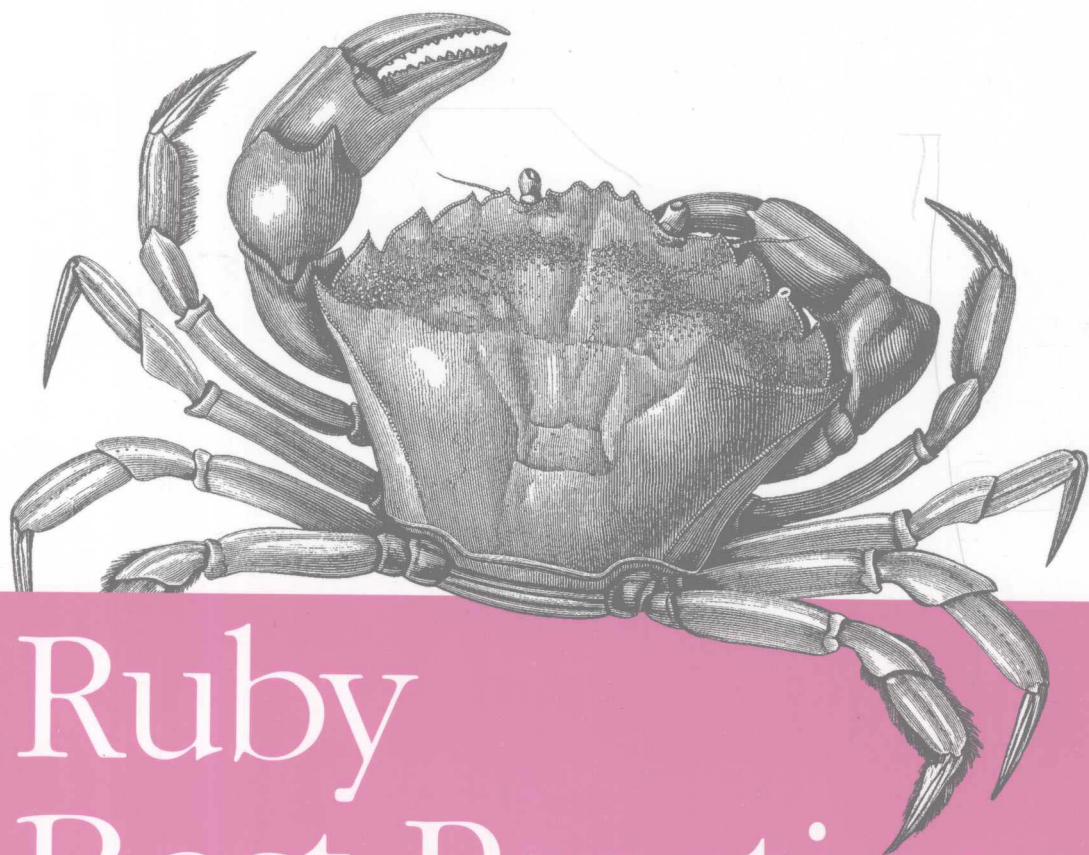


Ruby最佳实践 (影印版)



Ruby Best Practices

O'REILLY®

東南大學出版社

Gregory T. Brown 著
Yukihiro Matsumoto 序

Ruby最佳实践 (影印版)

Ruby Best Practices

Gregory Brown

foreword by Yukihiro “Matz” Matsumoto

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权东南大学出版社出版

东南大学出版社

图书在版编目 (CIP) 数据

Ruby 最佳实践: 英文 / (美) 布朗 (Brown, G.T.)
著. —影印本. —南京: 东南大学出版社, 2010.1

书名原文: Ruby Best Practices

ISBN 978-7-5641-1935-5

I . R… II . 布… III . 计算机网络—程序设计—英文
IV . TP393.09

中国版本图书馆 CIP 数据核字 (2009) 第 205661 号

江苏省版权局著作权合同登记

图字: 10-2009-246 号

©2009 by O'Reilly Media, Inc.

Reprint of the English Edition, jointly published by O'Reilly Media, Inc. and Southeast University Press, 2009. Authorized reprint of the original English edition, 2009 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2009。

英文影印版由东南大学出版社出版 2009。此影印版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式复制。

Ruby 最佳实践 (影印版)

出版发行: 东南大学出版社

地 址: 南京四牌楼 2 号 邮编: 210096

出 版 人: 江 汉

网 址: <http://press.seu.edu.cn>

电子邮件: press@seu.edu.cn

印 刷: 扬中市印刷有限公司

开 本: 787 毫米 × 980 毫米 16 开本

印 张: 20.75 印张

字 数: 349 千字

版 次: 2010 年 1 月第 1 版

印 次: 2010 年 1 月第 1 次印刷

书 号: ISBN 978-7-5641-1935-5

印 数: 1~1600 册

定 价: 48.00 元 (册)

本社图书若有印装质量问题, 请直接与读者服务部联系。电话 (传真): 025-83792328

Foreword

In 1993, when Ruby was born, Ruby had nothing. No user base except for me and a few close friends. No tradition. No idioms except for a few inherited from Perl, though I regretted most of them afterward.

But the language forms the community. The community nourishes the culture. In the last decade, users increased—hundreds of thousands of programmers fell in love with Ruby. They put great effort into the language and its community. Projects were born. Idioms tailored for Ruby were invented and introduced. Ruby was influenced by Lisp and other functional programming languages. Ruby formed relationships between technologies and methodologies such as test-driven development and duck typing.

This book introduces a map of best practices of the language as of 2009. I've known Greg Brown for years, and he is an experienced Ruby developer who has contributed a lot of projects to the language, such as Ruport and Prawn. I am glad he compiled his knowledge into this book.

His insights will help you become a better Ruby programmer.

—Yukihiro “Matz” Matsumoto
June 2009, Japan

Preface

Some programming languages excel at turning coders into clockwork oranges. By enforcing rigid rules about how software must be structured and implemented, it is possible to prevent a developer from doing anything dangerous. However, this comes at a high cost, stifling the essential creativity and passion that separates the masterful coder from the mediocre. Thankfully, Ruby is about as far from this bleak reality as you can possibly imagine.

As a language, Ruby is designed to allow developers to express themselves freely. It is meant to operate at the programmer's level, shifting the focus away from the machine and toward the problem at hand. However, Ruby is highly malleable, and is nothing more than putty in the hands of the developer. With a rigid mindset that tends to overcomplicate things, you will produce complex Ruby code. With a light and unencumbered outlook, you will produce simple and beautiful programs. In this book, you'll be able to clearly see the difference between the two, and find a clear path laid out for you if you choose to seek the latter.

A dynamic, expressive, and open language does not fit well into strict patterns of proper and improper use. However, this is not to say that experienced Rubyists don't agree on general strategies for attacking problems. In fact, there is a great degree of commonality in the way that professional Ruby developers approach a wide range of challenges. My goal in this book has been to curate a collection of these techniques and practices while preserving their original context. Much of the code discussed in this book is either directly pulled from or inspired by popular open source Ruby projects, which is an ideal way to keep in touch with the practical world while still studying what it means to write better code.

If you were looking for a book of recipes to follow, or code to copy and paste, you've come to the wrong place. This book is much more about how to go about solving problems in Ruby than it is about the exact solution you should use. Whenever someone asks the question "What is the right way to do this in Ruby?", the answer is always "It depends." If you read this book, you'll learn how to go with the flow and come up with good solutions even as everything keeps changing around you. At this point, Ruby stops being scary and starts being beautiful, which is where all the fun begins.

Audience

This book isn't really written with the Ruby beginner in mind, and certainly won't be very useful to someone brand new to programming. Instead, I assume a decent technical grasp of the Ruby language and at least some practical experience in developing software with it. However, you needn't be some guru in order to benefit from this book. The most important thing is that you actually care about improving the way you write Ruby code.

As long as you have at least an intermediate level of experience, reading through the book should be enjoyable. You'll want to have your favorite reference book handy to look things up as needed. Either *The Ruby Programming Language* (<http://oreilly.com/catalog/9780596516178/>) by David Flanagan and Yukihiro Matsumoto (O'Reilly) or *Programming Ruby*, Third Edition, by Dave Thomas (Pragmatic Bookshelf) should do the trick.

It is also important to note that this is a Ruby 1.9 book. It makes no attempt to provide notes on the differences between Ruby 1.8 and 1.9 except for in a brief appendix designed specifically for that purpose. Although many of the code samples will likely work with little or no modifications for earlier versions of Ruby, Ruby 1.9 is the way forward, and I have chosen to focus on it exclusively in this book. Although the book may still be useful to those maintaining legacy code, it is admittedly geared more toward the forward-looking crowd.

About This Book

This book is designed to be read by chapter, but the chapters are not in any particular order. The book is split into two parts, with eight chapters forming its core and three appendixes included as supplementary material. Despite the fact that you can read these topics in any order that you'd like, it is recommended that you read the entire book. Lots of the topics play off of each other, and reading through them all will give you a solid base in some powerful Ruby techniques and practices.

Each of the core chapters starts off with a case study that is meant to serve as an introduction to the topic it covers. Every case study is based on code from real Ruby projects, and is meant to provide a practical experience in code reading and exploration. The best way to work through these examples is to imagine that you are working through a foreign codebase with a fellow developer, discussing the interesting bits as you come across them. In this way, you'll be able to highlight the exciting parts without getting bogged down on every last detail. You are not expected to understand every line of code in the case studies in this book, but instead should just treat them as useful exercises that prepare you for studying the underlying topics.

Once you've worked your way through the case study, the remainder of each core chapter fills in details on specific subtopics related to the overall theme. These tend to

mix real code in with some abstract examples, preferring the former but falling back to the latter when necessary to keep things easy to understand. Some code samples will be easy to run as they are listed; others might only be used for illustration purposes. This should be easy enough to figure out as you go along based on the context. I wholeheartedly recommend running examples when they're relevant and stopping frequently to conduct your own explorations as you read this book. The sections are kept somewhat independent of one another to make it easy for you to take as many breaks as you need, and each wraps up with some basic reminders to refresh your memory of what you just read.

Although the core chapters are the essential part of this book, the appendixes should not be overlooked. You'll notice that they're slightly different in form and content from the main discussion, but maintain the overall feel of the book. You'll get the most out of them if you read them after you've completed the main part of the book, as they tend to assume that you're already familiar with the rest of the content.

That's pretty much all there is to it. The key things to remember are that you aren't going to get much out of this book by skimming for content on a first read, and that you should keep your brain engaged while you work your way through the content. If you read this entire book without writing any code in the process, you'll probably rob yourself of the full experience. So pop open your favorite editor, start with the topic from the chapter listing that interests you most, and get hacking!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example,

writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Ruby Best Practices* by Gregory Brown. Copyright 2009 Gregory Brown, 978-0-596-52300-8."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at <http://my.safaribooksonline.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

O'Reilly has a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

<http://oreilly.com/catalog/9780596523008/>

Gregory maintains a community-based page for this book at:

<http://rubybestpractices.com>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

<http://www.oreilly.com>

Acknowledgments

Over the course of writing *Ruby Best Practices*, I was thoroughly supported by my friends, family, and fellow hackers. I want to thank each and every one of the folks who've helped out with this book, because it would not exist without them.

This book did not have a typical technical review process, but instead was supported by an excellent advisory board whose members participated in group discussion and the occasional review as each chapter was released. These folks not only helped catch technical errors, but helped me sketch out the overall vision for how the book should come together as well. Participants included James Britt, Francis Hwang, Hart Larew, Chris Lee, Jeremy McAnally, and Aaron Patterson.

Rounding out the group was the best pair of guiding mentors I could hope for, Brad Ediger and James Edward Gray II. Both have published Ruby books, and have worked with me extensively on a number of Ruby projects. James and Brad were both instrumental in producing this book, and to my career as a software developer in general. I have learned a ton from each of them, and thanks to their help with *RBP*, I can now pass their knowledge on to you.

Much of the source code in this book comes from the open source Ruby community. Although I talk about my own projects (Prawn and Ruport) a lot, most of the code I show is actually from other contributors or at least originated from good ideas that came up in mailing list discussions, feature requests, and so on. In addition to these two projects, I also have benefited from studying a whole slew of other gems, including but not limited to: *activesupport*, *builder*, *camping*, *faker*, *flexmock*, *gibberish*, *haml*, *highline*, *lazy*, *nokogiri*, *pdf-writer*, and *rspec*. Great thanks go out to all of the developers of these projects, whom I've tried to acknowledge directly wherever I can throughout the text.

Of course, without Yukihiro Matsumoto (Matz), we wouldn't have Ruby in the first place. After writing this book, I am more impressed than ever by the language he has designed. If I'm lucky, this book will help show people just how beautiful Ruby can be.

Producing the technical content for this work was daunting, but only part of the overall picture. My editor, Mike Loukides, and the entire O'Reilly production team have made publishing this book a very comfortable experience. After overcoming major fears about the hurdles of working with a mainstream publisher, I've found the folks at O'Reilly to be helpful, accommodating, and supportive. It is especially nice that this book will become an open community resource less than a year after it prints. This

measure is one I hope to see other technical book publishers adopt, and one I'm very thankful that O'Reilly was open to.

Finally, I need to thank the folks who've helped me keep my sanity while working on this huge project. My future wife, Jia Wu, has been amazingly supportive of me, and helped make sure that I occasionally ate and slept while working on this book. On the weekends, we'd usually escape for an bit and spend time with my close friends and family. Though they didn't have anything to do with the project itself, without Pete, Paul, Mom, Dad, and Vinny, I doubt you'd be reading this book right now. Thanks to all of you, even if you'll never need to read this book.

So many people helped out in countless different ways, that I'm sure I've missed someone important while compiling this list. To make sure these folks get their well-deserved credit, please keep an eye on the acknowledgments page at <http://rubybestpractices.com> and let me know if there is someone who needs to be added to the list. But for now, if I've failed to list you here, thank you and please know that I've not forgotten what you've done to help me.

Table of Contents

Foreword	ix
Preface	xi
1. Driving Code Through Tests	1
A Quick Note on Testing Frameworks	2
Designing for Testability	2
Testing Fundamentals	10
Well-Focused Examples	10
Testing Exceptions	11
Run the Whole Suite at Once	13
Advanced Testing Techniques	14
Using Mocks and Stubs	14
Testing Complex Output	22
Keeping Things Organized	26
Embedding Tests in Library Files	27
Test Helpers	27
Custom Assertions	29
Conclusions	30
2. Designing Beautiful APIs	31
Designing for Convenience: Ruptor's Table() feature	31
Ruby's Secret Power: Flexible Argument Processing	35
Standard Ordinal Arguments	36
Ordinal Arguments with Optional Parameters	36
Pseudo-Keyword Arguments	37
Treating Arguments As an Array	38
Ruby's Other Secret Power: Code Blocks	40
Working with Enumerable	41
Using Blocks to Abstract Pre- and Postprocessing	43
Blocks As Dynamic Callbacks	45
Blocks for Interface Simplification	47

Avoiding Surprises	48
Use attr_reader, attr_writer, and attr_accessor	48
Understand What method? and method! Mean	50
Make Use of Custom Operators	53
Conclusions	55
3. Mastering the Dynamic Toolkit	57
BlankSlate: A BasicObject on Steroids	57
Building Flexible Interfaces	62
Making instance_eval() Optional	63
Handling Messages with method_missing() and send()	65
Dual-Purpose Accessors	69
Implementing Per-Object Behavior	70
Extending and Modifying Preexisting Code	74
Adding New Functionality	75
Modification via Aliasing	79
Per-Object Modification	81
Building Classes and Modules Programmatically	84
Registering Hooks and Callbacks	88
Detecting Newly Added Functionality	89
Tracking Inheritance	91
Tracking Mixins	93
Conclusions	96
4. Text Processing and File Management	99
Line-Based File Processing with State Tracking	99
Regular Expressions	103
Don't Work Too Hard	105
Anchors Are Your Friends	105
Use Caution When Working with Quantifiers	106
Working with Files	109
Using Pathname and FileUtils	109
The tempfile Standard Library	112
Automatic Temporary Directory Handling	113
Collision Avoidance	113
Same Old I/O Operations	114
Automatic Unlinking	114
Text-Processing Strategies	115
Advanced Line Processing	116
Atomic Saves	118
Conclusions	120

5. Functional Programming Techniques	121
Laziness Can Be a Virtue (A Look at lazy.rb)	121
Minimizing Mutable State and Reducing Side Effects	129
Modular Code Organization	133
Memoization	138
Infinite Lists	145
Higher-Order Procedures	149
Conclusions	152
6. When Things Go Wrong	153
A Process for Debugging Ruby Code	153
Capturing the Essence of a Defect	157
Scrutinizing Your Code	160
Utilizing Reflection	160
Improving inspect Output	162
Finding Needles in a Haystack	166
Working with Logger	168
Conclusions	176
7. Reducing Cultural Barriers	177
m17n by Example: A Look at Ruby's CSV Standard Library	178
Portable m17n Through UTF-8 Transcoding	182
Source Encodings	183
Working with Files	183
Transcoding User Input in an Organized Fashion	185
m17n in Standalone Scripts	188
Inferring Encodings from Locale	189
Customizing Encoding Defaults	191
m17n-Safe Low-Level Text Processing	193
Localizing Your Code	195
Conclusions	204
8. Skillful Project Maintenance	205
Exploring a Well-Organized Ruby Project (Haml)	205
Conventions to Know About	210
What Goes in a README	211
Laying Out Your Library	213
Executables	216
Tests	216
Examples	217
API Documentation via RDoc	219
Basic Documentation Techniques and Guidelines	220
Controlling Output with RDoc Directives	222

The RubyGems Package Manager	227
Writing a Gem::Specification	228
Working with Dependencies	231
Rake: Ruby's Built-in Build Utility	234
Conclusions	237
A. Writing Backward-Compatible Code	239
B. Leveraging Ruby's Standard Library	251
C. Ruby Worst Practices	283
Index	299

Driving Code Through Tests

If you've done some Ruby—even a little bit—you have probably heard of *test-driven development* (TDD). Many advocates present this software practice as the “secret key” to programming success. However, it's still a lot of work to convince people that writing tests that are often longer than their implementation code can actually lower the total time spent on a project and increase overall efficiency.

In my work, I've found most of the claims about the benefits of TDD to be true. My code is better because I write tests that document the expected behaviors of my software while verifying that my code is meeting its requirements. By writing automated tests, I can be sure that once I narrow down the source of a bug and fix it, it'll never resurface without me knowing right away. Because my tests are automated, I can hand my code off to others and mechanically assert my expectations, which does more for me than a handwritten specification ever could do.

However, the important thing to take home from this is that automated testing is really no different than what we did before we discovered it. If you've ever tried to narrow down a bug with a print statement based on a conditional, you've already written a primitive form of automated testing:

```
if foo != "blah"
  puts "I expected 'blah' but foo contains #{foo}"
end
```

If you've ever written an example to verify that a bug exists in an earlier version of code, but not in a later one, you've written something not at all far from the sorts of things you'll write through TDD. The only difference is that one-off examples do not adequately account for the problems that can arise during integration with other modules. This problem can become huge, and is one that unit testing frameworks handle quite well.

Even if you already know a bit about testing and have been using it in your work, you might still feel like it doesn't come naturally. You write tests because you see the long-term benefits, but you usually write your code first. It takes you a while to write your tests, because it seems like the code you wrote is difficult to pin down behavior-wise.

In the end, testing becomes a necessary evil. You appreciate the safety net, but except for when you fall, you'd rather just focus on keeping your balance and moving forward.

Masterful Rubyists will tell you otherwise, and for good reason. Testing may be hard, but it truly does make your job of writing software easier. This chapter will show you how to integrate automated testing into your workflow, without forcing you to relearn the troubleshooting skills you've already acquired. By making use of the best practices discussed here, you'll be able to more easily see the merits of TDD in your own work.

A Quick Note on Testing Frameworks

Ruby provides a unit testing framework in its standard library called *minitest/unit*. This library provides a user-level compatibility layer with the popular *test/unit* library, which has been fairly standard in the Ruby community for some time now. There are significant differences between the *minitest/unit* and *test/unit* implementations, but as we won't be building low-level extensions in this chapter, you can assume that the code here will work in both *minitest/unit* and *test/unit* without modification.

For what it's worth, I don't have a very strong preference when it comes to testing frameworks. I am using the `Test::Unit` API here because it is part of standard Ruby, and because it is fundamentally easy to hack on and extend. Many of the existing alternative testing frameworks are built on top of `Test::Unit`, and you will almost certainly need to have a working knowledge of it as a Ruby developer. However, if you've been working with a noncompatible framework such as RSpec (<http://rspec.info>), there's nothing wrong with that. The ideas here should be mostly portable to your framework of choice.

And now we can move on. Before digging into the nuts and bolts of writing tests, we'll examine what it means for code to be easily testable, by looking at some real examples.

Designing for Testability

Describing testing with the phrase “Red, Green, Refactor” makes it seem fairly straightforward. Most people interpret this as the process of writing some failing tests, getting those tests to pass, and then cleaning up the code without causing the tests to fail again. This general assumption is exactly correct, but a common misconception is how much work needs to be done between each phase of this cycle.

For example, if we try to solve our whole problem all in one big chunk, add tests to verify that it works, then clean up our code, we end up with implementations that are very difficult to test, and even more challenging to refactor. The following example illustrates just how bad this problem can get if you're not careful. It's from some payroll management code I wrote in a hurry a couple of years ago:

```

def time_data_for_week(week_data,start,employee_id)

  data = Hash.new { |h,k| h[k] = Hash.new }

  %w[M T W TH F S].zip((0..6).to_a).each do |day,offset|

    date = (start + offset.days).beginning_of_day

    data[day][:lunch_hours] = LunchTime.find(:all, conditions:
      ["employee_id = ? and day between ? and ?",
        employee_id, date, date + 1.day - 1.second] ).inject(0) { |s,r|
        s + r.duration
      }

    times = [[:sick_hours, "Sick"],
      [:personal_hours, "Personal"],
      [:vacation_hours, "Vacation"],
      [:other_hours, "Other"] ]

    times.each do |a,b|
      data[day][a] = OtherTime.find(:all, conditions:
        ["employee_id = ? and category = '#{b}' and date between ? and ?",
          employee_id, date, date + 1.day - 1.second] ).inject(0) { |s,r|
          s + r.hours
        }
    end

    d = week_data.find { |d,_| d == date }
    next unless d

    d = d[-1]
    data[day].merge!(
      regular_hours: d.inject(0) { |s,e|
        s + (e.end_time ? (e.end_time - e.start_time) / 3600 : 0)
      } - data[day][:lunch_hours],
      start_time: d.map { |e| e.start_time }.sort[0],
      end_time: d.map { |e| e.end_time }.compact.sort[-1]
    )
  end

  sums = Hash.new(0)

  data.each do |k,v|
    [:regular_hours, :lunch_hours, :sick_hours,
      :personal_hours, :vacation_hours, :other_hours].each { |h|
      sums[h] += v[h].to_f }
  end

  Table(:day,:start_time,:end_time,:regular_hours,:lunch_hours,
    :sick_hours,:personal_hours,:vacation_hours, :other_hours) do |t|
    %w[M T W TH F S].each { |d| t << {day: d}.merge(data[d]) }
    t << []
    t << { day: "<b>Totals</b>" }.merge(sums)
  end
end

```