1 期 Java 架构师训练营第一阶段课程答疑答案

一、电商系统在双 11 大促活动下的架构体系：

普通微服务系统的架构体系如何减少项目崩溃

项目崩溃可能有多方面的因素，硬件层面可能有网络，端口不可达，或者内存吃爆，中间件服务 down 掉等
而与微服务体系的关系不是很大，这里的微服务造成项目崩溃，可能是某个服务 down 掉，这就要求部署层面做到高可用，既然是微服务，那做横向扩展和服务负载相对容易，提升了可靠度。
另一方面，项目崩溃可能是微服务之间搭配不当，比如网关超时熔断时间是 5s，而里面某个服务没有做好优化达到了 10s，这就造成整个服务链路不可用。外部看来就是项目崩溃了。这就需要优化服务响应时间，或者实当放开最外层的熔断策略。同时对微服务单元做好 fallback 降级

二、从架构层面看设计模式

1、平时都用不到，怎么才能 用设计模式来写程序 有什么好的办法 可以提升设计方面的能力？

1) 用设计模式写程序，首先要明白不同设计模式的不同作用，可以根据设计模式分类去指定学习。
创建型模式：用于描述“怎样创建对象”，对应的设计模式有单例、原型、工厂方法、抽象工厂、建造者 5 类。
结构型模式：用于描述如何将类或对象按某种布局组成更大的结构，对应设计模式有代理、适配器、桥接、装饰、外观、享元、组合。
行为型模式：用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。对应设计模式有模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器。
2) 基于不同设计模式去想案例场景或者技术场景
技术场景：数据库连接池(单利)、线程池(单利)、解析文件创建对象(建造者)、多线程场景创建对象(原型)、创建不同对象(工厂或抽象工厂)、多数据源切换(适配器)、切换不同加密方式(策略)、数据库操作(模板方法)、获取用户会话(享元模式)
业务场景：计数器(单利)、在线人数统计(单利)、日志操作(单利)、商品价格优惠嵌套运算(策略)、订单状态更新(状态模式)、不同服务文件上传(代理模式)、打印不同类型发票(模板方法)。
3) 设计能力提升最需要的是经验和总结。需要日积月累开发经验和业务经验，同时还学学习别人的思维模式，不断的优化自己的程序，在优化中不断总结，不断优化。

2、模板方法和钩子函数，这两个还是没区分开？

钩子方法源于设计模式中模板方法（Template Method）模式，模板方法模式的概念为：在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。其主要分为两大类：模版方法和基本方法，而基本方法又分为：抽象方法（Abstract Method），具体方法（Concrete Method），钩子方法（Hook Method）。
四种方法的基本定义（前提：在抽象类中定义）：
（1）抽象方法：由抽象类声明，由具体子类实现，并以 abstract 关键字进行标识。
（2）具体方法：由抽象类声明并且实现，子类并不实现或者做覆盖操作。其实质就是普遍适用的方法，不需要子类来实现。
（3）钩子方法：由抽象类声明并且实现，子类也可以选择加以扩展。通常抽象类会给出一个空的钩子方法，也就是没有实现的扩展。它和具体方法在代码上没有区别，不过是一种意识的区别；而它和抽象方法有时候也是没有区别的，就是在子类都需要将其实现的时候。而不同的是抽象方法必须实现，而钩子方法可以不识闲。也就是说钩子方法为你在实现某一个抽象类的时候提供了可选项，相当于预先提供了一个默认配置。
（4）模板方法：定义了一个方法，其中定义了整个逻辑的基本骨架。
案例：

```java
/**
 * 抽象类，定义模板方法和基本方法
 */
abstract class abstractClass{
    /**
     * 具体方法，声明并实现，继承此抽象类不需实现此方法
     */
    public void concreteMethod(){
        System.out.print("这是一个具体方法");
    }

    /**
     * 抽象方法，abstract 关键字标识，只声明，并不实现，继承此抽象类必须实现此方法
     */
    protected abstract void abstractMethod();

    /**
     * 钩子方法，声明并实现（空实现或者定义相关内容皆可），继承此抽象类的子类可扩展实现或者不实现
     */
    public void hookMethod(){
```

```java
        //可定义一个默认操作，或者为空
        //System.out.print("此钩子方法有个默认操作")
    };

    /**
     * 模板方法，整个算法的骨架
     */
    public void templateMethod(){
        abstractMethod();
        concreteMethod();
        hookMethod();
    }
}


public class childClass2 {
    public void bond(abstractClass abstractClass){
        abstractClass.templateMethod();
    }
}
public class Test{
    public static void main(String[] args) {
        childClass2 childClass2=new childClass2();
        childClass2.bond(new abstractClass(){//匿名内部类实现回调
            @Override
            protected void abstractMethod(){
                System.out.print("子类实现父类抽象类中的抽象方法");
            }

            /**
             * 重构钩子方法
             */
            //public void hookMethod(){
                   System.out.print("子类可以在父类钩子方法实现的基础上进行扩展");
        }
    }
}
```

3、spring 设计模式讲解?

Spring 中用到了很多设计模式，我们这里列举常用的 9 种：

1) 简单工厂
2) 工厂方法（**Factory Method**）
3) 单例模式（**Singleton**）
4) 适配器（**Adapter**）
5) 包装器（**Decorator**）
6) 代理（**Proxy**）
7) 观察者（**Observer**）
8) 策略（**Strategy**）
9) 模板方法（**Template Method**）

4、能不能介绍下享元模式和状态模式的其他应用场景?

享元模式和状态模式在工作中应用场景一般都是很实用功能，我们这里对相关应用场景做一些总结。

享元模式：

1、系统有大量相似对象。

2、需要缓冲池的场景。

3、系统中有大量对象，这些对象消耗大量内存，并且对象的状态大部分可以外部化。

工作场景中，多线程会话共享，单点登录都是一种享元模式思想，分布式锁也是基于享元模式的思想实现，多个系统同步认证用户身份信息。

状态模式：

1、状态模式适用于行为随状态改变的业务场景，比如状态改变了，行为也会做成改变。

2、业务代码中很多条件的情况，加入一些代码有很多的 if...else，并且经常改变，这种情况就可以使用状态模式进行编写。

工作场景中，游戏角色发生变更或者 OA 系统中用户角色发生变更，都会拥有不同的权限，订单下单、支付、取消订单，不同的状态变化也会引起不同的操作。

5、需要更多例子去学习，该模块比较抽象

学习设计模式，不需要全面去学，需要掌握使用、有用、经常用的设计模式即可，例如：单利、工厂模式、状态模式、代理模式、策略模式、模板方法，学习这些已经可以解决工作中大部分问题了。

GOF 中 23 种设计模式：

1、单例（Singleton）模式
某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，其拓展是有限多例模式。

2、原型（Prototype）模式
将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。

3、工厂方法（Factory Method）模式
定义一个用于创建产品的接口，由子类决定生产什么产品。

4、抽象工厂（Abstract Factory）模式
提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。

5、建造者（Builder）模式
将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。

6、代理（Proxy）模式
为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。

7、适配器（Adapter）模式
将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

8、桥接（Bridge）模式
将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。

9、装饰（Decorator）模式
动态的给对象增加一些职责，即增加其额外的功能。

10、外观（Facade）模式
为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。

11、享元（Flyweight）模式
运用共享技术来有效地支持大量细粒度对象的复用。

12、组合（Composite）模式
将对象组合成树状层次结构，使用户对单个对象和组合对象具有一致的访问性。

13、模板方法（TemplateMethod）模式
定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。

14、策略（Strategy）模式
定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。

15、命令（Command）模式
将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。

16、职责链（Chain of Responsibility）模式
把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。

17、状态（State）模式
允许一个对象在其内部状态发生改变时改变其行为能力。

18、观察者（Observer）模式
多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。

19、中介者（Mediator）模式
定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。

20、迭代器（Iterator）模式
提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

21、访问者（Visitor）模式
在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。

22、备忘录（Memento）模式

在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。

23、解释器（Interpreter）模式
提供如何定义语言的文法，以及对语言句子的解释方法，即解释器。

三、大促高并发系统下 JVM 图和调优指导

1、生产环境 Metaspace 设置的最大值偏小，导致 oom,如何进行问题的排查；2、为什么 metaspace 扩容会导致 fullGC；fullGC 是否会对 metaspace 进行内存回收，如果不回收为什么要出发 fullGC；3、运行中的程序在哪些情况下会导致 matespace 占用内存上生，我们如何避免 metaspace 占用内存上生

1. oom 异常中会有原因，判断是否是 Metaspace 导致，例如：OutOfMemoryError：MetaSpace
2. Metaspace 不足会引发 fullgc，会回收垃圾对象，但一般能回收的很少
3. 频繁使用反射类加载、动态代理类加载、BeanUtils 中的对象拷贝，程序中引入一些无用的 jar 等

2、正常项目生产环境中，怎么知道内存泄漏了？

最有效的方法就是分析内存的 dump 文件

3、如何调优 jvm 会使项目较小的占用内存

调优 jvm 的目的不是占用最小的内存，而是提升吞吐量、降低暂停时间。如果是需要测试最小内存，可以将最大内存设置的小一些，进行测试功能是否正常，即可得出最小的内存。

四、架构师有必要深入下 JMM

1、老师讲的例子懂了，遇到别的，自己还是分析不出来

- 可以参考 jcstress 官网的代码和例子
- 分析的目的是为了透彻理解 volatile，synchronized 对可见性、有序性的影响
- 一旦熟悉了 volatile 等的使用，只需要根据一些指导原则去编写代码即可，无需一条条分析

可以遵循以下原则

（1）确认有没有发生竞态条件，即有没有多线程共同读写共享变量

✧如果没有，代码无需考虑线程安全
✧如果有，是不是可以利用线程封闭技术避免共享？

- ThreadLocal

✧如果有，是不是可以考虑将变量只读避免竞态条件发生

（2）确实没法避免竞态条件

✧首先考虑使用成熟的线程安全类， JUC
✧如果该线程安全类不能满足要求，才需要自己来实现线程安全

- 例如，使用 synchronized 同步块来保证原子、可见、有序，但性能不那么理想，且缺少控制
- 可以使用 ReentrantLock 实现更多控制，例如可中断，超时，公平等特性
- 对于计数类的变量，可以使用原子类来其保证原子、可见、有序

（3）volatile 的使用场景

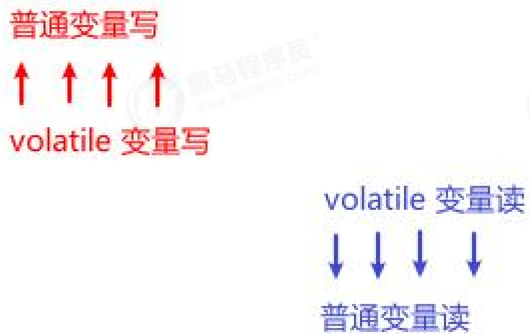
✧volatile 读无需加锁
✧用在单线程写共享变量时
✧写入时不依赖变量原有值
✧其它情况，需要配合 cas，甚至是 synchronized 一起用

（4）构造一个新对象，将其共享使用，遵从下面规则就能实现安全发布

✧任意某个成员变量用 final 修饰
✧或者，最后赋值的成员变量用 volatile 修饰

2、为什么 volatile 会影响普通变量的可见性，影响的范围是多大

原理就是内存屏障



- volatile 变量写，会连同普通变量的改动一起同步到主内存，普通变量不能越过屏障提前写到主存
- volatile 变量读，能够保证读取到自这个 volatile 变量写以来的，所有共享变量的最新值，普通变量的读不能越过屏障，读到过期值
- 因此 volatile 的 `写 -> 读`，就是保证线程切换时所有共享数据的可见性，这也是 hb 规则定义中有意义的一点

3、如何区分指令重排序与线程切换导致的结果不符合预期

- 重排序是线程内代码的执行的先后次序发生了变化
- 而线程切换会导致线程间的代码发生交错

通常不会根据这个结果去反推解决方法，而是根据实际代码来确定解决方法，例如，

- 一个线程对共享变量进行赋值，而另一个线程对该变量进行读取，这时出现问题就从可见和有序上去排查，因为赋值操作是单一原子操作，不能被线程切换所左右
- 如果是复合操作，例如 get-modify-write、check-then-act，多个共享变量，这种代码会受到线程切换的影响

  i++

  map containsKey if put

4、理论性比较强，怎么才能在做项目中 来加深对 jmm 的理解呢

如果你不是框架或类库的设计者，基本用不到这些知识，项目中一般仅仅跟业务打交道，用不上。

框架的设计者都会避免业务开发人员过多考虑线程安全问题，例如

- tomcat 是多线程的吗？是！但你编写一个 servlet 调用业务逻辑，执行数据操作时，基本不需要考虑线程安全问题，你可以认为你的代码在一个单线程内工作
- netty 是多线程的吗？是！但它把同一个 channel 的 io 操作都封在了一个线程中，因此你也无需考虑 channel handler 工作时的线程安全

即便你是框架或类库的设计者时，jmm 也仅仅提供了理论支撑，实际用的仍然是哪些多线程的知识

5、3 安全发布 使用 volatile 改进可以保证 t=s 赋值操作排在设置属性值后面吗？

可以，只要 volatile 变量的赋值放在构造的最后就可以

6、java 里面存在的是值传递还是引用传递, 怎么理解???是不是对基本类型采取的是值引用, 而引用类型采取的是引用传递??? 如果是这样为什么有很多的人都说 java 本身只有值传递???

必须只能是值传递，引用类型参数传递时，也传递的是引用的值

- 啥叫值传递？传递的是副本
- 啥叫引用传递？传递的是本身

首先明确一点，java 中的方法调用时，会为这些方法参数分配空间、而每个参数的值，来自于实参的拷贝。例如：

基本类型:

```
public static void foo(int i){
}
public static void main(String[] args) {
    /*
      将 a 的值 10 拷贝一份，传递给方法参数 i
```

```
        a 和 i 是完全不同的变量，占用不同的内存空间
        即使将 int i 的参数名改为 a 也一样
    */
    int a = 10;
    foo(a);
}
```

引用类型：

```
public static void foo(User u){
}
public static void main(String[] args) {
    /*
        将 x 的引用值拷贝一份，传递给方法参数 u
        x 和 u 是完全不同的变量，占用不同的内存空间
        即使将 User u 的参数名改为 x 也一样
    */
    User x = new User();
    foo(x);
}
```

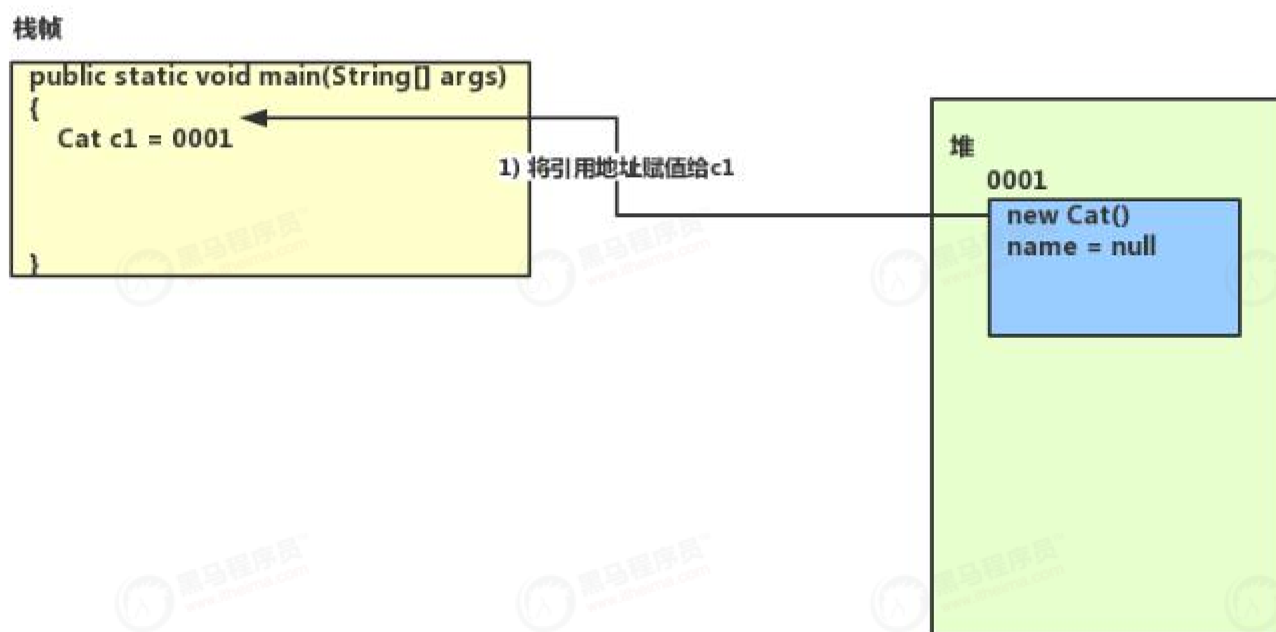
更详细的过程分析，请看下面的例子：

例 1：

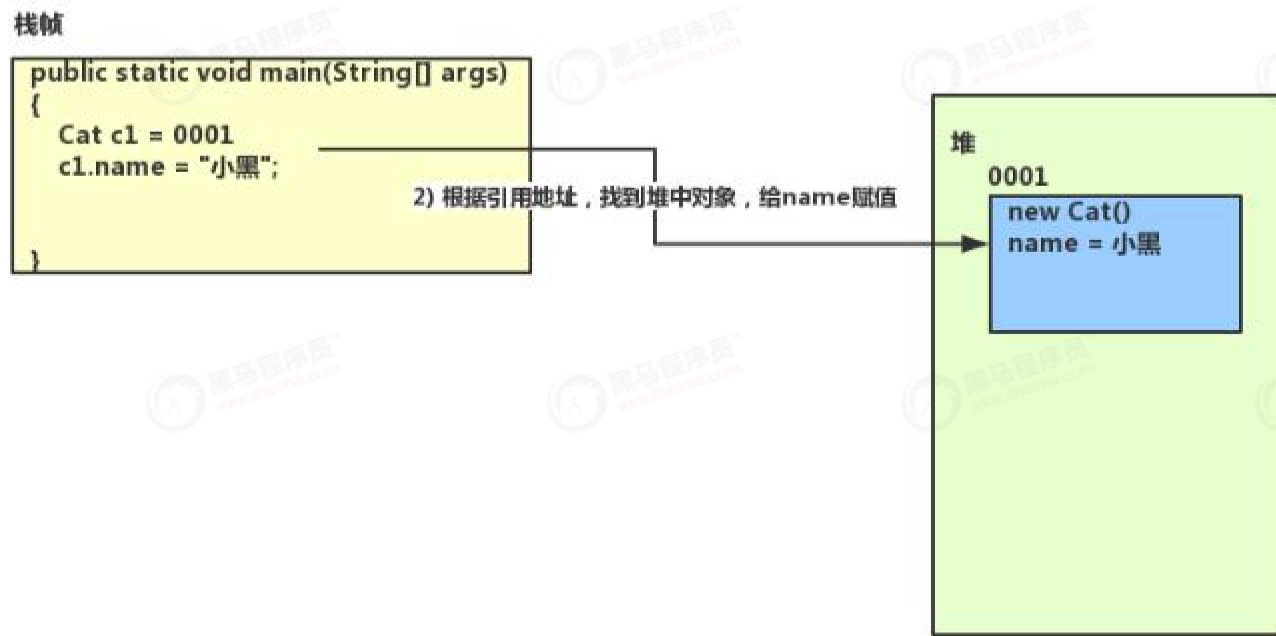
```
public static void main(String[] args) {
  Cat c1 = new Cat();
  c1.name = "小黑";
  foo(c1);
  System.out.println(c1.name);
}

public static void foo(Cat c2) {
  c2 = new Cat();
  c2.name = "小白";
}
```

执行过程分析如下：

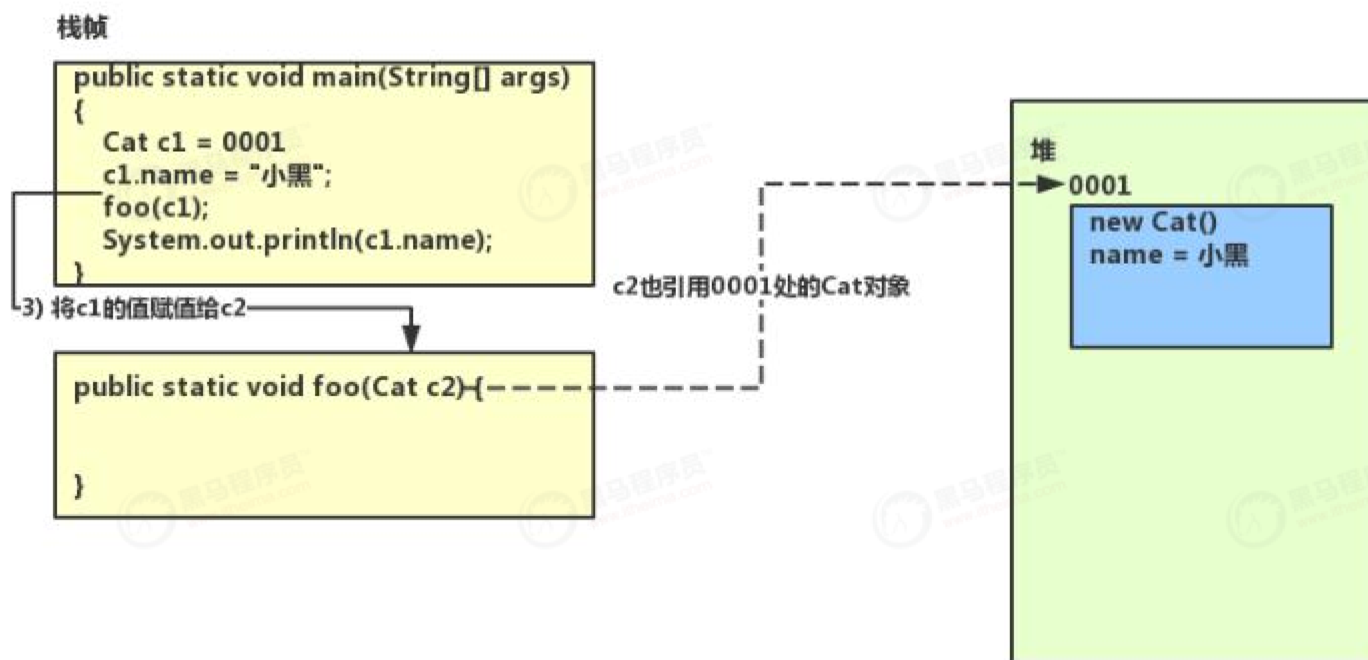
1、main 方法中执行 new Cat() 在堆中分配内存（假设地址为 0001），初始化 name=null，接下来将引用地址 0001 赋值给 main 方法中的 c1 变量



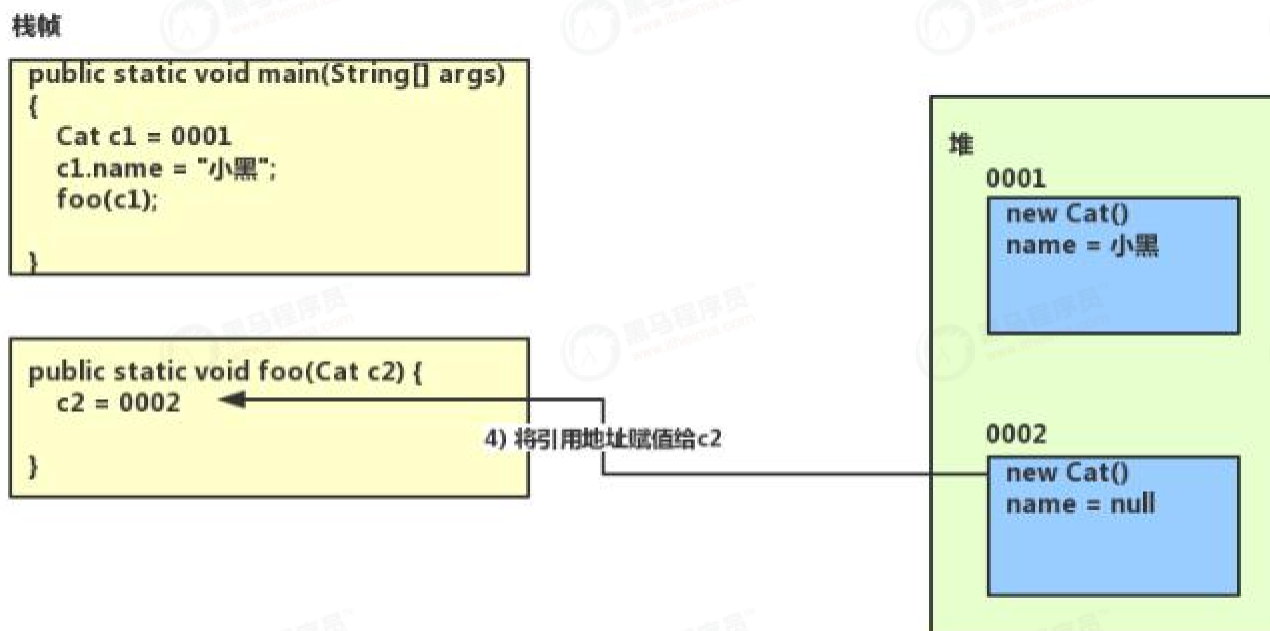
2、根据 c1 变量，引用到堆中 0001 处的 Cat 对象，修改 name=小黑



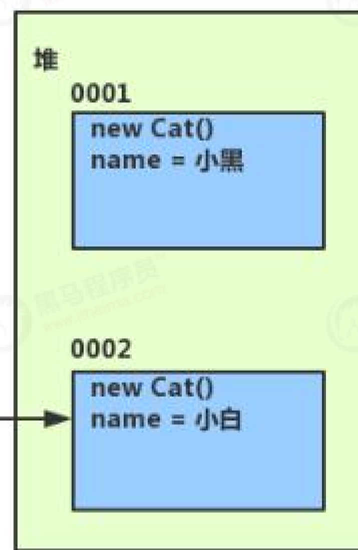
3、接下来调用 foo 方法，将 main 中 c1 变量的值（0001），赋值给 foo 方法中的 c2 变量，这里是传值，c1 和 c2 只是引用值一样，引用了同一个堆对象，而它们是独立的变量，占用不同的内存空间



4、在 foo 方法中执行 new Cat() 会在堆中新分配一块内存（假设地址为 0002），初始化 name=null，并将引用地址 0002 赋值给 foo 中的 c2 变量，这时 c1 不会受到任何影响，仍是 0001



5、根据 c2 变量，引用到堆中 0002 处的 Cat 对象，修改 name=小白



6、返回 main 方法，这时 c1 变量仍然引用的是 0001，因此打印 c1.name 还是小黑

但 c++ 中是有引用传递的：

例 2 C++引用传递：

```
#include <iostream>
using namespace std;
int main(){
    int a = 10;
    int b = 20;
    cout << "Before a:" << a << " b:" << b << endl;
    swap(a,b);
    cout << "After a:" << a << " b:" << b << endl;
    return 0;
}
void swap(int& a, int& b){
    int c = a;
    a = b;
    b = c;
}
```

执行

```
g++ hello.cpp -o hello
./hello
```

输出：

```
Before a:10 b:20
After a:20 b:10
```

java 中：

例 3 java 值传递（基本类型）：

```
public static void swap(int a, int b) {
  int c = a;
  a = b;
  b = c;
}

public static void main(String[] args) {
  int a = 23;
  int b = 47;
  System.out.println("Before, a:" + a + " b:"+ b);
  swap(a,b);
  System.out.println("After, a:" + a + " b:"+ b);
}
```

输出：

```
Before, a:23 b:47
After, a:23 b:47
```

例 4 java 值传递（引用类型）：

```
public static void swap(Integer a, Integer b) {
  Integer c = a;
  a = b;
  b = c;
}

public static void main(String[] args) {
  Integer a = new Integer(23);
  Integer b = new Integer(47);
  System.out.println("Before, a:" + a + " b:"+ b);
  swap(a,b);
  System.out.println("After, a:" + a + " b:"+ b);
}
```

输出仍是：

```
Before, a:23 b:47
After, a:23 b:47
```

7、实现原理还是有些不清楚

结论记住就好，原理留给 jvm 工程师就好

8、染色指针可以大幅减少在垃圾收集过程中内存屏障的使用数量，ZGC 只使用了读屏障。对 volatile 有影响吗？

- 没有影响，这是内存屏障在不同领域的应用，ZGC 只是在垃圾回收过程中只用读屏障，它提升的是垃圾回收的效率
- 而 volatile 仍然需要在写前加 StoreStore 屏障，写入后加 StoreLoad 屏障，在读取后加 LoadStore 与 LoadLoad 屏障，功能并没有任何改变

五、JDK 提高效率新特性探究

1. springboot 使用@Async 和 CompletableFuture 都可以达到异步编程，那么在项目实际应该中，我该怎么选取呢？

1）@Async 是 Spring 提供的，内部采用的是 SimpleAsyncTaskExecutor，若不断创建线程，可能导致 OOM。而且它也做不到线程复用。A,B,C-> D
2）其在使用时，要么返回值为 void，或者返回 Future。这点和 completableFuture 类似，但其无法做到任务编排。
3）如果没有 Spring 的话，@Async 就无法使用了。

2、lambda 表达式什么时候可以传参什么时候不可以传参?有什么界定?为什么数组不能直接使用 lambda 表达式进行遍历???

传参与不传参需要根据当前要操作的函数式接口或匿名内部类。

3、Stream 流的 filter 方法接收的是 Predicate 函数式接口, 这里传 Lambda 表达式的时候是相当于实现了 test 方法, 那么传方法引 Student::getName 也是实现 test 方法吗, 什么原理???

这点问题不会发生，编写时是一定会报错的。

六、多线程并发在电商系统下的追本溯源

1、threadlocal 如何“父子调用”顺序的多线程如何共享变量？或者说这种业务可能就不适用放 thread 里面了，那放在哪里比较好呢？

父子调用，这里的意思不太明确。如果是父子类之间方法调用，那么他们还属于同一个线程，threadlocal 可以正常使用。
如果是父线程中又开了子线程，比如 new 了一个 Thread，那不行，threadlocal 无效。
因为 threadlocal 的本意是线程间互相隔离，而不是共享。如果线程间想共享变量的话，直接外部定义就可以。

2、线程池的队列和消息队列这两个队列是不是同一种队列 2.如果消息队列满了，会不会和线程池的队列一样执行拒绝策略 3.重量级锁为什么适用于追求吞吐量的场景

一：线程池队列和消息队列不是同一种。线程池阻塞队列是 concurrent 包中的类，消息队列是中间件，比如 rabbitmq，kafka，rocketmq，activemq 都是常见的消息队列。
二：当然任何东西都是有限度的的，消息队列也不例外有最大容量限制，超出后拒绝接收。消息扔不进去

三：重量级锁适合应付大量线程同时访问同步块的情景。因为它会让申请锁失败的线程阻塞，你多少线程在等我是不怕的，因此吞吐量也就上来了。而轻量级锁和自旋，会不停占用 cpu，那么系统的响应能力就会下降，吞吐量收到影响。

3、ThreadLocal 案例

在课件中有详细案例，在 1.4.2 章节，后面的日志请求链路追踪课题也会有 web 环境下使用的详细案例。
总结来说，多线程时，如果有一个变量，它的值可以在每个线程里自由的赋值和读取，而不用担心别的线程干扰它。如 web 请求参数的上下文传递

4、怎么充分发挥 多线程 在普通项目中的作用 平时项目也没有用到高并发 多线程方面的 很困惑呀

举个很常见的例子。比如我们要为每个用户店铺生成一个静态页。我们知道文件写入 io 是很慢的，如果有 1000w 用户生成的话单线程可能会做很久，可以开启多个线程。
再比如财务对账单的生成。尤其是支付中心或者收银台，每天要调度为各个业务线或者细分为各个支付通道生成一份对账单，一般是一个 csv 文件，需要上传 ftp，这是个 io 耗时性任务，也是一个典型的多线程场景。

5、多线程 debug 实战，如何自测出多线程代码是否有死锁

死锁从直观感受上，可能会给你觉得你的程序卡死了，无响应。比如在 web 程序中，某个请求一直打不开。但是这个只是猜测，卡死可能有很多种原因，想要确认是不是死锁的话，就必须借助我们课堂上讲的 jdk 工具（3 种）

6、个人理解使用多线程的目的主要是为了减少响应时间，那么，在如今的开发技术体系下，（SpringMVC），如何利用多线程技术提高响应效率？例如，假如一个 service 方法中，有一个处理时间很长的子方法被调用，那么用另一条线程异步执行这个方法会不会提高整个请求的响应时间？ 2.Excel 导出这块，当数据量很大的时候，如何利用多线程提高响应时间？多线程提高效率的点在哪？是在数据处理过程中，还是利用 poi 写入 Excel 过程中？

问题 1：web 程序下手动使用多线程的情况相对较少。因为服务器层面本身已经做到了多线程（每个请求开启一个 tomcat 线程）。如果你在代码里有耗时操作，而且这个操作可以并行。比如用户上传了一个文件，那么你新开线程异步处理文件 io，而主线程继续执行，是能够提升响应速度的。
问题 2：先看数据处理阶段。比如你的数据量比较大，对每行数据都需要比较耗时的处理操作。那么你可以考虑开启多线程比如 fork/join 来并行处理，效果会很明显。再看导出阶段，如果我们只是生成 1 个 excel 文件，那么多线程也无处下手。如果分批导出多份，那么开多线程导出会让你的 io 并行，也会有很大的提升效果。

7、unsafe 可以讲一下吗

Unsafe 是把双刃剑，俗称一半魔鬼一半天使。这是一个类，在 sun.misc.Unsafe
因为它里面提供了像 C 语言一样直接操作内存值的方法，所以取名 Unsafe，也就是不安全的。
搞不好会造成 jvm 级别的崩溃，在公司项目中少用，不要做好奇宝宝
感兴趣的同学可以自己在本地玩一玩

8、hashmap 如果不扩容，是不是就不存在线程安全的问题？

如果仅仅是读数据，比如配置信息放到一个 Map 里就加载一次，那么不会对其结构造成更改，多线程下也没问题。

9、HashMap 数据结构是怎么样的 ？ ConcurrentHashMap 实现原理，源码没看懂；ThreadLocal 为什么会存在内存泄漏？各种锁的应用场景

一、HashMap 结构在基础班里有讲解，说白了就是一个数组，每个元素上放的是链表或者树。
二、ConcurrentHashMap 的原理，咱们有一个流程图，在资料库里。结合那个多看一下，会轻松一些。
三、ThreadLocal 因为用自己做 key，放在了各个线程里面，所以如果外面 threadlocal 变量销毁了，线程没销毁（如线程池里），那么线程里藏着的 value 就可能没人管存在内存泄露风险。而 threadlocal 用两个手段来避免这个问题。1 是 key 弱引用，会被垃圾回收掉变成 null，2 是 null 的 key 在你后续的 set/get 等操作的时候，会遍历一下，一旦发现就把 value remove 掉，也就帮我们清理了可能泄露的 value 内存。但是，用完手动 remove 是个好习惯。
四、常见的锁就是 synchronize，retreentlock，reentrantreadwritelock，sync 不能跨方法，lock 可以跨方法加锁和释放锁，这俩差不太多。如果有读写并行的地方，使用读写锁可以提高并发性能。

10、多个串行的线程池如何有效停止

这里串行的线程池不太准确。线程池里的线程是并行工作的。如果说串行的话，从队列里取任务是一个个领取，属于串行。
而线程池停止，调用课堂里说的两个方法。Shuntdownnow 和 shuntdown ， 推荐 shuntdown。

11、thread/threadlocal/threadlocalmap 之间的关系

首先，Thread，ThreadLocal 是普通类， ThreadLocalMap 是 ThreadLocal 里的一个内部类
然后他们之间如何搭配的呢？
Threadlocal 你可以定义一个这样的变量比如叫 a。
然后你 new 了很多个线程 Thread， 每个 Thread 对象里都藏着一个 ThreadLocalMap 类型的变量
当你在一个线程里面调 a.set(v)的时候，它会在这个线程自己的那个 ThreadLocalMap 里 put 一下，put 的 key 是变量 a，value 就是你要设置的这里的 v

12、定时任务过多如何调优线程使占用内存变小

这里的定时任务应该说的是 ScheduledThreadPoolExecutor
很遗憾，几乎没有办法。因为这个类只让你设置线程数，不准你设置队列。它内部的延迟队列是一个无界的，理论上可以无限往里扔任务。
那么你的任务扔进去后，时间还没到点，就只能占着内存。

一个可能调优的情况是，如果你的任务很多都是扎堆出现的，比如都是 8 点执行，这个情况下，实当的调大并发线程数有利于你的任务得到快速执行，内存 8 点的时候释放的快一点。但是，8 点之前，还是要占着，没有办法。

一个推荐的做法是，尽量不要在任务对象中放置太大的数据，比如大的集合，数据库的查询数据等。Task 本身只是作为一个定时事件。时间到了之后，会触发它，任务需要的数据再从外面查询获取。

13、currentskiphashmap 如何保证并发读安全的

这里应该指的 ConcurrentHashMap。因为读不会变更数据和 map 的结构，也就不需要加锁。它只会 hash 到对应的插槽里，取元素和当前 key 比对，找到返回就可以了。当然找不到给一个 null。这个操作是可以平行去执行的。相当于数据库的查询。

14、自定义线程池一般是怎么实现?有哪些实现方式?一般是指基于 spring-contex 的 ThreadPoolTaskExecutor 实现或者基于 java.util.concurrent 的 Executors 实现吗?

ThreadPoolTaskExecutor 是 spring 提供的，它对 ThreadPoolExecutor 进行了封装处理，底层用的还是 jdk 的线程池。如果你用的是 spring 环境，那就可以用 spring 的，它能让你使用 spring 的方式来配置相关参数，它会给你配置成一个 bean，这样就可以在需要的地方到处 Autoware。如果不用 spirng 的话，手动创建也能用，但是，你跨 bean 使用的时候呢？。。。

15、concurrenthashmap value 不能为 Null

对的，不但 value，key 也不行。原因就藏在 put 方法的一开始处

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
```

16、threadload 使用

问题 3 已经讨论过，不再赘述
提示一下，后续的业务篇，日志平台用户请求链路追踪实战里，对这个有实际 web 环境下的使用。到时候会讲到