

## 31 | 答疑课堂：模块五思考题集锦

2019-08-03 刘超

Java性能调优实战

[进入课程 >](#)



讲述：李良

时长 11:15 大小 10.30M



你好，我是刘超。

模块五我们都在讨论设计模式，在我看来，设计模式不仅可以优化我们的代码结构，使代码可扩展性、可读性强，同时也起到了优化系统性能的作用，这是我设置这个模块的初衷。特别是在一些高并发场景中，线程协作相关的设计模式可以大大提高程序的运行性能。

那么截至本周，有关设计模式的内容就结束了，不知你有没有发现这个模块的思考题都比较发散，很多同学也在留言区中写出了很多硬核信息，促进了技术交流。这一讲的答疑课堂我就来为你总结下课后思考题，希望我的答案能让你有新的收获。

[第 26 讲](#)

## 除了以上那些实现单例的方式，你还知道其它实现方式吗？

在[第 9 讲](#)中，我曾提到过一个单例序列化问题，其答案就是使用枚举来实现单例，这样可以避免 Java 序列化破坏一个类的单例。

枚举生来就是单例，枚举类的域（field）其实是相应的 enum 类型的一个实例对象，因为在 Java 中枚举是一种语法糖，所以在编译后，枚举类中的枚举域会被声明为 static 属性。


在[第 26 讲](#)中，我已经详细解释了 JVM 是如何保证 static 成员变量只被实例化一次的，我们不妨再来回顾下。使用了 static 修饰的成员变量，会在类初始化的过程中被收集进类构造器即 <clinit> 方法中，在多线程场景下，JVM 会保证只有一个线程能执行该类的 <clinit> 方法，其它线程将会被阻塞等待。等到唯一的一次 <clinit> 方法执行完成，其它线程将不会再执行 <clinit> 方法，转而执行自己的代码。也就是说，static 修饰了成员变量，在多线程的情况下能保证只实例化一次。

我们可以通过代码简单了解下使用枚举实现的饿汉单例模式：

 复制代码

```
1 // 饿汉模式 枚举实现
2 public enum Singleton {
3     INSTANCE;// 不实例化
4     public List<String> list = null;// list 属性
5
6     private Singleton() { // 构造函数
7         list = new ArrayList<String>();
8     }
9     public static Singleton getInstance(){
10         return INSTANCE;// 返回已存在的对象
11     }
12 }
```

该方式实现的单例没有实现懒加载功能，那如果我们要使用到懒加载功能呢？此时，我们就可以基于内部类来实现：

 复制代码

```
1 // 懒汉模式 枚举实现
2 public class Singleton {
3     INSTANCE;// 不实例化
```

```
4     public List<String> list = null;// list 属性
5
6     private Singleton(){// 构造函数
7         list = new ArrayList<String>();
8     }
9     // 使用枚举作为内部类
10    private enum EnumSingleton {
11        INSTANCE;// 不实例化
12        private Singleton instance = null;
13
14        private EnumSingleton(){// 构造函数
15            instance = new Singleton();
16        }
17        public static Singleton getSingleton(){
18            return instance;// 返回已存在的对象
19        }
20    }
21
22    public static Singleton getInstance(){
23        return EnumSingleton.INSTANCE.getSingleton();// 返回已存在的对象
24    }
25 }
```

## 第 27 讲

**上一讲的单例模式和这一讲的享元模式都是为了避免重复创建对象，你知道这两者的区别在哪儿吗？**

首先，这两种设计模式的实现方式是不同的。我们使用单例模式是避免每次调用一个类实例时，都要重复实例化该实例，目的是在类本身获取实例化对象的唯一性；而享元模式则是通过一个共享容器来实现一系列对象的共享。

其次，两者在使用场景上也是有区别的。单例模式更多的是强调减少实例化提升性能，因此它一般是使用在一些需要频繁创建和销毁实例化对象，或创建和销毁实例化对象非常消耗资源的类中。

例如，连接池和线程池中的连接就是使用单例模式实现的，数据库操作是非常频繁的，每次操作都需要创建和销毁连接，如果使用单例，可以节省不断新建和关闭数据库连接所引起的性能消耗。而享元模式更多的是强调共享相同对象或对象属性，以此节约内存使用空间。

除了区别，这两种设计模式也有共性，单例模式可以避免重复创建对象，节约内存空间，享元模式也可以避免一个类的重复实例化。总之，两者很相似，但侧重点不一样，假如碰到一些要在两种设计模式中做选择的场景，我们就可以根据侧重点来选择。

## 第 28 讲


除了以上这些多线程的设计模式（线程上下文设计模式、Thread-Per-Message 设计模式、Worker-Thread 设计模式），平时你还使用过其它的设计模式来优化多线程业务吗？

在这一讲的留言区，undifined 同学问到了，如果我们使用 Worker-Thread 设计模式，worker 线程如果是异步请求处理，当我们监听到有请求进来之后，将任务交给工作线程，怎么拿到返回结果，并返回给主线程呢？

回答这个问题的过程中就会用到一些别的设计模式，可以一起看看。


如果要获取到异步线程的执行结果，我们可以使用 Future 设计模式来解决这个问题。假设我们有一个任务，需要一台机器执行，但是该任务需要一个工人分配给机器执行，当机器执行完成之后，需要通知工人任务的具体完成结果。这个时候我们就可以设计一个 Future 模式来实现这个业务。

首先，我们申明一个任务接口，主要提供给任务设计：

 复制代码

```
1 public interface Task<T, P> {  
2     T doTask(P param); // 完成任务  
3 }
```


其次，我们申明一个提交任务接口类，TaskService 主要用于提交任务，提交任务可以分为需要返回结果和不需要返回结果两种：

 复制代码

```
1 public interface TaskService<T, P> {  
2     Future<?> submit(Runnable runnable); // 提交任务，不返回结果  
3     Future<?> submit(Task<T, P> task, P param); // 提交任务，并返回结果
```

```
4 }
```

接着，我们再申明一个查询执行结果的接口类，用于提交任务之后，在主线程中查询执行结果：

 复制代码

```
1 public interface Future<T> {
2
3     T get(); // 获取返回结果
4     boolean done(); // 判断是否完成
5 }
```

然后，我们先实现这个任务接口类，当需要返回结果时，我们通过调用获取结果类的 finish 方法将结果传回给查询执行结果类：

 复制代码

```
1 public class TaskServiceImpl<T, P> implements TaskService<T, P> {
2
3     /**
4      * 提交任务实现方法，不需要返回执行结果
5      */
6     @Override
7     public Future<?> submit(Runnable runnable) {
8         final FutureTask<Void> future = new FutureTask<Void>();
9         new Thread(() -> {
10             runnable.run();
11         }, Thread.currentThread().getName()).start();
12         return future;
13     }
14
15     /**
16      * 提交任务实现方法，需要返回执行结果
17      */
18     @Override
19     public Future<?> submit(Task<T, P> task, P param) {
20         final FutureTask<T> future = new FutureTask<T>();
21         new Thread(() -> {
22             T result = task.doTask(param);
23             future.finish(result);
24         }, Thread.currentThread().getName()).start();
25         return future;
26     }
```

最后，我们再实现这个查询执行结果接口类，FutureTask 中，get 和 finish 方法利用了线程间的通信 wait 和 notifyAll 实现了线程的阻塞和唤醒。当任务没有完成之前通过 get 方法获取结果，主线程将会进入阻塞状态，直到任务完成，再由任务线程调用 finish 方法将结果传回给主线程，并唤醒该阻塞线程：

 复制代码


```

1 public class FutureTask<T> implements Future<T> {
2
3     private T result;
4     private boolean isDone = false;
5     private final Object LOCK = new Object();
6
7     @Override
8     public T get() {
9         synchronized (LOCK) {
10             while (!isDone) {
11                 try {
12                     LOCK.wait(); // 阻塞等待
13                 } catch (InterruptedException e) {
14                     // TODO Auto-generated catch block
15                     e.printStackTrace();
16                 }
17             }
18         }
19         return result;
20     }
21
22     /**
23      * 获取到结果，并唤醒阻塞线程
24      * @param result
25      */
26     public void finish(T result) {
27         synchronized (LOCK) {
28             if (isDone) {
29                 return;
30             }
31             this.result = result;
32             this.isDone = true;
33             LOCK.notifyAll();
34         }
35     }
36
37     @Override
38     public boolean done() {

```


```
39         return isDone;
40     }
41 }
```

我们可以实现一个造车任务，然后用任务提交类提交该造车任务：

 复制代码

```
1 public class MakeCarTask<T, P> implements Task<T, P> {
2
3     @SuppressWarnings("unchecked")
4     @Override
5     public T doTask(P param) {
6
7         String car = param + " is created success";
8
9         try {
10             Thread.sleep(2000);
11         } catch (InterruptedException e) {
12             // TODO Auto-generated catch block
13             e.printStackTrace();
14         }
15
16         return (T) car;
17     }
18 }
```

最后运行该任务：


 复制代码

```
1 public class App {
2
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5
6         TaskServiceImpl<String, String> taskService = new TaskServiceImpl<String, String>();
7         MakeCarTask<String, String> task = new MakeCarTask<String, String>();
8
9         Future<?> future = taskService.submit(task, "car1");// 提交任务
10        String result = (String) future.get();// 获取结果
11
12        System.out.print(result);
13    }
14 }
```





运行结果：

 复制代码

```
1 car1 is created success
```



从 JDK1.5 起，Java 就提供了一个 Future 类，它可以通过 get() 方法阻塞等待获取异步执行的返回结果，然而这种方式在性能方面会比较糟糕。在 JDK1.8 中，Java 提供了 CompletableFuture 类，它是基于异步函数式编程。相对阻塞式等待返回结果，CompletableFuture 可以通过回调的方式来处理计算结果，所以实现了异步非阻塞，从性能上来说它更加优越了。

在 Dubbo2.7.0 版本中，Dubbo 也是基于 CompletableFuture 实现了异步通信，基于回调方式实现了异步非阻塞通信，操作非常简单方便。

## 第 29 讲

**我们可以用生产者消费者模式来实现瞬时高并发的流量削峰，然而这样做虽然缓解了消费方的压力，但生产方则会因为瞬时高并发，而发生大量线程阻塞。面对这样的情况，你知道有什么方式可以优化线程阻塞所带来的性能问题吗？**

无论我们的程序优化得有多么出色，只要并发上来，依然会出现瓶颈。虽然生产者消费者模式可以帮我们实现流量削峰，但是当并发量上来之后，依然有可能导致生产方大量线程阻塞等待，引起上下文切换，增加系统性能开销。这时，我们可以考虑在接入层做限流。

限流的实现方式有很多，例如，使用线程池、使用 Guava 的 RateLimiter 等。但归根结底，它们都是基于这两种限流算法来实现的：漏桶算法和令牌桶算法。

漏桶算法是基于一个漏桶来实现的，我们的请求如果要进入到业务层，必须经过漏桶，漏桶出口的请求速率是均衡的，当入口的请求量比较大的时候，如果漏桶已经满了，请求将会溢



出（被拒绝），这样我们就可以保证从漏桶出来的请求量永远是均衡的，不会因为入口的请求量突然增大，致使进入业务层的并发量过大而导致系统崩溃。

令牌桶算法是指系统会以一个恒定的速度在一个桶中放入令牌，一个请求如果要进来，它需要拿到一个令牌才能进入到业务层，当桶里没有令牌可以取时，则请求会被拒绝。Google 的 Guava 包中的 RateLimiter 就是基于令牌桶算法实现的。

我们可以发现，漏桶算法可以通过限制容量池大小来控制流量，而令牌算法则可以通过限制发放令牌的速率来控制流量。

## 第 30 讲

**责任链模式、策略模式与装饰器模式有很多相似之处。在平时，这些设计模式除了在业务中被用到之外，在架构设计中也经常被用到，你是否在源码中见过这几种设计模式的使用场景呢？欢迎你与大家分享。**

责任链模式经常被用在一个处理需要经历多个事件处理的场景。为了避免一个处理跟多个事件耦合在一起，该模式会将多个事件连成一条链，通过这条链路将每个事件的处理结果传递给下一个处理事件。责任链模式由两个主要实现类组成：抽象处理类和具体处理类。

另外，很多开源框架也用到了责任链模式，例如 Dubbo 中的 Filter 就是基于该模式实现的。而 Dubbo 的许多功能都是通过 Filter 扩展实现的，比如缓存、日志、监控、安全、telnet 以及 RPC 本身，责任链中的每个节点实现了 Filter 接口，然后由 ProtocolFilterWrapper 将所有的 Filter 串连起来。

策略模式与装饰器模式则更为相似，策略模式主要由一个策略基类、具体策略类以及一个工厂环境类组成，与装饰器模式不同的是，策略模式是指某个对象在不同的场景中，选择的实现策略不一样。例如，同样是价格策略，在一些场景中，我们就可以使用策略模式实现。基于红包的促销活动商品，只能使用红包策略，而基于折扣券的促销活动商品，也只能使用折扣券。

以上就是模块五所有思考题的答案，现在不妨和你的答案结合一下，看看是否有新的收获呢？如果你还有其它问题，请在留言区中提出，我会一一解答。最后欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他加入讨论。

# Java 性能调优实战

覆盖 80% 以上 Java 应用调优场景

刘超

金山软件西山居技术经理



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | 装饰器模式：如何优化电商系统中复杂的商品价格策略？

## 精选留言 (6)

写留言



发条橙子。

2019-08-03

老师，对于单例模式那块还是有些不解，希望老师解答：😊

在类初始化时 会将 static成员变量放到 <clinit> 方法中，在类加载准备阶段负责为其创建内存，到了初始化阶段执行<clinit>方法 进行赋值。

...

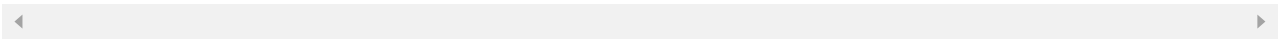
展开

作者回复: 1、外部类初次加载，会初始化静态变量、静态代码块、静态方法，但不会加载内部类和静态内部类。

如果我们使用内部类实现，不会在初始化类时加载内部类，只有在我们第一次调用 EnumSingleton.INSTANCE，才会加载 EnumSingleton 类，而只有在加载EnumSingleton类之后，才会实例化创建INSTANCE对象，由于INSTANCE是一个静态成员变量，所以在初始化时由

<clinit> 方法保证线程安全。

2、元空间存放的是类元素，而静态常量池存放的是类的字面量和字符引用，运行时常量池存放的是直接引用。这里的静态常量池也是很多人说的字符串常量池。



4



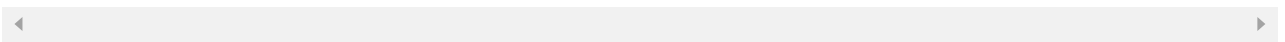
疯狂咸鱼

2019-08-03

老师会讲数据库调优么？

展开 ∨

作者回复: 下一讲开始讲到数据库调优了，欢迎一起学习成长。



2



Jxin

2019-08-03

打卡，转眼已过大半，感谢老师一致以来的分享。棒棒的。



失火的夏天

2019-08-03

老师我问一下，内部类和静态内部类的区别是什么，哪里有不同，我看jdk里map的node，还有list的node都是静态的。之前有看到设置成静态是为了防止内存泄露，但是没有想明白是为什么



-W.LI-

2019-08-03

老师好!感觉自己写代码很多时候都是面向过程的思维。纯纯的CRUD程序员平时99%的工作都是增删改查+处理业务逻辑。如何培养自己的面向对象思维?老师有好的建议书籍推荐么?万分感谢，我感觉这个应该是共性的问题。

展开 ∨



-W.LI-

2019-08-03

很干货谢谢老师。future那个感觉很不错，之前都是用的future+线程池

+countdownLanch。实现回调的，调用get方法的时候确实会阻塞等待最后一个任务完成为止，如果需要对一批任务做组合处理的化只能这样了吧。如果不需要聚合处理就可以使用CompletableFuture进行优化，回头看下CompletableFuture。之前好像看过future类的源码，没记错的话和老师的代码实现一样，依稀记得只是包了一层，future的run方法...  
展开 ∨

