个人技能

学会 Debug

调试（Debug）是作为一个程序员的基石。调试这个词第一个含义即是移除错误，但真正有意义的含义是，通过检查来观察程序的运行。一个不能调试的程序员等同于瞎子。

那些认为设计、分析、复杂的理论或其他东西是更基本的东西的理想主义者们不是现实的程序员。现实的程序员不会活在理想的世界里。即使你是完美的，你周围也会有，并且也需要与主要的软件公司或组织，比如 GNU，或者与你的同事，写的代码打交道。这里面大部分的代码以及它们的文档是不完美的。如果没有获得代码的执行过程可见性的能力，最轻微的颠簸都会把你永远地抛出去。通常这种可见性只能从实验获得，也就是，调试。

调试是一件与程序运行相关的事情，而非与程序本身相关。你从主要的软件公司购买一些产品，你通常不会看到（产品背后的）程序本身。但代码不遵循文档这样的情况（让你整台机器崩掉是一个常见又特殊的例子）或者文档没有说明的情况仍然会出现，不可避免的，这意味着你做的一些假设并不对，或者一些你没有预料到的情况发生了。有时候，神奇的修改源代码的技巧可能会生效。当它无效时，你必须调试了。

为了获得一个程序执行的可见性，你必须能够执行代码并且从这个过程中观察到什么。有时候这是可见的，比如一些正在呈现在屏幕上的东西，或者两个事件之间的延迟。在许多其他的案例中，它与一些不一定可见的东西相关，比如代码中一些变量的状态，当前真正在执行的代码行，或者是否一些断言持有了一个复杂的数据结构。这些隐藏的细节必须被显露出来。

通常的（一些）观察一个正在执行的程序的内部的方法可以如下分类：

- 使用一个调试工具；
- Printlining - 对程序做一个临时的修改，通常是加一些行去打印一些信息；
- 日志 - 用日志的形式为在程序的运行中创建一个永久的视窗。

当调试工具稳定可用时，它们是非常美妙的，但 Printlining 和写日志是更加重要的。调试工具通常落后于编程语言的发展，所以在任何时间点它们都可能是无效的。另外，调试工具可能轻微改变程序实际执行的方式。最后，调试有许多种，比如检查一个断言和一个巨大的数据结构，这需要写代码并改变程序的运行。当调试工具可用时，知道怎样使用调试工具是好的，但学会使用其他两种方式是至关重要的。

当需要修改代码时，一些初学者会害怕调试。这是可以理解的，这有点像探索型外科手术。但你需要学会打破代码，让它跳起来，你需要学会在它上面做实验，并且需要知道你临时对它做的任何事情都不会使它变得更糟。如果你感受到了这份恐惧，找一位导师 - （否则）在许多人面对这种恐惧的脆弱的开始时刻，我们会因此失去很多优秀的程序员。

如何通过分割问题空间来 Debug

调试是有趣的，因为它一开始是个迷。你认为它应该这样做，但实际上它却那样做。很多时候并不仅是这么简单---我给出的任何例子都会被设计来与一些偶尔在现实中会发生的情况相比较。调试需要创造力与智谋。如果说调试有简单之道，那就是在这个谜题上使用分治法。

假如，你创建了一个程序，它会在一个序列里做十件事情。当你运行它的时候，它崩溃了。因为你写的代码并不想让它崩溃，所以现在你有一个谜题了。当你查看输出时，你可以看到序列里前七件事情运行成功了。最后三件事情在输出里却看不到，所以你的谜题变小了："它是在执行第 8、9、10 件事的时候崩溃的"。

你可以设计一个实验来观察它是在哪件事情上崩溃的吗？当然，你可以用一个调试器或者我们可以在第 8 第 9 件事后面加一些 printlining 的语句（或者你正在使用的任何语言里的等价的事情），当我们重新运行它的时候，我们的谜题会变得更小，比如"它是在做第九件事的时候崩溃的"。我发现，把谜题是怎样的一直清楚地记在心里能让我们保持注意力。当几个人在一个问题的压力下一起工作时，很容易忘记最重要的谜题是什么。

调试技术中分治的关键和算法设计里的分治是一样的。你只要从中间开始划分，就不用划分太多次，并且能快速地调试。但问题的中点在哪里？这就是真正创造力和经验需要参与的地方。

对于一个真正的初学者来说，可能发生错误的地方好像在代码的每一行里都有。一开始，你看不到一些其他的你稍后将会学到的维度，比如执行过的代码段，数据结构，内存管理，与外部代码的交互，一些有风险的代码，一些简单的代码。对于一个有经验的程序员，这些其他的维度为整个可能出错的事情展示了一个不完美但是有用的思维模型。拥有这样的思维模型能让一个人更高效地找到谜题的中点。

一旦你最终划分出了所有可能出错的地方，你必须试着判断错误躲在哪个地方。比如：这样一个谜题，哪一行未知的代码让我的程序崩溃了？你可以这样问自己，出错的代码是在我刚才执行的程序中间的那行代码的前面还是后面？通常你不会那么幸运就能知道错误在哪行代码甚至是哪个代码块。通常谜题更像这个样子的："图中的一个指针指向了错误的结点还是我的算法里变量自增的代码没有生效？"，在这种情况下你需要写一个小程序去确认图中的指针是否都是对的，来决定分治后的哪个部分可以被排除。

如果移除一个错误

我曾有意把检查程序执行和修复错误分割开来，但是当然，调试也意味着移除 bug。理想状况下，当你完美的发现了错误以及它的修复方法时，你会对代码有完美的理解，并且有一种顿悟(啊哈！)的感觉。但由于你的程序会经常使用不具有可视性的、没有一致性注释的系统，所以完美是不可能的。在其他情况下，可能代码是如此的复杂以至于你的理解可能并不完美。

在修复 bug 时，你可能想要做最小的改变来修复它。你可能看到一些其他需要改进的东西，但不会同时去改进他们。试图使用科学的方法去改进一个东西，并且一次只改变一个东西。修复 bug 最好的方式是能够重现 bug，然后把你的修复替换进去，重新运行你的程序，观察 bug 不再出现。当然，有时候不止一行代码需要修改，但你在逻辑上仍然需要使用一个独立原子(译者注：以前人们认为原子不可再分，所以用用原子来代表不可再分的东西)的改变来修复这个 bug。

有时候，可能实际上有几个 bug，但表现出来好像是一个。这取决于你怎么定义 bug，你需要一个一个地修复它们。有时候，程序应该做什么或者原始作者想要做什么是不清晰的。在这种情况下，你必须多加练习，增加经验，评判并为代码赋予你自己的认知。决定它应该做什么,并注释/或用其他方式阐述清楚，然后修改代码以遵循你赋予的含义。这是一个进阶或高级的技能，有时甚至比一开始用原始的方式创建这些代码还难，但真实的世界经常是混乱的。你必须修复一个你不能重写的系统。

如何使用日志调试

Logging（日志）是一种编写系统的方式，可以产生一系列信息记录，被称为 log。*Printlining*只是输出简单的，通常是临时的日志。初学者一定要理解并且使用日志，因为他们对编程的理解是局限的。因为系统的复杂性,系统架构必须理解与使用日志。理想地，程序运行时，日志产生的信息的数量需要是可配置的。通常，日志提供了下面三个基本的优点：

- 日志可以提供一些难以重现的 bug 的有效信息，比如在产品环境中发生的、不能在测试环境重现的 bug。
- 日志可以提供统计和与性能相关的数据，比如语句间流逝过的时间。
- 可配置的情况下，日志允许我们获取普通的信息，使得我们可以在不修改或重新部署代码的情况下调试以处理具体的问题。

需要输出的日志数量总是一个简约与信息的权衡。太多的信息会使得日志变得昂贵，并且造成*滚动目盲*，使得发现你想要的信息变得很困难。但信息太少的话，日志可能不包含你需要的信息。出于这个原因，让日志的输出可配置是非常有用的。通常，日志中的每个记录会标记它在源代码里的位置，执行它的线程（如果可用的话），时间精度，并且，通常有，一些额外的有效信息，比如一些变量的值，剩余内存大小，数据对象的数量，等等。这些日志语句撒遍源码，但只出现在主要的功能点和一些可能出现危机的代码里。每个语句可以被赋予一个等级，并且将会在系统设置输出这个等级时输出这个记录。你应该设计好日志语句来标记你预期的问题。预估测量程序表现的必要性。

如果你有一个永久的日志，printling 现在可以用日志的形式来完成，并且一些调试语句可能会永久地加入日志系统。

如何理解性能问题

学习理解运行的程序的性能问题与学习 debug 是一样不可避免的。即使你完美地理解了你写的代码的代价，你的代码也会调用其他你几乎不能控制的或者几乎不可看透的软件系统。然而，实际上，通常性能问题和调试有点不一样，而且往往要更简单些。

假如你或你的客户认为你的一个系统或子系统运行太慢了。在你把它变快之前，你必须构建一个它为什么慢的思维模型。为了做到这个，你可以使用一个图表工具或者一个好的日志，去发现时间或资源真正被花费在什么地方。有一句很有名的格言：90%的时间会花费在 10%的代码上。在性能这个话题上，我想补充的是输入输出开销的重要性。通常大部分时间是以某种形式花费在 I/O 上。发现昂贵的 I/O 和昂贵的 10%代码是构建思维模型的一个好的开始。

计算机系统的性能有很多个维度，很多资源会被消耗。第一种资源是"挂钟时间"，即执行程序的所有时间。记录"挂钟时间"是一件特别有价值的事情，因为它可以告诉我们一些图表工具表现不了的不可预知的情况。然而，这并不总是描绘了整幅图景。有时候有些东西只是花费了稍微多一点点时间，并且不会引爆什么问题，所以在你真实要处理的计算机环境中，多一些处理器时间可能会是更好的选择。相似的，内存，网络带宽，数据库或其他服务器访问，可能最后都比处理器时间要更加昂贵。

竞争共享的资源被同步使用，可能导致死锁和线程饥饿，如果这是可预见的，最好有一种方式来合适地测量这种竞争。即使竞争不会发生，能够断言这种情况也是非常有帮助的。

如何修复性能问题

大部分软件都可以通过相对小得多的努力，变得比它们刚发布时，在时间上快 10 到 100 倍。在市场发布时间的压力下，选择一个简单快速的解决性能问题的方法而非其他方法是聪明而有效率的。然而，性能是可用性的一部分，而且通常它也需要被更仔细地考虑。

提高一个非常复杂的系统的性能的关键是，充分分析它，以发现"瓶颈"，或者资源耗费的地方。优化一个只占用 1%执行时间的函数是没有多大意义的。一个简要的原则是，你在做任何事情之前必须仔细思考，除非你认为它能够使系统或者它的一个重要部分至少快两倍。通常会有一种方法来达到这个效果。考虑你的修改会带来的测试以及质量保证的工作需要。每个修改带来一个测试负担，所以最好这个修改能带来一点大的优化。

当你在某个方面做了一个两倍提升后，你需要至少重新考虑并且可能重新分析，去发现系统中下一个最昂贵的瓶颈，并且攻破那个瓶颈，得到下一个两倍提升。

通常，性能的瓶颈的一个例子是，数牛的数目：通过数脚的数量然后除以 4，还是数头的数量。举些例子，我曾犯过的一些错误：没能在关系数据库中，为我经常查询的那一列提供适当的索引，这可能会使得它至少慢了 20 倍。其他例子还包括在循环里做不必要的 I/O 操作，留下不再需要的调试语句，不再需要的内存分配，还有，尤其是，不专业地使用库和其他的没有为性能充分编写过的子系统。这种提升有时候被叫做"低垂的水果"，意思是它可以被轻易地获取，然后产生巨大的好处。

你在用完这些"低垂的水果"之后，应该做些什么呢？你可以爬高一点，或者把树锯倒。你可以继续做小的改进或者你可以严肃地重构整个系统或者一个子系统。（不只是在新的设计里，在信任你的 boss 这方面，作为一个好的程序员，这是一个非常好的使用你的技能的机会）然而，在你考虑重构子系统之前，你应该问你自己，你的建议是否会让它好五倍到十倍。

如何优化循环

有时候你会遇到循环，或者递归函数，它们会花费很长的执行时间，可能是你的产品的瓶颈。在你尝试使循环变得快一点之前，花几分钟考虑是否有可能把它整个移除掉，有没有一个不同的算法？你可以在计算时做一些其他的事情吗？如果你不能找到一个方法去绕开它，你可以优化这个循环了。这是很简单的，move stuff out。最后，这不仅需要独创性而且需要理解每一种语句和表达式的开销。这里是一些建议：

- 删除浮点运算操作。
- 非必要时不要分配新的内存。
- 把常量都放在一起声明。
- 把 I/O 放在缓冲里做。
- 尽量不使用除法。
- 尽量不适用昂贵的类型转换。
- 移动指针而非重新计算索引。

这些操作的具体代价取决于你的具体系统。在一些系统中，编译器和硬件会为你做一些事情。但必须清楚，有效的代码比需要在特殊平台下理解的代码要好。

如何处理 I/O 代价

在很多问题上，处理器的速度比硬件交流要快得多。这种代价通常是小的 I/O，可能包括网络消耗，磁盘 I/O，数据库查询，文件 I/O，还有其他与处理器不太接近的硬件使用。所以构建一个快速的系统通常是一个提高 I/O 的问题，而非在紧凑的循环里优化代码或者甚至优化算法。

有两种基本的技术来优化 I/O：缓存和代表（译者注：比如用短的字符代表长的字符）。缓存是通过本地存储数据的副本，再次获取数据时就不需要执行 I/O,以此来避免 I/O（通常避免读取一些抽象的值）。缓存的关键在于让（上层对于）哪些数据是主干的，哪些数据是副本，完全透明。主干的数据只有一份-周期。缓存有这样一种危险：副本有时候不能立刻反映主干的修改。

代表是通过更高效地表示数据来让 I/O 更廉价。这通常会限制其他的要求，比如可读性和可移植性。

代表通常可以用他们第一实现中的两到三个因子来做优化。实现这点的技术包括使用二进制表示而非人类可识别的方式,传递数据的同时也传递一个符号表，这样长的符号就不需要被编码，极端的，可能会像哈弗曼编码。

一个偶尔可行的第三方技术是让计算更接近数据，来优化本地引用。例如，如果你正在从数据库读取一些数据并且在它上面执行一些简单的计算，比如求和，试着让数据库服务器去做这件事，这高度依赖于你正在工作的系统的类型，但这个方面你必须自己探索。

如何管理内存

内存是一种你不可以耗尽的珍贵资源。在一段时期里，你可以无视它，但最终你必须决定如何管理内存。

堆内存是在单一子程序范围外，需要持续（保留）的空间。一大块内存，在没有东西指向它的时候，是无用的，因此被称为*垃圾*。根据你所使用的系统的不同，你可能需要自己显式释放将要变成垃圾的内存。更多时候你可能使用一个有*垃圾回收器*的系统。一个垃圾回收器会自己注意到垃圾的存在并且在不需要程序员做任何事情的情况下释放它的内存空间。垃圾回收器是奇妙的：它减小了错误，然后增加了代码的简洁性。如果可以的话，使用垃圾回收器。 但是即使有了垃圾回收机制，你还是可能把所有的内存填满垃圾。一个典型的错误是把哈希表作为一个缓存，但是忘了删除对哈希表的引用。因为引用仍然存在，被引用者是不可回收但却无用的。这就叫做*内存泄露*。你应该尽早发现并且修复内存泄露。如果你会长时间运行系统，内存可能在测试中不会被耗尽，但可能在用户那里被耗尽。

创建新对象在任何系统里都是有点昂贵的。然而，在子程序里直接为局部变量分配内存通常很便宜，因为释放它的策略很简单。你应该避免不必要的对象创建。

当你可以定义你一次需要的数量的上界的时候，一个重要的情况出现了：如果这些对象都占用相同大小的内存，你可以使用单独的一块内存，或缓存，来持有所有的这些对象。你需要的对象可以在这个缓存里以循环的方式分配和释放，所以它有时候被称为环缓存。这通常比堆内存分配更快。（译者注：这也被称为对象池。）

有时候你需要显式释放已分配的内存，所以它可以被重新分配而非依赖于垃圾回收机制。然后你必须在每块内存上使用谨慎的智慧，并且为它设计一种在合适的时候重新分配的方式。这种销毁的方式可能随着你创建的对象的不同而不同。你必须确定每个内存分配方法的执行与最终都匹配一个内存释放操作。（译者注：在 C 里面，no malloc no free，在 C++ 里面，no new no free）。这通常是很困难的，所以程序员通常会实现一种简单的方式或者垃圾回收机制，比如引用计数，来为它们做这件事情。

如何处理偶现的 Bugs

偶现 bug 是外部不可见的 50 足的蝎子的亲戚。这种噩梦是如此稀少以至于它很难观察，但其出现频率使得它不能被忽视。你不能调试因为你不能找到它。

尽管在 8 个小时后你会开始怀疑，偶现的 bug 必须像其他事情一样遵循相同的逻辑规律。但困难的是它只发生在一些未知的情形。尝试记录这个 bug 会出现的情况，这样你可以猜测真实的影响变量是什么。情况可能跟数据的值相关，比如"这只是在我们把 *Wyoming* 作为一个值输入时发生"，如果这不是变量的根源，下一个怀疑应该是不合适的同步并发。

尝试，尝试，尝试去在一种可控的方式下重现这个 bug。如果你不能重现它，用日志系统给它设置一个圈套，来在你需要的时候，在它真的发生的时候，记录你猜想的，需要的东西。重新设计这个圈套，如果这个 bug 只发生在产品中，且不在你的猜想中的话，这可能是一个长的过程。你从日志中得到的（信息）可能不能提供解决方案，但可能给你足够的信息去优化这个日志。优化后的日志系统可能花很长时间才能被放入产品中使用。然后，你必须等待 bug 重新出现以获得更多的信息。这个循环可能会继续好几次。

我曾创建过的最愚蠢的偶现 bug 出现在，一个函数式编程语言里为类工程做多线程实现。我非常仔细地保证了函数式程序的并发估计，还有好的 CPU 使用（在这个例子里，是 8 个 CPU）。我简单地忘记了同步垃圾回收器。系统可能运行了很长一段时间，经常结束在我开始任何一个任务的时候，在任何能被注意到的事情出错之前。我很遗憾地承认在我理解我的错误之前，我甚至开始怀疑硬件了。

在工作中我们最近有这样一个偶现的 bug 让我们花了几个星期才发现。我们有一个多线程的基于 Apache™ 的 Java™web 服务器,在维护第一个页面跳转的时候，我们在四个独立线程里以 4 个独立的线程而非页面跳转线程为一个小的集合执行所有的 I/O 操作。每一次跳转会产生明显的卡顿然后停止做任何有用的事情，直到几个小时后，我们的日志允许我们去了解发生了什么。因为我们有四个线程，在一个线程内部发生这种情况并不是什么大问题，除非所有的四个线程都阻塞了。然后被这些线程排空的队列会迅速填充所有可用的内存，然后导致我们的服务器崩溃。这个 bug 花了我们一个星期去揪住这个问题，但我们仍然不知道什么导致了这个现象，不知道它什么时候会发生，甚至不知道它们阻塞的时候，线程们在干什么。

这表明了有关使用第三方软件的一些风险。我们在使用一段授权的代码，从文本中移除 HTML 标签。受它的起源的影响，我们把它叫做法国脱衣舞者。尽管我们有源代码（由衷感谢！），我们没有仔细研究它，直到查看我们服务器的日志的时候，我们最终意识到"法国脱衣舞者"中，通信线程阻塞了。

这个工具在大多数时候工作得很好，除了处理一些长而不常见的文本时。在那些文本里，代码复杂度是 N 平方或者更糟。这意味着处理时间与文本的长度的平方成正比。由于这些文本通常都会出现，我们可以马上发现这个 bug。如果他们从来都不会出现，我们永远都不会发现这个问题。当它发生时，我们花了几个星期去最终理解并且解决了这个问题。

如何学习设计技能

为了学习如何设计软件，你可以在导师做设计的时候，出现在他身边，学习他的行为。然后学习精心编写过的软件片段（译者注：比如 android 系统中的谷歌官方应用）。在这之后，你可以读一些关于最新设计技术的书。

然后你必须自己动手了。从一个小的工程开始，当你最后完成时，考虑为什么这个设计失败了或成功了，你是怎样偏离你最初的设想的。然后继续去着手大一点的工程，在与其他人结合时会更有希望。设计是一种需要花很多年去学习的关于评判的事情。一个聪明的程序员可以学习在两个月内充分学好这种基础然后从这里开始进步。

发展出你自己的风格是自然而有用的，但记住，设计是一种艺术，而不是一种科学。人们写的关于这个主题的书都有一种使得它好像是科学的既定的兴趣。不要武断对待特定的设计风格。

如何进行实验

已故的伟大的 Edsger Dijkstra 曾经充分解释过：计算机科学不是一门实验科学[ExpCS],并且不依赖于电子计算机。当他提出这个观点时，他指的是 19 世纪 60 年代。[Knife]

> ...危害已经出现：主题现在已经变成了“计算机科学” - 这实际上，像是把外科手术引用为“手术刀科学” - 这在人们心中深深植入了这样一个概念：计算机科学是关于机器和它们的外围设备的。

编程不应该是一门实验科学，但大多数职业程序员并没有保卫 Dijkstra 对于计算机科学的解释的荣耀。我们必须在实验的领域里工作，正如一部分，但非所有的，物理学家做的。如果三十年后，编程可以在不进行任何实验的前提下进行，这将是计算机科学的一个巨大成就。

你需要进行的实验包括：

- 用小的例子测试系统以验证它们遵循文档，或者在没有文档时，理解它们的反应；
- 测试一些小的代码修改去验证它们是否确实修复了一个 bug；
- 由于对一个系统不完全的理解，需要在两种不同情况下测量它们的性能表现；
- 检查数据的完整性，和
- 对困难的或者难以重现的 bug，收集解决方案中可能提示的统计数据。

我不认为在这篇文章里我可以讲述实验的设计，你会在实践中学习到这方面的知识。然而，我可以提供两点建议：

第一，对你的假设或者你要测试的断言要非常清楚。把假设写下来也是很有用的，尤其是如果你有点迷惑或者与其他人合作时。

你会经常发现你必须设计一系列的实验，它们中的每个都基于对最后一个实验的理解。所以，你应该设计你的实验去提供大部分可能的信息。不幸的是，这会影响保持实验简单的目的 - 你必须通过经验来发展这种评判的能力。

上一篇: 关于 (/project/how-to-be-a-programmer/)
下一篇: 团队技能 (/project/how-to-be-a-programmer/team-skills.html)

(http://my.jikexueyuan.com/0mgVaVPXW/)
jike_4739662 (http://my.jikexueyuan.com/0mgVaVPXW/) #1

由于时间关系，匆匆读完，感谢分享，和好，有时间，再仔细看一遍

2016年6月23日 0 回复