# 快捷工具

## myEclipse

ctrl+shift+t 查找类名

ctrl+1 强转对象

ctrl+t 查看继承结构

F2 改名，更改文件名

F3 进入目录

ctrl+shift+t 搜索原码

shift+alt+s source shift+r get\set方法

ctrl+方向下 向下复制

ctrl+1 强制类型转换

Alt+Shift+L 生产返回值

Alt+方向下 当前行下移

Alt+方向上 当前行上移

# SQL语句

## case when

#### 判断一个表的字段是否为空，若不为空则查另一个表的数据

select CONTACT,TEL,MOBILE,ADDR,ADDR\_NAME,IS\_DEFAULT,retion\_id,

(case when retion\_id is not null then

(select ao.administrative\_organ\_name from CUS\_ADDR a,

SYS\_ADMINISTRATIVE\_ORGAN ao WHERE ao.ADMINISTRATIVE\_ORGAN\_ID=a.RETION\_ID AND a.ADDR\_ID=14174709)

end) AS administrative\_organ\_name

FROM CUS\_ADDR ca WHERE ADDR\_ID=50001870401;

#### 创建序列

create sequence SEQ\_SELF\_SM\_TRANSFORMLOG

minvalue 1

maxvalue 9999999999999999999999999999

start with 1

increment by 1

cache 20;

# JAVA工具类

## MapUtils

#### 判断时间到期

**int** expire = MapUtils.*getIntValue*(listMap, "charge");

String endDate = "";

**if**(expire > 0) {

**if**(expire > 30) {

endDate = "服务未到期！";

}**else** {

endDate = "服务将到期！";

}

}**else** {

endDate = "服务已到期！";

}

listMap.put("endDate",endDate);

=============================

# Git

### 下载安装

安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！安装成功后，还需要最后一步设置，在命令行输入：

git config --global user.name “your name”

git config --global user.email [“email@example.com”](mailto:\“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门：你的名字和Email地址。你也许会担心，如果有人故意冒充别人怎么办？这个不必担心，首先我们相信大家都是善良无知的群众，其次，真的有冒充的也是有办法可查的。

注意git config命令的--global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

### 创建版本库

什么是版本库呢？版本库又名仓库，英文名repository，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

所以，创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：

cd f: //进入F盘

mkdir GitManager //创建文件夹GitManager 目录名包括父目录，最好不要包括中文

cd GitManager //进入文件夹GitManager

pwd //显示当前目录

然后通过 git init命令把这个目录变成Git可以管理的仓库：

git init //把当前目录交给Git管理

JandMin@jiangdm MINGW64 /f/GitManager

$ git init

Initialized empty Git repository in F:/GitManager/.git/

当前目录下多了一个.git的目录，这个目录是Git来跟踪管理版本库的，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了

### 把文件添加到版本库

首先这里再明确一下，所有的版本控制系统，其实只能跟踪文本文件的改动，比如TXT文件，网页，所有的程序代码等等，Git也不例外。版本控制系统可以告诉你每次的改动，比如在第5行加了一个单词“Linux”，在第8行删了一个单词“Windows”。而图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100KB改成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

不幸的是，Microsoft的Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的，前面我们举的例子只是为了演示，如果要真正使用版本控制系统，就要以纯文本方式编写文件。

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++](http://notepad-plus-plus.org/" \t "http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)代替记事本，不但功能强大，而且免费！记得把Notepad++的默认编码设置为UTF-8 without BOM即可。

一定要放到GitManager目录下(子目录也行)，因为这是一个Git仓库，放到其他地方Git再厉害也找不到这个文件。把一个文件房间Git仓库只需要两步：

第一步，用命令git add 告诉Git,把文件添加到仓库：

$ git add coding\_and\_debugging.txt

第二步，用命令git commit告诉Git，把文件提交到仓库：

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git commit -m "wrote a note first time"

[master (root-commit) 025a992] wrote a note first time

1 file changed, 189 insertions(+)

create mode 100644 coding\_and\_debugging.txt

git commit命令，-m后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录

为什么Git添加文件需要add，commit一共两步呢？因为commit可以一次提交很多文件，所以你可以多次add不同的文件。

### 时光穿梭机

我们成功添加并提交了文件coding\_and\_debugging.txt，现在可以修改一下文件的内容，然后执行一下git status命令，git status命令可以让我们时刻掌握仓库当前的状态，上面的命令告诉我们，readme.txt被修改过了，但还没有准备提交的修改。

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: coding\_and\_debugging.txt

no changes added to commit (use "git add" and/or "git commit -a")

虽然Git告诉我们coding\_and\_debugging.txt被修改了，但如果能看看具体修改了什么内容，自然是很好的。比如你休假两周从国外回来，第一天上班时，已经记不清上次怎么修改的coding\_and\_debugging.txt，所以，需要用git diff这个命令看看：

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git diff coding\_and\_debugging.txt

**diff --git a/coding\_and\_debugging.txt b/coding\_and\_debugging.txt**

**index a469546..e0e4b5c 100644**

**--- a/coding\_and\_debugging.txt**

**+++ b/coding\_and\_debugging.txt**

@@ -157,7 +157,37 @@ SQL:sum 和 case when

(SM.SERVICE\_TYPE\_ID = 1 or SM.SERVICE\_TYPE\_ID = 4 or (SM.SERVICE\_TYPE\_ID=2 and (SM.PRODUCT\_ID = 8 or SM.PRODUCT\_ID = 729)));

=================================================================

-

+/\*\*

+ \* 派工 Controller

+ \*

+ \*/

+@Controller

+@RequestMapping(value="/crmapp/allow/smDispatchCtl",produces="text/plain;charset=utf-8")

+public class SmDispatchController extends BaseController{

+ @Autowired

+ private SmDispatchService smDispatchService;

:

git diff顾名思义就是查看difference，显示的格式正是Unix通用的diff格式，可以从上面的命令输出看到，我们在第一行添加了一个“distributed”单词。

知道了对readme.txt作了什么修改后，再把它提交到仓库就放心多了，提交修改和提交新文件是一样的两步，第一步是git add：

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git add coding\_and\_debugging.txt

同样没有任何输出。在执行第二步git commit之前，我们再运行git status看看当前仓库的状态：

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: coding\_and\_debugging.txt

git status告诉我们，将要被提交的修改包括readme.txt，下一步，就可以放心地提交了：

$ git commit -m "add controller"

[master 443cbac] add controller

1 file changed, 31 insertions(+), 1 deletion(-)

提交后，我们再用git status命令看看仓库的当前状态：

$ git status

On branch master

nothing to commit, working tree clean

Git告诉我们当前没有需要提交的修改，而且，工作目录是干净（working directory clean）的。

#### 版本退回

现在，已经学会了修改文件，然后把修改提交到Git版本库，现在，再练习一次。

像这样，你不断对文件进行修改，然后不断提交修改到版本库里，就好比玩RPG游戏时，每通过一关就会自动把游戏状态存盘，如果某一关没过去，你还可以选择读取前一关的状态。有些时候，在打Boss之前，你会手动存盘，以便万一打Boss失败了，可以从最近的地方重新开始。Git也是一样，每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在Git中被称为commit。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

版本控制系统肯定有某个命令可以告诉我们历史记录，在Git中，我们用git log命令查看：

$ git log

commit 969f3e4e3f961cf26f186755370482c584a99b18

Author: JandMin <jiang\_dong\_ming@163.com>

Date: Mon Feb 13 14:51:27 2017 +0800

add javaBean

commit 443cbac234ff01fa0c291e2694fa342e3acf16e5

Author: JandMin <jiang\_dong\_ming@163.com>

Date: Mon Feb 13 14:45:23 2017 +0800

add controller

commit 025a99263f9a872461ede557b42766b16347f970

Author: JandMin <jiang\_dong\_ming@163.com>

Date: Mon Feb 13 13:31:24 2017 +0800

wrote a note first time

如果嫌输出信息太多，看的眼花缭乱的话，可以试试加上 --pretty=oneline参数：

$ git log --pretty=oneline

969f3e4e3f961cf26f186755370482c584a99b18 add javaBean

443cbac234ff01fa0c291e2694fa342e3acf16e5 add controller

025a99263f9a872461ede557b42766b16347f970 wrote a note first time

需要友情提示的是，你看到的一大串类似3628164...882e1e0的是commit id（版本号），和SVN不一样，Git的commit id不是1，2，3……递增的数字，而是一个SHA1计算出来的一个非常大的数字，用十六进制表示，而且你看到的commit id和我的肯定不一样，以你自己的为准。为什么commit id需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

现在我们启动时光穿梭机，准备把coding\_and\_debugging.txt回退到上一个版本，也就是“add controller”的那个版本，怎么做呢？

首先，Git必须知道当前版本是哪个版本，在Git中，用HEAD表示当前版本，也就是最新的提交969f3e4e...4a99b18（注意我的提交ID和你的肯定不一样），上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

现在，我们要把当前版本“add javaBean”回退到上一个版本“add controller”，就可以使用git reset命令：

$ git reset --hard HEAD^

HEAD is now at 443cbac add controller

还可以继续回退到上一个版本wrote a note first time，不过且慢，然我们用git log再看看现在版本库的状态：

$ git log

commit 443cbac234ff01fa0c291e2694fa342e3acf16e5

Author: JandMin <jiang\_dong\_ming@163.com>

Date: Mon Feb 13 14:45:23 2017 +0800

add controller

commit 025a99263f9a872461ede557b42766b16347f970

Author: JandMin <jiang\_dong\_ming@163.com>

Date: Mon Feb 13 13:31:24 2017 +0800

wrote a note first time

想再回去最新的版本已经回不去了，肿么办？

办法其实还是有的，只要上面的命令行窗口还没有被关掉，你就可以顺着往上找啊找啊，找到那个add javaBean的commit id是969f3e4e3f...，于是就可以指定回到未来的某个版本：

$ git reset --hard 969f3e4e3f

HEAD is now at 969f3e4 add javaBean

版本号没必要写全，前几位就可以了，Git会自动去找。当然也不能只写前一两位，因为Git可能会找到多个版本号，就无法确定是哪一个了。

再小心翼翼地看看readme.txt的内容：

$ cat coding\_and\_debugging.txt

=====================================================================

获取时间：本周、本月、上个月、前三个月 Calendar

。。。。。。 。。。。。。

javaBean的操作：

可以用VO的方式，添加数据库中没有，但页面需要的现实的字段。例如根据服务方式Id来现实服务名称。

==================================================================

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git log

commit 969f3e4e3f961cf26f186755370482c584a99b18

Author: JandMin <jiang\_dong\_ming@163.com>

Date: Mon Feb 13 14:51:27 2017 +0800

add javaBean

commit 443cbac234ff01fa0c291e2694fa342e3acf16e5

Author: JandMin <jiang\_dong\_ming@163.com>

Date: Mon Feb 13 14:45:23 2017 +0800

add controller

commit 025a99263f9a872461ede557b42766b16347f970

Author: JandMin <jiang\_dong\_ming@163.com>

Date: Mon Feb 13 13:31:24 2017 +0800

wrote a note first time

现在，你回退到了某个版本，关掉了电脑，第二天早上就后悔了，想恢复到新版本怎么办？找不到新版本的commit id怎么办？

在Git中，总是有后悔药可以吃的。当你用$ git reset --hard HEAD^回退到add controller版本时，再想恢复到add javaBean，就必须找到add javaBean的commit id。Git提供了一个命令git reflog用来记录你的每一次命令：

$ git reflog

969f3e4 HEAD@{0}: reset: moving to 969f3e4e3f

443cbac HEAD@{1}: reset: moving to HEAD^

969f3e4 HEAD@{2}: commit: add javaBean

443cbac HEAD@{3}: commit: add controller

025a992 HEAD@{4}: commit (initial): wrote a note first time

#### 工作区和暂存区

Git和其它版本控制系统如SVN的一个不同之处就是有暂存区的概念。

**工作区（Working Directory）**

就是你再电脑里能看到的目录，比如我的GitManager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西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分之master,以及指向master的一个指针叫HEAD。



前面我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分之。

因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在git commit就是往master分支上提交更改。可以简单的理解为，需要提交的文件修改通通放到暂存区，然后一次性提交暂存区的所有修改。现在，我们再联系一遍，先对redme.txt做个修改，比如加上一行内容，然后，在工作区新增一个license文本文件（内容随便写）。

先用git status查看一下状态：

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: coding\_and\_debugging.txt

Untracked files:

(use "git add <file>..." to include in what will be committed)

license.txt

no changes added to commit (use "git add" and/or "git commit -a")

Git非常清楚地告诉我们，readme.txt被修改了，而LICENSE还从来没有被添加过，所以它的状态是Untracked。

现在，使用两次命令git add，把coding\_and\_debugging.txt和license.txt都添加后，用git status再查看一下：

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: coding\_and\_debugging.txt

new file: license.txt

现在，暂存区的状态就变成这样了：



所以，git add命令实际上就是把要提交的所有修改放到暂存区(Stage)，然后，执行git commit就可以一次性把暂存区的所有修改提交到分支。

$ git commit -m "understand how stage works"

[master f82179c] understand how stage works

2 files changed, 30 insertions(+)

create mode 100644 license.txt

一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

$ git status

On branch master

nothing to commit, working tree clean

现在版本库变成了这样，暂存区就没有任何内容了：



#### 管理修改

为什么Git比其他版本控制系统设计的优秀，因为Git跟踪并管理的是修改，而非文件。什么是修改？比如你新增一个行，这就是一个修改，删除了一行，也是一个修改，更改了某些字符，也是一个修改，删了一些又加了一些，也是一个修改，甚至创建一个新文件，也算是一个修改。

我们可以把文件修改一次，然后git add，再查看下状态，在把文件修改一次，再git commit，会发现第二次的修改没有被提交。操作过程：

第一次修改 -> git add -> 第二次修改 -> git commit

Git管理的是修改，当你用git add命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

提交后，用git diff HEAD -- license.txt命令可以查看工作区和版本库里面最新版本的区别：

$ git diff HEAD -- license.txt

**diff --git a/license.txt b/license.txt**

**index 6b069c6..0397a67 100644**

**--- a/license.txt**

**+++ b/license.txt**

@@ -1,4 +1,5 @@

aaa

bbbbbb

ccccccccc

-dddddddddddd

\ No newline at end of file

+dddddddddddd

+eeeeeeeeeeeeeeee

\ No newline at end of file

那么如何提交第二次修改呢？继续git add在git commit，也可以别着急提交第一次修改，先git add第二次修改，再git commit ，就相当于把两次修改合并后一块提交了。

#### 撤销修改

如果对文件做了修改，但是想回到之前的版本，可以手动删掉修改了的内容，如果不记得修改了哪些内容，就可以做撤销修改处理。git checkout --file可以丢弃工作区的修改：

$ git checkout -- license.txt

命令git checkout -- license.txt意思就是，把license.txt文件在工作区的修改全部撤销，这里有两种情况：一种是license.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；一种是license.txt已经添加到暂存区后，又做了修改，现在，撤销修改就是回到添加到暂存区后的状态。

总之，就是染跟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，如果没有-- 就变成了“切换到另一个分支”的命令。

如果把修改已经添加到了暂存区，也就是修改后执行了git add命令：

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git add license.txt

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: license.txt

Git同样告诉我们，用命令git reset HEAD file可以把暂存区的修改撤销掉(unstage),重新回到工作区：

$ git reset HEAD license.txt

Unstaged changes after reset:

M license.txt

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: license.txt

no changes added to commit (use "git add" and/or "git commit -a")

#### 删除文件

一般情况下，你通常直接在文件管理其中把没用的文件删除了，或者用rm命令删除了：

rm leaning.docx

这个时候，Git知道你删除了文件，因此，工作区和版本库就不一致了，git status命令会立刻告诉你哪些文件被删除了：

$ git status

On branch master

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: learning.docx

no changes added to commit (use "git add" and/or "git commit -a")

现在有两个选择，一时确实要从版本库中删除文件，那就用命令git rm删掉，并且 git commit：

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git rm learning.docx

rm 'learning.docx'

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git commit -m "learning.docx"

[master f7075c4] learning.docx

1 file changed, 0 insertions(+), 0 deletions(-)

delete mode 100644 learning.docx

git reset命令即可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。现在暂存区是干净的，工作区有修改，就可以在用git checkout -- file丢弃工作区的修改了：

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git checkout -- license.txt

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git status

On branch master

nothing to commit, working tree clean

整个世界终于清静了。

### 远程仓库

Git是分布式版本控制系统，同一个Git仓库，可以分布到不同的机器上。怎么分布呢？最早，肯定只有一台机器有一个原始的版本库，此后，别的机器可以“克隆”这个原始版本库，而且每台机器的版本库其实都是一样的，并没有主次之分。

完全可以自己搭建一台运行Git的服务期，不过现阶段，为了学Git先搭个服务器绝对是小题大作。好在这个世界上有个叫GitHub的神奇的网站，从名字就可以看出，这个网站就是提供Git仓库托管服务的，所以，只要注册一个GitHub账号，就可以免费会的Git远程仓库。

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：

**第一步：**创建SSH Key，在用户主目录下，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这两个文件，如果已经有了，可以直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash）,创建SSH Key:

ssh-keygen -t rsa -C [“jiang\_dong\_ming@163.com”](mailto:\“jiang_dong_ming@163.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，使用默认即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

$ ssh-keygen -t rsa -C "jiang\_dong\_ming@163.com"

Generating public/private rsa key pair.

Enter file in which to save the key (/c/Users/JandMin/.ssh/id\_rsa):

Created directory '/c/Users/JandMin/.ssh'.

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /c/Users/JandMin/.ssh/id\_rsa.

Your public key has been saved in /c/Users/JandMin/.ssh/id\_rsa.pub.

The key fingerprint is:

SHA256:uTlG78xwwJ25yaOJ5snvmM1VY+7x+2nSnbA55BJ3JT4 jiang\_dong\_ming@163.com

The key's randomart image is:

+---[RSA 2048]----+

| |

| |

| |

| . o o . .|

| S + +. ..|

| . \* B =E. |

| \* O B \*.o|

| ..O @ + O +o|

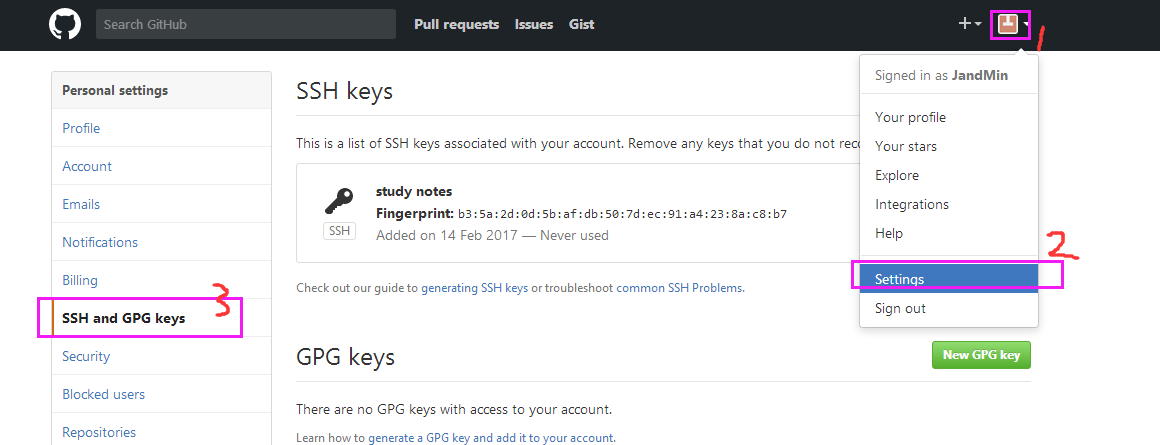
| oBoB + o \*+.|

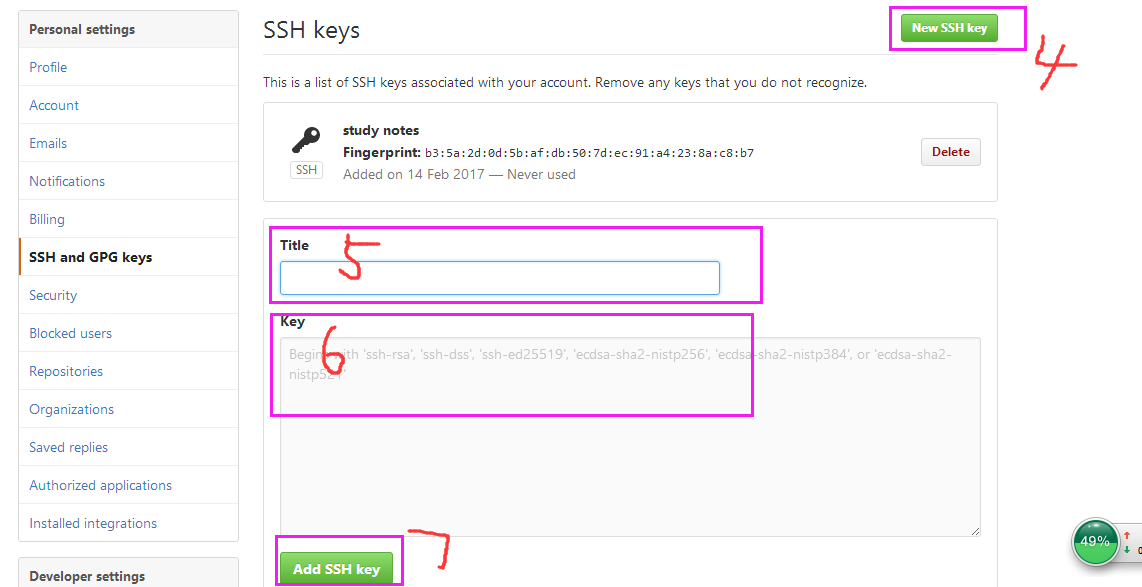
+----[SHA256]-----+

如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的密钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

第二步：登录GitHub,打开”Account settings”,”SSH Keys”页面：

然后，点击“Add SSH Key”,填上任意Title,在Key文本框里粘贴id\_rsa.pub文件的内容：



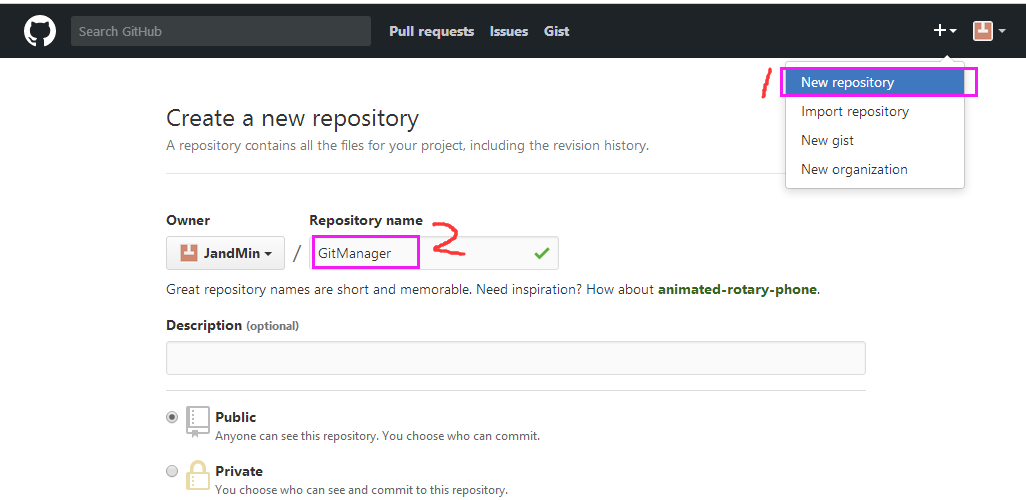


为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。最后友情提示，在GitHub上免费托管的Git仓库，任何人都可以看到喔（但只有你自己才能改）。所以，不要把敏感信息放进去。如果你不想让别人看到Git库，有两个办法，一个是交点保护费，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

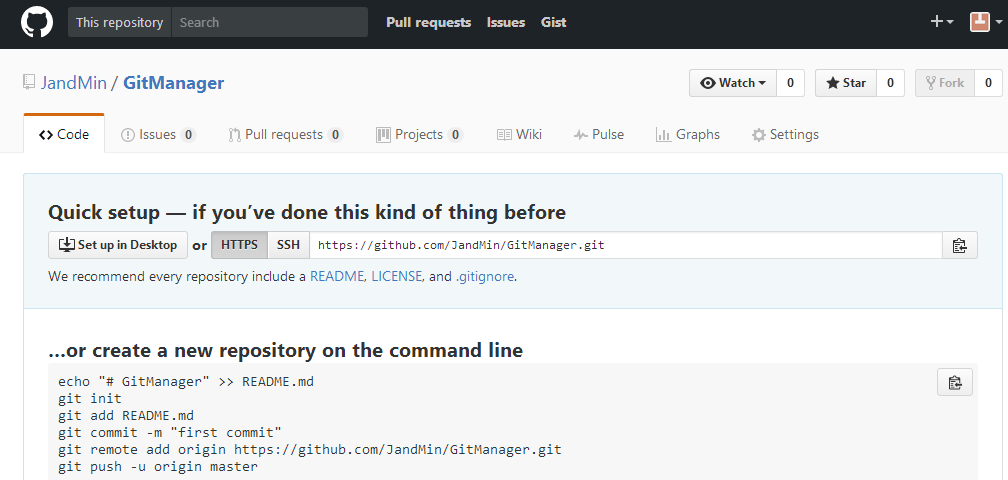
#### 添加远程仓库

现在的情景是，你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作，真是一举多得。

首先，登录GitHub，然后，在右上角找到“Create a new repp”按钮，创建一个新的仓库：



在Repository name填入GitManager，其他保持默认设置，点击底下的“Create repository”按钮，就成功地创建了一个新的Git仓库。



目前，在GitHub上的这个GitManager仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

现在，我们根据GitHub的提示，在本地的GitManager仓库下运行命令：

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ git remote add remote git@github.com:JandMin/GitManager.git

请千万注意。把上面的JandMin替换成你自己的GitHub账户名，否则，你再本地关联的就是我的远程库，关联没有问题，但是你以后是推不上去的，因为你的SSH Key公钥不在我的账户列表中。

添加后，远程库的名字就是remote,Git默认的叫法是origin,也可以改成别的，我就改成了remote.

下一步，就是把本地仓库的所有内容推送到远程库上：

$ git push -u remote master

The authenticity of host 'github.com (192.30.253.113)' can't be established.

RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'github.com,192.30.253.113' (RSA) to the list of known hosts.

Counting objects: 23, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (21/21), done.

Writing objects: 100% (23/23), 1.86 MiB | 13.00 KiB/s, done.

Total 23 (delta 6), reused 0 (delta 0)

remote: Resolving deltas: 100% (6/6), done.

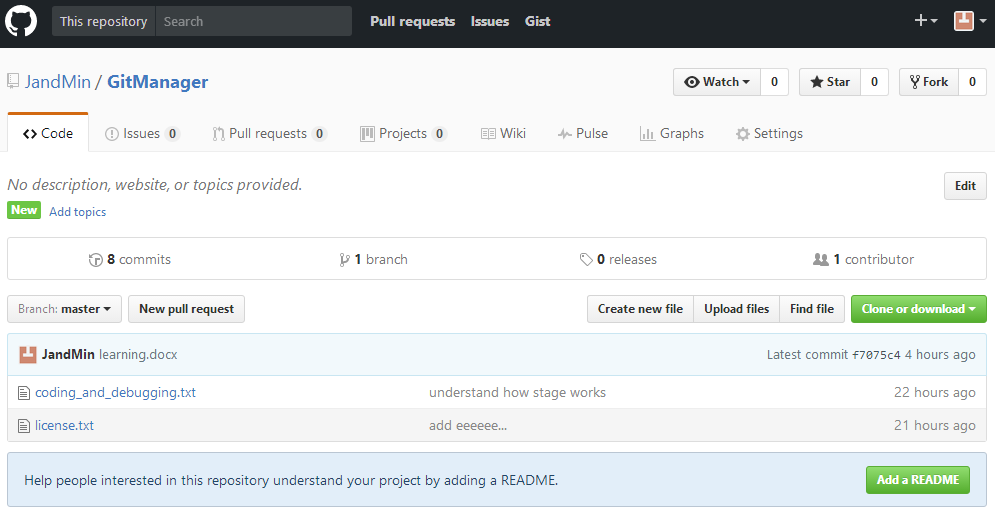
To github.com:JandMin/GitManager.git

\* [new branch] master -> master

Branch master set up to track remote branch master from remote.

把本地库的内容推送到远程，用git push命令，实际上是把当前分之master推送到远程。由于远程库是空的，我们第一次推送master分支时，加上了 -u 参数，Git 不但会把本地的master分支内容推送到远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一模一样：



从现在起，只要本地做了提交，就可以通过命令：

git push remote master

**SSH警告**

当你第一次使用Git的clone或者push命令链接GitHub时，会得到一个警告：

The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.

RSA key fingerprint is xx.xx.xx.xx.xx.

Are you sure you want to continue connecting (yes/no)?

这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入yes回车即可。Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

Warning: Permanently added 'github.com' (RSA) to the **list** of known hosts.

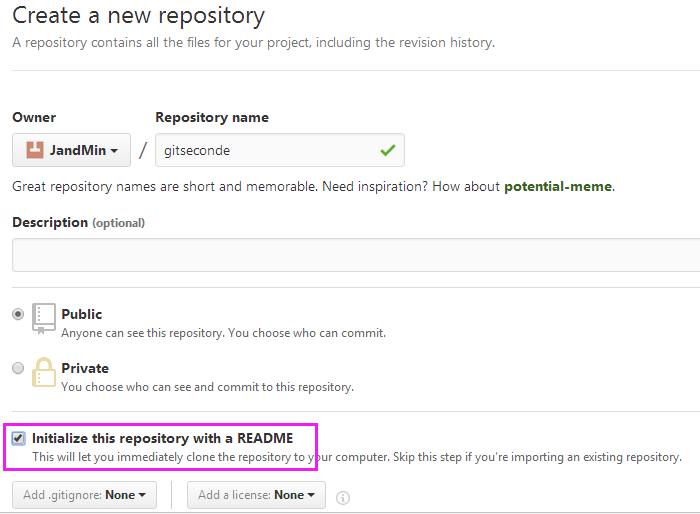
这个警告只会出现一次，后面的操作就不会有任何警告了。

如果你实在担心有人冒充GitHub服务器，输入yes前可以对照GitHub的RSA Key的指纹信息是否与SSH连接给出的一致。

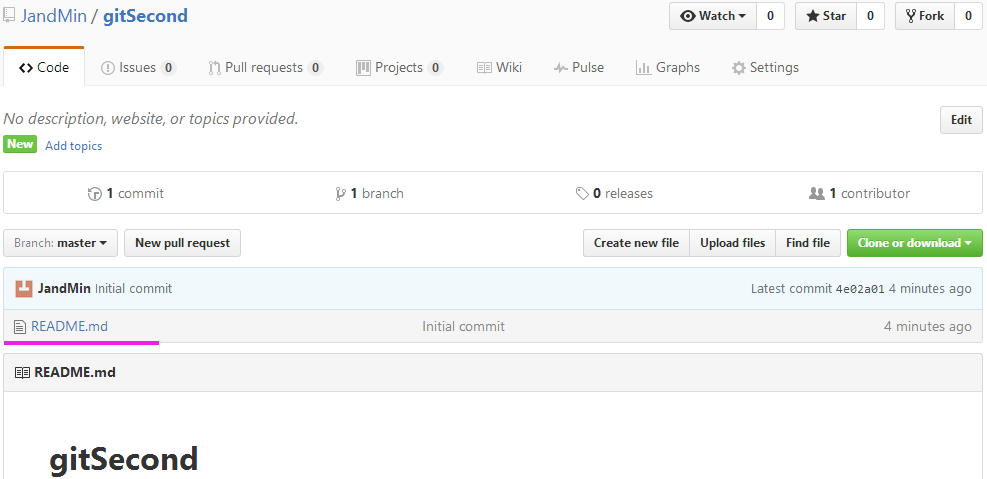
#### 从远程库克隆

现在，假设我们从零开发，那么最好的方式是先创建远程库，然后，从远程库克隆。

首先，登录GitHub,创建一个新的仓库，名字叫gitSecond:



我们勾选Initialize this repository with a README，这样GitHub会自动为我们创建一个README.md文件。创建完毕后，可以看到README.md文件：



现在，远程库已经准备好了，下一步是用命令git clone克隆一个本地库：

$ git clone git@github.com:JandMin/gitSecond.git

Cloning into 'gitSecond'...

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

Receiving objects: 100% (3/3), done.

JandMin@jiangdm MINGW64 /f/GitManager (master)

$ cd gitSecond

JandMin@jiangdm MINGW64 /f/GitManager/gitSecond (master)

$ ls

README.md

注意把Git库的地址换成你自己的，然后进入gitSecond目录看看，已经有README.md文件了。

如果有多个人协作开发，那么每个人各自从远程克隆一份就可以了。

GitHub给出的地址不止一个，还可以用https://github.com/JandMin/gitSecond.git这样的地址。实际上，Git支持多种协议，默认的是git://使用ssh，但也可以使用https等其他协议。

使用https除了速度慢以外，还有个最大的麻烦是每次推送都必须输入口令，但是在某些只开放http端口的公司内部就无法使用ssh协议而只能用https。

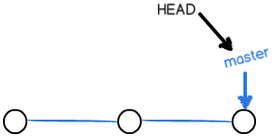
### 分支管理

分支就是科幻电影里面的平行宇宙，当你正在电脑前努力学习Git的时候，另一个你正在另一个平行宇宙里努力学习SVN。如果两个平行宇宙互不干扰，那对现在的你也没啥影响。不过，在某个时间点，两个平行宇宙合并了，结果，你既学会了Git又学会了SVN!

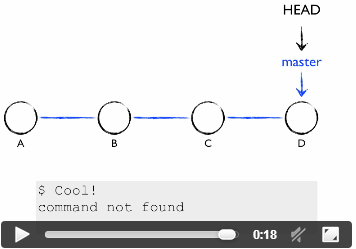
分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

#### 创建和合并分支

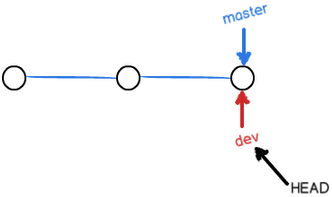
在版本回退里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支的提交点：



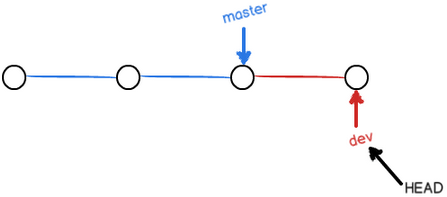
每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长：



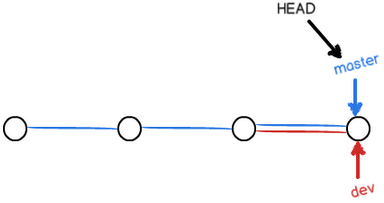
当我们创建新的分支，例如dev时，Git新建了一个指针叫做dev，指向master相同的提交，再把HEAD指向dev，就表示当前分在dev上：



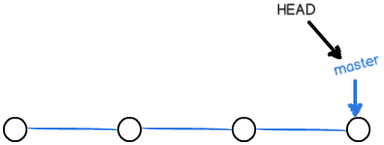
你看，Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！不过，从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变：



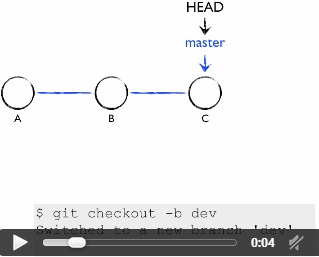
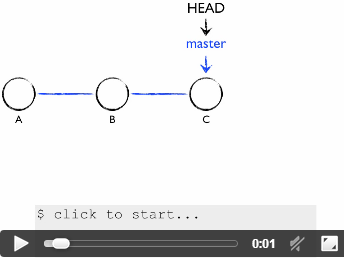
假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：

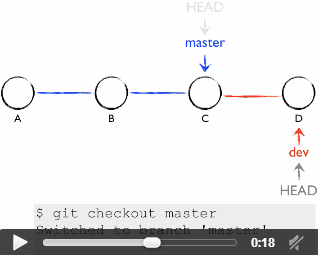
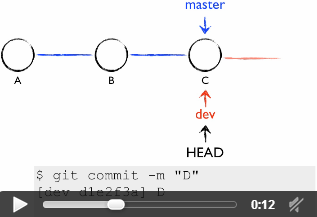


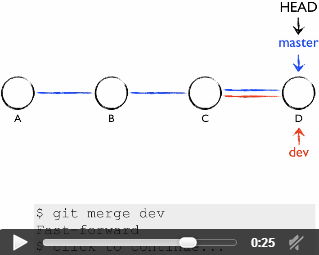
所以Git合并分支也是很快！就改改指针，工作区内容也不变！合并完分之后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下一条master分支：

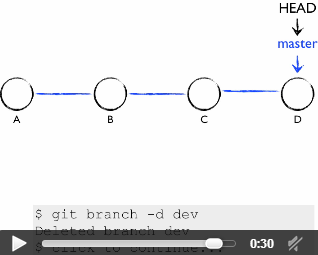


真是太神奇了，你看的出来有些提交是通过分支完成的吗？









下面开始实战。

首先，我们创建dev分支，然后切换到dev分支：

JandMin@jiangdm MINGW64 /f/gitSecond (master)

$ git checkout -b dev

M learning.docx

Switched to a new branch 'dev'

git checkout 命令加上-b参数表示创建并切换，当当与一下两条命令：

$ git branch dev

$ git checkout dev

Switched to branch 'dev'

然后，用git branch命令查看当前分支：

$ git branch

\* dev

master

git branch命令会列出所有分支，当前分支前面会标一个\*号。

然后，我们可以在dev分支上正常提交，比如对readme.mc做个修改，加上一行新内容，然后提交：

JandMin@jiangdm MINGW64 /f/gitSecond (dev)

$ git add README.md

JandMin@jiangdm MINGW64 /f/gitSecond (dev)

$ git add learning.docx

JandMin@jiangdm MINGW64 /f/gitSecond (dev)

$ git commit -m "create a new branch test"

[dev 74e9257] create a new branch test

2 files changed, 2 insertions(+), 1 deletion(-)

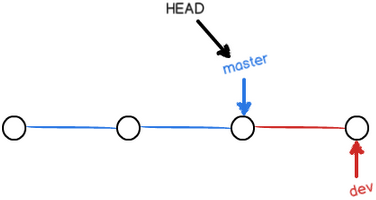
现在，dev分支的工作完成，我们就可以切换回master分支：

JandMin@jiangdm MINGW64 /f/gitSecond (dev)

$ git checkout master

Switched to branch 'master'

切换回master分之后，再查看一下README.md文件，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：



现在，我们把dev分支的工作成果合并到master分支上：

JandMin@jiangdm MINGW64 /f/gitSecond (dev)

$ git merge master

Updating 74e9257..874cc9c

Fast-forward

learning.docx | Bin 2214726 -> 2368397 bytes

1 file changed, 0 insertions(+), 0 deletions(-)

git merger命令用于合并指定分支到当前分支。合并后，再查看文件内容，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Fast-forward信息，Git告诉我们，这次合并是“快捷模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。当然，也不是每次合并都能fast-forward,我们后面会讲到其它方式的合并。

合并完成后，就可以放心地删除dev分支了：

JandMin@jiangdm MINGW64 /f/gitSecond (master)

$ git branch -d dev

Deleted branch dev (was 874cc9c).

删除后，查看branch，就只剩下master分支了：

$ git branch

\* master

因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删除掉分支，这和直接在master分支上工作效率是一样的，但过程更安全。

#### 解决冲突

#### 分支策略

#### Bug分支

#### Feature分支

#### 多人协作

### git命令集锦

cd f: //进入F盘

ls //查看所有文件

mkdir GitManager //创建文件夹GitManager 目录名包括父目录，最好不要包括中文

cd GitManager //进入文件夹GitManager

pwd //显示当前目录

git init //把当前目录交给Git管理

git add coding\_and\_debugging.txt //把文件添加到仓库

git commit -m "wrote a note first time" //把文件提交到仓库

git status //查看当前仓库文件状态

git diff coding\_and\_debugging.txt //查看不同，即查看修改了哪些地方

git log //查看历史版本记录，显示最近到最远的提交日志

git log --pretty=oneline //显示提交日志，只显示commit id和提交时的备注信息

git reset --hard commit\_id //返回上一个版本 HEAD当前版本，HEAD^上一个版本

cat coding\_and\_debugging.txt //查看文件具体内容

git reflog //查看历史命令，可以获得commit id

git diff HEAD -- license.txt //查看工作区和版本库里面最新版本的区别

git checkout -- license.txt //回到最近一次git commit或git add时的状态，即撤销修改

git reset HEAD license.txt //退回版本，把暂存区的修改回退到工作区

rm learning.docx //删除文件

git rm learning.docx //确定删除文件

git commit -m “remover learning.docx” //提交删除信息

ssh-keygen -t rsa -C [“jiang\_dong\_ming@163.com”](mailto:\“jiang_dong_ming@163.com\”) //创建SSH Key

git remote add origin [git@github.com:JandMin/GitManager.git](mailto:git@github.com:JandMin/GitManager.git) //与远程库建立连接

git remote add origin git@server-name:path/repo-name.git

git push -u origin master //第一次把本地库内容推送到远程库上

git push origin master //后期每次提交后把本地内容推送到远程库上

git clone [git@github.com:JandMin/gitSecond.git](mailto:git@github.com:JandMin/gitSecond.git) //从远程库克隆一个本地库

git checkout -b 分支名称 //创建分支，并切换到分支

git branch 分支名称 //创建分支

git checkout 分支名称 //切换到分支

git branch //查看当前分支

git merge 分支名称 //合并分支内容到当前分支上来

git branck -d 分支名称 //删除分支

=======================================

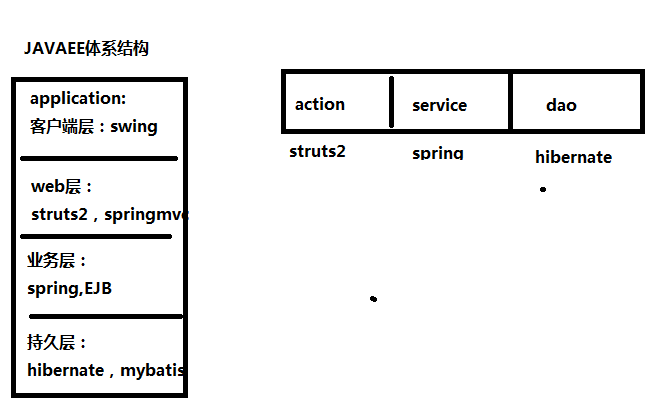
# MyBatis

========================================

# SpringMVC

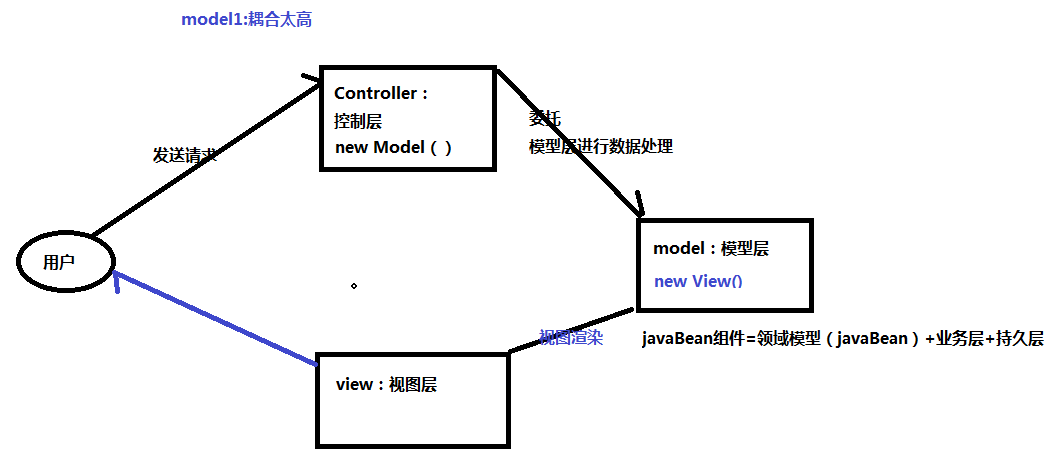
## springMvc基本框架

### JavaEE体系结构

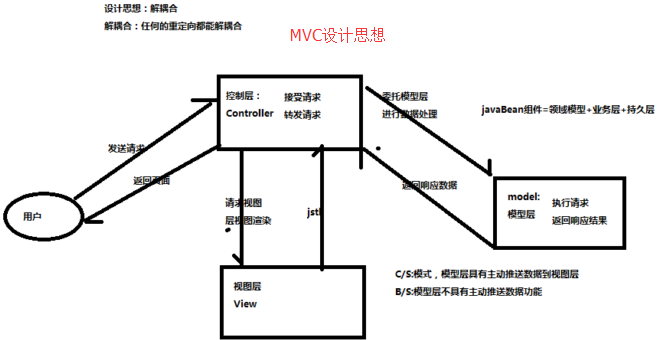


### 什么是mvc

* 原始开发model1



* mvc模式mpdel2



* Springmvc是什么

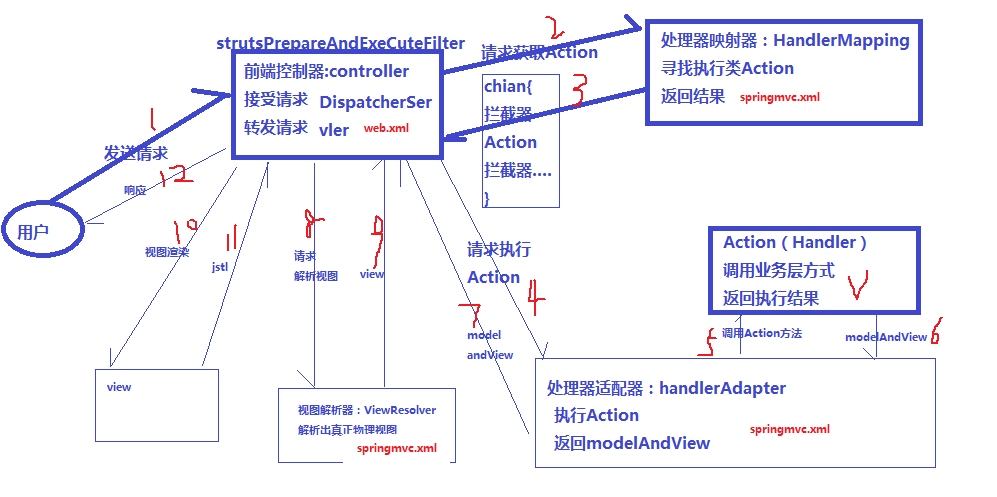
Springmvc是一个web层mvc框架。类似Struts2。

Struts2执行流程：

* strutsPrepareAndExcuteFilter拦截请求（控制层），拦截请求，转发请求
* 寻找Action执行
* ActionProxy：strutsActionProxy extends defaultActionProxy
* ActionMapping去寻找执行类Action

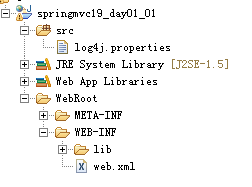
根据mvc设计模式：

### Springmvc执行流程

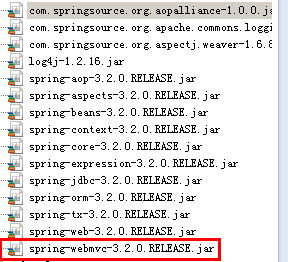


## Springmvc入门程序

### 创建一个web工程



### 导入jar包



### 配置web.xml：前端控制器：DispatcherServlet



### 配置springmvc.xml:

处理器映射器 HandlerMapping

配置处理器适配器 HandlerAdapter

配置自定义Controller

配置视图解析器

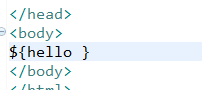
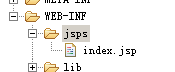


### 自定义Controller（struts2的action）

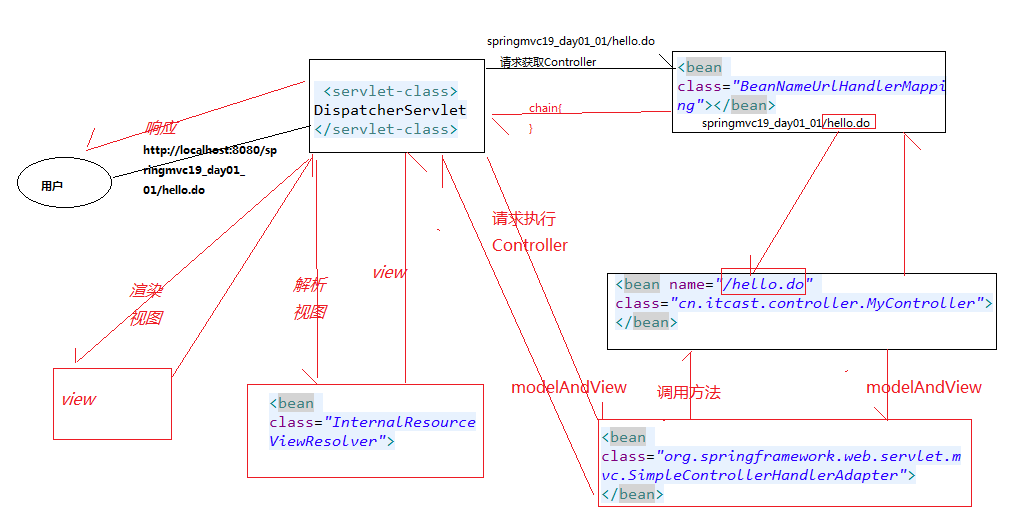


### 定义视图页面：jsps/index.jsp

根据视图解析路径：WEB-INF/jsps/index.jsp



### 根据代码分析springmvc执行流程

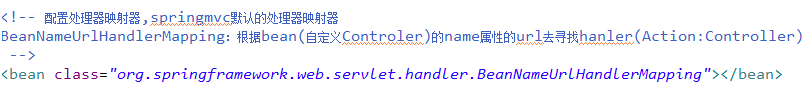


## 三个处理器映射器（3个可以共存）

### BeanNameUrlHandlerMapping

功能：寻找Controller

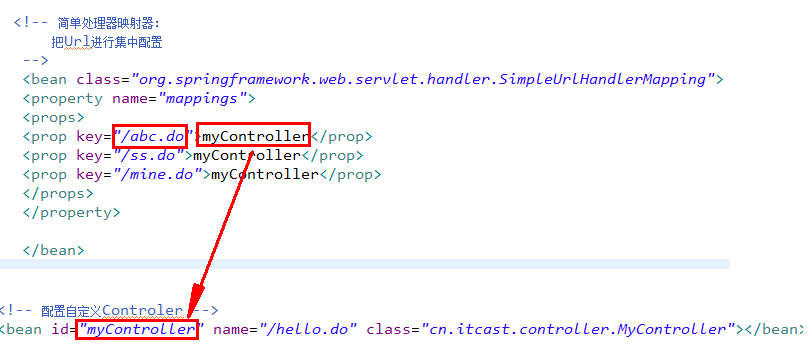
根据url请求去匹配bean的name属性url，从而获取Controller



### SimpleUrlHandlerMaping

功能：寻找Controller

根据浏览器url匹配简单url的key，key又Controller的id找到Controller



### ControllerClassNameHandlerMapping

功能：寻找Controller

根据类名（MyController）类名.do来访问,类名首字母小写



## 两个处理器适配器（2个处理器适配器可以共存）

### SimpleControllerHandlerAdapter

功能：执行controller

调用controller里面方法，返回modelAndView。



### HttpRequestHandlerAdapter

功能：执行controller

负责执行实现接口HttpRequstHandler接口的Controller



首先要自定义实现了接口HttpRequestHandler的controller的类



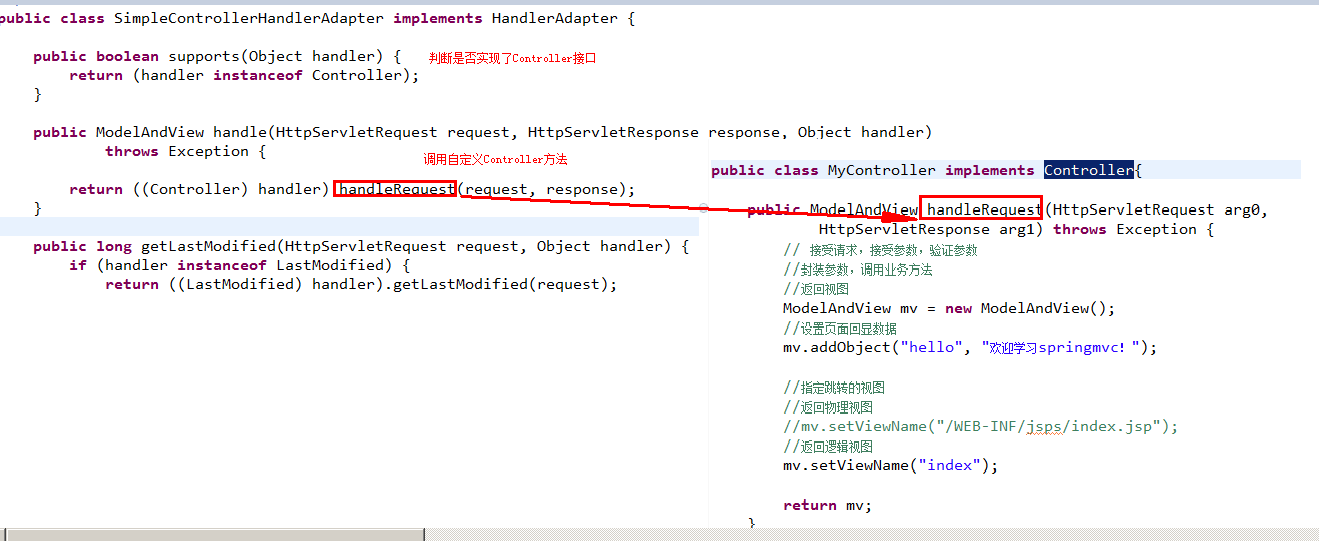
配置自定义bean



使用simpleUrlHandlerMapping来映射请求



### 适配器原码



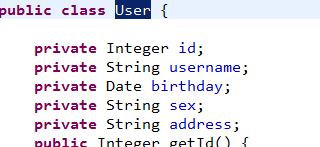
## 命令控制器

Springmvc通过命令设计模式接受页面参数。

### 自定义命令控制器

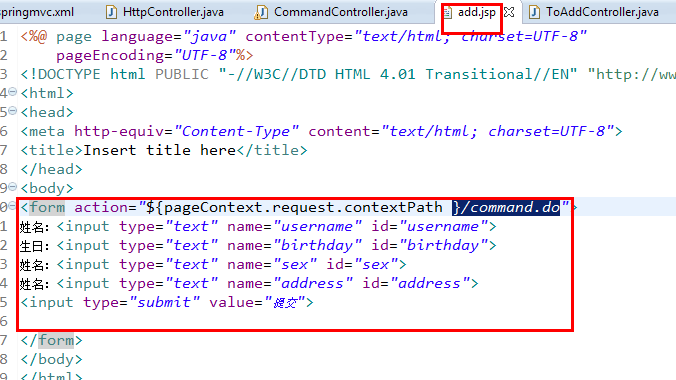


### 定义javabean



### 封装参数页面

add.jsp



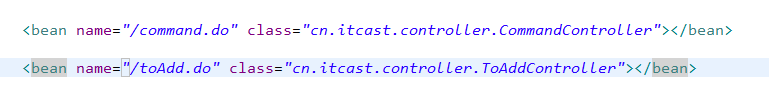
index.jsp

### 跳转到add页面

由于add页面在WEB-INF下面不能直接访问，需要通过Controller来访问。

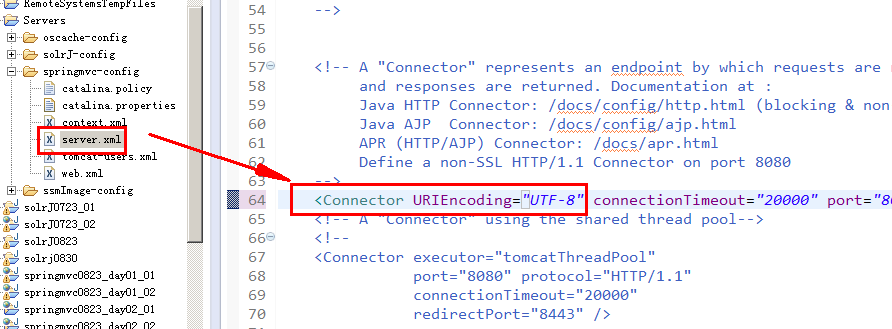


### 在springmvc中配置bean



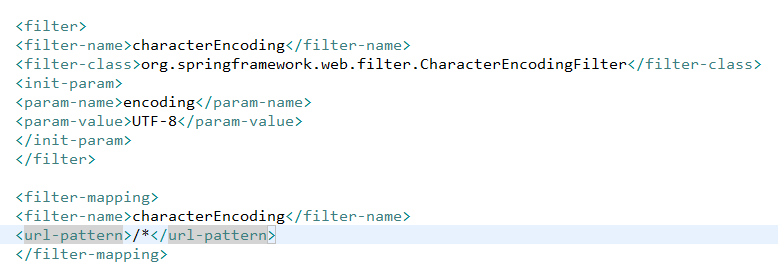
## 中文乱码解决

### GET请求乱码

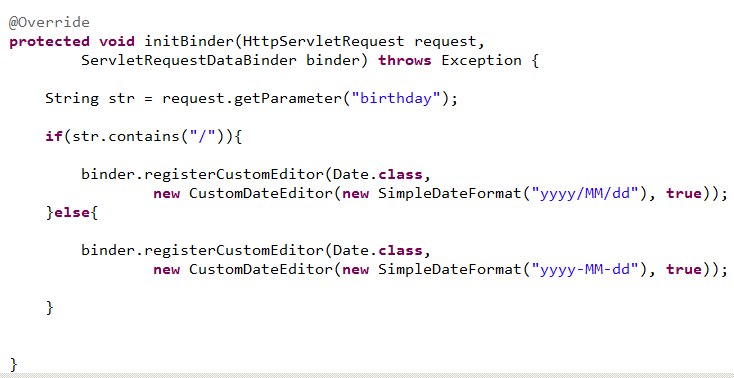


### POST请求乱码

Spring编码过滤器：在web.xml配置



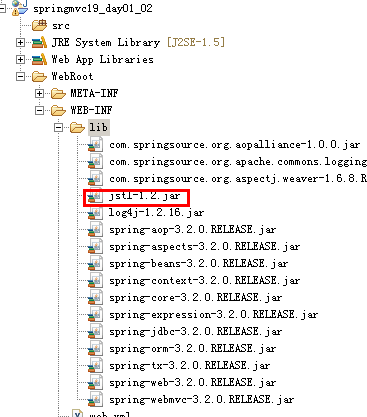
## 时间类型转换



## 注解开发

### 注解入门

创建一个web工程并导入jar包



配置web.xml项目的入口文件

<filter>

<filter-name>characterEncoding</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>characterEncoding</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!-- 默认加载方式

默认加载必须规范：

\* 文件命名：servlet-name-servlet.xml====springmvc-servlet.xml

\* 路径规范：必须在WEB-INF目录下面

-->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:springmvc.xml</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<url-pattern>\*.do</url-pattern>

</servlet-mapping>

修改springmvc.xml

<context:component-scan base-package=*"cn.itcast"*></context:component-scan>

<!-- 配置注解处理器映射器

功能：寻找执行类Controller

-->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"*></bean>

<!-- 配置注解处理器适配器

功能：调用controller方法，执行controller

-->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"*></bean>

<!-- 配置sprigmvc视图解析器：解析逻辑试图

后台返回逻辑试图：index

视图解析器解析出真正物理视图：前缀+逻辑试图+后缀====/WEB-INF/jsps/index.jsp

-->

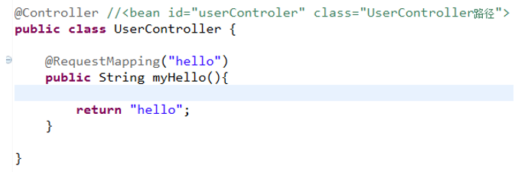
<bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/WEB-INF/jsps/"*></property>

<property name=*"suffix"* value=*".jsp"*></property>

</bean>

定义自定义的Controller



注意：return ”hello”,表示springmvc返回了一个逻辑视图hello

定义hello页面

根据视图解析，需要在web-inf下面定义jsps文件夹，在里面定义一个hello.jsp

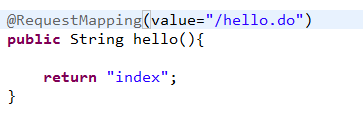
### 注解

#### @RequestMapping

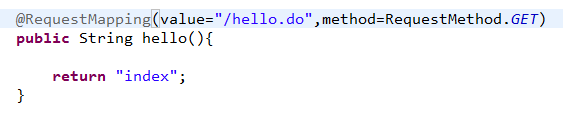
功能：请求映射

几种写法：

* **requestMapping(“hello”) ---->可以匹配任何的扩展名**
* requestMapping(“/hello.do”)
* requestMapping(value=”/hello.do”)

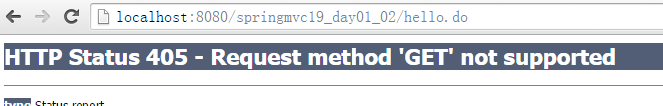


* requestMapping(value=”/hello.do”,method=RequestMethod.GET)



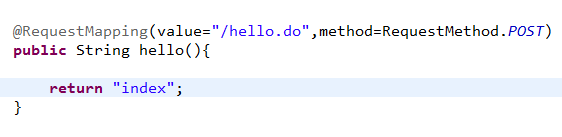
* requestMapping(value=”/hello.do”,method=RequestMethod.POST)

直接在浏览器进行访问，会报错



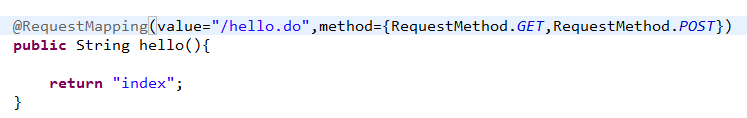
浏览器直接写url或者a标签都是get请求

一般情况下，如果使用form表单提交都是post请求，还有ajax请求指定post请求



* requestMapping(value=”/hello.do”,method={RequestMethod.POST, RequestMethod.GET})

两种请求都支持，什么都不写，默认就是两种都支持。所以，一般不这么写。



#### @requestMapping根路径

@RequestMapping（”/user”）

UserController｛

requestMapping(“save”)

Save()

requestMapping(“update”)

Update{}

requestMapping(“find”)

Fiind()

｝

项目名/user/save.do

@RequestMapping（”/items”）

ItemsController｛

requestMapping(“save”)

Save()

requestMapping(“update”)

Update{}

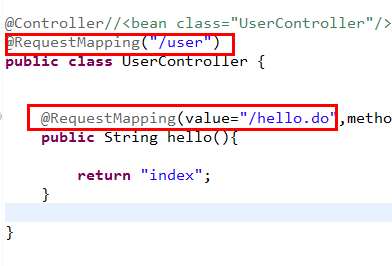
requestMapping(“find”)

Fiind()

｝

项目名/items/save.do

根路径就是用来隔离controller里面相同的方法名。



#### Springmvc封装参数

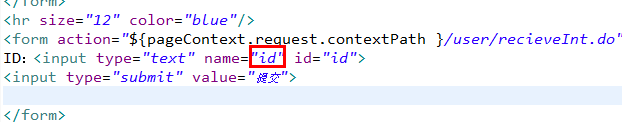
注意：springmvc没有成员变量，需要传递的参数对象放入方法中，当你请求这个方法的时候，这个方法里面对象会自动被创建，需要封装的参数自动被封装方法的对象。

基本类型

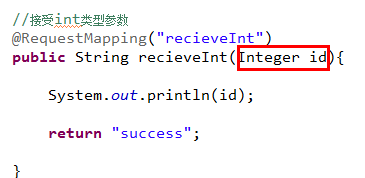
**Integer类型**

**页面**

页面传递参数都是字符串。

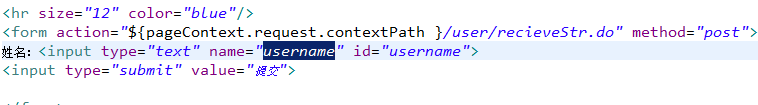


**接收参数方法**

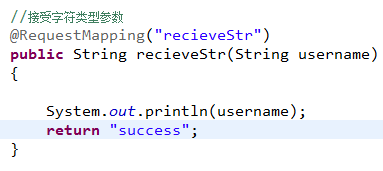


**String类型**

**页面**



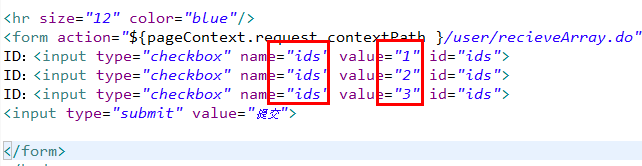
**接收参数方法**



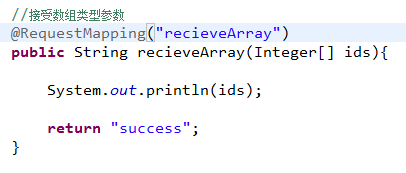
**接收数组类型参数**

分析：批量删除：checkbox复选框。Value必须有值。

**页面**



**接收参数方法**

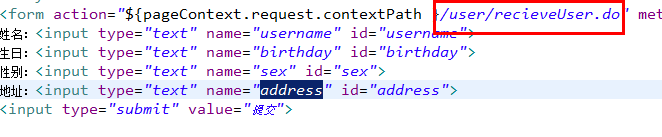


接收pojo类型参数

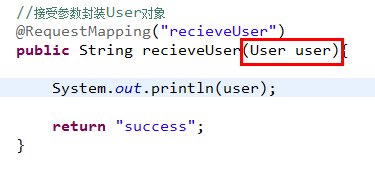
**创建pojo**

**User.java**

**页面**



**接收参数方法**



接收包装类型参数

**创建包装类**UserCustom.java

userCustom｛

private user user；

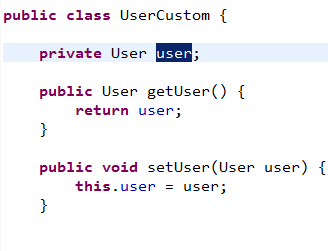
private List<User> userList;

private Map<String,Object> maps = new HashMap<String,Object>();;

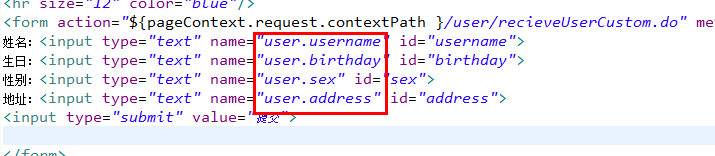
private items items;

set...get....

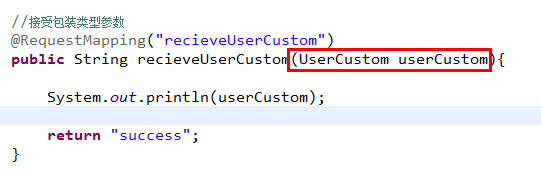
｝



**页面**

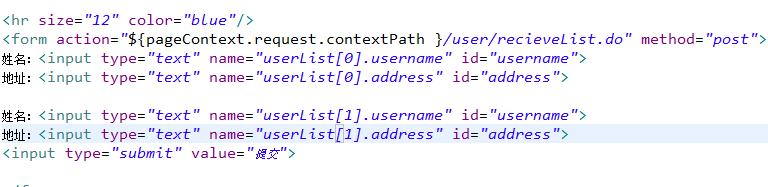


**接收参数方法**

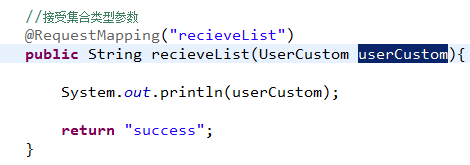


接收集合型参数

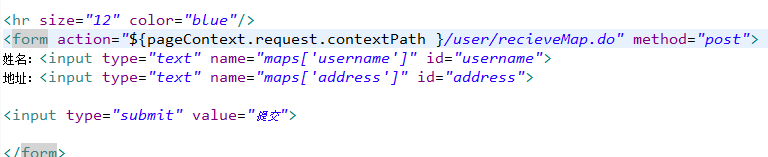
* **接收list类型参数**



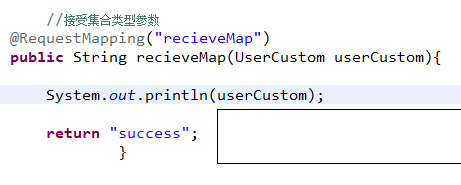
**接收参数方法**



* **接收map类型参数**



**接收参数方法**



### 有了Struts2，为什么还需要使用springmvc?

实现机制：

Struts2是基于过滤器实现的。

Springmvc基于servlet实现。Servlet比过滤器快。

运行速度：

Struts2是多列

请求来了以后，struts2创建多少个对象：

ActionContext，valuestack，UserAction，ActionSuport，ModelDriven

userAction里面属性：User对象，userlist集合等

Springmvc是单列

创建了1个对象，如果访问方法，只需要创建方法里面的对象，是局部对象，不用时被回收。

参数封装来分析：

Struts基于属性进行封装。

Springmvc基于方法封装。

### 页面参数回显Model

springmvc使用Model对象，Model对象相当于application;

application对象中数据可以是el表达式进行获取。

#### 查询所有

Model传值

@RequestMapping("list")

**public** String list(Model model){

//model 相当于application域对象

List<User> userList = **new** ArrayList<User>();

User user1 = **new** User();

user1.setId(1);

user1.setSex("男");

user1.setUsername("张山峰");

user1.setAddress("武当山");

user1.setBirthday(**new** Date());

User user2 = **new** User();

user2.setId(2);

user2.setSex("男2");

user2.setUsername("张山峰222");

user2.setAddress("武当山222");

user2.setBirthday(**new** Date());

User user3 = **new** User();

user3.setId(3);

user3.setSex("男3");

user3.setUsername("张山峰333");

user3.setAddress("武当山333");

user3.setBirthday(**new** Date());

userList.add(user1);

userList.add(user2);

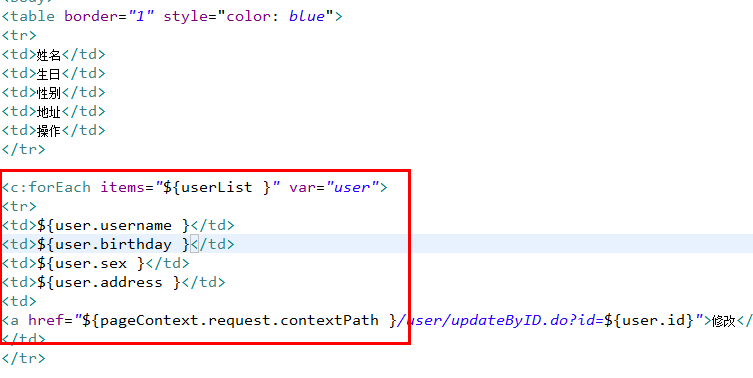
userList.add(user3);

model.addAttribute("userList", userList);

**return** "list";

}

页面获取：

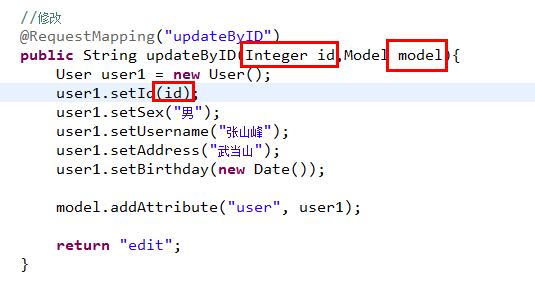


#### 修改

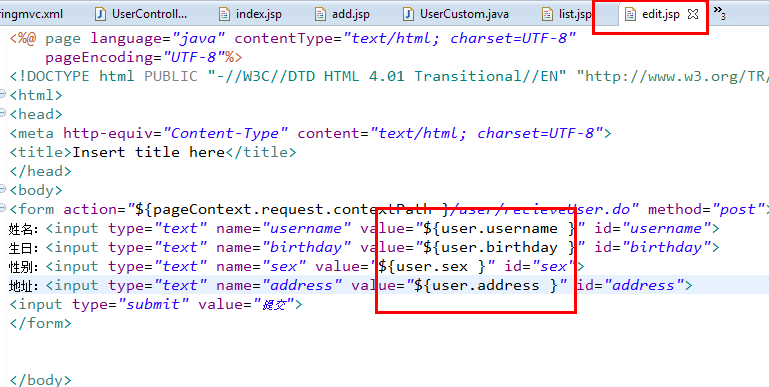
##### 页面传值



##### 修改代码



##### 回显：



### URL模版映射@PathVariable

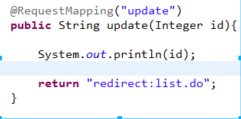
url模版映射可以restfull软件架构。主要是为了请求restfull设计模式。

Restfull软件架构设计模式：请求更简洁，更安全，方便与搜索引擎收录。

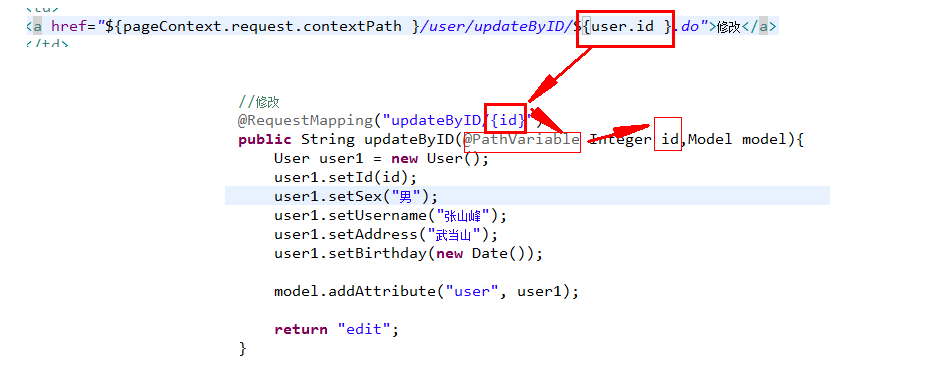
#### 普通模式修改



代码：



#### url模版映射过程@PathVariable

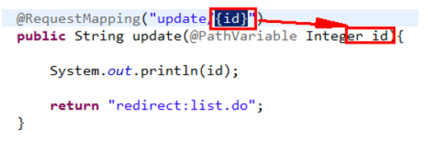


#### Restfull风格设计

##### 1.初始restfull风格，带扩展名的



代码：



##### 真正的restfull风格，不带扩展名

Web.xml拦截方式：在rest目录下所有请求都被拦截，servlet可以拦截目录。

web.xml添加约定：



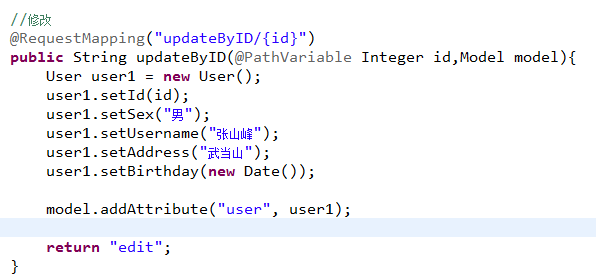


后台代码不变：

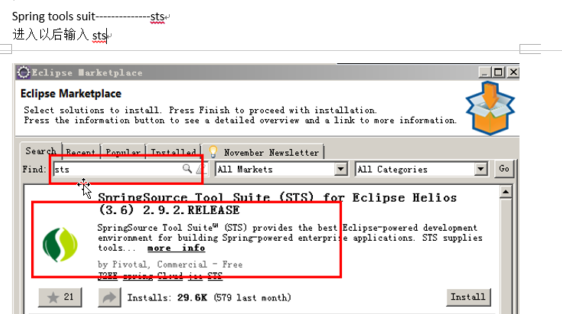


{}:匹配接受页面Url路径参数

@Pathariable：{}里面参数注入后面参数里面



安装提示信息插件：



#### RequestParam给参数定义别名

value：参数名字，即入参的请求参数名字，如value=“studentid”表示请求的参数区中的名字为studentid的参数的值将传入；

required：是否必须，默认是true，表示请求中一定要有相应的参数，否则将报400错误码；

defaultValue：默认值，表示如果请求中没有同名参数时的默认值

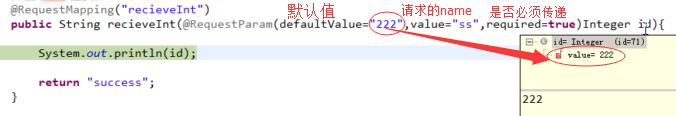
定义如下：

public String userlist( @RequestParam(defaultValue="2",value="group",required=true) String groupid) {

}

形参名称为groupid，但是这里使用value="group"限定参数名为group，所以页面传递参数的名必须为group。这里通过required=true限定groupid参数为必需传递，如果不传递则报400错误，由于使用了defaultvalue=”2”默认值即使不传group参数它的值为”2”，所以页面不传递group也不会报错，如果去掉defaultvalue=”2”且定义required=true则如果页面不传递group则会报错。

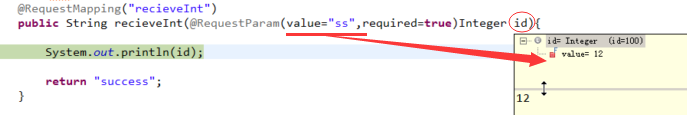
1. 第一种情况：没有传值，用默认值。使用别名ss传参数



前台请求：



1. 不要默认值



前台请求：



注意：所有页面不传值的时候，其实也传了值，穿了空字符串，从url里面直接访问可以不传值。

## 重定向和转发

### 转发

关键字：forward

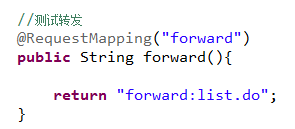
##### 本类进行转发：

本类方法与方法之间进行forward

转发方式：

方式一：return ”forward：list.do“；

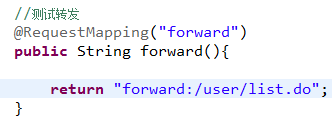
代码：



测试方式：在list方法打断点，如果断点能成功，证明转发成功。



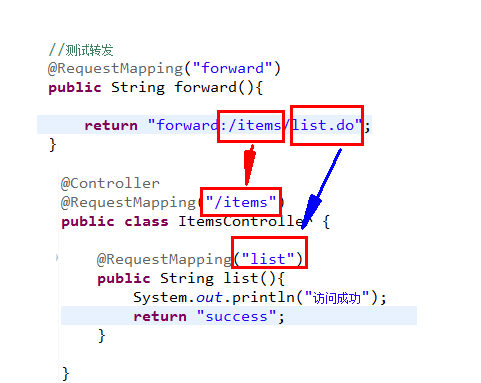
方式二：return ”forward：/user/list.do“；



注意：user根路径前面必须有/.

##### 跨类进行转发：

转发方式：return ”forward：/items/list.do“；



### 重定向

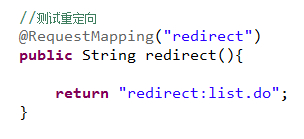
关键字：redirect

##### 本类进行重定向：

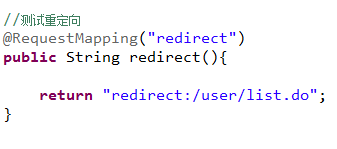
本类方法与方法之间进行redirect

重定向方式：

方式一：return ”redirect：list.do“；

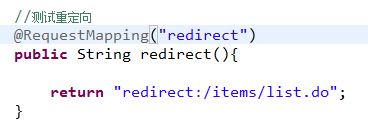


方式二：return ”redirect：/user/list.do“；



##### 跨类进行重定向：

转发方式：return ”redirect：/items/list.do“；



### <mvc:annotation-driver />配置：

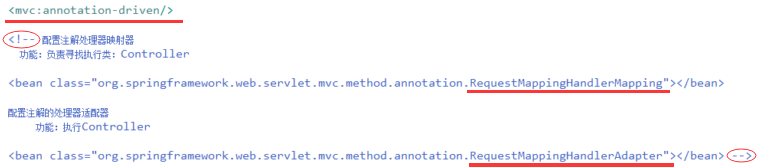
在spring配置中，注解映射器和注解适配器可以使用<mvc:annotation-driven />代替，

<mvc:annotation-driven />默认创建RequestMappingHandlerMapping ,

RequestMappingHandlerAdapter , 默认提供json数据格式的支持。

但是注意：javaBean不能添加@XmlRootElement。

@XmlRootElement提供对xml视图支持。在springmvc中的配置也不能与多视图配置一起使用。



即以下配置可用<mvc:annotation-driven />代替：

<!--注解映射器 -->

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>

<!--注解适配器 -->

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">

<property name="messageConverters">

<list>

<bean class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"></bean>

</list>

</property>

</bean

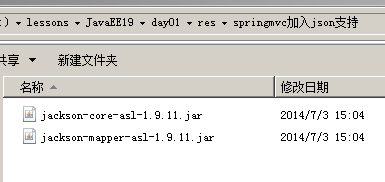
### @responseBody和@RequestBody

@responseBody把后台pojo转换json对象，返回到页面。

@RequestBody接受前台json数据，把json数据自动封装javaBean。

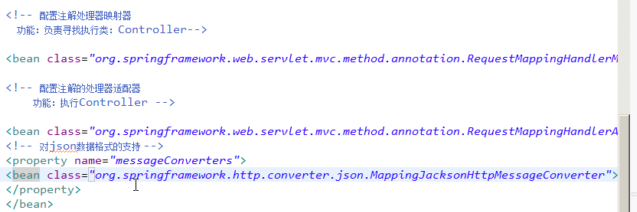
这连个注解不能直接使用，需要依赖Jackson的jar

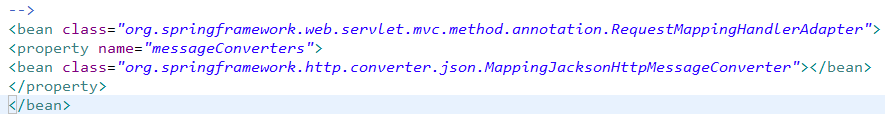
#### 导入json的jar



#### 配置json格式转换

在spring.xml中配置对json数据格式的支持





#### 前台页面引入jquery.js ，用ajax向传递json数据

Ajax传递 json格式数据



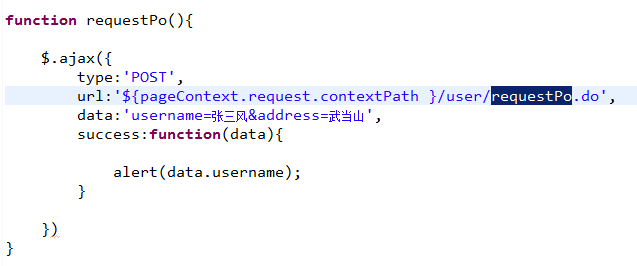
#### 后台接收json格式数据



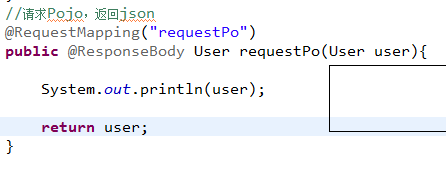


#### 需求：pojo，后台返回json

##### 前台请求数据构造：key=value&key=value.



##### 后台代码：



## Springmvc多视图

#### 导入xml格式支持jar



#### 配置springmvc支持多视图

<bean class=*"org.springframework.web.servlet.view.ContentNegotiatingViewResolver"*>

<!-- 配置支持媒体类型 -->

<property name=*"contentNegotiationManager"*>

<bean class=*"org.springframework.web.accept.ContentNegotiationManagerFactoryBean"*>

<property name=*"mediaTypes"*>

<map>

<entry key=*"json"* value=*"application/json"*></entry>

<entry key=*"xml"* value=*"application/xml"*></entry>

</map>

</property>

</bean>

</property>

<!-- 指定默认视图 -->

<property name=*"defaultViews"*>

<!-- 支持多个视图 -->

<list>

<!-- 对josn格式视图支持 -->

<bean class=*"org.springframework.web.servlet.view.json.MappingJacksonJsonView"*></bean>

<!-- xml格式视图支持 -->

<bean class=*"org.springframework.web.servlet.view.xml.MarshallingView"*>

<constructor-arg>

<bean class=*"org.springframework.oxm.jaxb.Jaxb2Marshaller"*>

<property name=*"classesToBeBound"*>

<list>

<value>cn.itcast.domain.User</value>

</list>

</property>

</bean>

</constructor-arg>

</bean>

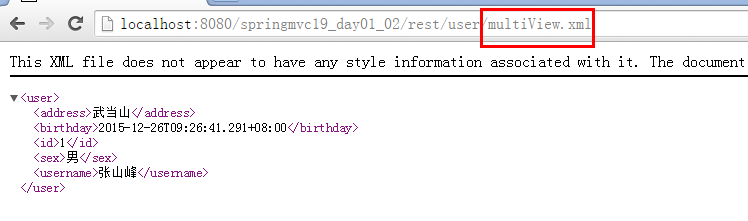
</list>

</property>

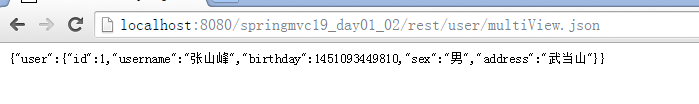
</bean>

#### 前台请求

约定rest目录下所有以json和xml扩展名都支持相应的视图

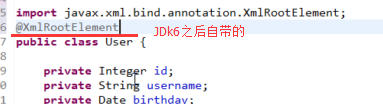


Json



#### 后台代码

支持xml时，javaBean需要加一个注解：@XmlRootElement

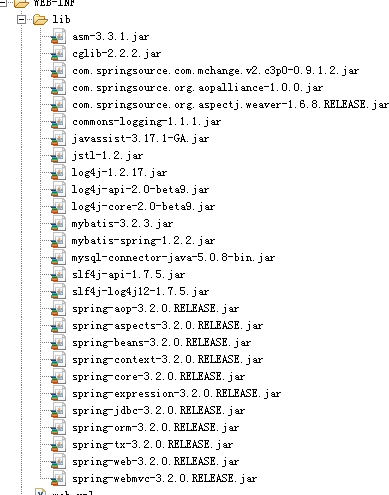




## ssm整合

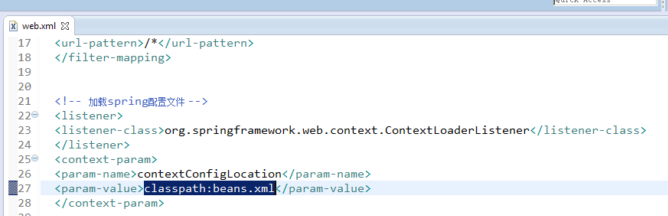
### 导入jar

导入spring(包含springmvc)，mybatis，mybatis-spring整合。数据库驱动，jstl，c3p0管理数据源，log4j.



### 配置web.xml入门文件

前面对springmvc的配置不变，再加上对spring的配置文件加载



### springmvc配置文件



### spring配置文件

##### beans.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.2.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.2.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.2.xsd"*>

<context:component-scan base-package=*"cn.itcast"*></context:component-scan>

<!-- 第一步：配置数据源 -->

<context:property-placeholder location=*"classpath:jdbc.properties"* />

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"jdbcUrl"* value=*"${jdbc.url}"*></property>

<property name=*"driverClass"* value=*"${jdbc.driver}"*></property>

<property name=*"user"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

<!-- 第二步：创建sqlSessionFactory。生产sqlSession -->

<bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

<property name=*"configLocation"* value=*"classpath:sqlMapConfig.xml"*></property>

</bean>

<!-- 配置mybatis接口代理开发

\* 接口类名和映射文件必须同名

\*　接口类和映射文件必须在同一个目录　下

\* 映射文件namespace名字必须是接口的全类路径名

\*　接口的方法名必须和映射Statement的ｉｄ一致

-->

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<property name=*"basePackage"* value=*"cn.itcast.dao"*></property>

<property name=*"sqlSessionFactoryBeanName"* value=*"sqlSessionFactory"*></property>

</bean>

<!-- 第三步：事务 -->

<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<!-- 配置通知 -->

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<tx:method name=*"save\*"* propagation=*"REQUIRED"* />

<tx:method name=*"update\*"* propagation=*"REQUIRED"* />

<tx:method name=*"delete\*"* propagation=*"REQUIRED"* />

<tx:method name=*"insert\*"* propagation=*"REQUIRED"* />

<tx:method name=*"\*"* propagation=*"REQUIRED"* />

</tx:attributes>

</tx:advice>

<!-- 配置拦截service -->

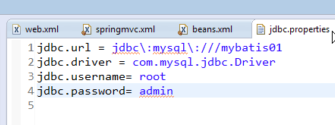
<aop:config>

<aop:advisor advice-ref=*"txAdvice"* pointcut=*"execution(\* cn.itcast.service.\*.\*(..))"*/>

</aop:config>

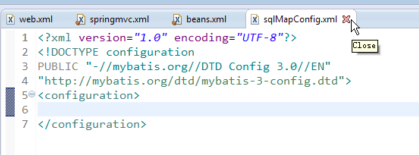
</beans>

##### jdbc.properties



##### sqlMapConfig.xml

配一个空的放这里即可，待需要配二级缓存等需求时再配置具体内容



========================================

# Spring Boot

## 一、简介

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式Spring Boot致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。

Spring Boot的四个核心：

-- 自动配置：针对很多Spring应用程序常见的应用功能，Spring Boot能够

自动提供相应配置。

-- 起步依赖：告诉Spring Boot需要什么功能，就引入需要的库。

-- 命令行界面：这是Spring Boot的可选特性，借此你只需写代码就能完成完整的

应用程序，无需传统的项目构建。

-- Actuator: 让你能够深入运行中的Spring Boot应用程序，一探究竟。

## 快速入门

========================================

========================================

========================================

========================================