

Flutter（十二） Dart的垃圾回收器

AlanGe

在学习 Flutter 的过程中，我们知道 Widget 只是最终渲染对象（RenderObject）的配置文件，它会在 build 的时候频繁的销毁和创建，那么，我们不需要担心他的创建和销毁带来的性能问题吗？

其实大可不必，因为 Dart 针对 Flutter 的 Widget 的创建和销毁专门做过优化，这也是 Flutter 在多种语言中选择 Dart 的一个重要因素，甚至我们还可以刻意利用这一点。

Garbage 垃圾；Collector 收集器

下面这篇文章解析了 Dart 的 GC（Garbage Collector），对它做了个翻译以及部分内容的解析，包括一些排版，有不对的地方大家多多指正。

原文地址：<https://medium.com/flutter/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30>

Flutter 使用 Dart 作为开发语言和运行时机制，Dart 一直保留着运行时机制，无论是在调试模式（debug）还是发布模式（release），但是两种构建方式之间存在很大的差异。

- 在调试模型下，Dart 将所有的管道（需要用到的所有配件）全部装载到设备上：运行时，JIT（the just-in-

time) 编译器/解释器 (JIT for Android and interpreter for iOS) , 调试和性能分析服务。

- 在发布模式下, 会除去 JIT 编译器/解释器 依然保留运行时, 因为运行时是 Flutter App 的主要贡献者。



图片

Dart 的运行时包括一个非常重要的组件: 垃圾回收器, 它主要的作用就是分配和释放内存, 当一个对象被实例化 (instantiated) 或者变成不可达 (unreachable) 。

在Flutter运行过程中，会有很多的Object。

- 在StatelessWidget在渲染前（其实上还有StatefulWidget），他们被创建出来。
- 当状态发生变化时候，他们又会被销毁。
- 事实上，他们有很短的寿命（lifespan）。
- 当我们构建一个复杂的UI界面时，会有成千上万这样的Widgets。

所以，作为Flutter开发者，我们需要担心垃圾回收器不能很好的帮助我们管理这些吗？（是不是会带来很多的性能问题呢）

- 当Flutter频繁的创建和销毁这些Widget（Objects），我们是否需要很迫切的限制这种行为呢？
- 非常普遍，对于新的Flutter开发者来说，当一个Widget的状态不需要改变时，他们会创建引用的Widget，来替代State中的Widget，以便于不会被销毁或者重建。

不需要这样做

担心Dart的GC是没有任何事实根据的（没有必要），这是因为它分代（generational）架构和实现，可以让我们频繁创建和销毁对象有一个最优解。在大多数情况下，我们只需

要 Flutter 引擎按照它的方式创建和销毁这些 Widgets 即可。

Dart 的 GC

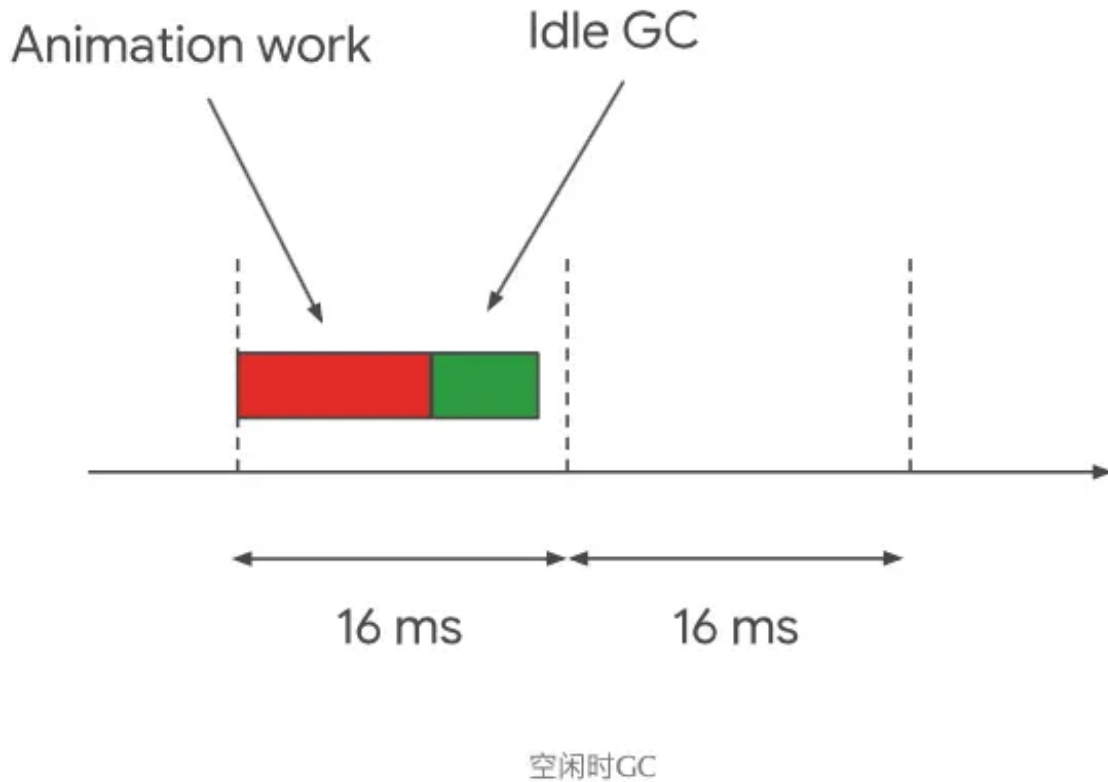
Dart 的 GC 是分代的 (*generational*) 和由两个阶段构成：
the young space scavenger (scavenger 针对年轻一代进行回收) and parallel mark sweep collectors (sweep collectors 针对老一代进行回收)

注解：事实上 V8 引擎也是这样的机制

调度安排 (Scheduling)

为了让 RG 最小化对 App 和 UI 性能的影响，GC 对 Flutter 引擎提供了 hooks，hooks 被通知，当 Flutter 引擎被侦测到这个 App 处于闲置的状态，并且没有用户交互的时候。这就给了 GC 一个空窗期来运行它的手机阶段，并且不会影响性能。
收集

垃圾收集器还可以在那些空闲间隔内进行滑动压缩 (sliding compaction)，从而通过减少内存碎片来最大程度地减少内存开销。



阶段一： Young Space Scavenger Scavenger 清洁工

这个阶段主要是清理一些**寿命很短**的对象，比如 **StatelessWidget**。当它处于阻塞时，它的清理速度远**快于**第二代的mark、sweep方式。并且结合调度，完成可以**消除**程序运行时的暂停现象。

本质上来讲，对象在内存中被分配一段**连续的、可用的内存**空间，直接被分配完为止。Dart使用 **bump pointer**（注解：**bump 碰撞，隆起**）如果**像 malloc 一样，维护 free_list 再分配，效率很低。**）分配新的空间，处理过程非常快。

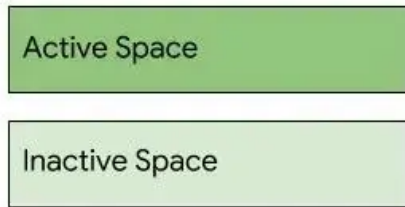
分配了新对象的新空间，被为两部分，称之为 semi semi 半挂车

spaces。一部分处于活动状态，另一部分处于非活动状态。新对象分配在活动状态，一旦填充完毕，依然存活的 Object，就会从活动状态 copy 到非活动状态，并且清除死亡的 Object。这个时候非活动状态变成了活动状态，上面的步骤一次重复。（注解：GC 来完成上面的步骤）

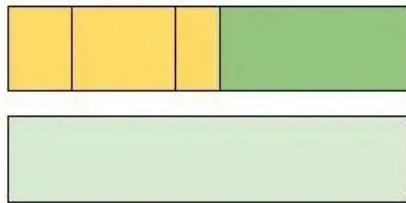
为了确定哪些 Object 是存活的或死亡的，GC 从根对象开始检测它们的应用。然后将有引用的 Object（存活的）移动到非活动状态，直接所有的存活 Object 被移动。死亡的 Object 就被留下；

有关此的更多信息，请查看 Cheney 算法。

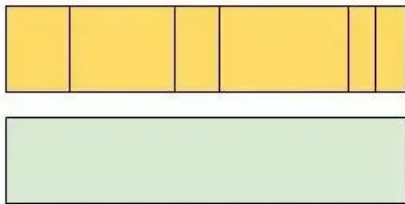
1: Semispaces



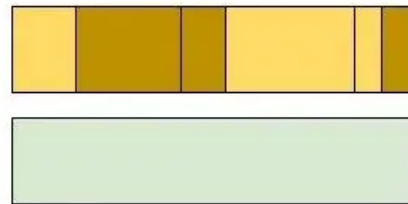
2: Objects allocated in active



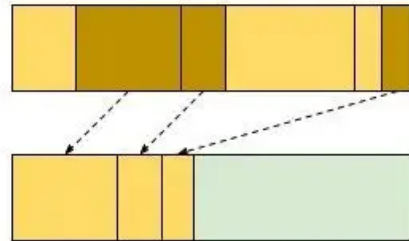
3: Active space fills



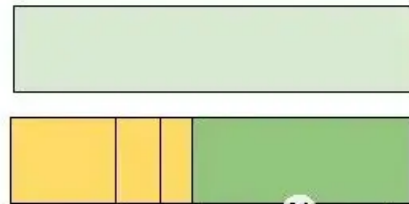
4: Live objects determined




5: Live objects moved



6: Spaces swap active state



 coderwhy

图片

阶段二：Parallel Marking and Concurrent Sweeping

当对象达到一定的寿命（在第一阶段没有被 GC 回收），它们将被提升由第二收集器管理的新内存空间：mark-sweep。

这个阶段的 GC 有两个阶段：第一阶段，首先遍历对象图

(the object graph) ，然后标记人在使用的对象。第二阶段，将扫描整个内存，并且回收所有未标记的对象。

这种 GC 机制在标记阶段会阻塞，^然不能有内存变化和 UI 线程也会被阻塞。但是由于短暂的对象在 Young Space Scavenger 阶段以及被处理，所有这个阶段非常少出现。不过由于 Flutter 可以调用收集时间，影响的性能也会被降到最低。

但是如果引用程序不遵守分代的机制，反而这种情况会经常发生。但是由于 Flutter 的 Widget 的机制，所有这种情况不经常发生，但是我们还是需要了解这种机制。

Isolate

值得注意的是，Dart 中的 Isolate 机制具有私有堆的概念，彼此是独立的。每个 Isolate 有自己单独的线程来运行，每个 Isolate 的 GC 不影响其他线程的性能。使用 Isolate 是避免阻塞 UI 和减轻密集型任务的好方法（注解：耗时操作可以使用 Isolate）。

总结

到这里你应该明白：Dart 使用了强大的分代 GC，以最大限

度的减少 Flutter 中 GC 带来的性能影响。

所以，你不需要担心 Dart 的垃圾回收器，这个反而是我们应用程序的核心所在。