老曹眼中的研发估点那些事儿

老板常问：“产品什么时候可以上线呢？”

产品经理常问：“完成这些功能需要多长时间呢？”

技术经理常问：”这个模块要开发多久呢？“

自己常问：“为啥又要delay呢？”

……

所有这些问题，都会指向一件事————研发中的估点。估点是计划的基础，不论你关注还是不关注它，它都在那里。估点不是拍脑袋，是一种对事件的客观描述方式。通过统计学可以让我们知道，用两个数字就能够描述世界——期望和方差。然而，如果没有历史数据的话，统计学的技术方法就无法应用。因此，估点既是获取研发中经验数据的开始，也贯穿于研发过程的始终。

从零开始——无历史参考的初始估点

对产品开发时间的估算有多重方式，其中目标分解后对每个子任务的时间估算一般被认为是估点，产品开发时间的估算是由估点后形成的关键路径决定的。对于估算本身而言，如果是基于一种次序性的尺度，而且有把握对***等距尺度***作出解释，那么就可以在这种类型的数据上安全地执行推断性统计分析，从而得到预测的结果。

如果使我们的研发时间估算相对准确，那么估点中的等距尺度是什么呢？

估点的单位

不论是TRIZ还是一般的架构思想，都会考虑***以终为始***。 对于估点中的等距尺度即估点的时间单位而言，也是如此。

既然要得到一个时间的数值，进一步提高准确度的话，还需要一个置信区间，所以估点应该依据一个相等的时长。就像我们在物理课上做测量那样，需要一个测量单位。如果单位是米，误差就可能是米或者更大，如果是厘米，那么误差就可能是厘米，以此类推。同理，如果估点的单位时长较大，那么整个估算的误差也会较大，如果估点的单位时长过小，那么操作起来就会比较复杂，就像我们学生时代使用游标卡尺去测量长度那样。

那么多大的时长是相对合适的呢？

互联网上有一种说法，“三个月就是一年”，不仅是形容了互联网的发展速度，而且是符合敏捷开发的思想和实践的——快速迭代。三个月就是一年，这是一个1:4的关系，一周顶四周用，一天相当于四天，那么两个小时（Double Hours，DHR）就相当于一天了。因此，对人 / 天的任务估算可以转化为对人 / DHR的估算，也就是说，估点的单位时长为两个小时（DHR）是相对合理，而且是可以接受的。

初始估点的方法

作为一个新组建的团队，如果没有可测量的历史数据作为支撑，那么初始估点的方式一般是：**将产品的目标转化为多个以两小时为单位的可开发实现的任务。**

将产品的目标转化为以DHR为单位的小任务是一个设计、建模和架构的过程，同样可以通过敏捷开发的方法来实现。具体地，就是明确我们的Sprint周期，根据需求来定义用户故事，将用户故事拆分为一个个以每两小时为单位的backlog。

估点中两个主要的难点是：需求的不确定性 和 思维的系统性。

需求的不断变化是导致估点无效的主要因素，这就要求对需求的边界有相对明确的定义和细化。软件估算的准确度取决于对软件定义的细化程度，必须通过排除可变性来源的方法来实现对需求边界的界定。 同时，由一个人来估算“有多少”，由另一个人来估算“有多不确定”，这是考虑不确定性影响的一种不错的方式。

对于思维的系统性是指我们思考问题过程中的盲点，也就是说，有一些我们可能遗漏的东西，可以通过建立一个检查列表的方式实现，这一列表可以根据自己的团队来补充完善。笔者曾经遇到过的功能性遗漏包括：

- 安装程序和构建环境。
- 数据转换和数据迁移的相关工具。
- 使用第三方API或者开源软件所需的集成代码和选型评估。
- 帮助和引导系统。
- 部署方式和监控管理。
- 与外部系统接口的集成、测试及评估。

非功能性需求往往是隐式的，但对于架构而言是必须关注的，笔者曾经遇到遗漏过的非功能性需求约束包括：

- 互操作性，产品所运行的环境与产品之间的相互影响。
- 可修改性，这是一个参数化的过程，要求对内容或展现形式的动态修改。
- 性能，具体的性能指标是否实现。
- 可靠性，结果是否确定，异常是否处理全面等。

- 可复用性，这一功能是否可以复用，粒度如何（函数，模块乃至服务的复用性）等。
- 可伸缩性，随着数据规模或者时间的变化是否可以实现弹性。
- 安全性，涉及安全的林林总总，例如SQL注入，跨域攻击等等。
- 抗毁性，是高可用性的一个分支，主要考虑服务可恢复的场景。
- 易用性，使用是否容易，不论是涉及用户交互，还是进程间或进程内的相互调用。

其中性能在估点时尤其是项目的初期是一个非常有争议的话题，那句“过早优化是万恶之源”实际上是我们对高德纳先生的断章取义，原文大概是这样的：

> 我们应该在例如97%的时间里，忘掉小处的效率；
>
> 过早优化是万恶之源。
>
> 但我们不应该错过关键的3%中的机会。

实际上，非关键路径上的优化是万恶之源，问题的核心所在————如何确定我们的代码是否在关键路径上。不论节省的时间是多少，花费在关键路径上的性能优化都是值得的，也是我们必须要重视的。

估点的简单示例

举一个最常见的例子——用户登陆，如何进行估点呢？

首先，确定这一功能的边界。这里不用赘述领域驱动开发或者5W1H等其他的设计方法，一个简单的用户故事描述可能是这样的：

> 作为一个XXX系统的用户，可以通过在客户端输入帐户信息登陆到XXX系统，看到XXX系统的主页面。

接着，把这一用户故事转换成可以实现的backlog。采用面向接口的方式，把它分割成前后端的设计，那么接口协议的设计可以作为一个backlog。对用户故事中的对象实体进行分析同样是一个backlog，用户是否分类？用户是否存在不同的类型，这涉及到后台的数据表设计。客户端有哪些类型，Android，iOS，还是网页？不同的客户端是否具有相同的呈现形式，还是有各自的特点？ 帐户信息指的的是什么？用户名 / 密码？ 用户名是否有规则呢？密码是否密文传输？……

简化起见，对各种客户端的登陆实现分别作为一个backlog，后台的登陆接口实现以及数据表设计作为一个backlog。得到的估点结果是，6个人 / DHR。

这就足够了吗？

对于功能性需求而言，如果前置条件不足，就需要考虑注册与登录的一致性，登录失败等异常处理和引导有可能又是一个backlog。如果允许使用第三方帐户登录，那又是至少一个backlog。登录页面的引导和帮助，又是一个backlog ……

对于非功能性而言，如果开发者使用的是新的电脑？那么环境的搭建也将是一个backlog。如果需要持续集成，那么Jinkens的搭建及各端构建脚本的编写同样至少是一个backlog。考虑性能因素，引入缓存不会少于两个backlog。对于安全性问题，客户端在输入的时候需要规则检验，同时要做简单的防SQL注入，至少是一个backlog。 至于易用性，是否要在客户端记住用户名 / 密码，在跟换用户登录时，如何处理本地的存储，往往涉及多用户使用同一终端登录的问题，至少又一个backlog。 如果一个用户在多个终端同时登录，会是一种怎样的表现呢？这往往用一个新的用户故事来描述更好。当用户总量和并发发生变化的时候，在一个怎样的范围内，应用的后台可以足够适应……

具体的情况还有很多，就一个登录的功能模块而言，backlog可以从6个到20多个不等，当产品的定义不能覆盖我们在技术上的定义要求的时候，我们有责任和义务就估点提出建议和解决方案。

在我们把目标分解为backlog 之后，具体的就是在两个小时内完成交付了。同样采用一比四的方式，两个小时被分为四段————半小时设计，半小时测试代码，半小时编码实现，最后半小时是测试和文档输出，这不是绝对的，可以交叉，但最好是相对清晰。幸运的是，半小时刚好满足***番茄工作法***对时间的要求。

在确定了估点之后，思考不确定性是必要的。例如对这一登录的示例而言，如果6个DHR是一个最大可能的时间，最乐观的估计可能是2个DHR，最悲观的估计是8个DHR，可以通过简单的经验公式得到一个估算值：

> 估算时间=（ 乐观估计 + 可能时间*4 + 悲观估计 ） / 6
>
> 或
>
> 估算时间=（ 乐观估计 + 可能时间*3 + 悲观估计*2 ） / 6

即 （ 2+6*4 +8 ） / 6= 5.7 DHR，这个数是可以作为一个期望值的。

尤其需要注意的是，对系统架构而言，往往要复杂的多，但是思路和方法是一致的。对整个产品而言，资源的约束和关键路径的组成，对产品开发周期的整体估算是至关重要的。

一般地，我们需要使用协同工具来关注资源的约束和关键路径。在自己曾经使用过的协同工具中，笔者认为trello 是非常出色的服务之一，可以对估点进行详细的记录和追踪，同时通过对支持trello 各种插件的使用，可以生成燃尽图等多种数据图表，从而能够更有效地了解产品开发过程的真实进度。

多元估点——数据方法的佐证

当进行了三个以上的sprint之后，相等于初步完成了对研发过程中相关数据的采集。这时候，对于新产品的研发估点而言，同样可以初始估点中的方法，因为将目标转化成以DHR为时间单位的思路和方法是相同的。同时，通过对历史数据的计数分析，可以采用统计学中的一些方法进行评估，得到对产品开发时间的另一种估算结果。将使用统计估算的结果与初始估点的估算结果进行比较，可以进一步判断估点的置信区间，从而提高估点的准确性和可信度。

对历史数据的提取和采集

对哪些历史数据进行选取并作为估点的依据呢？同样存在很多的方法，比较简单有效的历史数据就是代码行数了。尽管代码行数又着各种各样的局限，但是以其他数据作为估算的依据可能会更糟糕。

对于存储代码的版本管理工具而言，Git 几乎是大多数开发团队的首选。在Git的开源社区中有一些可视化的工具如gitk，giggle等，可以用来查看产品的开发历史。但对于大型的项目，这些简单的可视化工具就可能不足以了解完整的开发历史了，因为一些定量的统计数据（如每日提交量，行数等）更能反映开发进程和活跃性。GitStats是笔者推荐的一个好工具，它是一个Git仓库分析软件，可以帮助我们查看Git仓库的状态，自动生成相关的数据图表，它所生成的统计数据如下：

- 常规的统计：文件总数，行数，提交量，作者数。
- 活跃度：每天中每小时的、每周中每天的、每周中每小时的、每年中每月的、每年的提交量。
- 所有参与开发的作者数据：列举所有的开发者（提交数，第一次提交日期，最近一次的提交日期），并按月和年来进行划分。
- 文件数：支持按日期划分以及按文件的扩展名来划分。
- 以及代码行数：按日期划分。

GitStats 的下载网址为 http://gitstats.sourceforge.net/ ，也可以从 github 上获得：https://github.com/trybeee/GitStats， 这是一个基于Python 的程序，调用git 自身的相关命令获取数据，使用Gnuplot 作为绘图工具，最终生成HTML的文件作为输出结果。

GitStats的使用方法非常简单，示例如下：

```
./gitstats /home/abel/git/project ~/gitstats_html/project
```

Git项目在/home/abel/git/project下，生成的统计数据放在~/gitstats_html/project目录下。以笔者经历的一个产品为例，gitstats输出的常规信息如下：

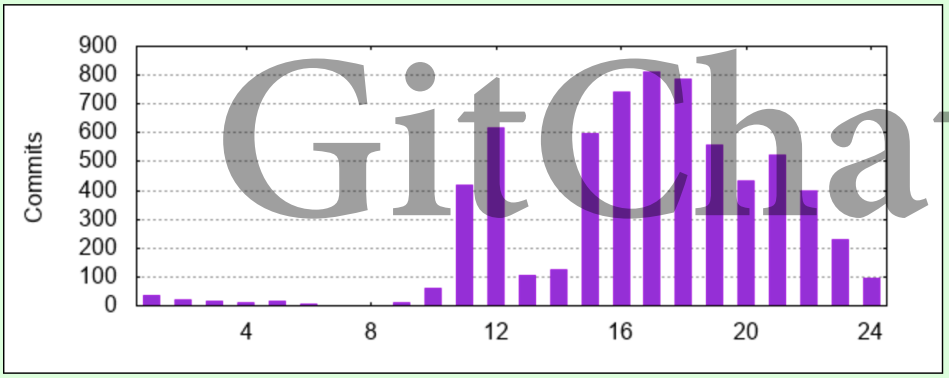
Generator:
GitStats (version 55c5c28), git version 2.8.4 (Apple Git-73), gnuplot 5.0 patchlevel 6
Report Period:
2016-03-01 09:15:42 to 2017-04-14 01:17:48
Age:
409 days, 329 active days (80.44%)
Total Files:
2772
Total Lines of Code:
489853 (1882629 added, 1392776 removed)
Total Commits:
6614 (average 20.1 commits per active day, 16.2 per all days)

从中可以发现一些有趣的数字，比如增加了1882629行代码，同时删除了1392776行代码。是代码的重构还是需求变化导致的呢？

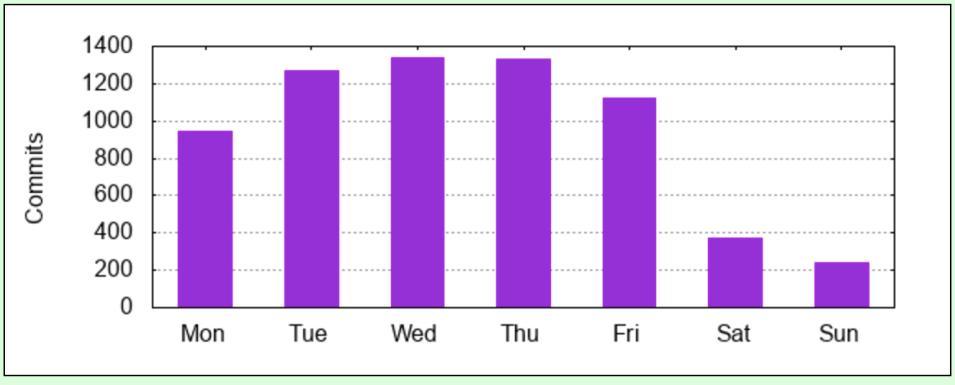
看一看每天中哪个时段或者一周中的哪一天代码的提交比较频繁：

Hour of Day

Hour	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Commits	37	21	15	12	17	7	2	2	10	60	416	617	105	122	595	739	812	786	559	432	523	400	230	95
%	0.56	0.32	0.23	0.18	0.26	0.11	0.03	0.03	0.15	0.91	6.29	9.33	1.59	1.84	9.00	11.17	12.28	11.88	8.45	6.53	7.91	6.05	3.48	1.44



Day of Week



Day	Total (%)
Mon	946 (14.30%)
Tue	1266 (19.14%)
Wed	1340 (20.26%)
Thu	1327 (20.06%)
Fri	1125 (17.01%)
Sat	374 (5.65%)
Sun	236 (3.57%)

还可以看到每个开发者在该产品中的贡献情况：

Commits (%)	+ lines	– lines	First commit	Last commit	Age	Active days	# by commits
1437 (21.73%)	99875	49204	2016-03-03	2017-04-13	406 days, 2:22:57	269	1
766 (11.58%)	56183	21487	2016-03-03	2017-04-14	406 days, 3:47:02	231	2
452 (6.83%)	162674	8727	2016-07-18	2017-04-13	269 days, 10:44:53	139	3
415 (6.27%)	17672	13807	2016-07-06	2017-04-13	281 days, 1:58:25	88	4
402 (6.08%)	0	0	2016-07-08	2017-04-14	279 days, 9:41:24	119	5
355 (5.37%)	41375	27353	2016-10-20	2017-04-13	174 days, 21:26:23	75	6
319 (4.82%)	17891	89623	2016-07-15	2017-04-13	272 days, 2:42:15	109	7
303 (4.58%)	16030	2822	2016-03-03	2016-07-01	119 days, 10:46:47	65	8
269 (4.07%)	24535	7718	2016-03-06	2016-09-30	208 days, 4:18:51	74	9
226 (3.42%)	12110	1854	2016-07-08	2017-04-14	279 days, 9:46:21	78	10
225 (3.40%)	18957	4653	2016-07-18	2017-04-13	268 days, 22:39:59	82	11
219 (3.31%)	16282	9137	2016-07-06	2017-04-01	268 days, 22:46:45	66	12
202 (3.05%)	22970	1132590	2016-06-17	2016-12-23	188 days, 21:32:02	62	13
183 (2.77%)	20586	8619	2016-03-03	2016-07-14	132 days, 20:14:06	65	14
131 (1.98%)	9050	3401	2016-07-07	2017-04-14	280 days, 14:46:20	47	15
127 (1.92%)	15678	20324	2016-07-07	2017-01-16	193 days, 1:31:56	53	16
101 (1.53%)	2965	694	2016-04-21	2016-06-29	68 days, 23:59:10	36	17
96 (1.45%)	7140	4121	2016-08-22	2017-04-13	234 days, 5:44:24	35	18
83 (1.25%)	8249	1793	2016-03-05	2016-04-07	32 days, 23:08:03	19	19
73 (1.10%)	1427102	45831	2016-03-01	2016-06-29	120 days, 8:10:39	20	20

如果有其他特殊的需要，可以参考gitstats的python 源代码，进行按需定制。

基于统计数据的估算

基于统计数据的估算有着一些基本的假设，例如开发人员的开发时间全部应用于某一产品的开发，而不是时分复用，不同产品之间是相对独立的等等。通过对大目标的估算分解成对小任务的估算，利用大数法则，让偏大的误差和偏小的误差在一定程度上相互抵消。

其中的一个难点和不确定性是backlog与代码行数之间的对应关系，一个功能的实现采用不同的编程语言代码量不同，比如通过http 请求获取一个页面，Java可能需要30行代码，而Python可能不超过5行。如果采用相同语言，使用不同的库导致代码量同样会有较大差别。即使采用相同的编程语言和相同的库，开发人员本身的技能水平同样会导致代码量差异。

因此，基于统计数据的估算一般来说是面向开发者个人的，也就是说，首先要保持团队的相对稳定，然后让开发者根据自己的数据进行估算比较好，因此，针对同一个业务的开发，不同的开发者建立的backlog 可能是不同的。

如何找到每个backlog对应的代码量呢？如果使用trello 来跟踪backlog状态的话，可以通过trello的开发者API 通过程序来获得每个backlog的时间段，同时在流程中约定，在每个backlog 的DHR过程中中必须提交代码，这样就可以从git仓库中针对每个开发者的每个backlog进行代码量的统计了。

至于backlog 之间的相似性，也是以开发者自身的纵向对比为主。因为一个资深的工程师和一个一般水平程序员之间的横向对比往往不具备可比性，这或许就是所谓“10倍生产率”的一个表现。

举个简单的例子，如果工程师A历史数据中每个backlog的代码行数平均值为100行，标准差是30行的话，就可以尝试根据正态分布计算置信区间了。

其他参考模型的估算

当然，这时也可以参考其他常见的软件估算模型进行多元估点，例如Putnam模型。Putnam是一种动态多变量模型，其中L代表源代码行数，K代表开发的工作量，Tdev表示开发时间，Ck是技术状态常数取值因开发环境而定，得到的开发时间估算公式如下：

$$Tdev = 1/4 \times \frac{L}{Ck \times K}$$

还有比较有名的COCOMO II 模型，在COCOMO II 模型中关于进度的估算公式如下：

$$Tdev = \frac{3.67 * (PM)^{0.28+0.2*(B-0.91)} * (SCED\%)}{100}$$

具体解释参见参考阅读。

需要注意的是，传统估算模型都是以人 / 天，人 / 月甚至人 / 年为单位的，我们要转换成以DHR为单位，那些参数也需要根据自己的历史数据进行不断的校准。

这样，我们就可以尝试用多元估点的方法对估算的最终结果进行比较和进一步评估了。

处理产品与研发间的估点矛盾

产品经理和研发人员的矛盾主要是开发周期的目标与估点结果之间的矛盾。作为一名研发人员或者技术管理者，在与产品经理或者项目经理进行沟通的时候最好保持以下原则：

- 把人和问题分开，也就是我们提倡的“对事不对人”。
- 更关注利益，产品的哪些功能交付可以为团队乃至公司带来怎样的利益，而不是出于不同分工的立场。
- 我们是一条绳上的蚂蚱，创造可以共同获利的可行方案。
- 坚持使用客观标准，任何的主观臆断都可能会导致相互间误会的加剧。

沟通中的要素

我们要记下估算中包含的假设，并就此进行沟通。同时，明确表达的是估算结果中的不确定性，而不是自己达到承诺的能力的不确定性。不要向其他干系人提供只有很小可能的估算结果，最好以图形代表文本作为估算值的表达形式。

不要用范围表示承诺，承诺应该是明确的，也就是说，我们承诺何时可以完成就要在那个时间点必须完成，这是一种职业的态度和操守。可以对承诺进行沟通，但不要对估算值进行谈判，让产品 / 项目经理了解有效的估算实践是有意义的，最好让他们帮助检查估算中10个问题：

1. 是否明确定义了估点目标？
2. 是否包括完成任务所需的所有工作类型？
3. 是否包含了完成任务所需的所有功能领域？
4. 是否被分解到足够详细的程度，可以揭示所有隐藏的工作？
5. 是否使用来自过去的历史数据，设定的生产率是否接近于类似工作所达到的生成率？
6. 是否被实际要完成开发工作的工程师所认可？
7. 是否分别包含了最好情况，最差情况和最可能情况,最差情况是否真的最差？是否还有更差？
8. 是否从这些情况中正确的计算出了预期情况？
9. 是记录了估算中的假设？
10. 估算做出后是否发生了改变？

除非有定量推算的方法，否则不要提供“百分之多少的置信度”形式的估算值（尤其是“90%置信度”），从个人经验上看，大部分直觉上的90%置信度实际上相当于30%置信度。

妥协与共赢

谨慎对待进度压缩和最短的可能进度，因为缩短名义上的进度实际上会增加总体工作量的。由于可能存在难以突破的或者难以实现的关键点，如果我们必须要面对压缩进度，最好不要让进度缩短的幅度超过25%。如果缩减团队规模，使进度变得宽松一些，通常会减少总工作量。也就是说，延长进度并采用较小的团队，可降低开发的成本。但需要注意的是，让进度延长超过30%很可能会产生各种低效的情况，反而会增加成本。

最后期限的压力往往是软件工程中最危险的敌人。过度紧张的或不合理的进度是对所有产品开发最具破坏力的影响因素。所以，尽量不要故意低估，低估带来的损失比高估带来的损失更严重。最好通过计划和控制来解决对高估的顾虑，而不要故意降低估算值。

高估带来的损失往往是线性而且是有限的，但是，低估带来的损失是非线性增长的而且是没有限制的，很多时候，更多bug所产生的损害比高估要严重的多。

在讨论进度的时候，提出尽可能多的可选计划，为达到开发的目标提供支持。在形成合作式解决问题的气氛时，千万不要根据即兴估算（拍脑袋）做出任何承诺。也就是说，不要对估算结果本身进行沟通，坚持由有资格的人来进行估算，参考所在开发组的历史数据和估算方式，经受住理念冲突的考验。 当遇到僵局的时候，只思考一个问题————“怎样才对我们的组织 / 公司最有利”。

回顾与小结

对软件开发的估算是对开发持续时间的一种预测，以期望达到产品和业务的目的，进而许诺在特定的日期之前以特定的质量水平交付规定的功能。

估算应该是相对客观的分析过程，目的是得到相对准确的结果；而规划与计划一般是主观的目标求解过程，目的是寻求一种特定的结果。在传统的软件工程中，估算的准确度最高可达±10%,也只有在控制很好的项目中才能达到。估算的首要目标不是预测最终的结果，而是确定目标是否能够实现，从而在可控的状态下完成这些目标，无需非常准确而是要有效。良好的估算是对项目实际情况有足够清晰的看法，让管理层可以作出可控而且能够达到目标的决策。

具体地说，以DHR作为初始估算的时间单位，确定目标需求的边界，进而检查功能的完备性以及非功能性约束是否遗漏，得到估点的期望值。进一步，以历史数据为依据，通过统计方法或其他估算模型进行多元估点，对多种估算结果进行比较，可以得到置信区间以及对估算的结果进行纠偏。 最后，与产品 / 管理团队沟通协商做出承诺，团结一致，全力做好产品。

参考阅读：

- 《软件成本估算：COCOMO II 模型方法》
- 《软件估算：“黑匣子”揭秘》
- 《software engineering metrics and models》