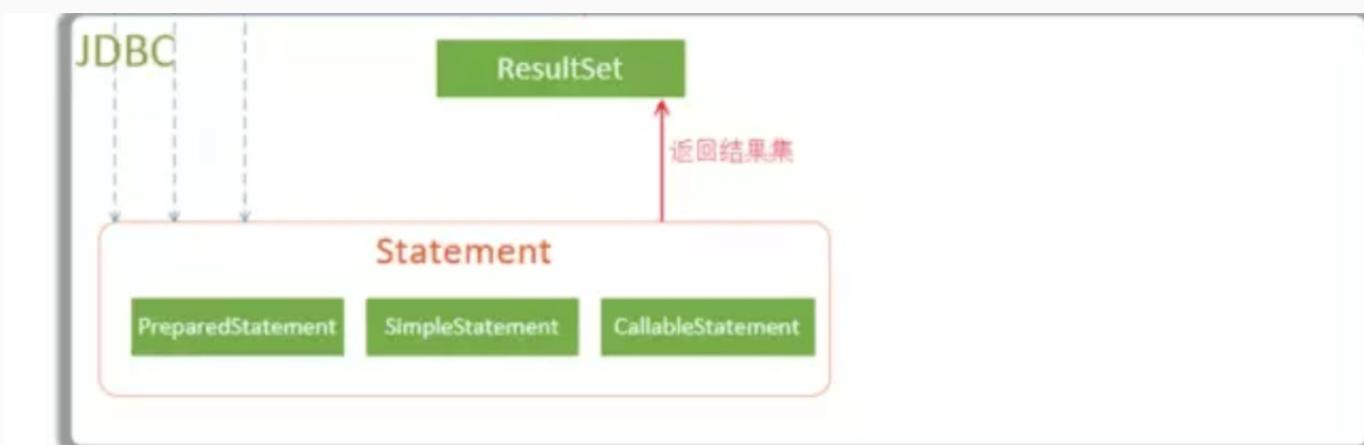
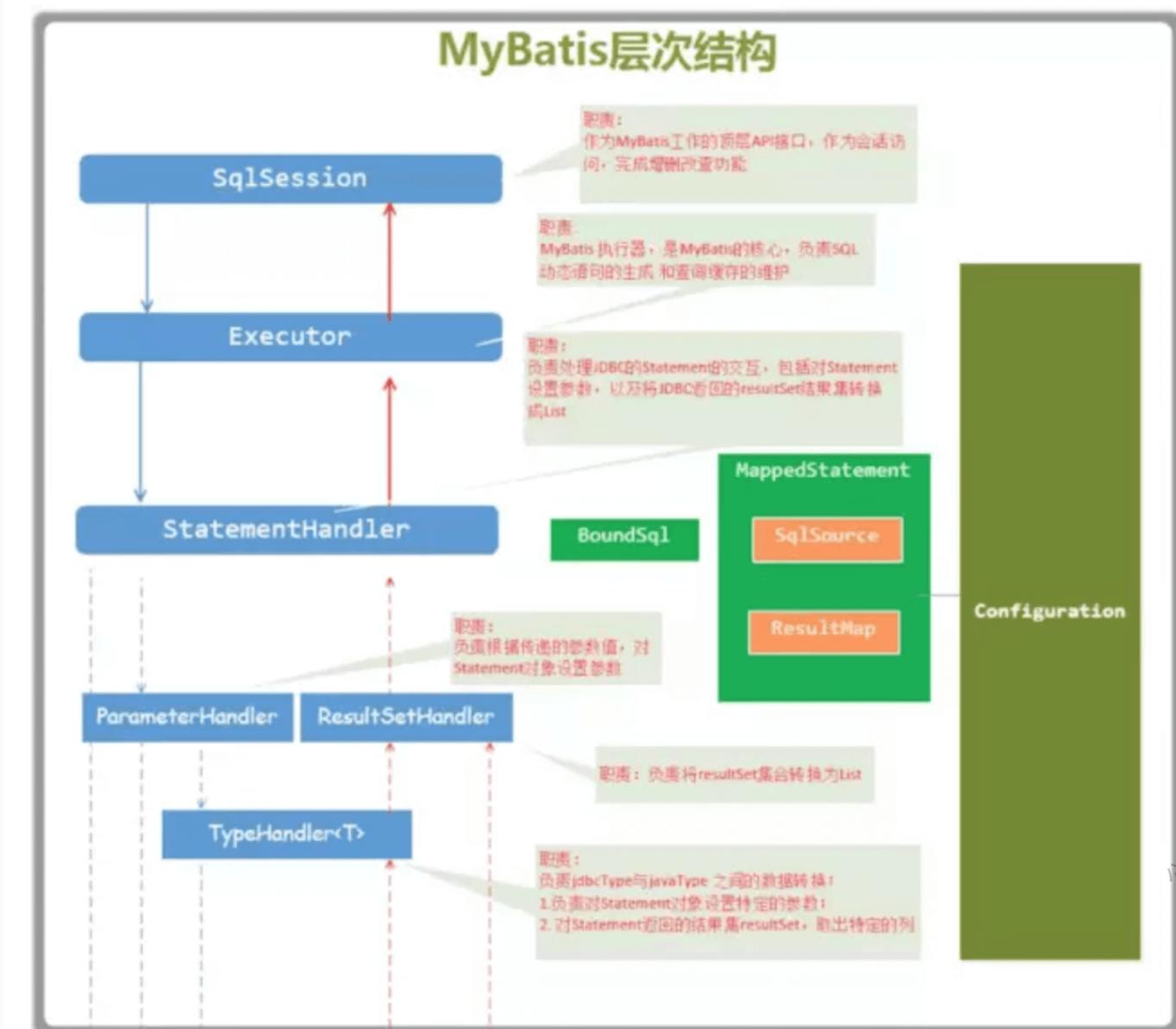
StatementHandler是真正访问数据库的地方，并调用ResultSetHandler处理查询结果。

6、ResultSetHandler

处理查询结果

MyBatis成员层次&职责

java架构师公众号：方志朋



- SqlSession 作为 MyBatis 工作的主要顶层 API，表示和数据库交互的会话，完成必要数据库增删改查功能
- Executor MyBatis 执行器，是 MyBatis 调度的核心，负责 SQL 语句的生成和查询缓存的维护

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- StatementHandler 封装了 JDBC Statement操作，负责对JDBCstatement的操作，如设置参数、将Statement结果集转换成List集合。
 - ParameterHandler 负责对用户传递的参数转换成JDBC Statement 所需要的参数
 - ResultSetHandler *负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；
 - TypeHandler 负责java数据类型和jdbc数据类型之间的映射和转换
 - MappedStatement MappedStatement维护了一条
 - 节点的封
- SqlSource 负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回
- BoundSql 表示动态生成的SQL语句以及相应的参数信息
 - Configuration MyBatis所有的配置信息都维持在Configuration对象之中

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



SSM:Mybatis常见面试题总结及答案

1、什么是Mybatis？

- 1、Mybatis是一个半ORM（对象关系映射）框架，它内部封装了JDBC，开发时只需要关注SQL语句本身，不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。程序员直接编写原生态sql，可以严格控制sql执行性能，灵活度高。
- 2、MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。
- 3、通过xml 文件或注解的方式将要执行的各种 statement 配置起来，并通过java对象和 statement中 sql的动态参数进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射为java对象并返回。（从执行sql到返回result的过程）。

2、Mybatis的优点：

- 1、基于SQL语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL写在XML里，解除sql与程序代码的耦合，便于统一管理；提供XML标签，支持编写动态SQL语句，并可重用。
- 2、与JDBC相比，减少了50%以上的代码量，消除了JDBC大量冗余的代码，不需要手动开关连接；
- 3、很好的与各种数据库兼容（因为MyBatis使用JDBC来连接数据库，所以只要JDBC支持的数据库MyBatis都支持）。
- 4、能够与Spring很好的集成；
- 5、提供映射标签，支持对象与数据库的ORM字段关系映射；提供对象关系映射标签，支持对象关系组件维护。

3、MyBatis框架的缺点：

- 1、SQL语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写SQL语句的功底有一定要求。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

2、SQL语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

4、MyBatis框架适用场合：

- 1、MyBatis专注于SQL本身，是一个足够灵活的DAO层解决方案。
- 2、对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis将是不错的选择。

5、MyBatis与Hibernate有哪些不同？

- 1、Mybatis和hibernate不同，它不完全是一个ORM框架，因为MyBatis需要程序员自己编写Sql语句。
- 2、Mybatis直接编写原生态sql，可以严格控制sql执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，因为这类软件需求变化频繁，一旦需求变化要求迅速输出成果。但是灵活的前提是mybatis无法做到数据库无关性，如果需要实现支持多种数据库的软件，则需要自定义多套sql映射文件，工作量大。
- 3、Hibernate对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件，如果用hibernate开发可以节省很多代码，提高效率。

6、#{ } 和 \${ } 的区别是什么？

、#{ } 是预编译处理，\${ } 是字符串替换。

Mybatis在处理#{ }时，会将sql中的#{ }替换为?号，调用PreparedStatement的set方法来赋值；

Mybatis在处理时，就是把{}替换成变量的值。

使用#{ }可以有效的防止SQL注入，提高系统安全性。

7、当实体类中的属性名和表中的字段名不一样，怎么办？

第1种：通过在查询的sql语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

```
1 <select
```

```
2
3 id
4 =
5 "selectorder"
6
7 parameterType
8 =
9 "int"
10
11 resultType
12 =
13 "me.gacl.domain.order"
14 >
15
16     select order_id id, order_no orderno ,order_price price fo
    rm orders where order_id=#{id};
17
18 </select>
```

第2种：通过 来映射字段名和实体类属性名的一一对应的关系。

```
1 <select
2
3 id
4 =
5 "getOrder"
6
7 parameterType
8 =
9 "int"
10
11 resultMap
12 =
13 "orderresultmap"
14 >
15
16 select * from orders where order_id=#{id}
17
```

公众号：方志朋

```
18 </select>
19
20
21
22 <resultMap
23
24 type
25 =
26 "me.gacl.domain.order"
27
28 id
29 =
30 "orderresultmap"
31 >
32
33     <!--用id属性来映射主键字段-->
34
35
36 <id
37
38 property
39 =
40 "id"
41
42 column
43 =
44 "order_id"
45 >
46
47
48
49     <!--用result属性来映射非主键字段，property为实体类属性名，column为数据
表中的属性-->
50
51
52 <result
53
54 property
55 =
56 "orderno"
```

公众号：方志朋

```
57
58 column
59 =
60 "order_no"
61 />
62
63
64 <result
65
66 property
67 =
68 "price"
69
70 column
71 =
72 "order_price"
73
74 />
75
76 </reslutMap>
```

8、 模糊查询like语句该怎么写？

第1种：在Java代码中添加sql通配符。

```
1 string wildcardname = "%smi%";
2
3 list
4 <name>
5 names = mapper.selectlike(wildcardname);
6
7
8
9 <select
10
11 id
12 =
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
13 "selectlike"
14 >
15
16 select * from foo where bar like #{value}
17
18 </select>
```

第2种：在sql语句中拼接通配符，会引起sql注入

```
1 string wildcardname = "smi";
2
3 list
4 <name>
5 names = mapper.selectlike(wildcardname);
6
7
8
9
10
11 <select
12
13 id
14 =
15 "selectlike"
16 >
17
18 select * from foo where bar like "%"#{value}%""
19
20 </select>
```

9、通常一个Xml映射文件，都会写一个Dao接口与之对应，请问，这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗？

Dao接口即Mapper接口。接口的全限名，就是映射文件中的namespace的值；接口的方法名，就是映射文件中Mapper的Statement的id值；接口方法内的参数，就是传递给sql的参数。

Mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为key值，可唯一定位一个MapperStatement。在Mybatis中，每一个`<select>`、`</select>`、`<update>`、`</update>`、`<insert>`、`</insert>`、`<delete>`、`</delete>`标签，都会被解析为一个MapperStatement对象。

举例：`com.mybatis3.mappers.StudentDao.findStudentById`，可以唯一找到namespace为`com.mybatis3.mappers.StudentDao`下面 id 为 `findStudentById` 的 MapperStatement。

Mapper接口里的方法，是不能重载的，因为是使用 全限名+方法名 的保存和寻找策略。Mapper 接口的工作原理是JDK动态代理，Mybatis运行时会使用JDK动态代理为Mapper接口生成代理对象proxy，代理对象会拦截接口方法，转而执行MapperStatement所代表的sql，然后将sql执行结果返回。

10、Mybatis是如何进行分页的？分页插件的原理是什么？

Mybatis使用RowBounds对象进行分页，它是针对ResultSet结果集执行的内存分页，而非物理分页。可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页参数。

11、Mybatis是如何将sql执行结果封装为目标对象并返回的？都有哪些映射形式？

第一种是使用`resultType`标签，逐一定义数据库列名和对象属性名之间的映射关系。

第二种是使用sql列的别名功能，将列的别名书写为对象属性名。

有了列名与属性名的映射关系后，Mybatis通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

12、如何执行批量插入？

首先，创建一个简单的insert语句：

```
1 <insert  
2
```

```
3 id  
4 =  
5 "insertname"  
6 >  
7  
8 insert into names (name) values (#${value})  
9  
10 </insert>
```

然后在java代码中像下面这样执行批处理插入：

```
1 list <  
2 string  
3 > names =  
4 new  
5 arraylist();  
6  
7 names.add("fred");  
8  
9 names.add("barney");  
10  
11 names.add("betty");  
12  
13 names.add("wilma");  
14  
15 // 注意这里 executortype.batch  
16  
17 sqlsession sqlsession = sqlsessionfactory.opensession(executortyp  
e.batch);  
18  
19 try  
20 {  
21  
22     namemapper mapper = sqlsession.getmapper(namemapper.  
23 class  
24 );  
25  
26
```

公众号：方志朋

```
27 for
28 {
29     string
30     name: names) {
31
32         mapper.insertname(name);
33
34     }
35
36     sqlsession.commit();
37
38 }
39 catch
40 (
41 Exception
42 e) {
43
44     e.printStackTrace();
45
46     sqlSession.rollback();
47
48
49 throw
50 e;
51
52 }
53
54 finally
55 {
56
57     sqlsession.close();
58
59 }
```

公众号：方志朋

13、如何获取自动生成的(主)键值？

insert 方法总是返回一个int值，这个值代表的是插入的行数。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

如果采用自增长策略，自动生成的键值在 insert 方法执行完后可以被设置到传入的参数对象中。

示例：

```
1 <insert
2
3 id
4 =
5 "insertname"
6
7 usegeneratedkeys
8 =
9 "true"
10
11 keyproperty
12 =
13 "id"
14 >
15
16     insert into names (name) values (#{name})
17
18 </insert>
19
20 name name = new name();
21
22 name.setName("fred");
23
24
25
26 int rows = mapper.insertname(name);
27
28 // 完成后，id已经被设置到对象中
29
30 System.out.println("rows inserted = " + rows);
31
32 System.out.println("generated key value = " + name.getId());
```

公众号：方志朋

14、在mapper中如何传递多个参数？

1、第一种：

DAO层的函数

```
1 public
2
3 UserselectUser
4 (
5 String
6 name,
7 String
8 area);
9
10 对应的xml,#{}
11 0
12 }代表接收的是dao层中的第一个参数, #{}
13 1
14 }代表dao层中第二参数, 更多参数一致往后加即可。
15
16 <select
17
18 id
19 =
20 "selectUser"
21 resultMap
22 =
23 "BaseResultMap"
24 >
25
26
27     select *  fromuser_user_t    whereuser_name = #{0} anduser_are
28     a=#{1}
29 </select>
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

2、第二种：使用 `@param` 注解：

```
1 public
2
3 interface
4 usermapper {
5
6     user selectuser(
7 @param
8 ("username") string username,
9 @param
10 ("hashedpassword") string hashedpassword);
11
12 }
```

然后，就可以在xml像下面这样使用(推荐封装为一个map,作为单个参数传递给mapper)：

```
1 <select
2
3 id
4 =
5 "selectuser"
6
7 resulttype
8 =
9 "user"
10 >
11
12         select id, username, hashedpassword
13
14         from some_table
15
16         where username = #{username}
17
18         and hashedpassword = #{hashedpassword}
19
20 </select>
```

公众号：方志朋

3、第三种：多个参数封装成map

```
1 try
2 {
3
4
5 //映射文件的命名空间.SQL片段的ID，就可以调用对应的映射文件中的SQL
6
7
8 //由于我们的参数超过了两个，而方法中只有一个Object参数收集，因此我们使用Map集
9 合来装载我们的参数
10
11 Map
12 <
13 String
14 ,
15 Object
16 > map =
17 new
18
19 HashMap
20 ();
21
22     map.put(
23 "start"
24 , start);
25
26     map.put(
27 "end"
28 , end);
29
30
31 return
32 sqlSession.selectList(
33 "StudentID.pagination"
34 , map);
35
```

公众号：方志朋

```
36 }
37 catch
38 (
39 Exception
40 e) {
41
42     e.printStackTrace();
43
44     sqlSession.rollback();
45
46
47 throw
48 e;
49
50 }
51 finally
52 {
53
54
55 MybatisUtil
56 .closeSqlSession();
57
58 }
```

公众号：方志朋

15、Mybatis动态sql有什么用？执行原理？有哪些动态sql？

Mybatis动态sql可以在Xml映射文件内，以标签的形式编写动态sql，执行原理是根据表达式的值 完成逻辑判断并动态拼接sql的功能。

Mybatis提供了9种动态sql标签：trim|where|set|foreach|if|choose|when|otherwise|bind。

16、Xml映射文件中，除了常见的select|insert|update|delete标签之外，还有哪些标签？

答：、、、、，加上动态sql的9个标签，其中 为sql片段标签，通过 标签引入sql片段， 为不支持自增的主键生成策略标签。

17、Mybatis的Xml映射文件中，不同的Xml映射文件，id是否可以重复？

不同的Xml映射文件，如果配置了namespace，那么id可以重复；如果没有配置namespace，那么id不能重复；

原因就是namespace+id是作为Map <String,MapperStatement>的key使用的，如果没有namespace，就剩下id，那么，id重复会导致数据互相覆盖。有了namespace，自然id就可以重复，namespace不同，namespace+id自然也就不同。

18、为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？

Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半自动ORM映射工具。

19、一对一、一对多的关联查询？

```
1  
2  
3  
4 <!--association 一对一关联查询 -->  
5  
6  
7  
8 <select  
9  
10 id  
11 =  
12 "getClass"  
13  
14 parameterType  
15 =  
16 "int"
```

公众号：方志朋

```
17
18 resultMap
19 =
20 "ClassesResultMap"
21 >
22
23
24         select * from class c,teacher t where c.teacher_id=t.t_i
25         d and c.c_id=#{id}
26
27 </select>
28
29
30
31
32
33 <resultMap
34
35 type
36 =
37 "com.lcb.user.Classes"
38
39 id
40 =
41 "ClassesResultMap"
42 >
43
44
45
46 <!-- 实体类的字段名和数据表的字段名映射 -->
47
48
49
50 <id
51
52 property
53 =
54 "id"
55
```

```
56 column
57 =
58 "c_id"
59 />
60
61
62
63 <result
64
65 property
66 =
67 "name"
68
69 column
70 =
71 "c_name"
72 />
73
74
75
76 <association
77
78 property
79 =
80 "teacher"
81
82 javaType
83 =
84 "com.lcb.user.Teacher"
85 >
86
87
88
89 <id
90
91 property
92 =
93 "id"
94
95 column
```

公众号：方志朋

```
96 =
97 "t_id"
98 />
99
100
101
102 <result
103
104 property
105 =
106 "name"
107
108 column
109 =
110 "t_name"
111 />
112
113
114
115 </association>
116
117
118
119 </resultMap>
120
121
122
123
124
125
126
127 <!--collection 一对多关联查询 -->
128
129
130
131 <select
132
133 id
134 =
135 "getClass2"
```

```
136
137 parameterType
138 =
139 "int"
140
141 resultMap
142 =
143 "ClassesResultMap2"
144 >
145
146
147         select * from class c,teacher t,student s where c.teache
r_id=t.t_id and c.c_id=s.class_id and c.c_id=#{id}
148
149
150 </select>
151
152
153
154
155
156 <resultMap
157
158 type
159 =
160 "com.lcb.user.Classes"
161
162 id
163 =
164 "ClassesResultMap2"
165 >
166
167
168
169 <id
170
171 property
172 =
173 "id"
174
```

公众号：方志朋

```
175 column
176 =
177 "c_id"
178 />
179
180
181
182 <result
183
184 property
185 =
186 "name"
187
188 column
189 =
190 "c_name"
191 />
192
193
194
195 <association
196
197 property
198 =
199 "teacher"
200
201 javaType
202 =
203 "com.lcb.user.Teacher"
204 >
205
206
207
208 <id
209
210 property
211 =
212 "id"
213
214 column
```

公众号：方志朋

```
215 =
216 "t_id"
217 />
218
219
220
221 <result
222
223 property
224 =
225 "name"
226
227 column
228 =
229 "t_name"
230 />
231
232
233
234 </association>
235
236
237
238
239
240 <collection
241
242 property
243 =
244 "student"
245
246 ofType
247 =
248 "com.lcb.user.Student"
249 >
250
251
252
253 <id
254
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymysq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
255 property
256 =
257 "id"
258
259 column
260 =
261 "s_id"
262 />
263
264
265
266 <result
267
268 property
269 =
270 "name"
271
272 column
273 =
274 "s_name"
275 />
276
277
278
279 </collection>
280
281
282
283 </resultMap>
284
285
286 </mapper>
```

公众号：方志朋

20、MyBatis实现一对一有几种方式?具体怎么操作的?

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次,通过在resultMap里面配置association节点配置一对一的类就可以完成;

嵌套查询是先查一个表,根据这个表里面的结果的外键id,去再另外一个表里面查询数据,也是通过association配置,但另外一个表的查询通过select属性配置。

21、MyBatis实现一对多有几种方式,怎么操作的?

有联合查询和嵌套查询。联合查询是几个表联合查询,只查询一次,通过在resultMap里面的collection节点配置一对多的类就可以完成; 嵌套查询是先查一个表,根据这个表里面的 结果的外键id,去再另外一个表里面查询数据,也是通过配置collection,但另外一个表的查询通过select节点配置。

22、Mybatis是否支持延迟加载? 如果支持, 它的实现原理是什么?

答: Mybatis仅支持association关联对象和collection关联集合对象的延迟加载, association指的就是一对一, collection指的就是一对多查询。在Mybatis配置文件中, 可以配置是否启用延迟加载 lazyLoadingEnabled=true|false。

它的原理是, 使用CGLIB创建目标对象的代理对象, 当调用目标方法时, 进入拦截器方法, 比如调用 a.getB().getName(), 拦截器invoke()方法发现a.getB()是null值, 那么就会单独发送事先保存好的查询关联B对象的sql, 把B查询上来, 然后调用a.setB(b), 于是a的对象b属性就有值了, 接着完成 a.getB().getName()方法的调用。这就是延迟加载的基本原理。

当然了, 不光是Mybatis, 几乎所有的包括Hibernate, 支持延迟加载的原理都是一样的。

23、Mybatis的一级、二级缓存:

- 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, 默认打开一级缓存。
- 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。默认不打开二级缓存, 要开启二级缓存, 使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态), 可在它的映射文件中配置 ;
- 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。

24、什么是MyBatis的接口绑定? 有哪些实现方式?

接口绑定, 就是在MyBatis中任意定义接口,然后把接口里面的方法和SQL语句绑定, 我们直接调用接口方法就可以,这样比起原来SqlSession提供的方法我们可以有更加灵活的选择和设置。

接口绑定有两种实现方式,一种是通过注解绑定, 就是在接口的方法上面加上 @Select、@Update等注解, 里面包含Sql语句来绑定; 另外一种就是通过xml里面写SQL来绑定, 在这种情况下,要指定xml映射文

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

件里面的namespace必须为接口的全路径名。当Sql语句比较简单时候,用注解绑定,当SQL语句比较复杂时候,用xml绑定,一般用xml绑定的比较多。

25、使用MyBatis的mapper接口调用时有哪些要求？

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。

26、Mapper编写有哪几种方式？

第一种：接口实现类继承SqlSessionDaoSupport：使用此种方法需要编写mapper接口， mapper接口实现类、 mapper.xml文件。

1、在sqlMapConfig.xml中配置mapper.xml的位置

```
1 <mappers>
2
3
4 <mapper
5
6 resource
7 =
8 "mapper.xml文件的地址"
9
10 />
11
12
13 <mapper
14
15 resource
16 =
17 "mapper.xml文件的地址"
18
19 />
20
```

公众号：方志朋

```
21 </mappers>
```

1、定义mapper接口

3、实现类集成SqlSessionDaoSupport

mapper方法中可以this.getSqlSession()进行数据增删改查。4、spring 配置

```
1 <bean  
2  
3 id  
4 =  
5 ""  
6  
7 class  
8 =  
9 "mapper接口的实现"  
10 >  
11  
12  
13 <property  
14  
15 name  
16 =  
17 "sqlSessionFactory"  
18  
19 ref  
20 =  
21 "sqlSessionFactory"  
22 ></property>  
23  
24 </bean>
```

第二种：使用 org.mybatis.spring.mapper.MapperFactoryBean：

1、在sqlMapConfig.xml中配置mapper.xml的位置，如果mapper.xml和mappre接口的名称相同且在同一个目录，这里可以不用配置

```
1 <mappers>
```

```
2  
3  
4 <mapper  
5  
6 resource  
7 =  
8 "mapper.xml文件的地址"  
9  
10 />  
11  
12  
13 <mapper  
14  
15 resource  
16 =  
17 "mapper.xml文件的地址"  
18  
19 />  
20  
21 </mappers>
```

2、定义mapper接口：

- 1、mapper.xml中的namespace为mapper接口的地址
- 2、mapper接口中的方法名和mapper.xml中的定义的statement的id保持一致
- 3、Spring中定义

```
1 <bean  
2  
3 id  
4 =  
5 ""  
6  
7 class  
8 =  
9 "org.mybatis.spring.mapper.MapperFactoryBean"  
10 >  
11
```

公众号：方志朋

```
12
13 <property
14
15 name
16 =
17 "mapperInterface"
18
19 value
20 =
21 "mapper接口地址"
22
23 />
24
25
26
27 <property
28
29 name
30 =
31 "sqlSessionFactory"
32
33 ref
34 =
35 "sqlSessionFactory"
36
37 />
38
39
40 </bean>
```

公众号：方志朋

第三种：使用mapper扫描器：

1、 mapper.xml文件编写：

mapper.xml中的namespace为mapper接口的地址； mapper接口中的方法名和mapper.xml中的定义的statement的id保持一致；如果将mapper.xml和mapper接口的名称保持一致则不用在sqlMapConfig.xml中进行配置。

2、 定义mapper接口：

注意mapper.xml的文件名和mapper的接口名称保持一致，且放在同一个目录

3、配置mapper扫描器：

```
1 <bean
2
3   class
4   =
5   "org.mybatis.spring.mapper.MapperScannerConfigurer"
6   >
7
8
9   <property
10
11     name
12     =
13     "basePackage"
14
15     value
16     =
17     "mapper接口包地址"
18   ></property>
19
20
21   <property
22
23     name
24     =
25     "sqlSessionFactoryBeanName"
26
27     value
28     =
29     "sqlSessionFactory"
30   />
31
32
33 </bean>
```

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

4、使用扫描器后从spring容器中获取mapper的实现对象。

27、简述Mybatis的插件运行原理，以及如何编写一个插件。

答：Mybatis仅可以编写针对ParameterHandler、ResultSetHandler、StatementHandler、Executor这4种接口的插件，Mybatis使用JDK的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这4种接口对象的方法时，就会进入拦截方法，具体就是InvocationHandler的invoke()方法，当然，只会拦截那些你指定需要拦截的方法。

编写插件：实现Mybatis的Interceptor接口并复写intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

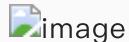
作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



公众号：方志朋

程序员理财，请关注：



SSM:MyBatis中的\$和#, 用不好，准备走人！

这是一次代码优化过程中发现的问题，在功能优化后发现部分数据查不到出来了，问题就在于一条sql上的#和\$

下图为两条sql：

```
select ww.Id,ww.MainTitle,ww.SubTitle,ww.HomeImg,ww.SortId,ww.HomeLabel,.....,ww.SortId,ww.XcxUrl,ww.ListImg  
from WellWelfare ww ,WellWelfareLabelRelation wwlr  
where 1 = 1  
and ww.Id = wwlr.WellWelfareId  
and ww.IsValid = 1  
and wwlr.LabelId in(${showLabels})  
and ww.BeginTime <= #{nowDate}  
and ww.EndTime >= #{nowDate}  
and wwlr.Type = 1  
order by ww.SortId  
</select>  
  
select ww.Id,ww.MainTitle,ww.SubTitle,ww.HomeImg,ww.SortId,ww.HomeLabel,.....,ww.SortId,ww.XcxUrl,ww.ListImg,  
(select count(*) from MemberRecord mr where mr.EntranceId= 6 and mr.ActivityId = ww.id and mr.UnionId = ${unionId}) as status  
from WellWelfare ww ,WellWelfareLabelRelation wwlr  
where 1 = 1  
and ww.Id = wwlr.WellWelfareId  
and ww.IsValid = 1  
and wwlr.LabelId in(${showLabels})  
and ww.BeginTime <= #{nowDate}  
and ww.EndTime >= #{nowDate}  
and wwlr.Type = 1  
order by ww.SortId  
</select>
```

从图上可以看出 wwlr.LabelId in() 处理的方式是不一样的。

区别

1、#{ }是预编译处理，MyBatis在处理#{ }时，它会将sql中的#{ }替换为？，然后调用PreparedStatement的set方法来赋值，传入字符串后，会在值两边加上单引号，如上面的值“4,44,514”就会变成“ '4,44,514' ”；

2、是字符串替换，MyBatis在处理\${ }时，它会将sql中的\${ }替换为变量的值，传入的数据不会加两边加上单引号。

注意：使用\${ }会导致sql注入，不利于系统的安全性！

SQL注入：就是通过把SQL命令插入到Web表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

常见的有匿名登录（在登录框输入恶意的字符串）、借助异常获取数据库信息等

应用场景：

- 1、#{ }：主要用户获取DAO中的参数数据,在映射文件的SQL语句中出现#{ }表达式,底层会创建预编译的SQL；
- 2、：主要用于获取配置文件数据, **DAO**接口中的参数信息,当 出现在映射文件的SQL语句中时创建的不是预编译的SQL,而是字符串的拼接,有可能会导致SQL注入问题.所以一般使用\$接收dao参数时,这些参数一般是字段名,表名等,例如order by {column}。

注：

{}获取DAO参数数据时,参数必须使用@param注解进行修饰或者使用下标或者参数#{param1}形式；

、{}获取DAO参数数据时,假如参数个数多于一个可有选择的使用@param。

问题分析

其实刚开始我也没太去看sql里的#和\$, 我把sql放到数据库跑一切正常, 所以我就将代码的执行sql输出到控制台了, 具体是这么一个输出sql的配置文件

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6<configuration>
7
8<settings>
9    <!-- changes from the defaults -->
10   <setting name="LazyLoadingEnabled" value="false"/>
11</settings>
12
13<typeAliases>
14   <typeAlias alias="HashMap" type="java.util.HashMap"/>
15</typeAliases>
16
17 <!-- mybatis sql 拦截器，可打印mybatis执行的sql，及计算执行时间，线上环境时建议关闭-->
18<plugins>
19    <plugin interceptor="com.ly.mdd.frame.core.mybatis.SQLPrintPlugin">
20        <property name="show_sql" value="false"/>
21    </plugin>
22    <!-- mybatis 分页插件配置 -->
23    <plugin interceptor="com.github.pagehelper.PageInterceptor">
24        <!-- 配用会话化时，如果pageNum<1且pageNum>pages会返回空数组 -->
25        <property name="reasonable" value="true" />
26    </plugin>
27</plugins>
28
29</configuration>
```

输出后，终于发现了问题在哪里。。。

看了上面的区别介绍，相信大家其实都应该知道区别在哪里，我们的问题在哪里，其实就是sql在in的时候，里面的数据被加了两个双引号。“wwlr.LabelId in (4,44,514) 就会变成 wwlrl.LabelId in ('4,44,514')；所以导致部分数据查不到了。

解决办法

1、快速解决

最快的方法就是把#直接替换成\$，这样问题应该就可以解决了。

但是，我很无语，我确没有解决。

本地跑代码一点问题都没有，部署到公司的docker上问题一样没解决，给人的感觉就是代码根本没有从#变\$。

大家都知道

其实是有危险性，会容易被sql注入，据我所知道，我们公司的**docker**是会加一层防止sql注入的功能，所以不知道是不是这个功能把的无效掉了。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

当然，我也没有去再到服务上打出sql来看一下，因为本来\$就是不太安全的，所以我换了一种方式处理。

2、foreach标签的使用

foreach标签主要用于构建in条件，他可以在sql中对集合进行迭代。

先来看看语法：

```
<delete id="deleteBatch">
    delete from user where id in
        <foreach collection="array" item="id" index="index" open="(" close=")" separator=",">
            ={id}
        </foreach>
    </delete>

我们假设说参数为..... int[] ids = {1,2,3,4,5} ....那么打印之后的SQL如下：
delete form user where id in (1,2,3,4,5)

释义：
collection : collection属性的值有三个分别是list、array、map三种，分别对应的参数类型为：List、数组、map集合。我在上面传的参数为数组，所以值为array
item : 表示在迭代过程中每一个元素的别名
index : 表示在迭代过程中每次迭代到的位置（下标）
open : 前缀
close : 后缀
separator : 分隔符，表示迭代时每个元素之间以什么分隔
```

通过上图，大家也应该也了解和使用这个标签了吧。

那对于我们项目中的改造，其实就是把原来传进来的字符型参数变成List，这样问题就完美的解决了，既实现了我们的功能，又解决了安全性问题。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



java架构师公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

公众号：方志朋

算法：5亿整数的大文件，怎么排？

5亿整数的大文件，怎么排？

问题

给你1个文件bigdata，大小4663M，5亿个数，文件中的数据随机，如下一行一个整数：

```
1 6196302
2 3557681
3 6121580
4 2039345
5 2095006
6 1746773
7 7934312
8 2016371
9 7123302
10 8790171
11 2966901
12 ...
13 7005375
```

现在要对这个文件进行排序，怎么搞？

内部排序

先尝试内排，选2种排序方式：

```
1 private final int cutoff = 8;
2
3 public <T> void perform(Comparable<T>[] a) {
4     perform(a,0,a.length - 1);
5 }
6
```

```
7     private <T> int median3(Comparable<T>[] a, int x, int y, int z)
8     {
9         if(lessThan(a[x],a[y])) {
10             if(lessThan(a[y],a[z])) {
11                 return y;
12             }
13             else if(lessThan(a[x],a[z])) {
14                 return z;
15             }else {
16                 return x;
17             }
18         }else {
19             if(lessThan(a[z],a[y])){
20                 return y;
21             }else if(lessThan(a[z],a[x])) {
22                 return z;
23             }else {
24                 return x;
25             }
26         }
27     }
28
29     private <T> void perform(Comparable<T>[] a, int low, int high)
30     {
31         int n = high - low + 1;
32         //当序列非常小，用插入排序
33         if(n <= cutoff) {
34             InsertionSort insertionSort = SortFactory.createInser
35             tionSort();
36             insertionSort.perform(a,low,high);
37             //当序列小时，使用median3
38         }else if(n <= 100) {
39             int m = median3(a,low,low + (n >>> 1),high);
40             exchange(a,m,low);
41             //当序列比较大时，使用ninther
42         }else {
43             int gap = n >>> 3;
44             int m = low + (n >>> 1);
45             int m1 = median3(a,low,low + gap,low + (gap << 1));
46             int m2 = median3(a,m - gap,m,m + gap);
```

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

```
44     int m3 = median3(a,high - (gap << 1),high - gap,high)
45     ;
46     int ninther = median3(a,m1,m2,m3);
47     exchange(a,ninther,low);
48 }
49 if(high <= low)
50     return;
51 //lessThan
52 int lt = low;
53 //greaterThan
54 int gt = high;
55 //中心点
56 Comparable<T> pivot = a[low];
57 int i = low + 1;
58
59 /*
60 * 不变式：
61 *   a[low..lt-1] 小于pivot -> 前部(first)
62 *   a[lt..i-1] 等于 pivot -> 中部(middle)
63 *   a[gt+1..n-1] 大于 pivot -> 后部(final)
64 *
65 *   a[i..gt] 待考察区域
66 */
67
68 while (i <= gt) {
69     if(lessThan(a[i],pivot)) {
70         //i-> ,lt ->
71         exchange(a,lt++,i++);
72     }else if(lessThan(pivot,a[i])) {
73         exchange(a,i,gt--);
74     }else{
75         i++;
76     }
77 }
78
79 // a[low..lt-1] < v = a[lt..gt] < a[gt+1..high].
80 perform(a,low,lt - 1);
81 perform(a,gt + 1,high);
82 }
```

公众号：方志朋

归并排序：

```
1 /**
2      * 小于等于这个值的时候，交给插入排序
3      */
4     private final int cutoff = 8;
5
6 /**
7      * 对给定的元素序列进行排序
8      *
9      * @param a 给定元素序列
10     */
11    @Override
12    public <T> void perform(Comparable<T>[] a) {
13        Comparable<T>[] b = a.clone();
14        perform(b, a, 0, a.length - 1);
15    }
16
17    private <T> void perform(Comparable<T>[] src, Comparable<T>[] dest, int low, int high) {
18        if (low >= high)
19            return;
20
21        //小于等于cutoff的时候，交给插入排序
22        if (high - low <= cutoff) {
23            SortFactory.createInsertionSort().perform(dest, low, hi
gh);
24            return;
25        }
26
27        int mid = low + ((high - low) >>> 1);
28        perform(dest, src, low, mid);
29        perform(dest, src, mid + 1, high);
30
31        //考虑局部有序 src[mid] <= src[mid+1]
32        if (lessThanOrEqual(src[mid], src[mid+1])) {
```

公众号：方志朋

```
33             System.arraycopy(src, low, dest, low, high - low + 1);
34         }
35
36         //src[low .. mid] + src[mid+1 .. high] -> dest[low .. high]
37         merge(src, dest, low, mid, high);
38     }
39
40     private <T> void merge(Comparable<T>[] src, Comparable<T>[] dest, int low, int mid, int high) {
41
42         for(int i = low, v = low, w = mid + 1; i <= high; i++) {
43             if(w > high || v <= mid && lessThanOrEqual(src[v], src[w])) {
44                 dest[i] = src[v++];
45             } else {
46                 dest[i] = src[w++];
47             }
48         }
49     }
```

公众号：方志朋

数据太多，递归太深 ->栈溢出？加大Xss？

数据太多，数组太长 -> OOM？加大Xmx？

耐心不足，没跑出来.而且要将这么大的文件读入内存，在堆中维护这么大数据量，还有内排中不断的拷贝，对栈和堆都是很大的压力，不具备通用性。

sort命令来跑

跑了多久呢？24分钟.

为什么这么慢？

粗略的看下我们的资源：

内存

jvm-heap/stack, native-heap/stack,page-cache, block-buffer

外存

swap + 磁盘

数据量很大，函数调用很多，系统调用很多，内核/用户缓冲区拷贝很多，脏页面回写很多，io-wait很高，io很繁忙，堆栈数据不断交换至swap，线程切换很多，每个环节的锁也很多。

总之，内存吃紧，向磁盘要空间，脏数据持久化过多导致cache频繁失效，引发大量回写，回写线程高，导致cpu大量时间用于上下文切换，一切，都很糟糕，所以24分钟不细看了，无法忍受。

位图法

```
1 private BitSet bits;
2
3     public void perform(
4             String largeFileName,
5             int total,
6             String destLargeFileName,
7             Castor<Integer> castor,
8             int readerBufferSize,
9             int writerBufferSize,
10            boolean asc) throws IOException {
11
12         System.out.println("BitmapSort Started.");
13         long start = System.currentTimeMillis();
14         bits = new BitSet(total);
15         InputPart<Integer> largeIn = PartFactory.createCharBuffer
16             edInputPart(largeFileName, readerBufferSize);
17         OutputPart<Integer> largeOut = PartFactory.createCharBuff
18             eredOutputPart(destLargeFileName, writerBufferSize);
19         largeOut.delete();
20
21         Integer data;
22         int off = 0;
23         try {
24             while (true) {
25                 data = largeIn.read();
26                 if (data == null)
27                     break;
28                 int v = data;
29                 set(v);
```

公众号：方志朋

```
28                 off++;
29             }
30             largeIn.close();
31             int size = bits.size();
32             System.out.println(String.format("lines : %d ,bits :
33 %d", off, size));
33
34         if (asc) {
35             for (int i = 0; i < size; i++) {
36                 if (get(i)) {
37                     largeOut.write(i);
38                 }
39             }
40         }else {
41             for (int i = size - 1; i >= 0; i--) {
42                 if (get(i)) {
43                     largeOut.write(i);
44                 }
45             }
46         }
47
48         largeOut.close();
49         long stop = System.currentTimeMillis();
50         long elapsed = stop - start;
51         System.out.println(String.format("BitmapSort Complete
52 d.elapsed : %dms",elapsed));
52     }finally {
53         largeIn.close();
54         largeOut.close();
55     }
56 }
57
58     private void set(int i) {
59         bits.set(i);
60     }
61
62     private boolean get(int v) {
63         return bits.get(v);
64     }
```

nice!跑了190秒，3分来钟。

以核心内存4663M/32大小的空间跑出这么个结果，而且大量时间在用于I/O，不错。

问题是，如果这个时候突然内存条坏了1、2根，或者只有极少的内存空间怎么搞？

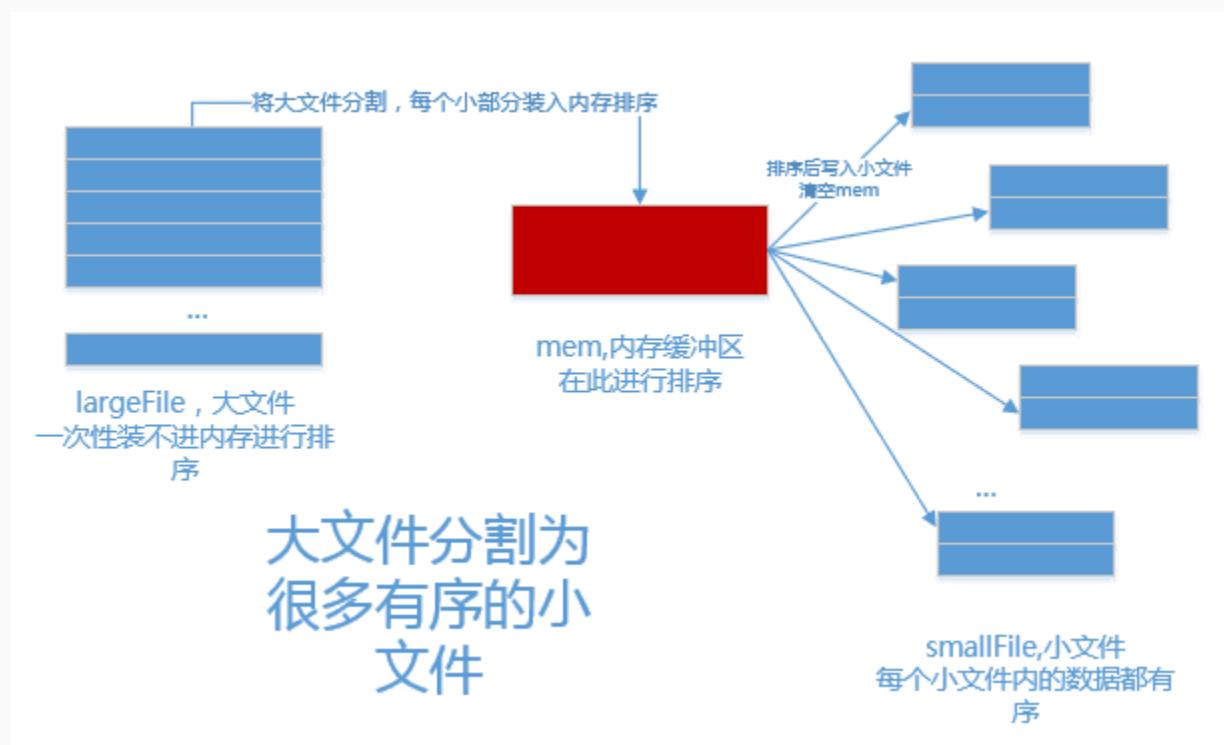
外部排序

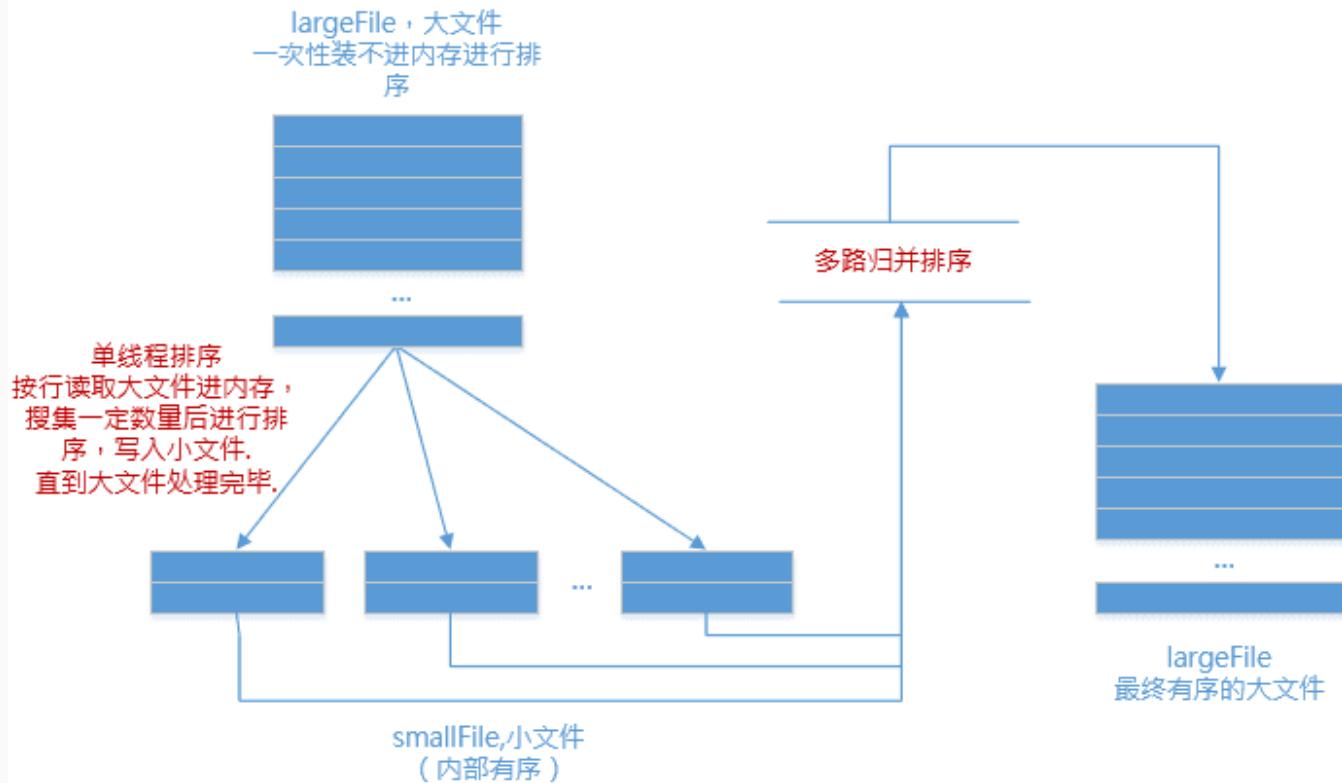
该外部排序上场了。

外部排序干嘛的？

| 内存极少的情况下，利用分治策略，利用外存保存中间结果，再用多路归并来排序；

map-reduce的嫡系。





1. 分

内存中维护一个极小的核心缓冲区`memBuffer`，将大文件`bigdata`按行读入，搜集到`memBuffer`满或者大文件读完时，对`memBuffer`中的数据调用内排进行排序，排序后将有序结果写入磁盘文件`bigdata.xxx.part.sorted`。

循环利用`memBuffer`直到大文件处理完毕，得到n个有序的磁盘文件：

<code>bigdata.13.part.sorted</code>	<code>bigdata.177.part.sorted</code>	<code>bigdata.35.part.sorted</code>
<code>bigdata.140.part.sorted</code>	<code>bigdata.178.part.sorted</code>	<code>bigdata.36.part.sorted</code>
<code>bigdata.141.part.sorted</code>	<code>bigdata.179.part.sorted</code>	<code>bigdata.37.part.sorted</code>
<code>bigdata.142.part.sorted</code>	<code>bigdata.17.part.sorted</code>	<code>bigdata.38.part.sorted</code>
<code>bigdata.143.part.sorted</code>	<code>bigdata.180.part.sorted</code>	<code>bigdata.39.part.sorted</code>
<code>bigdata.144.part.sorted</code>	<code>bigdata.181.part.sorted</code>	<code>bigdata.40.part.sorted</code>
<code>bigdata.145.part.sorted</code>	<code>bigdata.182.part.sorted</code>	<code>bigdata.41.part.sorted</code>
<code>bigdata.146.part.sorted</code>	<code>bigdata.183.part.sorted</code>	<code>bigdata.42.part.sorted</code>
<code>bigdata.147.part.sorted</code>	<code>bigdata.184.part.sorted</code>	<code>bigdata.43.part.sorted</code>
<code>bigdata.148.part.sorted</code>	<code>bigdata.185.part.sorted</code>	<code>bigdata.44.part.sorted</code>
<code>bigdata.149.part.sorted</code>	<code>bigdata.186.part.sorted</code>	<code>bigdata.45.part.sorted</code>
<code>bigdata.14.part.sorted</code>	<code>bigdata.187.part.sorted</code>	<code>bigdata.46.part.sorted</code>
<code>bigdata.150.part.sorted</code>	<code>bigdata.188.part.sorted</code>	<code>bigdata.47.part.sorted</code>
<code>bigdata.151.part.sorted</code>	<code>bigdata.189.part.sorted</code>	<code>bigdata.48.part.sorted</code>
<code>bigdata.152.part.sorted</code>	<code>bigdata.19.part.sorted</code>	<code>bigdata.49.part.sorted</code>

公众号：方志朋

2. 合

现在有了n个有序的小文件，怎么合并成1个有序的大文件？

把所有小文件读入内存，然后内排？

(⊙o⊙)...

no!

利用如下原理进行归并排序：

一个n个元素的有序小集合：

$$S = \{x \mid x_i \leq x_j, i, j \in [0, n)\}$$

现在我们有m个小集合： s_1, s_2, \dots, s_m

那么当前所有小集合中的最小值：

$$\min = \min(\min(s_1), \min(s_2), \dots, \min(s_m))$$

我们举个简单的例子：

文件1：3,6,9

文件2：2,4,8

文件3：1,5,7

第一回合：

文件1的最小值：3，排在文件1的第1行

文件2的最小值：2，排在文件2的第1行

文件3的最小值：1，排在文件3的第1行

那么，这3个文件中的最小值是： $\min(1,2,3) = 1$

也就是说，最终大文件的当前最小值，是文件1、2、3的当前最小值的最小值，绕么？

上面拿出了最小值1，写入大文件。

第二回合：

文件1的最小值：3，排在文件1的第1行

文件2的最小值：2，排在文件2的第1行

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

文件3的最小值：5，排在文件3的第2行

那么，这3个文件中的最小值是： $\min(5,2,3) = 2$

将2写入大文件。

也就是说，最小值属于哪个文件，那么就从哪个文件当中取下一行数据。（因为小文件内部有序，下一行数据代表了它当前的最小值）

最终的时间，跑了771秒，13分钟左右。

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

网络：TCP、IP协议族(一) HTTP简介、请求方法与响应状态码

接下来想系统的回顾一下TCP/IP协议族的相关东西，当然这些东西大部分是在大学的时候学过的，但是那句话，基础的东西还是要不时的回顾回顾的。接下来的几篇博客都是关于TCP/IP协议族的，本篇博客就先简单的聊一下TCP/IP协议族，然后聊一下HTTP协议，然后再聊一下SSL上的HTTP（也就是HTTPS）了。当然TCP/IP协议族是个老生常谈的话题，网络上关于该内容的文章一抓一大把呢，但是鉴于其重要性，还是有必要系统的总结一下的。

TCP/IP协议组简述

在聊HTTP与HTTPS之前呢，我们先简单的聊一下TCP/IP协议族。TCP/IP不单单指的就是TCP和IP这两个协议，而是指的与其相关的各种协议。比如HTTP, FTP, DNS, TCP, UDP, IP, SNMP等等都属于TCP/IP协议族的范畴。

TCP/IP协议的分层

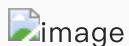
TCP/IP协议族是分层管理的，在OSI标准中可以分为7层（应用层、表示层、会话层、传输层、网络层、数据链路层、物理层，可记为：应表会传网数物），本篇博客我们采用的是TCP/IP协议族中的四层（应用层、传输层、网络层、链路层）。下方是对四层中每层的简单介绍：

- 应用层：该层是面向用户的一层，也就是说用户可以直接操作该层，该层决定了向用户提供应用服务时的通信活动。本篇博客要聊的HTTP (HyperText Transfer Protocol: 超文本传输协议) 就位于该层。我们经常使用的FTP(File Transfer Protocol: 文件传输协议)和DNS (Domain Name System: 域名系统)都位于该层。FTP简单的说就是用来文件传输的。而DNS则负责域名解析的，通过DNS可以将域名（比如：www.cnblogs.com）与IP地址（201.33.xx.09）进行相互的转换。在7层中，又将该层分为：应用层、表示层和会话层。
- 传输层：应用层的下方是传输层，应用层会将数据交付给传输层进行传输。TCP(Transmission Control Protocol: 传输控制协议)和UDP(User Data Protocol: 用户数据协议)位于该层，当然见名知意，该层是用来提供处于网络连接中的两台计算机直接的数据传输的。TCP建立连接是需要三次握手来确认连接情况，而UDP则没有三次握手的过程。稍后会介绍。
- 网络层：传输层的下方是网络层，网络层用来处理在网络上流动的数据包，IP(Internet Protocol: 网际协议)就位于这层。该层负责在众多网络线路中选择一条传输线路。当然这个选择传输线路的过程需要IP地址和MAC地址的支持。

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- 链路层：在7层协议中，将链路层分为数据链路层和物理层。该部分主要是用来处理网络的硬件部分，我们常说的NIC（Net Work Card），也就是网卡就位于这一部分，当然光纤也是链路层的一部分。



在TCP/IP协议族中的每次直接在传输数据时的协作关系，以及交互过程，还是引用《图解HTTP》一书上的一张图来解释吧。下图就是这四层协议在数据传输过程中的工作方式。下面这张图还是相当直观的。在发送端是应用层-->链路层这个方向的封包过程，每经过一层都会增加该层的头部。而接收端则是从链路层-->应用层解包的过程，每经过一层则会去掉相应的首部。

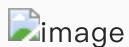


TCP协议的三次握手

在聊HTTP协议之前，我们先简单的聊一下TCP三次握手的过程，在后面的博客中我们将会对TCP和IP协议进行详述，本篇博客就先简单的聊一下做HTTP协议的基础。

TCP协议位于传输层，为了确保传输的可靠性，TCP协议在建立连接时需要三次握手（Three-way handshaking）。下方这个简图就是TCP协议建立连接时三次握手的过程。

- 第一次握手：发送端发送一个带SYN(Synchronize)标志的数据包给接收端，用于询问接收端是否可以接收。如果可以，就进行第二次握手。
- 第二次握手：接收端回传给发送端一个带有SYN/ACK(Acknowledgement)的数据包，给发送端说，我收到你给我发送的SYN标志了，我再给你传一个ACK标志，你能收到吗？如果发送端收到了SYN/ACK这个数据包，就可以确认接收端收到了之前发送的SYN，然后进行第三次握手。
- 第三次握手：发送端会给接收端发送一个带有ACK标志的数据包，告诉接收端我可以收到你给我发送的SYN/ACK标志。接收端如果收到了这个来自客户端的ACK标志，就意味着三次握手完成，连接建立，就可以开始传输数据了。



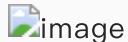
HTTP报文结构

HTTP协议全称是HyperText Transfer Protocol，即超文本传输协议，用户客户端和服务器之前的通信，目前普遍使用版本为HTTP/1.1。协议本质上就是规范，我们之前提到过的“面向接口”编程，其实就是“面向协议”编程。先定义好类的协议，也就是接口，相关类都遵循该协议，这样一来我们就规范了这些类的调用方式。而HTTP协议是规范客户端和服务器之间通信的协议。也就是说所有的客户端或者服务器都遵循了

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

HTTP这个通信协议，那么也就是意味着他们对外传输数据的接口是一直的，就可以在其中间连接上管道，这样一来就可以进行传输了。

这些协议就是接口，有着共同的通信协议，多个端就可以相互通信。采用相同的协议，就是便于个个设备之间进行沟通交流。HTTP协议的作用如下所示。



HTTP协议的作用是用来规范通信内容的，在HTTP协议中可以分为请求报文和响应报文。顾名思义，请求报文是请求方发出的信息，而响应报文是响应端收到请求后响应的内容。接下来我们就来看看请求报文和响应报文的整体结构。

请求报文（Request Message）结构

下方是请求报文的整体结构。请求报文主要分为两大部分，一个是请求头（Request Headers）另一个是请求体（Request Body）。这两者之间由空行分割。在请求头中又分为请求行（Request Line），请求头部字段，通用头部字段和实体头部字段等，这个稍后会详细介绍。下方就是请求报文的结构。



下方这个截图就是请求博客园某个页面时的Request Headers。在请求行中的第一个“GET”是当前请求的方法，稍后会做介绍。中间的就是请求资源的路径，最后一个HTTP/1.1就是当前使用请求协议及其版本。下方这些就是请求头了，稍后会对常用的请求头进行解说。而请求体就是你往服务端传输的数据，比如form表单神马的。



响应报文（Response Message）结构

聊完请求报文，接下来我们来聊聊响应报文，响应报文的结构与请求报文的结构类似，也分为报文头和报文体。下方就是响应报文的结构图。响应头（Response Headers）分为状态行（State Line），响应头部字段，通用头部字段、实体头部字段等。响应头与响应体中间也是有空行进行分割的。



下方截图就是上述请求报文发出后的响应头，响应体就是对于的HTML等前端资源了。在响应头中，第一行就是状态行，“HTTP/1.1”表示使用的HTTP协议的1.1版本，状态200表示响应成功，“OK”则是状态原因短语。常用状态，稍后会详细介绍。



HTTP的请求方法以及响应状态码

上面在介绍请求报文中提到的“GET”就是请求请求方法，而在响应报文中提到的“200”状态码，就是稍后要聊的响应状态码。请求方法和响应状态码在HTTP协议中算是比较重要的内容了。之前我们在使用Perfect框架开发服务器端的时候，曾聊过请求方法中的GET、POST、PUT以及DELETE，并且这四种方法可以结合着REST使用。本部分是以HTTP协议的角度来聊的请求方法，所以与之前会有稍稍的不同。本部分我们就来聊一下HTTP协议的请求方法和响应状态码。

请求方法

接下来我们要聊的请求方法有GET、POST、PUT、HEAD、DELETE、OPTIONS、TRACE、CONNECT。当然上述方法是基于HTTP/1.1的，HTTP/1.0中独有的方法就不说了。

- GET----获取资源

GET方法一般用来从服务器上获取资源的方法。服务器端接到GET请求后，就会明白客户端是要从服务器端获取相应的资源，然后就会根据请求报文中相应的参数，将需要的资源返回给客户端。使用GET方式的请求，传输的参数是拼接在URI上的。

- POST----数据提交

POST方法一般用于表单提交，将客户端的数据塞到请求体中发送给服务器端。

- PUT----上传文件

PUT方法主要用来上传文件，将文件内容塞到请求报文体中，传输给服务器。因为HTTP/1.1的PUT方法自身不带验证机制，所以任何人都可以上传文件，存在安全性，所以上传文件时不推荐使用。但是之前我们在设计接口使用REST标准时，可以使用PUT来做相应内容的更新。

- HEAD----获取响应报文头

响应端收到HEAD请求后，只会返回相应的响应头，不会返回响应体。

- DELETE----删除文件

DELETE用于删除URI指定的资源，与PUT一样，自身也是不带验证机制的，不过在REST标准中可以用来做相应API的删除功能。

- OPTIONS----查询支持的方法

OPTIONS方法是用来查询服务器可对那些请求方法做出相应，返回内容就是响应端所支持的方法。

- TRACE----追踪路径

TRACE方法可追踪请求经过的代理路径，在发送请求时会为Max-Forwards头部字段填入数字，每经过一个代理中转Max-Forwards的值就会减一，直至Max-Forwards为零后，才会返回200。因为该方法易引起XST(Cross-Site Tracing，跨站追踪)攻击，所以不常用呢。

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- CONNECT ---- 要求用隧道协议连接代理

CONNECT方法要求在与代理服务器通信时建立隧道，实现用隧道协议进行TCP通信。主要使用SSL(Secure Sockets Layer, 安全套接层)和TLS(Transport Layer Security, 传输安全层)协议将通信内容进行加密后经网络隧道传输。

响应状态码

聊完请求方法后，接下来我们来聊聊HTTP协议的响应状态码。顾名思义，响应状态码是用来标志HTTP响应状态的，响应状态由响应状态码和响应原因短语构成，当然状态码有很多中，本部分就挑出来常用的状态码进行讨论。下方是响应状态码可以分为的几大类：

- 1xx ---- Informational (信息性状态码)，表示接受的请求正在处理。
- 2xx ---- Success (成功)，表示请求正常处理完毕。
- 3xx ---- Redirection (重定向)，表示要对请求进行重定向操作，当然其中的304除外。
- 4xx ---- Client Error (客户端错误)，服务器无法处理请求。
- 5xx ---- Server Error (服务器错误)，服务器处理请求时出错。

上面是响应状态码的整体分类，接下来介绍一些常用的响应状态码。

- (01)、200 OK：表示服务端正确处理了客户端发送过来的请求。
- (02)、204 No Content：表示服务端正确处理请求，但没有报文实体要返回。
- (03)、206 Partial Content：表示服务端正确处理了客户端的范围请求，并按照请求范围返回该指定范围内的实体内容。
- (04)、301 Moved Permanently：永久性重定向，若之前的URI保存到了书签，则更新书签中的URI。
- (05)、302 Found：临时重定向，该重定向不会变更书签中的内容。
- (06)、303 See Other：临时重定向，与302功能相同，但是303状态码明确表示客户端应当采用GET方法获取资源。
- (07)、304 Not Modified：资源未变更，该状态码与重定向并没有什么关系，当返回该状态码时，告诉客户端请求的资源并没有更新，响应报文体中并不会返回所请求的内容。
- (08)、400 Bad Request：错误请求，表示请求报文中包含语法错误。
- (09)、401 Unauthorized：请求未认证，表示此发送的请求需要客户端进行HTTP认证（稍后会提到）。
- (10)、404 Not Found：找不到相应的资源，表示服务器找不到客户端请求的资源。
- (11)、500 Internal Server Error：服务器内部错误，表示服务器在处理请求时出现了错误，发生了异常。
- (12)、503 Service Unavailable：服务不可用，表示服务器处于停机状态，无法处理客户端发来的请求。

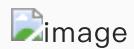
还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋

网络：TCP、IP协议族(二) HTTP报文头解析

本篇博客我们就来详细的聊一下HTTP协议的常用头部字段，当然我们将其分为请求头和响应头进行阐述。下方是报文头每个字段的格式，首先是头部字段的名称，如Accept，冒号后方紧跟的是该字段名所对应的值，每个值之间有逗号分隔。如果该值需要优先级，那么在值的后方跟上优先级q=0.8(q的值由0~1，优先级从低到高)。值与优先级中间由分号相隔。

头部字段名：值1, 值2;q=0.8

下方就是截取的网络请求中Request Headers的部分内容。红框中的Accept-Language就是头部字段名，冒号后边就是该字段相应的值了。如下所示：

image

HTTP头部字段可以分为通用头部字段，请求头部字段，响应头部字段以及实体头部字段，下方会给出详细的介绍。

通用头部字段（General Header Fields）

该字段在请求头和响应头都会使用到，下方是常用的通用头部字段：

Cache-Control

用来操作缓存的工作机制，下方截图响应头中的的Cache-Control的参数为private和max-age=10。private缓存是私有的，仅像特定用户提供相应的缓存信息。如果是public，那么就意味着可向任意方提供相应的缓存信息。max-age = 10表示缓存有效期为10秒。从下方的Expires(过期时间)和Last-Modified(最后修改时间)就可以看出，这两者之间的差值正好是10秒。

该字段还可以对应其他的参数：

- no-cache：如果是客户端的话，说明客户端不会接收缓存过的响应，要请求最新的内容。而服务器端则表示缓存服务器不能对相应的资源进行缓存。
- no-store：表示缓存不能在本地存储。
- max-age：该参数后方会被赋值上相应的秒数，在请求头中表示如果缓存时间没有超过这个值就返回给我。而在响应头中时，则表示资源在缓存服务器中缓存的最大时间。

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

- `only-if-cached`: 表示客户端仅仅请求缓存服务器上的内容，如果缓存服务器上没有请求的内容，那么返回504 Gateway Timeout。
- `must-revalidate`: 表示缓存服务器在返回资源时，必须向资源服务器确认其缓存的有效性。
- `no-transform`: 无论请求还是响应，都不能在传输的过程中改变报文体的媒体类型。



Connection

该字段可以控制不转发给代理服务器的首部字段以及管理持久连接，下方这个响应报文头中的`Connection`就是用来管理持久连接的，其参数为`keep-alive`，就是保持持久连接的意思。可以使用`close`参数将其关闭。



Transfer-Encoding

该字段表示报文在传输过程中采用的编码方式，在HTTP/1.1的报文传输过程中仅对分块编码有效。下方这个截图就是`Transfer-Encoding`在Response Header中的使用，后边跟的`chunked`(分块)的参数，说明报文是分块进行传输的。



Via

该字段是为了追踪请求和响应报文的传输路径，报文经过代理或者网关时会在`Via`字段添加该服务器的信息，然后再进行转发。

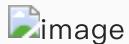


请求头部字段（Request Header Fields）

顾名思义，请求头部字段当然是在请求头中才使用的字段。该字段用于补充请求的附加信息，客户端信息等。接下来将给出常用而且比较重要的几个请求头部字段。

Accept

该字段可通知服务器用户代理能够处理的媒体类型以及该媒体类型对应的优先级。媒体类型可使用“type/subtype”这种形式来指定，分号后边紧跟着的是该类型的优先级。如下所示。



Accept-Encoding

该字段用来告知服务器，客户端这边可支持的内容编码以及相应内容编码的优先级，下方就是Accept-Encoding的用法。gzip表示由文件压缩程序gzip(GNU zip)生成的编码格式。compress表示UNIX文件压缩程序compress生成的编码格式。deflate表示组合使用 zlib 格式以及有 deflate 压缩算法生成的编码格式。identity表示不执行压缩或者使用一致的默认编码格式。



Accept-Language

该字段用来告知服务器，客户端可处理的自然语言集，以及对应语言集的优先级。以下方的截图为例，Accept-Language后方跟了三个属性，分别是“zh-CN”，“zh;q=0.8”，“en;q=0.6”。也就是说客户端可处理三种自然语言集，zh-CN，其优先级是1（最高）。第二种是zh，其优先级是0.8，次之。第三个是en，优先级为0.6，优先级在三者之间最低。



Authorization

用来告知服务器用户端的认证信息，下方就是连接公司内部SVN系统时需要认证时的请求头部信息。



如果你没有填写认证信息的话，那么就会返回401 Unauthorized。如下所示：



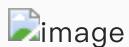
If-Match 与 If-None-Match

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

上面这两个请求头部字段都是带有逻辑判断的，从上面的英文我们不难看出两者恰好相反。两者后方都跟着串字符串，如If-Match "xcsldjh49773hce"，后边这个字符的匹配对象是ETag(稍后会介绍)。If-Match的请求是如果后方的字符串与ETag相等则服务器端进行请求，否则不进行处理。If-None-Match是If-Match的非操作，同样是匹配ETag，如果Etag没有匹配成功就处理请求，否则不处理。

If-Modified-Since与If-Unmodified-Since

If-Modified-Since也是带有逻辑判断的请求头部字段，该字段后方跟的是一个日期，意思是在该日期后发生了资源更新，那么服务器就会处理该请求。If-Unmodified-Since就是 If-Modified-Since的非操作。



If-Range

If-Range字段后方也是跟的Etag，该字段要结合着Range字段进行使用。其所代表的意思就是如果Etag匹配成功，请求的内容就按照Range字段所规定的范围进行返回，否则返回全部的内容。用法如下所示：

```
If-Range: "etag_code"  
Range: bytes=1000-5000
```

公众号：方志朋

Referer

其实Referer是一个错误的拼写，但是一直在使用。正确的英文单词应该是Referrer(此处可翻译为：来历、来路)。Referer字段后方跟的是一个URI，该URI就是发起请求的URI，具体如下所示：



User-Agent

该字段会将请求方的浏览器和用户代理名称等信息传达给服务器。下方就是从我当前笔记本的Chrome浏览器请求网络时的User-Agent信息。



响应头部字段（Request Header Fields）

聊完请求报文头部字段后，我们接下来来聊一下响应报文头部字段。响应头是由Server向Client返回响应报文中使用的头部信息。用户补充响应的附加信息、服务信息等。下方是几个常见响应头部字段。

Accept-Ranges

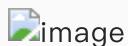
该字段用来告知客户端服务器那边是否支持范围请求（请求部分内容，请求头中使用Range字段）。

Accept-Ranges的值为bytes时，就说明服务器支持范围请求，为none时，说明服务器不支持客户端的范围请求。下方是博客园的页面的加载，从下方可以看出是支持范围请求的，如下所示：



Age

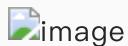
该字段告知客户端，源服务器在多久前创建了该响应。



Etag

Etag是服务器当前请求的服务器资源（图片，HTML页面等）所对应的一个独有的字符串。不同资源间的Etag是不同的，当资源更新时Etag也会进行更新。

所以结合着请求头中的If-Match等逻辑请求头，可以判断当前Client端已经加载的资源在服务器端是否已经更新了。当初次请求一个资源，如图片时，我们可以将其Etag进行保存，在此请求时，可放在If-None-Match后方，进行资源更新。如果服务器资源并未修改，就不对该请求做出响应。下方就是网页中某张图片对应着的Etag，如下所示。

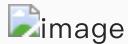


Location

Location字段一般与重定向结合着使用。下方是我访问“www.baidu.com/hello”这个连接的响应报文。因为服务器上并没有/hello这个资源路径，所以给我重定向了error.html页面，这个重定向的URL就存储在

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

Location字段中，如下所示：



Server

该响应字段表明了服务器端使用的服务器型号，下方是博客园某张图片的响应头，使用的Web服务器是Tengine，Tengine是淘宝发起的Web服务器项目，是基于Nginx的，关于Tengine的相关内容，请自行Google吧。



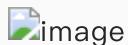
Vary

Vary可对缓存进行控制，通过该字段，源服务器会向代理服务器传达关于本地缓存使用方法的命令。下方就是Vary的使用，Vary后方的参数是Accept-Encoding。其意思是返回的缓存要以Accept-Encoding为准。当请求的Accept-Encoding的参数与缓存内容的Accept-Encoding参数一致时就返回缓存内容，否则就请求源服务器。



WWW-Authenticate

该字段用于HTTP的访问认证，在状态码401 Unauthorized中肯定带有此字段，该字段用来指定客户端的认证方案（Basic或者Digest）。参数realm的字符串是为了辨别请求URL指定资源所受到的保护策略。如下所示：



实体头部字段（Content Header Fields）

接下来我们就来聊聊常见的实体头部字段，实体头部字段是报文实体所使用的头部，用来补充与报文实体相关的信息。

Allow

该字段用于服务器通知客户端服务器这边所支持的所有请求方法（GET、POST等）。如果服务器找不到客户端请求中所提到的方法的话，就会返回405 Method Not Allowed，于此同时还会把所有能支持的HTTP方法写入到首部字段Allow后返回。

Allow : GET, POST, HEAD, PUT, DELETE

Content-Encoding

该字段用来说明报文实体的编码方式，下方这段报文头中的Content-Encoding的参数为gzip，说明是使用gzip对报文实体进行压缩的。

image

Content-Language

该字段表示报文实体使用的自然语言，使用方式如下所示：

Content-Language: zh-CN

公众号：方志朋

Content-Length

顾名思义，该字段用来指定报文实体的字节长度，如下所示：

image

Content-MD5

该字段中存储的是报文实体进行MD5加密然后再使用Base64进行编码的字符串。客户端收到响应报文后，可以对报文实体进行MD5加密，然后再对其进行Base64编码，然后与Content-MD5中的字符串进行比较来确定报文是否进行修改，可以说这是一个简单的验签功能。但是此方法并不能确定报文是否被修改了，因为Content-MD5这个值也有可能被篡改。

Cookie相关的头部字段

还有超过100本优质java高清电子书，添加微信cxymysq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

因为HTTP协议本身是无状态的，在Web站点中使用Cookie来管理服务器与客户端之间的状态。解析来我就来介绍一下Cookie相关的头部字段。

Set-Cookie

响应报文中会使用到该字段。当服务器准备开始管理客户端的状态时，会事先告知其各种信息。下方字段是登录知乎时所返回的所要设置的Cookie信息。接下来我们就要对这串Cookie信息进行解析。

- 键值对：在Set-Cookie字段中，“z_co=Mi4.....”这就是要存入Cookie中的信息，当然可以是多个键值对，中间使用逗号进行分割即可。
- Domain：然后是Domain属性，由下方不难看出，Domain中存储的就是Cookie适用对象的域名，若不指定Domain的值，那么默认就是创建Cookie的服务器的域名。
- expire：该字段属性的值是一个时间，也就是Cookie的有效期，若不指定该属性的值，默认就是当前会话有效，关闭浏览器Cookie即失效。
- httponly：设置该属性的目的是让JavaScript脚本无法获取Cookie，其主要目的是防止跨站脚本攻击对Cookie信息的窃取。
- path：用于限制指定Cookie的发送范围的文件目录。
- Secure：仅在HTTPS安全通信时才会发送Cookie。



Cookie

请求报文头中会使用该字段，用于将本地存储的Cookie信息发送给服务端。下方就是知乎上每次请求文章所带有的Cookie信息，当然下方只是部分信息，但是我们还是从中可以找到之前我们存储的“z_co=Mi4.....”这个键值对的。



其他比较常见而且比较简单的头部字段就不做过多赘述了，今天博客就先到这儿吧。

作者：方志朋，一线大厂架构师，

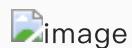
来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：

公众号：方志朋

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！



程序员理财，请关注：



公众号：方志朋

网络：TCP、IP协议族(三) 数字签名与HTTPS详解

前面几篇博客聊了HTTP的相关东西，今天就来聊一聊HTTPS的东西。因为HTTP协议本身存在着明文传输、不能很好的验证通信方的身份和无法验证报文的完整性等一些安全方面的确点，所以才有了HTTPS的缺陷。HTTPS确切的说不是一种协议，而是HTTP + SSL (TSL)的结合体。HTTP报文经过SSL层加密后交付给TCP层进行传输。SSL(安全套接层)主要采取的是RSA (非对称加密) 与AES (对称加密) 结合的加密方式。先通过RSA交互AES的密钥，然后通过AES进行报文加密和解密。本篇博客主要聊的就是HTTPS具体的工作过程。

RSA与AES简述

在本篇博客的第一部分呢，先聊一下RAS与AES这两个加密策略，如果你在公司做过支付相关的东西，对数据传输的安全性要求比较高，这时候就得采取一些加密措施将传输的报文进行加密，必要时再进行MD5验签。当然本部分聊的RAS与AES是比较简洁的，关于这两者具体的内容，请自行Google吧。因为HTTPS在传输的过程中使用到了RSA与AES加密算法，所以在聊HTTP+SSL之前呢，我们先简单的聊一下AES与RSA。

Advanced Encryption Standard (AES: 高级加密标准)

AES，全称：Advanced Encryption Standard----高级加密标准。该加密算法有一个密钥，该密钥可以用来加密，也可以用来解密，所以AES是对称加密算法。下方这个就是AES加密和解密的过程。Client端与Server端有一个共同的Key，这个Key是用来加密和解密的。如果报文在传输的过程中被窃取了，没有这个key，对加密的内容进行破解是非常困难的，当然窃取者如果有key的话，也是可以轻而易举的解密的。所以在AES中，key是关键。这也就相当于你们家的门钥匙，谁拿到钥匙后都可以打开你们家的门。即使门锁再结实，再安全，在钥匙面前也是不行呢。

所以对于AES加密策略来说这个Key的保密措施要做足一些，如果之后有时间的话可以分享一些具体的AES加密策略。比如每次加密的Key都是从一个密码本中动态生成的，而这个密码本服务端和客户端都有同一本，每次传输的是一些参数。这些参数在经过一些算法的映射，从密码本中取出相应的key用来解密。这样一来，就相当于给AES加了一层防盗门，加大了破解的难度。这样做的好处是每次加密的key都是不同的，而且需要密码本以及映射算法的支持。



RSA 公钥加密算法

RAS这个名字，就是该算法三位发明者的名字的首字母的组合。RAS是非对称加密，其在加密和解密的过程中，需要两个Key，一个公钥（public key），一个是私钥（private key）。公钥负责加密，而私钥负责解密。从名字就可以看出，公钥是可以开放出去的，任何人都可以持有公钥进行加密。而私钥必须得进行保护，因为是用来解密的。

这样一来，加密和解密就可以用不同的钥匙来处理。对于加密方来说，即使你可以对报文进行加密，如果没有私有的话也是不可以对你加密的内容进行解密的。这就相当于一个盒子，盒子上有把锁。你可以把东西放进去，然后再锁上盒子。但是如果你没有钥匙的话，也是打不开这把锁的。

下方这个简图就是服务端单向验证的RAS非对称加密算法，Client内置了一个公钥，该公钥与Server端的私钥是配对的，所以Client端可以使用这个内置的Public key加密，而服务端就可以使用这个private key进行解密。目前最常用的是服务端单向认证机制。



CA证书

如果你自己通过RAS算法生成了一个私钥和公钥，在公钥发送给客户端的过程中有可能被篡改成其他的公钥，而客户端在没有其他措施的保护下是不知道该公钥是否就是服务器那边的私钥对应的公钥的。这种自己做的RAS的公钥和私钥有可能在公钥分发的过程中被篡改。下方就是Client从Server端获取公钥时被中间者篡改了，将public换成了自己的伪public key，同样这个中间者持有这个伪public key所对应的伪private key。如果客户端使用的伪public key进行加密传输的话，那么中间者是可以使用自己的private key进行解密的。

举个例子来类比一下这个问题。

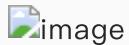
假设你在古代，你出门在外，妻子在家养子。你们家有个箱子，箱子上有把锁，这就是你和你妻子互通的工具。你媳妇儿负责往箱子里放东西，然后上锁。你有把独特的钥匙，你负责开锁，取东西。可是你再将箱子给镖局托运的过程中，被镖局的“小黑”掉包了，箱子的外表一致，锁看起来也一样，可是已经不是你的箱子了。因为路途遥远，古代又没有什么iPhone啥的，你媳妇没办法来辨别该箱子是否是原装的。然后就将一些东西放在了箱子里边，然后上锁交给了镖局的“小黑”。

因为“小黑”掉包的箱子，所以小黑有箱子的钥匙呢，然后就可以打开这个箱子，取东西了。原来的箱子又在小黑那，小黑就可以往原来的箱子里边随便往箱子放点没有价值的东西给你就行了。当你发现箱子里的东西不是你想要的时候，完了，小黑从镖局辞职了，找不到人了。找镖局的人讨说法，可是镖局的人说“小

还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

“黑”是镖局的临时工，这个责任镖局说了，我们不能担。鉴于你无权无势，这事儿也就此罢了。（故事纯属虚构，如有雷同纯属巧合）

关于更多骗子的故事请移步网络剧《毛骗》一二三季。



为了防止“小黑”再次作案，所以颁布一个公正机构来证明你媳妇收到的箱子就是你发出的箱子。在RAS加密中也有一个第三方机构来充当这个角色，负责证明客户端收到的证书就是你发送的证书，中间没有被篡改。这个中间认证机构，就是数组证书认证机构，其颁发的证书也就是我们常说的CA证书（CA，Certificate Authority）。

下面我们就来详细的叙述一下证书签名，证书分发以及证书验证的整个过程。

- 1、服务端人员使用RSA算法生成两个密钥，一个用来加密一个用来解密。将负责加密的那个密钥公布出去，所以我们称之为公钥（Public Key），而用来解密的那个密钥，不能对外公布，只有服务端持有，所以我们称之为私钥（Private Key）。服务端在将Public Key进行分发证书之前需要向CA机构申请给将要分发的公钥进行数字签名。（服务器公钥负责加密，服务器私钥负责解密）
- 2、生成数字签名公钥证书：对于CA机构来说，其也有两个密钥，我们暂且称之为CA私钥和CA公钥。CA机构将服务端的Public Key作为输入参数将其转换为一个特有的Hash值。然后使用CA私钥将这个Hash值进行加密处理，并与服务端的Public Key绑定在一起，生成数字签名证书。其实数字签名证书的本质就是服务端的公钥+CA私钥加密的Hash值。（CA私钥负责签名，CA公钥负责验证）
- 3、服务器获取到这个已经含有数字签名并带有公钥的证书，将该证书发送给客户端。当客户端收到该公钥数字证书后，会验证其有效性。大部分客户端都会预装CA机构的公钥，也就是CA公钥。客户端使用CA公钥对数字证书上的签名进行验证，这个验证的过程就是使用CA公钥对CA私钥加密的内容进行解密，将解密后的内容与服务端的Public Key所生成的Hash值进行匹配，如果匹配成功，则说明该证书就是相应的服务端发过来的。否则就是非法证书。
- 4、验证完服务端公钥的合法性后，就可以使用该公钥进行加密通信了。



下方这个截图就是苹果的根证书的一些信息，从下方可以看出，CA证书内容中包括加密算法，公共密钥以及数字签名。



下方就是公钥以及数字签名的具体内容，当对下方公共密钥进行验证时，需要使用内置的CA公钥将数字签名进行解密。然后将解密后的内容，与公钥生成的Hash值进行比较，如果匹配成功，那么该证书就是CA机构颁布的合法证书。



HTTPS安全通信机制的建立

上面我们聊完AES与RSA加密策略，然后又聊了带有数字签名的公共密钥。上面这两部分内容都是为HTTPS做铺垫的，接下来就看一看HTTP+SSL是如何进行数据传输的。

HTTPS简介

在开头的部分也说了，HTTPS不是一个新的通信协议，而是HTTP与SSL（或TSL）的组合。SSL--安全套接层(Secure Socket Layer), TSL (Transport Layer Security 安全传输层) 是以SSL为原型开发的协议，IETF以SSL3.0为基准后又制定了TLS1.0、TLS1.1和TLS1.2，当前主流版本为SSL3.0与TLS1.0。

HTTPS就是在HTTP与TCP层中间添加了一个SSL层。因为HTTPS被HTTP多了这层加密的流程，所以HTTPS的速度要比HTTP慢的多。



HTTPS的通信过程

SSL的加密过程是RSA与AES混合进行的。简单概括一下，就是通过RSA加密方式来交换AES加解密的密钥，然后使用AES加密的方式来传输报文。下方是SSL建立连接以及传输数据的图解。在下图中大体可以分为四步：

- 第一步：有客户端发起的第一次握手，此次握手过程的主要目的是从服务端获取数字签名证书，服务端在发送数字签名证书之前要先确认客户端的SSL版本、加密算法等信息。
- 第二步：完成第一次握手后，接着进行第二次握手。第二次握手是在客户端收到证书后发起的，主要目的是将AES加解密使用的Key（Pre-master secret）发送给服务端。当然这个AES_KEY是使用第一次握手获取的公钥进行加密的。客户端收到这个使用公钥加密后的AES_KEY，使用服务端的私钥进行解密。这样客户端和服务端经过二次握手后都持有了AES加解密的KEY。
- 第三步：当Client与Server端都持有AES_KEY后，就可以对HTTP报文进行加解密了。
- END: 最后就是断开连接了。具体如下图所示：



还有超过100本优质java高清电子书，添加微信cxymsq2获取。另外作者实盘投资公众号：价投之道，欢迎关注！

作者：方志朋，一线大厂架构师，

来源：<https://fangzhipeng.com>，欢迎关注作者的公众号，扫一扫：



程序员理财，请关注：



公众号：方志朋