**白板编程浅谈——Why, What, How**

2015年 5月31日 | [评论](http://zh.lucida.me/blog/whiteboard-coding-demystified/" \l "disqus_thread)

作者：[Lucida](http://lucida.me/about)

* 微博：[@peng\_gong](http://www.weibo.com/pegong/)
* 豆瓣：[@figure9](http://www.douban.com/people/figure9/)

原文链接：<http://lucida.me/blog/whiteboard-coding-demystified/>

*这篇文章节选自我正在撰写的一本关于应届生面试求职的书籍，欢迎在评论或微博（[@peng\_gong](http://www.weibo.com/pegong)）上留言反馈。*

面试很困难，技术面试更加困难——只用 45 ~ 60 分钟是很难考察出面试者的水平的。所以 [刘未鹏](http://mindhacks.cn/)在他的 [怎样花两年时间去面试一个人](http://mindhacks.cn/2011/11/04/how-to-interview-a-person-for-two-years/) 一文中鼓励面试者创建 GitHub 账号，阅读技术书籍，建立技术影响力，从而提供给面试官真实，明确，可度量的经历。

这种方法对面试者效果很好，但对面试官效果就很一般——面试官要面对大量的面试者，这些面试者之中可能只有很少人拥有技术博客，但这并不代表他们的技术能力不够强（也许他们对写作不感兴趣）；另一方面，一些人拥有技术博客，但这也不能说明他们的水平就一定会很牛（也许他们在嘴遁呢）。

总之，技术博客和 GitHub 账号是加分项，但技术面试仍然必不可少。所以，问题又回来了，如何进行高效的技术面试？或者说，如何在 45 ~ 60 分钟内尽可能准确的考察出面试者的技术水平？

回答这个问题之前，让我们先看下技术面试中的常见问题都有什么：

**技术面试中的常见问题**

技术面试中的问题大致可以分为 5 类：

1. 编码：考察面试者的编码能力，一般要求面试者在 20 ~ 30 分钟之内编写一段需求明确的小程序（例：编写一个函数划分一个整形数组，把负数放在左边，零放在中间，正数放在右边）；
2. 设计：考察面试者的设计/表达能力，一般要求面试者在 30 分钟左右内给出一个系统的大致设计（例：设计一个类似微博的系统）
3. 项目：考察面试者的设计/表达能力以及其简历的真实度（例：描述你做过的 xxx 系统中的难点，以及你是如何克服这些难点）
4. 脑筋急转弯：考察面试者的『反应/智力』（例：如果你变成蚂蚁大小然后被扔进一个搅拌机里，你将如何脱身？）
5. 查漏：考察面试者对某种技术的熟练度（例：Java 的基本类型有几种？）

这 5 类问题中，脑筋急转弯在外企中早已绝迹（因为它无法判定面试者的真实能力），查漏类问题因为实际价值不大（毕竟我们可以用 Google）在外企中出现率也越来越低，剩下的 3 类问题里，项目类和设计类问题要求面试官拥有同类项目经验，只有编码类问题不需要任何前提，所以，几乎所有的技术面试中都包含编码类问题。

然而，最令面试者头痛的也是这些编码类问题——因为几乎所有的当面（On-site）技术面试均要求面试者在白板上写出代码，而不是在面试者熟悉的 IDE 或是编辑器中写出。在我的面试经历里，不止一个被面试者向我抱怨：『如果能在计算机上编程，我早就把它搞定了！』就连我自己在面试初期也曾怀疑白板代码的有效性：『为什么不让面试者在计算机上写代码呢？』

然而在经历了若干轮被面试与面试之后，我惊奇的发现白板编程竟然是一种相当有效的技术考察方式。这也是我写这篇文章的原因——我希望通过这篇文章来阐述为什么要进行白板编程（WHY），什么是合适的白板编程题目（WHAT），以及如何进行白板编程（HOW），从而既帮助面试者更好的准备面试，也帮助面试官更好的进行面试。

**为什么要进行白板编程**

很多面试者希望能够在 IDE 中（而不是白板上）编写代码，因为：

1. 主流 IDE 均带有智能提示，从而大大提升了编码速度
2. IDE 可以保证程序能够编译通过
3. 可以通过 IDE 运行/调试代码，找到程序的 Bug

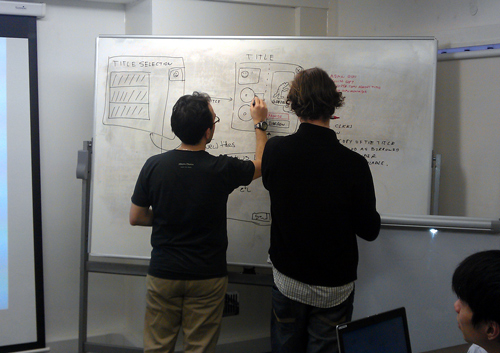
我承认第 1 点，白板编程要比 IDE 编程慢很多，但这并不能做为否认白板编程的理由——因为白板编程往往是 API 无关（因此并不需要你去背诵 API）的一小段（一般不超过 30 行）代码，而且面试官也会允许面试者进行适当的缩写（比如把Iterable类型缩写为Iter），因此它并不能成为否认白板编程的理由。

至于第 2 点和第 3 点，它们更不能成为否认白板编程的借口——如果你使用 IDE 只是为了在其帮助下写出能过编译的代码，或是为了调试改 Bug，那么我不认为你是一名合格的程序员——我认为程序员可以被分为两种：

1. 先确认前条件/不变式/终止条件/边界条件，然后写出正确的代码
2. 先编写代码，然后通过各种用例/测试/调试对程序进行调整，最后得到似乎正确的代码

我个人保守估计前者开发效率至少是后者的**10 倍**，因为前者不需要浪费大量时间在 **编码-调试-编码** 这个极其耗时的循环上。通过白板编程，面试官可以有效的判定出面试者属于前者还是后者，从而招进合适的人才，并把老油条或是嘴遁者排除在外。

除了判定面试者的开发效率，白板编程还有助于展示面试者的编程思路，并便于面试者和面试官进行交流：



白板编程的目标并不是要求面试者一下子写出完美无缺的代码，而是：

* 让面试者在解题的过程中将他/他的思维过程和编码习惯展现在面试官面前，以便面试官判定面试者是否具备清晰的逻辑思维和良好的编程素养
* 如果面试者陷入困境或是陷阱，面试官也可以为其提供适当的辅助，以免面试陷入无人发言的尴尬境地

**什么是合适的白板编程题目**

正如前文所述，白板编程是一种很有效的技术面试方式，但这是建立在**有效的编程题目**的基础之上：如果编程题目过难，那么面试很可能会陷入『大眼瞪小眼』的境地；如果编程题目过于简单（或者面试者背过题目），那么面试者无需思考就可以给出正确答案。这两种情况都无法达到考察面试者思维过程的目的，从而使得面试官无法正确评估面试者的能力。

既然编程题目很重要，那么问题来了，什么才是合适（合理）的编程题目呢？

在回答这个问题之前，让我们先看看什么编程题目不合适：

**什么不该问**

**被问滥的编程问题**

我在求职时发现，技术面试的编程题目往往千篇一律——拿我自己来说，反转单链表被问了 5 次，数字转字符串被问了 4 次，随机化数组被问了 3 次，最可笑的是在面试某外企时三个面试官都问我如何反转单链表，以至于我得主动要求更换题目以免误会。

无独有偶，我在求职时同时发现很多面试者都随身带一个本子或是打印好的材料，上面写满了常见的面试题目，一些面试者甚至会祈祷能够被问到上面的题目。

就这个问题，我和我的同学以及后来的同事讨论过，答案是很多面试官在面试前并不会提前准备面试题，而是从网络上（例如 [July 的算法博客](http://blog.csdn.net/v_JULY_v)）或 [编程之美](http://www.amazon.cn/gp/product/B0016K8XGQ/ref=as_li_ss_tl?ie=UTF8&camp=536&creative=3132&creativeASIN=B0016K8XGQ&linkCode=as2&tag=lucida-23) 之类的面试题集上随机挑一道题目询问。如果面试者做出来（或背出来）题目那么通过，如果面试者做不出来就挂掉。

这种面试方式的问题非常明显：如果面试者准备充分，那么这些题目根本没有区分度——面试者很可能会把答案直接背下来；如果面试者未做准备，他/她很可能被一些需要 [aha! moment](http://en.wikipedia.org/wiki/Eureka_effect) 的题目困住。总之，如果面试题不能评估面试者水平，那么问它还有什么意义呢？

下面是一些**问滥的编程问题**：

* [编程之美](http://www.amazon.cn/gp/product/B0016K8XGQ/ref=as_li_ss_tl?ie=UTF8&camp=536&creative=3132&creativeASIN=B0016K8XGQ&linkCode=as2&tag=lucida-23) 书里的所有题目；
* [July 的算法博客](http://blog.csdn.net/v_JULY_v) 中的绝大多数题目（包括 [面试 100 题](http://blog.csdn.net/v_JULY_v/archive/2010/11/17/6015165.aspx) 中的所有题目）；
* [leecode](https://leetcode.com/) 里的大部分题目；

**涉及到库函数或 API 调用**

白板编程的目标在于考察面试者的编程基本功，而不是考察面试者使用某种语言/类库的熟练度。所以白板编程题目应尽可能库函数无关——例如：编写一个 XML 读取程序就是不合格的题目，因为面试者没有必要把 XML 库中的函数名背下来（不然要 Intellisense 干甚）；而原地消除字符串的重复空白（例："ab c d e" => "ab c d e"）则是一道合格的题目，因为即便不使用库函数，合格的面试者也能够在 20 分钟内完成这道题目。

**过于直接（或简单）的算法问题**

这类问题类似 [被问滥的编程问题](http://zh.lucida.me/blog/whiteboard-coding-demystified/" \l "abused_problems)，它们的特点在于过于直接，以至于面试者不需要思考就可以给出答案，从而使得面试官无法考察面试者的思维过程。快速排序，深度优先搜索，以及二分搜索都属于这类题目。

需要注意的是，尽管过于直接的算法题目不适合面试，但是我们可以将其进行一点改动，从而使其变成合理的题目，例如稳定划分和二分搜索计数（给出有序数组中某个元素出现的次数）就不错，尽管它们实际是快速排序和二分搜索的变种。

**过于复杂的题目**

同 [过于直接的算法问题<](http://zh.lucida.me/blog/whiteboard-coding-demystified/" \l "too_straightforward_problems) 相反，[过于复杂的题目](http://zh.lucida.me/blog/whiteboard-coding-demystified/" \l "too_complicated_problems) 属于另一个极端：这些题目往往要求面试者拥有极强的算法背景，尽管算法问题是否过于复杂因人而异（在一些 ACM 编程竞赛选手的眼里可能就没有复杂的题目 –\_-），但我个人认为如果一道题满足了下面任何一点，那么它就太复杂，不适合面试（不过如果面试者是 ACM 编程竞赛选手，那么可以无视此规则）：

* 需要 [aha! moment](http://en.wikipedia.org/wiki/Eureka_effect)（参考 [脑筋急转弯](http://zh.lucida.me/blog/whiteboard-coding-demystified/" \l "brain_teaser)）
* 需要使用某些『非主流』数据结构/算法才能求解
* 耗时过长（例如实现红黑树的插入/删除）

**脑筋急转弯**

什么是脑筋急转弯？

* 不考察编程能力
* 依赖于 [aha! moment](http://en.wikipedia.org/wiki/Eureka_effect)
* All or nothin：或者做不出来，或者是最终答案

在一些书（例如 [谁是谷歌想要的人才？：破解世界最顶尖公司的面试密码](http://www.amazon.cn/gp/product/B00BB81BB2/ref=as_li_ss_tl?ie=UTF8&camp=536&creative=3132&creativeASIN=B00BB81BB2&linkCode=as2&tag=lucida-23)）和电影的渲染下，Google 和微软这些外企的面试被搞的无比神秘，以至于很多人以为外企真的会问诸如『井盖为什么是圆的』或是『货车能装多少高尔夫球』这样的奇诡问题。而实际上，这些题目由于无法考察面试者的技术能力而早已在外企中绝迹。反倒是一些国内公司开始使用[脑筋急转弯](http://zh.lucida.me/blog/whiteboard-coding-demystified/" \l "brain_teaser) 作为面试题目 –\_–#

**应该问什么问题**

所以，技术面试题目不应该太难，也不应太简单，不能是脑筋急转弯，也不能直接来自网络。

前三点并不难满足：我们可以去 [算法导论](http://www.amazon.cn/gp/product/B00AK7BYJY/ref=as_li_ss_tl?ie=UTF8&camp=536&creative=3132&creativeASIN=B00AK7BYJY&linkCode=as2&tag=lucida-23)，[编程珠玑](http://www.amazon.cn/gp/product/B00SFZH0DC/ref=as_li_ss_tl?ie=UTF8&camp=536&creative=3132&creativeASIN=B00SFZH0DC&linkCode=as2&tag=lucida-23)，以及 [计算机程序设计艺术](http://www.amazon.cn/gp/product/B00478TO44/ref=as_li_ss_tl?ie=UTF8&camp=536&creative=3132&creativeASIN=B00478TO44&linkCode=as2&tag=lucida-23) 这些经典算法书籍中的课后题/练习题挑选合适的题目，也可以自己创造题目。然而，由于 [careercup](http://www.careercup.com/page) 这类网站的存在，没有什么题目可以做到绝对原创——毕竟没有人能阻止面试者把题目发到网上，所以任何编程题目都逃脱不了被公开的命运。

不过，尽管面试者会把编程题目发到网上，甚至会有一些『好心人』给出答案，但这并不代表面试官不能继续使用这道题：因为尽管题目被公开，但题目的考察点和延伸问题依然只有面试官才知道。这有点像 [公钥加密](http://zh.wikipedia.org/wiki/%E5%85%AC%E5%BC%80%E5%AF%86%E9%92%A5%E5%8A%A0%E5%AF%86)，公钥（面试题）是公开的，但私钥（解法，考察点，以及延伸问题）只有面试官才知道。这样即便面试者知道面试题，也不会妨碍面试官考察面试者的技术能力。

接下来，让我们看看什么问题适合白板编程。

**不止一种解法**

良好的编程问题都会有不止一种解法。这样面试者可以在短时间内给出一个不那么聪明但可实现的『粗糙』算法，然后通过思考（或面试官提示）逐步得到更加优化的解法，面试官可以通过这个过程观察到面试者的思维方式，从而对面试者进行更客观的评估。

以 [数组最大子序列和](http://en.wikipedia.org/wiki/Maximum_subarray_problem) 为例，它有一个很显然的 O(n3) 解法，将 O(n3) 解法稍加改动可以得到 O(n2) 解法，利用分治思想，可以得到 O(n\*logn) 解法，除此之外它还有一个 o(n) 解法。（[编程珠玑](http://www.amazon.cn/gp/product/B00SFZH0DC/ref=as_li_ss_tl?ie=UTF8&camp=536&tag=lucida-23&creativeASIN=B00SFZH0DC&linkCode=as2&creative=3132) 和 [数据结构与算法分析 C语言描述](http://www.amazon.cn/gp/product/B002WC7NGS/ref=as_li_ss_tl?ie=UTF8&camp=536&creative=3132&creativeASIN=B002WC7NGS&linkCode=as2&tag=lucida-23) 对这道题均有非常精彩的描述，有兴趣的朋友可以自行阅读）

**考察点明确**

良好的编程问题应拥有大量考察点，面试官应对这些考察点烂熟于心，从而给出更加客观量化的面试结果。这里可以参考我之前在 [从武侠小说到程序员面试](http://lucida.me/blog/from-wuxia-to-programmer-interview/) 提到的 to\_upper。

**延伸问题**

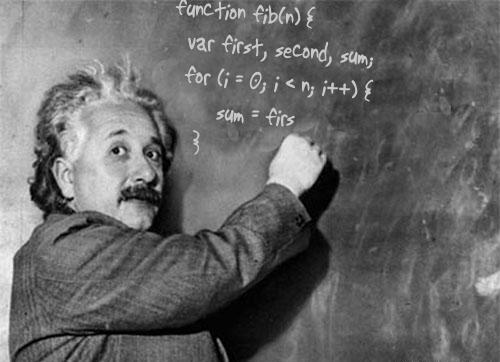
良好的编程问题应拥有延伸问题。延伸问题既可以应对面试者背题的情况，也可以渐进的（Incremental）考察面试者的编程能力，同时还保证了面试的延续性（Continuity）。

以 [遍历二叉树](http://zh.wikipedia.org/wiki/%E6%A0%91%E7%9A%84%E9%81%8D%E5%8E%86) 为例：面试官可以从非递归中序遍历二叉树开始提问，面试者有可能会很快的写（或是背）出一个使用栈的解法。这时面试官可以通过延伸问题来判别面试者是否在背题：使用常量空间中序遍历**带有父节点指针**的二叉树，或是找到二叉搜索树中第 n 小的元素。下面是中序遍历二叉树的一些延伸问题：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | |--中序遍历二叉树  |  |--非递归中序遍历二叉树  |  |--常量空间，非递归遍历带父节点的二叉树  | |  | |--在带父节点的二叉搜索树寻找第 N 小的元素  | |  | |--可否进一步优化时间复杂度？  |  |--常量空间，非递归遍历不带父节点的二叉树 |

上面的问题不但可以被正向使用（逐步加强难度），也可以被逆向使用（逐步降低难度）：同样从非递归中序二叉树遍历开始提问，如果面试者无法完成这个问题，那么面试官可以降低难度，要求面试者编写一个递归版本的中序遍历二叉树。

**如何进行白板编程**



**面试官应该做什么**

**面试前**

面试之前，面试官应至少得到以下信息：

1. 面试者的简历
2. 面试者的应聘职位
3. 面试者之前被问过哪些面试题

接下来，面试官应根据面试者的简历/职位确认对面试者的期望值，然后准备好编程题目（而不是面试时即兴选择题目）。面试官应至少准备 4 道题目（2 道简单题，2 道难题），以应对各种情况。

**面试中**

面试时，面试官应清楚的陈述题目，并通过若干组用例数据确认面试者真正的理解题目（以免面试者花很长时间去做不相关的题目，我在之前的面试就办过这种挫事 –\_–#）

在面试者解题时，面试官应全程保持安静（或倾听的状态），如果面试者犯下特别严重的错误或是陷入苦思冥想，面试官应给出适当的提示，以帮助面试者走出困境完成题目，如果面试者还是不能完成题目，那么面试官应换一道略简单的题目，要知道面试的目的是**发现面试者的长处，而非为难面试者。**（一些国内企业似乎正好相反）

**面试后**

面试之后，面试官应拍照（或誊写）面试者写下的代码，然后把提问的问题发给 HR 和接下来的面试者（以确保问题不会重复）。接下来，面试官应根据面试者的代码以及其面试表现，尽快写出面试反馈（Interview Feedback）发给 HR，以便接下来的招聘流程。

**面试者应该做什么**

**面试前**

面试之前，面试者应至少做过以下准备：

1. 拥有扎实的数据结构/算法基础
2. 知道如何利用 前条件/不变式/后条件 这些工具编写正确的程序
3. 能够在白板（或纸上）实现基本的数据结构和算法（如果 1 和 2 做到这一步是水到渠成）
4. 在 [leetcode](https://leetcode.com/) 或 [careercup](http://www.careercup.com/page) 上面进行过练习，了解常见的技术面试题目（我个人不鼓励刷题，但在面试前建立起对面试题的**『感觉』**非常重要）

**面试中**

**确定需求**

面试者在白板编程时最重要的任务是**理解题目，确认需求**——确定输入/输出，确定数据范围，确定时间/空间要求，确定其它限制。以最常见的**排序**为例：

* 输入：来自数组？链表？或是不同的机器？
* 输出：是否有重复？是否要求稳定？
* 数据范围：排序多少个元素？100 个？ 100 万个？ 1 亿个？这些元素是否在某个范围内？
* 时间要求：1 分钟？1 刻钟？一小时？
* 空间要求：是否常量空间？是否可以分配新的空间？如果可以，能分配多少空间？是否在内存中排序？
* 其它限制：是否需要尽可能少的赋值？是否需要尽可能少的比较？

有时面试官不会把题目说的特别清楚，这时就需要面试者自己去确认这些需求，不要认为这是在浪费时间，不同的需求会导致截然不同的解法，此外确认需求会留给面试官良好的印象。

**白板编程**

理解题目确认需求之后，面试者就可以开始在白板上编写代码，下面是一些我自己的白板编程经验：

* 先写出轮廓（大纲）

白板编程没法复制粘贴，所以后期调整代码结构非常困难。因此我们最好在开头写出程序的大致结构，从而保证之后不会有大改；

* 确定前条件/不变式/后条件

我们可以通过注释的形式给出代码的前条件/不变式/后条件，以划分为例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int\* partition(int \*begin, int \*end, int pivot) {  int \*par = begin;  for ( ; begin < end; begin++) {  if (\*begin < pivot) {  swap(begin, par++)  }  }  return par;  } |

就不如

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | int\* partition(int \*begin, int \*end, int pivot) {  // [begin, end) should be a valid range  int \*par = begin;  // Invariant: All [0, par) < pivot && All [par, begin) >= pivot  for ( ; begin < end; begin++) {  if (\*begin < pivot) {  swap(begin, par++)  }  }  // Now All [0, par) < pivot && All [par, end) >= pivot  return par;  } |

* 使用实例数据验证自己的程序

尽管不变式足以验证程序的正确性，但适当的使用实例数据会大大增强代码的可信性，以上面的划分程序为例：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | Given range [2, 3, 4, 5, 1] and pivot 3  [ 2, 3, 4, 5, 1 ]  ^ ^  p,b e  [ 2, 3, 4, 5, 1 ]  ^ ^  p,b e  [ 2, 3, 4, 5, 1 ]  ^ ^ ^  p b e  [ 2, 3, 4, 5, 1 ]  ^ ^ ^  p b e  [ 2, 1, 4, 5, 3 ]  ^ ^ ^  p b e  [ 2, 1, 4, 5, 3 ]  ^ ^  p b,e  Now we have all [0, p) < 3 and all [p, e) >= 3 |

* 使用缩写

白板编程并不需要面试者在白板上写出能够一次通过编译的代码。为了节省时间，面试者可以在和面试官沟通的基础上使用缩写。例如使用 Iter 替代 Iterable，使用 BQ 替代 BlockingQueue。（此法尤其适合于 Java –\_–#）

* 至少留一行半行宽

出于紧张或疏忽，一般面试者在白板编程时会犯下各种小错误，例如忘了某个判断条件或是漏了某条语句，空余的行宽可以帮助面试者快速修改代码，使得白板上的代码不至于一团糟。

这就延伸出了另一个问题，如果使用大行宽，那么白板写不下怎么办？一些面试者聪明的解决了这个问题：**他们在面试时会自带一根细笔迹的水笔，专门用于白板编程。**

**不会做怎么办**

相信大多数面试者都碰到过面试题不会做的情况，这里说说我自己的对策：

1. 至少先给出一个暴力（Brute force）解法
2. 寻找合适的数据结构（例如栈/队列/树/堆/图）和算法（例如分治/回溯/动态规划/贪婪）
3. 从小数据集开始尝试
4. 如果还是没有头绪，重新考虑题目的前条件，思考是否漏掉了条件（或是隐含的条件）
5. 如果 3 分钟过后还是没有任何思路，请求面试官提示，不要觉得不好意思——经过提示给出答案远强于没有答案

**面试后**

个人不建议面试者在面试之后把题目发到网上，很多公司在面试前都会和面试者打招呼，有的会签订 NDA（Non Disclosure Agreement）条款以确保面试者不会泄露面试题目。尽管他们很少真的去查，但如果被查到那绝对是得不偿失。

我自己在面试之后会把面试中的编程题目动手写一遍（除非题目过于简单不值得），这样既能够验证自己写的代码，也可以保证自己不会在同一个地方摔倒两次。

**参考**

**书籍**

1. [Elements of Programming Interviews: The Insiders’ Guide](http://www.amazon.com/Elements-Programming-Interviews-Insiders-Guide/dp/1479274836)
2. [编程原本](http://www.amazon.cn/gp/product/B006P7V73G/ref=as_li_ss_tl?ie=UTF8&camp=536&tag=lucida-23&creativeASIN=B006P7V73G&linkCode=as2&creative=3132)
3. [程序员面试金典（第5版）](http://www.amazon.cn/gp/product/B00G8VOQOG/ref=as_li_ss_tl?ie=UTF8&camp=536&tag=lucida-23&creativeASIN=B00G8VOQOG&linkCode=as2&creative=3132)

**文章**

1. [怎样花两年时间去面试一个人](http://mindhacks.cn/2011/11/04/how-to-interview-a-person-for-two-years/)
2. [5 Whiteboard Coding Tips for Interviews](http://product.hubspot.com/blog/5-whiteboard-coding-tips-for-interviews)
3. [Is “White-Board-Coding” inappropriate during interviews?](http://programmers.stackexchange.com/questions/119168/is-white-board-coding-inappropriate-during-interviews)

以上。