# 编程基础 IV - 函数式编程范式

刘 钦

# Outline

- 函数式编程范式 I
- 证明程序的正确性
- Git
- Java 编程基础 4

# Outline

- 函数式编程范式
  - 避免过程和数据的重复
  - 高阶函数
  - 信号流（枚举器+过滤器+映射+累积器）
  - Java 8 stream api
  - 函数式编程范式的特点
- 证明程序的正确性
- Git
- Java 编程基础 4

# 避免重复

# 过程

# Fibonacci数列

- Fib(n)
- = 0  n=0
- 1  n=1
- Fib(n-1)+Fib(n-2) 否则

# Fibonacci数列

- (define (fib n)
  - (cond ((= n 0) 0)
    - ((= n 1) 1)
    - (else (+ (fib (- n 1))
      - (fib (- n 2))))))

# 最大公约数

- 如果r是a除以b的余数，那么a和b的公约数正好也是b和r的公约数

# 最大公约数

- (define ( gcd a b)
  - (if (= b 0)
    - a
    - (gcd b (remainder a b))))

# 换零钱

- 将1美元换成半美元、四分之一美元、10美分、5美分、1美分总共有多少种换法?

# 换零钱

```
(define (count-change amount)
  (cc amount 5))
```

```
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1))
                 (cc (- amount
                        (first-denomination kinds-of-coins))
                     kinds-of-coins)))))
```

```
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

# 高阶函数

- 也是一种抽象
- 过程作为参数

# 案例

- 考虑三个过程
  - 计算从a到b的各个整数之和
  - 计算给定范围内的整数的立方之和
  - 计算下列序列之和
    - 1/(1*3)+1/(5*7)+1/(9*11)+…

# 计算从a到b的各个整数之和

- (define ( sum-integers a b)
- (if (> a b)
- 0
- (+ a (sum-integers( + a 1) b))))

# 计算给定范围内的整数的立方之和

- (define ( sum-cubes a b)
- (if (> a b)
- 0
- (+ (cube a) (sum-cubes( + a 1) b))))

# 计算下列序列之和
# 1/(1*3)+1/(5*7)+1/(9*11)+…

- (define ( sum-pi a b)
- (if (> a b)
- 0
- (+ (/ 1.0 (* a (+ a 2))) (sum-pi( + a 4) b))))

# 模板

- (define ( <name> a b)
- (if (> a b)
- 0
- (+ (<term> a) (<name>(<next> a) b))))

# 形参是函数

- (define ( sum term a next b)
- (if (> a b)
- 0
- (+ (term a)
- (sum term (next a) next b))))

# 计算从a到b的各个整数之和 - 高阶函数版

- (define (identity x) x)
- (define ( sum-integer a b)
- (sum identity a inc b))

# 计算给定范围内的整数的立方之和 - 高阶函数版

- (define (inc n) (+ n 1))
- (define ( sum-cubs a b)
- (sum cube a inc b))

# 计算下列序列之和
# 1/(1*3)+1/(5*7)+1/(9*11)+…  - 高阶函数版

- (define ( sum-pi a b)
- (define (pi-term x)
  - (/ 1.0 (* x (+ x 2))))
- (define ( pi-next x)
  - (+ x 4)
  - (sum pi-term a pi-next b))

# 利用lambda表达式构造过程

- 表达高阶函数
- (define (pi-sum a b)
  - (sum ( lambda (x) (/ 1.0 (* x (+ x 2))))
    - a
    - (lambda (x) (+ x 4))
    - b))

# 例子

- (define (f x y)
- (define (f-helper a b)
- (+ (* x (square a))
- (* y b)
- (* a b)))
- (f-helper (+ 1 (* x y))
- (- 1 y)))

# 利用lambda表达式构造过程

- 表达局部变量
- (define (f x y)
- ((lambda (a b)
- (+ (* x (square a))
- (* y b)
- (* a b)))
- (+ 1 (* x y))
- (- 1 y)))

# let关键字

- (define (f x y)
- (let ((a (+ 1 (* x y)))
- (b (- 1 y)))
- (+ (* x (square a))
- (* y b)
- (* a b))))

# 数据

# 有理数

- 有理数表示为两个证书的有序对

# 有理数

- (define (make-rat n d) (cons n d))
- (define (numer x ) (car x)
- (define (denom x) (cdr x))

# 考虑有理数约化到最简形式

- (define ( make-rat n d)
  - (let ( ( g (gcd n d)))
    - (cons (/ n g) (/ d g))))

# 链表

- 1-》2 -》3-》4

# 链表

- (list 1 2 3 4)
- 等价于
- (cons 1
  - (cons 2
    - (cons 3
      - (cons 4 nil))))

# list-ref

- 返回第n个项（从0开始计数）
  - n = 0 list-ref 返回表的car
  - 否则，返回表的cdr的第n-1个项

# list-ref

- (define (list-ref items n)
  - (if ( = n 0)
    - (car items)
    - (list-ref (cdr items) (- n 1))))

# scale-list

- 将一个表里所有元素按给定的因子做一次缩放

# scale-list

```
(define (scale-list items factor)
  (if (null? items)
      nil
      (cons (* (car items) factor)
            (scale-list (cdr items) factor))))
```

# map

- 返回将这一过程应用于表中各个元素得到的结果形成的表

# map

- (define (map proc items)
- (if (null? items)
- nil
- (cons (proc (car items))
- (map proc (cdr items)))))

# map的应用

- (map abs (list -10 2.5 -11.6 17))
- (10 2.5 11.6 17)
- (map (lambda (x) (* x x))
- (list 1 2 3 4))
- (1 4 9 16)

- Now we can give a new definition of scale-list in terms of map:

- (define (scale-list items factor)
- (map (lambda (x) (* x factor))
- items))

# 层次性结构

- （（1 2） 3 4)

# 层次性结构

- (cons (list 1 2) (list 3 4)

# count-leaves

- 数树的叶节点个数
  - 空表的叶节点个数为0；
  - 一个树叶的叶节点为1；
  - 否则为 car的叶节点个数+ cdr的叶节点个数。

# count-leaves

- (define (count-leaves x)
  - (cond (( null ? x) 0)
    - ((not (pair ? x )) 1)
    - (else ( + ( count-leaves (car x))
      - (count-leaves (cdr x))))))

# scale-tree

- 数值树
- 返回一颗具有同样形状的树，树中的每个数值都乘以这个因子

# scale-tree

- (define (scale-tree tree factor)
- (cond ((null? tree) null)
- ((not (pair? tree)) (* tree factor))
- (else (cons (scale-tree (car tree) factor)
- (scale-tree (cdr tree) factor)))))

# scale-tree lambda表达

- (define (scale-tree tree factor)
- (map (lambda (sub-tree)
- (if (pair? sub-tree)
- (scale-tree sub-tree factor)
- (* sub-tree factor)))
- tree))

# 信号流

- 枚举器-》过滤器-》映射-》累积器

# 计算值为奇数的叶子的平方和

- 枚举器-》过滤器-》映射-》累积器
- tree leaves-》odd？-》square-》+，0

# 计算值为奇数的叶子的平方和

- (define (sum-odd-squares tree)
- (cond ((null? tree) 0)
- ((not (pair? tree))
- (if (odd? tree) (square tree) 0))
- (else (+ (sum-odd-squares (car tree))
- (sum-odd-squares (cdr tree))))))

# 所有偶数的fibnacci数列

- 枚举器-》映射-》过滤器-》累积器
- integer-》fib -》even?》cons,()

# 所有偶数的fibnacci数列

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

# 案例

- 创建一个Messages Collection
- List<Message> messages = new ArrayList<>();
- messages.add(new Message("aglover", "foo", 56854));
- messages.add(new Message("aglover", "foo", 85));
- messages.add(new Message("aglover", "bar", 9999));
- messages.add(new Message("rsmith", "foo", 4564));

- 具体来讲，我希望找到Message当中所有延迟周期超过3000秒的条目并计算它们的总计延迟时长。

# 普通Java青年写法

```
long totalWaitTime = 0;
for (Message message : messages)
{
    if (message.delay > 3000)
    {
        totalWaitTime += message.delay;
    }
}
```

# 文艺Java 8青年写法

- Java 8 Stream API
  - long totWaitTime = messages.stream().filter(m -> m.delay > 3000).mapToLong(m -> m.delay).sum();

# 函数式编程

- From wiki：
  - In computer science, functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
- 参考 WIKI 的解释，函数式编程（Functional programming）指的是一种编程范式（Programming paradigm），将计算机运算看作是数学中函数的计算，并且避免了状态以及变量的概念。
- 　　我们所了解到的面向对象编程、指令式编程以及所要讲的函数式编程均为不同的编程范式。编程范式的主要作用在于提供且同时决定了程序员对于程序执行的看法。例如，在面向对象编程中，程序员认为程序是一系列相互作用的对象，而在函数式编程中一个程序会被看作是一个无状态的函数计算的序列。

# 函数式编程范式

1. Functions As First-class Citizens
    1. 函数被当作头等公民，意味着函数可以作为别的函数的参数、函数的返回值，赋值给变量或存储在数据结构中，通常以高阶函数（Higher Order Functions）的形式存在。
2. No Side Effects
    1. 在计算机科学中，函数副作用（Side Effect）指当调用函数时，除了返回函数值之外，还对外部作用域产生附加的影响，例如修改全局变量（函数外的变量）或修改参数。而严格的函数式编程是要求函数必须没有副作用，这意味着影响函数返回值的唯一因素就是它的参数。
3. No Changing-state
    1. 在函数式编程中，函数就是基础元素，可以完成几乎所有的操作，哪怕是最简单的计算，也是用函数来完成的，而我们平时在其他类型中所理解的变量（可修改，往往用来保存状态）在函数式编程中，是不可修改的，这意味着状态（State）不能保存在变量中，而事实上函数式编程是使用函数参数来保存状态，最好的例子便是递归。

```
String reverse(String arg) {
    if(arg.length == 0) {
        return arg;
    }
    else {
        return reverse(arg.substring(1, arg.length)) + arg.substring(0, 1);
    }
}
```

# 翻转一个字符串

4. **Currying**

在计算机科学中，柯里化（ Currying ），又译为卡瑞化或加里化，是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

举个简单的例子，在 Java 中计算一个整数的平方：

```
int pow(int i, int j);
int square(int i) {
    return pow(i, 2);
}
```

若编程语言支持 Currying 技术，是没有必要为某个函数手工创建另外一个函数去包装并转换它的接口，如在 Java 中，可以直接这样：

```
square = int pow(int i, 2);
```

在上述例子中，利用 Currying， 我们可以很方便地把接受两个参数的 `pow` 函数转换成只接受一个参数的 `square` ，它的功能是计算一个整数的平方。

简单地说，在函数式编程中，Currying 是一种可以快速且简单地实现函数封装的技术。

# Currying

5. **Concurrency**

在函数式编程中，由于没有副作用，函数不会影响或者依赖于全局状态，所以程序是支持并发（ Concurrency ）执行的。因为不需要采用锁机制，所以完全不用担心死锁或者并发竞争的情况会发生。

举个简单的例子：

```
String s1 = somewhatOperation1();
String s2 = somewhatOperation2();
String s3 = concatenate(s1, s2);
```

如果这是一门函数式编程语言而写的程序，编译器就会对代码进行分析，也许会发现生成 `s1` 和 `s2` 字符串的两个函数耗时比较大，进而安排它们并行运行。这在指令式编程中是不可能做到的，因为每一个函数都有可能修改其外部状态，然后接下来运行的函数又有可能会依赖于这些状态的值。

所以，函数式编程是十分适合那些需要高度并行的应用程序的。

# Concurrency

6. **Lazy Evaluation**

惰性求值（Lazy Evaluation），又称惰性计算、懒惰求值，是一个计算机编程中的一个概念，它的目的是要最小化计算机要做的工作。

以在上面介绍的并发中所贴的代码为例，我们知道，函数 `somewhatOperation1` 和 `somewhatOperation2` 是可以并发执行的，但其实由于支持 Lazy Evaluation，编译器会进一步把代码优化，不会在被赋值到 `s1` 与 `s2` 时就立即求值，而是在真正需要 `s1` 与 `s2` 的地方才求值，即推迟到在 `concatenate` 函数中需要 `s1` 与 `s2` 的时候才求值。

或许有人会觉得，这些运算迟早都要计算，那延迟的意义体现在哪里？其实这种优化恰恰能省去无谓的计算。譬如说，在 `concatenate` 函数中有一个条件判断语句，分别用到了两个参数 `s1` 与 `s2` 中的其中一个，那么在执行该函数时，由于只会用到其中一个参数，另外一个在执行过程中是不会被计算的。

惰性求值使得代码具备了巨大的优化潜能。支持惰性求值的编译器会像数学家看待代数表达式那样看待函数式编程的程序：抵消相同项从而避免执行无谓的代码，安排代码执行顺序从而实现更高的执行效率甚至是减少错误。

惰性求值另一个重要的好处是它可以构造一个无限的数据类型。譬如构建一个存储斐波那契数列数字的列表，很显然我们是无法在有限的时间内计算出这个无限的列表并且存储在内存中。在 Java 中，我们可以通过循环来返回这个数列中的某个数字，而在 Haskell 等函数式编程语言中，我们真的可以定义一个斐波那契数列的无穷列表结构，由于语言本身支持惰性求值，在这个列表中的数，只有真正被用到的时候才会被计算出来。这个特性能让我们把很多问题抽象化，然后在更高层次上去解决它们。

# Lazy Evaluation

# Outline

- 函数式编程范式
- 证明程序的正确性
  - Dijkstra
  - Hoare
- Git
- Java 编程基础 4

# How to proof your program’s correctness

- Edsger W. Dijkstra
  - Enumeration
  - Mathematical induction
  - Abstraction
- C. A. R. HOARE
  - Axioms proof

# How to proof — Edsger W. Dijkstra

- Two statements
  - int r = a; int dd =d;
  - while (dd<=r)
    - dd=2*dd;
  - while (dd!=d){
    - dd = dd/2;  //halve dd;
    - if （dd<=r）{ r= r-dd;} //reduce r modulo dd
  - }
- operating on the variable “r” and “dd” leaves the relations
  - 0 <= r < dd
    - which is satisfied to start with

satisfied to start with. After the execution of the first statement, which halves the value of *dd*, but leaves *r* unchanged, the relations

$$0 \leqslant r < 2*dd \tag{2}$$

will hold. Now we distinguish two mutually exclusive cases.

(1) $dd \leqslant r$. Together with (2) this leads to the relations

$$dd \leqslant r < 2*dd; \tag{3}$$

In this case the statement following **do** will be executed, ordering a decrease of *r* by *dd*, so that from (3) it follows that eventually

$$0 \leqslant r < dd,$$

i.e. (1) will be satisfied.

(2) **non** $dd \leqslant r$ (i.e. $dd > r$). In this case the statement following **do** will be skipped and therefore also *r* has its final value. In this case "$dd > r$" together

# Enumeration

TABLE I

| | | |
|---|---|---|
| A1 | $x + y = y + x$ | addition is commutative |
| A2 | $x \times y = y \times x$ | multiplication is commutative |
| A3 | $(x + y) + z = x + (y + z)$ | addition is associative |
| A4 | $(x \times y) \times z = x \times (y \times z)$ | multiplication is associative |
| A5 | $x \times (y + z) = x \times y + x \times z$ | multiplication distributes through addition |
| A6 | $y \leqslant x \supset (x - y) + y = x$ | addition cancels subtraction |
| A7 | $x + 0 = x$ | |
| A8 | $x \times 0 = 0$ | |
| A9 | $x \times 1 = x$ | |

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

| | |
|---|---|
| A5 | $(r - y) + y \times (1 + q)$ |
| | $= (r - y) + (y \times 1 + y \times q$ |
| A9 | $= (r - y) + (y + y \times q)$ |
| A3 | $= ((r - y) + y) + y \times q$ |
| A6 | $= r + y \times q$ provided $y \leqslant r$ |

# How to proof — Hoare

D0 Axiom of Assignment

$\vdash P_0\{x := f\}\, P$

where

$x$ is a variable identifier;

$f$ is an expression;

$P_0$ is obtained from $P$ by substituting $f$ for all occurrences of $x$.

D1 Rules of Consequence

If $\vdash P\{Q\}R$ and $\vdash R \supset S$ then $\vdash P\{Q\}S$

If $\vdash P\{Q\}R$ and $\vdash S \supset P$ then $\vdash S\{Q\}R$

D2 Rule of Composition

If $\vdash P\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R$ then $\vdash P\{(Q_1\,;\, Q_2)\}R$

D3 Rule of Iteration

If $\vdash P \wedge B\{S\}P$ then $\vdash P\{\textbf{while } B \textbf{ do } S\}\neg B \wedge P$

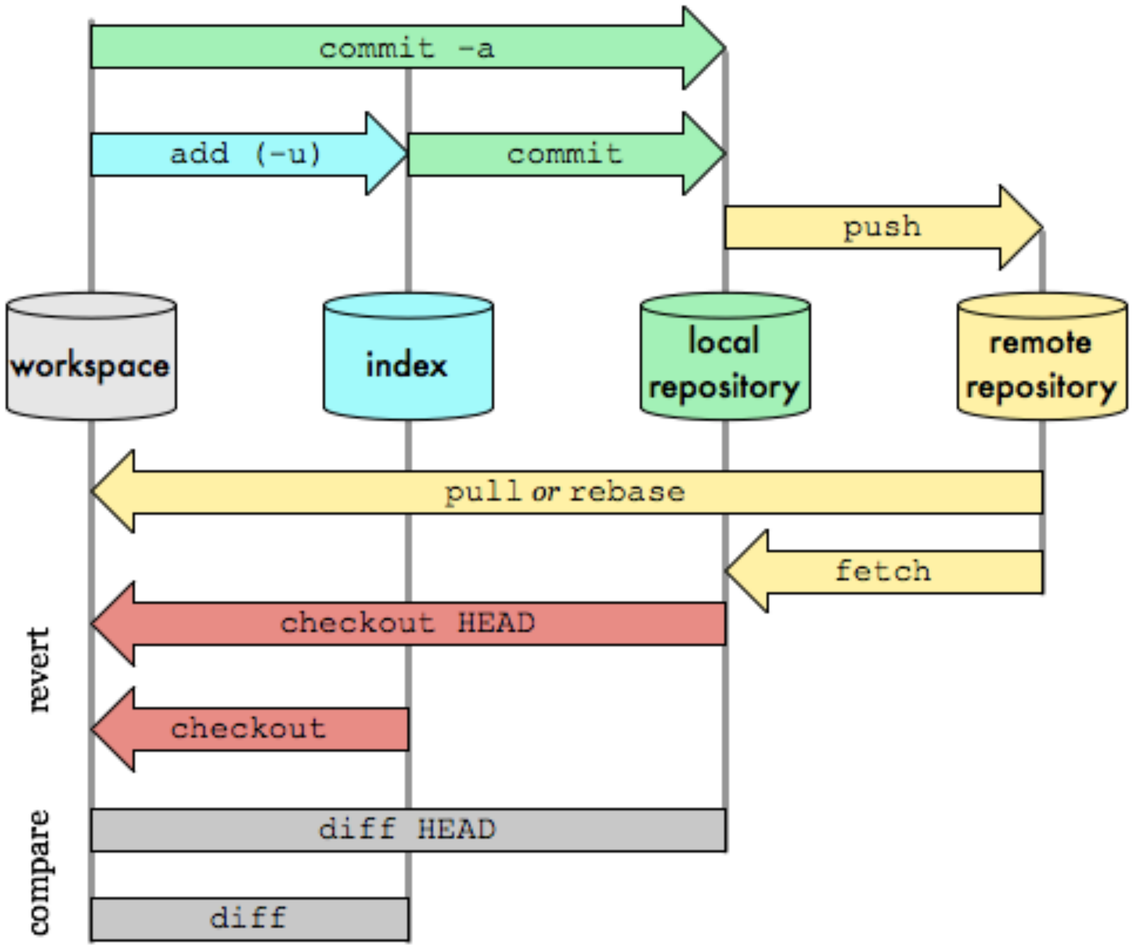
TABLE III

| Line number | Formal proof | Justification |
|---|---|---|
| 1 | $\textbf{true} \supset x = x + y \times 0$ | Lemma 1 |
| 2 | $x = x + y \times 0\{r := x\}x = r + y \times 0$ | D0 |
| 3 | $x = r + y \times 0\ \{q := 0\}\ x = r + y \times q$ | D0 |
| 4 | $\textbf{true}\ \{r := x\}\ x = r + y \times 0$ | D1 (1, 2) |
| 5 | $\textbf{true}\ \{r := x;\ \ q := 0\}\ x = r + y \times q$ | D2 (4, 3) |
| 6 | $x = r + y \times q \wedge y \leqslant r \supset x = (r-y) + y \times (1+q)$ | Lemma 2 |
| 7 | $x = (r-y) + y \times (1+q)\{r := r-y\}x = r + y \times (1+q)$ | D0 |
| 8 | $x = r + y \times (1+q)\{q := 1+q\}x = r + y \times q$ | D0 |
| 9 | $x = (r-y) + y \times (1+q)\{r := r-y;\ q := 1+q\}\ x = r + y \times q$ | D2 (7, 8) |
| 10 | $x = r + y \times q \wedge y \leqslant r\ \{r := r-y;\ q := 1+q\}\ x = r + y \times q$ | D1 (6, 9) |
| 11 | $x = r + y \times q\ \{\textbf{while } y \leqslant r \textbf{ do } (r := r-y;\ \ q := 1+q)\}$ $\neg y \leqslant r \wedge x = r + y \times q$ | D3 (10) |
| 12 | $\textbf{true}\ \{((r := x;\ \ q := 0);\ \textbf{while } y \leqslant r \textbf{ do } (r := r-y;\ \ q := 1+q))\}\ \neg y \leqslant r \wedge x = r + y \times q$ | D2 (5, 11) |

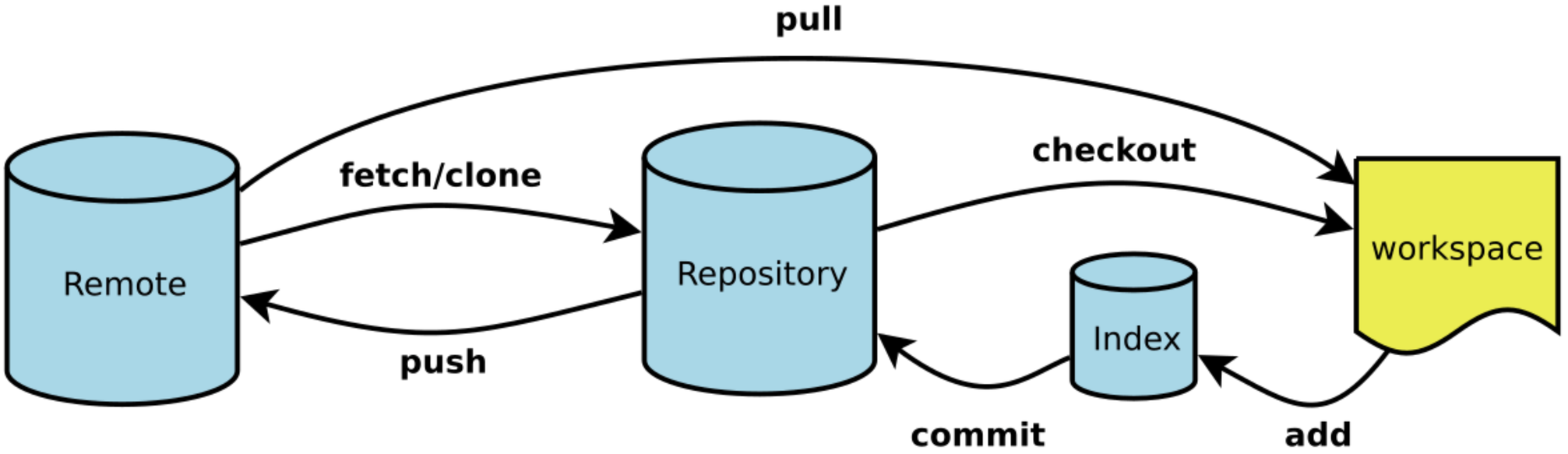
# Outline

- 函数式编程范式 I
- 证明程序的正确性
- Git
- Java 编程基础 4

# 工作流程


commit -a
add (-u)
commit
push
workspace
index
local repository
remote repository
pull or rebase
fetch
checkout HEAD
revert
checkout
compare
diff HEAD
diff

# 6个最常用的命令


pull
fetch/clone
checkout
Remote
Repository
workspace
Index
push
commit
add

# 下载项目

- # 下载一个项目和它的整个代码历史
- $ git clone [url]

# 添加文件

- # 添加指定文件到暂存区
- $ git add [file1] [file2] ...

- # 添加指定目录到暂存区，包括子目录
- $ git add [dir]

- # 添加当前目录的所有文件到暂存区
- $ git add .

# 代码提交

- # 提交暂存区到仓库区
- $ git commit -m [message]

- # 提交暂存区的指定文件到仓库区
- $ git commit [file1] [file2] ... -m [message]

- # 提交工作区自上次commit之后的变化，直接到仓库区
- $ git commit -a

- # 提交时显示所有diff信息
- $ git commit -v

# 查看

- # 显示有变更的文件
- $ git status

- # 显示当前分支的版本历史
- $ git log

- # 显示暂存区和工作区的差异
- $ git diff

# 撤销

- # 恢复暂存区的指定文件到工作区
- $ git checkout [file]

- # 恢复某个commit的指定文件到暂存区和工作区
- $ git checkout [commit] [file]

- # 恢复暂存区的所有文件到工作区
- $ git checkout .

- # 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变
- $ git reset [file]

- # 重置暂存区与工作区，与上一次commit保持一致
- $ git reset --hard

- # 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变
- $ git reset [commit]

- # 重置当前分支的HEAD为指定commit，同时重置暂存区和工作区，与指定commit一致
- $ git reset --hard [commit]

# 远程同步

- # 显示所有远程仓库
- $ git remote -v

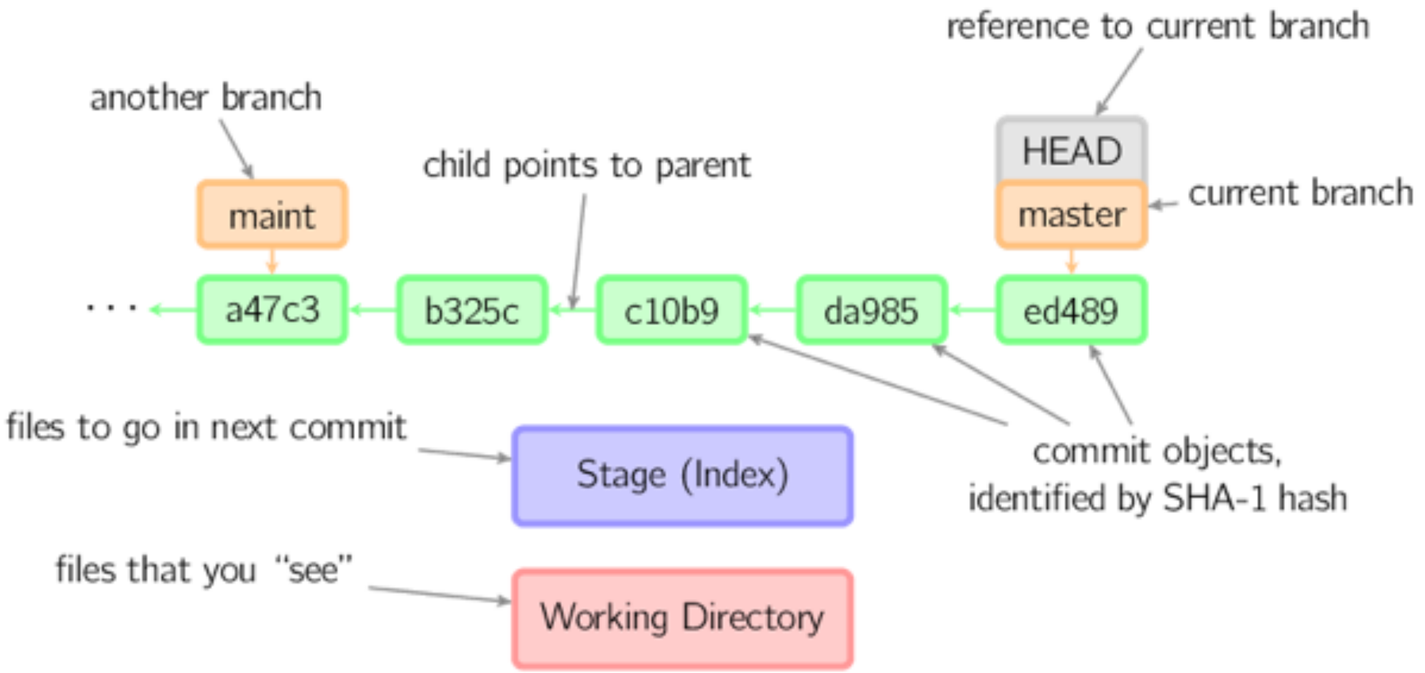
- # 显示某个远程仓库的信息
- $ git remote show [remote]

- # 下载远程仓库的所有变动
- $ git fetch [remote]

- # 取回远程仓库的变化，并与本地分支合并
- $ git pull [remote] [branch]

- # 上传本地指定分支到远程仓库
- $ git push [remote] [branch]

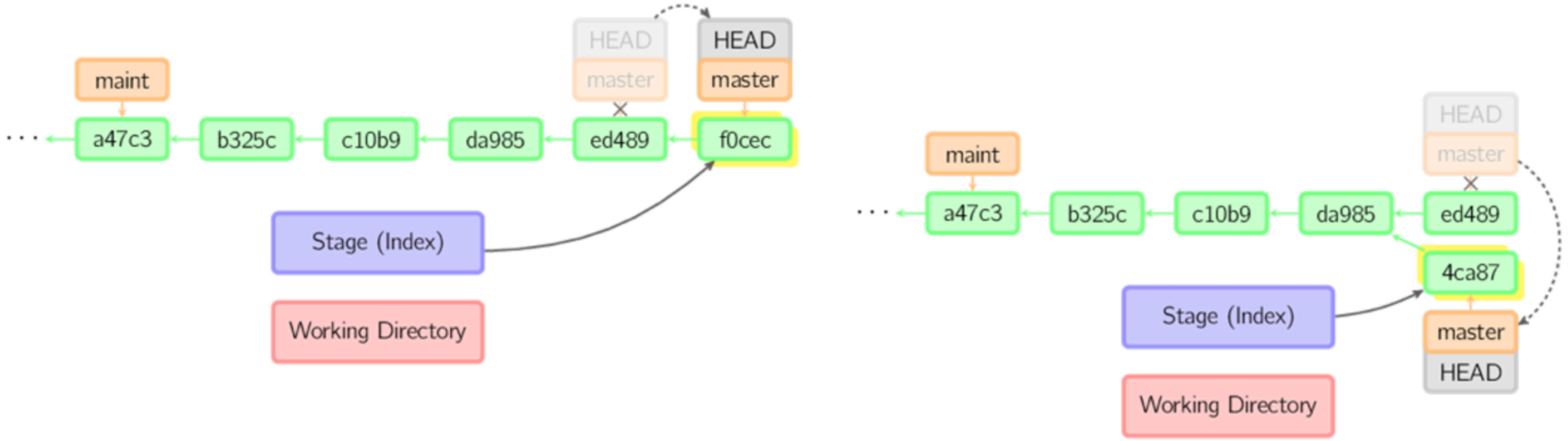
# 图例


reference to current branch
another branch
HEAD
child points to parent
maint
master
current branch
. . .
a47c3
b325c
c10b9
da985
ed489
files to go in next commit
Stage (Index)
commit objects,
identified by SHA-1 hash
files that you "see"
Working Directory

# Commit

## commit

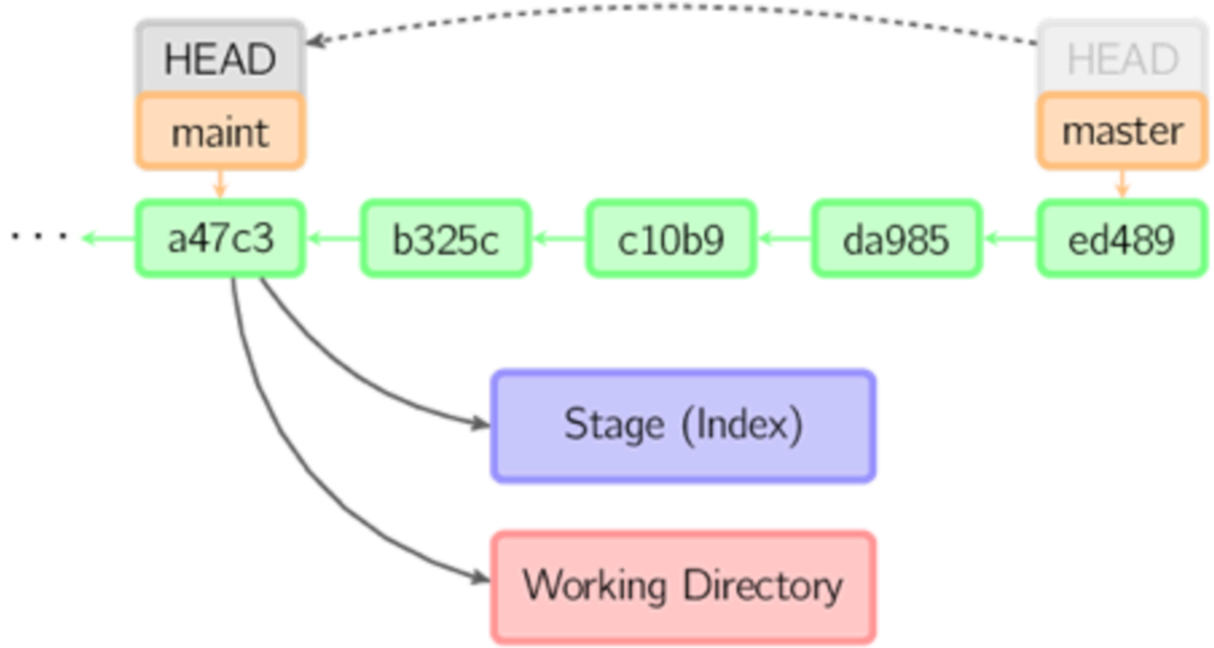
commit把暂存区的内容存入到本地仓库，并使得当前分支的HEAD向后移动一个提交点。如果对最后一次commit不满意，可以使用`git commit --amend`来进行撤销，修改之后再提交。如图所示的，ed489被4ca87取代，但是git log里看不到ed489的影子，这也正是amend的本意：原地修改，让上一次提交不露痕迹。



# Checkout

## checkout

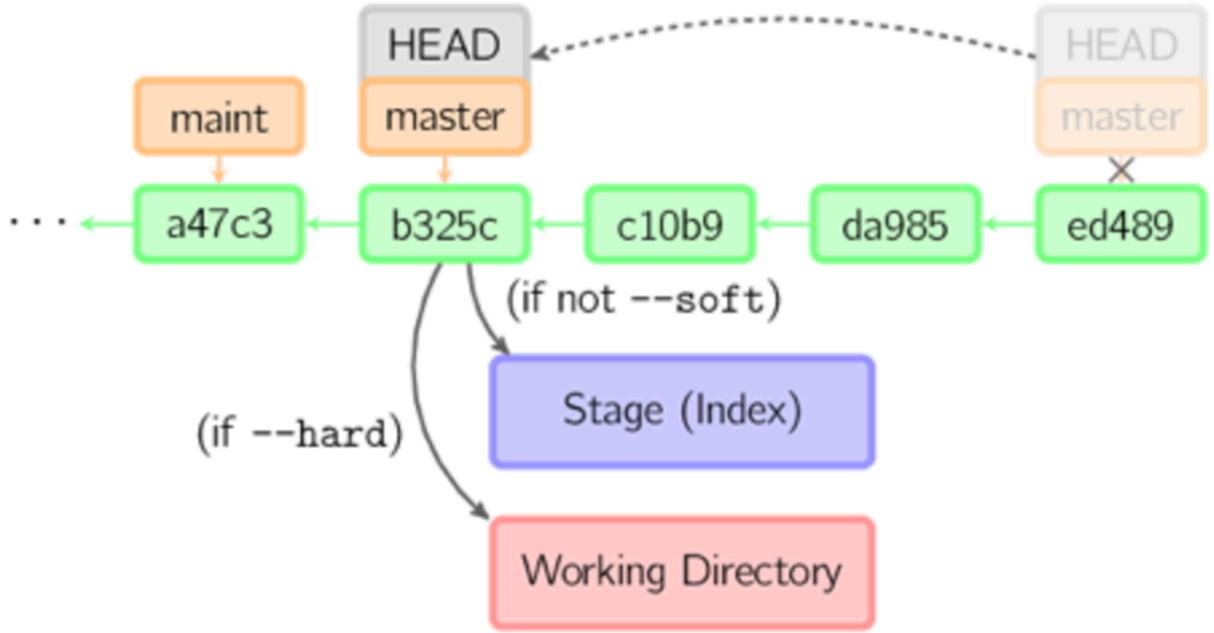
checkout用来检出并切换分支。checkout成功后，HEAD会指向被检出分支的最后一次提交点。对应的，工作目录、暂存区也都会与当前的分支进行匹配。下图是执行`git checkout maint`后的结果：



# Reset

## reset

reset命令把当前分支指向另一个位置，并且相应的变动工作目录和索引。如下图，执行`git reset HEAD~3`后，当前分支相当于回滚了3个提交点，由ed489回到了b325c：

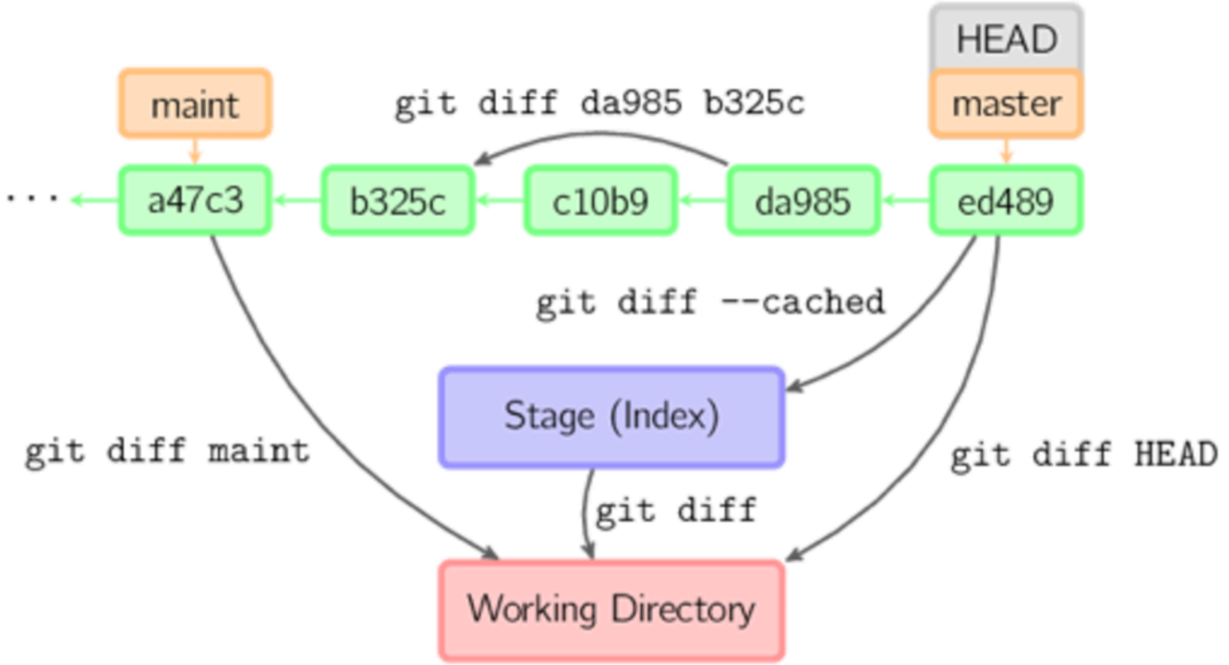



reset有3种常用的模式：
–soft，只改变提交点，暂存区和工作目录的内容都不改变
–mixed，改变提交点，同时改变暂存区的内容。这是默认的回滚方式
–hard，暂存区、工作目录的内容都会被修改到与提交点完全一致的状态

# Diff

## diff

我们在commit、merge、rebase、打patch之前，通常都需要看看这次提交都干了些什么，于是diff命令就派上用场了：



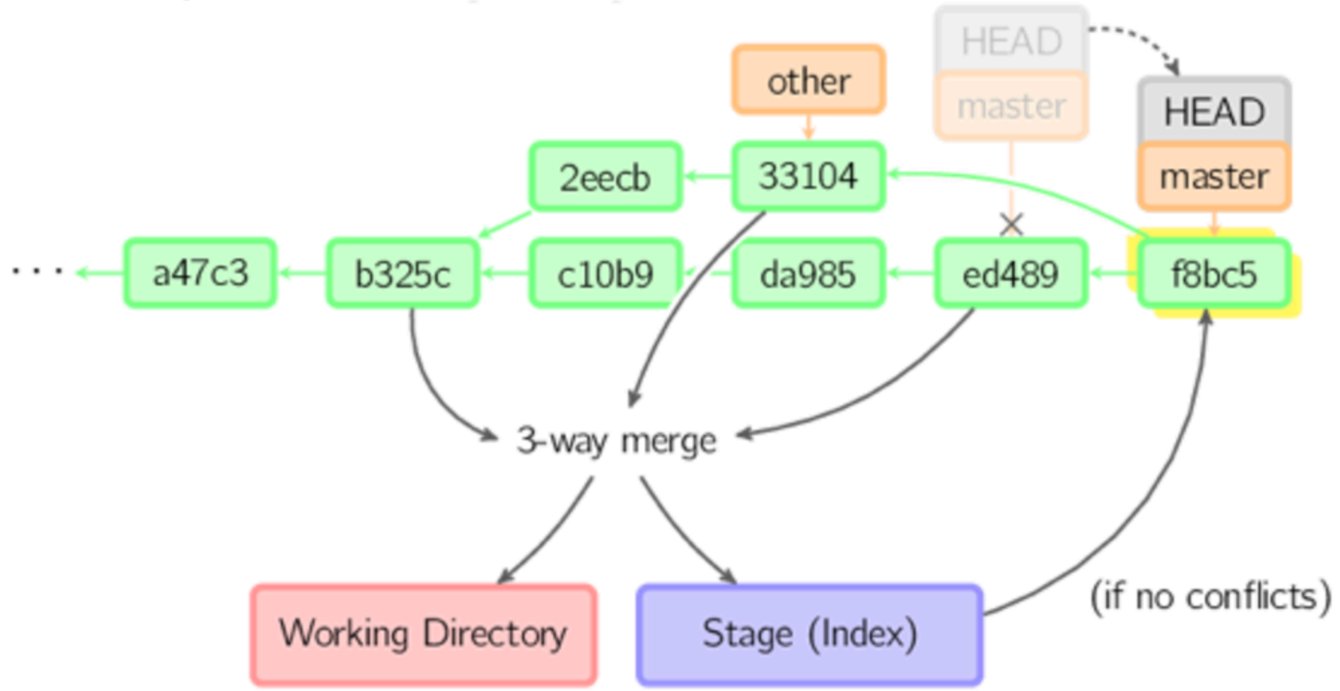

来比较下上图中5种不同的diff方式：

比较不同的提交点之间的异同，用`git diff 提交点1 提交点2`
比较当前分支与其他分支的异同，用`git diff 其他分支名称`
在当前分支内部进行比较，比较最新提交点与当前工作目录，用`git diff HEAD`
在当前分支内部进行比较，比较最新提交点与暂存区的内容，用`git diff --cached`
在当前分支内部进行比较，比较暂存区与当前工作目录，用`git diff`

# Merge

## merge

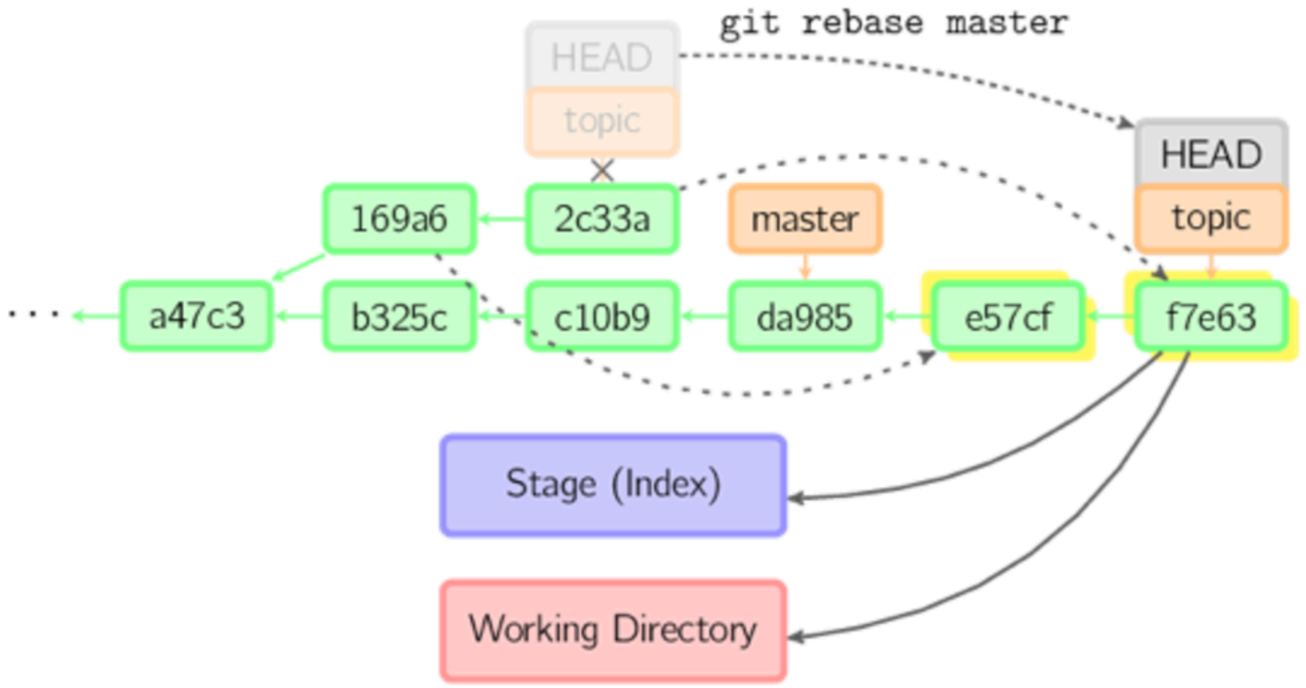
merge命令把不同的分支合并起来。如下图，HEAD处于master分支的ed489提交点上，other分支处于33104提交点上，项目负责人看了下觉得other分支的代码写的不错，于是想把代码合并到master分支，因此直接执行 `git merge other`，如果没有发生冲突，other就成功合并到master分支了。



# Rebase

## rebase

rebase又称为衍合，是合并的另外一种选择。merge把两个分支合并到一起进行提交，无论从它们公共的父节点开始(如上图，other分支与 master分支公共的父节点b325c)，被合并的分支(other分支)发生过多少次提交，合并都只会在当前的分支上产生一次提交日志，如上图的 f8bc5。所以merge产生的提交日志不是线性的，万一某天需要回滚，就只能把merge整体回滚。而rebase可以理解为verbosely merge，完全重演下图分支topic的演化过程到master分支上。如下图：

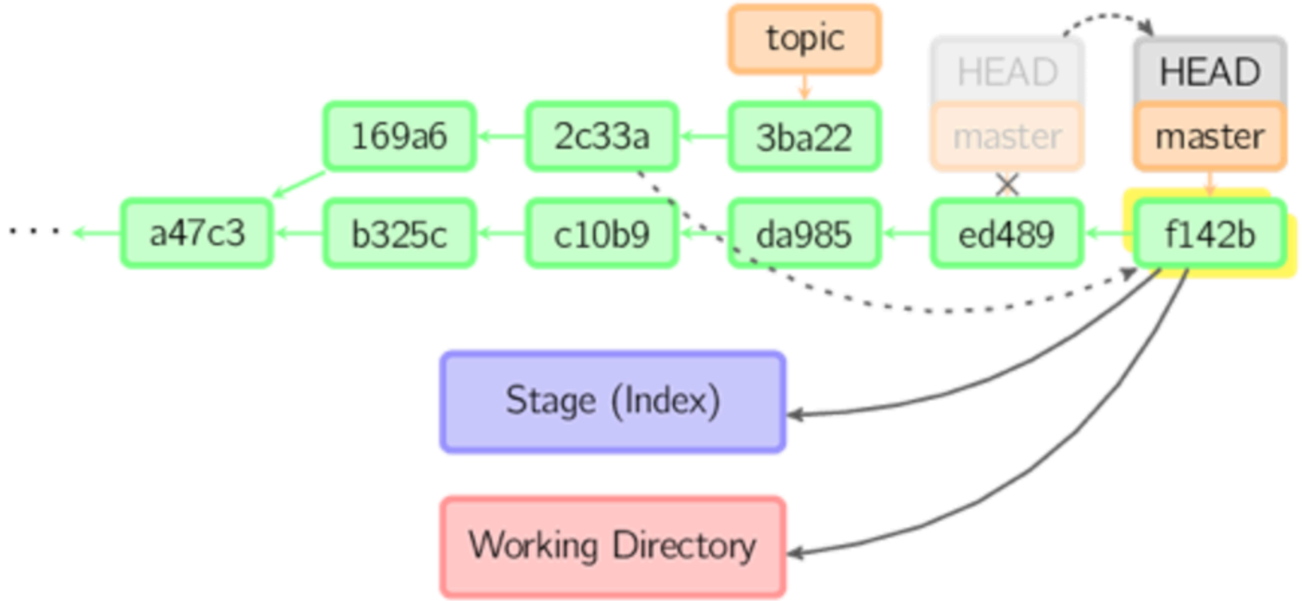



在开始阶段，我们处于topic分支上，执行`git rebase master`，那么169a6和2c33a上发生的事情都在master分支上重演一遍，分别对应于master分支上的e57cf和f7e63，最后checkout切换回到topic分支。这一点与merge是一样的，合并前后所处的分支并没有改变。`git rebase master`，通俗的解释就是topic分支想站在master的肩膀上继续下去。

# Cherry-pick

## cherry-pick

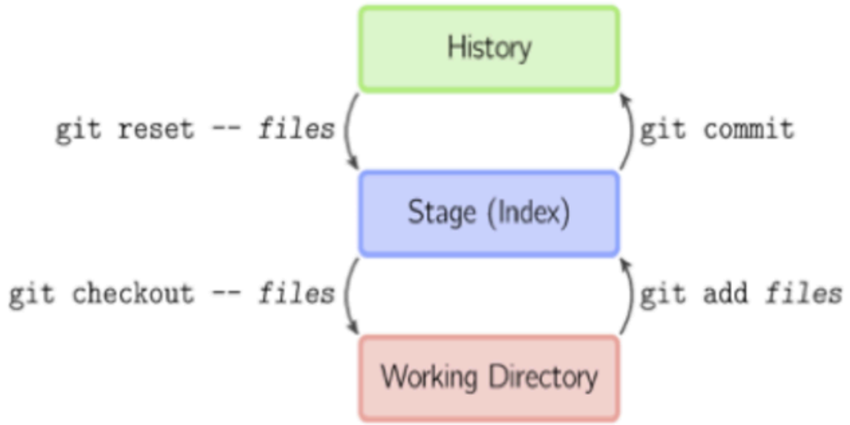
cherry-pick命令复制一个提交点所做的工作，把它完整的应用到当前分支的某个提交点上。rebase可以认为是自动化的线性的cherry-pick。
例如执行`git cherry-pick 2c33a`:

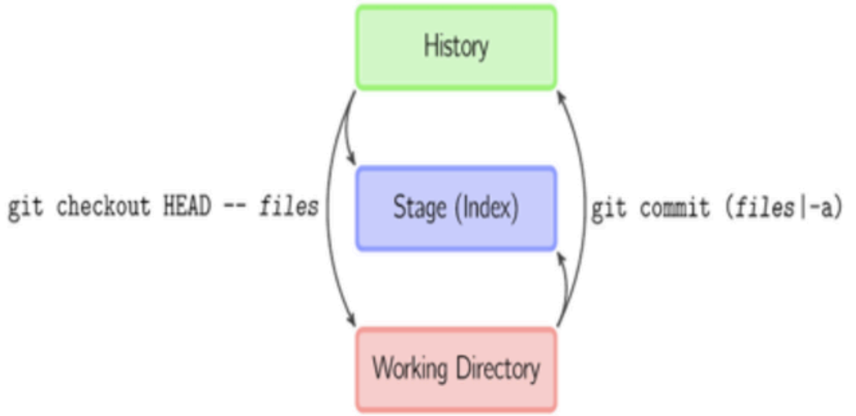


# 正反过程对比

## 正反过程对比

理解了上面最晦涩的几个命令，我们来从正反两个方向对比下版本在本地的3个阶段之间是如何转化的。如下图(history就是本地仓库)：




如果觉得从本地工作目录到本地历史库每次都要经过index暂存区过渡不方便，可以采用图形右边的方式，把两步合并为一步。

# Outline

- 函数式编程范式 I
- 证明程序的正确性
- Git
- Java 编程基础 4
  - 文件

# 持久化

# 持久化

- 文件
- 对象序列化
- 网络

# 文件输入输出

- File
- FileWriter & FileReader
- BufferWriter & BufferReader

# Java.io.File类

① Make a File object representing an existing file

```
File f = new File("MyCode.txt");
```

② Make a new directory

```
File dir = new File("Chapter7");
dir.mkdir();
```

③ List the contents of a directory

```
if (dir.isDirectory()) {
      String[] dirContents = dir.list();
      for (int i = 0; i < dirContents.length; i++) {
          System.out.println(dirContents[i]);
      }
}
```

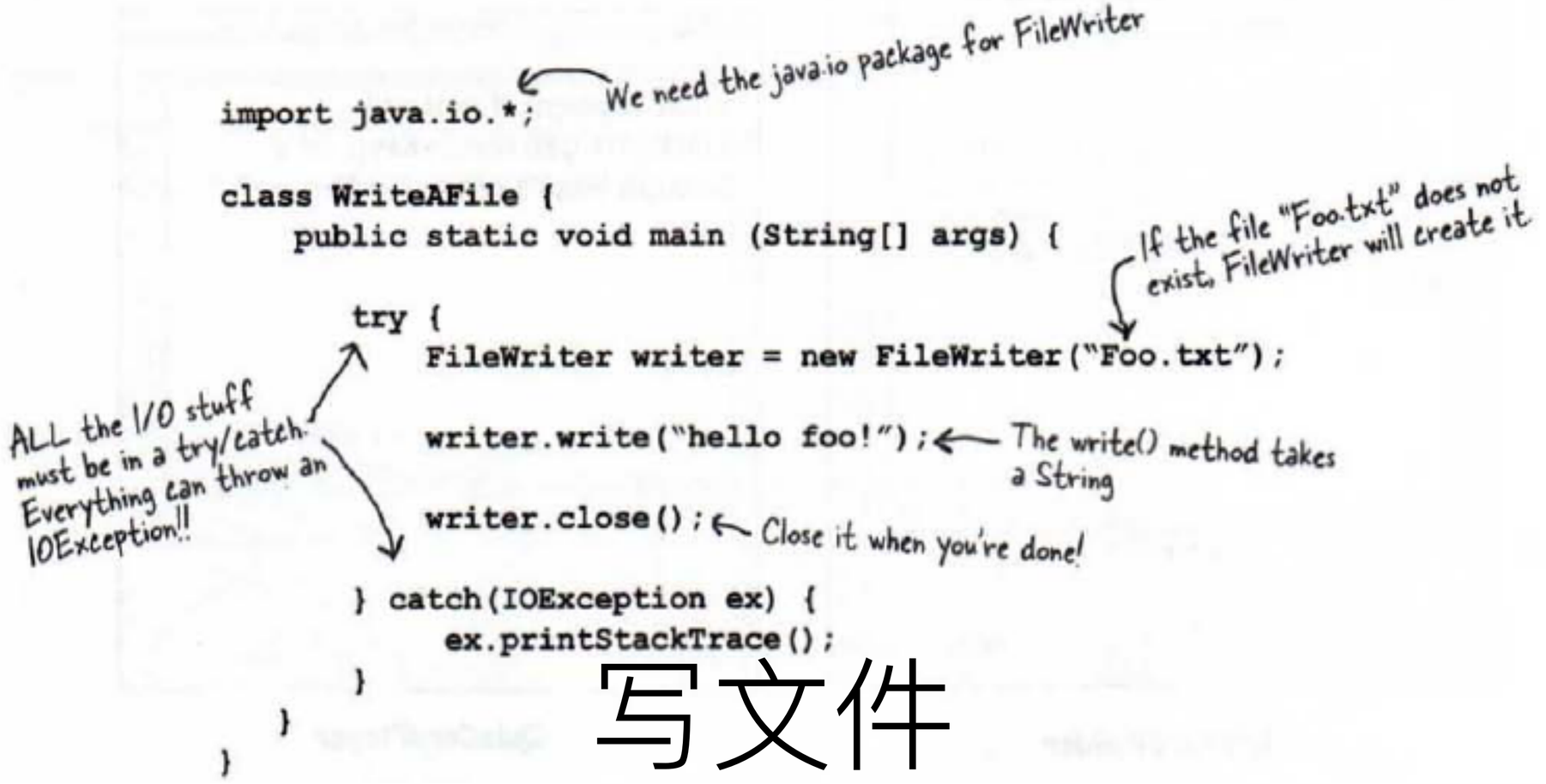
④ Get the absolute path of a file or directory

```
System.out.println(dir.getAbsolutePath());
```

⑤ Delete a file or directory (returns true if successful)

**To write a String:**

```
fileWriter.write("My first String to save");
```




# 写文件

```
class ReadAFile {
    public static void main (String[] args) {

        try {
            File myFile = new File("MyText.txt");
            FileReader fileReader = new FileReader(myFile);

            BufferedReader reader = new BufferedReader(fileReader);

            String line = null;

            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

A FileReader is a connection stream for characters, that connects to a text file

Chain the FileReader to a BufferedReader for more efficient reading. It'll go back to the file to read only when the buffer is empty (because the program has read everything in it).

Make a String variable to hold each line as the line is read

This says, "Read a line of text, and assign it to the String variable 'line'. While that variable is not null (because there WAS something to read) print out the line that was just read."

Or another way of saying it, "While there are still lines to read, read them and print them."

# 读文件

# CSV文件

- 逗号分隔值（Comma-Separated Values，CSV，有时也称为字符分隔值，因为分隔字符也可以不是逗号），其文件以纯文本形式存储表格数据（数字和文本）。纯文本意味着该文件是一个字符序列，不含必须象二进制数字那样被解读的数据。CSV文件由任意数目的记录组成，记录间以某种换行符分隔；每条记录由字段组成，字段间的分隔符是其它字符或字符串，最常见的是逗号或制表符。通常，所有记录都有完全相同的字段序列。

- CSV文件格式的通用标准并不存在，但是在RFC 4180中有基础性的描述。使用的字符编码同样没有被指定，但是7-bit ASCII是最基本的通用编码。

# 样例

- Year,Make,Model,Description,Price
- 1997,Ford,E350,"ac, abs, moon",3000.00
- 1999,Chevy,"Venture ""Extended Edition""","",4900.00
- 1999,Chevy,"Venture ""Extended Edition, Very Large""",,5000.00
- 1996,Jeep,Grand Cherokee,"MUST SELL!
- air, moon roof, loaded",4799.00

| 年份 | 品牌 | 型号 | 描述 | 价格 |
|---|---|---|---|---|
| 1997 | Ford | E350 | ac, abs, moon | 3000.00 |
| 1999 | Chevy | Venture "Extended Edition" | | 4900.00 |
| 1999 | Chevy | Venture "Extended Edition, Very Large" | | 5000.00 |
| 1996 | Jeep | Grand Cherokee | MUST SELL!<br>air, moon roof, loaded | 4799.00 |

# Homework09 - Stream

- Deadline: March 29, 23:59:59
- 假如给定一个名称列表，其中一些名称包含一个字符。系统会要求您在一个逗号分隔的字符串中返回名称，该字符串中不包含单字母的名称，每个名称的首字母都大写。
  - 输入List("neal", "s", "stu", "j", "rich", “bob")
  - 输出”Neal,Stu,Rich,Bob”
- 分别用命令式范式和函数式范式实现。
- 提交源代码。

# Homework10 - CSVFile

- Deadline: March 29, 23:59:59
- 编写一个程序，读入以下数据文件
- Ling,Mai,55900
- Johnson,Jim,56500
- …
- Zarnecki,Sabrina,51500
- 处理改记录，并以格式化的表格形式显示结果，间隔均匀（4个空格），如示例输出。
  - Last  Fisrt  Salary
  - Ling  Mai  55900
  - Johnson  Jim  56500
  - …
  - Zarnecki  Sabrina  51500
- 提交源代码。