

“系统架构部”周刊2020年02期

0. 目录

读完本期周刊，大概需要花费您10分钟的宝贵时间，不过超值哦。

1. 心理测试: 你是不是一个完美主义者?
2. 微基准测试 Microbenchmarks
3. 让人心慌的抖动(Jitter)
4. 正则表达式如何方便地测试?
5. “吃着火锅还唱着歌”，友好API设计之范例CopyFile
6. IT人的十级自由，快来测测，你在第几级?
7. 理解TCP三次握手
8. 用YYYY-MM-dd的程序员，小心“被祭天”
9. 谷歌的代码回顾最佳实践，Code Review应该关注什么?
10. 历史周刊回顾

周刊每周五准时发布，欢迎您[提交Issue](#)或者联系主编黄老邪(微信号bingoohuang)，向本周刊投递有值得分享的内容，简短的一张图一段文亦可。

1. 心理测试: 你是不是一个完美主义者(Perfectionist)



你是不是经常为自己设定目标，而且最终往往达不到那些目标？

1. 当一件事件做得不够好时，你是否有再试一次的想法?
2. 你是否休息时仍想着学习、工作、或者其他还没有解决的事?
3. 如果遇到别人说话或打岔，破坏了你的注意力，你是否觉得不太高兴?
4. 你是否常常会在事后想，如果当时能换成另一种方式来解决，也许会更加理想?

如果上面的问题，你的回答全是肯定的，那么你就有完美主义倾向。简单说，完美主义就是追求一个较高水平的目标，不接受一个较低水平的、但可用的结果。这是不好的。我就一直告诫自己，千万不要追求完美。就像一篇文章说的，完美主义的最大问题是，它实际上让你追求高成本。

完美主义是一种压力，它让你为自己创造不切实际的期望。你明明做到了正常水平，但是因为设定的目标太高，所以看上去距离目的地仍然很遥远。你的心态变成了：这一切还不够好，依然可以改进。完美主义消耗了我们最宝贵的资源和时间，让你将注意力从真正的优先事项上移开。

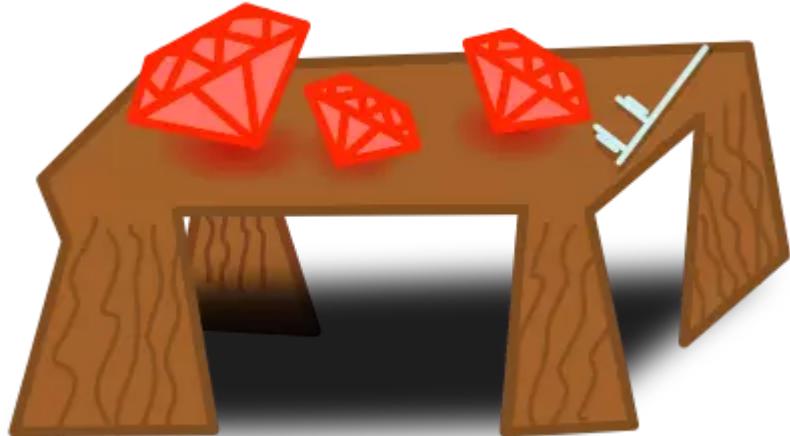
“做得快”比“做得好”更重要。我们需要的是，低成本地做出尽量多的成果，而不是高成本地创造一件精品。完美主义会妨碍我们“做得快”。完美主义带来的高压力，也不利于身心健康。

2. 微基准测试(Microbenchmarks)

一天，徐梓郡问我：黃老邪， Runtime.getRuntime().totalMemory()这个方法耗性能吗？

我说：我亦不知，用微基准跑一下可知也。

什么是微基准测试？



微基准测试测试的代码很少，有时仅仅是一个微小的操作。如果你想知道一个微小操作有多快(例如传递一个块，一个循环、一个整数自增或一个 map)，一个微基准测试就可以非常精确地测量它，而几乎不测量其他任何东西。

JAVA中的基准测试框架JMH



```
java -jar target/benchmarks.jar
# JMH version: 1.22
# VM version: JDK 1.8.0_212, OpenJDK 64-Bit Server VM, 25.212-b04
# Warmup: 3 iterations, 10 s each
# Measurement: 8 iterations, 10 s each
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark:
com.github.bingoohuang.gotcha.jmh.BenchmarkRuntime.totalMemory
...
# Run progress: 50.00% complete, ETA 00:01:50
# Fork: 2 of 2
...
# Warmup Iteration 3: 38.898 ns/op
Iteration 1: 41.039 ns/op
...
Iteration 8: 38.257 ns/op

Result "com.github.bingoohuang.gotcha.jmh.BenchmarkRuntime.totalMemory":
39.231 ±(99.9%) 1.293 ns/op [Average]
(min, avg, max) = (38.198, 39.231, 42.260), stdev = 1.270
CI (99.9%): [37.937, 40.524] (assumes normal distribution)

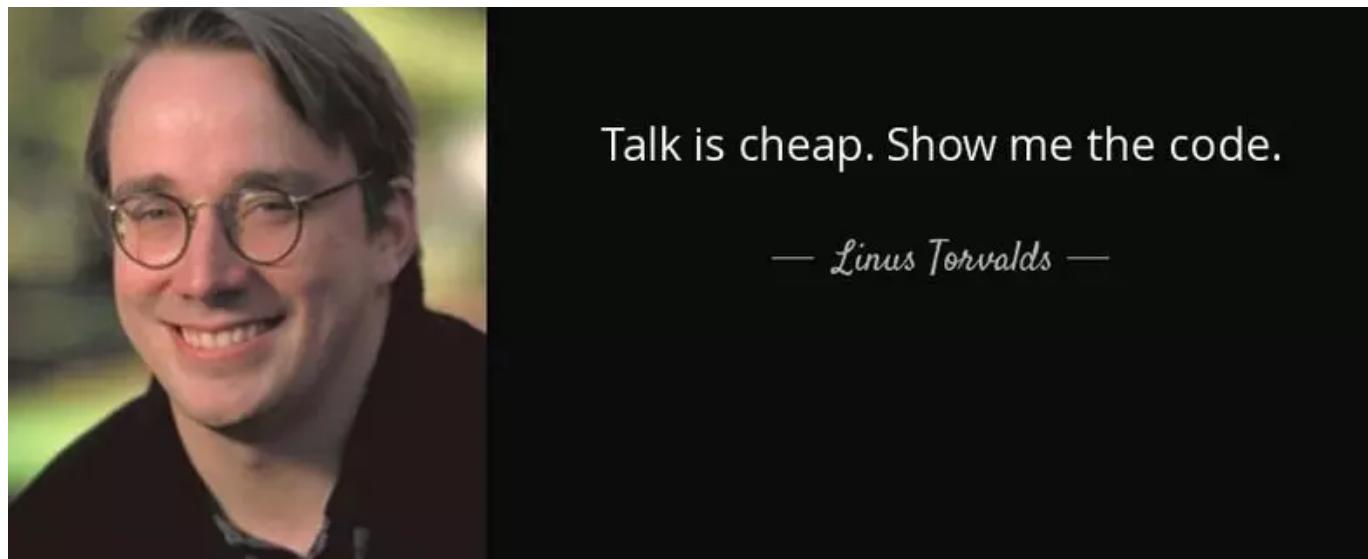
# Run complete. Total time: 00:03:40
...
Benchmark Mode Cnt Score Error Units
BenchmarkRuntime.loopFor avgt 16 39.231 ± 1.293 ns/op
```

GO中的微基准测试（内建）

```
$ go test -bench=.
goos: darwin
goarch: amd64
pkg: github.com/bingoohuang/golang-trial/synk
BenchmarkOnce2-12 1000000000 0.165 ns/op
```

BenchmarkOnce-12	1000000000	0.168 ns/op
PASS		
ok	github.com/bingoohuang/golang-trial/synk	0.379s

计算机科学有个好处，不用猜，可以实证。就像大神Linus说的那样：



3. 让人心慌的抖动(Jitter)

负载抖动了

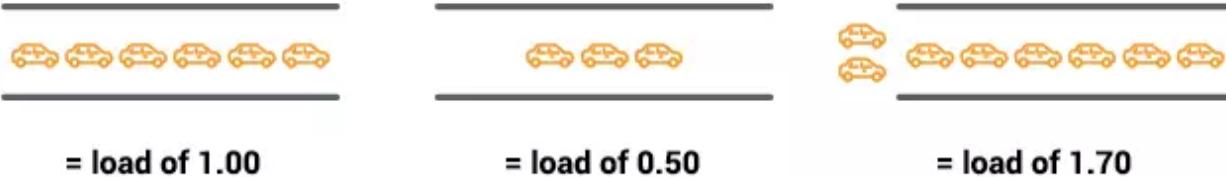
一台全新的机器，还没来得及安装任何应用，先把监控(telegraf)加上去，然后就发现，机器负载(Load)竟然不定期规律抖动(Jitter)，让我的小心脏心慌慌，到底咋了？



图片来自telegraf issues 3465

负载是个啥

先紧急补充一下对于负载(Load)的理解，[一图胜千言](#)：



一个很生动的比喻：有一座只有单车道的大桥，这座大桥能同时容纳的通车数是有上限的，假设为10。那么：

- $\text{load} = 0$ 时，代表桥上没有车辆通行，桥完全处于空闲状态。
- $\text{load} = 0.5$ 时，代表桥上有50%的车辆，但还剩下50%的容量。事实上，只要Load还未达到1，桥就还有空余容量，所以此时如果有新的车辆到来，是可以直接上桥不用排队的。
- $\text{load} = 1$ 时，代表桥上有100%的车辆，已经没有空余，但也没有车在排队。
- $\text{load} = 1.7$ 时，代表桥上有100%的车辆，已经没有空余，而且有70%的车在排队等待上桥。事实上，只要Load大于等于1，就代表桥已经满载，所以此时如果有新的车辆到来，就需要在外面排队等待。

对应到计算机术语里来，大桥代表着CPU，汽车代表着任务，进车过桥代表着任务在使用CPU时间，汽车排队代表着任务在等待CPU分配时间片。

咋抖了呢

注：以下内容，公式跳过，不明觉厉即可，认真解读需要交流的请联系刘金良同学。

根据 linux内核`include/linux/sched/loadavg.h` 定义`EXP_1=1884`而`FIXED_1=1<<11`, 通过替换 $\Delta t=5$, $t=60$ 后得到的公式为

$$L(t) = (1 - e^{-1/12})p(t) + e^{-1/12}L(t - \Delta t)$$

Linux内核为了加速计算，采用shift 11位来计算，也就是 $e^{-1/12} * 2^{11}$ 的近似计算值为1884。

据此，我们理解了loadavg的计算，实际上linux内核的采样周期为5.001，这样的原因是，如果一个定时任务总是落在5s内，那么linux内核就可能采样不到真实的负载情况，加这个偏移，那么就可以在一个固定的周期内体现出系统的复杂。这也是我们系统的监控图总是周期化尖峰的原因。如果需要关注具体案例请参考[cpuload 负载高问题跟踪](#)

咋消除抖动

在telegraf的配置文件中增加`collection_jitter = "5s"`，即可，[详见](#)。

4. 正则表达式如何方便地测试

不想装什么软件，也不想搞什么破解，互联网时代，我就想单纯地用个在线的版本（多么美好的事情）。那小编就推荐[regex101](#)了，它提供表达式高亮，解读，测试，以及分享，非常强大，需要写正则人从此有福了。表达式语法高亮，解释，测试，一气呵成。对正则表达式还是很了然的，右下角有Reference哦。对英文不甚了然的，上期有推荐有[彩云小译火力全开](#)哦。

The screenshot shows the regex101.com interface. On the left, there's a sidebar with 'SAVE & SHARE' (Update Regex, Fork Regex, Delete Regex), 'FLAVOR' (PCRE (PHP) selected, ECMAScript (JavaScript), Python, Golang), and 'TOOLS' (Code Generator, Regex Debugger). The main area has a 'REGULAR EXPRESSION v1' section with the pattern `\n|\N\K<{.*?}>(?=\\n|\\x)` and a 'TEST STRING' containing several lines of log file content. The 'EXPLANATION' panel on the right details the regex components: '\n' matches the character 'n' literally (case sensitive), '\N' resets the starting point of the reported match, and the capturing group '<{.*?}>' matches any character (except for line terminator) zero or unlimited times, as few times as possible, expanding as needed (Lazy). The 'MATCH INFORMATION' panel shows two matches found. The 'QUICK REFERENCE' panel on the bottom right lists various regex symbols and their meanings.

5. “吃着火锅还唱着歌”，友好API设计之范例CopyFile

假设有这么一个API，这里以Go语言为例（Java类似）

```
func CopyFile(to, from string) error
```

虽然从源代码看，to和from的函数参数命名所包含的意思很明显，勿须过多解释（自解释），但是如果阅读实际调用代码，你还会清楚地记得哪个参数是from，哪个参数又是to么？

```
CopyFile("/tmp/backup", "presentation.md")
CopyFile("presentation.md", "/tmp/backup")
```

大概率是，你不得不查阅文档/代码，然后才分辨出来了（无法自解释）。本来看业务代码，看得正起劲（你带着老婆出了城，吃着火锅还唱着歌），突然顿住（到底从那拷贝文件到哪啊）了，要查阅其文档/代码，就从阅读的当前位置打断了（突然就被麻匪劫了），“跳走了”，看完还得跳回来，重新整理思绪，继续“前行”。这就破坏了阅读代码的连续性，增加了“额外的上下文切换的成本”，读代码就缺乏自然性了。如果换一种API的设计方式，如下：

```
type Source string
```

```

func (src Source) CopyTo(dest string) error {
    return CopyFile(dest, string(src))
}

func main() {
    var from Source = "presentation.md"
    from.CopyTo("/tmp/backup")
}

```

这种API的设计方式就非常友好，每当读到CopyTo的时候，就非常清晰，不需要去查阅文档或者源代码了，阅读代码，就非常自然清晰了。（如果此时，你还要去查阅CopyTo的源代码，只能说明一个问题，你不信任实现CopyTo的小伙伴，你累不累啊。）

以整个团队为单位，读代码与写代码的时间有一个比例，大概是9:1，也就是说花1小时时间写的代码，未来整个团队可能要花9个小时的时间来阅读它。代码可读性长期来看，是对团队的一种“极大的友好”，但是写完代码后的相当一段时间，都是自己阅读，代码可读性从近期来看，却是对“自己”的一种“友好”。

[本文摘自编写可维护程序的建议](#)

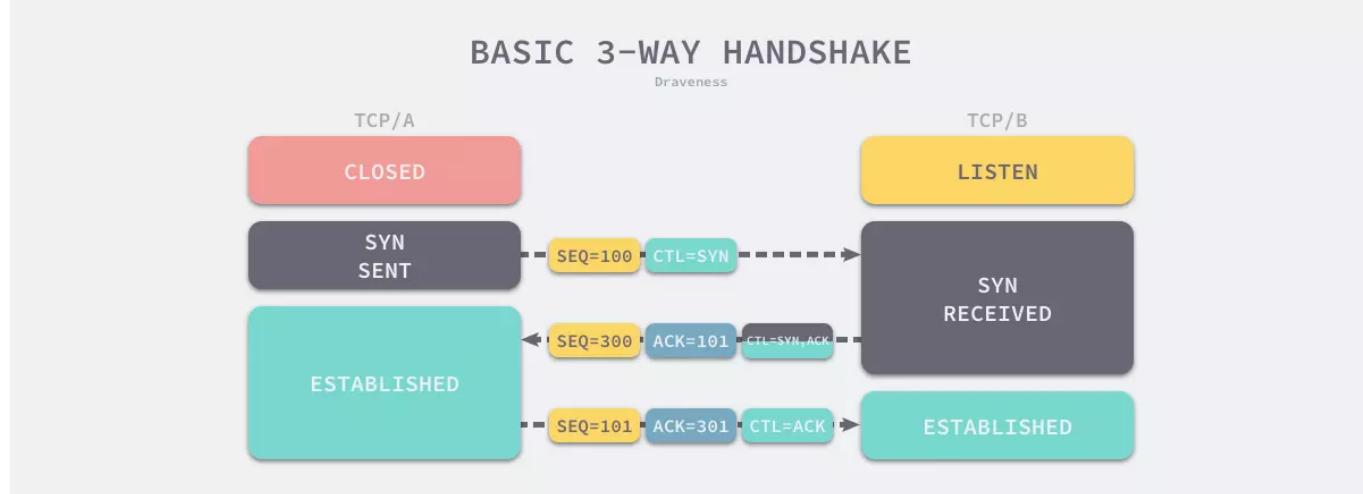
6. IT人的十级自由，快来测测，你在第几级



[来源](#)

7. 理解TCP三次握手(TCP three-way handshakes)

三次握手怎么握的



图片来源

因为双方都需要和对方同步 seq 号，所以需要来回确认。服务器把 SYN 和 ACK 合并成一条消息，所以最终只需要三次交流就可以了。当然如果你想把 SYN 和 ACK 拆开成两个消息也可以，只不过协议栈一般不这样实现。

比如你参加一次相亲聚会，看到一个漂亮的姑娘，你想去搭讪，首先得先了解一下。

你：姑娘您好，贵庚啊？
 姑娘：小女子18，请问大叔您贵庚啊？
 你：我81了

这样寒暄之后你们双方就可以进一步的深入的交流了。

那么，从北京到上海的建立一个TCP连接大概需要多久呢？

内容来自[为什么 TCP 会被 UDP 取代](#)。北京到上海的直线距离约为 1000 多公里，而光速是目前通信速度的极限，所以 RTT 一定会大于 6.7ms：

$$RTT = 1,000,000m \div 300,000m/ms \times 2 = 6.7ms$$

在网络通信中，从发送方发出数据开始到收到来自接收方的确认的时间被叫做往返时延（Round-Trip Time, RTT）。

光在光纤中不是直线传播的，真正的传输速度会比光速慢 ~31%，而且数据需要在各种网络设备之间来回跳转，所以很难达到理论的极限值。实际环境中从北京到上海的 RTT 大概在 40ms 左右，所以 TCP 建立连接所需要最短时间也需要 60ms (1.5RTT)。顺便问一个问题：**60毫秒，对于CPU而言，是一个什么样的存在**，答案将在下期揭晓。

8. 用YYYY-MM-dd的程序员，小心“被祭天”

来自[就在几天前，听说用了 YYYY-MM-dd 的程序员，都在加班改 Bug !](#)

一大叔北漂十多年，一直未摇到京牌，每周都要办理“进京证”。当他办理2019年最后一次进京证的时候，APP给出了这样的提示：



日期显示：2020-12-31！

立马有人不淡定了，虽然大家都猜出来，这应该是 APP 的 Bug，但还是难免要吐槽一下，讨论到最后，就快要杀个程序员祭天了！

那么产生这个 Bug 的原因是什么呢？其实很简单，就是把 yyyy-MM-dd 写成了 YYYY-MM-dd。

SimpleDateFormat的javadoc

Letter	Date or Time Component	Presentation	Examples
y	Year 正正经经的年	Year	1996; 96
Y	Week year 不正经的年，若本周跨年就算入下一年	Year	2009; 09
M	Month in year (context sensitive)	Month	July; Jul; 07
D	Day in year	Number	189
d	Day in month	Number	10
H	Hour in day (0-23)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978

[在线可运行版本](#)，可以点进去运行一下以下代码

```
import java.util.Calendar;
import java.util.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

class Main {
    public static void main(String[] args) {
        Calendar calendar = Calendar.getInstance();
        calendar.set(2019, Calendar.DECEMBER, 31);
        Date strDate = calendar.getTime();

        DateFormat f1 = new SimpleDateFormat("yyyy-MM-dd");
        System.out.println("2019-12-31 to yyyy-MM-dd: " + f1.format(strDate));
        // 2019-12-31 to yyyy-MM-dd: 2019-12-31

        DateFormat f2 = new SimpleDateFormat("YYYY-MM-dd");
        System.out.println("2019-12-31 to YYYY-MM-dd: " + f2.format(strDate));
        // 2019-12-31 to YYYY-MM-dd: 2020-12-31
    }
}
```

黄老邪的经验是，不要手写，要学会拷贝黏贴，虽然“大拷伤身”，但是“小拷怡情”，需要什么格式，从下面表格中拷贝，拿走不谢：

SimpleDateFormat格式Pattern	Result
---------------------------	--------

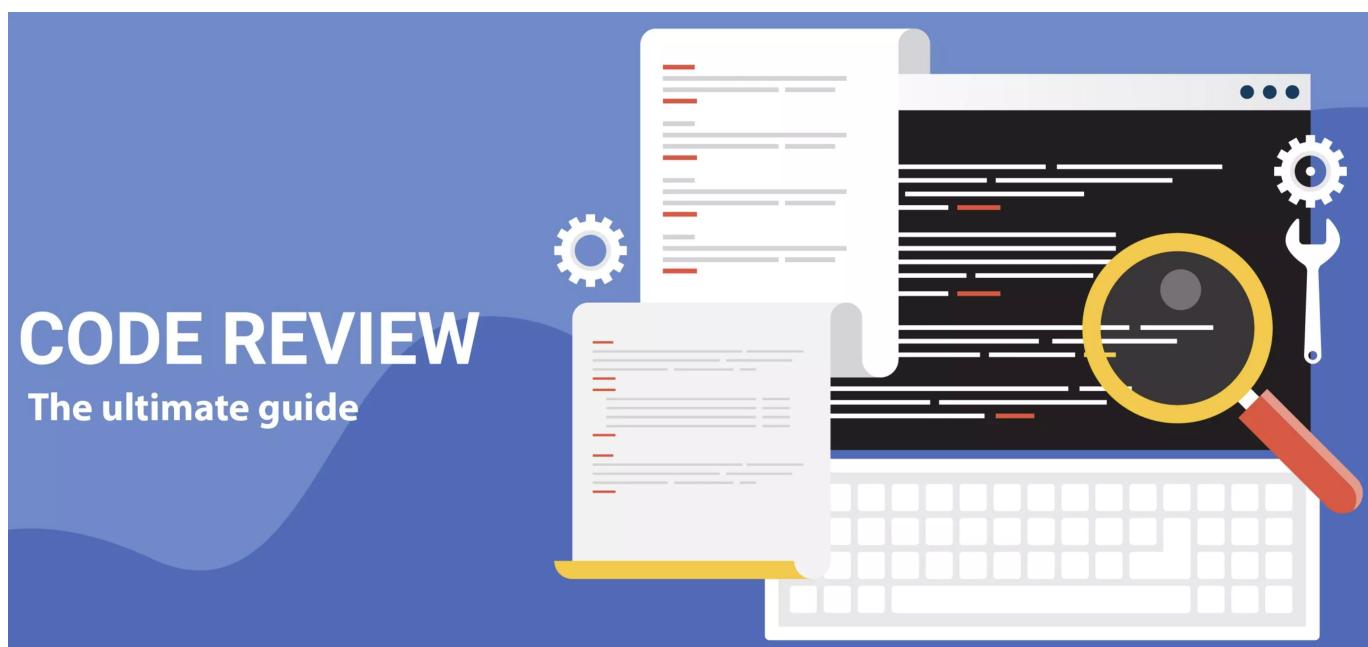
SimpleDateFormat格式Pattern	Result
yyyy.MM.dd G 'at' HH:mm:ss z	2001.07.04 AD at 12:08:56 PDT
yyyyy.MMMM.dd GGG hh:mm aaa	02001.July.04 AD 12:08 PM
yyMMddHHmmssZ	010704120856-0700
yyyy-MM-dd'T'HH:mm:ss.SSSZ	2001-07-04T12:08:56.235-0700
yyyy-MM-dd'T'HH:mm:ss.SSSXXX	2001-07-04T12:08:56.235-07:00

帮人帮到底吧，黄老邪再贴心一点，把MySQL中的日期格式，也摘录一下附送了，方便你拷贝，[原始版本](#)，[在线验证](#)：

DATE_FORMAT(NOW(),'%Y-%m-%d %T')

2020-01-08 06:16:18

9. 谷歌的代码回顾最佳实践，Code Review应该关注什么



图片来自[Code Review — The Ultimate Guide](#)

上一期，讲到[评审者指南](#)，其中提到

指导性: 代码回顾有个重要的作用，那就是可以教会开发者关于语言、框架或者通用软件设计原理。

那么Code Review应该关注什么呢，参考以下列表，以后代码回顾，就有参考了：

1. 代码设计良好。
2. 代码功能对用户有用。
3. 任何UI改动应当是深思熟虑过而且看起来不错的。
4. 保证线程安全。
5. 代码没有增加不必要的复杂性。
6. 开发者没有写有些将来需要但现在不知道是否需要的东西。
7. 代码有适当的单元测试。

8. 测试逻辑设计良好。
9. 开发者使用了清晰明了的命名。
10. 注释清晰明了实用，通常解释清楚了为什么这么做，而不是做了啥。
11. 代码又相应完善的文档。
12. 代码风格符合规范。

10. 历史周刊回顾

[上一期周刊](#)

