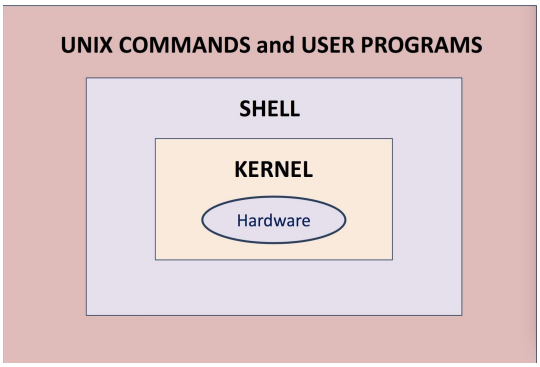
# CSE 15L Final Review

- Pipe and redirect
  - Eg: who | sort > curr
  - Eg: (who | sort) > curr
- Shell scripting
  - Unix commands and syntax
- Chmod command
  - Change permissions of file
  - Chmod rwx rwx rwx
- Quote character
  - Single quote: taken literally
  - Back quote: execute everything
  - Double quote: weak quote
- Global variables (environment variables)
  - $PATH …
- Defining local variables
  - NO SPACES AROUND = (if space, more like a comparison )
- Positional parameters
  - Eg: $?, $1, $@ …
- The process ID (PID)
  - How to kill
  - What is a child …
  - Kernel with PID 0
  - Eg: ps -eaf | grep username
- Introduction to debugging
  - Steps … one to six in order
    - Understand the system
    - …
  - Memory management … leaks …
  - Tools : GDB, JDB, Valgrind
- Makefiles
  - Target, dependency, and actions
  - Make -C dir <target>
- TDD
  - Build before write the code
  - JUnit
  - Failure and error: (tests fail & exception - did not include in ur test cases)
- Version control (Git)
  - Distributed model
  - Repository
  - Github : public (is your portfolio)

- XML
    - Hierarchical, human readable format; sibling to html;
    - encodes doc/data
    - Always parsable

- General command format: **NCM**
    - N: optional multiplier number
    - C: the command
    - M: optional scalar modifier
    - Eg: 3dw (delete three words)

- Unix relationship
    - 


- **Everything is a file or process.**

- PATHs:
    - Absolute: start at the root and follow the tree
    - Relative: start at working directory

- **Each shell opens three files automatically: stdin, stdout, stderr.**
- **Each shell opens automatically: .bashrc .profile**

- IDE: integrated development environment
    - Pros:
        - Faster
        - Open source
        - Extensible
    - Cons:
        - Learning curve
        - Requires JRE (java runtime environment)
        - Pretty heavyweight

- **Redirection**

- > overwrites a file
- >> appends to a file

- **Debugging (debugging is NOT algorithmic)**
  - Steps
    1. Understand the system
    2. Identify the problem
    3. Reproduce the problem
    4. Diagnose the cause of the problem
    5. Fix the problem
    6. Reflect and learn from the problem
  - Reproduce the problem with minimal input
  - Methods
    - First, guess the problem
    - Second, conduct an experiment
      - Make one change at one time

- **Pipe**
  - two or more commands separated by pipe char '|'. That tells the shell to arrange for the output of the preceding command to be passed as input to the following command. Example : **ls -l | pr**
    - The output for a command ls is the standard input of pr. When a sequence of commands are combined using pipe, then it is called pipeline

- **Computer can be divided into 3 parts: CPU (central processing unit), memory, input/output subsystem.**
- **Grep and tail command:**
  Grep “pattern” + filename
  Tail + filename: display the last part of the file
  Wc: count line, word and characters
- Debug tools: JDB, GDB, Valgrind.

- **Shell script**
  - Start with **#!/bin/bash**
  - Need to chmod and make it executable
  - Execute:
    - ./script.sh
    - . script.sh
    - source script.sh
    - sh script.sh
    - 这几个好像有区别 有些是在current process里面run有些事其他的 不懂

    - 有些不需要chmod permission。sh就不需要; source run as current process, while ./ run as a new process

- **Chmod command**
    - Change the permission (read[r], write[w], execute[x]) of files for three categories of users (owner[o], group[g], others[o])
    - Apply the chmod command to all files in a directory with **-R**
    - Read permission for directory: ls can read the list of filenames in the directory
    - Write permission: can move or create or delete files
    - Execute permission: can pass through the directory in searching for subdirectories.
- **Quote**
    - double: weak.
    - single: strong. Everything inside single quotes is taken literally.
    - back: treat as a unix command.
        - Eg:
            - *world=Earth*
            - *foo='hello $world'*
            - *bar="hello $world"*
            - *echo $foo -> hello $world*
            - *echo $bar -> hello Earth*
            - *echo '$bar' -> $bar*

    File Inquiry:

## File inquiry operations

| | |
|---|---|
| -d file | Test if file is a directory |
| -f file | Test if file is not a directory |
| -s file | Test if the file has non zero length |
| -r file | Test if the file is readable |
| -w file | Test if the file is writable |
| -x file | Test if the file is executable |
| -o file | Test if the file is owned by the user |
| -e file | Test if the file exists |
| -z file | Test if the file has zero length |

All these conditions return true if satisfied and false otherwise.

- **TDD (Test-Driven Development)**
    - Pros:
        - Increase development speed
        - Increase your confidence and reliability of your code
        - Help you ensure that your program is correct
    - Cons:

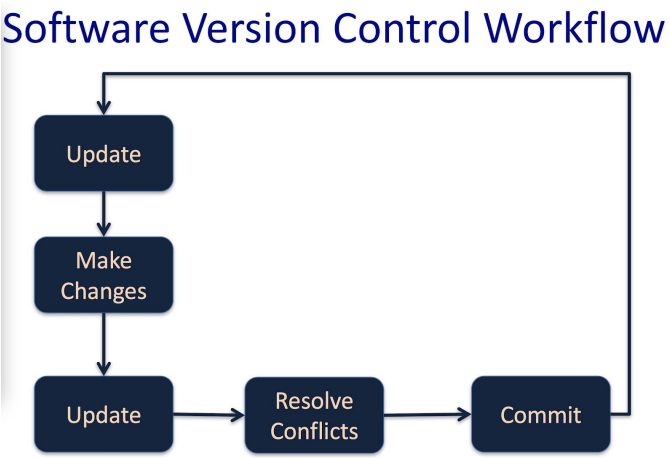
    - Doesn't test the whole program
    - Same developer coding and testing
    - time-consuming
- **Write the test before writing your program**
- **Regression testing**
    - Everytime change your code, run original test first. To make sure old features work after adding new features.
- **Unit testing**
    - Test one method at a time
    - Junit testing
        - Framework for testing of java software
    - All have:
        - Test runner
        - Test cases
        - Test suites
        - Test execution
    - Strategies
        - Identify and prioritize testing of
            - Core functionality
            - Corner cases
            - Special input cases
            - Commonly used functionality
        - Test related functionality as test suites
        - Test both positive and negative paths
- Failure and error:
    - Fail: assertion is incorrect
    - Error: unexpected exceptions
    - Cons of JUNIT: Not every method has return value, etc.

- **Shell programming**
    - Variables:
        - Global: HOME, PATH, USERNAME, PWD…
        - Local: user created (no space around "=")
        - Special: $1, $2, $@...

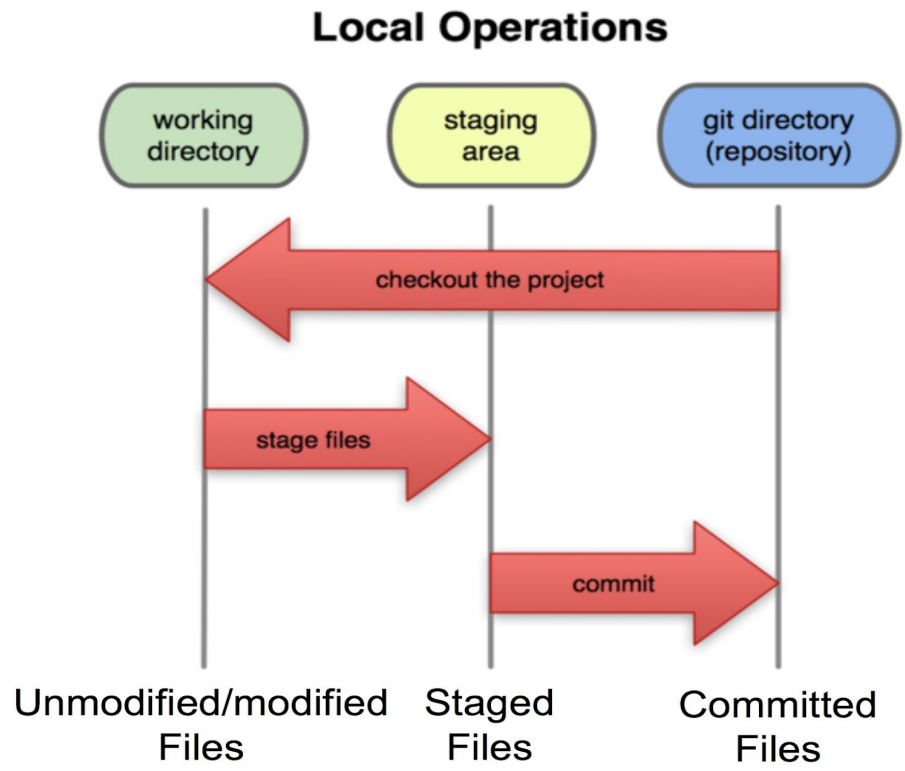
$0 This variable that contains the name of the script
$1, $2, ….. $9 1st, 2nd 3rd command line parameter
$# Number of command line parameters
$$ process ID of the shell
$@ same as $* but as a list one at a time
$? Return code 'exit code' of the last command
`Shift` command: This shell command shifts the positional parameters by one towards the beginning and drops $1 from the list. After a shift $2 becomes $1 , and so on … It is a useful command for processing the input parameters one at a time.

- To see environment variables: **printenv**
- Referencing variables
    - $var or ${var} (with bracket much safer)
- Lists:
    - a=(1 2 3)
    - echo ${a[1]} -> 2
    - echo ${a[*]} -> 1 2 3
- Operators:
    - Define a variable
        - set x=37; echo $x -> 37 (**csh**)
        - unset x; echo $x -> undefined variable (**csh**)
    - Arithmetic
        - count=5
        - count=`expr $count + 1`
        - echo $count -> 6
- Logic structures
    - Conditional
        - if [ … ]; then
        - elif [ … ]; then
        - else
        - fi
    - Loops
        - for arg in **list (eg: for i in {1..10} )**
            - **do**
            - **…**
            - **Done**
        - **while test** … (while [ … ])
            - do
            - ...
            - Done
        - until test …
            - do
            - …
            - Done
    - Switch logic
        - case arg in
            - Pattern 1) … ;;
            - Pattern 2) … ;;
        - esac
    - Functions
        - functionname(){
            …
          }

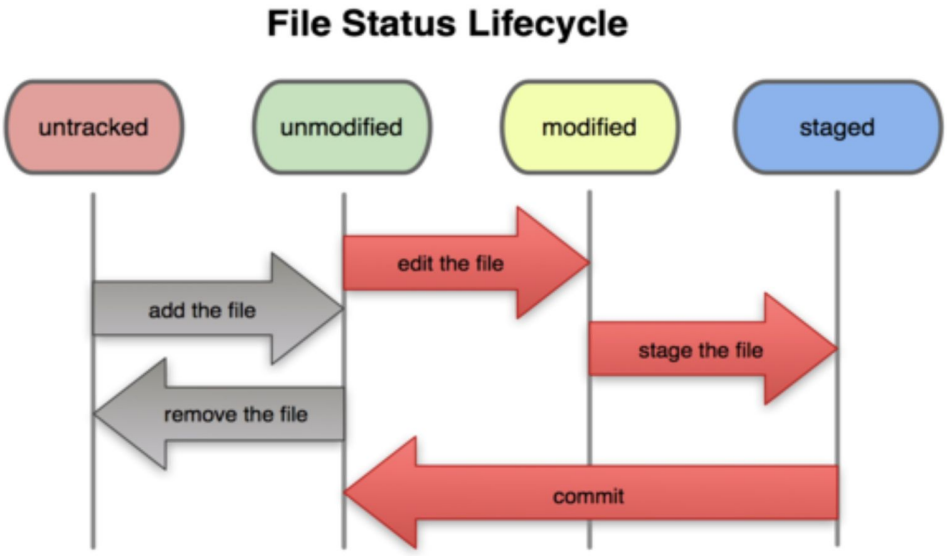
- **Makefile**
  - If a task needs doing repeatedly, you could write a program to do it.
  - Tab at the beginning of every action
  - **To make the target, make the dependencies first, then perform all the actions**
  - **.PHONY: always out of date**
  - Marcos:
    - CLASSES = a.class b.class. C.class
    - all: $(CLASSES)
  - **Call make from subdirectories: (!!!!kan)**
    - make -C ../ target
  - .SUFFIX directive
    - .SUFFIXES: .java .class
      .java.class:
      Javac &< (the dependencies, whatever it is)

- **Version control**
  - Non-version control cons:
    - If one lost his laptop
    - If one wants to go back to previous version
    - Last one who saves wins
  - 


- **Git**
  - Goal:
    - Speed
    - Non-linear development
    - Fully distributed
    - Able to handle large projects efficiently
  - A local git project has 3 areas




-




-

- Create a local copy of a repository
  - Clone an existing repo to current directory
    - git clone <url> [local directory]
  - Create a git repo in current directory
    - git init
- Committing
  - Before committing a file we first need to add it to the staging area

        - git add file1
    - And then move the staged changes into the repo
        - git commit -m “Fixing bug#10”
- **Those commands just act on your local repo**
- status: view the status of your files in the working directory
    - git status
- diff: see what is modified but unstaged
    - git diff
- Pulling and pushing
    - Add and commit your changes to your local repo
    - pull: to pull from remote repo to get the most recent changes
        - **Two steps:**
            - **git fetch**
            - **git merge [ remote fetched repo]**
    - push: push your changes into the remote repo
    - branch : git branch experiment

- **Unix process**
    - Grandparent of all process: PID = 1
    - Kernel has PID = 0
    - Orphan process: when parent process dies, immediately adopted by the grandparent: the init process
    - **Daemon process:**
        - **Parent: init**
        - **Run as a background process without being controlled by an interactive user**
    - **Zombie process:**
        - **Completed execution but still has an entry in the process table**
    - How to kill
        - kill PID
        - ps -efl (make sure the process is dead)
        - kill -9 PID (kill it dead)
    - trap command
        - Execute a command when a signal is received by your script
- **XML and Ant**
    - XML: (**by default, XML file is named build.xml**)
        - Hierarchical, human-readable format
            - Blends data and schema (structure)
        - XML Anatomy
            - <title>**ABC**</title>
                - <> is open tag </> is closed tag. The whole thing is the **element**

        - <article mdate="2003-10-10" key="tr/abc">
            <editor>xxx</editor>
          </article>
            - mdate and key are **attributes**
    - Any well-formed XML is always parsable
    - Attributes only appear once in an element
    - **XML is case-sensitive**
- Ant: (Apache Ant) (Another Neat Tool)
    - Java based tool for automating the build process
    - Ant's build files are written in XML
    - Each build file contains
        - A project
        - At least 1 target
    - Comments: <!-- -- >
    - <u>Projects:</u>
        - Usually contains 3 attributes
            - Name
            - Default
            - Basedir

```
<project name="Sample Project" default="compile" basedir=".">

 <description>
  A sample build file for this project
 </description>

</project>
```

    - Build files may contain constants, known as **properties.** <u>To assign a value to a variable that can be used throughout the project.</u>
        - Consist of a pair of name/value
        - ${property name}

```
<project name="Sample Project" default="compile" basedir=".">

 <description>
  A sample build file for this project
 </description>

 <!-- global properties for this build file -->
 <property name="source.dir" location="src"/>
 <property name="build.dir" location="bin"/>
 <property name="doc.dir" location="doc"/>

</project>
```

    - **Targets tags** has the following required attributes
        - Name - logic name

- Optional
  - Depends
  - Descriptions
- A build file may additionally specify a default target

```
<project name="Sample Project" default="compile" basedir=".">

  ...

  <!-- set up some directories used by this project -->
  <target name="init" description="setup project directories">
  </target>

  <!-- Compile the java code in src dir into build dir -->
  <target name="compile" depends="init" description="compile java sources">
  </target>

  <!-- Generate javadocs for current project into docs dir -->
  <target name="doc" depends="init" description="generate documentation">
  </target>

  <!-- Delete the build & doc directories and Emacs backup (*~) files -->
  <target name="clean" description="tidy up the workspace">
  </target>

</project>
```

- **Tasks**
  - Represents an action that needs to be executed
  - Initialization target and task

```
<project name="Sample Project" default="compile" basedir=".">

  ...

  <!-- set up some directories used by this project -->
  <target name="init" description="setup project directories">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${doc.dir}"/>
  </target>

  ...

</project>
```

  - Compile target and task

```
<project name="Sample Project" default="compile" basedir=".">

  ...

  <!-- Compile the java code in ${src.dir} into ${build.dir} -->
  <target name="compile" depends="init" description="compile java sources">
    <javac srcdir="${source.dir}" destdir="${build.dir}"/>
  </target>

  ...

</project>
```

- javadoc target and task
- Generate HTML documentation for all sources

```
<project name="Sample Project" default="compile" basedir=".">

 ...

 <!-- Generate javadocs for current project into ${doc.dir} -->
 <target name="doc" depends="init" description="generate documentation">
  <javadoc sourcepath="${source.dir}" destdir="${doc.dir}"/>
 </target>

 ...

</project>
```

-
- Cleanup target and task
- Removes all files/directories from the system

```
<project name="Sample Project" default="compile" basedir=".">
 ...
 <!-- Delete the build & doc directories and Emacs backup (*~) files -->
 <target name="clean" description="tidy up the workspace">
  <delete dir="${build.dir}"/>
  <delete dir="${doc.dir}"/>
  <delete>
   <fileset defaultexcludes="no" dir="${source.dir}" includes="**/*~"/>
  </delete>
 </target>
 ...
</project>
```

-
- Running ant
  - cd to the directory with build file and type **ant** will run the default target (or type **ant** followed by the name of target)

```
<project name="Sample Project" default="compile" basedir=".">

 <description>
  A sample build file for this project
 </description>

 <!-- global properties for this build file -->
 <property name="source.dir" location="src"/>
 <property name="build.dir" location="bin"/>
 <property name="doc.dir" location="doc"/>

 <!-- set up some directories used by this project -->
 <target name="init" description="setup project directories">
  <mkdir dir="${build.dir}"/>
  <mkdir dir="${doc.dir}"/>
 </target>

 <!-- Compile the java code in ${src.dir} into ${build.dir} -->
 <target name="compile" depends="init" description="compile java sources">
  <javac srcdir="${source.dir}" destdir="${build.dir}"/>
 </target>
```

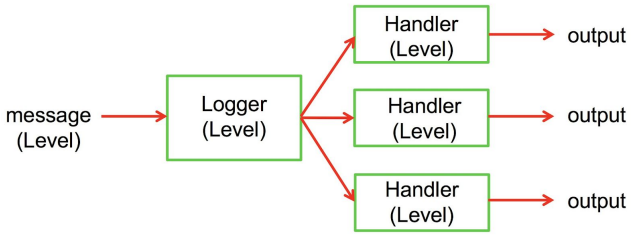
```
  <!-- Generate javadocs for current project into ${doc.dir} -->
  <target name="doc" depends="init" description="generate documentation">
    <javadoc sourcepath="${source.dir}" destdir="${doc.dir}"/>
  </target>

  <!-- Delete the build & doc directories and Emacs backup (*~) files -->
  <target name="clean" description="tidy up the workspace">
    <delete dir="${build.dir}"/>
    <delete dir="${doc.dir}"/>
    <delete>
      <fileset defaultexcludes="no" dir="${source.dir}" includes="**/*~"/>
    </delete>
  </target>

</project>
```

- **Diagnose the output**
  - Need
    - Tracing
    - Timing
    - Profiling
    - Logging
    - Error reporting
  - Tools
    - Standard output and error
    - Debuggers
    - …

- **Loggings**
  - Logging framework or API
  - java.util.logging
    - Important classes
      - Logger
      - Handler
      - Formatter
      - Level
    - **Level**
      - Specify the importance level of log information
        - Level.SEVERE
        - Level.WARNING
        - Level.INFO
        - Level.CONFIG
        - Level.FINE
        - Level.FINER
        - Level.FINEST
      - When logging a message, a level must be specified
      - Each logger and Handler has a log level, log messages lower than that level will be ignored.
        - eg: level.ALL / level.OFF
          Log/ignore all messages
      - **A message will not be logged if the Logger's level or Handler's level is higher than the message's level**
    - **Handler**
      - Each logger object must have **one or more** handler object associate with it
      - ConsoleHandler: log to stderr, has by default
      - FileHandler: log to file
      - SocketHandler: connect to a logging server
    - **Formatter**
      - Each handler must have a Formatter object to use to format logging messages
      - **SimpleFormatter: default of ConsoleHandler**
      - XMLFormatter: default for File and SocketHandler
  - If you want to change the logging level, you have to edit the program and recompile. **Easier and more powerful is using a properties file (recompile not necessary)**




- **Controlling logging (what if logging too much)**
  - Logging at a low level
  - In production code, only high level messages should be logged
  - To improve the efficiency, or lower the cost
    - **Guarded logger**
      - **eg: if (logger.isLoggable(Level.FINE))**
        **logger.fine("x = " + x);**
      - This reduces runtime cost
      - Better:
        - Global boolean
        - **eg: if (DOLOGGING)**
          **logger.fine("x = " + x);**
        - **Set to false and recompile**
    - To decrease the size of .class files
      - Soln: using **final** static boolean
      - Unnecessary codes don't get compiled

- **Efficiency**
  - Runtime cost: functions of the size N of the problem the software is given to solve
    - Time, space, energy
  - Measuring:
    - Profiling frameworks
    - Unix time command
      - time & usr/bin/time (first time is bash shell built-in)
      - **eg: time java app**
        **/usr/bin/time -v java app**

    *eg: time ls 2> out*

    *Redirect the error output of ls to "out" (which is nothing) and print out the "ls" and then time command will measure the time of the operation and print out the time it takes*

    - *eg: time ls > out*

    *Redirect the output of ls to "out" and time command will measure the time the operation takes and print out the time*

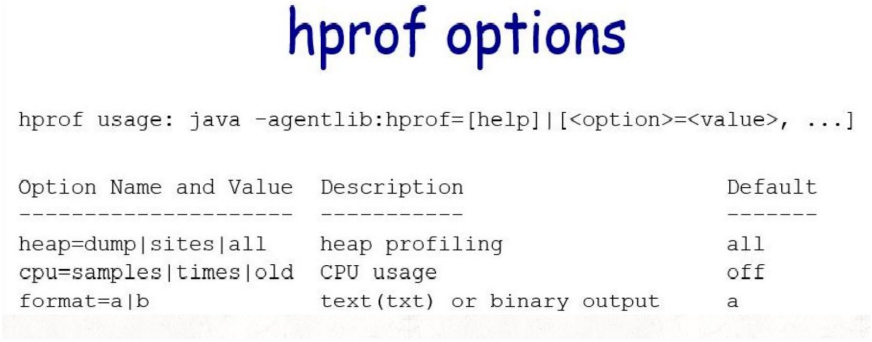
    - *eg: (time ls) 2> out*

    *Redirect the output of time command to "out" and print out "ls"*

    - **Basic procedure of measuring**

    - **Standard deviation;**
        - Small: close to the average
        - Large: far. Some corrupted by noise

- **Profiling using hprof**
    - Measuring cost functions: Profiling
    - hprof and JVMTI(tool interface)
        - **java -agentlib:hprof MyProg**




    -
        - **heap=dump: get a dump of all live objects in the heap**
        - **heap=sites: get a sorted list of sites with the most heavily allocated objects at the top**
        - **By default, all gives both output**
    - **Hprof cpu options:**
        - samples
            - Outputs the ranked estimate of time spent in methods based on this sampling
        - times
            - Injects bytecodes at start and end of every method to record time spent in each method. Outputs ranked result. Slow down the application significantly.
    - jhat: a tool to help browse the heap data
        - Use format=b option, and run jhat directing it to the data file:
            - jhat java.hprof

- **Reporting bugs**
    - A bug reporter should
        - Understand the system
        - Identify the problem
        - Reproduce the problem
    - Frameworks
        - Bugzilla (go to fucking slides)
        - Debbugs
        - MantisBT
    - Bug reporting guidelines

- Check if updated to the latest version
- Search if the bug has been reported and then get ready to report a new bug

To be tested：162 Points

1. Grep tail
2. Git distributed
3. Advanced makefile: make -C subdir target
4. Midterm has TF
5. Hprof options
6. 10 vim commands
    a. A, a, O, o, I, i, “.”, yy, dd, p, }, {, w, b, e, u, c, r, /, ?,0
7. Review slide（handler）