

<<真实世界的Haskell>>

图书基本信息

书名：<<真实世界的Haskell>>

13位ISBN编号：9787564119256

10位ISBN编号：756411925X

出版时间：2010-1

出版时间：东南大学出版社

作者：[美] 沙利文,[美] 戈尔,[美] 斯图尔特

页数：670

版权说明：本站所提供下载的PDF图书仅提供预览和简介，请支持正版图书。

更多资源请访问：<http://www.tushu007.com>

## 前言

Have We Got a Deal for You! Haskell is a deep language; we think learning it is a hugely rewarding experience. We will focus on three elements as we explain why. The first is novelty : we invite you to think about programming from a different and valuable perspective. The second is power : we'll show you how to create software that is short , fast , and safe. Lastly , we offer you a lot of enjoyment : the pleasure of applying beautiful programming techniques to solve real problems.

**Novelty** Haskell is most likely quite different from any language you've ever used before. Compared to the usual set of concepts in a programmer's mental toolbox , functional programming offers us a profoundly different way to think about software. In Haskell , we deemphasize code that modifies data. Instead , we focus on functions that take immutable values as input and produce new values as output. Given the same inputs , these functions always return the same results. This is a core idea behind functional programming.

Along with not modifying data , our Haskell functions usually don't talk to the external world; we call these functions pure. We make a strong distinction between pure code and the parts of our programs that read or write files , communicate over network connections , or make robot arms move. This makes it easier to organize , reason about , and test our programs.

We abandon some ideas that might seem fundamental , such as having a for loop built into the language. We have other , more flexible , ways to perform repetitive tasks. Even the way in which we evaluate expressions is different in Haskell. We defer every computation until its result is actually needed——Haskell is a lazy language. Laziness is not merely a matter of moving work around , it profoundly affects how we write programs.

**Power** Throughout this book , we will show you how Haskell's alternatives to the features of traditional languages are powerful and flexible and lead to reliable code. Haskell is positively crammed full of cutting-edge ideas about how to create great software.

Since pure code has no dealings with the outside world , and the data it works with is never modified , the kind of nasty surprise in which one piece of code invisibly corrupts data used by another is very rare. Whatever context we use a pure function in , the function will behave consistently.

## <<真实世界的Haskell>>

### 内容概要

Haskell is most likely quite different from any language youve ever used before. Compared to the usual set of concepts in a programmers mental toolbox , functional programming offers us a profoundly different way to think about software. In Haskell , we deemphasize code that modifies data. Instead , we focus on functions that take immutable values as input and produce new values as output. Given the same inputs , these functions always return the same results. This is a core idea behind functional programming.

## 书籍目录

Preface  
1. Getting Started  
Your Haskell Environment  
Getting Started with ghci , the Interpreter  
Basic Interaction : Using ghci as a Calculator  
Simple Arithmetic  
An Arithmetic Quirk : Writing Negative Numbers  
Boolean Logic , Operators , and Value Comparisons  
Operator Precedence and Associativity  
Undefined Values , and Introducing Variables  
Dealing with Precedence and Associativity Rules  
Command-Line Editing in ghci  
Lists  
Operators on Lists  
Strings and Characters  
First Steps with Types  
A Simple Program  
2. Types and Functions  
Why Care About Types?  
Haskell ' s Type System  
Strong Types  
Static Types  
Type Inference  
What to Expect from the Type System  
Some Common Basic Types  
Function Application  
Useful Composite Data Types : Lists and Tuples  
Functions over Lists and Tuples  
Passing an Expression to a Function  
Function Types and Purity  
Haskell Source Files , and Writing Simple Functions  
Just What Is a Variable , Anyway?  
Conditional Evaluation  
Understanding Evaluation by Example  
Lazy Evaluation  
A More Involved Example  
Recursion  
Ending the Recursion  
Returning from the Recursion  
What Have We Learned?  
Polymorphism in Haskell  
Reasoning About Polymorphic Functions  
Further Reading  
The Type of a Function of More Than One Argument  
Why the Fuss over Purity?  
Conclusion  
3. Defining Types , Streamlining Functions  
Defining a New Data Type  
Naming Types and Values  
Type Synonyms  
Algebraic Data Types  
Tuples , Algebraic Data Types , and When to Use Each  
Analogues to Algebraic Data Types in Other Languages  
Pattern Matching  
Construction and Deconstruction  
Further Adventures  
Variable Naming in Patterns  
The Wild Card Pattern  
Exhaustive Patterns and Wild Cards  
Record Syntax  
Parameterized Types  
Recursive Types  
Reporting Errors  
A More Controlled Approach  
Introducing Local Variables  
Shadowing  
The where Clause  
Local Functions , Global Variables  
The Offside Rule and Whitespace in an Expression  
A Note About Tabs Versus Spaces  
The Offside Rule Is Not Mandatory  
The case Expression  
Common Beginner Mistakes with Patterns  
Incorrectly Matching Against a Variable  
Incorrectly Trying to Compare for Equality  
Conditional Evaluation with Guards  
4. Functional Programming  
Thinking in Haskell  
A Simple Command-Line Framework  
Warming Up : Portably Splitting Lines of Text  
A Line-Ending Conversion Program  
Infix Functions  
Working with Lists  
Basic List Manipulation  
Safely and Sanely Working with Crashy Functions  
Partial and Total Functions  
More Simple List Manipulations  
Working with Sublists  
Searching Lists  
Working with Several Lists at Once  
Special String-Handling Functions  
How to Think About Loops  
Explicit Recursion  
Transforming Every Piece of Input  
Mapping over a List  
Selecting Pieces of Input  
Computing One Answer over a Collection  
The Left Fold  
Why Use Folds , Maps , and Filters?  
Folding from the Right  
Left Folds , Laziness , and Space Leaks  
Further Reading  
Anonymous (lambda) Functions  
Partial Function Application and Currying  
Sections  
As-patterns  
Code Reuse Through Composition  
Use Your Head Wisely  
Tips for Writing Readable Code  
Space Leaks and Strict Evaluation  
Avoiding Space Leaks with seq  
Learning to Use seq  
5. Writing a Library : Working with JSON Data  
A Whirlwind Tour of JSON  
Representing JSON Data in Haskell  
The Anatomy of a Haskell Module  
Compiling Haskell Source  
Generating a Haskell Program and Importing Modules  
Printing JSON Data  
Type Inference Is a Double-Edged Sword  
A More General Look at Rendering  
Developing Haskell Code Without Going Nuts  
Pretty Printing a String  
Arrays and Objects , and the Module Header  
Writing a Module Header  
Fleshing Out the Pretty-Printing Library  
Compact Rendering  
True Pretty Printing  
Following the Pretty Printer  
Creating a Package  
Writing a Package Description  
GHC ' s Package Manager  
Setting Up , Building , and Installing  
Practical Pointers and Further Reading  
6. Using Typeclasses  
The Need for Typeclasses  
What Are Typeclasses?  
Declaring Typeclass Instances  
Important Built-in Typeclasses  
Show/Read/Serialization with read and show  
Numeric Types  
Equality , Ordering , and Comparisons  
Automatic Derivation  
Typeclasses at Work : Making JSON Easier to Use  
More Helpful Errors  
Making an Instance with a Type Synonym  
Living in an Open World  
When Do Overlapping Instances Cause Problems?  
Relaxing Some Restrictions on Typeclasses  
How Does Show Work for Strings?  
How to Give a Type a New Identity  
Differences Between Data and Newtype  
Declarations  
Summary : The Three Ways of Naming Types  
JSON Typeclasses Without Overlapping Instances  
The Dreaded Monomorphism Restriction  
Conclusion  
7. I/O  
Classic I/O in Haskell  
Pure Versus I/O  
Why Purity Matters  
Working with Files and Handles  
More on openFile  
Closing Handles  
Seek and Tell  
Standard Input , Output

, and ErrorDeleting and Renaming FilesTemporary FilesExtended Example : Functional I/O and Temporary FilesLazy I/OgetFileContentsreadFile and writeFileA Word on Lazy OutputinteractThe IO MonadActionsSequencingThe True Nature of ReturnIs Haskell Really Imperative?Side Effects with Lazy I/OBufferingBuffering ModesFlushing The BufferReading Command-Line ArgumentsEnvironment Variables8. Efficient File Processing , Regular Expressions , and Filename MatchingEfficient File ProcessingBinary I/O and Qualified ImportsText I/OFilename MatchingRegular Expressions in HaskellThe Many Types of ResultMore About Regular ExpressionsMixing and Matching String TypesOther Things You Should KnowTranslating a glob Pattern into a Regular ExpressionAn important Aside : Writing Lazy FunctionsMaking Use of Our Pattern MatcherHandling Errors Through API DesignPutting Our Code to Work9. I/O Case Study : A Library for Searching the FilesystemThe find CommandStarting Simple : Recursively Listing a DirectoryRevisiting Anonymous and Named FunctionsWhy Provide Both mapM and forM?A Naive Finding FunctionPredicates : From Poverty to Riches , While Remaining PureSizing a File SafelyThe Acquire-Use-Release CycleA Domain-Specific Language for PredicatesAvoiding Boilerplate with LiftingGluing Predicates TogetherDefining and Using New OperatorsControlling TraversalDensity , Readability , and the Learning ProcessAnother Way of Looking at TraversalUseful Coding GuidelinesCommon Layout Styles10. Code Case Study : Parsing a Binary Data FormatGrayscale FilesParsing a Raw PGM FileGetting Rid of Boilerplate CodeImplicit StateThe Identity ParserRecord Syntax , Updates , and Pattern MatchingA More Interesting ParserObtaining and Modifying the Parse StateReporting Parse ErrorsChaining Parsers TogetherIntroducing FunctorsConstraints on Type Definitions Are BadInfix Use of fmapFlexible InstancesThinking More About FunctorsWriting a Functor Instance for ParseUsing Functors for ParsingRewriting Our PGM ParserFuture Directions11. Testing and Quality AssuranceQuickCheck : Type-Based TestingTesting for PropertiesTesting Against a ModelTesting Case Study : Specifying a Pretty PrinterGenerating Test DataTesting Document ConstructionUsing Lists as a ModelPutting It All TogetherMeasuring Test Coverage with HPC12. Barcode RecognitionA Little Bit About BarcodesEAN-13 EncodingIntroducing ArraysArrays and LazinessFolding over ArraysModifying Array ElementsEncoding an EAN-13 BarcodeConstraints on Our DecoderDivide and ConquerTurning a Color Image into Something TractableParsing a Color ImageGrayscale ConversionGrayscale to Binary and Type SafetyWhat Have We Done to Our Image?Finding Matching DigitsRun Length EncodingScaling Run Lengths , and Finding Approximate MatchesList ComprehensionsRemembering a Match ' s ParityChunking a ListGenerating a List of Candidate DigitsLife Without Arrays or Hash TablesA Forest of SolutionsA Brief Introduction to MapsFurther ReadingTurning Digit Soup into an AnswerSolving for Check Digits in ParallelCompleting the Solution Map with the First DigitFinding the Correct SequenceWorking with Row DataPulling It All TogetherA Few Comments on Development Style13. Data StructuresAssociation ListsMapsFunctions Are Data , TooExtended Example : /etc/passwdExtended Example : Numeric TypesFirst StepsCompleted CodeTaking Advantage of Functions as DataTurning Difference Lists into a Proper LibraryLists , Difference Lists , and MonoidsGeneral-Purpose Sequences14. MonadsRevisiting Earlier Code ExamplesMaybe ChainingImplicit StateLooking for Shared PatternsThe Monad TypeclassAnd Now , a Jargon MomentUsing a New Monad : Show Your Work!Information HidingControlled EscapeLeaving a TraceUsing the Logger MonadMixing Pure and Monadic CodePutting a Few Misconceptions to RestBuilding the Logger MonadSequential Logging , Not Sequential EvaluationThe Writer MonadThe Maybe MonadExecuting the Maybe MonadMaybe at Work , and Good API DesignThe List MonadUnderstanding the List MonadPutting the List Monad to WorkDesugaring of do BlocksMonads as a Programmable SemicolonWhy Go Sugar-Free?The State MonadAlmost a State MonadReading and Modifying the StateWill the Real State Monad Please Stand Up?Using the State Monad : Generating Random ValuesA First Attempt at PurityRandom Values in the State MonadRunning the State MonadWhat About a Bit More State?Monads and FunctorsAnother Way of Looking at MonadsThe Monad Laws and Good Coding Style15. Programming with MonadsGolfing Practice : Association ListsGeneralized LiftingLooking for AlternativesThe Name mplus Does Not Imply AdditionRules for Working with MonadPlusFailing Safely with MonadPlusAdventures in Hiding the PlumbingSupplying Random NumbersAnother Round of GolfSeparating

Interface from Implementation Multiparameter Typeclasses Functional Dependencies Rounding Out Our Module Programming to a Monad 's Interface The Reader Monad A Return to Automated Deriving Hiding the IO Monad Using a newtype Designing for Unexpected Uses Using Typeclasses Isolation and Testing The Writer Monad and Lists Arbitrary I/O Revisited 16. Using Parsec First Steps with Parsec : Simple CSV Parsing The sepBy and endBy Combinators Choices and Errors Lookahead Error Handling Extended Example : Full CSV Parser Parsec and MonadPlus Parsing a URL-Encoded Query String Supplanting Regular Expressions for Casual Parsing Parsing Without Variables Applicative Functors for Parsing Applicative Parsing by Example Parsing JSON Data Parsing a HTTP Request Backtracking and Its Discontents Parsing Headers 17. Interfacing with C : The FFI Foreign Language Bindings : The Basics Be Careful of Side Effects A High-Level Wrapper Regular Expressions for Haskell : A Binding for PCRE Simple Tasks : Using the C Preprocessor Binding Haskell to C with hsc2hs Adding Type Safety to PCRE Binding to Constants Automating the Binding Passing String Data Between Haskell and C Typed Pointers Memory Management : Let the Garbage Collector Do the Work A High-Level Interface : Marshaling Data Marshaling ByteStrings Allocating Local C Data : The Storable Class Putting It All Together Matching on Strings Extracting Information About the Pattern Pattern Matching with Substrings The Real Deal : Compiling and Matching Regular Expressions 18. Monad Transformers Motivation : Boilerplate Avoidance A Simple Monad Transformer Example Common Patterns in Monads and Monad Transformers Stacking Multiple Monad Transformers Hiding Our Work Moving Down the Stack When Explicit Lifting Is Necessary Understanding Monad Transformers by Building One Creating a Monad Transformer More Typeclass Instances Replacing the Parse Type with a Monad Stack Transformer Stacking Order Is Important Putting Monads and Monad Transformers into Perspective Interference with Pure Code Overdetermined Ordering Runtime Overhead Unwieldy Interfaces Pulling It All Together 19. Error Handling Error Handling with Data Types Use of Maybe Use of Either Exceptions First Steps with Exceptions Laziness and Exception Handling Using handle Selective Handling of Exceptions I/O Exceptions Throwing Exceptions Dynamic Exceptions Error Handling in Monads A Tiny Parsing Framework 20. Systems Programming in Haskell Running External Programs Directory and File Information Program Termination Dates and Times Clock Time and Calendar Time File Modification Times Extended Example : Piping Using Pipes for Redirection Better Piping Final Words on Pipes 21. Using Databases Overview of JDBC Installing JDBC and Drivers Connecting to Databases Transactions Simple Queries SqlValueQuery Parameters Prepared Statements Reading Results Reading with Statements Lazy Reading Database Metadata Error Handling 22. Extended Example : Web Client Programming Basic Types The Database The Parser Downloading Main Program 23. GUI Programming with gtk2hs Installing gtk2hs Overview of the GTK+ Stack User Interface Design with Glade Glade Concepts Event-Driven Programming Initializing the GUI The Add Podcast Window Long-Running Tasks Using Cabal 24. Concurrent and Multicore Programming Defining Concurrency and Parallelism Concurrent Programming with Threads Threads Are Nondeterministic Hiding Latency Simple Communication Between Threads The Main Thread and Waiting for Other Threads Safely Modifying an MVar Safe Resource Management : A Good Idea , and Easy Besides Finding the Status of a Thread Writing Tighter Code Communicating over Channels Useful Things to Know About MVar and Chan Are Nonstrict Chan Is Unbounded Shared-State Concurrency Is Still Hard Deadlock Starvation Is There Any Hope? Using Multiple Cores with GHC Runtime Options Finding the Number of Available Cores from Haskell Choosing the Right Runtime Parallel Programming in Haskell Normal Form and Head Normal Form Sequential Sorting Transforming Our Code into Parallel Code Knowing What to Evaluate in Parallel What Promises Does par Make? Running Our Code and Measuring Performance Tuning for Performance Parallel Strategies and MapReduce Separating Algorithm from Evaluation Separating Algorithm from Strategy Writing a Simple MapReduce Definition MapReduce and Strategies Sizing Work Appropriately Efficiently Finding Line-Aligned Chunks Counting Lines Finding the Most Popular URLs Conclusions 25. Profiling and Optimization Profiling Haskell Programs Collecting Runtime Statistics Time Profiling Space Profiling Controlling Evaluation Strictness and Tail Recursion Adding Strictness Understanding Core Advanced Techniques : Fusion Tuning the Generated Assembly Conclusions 26. Advanced Library Design : Building a Bloom

<<真实世界的Haskell>>

FilterIntroducing the Bloom FilterUse Cases and Package LayoutBasic DesignUnboxing , Lifting , and BottomThe ST MonadDesigning an API for Qualified ImportCreating a Mutable Bloom FilterThe Immutable APICreating a Friendly InterfaceRe-Exporting Names for ConvenienceHashing ValuesTurning Two Hashes into ManyImplementing the Easy Creation FunctionCreating a Cabal PackageDealing with Different Build SetupsCompilation Options and Interfacing to CTesting with QuickCheckPolymorphic TestingWriting Arbitrary Instances for ByteStringsAre Suggested Sizes Correct?Performance Analysis and TuningProfile-Driven Performance Tuning27. Sockets and SyslogBasic NetworkingCommunicating with UDPUDP Client Example : syslogUDP Syslog ServerCommunicating with TCPHandling Multiple TCP StreamsTCP Syslog ServerTCP Syslog Client28. Software Transactional MemoryThe BasicsSome Simple ExamplesSTM and SafetyRetrying a TransactionWhat Happens When We Retry?Choosing Between AlternativesUsing Higher Order Code with TransactionsI/O and STMCommunication Between ThreadsA Concurrent Web Link CheckerChecking a LinkWorker ThreadsFinding LinksCommand-Line ParsingPattern GuardsPractical Aspects of STMGetting Comfortable with Giving Up ControlUsing InvariantsA. Installing GHC and Haskell LibrariesB. Characters , Strings , and Escaping RulesIndex

## 章节摘录

In this section, we've discussed how Haskell, unlike most languages, draws a clear distinction between pure code and I/O actions. In languages such as C or Java, there is no such thing as a function that is guaranteed by the compiler to always return the same result for the same arguments or a function that is guaranteed to never have side effects. The only way to know if a given function has side effects is to read its documentation and hope that it's accurate. Many bugs in programs are caused by unanticipated side effects. Still more are caused by misunderstanding circumstances in which functions may return different results for the same input. As multithreading and other forms of parallelism grow increasingly common, it becomes more difficult to manage global side effects. Haskell's method of isolating side effects into I/O actions provides a clear boundary. You can always know which parts of the system may alter state and which won't. You can always be sure that the pure parts of your program aren't having unanticipated results. This helps you to think about the program. It also helps the compiler to think about it. Recent versions of `ghc`, for instance, can provide a level of automatic parallelism for the pure parts of your code—something of a holy grail for computing. For more discussion on this topic, refer to "Side Effects with Lazy I/O" on page 188.



## <<真实世界的Haskell>>

### 媒体关注与评论

“现代软件的最大问题在于性能、模块化、可靠性和并发性。在《真实世界的Haskell》中，作者很好地讲授了如何使用Haskell这一超前于当今主流的语言，来逐一化解这些问题。

” ——Trim Sweeney, Epic Games创始人，同时也是Unreal 游戏引擎设计者 “这是第一本涵盖了现实世界程序员所需一切技术的书籍。

当读罢此书，你将能够用当前所钟爱的语言写出更优秀的代码。

” ——Simon Peyton Jones. Microsoft Research Haskell语言架构师，GlasgowHaskell 编译器设计者

## <<真实世界的Haskell>>

### 编辑推荐

《真实世界的Haskell(影印版)》是一本上手快且易于使用的指导书，它向你介绍这门日趋流行的编程语言。

你将学习如何将Haskell应用于不同实践当中，从简短的脚本到要求苛刻的大型应用。

《真实世界的Haskell(影印版)》向你讲解了函数式编程的基础，帮助你加深对如何在现实世界中应用Haskell的理解，例如输入/输出性能、数据处理、并发等等。

《真实世界的Haskell》能帮助你：  
· 理解过程式与函数式编程之间的差异  
· 学习Haskell的特性，以及如何使用它来开发有用的程序  
· 与文件系统、数据库和网络服务交互  
· 编写可以进行自动测试、代码覆盖和错误处理的代码  
· 通过并发和并行编程发挥多核系统的威力  
在《真实世界的Haskell(影印版)》中你将发现大量的实用习题和真实的Haskell程序示例，你可以修改、编译及运行它们。

无论是否曾经使用过函数式语言，如果想要了解Haskell为何成为众多组织所选用的实用语言，《真实世界的Haskell》是你的首选。

<<真实世界的Haskell>>

版权说明

本站所提供下载的PDF图书仅提供预览和简介，请支持正版图书。

更多资源请访问:<http://www.tushu007.com>