



下载APP



14 | 内存使用篇：如何高效使用内存来优化软件性能？

2021-06-17 尉刚强

《性能优化高手课》

课程介绍 >




讲述：尉刚强

时长 19:20 大小 17.72M



你好，我是尉刚强。今天，我们来聊聊如何通过内存的高效使用，来进一步优化和提升软件性能。

软件的实现是通过变量和变量之上的计算逻辑组成的，而在计算机运行期间，变量主要依赖于内存来承载。所以，如何高效地使用内存，就成为了高性能编码优化的重要手段之一。而在软件编码的过程中，不同实现方式对内存的影响，则主要体现在这三个场景：**内存的空间与布局、内存的申请与释放、内存的读取与修改。**

不过就我的观察发现，很多研发团队在软件的开发阶段，并不会去关注内存使用优化， 样当业务上线后，伴随着用户规模的快速增长，各种内存引起的性能问题就逐渐暴露出来了，比如内存空间不够、内存操作引起比较大的时延抖动等。

只有到了这个时候，他们才会意识到内存使用优化的重要性，但这时与内存相关的代码实现已经侵入业务的各个地方，调整重构变得举步维艰。所以可见，我们应该在编码实现的过程中，就去掌握优化内存使用的技巧和方法，以避免软件后期引起比较严重的性能问题。

那么今天这节课，我就会从这三个场景出发，带你了解如何通过不同的编码方式，来调整优化内存使用效率，从而提升软件性能。

不过在开始之前，我要先说明一点，就是不同编程语言的语法、解析运行机制的差异都很大，在高性能编码实现的技巧手法上也都不太一样。所以今天，我主要是以使用范围和人群都很广泛的 Java 语言为主，给你讲解如何从内存使用的角度进行高性能编码，从而开发出性能更加优越的软件。而在一些特定场景下，我还会选用一些 C/C++ 代码片段进行对比分析，这样有助于你理解背后的原理与意义。

好了，接下来，我们就一起看看如何通过代码实现来优化内存的空间与布局吧。

内存的空间与布局优化

首先你要清楚的是，通过编码实现手段减少对内存空间的使用，不仅能帮助你节省软件运行期间占用的内存开销，还可以减少代码运行期间对内存空间的操作，从而提升软件运行速度。

那么这样问题也就来了：我们要如何通过编码实现，来减少使用的内存空间呢？

这里，我给你总结了三种优化思路，虽然它们的关注视角不太一样，但最终都可以改善内存使用的性能。当你理解了这三种思路背后的原理，自然就会在编码过程中，朝着提升内存使用效率的方向改进。

下面我就详细给你介绍下。

按照变量存储信息量选择对应的类型定义

第一个手段是：按照变量存储信息量来选择对应的类型定义。这要怎么理解呢？我先带你来看一个具体的例子。

这个例子针对的是一个学生年龄的数据信息，你可以先想一想，一个在校学生的年龄有效范围会是多少呢？首先，基于常识判断，你可以认为该学生年龄是小于 100 岁的。因此针对这个场景，你在 Java 中使用一个 byte 基本类型，差不多就能满足信息的存取需求了。

可是，如果你使用 long 类型来保存，就会造成额外空间的浪费，并且也会潜在地影响程序的执行速度。所以，这就是一个没有根据存储信息量来选择类型定义的反例。遗憾的是，很多开发工程师在实际的代码开发过程中，并没有关注到这样的代码实现细节。

OK，现在我们换个思路，看看还有没有更极致的优化内存空间的方法。

首先我想问你的是，在计算机上最小的存储单位是多大呢？答案是一个 bit 位。那么这里，我们就来看下，在 C/C++ 中是如何记录该学生的结构体定义的，如下所示：

[复制代码](#)

```
1 struct Student
2 {
3     unsigned char gender: 1;
4     unsigned char gradeId: 3; //年级号 (1~6)
5     unsigned char classId: 5; //班级号 (1~20)
6 };
```

可以看到，在这个代码结构体中，使用了一个字节表示该学生的性别（gender）、年级号（gradeId）和班级号（classId）。因为小学一共只有 6 个年级，所以年级号（gradeId）使用 3 个 bit 位保存就足够了，其他字段也是相同原理。

这样一来，因为在 C/C++ 语言中支持**位域操作**（即可以针对 bit 位来记录变量信息），我们就可以进一步缩减内存空间。而利用 bit 位来节省内存，正是嵌入式或高性能系统中重要的内存优化手段之一，但比较遗憾的是，Java 语言并不提供原生位域能力，因此直接使用 bit 位变量来压缩内存空间会有点不方便。

但是，Java 中有 **BitSet** 这个类型，它可以支持位操作，不过它的主要思想是通过压缩存储来节省空间开销。

比如，假设你要保存元素值为 64 以内且不重复的数组：

[1,3,5,6,10,11,12,25,44,56,2,55]，那么如果你使用正常 byte 数组来保存的话，可能需要

十几个字节才可以。而使用 BitSet，使用每一 bit 位来表示一个数字，那么用 8 个字节就可以记录很多个数字了（当然，Java 使用 BitSet 压缩存储的应用场景也并不只限于这种方式）。

不过这样问题也就来了：**针对 C/C++ 语言的位域优化实现，Java 是否也可以实现这样类似的功能呢？**

其实当然是可以的，但你可能需要借助**位运算**。这里我们来看一段 Java 代码示例，同样是实现类在一个字节保存 classID，gradeID，gender 等多个信息的能力，从中我们会观察到，在 Java 内也可以使用一个字节，来保存多个有效的字段信息。

[复制代码](#)

```
1 public class Student {
2
3     byte data; // |classID(4bit) |gradeID(3bit)| gender(1bit)|
4
5     Student() {
6         data = 0;
7     }
8
9     public void setGender(boolean isMale) {
10         data = (byte) (isMale ? (data | 0x01) : (data & 0xFE));
11     }
12
13     public boolean isMale() {
14         return (data & 0x01) == (byte) 1 ? true : false;
15     }
16
17     public byte getGradeId() {
18         return (byte) ((data & 0x0E) >> 1);
19     }
20
21     public void setGradeId(byte gradeID) {
22         data = (byte) (data | ((gradeID & 0x07) << 1));
23     }
24
25     public byte getClassId() {
26         return (byte) ((data & 0xF0) >> 4);
27     }
28
29     public void setClassId(byte ClassId) {
30         data = (byte) (data | ((ClassId & 0x0F) << 4));
31     }
32
33 }
```


当然，上面这种实现可以在一些极端性能场景下（比如有大规模实例对象的场景）使用，但我并不建议你经常使用，因为它会导致代码的实现复杂度提升。实际上，针对 Java 语言而言，在绝大部分的应用场景下，选择合适的变量类型就已经能很大程度上减少内存空间的浪费了。

对齐对象实例内的变量空间

好，我们再来了解下第二个手段：对象实例内变量空间对齐。

我们知道，如果一个变量的内存起始地址是字节对齐的，那么对这个变量的读取就是高效且安全的，因为不需要将变量分为多个指令周期进行读取和拼凑。

所以这里，我们来看下这个代码片段，这是一个 C/C++ 的结构体定义（这是不考虑 bit 位压缩存储的实现）：


 复制代码

```
1 struct Student
2 {
3     unsigned char flag;
4     unsigned int classId;
5     unsigned short gradeId;
6 };
```

其中，classId 的字段类型 int 长度为 4 个字节，所以如果起始地址没有 4 个字节对齐，那么程序在读取 classId 时，就会将其拆分在两个指令周期内完成，这样势必运行效率就很低下。

因此，**为了提升读取操作的性能，我们只能使用空间换时间的策略**。在 C/C++ 编译的过程中，一般会使用字节填充技术，在 char 类型后面填充 3 个无效字节，从而保证 classId 的起始地址是 4 个字节对齐的，这样即可实现高效读取（当然你也可以通过编译器指令显示，告知编译器不做这样的优化）。

但很明显，这样会带来一个副作用，就是**空间浪费**。而解决这个问题的手段，就是手动调整字段顺序来实现字节对齐。比如说，你可以把 classId，gradeId 放到结构体的前面来定义：

 复制代码

```
1 struct Student
2 {
3     unsigned int classID;
4     unsigned short gradeID;
5     unsigned char flag;
6 };
```

这样一来，我们就可以在满足字节对齐的场景下，最大化地节省内存空间。

比较幸运的是，JVM 在优化的过程中，使用了**字段重排优化技术**，这个技术是通过将相同类型的字段放在一起，来减少一些补白操作。所以在通常情况下，你只需要根据变量选择合适的字段类型即可。

尽量使用栈上的内存

OK，现在我们来学习下第三个手段：尽量使用栈上内存。

栈内存就是程序调用栈上使用的内存空间，**当调用栈退出之后就可以被自动回收重复利用**。在 C/C++ 中，所有的局部变量、结构体和类的实例，都可以在调用栈上临时分配。而在 Java 中，虽然不可以显式地在栈上去分配对象，但对于变量而言，你其实可以选择将它定义在函数内或者在类对象上，这样就可以避免在堆上分配资源。

这里需要说明的是，**如果尽量将变量定义在函数内，其实会对性能更加有利**。

因为在 Java 中，如果将变量定义在类对象中，当申请内存之后，由于内存回收需要依赖 GC 实现，因此可能在很长一段时间内，这块内存都不能再被回收利用了。另外，虽然在 JVM 编译优化中，会有些栈上分配的优化机制，但它们需要满足很多条件才可以实现，所以在开发代码时并不能完全依赖这个机制。

OK，以上就是通过调整代码实现，来优化内存空间与布局的常用方法和思路，它们的目标都是基于缩减内存空间来实现优化性能的效果，不过仅通过这个方式来提升内存效率还是不够的。

所以接下来，我们再来看看针对内存的申请与释放这一实现方式来说，还有哪些手段也可以提升软件性能。

内存的申请与释放优化

对 Java 语言而言，针对一个对象的 new 操作是非常耗时的操作，不仅需要动态在堆空间上分配内存并进行初始化，而且在使用结束之后，还需要基于 GC 来管理跟踪释放流程。


那么针对内存的申请和释放过程，我们可以通过编码实现来优化它的性能吗？

答案当然是可以的，这里常规的优化思路主要有三点，下面我就给你详细介绍下。

调整内存申请释放发生的时间点

首先是调整内存申请释放发生的时间点。这个优化思路的出发点是：**将内存申请操作提前到软件程序的启动阶段中，从而减少运行期间申请内存资源的开销。**

这里我们同样来看一个代码片段，这是一个单例模式的示例：

 复制代码

```
1 public class Singleton {  
2  
3     private static final Singleton instance = new Singleton();  
4     private Singleton() {  
5     }  
6     public static Singleton getInstance() {  
7         return instance;  
8     }  
9 }
```

从中我们可以观察到，程序在类初始化的过程中会自动去创建对象实例，这样就可以实现两个优化点：程序在运行期间直接使用对象，而不需要再进行动态申请内存；同时，由于静态对象实例也不会被 JVM 的 GC 垃圾回收，所以在一定程度上减少了 GC 的负荷。

实际上，这个例子也可以认为是预计算性能模式在内存申请场景下的一个应用示例，也就是把内存申请操作提前到启动运行阶段，从而减少运行期的动态申请开销（关于性能优化模式的更多内容，你可以回顾第 [9](#)、[10](#) 讲）。

减少内存的申请与释放次数

好了，除了调整内存申请发生的时机之外，我们还可以通过减少内存的申请与释放次数来优化性能。

我们知道，在 Java 语言中，一共包含了 8 种基本类型，分别是 byte、short、int、long、float、double、boolean、char。

那么首先，在业务允许的情况下，我们会**优先使用基本类型**，避免使用包装类型（如 Integer 等），这样可以减少额外的内存申请操作。

其次，在编码过程中，我们也要**避免写一些冗余的对象申请操作**，如下代码所示：

```
1    String str = "test ";
2    String str_2 = new String("test ");
```

[复制代码](#)

可以看到，第一行代码里的 str 并不会动态申请对象，而第二行的 str_2 则额外申请了一次内存，从而就会导致额外的性能损耗。

最后，在 Java 中也是最重要的一点，就是**尽量使用预分配方式**。比如，你可以看看下面给出的这段代码示例，其核心逻辑就是将内存提前申请好，以避免支出运行期动态申请空间的开销：

```
1    StringBuilder sb = new StringBuilder(1024);
2    sb.append("test append alloc")
```

[复制代码](#)

如果你在 StringBuilder 的构造函数中，通过参数提前指定了空间大小，并且已经一次性分配好，那么就可以避免在调用 append 的操作中，触发额外的内存申请操作，进而就可以提升性能。而且，这种预分配空间的机制，在 Java 的各种数据结构类型中，如 ArrayList、HashMap 等的使用过程中，你都可以去使用。

定制化内存申请与释放实现

好，最后一种优化手段，就是定制化内存申请与释放实现。也就是说，针对特定的业务场景，你可以根据使用内存的方式与特点，单独定义一套内存申请和释放的算法实现，来提升软件在内存申请与释放操作的性能。

在动态内存申请与释放的实现过程中，我们需要注意以下几个地方：

首先，需要一个查找算法来寻找合适的内存大小空间；

其次，需要考虑在并发场景下，通过同步手段来保证线程安全性；

最后，要注意当堆空间不足时，有可能会触发内核态 API 调用，从而造成内核态与用户态切换的更大开销（目前不确定在 JVM 实现中是否会发生这个阶段，但在 C/C++ 中会存在）。

实际上，在我以往参与的很多 C/C++ 开发高性能的系统中，定制化内存申请与释放是一个非常重要的性能优化手段。举个简单的例子，针对固定大小的内存申请操作，我们可以基于队列的出队和入队等类似算法实现，来改善内存申请与释放速度。

不过，在一些高实时性嵌入式系统中，动态内存使用是被禁止的，程序的所有内存都需要通过静态预分配与管理来实现。

可是在 Java 中，如果一些业务逻辑内，频繁地申请和释放对象操作对性能的影响比较大，那么我们是否有办法去优化解决呢？

当然是有的！这里我想给你介绍的一个性能优化手段，就是**对象池共享技术**。

对象池本质上是通过集合来管理已经申请过的对象，如果线程需要这种类型的对象，就直接从集合中取一个元素，但是使用完也一定要归还，否则就会造成内存泄漏。另外，在使用对象池来管理一些创建比较耗时的对象时，我们还可以通过减少对象申请与创建的过程，来提高软件的运行速度和性能。

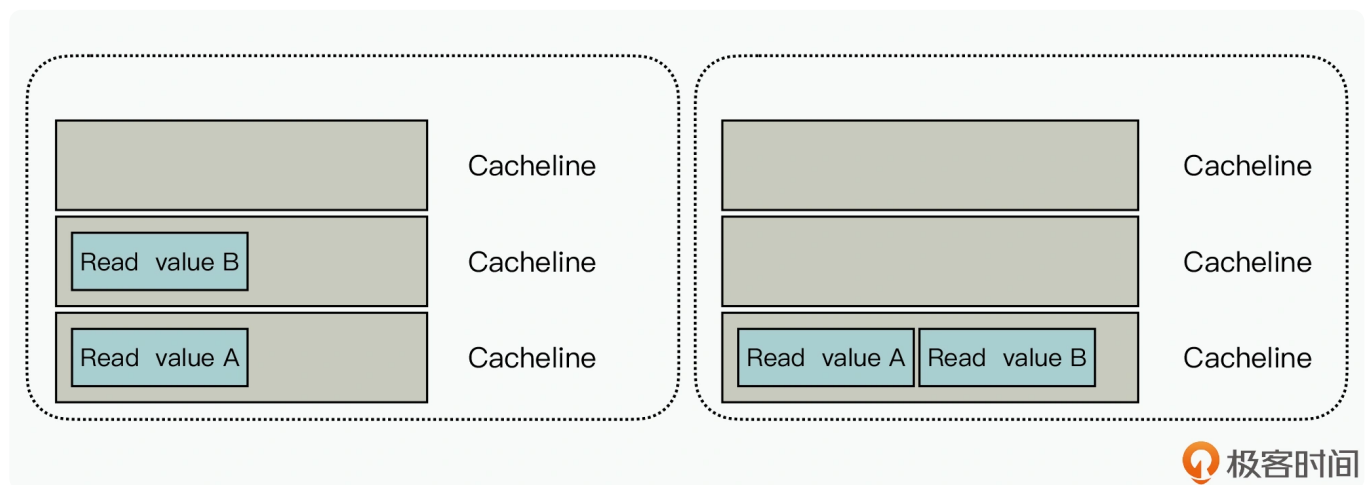
好了，简而言之，对于内存的申请与释放优化，就是通过减少内存的申请和释放操作，或提升内存申请和释放的速度，来达到提升软件运行性能的目的。那么下面，我们接着来看看，要如何优化内存的读取和修改，来进一步提升软件性能。

内存的读取与修改优化

在 CPU 中，Cache 的存取与替换都是以缓存行（Cacheline）为单位，而且在不同的 CPU 体系架构实现中，缓存行的长度也都不一样，具体从 8 字节到 128 字节之间不等。因此，如果说我们可以**把变量空间按照缓存行对齐**，那么就可以提升 Cache 的读写效率，从而就能够达到提升性能的效果。

既然如此，具体我们该怎么做呢？

这里我们先来看一个例子，这个例子主要用来说明：变量 A、变量 B 是否在一个 Cacheline 中，并会对 Cache 的读取操作产生影响。

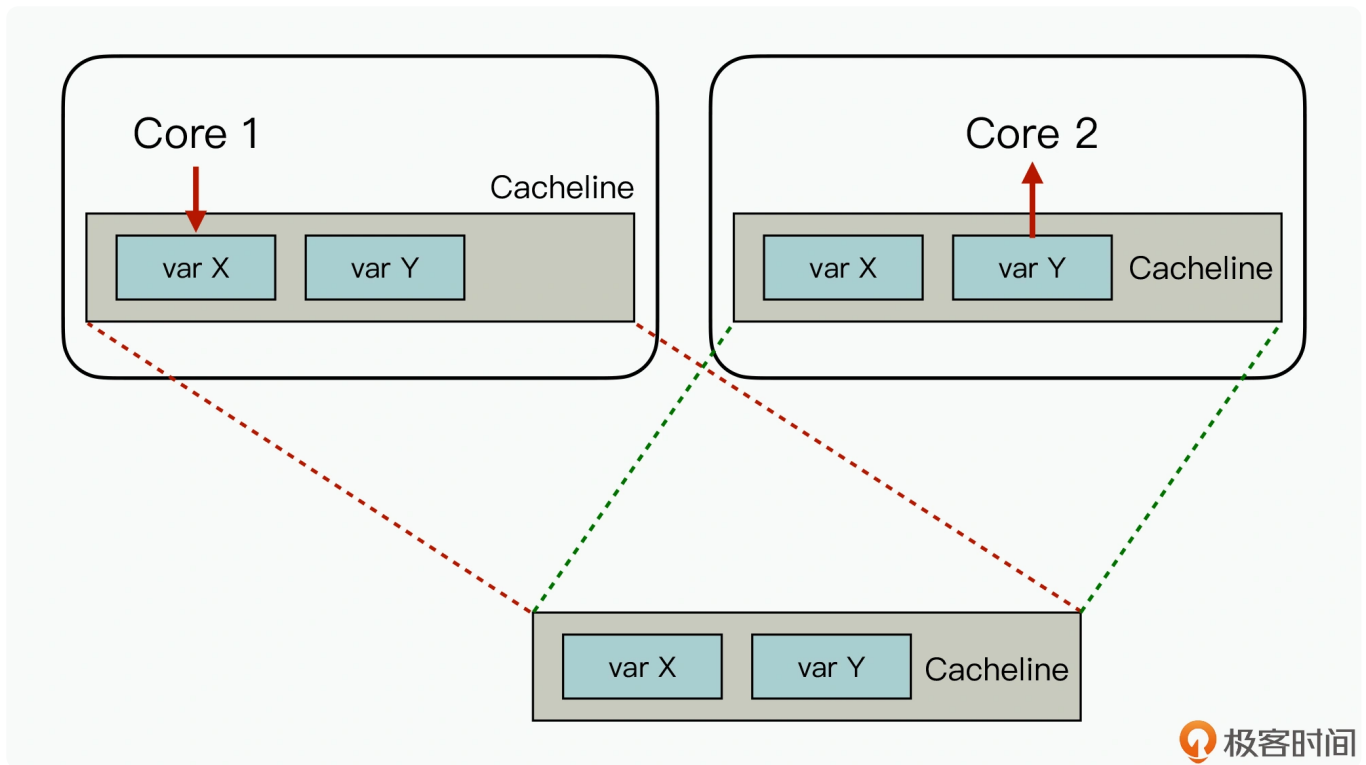


如上图的左半部分所示，如果两个变量不在一个 Cacheline 中，那么在读取变量 A、变量 B 时，就需要读取两个 Cacheline。而在图的右侧，由于两个变量在一个 Cacheline 中，所以只需要读取一个 Cacheline 即可。

也就是说，你在编码的过程中，就需要**充分利用局部性原理**，把经常一起使用的变量放在一起，从而最大化地实现 Cacheline 的长度对齐，来优化提升软件的运行效率。

注意：由于 CPU 的**指令预取技术**，通常情况下在串行程序中，Cacheline 对齐对性能的影响可能不是那么明显，所以很容易被我们忽视。

好，除此之外，我们还要知道在多核并发的场景下，由于 Cacheline 没有对齐，造成的**伪共享**（False Sharing），也会显著地影响程序的运行效率。如下图所示：



已知在 Core 1 上需要更新变量 X，而 Core 2 需要读取变量 Y。但是由于变量 X 与 Y 在一个 Cacheline 中，它们会映射在相同的内存地址上，所以每次当 Core1 上更新变量 X 之后，就会造成 Core 2 上的变量 Y 对应的 Cacheline 失效，需要重新读取，进而就会影响性能。

所以在并发系统的设计中，针对 Cacheline 未对齐造成软件性能受影响的场景，我们可以通过**显式字段的冗余**来实现 Cacheline 对齐，以避免这种情况的发生。不过好在，Java 8 中引入了 **sun.misc.Contended** 注解，它就可以针对性地识别并发场景下存在的 Cacheline 伪共享问题。

那么，除了伪共享问题外，在实现编码的过程中，还有一些手段也可以优化内存读取和修改的性能。

比如说，在 **Java 语言** 中，你可以借助一些 **native 方法** 来优化拷贝赋值数据的性能，而其中最常用的，就是使用接口的 `System.arraycopy` 方法、对象的 `clone` 方法。那么再进一步，对内存拷贝的性能优化极限就是**零拷贝**，你可以通过业务逻辑或算法优化来尽量减少拷贝操作，从而进一步优化性能。

而对于 **C/C++ 语言** 来说，对一块内存进行修改时，使用 **memcpy 操作** 性能则优于直接赋值操作，这是因为 `memcpy` 在汇编过程，使用了特殊的汇编指令来优化连续内存的拷贝操作。

小结

学完了这节课，我们需要明确一点，就是不同编程语言在如何高效地使用内存上，存在不同的权衡策略，但不管是静态类型语言（如 C/C++、Java 等），还是动态类型语言（如 Ruby、Python、Node.js 等），高效使用内存都是编码性能优化的重要考量因素之一。

在面对不同编程语言进行业务编码的过程中，你都可以充分利用编程语言内置的内存使用策略，并从内存空间与布局优化、内存申请与释放优化、内存读取与修改优化三个维度进行思考，从而就能够写出高性能的代码。

思考题

代码在运行期间，也是从内存中加载到 Cache 来运行的吗？你知道还有哪些手段，可以优化代码逻辑使用内存的效率呢？

欢迎在留言区分享你的思考和看法。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 13 | 编译期优化：只有修改业务代码才能提升系统性能？

下一篇 15 | 并发实现：掌握不同并发框架的选择和使用秘诀

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 (1)

写留言



Geek_094d56

2021-06-23

老师：请问c++怎么减少伪共享呢？

展开

作者回复: 你可以考虑将共享内存统一规划使用。