30丨“代码经济篇”答疑汇总

范学雷 2019-03-13

00:00 08:36

讲述：刘飞　　大小：15.76M

到这一篇文章，意味着专栏第二模块“经济的代码”已经更新完毕了。非常感谢同学们积极踊跃地留言，提出了很多独到的见解，我自己也学到了不少新东西。

今天，我来集中解答一下留言区里的一些疑问。有很多问题，我们已经在留言区里讨论了。这里，我们就挑选一些还没有解决掉的问题，深入讨论一下。

@秦凯

> 对性能和资源消耗有一定的意识，但是在具体的开发过程中或者应用运行过程中要对性能进行监控、评测、分析，就束手无策了。

答：我一直都认为，有了意识，其实就成功一大半了。有了意识，我们就会去寻找技术，寻找工具，寻找解决的办法。到了这个专栏的第三个部分（也是接下来要更新的“安全篇”），我们就会更强烈地感受到，“要有意识”是我们首先要获得的能力。大部分代码的问题，其实都是意识和见识带来的问题。

回到这个问题本身，性能的监控、评测和分析，我们通常要通过一定的工具来解决。

第一个常用的工具是 JMH，我们在第 26 篇里简单介绍了这个小工具的用法。JMH 可以用来测试一个接口、一段代码跑得有多快。特别是当我们面临两个选择，并且犹豫不决的时候，对比 JMH 的测试结果，就可以给我们提供一个大致准确的方向。

第二个常用的工具是性能回归测试。我们经常修改代码，修改后的代码性能有没有变坏？这是一个值得我们警惕的问题。我们可以通过自动化的性能回归测试，来检测修改有没有恶化代码的性能。就像普通的回归测试一样，性能回归测试也需要写测试代码。编写自动的回归测试代码就相当于我们制造了一个工具。从长远看，工具可以提升我们的工作效率。

第三个就是找一款成熟的性能调试工具，比如 NetBeans Profiler、JProfiler、Java Mission Control、Stackify Prefix 等。这些性能调试工具，能够综合地检测性能问题，比如内存、CPU、线程的使用状况，或者最耗费资源的方法等。

第四个办法就是用实时的性能监控工具，比如 New Relic APM，Stackify Retrace 等。这个工具可以用在运营环境上，预警性能障碍，检测性能瓶颈。

如果掌握了这四样工具，很多性能问题，我们都可以早发现、早解决。

@悲劇の輪廻

> 某些银行的客户端已经奔着 150M+ 去了……我怀疑他们的开发人员是不是也经过层层外包，根本不考虑客户终端的运行环境。

答：@悲劇の輪廻 提出了一个好问题。代码的尺寸，也是一个我们需要考量的重要指标。

在 JDK 9 中，Java 开始支持模块化（Java module）。Java 模块化背后的一个重要推动力量，就是 JDK 的尺寸。

在云服务，特别是微服务，和移动计算到来之前，我们认为硬盘不值钱，所以一个软件的尺寸大一点也没有关系。

但是对于云服务和微服务来说，使用了多少硬盘空间，也是一个重要的计费项目。这时候，软件的尺寸带来的开销可就是一个常规的日常费用了。

对于移动计算，比如我的手机，空间只有可怜的 16G。一旦存储报警，我几乎没有太多的选择余地，只好删除那些不太常用的、占用空间又大的 App。是不是 App 的开发者，也应该琢磨下怎么节省用户的手机空间呢？

@风清扬笑

> 话说“第一眼看到钱”这个需求貌似是很多人想要的，但是我觉得没有这么做的部分原因也是基于安全考虑，一些 APP 设计把余额放到二级菜单里，而且想看的话还得输入密码。

@IamNigel

> 对于银行 App 我最想看到钱，可以转账，可以管理我的银行卡信息，最近两年在用平安银行的手机 App，看余额得输入密码，这个应该是安全考虑，但其中的很多功能从来不会去点，特别是任意门，一不小心就点上去了。好的地方也有，像转帐以后会记录我最近转过的信息，也是比较方便。

答：@风清扬笑和 @IamNigel 都提到了密码登录的问题。这个问题，是一个传统而又典型的身份认证（Authentication）方式。

有人开玩笑， 我们受够了密码，但是离了密码又活不了 (We cannot live with password; we cannot live without password.)。 密码导致的问题太多了，我们实在没有办法爱上它。二十年前，就有人断言，密码必死无疑。无密码的解决方案也是一茬接一茬地出现。可实际情况是，密码自己越活越洒脱，越活越有样子。现代的新技术，比如指纹、瞳膜、面部识别，都有着比密码更严肃的的缺陷，替代不了传统的密码。

有没有可以降低对密码依赖的技术呢？比如说，使用银行 App，能不能就输一次密码，然后就可以不用再使用密码了。其实有很多这样的技术，比如手机的指纹识别、面部识别，都可以降低密码的输入频率。

如果你想系统地了解有关这方面最新的技术，我建议你从 2019 年 3 月 4 日发布的 WebAuthn 协议开始。深入阅读、理解这份协议，你可以掌握很多现代身份认证技术的概念和技术。了解了这些技术，像是银行 App 输入密码这种麻烦事，你就可能有比较靠谱的解决办法。

关于 WebAuthn 的具体技术细节和方案，请你去阅读 W3C 的协议，或者搜索相关的介绍文章。

@Tom

> 签名数据太大，比如文件图片，占用内存大，使用流处理可以减少内存占用吗？

答：签名数据可以分割开来，一段一段地传递给签名的接口。 比如要分配一个 2048 个字节的数组，每次从文件中读取不多于 2048 个字节的数据，传递给 Signature.update()。文件读取结束，再调用 Signature.sign() 方法完成签名。这种方法只需要在应用层分配一块小的内存，然后反复使用。

不太清楚你说的流处理是什么意思。如果一个字节一个字节或者一小块一小块地读取文件数据，就会涉及太多的 I/O。虽然节省了内存，但是 I/O 的效率可能就成了一个问题。

这个数组的尺寸多大合适呢？这和具体的签名算法，以及文件 I/O 有关系。目前来看，2048 个字节是一个常用的经验值。

问题（第 24 篇）：延迟分配的例子中，为什么要使用 temporaryMap 变量以及 temporaryMap.put() 而不是 helloWordsMap.put()?

为了方便阅读，我把这段要讨论的代码拷贝到了下面：

复制代码

```
public class CodingExample {
    private volatile Map<String, String> helloWordsMap;

    private void setHelloWords(String language, String greeting) {
        Map<String, String> temporaryMap = helloWordsMap;
        if (temporaryMap == null) {    // 1st check (no locking)
            synchronized (this) {
                temporaryMap = helloWordsMap;
                if (temporaryMap == null) {    // 2nd check (locking)
                    temporaryMap = new ConcurrentHashMap<>();
                    helloWordsMap = temporaryMap;
                }
            }
        }

        temporaryMap.put(language, greeting);
    }

    // snipped
}


```

@yang

> 使用局部变量，可以减少主存与线程内存的拷贝次数。

@轻歌赋

> 双检锁在多 CPU 情况下存在内存语义 bug，通过 volatile 实现其内存语义。

@唐名之

> 使用局部变量，可以减少主存与线程内存的拷贝次数。这个点还是有点不明白能解释下嘛?

答：要解释这个问题，我们需要了解 volatile 这个关键字，要了解 volatile 这个关键字，就需要了解计算机和 Java 的内存模型。这些问题，在杨晓峰老师的《Java 核心技术 36 讲》和郑雨迪老师的《深入拆解 Java 虚拟机》专栏里，都有讲解。详细的技术细节，请参考两位老师的文章（点击链接即可直接看到文章）。

简单地说，线程的堆栈、CPU 的缓存、计算机的内存，可以是独立的物理区域。共享数据的读取，要解决好这些区域之间的一致性问题。也就是说，不管从线程堆栈读写（线程内），还是从 CPU 缓存读写（线程间），还是从计算机的内存读写（线程间），对于每个线程，这些数据都要一样。这就需要在这些不同的区域之间做好数据同步。

我们再来看这个例子。声明为 volatile 的 helloWordsMap 是一个共享的资源。它的读写，需要在不同的线程间保持同步。而同步有损效率。有没有办法降低一点读写的频率呢?

如果我们不使用共享的资源，也就没有了数据在不同内存间同步的需求。temporaryMap 变量是一个方法内的局部变量，这个局部变量，只在这个线程内起作用，不需要和其他线程分享。所以，它的访问就不存在同步的问题了。

把共享的 volatile 变量的引用，赋值给一个局部的临时变量，然后使用临时变量进行操作，就起到了降低共享变量读写频率的效果。

这种办法有一个适用场景，就是 volatile 变量的引用（地址）一旦初始化，就不再变更。如果 volatile 变量的引用反复变更，这种办法就有潜在的数据同步的问题了。

以上就是答疑篇的内容。如果这些问题是你和朋友，或者同事经常遇到的问题，不妨把这篇文章分享给他们，一起交流一下。

从下一篇文章起，我们就要开始这个专栏的第三部分“安全的代码”的学习了。在这一部分，我们将主要采用**案例分析**的形式来进行学习。下一篇文章见！

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表　　0/2000字　　提交留言

精选留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。