# [01: Static Checking](http://web.mit.edu/6.005/www/fa16/classes/01-static-checking/)

## Mutating Values vs. Reassigning Variables

It’s good practice to use final for declaring the parameters of a method and as many local variables as possible. Like the type of the variable, these declarations are important documentation, useful to the reader of the code and statically checked by the compiler.

There are two variables in our hailstoneSequence method: can we declare them final, or not?

public static List<Integer> hailstoneSequence(final int n) {

final List<Integer> list = new ArrayList<Integer>();

## Documenting Assumptions

Programs have to be written with two goals in mind:

communicating with the computer. First persuading the compiler that your program is sensible – syntactically correct and type-correct. Then getting the logic right so that it gives the right results at runtime.

communicating with other people. Making the program easy to understand, so that when somebody has to fix it, improve it, or adapt it in the future, they can do so.

## Hacking vs. Engineering

We’ve written some hacky code in this class. Hacking is often marked by unbridled optimism:

Bad: writing lots of code before testing any of it

Bad: keeping all the details in your head, assuming you’ll remember them forever, instead of writing them down in your code

Bad: assuming that bugs will be nonexistent or else easy to find and fix

But software engineering is not hacking. Engineers are pessimists:

Good: write a little bit at a time, testing as you go. In a future class, we’ll talk about test-first programming.

Good: document the assumptions that your code depends on

Good: defend your code against stupidity – especially your own! Static checking helps with that.

## The Goal of 6.005

Our primary goal in this course is learning how to produce software that is:

**Safe from bugs**. Correctness (correct behavior right now), and defensiveness (correct behavior in the future).

**Easy to understand**. Has to communicate to future programmers who need to understand it and make changes in it (fixing bugs or adding new features). That future programmer might be you, months or years from now. You’ll be surprised how much you forget if you don’t write it down, and how much it helps your own future self to have a good design.

**Ready for change**. Software always changes. Some designs make it easy to make changes; others require throwing away and rewriting a lot of code.

## Summary

The main idea we introduced today is **static checking**. Here’s how this idea relates to the goals of the course:

**Safe from bugs.** Static checking helps with safety by catching type errors and other bugs before runtime.

**Easy to understand.** It helps with understanding, because types are explicitly stated in the code.

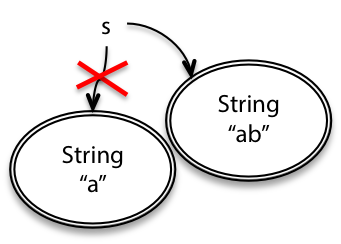
**Ready for change.** Static checking makes it easier to change your code by identifying other places that need to change in tandem. For example, when you change the name or type of a variable, the compiler immediately displays errors at all the places where that variable is used, reminding you to update them as well.

# Reading 2: Basic Java

## Snapshot diagrams

### Mutating values vs. reassigning variables

#### Reassignment and immutable values



For example, if we have a [String](http://docs.oracle.com/javase/8/docs/api/?java/lang/String.html) variable s, we can reassign it from a value of "a" to "ab".

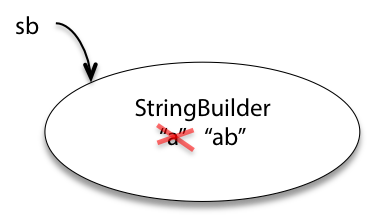
String s = "a";

s = s + "b";

String is an example of an immutable type, a type whose values can never change once they have been created. Immutability (immunity from change) is a major design principle in this course, and we’ll talk much more about it in future readings.

Immutable objects (intended by their designer to always represent the same value) are denoted in a snapshot diagram by a double border, like the String objects in our diagram.

#### Mutable values



By contrast, [StringBuilder](http://docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html) (another built-in Java class) is a mutable object that represents a string of characters, and it has methods that change the value of the object:

StringBuilder sb = new StringBuilder("a");

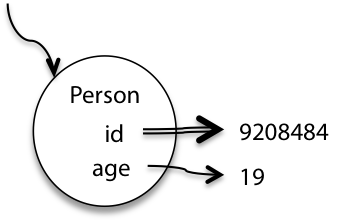
sb.append("b");

These two snapshot diagrams look very different, which is good: the difference between mutability and immutability will play an important role in making our code safe from bugs.

#### Immutable references

Java also gives us immutable references: variables that are assigned once and never reassigned. To make a reference immutable, declare it with the keyword final:

final int n = 5;



If the Java compiler isn’t convinced that your final variable will only be assigned once at runtime, then it will produce a compiler error. So final gives you static checking for immutable references.

In a snapshot diagram, an immutable reference (final) is denoted by a double arrow. Here’s an object whose id never changes (it can’t be reassigned to a different number), but whose age can change.

## Java Collections

In a snapshot diagram, we represent a List as an object with indices drawn as fields:

In a snapshot diagram, we represent a Set as an object with no-name fields:

In a snapshot diagram, we represent a Map as an object that contains key/value pairs:

# Reading 4: Code Review

## Objectives for Today’s Class

In today’s class, we will practice:

code review: reading and discussing code written by somebody else

general principles of good coding: things you can look for in every code review, regardless of programming language or program purpose

## Style Standards

Google Java Style Guide Online Guide

<https://google.github.io/styleguide/javaguide.html>

## Don’t Repeat Yourself

A switch statement is a more compact way to represent a series of if-else-if statements

## Comments Where Needed

One kind of crucial comment is a specification, which appears above a method or above a class and documents the behavior of the method or class. In Java, this is conventionally written as a Javadoc comment, meaning that it starts with /\*\* and includes @-syntax, like @param and @return for methods.

## Fail Fast

*Failing fast* means that code should reveal its bugs as early as possible.

## Avoid Magic Numbers

One way to explain a number is with a comment, but a far better way is to declare the number as a named constant with a good, clear name.

## One Purpose For Each Variable

Don’t reuse parameters, and don’t reuse variables. Variables are not a scarce resource in programming. Introduce them freely, give them good names, and just stop using them when you stop needing them.

Method parameters, in particular, should generally be left unmodified. (This is important for being ready-for-change — in the future, some other part of the method may want to know what the original parameters of the method were, so you shouldn’t blow them away while you’re computing.) It’s a good idea to use final for method parameters, and as many other variables as you can. For example:

public static int dayOfYear(final int month, final int dayOfMonth, final int year) {

...

}

## Use Good Names

Good method and variable names are long and self-descriptive. Comments can often be avoided entirely by making the code itself more readable, with better names that describe the methods and variables.

## Use Whitespace to Help the Reader

## Don’t Use Global Variables

A global variable is:

a *variable*, a name whose meaning can be changed

that is *global*, accessible and changeable from anywhere in the program.

In general, change global variables into parameters and return values, or put them inside objects that you’re calling methods on.

## Methods Should Return Results, not Print Them

In general, only the highest-level parts of a program should interact with the human user or the console. Lower-level parts should take their input as parameters and return their output as results. The sole exception here is debugging output, which can of course be printed to the console.

## Summary

Code review is a widely-used technique for improving software quality by human inspection. Code review can detect many kinds of problems in code, but as a starter, this reading talked about these general principles of good code:

Don’t Repeat Yourself (DRY)

Comments where needed

Fail fast

Avoid magic numbers

One purpose for each variable

Use good names

No global variables

Return results, don’t print them

Use whitespace for readability

The topics of today’s reading connect to our three key properties of good software as follows:

**Safe from bugs.** In general, code review uses human reviewers to find bugs. DRY code lets you fix a bug in only one place, without fear that it has propagated elsewhere. Commenting your assumptions clearly makes it less likely that another programmer will introduce a bug. The Fail Fast principle detects bugs as early as possible. Avoiding global variables makes it easier to localize bugs related to variable values, since non-global variables can be changed in only limited places in the code.

**Easy to understand.** Code review is really the only way to find obscure or confusing code, because other people are reading it and trying to understand it. Using judicious comments, avoiding magic numbers, keeping one purpose for each variable, using good names, and using whitespace well can all improve the understandability of code.

**Ready for change.** Code review helps here when it’s done by experienced software developers who can anticipate what might change and suggest ways to guard against it. DRY code is more ready for change, because a change only needs to be made in one place. Returning results instead of printing them makes it easier to adapt the code to a new purpose.

# Week 2

1. Make sure **assertions are always on**. Assertions are a great tool for keeping your code safe from bugs, but Java has them off by default.

In preferences, go to Java → Installed JREs. Click "Java SE 8", click "Edit…", and in the "Default VM arguments" box enter: -ea (which stands for enable assertions).

**(论坛上的讨论)Java Assertions are not to be used on production code and should be restricted to private methods**. //个人理解是不用于 production code 的意思是说默认编译时Assertion是不开起的。若开启，也可用于production code**(论坛上的讨论)**

2. Make sure your **Eclipse workspace refreshes automatically** whenever files are changed on the filesystem. This will be important when you run the grader script, so that you see the output files it generates.

In preferences, go to General → Workspace. Turn on both checkboxes that talk about refresh: "Refresh using native hooks or polling" and "Refresh on access."

3. **Tabs**. If you configure the editor to use spaces instead of tabs, then your code will look the same in all editors regardless of how that editor displays tab characters.

In preferences, go to Java → Code Style → Formatter. Click the "Edit…" button next to the active profile. In the new window, change the Tab policy to "Spaces only." Keep the Indentation size and Tab size at 4. To save your changes, enter a new "Profile name" at the top of the window and click OK.

# Reading 3: [Testing](http://web.mit.edu/6.005/www/sp16/classes/03-testing/)

## Objectives for Today’s Class

After today’s class, you should:

understand the value of testing, and know the process of test-first programming;

be able to design a test suite for a method by partitioning its input and output space and choosing good test cases;

be able to judge a test suite by measuring its code coverage; and

understand and know when to use blackbox vs. whitebox testing, unit tests vs. integration tests, and automated regression testing.

## Test-first Programming

In test-first-programming, you write tests before you even write any code. The development of a single function proceeds in this order:

Write a specification for the function.

Write tests that exercise the specification.

Write the actual code. Once your code passes the tests you wrote, you’re done.

## Choosing Test Cases by Partitioning

we divide the input space into **subdomains.**

The idea behind subdomains is to partition the input space into sets of similar inputs on which the program has similar behavior.

(论坛上的讨论)：对于function test是否需要考虑异常值(这里的异常值是指不符合specification说明范围的值)

You are mixing up different kinds of tests. You want to test that your web application (for example) doesn't crash or worse against all kinds of inputs. You don't want to do that with every single individual function though, because of (2).  
  
You want to be a) As flexible as possible towards inputs from the outside (users, "foreign" data), b) Be strict towards internal communication partners.

This is the difference between testing the program and testing functions. Few functions get into contact with the bad, bad outside world and all its hackers, if you designed it right.

//总结：program test需要考虑异常值，单个function test 不需要考虑异常值。

(论坛上的讨论)

### Include Boundaries in the Partition

Bugs often occur at *boundaries* between subdomains. Some examples:

0 is a boundary between positive numbers and negative numbers

the maximum and minimum values of numeric types, like int and double

emptiness (the empty string, empty list, empty array) for collection types

the first and last element of a collection

Why do bugs often happen at boundaries? One reason is that programmers often make **off-by-one mistakes** (like writing <= instead of <, or initializing a counter to 0 instead of 1). Another is that some boundaries may need to be handled as special cases in the code. Another is that boundaries may be places of discontinuity in the code’s behavior.

### Two Extremes for Covering the Partition

#### ****Full Cartesian product****

#### ****Cover each part****

//这一部分是说要包含所有情况，但是每种情况不需要单独一个测试用例，是需要有测试用例能包含该情况。举例：

Partition into:

relationship between a and b

a < b

a = b

a > b

value of a

a = 0

a < 0

a > 0

a = minimum integer

a = maximum integer

value of b

b = 0

b < 0

b > 0

b = minimum integer

b = maximum integer

 it would mean up to 3 × 5 × 5 = 75 test cases. In practice not all of these combinations are possible.

Every part of each dimension is covered by at least one test case, but not necessarily every combination. With this approach, the test suite for max might be as small as 5 test cases if carefully chosen.

Now let’s pick test values that cover all these classes:

(1, 2) covers a < b, a > 0, b > 0

(-1, -3) covers a > b, a < 0, b < 0

(0, 0) covers a = b, a = 0, b = 0

(Integer.MIN\_VALUE, Integer.MAX\_VALUE) covers a < b, a = minint, b = maxint

(Integer.MAX\_VALUE, Integer.MIN\_VALUE) covers a > b, a = maxint, b = minint

## Blackbox and Whitebox Testing

**Blackbox testing** means choosing test cases only from the specification, not the implementation of the function.

**Whitebox testing** (also called glass box testing) means choosing test cases with knowledge of how the function is actually implemented.

## Documenting Your Testing Strategy

For the example function on the left, on the right is how we can document the testing strategy.

|  |  |
| --- | --- |
| /\*\*  \* Reverses the end of a string.  \*  \* For example:  \* reverseEnd("Hello, world", 5)  \* returns "Hellodlrow ,"  \*  \* With start == 0, reverses the entire text.  \* With start == text.length(), reverses nothing.  \*  \* @param text non-null String that will have  \* its end reversed  \* @param start the index at which the  \* remainder of the input is  \* reversed, requires 0 <=  \* start <= text.length()  \* @return input text with the substring from  \* start to the end of the string  \* reversed  \*/  static String reverseEnd(String text, int start) | Document the strategy at the top of the test class:  /\*  \* Testing strategy  \*  \* Partition the inputs as follows:  \* text.length(): 0, 1, > 1  \* start: 0, 1, 1 < start < text.length(),  \* text.length() - 1, text.length()  \* text.length()-start: 0, 1, even > 1, odd > 1  \*  \* Include even- and odd-length reversals because  \* only odd has a middle element that doesn't move.  \*  \* Exhaustive Cartesian coverage of partitions.  \*/  Document how each test case was chosen, including white box tests:  // covers test.length() = 0,  // start = 0 = text.length(),  // text.length()-start = 0  @Test public void testEmpty() {  assertEquals("", reverseEnd("", 0));  }  // ... other test cases ... |

## Coverage

One way to judge a test suite is to ask how thoroughly it exercises the program. This notion is called *coverage*. Here are three common kinds of coverage:

**Statement coverage**: is every statement run by some test case?

**Branch coverage**: for every if or while statement in the program, are both the true and the false direction taken by some test case?

**Path coverage**: is every possible combination of branches — every path through the program — taken by some test case?

In industry, 100% statement coverage is a common goal, but even that is rarely achieved due to unreachable defensive code (like “should never get here” assertions). 100% branch coverage is highly desirable. Unfortunately 100% path coverage is infeasible, requiring exponential-size test suites to achieve.

A good code coverage tool for Eclipse is [EclEmma](http://www.eclemma.org/). Lines that have been executed by the test suite are colored green, and lines not yet covered are red.

For  [EclEmma](http://www.eclemma.org/), Preferences / General / Appearance / Editors / Text Editors / Annotations. The annotations to change are called Full Coverage, Partial Coverage, and No Coverage.

## Unit Testing and Stubs

The opposite of a unit test is an **integration test**, which tests a combination of modules, or even the entire program.

Isolating a higher-level module is possible if we write **stub** versions of the modules that it calls.// stub的意思是打桩。通过对底层函数的打桩(直接返回所需结果)，来实现对上层调用函数的单元测试

## Automated Testing and Regression Testing

**Automated testing** means running the tests and checking their results automatically.

Once you have test automation, it’s very important to rerun your tests when you modify your code. This prevents your program from regressing — introducing other bugs when you fix new bugs or add new features. Running all your tests after every change is called **regression testing**.// 这一段不重要

## Summary

In this reading, we saw these ideas:

Test-first programming. Write tests before you write code.

Partitioning and boundaries for choosing test cases systematically.

White box testing and statement coverage for filling out a test suite.

Unit-testing each module, in isolation as much as possible.

Automated regression testing to keep bugs from coming back.

The topics of today’s reading connect to our three key properties of good software as follows:

**Safe from bugs.** Testing is about finding bugs in your code, and test-first programming is about finding them as early as possible, immediately after you introduced them.

**Easy to understand.** Testing doesn’t help with this as much as code review does.

**Ready for change.** Readiness for change was considered by writing tests that only depend on behavior in the spec. We also talked about automated regression testing, which helps keep bugs from coming back when changes are made to code.

# Reading 5: Version Control

### Version control terminology

**Repository**: a local or remote store of the versions in our project

**Working copy**: a local, editable copy of our project that we can work on

**File**: a single file in our project

**Version** or **revision**: a record of the contents of our project at a point in time

**Change** or **diff**: the difference between two versions

**Head**: the current version

# 1.1.29 Git专题

###### 各类版本控制的定义

<https://git-scm.com/book/zh/v2/%E8%B5%B7%E6%AD%A5-%E5%85%B3%E4%BA%8E%E7%89%88%E6%9C%AC%E6%8E%A7%E5%88%B6>

###### 配置

安装完成后，还需要最后⼀一步设置，在命令⾏行输⼊入：

$ git config --global user.name "Your Name"

$ git config --global user.email "email@example.com"

因为Git是分布式版本控制系统，所以，每个机器都必须自报家⻔门：你的名字和Email地址。

###### 创建版本库

$ git init

###### 文件添加到版本库中

4.1 使用Windows的童鞋要特别注意：

千万不要使用Windows自带的记事本编辑任何文本文件。原因是Microsoft开发记事本的团队使用了一个非常弱智的行为来保存UTF-8编码的文件，他们自作聪明地在每个文件开头添加了0xefbbbf（十六进制）的字符，你会遇到很多不可思议的问题，比如，网页第一行可能会显示一个“?”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载Notepad++代替记事本，不但功能强⼤大，而且免费！记得把Notepad++的默认编码设置为UTF-8 without BOM即可。

4.2 第一步，用命令git add告诉Git，把文件添加到仓库：

$ git add readme1.txt

$ git add readme1.txt

4.3 第二步，⽤用命令git commit告诉Git，把⽂文件提交到仓库：

$ git commit -m "wrote a readme file"

###### 查看git仓库状态和比较差异

5.1 查看git仓库状态($ git status)

5.2 比较工作区和版本库同名文件差异($ git diff readme.txt)

比较readme.txt文件的差异

“git diff HEAD -- readme.txt”命令可以查看⼯工作区和版本库⾥里⾯面最新版本的区别

具体内容可参见：

https://blog.csdn.net/asheandwine/article/details/78982919

5.3 如果git status告诉你有⽂文件被修改过，⽤用git diff可以查看修改内容。

###### 版本回退

$ git log

*git log*命令显⽰示从最近到最远的提交⽇日志。

如果嫌输出信息太多，看得眼花缭乱的，可以试试加上

--pretty=oneline参数：

$ git log --pretty=oneline

$ git reset

如果嫌输出信息太多，看得眼花缭乱的，可以试试加上

Git必须知道当前版本是哪个版本，在Git中，用HEAD表⽰示当前版本，也就是最新的提交“ 3628164...882e1e0”（注意我的提交ID和你的肯定不⼀一样），上⼀一个版本就是HEAD^，上上⼀一个版本就是HEAD^^，当然往上100 个版本写100个^⽐比较容易数不过来，所以写成HEAD~100

$ git reset --hard HEAD^

或者 $git reset –hard head~1

或者$git reset –hard 提交ID(3628164),(回到当前最新版本，只需提交ID的前几位)

$ git reflog

Git提供了⼀一个命令*git reflog*⽤用来记录你的每⼀一次命令

举例：

$ git reflog

ea34578 HEAD@{0}: reset: moving to HEAD^

3628164 HEAD@{1}: commit: append GPL

第⼆行显⽰示“append GPL”的commit id是3628164。

你可以简单理解为，需要提交的⽂文件修改通通放到暂存区，然后，⼀一次性提交暂存区的所有修改。

###### 工作区和暂存区

工作区（Working Directory）**：**就是你在电脑⾥里能看到的目录，比如我的learngit文件夹就是一个⼯作区.

版本库（Repository）：⼯作区有一个隐藏目录“.git”，这个不算⼯作区，⽽是Git的版本库。Git的版本库里存了很多东⻄，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们⾃自动创建的第一个分⽀master，以及指向master的一个指针叫HEAD。



###### 撤销修改

8.1 工作区的修改撤销(git checkout -- readme.txt)

命令*git checkout -- readme.txt*意思就是，把readme.txt⽂文件在⼯作区的修改全部撤销，这⾥有两种情况：

1. 一种是readme.txt自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态。

2. 一种是readme.txt已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次*git commit*或*git add*时的状态。

*git checkout -- file*命令中的“--”很重要，没有“--”，就变成了“创建一个新分⽀支”的命令，我们在后⾯面的分支管理中会再次遇到*git checkout*命令。

8.2 暂存区的修改撤销(git reset)(和6.2是相同的命令，用法不同)

Git同样告诉我们，⽤用命令*git reset HEAD file*可以把暂存区的修改撤销掉（unstage），重新放回工作区：

$ git reset HEAD readme.txt

Unstaged changes after reset:

M readme.txt

*git reset*命令既可以回退版本，也可以把暂存区的修改回退到⼯作区。当我们⽤用HEAD时，表⽰示最新的版本。

8.3 小结

又到了小结时间。

场景1：当你改乱了工作区某个文件的内容，想直接丢弃⼯作区的修改时，⽤命令*git checkout -- file*。

场景2：当你不但改乱了⼯作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步⽤命令*git reset HEAD file*，就回到了场景1，第⼆步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。

###### 删除文件

10.1 版本库中删除文件($ git rm test.txt)

从版本库中删除该⽂文件，那就⽤用命令*git rm*删掉

$ git checkout -- test.txt（同8.1是相同的命令）

将工作区已删除的文件恢复为版本库里的文件版本。

9.3 $ git checkout 总结：

*git checkout*其实是⽤版本库⾥的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

## 远程

###### 远程仓库（配置公钥和私钥）

在继续阅读后续内容前，请自行注册GitHub账号。由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置：

第1步：创建SSH Key

在⽤用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有id\_rsa和id\_rsa.pub这两个⽂文件，如果已经有了，可直接 跳到下⼀一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

$ ssh-keygen -t rsa -C "youremail@example.com"

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是⽤用于军事目的，所以也无需设置密码。

第2步：GitHub上填写公钥

登陆GitHub，打开“Account settings”，“SSH Keys”页⾯：

然后，点“Add SSH Key”，填上任意Title，在Key文本框⾥里贴id\_rsa.pub文件的内容

**xzguo begin**

注意：

若提示如下：

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

Permissions 0674 for '/home/guoxiaozhou/.ssh/id\_rsa' are too open.

It is recommended that your private key files are NOT accessible by others.

This private key will be ignored.

Load key "/home/guoxiaozhou/.ssh/id\_rsa": bad permissions

Permission denied (publickey).

则需将id\_rsa的文件权限修改，命令如下：

chmod 700 /home/guoxiaozhou/.ssh/id\_rsa

**xzguo end**

###### 添加远程库(新建远程库，将本地库和远程库相关联)

添加后，远程库的名字就是origin，这是Git默认的叫法，也可以改成别的，但是origin这个名字一看就知道是远程库。

12.1 小结

要关联一个远程库(该远程库必须为空，即连[README.md](https://github.com/gxzmail/learngit/blob/master/README.md" \o "README.md)也不可以有)，使⽤用命令git remote add origin git@server-name:path/repo-name.git；

关联后，使⽤命令*git push -u origin master*第一次推送master分支的所有内容；

此后，每次本地提交后，只要有必要，就可以使⽤命令*git push origin master*推送最新修改；

###### 从远程库克隆

12.1$ git clone

$ git clone [git@github.com:michaelliao/gitskills.git](mailto:git@github.com:michaelliao/gitskills.git)

或者 $ git clone git@github.com: gxzmail/gitskills.git

12.2 小结

要克隆一个仓库，⾸先必须知道仓库的地址，然后使⽤*git clone*命令克隆。

###### 创建与合并分支

13.1 切换分支

13.1.1 创建且切换分支($ git branch + git checkout/git checkout –b)

首先，我们创建dev分支，然后切换到dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

*git checkout*命令加上*-b*参数表⽰创建并切换，相当于以下两条命令：

$ git branch dev ----创建dev分支

$ git checkout dev -----切换到dev分支

13.1.2 显示所有分支($ git branch)

*git branch*命令会列出所有分支，当前分支前⾯面会标一个\*号。

13.1.3 $ git merge

举例：

$ git merge dev

Updating d17efd8..fec145a

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

*git merge*命令用于合并指定分⽀到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

13.1.4 删除分支($ git branch -d )

举例：

$ git branch -d dev

Deleted branch dev (was fec145a).

放⼼地删除dev分支了

13.2 小结

Git⿎鼓励⼤大量使⽤用分⽀支：

查看分⽀支：git branch

创建分⽀支：git branch name

切换分⽀支：git checkout name

创建+切换分⽀支：git checkout -b name

合并某分⽀支到当前分⽀支：git merge name

删除分⽀支：git branch -d name

###### 解决冲突(该章节看pdf和doc的19节)

14.1 $ git log

用带参数的git log也可以看到分支的合并情况：

$ git log --graph --pretty=oneline --abbrev-commit

14.2 小结

⽤git log --graph命令可以看到分支合并图。

###### 分⽀管理策略

$ git merge --no-ff

准备合并dev分支，请注意--no-ff参数，表⽰禁用“Fast forward”：

$ git merge --no-ff -m "merge with no-ff" dev

小结

合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

###### Bug分支

$ git stash

Git还提供了⼀个stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

$ git stash

Saved working directory and index state WIP on dev: 6224937 add

merge

HEAD is now at 6224937 add merge

$ git stash list

⼯作区是干净的，刚才的⼯作现场存到哪去了？⽤git stash list命令看

$ git stash list

stash@{0}: WIP on dev: 6224937 add merge

$git stash apply/$git stash drop/$ git stash pop

工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除。

另一种⽅式是⽤git stash pop，恢复的同时把stash内容也删了。

小结

修复bug时，我们会通过创建新的bug分支进⾏修复，然后合并，最后删除；

当⼿头工作没有完成时，先把工作现场git stash一下，然后去修复bug，修复后，再git stash pop，回到工作现场。

###### Feature分支

强行删除分支($ git branch –D)

开发一个新feature，最好新建一个分支；

如果要丢弃一个没有被合并过的分支，可以通过git branch -D name强行删除。

###### 多人协作(重要，看pdf)

查看远程库信息($ git remote -v)

$ git remote -v

origin https://github.com/gxzmail/readme.git (fetch)

origin https://github.com/gxzmail/readme.git (push)

上⾯显⽰了可以抓取和推送的origin的地址。如果没有推送权限，就看不到push的地址。

推送分支($ git push origin dev)

如果要推送其他分支，比如dev，就改成：

$ git push origin dev

注意：

获取分支($ git pull)

$ git pull

如果git pull提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream branch-name origin/branch-name。

如果提示”fatal: [refusing to merge unrelated histories](https://blog.csdn.net/m0_37402140/article/details/72801372)”, 则说明它们是两个不同的项目，要把两个不同的项目合并，git需要添加一个参数，使用命令git pull origin master --allow-unrelated-histories即可。

###### 标签管理

创建标签

切换到需要打标签的分支上

$ git branch

\* dev

master

$ git checkout master

Switched to branch 'master'

打标签(git tag name)

20.1.2.1 当前版本打标签

$ git tag v1.0

20.1.2.2 历史版本打标签

首先方法是找到历史提交的commit id：

$ git log --pretty=oneline --abbrev-commit

其次根据commit id打上标签

$ git tag tagname commit\_id

20.1.2.3 带说明的标签

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字

举例：

$ git tag -a v0.1 -m "version 0.1 released" 3628164

20.1.2.4 带私钥的标签

还可以通过-s用私钥签名一个标签

举例：

$ git tag -s v0.2 -m "signed version 0.2 released" fec145a

操作标签

20.1.3.1删除本地标签

$ git tag -d v0.1

因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

20.1.3.2 推送标签到远程

使用命令git push origin tagname

举例：

$ git push origin v1.0

***或者***一次性推送全部尚未推送到远程的本地标签：

$ git push origin --tags

20.1.3.3 删除已推送到远程的标签

首先删除本地标签，具体参见20.1.3.1删除本地标签

其次，删除远程，命令如下：

$ git push origin :refs/tags/tagname

举例：

$ git push origin :refs/tags/v0.9

查看标签(git tag)

$ git tag

v1.0

查看标签信息

$git show tagname

举例：

$ git show v0.9

小结

• 命令*git tag name*⽤用于新建一个标签，默认为HEAD，也可以指定一个commit id；

• -a tagname -m "blablabla..."可以指定标签信息；

• -s tagname -m "blablabla..."可以⽤用PGP签名标签；

• 命令*git tag*可以查看所有标签；

• 命令git push origin tagname可以推送⼀一个本地标签；

• 命令git push origin --tags可以推送全部未推送过的本地标签；

• 命令git tag -d tagname可以删除⼀一个本地标签；

• 命令git push origin :refs/tags/tagname可以删除⼀一个远程标签。

###### 使用GitHub(参与开源项目)

略。具体看文档

###### 自定义Git（配置git）

略。具体看文档

###### 忽略特殊⽂文件

没看懂。具体看文档

###### 配置别名

$ git config --global alias.别名 实际命令

举例：

$ git config --global alias.st status

则git st 等同于 git status

###### 搭建Git服务器

略。具体看文档

# MOOC.FI

## Object-Oriented Programming with Java, part I

### WEEK 1

#### The program and the source code

#### 2. Main program body

#### 3. Getting to know your development environment