



下载APP



26 | 一个嵌入式实时系统都要从哪些方面做好性能优化？

2021-07-15 尉刚强

《性能优化高手课》

课程介绍 >



讲述：尉刚强

时长 15:44 大小 14.42M



你好，我是尉刚强。从这节课开始，我们就进入课程的案例分享模块了。在这个模块中，我会通过之前参与的一些真实项目案例，来帮助你巩固前面课程中学习到的各种性能优化技术，并带你进一步深入了解实际项目中的技术落地细节。

今天，我要给你分享的是一个完整的性能优化案例，我会从启动这个性能优化任务开始，带你了解每一步的工作内容，包括如何启动性能分析、设计、实施性能优化工作，以及中间的思考过程，直到最后达成性能优化目标。

在这个具体剖析的过程中，你会发现做好性能优化并不是一锤子的买卖，而是一个系统的软件工程活动。同时，案例中涵盖的高性能软件设计、高性能编码、性能测试与看护、性能调优等很多方面的具体技巧，你也可以直接拿来在自己参与的项目中使用。



这个性能优化攻关案例，我会按照案例背景、性能分析诊断、性能测试看护、设计与实现优化、优化成果对比的顺序进行介绍，这与我们性能优化工作的开展节奏也是基本一致的。另外为了方便理解，我在讲解案例的过程中，也会省略或简化掉很多跟领域相关的知识，主要聚焦在项目中所使用的性能优化技术。所以这节课，你需要重点关注的就是性能优化的完整实施过程。

那么下面，我们就先来了解下这个案例的背景吧。

案例背景

这是一个基于 C/C++ 开发的嵌入式实时软件子系统，它的**核心业务逻辑**是采用一定排序算法，选择出一部分优先级比较高的用户，分配相应的传输带宽资源。其中有几个关键的约束分别是：带宽资源、用户优先级、用户的需求都在实时动态变化中；核心业务逻辑的处理时间不能超过 1ms，否则就会导致系统业务出现错误。

然后，这个软件系统**最核心的性能指标**是 1ms 的调度用户数（完成排序选择和请求带宽资源分配的用户数目），同时还有一些隐含的性能要求，比如版本二级制大小不能增加、调度时延抖动情况不能增加、内存占用空间不能上升，等等。当然，与其他很多性能优化项目一样，这个团队也设定了**性能优化攻关目标：1ms 内调度用户数性能提升一倍**（比如从 8 用户 /s 提升到 16 用户 /s）。

那么面对这样的需求任务，接下来我们该从哪里开始呢？

性能分析诊断

实际上，我们首先应该进行系统化的性能分析诊断，这样才能找出系统中的所有性能瓶颈资源，从而识别出潜在的性能优化点。

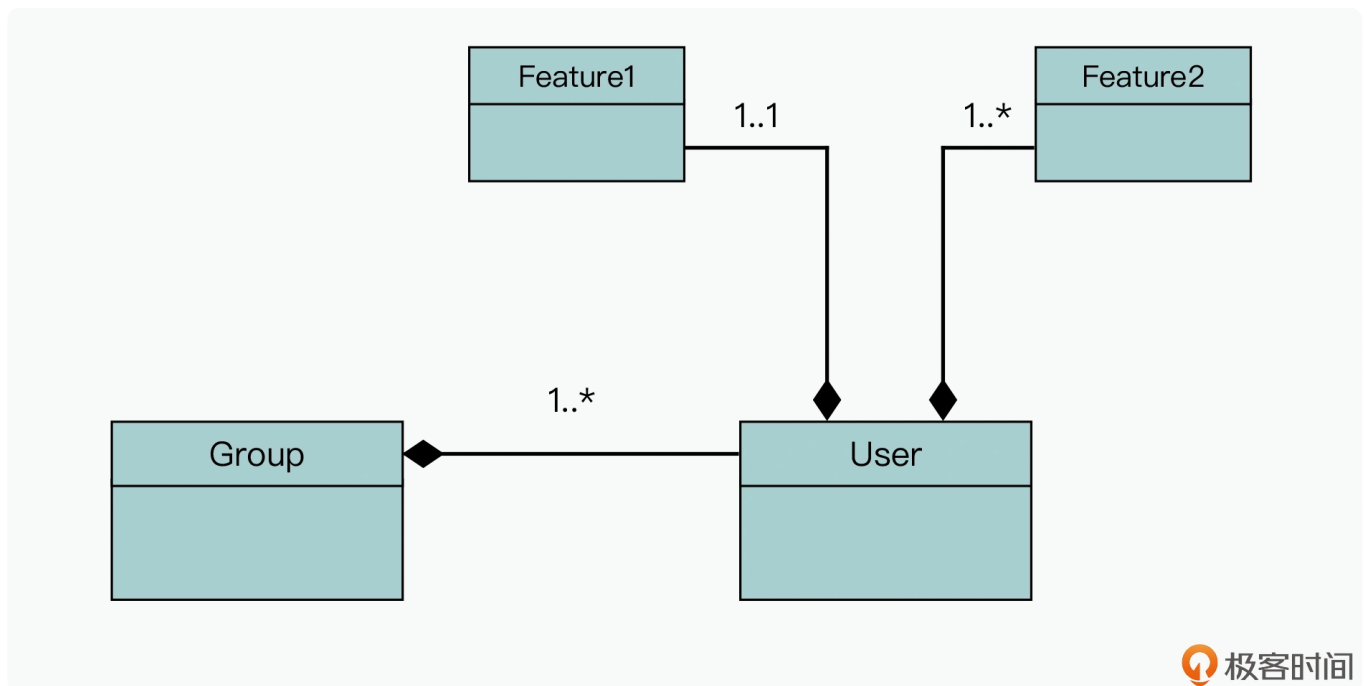
那么，在对这个软件系统进行性能分析和诊断时，采用的就是🔗**自顶向下的性能分析方法**，从高性能设计出发（如并发设计、通信设计等），再具体到编码实现技术（如性能模式、数据结构与算法等）。当然，在性能分析的过程中，我们需要同步获取相关的监控观测数据，来帮助验证这些性能分析结论是否是有效的。

在经过系统性的性能分析诊断后，我们最终识别出了一系列的性能优化点，比如并发任务拆分不均衡、消息交互过度互斥、排序算法效率不高、大量的重复计算逻辑，等等。不过

这么多的性能优化点，用一节课的时间并不能完全分析清楚，所以这里我选择了其中两个比较有借鉴意义的优化点，来给你具体介绍下，这样你在真实的项目开发过程中，也能识别出这类引起性能劣化的软件设计问题。

优化点 1：类对象存储模型对 Cache 不友好

为了方便理解，我把这个软件系统中几个比较关键的类的领域关系，进行了简化处理，然后使用了一个类图来表示，具体如下图所示：



在这个软件系统中，存在多个 Group 的对象实例，其中每个 Group 对象都包含了一组 User 对象实例（规模一般在 1000 左右），而每个 User 对象（用户）又包含了多个特性实例，比如 Feature1 和 Feature2 的对象实例。


而这个软件系统的核心计算逻辑是：**按照不同 User 中的 Feature 对象实例进行遍历，然后进行排序。**

那么这个时候，你其实会发现这些 Feature 对象实例空间，在内存中完全是离散分布的，所以就导致了在数据遍历的过程中，数据局部性不友好。当然，这也是该软件系统在运行过程中，数据 Cache Miss 的概率比较高的主要原因之一。

实际上，这个优化点反映了一个比较重要的现象，也就是很多系统的对象存储模型，会影响到软件的执行性能表现。

优化点 2：业务代码未进行预裁剪，导致代码执行效率不高

在走读和分析业务代码的过程中，我们发现有大量使用模式的分支选择逻辑（有几百次以上），如下所示：

 复制代码

```
1  if(mode == TDD1)
2  {
3      calcBandResTDD1();
4  }
5  if(mode == TDD2)
6  {
7      calcBandResTDD2();
8  }
```

你要知道，**如果业务代码中存在大量的重复分支逻辑判断，就会潜在地影响到指令分支预判成功率，具体表现在指令 Cache Miss 概率比较高**。所以我们再进一步分析业务逻辑，发现这个软件系统一共有 7 种工作模式，它们核心的处理流程都是相似的，只是具体的处理细节中存在差异，这样就会导致系统中存在大量的分支选择判断。

但其实，这个软件系统的工作模式选择在运行之前就可以确定了，所以没必要像这个软件编码实现一样，把这种运行前的选择逻辑放到运行过程中去实现。也就是说，我们可以认为本质上所有的这些分支选择逻辑代码，都可以在运行前裁剪掉，从而减少运行时引入的开销。而且，这样也能避免因为在代码运行时加载了很多不需要的代码，而导致出现一些其他的性能问题，比如冗余的内存空间浪费、二进制文件变大等。

其实，这个软件设计与实现问题，也是一个比较典型的导致软件性能劣化的原因之一。就举个简单的例子，在 ToB 的企业应用开发中，给企业 A 提供的软件中会包含给企业 C 开发的一些定制功能，从而就导致企业 A 的软件运行效率不是最佳的；或者说，某一款电话手表开发的软件 App，在运行中加载了很多其他型号电话手表中，定制化的业务代码和对象实例，从而就会引起执行性能不佳，导致 App 卡顿等问题。

另外不知你发现了没有，这里识别出的这个软件设计实现优化点，并不是通过工具链扫描发现的，而是通过主观地分析和走读业务代码发现的。而且，我们使用工具进一步获取的观测数据，只是为了验证性能分析结论的正确性，以及它对性能产生的具体影响大小。也

就是说，在性能优化过程中，你依赖的应该是自己的独立思考和分析能力，而不是一个具体的工具。

好，那么当识别出了这么多性能优化点之后，我们是不是就可以直接去动手修改优化代码实现了呢？

实际上，我们并没有这么做，因为在代码设计与实现优化前，我们首先应该构筑性能测试看护网。

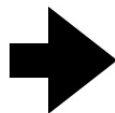
性能测试看护

因为原来这个软件系统的性能测试是基于真实设备上运行的，所以获取软件性能表现的反馈周期会比较长，而且也不方便获取，软件内部模块级细粒度执行开销的观测统计数据。那么在这种现状下，对软件设计与实现进行优化重构修改，其实会很难获取到及时的反馈信息，从而就会影响到性能调优的工作效率。

因此，我们首先就针对这个软件子系统，开发了本地化的微基准性能测试用例，希望可以支持快速获取软件优化重构后，其内部细粒度的观测数据和性能提升的效果反馈。

下面给出的是针对这个软件系统的观测数据伪代码，以及本地性能测试结果的打印效果图，具体如下：

```
TR_ENTER(TIME_RECORD_ID);
{
    Block code;
    Block code;
    Block code;
}
TR_ENTER(TIME_RECORD_ID);
```



```
*****
Test case name: FDD CELL 20M 16UE ...
-----
record id  |   time   | inter
-----
SELECT_UE  |  345435  |  10
ALLOC_DCI  |   23435  |   8
*****
Test case name: FDD CELL 20M 8UE ...
-----
record id  |   time   | inter
-----
SELECT_UE  |  345435  |  10
ALLOC_DCI  |   23435  |   8
```

在图中，左边是添加打点代码的示例，其中大括号中的代码会按照 `TIME_RECORD_ID`，来统计执行花费的 CPU 的 Cycle 数目（指令周期）；右边则是根据性能测试场景，打印出的各个业务流程的具体执行开销。

这样一来，由于这个测试用例可以快速本地化执行，所以我们可以很方便地验证软件优化重构后的性能提升效果。接下来，我们就可以开始对软件进行设计与实现的优化了。

设计与实现优化

首先我们都知道，**软件设计与实现的优化工作是针对性能分析识别出的性能优化点，采用高性能设计与编码技巧，对软件代码设计与实现进行重构，然后逐步优化提升软件性能的过程。**

当然，每一种性能优化点对应的修改解决方法是不一样的，这里我们再回到前面提到的两个性能优化点，来分别看下怎么去对应解决。

优化点 1：类对象存储模型优化设计

针对前面提到的那个类对象模型场景，核心的解决思路应该是将所有 User 对象间的 Feature 的对象实例内存空间放到一起，比如把系统中所有的 Feature1 对象实例放在一起，所有的 Feature2 对象实例放在一起。这样就可以在遍历过程中，实现数据访问有更好的局部性。

具体的设计与实现有很多种方式，这里我就不进行深入介绍了。不过在游戏开发领域，有一个典型的 ECS 架构模式，其特点之一也是解决了对象存储模型与计算模型解耦的问题，从而提升了软件的 Cache 友好性，具体你可以参考 [ECS 架构简介](#)。

优化点 2：裁剪掉未使用的业务代码逻辑

我们再回到之前介绍的那个存在大量重复分支判断的场景，现在我们已经知道，为了提升运行时的性能，本质上就需要在软件运行过程中裁剪不需要的功能与代码。那么这一步具体要如何实现呢？

对于 C/C++ 语言来说，其实有很多种手段来解决这个问题，下面我就来给你具体介绍一下。

一般情况下，你可以通过**预编译宏**或者构建时的**静态多态技术**，来为不同的工作模式生成不同的二进制交付版本。但这种方式有些局限性，引入过多的编译宏会导致代码的可阅读性变差，同时也会给版本管理增加复杂度。

当然，你还可以使用**运行时多态**，通过为不同的模式创建对应的子类，来减少这种类型的分支判断处理。但这种方式也存在局限性，那就是虚指针调用本质上也是运行时判断的，所以也会影响运行时开销。

最后，我们在优化实现的过程中，可以采用**组合式模式**，比如使用泛型技术，去组合通用计算逻辑和各种工作模式下的差异计算逻辑，从而生成在特定工作模式下的代码，来规避虚指针引入的额外性能开销，以及过多编译宏导致代码可读性差的问题。

补充：对于 Java 来说，在采用多态的方式来解决这种问题的时候，可以充分利用 JVM 在运行的过程中，只会加载需要的 class 文件的机制，来避免一些不必要的执行开销；而像 Ruby 这样的动态语言，你还可以在软件执行期间，通过动态删除或修改函数方法实现来优化性能。所以说，不同语言的解决方法有很多差异，当碰到具体的代码实现优化问题时，你还需要结合特定语言的机制，来重构和优化代码。

总之，在性能优化攻关期间，我们需要持续重构优化后的代码并合入到主线中，同时为了保证性能优化结果可以长期有效，我们需要将微基准性能测试集成到 CI 流水线中，来帮助我们在软件性能劣化时，可以第一时间发现问题。

最后，这个嵌入式性能优化攻关项目，在经过系统性的性能分析与诊断，以及持续地重构与优化之后，在一年半的时间内，其软件关键性能指标就取得了近一倍的提升。同时还实现了在性能优化的过程中，软件设计更加清晰，代码更加简洁。

小结

对于性能优化领域来说，凡是工具可以直接定位和解决的问题，很有可能都不是系统最核心的软件性能问题。所以在对一个复杂软件系统进行性能优化时，需要你具备很强的软件设计和编码能力，同时还需要深入到业务设计与实现中，去识别和发现各种性能优化点，并且还需要逐步修改和重构代码来优化性能，这样才有可能设计和开发出高性能的软件系统。

那么今天这节课，我给你讲解了一个嵌入式实时系统的性能优化攻关项目案例。你应该重点关注的是这个性能优化案例的实施过程，包括其中每一个步骤的核心关注点和解决的问题都是什么，以及这样做的好处是什么。在掌握了实际项目中落地实施性能优化技术的细节之后，你就可以在具体的性能优化项目中，去借鉴和采用这些方法，从而可以在更大程度上去帮助改进软件产品的性能表现。

思考题

在你参与的性能优化项目中，关键的性能优化点是通过深入分析业务系统的设计与实现发现的，还是通过监控分析工具来发现的呢？

欢迎在留言区分享出你的答案和分析诊断思路。如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

 赞 0  提建议

© 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 25 | 性能调优什么时候应该停止？

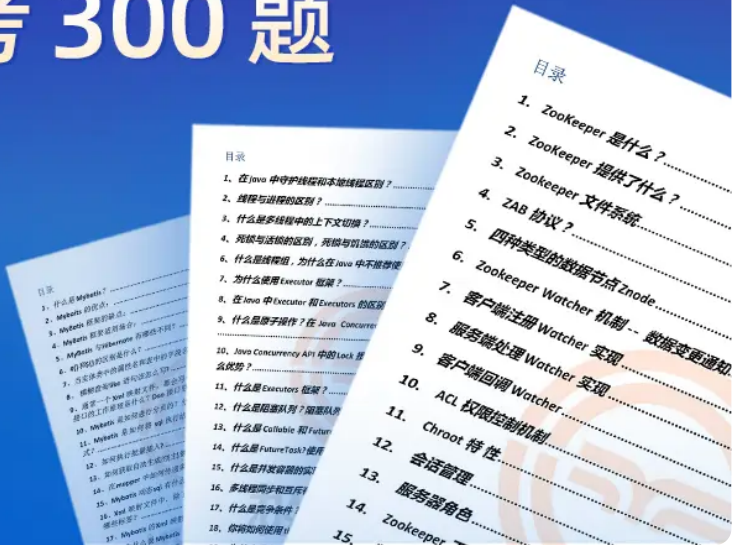
下一篇 27 | 解决一个互斥问题，系统并发用户数提升了10倍！

更多学习推荐

Java 面试必考 300 题

最新汇总

限时免费领取



精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。