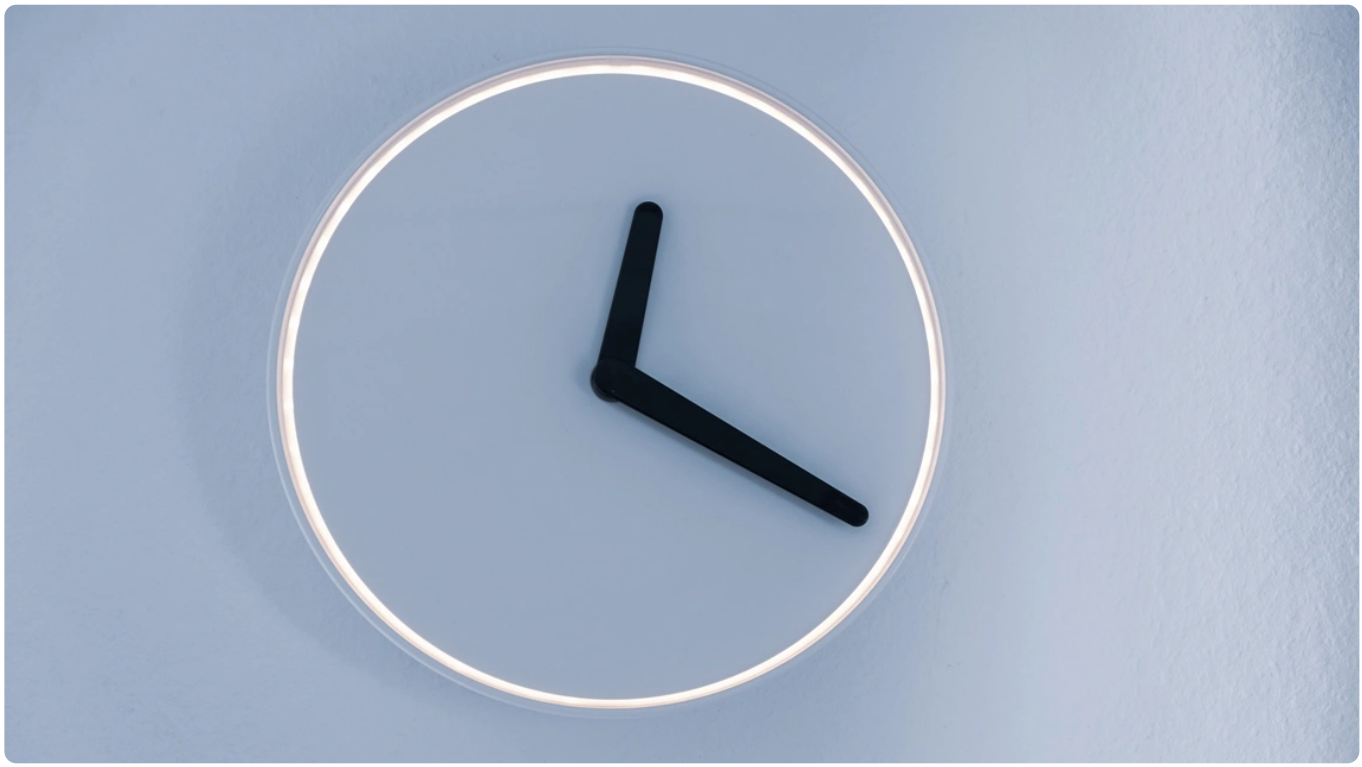
10 | 性能模式（下）：如何解决核心的性能问题？

2021-06-08 尉刚强

《性能优化高手课》 课程介绍 >



讲述：尉刚强
时长 14:18 大小 13.10M

你好，我是尉刚强。今天我们接着上节课的话题，继续来学习另外 4 种性能模式，分别是预计算模式、耦合模式、搬移计算模式以及丢弃模式。

现在我们已经知道，性能模式是为了提升性能指标，针对软件设计与实现的一种调整方法和手段。理解了这些性能模式，我们就能够在优化系统性能的过程中，快速找到调整设计实现的出发点与思路。

在开始这节课的学习之前，我还想给你强调两点：

首先，与设计模式一样，每种性能模式都只是解决特定业务场景下的性能问题，如果你使用不当，很有可能会取得反效果。所以你一定不要局限于这几种性能模式，而是要掌握这种解决性能问题的思路。

其次，基于性能模式对软件设计实现的调整，它带来的性能收益其实并不是确定的。因此，在做调整优化前，你需要通过测试获取性能提升收益的准确数据后，再去权衡考虑是否真的需要修改，这样也有利于节省成本。

好了，接下来我就从预计算模式开始，来带你了解下它的设计原理和工作机制。

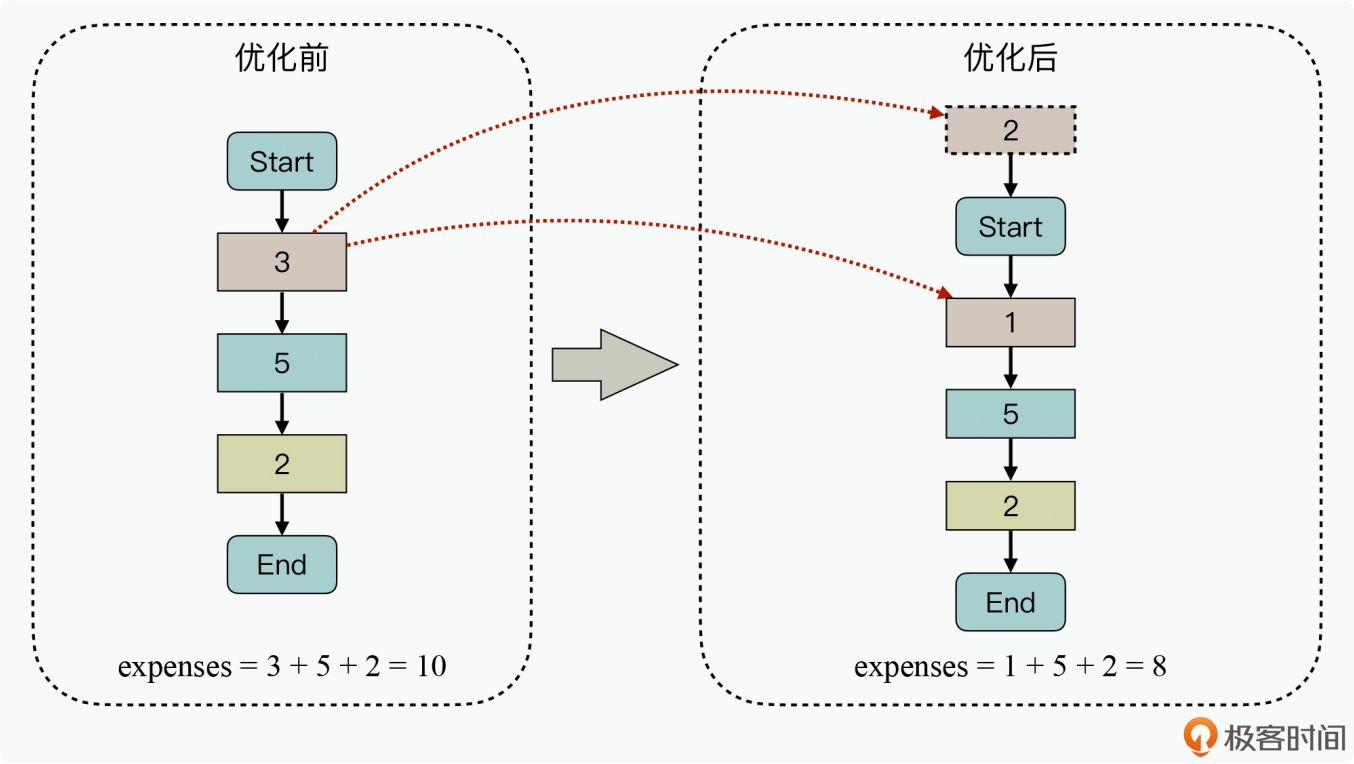
预计算模式

我们可以先来试想下这样的场景：小明喜欢在家吃早餐，但又不想太早起床，所以他选择了前天晚上就把菜洗好切好，这样早上起来直接炒一下就可以吃了，节省了早上要花费的时间，可以多睡一会儿懒觉。

那么回到软件实现的业务逻辑中，是否也有一些计算逻辑可以提前执行呢？

当然有，下面我要介绍的预计算模式，就是通过挖掘出提前计算的业务逻辑，并在程序启动前执行完毕，从而有效地提升了业务的处理速度。

好，现在我们来看下预计算性能模式的具体工作流程：



在图上左侧优化前的代码中，第一个矩形代码块的执行开销为 3。通过分析重复的计算逻辑，发现其中一部分的计算逻辑可以提前执行。针对这种场景，我们就可以将这部分计算

从业务中剥离掉，使用预计算模式提前到程序启动前来执行。

现在我们来看一个具体的例子。如下所示的代码示例中，实现的功能是根据员工请假天数来计算当月薪水，其中 calcSalaryBeforeOptimize 代表优化前的实现，calcSalaryAfterOptimize 代表优化后的实现：

复制代码

```
public class ClacOptimze {
    public static int calcSalaryBeforeOptimize(int leaveDay) {
        int fullSalary = 12000;
        if (leaveDay <= 1) {            //使用条件判断，潜在分支预测失败
            return fullSalary;
        }
        if (leaveDay < 5) {
            return fullSalary - leaveDay * 400; //使用了乘法运算逻辑
        }

        return fullSalary - leaveDay * 800;

    }

    final static int[] salarys = { 12000, 12000, 11800, 11600, 11400, 11200, 1
    public static int calcSalaryAfterOptimize(int leaveDay) {
        return salarys[leaveDay];  //这里使用查表
    }
}
```

可以看到，在优化前的代码实现中，需要进行多次 if 判断，还需要进行乘法运算；而优化后的代码中，每次运行只需要查表就可以返回，执行速度会快很多。

实际上，以上代码示例所使用的预计算模式，采用的策略是**通过空间换时间，这是预计算性能模式实现过程中比较常见的一种方式**。

在通常情况下，针对预计算工作量比较小的方式，我们完全可以手工计算，但当计算量比较大时，我们可能还需要开发单独的针对预计算的程序。当然，预计算模式并不是只有空间换时间的实现方式，还有很多种实现并不会带来额外的内存开销，比如业务中内存的预申请、业务数据的预初始化，等等。

另外，还有一些编程语言提供了编译期计算的能力，针对这种场景，我们也可以**将计算逻辑提前到编译期执行，来减少运行期的时间开销**。比如 C++ 的常量表达式、模板泛型编程

等，都提供了比较强大的编译期计算能力。

这里我给你举个真实的例子。我曾经参与过一个 SaaS 服务时延的优化项目，就使用过多次在数据库中添加冗余数据来记录预计算结果，从而减少了业务处理运行期开销，达到降低时延的效果；此外，在嵌入式实时性的优化中，我们还通过挖掘业务中所有预计算逻辑，多次帮我们大幅度提升了产品性能。

不过在使用预计算模式时，你还**需要注意一点**，就是当需要对计算逻辑进行比较大的调整时，你需要进行完备的测试，以免引入新的故障。

耦合模式

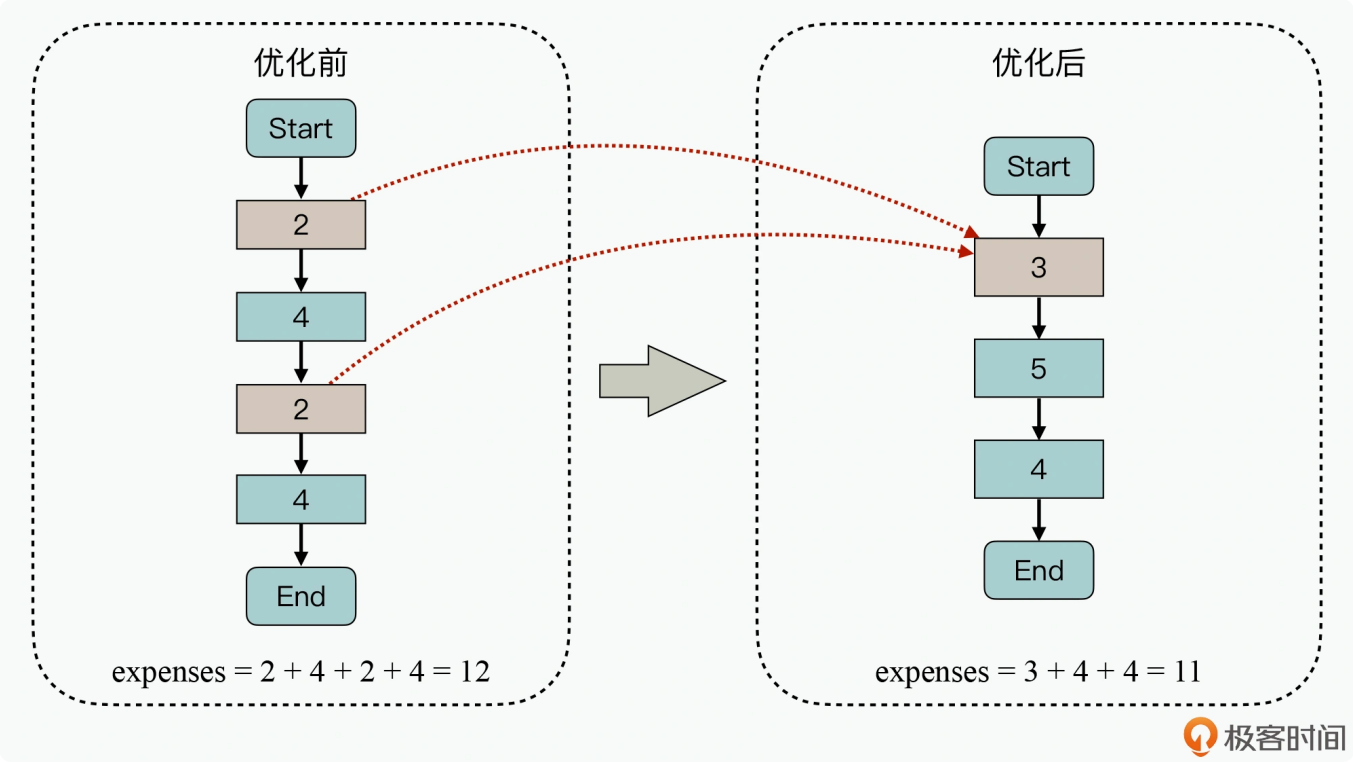
好，我们接着来看看耦合模式。

这种性能模式的原理其实非常简单，我就拿出行服务来给你举个例子。我们知道，出租车司机在开车运营期间，喜欢选择拼车模式同时接送多位乘客，因为当乘客路线重合比较多的时候，他们就可以获得更大的现金收益，而这就是使用了耦合模式的解决思路。

所以，**耦合模式的意思就是当你做一件事情的时候，不要把目光单独停留在这一件事情上，你还可以思考下是不是可以顺带把其他事情也一并处理掉。**

这里你可能马上就会想到，这与面向对象设计原则中的“单一职责原则”有冲突啊？的确，耦合模式在一些场景下会与单一职责存在冲突（单一职责推荐一个方法只实现一个功能，而耦合模式需要一个方法内同时实现多个功能），所以**我更推荐你只在性能影响权重比较大的关键场景中使用它。**

现在，我们先来了解下耦合性能模式的优化过程：



可以看到，图中左侧粉色的两个代码块是相对独立的，执行开销分别为 2，在优化过程中将两个代码块逻辑合并到一起后，执行总开销变为了 3。这样在使用这种方式优化后，系统的总执行开销就从原来的 12 降低到了 11，处理时延也就降低了。

这里我们来看一个具体的例子。下面是一个 Java 使用 MyBatis 访问数据库场景的代码片段，其中 UserMapperBeforeOptimize 代表的是优化前访问数据库的接口，UserMapperAfterOptimize 则代表优化后访问数据库的接口。

> 注：代码中我省略了很多关于数据库的相关配置与代码，因为这对理解耦合模式并无太大帮助。

复制代码

```
1  public class User {
2          private String name;
3          private Integer age;
4
5          public String getName() {
6          return name;
7          }
8
9          public void setName(String name) {
10         this.name = name;
11         }
12
13         public Integer getAge() {
```

```
14          return age;
15          }
16
17          public void setAge(Integer age) {
18          this.age = age;
19          }
20
21  }
22
23  public interface UserMapperBeforeOptimize {
24      public String findNameById(String Id);  //单一职责接口
25      public String findAgeById(String Id);   //单一职责接口
26  }
27
28  public interface UserMapperAfterOptimize {
29      public String findNameById(String Id);  //单一职责接口
30      public String findAgeById(String Id);  //单一职责接口
31      public User FindUserById(String Id);  //新增的获取多个字段的耦合接口
32  }
```

可以发现，优化之前的接口中包含了两个方法：根据 ID 获取名字、根据 ID 获取年龄。当很多的客户代码同时需要获取名字和年龄时，就可以通过在接口中增加一次性返回姓名和年龄信息的方法，来减少业务两次访问数据库带来的网络和查询的额外开销。

耦合模式的应用场景比较多，比如说：

> 在数据库设计的过程中，基于性能考虑，我们可以将多个表中的字段信息融合记录到一个大表内，从而实现原来需要多次查询操作，变成一次查询就全部获取。
>
> 在微服务接口设计中，通常 REST 接口并不是正交的，其中会包含一些基本功能接口和一些复合功能接口。而在一些典型的性能优化场景下，使用复合接口就可以一次实现原来多个基本功能接口请求的功能，从而就能通过减少 REST 接口调用次数来优化性能。
>
> 在嵌入式场景中，子系统间交互使用的 TLV（Tag、Length、Value）数据结构类型，也是典型的耦合模式的应用。

另外，**耦合模式也并不局限在接口层面，你也可以在计算逻辑中去使用**。同时你需要注意，在实现包含复合功能的接口与业务逻辑时，不建议删除掉原来的单一功能实现，这是为了防止对只使用简单接口与功能的客户带来额外的开销。

搬移计算模式

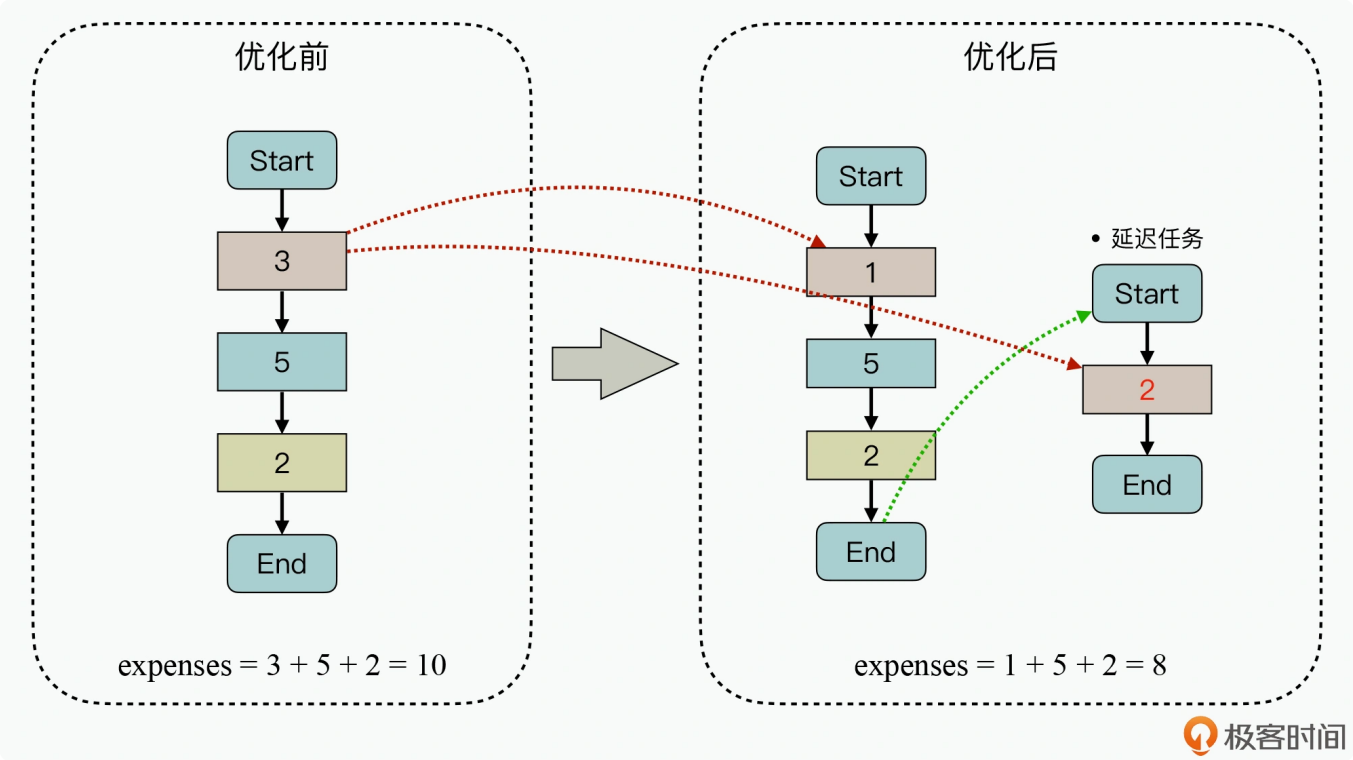
不论在工作还是生活中，你可能很擅长基于时间来统筹安排每个时间段中要做的事情。那么在软件的业务计算过程当中，你同样可以基于性能和效率考量，去调整安排计算逻辑在运行时间与物理位置上的分布。

在实时高性能的系统中，软件工程师在设计时通常会把业务逻辑划分为关键路径和非关键路径。而我们知道，系统时延在更大程度上取决于关键业务逻辑的处理时延。

所以，如果你可以**把计算逻辑从关键路径搬移到非关键路径，就可以提升产品的性能。**

在使用搬移计算模式来调整计算业务逻辑的过程中，你要重点关注的是处理时延的性能提升。但你也要注意，在这样优化处理之后，其实系统的总负荷通常并没有减少。所以，你可以在系统核心目标是追求用户侧的时延最小化的业务场景中，选择使用这种性能模式。

下面我们就来看一下搬移计算模式的具体实现流程：



在图中左侧第一个矩形执行代码块中，我们通过分析业务流程和度量数据，发现部分业务逻辑可以推后计算，于是把这部分业务拆分到另外一个任务中去执行，从而就减少了客户关注的处理时延。

这里我给你举一个真实的使用案例。在互联网 SaaS 服务中，对我们这样的普通用户而言，会对请求的响应时延十分感兴趣，而并不关注服务器处理业务的全部处理时间，所以

这时候，我们就可以使用搬移计算模式来进行优化。因此，我曾经就在这类项目的优化过程中，引入了延迟计算服务，并通过对业务逻辑优化，剥离了非关键业务，交给延迟计算任务进行处理，从而实现了在短时间内，将时延性能指标提升到 40% 以上的目标。

而除了与 SaaS 服务相关的业务场景，在**嵌入式场景和云服务场景**中，使用搬移计算模式优化性能也都能取得很好的效果。但是你需要认识到，**搬移计算的设计与实现其实容易引起整个系统的复杂度提升，进而也容易引入额外的故障**，所以你在引用这种性能模式时，一定要注意进行充分的功能验证与性能优化的提升分析。

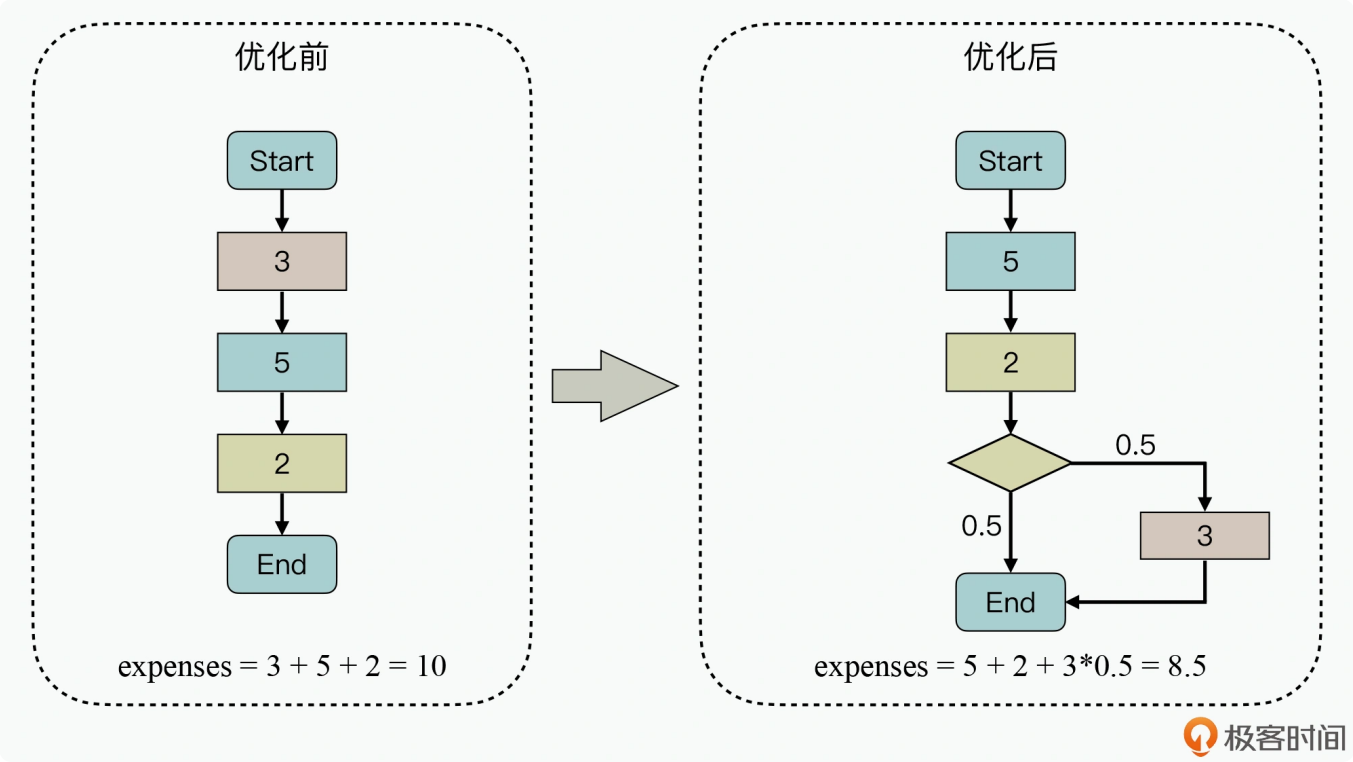
丢弃模式

好，现在我们来学习下最后一种性能模式，也就是丢弃模式。

你可以先来试想这样一种场景：在正常情况下，你每天早上起来上班前，都会洗脸、刷牙，然后再带上电脑出门；而当某天起来晚了，你可能就会把洗脸、刷牙这步省掉了，直接背着电脑出门。

所以发现没有，我们做的很多事情在特定场景下其实都是可以丢弃的。

那么回到软件业务处理的过程中，也是一样的道理。在软件系统中，我们在一些特殊的场景下使用丢弃模式，就可以达到非常好的性能优化效果，这里我们先来了解下它的具体工作流程：



在上图的左侧部分，可以看到第一个矩形代码块的执行开销为 3，而我们通过分析业务流程和度量数据，发现这个功能的优先级比较低。因此可以使用的优化策略就是：把优先级比较低的代码块放到业务最后，在极端场景下，也可以通过直接丢弃不处理来保证系统性能不恶化。

丢弃模式，在实时嵌入式的场景中使用的比较多，这种模式非常好理解和操作。你在实际的业务场景下，要注意先识别出业务中的非关键部分逻辑，确认其支持可关闭，这样当系统处于超负荷运行时，就可以直接将这部分业务停掉。

小结

通过这两节课的介绍和讲解，现在你应该就能理解这八种性能模式的核心优化思想了。这里我想提醒你一点，在前面的代码示例中，我使用的是非常小的代码来进行演示的，因此并不建议你在实际代码实现的细粒度级别中去使用这些代码例子。

最后我想说，性能模式是软件优化中非常重要的手段之一，但是很多的性能优化工程师，目光只停留在编译层面的优化性能，而忽略了最直接且高效的性能模式。所以我希望，你在碰到具体的性能问题与挑战时，可以运用今天学到的性能模式，并可以在较大的业务粒度上使用，来帮助产品提升性能竞争力。

思考题

在你以往参与的性能优化项目中，有没有也使用过一些巧妙的处理模式，却带来了很明显的性能提升呢？

欢迎在留言区中分享你的答案，我们一起交流讨论。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。

上一篇 09 | 性能模式（上）：如何有效提升性能指标？

下一篇 11 | 如何针对特定业务场景设计数据结构和高性能算法？

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。