

开发记录

卷一

Dolphin

Copyright © 2017 Xiaoqiang Jiang

EDITED BY XIAOQIANG JIANG

[HTTP://JIANGXIAOQIANG.GITHUB.COM/](http://jiangxiaoqiang.github.com/)

All Rights Reserved.

Version 16:02, December 2, 2017

Contents

I	Framework	
1	Spring	3
1.1	Common Sense	3
1.1.1	classpath	3
1.1.2	Spring Boot 开发者工具	6
1.1.3	immutable	6
1.1.4	集合操作 (Collection Operate)	7
1.1.5	正则表达式 (Regular Expression)	11
1.1.6	Code Snippets	12
1.2	接口 (Application Interface)	14
1.2.1	请求消息 (Request)	14
1.2.2	返回消息封装 (Message Encapsulation)	16
1.2.3	接口认证 (Authorization)	17
1.2.4	日志记录 (Logging)	20
1.2.5	异常处理 (Exception Handle)	23
1.2.6	版本管理 (Interface Version Control)	24
1.2.7	接口缓存 (Interface Cache)	25

1.3	数据库操作 (Database Operate)	26
1.3.1	Could not get JDBC connection	26
1.3.2	JDBC 数据库重连 (JDBC Database Reconnect)	26
1.3.3	HikariCP 数据库连接	29
1.3.4	Spring Boot 缓存	32
1.3.5	用户重复登录问题	33
1.3.6	读取 properties 属性	33
1.4	单元测试 (Unit Test)	34
1.4.1	Gradle 中执行单元测试	34
1.4.2	TestNG 单元测试	34
1.4.3	Spring mockMvc 测试	35
1.5	常见问题 (Frequently Encounted Question)	36
1.5.1	java.lang.OutOfMemoryError: Java heap space	40
1.6	MyBatis	42
1.6.1	打印 SQL(Print SQL)	42
1.6.2	多数据源 (Multi-Datasource)	43
1.6.3	不同配置文件	44
1.6.4	多参数传递 (Multi-Parameter)	44
1.6.5	批量写入	45
1.6.6	resultMap 和 resultType 区别	46
1.6.7	MyBatis 常见问题	46
2	Scala	49
2.1	函数 (Function)	49
2.1.1	排序	50
2.1.2	集合操作	50
3	Python	51
3.0.1	基础	51
3.0.2	Beautiful Soup	52
3.0.3	爬取豆瓣书籍	53
3.0.4	抓取技巧	53
3.0.5	读取 Properties 文件	54
3.0.6	sh: mysql_config: command not found 解决办法	56

4	前端 (Front End)	57
4.1	React	57
4.1.1	组件生命周期 (Life Cycle)	57
4.1.2	第一个 React	57
4.1.3	React 数据的流转过程	58
4.1.4	规范 (Specification)	58
4.1.5	Router	58
4.1.6	Provider	60
4.1.7	Connect	61
4.1.8	Actions	62
4.1.9	Store	62
4.1.10	Dispatcher	62
4.1.11	Reducer	64
4.1.12	state	64
4.1.13	Props	67
4.1.14	Reducers	69
4.1.15	Immutable	69
4.1.16	前端 Mock	70
4.2	MobX	70
4.2.1	简介	70
4.3	gulp	70
4.3.1	压缩图片	70
4.3.2	清除文件	71
4.3.3	设置默认任务	71
4.4	webpack	71
4.5	Library	71
4.5.1	lodash	71
4.5.2	Axios	72
4.5.3	Axios 请求	73
4.5.4	Axios 拦截响应	74
4.5.5	Promise	74
4.6	Cookie	75
4.7	Javascript	77
4.7.1	prototype	77

4.7.2	常用函数	78
4.7.3	日期处理	79
4.8	常见问题	80

II

OS

5	Linux	87
----------	--------------------	-----------

5.1	Shell	87
------------	--------------	-----------

5.1.1	自动部署 (Auto Deploy)	87
5.1.2	调试 Bash 脚本 (Bash Debugging)	94
5.1.3	命令 (Command)	94
5.1.4	tmux	98
5.1.5	环境变量 (Environment Variable)	99
	screen	101
	查看 Linux 信息	102

5.2	常用设置	102
------------	-------------	------------

5.2.1	XFCE 桌面	102
5.2.2	Fedora 访问 Windows 共享文件夹	102
5.2.3	Ubuntu 访问 Windows 共享文件夹	103
5.2.4	软件包管理 (Package Management)	103
5.2.5	Ubuntu 打印	104
5.2.6	输入法 (Input Method)	104

III

Tool

6	IntelliJ Idea	109
----------	----------------------------	------------

6.0.1	IntelliJ Idea 远程调试	109
6.0.2	IntelliJ Idea 调试慢	110
6.0.3	无法查看 Scala 变量	110
6.0.4	常见设置	110
6.0.5	Tips	112

7	Gradle	115
7.1	基础	115
7.1.1	初始化项目	116
7.1.2	引用本地文件	116
7.1.3	Gradle Properties 支持	116
7.1.4	执行流程	117
7.1.5	默认 JVM	117
7.1.6	repositories	117
7.1.7	常见问题	118
7.2	Gradle 对象	119
7.2.1	属性 (Properties)	119
	Extra Properties	119
7.2.2	Task	120
8	OWASP ZAP	123
8.1	基础	123
8.1.1	调整	123
8.1.2	分析	123
9	OpenVPN	125
9.0.1	安装	125
9.0.2	生成客户端证书	126
9.0.3	Mac Book 客户端配置	128
9.0.4	Ubuntu 远程 Raspberry	129
9.0.5	Ubuntu 远程 Ubuntu	130
9.0.6	Mac Book 远程 Fedora	131
9.0.7	常见问题	131
	ssl3_get_server_certificate:certificate verify failed	131
10	Nmap(GNU General Public License)	133
10.0.1	主机发现 (Host Discovery)	133
10.0.2	端口扫描 (Port Scanning)	134
10.0.3	版本侦测 (Version Detection)	136
10.0.4	Zenmap	137

11 Raspberry Pi	139
11.1 基础	139
11.1.1 连接 (Connection)	139
12 Widgets	141
12.1 Vim Editor	141
12.2 Little Tool	141
12.2.1 SQuirreL SQL Client	141
12.2.2 You-Get	144
12.2.3 transmission	144
12.2.4 rsync	144
12.2.5 ag	144
12.2.6 Pandoc	144
12.2.7 Curl	145
12.2.8 iostat	146
12.2.9 Zeal	146
12.2.10Thunderbird 邮件客户端	146
12.2.11jq	148
12.3 Nginx	148
12.3.1 X-Forwarded-For	149
12.3.2 Nginx 获取真实 IP	149
12.3.3 Nginx 并发	153
12.3.4 URI 长度限制	154
12.3.5 开启 gzip 压缩	154
12.3.6 Nginx 配置	155
12.3.7 常见问题	156
12.4 Ansible	158
12.4.1 Ansible 模块	158
12.4.2 Ansible 代理 (Ansible Proxy)	159
12.5 ssh(Secure Shell)	159
12.5.1 免密登陆	159
12.5.2 ssh 连接慢	159
12.5.3 Permission denied (publickey)	160
12.5.4 Session 时间	161

12.5.5 代理转发	162
12.5.6 ssh 查看日志	162
12.6 VisualVM	163
12.6.1 VisualVM 远程监控	163
12.6.2 autojump	165
12.7 Graphviz	166
12.7.1 编译	166
12.7.2 subgraph	166
12.8 MyBatis Generator	167
13 Git	169
13.1 基础	169
13.1.1 常用 Git 命令	169
13.1.2 返回到指定版本	170
13.1.3 标签 (Tag)	170
13.1.4 查看修改历史	171
13.1.5 merge	172
13.1.6 Git Hook	173
13.1.7 常见问题	173
14 Google Chrome	175
14.1 DevTools	175
14.1.1 Console	175
14.1.2 Source Map	177
14.1.3 Network	177
14.1.4 Elements	179
14.1.5 Performance	179
14.1.6 Sources	179
14.1.7 Audits	180
14.1.8 Google Chrome SSH	180
15 LaTeX	181
15.1 安装	181
15.1.1 字体	181

15.1.2 版面	182
15.1.3 文献引用	182
15.1.4 支持语言	183
15.1.5 常见问题	183

IV

DB

16 性能 (Performance) 187

16.1 索引 (Index) 187

16.1.1 索引常识	187
16.1.2 索引类型	189
16.1.3 like 查询优化	190

16.2 Redis 192

16.2.1 安装	192
-----------------	-----

17 PostgreSQL 193

17.1 基础操作 193

17.1.1 安装 (Install)	193
---------------------------	-----

18 MySQL 197

18.0.1 查询	197
18.0.2 Mac 中 MySQL 初始密码	198
18.0.3 基础	198
18.0.4 备份	200
18.0.5 存储过程	200
18.0.6 mycli	201
18.0.7 常见问题	201
18.0.8 中文查询	201

19 达梦 203

19.1 SQL 203

19.1.1 索引	203
19.1.2 函数	204
19.1.3 常用 SQL 记录	206

19.2	通用知识	207
19.2.1	查询优化	207
19.2.2	系统架构	208
19.2.3	注意事项	208

V	Network
----------	----------------

20	HTTP	213
20.0.1	性能分析	213
20.0.2	Session	218
20.1	PAC(Proxy Auto-Configuration)	218
20.1.1	PAC 简介 (Pac Introduce)	218
20.1.2	PAC 实例 (PAC Example)	218
20.1.3	同时连接内外网	220

VI	Engineering
-----------	--------------------

20.2	通用规范	231
20.2.1	版本管理	231
20.2.2	项目发布 (Publish)	232

Bibliography	235
Books	235
Articles	235

这里记录的是一些比较杂乱的笔记，绝大多数文字皆来源于网络，不是自己的原创，这里没有高深的算法，没有宏伟的技术及系统架构，只是一些平时工作中遇到的一些问题，和解决问题的思路以及所采用的方案。由于平时工作时还没有遇到前人没有遇到过的问题需要自己发明方去解决(其实真的遇到估计也是没辙)，所以绝大部分内容是为了避免再遇到同样的问题时，又需要到处去搜寻，索性将之记录下来，以便于下次可以快刀斩乱麻，迅速解决问题。

Framework

1	Spring	3
1.1	Common Sense	
1.2	接口 (Application Interface)	
1.3	数据库操作 (Database Operate)	
1.4	单元测试 (Unit Test)	
1.5	常见问题 (Frequently Encounted Question)	
1.6	MyBatis	
2	Scala	49
2.1	函数 (Function)	
3	Python	51
4	前端 (Front End)	57
4.1	React	
4.2	MobX	
4.3	gulp	
4.4	webpack	
4.5	Library	
4.6	Cookie	
4.7	Javascript	
4.8	常见问题	

1. Spring

1.1 Common Sense

Java 8 (codename: Spider) was released on 18 March 2014, and included some features that were planned for Java 7 but later deferred.

1.1.1 classpath

理解 classpath 花了不短的时间，classpath 到底是哪个目录？配置文件中指定的 classpath 程序到底在哪里找的？都困扰着我。其实有很好的方法来深深的理解 classpath，如下代码片段打印出来当前项目的 classpath 以及包含的文件。

```
1 public class Application {  
2     public static void main(String[] args) {  
3         ClassLoader classLoader = ClassLoader.  
4             getSystemClassLoader();  
5         URL[] urls = ((URLClassLoader) classLoader).  
6             getURLs();  
7         for (URL url : urls) {  
8             System.out.println(url.getFile());  
9         }  
10    }
```

从代码的输出结果即可看出，classpath 具体是哪些文件夹。

```
1 # 与JDK相关的文件夹
2 /home/local/jdk1.8.0_111/jre/lib
3 /home/local/jdk1.8.0_111/jre/lib/ext
4 /home/local/jdk1.8.0_111/lib
5 # 与项目相关的文件夹，这里仅仅是一个module
6 /home/credit-system/cc-data/build/classes/main/
7 /home/credit-system/cc-data/build/resources/main/
8 # 与Gradle相关的文件夹
9 /home/hldev/.gradle/caches
```

所以在项目中看到如下的配置就不难理解了：

```
1 bean.setMapperLocations(resolver.getResources("
    classpath:/mybatis/*.xml"));
```

它实际上就是映射的如图1.1中所示的 XML 文件。

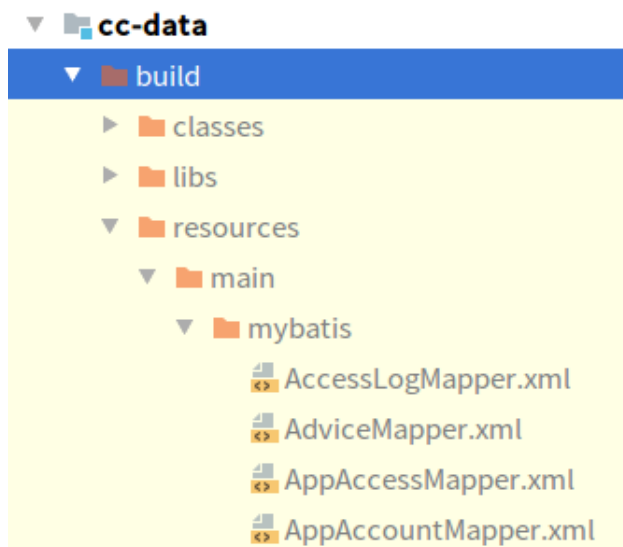


Figure 1.1: MyBatis 映射 classpath 中的文件

classpath 是指编译过后的 WEB-INF 文件夹下的 classes 目录。配置文件一般放在 resources 目录下，当编译之后会自动复制到 classes 目录下。classpath*: 的使用是为了多个 component(最终发布成不同的 jar 包) 并行开发，各自的 bean 定义文件按照一定的规则: package+filename，而使用这些 component 的调用者可以把这些文件都加载进来。不仅包含 classes 路径，还包括 jar 文件 classes 路径进行查

找。classpath: 只能加载找到的第一个文件。然而，使用 classpath*: 需要遍历所有的 classpath，加载速度很慢，因此您应该尽量避免使用 classpath*。有时提示找不到资源文件，首先检查资源文件夹的名称，资源文件夹的名称默认为 resources，注意有一个 s，其次是要检查此文件夹是否定义成了资源文件夹，资源文件夹与普通文件夹不同，资源文件夹有一个层叠的标识，如图1.2所示。

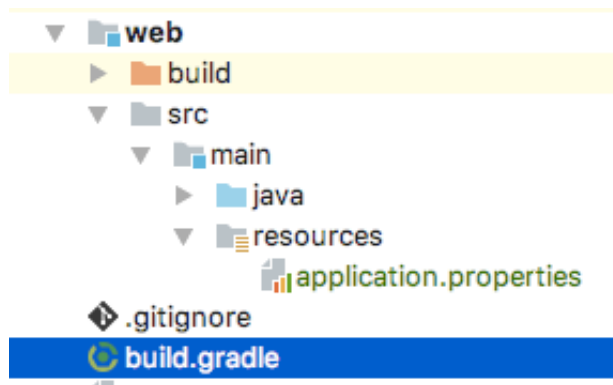


Figure 1.2: IntelliJ Idea 中的资源文件夹

仅仅是建立一个 resources 文件夹是不够的，还需要将之定义成资源文件夹，在文件夹上右键，选择 Mark Directory As，将之定义为资源文件夹。定义为资源文件夹后，即可通过如下方式指定配置文件的位置：

```
1 @Configuration
2 @Data
3 @Component
4 @PropertySource("classpath:application.properties")
5 @ConfigurationProperties(prefix = "spring.datasource"
6     )
7 public class DataSourceConfig {
8     private String jdbcUrl;
9
10    private String username;
11
12    private String driverClassName;
13
14    private String password;
15 }
```

在 IntelliJ Idea 中，如果要把配置文件加入 classpath，可以将配置文件标记为资源文件。

1.1.2 Spring Boot 开发者工具

spring-boot-devtools 为应用提供一些开发时特性，包括默认值设置，自动重启，livereload 等。在 build.gradle 文件里添加开发工具配置：

```
compile "org.springframework.boot:spring-boot-  
devtools"
```

在激活了开发者工具后，Classpath 里对文件做任何修改都会触发应用程序重启。为了让重启速度够快，不会修改的类（比如第三方 JAR 文件里的类）都加载到了基础类加载器里，而应用程序的代码则会加载到一个单独的重启类加载器里。检测到变更时，只有重启类加载器重启。在 IntelliJ Idea 中，需要在设置中配置 Compiler，勾选上 build project automatically 选项。在 Mac 中，按下 command + alt + shift + / 出现的界面中，选择 Registry。找到“compiler.automake.allow.when.app.running”这个选项，并且勾选。遗憾的是在自己的 Mac 中，IntelliJ Idea 按照如上配置后还是不能正常自动 Reload，不过当在点击了 Reload Changed Classes 之后，Spring Boot 会自动重启，可以配置 Reload Changed Classes 的快捷键来实现。

1.1.3 immutable

多线程共享变量的情况下，为了保证数据一致性，往往需要对这些变量的访问进行加锁。而锁本身又会带来一些问题和开销。Immutable Object 模式使得我们可以在不使用锁的情况下，既保证共享变量访问的线程安全，又能避免引入锁可能带来的问题和开销。多线程环境中，一个对象常常会被多个线程共享。这种情况下，如果存在多个线程并发地修改该对象的状态或者一个线程读取该对象的状态而另外一个线程试图修改该对象的状态，我们不得不做一些同步访问控制以保证数据一致性。而这些同步访问控制，如显式锁和 CAS 操作，会带来额外的开销和问题，如上下文切换、等待时间和 ABA 问题等。Immutable Object 模式的意图是通过使用对外可见的状态不可变的对象（即 Immutable Object），使得被共享对象“天生”具有线程安全性，而无需额外的同步访问控制。从而既保证了数据一致性，又避免了同步访问控制所产生的额外开销和问题，也简化了编程。可以遵照以下几点来编写一个不可变类：1. 类应该定义成 final，避免被继承。将类声明为 final（强不可变类），或者将所有类方法加上 final（弱不可变类）。或者使用静态工厂并声明构造器为 private。2. 声明属性为 private 和 final。3. 不要提供任何可以修改对象状态的方法：不仅仅是 set 方法，还有任何其它可以改变状态的方法。4. 如果类有任何可变对象属性，那么当它们在类和类的调用者间传递的时候必须被保护性拷贝。

如果某一个类成员不是原始变量 (primitive) 或者不可变类, 必须通过在成员初始化 (in) 或者 get 方法 (out) 时通过深度 clone 方法, 来确保类的不可变。

不可变对象有几个优点: 线程安全、易于理解、比可变对象有更高的安全性。

1.1.4 集合操作 (Collection Operate)

Java 8 的 stream 可以像操作数据库那样操作内存, 代码简洁清晰易懂, 缺点是性能比传统处理方式较低。List 移除空元素, 移除重复元素:

```
1 request.getParamList().stream()
2     .filter(StringUtils::isEmpty)
3     .distinct()
4     .forEach(param -> {
5         if (param.length() == 18 && CommonUtil.
6             isLetterOrNumber(param)) {
7             });
```

对象做非空判断:

```
1 f (!"".equals(name)) {
2     //将"" 写在前头
3     //不管 name 是否为 null, 都不会出错
4 }
5
6 if (StringUtils.isNotBlank((String) pageable.
7     getParams().get("qymc"))) {
8     //用 String 强制转换也不会报空指针异常
9 }
```

获取 Json 单个值:

```
1 JSONObject json = new JSONObject(responseContent);
2 String total = json.getString("
    totalElementsCondition1");
```

去除 List 元素中的特殊字符:

```
1 List<AccessLog> logs = operationLogs.stream()
```

```
2         .peek(log -> log.setQueryResult(log.  
3         getQueryResult().replaceAll("\u0000", "")))  
         .collect(Collectors.toList());
```

peek()-> 对每个遇到的元素执行一些操作。如上方法会遍历 List 列表 operationLogs 中的每个元素，将特殊字符\u0000去掉。

数组转换为 ArrayList

数组转 List 有如下方式：

```
1 Integer[] integers = new Integer[] {1,2,3,4,5};  
2 // Cannot modify returned list  
3 List<Integer> list21 = Arrays.asList(integers);  
4 // good  
5     List<Integer> list22 = new ArrayList<>(Arrays.  
6     asList(integers));  
7 //Java 8  
8 List<Integer> list23 = Arrays.stream(integers).  
9     collect(Collectors.toList());
```

采用 Arrays.asList 转时，返回的 List 不能修改，因为 asList() 返回的列表是由原始数组支持的固定大小的列表。这种情况下，如果添加或删除列表中的元素，程序会抛出异常 UnsupportedOperationException。

获取数据

获取 List 对象对象的某一个字段：

```
1 List<String> redblackLogKeys = redBlackLogs.stream()  
2     .map(redBalckLog -> redBalckLog.getResponseKey())  
3     .collect(Collectors.toList());
```

List

如下代码片段按照 responsekey 对集合进行分组：

```
1 Map<String, List<FeedbackRecord>> groupByLists =  
2     feedbackRecords  
3     .stream()
```



```
3         .collect(Collectors.groupingBy(FeedbackRecord  
        ::getResponsekey));
```

如果需要在创建 List 时对其初始化：

```
1 List<String> defaultUnion = new ArrayList<String>() {  
2     {  
3         add(PublicVariable.DEFAULT_ACCORDING_IMPLEMENT  
4             );  
5     }  
6 };
```

有时需要将 List 对象按照对象某一个字段是否重复进行去重，可以先重写对象的 equals 方法：

```
1 @Override  
2 public boolean equals(Object o) {  
3     if (this == o) return true;  
4     if (!(o instanceof UnitedAwardsPenalties))  
5         return false;  
6     UnitedAwardsPenalties stu = (  
7         UnitedAwardsPenalties) o;  
8     return implementContent != null ?  
9         implementContent.equals(stu.implementContent)  
10        : stu.implementContent == null;  
11 }  
12  
13 @Override  
14 public int hashCode() {  
15     int result = id.hashCode();  
16     result = 31 * result + memo.hashCode();  
17     return result;  
18 }
```

就可以对对象进行直接比较：

```
1 /**  
2  * 去重
```

```
3 */
4 unitedAPMapper.pageUnitedAPList(paramsMap)
5     .stream()
6     .forEach(element -> {
7         if (!distinctUnitedAwardsPenaltiesList.contains(
8             element)) {
9             distinctUnitedAwardsPenaltiesList.add(element
10            );
11        }
12    });
```

查找

如下代码片段查找第一个符合条件的元素：

```
1 Optional<FeedbackRecord> filterFeedbackRecord = entry
2     .getValue()
3     .stream()
4     .filter(feedbackRecord -> "0".equals(
5         feedbackRecord.getIsFeedback()))
6     .findFirst();
7 //更加直接的方式判断是否有符合条件的元素
8 boolean isChild = users.stream().anyMatch(t->t.age
9     <18);
```

parallelStream

在采用 `parallelStream` 时，不能使用全局的可变对象，要么使用不可变对象，要么使用局部对象，如下代码片段所示：

```
1 List<String> example = new ArrayList<>();
2 example.add("a");
3 example.add("b");
4 example.add("c");
5 example.add("d");
6 Pageable pageable = DataUtils.pageable(1, 2, "a", "
7     asc");
8 for (int i = 0; i < 10000; i++) {
```

```
8     example.parallelStream().forEach(element -> {
9         pageable.addSomeParam("a", element);
10        if ("a".equals(element)) {
11            System.out.println(pageable.getParams().
12                get("a"));
13        }
14    });
15 }
```

运行此段代码，会发现和字符串 a 相等的元素输出有可能为 b、c、d。为什么会出现这样的现象呢，是由于使用了全局的可变对象 `pageable`，当执行到输出语句时，`pageable` 可能已经被其他线程设置成了另外的值。解决的办法要么是把 `pageable` 做成不可变的对象，并只能在初始化的时候改变其内容，要么将之移动到局部，每次循环的时候创建新的 `pageable` 对象。

Object 转为 Map

有时需要将 Object 实体转换为 Map 的方式：

```
1 public static Map<?, ?> objectToMap(Object obj) {
2     if (obj == null)
3         return null;
4     return new org.apache.commons.beanutils.BeanMap(
5         obj);
6 }
7 //调用
8 Map<String, Object> stringObjectMap = (Map<String,
9     Object>) objectToMap(documentList.get(i));
```

需要引用 BeanUtils 包，Gradle 引用如下：

```
1 compile group: 'commons-beanutils', name: 'commons-
    beanutils', version: '1.9.3'
```

BeanUtils 主要提供了对于 JavaBean 进行各种操作。

1.1.5 正则表达式 (Regular Expression)

匹配电话号码的正则表达式：

```
1 "^1(3[4-9]|4[7]|5[0-27-9]|7[08]|8[2-478])\\d{8}$"
```

[] 匹配模式能够匹配方括号内任意字符，[08] 可匹配 0 或者 8。以上匹配电话号码的正则表达式会随着时间过期，比如某些号码段新开放时，相应的正则表达式会过期，比如以上正则表达式就无法匹配 17623083674 这个电话号码。Javascript 根据空格和换行做分割：

```
1 let paramList = value.split(/\n+|\s+/);
```

\n 表示匹配一个换行符。+ 字符表示匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。替换空格：

```
1  /*
2   其实不能替换空格
3   \s匹配任何不可见字符，包括空格、制表符、换页符等等。
4       等价于[ \f\n\r\t\v]
5   \f:form feed character
6   \n:new line character
7   \r:carriage return character
8   \t:tab character
9   \v:vertical tab character
10  \S匹配任何可见字符。等价于[^\f\n\r\t\v]
11 */
12 values.replace(/\s/g, '').split(/\n+/);
13 // 方式三
14 values.replace(/[ ]/g, '').split(/\n+/);
```

\s 元字符用于查找空白字符。空白字符可以是：空格符 (space character) 制表符 (tab character) 回车符 (carriage return character) 换行符 (new line character) 垂直换行符 (vertical tab character) 换页符 (form feed character)。/g 是全局 (Global) 匹配。

1.1.6 Code Snippets

有时想测试某段代码的运行效果，代码足够短也不太需要使用 IDE 来运行，相当于一个代码草稿本类似的功能。那么此时可以使用 Visual Studio Code 中使用 Code Runner 插件，此插件可以支持多种语言，测试草稿代码，如图1.3所示，想要查看对一个空字符串做 toString() 操作会有什么效果：

ArrayList 分页：

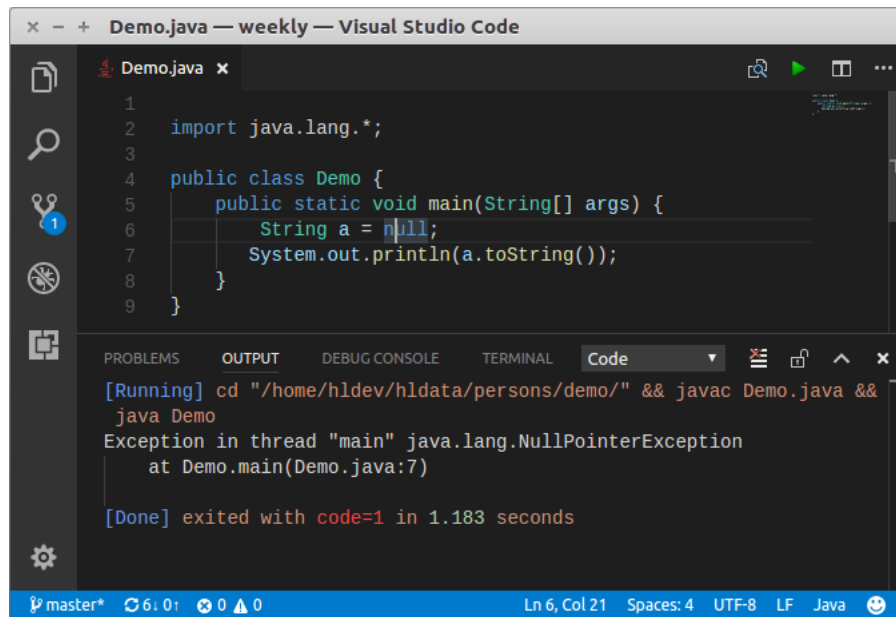


Figure 1.3: Visual Studio Code 代码草稿

```

1 return new Page<>(pageable, menuList.subList((page -
    1) * size, page * size));

```

Optional 避免空指针:

```

1 public static String getChampionName(Competition
    comp) throws IllegalArgumentException {
2     return Optional.ofNullable(comp)
3         .map(c->c.getResult())
4         .map(r->r.getChampion())
5         .map(u->u.getName())
6         .orElseThrow(()->new IllegalArgumentException
7             ("The value of param comp isn't available."))
8         ;
9 }

```

Optional 参数验证:

```

1 public void setName(String name) throws
    IllegalArgumentException{
2     this.name = Optional.ofNullable(name)

```

```
3         .filter(User::isNameValid)
4         .orElseThrow(()->new
        IllegalArgumentException("Invalid username.")
        );
5     }
```

String.valueOf() 的坑:

```
1 //如果 article.getCategory() 为空返回 null 字符串
2 String categoryId = String.valueOf(article.
        getCategory());
```

String.valueOf 中如果对象为空, 就返回字符串的"null", 注意不是 null 对象, 是字符串、字符串、字符串。

1.2 接口 (Application Interface)

1.2.1 请求消息 (Request)

HttpServletResponse 对象 (HttpServletRequest 同理) 的字符输出流在编码时, 默认采用的是 ISO 8859-1 编码, 该编码方式不兼容中文, 比如会将"中国" 编码为"63 63"(在 ISO 8959-1 的码表中查不到的字符会显示 63)。当浏览器对接收到的数据进行解码时, 会默认采用 GB2312, 将"63" 解码为"?", 浏览器就将"中国" 两个字符解码为"??"。在获取请求消息记录日志时, 获取到的汉字乱码。获取请求参数的语句为:

```
1 // 获取未经过解码的参数 (汉字会显示乱码)
2 String queryString = request.getQueryString();
3 // 获取经过解码的参数
4 String qymc = request.getParameter("qymc");
```

使用 request.getParameter 方式获得的参数是已经经过 web 服务器解码的。使用 request.getQueryString 可以获得未解码的原始参数。对于 tomcat 解码造成的乱码问题可以通过 2 种途径解决, 修改 tomcat 配置文件设置解码方式, 服务器端对于获取到的参数进行 new String(param.getBytes("ISO-8859-1")," 页面指定编码") 转换, 如下代码片段所示:

```
1 // 将获取到的消息解码
```

```
2 String decodedQueryString=new String(queryString.  
    getBytes("ISO-8859-1"),"UTF-8");
```

POST 请求

发送 POST 请求时, 客户端统一使用 Json 发送, 服务端通过定义实体来获取。

```
1 @PostMapping("create")  
2 @ApiOperation(value = "创建文章", notes = "创建文章")  
3 public ApiResult createArticle(@RequestBody Article  
    article) {  
4 }
```

使用 curl 模拟 POST 请求:

```
1 curl -X POST -H "APPID:hlb1451j6d136334gh2" \  
2 -H "TIMESTAMP:2016-12-19 16 (tel:2016121916):58:02" \  
3 -H "ECHOSTR:sdsaasf" \  
4 -H "TOKEN:e02016ed4adfd9d9743ab70cf389ef9877c4a0e" \  
5 -H 'Content-Type: application/json;charset=UTF-8' \  
6 http://59.214.215.6:28080/api/article/create \  
7 --data-binary $'{"title":"a123","content":"123456","  
    category":12,"publisher":"ab"}'|jq '.'
```

需要传递请求头 Content-Type, 否则会提示如下错误:

```
1 Content type 'application/x-www-form-urlencoded;  
    charset=UTF-8' not supported
```

编辑请求 (Edit Request)

有时在写好接口之后, 修改可能是非常细小的, 比如原来的接口路径是/page, 但是现在需要返回全部的数据可能就是/page/all(也可以在原来的接口方法改造, 这里不考虑哪种方法实现更优)。需要测试一下新接口的返回, 一种可以在浏览器的 Network 页中拷贝出 curl 请求命令来修改, 修改后直接通过 curl 请求; 一种方法在浏览器中安装 tamper 插件, 抓取请求修改; 一种方法可以使用 Zed Attack Proxy¹代理, 在代理中编辑 HTTP 请求信息。

¹<https://github.com/zaproxy/zaproxy>

1.2.2 返回消息封装 (Message Encapsulation)

一个完整的标准化服务描述性语言 SOAP (Simple Object Access Protocol, 简单对象访问协议) 在某些环境下表现的还不错。尽管如此, SOAP 产品的性能开销很大, 它是一个巨大的性能杀手。这也是 Restful 接口及接口采用 Json 交互的优势之一, 在编写 Restful 接口时, 需要将返回的内容用标准格式进行封装。通过编写统一的返回实体来实现:

```
1 @RequestMapping("books")
2 public RestApiResponse<List<Book>> getUserShelfBooks
   (long userId){
3     List<Book> userShelfBooks = bookShelfComposite.
       getUserShelfBooks(userId);
4     return new RestApiResponse<>(ResponseCode.
       REQUEST_SUCCESS_MESSAGE, ResponseCode.
       REQUEST_SUCCESS, userShelfBooks);
5 }
```

RestApiResponse 为统一定义返回的返回实体, 所有的接口都会返回标准的 RestApiResponse 对象, 对象通过泛型定义了不同的返回内容。

接口返回 (Interface Response)

接口返回的格式如图所示:

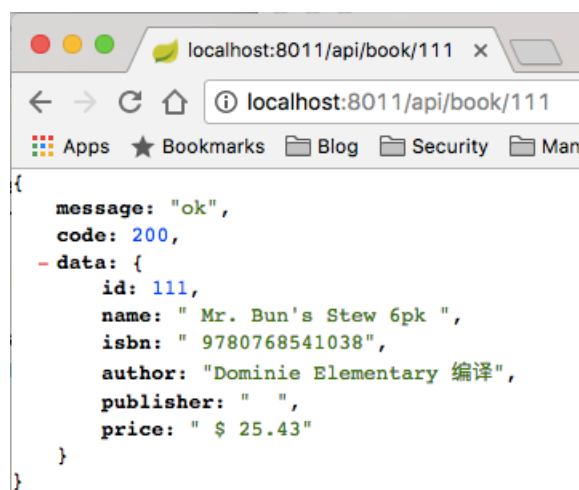


Figure 1.4: Restful 接口返回数据示例

其中, message 可能缩写为 mesg、msg 等, 为了避免歧义, 没有使用缩写, 也没有命名为 errorMessage, 是因为考虑到返回时不仅仅只有错误消息。针对不同的

接口返回不同的字段，可以使用 MapperFacade 的 Mapper 方法进行映射，如下代码片段所示：

```
1 @Autowired
2 private MapperFacade corporationMapperFacade;
3
4 public XydmResponse mapToCorporationResponse(
5     Corporation corporation) {
6     return corporationMapperFacade.map(corporation,
7         XydmResponse.class);
8 }
```

1.2.3 接口认证 (Authorization)

目前接口认证采用根据 TIMESTAMP、ECHOSTR、APPKEY、APPID 生成的 Token，将之发送到服务端，服务端采用同样的加密方式生成 Token 来进行比对认证。但是目前未做 Token 时间过期判断，容易遭受重放攻击 (Replay Attack²)。而且从流量中是可以轻松拿到请求头，对于专业选手来说，相当于大门敞开。采用如下 Shell 脚本³生成 SHA1 串。

```
1 # Mac 生成 sha256 串
2 echo -n "foobar" | shasum -a 256
3 # Mac 生成 sha1 串
4 echo -n "foobar" | shasum -a 1
5 readonly TOKEN = echo -n "foobar" | openssl dgst -
6     sha256
```

脚本的目的是要使用不同的时间，随机生成的字符串，保证每次请求的 Token 都需要不一样，相当于一次一密。使用如下脚本生成随机 token：

```
1 #!/bin/bash
2
3 #
4 # 认证请求脚本
```

²https://en.wikipedia.org/wiki/Replay_attack

³风格参考 Google Shell 风格指南：<http://zh-google-styleguide.readthedocs.io/en/latest/google-shell-styleguide/formatting/>

```

5 # 根据时间戳生成请求，每次请求使用不同 Token
6 # 避免重放攻击 (Replay Attack)
7 # Author:
8 # dolphin
9 # Globals:
10 # APP_ID:
11 # Arguments:
12 # None
13 # Returns:
14 # 接口调用返回结果
15 # TODO: 根据不同的随机串排序，生成不同 Token
16 #
17
18 readonly APP_ID="h1b1451j6d136334gh2"
19 readonly APP_KEY="test"
20
21 ECHOSTR='head -n 1 /dev/urandom|sed 's/[^a-z0-9]//g'|
           md5sum|awk '{print $1}''
22 CURRENT_TIME='date +%Y-%m-%d %H:%M:%S'
23
24 #
25 # 生成 TOKEN
26 # h1b1451j6d136334gh2 接口各项认证参数的排列顺序是:
27 # 时间戳 (timestamp)、AppKey、随机字符串 (echostr)
28 # $1 接收来自命令行传入的参数
29 # 第二个参数用 $2 接收，以此类推
30 # $@ 表示所有的参数，相当于 Java 中 main 函数中定义的
    参数数组
31 #
32 SORTED_TOKEN='echo -n ${CURRENT_TIME}_${APP_ID}_${
           APP_KEY}_${ECHOSTR}|tr '_' '\t'|sort|tr -d
           '\011''
33 TOKEN='echo -n ${APP_ID}${APP_KEY}${CURRENT_TIME}${
           CONST_STR}|md5sum|awk '{print $1}''
34 SEQUENCE_TOKEN='echo -n ${CURRENT_TIME}${APP_ID}${
           APP_KEY}${CONST_STR}|shasum -a 1|awk '{print

```

```

35         $1}''
36 # 请求服务端
37 COMMAND='curl -H "APPID:$APP_ID" -H "TIMESTAMP:
           $CURRENT_TIME" -H "ECHOSTR:$CONST_STR" -H "
           TOKEN:$SEQUENCE_TOKEN" \
38 http://localhost:28080/api/blacklist/page?xdr= | jq
           , , '

```

将 APPID、APPKEY、时间戳，随机字符串采用 SHA1 加密后，生成 Token 发送给服务端，服务端再按照同样的步骤产生 Token，两个 Token 进行比对，这里的关键是 APPKEY 只有服务端和客户端知道，其他非法用户不知道 APPKEY，无法伪造 Token。echo 加上 -n(newline) 参数表示不换行输出。Linux 中的随机数可以从两个特殊的文件中产生，一个是 /dev/urandom。另外一个为 /dev/random。他们产生随机数的原理是利用当前系统的熵池来计算出一定数量的随机比特，然后将这些比特作为字节流返回。熵池就是当前系统的环境噪音，熵指的是一个系统的混乱程度，系统噪音可以通过很多参数来评估，如内存的使用，文件的使用量，不同类型的进程数量等等。如果当前环境噪音变化的不是很剧烈或者当前环境噪音很小，比如刚开机的时候，而当前需要大量的随机比特，这时产生的随机数的随机效果就不是很好了。这就是为什么会有 /dev/urandom 和 /dev/random 这两种不同的文件，后者在不能产生新的随机数时会阻塞程序，而前者不会 (ublock)，当然产生的随机数效果就不太好了，这对加密解密这样的应用来说就不是一种很好的选择。/dev/random 会阻塞当前的程序，直到根据熵池产生新的随机字节之后才返回，所以使用 /dev/random 比使用 /dev/urandom 产生大量随机数的速度要慢。

不过在接口中还多了一个排序的步骤，就是将 APPID、APPKEY、时间戳，随机字符串按照一定的规则排序，因为随机字符串每次都是不同的，那么加密的 APPID、APPKEY、时间戳，随机字符串顺序也会相应的发生变化。在 Shell 中，采用如下命令对 APPID、APPKEY、时间戳，随机字符串进行排序：

```

1 # 测试排序语句
2 echo -n 'b a'|tr ' ' '\t'|sort|tr -d
   '\040\011\012\015'
3 # 实际排序语句
4 aSEQUENCE_TOKEN='echo -n ${CURRENT_TIME}_${APP_ID}_${
   APP_KEY}_${CONST_STR}|tr ' ' '\t'|sort|tr -d
   '\011'

```

040 表示空格, 011 表示制表符, 012 表示换行符 (carriage returns), 015 表示新行 (newlines)。最后的 `tr` 命令去掉排序后字符串中的制表符, 这样排序后的语句就可以直接使用 `md5sum` 命令生成 Token 了。通过使用 `tr`, 可以非常容易地实现 `sed` 的许多最基本功能。您可以将 `tr` 看作为 `sed` 的 (极其) 简化的变体: 它可以用一个字符来替换另一个字符, 或者可以完全除去一些字符。您也可以用它来除去重复字符。这就是所有 `tr` 所能够做的。`-d` 参数表示删除 (delete)。

1.2.4 日志记录 (Logging)

接口日志记录时, 自己的习惯是将之放在拦截器 (Interceptor) 中而不是单独写在方法中。写在方法里总有一些不可避免的重复, 一旦修改起来许多地方都需要修改, 而且非常容易遗漏。拦截器中缺点是无法很好的支持自定义日志的记录, 而且对系统的性能等有影响。记录日志时需要获取响应 Json 的内容进行记录, 通过反射获取。

获取 Response 返回内容

在拦截器中记录日志时, 有时需要获取 HTTP Response 中的内容, 从图1.5可以看出, 只要获取到 `outputChunk` 的内容即可。

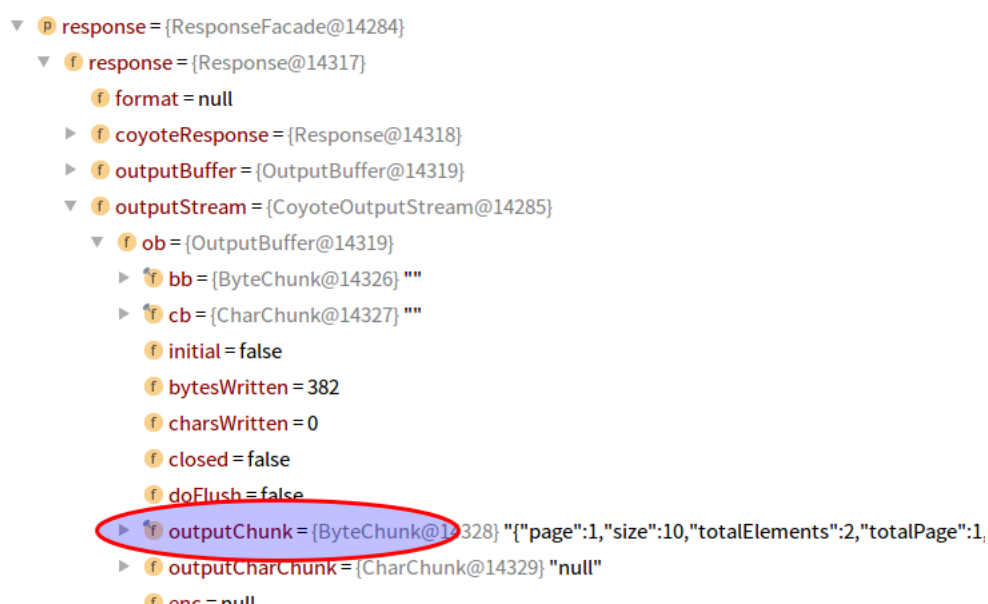


Figure 1.5: 获取 HTTP Response 返回的内容

可以通过反射来实现:

```
public String getResponseContent(HttpServletResponse
    response) throws IOException,
    NoSuchFieldException, IllegalAccessException
```

```
    {
2      String responseContent = null;
3      CoyoteOutputStream outputStream = (
          CoyoteOutputStream) response.getOutputStream
          ();
4      Class<CoyoteOutputStream> coyoteOutputStreamClass
          = CoyoteOutputStream.class;
5      Field obField = coyoteOutputStreamClass.
          getDeclaredField("ob");
6      if (obField.getType().toString().endsWith("
          OutputBuffer")) {
7          obField.setAccessible(true);
8          OutputBuffer outputBuffer = (OutputBuffer)
          obField.get(outputStream);
9          Class<OutputBuffer> opb = OutputBuffer.class;
10         Field outputChunkField = opb.getDeclaredField
          ("outputChunk");
11         outputChunkField.setAccessible(true);
12         if (outputChunkField.getType().toString().
          endsWith("ByteChunk")) {
13             // 取到 byte 流
14             ByteChunk bc = (ByteChunk)
          outputChunkField.get(outputBuffer);
15             // 最终的值
16             responseContent = new String(bc.getBytes
          (), "UTF-8");
17         }
18     }
19     return responseContent;
20 }
```

OutputBuffer 的命名空间为：org.apache.catalina.connector。在获取响应 Json 时，有时 Json 数据会比较长，一个无法避免的问题是只能获取到部分 Json 数据，那是因为在 HTTP 传输时，动态传输内容时，将返回内容以 chunk 的形式进行传输。HTTP 1.1 时，Response 要嘛通过 Content-Length 来指定要传输的内容大小，要嘛通过 Transfer-Encoding: chunked 来传输动态大小的内容，此时要求 Response 传输

的内容要符合 chunk encoding 的规定。在交互式应用程序中有 1 个问题，就是它并不知道将要传输的数据有多大。在 HTTP1.0 中，服务器就会省略 response 头中的 Content-Length 而持续写数据出去，当服务器挂了的话，它简单地断开连接。而经典的 HTTP 客户端会一直读数据直到碰到 -1（传输结束的标识符）。为了处理这个问题，HTTP1.1 中增加了一个特殊的 header: Transfer-Encoding: chunked，允许响应 response 被分块 chunked。每次向连接写数据的时候会先计算大小，最后在 response 的尾部以一个 0 长度的 chunk 块标志着此次传输的结束。即 HTTP1.1 支持 chunked 编码，它允许 HTTP 消息被分成多块后再进行传输。Chunking 一般用在服务器响应 response 的时候，但是客户端也可以 chunk 大的请求 request。即 Chunk 编码允许服务器在发送完 Header 后，发送更多的 Body 内容。Chunked 编码使用若干个 Chunk 块串连而成，每个 Chunk 块都以一个表明 chunk 块大小的 16 进制数字和一个额外的 CRLF（回车换行）开始，后面跟着指定大小的内容。即每个 Chunk 块分为头部和正文两部分，头部内容指定下一段正文的字符总数（十六进制的数字）和数量单位（一般不写），正文部分就是指定长度的实际内容，两部分之间用回车换行 (CRLF) 隔开。最后以一个长度为 0 的 Chunk 表明本次传输结束。查看 getOutputStream 方法（在 org.apache.catalina.connector 命名空间下），代码如下：

```
1 public ServletOutputStream getOutputStream() throws
    IOException {
2     if (usingWriter) {
3         throw new IllegalStateException(sm.getString(
4             "coyoteResponse.getOutputStream.ise"));
5     }
6     usingOutputStream = true;
7     if (outputStream == null) {
8         outputStream = new CoyoteOutputStream(
9             outputBuffer);
10    }
11    return outputStream;
12 }
```

可以看出实际是从 outputBuffer 中取的内容，而 outputBuffer 的默认长度是 8*1024(在 OutputBuffer 类下):

```
1 public static final int DEFAULT_BUFFER_SIZE =
    8*1024;
```

可以在 `preHandle` 中手动通过代码指定长度来解决这个问题：

```
1  /**
2   * 在记录接口日志时
3   * 由于有时返回的内容过长，在getResponseContent通过反射
      获取到的只是一部分Json
4   * 导致后面Json解析失败
5   * 所以这里手动修改buffer的长度
6   * 使得一次能过获取较大的Json内容
7   * 不过当返回的Json内容超过2048 * 20时
8   * 同样只能获取部分
9   * 默认的长度是8*1024    public static final int
      DEFAULT_BUFFER_SIZE = 8*1024;
10  */
11  response.setBufferSize(2048 * 20);
```

1.2.5 异常处理 (Exception Handle)

在系统中，异常处理一般通过全局异常捕获，这样就几乎不用在代码中使用异常捕获块，过多的异常捕获代码块也会造成代码的污染和重复。在 Spring 中，可以在 class 注解上 `@ControllerAdvice` 来捕获全局异常：

```
1  @ControllerAdvice(basePackages = "creditsystem.web.
      controllers")
2  public class CreditsystemControllerAdvice extends
      ResponseEntityExceptionHandler {
3
4      @ExceptionHandler(HlValidateException.class)
5      @ResponseBody
6      ResponseEntity<?> handleValidateException(
          HttpServletRequest request, HlBaseException
          ex) {
7          HttpStatus status = HttpStatus.OK;
8          return new ResponseEntity<>(ApiResult.error(
              ex.getErrCode(), ex.getErrMsg(), ex.getData()
              ), status);
9      }
```



```
10 }
```

这样在业务逻辑层面遇到异常，只需要抛出异常即可，具体的异常处理流程由全局的异常处理模块来负责。以上代码还有一个问题，在添加了 `basePackages` 的扫描范围限制后，在拦截器中抛出异常时，全局异常处理类无法捕获异常。即使将拦截器的包空间加入到 `basePackages` 中也无效，暂时未找到原因。去掉 `basePackages` 的限制是可以捕获到异常的，这里捕获到异常时，接口返回标准的 `Json` 格式内容，这样调用方就方便解析并处理异常了。`ExceptionHandler` 指定捕获的异常类型，不同的异常类型会进入到不同的方法来处理。

1.2.6 版本管理 (Interface Version Control)

接口升级时，有的时候的修改是会影响到下游调用的，无法做到完全兼容，当升级了接口之后，原来的接口调用逻辑就无法正常使用了。所以，考虑做接口的版本管理，当发布接口时，发布一版新的接口，旧有的接口继续提供服务。升级接口由下游的用户来决定，这样在做不兼容升级时，不会影响原来的接口。

URI Parameter Versioning

将 `Api` 的版本信息放到 `URI` 的参数里：

```
1 http://example.com/api/user?version=v1.1.30
```

这种方式被 Amazon、Google、Netflix 采用，流行度高。客户端可以通过指定参数选择调用服务端的 `Api` 版本，比如服务端提供 `v1`、`v2` 版本，当 `v1` 不可用时，可以在接口调用时指定 `v2` 版本的接口。项目也选用这种实现方式，后端部署多个版本对应部署多个服务，在 `Nginx` 中根据 `URL` 的传入参数路由到不同版本的服务上。

```
1 server{
2     location /api {
3         if($query_string ~ version=v1.1.30){
4             //如果query string中包含"version=v1.1.30"
5             //流量转发到18081端口
6             proxy_pass http://10.10.1.12:18081;
7             proxy_redirect off;
8         }else{
9             proxy_pass http://10.10.1.12:18082;
10            proxy_redirect off;
```



```
11     }  
12 }  
13 }
```

根据 URL 请求的版本参数路由到后端不同版本的服务。`$query_string` 是 Nginx 的内置变量。`~` 表示区分大小写的正则匹配, `~*` 表示不区分大小写的正则匹配。

URI Versioning

将 Api 的版本信息放到 URI 里:

```
1 http://example.com/api/v1.1.30/user
```

这种方式被 Twitter、Facebook⁴采用。

Header Versioning

通过添加自定义请求头来指定 Api 的版本:

```
1 x-ms-version:2017-04-17
```

这种方式被 Microsoft 使用⁵。

1.2.7 接口缓存 (Interface Cache)

由于每次接口请求都需要调用认证信息, 认证数据一旦生成, 变动较少, 所以将接口认证数据缓存。

```
1 @Cacheable(value = "appAccount", key = "#id")  
2 public Optional<AppAccount> findAppAccountById(String  
    id){  
3     return Optional.ofNullable(appAccountMapper.  
        findOne(id));  
4 }
```

⁴<https://developers.facebook.com/docs/marketing-api/versions>

⁵<https://docs.microsoft.com/en-us/rest/api/storageservices/Versioning-for-the-Azure-Storage-Services?redirectedfrom=MSDN>

1.3 数据库操作 (Database Operate)

1.3.1 Could not get JDBC connection

接口有时会查询不出数据，总是报超时错误。查看程序日志，提示 Could not get JDBC Connection。查看了数据库的 Session 数量 (40) 远远未到达最大限制数量 (500)。

```
1 #达梦DB查看最大会话数量
2 select *
3 from v$dm_ini
4 where para_name = 'MAX_SESSIONS'
5
6 #达梦DB查看当前会话数量
7 SELECT count(*)
8 FROM v$sessions;
```

经过分析，是由于连接池的默认设置数量问题，由于配置中对 HikariCP 的连接池数量未做指定，HikariCP 默认的连接池数量 (maximumPoolSize) 是 10，正是由于连接池数量较小，很容易出现获取不到连接而出现超时的现象。This property controls the maximum size that the pool is allowed to reach, including both idle and in-use connections. Basically this value will determine the maximum number of actual connections to the database backend. A reasonable value for this is best determined by your execution environment. When the pool reaches this size, and no idle connections are available, calls to getConnection() will block for up to connectionTimeout milliseconds before timing out. Default: 10. 如下配置 HikariCP 的连接池数量：

```
1 #
2 # Pool 数太少会出现连接超时,Hikari 默认的大小是 10
3 # 用户数 200-3000 连接池目前设置为 200
4 # 调大时综合服务器内存、CPU 负荷进行考虑
5 # 数据库目前最大 Session 数量为 500
6 #
7 spring.datasource.maximumPoolSize=200
```

1.3.2 JDBC 数据库重连 (JDBC Database Reconnect)

在数据库重启后，程序无法重新连接到数据库。程序使用的是 Hikari，Hikari 自带数据库重连机制。可以将连接池到 c3p0，c3p0 连接池本身具有数据库重连机

Table 1.1: 常见数据库连接池

名称	协议	备注
c3p0	LGPL v.2.1	C3P0 是一个开源数据连接池，Hibernate3.0 默认自带的数据库连接池，性能比较稳定。
HikariCP	Apache 2.0	Fast, simple, reliable. HikariCP is a "zero-overhead" production ready JDBC connection pool. At roughly 130Kb, the library is very light.
Druid	Apache 2.0	Druid 是 Java 语言中最好的数据库连接池。Druid 能够提供强大的监控和扩展功能。
DBCP	Apache 2.0	DBCP(DataBase connection pool), 是 apache 上的一个 java 连接池项目, 也是 tomcat 使用的连接池组件。单独使用 dbcp 需要 2 个包: commons-dbcp.jar,commons-pool.jar
BoneCP	Apache 2.0	使用 HikariCP 替代

制。目前常见到连接池如表1.1所示：

Tomcat 在 7.0 以前的版本都是使用 commons-dbcp 做为连接池的实现，但是 dbcp 饱受诟病，原因有：

- dbcp 是单线程的，为了保证线程安全会锁整个连接池
- dbcp 性能不佳
- dbcp 太复杂，超过 60 个类

dbcp 使用静态接口，在 JDK 1.6 编译有问题 dbcp 发展滞后因此很多人会选择一些第三方的连接池组件，例如 c3p0, bonecp, druid。为此，Tomcat 从 7.0 开始引入一个新的模块：Tomcat jdbc pool。登陆服务器，使用命令查看正常的数据库连接：

```
lsof -i:5236
```

在程序启动时，并没有真正建立 TCP 连接。只有真正的接受到请求查询数据库时，程序才与数据库建立 TCP 连接。此处建立了 10 个 TCP 连接。而 Hikari 默认的最大连接池数量也为 10 个。如下配置设启连接池连接池时，初始建立的连接数量为 5 个：

```
1 spring.datasource.initial-size=5
```

在 Spring 中配置数据源类型如下：

```
1 <!-- 指定数据源类型 -->
2 spring.datasource.type=com.zaxxer.hikari.
    HikariDataSource
```

Hikari 默认的数据源默认的连接池数量为 10 个，默认的数量在 HikariConfig 类中进行的设置。为什么修改 Spring 的 Initial Size 数量会影响 Hikari 的连接数量呢？如果不配置数据源，Spring Boot 默认的数据源是：

```
1 org.apache.tomcat.jdbc.pool.DataSource
```

但是在实际开发中，可能需要使用自己是熟悉的数据源或者其他性能比较高的数据源。此时就可以通过制定 Spring 的数据源类型来实现。

停止数据库：

```
1 nohup sudo /opt/dmdbms/bin/dmserver /opt/dmdbms/data/
    DAMENG/dm.ini -noconsole &
```

在反复研究后发现，程序并未使用 HikariCP 连接池，而是使用的 Spring JDBC 连接池，所以在初始化 Datasource 时指定连接池，如下代码片段所示：

```
1 @Configuration
2 @Data
3 @ConfigurationProperties(prefix = "spring.datasource"
4     )
5 public class DataSourceConfig {
6     private String jdbcUrl;
7
8     private String username;
9
10    private String driverClassName;
11
12    private String password;
```

```
13
14     @Bean
15     @Primary
16     public DataSource primaryDataSource() {
17         HikariConfig hikariConfig = new HikariConfig
18             ();
19         hikariConfig.setDriverClassName(
20             driverClassName);
21         hikariConfig.setJdbcUrl(jdbcUrl);
22         hikariConfig.setUsername(username);
23         hikariConfig.setPassword(password);
24         hikariConfig.setMaximumPoolSize(5);
25         hikariConfig.setConnectionTestQuery("SELECT 1
26             ");
27         hikariConfig.setPoolName("springHikariCP");
28         hikariConfig.addDataSourceProperty("
29             dataSource.cachePrepStmts", "true");
30         hikariConfig.addDataSourceProperty("
31             dataSource.prepStmtCacheSize", "250");
32         hikariConfig.addDataSourceProperty("
33             dataSource.prepStmtCacheSqlLimit", "2048");
34         hikariConfig.addDataSourceProperty("
35             dataSource.useServerPrepStmts", "true");
36         HikariDataSource dataSource = new
37             HikariDataSource(hikariConfig);
38         return dataSource;
39     }
40 }
```

1.3.3 HikariCP 数据库连接

在配置 HikariCP 数据库连接时，总是提示如下错误：

```
1 org.springframework.beans.factory.
    BeanCreationException: Error creating bean
    with name 'dataSource' defined in class path
```

```
resource [org/springframework/boot/
autoconfigure/jdbc/DataSourceConfiguration$
Hikari.class]: Bean instantiation via factory
method failed; nested exception is org.
springframework.beans.
BeanInstantiationException: Failed to
instantiate [com.zaxxer.hikari.
HikariDataSource]: Factory method 'dataSource
' threw exception; nested exception is java.
lang.NullPointerException
```

经过源码跟踪,发现是 url 为空导致的这个问题,后来增加了 url 的配置才得以成功连接数据库。所以除了要定义 jdbc-url 外,还需要定义 url。经过查看 HikariCP 的文档,属性 jdbc-url 将使 HikariCP 使用“基于驱动管理器”(DriverManager-based)的配置。一般情况下,基于数据源(DataSource-based)的配置是更好的选择。但对许多部署实例来讲却也区别不大。当使用 jdbc-url 来配置“旧”的 JDBC 驱动时,你可能也需要设置 driverClassName 属性,但可以试一试不设置是否能行得通。所以,最终连接数据库的配置如下:

```
1 spring.datasource.hikari.jdbc-url=jdbc:mysql:
    //10.0.0.14:3306/dolphin
2 spring.datasource.hikari.url=jdbc:mysql://10.0.0.14
    :3306/dolphin
3 spring.datasource.hikari.username=root
4 spring.datasource.hikari.password=dolphin
5 spring.datasource.hikari.type=com.zaxxer.hikari.
    HikariDataSource
6 spring.datasource.hikari.driver-class-name=com.mysql.
    jdbc.Driver
```

在某些安全定义较为严格的网络,如果数据库连接经过一定时间无通信,会自动断开连接。可以设置 HikariCP:

```
1 hikariConfig.setConnectionTestQuery("SELECT 1");
```

This is the query that will be executed just before a connection is given to you from the pool to validate that the connection to the database is still alive.

DataSource 跟 DriverManager 区别及联系

In basic terms, a data source is a facility for storing data. It can be as sophisticated as a complex database for a large corporation or as simple as a file with rows and columns. A data source can reside on a remote server, or it can be on a local desktop machine. Applications access a data source using a connection, and a DataSource object can be thought of as a factory for connections to the particular data source that the DataSource instance represents.

两者相同点：Using a DataSource object is the preferred alternative to using the DriverManager for establishing a connection to a data source. They are similar to the extent that the DriverManager class and DataSource interface both have methods for creating a connection, methods for getting and setting a timeout limit for making a connection, and methods for getting and setting a stream for logging.

两者不同点：1.Unlike the DriverManager, a DataSource object has properties that identify and describe the data source it represents. Also, a DataSource object works with a Java™ Naming and Directory Interface™ (JNDI) naming service and is created, deployed, and managed separately from the applications that use it. A driver vendor will provide a class that is a basic implementation of the DataSource interface as part of its JDBC 2.0 or 3.0 driver product.

2.The second major advantage is that the DataSource facility allows developers to implement a DataSourceclass to take advantage of features like connection pooling and distributed transactions.

传统的数据库连接方式（指通过 DriverManager 和基本实现 DataSource 进行连接）中，一个数据库连接对象均对应一个物理数据库连接，数据库连接的建立以及关闭对系统而言是耗费系统资源的操作，在多层结构的应用程序环境中这种耗费资源的动作对系统的性能影响尤为明显。

在多层结构的应用程序中通过连接池（connection pooling）技术可以使系统的性能明显得到提高，连接池意味着当应用程序需要调用一个数据库连接的时，数据库相关的接口通过返回一个通过重用数据库连接来代替重新创建一个数据库连接。通过这种方式，应用程序可以减少对数据库连接操作，尤其在多层环境中多个客户端可以通过共享少量的物理数据库连接来满足系统需求。通过连接池技术 Java 应用程序不仅可以提高系统性能同时也为系统提高了可测量性。

数据库连接池是运行在后台的而且应用程序的编码没有任何的影响。此中状况存在的前提是应用程序必须通过 DataSource 对象（一个实现 javax.sql.DataSource 接口的实例）的方式代替原有通过 DriverManager 类来获得数据库连接的方式。一个实现 javax.sql.DataSource 接口的类可以支持也可以不支持数据库连接池，但是两者获得数据库连接的代码基本是相同的。

1.3.4 Spring Boot 缓存

SpringBoot 支持很多种缓存方式：redis、guava、ehcache、jcache 等等。出现 At least one cache should be provided per cache operation 错误时，表示没有指定 Cache 缓存框架，需要在 XML 中配置。EhCache 是一个纯 Java 的进程内缓存框架，具有快速、精干等特点，是 Hibernate 中默认的 CacheProvider。ehcache 提供了多种缓存策略，主要分为内存和磁盘两级，所以无需担心容量问题。

@Cacheable：Spring 在每次执行前都会检查 Cache 中是否存在相同 key 的缓存元素，如果存在就不再执行该方法，而是直接从缓存中获取结果进行返回，否则才会执行并将返回结果存入指定的缓存中。

@CacheEvict：清除缓存。

@CachePut：@CachePut 也可以声明一个方法支持缓存功能。使用 @CachePut 标注的方法在执行前不会去检查缓存中是否存在之前执行过的结果，而是每次都会执行该方法，并将执行结果以键值对的形式存入指定的缓存中。

这三个方法中都有两个主要的属性：value 指的是 ehcache.xml 中的缓存策略空间；key 指的是缓存的标识，同时可以用 # 来引用参数。在 ehcache.xml 中配置缓存策略空间：

```
1 <ehcache>
2   <cache name="summaryXzxk" maxBytesLocalHeap="1m"
      timeToLiveSeconds="200"/>
3 </ehcache>
```

在方法上指定缓存，语法采用 SpEL(Spring Express Language)，Spring 表达式语言全称为 Spring Expression Language(缩写为 SpEL)，能在运行时构建复杂表达式、存取对象图属性、对象方法调用等等，并且能与 Spring 功能完美整合，如能用来配置 Bean 定义。表达式语言给静态 Java 语言增加了动态功能。SpEL 是单独模块，只依赖于 core 模块，不依赖于其他模块，可以单独使用。：

```
1 @Cacheable(value = "summaryXzxk",
2 key = "'findPage'+#pageable.page+#pageable.params.get
      ('xdr')",
3 condition = "#pageable.params.get('xdr') eq null or #
      pageable.params.get('xdr') eq ''"
4 )
```

@Cacheable 注解有三个参数，value 是必须的，还有 key 和 condition。第一个参数，也就是 value 指明了缓存将被存到什么地方。其中 summaryXzxk 为定义在

配置文件中的缓存器名称。缓存的 Key, 当我们没有指定该属性时, Spring 将使用默认策略生成 key(表示使用方法的参数类型及参数值作为 key), key 属性是用来指定 Spring 缓存方法的返回结果时对应的 key 的。该属性支持 SpringEL 表达式。我们还可以自定义策略: 自定义策略是指我们可以通过 Spring 的 EL 表达式来指定我们的 key。这里的 EL 表达式可以使用方法参数及它们对应的属性。使用方法参数时我们可以直接使用“# 参数名”或者“#p 参数 index”。key 的生成策略有两种: 一种是默认策略, 一种是自定义策略。默认的 key 生成策略是通过 KeyGenerator 生成的, 其默认策略如下: 1. 如果方法没有参数, 则使用 0 作为 key。2. 如果只有一个参数的话则使用该参数作为 key。3. 如果参数多余一个的话则使用所有参数的 hashCode 作为 key 满足一定条件才缓存, 写法如下:

```
condition = "#pageable.params.get('xdr') eq null or #  
            pageable.params.get('xdr') eq ''"
```

如上条件, 取出 xdr 为空的数据进行缓存。如下情形缓存将会不起作用:

- 同一个 bean 内部方法调用, 这个问题遇到过, 也是查了一些时间才发现
- 子类调用父类中有缓存注解的方法

1.3.5 用户重复登录问题

在服务端维护登录用户与其 SessionID 的对应关系。同一用户, 后登陆者将前登陆者的 SessionID 替换, 这样, 前者如果再请求服务器资源, 无法与新生成的 Session ID 匹配, 此时服务端会返回一个错误码, 前端捕获到此错误码, 就会认为其未登录, 要求重新登录。这样, 即使出现断电等无法注销的情况, 也会在他下次登陆时自动取代掉, 同时也解决了用户登陆的唯一性问题。

1.3.6 读取 properties 属性

读取自定义属性

有时需要读取自定义属性, 以配置 HikariCP 连接池为例, 在 application.properties 文件指定指定配置:

```
spring.datasource.hikari.jdbc-url=jdbc:dm://dn4:5236/  
DMSERVER
```

在类中获取配置相应的值, 注意需要添加 Data 注解, 否则需要手写 get 和 set 方法:

```
@Configuration
```

```
2 @Data
3 @ConfigurationProperties(prefix = "spring.datasource.
    hikari")
4 public class DataSourceConfig {
5     private String jdbcUrl;
6 }
```

由于此处注解是默认写在 application.properties 配置文件中，所以在 ConfigurationProperties 中可以不指定路径。否则需要使用 locations 指定配置文件路径。

1.4 单元测试 (Unit Test)

1.4.1 Gradle 中执行单元测试

项目使用 Gradle 进行构建，写好单元测试类后，在 Gradle 中做如下配置：

```
1 project(':cc-api') {
2     test{
3         include('**/*Test.class')
4     }
5 }
```

在每次构建时都会运行单元测试。Mock 和 Stub 是两种测试代码功能的方法。Mock 侧重于对功能的模拟。Stub 侧重于对功能的测试重现。比如对于 List 接口，Mock 会直接对 List 进行模拟，而 Stub 会新建一个实现了 List 的 TestList，在其中编写测试的代码。强烈建议优先选择 Mock 方式，因为 Mock 方式下，模拟代码与测试代码放在一起，易读性好，而且扩展性、灵活性都比 Stub 好。

1.4.2 TestNG 单元测试

项目经常涉及到修改，水平不够足，有些地方始终考虑的不够周到，会互相影响，所以考虑添加单元测试。系统前后端分离，分为各类接口，根据不同类的接口分组进行测试。在测试类上指定测试接口的分组：

```
1 /**
2  * 组名是数组类型，， 所以一个测试方法可以属于多个组
3  * group={"group name 1","group name 2"}
4  */
5 @Test(groups = "xycq-api")
```

```

6 public void testGetXzcfList() throws Exception {
7     TestPublicVariable.test("/xzcf?xdr=dolphin");
8 }

```

配置跑单元测试如图1.6所示。

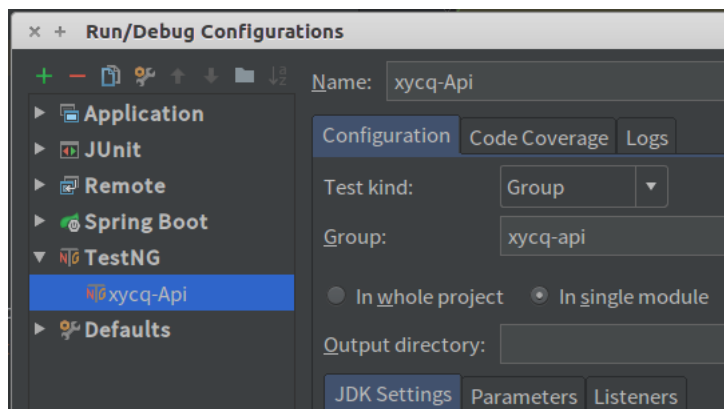


Figure 1.6: TestNG 单元测试

1.4.3 Spring mockMvc 测试

MockMvc 实现了对 Http 请求的模拟，能够直接使用网络的形式，转换到 Controller 的调用，这样可以使得测试速度快、不依赖网络环境，而且提供了一套验证的工具，这样可以使得请求的验证统一而且很方便。引入在 Gradle 构建脚本中增加 Spring 测试包：

```

1 compile("org.springframework.boot:spring-boot-test:"
2         + springBootVersion)
3 testCompile group: 'junit', name: 'junit', version: '
4         4.12'

```

增加测试类：

```

1 @RunWith(SpringRunner.class)
2 @SpringBootTest(classes = DolphinApplication.class)
3 @Commit
4 @Transactional
5 public class BookControllerTest {
6
7     private MockMvc mockMvc;

```

```
8
9     @Autowired
10    private WebApplicationContext wac;
11
12    @Before
13    public void setUp() {
14        this.mockMvc = webApplicationContextSetup(this.wac).
15            build();
16    }
17
18    @Test
19    public void getBooksByName() throws Exception {
20        String responseString = mockMvc.perform(
21            get("/api/
22            book?name=小学生")
23                .andExpect(
24                    status().isOk())
25                .andDo(print
26                    ())
27                .andReturn()).
28            getResponse().getContentAsString();
29        System.out.println("-----返回的json = " +
30            responseString);
31    }
32 }
```

1.5 常见问题 (Frequently Encounted Question)

getAttribute: Session already invalidated

当用户多次登录后，由于服务端缓存了所有的 Session，在取出 Session 进行对比的时候，会出现此错误，此错误指示从 Session 中获取属性值的时候，Session 已经无效。有两种可能导致 Session 无效：session timeout 或者程序中调用了 session.invalidate() 方法。检查代码发现，在移除 Session 时确实使用了：

```
1 // 使 Session 无效的语句
2 getSessionMap().get(sessionId).invalidate();
```

```
3 // 出错的语句
4 User user = (User) session.getAttribute("
    LOCAL_CLINET_USER");
```

当在无效的 session 中获取 Attribute 时, 出现了上面的错误, 解决办法就是不调用 invalidate 方法。

java.lang.NoClassDefFoundError: org/apache/juli/logging/LogFactory

java.lang.NoClassDefFoundError, 这个报错是因为 Java 虚拟机在编译时能找到合适的类, 而在运行时不能找到合适的类导致的错误. 可以查看目标目录, 是否有编译成功的 example.class 文件存在. 在 IntelliJ Idea 中查看文件夹是 module-name/build 文件夹, 如图1.7所示, 可以将其删除, 删除之后即可触发重新编译类。

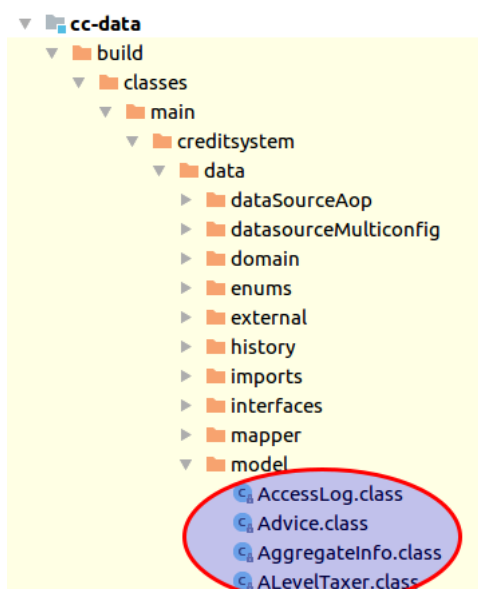


Figure 1.7: 构建.class 文件输出目录

Tomcat 的内部日志使用 JULI 组件, 这是一个 Apache Commons 日志的重命名的打包分支, 默认被硬编码, 使用 java.util.logging 架构。这能保证 Tomcat 内部日志与 Web 应用的日志保持独立, 即使 Web 应用使用的是 Apache Commons Logging。在 Gradle 构建脚本中添加如下配置即可:

```
1 dependencies{
2     compile group: 'org.apache.tomcat', name: 'tomcat
3         -juli', version: property('tomcat.version')
4 }
```

连接时从内存找不到需要的 class 就出现 `NoClassDefFoundError`，当 Java 虚拟机或 `ClassLoader` 实例试图在类的定义中加载（作为通常方法调用的一部分或者作为使用 `new` 表达式创建的新实例的一部分），但无法找到该类的定义时，抛出此异常。

Java 事务的类型有三种：JDBC 事务、JTA(Java Transaction API) 事务、容器事务。常见的容器事务如 Spring 事务，容器事务主要是 J2EE 应用服务器提供的，容器事务大多是基于 JTA 完成，这是一个基于 JNDI 的，相当复杂的 API 实现。在本机请求会报此错误，但是部署到服务器上 OK 的。在本地引入如下 2 个 jar 包即可修复此问题：

```
1 compile group: 'javax.transaction', name: 'jta',  
    version: '1.1'  
2 compile group: 'ma.glasnost.orika', name: 'orika-core',  
    version: '1.5.1'
```

Orika 是一个 Java Bean 映射框架，递归复制数据从一个对象到另一个。对于开发多层应用程序时，非常有用。Java 事务编程接口（JTA：Java Transaction API）和 Java 事务服务（JTS；Java Transaction Service）为 J2EE 平台提供了分布式事务服务。分布式事务（Distributed Transaction）包括事务管理器（Transaction Manager）和一个或多个支持 XA 协议的资源管理器（Resource Manager）。我们可以将资源管理器看做任意类型的持久化数据存储；事务管理器承担着所有事务参与单元的协调与控制。JTA 事务有效的屏蔽了底层事务资源，使应用可以以透明的方式参与到事务处理中；但是与本地事务相比，XA 协议的系统开销大，在系统开发过程中应慎重考虑是否确实需要分布式事务。若确实需要分布式事务以协调多个事务资源，则应实现和配置所支持 XA 协议的事务资源，如 JMS、JDBC 数据库连接池等。如果是引用包里面的类的路径找不到（例如：`org/Spring/Application`），那么就可以先查看项目的 `CLASSPATH` 是否包含引用包的路径：

```
1 //打印项目的 CLASSPATH  
2 ClassLoader classLoader = ClassLoader.  
    getSystemClassLoader();  
3 URL[] urls = ((URLClassLoader) classLoader).getURLs()  
    ;  
4 for (URL url : urls) {  
5     System.out.println(url.getFile());  
6 }
```

可以看到 CLASSPATH 路径里面没有包含引用包的路径, 所以会提示找不到类。

no Scala SDK found in module

项目编译的时候提示 module 里没有找到 Scala SDK, 因为 module 没有指定 Scala SDK(或者是原来的配置由于某种原因失效了)。此时可以将 Scala SDK 添加到指定 module 里, File->Project Structure->Project Settings->Modules, 在对应的 module 上右键添加 Scala SDK, 如图1.8所示。

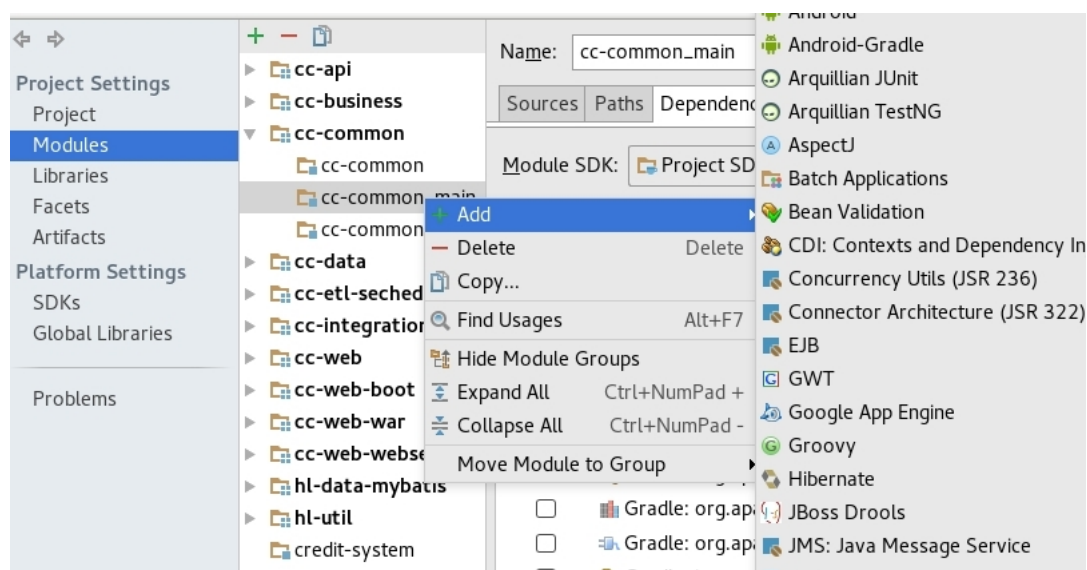


Figure 1.8: IntelliJ Idea 中添加 Scala SDK

添加了 Scala SDK 后, 编译的时候就可以编译 module 里面的 scala 文件了。

Unregistering JMX-exposed beans on shutdown

运行之后控制台输出 “Unregistering JMX-exposed beans on shutdown”, 原因: SpringBoot 内置 Tomcat 没有正常启动. 原来是在引入时没有写版本号, 在没有版本号的情况下此次引入其实是不成功的, 难怪在类中始终不能够自动导入对应的命名空间。

```
compile('org.springframework.boot:spring-boot-starter-web:1.5.2.RELEASE')
```

忘记了写: 1.5.2.RELEASE, 其实这里的版本号可以省略 (应该是不能省略才是), 但是目前还不清楚具体的省略机制, 在什么情况下可以省略。

AES 加密失败

因为某些国家的进口管制限制，Java 发布的运行环境包中的加解密有一定的限制。比如默认不允许 256 位密钥的 AES 加解密，解决方法就是修改策略文件。将下载的没有限制的文件替换掉原来的文件即可。替换的路径为：

```
1 /home/hl/local/jdk1.8.0_144/jre/lib/security/  
    local_policy.jar  
2 /home/hl/local/jdk1.8.0_144/jre/lib/security/  
    US_export_policy.jar
```

由于信息安全在军事等方面极其重要，如在第二次世界大战期间，使用了无线电，若是能够成功解密敌方的机密情报，往往预示着战争的胜利，因此美国对加密解密等软件进行了出口限制，JDK 中默认加密的密钥长度较短，加密强度较低，而 UnlimitedJCEPolicyJDK7 中的文件则没有这样的限制，因此为了获得更好的加密强度，需要替换掉那两个文件。如何知道文件是否已经替换成了允许 256 位加密的那个文件呢？只需要看文件的 MD5 即可，在 Mac 下：

```
1 # 切换到需要替换的文件所在目录  
2 /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/  
    Contents/Home/jre/lib/security/  
3 # 查看文件的 MD5 编码  
4 # 前提是需要知道替换后的那个文件的 MD5 编码  
5 # 与当前生成的 MD5 编码的文件比对即可  
6 MD5 local_policy.jar  
7 # 在 Ubuntu 下使用此命令查看 MD5  
8 md5sum local_policy.jar
```

1.5.1 java.lang.OutOfMemoryError: Java heap space

内存溢出是指应用系统中存在无法回收的内存或使用的内存过多，最终使得程序运行要用到的内存大于虚拟机能提供的最大内存。看到 heap 相关的时候就肯定是堆栈溢出了，此时如果代码没有问题的情况下，适当调整-Xmx 和-Xms 是可以避免的，不过一定是代码没有问题的前提，为什么会溢出呢，要么代码有问题，要么访问量太多并且每个访问的时间太长或者数据太多，导致数据释放不掉，因为垃圾回收器是要找到那些是垃圾才能回收，这里它不会认为这些东西是垃圾，自然不会去回收了；主意这个溢出之前，可能系统会提前先报错关键字为：


```
1 java.lang.OutOfMemoryError:GC overhead limit exceeded
```

这种情况是当系统处于高频的 GC 状态，而且回收的效果依然不佳的情况，就会开始报这个错误，这种情况一般是产生了很多不可以被释放的对象，有可能是引用使用不当导致，或申请大对象导致，但是 java heap space 的内存溢出有可能提前不会报这个错误，也就是可能内存就直接不够导致，而不是高频 GC。JVM 在启动的时候会自动设置 Heap size 的值，其初始空间 (即-Xms) 是物理内存的 1/64，最大空间 (-Xmx) 是物理内存的 1/4。可以利用 JVM 提供的-Xmn -Xms -Xmx 等选项可进行设置。例如：

```
1 nohup $JAVA_HOME/bin/java -Xmx8192M -Xms4096M -jar \  
2 -Xdebug -Xrunjdwp:transport=dt_socket,suspend=n,\  
   server=y,address=5006 \  
3 $APP_PATH/credit-system-web-api-$VERSION.jar \  
4 --spring.config.location=application-api.properties>/\  
   dev/null &
```

以上启动项设置初始 (initial size)Heap Size(Xms) 是 4096MB 最大空间 xmx(maximum size)8192MB。一种查明方法是不间断地监控 GC 的活动，确定内存使用量是否随着时间增加。如果确实如此，就可能发生了内存泄漏。如果发生了内存泄漏，可以使用 JMap 生成 dump 文件以供分析：

```
1 jmap -dump:format=b,file=heap.bin <pid>
```

此种方式在执行时，JVM 是暂停服务的，所以对线上的运行会产生影响。也可以在启动参数上设置发生 OutOfMemory 时生成 dump 文件：

```
1 # 发生 OOM 时生成 dump 文件  
2 # 标准参数 (-)  
3 # 所有的 JVM 实现都必须实现这些参数的功能，而且向后兼容；  
4 # 非标准参数 (-X)  
5 # 默认 jvm 实现这些参数的功能  
6 # 但是并不保证所有 jvm 实现都满足，且不保证向后兼容；  
7 # 非 Stable 参数 (-XX)  
8 # 此类参数各个 jvm 实现会有所不同，将来可能会随时取消  
9 # 需要慎重使用；  
10 -XX:+HeapDumpOnOutOfMemoryError
```

```
11 # dump 文件存放路径
12 -XX:HeapDumpPath=/home/app
```

一般说来，一个正常的系统在其运行稳定后其内存的占用量是基本稳定的，不应该是无限制的增长的。同样，对任何一个类的对象的使用个数也有一个相对稳定的上限，不应该是持续增长的。根据这样的基本假设，我们持续地观察系统运行时使用的内存的大小和各实例的个数，如果内存的大小持续地增长，则说明系统存在内存泄漏，如果特定类的实例对象个数随时间而增长（就是所谓的“增长率”），则说明这个类的实例可能存在泄漏情况。

1.6 MyBatis

mybatis-spring-boot-starter 1.3.x 版本对应 Spring Boot 1.5.x 版本。

1.6.1 打印 SQL(Print SQL)

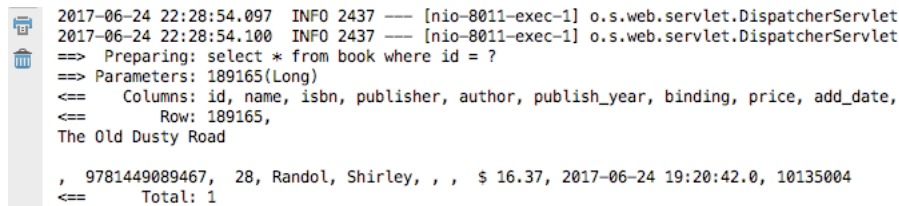
开发中，配置日志中打印 SQL 语句，在 application.properties 文件中指定 mybatis 配置文件路径：

```
1 mybatis.config-location=classpath:mybatis-config.xml
```

在 Mybatis 配置文件 mybatis-config.xml 中，做如下配置：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4 "http://mybatis.org/dtd/mybatis-3-config.dtd">
5 <configuration>
6     <settings>
7         <!-- 打印查询语句 -->
8         <setting name="logImpl" value="STDOUT_LOGGING" />
9     </settings>
10 </configuration>
```

打印 SQL 语句效果如图1.9所示。



```

2017-06-24 22:28:54.097 INFO 2437 --- [nio-8011-exec-1] o.s.web.servlet.DispatcherServlet
2017-06-24 22:28:54.100 INFO 2437 --- [nio-8011-exec-1] o.s.web.servlet.DispatcherServlet
==> Preparing: select * from book where id = ?
==> Parameters: 189165(Long)
<== Columns: id, name, isbn, publisher, author, publish_year, binding, price, add_date,
<== Row: 189165,
The Old Dusty Road
, 9781449089467, 28, Randol, Shirley, , , $ 16.37, 2017-06-24 19:20:42.0, 10135004
<== Total: 1

```

Figure 1.9: MyBatis 打印 SQL

1.6.2 多数据源 (Multi-Datasource)

项目开始的时候是将日志写在主数据库中，过程中考虑到数据库到压力，决定将日志单独放在另一台数据库中。目前还没有到需要使用 MQ 到程度。项目中数据持久化使用的是 MyBatis，所以此时就需要配置 MyBatis 的多数据源支持。多数据源可以采用 AOP 动态切换的方式来实现。一种最简便的方式是初始化多个数据源，在创建 jdbcTemplate 实例时指定某个数据源即可，如下代码片段所示：

```

1 @Autowired
2 @Qualifier("secondaryJdbcTemplate")
3 private JdbcTemplate jdbcTemplate1;

```

那么采用 jdbcTemplate1 执行的 SQL 都在数据源 secondaryJdbcTemplate 上操作，不过麻烦的地方是需要手动写 SQL，但是由于有时采用的 Mapper 操作的数据库，如果可以直接通过注解来指定数据源岂不更加方便。如下是采用自定义注解 + AOP 的方式实现数据源动态切换。动态数据源能进行自动切换的核心就是 spring 底层提供了 AbstractRoutingDataSource 类进行数据源的路由的，我们主要继承这个类，实现里面的方法即可实现我们想要的，这里主要是实现方法：determineCurrentLookupKey()，而此方法只需要返回一个数据库的名称即可，所以我们核心的是有一个类来管理数据源的线程池，这个类才是动态数据源的核心处理类。还有另外就是我们使用 AOP 技术在执行事务方法前进行数据源的切换。

Spring Boot AOP 多数据源 (未成功)

动态数据源能进行自动切换的核心就是 spring 底层提供了 AbstractRoutingDataSource 类进行数据源的路由的，我们主要继承这个类，实现里面的方法即可实现我们想要的，这里主要是实现方法：determineCurrentLookupKey()，而此方法只需要返回一个数据库的名称即可，所以我们核心的是有一个类来管理数据源的线程池，这个类才是动态数据源的核心处理类。还有另外就是我们使用 AOP 技术在执行事务方法前进行数据源的切换。AbstractRoutingDataSource 获取数据源之前会先调用 determineCurrentLookupKey 方法查找当前的 lookupKey，这个 lookupKey 就是数据源标识。因此通过重写这个查找数据源标识的方法就可以让 spring 切换到

指定的数据源了。

- xml<aop> 拦截到数据源名称
- 执行切面 DataSourceAspect 中的 before 方法，将数据源名称放入 DynamicDataSourceHolder 中
- Spring JDBC 调用 determineCurrentLookupKey() 方法 <DynamicDataSource 中重写 AbstractRoutingDataSource 类中的方法>，从 DynamicDataSourceHolder 取出当前的数据库名称，并返回
- AbstractRoutingDataSource 类中 determineTargetDataSource() 方法调用 determineCurrentLookupKey() 匹配到指定的数据库，并建立链接，即为切换到相应的数据库；
- 在指定的数据库中执行相应的 sql

AbstractRoutingDataSource，它的一个作用就是可以根据用户发起的不同请求去转换不同的数据源。若 service 没有启动事务机制，则执行的顺序为：切面——>determineCurrentLookupKey——>Dao 方法。数据源真正切换的关键是 AbstractRoutingDataSource 的 determineCurrentLookupKey() 被调用，此方法是在 open connection 时触发。

1.6.3 不同配置文件

使用不同源配置文件的方式需要多个 datasource、多个 SqlSessionFactory、多个 config 文件。连接不同数据库的 Mapper 文件存放在不同的文件夹下，如何确定配置使用的不同数据源？可以直接查看 Mapper 的 sqlSession 配置，如图1.10所示。

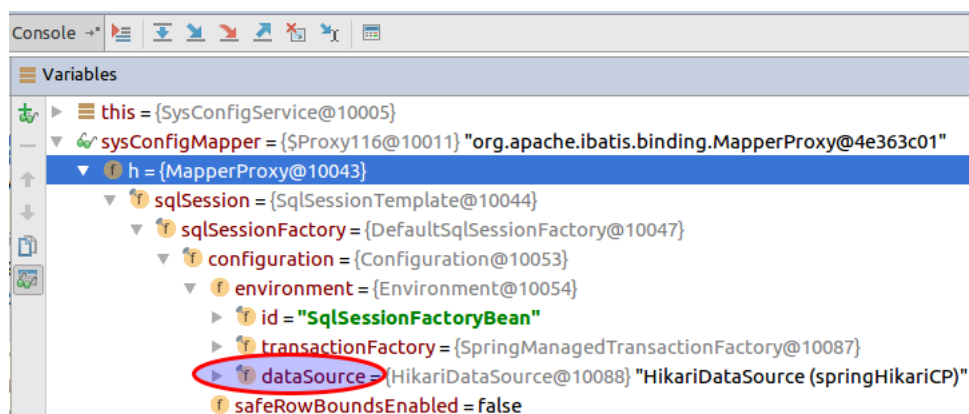


Figure 1.10: 查看 Mapper 使用的 Datasource

1.6.4 多参数传递 (Multi-Parameter)

在使用 Mybatis 的过程中，有时需要传递多个参数，例如既要传入机构 ID 列表，又要传入其他参数，那么此时 Mapper 如下：

```
1 int batchUpdate(@Param("ids") List<String> ids,@Param  
    ("status") String status);
```

在 XML 中使用：

```
1 <update id="batchUpdate" parameterType="java.util.  
    List">  
2     UPDATE TD_S_ORG  
3     SET STATUS = #{status}  
4     WHERE ORG_ID IN  
5     <foreach collection="ids" item="item" index="  
        index" open="(" separator="," close=")">  
6         #{item}  
7     </foreach>  
8 </update>
```

1.6.5 批量写入

在导入 Excel 的过程中，可以利用批量写入功能。可以使写入速度得到质的提高。批量写入在 MyBatis 中作如下配置：

```
1 <insert id="createBatch" parameterType="system.data.  
    model.Reporting">  
2     insert into TI_B_XZXK_REPORTING (  
3         ID,  
4         SFGK,  
5         BGKYY  
6     ) values  
7     <foreach collection="list" item="item" index="  
        index" separator=",">  
8         (  
9             #{item.id},  
10            #{item.sfgk},  
11            #{item.bgkyy}  
12        )  
13    </foreach>
```

```
14 </insert>
```

参数传入 List 即可，注意列表的数量不宜过大，一般数据库对传入的参数有限制，也就是同时一批写入的数据量不能太大。

1.6.6 resultMap 和 resultType 区别

使用 resultType 进行输出映射，只有查询出来的列名和 pojo 中的属性名一致，该列才可以映射成功。如果查询出来的列名和 pojo 中的属性名全部不一致，没有创建 pojo 对象。只要查询出来的列名和 pojo 中的属性有一个一致，就会创建 pojo 对象。mybatis 通过 resultMap 能帮助我们很好地进行高级映射。遇到一个问题是将分页的本应该写成 resultMap 的地方写成了 resultType，在本地可以正常运行，但是打包成 jar 包放到服务器就无法运行了，提示 org.apache.ibatis.type.TypeException: Could not resolve type alias 'AccessLog':

```
1 <!--这里的resultType应该写成resultMap-->
2 <select id="countAccessLog" parameterType="com.
    hualongdata.data.mybatis.Pageable" resultType
    ="AccessLog">
3 </select>
```

如果改为 resultMap 又会出现如下错误：

```
1 org.apache.ibatis.builder.IncompleteElementException:
    Could not find result map creditsystem.data.
    mapper.RedBalckLogMapper.RedBalckLog
```

出现这个错误的原因是在 resultMap 里面指定的是 RedBalckLog(大写)，而 Mapper 中定义的是 redBalckLog(小写)。这里 resultMap 指定的 Bean 需要和 Mapper 文件中定义的 Bean 的 id 一致。

1.6.7 MyBaits 常见问题

Invalid bound statement (not found)

在配置 MyBatis Mapper 时，出现如下错误：

```
1 Request processing failed; nested exception is org.
    apache.ibatis.binding.BindingException:
    Invalid bound statement (not found)
```

配置文件检查了没有问题，后来看网上说是在 IntelliJ Idea 中，需要在 resource 目录下新建一个文件夹，并设置成资源文件夹，放置 Mapper.xml 文件。还真能够解决问题，但是不明白为什么需要单独建立一个文件夹，按道理 resource 也是资源文件夹啊，应该可以读取才对。还有建立的文件夹也有讲究，文件夹的名称应该和放 Mapper.java 文件的名称一致，不能随意起名。问题虽然解决，但是其中原理始终没有弄个明白，最终文件夹结构如图1.11所示。

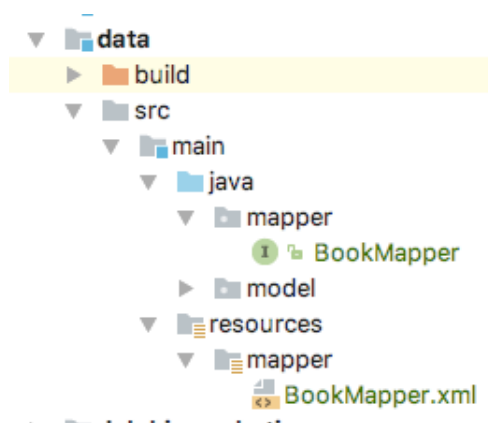


Figure 1.11: Mapper XML 放置目录结构

在配置多数据源时，除了要检查包名是否对应，还要检查不同数据源不要使用同一个包路径。

Parameter 'keys' not found. Available parameters are (list)

在使用 MyBatis 时，出现如下错误：

```
org.apache.ibatis.binding.BindingException: Parameter
    'studentNameList' not found. Available
    parameters are [list]
```

传递一个 List 实例或者数组作为参数对象传给 MyBatis。当你这么做的时候,MyBatis 会自动将它包装在一个 Map 中,用名称在作为键。List 实例将会以 “list” 作为键,而数组实例将会以 “array” 作为键。因为此时传的参数只有一个,而且传入的是一个 List 集合,所以 mybatis 会自动封装成 Map<“list”,keys>。在解析的时候会通过 “list” 作为 Map 的 key 值去寻找。但是在 xml 中却声明成 keys 了,所以自然会报错找不到。解决办法就是修改 Mapper.xml 文件。或者将 List 封装到 Map 里面传递。第一种方式以后如果要扩展该方法,增加集合参数的时候,需要修改 xml 中的内容。

2. Scala

截至 2017 年，采用 Scala 编写的主要有 Spark、Akka。许多企业非关键性组件也会使用 Scala 构建，Scala 相对于 Java 的优势在语法更加简洁，采用完成相同功能代码量更少，综合来看 Scala 生产力相对较高。随着硬件性能不断的提升，性能已经不是主要的考虑因素。

2.1 函数 (Function)

Scala 有两种变量，val 和 var。val 就不能再赋值了。与之对应的，var 可以在它生命周期中被多次赋值。数据转换：

```
1 //字符串转整型
2 val errorCodeId: Int = singleErrorCode.toInt;
3 //split 函数
4 val dstArray: Array[String] = errorCode.toString.
    split(',')
```

```
1 case class User(
2     id: Int,
3     firstName: String,
4     lastName: String,
5     age: Int,
```

```
6     gender: Option[String])
7
8     object UserRepository {
9         private val users = Map(1 -> User(1, "John", "
10             Doe", 32, Some("male")),
11             2 -> User(2, "Johanna", "Doe", 30, None))
12         def findById(id: Int): Option[User] = users.
13             get(id)
14         def findAll = users.values
15     }
```

2.1.1 排序

对 Vector 中的元素做倒序排序。

```
1  /**
2   * 对要排序的列做反向排序 (reverse)
3   * 全文索引列必须要置于条件的开始
4   * 否则无法利用索引
5   * 行政许可XDR列有全文索引，需要排序在前
6   */
7  val params = pageable.getParams.asScala.toVector.
    reverse
```

2.1.2 集合操作

Scala 集合操作如下：

```
1  processResult = errorCode.get.split(',')
2      .filter(s => StringUtils.isNotBlank(s))
3      .map(code => etlErrorTypeService.
4          findById(IntegerId(code.toInt))
5          .filter(_ != null)
6          .map(eet => eet.getErrDesc)
7          .mkString("; "))
```

mkString 把集合元素转化为字符串，可以添加分隔符，前缀，后缀。

3. Python

Python 是 FLOSS（自由/开放源码软件）之一。你可以自由地发布这个软件的拷贝、阅读它的源代码、对它做改动、把它的一部分用于新的自由软件中。Python 希望看到一个更加优秀的人创造并经常改进。由于它的开源本质，Python 已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。如果你小心地避免使用依赖于系统的特性，那么你的所有 Python 程序无需修改就可以在下述任何平台上面运行。这些平台包括 Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE 甚至还有 PocketPC、Symbian 以及 Google 基于 linux 开发的 Android 平台！

3.0.1 基础

查看 Python 已经安装了哪些模块，可以直接在 Python 环境中输入 `help('modules')` 命令。在 Pycharm 中设置 Python 路径在 Preference->Project Interpreter 中。

```
1 # 安装 MySQL
2 sudo apt-get install mysql-server -y
3 # 安装 pip
4 sudo apt-get install python-pip python-dev build-essential
5 # 安装 MySQL 驱动
6 sudo pip install mysql-python
```

```
7 # 安装 beautifulsoup4
8 sudo pip install beautifulsoup4
9 # openpyxl 来处理 xml 文件
10 sudo pip install openpyxl
11 # 安装 Redis 支持模块
12 sudo pip install redis
```

在 Mac OS X 上，提示 No Module named requests，使用 pip 安装 requests 模块也无法解决，输入如下命令此问题消失：

```
1 sudo easy_install -U requests
```

pip 改善了不少 easy_install 的缺點，如此說來 pip 應該是略勝一籌，不過它還不能完全取代對方，因为目前有很多套件還是得用 easy_install 安裝。easy_install 的作用和 perl 中的 cpan，ruby 中的 gem 类似，都提供了在线一键安装模块的傻瓜方便方式，而 pip 是 easy_install 的改进版，提供更好的提示信息，删除 package 等功能。老版本的 python 中只有 easy_install，没有 pip。

3.0.2 Beautiful Soup

Beautiful Soup 是用 Python 写的一个 HTML/XML 的解析器，它可以很好的处理不规范标记并生成剖析树 (parse tree)。它提供简单又常用的导航 (navigating)，搜索以及修改剖析树的操作。Beautiful Soup 提供一些简单的、python 式的函数用来处理导航、搜索、修改分析树等功能。它是一个工具箱，通过解析文档为用户提供需要抓取的数据，因为简单，所以不需要多少代码就可以写出一个完整的应用程序。如下代码片段展示爬取豆瓣书籍详情页的书籍信息：

```
1 def get_single_book_detail_info(url):
2     try:
3         req = urllib2.Request(url, headers=hds[np.random.
4                               randint(0, len(hds))])
5         source_code = urllib2.urlopen(req).read()
6         plain_text = str(source_code)
7     except (urllib2.HTTPError, urllib2.URLError), e:
8         print e
9     soup = BeautifulSoup(plain_text)
10    list_soup = soup.find('div', {'id': 'info'}).findAll
11                    ('span')
```

```
10 return list_soup
```

find 方法直接返回结果，findAll 方法返回符合条件的结果。Python 在使用前，可以不需要初始化变量，在外层可以直接使用内层函数的变量。

3.0.3 爬取豆瓣书籍

向数据库中保存数据：

```
1 def save_to_mysql():
2     try:
3         conn = MySQLdb.connect(host='localhost', user
4         = 'root', passwd='123456', port=3306)
5         cur = conn.cursor()
6         conn.select_db('dolphin')
7         value = ['hi rollen']
8         cur.execute('insert into book(name) values(%s
9         )', value)
10        conn.commit()
11        cur.close()
12        conn.close()
13
14    except MySQLdb.Error, e:
15        print "Mysql Error %d: %s" % (e.args[0], e.
16        args[1])
```

在根据标签爬取的书籍列表中，信息不够全面，比如 ISBN 号码就没有。所以需要爬到每本书籍的详情页面。

3.0.4 抓取技巧

防止 IP 被封

在爬取豆瓣的页面时，如果一分钟请求超过一定次数（一说 40 次），豆瓣会将请求的 IP 加入黑名单。所以一是爬取的频率不要太高，而是可以使用代理来爬取。服务端对 IP 访问对阈值也有限制，因为按照正常情况下，一个用户不会大批量对请求一个网站的，所以超过了规定的请求次数也会触发封锁。还可以让请求不那么有规律，让服务器无法通过请求的频率判断是机器还是人类，一般人类访问请求的频率是不固定的。

模仿浏览器

有些网站会检查你是不是真的浏览器访问，还是机器自动访问的。这种情况，加上 User-Agent，表明你是浏览器访问即可。有时还会检查是否带 Referer 信息还会检查你的 Referer 是否合法，一般再加上 Referer。在此次抓取的过程中，模仿浏览器：

```
1 # Some User Agents
2 headers = [{'User-Agent': 'Mozilla/5.0 (Windows; U;
    Windows NT 6.1; en-US; rv:1.9.1.6) Gecko
    /20091201 Firefox/3.5.6'}, \
3 {
4 'User-Agent': 'Mozilla/5.0 (Windows NT 6.2)
    AppleWebKit/535.11 (KHTML, like Gecko) Chrome
    /17.0.963.12 Safari/535.11'}, \
5 {'User-Agent': 'Mozilla/5.0 (compatible; MSIE 10.0;
    Windows NT 6.2; Trident/6.0)'}]
```

在 HTTP 请求头中，添加浏览器相关标识，让服务器认定为来自浏览器的请求。

避免重复抓取

豆瓣采用 Restful 风格来设计的 API，最后的值就相当于每本书籍的唯一标识，在数据库中预先生成所有的 ID 集合，每抓取了一个 ID 的数据，都会更新 ID 的状态，表示此数据已经抓取。这样的方式来确保每本书籍只被抓取一次。

海量数据

3.0.5 读取 Properties 文件

在项目中持续集成的 Python 部署脚本中，需要读取 Java 的 Properties 文件，读取的方法如下：

```
1 def getVersion():
2     #
3     #动态从版本配置文件中读取当前最新版本号
4     #
5     file_path = VERSION_CONFIG_PATH
6     props = parse(file_path) #读取文件
7     print('version: '+props.get('VERSION'))
```

```
8     #根据 key 读取 value
9     return props.get('VERSION')
10
11 def parse(file_name):
12     return Properties(file_name)
```

Propertise 类的代码如下:

```
1 import re
2 import os
3 import tempfile
4
5 class Properties:
6
7     def __init__(self, file_name):
8         self.file_name = file_name
9         self.properties = {}
10        try:
11            fopen = open(self.file_name, 'r')
12            for line in fopen:
13                line = line.strip()
14                if line.find('=') > 0 and not line.
15                    startswith('#'):
16                    strs = line.split('=')
17                    self.properties[strs[0].strip()] = strs[1].
18                        strip()
19        except Exception, e:
20            raise e
21        else:
22            fopen.close()
23
24    def has_key(self, key):
25        return key in self.properties
26
27    def get(self, key, default_value=''):
28        if key in self.properties:
```

```
27         return self.properties[key]
28     return default_value
```

3.0.6 sh: mysql_config: command not found 解决办法

首先需要安装 MySQL，安装完成后，打开终端，输入下方语句修改环境变量：

```
1 # 安装 python-devel
2 sudo dnf install python-devel -y
3 # 解决 fatal error: Python.h: No such file or
   directory
4 sudo dnf install python-devel --allowdowngrading
5 PATH="$PATH":/usr/local/mysql/bin
```

设置完成之后，输入如下语句安装 MySQL Python：

```
1 sudo pip install mysql-python
```




4. 前端 (Front End)

4.1 React

只有动画或视频达到 60fps，人眼才不会感觉卡顿。

4.1.1 组件生命周期 (Life Cycle)

页面的交互流程，页面加载或者用户操作界面，调用相关函数通过 Ajax 从服务端获取数据，触发 dispatch，将 action 传递给 reducer 更改 state(要赋给 props 的值都存在 state 中)，然后再将 state 的新的值赋给 props，重新渲染页面。connect 中的函数当 state 改变时都会被执行，可以理解成 subscribe 函数或者监听函数。

4.1.2 第一个 React

在 Web 开发中，我们总需要将变化的数据实时反应到 UI 上，这时就需要对 DOM 进行操作。而复杂或频繁的 DOM 操作通常是性能瓶颈产生的原因（如何进行高性能的复杂 DOM 操作通常是衡量一个前端开发人员技能的重要指标）。React 为此引入了虚拟 DOM (Virtual DOM) 的机制：在浏览器端用 Javascript 实现了一套 DOM API。基于 React 进行开发时所有的 DOM 构造都是通过虚拟 DOM 进行，每当数据变化时，React 都会重新构建整个 DOM 树，然后 React 将当前整个 DOM 树和上一次的 DOM 树进行对比，得到 DOM 结构的区别，然后仅仅将需要变化的部分进行实际的浏览器 DOM 更新。如下定义一个最简单的 React 示例，首先写基本的一个程序入口的组件：

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 ReactDOM.render(
5   <div>Hello World!</div>,
6   document.getElementById('root')
7 );
```

随后定义一个默认的 HTML 文件：

```
1 <body>
2   <!-- 容器 -->
3   <div id="root"></div>
4   <!-- 引入编译好的js文件 -->
5   <script type="text/javascript" src="./bundle.js"
6     ></script>
7 </body>
```

4.1.3 React 数据的流转过程

通过 Axios 向服务端发送 HTTP 请求，获取到服务端返回端 Json 数据后，触发对应 Type 的 dispatch(serviceCommon.js) 去更新 state。当 state 变化，或者 ownProps 变化的时候，mapStateToProps(connect 方法的第一个参数) 都会被调用，计算出一个新的 stateProps，（在与 ownProps merge 后）更新给对应的组件。

4.1.4 规范 (Specification)

React 模块使用 .jsx 扩展名，如 BookMain.jsx。对于 JSX 属性值总是使用双引号 (")，其他均使用单引号。为什么？JSX 属性不能包括转译的引号，因此在双引号里包括像 "don't" 的属性值更容易输入。HTML 属性也是用双引号，所以 JSX 属性也遵循同样的语法。

4.1.5 Router

Web 服务并不会解析 hash，也就是说 # 后的内容 Web 服务都会自动忽略，但是 JavaScript 是可以通过 window.location.hash 读取到的，读取到路径加以解析之后就可以响应不同路径的逻辑处理。history 是 HTML5 才有的新 API，可以用来操作浏览器的 session history (会话历史)。在 HTML5 的 history API 出现之前，前端的路由都是通过 hash 来实现的，hash 能兼容低版本的浏览器。React Router 的设计思

想来源于 Ember 的路由，如果原来有用过 Ember 的路由，那么应该对 React Router 不会陌生。React Router 4.0（以下简称 RR4）已经正式发布，它遵循 react 的设计理念，即万物皆组件。所以 RR4 只是一堆提供了导航功能的组件（还有若干对象和方法），具有声明式（引入即用），可组合性的特点。添加一个 Router 组件如下：

```
1 import React from "react";
2 import {BrowserRouter, Route} from 'react-router-dom
  ,
3
4 const User = ({match}) => {
5   return <h1>Hello {match.params.username}!</h1>
6 }
7
8 const routes = (
9   <BrowserRouter>
10     <Route path="/user/:username" component={User
11       }/>
12   </BrowserRouter>
13 );
14 export default routes;
```

`<BrowserRouter>` 是一个使用了 HTML5 history API 的高阶路由组件，保证你的 UI 界面和 URL 保持同步。使用 HTML5 历史记录 API（pushState, replaceState 和 popstate 事件）的 `<Router>` 来保持 UI 与 URL 的同步。`<Route>` 也许是 RR4 中最重要的组件了。它最基本的职责就是当页面的访问地址与 Route 上的 path 匹配时，就渲染出对应的 UI 界面。`<Route>` 自带三个 render method 和三个 props，props 分别是：match、location、history。所有的 render method 无一例外都将被传入这些 props。访问效果如图4.1所示：

传入初始值

项目中需要判断页面是否初次进入，根据是否第一次进入页面还是返回时进入的页面进行不同的逻辑操作，在链接到项目对应模块的地方增加初始参数：

```
1 <Link to={'/profile/redblackLog?operation=init'}>红黑
  名单访问记录</Link>
```

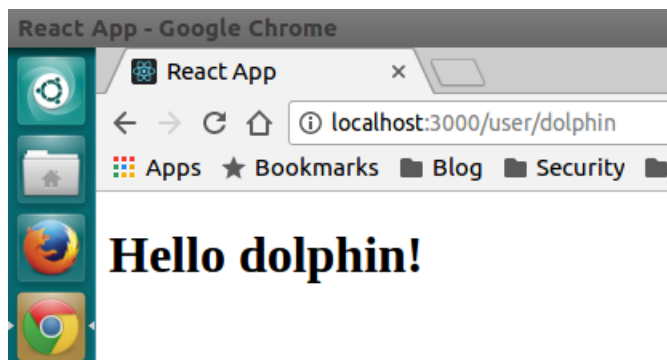


Figure 4.1: React Route 效果

在页面中判断是否第一次进入：

```
1 if (this.props.location.query.operation === "init") {  
2     //第一次进入页面  
3 }
```

4.1.6 Provider

Provider 方法会在 react 组件外边包一层名为 provider 的组件，使其所有子组件共享 store 提供的方法。connect 方法生成容器组件以后，需要让容器组件拿到 state 对象，才能生成 UI 组件的参数。一种解决方法是将 state 对象作为参数，传入容器组件。但是，这样做比较麻烦，尤其是容器组件可能在很深的层级，一级级将 state 传下去就很麻烦。React-Redux 提供 Provider 组件，可以让容器组件拿到 state。

```
1 ReactDOM.render(  
2     <Provider store={store}>{routes}</Provider>,  
3     document.getElementById('root')  
4 );
```

Provider 在根组件外面包了一层，这样一来，App 的所有子组件就默认都可以拿到 state 了。同理 store 只需要在最顶层导入一次，否则在下级的每一个组件都需要导入 store，非常不利于组件复用，每个组件都导入显然不够优雅，它的原理是 React 组件的 context 属性。

4.1.7 Connect

React-Redux 提供 connect 方法，用于从 UI 组件生成容器组件。connect 的意思，就是将这两种组件连起来。

```
1 connect(mapStateToProps, mapDispatchToProps,  
    mergeProps)(MyComponent)
```

项目中 connect 定义如下：

```
1 /**  
2  * mapStateToProps:这个函数的作用是确定哪些 Redux 全局  
    的 state 是我们组件想要通过 props 获取  
3  */  
4 const mapStateToProps = (state) => {  
5     return {  
6         book: state.book  
7     }  
8 }  
9  
10 /**  
11  * mapDispatchToProps:这个方法的作用是确定 哪些action  
    创建函数是我们想要通过props获取  
12  */  
13 const mapDispatchToProps = (dispatch) =>{  
14     return{  
15         bookService: bindActionCreators(bookService,  
            dispatch)  
16     }  
17 }  
18  
19 connect(mapStateToProps, mapDispatchToProps)  
20 class BookContainer extends Component {  
21     render() {  
22         return React.cloneElement(this.props.  
            children, {...this.props});  
23     }  
}
```

```
24 }
```

4.1.8 Actions

Action 向 store 派发指令，action 函数会返回一个带有 type 属性的 Javascript Plain Object，store 将会根据不同的 action.type 来执行相应的方法。

4.1.9 Store

为什么要使用 store？如果不使用 store，那么各个组件的数据必须由各个组件来维护，这种情况会出现一些问题，第一就是组件之间的数据可能会有重复，比如一个组件父组件统计子组件的数据，父组件需要存储子组件的数据，子组件也会存储自己的数据，数据重复又会带来问题，数据之间的维护需要大量的工作（子组件数据更新后父组件的数据也要更新），多份数据的情况下非常容易造成数据不一致。这也就是为什么要使用 store，而且整个应用中只能有一个 store 的原因¹。store 是应用的状态管理中心，保存着是应用的状态（state），当收到状态的更新时，会触发视觉组件进行更新。Provider 是 react-redux 提供的组件，它的作用是把 store 和视图绑定在了一起，如下语句创建一个简单的 store。

```
1 import {createStore} from 'redux';
2 //创建 store
3 const store = createStore(reducer, 1);
```

第一个参数为 reducer，第二个参数是初始化的状态 (initialState)，这里为 1。

4.1.10 Dispatcher

接收 action，并且向注册的回调函数广播 payloads。Facebook 还提供了 Dispatcher 的开源库。你可以将 Dispatcher 认为是一种全局 pub/sub 系统的事件处理器，用于向所注册的回调函数广播 payloads。通常情况下，如果你使用 Flux 架构，你可以借助于 Facebook 提供的这个 Dispatcher 实现，并且可以借助 NodeJS 中的 EventEmitter 模块来配置事件系统，从而帮助管理应用程序的状态。Store 中包括了注册给 Dispatcher 的回调函数。actionCreator 还有很简洁的用法：对 actionCreator 做 dispatch 级别的封装，这个过程我们可以使用 redux 提供的 bindActionCreators 函数自动完成。然后就可以直接调用 action，而不需要使用 dispatch 方法去调用 actionCreator。dispatch 调用如下代码片段所示。

```
1 store.dispatch({
```

¹参考《深入浅出 React 和 Redux》2.5 小节


```
2     type: "ARTICLE_PAGE",
3     payload: 10
4   });
```

dispatch 的参数是一个 Action。如下代码片段定义一个 Action：

```
1 const action = {
2   type: "ARTICLE_PAGE",
3   payload: 10
4 };
```

其中的 type 属性是必须的，表示 Action 的名称。上面代码中，Action 的名称是 ARTICLE_PAGE，它携带的信息是数字 10。

redux-thunk

解决了 dispatch 不好获取的问题。Redux 官网说，action 就是 Plain JavaScript Object。Thunk 允许 action creator 返回一个函数，而且这个函数第一个参数是 dispatch。例如：

```
1 export function findPage(params) {
2   return dispatch =>
3     requestJSON(
4       dispatch,
5       {method: 'get', url: '/pubapi/article/
6         page', params: {...params}},
7       {type: articleType.ARTICLE_PAGE},
8       {type: articleType.ARTICLE_PAGE_ERROR,
9         message: '获取文章列表失败'});
10 }
```

在异步 Action 时也需要使用 thunk 中间件。以上代码片段就是一个更好的例子，以上代码表示向服务器发送一个请求获取文章列表，但是这里返回的是一个函数，而不是一个对象，redux-thunk 的工作就是检查 action 对象是不是函数，如果不是函数就放行，让其执行完整的 action-reducer-view 生命周期。如果发现对象是函数，就执行这个函数，并把 Store 的 dispatch 函数和 getState 函数作为参数传递到函数中去，处理过程到此为止，不会让异步对象再派发到 reducer 函数。异步

调用的 API 再根据合适的时机派发 Action，触发界面 Render²。

4.1.11 Reducer

Store 收到 Action 以后，必须给出一个新的 State，这样 View 才会发生变化。这种 State 的计算过程就叫做 Reducer。Reducer 是一个函数，它接受 Action 和当前 State 作为参数，返回一个新的 State。如下语句创建一个简单的 Reducer：

```
1 const reducer = (state, action) => {  
2   switch (action.type) {  
3     case "ADD":  
4       state = state + action.payload;  
5       break;  
6     case "SUBTRACT":  
7       break;  
8   }  
9   return state;  
10 };
```

4.1.12 state

State (也称为 state tree) 是一个宽泛的概念，但是在 Redux API 中，通常是指一个唯一的 state 值，由 store 管理且由 getState() 方法获得。它表示了 Redux 应用的全部状态，通常为一个多层嵌套的对象。约定俗成，顶层 state 或为一个对象，或像 Map 那样的键-值集合，也可以是任意的数据类型。然而你应尽可能确保 state 可以被序列化，而且不要把什么数据都放进去，导致无法轻松地把 state 转换成 JSON³。state 是 React 中组件的一个对象。React 把用户界面当做是状态机，想象它有不同的状态然后渲染这些状态，可以轻松让用户界面与数据保持一致。React 中，更新组件的 state，会导致重新渲染用户界面 (不要操作 DOM)。

```
1 //会重新渲染页面  
2 this.setState({query: query});  
3 /*  
4  * 不会重新渲染页面  
5  * 调用this.forceUpdate()手动强制重新渲染页面  
6  */
```

²参考《深入浅出 React 和 Redux》7.2.2 节

³来源：<http://cn.redux.js.org/docs/Glossary.html>


```
7 this.setState.query = query;  
8 this.forceUpdate();
```

一般情况下应该用 `setState` 来设置 `state` 值, 不应该对 `state` 直接赋值⁴(It is better to use `this.setState` because it is init native check-state mecanizm of React engine which is better then force update.If you just update any param of `this.state` directly without `setState` react render mecanizm will not know that some params of state is updated.), 对 `state` 直接赋值可以在构造方法中 (The only place where you can assign `this.state` is the constructor). 还有一点需要注意的是 `setState` 方法是异步的, 可以使用回调 (Callback) 函数来获取最终结果:

```
1 //使用回调函数获取设置的值  
2 this.setState({data: data}, function(){  
3     console.log(this.state.data);  
4 }.bind(this));
```

初始化 state

有 2 种方法来初始化应用的 `state`, 一种是在 `createStore` 方法中的第二个可选参数 `preloadState`。也可以在 `reducer` 中为 `undefined` 的 `state` 参数指定默认初始值。`state` 的结构如图4.2所示。

redux logger

Redux Logger 是一个中间件, 它会打印出当前应用的 `state` 变化, 配置如下代码所示。

```
1 /**  
2  * ES6 with npm, you can write import React from '  
3     react'.  
4  * ES5 with npm, you can write var React = require('  
5     react').  
6  */  
7 import {createLogger} from 'redux-logger';  
8  
9 if (process.env.NODE_ENV === 'development') {  
10     const logger = createLogger({
```

⁴<https://reactjs.org/docs/state-and-lifecycle.html>

```

13:41:41.898 ▶XHR finished loading: POST "http://creditsystem.test/inapi/redE
13:41:42.008 ▶ (12) [Object, Object, Object, Object, Object, Object, Object, (
13:41:42.075 action @ 13:41:41.900 RED_BLACK_BATCH_PAGE
13:41:42.076 prev state ▶Object {global: Object, share: Object, maintype:
13:41:42.076 action ▶Object {type: "RED_BLACK_BATCH_PAGE", data: Object}
13:41:42.077 next state
▼Object {global: Object, share: Object, maintype: Object, info:
  ▼article: Object
    ▼ARTICLE_GZDT_PAGE: Object
      ▼content: Array(10)
        ▼0: Object
          browseCount: 107
          category: 14
          content: "<p style='";margin-bottom:0;text-align:cent
          createBy: 1
          createDate: "2017-02-28T14:17:01.541"
          description: "重庆市发改委副主任欧阳林在培训会上通报全市社会
          id: 305
          parentCategory: 1
          status: 1
          title: "重庆正加快信用体系建设步伐 研究信用红黑名单管理办法"
          updateBy: 1
          updateDate: "2017-06-01T12:55:57.889"
          __proto__: Object
        ▶1: Object
        ▶2: Object
        ▶3: Object

```

Figure 4.2: State 状态示例

```

9   stateTransformer: (state) => {
10     let newState = {};
11     for (let i of Object.keys(state)) {
12       if (Iterable.isIterable(state[i])) {
13         newState[i] = state[i].toJS();
14       } else {
15         newState[i] = state[i];
16       }
17     }
18     return newState;
19   }
20 });
21 middleware.push(logger);
22 }

```

打印出 action 的效果如图4.3所示。

从图中看出，state 的状态已经改变。说明是 state 改变后的某些动作还未触发。

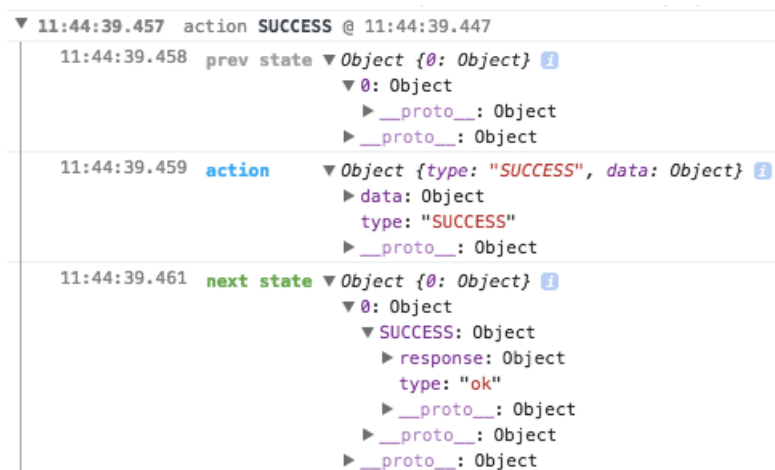


Figure 4.3: Redux Logger 打印效果

4.1.13 Props

当你用 React Redux 的 connect 进行封装的时候，connect 方法会把 dispatch 放到 props 中。

React Router 父子 Props 传递

在 React 中需要将 bookService 由父组件中传递到子组件中，写法如下：

```
1 const routes = (  
2   <BrowserRouter>  
3     <BookContainer>  
4       <switch>  
5         <Route path="/book/:id" render={() => <  
6           Book bookService={bookService}/>}/>  
7       </switch>  
8     </BookContainer>  
9   </BrowserRouter>  
10 );
```

其中 Book 处于内层，负责渲染界面，所以叫做展示组件 (Presentational Container)。BookContainer 与 Book 组件属于父子组件的关系，BookContainer 负责和 Redux Store 打交道，处于外层，称作容器组件 (Container Component)。容器组件的定义如下：

```
1 /**  
2  * 容器组件，负责与 Redux Store 打交道
```

```

3  * 通过props把状态传递给内层组件
4  */
5  class BookContainer extends Component {
6      render() {
7          return React.cloneElement(this.props.
            children, {...this.props});
8      }
9  }

```

在浏览器中，切换到相应组件上，可以看到对应的属性，如图4.4所示。

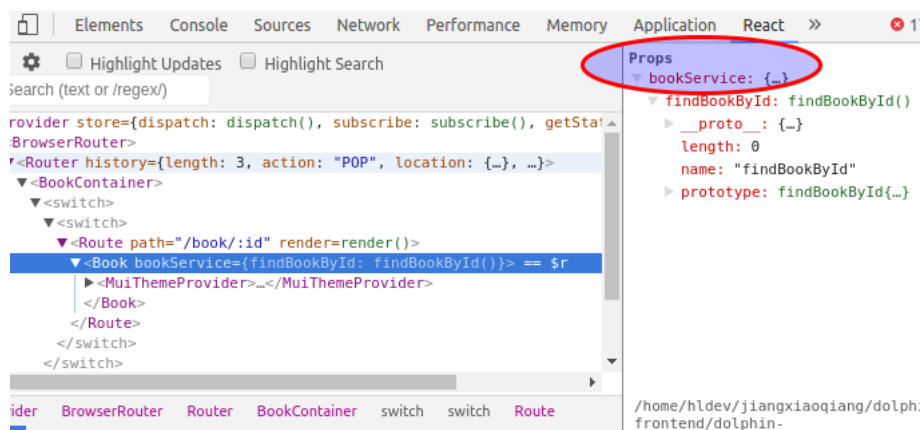


Figure 4.4: React 组件 Props 传递

还可以传递函数，例如有时候在子组件上操作后需要更新父组件的界面，可以将更新方法通过 Props 传递给子组件，子组件直接调用函数进行更新即可。

```

1  /**
2  * refreshFetch是父组件定义的函数
3  */
4  return <FeedbackPanel query={query}
5              refreshFetch={this.
6                  refreshFetch}
7              creditFormTable={
8                  creditFormTable}
9              key={i}/>

```

4.1.14 Reducers

Redux 有且只有一个 State 状态树，为了避免这个状态树变得越来越复杂，Redux 通过 Reducers 来负责管理整个应用的 State 树，而 Reducers 可以被分成一个个 Reducer。

onEnter

Route 的 onEnter 钩子将用于在渲染对象的组件前做拦截操作，比如验证权限。单页应用路由的权限控制的基本思路是：监听路由的改变，每当路由将要发生改变，我们就使用一个中间服务（该服务介于上一级路由和将要到达路由之间启动），来判断我们是否有进入这个路由的权限，有的话直接进入，没有的话就 redirect。在 React 中，为某个路由进行权限监听的方式是：

```
<Route path="page" component={Page} onEnter={
  requireCredentials}/>
```

该 onEnter 属性对应连着一个具有判断权限的中间服务。

4.1.15 Immutable

JavaScript 中的对象一般是可变的（Mutable），因为使用了引用赋值，新的对象简单的引用了原始对象，改变新的对象将影响到原始对象。如 `foo=a: 1; bar=foo; bar.a=2` 你会发现此时 `foo.a` 也被改成了 2。虽然这样做可以节约内存，但当应用复杂后，这就造成了非常大的隐患，Mutable 带来的优点变得得不偿失。为了解决这个问题，一般的做法是使用 `shallowCopy`（浅拷贝）或 `deepCopy`（深拷贝）来避免被修改，但这样做造成了 CPU 和内存的浪费。Immutable 可以很好地解决这个问题。

IMMUTABLE DATA

Immutable Data 就是一旦创建，就不能再被更改的数据。对 Immutable 对象的任何修改或添加删除操作都会返回一个新的 Immutable 对象。Immutable 实现的原理是 Persistent Data Structure（持久化数据结构），也就是使用旧数据创建新数据时，要保证旧数据同时可用且不变。同时为了避免 `deepCopy` 把所有节点都复制一遍带来的性能损耗，Immutable 使用了 Structural Sharing（结构共享），即如果对象树中一个节点发生变化，只修改这个节点和受它影响的父节点，其它节点则进行共享。

4.1.16 前端 Mock

有的时候前端没必要等待后端接口写好，可以直接 Mock 接口的返回数据。这里使用 json-server⁵来实现。

4.2 MobX

4.2.1 简介

mobx 是一个库 (library)，不是一个框架 (framework)。可以和 React 组合使用，MobX is a battle tested library that makes state management simple and scalable by transparently applying functional reactive programming (TFRP). React and MobX together are a powerful combination. redux 管理的是 (STORE -> VIEW -> ACTION) 的整个闭环，而 mobx 只关心 STORE -> VIEW 的部分。

4.3 gulp

Gulp 基于 Node.js 的前端构建工具，通过 Gulp 的插件可以实现前端代码的编译 (sass、less)、压缩、测试；图片的压缩；浏览器自动刷新。比起 Grunt 不仅配置简单而且更容易阅读和维护。gulpjs 是一个前端构建工具，与 gruntjs 相比，gulpjs 无需写一大堆繁杂的配置参数，API 也非常简单，学习起来很容易，而且 gulpjs 使用的是 nodejs 中 stream 来读取和操作数据，其速度更快。使用 browserify 进行打包，gulp 进行任务构建。由于使用了 ES2015 和 JSX 语法，因此使用 Babel 进行转换。

- 压缩 js 代码 (gulp-uglify)
- 压缩图片 (gulp-imagemin)

4.3.1 压缩图片

首先，定义一个压缩图片的任务：

```
1 gulp.task('images', function() {  
2   return gulp.src('src/images/**/*')  
3     .pipe(imagemin({ optimizationLevel: 3,  
4       progressive: true, interlaced: true })))  
5     .pipe(gulp.dest('dist/assets/img'))  
     .pipe(notify({ message: 'Images task  
complete' })));
```

⁵<https://github.com/typicode/json-server>

```
6 });
```

定义一个名称为 `images` 的任务，这个任务使用 `imagemin` 插件把所有在 `src/images/` 目录以及其子目录下的所有图片（文件）进行压缩，可以进一步优化，利用缓存保存已经压缩过的图片。在终端中运行如下命令执行此任务。

```
1 gulp images
```

4.3.2 清除文件

在再次生成文件前，最好先清除之前生成的文件：

```
1 gulp.task('clean', function(cb) {  
2     del(['dist/assets/css', 'dist/assets/js', 'dist/  
3         assets/img'], cb)  
});
```

在这里没有必要使用 Gulp 插件了，可以使用 NPM 提供的插件。用一个回调函数（cb）确保在退出前完成任务。

4.3.3 设置默认任务

```
1 gulp.task('default', ['clean'], function() {  
2     gulp.start('styles', 'scripts', 'images');  
3 });
```

`clean` 任务执行完成了才会去运行其他的任务，在 `gulp.start()` 里的任务执行的顺序是不确定的，所以将要在它们之前执行的任务写在数组里面。

4.4 webpack

使用 `install -save` 时，将默认在 `package.json` 中添加小尖尖[^]不是小波浪[~]为依赖版本的前缀。

4.5 Library

4.5.1 lodash

lodash 是一个具有一致接口、模块化、高性能等特性的 JavaScript 工具库。

去除重复数据

去除重复数据:

```
1 import lodashUniqBy from "lodash/uniqBy";
2 const uniqOrgList = lodashUniqBy(orgList, 'dfName');
```

分组

将数组中的元素按照新的方式返回。

```
1 /**
2  * 将部门按照10个1组组成一个新数组返回
3  */
4 return lodashChunk(dfbmList, 10).map((dfbms, rowNo)
5     => {
6     const items = this.renderItems(groups, dfbms)
7     ;
8     return (
9     });
```

排序

数组使用 lodash 排序如下代码所示:

```
1 import lodash from 'lodash';
2 /**
3  * 按照登录数量排序
4  */
5 const top10Datasource = lodash.sortBy(this.loginCount
6     (), function (item) {
7     item.count;
8 });
```

4.5.2 Axios

axios 是一个基于 Promise 用于浏览器和 nodejs 的 HTTP 客户端。ES6 原生提供了 Promise 对象。所谓 Promise，就是一个对象，用来传递异步操作的消息。它代表了某个未来才会知道结果的事件（通常是一个异步操作），并且这个事件提供统一的 API，可供进一步处理。

Promise 对象有以下两个特点。

(1) 对象的状态不受外界影响。Promise 对象代表一个异步操作，有三种状态：Pending（进行中）、Resolved（已完成，又称 Fulfilled）和 Rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是 Promise 这个名字的由来，它的英语意思就是「承诺」，表示其他手段无法改变。

(2) 一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise 对象的状态改变，只有两种可能：从 Pending 变为 Resolved 和从 Pending 变为 Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对 Promise 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

有了 Promise 对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise 对象提供统一的接口，使得控制异步操作更加容易。

Promise 也有一些缺点。首先，无法取消 Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部。第三，当处于 Pending 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

4.5.3 Axios 请求

一个简单的 Axios 请求示例如下：

```
1 import axios from 'axios';
2
3 export function request() {
4     return axios({
5         method: 'get', url: 'http://localhost
6         :8011/api/book/111'
7     }).then(
8         response => {
9             console.log(response)
10        }
11    ).catch(
12        error => {
13            console.error(error);
14        }
15    );
16 }
```

```
13     }  
14     );  
15 }
```

这里需要解决跨域问题，通过 Node 或 Nginx 等做 Proxy，或者在服务器端配置 CORS。

4.5.4 Axios 拦截响应

拦截响应可以对服务端抛出的错误做统一的 handle:

```
1  /**  
2   * axios响应拦截器  
3   */  
4  axios.interceptors.response.use(  
5      response =>{  
6          return response;  
7      },  
8      error => {  
9          if(error.response){  
10             switch (error.response.status){  
11                 case 401:  
12                     responseError401PreHandle(error.  
13                         response.data);  
14                     break;  
15                 case 403:  
16                     }  
17             }  
18 );
```

4.5.5 Promise

为了降低异步编程的复杂性，开发人员一直寻找简便的方法来处理异步操作。其中一种处理模式称为 promise，它代表了一种可能会长时间运行而且不一定必须完整的操作的结果。这种模式不会阻塞和等待长时间的操作完成，而是返回一个代表了承诺的 (promised) 结果的对象。promise 模式通常会实现一种称为 then 的方法，用来注册状态变化时对应的回调函数。promise 模式在任何时刻都处于以下三种状

态之一：未完成 (unfulfilled)、已完成 (resolved) 和拒绝 (rejected)。Promise.reject 返回了一个拒绝状态的 Promise 对象。对于这样的 Promise 对象，如果其后续 then 或 catch 中都没有声明 onRejected 回调，它将会抛出一个 “Uncaught (in promise) ...” 的错误。Promise 之间泾渭分明，内部 Promise 抛出的任何错误，外部 Promise 对象都无法感知并捕获。同时，由于 promise 是异步的，try catch 语句也无法捕获其错误。因此养成良好习惯，promise 记得写上 catch。

4.6 Cookie

The browser is expected to support 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4 KB each. 在 Chrome 中查看 Cookie 可以直接在 Console 中输入：

```
1 document.cookie
```

服务器端像客户端发送 Cookie 是通过 HTTP 响应报文实现的，在 Set-Cookie 中设置需要像客户端发送的 cookie，cookie 格式如下：

```
1 Set-Cookie: "name=value;domain=.domain.com;path=/;  
    expires=Sat, 11 Jun 2016 11:29:42 GMT;  
    HttpOnly;secure"
```

其中 name=value 是必选项，其它都是可选项。如果在 Cookie 中设置了 "HttpOnly" 属性，那么通过程序 (JS 脚本、Applet 等) 将无法读取到 Cookie 信息，这样能有效防止 XSS 攻击。当 secure 设置为 true 时，表示创建的 Cookie 会以安全的形式向服务器传输，也就是只能在 HTTPS 连接中被浏览器传递到服务器端进行会话验证，如果是 HTTP 连接则不会传递该信息，所以不会被盗取到 Cookie 的具体内容。Cookie 的主要构成如下：

- 1 name: 一个唯一确定的 cookie 名称。通常来讲 cookie 的名称是不区分大小写的。
- 2 value: 存储在 cookie 中的字符串值。最好为 cookie 的 name 和 value 进行 url 编码
- 3 domain: cookie 对于哪个域是有效的。所有向该域发送的请求中都会包含这个 cookie 信息。这个值可以包含子域 (如: yq.aliyun.com)，也可以不包含它 (如: .aliyun.com，则对于 aliyun.com 的所有子域都有效)

- 4 **path**: 表示这个cookie影响到的路径, 浏览器跟会根据这项配置, 像指定域中匹配的路径发送cookie。
- 5 **expires**: 失效时间, 表示cookie何时应该被删除的时间戳(也就是, 何时应该停止向服务器发送这个cookie)。如果不设置这个时间戳, 浏览器会在页面关闭时即将删除所有cookie; 不过也可以自己设置删除时间。这个值是GMT时间格式, 如果客户端和服务端时间不一致, 使用expires就会存在偏差。
- 6 **max-age**: 与expires作用相同, 用来告诉浏览器此cookie多久过期(单位是秒), 而不是一个固定的时间点。正常情况下, max-age的优先级高于expires。
- 7 **HttpOnly**: 告知浏览器不允许通过脚本document.cookie去更改这个值, 同样这个值在document.cookie中也不可见。但在http请求中仍然会携带这个cookie。注意这个值虽然在脚本中不可获取, 但仍在浏览器安装目录中以文件形式存在。这项设置通常在服务器端设置。
- 8 **secure**: 安全标志, 指定后, 只有在使用SSL链接时候才能发送到服务器, 如果是http链接则不会传递该信息。就算设置了secure 属性也并不代表他人不能看到你机器本地保存的 cookie 信息, 所以不要把重要信息放cookie就对了

Max Age

查看 Cookie 类的 `setMaxAge(int expiry)` 方法说明: 这个类用于给 Cookie 设置最大的年龄, 以秒为单位。expiry - an integer specifying the maximum age of the cookie in seconds; if negative, means the cookie is not stored; if zero, deletes the cookie. 如果这个方法的参数给的是正值, 表明 cookie 在超过指定的年龄时间后会消亡。负值意味着 cookie 不是永久性存储的, 在浏览器关闭的时候将会被删除。这个方法的参数如果是零值将会导致 cookie 被删除。Expires 指定了 cookie 的生存期, 默认情况下 cookie 是暂时存在的, 他们存储的值只在浏览器会话期间存在, 当用户推出浏览器后这些值也会丢失, 如果想让 cookie 存在一段时间, 就要为 expires 属性设置为未来的一个过期日期。现在已经被 max-age 属性所取代, max-age 用秒来设置 cookie 的生存期。

4.7 Javascript

快捷键	说明
nbsp	" "

4.7.1 prototype

prototype 是用于解决实例对象之间无法共享属性和方法的。

```
1 function Tiger(name){
2     this.name = name;
3     this.species = "猫科";
4 }
5 var tigerA = new Tiger("大虎");
6 var tigerB = new Tiger("二虎");
7 tigerA.species="犬科";
8 console.log(tigerB.species);//猫科
```

此处无法做到数据共享，也是极大的资源浪费。其实 species 共享是符合设计的。使用 prototype 来实现共享：

```
1 function Tiger(name) {
2     this.name = name;
3 }
4 Tiger.prototype = {
5     species: '犬科'
6 };
7 var tigerA = new Tiger("大虎");
8 var tigerB = new Tiger("二虎");
9 console.log(tigerA.species);//犬科
10 console.log(tigerB.species);//犬科
11 Tiger.prototype.species = '猫科';
12 console.log(tigerA.species);//猫科
13 console.log(tigerB.species);//猫科
```

注意这里是设置 Tiger 的 prototype 而不是 Tiger 实例的 prototype。一个未初始化的变量的值为 undefined，一个没有传入实参的形参变量的值为 undefined，如果一个函数什么都不返回，则该函数默认返回 undefined。所以这里在控制台最后会打印一行 undefined。普通对象设置 prototype 使用如下语句：

```
1 tigerA.__proto__.species = '犬科';
```

4.7.2 常用函数

正则匹配定长数字

采用如下函数验证定长数字：

```
1 isFixedNumber = (rule, value, callback) => {
2   if (/^\d{1,6}$/.test(value)) {
3     callback();
4   } else {
5     callback('机构ID只能是6位数字');
6   }
7 }
```

下划线命名转驼峰命名

如下函数将下划线命名的方式转成驼峰命名的方式：

```
1 function camelCase(input) {
2   return input.toLowerCase().replace(/_(.)/g,
3     function (match, group1) {
4       return group1.toUpperCase();
5     });
6 }
7 //XDR_SHXYM 转换结果 xdrShxym
const camelName = camelCase("XDR_SHXYM");
```

数学函数：

```
1 f = Math.round(x / 10000 * Math.pow(10, y)) / Math.
   pow(10, y);
```

Math.pow 计算 10 的 y 次幂，Math.round 计算与 round 函数最接近的整数。

```
1 #默认会执行一次
2 <button onClick={this.getBookInfo()}>查询</button>
3 #默认不会执行
```

```
4 <button onClick={() => this.getBookInfo()}>查询</button>
```

对象转数组：

```
1 const books = this.props.books;
2   let arr = [];
3   for(let i in books){
4     arr.push(books[i]);
5   }
```

判断值是不是空：

```
1 if (!value || (value + ' ').trim().length === 0) {
2   /**
3    * 值为空的字段在界面上不显示
4    * 如果value是整数，不能使用trim()，所以加字符串手动转换成String类型
5    */
6   return;
7 }
8
9 /**
10 * 如果需要判断字符串为空，直接if判断即可
11 */
12 if(value){
13   /**
14    * 此种写法等价于if(value===null||value===undefined||value==='')
15    */
16 }
```

4.7.3 日期处理

Moment.js 是一个 JavaScript 日期处理类库，如下代码片段使用 moment 格式化日期。

```
1 import moment from "moment";
```

```
2
3 tableDataSource = (logList, orgList) => {
4   const content = logList.content || [];
5   return content.map((item, idx) => {
6     const org = orgList.find(org => org.orgId ==
7       item.ORG_ID);
8     return {
9       ...item,
10      //格式化日期
11      QUERY_DATE: moment(item.QUERY_DATE).
12        format("YYYY-MM-DD hh:mm:ss"),
13    };
14  })
15};
```

获取本周的第一天，moment 中可以用如下代码实现：

```
1 moment.lang('zh-cn', {
2   week: {
3     dow: 1 // Monday is the first day of the week
4   }
5 });
6 // date now is the first day of the week, (i.e.,
7   Monday)
8 const date = moment().weekday(0);
```

4.8 常见问题

页面卡顿

一次页面卡顿是由于服务端某一个页面返回了过多的数据 6w 多。

index.js:83 TypeError: Cannot read property 'get' of undefined

出现此错误的原因是获取到的参数为空。

```
1 TypeError: Cannot read property 'get' of undefined
2 at Object.getToJS (Utils.js:171)
3 at Object.getToArray (Utils.js:166)
```



```

4 at CreditDataPeopleMain.render (CreditDataPeopleMain.
    js:299)
5 at eval (ReactCompositeComponent.js:796)
6 at measureLifeCyclePerf (ReactCompositeComponent.
    js:75)
7 at ReactCompositeComponentWrapper.
    _renderValidatedComponentWithoutOwnerOrContext
    (ReactCompositeComponent.js:795)
8 at ReactCompositeComponentWrapper.
    _renderValidatedComponent (
    ReactCompositeComponent.js:822)
9 at ReactCompositeComponentWrapper.performInitialMount
    (ReactCompositeComponent.js:362)
10 at ReactCompositeComponentWrapper.mountComponent (
    ReactCompositeComponent.js:258)
11 at Object.mountComponent (ReactReconciler.js:46)

```

獲取到的空數據如圖4.5所示。

```

    key: "render",
    value: function render() {
      var TYPES = this.props.TYPES; TYPES = {globalTYPE: {...}, "CREDIT_COUNT_INFO TYPE: LIST"
      var peopleCreditCountTYPE = TYPES.peopleCreditCountTYPE, "CREDIT_COUNT ORG LIST", CRE
      orgTYPE = TYPES.orgTYPE, orgTYPE = {OR: creditFormTYPE, "CREDIT FORM FIELD LIST:
      creditFormTYPE = TYPES.creditFormTYPE; CREDIT FORM (A) undefined CREDIT FORM TABLE

      var creditCountList = Utils2.default.getToArray(this.props.peopleCreditCount, peopleC
      var orgList = _Utils2.default.filterMainOrgList(this.props.org.get(orgTYPE.ORG LIST));
      var creditTableList = _Utils2.default.getToArray(this.props.creditForm, creditFormTYPE.
      var signUser = _Utils2.default.getToObject(this.props.sign, 'user');

      var rows = this.renderRow(signUser, creditTableList, creditCountList, orgList);
      var breadcrumbItems = [{

```

Figure 4.5: undefined 錯誤

由于采用的是 React，说明获取到的数据没有映射到对象里面。在 store.js 里面添加对应的缺少的实体即可。在错误处点击进去跟踪到相应的实体值为 undefined 的，添加到 store 文件中即可。

Each child in an array or iterator should have a unique "key" prop. Check the render method of 'CreditDataPeopleMain'

提示在數組和可枚舉中需要給定一個唯一的 key 屬性：

```

1 Warning: Each child in an array or iterator should
    have a unique "key" prop. Check the render
    method of 'CreditDataPeopleMain'. See https:

```

```

        //fb.me/react-warning-keys for more
        information.
2   in a (created by CreditDataPeopleMain)
3   in CreditDataPeopleMain (created by RouterContext)
4   in SharingExchangeContainer (created by Connect(
        SharingExchangeContainer))
5   in Connect(SharingExchangeContainer) (created by
        RouterContext)
6   in div (created by AppContainer)
7   in div (created by AppContainer)
8   in div (created by AppContainer)
9   in AppContainer (created by RouterContext)
10  in RouterContext (created by Router)
11  in Router
12  in Provider

```

key 屬性幫助 react 確定項目是否已經更新 (Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity)。

Cannot read property 'apply' of undefined

Module build failed: SyntaxError: Unexpected token ...

... 是 ES6 中的擴展运算符 (Spread Operator)，需要將之轉化為 ES5 語法，轉換使用 babel 來完成，presets 字段設定轉碼規則，檢查是否安裝包：

```

1  "devDependencies": {
2    "babel-core": "^6.25.0",
3    "babel-loader": "^7.1.0",
4    "babel-preset-es2015": "^6.24.1", //ES2015 轉碼規
    則
5    "babel-preset-react": "^6.24.1",
6    //ES7 不同階段語法提案的轉碼規則（共有 4 個階段）
7    //從 stage-0 到 stage-3
8    "babel-preset-stage-0": "6.22.0"
9  }

```

檢查 Webpack 中的配置：

```
1 module: {
2   loaders: [{
3     test: /\.jsx?$/,    // 匹配 jsx 语法规则
4     exclude: /node_modules/, // 排除的模块
5     loaders: 'babel-loader', // loader 名
6     query: {
7       // 将 react 预置为 es5
8       presets: ['es2015', 'react', 'stage-0']
9     }
10  }]
11 }
```

Store does not have a valid reducer

提示错误：Store does not have a valid reducer. Make sure the argument passed to combineReducers is an object whose values are reducers. store 在 index.js 文件中引入，在 store.js 中添加打印语句将 reducer 对象打印出来，看到实际是有 reducer 对象的，所以怀疑是 reducer 的定义有些问题。最后发现原来是 require 的使用有问题。

```
1 const state = {};
2 for (const name of nameList) {
3   state[name] = require(`./${name}/${name}Reducer`)
4   ;
5 }
```

这里是用 Require 动态加载，但是实际有点问题，暂时还找不到原因。暂时采用如下的写法：

```
1 import bookReducer from './book/bookReducer';
2 state[0] = bookReducer;
```




OS

5 Linux 87

- 5.1 Shell
- 5.2 常用设置

5. Linux

在使用 apt-get 时，有时可能会遇到依赖损坏的情况，那么可以执行如下命令，尝试修复依赖：

```
1 sudo apt-get -f install
```

f(fix) 参数表示尝试修正系统依赖损坏处。

5.1 Shell

5.1.1 自动部署 (Auto Deploy)

最初使用 Shell 脚本是将之用于自动化部署应用程序。一般部署程序步骤皆是重复的，编译、构建、打包、拷贝、备份网站、停止旧版网站启动新版 Beta 网站，Beta 网站出现问题时，Rollback 到 Stable 版本，可以使用 Shell 脚本将重复的操作自动化。将项目在本地打包，打包文件拷贝到目标服务器（一台或多台）上，自动启动应用程序。

```
1 #!/usr/bin/env bash
2
3 #
4 # 一键部署后端项目到远程 1 台或多台服务器
5 # 发布流程为：编译构建-打包-发布包-拷贝包到指定目录-解
  压包
```

```
6 # 注：本机需要安装 Ansible 并与服务器做免密登录
7 #
8 # 2017-05-25 增加动态读取版本号，根据版本自动发布程序
9 #
10
11 # 当使用未初始化的变量时，程序自动退出
12 set -u
13
14 # 当任何一行命令执行失败时，自动退出脚本
15 set -e
16
17 # send='dateã'+CURRENT_TIME='date '+
18 echo "$CURRENT_TIME"
19
20 PROGRAM_PATH_INFORMATION_CENTER="/opt/app/backend"
21 SERVER_IP="192.168.1.101"
22 LOGIN_USER="root"
23 LOCAL_PATH="./cc-web-boot/build/libs"
24
25 # Read program version number
26 source version.properties
27 echo "$VERSION"
28
29 PROGRAM_NAME="credit-system-web-boot-$VERSION.jar"
30 echo "$PROGRAM_NAME"
31
32 echo "开始构建..."
33 # Build project
34 ./gradlew -p cc-web-boot -x test build
35 ./gradlew -p cc-etl-sechedule -x test build
36
37 echo "开始拷贝..."
38 # publish project
39 scp $LOCAL_PATH/$PROGRAM_NAME $LOGIN_USER@$SERVER_IP
40      :~/app-soft/
41 scp start.sh $LOGIN_USER@$SERVER_IP:
```



```

        $PROGRAM_PATH_INFORMATION_CENTER
41 scp version.properties $LOGIN_USER@$SERVER_IP:
        $PROGRAM_PATH_INFORMATION_CENTER
42
43 ansible webservers -m command -a "date"
44
45 echo "停止站点..."
46 ansible webservers -m command -a "chdir=
        $PROGRAM_PATH_INFORMATION_CENTER bash ./stop.
        sh"
47
48 echo "等待站点停止..."
49 sleep 10
50
51 echo "备份站点..."
52 # 备份当前站点
53 ansible webservers -m command -a "mv
        $PROGRAM_PATH_INFORMATION_CENTER/
        $PROGRAM_NAME
        $PROGRAM_PATH_INFORMATION_CENTER/
        $PROGRAM_NAME-$CURRENT_TIME"
54
55 echo "拷贝新文件..."
56 ansible webservers -a "mv ~/app-soft/$PROGRAM_NAME
        $PROGRAM_PATH_INFORMATION_CENTER"
57
58 echo "启动站点..."
59 ansible webservers -m shell -a "bash ./start.sh chdir
        =$PROGRAM_PATH_INFORMATION_CENTER"
60
61 echo "部署完成"
```

start.sh 和 stop.sh 为启动脚本，通过 Ansible 远程调用执行。服务器程序路径和启动脚本路径固定。另外编写了一个用于 API 授权测试、API 调用的脚本（接口根据密钥和随机字符动态生成 Token，没有找到现成的工具），用于动态生成 API Token，提高 API 测试的方便性和 API 交互的安全性，具体可以参见 1.2.3 节。set -u

Table 5.1: 终端快捷键

快捷键	说明
Alt + F	向前移动一个单词
Alt + B	向后移动一个单词
Ctrl + ;	打开剪贴板历史，默认显示 5 个历史记录，选择相应的数字粘贴 (Ubuntu 下有效)
Ctrl + R	查找历史命令 (输入过滤条件之后，再次按 Ctrl + R 可以往回搜索上一条符合过滤条件的命令)
Ctrl + K	从光标当前位置向后删除所有字符
Ctrl + U	从光标当前位置向前删除所有字符

用来改变脚本的行为。脚本在头部加上它，遇到不存在的变量等情况就会报错，并停止执行。

终端快捷键 (Terminal Shortcut)

Shell 终端中常用的快捷键如表5.1：

source

在使用 source 命令时，提示找不到文件。如果 filename 不包含斜杠 (/)，那么从 PATH 环境变量指定的那些路径搜索 filename，这个文件不必是可执行的。(If filename does not contain a slash, file names in PATH are used to find the directory containing filename. The file searched for in PATH need not be executable.) 如果在 PATH 中找不到指定文件，当 bash 不是 posix 模式时，将在当前目录下搜索该文件。(When bash is not in posix mode, the current directory is searched if no file is found in PATH.) 如果 shopt 里的 sourcepath 关闭，则不在 PATH 中搜索指定文件。(If the sourcepath option to the shopt builtin command is turned off, the PATH is not searched.) 直接写文件的绝对路径解决。source filename 其实只是简单地读取脚本里面的语句依次在当前 shell 里面执行，没有建立新的子 shell。那么脚本里面所有新建、改变变量的语句都会保存在当前 shell 里面。所以为什么修改了.bashrc 文件之后都会使用 source 命令加文件名来让设置生效。但是 source 命令生效也有局限，就是智能在当前的 shell 中生效，比如在/etc/profile 中修改了全局环境变量，使用 source 命令生效只是在当前的 shell，如果需要全局生效，还是需要重新登陆当前用户。

修改登录路径

每次登录后都要切换到项目路径，直接登录时切换到项目路径，在当前用户的.bashrc 中添加如下设置：

```
1 if [ -d /home/app/ ];then
2     cd /home/app/
3 fi
```

输入长命令

有时在 shell 中，会输入比较长的命令，直接在 shell 中输入可能不太方便。可以直接使用快捷键 Ctrl+X,E，即可调出默认编辑器。因为自己习惯使用 vim，所以使用如下命令设置 vim 为默认的编辑器：

```
1 export EDITOR=vim
```

在 vim 中，编辑好了命令之后，直接按下 ZZ 命令，即可退出 vim，shell 即会执行编辑完成后的命令。

创建路径

创建目录及目录树可以批量进行，一个命令，即可创建整个目录树。

```
1 #新建后端开发目录
2 mkdir -p /home/dolphin/hldata/backend/credit-system
3 #新建开发目录树
4 mkdir -p /home/dolphin/hldata/{backend/credit-system,
    frontend/credit-system-frontend,doc/{html,
    info,pdf},demo/}
```

与控制操作符组合使用

|| 控制操作符分隔两个命令，并且仅当第一个命令返回非零退出状态时才运行第二个命令。换句话说，如果第一个命令成功，则第二个命令不会运行。如果第一个命令失败，则第二个命令才会运行。

```
1 #如果目录不存在，则创建相应的目录
2 cd /home/hldev/app-soft || mkdir -p /home/hldev/app-
    soft
```

source 文件路径

在使用 `source` 命令时，提示找不到文件。如果 `filename` 不包含斜杠 (/)，那么从 `PATH` 环境变量指定的那些路径搜索 `filename`，这个文件不必是可执行的。(If `filename` does not contain a slash, file names in `PATH` are used to find the directory containing `filename`. The file searched for in `PATH` need not be executable.) 如果在 `PATH` 中找不到指定文件，当 `bash` 不是 `posix` 模式时，将在当前目录下搜索该文件。(When `bash` is not in `posix` mode, the current directory is searched if no file is found in `PATH`.) 如果 `shopt` 里的 `sourcepath` 关闭，则不在 `PATH` 中搜索指定文件。(If the `sourcepath` option to the `shopt` builtin command is turned off, the `PATH` is not searched.) 直接写文件的绝对路径解决。

读取配置文件

在编写自动化部署脚本时，需要通过 `shell` 脚本读取配置文件 `version.properties` 中的版本号，直接使用 `source` 命令即可：

```
1 # version.properties 文件中直接写 VERSION_CODE=1.0.2
   即可
2 source /Users/dolphin/source/credit-system/gradle/
   version.properties
3 # 输出 1.0.2
4 echo $VERSION_CODE
```

点空格点斜杠执行脚本 (`./`)，是相当于 `source ./` 执行脚本，`source` 是执行脚本当中的命令，也就是说在当前进程中执行命令，所以其中的环境变量的设置会对当前 `Shell` 起作用。点斜杠执行脚本是启动了另一个 `Shell` 去执行脚本（另一个进程），所以点斜杠执行脚本时，设置的环境变量会随着进程的退出而结束，其中的环境变量设置对当前 `Shell` 不起作用。在 `Gradle` 中也可以直接读取 `version.properties` 中的内容，这样编译打包发布就可以通过脚本自动化完成。

文件是否存在

`Shell` 下判断文件是否存在：

```
1 myFile="/opt/app/backend/app.pid"
2
3 if [ -f "$myFile" ]; then
4     kill `cat /opt/app/backend/app.pid`
5 fi
```

Terminal 自动提示大小写不敏感

有时文件或者文件夹是以大写字母开头，在使用 Tab 按键自动提示时就必须首先输入大写字母匹配才有效果，总体来说不是非常方便。所以可以设置 shell 智能提升时大小写不敏感，就不需要再重复做一次大小写转换操作了：

```
1 # Mac OS X 和 Linux 下皆可
2 echo "set completion-ignore-case on">> ~/.inputrc
3 echo "set show-all-if-ambiguous on">> ~/.inputrc
```

设置完毕后重新打开终端即可，必须要重新打开终端才会生效。以上设置针对当前用户，如果需要在全局范围内生效，可以修改 inputrc 文件：

```
1 # Mac OS X 和 Linux 下皆可
2 echo "set completion-ignore-case on">> /etc/inputrc
3 #重启系统使之生效
4 reboot
```

统计代码行数

统计项目代码行数使用如下命令：

```
1 # Java 行数: 35400
2 find . -name "*.java"|xargs cat|grep -v ^$|wc -l
3 # 42178, 会显示每个 Java 文件的行数
4 find . -name '*.java' |xargs wc -l
5
6 # Scala 行数: 1500
7 find . -name "*.scala"|xargs cat|grep -v ^$|wc -l
8 # XML 行数: 7336
9 find . -name "*.xml"|xargs cat|grep -v ^$|wc -l
10 # js 行数: 151312
11 find . -name "*.jsr"|xargs cat|grep -v ^$|wc -l
12
13 # 统计代码行数
14 find sourcecode "(" -name "*.js" -or -name "*.java" -
    or -name "*.scala" -or -name "*.xml" ")"|
    xargs cat|grep -v ^$|wc -l
```

符号 `^` 表示行首，`$` 表示行尾，使用 `grep -v` 可以实现 NOT 操作。`-v` 选项用来实现反选匹配的 (`invert match`)。`xargs` 是一个强有力的命令，它能够捕获一个命令的输出，然后传递给另外一个命令。

查看端口情况

查看某个端口是否被占用：

```
lsof -i:18080
```

`lsof` (`list open files`) 是一个列出当前系统打开文件的工具。`FD`(`File Descriptor`): 文件描述符，应用程序通过文件描述符识别该文件。在 `Linux` 系统中一切皆可以看成是文件，文件又可分为：普通文件、目录文件、链接文件和设备文件。文件描述符 (`file descriptor`) 是内核为了高效管理已被打开的文件所创建的索引，其是一个非负整数 (通常是小整数)，用于指代被打开的文件，所有执行 `I/O` 操作的系统调用都通过文件描述符。程序刚刚启动的时候，`0` 是标准输入，`1` 是标准输出，`2` 是标准错误。如果此时去打开一个新的文件，它的文件描述符会是 `3`。

根据具体操作系统的不同，将文件和目录称为 `REG` 和 `DIR` (在 `Solaris` 中，称为 `VREG` 和 `VDIR`)。其他可能的取值为 `CHR` 和 `BLK`，分别表示字符和块设备；或者 `UNIX`、`FIFO` 和 `IPv4`，分别表示 `UNIX` 域套接字、先进先出 (`FIFO`) 队列和网际协议 (`IP`) 套接字。进而查看被哪个程序占用：

```
ps pid
```

5.1.2 调试 Bash 脚本 (Bash Debugging)

有时，我们想看看 `Bash` 脚本到底执行了哪些语句，可以在执行前添加 `-x` 选项来运行：

```
bash -x ./auth.sh
```

命令行前面的 `+` 号表示嵌套。

5.1.3 命令 (Command)

`which` 命令用于查找并显示给定命令的绝对路径，环境变量 `PATH` 中保存了查找命令时需要遍历的目录。`which` 指令会在环境变量 `$PATH` 设置的目录里查找符合条件的文件。也就是说，使用 `which` 命令，就可以看到某个系统命令是否存在，以及执行的到底是哪一个位置的命令。查看软件运行目录：

Table 5.2: 文件描述符

文件描述符	说明
cwd	表示 current work dirctory, 即: 应用程序的当前工作目录, 这是该应用程序启动的目录, 除非它本身对这个目录进行更改
txt	该类型的文件是程序代码, 如应用程序二进制文件本身或共享库, 如上列表中显示的/sbin/init 程序
lnn	library references (AIX)
ltx	shared library text (code and data)
er	FD information error (see NAME column)
jld	jail directory (FreeBSD)
mxx	hex memory-mapped type number xx
m86	DOS Merge mapped file
mem	memory-mapped file
mmap	memory-mapped device
d	parent directory
rtd	root directory
tr	kernel trace file(OpenBSD)
v86	VP/ix mapped file
0	表示标准输出
1	表示标准输入
2	表示标准错误

```
1 which scala
```

ls

有时一个文件夹里面的文件很多，为了避免从太多文件里面去找目标文件，而只想查看一个文件的属性：

```
1 #列出指定文件的信息
2 ls -lh authorized_keys
3 #只显示当前文件夹下的目录
4 ls -al | grep "^d"
```

find

在当前目录下查找大文件：

```
1 find . -type f -size +1000k
2 #查找大于 100MB 的文件
3 find . -type f -size +100M
```

size 默认单位是 b，而它代表的是 512 字节，所以 2 表示 1K，1M 则是 2048，如果不想自己转换，可以使用其他单位，如 c、K、M 等。如何查找指定大小的文件，并列出现实文件的大小。find 查找文件时需要添加通配符：

```
1 find / -name "*工作*"
```

kill

tar

创建压缩文档：

```
1 #打包
2 tar -czvf mybatisDemo.tar.gz mybatisDemo
```

解压 tar.xz 压缩文件，先 xz -d xxx.tar.xz 将 xxx.tar.xz 解压成 xxx.tar 然后，再用 tar xvf xxx.tar 来解包。

```
1 xz -d xxx.tar.xz
2 tar xvf xxx.tar
3 # 解压 tgz 文件
```



```

4 # 其实 tgz 文件和 tar.gz 文件是一样的
5 # 可以使用相同的命令解压
6 tar -zxcf example.tgz

```

grep

有时需要从返回的列表中，查看某一个字段的返回结果，如果列表太长或者字段太多，单凭肉眼分辨非常吃力，此时可以通过 `grep` 来查看数组返回数据：

```

1 curl -H "APPID:" -H "TIMESTAMP:2016-12-19 16 (tel
      :2016121916):58:02" -H "ECHOSTR:" -H "TOKEN:"
      http://10.10.1.12:28080/api/xzxk?xdr= | jq
      ').' |grep jdrq

```

效果如图5.1所示。

```

hldev@hldev-ThinkPad-S2:~/jiangxiaoqiang/d
-H "TOKEN:31c94bbf40b70bc5575e0e68f1c18f14
% Total    % Received % Xferd  Average S
Dload Up
100 8272    0 8272    0    0 477k
"jdrq": "2017-08-15T00:00:00",
"jdrq": "2017-08-15T00:00:00",
"jdrq": "2017-08-15T00:00:00",
"jdrq": "2017-08-15T00:00:00",
"jdrq": "2017-08-15T00:00:00",
"jdrq": "2017-08-15T00:00:00",
"jdrq": "2017-08-14T18:22:00",
"jdrq": "2017-08-15T00:00:00",
"jdrq": "2017-08-15T00:00:00",
"jdrq": "2017-08-15T00:00:00",
hldev@hldev-ThinkPad-S2:~/jiangxiaoqiang/d

```

Figure 5.1: Grep 过滤 Json 结果

这样就可以轻松的看出来某个关键字段的返回结果，不需要再一个个比对。

sed(Stream Editor)

`sed` 全名叫 stream editor，流编辑器，用程序的方式来编辑文本，相当的 hacker 啊。`sed` 基本上就是玩正则模式匹配，所以，玩 `sed` 的人，正则表达式一般都比较强。過濾某一個時間段的日志：

```

1 sed -n '/^2017-08-22 18:50/','/^2017-08-22 18:59/p'
      cc-web.2017-08-22.0.log >> filter.log

```

-n取消默认的输出,使用安静 (silent) 模式。在一般 sed 的用法中,所有来自 STDIN 的资料一般都会被列出到屏幕上。但如果加上 -n 参数后,则只有经过 sed 特殊处理的那一行(或者动作)才会被列出来。^匹配一行的开始。-n 选项和 p 命令一起使用表示只打印那些发生替换的行。

ag

ag¹可以递归搜索文件内容,输入如下命令安装:

```
1 # Ubuntu 安装 ag
2 apt-get install silversearcher-ag
3 dnf install the_silver_searcher
```

traceroute

如果目标系统在 3 秒的超时间隔内没有响应,所有的查询都会发生超时,结果会采用星号 (*) 显示,由此可以断定是目标系统的问题。

```
1 # Linux
2 traceroute 10.55.10.9
3 # Windows
4 tracert 10.55.10.9
```

在 traceroute 输出中,收到了 ICMP²错误信息(不包括时间超限和不能到达的端口),! H 表示不能到达的主机。在同一局域网里面的 2 台主机, ping 会偶尔出现丢包的情况。使用 Wireshark 监听 ping 流量如图5.2所示。其中 10.55.10.8 表示 ping 主机, 10.55.10.9 表示目标主机。在图中提示 no response found 的,会在 ping 主机上提示“请求超时”。这里的 Wireshark 监听条件是 host==10.55.10.8。

5.1.4 tmux

tmux 是一个优秀的终端复用软件,类似 GNU Screen,但来自于 OpenBSD,采用 BSD 授权。使用 tmux 的好处之一就是可以保存会话,避免网络不稳定带来的影响。

键	作用
Ctrl + b d	返回主 shell , detach, tmux 依旧在后台运行,里面的命令也保持运行状态
C-b ?	显示快捷键帮助

¹https://github.com/ggreer/the_silver_searcher

²https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

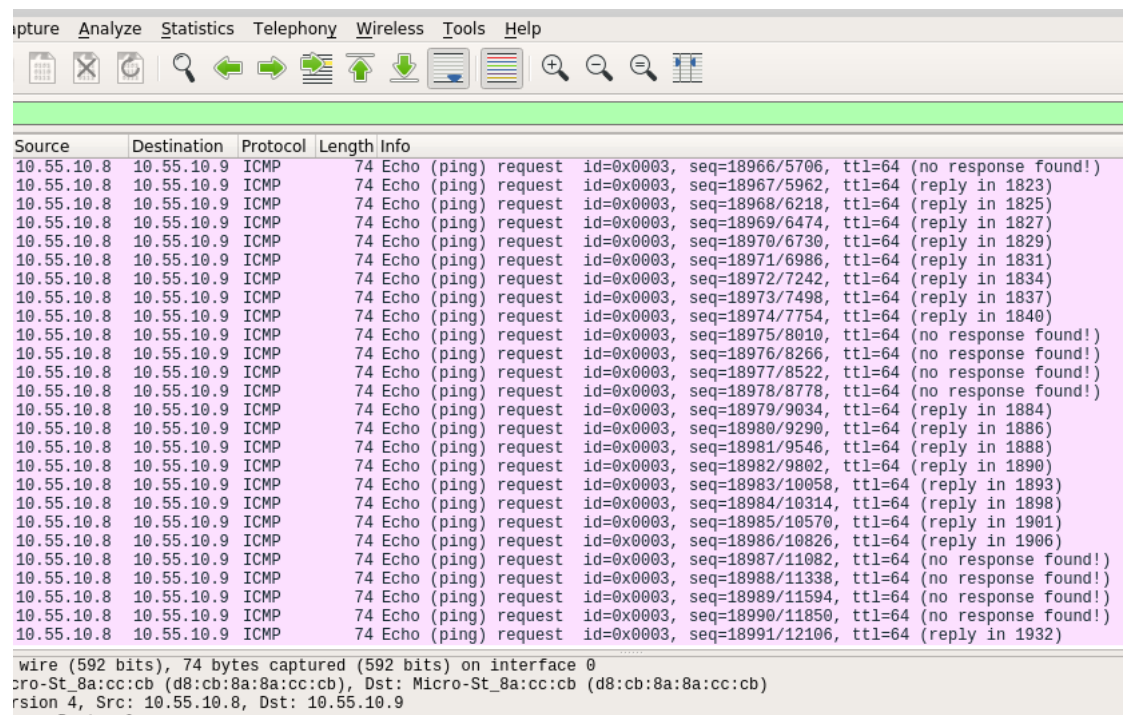


Figure 5.2: Wireshark 监听 Ping

使用终端,当然需要漂亮的字体.TeX Gyre Cursor 可以作为终端的字体.The TeX Gyre Cursor family of monospaced serif fonts is based on the URW Nimbus Mono L family . and are available in PostScript, TeX and OpenType formats. 效果如图5.3所示.

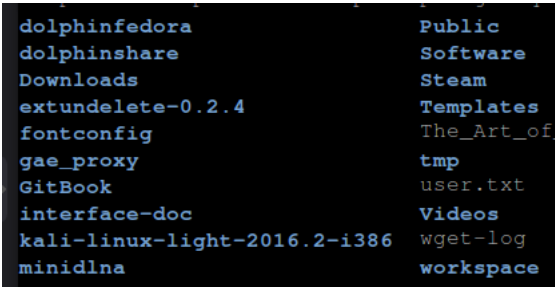


Figure 5.3: 终端使用 Tex Gyre Cursor 字体

也可以使用 TlwgMono Oblique 和 FreeMono 字体.

5.1.5 环境变量 (Environment Variable)

Linux 分 shell 变量 (set), 用户变量 (env), shell 变量包含用户变量, export 是一种命令工具, 是显示那些通过 export 命令把 shell 变量中包含的用户变量导入给用户变量的那些变量. 删除 Linux 环境变量:

```
# 查看环境变量
```

Table 5.3: Linux profile 配置文件

序号	名称
/etc/profile	此文件为系统的每个用户设置环境信息, 当用户第一次登录时, 该文件被执行. 并从/etc/profile.d 目录的配置文件中搜集 shell 的设置
/etc/environment	在登录时操作系统使用的第二个文件, 系统在读取你自己的 profile 前, 设置环境文件的环境变量
/etc/bashrc	为每一个运行 bash shell 的用户执行此文件. 当 bash shell 被打开时, 该文件被读取
/.profile	每个用户都可使用该文件输入专用于自己使用的 shell 信息, 当用户登录时, 该文件仅仅执行一次! 默认情况下, 它设置一些环境变量, 执行用户的.bashrc 文件
/.bashrc	该文件包含专用于你的 bash shell 的 bash 信息, 当登录时以及每次打开新的 shell 时, 该文件被读取

```

2 echo $http_proxy
3 env
4 # 删除环境变量
5 unset http_proxy

```

set 命令显示 (设置)shell 变量包括的私有变量以及用户变量, 不同类的 shell 有不同的私有变量 bash,ksh,csh 每中 shell 私有变量都不一样。env 命令显示 (设置) 用户变量变量。export 命令显示 (设置) 当前导出成用户变量的 shell 变量。Linux profile 文件如表5.3所示。

有时在 Bash 记录历史命令时, 需要忽略掉重复的命令等。可以做如下设置:

```

1 # 忽略 [连续] 重复命令
2 HISTCONTROL=ignoredups
3 # 清除重复命令
4 # HISTCONTROL=erasedups

```

```
5 # 忽略特定命令
6 HISTIGNORE="[ ]*:ls:ll:cd:vi:pwd:sync:exit:history*"
7 # 命令历史文件大小 10M
8 HISTFILESIZE=1000000000
9 # 保存历史命令条数 10W
10 HISTSIZE=1000000
```

以上配置可以通过 `set | grep HIST` 查看可选项。当打开多个终端，关闭其中一个终端时，会覆盖其他终端的命令历史，这里我们采用追加的方式避免命令历史文件 `.bash_history` 文件被覆盖。再次打开 `./bashrc` 文件添加下面这一句。

```
1 shopt -s histappend
```

更多 `shopt` 可选项可以通过 `echo $SHELLOPTS` 命令查看。

screen

GNU Screen 是一款由 GNU 计划开发的用于命令行终端切换的自由软件。用户可以通过该软件同时连接多个本地或远程的命令行会话，并在其间自由切换。在使用 `nohup` 时，有一个缺点就是交互性很低，例如后台启动 OpenVPN 时，需要输入预先设置的认证密码。但是使用 `nohup` 命令启动时，无法显示密码输入入口。此时就需要使用 `screen` 命令了。输入如下的命令：

```
1 screen sudo openvpn client.conf
```

接着在跳出的窗口中，输入密码。按下 `Ctrl + A` 后再按下 `d` 键，暂时断开 `screen` 会话即可。只要 `Screen` 本身没有终止，在其内部运行的会话都可以恢复。这一点对于远程登录的用户特别有用——即使网络连接中断，用户也不会失去对已经打开的命令行会话的控制。只要再次登录到主机上执行 `screen -r` 就可以恢复会话的运行。同样在暂时离开的时候，也可以执行分离命令 `detach`，在保证里面的程序正常运行的情况下让 `Screen` 挂起（切换到后台）。这一点和图形界面下的 VNC 很相似。查看当前系统中有哪些 `screen` 会话：

```
1 screen -ls
```

该命令会列出当前系统中所有的 `screen` 会话，重新打开会话：

```
1 screen -r 12865
```

正常情况下，当你退出一个窗口中最后一个程序（通常是 `bash`）后，这个窗口就关闭了。另一个关闭窗口的方法是使用 `C-a k`，这个快捷键杀死当前的窗口，同时也将杀死这个窗口中正在运行的进程。如果一个 `Screen` 会话中最后一个窗口被关闭了，那么整个 `Screen` 会话也就退出了，`screen` 进程会被终止。除了依次退出/杀死当前 `Screen` 会话中所有窗口这种方法之外，还可以使用快捷键 `C-a :`，然后输入 `quit` 命令退出 `Screen` 会话。需要注意的是，这样退出会杀死所有窗口并退出其中运行的所有程序。其实 `C-a :` 这个快捷键允许用户直接输入的命令有很多，包括分屏可以输入 `split` 等，这也是实现 `Screen` 功能的一个途径

查看 Linux 信息

```
1 # Linux 查看版本当前操作系统内核信息
2 uname -a
```

5.2 常用设置

5.2.1 XFCE 桌面

占用资源较 `GNOME`，`KDE` 较少。适合老机器，轻量级桌面。与 `windows` 界面环境类似。许多不习惯 `GNOME 3`，`Unity` 新桌面的同学，很多选择了 `XFCE 4.8`，包括 `Linus` 大神同学。

5.2.2 Fedora 访问 Windows 共享文件夹

在局域网里，有时需要使用 `Fedora 24` 访问 `Windows 7` 共享文件夹。在 `Fedora` 中输入命令：

```
1 #密码 aaa321
2 sudo mount -t cifs -o username="aaa",uid="dolphin",
    gid="dolphin" //10.55.10.2/data /home/dolphin
    /software
```

一般提示权限错误是由于用户名密码输入错误。`aaa` 为 `Windows` 机器的用户名，`uid` 为当前 `Fedora` 账户的名称，`gid` 为当前 `Fedora` 用户组的名称。`10.55.10.2` 为 `Windows` 账户的 IP，`data` 为共享文件夹的名称，与盘符无关。`/home/dolphin/software` 本地 `Fedora` 机器的目录。`cifs` 为通用网络文件系统，`The Common Internet File System (CIFS)` is the standard way that computer users share files across corporate intranets and the Internet. An enhanced version of the Microsoft open, cross-platform Server Message Block (SMB) protocol, `CIFS` is a native file-sharing protocol in `Windows 2000`.

5.2.3 Ubuntu 访问 Windows 共享文件夹

在 Windows 7 中设置了文件夹共享后，在 Ubuntu 文件管理器中，点击左侧的“Connect to Server”选项卡，填写 Windows 的 IP，示例如下：

```
1 smb://10.55.10.2
```

点击确定后，在弹出的对话框中输入相应的用户名和密码即可。

5.2.4 软件包管理 (Package Management)

根据软件包的名称模糊匹配，查看系统已经安装了哪些软件包：

```
1 dpkg -l|grep pcre
2 yum list installed|grep openssl
3 # 查看 openssl 版本
4 openssl version -a
5 # yum 查看已经安装的软件包
6 yum list installed
7 # 下载指定版本软件
8 sudo apt-get install scala=2.12.3
9 # Ubuntu 显示包的详细信息
10 sudo apt show wget
11 # 删除软件及其配置文件
12 sudo apt-get --purge -y remove <package>
```

在升级软件包的时候，可以注意软件包管理器下方的提示，在网络速度不是很理想的情况下，可以优先升级已经下载完毕后的软件包。下载软件包一个比较头疼的问题就是下载速度有时非常慢，此时如果是国外的镜像源（默认是 US 的镜像源），可以替换为国内的镜像源。可以直接修改/etc/apt/source.list 文件和通过 Software&Updater 图形界面调整。其中/etc/apt/source.list.d 中存放了第三方软件的源。升级时，默认会下载服务器上最新版本的软件包，但是有时候对下载的包的版本有要求，必须是指定版本，那么可以提前查看服务器的版本，以决定是否需要下载还是手动安装：

```
1 # 查看服务器 Scala 版本
2 sudo apt-cache madison scala
3 sudo apt-cache policy scala
4 # 查看版本
```

```
5 apt-show-versions -a scala
```

madison 是一个 apt-cache 子命令。

5.2.5 Ubuntu 打印

在使用 ProjectLibre 撰写项目计划时，打印出来的 pdf 中文竟然未显示。此时需要使用虚拟打印机来打印文档了，第一步安装 Linux 下的虚拟打印机 cups-pdf：

```
1 # 安装 cups-pdf
2 sudo apt-get install cups-pdf -y --fix-missing
```

cups-pdf 是 Ubuntu 下的 PDF 虚拟打印机。第二步安装 ghostscript 将生成的 ps 文件转换为 pdf：

```
1 # 安装 ghostscript 包
2 sudo apt-get install ghostscript
3 # 将 ps 转换为 pdf
4 ps2pdf out.ps out.pdf
```

ghostscript package contains a group of command line tools, in which ps2pdf converts .ps file to .pdf file. The package also contains other command tools, they're dumphint, dvipdf, eps2eps, font2c, ghostscript, gs, gsbj, gsdj, gsdj500, gslj, gslp, gsnd, pdf2dsc, pdf2ps, pdfopt, pf2afm, pfbtopfa, pphs, printafm, ps2ascii, ps2epsi, ps2pdf, ps2pdf12, ps2pdf13, ps2pdf14, ps2pdfwr, ps2ps, ps2ps2, ps2txt, update-gsfontmap, wftopfa.

5.2.6 输入法 (Input Method)

有时在 Fedora 24 中，突然中文输入法切换后无法输入中文了。可以到系统区域和语言 (Region&Language) 设置界面，重新删除后再添加即可修复，如图5.4所示。

在使用 Ubuntu 16.04 的搜狗拼音时，有时也会出现各种问题。可以尝试的解决方案包括，登录系统再次重新登入。删除输入法配置后，重新添加输入法，或者使用 Google 拼音输入法，输入如下命令进行安装：

```
1 sudo apt install fcitx-googlepinyin
```

安装好后需要重新登入系统，添加输入法配置时才可以选择 Google 输入法。

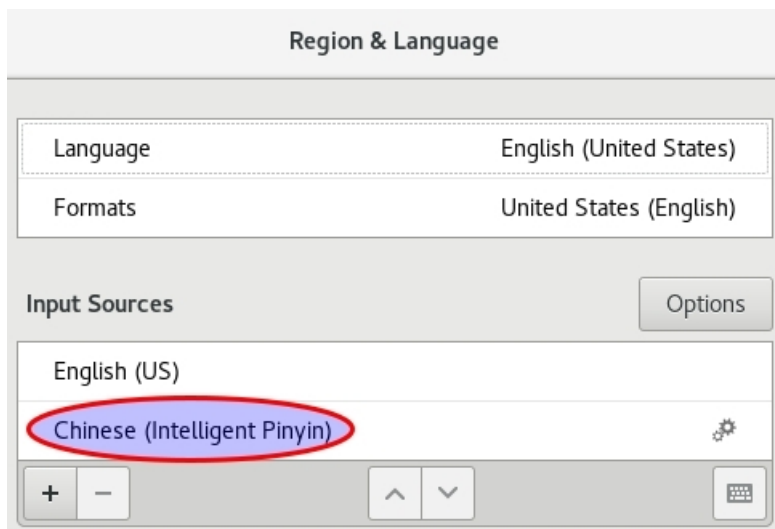


Figure 5.4: Fedora 区域和语言设置

Tool

6	Intellij Idea	109
7	Gradle	115
7.1	基础	
7.2	Gradle 对象	
8	OWASP ZAP	123
8.1	基础	
9	OpenVPN	125
10	Nmap(GNU General Public License)	133
11	Raspberry Pi	139
11.1	基础	
12	Widgets	141
12.1	Vim Editor	
12.2	Little Tool	
12.3	Nginx	
12.4	Ansible	
12.5	ssh(Secure Shell)	
12.6	VisualVM	
12.7	Graphviz	
12.8	MyBatis Generator	
13	Git	169
13.1	基础	
14	Google Chrome	175
14.1	DevTools	
15	LaTeX	181
15.1	安装	

6. IntelliJ Idea

第一次配置 JDK 在 Configure->Project Defaults->Project Structure 设置里。Mac OS X 的路径一般如下：

```
1 # 查看 Java 路径
2 echo $JAVA_HOME
3 /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk
4 # Mac OS X 下 Gradle 的路径
5 /usr/local/Cellar/gradle/3.2.1/libexec
```

6.0.1 IntelliJ Idea 远程调试

输入如下命令，启动远程调试：

```
1 nohup /home/app/local/jdk1.8.0_111/bin/java -
    Xmx8192M -Xms4096M -jar -Xdebug -Xrunjdwp:
    transport=dt_socket,suspend=n,server=y,
    address=5005 /home/app/backend/credit-system-
    web-boot-1.1.9.jar --spring.config.location=
    application.properties>/dev/null &
```

在远程调试 Web 应用时，调试时 Block 住了整个网站的请求，其实可以在远程调试打断点时选择阻断线程而不是整个 JVM，如图所示：

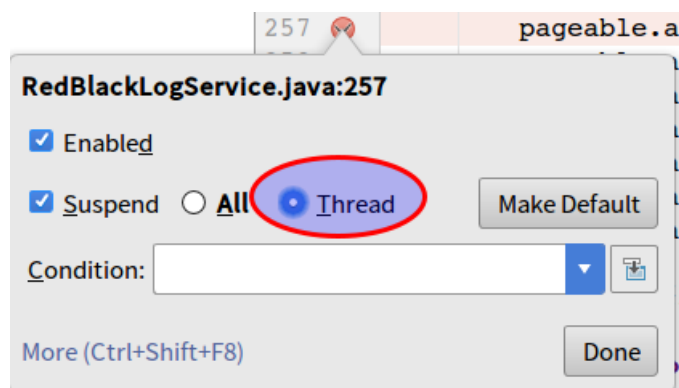


Figure 6.1: IntelliJ Idea 设置断点阻断当前 Thread

这样就不会阻断其他用户的正常请求了。

6.0.2 IntelliJ Idea 调试慢

在使用 IntelliJ Idea 时调试 Spring Boot 项目时，启动时突然变慢了，原来启动 10 几秒，后来启动莫名其妙需要 1 分钟左右。经过排查是由于打了过多的断点的缘故，删除断点后，启动立即变快了。所以在项目中不能标记太多断点，调试后立即将断点删除。

6.0.3 无法查看 Scala 变量

在 IntelliJ Idea 调试 Scala 时，无法查看 Scala 的 Vector 等变量，提示 Unable to evaluate the expression java.lang.UnsupportedOperationException。解决方案是，重新安装 IntelliJ Idea:

```
1 #找到 Idea 的安装目录
2 sudo find / -name "idea.sh"
```

找到安装目录后直接删除目录即可。重装后调试即可看到相应的变量，如图6.2所示。

6.0.4 常见设置

有时在 IntelliJ Idea 编译时找不到类，输出 CLASSPATH 时，发现并没有相应的 Gradle 库的路径在 CLASSPATH 中，一般使用 Gradle 编译时，CLASSPATH 都会默认包含有 .gradle 路径，如：/home/hldev/.gradle/caches/modules-2/files-2.1。如果没有包含引用 Library 路径，可以在 Project Structure->Project Settings->Libraries。

有时在使用列编辑 SQL 语句时，会自动换行，而且会自动添加一些符号，导致编辑出来的 SQL 语句语法错误，在 Settings->Editor->Code Style 下的 Default Options



Figure 6.2: 调试查看 Scala 变量

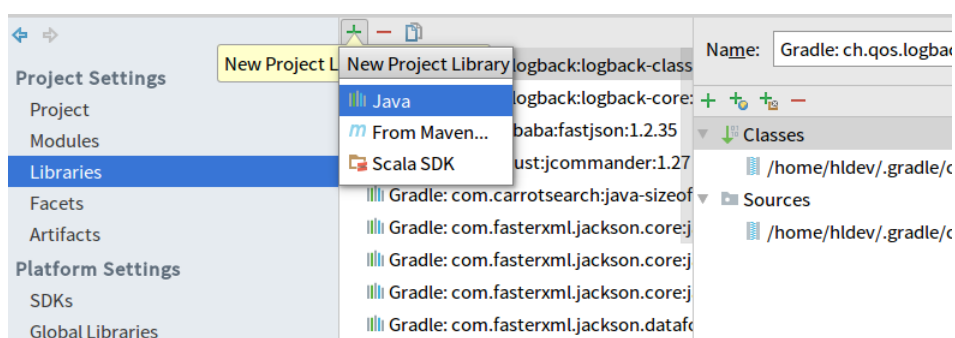


Figure 6.3: IntelliJ Idea 添加项目 Library 路径

Table 6.1: IntelliJ Idea 快捷键

快捷键	作用
Ctrl + Left/Right	到行首/尾
Command + Shift + A(Mac)	Find Action
Ctrl + Left/Right(Ubuntu)	左移/右移一个单词
Home/End(Ubuntu)	光标移动到行首或者行尾
Shift + Ctrl + Alt + J	Select all occurrence，选中当前单词所有出现的地方
Alt + J	Next Occurence，上下左右搜寻，选择离当前光标最近的下一个出现的地方，按下 Shift 就是反向选择
F2/Shift + F2	定位到下一个或者上一个错误
Ctrl + [/Ctrl +]	定位到代码块的开始、结束
Alt + Left / Alt + Right	回到之前的文件

下调整最大列文字数量即可。

6.0.5 Tips

IntelliJ Idea 常用快捷键，如表6.1。

Class JavaLaunchHelper is implemented

启动时提示错误：

```
objc[3648]: Class JavaLaunchHelper is implemented in
both /Library/Java/JavaVirtualMachines/jdk1
.8.0_121.jdk/Contents/Home/bin/java (0
x10d19c4c0) and /Library/Java/
JavaVirtualMachines/jdk1.8.0_121.jdk/Contents
/Home/jre/lib/libinstrument.dylib (0
x10ea194e0). One of the two will be used.
Which one is undefined.
```

检查启动配置里启动模块是否选择正确。

列编辑模式

最近要做一些数据抓取转换的工作，需要编辑一些 SQL 脚本，突然发现可以根据 SQL 的注释，通过列编辑模式轻轻松松生成 SQL 的 Insert 语句。在 Mac 下，按住 Command + Shift + Alt 组合键之后，再用鼠标选择相应的区域即可。在列编辑的过程中，需要好好利用 Command + Left 到行首，Command + Right 到行尾的功能，有时候单个字符移动光标之后，也许不是想要的光标位置，此时可以整个单词的移动，利用 Alt + Left 和 Alt + Right 快捷键。在使用列编辑模式编辑 SQL 的时候，如果列自动对应上了，会出现灰色的列名，这个小细节也非常有用，在调整 SQL 语句的时候，如图6.4所示。

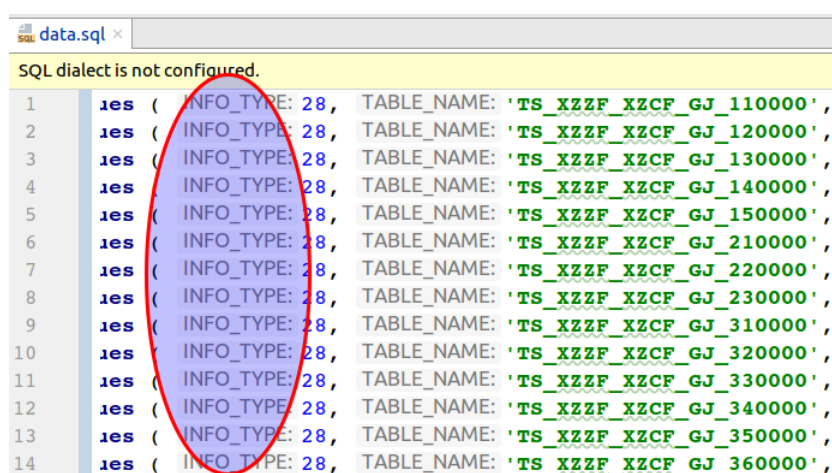


Figure 6.4: IntelliJ Idea SQL 编辑时列匹配成功提示

自动换行

有时候，代码太长，滚动滑块不是非常方便。需要设置代码自动换行。

匹配文件

有时需要使用 IntelliJ Idea 的全局搜索，在匹配文件时，如果想匹配某几类文件，可以输入如下的过滤条件。

```
1 #在 Java 文件和 log 文件中进行匹配
2 *.java,*.log
```

智能提示

在使用 IntelliJ Idea 的过程中，有时总是不出现智能提示，不出现智能提示大致分为 3 种情况。其一是勾选了省电模式，在省电模式下没有智能提示。第二是缓存问题，解决办法删除缓存 file -> invalidate caches，根据提示重启 intellij idea。情

况三配色问题。解决方法是，切换到其他配色，比如内置配色 darchla。自己的是属于勾选了省电模式。

Template

类模版：

```
1 #if (${PACKAGE_NAME} && ${PACKAGE_NAME} != "")
    package ${PACKAGE_NAME};#end
2 #parse("File Header.java")
3 /**
4  * @author jiangtingqiang@gmail.com
5  * @create ${YEAR}-${MONTH}-${DAY}-${TIME}
6  */
7 public class ${NAME} {
8 }
```

7. Gradle

一般在 Mac 下 Gradle 的安装路径为：/usr/local/Cellar/gradle/3.2.1/libexec，在 Mac 的 IntelliJ Idea 指定 Gradle 路径时，即用此路径。

7.1 基础

Gradle 使用的 Groovy 语言，Groovy 语言字符串的表示形式很多，有一些细微的区别：

```
1 def a = '单引号形式'
2 def b = "双引号形式，${v}"
3 def c = '''三个单引号形式
4 支持多行
5 不支持占位符'''
6 def d = """三个双引号形式
7 支持多行
8 支持占位符，${v}"""
9 def e = /反斜杠形式
10 支持多行
11 支持占位符，${v}/
12 def f = $/美刀反斜杠形式
13 支持多行
```

14 支持占位符, \${v}/\$

建议使用'和"这两种形式。单引号'中的内容严格对应 Java 中的 String, 不对 \$ 符号进行转义。双引号"的内容则和脚本语言的处理有点像, 如果字符中有 \$ 号的话, 则它会 \$ 表达式先求值。三个引号'''xxx''' 中的字符串支持随意换行。

7.1.1 初始化项目

输入如下命令来初始化一个项目:

```
1 gradle init
```

运行命令后, 会在项目中新建 gradlew/gradlew.bat/settings.gradle/build.gradle 文件。

7.1.2 引用本地文件

有时候项目依赖的可能不是公布在网络上一个公共库, 可能只是一个闭源的 jar 包, 只是一个本地文件而已。Gradle 中引用本地文件写法如下:

```
1 def ccDataLibs = fileTree(dir: "$rootDir/cc-data/lib"  
    , include: '*.jar')  
2  
3 dependencies {  
4     compile ccDataLibs  
5 }
```

dir 表示文件所在目录, include 表示包含哪些文件。

7.1.3 Gradle Properties 支持

Gradle 支持三种 Properties, 这三种 Properties 的作用域和初始化阶段都不一样:

System Properties: 可通过 gradle.properties 文件, 环境变量或命令行-D 参数设置, 可在 setting.gradle 或 build.gradle 中动态修改, 在 setting.gradle 中的修改对 buildscript block 可见; 所有工程可见, 不建议在 build.gradle 中修改。多子工程项目中, 子工程的 gradle.properties 会被忽略掉, 只有 root 工程的 gradle.properties 有效。

Project Properties: 可通过 gradle.properties 文件, 环境变量或命令行-P 参数设置, 优先级是: 可在 build.gradle 中动态修改, 但引用不存在的 project properties 会立即抛错, 动态修改过的 project properties 在 buildscript block 中不可见。

Project ext properties: 可在项目的 `build.gradle` 中声明和使用, 本工程和子工程可见, 不能在 `setting.gradle` 中访问, 如果有多处设置, 加载次序如下 (注意: `gradle 2.0` 是这样的, `1.10 1.12` 有 bug), 后面的覆盖前面的设置。

- from `gradle.properties` located in project build dir.
- from `gradle.properties` located in gradle user home.
- from system properties, e.g. when `-Dsome.property` is used in the command line.
- `setting.gradle`
- `build.gradle`

7.1.4 执行流程

There is a one-to-one relationship between a Project and a "build.gradle" file. During build initialisation, Gradle assembles a Project object for each project which is to participate in the build, as follows:

Create a Settings instance for the build. Evaluate the "settings.gradle" script, if present, against the Settings object to configure it. Use the configured Settings object to create the hierarchy of Project instances. Finally, evaluate each Project by executing its "build.gradle" file, if present, against the project. The projects are evaluated in breadth-wise order, such that a project is evaluated before its child projects. This order can be overridden by calling `evaluationDependsOnChildren()` or by adding an explicit evaluation dependency using `evaluationDependsOn(String)`.

7.1.5 默认 JVM

在 IntelliJ Idea 中首次导入 Gradle 项目时, 可能提示未设置 Gradle JVM, 此时在项目初始选择页面选择 `Configure > Project Defaults > Project Structure`, 在项目 Structure 中设置 JVM 即可。

7.1.6 repositories

在 Gradle 构建标本 `build.gradle` 里, 经常会看到如下脚本:

```
1 repositories {  
2     maven {  
3         url 'http://www.eveoh.nl/files/maven2'  
4     }  
5     maven {  
6         url 'http://repox.gtan.com:8078'  
7     }  
}
```

```
8     mavenCentral()
9     jcenter()
10    maven { url 'http://repo.spring.io/plugins-
        release' }
11 }
```

总的来说，只有两个标准的 Android library 文件服务器：JCenter 和 Maven Central。起初，Android Studio 选择 Maven Central 作为默认仓库。如果你使用老版本的 Android Studio 创建一个新项目，mavenCentral() 会自动的定义在 build.gradle 中。但是 Maven Central 的最大问题是对开发者不够友好。上传 library 异常困难。上传上去的开发者都是某种程度的极客。同时还因为诸如安全方面的其他原因，Android Studio 团队决定把默认的仓库替换成 JCenter。正如你看到的，一旦使用最新版本的 Android Studio 创建一个项目，JCenter() 自动被定义，而不是 mavenCentral()。mavenCentral() 表示依赖是从 Central Maven 2 仓库中获取的，库的地址是 <https://repo1.maven.org/maven2>。jcenter 表示依赖是从 Bintary' s JCenter Maven 仓库中获取的，仓库的地址是 <https://jcenter.bintray.com>，bintray 是一家提供全球企业软件开发包托管的商业公司。

7.1.7 常见问题

could not find method compile()

gradle 项目配置中引用 java 插件。

Gradle 找不到 Tasks

有时在使用 Gradle 命令编译时找不到相应到任务 (Tasks)。比如在部署脚本在 project/script/deploy 目录下，当在此目录下运行脚本时，会提示在项目目录 deploy 下找不到任务。命令如下：

```
1 ${PROGRAM_DEVELOP_PATH_BACKEND}/gradlew copyTask -x
    test
```

此时就需要指定项目的路径，将命令调整为：

```
1 ${PROGRAM_DEVELOP_PATH_BACKEND}/gradlew -p ${
    PROGRAM_DEVELOP_PATH_BACKEND} copyTask -x
    test
```

p 参数指定项目的路径。在不同目录下面执行 gradle tasks 命令，可以看到不同的任务，当在 /project/script/deploy 目录下执行 gradle tasks 命令时，是看不到

copyTask 命令的，但是在项目的根目录下执行是可以看到 copyTasks 任务，如图所示。

```
Verification tasks
-----
check - Runs all checks.
test - Runs the unit tests.

Other tasks
-----
copyTask
wrapper - Generates gradlew[.bat] scripts
```

Figure 7.1: Gradle 查看 Tasks

7.2 Gradle 对象

7.2.1 属性 (Properties)

Extra Properties

extra 属性一般用于定义常量，All extra properties¹ must be defined through the "ext" namespace. Once an extra property has been defined, it is available directly on the owning object (in the below case the Project, Task, and sub-projects respectively) and can be read and updated. Only the initial declaration that needs to be done via the namespace.

```
1 buildscript {
2     ext {
3         springBootVersion = '1.4.5.RELEASE'
4         jacksonVersion = '2.8.7'
5         springfoxVersion = '2.6.1'
6         poiVersion = "3.14"
7         aspectjVersion = '1.7.4'
8     }
9 }
```

Reading extra properties is done through the "ext" or through the owning object.

```
ext.isSnapshot = version.endsWith("-SNAPSHOT") if (isSnapshot) // do snapshot stuff
```

¹<https://docs.gradle.org/current/javadoc/org/gradle/api/Project.html#extraproperties>

定义目标 jar 包名

有时在构建后需要定义目标 jar 包的名称。

```
1 project(":web"){
2
3     description = "web"
4
5     jar{
6         baseName = "dolphin-web"
7     }
8 }
```

编译打包后的 jar 包名称即为 dolphin-web.jar。

7.2.2 Task

对于 build.gradle 配置文件，当运行 Gradle <Task> 时，Gradle 会为我们创建一个 Project 的对象，来映射 build.gradle 中的内容。其中呢，对于不属于任何 Task 范畴的代码，Gradle 会创建一个 Script 类的对象，来执行这些代码；对于 Task 的定义，Gradle 会创建 Task 对象，并将它作为 project 的属性存在（实际上是通过 getTaskName 完成的）。

拷贝文件

在项目中需要根据不同的用途生成不同的文件，比如 api 需要单独打包，平台需要单独打包，但是 api 和平台的代码都是一套代码，只是配置不同罢了。此时需要 Gradle 在构建之后根据不同的用途生成不同的包名称，使用 copy 功能来实现。首先定义项目的动态属性：

```
1 buildscript {
2     ext {
3         projectVersion = '1.1.11'
4     }
5 }
```

然后在 task 中使用此属性即可。

```
1 task copyfile(){
2     println(projectVersion)
3 }
```


一个简单的拷贝任务，在生成完毕文件后，目的文件拷贝一个接口部署包副本，接口与应用程序单独部署，并重新命名。

```
1 def ccCommonBuildScript = file("$rootDir/gradle/
    common.gradle")
2 apply from: ccCommonBuildScript
3 /**
4  * 复制一份接口的部署包
5  * 接口与平台单独部署，避免互相影响
6  */
7 task copyTask(type: Copy) {
8     from 'build/libs/credit-system-web-boot-' +
        version + '.jar'
9     into 'build/libs/api'
10    rename 'credit-system-web-boot-' + version + '.
        jar','credit-system-web-api-' + version + '.
        jar'
11 }
12
13 /**
14  * 复制一份接口的部署包
15  * 接口与平台单独部署，避免互相影响
16  * 指定路径
17  */
18 task copyTask(type: Copy) {
19     from "$projectDir/build/libs/credit-system-web-
        boot-${version}.jar"
20     into "$projectDir/build/libs/api"
21     rename "credit-system-web-boot-${version}.jar","
        credit-system-web-api-${version}.jar"
22 }
```

在这里，version 变量是来自另外一个 gradle 文件。version 变量在 common.gradle 中定义，在 build.gradle 文件中使用。



8. OWASP ZAP

8.1 基础

8.1.1 调整

OWASP 消息体太长默认无法显示，可在 Option/Display 下调整消息体的长度。

8.1.2 分析

9. OpenVPN

OpenVPN 从 2001 年开始开发，使用的是 C 语言。此处使用的 OpenVPN 版本是 2.4.1。如果使用 Mac 下的 brew 工具安装，则 OpenVPN 目录在：/usr/local/Cellar/openvpn/2.4.1，OpenVPN 的配置文件在：/usr/local/etc/openvpn。目前 OpenVPN 能在 Solaris、Linux、OpenBSD、FreeBSD、NetBSD、Mac OS X 与 Microsoft Windows 以及 Android 和 iOS 上运行，并包含了许多安全性的功能。此处的服务器使用的是 CentOS 7.3，客户端包含 Fedora 24、Ubuntu 14.04、Ubuntu 16.04、Windows 7、Windows 10。在 OpenVPN 网络中查看存活的主机：

```
1 nmap -A -T4 10.0.0.*
```

9.0.1 安装

安装基础包：

```
1 sudo yum -y install openssl openssl-devel lzo openvpn  
   easy-rsa --allowerasing  
2 #手动安装  
3 wget -c https://swupdate.openvpn.org/community/  
   releases/openvpn-2.4.1.tar.gz  
4 tar -zxvf openvpn-2.4.1.tar.gz  
5 ./configure
```

```
6 make && make install
```

LZO 是致力于解压速度的一种数据压缩算法, LZO 是 Lempel-Ziv-Oberhumer 的缩写。这个算法是无损算法, 参考实现程序是线程安全的。实现它的一个自由软件工具是 `lzop`。最初的库是用 ANSI C 编写、并且遵从 GNU 通用公共许可证发布的。现在 LZO 有用于 Perl、Python 以及 Java 的各种版本。代码版权的所有者是 Markus F. X. J. Oberhumer。

9.0.2 生成客户端证书

生成客户端证书端步骤如下:

```
1 #建立一个空的 pki 结构, 生成一系列的文件和目录
2 ./easysrsa init-pki
```

PKI: Public Key Infrastructure 公钥基础设施。生成请求:

```
1 ./easysrsa gen-req dolphinfedora
```

输入 PEM 验证码。PEM - Privacy Enhanced Mail, 打开看文本格式, 以"—BEGIN..." 开头, "—END..." 结尾, 内容是 BASE64 编码。Apache 和 *NIX 服务器偏向于使用这种编码格式。签约:

```
1 #切换到服务端生成 rsa 的目录
2 #导入 req
3 ./easysrsa import-req ~/client/easysrsa/easy-rsa-master
    /easysrsa3/pki/reqs/dolphinfedora.req
    dolphinfedora
4 #用户签约, 根据提示输入服务端的 ca 密码
5 ./easysrsa sign client dolphinfedora
```

PKI: Public Key Infrastructure 公钥基础设施。输入 PEM 验证码。PEM - Privacy Enhanced Mail, 打开看文本格式, 以"—BEGIN..." 开头, "—END..." 结尾, 内容是 BASE64 编码。查看 PEM 格式证书的信息:`openssl x509 -in certificate.pem -text -noout`。Apache 和 *NIX 服务器偏向于使用这种编码格式。服务端生成的文件有:

文件名称	说明 (Purpose)	位置
ca.crt	根证书 (Root CA certificate) 件	Server+All Clients
reqs/server.req		
reqs/dolphin.req		
private/ca.key	根证书私钥 (Root CA key)	key signing machine only
private/server.key		
issued/server.crt	服务器证书 Server Certificate	server only
issued/dolphin.crt		
dh.pem	Diffie Hellman parameters	server only

客户端生成的文件有：

序号	名称
	private/dolphinclient.key
	reqs/sdolphinclient.req

拷贝出客户端证书文件：

```

1 cp easyrsa/easy-rsa-master/easyrsa3/pki/ca.crt ~/
   dolphinfedora/
2 cp easyrsa/easy-rsa-master/easyrsa3/pki/issued/
   dolphinfedora.crt ~/dolphinfedora/
3 cp ~/client/easyrsa/easy-rsa-master/easyrsa3/pki/
   private/dolphinfedora.key ~/dolphinfedora/

```

启动 OpenVPN：

```

1 sudo openvpn server.conf
2 # Mac 下启动 OpenVPN
3 sudo /usr/local/Cellar/openvpn/2.4.1/sbin/openvpn /
   usr/local/etc/openvpn/client.conf
4 # 需要以后台交互方式启动时
5 screen sudo openvpn client.conf

```

客户端端配置如下：

```

1 client          #指定当前 VPN 是客户端
2 dev tun         #必须与服务器的保持一致

```

```

3 proto udp      #必须与服务器的保持一致
4 #指定连接的远程服务器的实际 IP 地址和端口号
5 remote 192.168.1.106 1194
6 #断线自动重新连接
7 #在网络不稳定的情况下（例如：笔记本电脑无线网络）非常
   有用
8 resolv-retry infinite
9 nobind         #不绑定特定的本地端口号
10 persist-key
11 persist-tun
12 ca ca.crt     #指定 CA 证书的文件路径
13 cert client1.crt #指定当前客户端的证书文件路径
14 key client1.key #指定当前客户端的私钥文件路径
15 ns-cert-type server #指定采用服务器校验方式
16 #如果服务器设置了防御 DoS 等攻击的 ta.key
17 #则必须每个客户端开启；如果未设置，则注释掉这一行；
18 tls-auth ta.key 1
19 comp-lzo      #与服务器保持一致
20 #指定日志文件的记录详细级别，可选 0-9，等级越高日志内
   容越详细
21 verb 3

```

配置 ns-cert-type(Netscape Cert Type) 指定为 server 主要是防止中间人攻击 (Man-in-the-Middle Attack)。在服务端做如下配置：

```

1 nsCertType server

```

9.0.3 Mac Book 客户端配置

在 Mac 下连接 OpenVPN 需要安装 Tunnelblick，Tunnelblick 是 Mac OS X 上的一个 OpenVPN 的图形化前端界面。Tunnelblick is free software licensed under the GNU General Public License, version 2 and may be distributed only in accordance with the terms of that license. 安装好 Tunnelblick 之后，直接导入相应的设置即可。如下是 Mac Book 下一个 OpenVPN 客户端可用配置的示例。

```

1 # 定义一个客户端
2 client
3 # 和服务器保持一致

```



```
4 dev tun
5 # 用 TCP 协议
6 proto tcp
7 # 指定服务器的 IP 地址和端口，可以用多行指定多台服务器，
  实现负载均衡
8 remote 106.14.30.1 1194
9 ;remote-random
10 resolv-retry infinite
11 # 客户端不需要绑定端口
12 nobind
13 persist-key
14 persist-tun
15 mute-replay-warnings
16 comp-lzo
17 verb 3
18 ;mute 20
19 ca ca.crt
20 cert dolphin.crt
21 key dolphinclient.key
```

9.0.4 Ubuntu 远程 Raspberry

OpenVPN 的一个应用就是可以方便的在不同局域网之间的计算机进行远程。
在 Raspberry 安装 VNC 服务端：

```
1 sudo apt-get install tightvncserver
```

设置密码：

```
1 tightvncserver
```

启动：

```
1 vncserver :2 -geometry 800x600 -depth 24
```

在 Ubuntu 下远程配置如图所示：

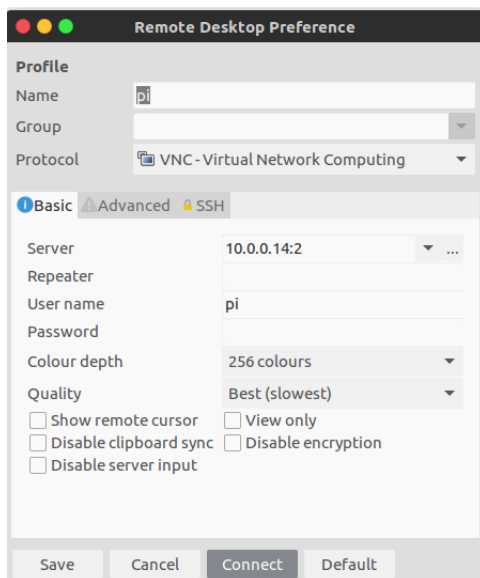


Figure 9.1: Ubuntu 远程连接配置

9.0.5 Ubuntu 远程 Ubuntu

这里的两台机器都是 Ubuntu 16.04 LTS 系统，xrdp 支持不了 13.10 的 GNOME 了，解决办法是安装 xfce 界面。被远程的机器上安装必须的组件：

```
1 #安装 xrdp
2 sudo apt-get install xrdp -y
3 #安装 vnc4server
4 sudo apt-get install vnc4server tightvncserver -y
5 #安装 xfce4
6 sudo apt-get install xubuntu-desktop -y
7 sudo /etc/init.d/xrdp restart
8 # 指定桌面管理器
9 echo "gnome-session --session=gnome-classic" > ~/.
    xsession
```

最后一个命令的作用是由于安装了 gnome 桌面，ubuntu16.04 中同时存在 unity、GNOME 多个桌面管理器，需要启动的时候指定一个，不然即使远程登录验证成功以后，也只是背景。在 Ubuntu 上使用 Remmina Remote Desktop Client 即可链接远程桌面。

9.0.6 Mac Book 远程 Fedora

在 Mac Book 远程 Linux 主机选择的是 XQuartz，安装好 XQuartz 后，直接从 XQuartz 打开的终端中运行：

```
ssh -X dolphin@10.0.0.10
```

直接在命令行中启动图形化界面，会在 Mac 上显示 Linux 主机上的软件原生 GUI 界面，就像软件在本机运行一样。X11 是 X Window System 主版本 11 的缩写，它不光是一个基本的 GUI 软件，X11 也被定义为一个网络协议，因为 X11 提供了非常灵活的网络访问接口。SSH 的 X11 forwarding 特性可以使 X client 和 X server 安全地通讯。使用 X11 forwarding 后，从 X client 到 X Server 方向的数据先被送至 SSH server，SSH server 利用和 SSH client 的安全通道转发给 SSH client，再由 SSH client 转发给 X server，从 X server 到 X client 的数据流同理。这里 SSH server 和 SSH client 充当了 X client 和 X server 间数据的转发器，由于 SSH server 和 X client、SSH client 和 X server 一般在同一台机器上，它们之间是一种安全的进程间通讯，而 SSH server 和 SSH client 间的通讯也是安全的，所以 X client 和 X server 间的通讯就是安全的。X 在 Unix-like 系统上几乎完全占据统治地位。但是仍然有人尝试提供替代品和更多的选择，过去曾经有 Sun Microsystems 的 NeWS，但它遭到市场失败；还有 NeXT 的 Display PostScript，它最终转变为苹果电脑的 Quartz for Mac OS X。

9.0.7 常见问题

得到了 Initialization Sequence Completed 消息，但是 ping 失败了 – 这通常是服务端或客户端上的防火墙过滤了 TUN/TAP 网络接口从而阻止了 VPN 网络的流量。

ssl3_get_server_certificate:certificate verify failed

在 Mac 中，连接好了并没有显示 IP 的变化。This computer's apparent public IP address was not different after connecting to client. It is still 45.76.205.201.This may mean that your VPN is not configured correctly.

OpenVPN is connected to the server but your IP address does not change

If OpenVPN connects to the server properly but your IP address does not change, you are probably missing the "--redirect-gateway" option. Add the following line: --redirect-gateway def1 to your configuration file.

By default, OpenVPN only sends some traffic through the VPN —traffic that is specifically destined for the VPN network itself. The "--redirect-gateway" option tells OpenVPN to send all traffic through the VPN. (Leave out the "--" when it is put in the

configuration file.)

An alternative to putting "redirect gateway def1" in the configuration file is to "push" it from the VPN server to the client.

10. Nmap(GNU General Public License)

Nmap(Network Mapper), 网络映射器, 是一款开放源代码的网络探测和安全审核的工具。它的设计目标是快速地扫描大型网络, 当然用它扫描单个主机也没有问题。Nmap 以新颖的方式使用原始 IP 报文来发现网络上有哪些主机, 那些主机提供什么服务 (应用程序名和版本), 那些服务运行在什么操作系统 (包括版本信息), 它们使用什么类型的报文过滤器/防火墙, 以及一堆其它功能。虽然 Nmap 通常用于安全审核, 许多系统管理员和网络管理员也用它来做一些日常的工作, 比如查看整个网络的信息, 管理服务升级计划, 以及监视主机和服务的运行。Nmap 脚本引擎 (Nmap Script Engine) 是 Nmap 最有力灵活的的一个特性。它允许用户撰写和分享一些简单的脚本来一些较大的网络进行扫描任务。基本上这些脚本是用 Lua 编程语言来完成的。通常 Nmap 的脚本引擎可以完成很多事情。

10.0.1 主机发现 (Host Discovery)

主机发现顾名思义就是发现所要扫描的主机是否是正在运行的状态。输入如下命令开始主机扫描

```
1 nmap -F -sT -v nmap.org
2 # 主机发现, 生成存活主机列表
3 nmap -sn -T4 -oG Discovery.gnmap 192.168.24.0/24
4 grep "Status: Up" Discovery.gnmap | cut -f 2 -d ' ' >
   LiveHosts.txt
```

- -F 扫描 100 个最有可能开放的端口,F 表示快速模式 (Fast Model), 比默认模式扫描更少的端口 (Scan fewer ports than the default scan), 如果需要扫描更多的端口, 指定 p 参数, 如下命令行片段所示:

```
1 nmap -p 0-10000 -A -v <Host>
```

端口号最好分段指定, 否则扫描的速度会非常缓慢, 例如需要扫描 0-65535 端口, 可以先扫描 0-10000 端口, 再扫描 10001-20000, 依次类推。

- -v 获取扫描的信息
- -sT(Scan using TCP) 采用的是 TCP 扫描, 不写也是可以的, 默认采用的就是 TCP 扫描
- -sn 参数表示 ping 扫描, 禁用端口扫描
- -oG 表示输出 Grepable 格式

Nmap 输出的是扫描目标的列表, 以及每个目标的补充信息, 至于哪些信息则依赖于所使用的选项。“所感兴趣的端口表格”是其中的关键。那张表列出端口号, 协议, 服务名称和状态。状态可能是 open(开放的), filtered(被过滤的), closed(关闭的), 或者 unfiltered(未被过滤的)。Open(开放的)意味着目标机器上的应用程序正在该端口监听连接/报文。filtered(被过滤的)意味着防火墙, 过滤器或者其它网络障碍阻止了该端口被访问, Nmap 无法得知它是 open(开放的) 还是 closed(关闭的)。closed(关闭的) 端口没有应用程序在它上面监听, 但是他们随时可能开放。当端口对 Nmap 的探测做出响应, 但是 Nmap 无法确定它们是关闭还是开放时, 这些端口就被认为是 unfiltered(未被过滤的) 如果 Nmap 报告状态组合 open|filtered 和 closed|filtered 时, 那说明 Nmap 无法确定该端口处于两个状态中的哪一个状态。当要求进行版本探测时, 端口表也可以包含软件的版本信息。当要求进行 IP 协议扫描时 (-sO), Nmap 提供关于所支持的 IP 协议而不是正在监听的端口的信息。获取远程主机的系统类型及开放端口:

```
1 nmap -sS -P0 -sV -O <target>
```

10.0.2 端口扫描 (Port Scanning)

端口扫描是 Nmap 最基本最核心的功能, 用于确定目标主机的 TCP/UDP 端口的开放情况。端口扫描简单示例如下代码片段所示:

```
1 #服务扫描
2 nmap-T4 -sV targetip
3
```

```

4 #扫描淘宝 IP
5 nmap -p 0-1000 -v 218.201.46.124
6 nmap -T4 -A -v 218.201.46.124
7
8 # 扫描指定主机的 3306 端口
9 nmap -sS -p 3306 -v 192.168.31.25

```

通过扫描可以发现一些主机信息，比如淘宝的数字证书用的是 GlobalSign，加密位数是 2048，签名算法是 sha256WithRSAEncryption，操作系统猜测是 Linux 系列的，内核版本可能是 3.X 或者 2.6.X，具体信息如下。

```

1 443/tcp open  ssl/http Tengine httpd
2 |_http-server-header: Tengine
3 |_http-title: 501 Not Implemented
4 | ssl-cert: Subject: commonName=*.tmall.com/
      organizationName=Alibaba (China) Technology
      Co., Ltd./stateOrProvinceName=ZheJiang/
      countryName=CN
5 | Issuer: commonName=GlobalSign Organization
      Validation CA - SHA256 - G2/organizationName=
      GlobalSign nv-sa/countryName=BE
6 | Public Key type: rsa
7 | Public Key bits: 2048
8 | Signature Algorithm: sha256WithRSAEncryption
9 | Not valid before: 2015-12-14T10:38:38
10 | Not valid after: 2016-12-14T10:38:38
11 | MD5: 5d67 3ca3 7f0e 2ab7 7b4f 59d5 0700 7223
12 |_SHA-1: d048 e9cf 5487 5030 9f26 e638 7d3f 94ad a2b3
      e6fa

```

默认情况下，Nmap 会扫描 1000 个最有可能开放的 TCP 端口。

1) 公认端口 (0 ~ 1023)，又称常用端口，为已经公认定义或为将要公认定义的软件保留的。这些端口紧密绑定一些服务且明确表示了某种服务协议。如 80 端口表示 HTTP 协议。2) 注册端口 (1024 ~ 49151)，又称保留端口，这些端口松散绑定一些服务。3) 动态/私有端口 (49152 ~ 65535)。理论上不应为服务器分配这些端口。按协议类型可以将端口划分为 TCP 和 UDP 端口。1) TCP 端口是指传输控制协议端口，需要在客户端和服务端之间建立连接，提供可靠的数据传输。如

Telnet 服务的 23 端口。2) UDP 端口是指用户数据包协议端口，不需要在客户端和服务器之间建立连接。常见的端口有 DNS 服务的 53 端口。有的服务器为了安全会修改默认端口，比如将 ssh 默认的 22 端口修改为 50000，不过通过 Nmap 扫描也会很容易的发现。详细的扫描指令如下所示：

```
nmap -sS -sU -T4 -A -v -PE -PP -PS80,443 -PA3389 -
    PU40125 -PY -g 53 --script "default or (
    discovery and safe)" 12.26.32.14
```

- -sS 参数 (TCP SYN Scan) 可以利用基本的 SYN 扫描方式探测其端口开放状态
- -sU 参数代表 UDP Scan
- -T<0-5>: Set timing template (higher is faster)
- -A: Enable OS detection, version detection, script scanning, and traceroute
- -v: Increase verbosity level (use -vv or more for greater effect)
- -PE/PP/PM: ICMP echo, timestamp, and netmask request discovery probes
- -PS/PA/PU/PY[portlist]: TCP SYN/ACK, UDP or SCTP discovery to given ports
- -g/--source-port <portnum>: Use given port number

"ppp0" is not an ethernet device

在使用 nmap 扫描时候，提示

```
Only ethernet devices can be used for raw scans on
Windows, and "ppp0" is not an ethernet device
. Use the --unprivileged option for this scan
. QUITTING!
```

因为用 Nmap 扫描时，不能用 pppoe 的拨号口，只能用以太网口。根据提示带参数--unprivileged 就可以了，注意扫描的命令不能带-A 参数，否则会要求 root 权限，无法启动扫描。

10.0.3 版本侦测 (Version Detection)

版本侦测在后面添加 sV(Service/Version) 参数即可。

```
-sV: Probe open ports to determine service/version
    info
```


10.0.4 Zenmap

Zenmap 是 Nmap 的官方 GUI，可以运行在 Linux, Windows, Mac OS X, BSD 等操作系统上。

```
1 # Mac 下获取 Zenmap 安装文件
2 wget -c https://nmap.org/dist/nmap-7.31.dmg
```


11. Raspberry Pi

11.1 基础

11.1.1 连接 (Connection)

Raspberry 可以通过网线直接与 PC 连接，输入如下命令查看 Raspberry 的 IP：

```
1 arp -a
```

初次登录需要通过有线连接，设置完成后，如果是连接的是隐藏网络，有可能无线无法获取 IP。此时网络设置需要添加 scan_ssid：

```
1 network={
2     ssid="爱生活"
3     scan_ssid=1
4     psk="15023396614"
5     priority=3
6 }
```

SSID 是个笼统的概念，包含了 ESSID 和 BSSID，用来区分不同的网络，最多可以有 32 个字符，无线网卡设置了不同的 SSID 就可以进入不同网络，SSID 通常由 AP 广播出来，通过 XP 自带的扫描功能可以查看当前区域内的 SSID。出于安全考虑可以不广播 SSID，此时用户就要手工设置 SSID 才能进入相应的网络。简单说，SSID 就是一个局域网的名称，只有设置为名称相同 SSID 的值的电脑才能互

相通信。

检查配置文件是否有错误:

```
1 sudo wpa_supplicant -c /etc/wpa_supplicant/  
    wpa_supplicant.conf -i wlan0
```

安装步骤:

```
1 # 安装tightvncserver  
2 sudo apt-get install tightvncserver - y  
3 # 设置密码  
4 vncpasswd
```

Raspberry Pi 的网络配置, 需要编辑/etc/wpa_supplicant/wpa_supplicant.conf 文件。编辑后重启网络:

```
1 /etc/init.d/networking restart
```



12. Widgets

12.1 Vim Editor

复制全部:

全部删除: 按 `esc` 后, 然后 `dG` 全部复制: 按 `esc` 后, 然后 `ggyG` 全选高亮显示: 按 `esc` 后, 然后 `ggvG` 或者 `ggVG`

12.2 Little Tool

12.2.1 SQuirreL SQL Client

目前常见的数据库有几十种, 每一种又有自己连接客户端, 平时工作中会经常接触不同类型的数据库, 一个通用的数据库客户端非常有必要。SQuirreL SQL Client¹优点是开源、跨平台、支持不同的数据库, 关系型和非关系型。只需要使用一个工具即可操作多种数据库。可以在 `Session->Session Properties->SQL` 中设置编辑器的字体 (默认字体比较小), 还可以设置将每张表包含的数据总量显示出来 (Show SQL Results Metadata), 如图12.1所示。在使用数据库的过程中, 还有一个比较不方便的地方就是一段时间后, 数据库会断开连接, 也许是安全防火墙设置的缘故。可以设置定期执行一个 SQL, 防止连接断开, this tab allows you to configure an SQL statement to execute periodically in the background. This can help keep the session connection alive for firewalls that terminate connections due to inactivity.

¹<http://www.squirrelsql.org/>

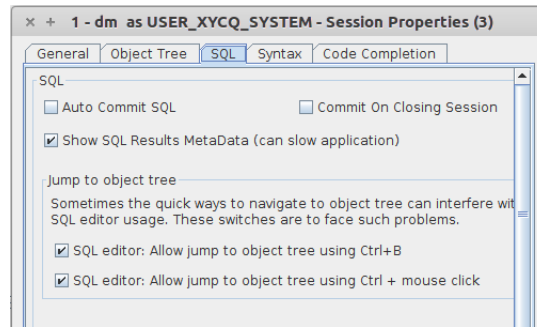


Figure 12.1: Squirrel SQL Client 显示表总数

如果需要在对象树上搜索对象,则可以使用类似 SQL 通配符的方式,如图12.2所示。匹配的对象会高亮显示。

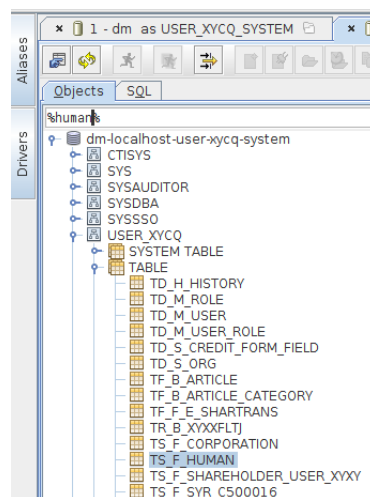


Figure 12.2: Squirrel SQL Client 搜索对象

Code Completion

Squirrel SQL Client 有代码自动完成插件 codecompletion, 输入 SQL 的过程中按下快捷键 Ctrl + Space 即可。但是 Ctrl + Space 快捷键有可能被输入法占用了, 此时修改输入法的快捷键。在路径 Text Entry->Input Method Configuration->Global Config 的热键 (Hotkey) 设置中修改。在 Session Properties 中可以设置显示代码自动提示时显示列的类型, 最终的效果如图12.3所示。

DBCoppy

可以在同一个数据库或不同数据库中, 直接迁移一个或多个数据表! 如此, 我们就可以省去了不同方言的建表语句, 以及数据导出成 txt, 再导入到另外数据库的繁琐操作。直接右键表, 复制表, 粘贴表即可, 如图12.4所示。当数据量比较大时, 拷贝速度还是比较慢的。

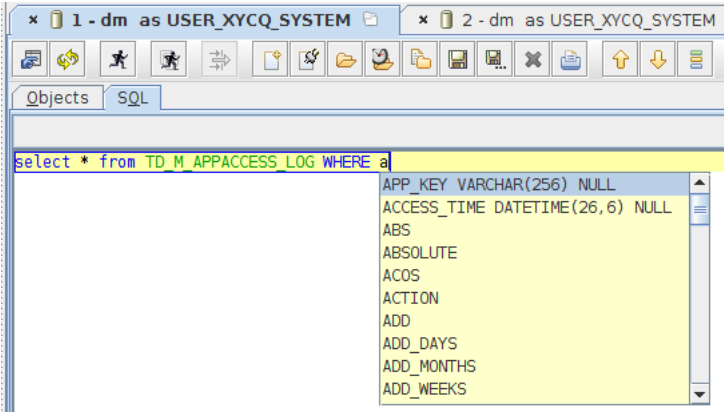


Figure 12.3: 代码自动完成

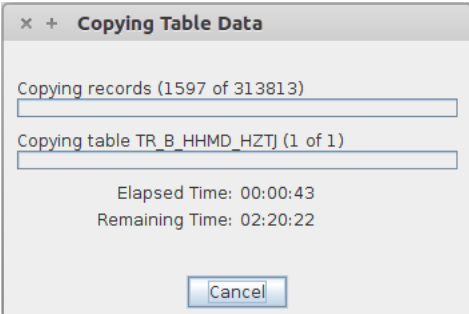


Figure 12.4: 复制表数据示例

12.2.2 You-Get

有时看点视频真闹心, 广告多, 时间长, 还有各种限制, 如果网速不是很快, 不稳定的话, 还非常卡, 还有 Linux 下对 Flash 支持的也不是很好, 还要安装 Flash 插件, 非常麻烦, You-Get 工具可以解决这个问题. 使用 You-Get 下载的命令为:

```
1 you-get <url>
2 you-get http://www.fsf.org/blogs/rms/20140407-geneva-
   tedx-talk-free-software-free-society
```

12.2.3 transmission

在使用磁力链接下载元素据时, 如果下载时间太长, 一般是网络的问题, 电信的网络普遍优于长城等。

12.2.4 rsync

rsync 命令是一个远程数据同步工具, 可通过 LAN/WAN 快速同步多台主机间的文件。rsync 使用所谓的“rsync 算法”来使本地和远程两个主机之间的文件达到同步, 这个算法只传送两个文件的不同部分, 而不是每次都整份传送, 因此速度相当快。使用一个远程 shell 程序 (如 rsh、ssh) 来实现将本地机器的内容拷贝到远程机器。当 DST 路径地址包含单个冒号":" 分隔符时启动该模式。

```
1 rsync -avz pi@10.42.0.22:app-soft .
```

12.2.5 ag

silversearcher-ag 可递归搜索文件内容。

```
1 apt-get install silversearcher-ag
```

12.2.6 Pandoc

Pandoc 使用 haskell 开发的一个文档格式转换软件。在编写文档时习惯使用 \LaTeX 来编写, \LaTeX 生成的 pdf 文档美观, Word 格式在不同平台 (Mac OS X/Linux/Windows) 不同的软件 (Microsoft Office/金山/Libre Office) 打开格式可能会不一致, 但是有许多同事习惯使用 Word, 有时发给对方时也需要作一些编辑和调整, Pandoc 就可以满足既可以用 \LaTeX 来编辑文档, 也可以将 Tex 文档转换为 Word 方便同事:


```
1 pandoc -s interface.tex -o example30.docx
```

另外使用 \LaTeX 还有一个比较方便的地方是可以容易的以模块来管理文档，比如接口文档分为公共部分，所有调用方都可以看到，但是针对每个调用方可能会有一些定制的内容，当要修改公共部分时，如果是采用 Word 文件分散管理会出现不一致的情况，有的文件修改了，有的没有改。使用 \LaTeX 就只需要改动一个公共模块即可。其他引用公共的地方自动更新。

12.2.7 Curl

内部接口测试：

```
1 curl -i -H "APPID:hlb1451j6d136334gh2" \  
2 -H "ECHOSTR:sdsaasf" \  
3 -H "TOKEN:e02016ed4adfd9d9743ab70cf389ef9877c4a0e" \  
4 -H "TIMESTAMP:2016-12-19 16 (tel:2016121916):58:02" \  
    http://localhost:28080/api/article
```

使用如下命令上传文件：

```
1 curl -i -X POST -F "filename=@example.tar.gz;  
    filename1=@example1.tar.gz" http://localhost  
    :28080/api/article  
2 curl -i -X POST host:port/post-file -H "Content-Type:  
    text/xml" --data-binary "@path/to/file"  
3 # 使用 curl 將文件上傳到服務器，文件類型可以任意  
4 curl -i -X POST -H "APPID:hlb1451j6d136334gh2" \  
5 -H "ECHOSTR:sdsaasf" \  
6 -H "TOKEN:e02016ed4adfd9d9743ab70cf389ef9877c4a0e" \  
7 -H "TIMESTAMP:2016-12-19 16 (tel:2016121916):58:02" \  
8 -F "upfile=@main.synctex.gz" http://localhost:28080/  
    api/article
```

The `-i` option tells curl to show the response headers as well。如果使用了 `-F` 参数，curl 就会以 `multipart/form-data` 的方式发送 POST 请求。`-F` 参数以 `name=value` 的方式来指定参数内容，如果值是一个文件，则需要以 `name=@file` 的方式来指定。文件的路径需要有一个 `@` 符号，所以 curl 知道从文件中读取。`upfile` 對應服務端的方法參數。有时需要查看接口详细的响应时间，采用如下命令：

```
1 # 文件 curl-format.txt 写入的内容
2 time_namelookup:  %{time_namelookup}\n
3 time_connect:    %{time_connect}\n
4 time_appconnect:  %{time_appconnect}\n
5 time_pretransfer: %{time_pretransfer}\n
6 time_redirect:    %{time_redirect}\n
7 time_starttransfer:  %{time_starttransfer}\n
8 -----\n
9 time_total:  %{time_total}\n
10
11 curl -w "@curl-format.txt" -s -H "APPID:" \
12 -H "TIMESTAMP:2016-12-19 16 (tel:2016121916):58:02" \
13 -H "ECHOSTR:sdsaasf" -H "TOKEN:" \
14 -H "Accept: application/json, text/plain" \
15 http://localhost:28080/api/xysj/search?qymc=长安福特
```

12.2.8 iostat

查看磁盘的 IO 情况，每秒刷新一次：

```
1 iostat 2 -cdxk
```

参数 -d(Device) 表示，显示设备（磁盘）使用状态；-k 某些使用 block 为单位的列强制使用 Kilobytes 为单位；2 表示，数据显示每隔 2 秒刷新一次。iostat 还有一个比较常用的选项 -x，该选项将用于显示和 io 相关的扩展数据。-c 用来获取 cpu 部分状态值。为显示更详细的 io 设备统计信息，可以使用 -x 选项。

12.2.9 Zeal

Zeal 是一个离线的文档浏览工具。查看 Nginx Stream 相关文档信息，示例如图12.5所示。搜索以 nginx 和冒号开始，表示搜索与 Nignx 相关的信息。

12.2.10 Thunderbird 邮件客户端

Thunderbird 邮件客户端配置腾讯企业邮箱如图12.6所示。

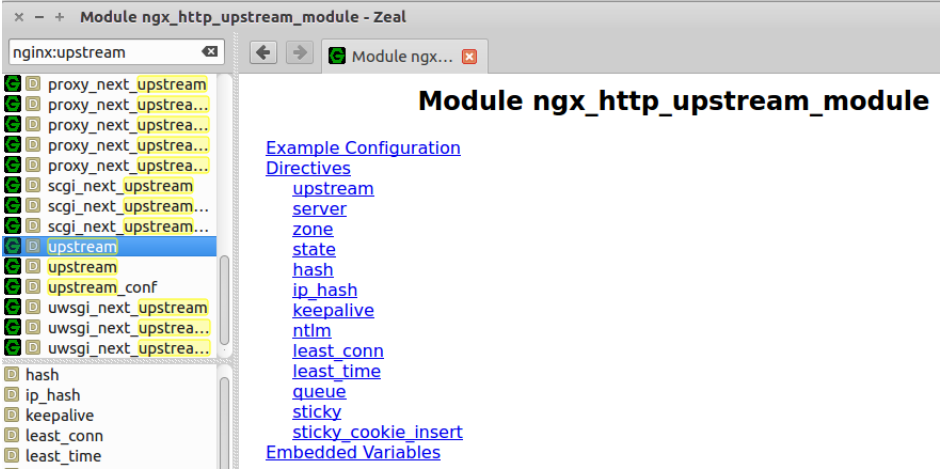


Figure 12.5: Zeal 文档搜索示例

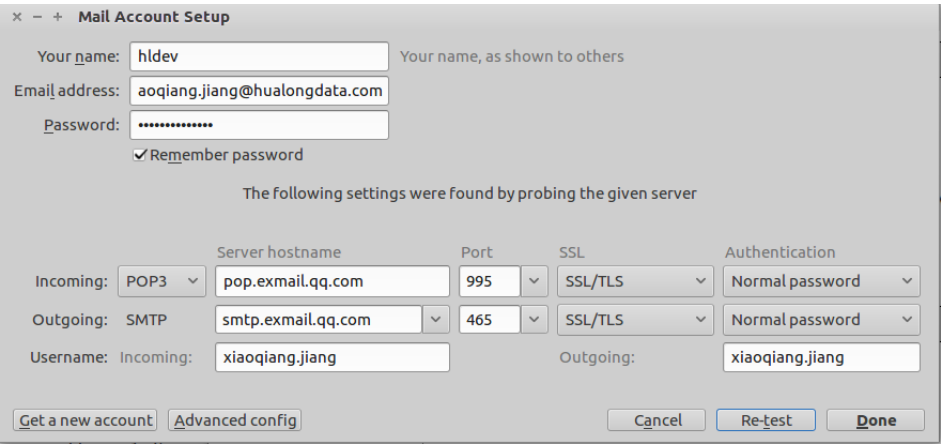


Figure 12.6: Thunderbird 配置企业邮箱

12.2.11 jq

项目里同事根据接口返回的数据建立数据库表格时，需要看格式化后的 Json，由于只能在终端中请求接口数据，未格式化的 Json 返回根本没法阅读。由于目前还未发现 Json 格式化工具，所以只能时粘贴 Json 到网络上的在线网页里面进行格式化，由于网络用的是物理隔离的内网，甭提有多麻烦了，而且都是重复劳动，而 jq 采用 C 语言编写 (开始还以为是用 Python 写的)，没有运行时依赖 (jq is written in portable C, and it has zero runtime dependencies.)，直接在服务器上源码编译安装即可 (服务器物理隔离，只能源码安装)。没有它，估计耗费的时间不下几个小时。jq 直接在终端中就可以查看格式化好的 Json。需要说明的是 jq 只能接受 well form 的 JSON 字符串作为输入内容。也就是说输入内容必须严格遵循 JSON 格式的标准。所有的属性名必须是以双引号包括的字符串。对象的最后一个属性的末尾或者数组的最后一个元素的末尾不能有逗号。否则 jq 会抛出无法解析 JSON 的错误。

常用模式

获取所有的 key:

```
1 echo '{"foo": 42, "bar": "less interesting data"}' |  
    jq 'keys'  
2 # 获取某一个 value 的值  
3 echo '{"foo": 42, "bar": "less interesting data"}' |  
    jq '.bar'
```

嵌套数据查看:

```
1 curl -H "APPID:" -H "TIMESTAMP:2016-12-19 16 (tel  
    :2016121916):58:02" -H "ECHOSTR:" -H "TOKEN:"  
    http://10.10.1.12:28080/api/xzxk?xdr= | jq  
    '.data.totalElements'
```

12.3 Nginx

安装配置好 nginx 服务器后默认目录是 /usr/share/nginx/html。nginx 模块一般被分成三大类: handler、filter 和 upstream。Mac 下 Nginx 配置文件的路径为: /usr/local/etc/nginx/nginx.conf。

12.3.1 X-Forwarded-For

X-Forwarded-For 是一个 HTTP 扩展头部。HTTP/1.1 (RFC 2616) 协议并没有对它的定义，它最开始是由 Squid 这个缓存代理软件引入，用来表示 HTTP 请求端真实 IP。如今它已经成为事实上的标准，被各大 HTTP 代理、负载均衡等转发服务广泛使用，并被写入 RFC 7239 (Forwarded HTTP Extension) 标准之中。在默认情况下，Nginx 并不会对 X-Forwarded-For 头做任何的处理，除非用户使用 `proxy_set_header` 参数设置：

```
1 proxy_set_header X-Forwarded-For  
    $proxy_add_x_forwarded_for;
```

X-Forwarded-For 请求头格式：

```
1 X-Forwarded-For: IP0, IP1, IP2
```

XFF 的内容由「英文逗号 + 空格」隔开的多个部分组成，最开始的是离服务端最远的设备 IP，然后是每一级代理设备的 IP。

12.3.2 Nginx 获取真实 IP

做异地登陆的判断，或者统计 ip 访问次数等，通常情况下我们使用 `request.getRemoteAddr()` 就可以获取到客户端 ip，但是当我们使用了 nginx 作为反向代理后，使用 `request.getRemoteAddr()` 获取到的就一直是 nginx 服务器的 ip 的地址。Nginx 作为 HTTP 代理转发前端时，后端服务无法获知前端访问客户的 IP 地址。要获取客户端 IP，需要将 `http_realip_module` 编译进入 Nginx 中。查看 Nginx 安装了哪些模块，使用如下命令：

```
1 nginx -V
```

在命令的输出结果中，可以看到 Nginx 的版本，Nginx 的编译参数，和 Nginx 已经包含有哪些模块。此处显示已经编译了 `http_realip_module` 模块。在服务器上的项目使用的 `tengine`，经过查看是没有编译 `http_realip_module` 模块，从 `tengine` 官网上下载好源码，输入如下命令进行编译：

```
1 # 在 Ubuntu 14.04 LTS 下安装 pcre 依赖  
2 # pcre:Perl Compatible Regular Expressions  
3 sudo apt-get install libpcre3 libpcre3-dev  
4 # 指定预编译参数，仅仅指定路径  
5 sudo ./configure --prefix=/opt/tengine  
6 # 将 realip 模块编译进入
```

```
7 sudo ./configure --prefix=/opt/tengine --with-
    http_realip_module
8 # 编译
9 sudo make
10 # 安装
11 sudo make install
12 ./configure --prefix=/usr/local/tengine-2.1.2 --with-
    openssl=/Users/dolphin/source/openssl
```

不指定 prefix, 则可执行文件默认放在 /usr/local/bin, 库文件默认放在 /usr/local/lib, 配置文件默认放在 /usr/local/etc。其它的资源文件放在 /usr/local/share。你要卸载这个程序, 要么在原来的 make 目录下用一次 make uninstall (前提是 make 文件指定过 uninstall), 要么去上述目录里面把相关的文件一个个手工删掉。指定 prefix, 直接删掉一个文件夹就够了, 这也是制定 prefix 一个比较方便的地方。在编译时, 还需要指定 pcre 的版本, 因为 Ubuntu 16.04 LTS 里有时安装有 pcre 3, 而部署的电脑上不一定安装有 pcre 3, 而且 pcre 3 的源码不容易找到。所以编译命令如下:

```
1 # 将 realip 模块编译进入, 并指定 pcre
2 sudo ./configure --prefix=/opt/tengine --with-
    http_realip_module --with-pcre=/root/software
    /pcre-8.40 --with-openssl=/root/software/
    openssl-OpenSSL_1_1_0e --without-
    http_gzip_module
3
4 ./configure --prefix=/opt/tengine --with-
    http_realip_module --with-pcre=/root/software
    /pcre-8.40 --without-http_gzip_module
5
6 #本机编译 (不成功)
7 ./configure --prefix=/opt/tengine --with-
    http_realip_module --with-pcre=/home/hldev/
    Downloads/pcre-8.40 --with-openssl=/home/
    hldev/Downloads/openssl-OpenSSL_1_0_1e --
    without-http_gzip_module
8
9 #本机编译 (成功)
```

```
10 -prefix=/opt/tengine --with-http_realip_module --with
    -pcre=/home/hldev/Downloads/pcre-8.40 --with-
    openssl=/home/hldev/software/openssl-
    OpenSSL_1_0_2g --without-http_gzip_module
11
12 # 构建程序
13 make
14
15 #安装程序
16 make install
17
18 #服务器
19 ./configure --prefix=/opt/tengine
20
21 ./configure: error: SSL modules require the OpenSSL
    library.
22 You can either do not enable the modules, or install
    the OpenSSL library
23 into the system, or build the OpenSSL library
    statically from the source
24 with nginx by using --with-openssl=<path> option.
```

其中`--with-pcre`表示 pcre(Perl Compatible Regular Expressions) 的源代码目录。在编译时，还需要注意 openssl 的版本，本地电脑版本是 1.0.2g，服务端的版本是 1.0.1e。服务器端编译 Nginx 的一个命令：

```
1 ./configure --prefix=/opt/tengine --with-openssl=/usr
    /local/src/openssl-OpenSSL_1_0_2g/ --without-
    http_gzip_module --with-pcre=/usr/local/src/
    pcre-8.40/
```

这里编译的时候有一个小细节需要注意，需要将源码包存放到 `/usr/local/src` 目录下。一直没有编译成功也许是这个小小的细节，开始时是将源码包随意存放的一个目录。以上编译的命令没有包含 Zlib，也没有包含 `http_realip_module` 模块。最终的编译命令如下：

```
1 ./configure --prefix=/opt/tengine --with-openssl=/usr
```

```
/local/src/openssl-OpenSSL_1_0_2g/ --with-  
pcre=/usr/local/src/pcre-8.40/ --with-zlib=/  
usr/local/src/zlib-1.2.11 --with-  
http_realip_module
```

编译完毕之后，输入如下命令：

```
1 # 查看 Tengine 版本、模块信息，编译时所使用的参数  
2 ./nginx -V
```

即可看到 http_realip_module 已经编译到 Nginx 中了。Nginx 的 http realip module 等于 Apache 的 mod_rpaf，用于接受前端发来的 IP head 信息，从获取到真实的用户 IP。将获取真实 IP 的模块编译进入 Nginx 后，还需要在对应的 http、server、location 中加入以下参数：

```
1 #指令是告诉 nginx，10.10.1.11 是我们的反向代服务器  
2 #不是真实的用户 IP  
3 set_real_ip_from 10.10.1.11;  
4 #告诉 nginx 真正的用户 IP 是存在 X-Forwarded-For 请求  
   头中  
5 real_ip_header X-Real-IP;
```

其中 set_real_ip_from 可以指定某个网段。这个指令指定信任的代理 IP，它们将会以精确的替换 IP 转发。0.8.22 后可以指定 Unix sockets。real_ip_header 设置需要使用哪个头来确定替换的 IP 地址。由于项目中使用的 Nginx 作为反向代理，所以配置如下：

```
1 location /inapi {  
2 proxy_pass http://localhost:28080;  
3 proxy_set_header X-Real-IP $remote_addr;  
4 proxy_redirect off;  
5 }
```

在另一台电脑上访问服务端，在后端根据新添加的 Header 获取到的 IP 地址如图12.7所示：

此处获取到的 IP 是用户的真实 IP，而不是反向代理服务器的 IP。

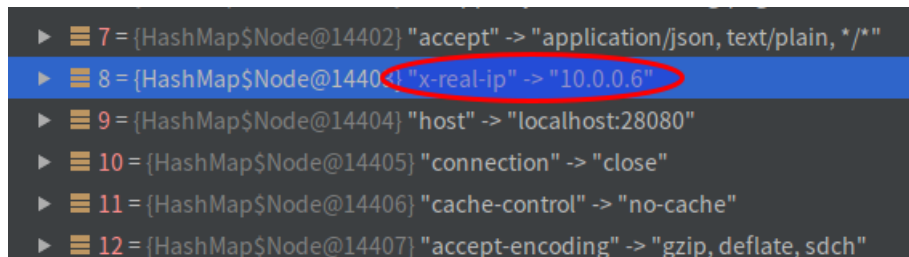


Figure 12.7: Nginx 获取用户真实访问 IP

12.3.3 Nginx 并发

如何设置能限制某个 IP 某一段时间段的访问次数是一个让人头疼的问题，特别面对恶意的 ddos 攻击的时候。其中 CC 攻击（Challenge Collapsar）是 DDOS（分布式拒绝服务）的一种，也是一种常见的网站攻击方法，攻击者通过代理服务器或者肉鸡向受害主机不停地发大量数据包，造成对方服务器资源耗尽，一直到宕机崩溃。

cc 攻击一般就是使用有限的 ip 数对服务器频繁发送数据来达到攻击的目的，nginx 可以通过 `HttpLimitReqModule` 和 `HttpLimitZoneModule` 配置来限制 ip 在同一时间段的访问次数来防 cc 攻击。

`HttpLimitReqModule` 用来限制单位时间内连接数的模块，使用 `limit_req_zone` 和 `limit_req` 指令配合使用来达到限制。一旦并发连接超过指定数量，就会返回 503 错误。`limit_req_zone` 用来限制单位时间内的请求数，即速率限制，采用的漏桶算法“leaky bucket”。

```

1 http {
2     limit_conn_log_level error;
3     limit_conn_status 503;
4     #Zone=one 或 allips 表示设置了
5     #名为 “one” 或 “allips” 的存储区
6     #大小为 10 兆字节
7     limit_conn_zone $binary_remote_addr zone=one:10m;
8     limit_conn_zone $server_name zone=perserver:10m;
9     #rate=10r/s 的意思是允许 1 秒钟不超过 1000 个请求
10    limit_req_zone $binary_remote_addr zone=allips
        :100m rate=1000r/s;
11    server {
12        limit_conn one 100;
13        limit_conn perserver 3000;

```

```
14     limit_req zone=allips burst=5 nodelay;  
15 }  
16 }
```

HttpLimitConnModule 用来限制单个 ip 的并发连接数,使用 limit_zone 和 limit_conn 指令,这两个模块的区别前一个是对一段时间内的连接数限制,后者是对同一时刻的连接数限制。

12.3.4 URI 长度限制

在 Http1.1 协议中并没有提出针对 URL 的长度进行限制, RFC 协议里面是这样描述的, HTTP 协议并不对 URI 的长度做任何的限制, 服务器端必须能够处理任何它们所提供服务多能接受的 URI, 并且能够处理无限长度的 URI, 如果服务器不能处理过长的 URI, 那么应该返回 414 状态码。接触的最多的服务器类型就是 Nginx 和 Tomcat, 对于 url 的长度限制, 它们都是通过控制 http 请求头的长度来进行限制的, nginx 的配置参数为 large_client_header_buffers, tomcat 的请求配置参数为 maxHttpRequestSize。

```
1 # Nginx 配置 HTTP 请求头长度  
2 client_header_buffer_size 2048k;
```

项目采用 Spring Boot, 所使用的 Tomcat 为内嵌的 Tomcat。在 application.properties 中, 增加如下配置:

```
1 #单位为 KB, 如果没有指定, 默认为 8192(8KB)  
2 server.max-http-header-size=1024
```

奇怪的是, 在本机设置的 1024KB 可以查询, 部署到服务器上, 参数必须再调大才能够查询, 否则提示请求头太大的错误。真的遇到鬼了。在不同的操作系统上单位不一样吗? 见鬼

12.3.5 开启 gzip 压缩

Nginx 开启 Gzip 压缩配置如下:

```
1 gzip on;  
2 gzip_min_length 1k;  
3 gzip_buffers 4 16k;  
4 gzip_comp_level 2;
```

```
5 gzip_types application/javascript application/json
   text/plain application/x-javascript text/css
   application/xml text/javascript application/x
   -httpd-php image/jpeg image/gif image/png;
6 gzip_vary off;
7 gzip_disable "MSIE [1-6]";
```

用 curl 测试是否开启了压缩：

```
1 curl -I -H "Accept-Encoding: gzip, deflate" "http://
   creditsystem.test/main"
2 curl -I -H "Accept-Encoding: gzip, deflate" "http
   ://10.10.1.11/main"
```

-I 选项与 -header 选项是等价的，表示只打印出 HTTP 头部。开启压缩成功后，可以看到一句：Content-Encoding: gzip。

12.3.6 Nginx 配置

简单的配置：

```
1 user www-data;
2 worker_processes 4;
3 pid /run/nginx.pid;
4
5 events {
6     worker_connections 768;
7 }
8
9 http {
10     server{
11         listen 80;
12         # 定义网站默认根目录
13         root /opt/dolphin/frontend/dist;
14
15         location /api {
16             proxy_pass http://localhost:8011;
17         }
```

```
18
19     location / {
20         root /opt/dolphin/frontend/dist;
21         index index.html;
22     }
23 }
24 }
```

12.3.7 常见问题

open() "/opt/tengine/conf/nginx.conf" failed (2: No such file or directory)

以上的路径是编译时 Nginx 配置文件的路径，如果在启动时不指定配置文件和日志文件的路径，那么 Nginx 默认使用的时编译时的文件路径，但是在这里已经将编译文件移动到另外的文件夹里面了，所以需要显示的指定配置文件的路径：

```
1 ./nginx -c /opt/app/local/tengine/conf/nginx.conf
```

也可以指定 prefix 文件目录，这样就会更改默认的 Nginx 主目录了：

```
1 #重新加载配置（测试环境）
2 /opt/app/local/tengine/sbin/nginx -s reload -p /opt/
  app/local/tengine
3 /opt/app/local/tengine/sbin/nginx -c /opt/app/local/
  tengine/conf/nginx.conf -p /opt/app/local/
  tengine
4 /opt/home/local/tengine/sbin/nginx -c /home/app/
  local/tengine/conf/nginx.conf -p /home/app/
  local/tengine
```

其中 opt 目录是编译时指定的目录，/opt/app/local 是新的目录，用 p 参数指定新目录即可。同理，在刷新配置的时候也需要显示的指定 Nginx 的主目录：

```
1 #重新加载配置（正式环境 Nginx Reload）
2 /home/app/local/tengine/sbin/nginx -p /home/app/
  local/tengine -s reload
3 ./nginx -p /home/app/local/tengine -V
```

error while loading shared libraries: libpcre.so.3: cannot open shared object file

找不到 libpcre.so.3 文件，在本机搜索了之后，发现文件在目录下：

```
1 /lib/i386-linux-gnu/libpcre.so.3
2 /lib/x86_64-linux-gnu/libpcre.so.3
```

error while loading shared libraries: libssl.so.1.0.0: cannot open shared object file

有可能是由于 openssl 库的位置不对导致的，可以尝试创建软链接解决：

```
1 ln -s /usr/local/lib64/libssl.so.1.1 /usr/lib64/
    libssl.so.1.1
2 ln -s /usr/local/lib64/libcrypto.so.1.1 /usr/lib64/
    libcrypto.so.1.1
```

.so 为共享库，是 shared object，用于动态连接的，和 dll 差不多。openssl：多用途的命令行工具，各功能分别使用子命令实现，libcrypto：公共加密库（存放了各种加密算法），libssl：ssl 协议的实现。在 Ubuntu 下可以通过如下命令安装 libssl.so.1.0.0：

```
1 sudo apt-get install libssl1.0.0:amd64
2 sudo apt-get install libssl1.0.0 libssl-dev
3
4 #查看 OpenSSL 版本
5 openssl version
6
7 #查看系统中对 libssl
8 find / -name libssl.s0*
9
10 # 列出 openssl 包
11 yum list \*openssl\*
```

不过此处不能通过命令安装，只能通过源码编译安装。这里有包：<https://pkgsrc.org/download/libssl1.0.0>。查看当前操作系统支持的 openssl：

```
1 yum --showduplicates list openssl
```

寻找包含 libssl.so.1.0.0 的安装包：

```
1 yum provides */libssl.so.1.0.0
```

在 <http://rpm.pbone.net/> 上搜索 openssl-1.0.0, 搜到 openssl-1.0.0-4.fc24.x86_64.rpm。
<http://rpmfind.net/linux/rpm2html/search.php?query=openssl&submit=Search+...&system=CentOS&arch=>

nginx: (emerg) unknown directive "upstram"

粘贴复制时有特殊字符。

12.4 Ansible

12.4.1 Ansible 模块

Ansible 在 Mac 下的安装路径 `/usr/local/Cellar/ansible/2.3.1.0`。在 Mac 下没有默认创建 Ansible 配置文件，手动创建即可。

Ansible 配置

Ansible 主机配置：

```
1 [pi]
2 192.168.31.25 ansible_ssh_port=22 ansible_ssh_user=pi
```

shell 模块

使用 shell 模块，在远程命令通过 `/bin/sh` 来执行；所以，我们在终端输入的各种命令方式，都可以使用；但是我们自己定义在 `.bashrc/.bash_profile` 中的环境变量 shell 模块由于没有加载，所以无法识别；如果需要使用自定义的环境变量，就需要在最开始，执行加载自定义脚本的语句；对 shell 模块的使用可以分成两块：如果待执行的语句少，可以直接写在一句话中：

```
1 ansible myservers -a ". .bash_profile;ps -fe |grep
   sa_q" -m shell
```

如果在远程待执行的语句比较多，可写成一个脚本，通过 copy 模块传到远端，然后再执行；但这样就又涉及到两次 ansible 调用；对于这种需求，ansible 已经为我们考虑到了，script 模块就是干这事的；

copy 模块

copy 模块是将本机中的文件复制到远程主机当中。如果在复制文件过程中需要使用变量，可以使用 template 模块。

```
1 ansible shuitu-front-webservers -m copy -a "src=./  
    dist.tar.gz dest=/root/app-soft/"
```

12.4.2 Ansible 代理 (Ansible Proxy)

发布系统时,有时无法直接连接目标机器,通过设置 Ansible 代理地址来发布,貌似也没有许多特别的地方:

```
1 [proxy-ic-webservers]  
2 #通过跳板机器访问测试环境  
3 10.55.10.77 ansible_ssh_port=2222 ansible_ssh_user=  
    root
```

其中 10.55.10.77 是配置的代理机器,代理机器通过 ssh 端口转发 (SSH Port Forwarding) 来实现代理,2222 为代理机器的监听端口。ssh 代理转发将代理机器 2222 端口的流量转发到目标机器的 ssh 端口上。

12.5 ssh(Secure Shell)

12.5.1 免密登陆

免密登录需要注意的是, .ssh 文件夹下的 authorize_key 文件的权限需要是 600。

```
1 ssh -p 22 pi@192.168.31.25 'mkdir -p .ssh && cat >> .  
    ssh/authorized_keys' < ~/.ssh/id_rsa.pub  
2 # 如果本地没有生成过 ssh key  
3 # 使用如下命令生成 ssh key  
4 ssh-keygen -t rsa -C "a@gmail.com"
```

在 Mac OS X 中,有时自动登陆需要反复输入密码,解决问题的方法是可以配置 Serria 记住密码:

```
1 usekeychain yes
```

12.5.2 ssh 连接慢

在使用 SSH 时,每次连接建立都相当的慢啊,要 20 秒以上才能够登录上去,严重影响工作效率。在服务端的/etc/ssh/sshd_config 文件中,修改配置,将 GSSAPI-

Authentication 默认设置为关闭即可，配置文件修改后需要重新启动 sshd 守护进程配置才能够生效：

```
1 GSSAPIAuthentication no
2 # 重启 sshd 守护进程
3 sudo service sshd restart
```

GSSAPI (Generic Security Services Application Programming Interface) 是一套类似 Kerberos 5 的通用网络安全系统接口。该接口是对各种不同的客户端服务器安全机制的封装，以消除安全接口的不同，降低编程难度。但该接口在目标机器无域名解析时会有问题。使用 strace 查看后发现，ssh 在验证完 key 之后，进行 authentication gssapi-with-mic，此时先去连接 DNS 服务器，在这之后会进行其他操作。

12.5.3 Permission denied (publickey)

在 ssh 登陆机器时，提示 Permission denied (publickey)。可以尝试的方法，在连接命令中加上 -vvv 参数，观察详细的调试输出：

```
1 ssh -p 2222 -vvv hldev@10.0.0.22
```

设置文件夹对应的权限：

```
1 sudo chmod 700 .ssh
2 sudo chmod 600 .ssh/authorized_keys
```

登陆服务器观察相应的日志输出：

```
1 tail -f /var/log/auth.log
```

是不是服务器关闭了密码登陆？只能使用公钥认证登陆。后面检查确实如此，服务器端为了安全考虑，关闭了基于密码登录的方式，但是需要登录的主机并没有将自己的公钥拷贝到服务器上。所以服务器直接提示了 Permission denied (publickey)，解决的办法就是在服务器端暂时开启密码登录，在 /etc/ssh/sshd_config 中调整配置：

```
1 #允许使用基于密钥认证的方式登陆
2 PubkeyAuthentication yes
```


12.5.4 Session 时间

在使用 ssh 的过程中，经常会遇到一会儿没有操作就自动断开了，不是非常方便。

ClientAliveInterval

修改/etc/ssh/sshd_config 配置文件 ClientAliveInterval 300（默认为 0），参数的意思是每 5 分钟，服务器向客户端发一个消息，用于保持连接，使用 service sshd reload 让其修改后生效。如果发现还是有问题，可以试着把 300 设置小一点，例如 60。

ClientAliveCountMax

另外，至于 ClientAliveCountMax，使用默认值 3 即可。ClientAliveCountMax 表示服务器发出请求后客户端没有响应的次数达到一定值，就自动断开。

ControlPersist 4h

在./ssh/config 中添加一行：

```
1 ControlPersist 4h
```

When used in conjunction with ControlMaster, specifies that the master connection should remain open in the background (waiting for future client connections) after the initial client connection has been closed. If set to no, then the master connection will not be placed into the background, and will close as soon as the initial client connection is closed. If set to yes or 0, then the master connection will remain in the background indefinitely (until killed or closed via a mechanism such as the “ssh -O exit”). If set to a time in seconds, or a time in any of the formats documented in sshd_config, then the backgrounded master connection will automatically terminate after it has remained idle (with no client connections) for the specified time². 现在你每次通过 SSH 与服务器建立连接之后，这条连接将被保持 4 个小时，即使在你退出服务器之后，这条连接依然可以重用，因此，在你下一次（4 小时之内）登录服务器时，你会发现连接以闪电般的速度建立完成，这个选项对于通过 scp 拷贝多个文件提速尤其明显，因为你不在需要为每个文件做单独的认证了。

²http://man.openbsd.org/sshd_config.5

12.5.5 代理转发

本地转发 Local Forward

将本地机(客户机)的某个端口转发到远端指定机器的指定端口。工作原理是这样的,本地机器上分配了一个 socket 侦听 port 端口,一旦这个端口上有了连接,该连接就经过安全通道转发出去,同时远程主机和 host 的 hostport 端口建立连接。可以在配置文件中指定端口的转发。只有 root 才能转发特权端口。应用实例可以参看20.1.3。

```
1 # Mac OS X 端口转发
2 /usr/bin/ssh -g -L 2222:10.10.30.1:22222 127.0.0.1
3 # 接口服务器设置代理转发
4 ssh -g -L 7805:192.168.250.100:7805 10.10.1.32
```

有时在本地转发会遇到一些问题,比如 Connection Refused。首先要确定本地要运行有 ssh 服务端,使用如下命令启动 sshd:

```
1 # Mac OS X 启动 sshd
2 sudo /usr/bin/sshd
3 # Ubuntu 启动 sshd
4 sudo /etc/init.d/ssh
```

启动 SSHD 的时候系统提示:Could not load host key: /etc/ssh/ssh_ed25519_key。新版的 opensshd 中添加了 Ed25519 做签名验证,而之前系统里没这个算法的证书,所以办法也很简单新生成下证书即可。

```
1 sudo ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key
2 sudo ssh-keygen -t dsa -f /etc/ssh/ssh_host_dsa_key
3 sudo ssh-keygen -t ecdsa -f /etc/ssh/
    ssh_host_ecdsa_key
4 ssh-keygen -t ed25519 -f /etc/ssh/
    ssh_host_ED25519_key
```

远程转发 Remote Forward

12.5.6 ssh 查看日志

如果需要查看 ssh 日志,可在/etc/ssh/sshd_config 配置日志输出:

```
1 SysLogFacility LOCAL7
```

Facility: 设施, 是 rsyslog 引入的概念, 从功能或者程序上对日志进行分类, 并由专门的工作负责记录相对应的信息, 分类有: auth(授权), authpriv, cron(定时任务), daemon(守护进程), kern(内核), lpr, mail(邮件相关), mark, news, security(安全), syslog, user, uucp, local0, local1, local2, local3, local4, local5, local6, local7; mysql, postgresql, oracle 等多种关系数据中, 强大的过滤器, 可实现过滤系统信息中的任意部分。自定义输出格式。适用于企业级别日志记录需求。ssh 配置文件配置后, 在 /etc/rsyslog.conf 文件中作如下配置:

```
1 LOCAL7.*    /var/log/sshd.log
```

12.6 VisualVM

使用 nmap 扫描 1099 端口, 看 jstatd 是否生效。

12.6.1 VisualVM 远程监控

jstatd 是一个监控 JVM 从创建到销毁过程中资源占用情况并提供远程监控接口的 RMI (Remote Method Invocation, 远程方法调用) 服务器程序, 它是一个 Daemon 程序, 要保证远程监控软件连接到本地的话需要 jstatd 始终保持运行。jstatd 运行需要通过 -J-Djava.security.policy=*** 指定安全策略, 因此我们需要在服务器的 JDK 根目录 (如: /opt/app/local/jdk1.8.0_111) 上建立一个指定安全策略的文件 jstatd.all.policy, 文件内容如下:

```
1 grant codebase "file:${java.home}/../lib/tools.jar" {
2     permission java.security.AllPermission;
3 };
```

建立好安全策略文件后, 在服务端运行 jstatd 守护程序, jstatd 在 JDK 目录下:

```
1 ./bin/jstatd -J-Djava.security.policy=jstatd.all.
    policy -J-Djava.rmi.server.hostname
    =10.10.1.12 -J-Djava.rmi.server.logCalls=true
    -p 1011
2 #在树莓派上启动 jstatd
3 sudo /usr/bin/jstatd -J-Djava.security.policy=/opt/
    dolphin/backend/jstatd.all.policy -J-Djava.
    rmi.server.hostname=192.168.31.25 -J-Djava.
    rmi.server.logCalls=true -p 1011
```

使用 VisualVM 监控远程主机上 JAVA 应用程序时，需要开启远程主机上的远程监控访问，或者在远程 JAVA 应用程序启动时，开启远程监控选项，两种方法，选择其中一种就可以开启远程监控功能，配置完成后就可以在本地对远程主机上的 JAVA 应用程序进行监控。注意下载地址³需要手动修改一下，原装的地址已经失效。VisualVM 中做相应配置即可。连接后如图12.8所示。

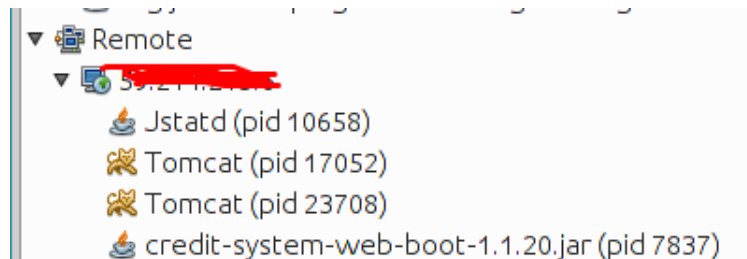


Figure 12.8: VisualVM 远程监控

目前可以通过 VisualVM 远程监控实现查看远程应用的 PID、查看远程应用的启动参数、所使用的配置文件、JVM 的版本以及路径、GC 信息等等。Java 的对象分为年轻代 (Young Generation) 和老年代 (Old Generation)。年轻代又分为 Eden(伊甸园) 区和 Survivor Space(幸存者)，老年代是 Tenured。当 JVM 无法在 Eden 区创建新对象时，则会出现 Out of memory 错误。使用 JVM 监控系统如图12.9所示：

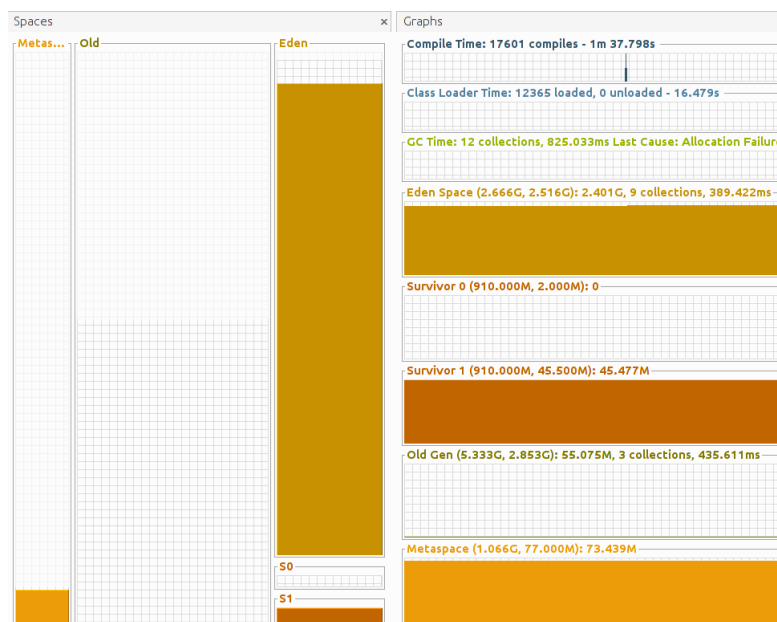


Figure 12.9: VisualVM 远程监控 GC

³<https://visualvm.github.io/uc/release139/updates.xml.gz>

12.6.2 autojump

autojump 可以让你一键跳转到目标目录，一般情况下，很难准确的记忆清楚目标目录的准确路径，特别是目录嵌套比较深的时候。不停的重复使用 `cd` 命令一级或多级跳转显得有点笨拙。此时就可以使用 `autojump` 命令，只需要输入路径的关键字即可，比如只记得路径中隐约包含有关键字 “jiangxiaoqiang”，那么仅仅输入命令 `autojump jiang`，就可以自动跳转到对应的路径下。安装 `autojump`：

```
1 sudo apt-get install autojump -y
2 # 给 autojump 命令设置别名
3 alias j='autojump'
4 # 从源代码安装 autojump
5 git clone git://github.com/joelthelion/autojump.git
6 chmod 755 install.py
7 ./install.py
8
9 # 将当前路径添加到 autojump 的搜索库
10 autojump -a 'pwd'
```

为了进一步简化目录跳转命令，可以直接添加命令 `autojump` 的别名：

```
1 alias j='autojump'
```

有时会出现如下的提示：Please source the correct autojump file in your shell's startup file. For more information, please reinstall autojump and read the post installation instructions. 为了暂时激活 `autojump` 应用，即直到你关闭当前会话或打开一个新的会话之前让 `autojump` 均有效，需要以常规用户身份运行下面的命令：

```
1 # 在 Linux 系统下
2 source /usr/share/autojump/autojump.sh on startup
3 # 在 Mac OS X 下
4 source ~/.autojump/etc/profile.d/autojump.sh
5 # 也可运行如下命令
6 [[ -s /Users/dolphin/.autojump/etc/profile.d/autojump
   .sh ]] && source /Users/dolphin/.autojump/
   etc/profile.d/autojump.sh
```

为了使得 `autojump` 在 `BASH` shell 中永久有效，需要运行下面的命令。

```
1 echo '. /usr/share/autojump/autojump.sh'>> ~/.bashrc
2 # 在 Mac OS X 下
3 echo '. ~/.autojump/etc/profile.d/autojump.sh'>> ~/.
    bashrc
```

12.7 Graphviz

Graphviz（英文：Graph Visualization Software 的缩写）是一个由 AT&T 实验室启动的开源工具包，用于绘制 DOT 语言脚本描述的图形。它也提供了供其它软件使用的库。Graphviz 是一个自由软件，其授权为 Eclipse Public License。

12.7.1 编译

一个简单的图形示例：

```
1 digraph G {
2     node [peripheries=2 style=filled color="#eccc80"]
3     edge [color="sienna" fontcolor="green"]
4     "componentWillMount" -> "serviceComponent";
5     "serviceComponent" -> "axios";
6     "axios" -> "dishatch data to store"[label="XHR(
        GET/POST/PUT/DELETE)"] ;
7     "dishatch data to store" -> "props get data";
8 }
```

编译图形：

```
1 dot -Tjpg -Gdpi=1024 maven-lifecycle.dot -o maven-
    lifecycle.jpg
```

12.7.2 subgraph

子图的名称必须以 cluster 开头，否则 graphviz 无法识别。

```
1 digraph G {
2     node [peripheries=2 style=filled color="#eccc80"
        ];
```

```
3 //edge [color="sienna" fontcolor="green"];
4 //node [shape="record"];
5 edge [style="dashed"];
6
7
8 "" [shape="none",image="chrome-512.png",color=""];
9
10 "" -> "关键字";
11 "关键字" -> "查询视图";
12
13 subgraph cluster1{
14     label = "展示层";
15     fillcolor = darkslategray;
16     "平台用户" -> "浏览器";
17 }
18
19 }
```

12.8 MyBatis Generator

13. Git

13.1 基础

13.1.1 常用 Git 命令

常用语句。

```
1 git config --global user.name "dolphin"
2 git config --global user.email "jiangtingqiang@gmail.
  com"
3 #增加远程库
4 git remote add company-internal http://xiaoqiang.
  jiang@gitlab.data.com/backend/system.git
5 #删除远程库
6 git remote rm company-
7 #修改远程库地址
8 git remote set-url origin http://gitlab.hualongdata.
  com/internal/work-record.git
9 #列出 git 配置
10 git config --list
11 #查看远程 (remote) 分支
12 git branch -r
13 # 比较 2 个分支文件的差异
```

```
14 git diff v1.3_xiaoqiang v1_xiaoqiang filename.java
```

克隆指定的分支：

```
1 git clone -b v1.3 <url>
```

13.1.2 返回到指定版本

查看提交历史：

```
1 git log --all --graph -12
```

回滚到指定版本：

```
1 git reset --hard  
e4177ea9061a9726efd42d289f73553409df793d
```

结果如图13.1所示，即可看到由谁修改。



```
commit e78c8f6561060799a3db1aeab3cd00b8d30b9820  
Author: [REDACTED]  
Date: Wed Aug 23 11:28:40 2017 +0800  
  
    修改上传大小的最大值  
  
diff --git a/application-jjxxzx-test.properties b/application-jjxxzx-test.properties  
index 710da92..b74a5d0 100644  
--- a/application-jjxxzx-test.properties  
+++ b/application-jjxxzx-test.properties  
@@ -6,7 +6,7 @@ spring.datasource.type=com.zaxxer.hikari.HikariDataSource  
 #  
 # 部署到生产环境时需要修改链接地址  
 #  
-spring.datasource.jdbc-url=jdbc:dm://dn4:5236/DMSERVER  
+spring.datasource.jdbc-url=jdbc:dm://10.10.10.1:5236/DMSERVER
```

Figure 13.1: Git 查看文件修改历史

13.1.3 标签 (Tag)

有的时候需要修复线上某个版本的问题，但是当前主分支已经合并了中途开发的部分功能，还未经过严格测试。为了满足这样的要求，需要在每次发布版本的时候打上标记 (Tag)，需要修复线上问题时，可以回到标记的版本，修复线上的问题后，将 fix 分支合并到主干。

```
1 # 发布时打上 Tag，带有信息
```

```
2 git tag -a v1.1.31 -m "v1.0版本发布"
3 git tag -a v1.1.35 -m "v1.1.35-Release版本"
4 # 查看有哪些 Tag
5 git tag
6 # 查看具体 Tag 的内容
7 git show v1.1.31
8 # 将代码回退到发布时的版本
9 # 实际 Commit ID 为
10 # eb8f7df1935da2e191163610d8b0374beff011e0
11 # 只需要前几位即可
12 git reset --hard eb8f7d
13 # 回退版本后，拉取新的分支，这里取名 HotFix1.1.31 分支
14 git checkout -b HotFix1.1.31
```

需要注意的是，创建好 HotFix 分支之后，要马上回到主分支将代码前进到最新版本：

```
1 # 切换到主分支
2 git checkout master
3 # 使用 reflog 命名检索最新版本的 commit id
4 git reflog
5 # 前进到最新版本
6 git reset --hard 9e4c92
```

剩下的工作就是切换到刚才到 HotFix1.1.31 分支修复问题即可。修复之后，将问题修复后的代码合并到主干分支。

13.1.4 查看修改历史

在项目中看到有几行代码，百思不得其解，也没有注释，所以想看一看作者，与作者沟通，使用如下命令即可：

```
1 git blame PubapiOrgController.java
```

会显示每一行代码的作者(最后修改者)和最后的修改时间。有的时候，不仅需要看某(几)行代码最后的修改，也希望看到最近几个版本的修改。比如有一次部署上去后，发现数据库链接被改掉了，那么可以使用如下命令查看文件的修改历史：

```
1 git log --follow application-jjxxzx-test.properties
```

会显示出指定文件的修改历史记录，使用如下命令查看详细的修改内容：

```
1 git show e78c8f6561060799a3db1aeab3cd00b8d30b9820
```

查看文件删除记录：

13.1.5 merge

配置 Meld 为默认合并工具：

```
1 git config --global merge.tool meld
```

合并指定分支

项目开发有 v1 分支和 v1.3 分支，v1 分支是旧版系统的分支，针对旧版系统的某些改造特性想在新版中集成，而其他的内容和问题修复不一定会集成到新版系统中。此时为了避免重复修改，可以使用 git 的 cherry pick 将指定的 commit 合并到新版分支中。使用如下命令：

```
1 git cherry-pick <commit id>
2 #合并指定分支到当前分支
3 git merge --no-ff v1_xiaoqiang
```

和并指定文件

在开发过程中，有一个旧的分支 v1_xiaoqiang 和一个新的分支，在新旧分支同步开发的过程中，旧分支添加了一些特性，需要和并指定特性到分支上。也就是需要将旧分支的某些修改和并到新的分支，可以直接使用 merge 命令，但是所有的内容都会和并到新分支，也可以使用 git diff 命令对比两个分支。

```
1 git diff branch1 branch2
```

但是采用 git diff 命令比较头疼的是完全是文本行，不是非常直观，可以直接使用 merge 工具 meld 来和并：

```
1 git difftool -t meld -y v1.3_xiaoqiang v1_xiaoqiang
```

使用 Meld 工具对比效果如下图所示。

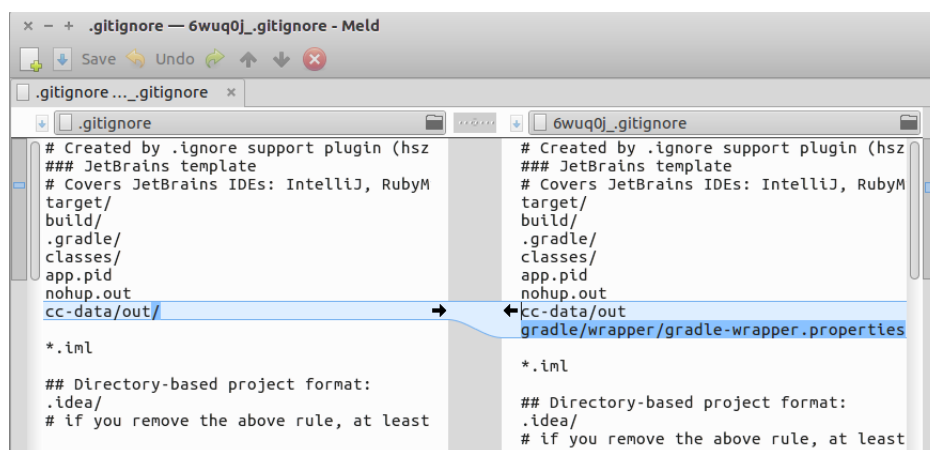


Figure 13.2: Git 使用 Meld 工具对比和并不同分支文件

图中左侧表示 v1.3_xiaoqiang 分支的文件，右侧表示 v1_xiaoqiang 分支的文件。如果不想和并文件，直接关闭 Meld 窗体即可，如果需要和并，点击对应的箭头。

13.1.6 Git Hook

项目的 v1 分支可以合并到 v1.3 分支，而 v1.3 分支不能够合并到 v1 分支。为了保护有时分支失误合并，采用钩子的方式来解决，每次推送的时候验证：

13.1.7 常见问题

error setting certificate verify location

当你通过 HTTPS 访问 Git 远程仓库，如果服务器的 SSL 证书未经过第三方机构签署，那么 Git 就会报错。这是十分合理的设计，毕竟未知的没有签署过的证书意味着很大安全风险。但是，如果你正好在架设 Git 服务器，而正式的 SSL 证书没有签发下来，你为了赶时间生成了自签署的临时证书。

```
env GIT_SSL_NO_VERIFY=true git clone https://<
    host_name/git/project.git
```

或者将 Git 设置为不允许 ssl 认证：

```
git config --system http.sslverify false
```


14. Google Chrome

14.1 DevTools

快捷键	备注
Ctrl+Shift+J (Windows/Linux)	打开 DevTools Console 面板
Ctrl+Shift+I (Windows/Linux)	打开 DevTools Network 面板, I 是 Inspect 的缩写
Ctrl+Shift+P (Windows/Linux)	打开所有窗口

14.1.1 Console

Log XMLHttpRequests

想要看到使用中应用发送的 XHR 请求, 可以打开 settings 面板 (译注: 打开调试面板按 F1 呼出) 选中 “Log XMLHttpRequests” 选项, 然后就可以在 “Console” 面板中看到请求了。点击此请求可以导航到 Network 选项卡上, 当然也可以直接在 Network 选项卡中查看 XHR HttpRequest 请求, 不过使用 Console 打印出来到导航还是要方便一些, 多了一个入口。

脚本添加到黑盒

当你使用 “Event listener breakpoint :: Mouse :: Click” 调试事件时, 有很大的可能是代码在第三方的库中中断了 (例如 jquery), 这时如果你想找到属于自己项目的调用需要在调试器里面点很多次的下一步才能看到。一种很棒的方式就是把这些第三方的脚本放到黑盒里面, 调试器永远不会在黑盒中的脚本内停止, 它会一直向后运行, 直到运行的代码行位于黑盒之外的文件。你可以在调用栈面板中

右键点击第三方脚本的文件名，然后左键在下拉菜单中点击 “Blackbox Script”。

定位按钮触发代码

快速定位某个特定按钮或者链接点击后所触发的代码，只需要在你真正点击按钮之前，激活一个 “Event listener breakpoint”，选中右侧面板下 mouse 下面的 click 确认框即可。

Async 调试

在 Promise 被广泛应用的今天，我们都知道，Promise 的回调是异步执行的，没有开启 Async 模式前，调用栈只记录到回调函数本身，我们无法找到代码执行的顺序，这给我们调试带来巨大的困难。Async 模式可以解决这个问题。

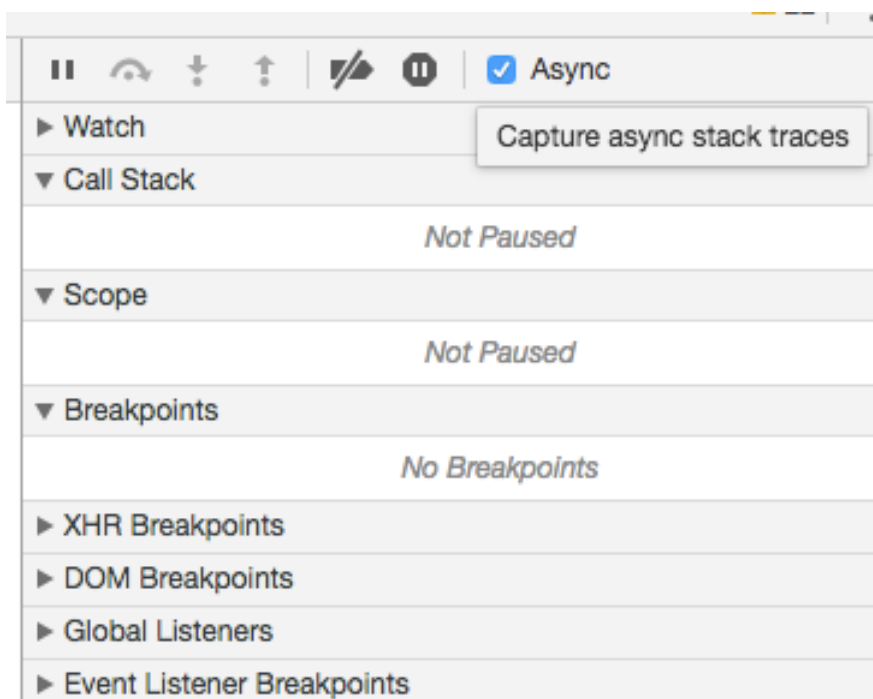


Figure 14.1: Google Chrome 异步调试

查找成员函数

Ctrl+Shif+O 文件中定位成员函数。

打印对象数组

在 Javascript 中可能返回有数组，有的时候可能某一个数组元素感觉数据不对，在 Javascript 调试跟踪时，输出了一些 Object 对象，如果是几个元素，还可以一个一个点开查看，如果有成百上千的元素，一个一个点开查看几乎是不可能的，至今没有发现在结果中按关键字搜索的功能，此时可以使用如下方法：


```
1 console.table(data);
```

即可以表格的形式显示所有元素,其中 data 表示一个对象或者数组,如图14.2所示。

```
12:37:15.662 console.table(data);
12:41:09.525
```

(index)	name	value	dfbm
0	"市级部门"	0	"500000"
1	"万州区"	14296	"500101"
2	"涪陵区"	10788	"500102"
3	"渝中区"	30481	"500103"
4	"大渡口区"	6853	"500104"
5	"江北区"	18064	"500105"
6	"沙坪坝区"	22700	"500106"
7	"九龙坡区"	46013	"500107"
8	"南岸区"	40452	"500108"
9	"北碚区"	13938	"500109"
10	"綦江区"	22234	"500110"
11	"大足区"	8913	"500111"
12	"渝北区"	32595	"500112"
13	"巴南区"	20810	"500113"
14	"黔江区"	7957	"500114"
15	"长寿区"	9247	"500115"
16	"江津区"	14723	"500116"
17	"合川区"	17022	"500117"
18	"永川区"	12604	"500118"
19	"南川区"	9071	"500119"
20	"璧山区"	16957	"500120"

Figure 14.2: Google Chrome 查看对象数组

14.1.2 Source Map

截止目前几乎没有浏览器完全原生支持 es6 标准,对于这种情况,Chrome 引入了 source-map 文件,标识 es5 代码对应的转码前的 es6 代码哪一行,唯一要做的就是配置 webpack 自动生成 source-map 文件,这也很简单,在 webpack.config.js 中增加一行配置即可(需要重新启动 webpack-dev-server 使配置生效)

14.1.3 Network

Replay XHR

有时在开发时,可以直接在 Network 中的网络请求列表中点击右键,在菜单中重复发送 XHR 请求 (Replay XHR),不需要再次进入界面模拟请求,可以提高不少效率。

Disable cache

勾选上 Disable cache 后,浏览器不会利用本地的 Javascript 缓存,每次都会到服务器上重新取。

Preserve Log

保留请求日志，勾选后，即使在当前页面刷新后，日志也不会清空。在使用 large request row 后，size 和 time 会出现 2 行内容，如图 14.3 所示。其中时间行，上面的时间代表从请求开始到接收完最后一个 byte 为止的时间，下面一行是 Latency，代表从请求接收完后到读取该资源第一个 byte 之间的等待时间。按照这个逻辑，那么上面一行的时间减去下面一行的时间就是数据的下载时间，将鼠标移动到右侧 waterfall 图标上，可以看到下载时间 (Content Download) 正好是上下两行的时间差。另外 Size 列也有 2 行，上面一行代表网络上实际传输的文件大小，下面一行代表原始的未经过压缩的文件大小。一般服务器会对资源使用 Gzip 进行压缩。






Filter	<input type="checkbox"/> Regex	<input type="checkbox"/> Hide data URLs	All	XHR	JS	CSS	Img	Media
Name	Status	Type	Initiator	Size	Time			
 globalInfo /pubapi/global	200 OK	xhr	xhr.js:175 Script	179 KB 1.4 MB	517 ms 89 ms			
 page?parentCate... /pubapi/article	200 OK	xhr	xhr.js:175 Script	87.9 KB 336 KB	49 ms 21 ms			
 list /pubapi/creditCo...	200 OK	xhr	xhr.js:175 Script	18.1 KB 169 KB	44 ms 28 ms			
 page?parentCate... /pubapi/article	200 OK	xhr	xhr.js:175 Script	20.9 KB 60.5 KB	42 ms 29 ms			
 page?parentCate... /pubapi/article	200 OK	xhr	xhr.js:175 Script	14.7 KB 51.4 KB	28 ms 22 ms			

Figure 14.3: Network 宽行显示

DOMContentLoaded 和 load 事件信息

DOMContentLoaded 和 load 这两个事件会高亮显示。DOMContentLoaded 事件会在页面上 DOM 完全加载并解析完毕之后触发，不会等待 CSS、图片、子框架加载完成。load 事件会在页面上所有 DOM、CSS、JS、图片完全加载完毕之后触发。DOMContentLoaded 事件在 Overview 上用一条蓝色竖线标记，并且在 Summary 以蓝色文字显示确切的时间。load 事件同样会在 Overview 和 Requests Table 上用一条红色竖线标记，在 Summary 也会以红色文字显示确切的时间。

资源的发起者 (请求源) 和依赖项

通过按住 Shift 并且把光标移到资源名称上，可以查看该资源是由哪个对象或进程发起的 (请求源) 和对该资源的请求过程中引发了哪些资源 (依赖资源)。在该资源的上方第一个标记为绿色的资源就是该资源的发起者 (请求源)，有可能会有第二个标记为绿色的资源是该资源的发起者的发起者，以此类推。在该资源的下方标记为红色的资源是该资源的依赖资源，如图 14.4 所示。















	main	304	Not Modified	document	webpack:///./src/entrie
	gray.png	200	OK	png	InkTabBarMixin.js:47
	/assets/img/indexImage	200	OK	png	Script
	icon.png	200	OK	png	InkTabBarMixin.js:47
	/assets/img/indexImage	200	OK	png	Script
	head.png	200	OK	png	InkTabBarMixin.js:47
	/assets/img/indexImage	200	OK	png	Script
	title.png	200	OK	png	Other
	/assets/img/indexImage	200	OK	font	main
	Fontawesome-webfont.woff2?v=...	200	OK	font	Parser
	/assets/vendor/font-awesome/fo...	200	OK	script	main
	echarts.min.js	200	OK	script	Parser
	/assets/vendor/echarts	200	OK	script	main
	dataSource.js	200	OK	script	main
26 requests 321 KB transferred Finish: 5.33 s DOMContentLoaded: 2.55 s Load: 2.55 s					

Figure 14.4: 依赖资源

14.1.4 Elements

在元素上，按住 Ctrl + Alt + 鼠标左键，可以展开元素及下级元素的所有节点。

14.1.5 Performance

14.1.6 Sources

面板中查看变量

在调试的过程中，虽然鼠标可以移动到变量上会自动 Pop 出对应的变量，但是有时候鼠标移出区域后变量也会自动消失。那么此时可以在右侧面板中的 Scope 查看变量，如图14.5所示：

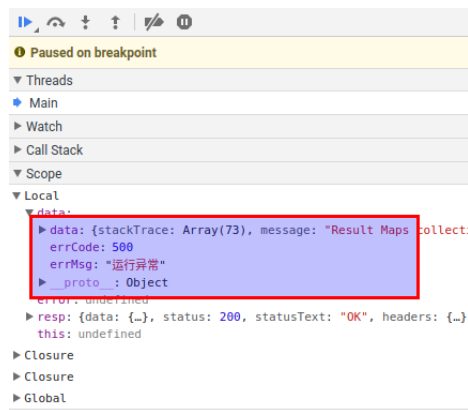


Figure 14.5: Devtool Scope 中查看变量

查看 Render 区域 有时需要查看浏览器重绘制的区域,页面的绘制时间(paint time)是每一个前端开发都需要关注的的重要指标，它决定了你的页面流畅程度。使用 Ctrl + Shift + P，输入 Render 打开对应页面即可。而如何去观察页面的绘制

时间，找到性能瓶颈，可以借助 Chrome 的开发者工具，激活页面绘制区域显示如图14.6所示。

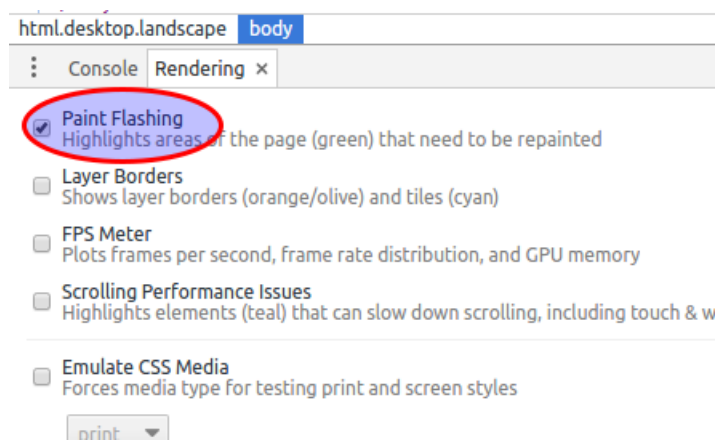


Figure 14.6: 激活页面绘制区域高亮

14.1.7 Audits

14.1.8 Google Chrome SSH

15. LaTeX

15.1 安装

输入如下命令安装 TexLive:

```
1 # Ubuntu 下安装 Texlive
2 sudo apt-get install texlive -y
```

15.1.1 字体

Computer Modern 是自由软件 TeX 的默认字体，为美国计算机科学家高德纳 (Donald Knuth) 使用 METAFONT 软件创造。但是此字体不是很漂亮，所以考虑换一个字体。

lstlisting 字体

在使用 lstlisting 代码块时，希望使用等宽字体 (Monospaced Font)，代码使用等宽字体更加美观。lstlisting 设置如下：

```
1 \lstset{
2   % 设定代码的基本字体为等宽字体（类似于 Courier）
3   basicstyle=\ttfamily,
4   % 设定注释部分字体为等宽字体，用深绿色显示
5   commentstyle=\ttfamily\color{green!40!black},
```

```
6 }
```

15.1.2 版面

PDF 导航深度

LaTeX 生成的 pdf 导航默认深度是 3，如果需要显示更深层次。那么可以使用 hyperref 包，如下代码片段。

```
1 \usepackage[bookmarksopen,bookmarksdepth=3]{hyperref}
```

奇怪的是，这种方式在 Mac OS X 下有效，但是在 Ubuntu 16.04 LTS 下无效。

换行

Latex 的自动拆单词换行是依靠字典的。如果遇到 LaTeX 不认识的单词，他就不会换行了。后面的内容就会溢出。

15.1.3 文献引用

BibTeX 是一种方式，它本身不需要加载任何的包 (package)，但编译的时候需要使用 bibtex.exe。由于它不太灵活，又出现了新的方式—biblatex 方式。biblatex 包是一个更加灵活的文献处理方式，它不仅支持更多的 entry type，而且支持多次加入 bib 文件，支持多种不同的 bib 内容书写格式，也支持从远程加入 bib 文件，支持在文档的任何位置显示参考文献的内容。比如，你可以在论文的每一章后面添加参考文献的显示。从发展的眼光来看，biblatex 是一个比 bibtex 更加先进的技术，在未来的某个阶段肯定会取代 bibtex。biblatex 需与 biber 命令配合使用。首先在该文件的导言区加上下面指令：

```
1 \usepackage[hyperref=true,backend=biber,sorting=none,  
    backref=true]{biblatex}  
2 \addbibresource{Bronvermelding.bib}
```

backend=biber 表示后端处理的程序为 biber.exe；hyperref=true 和 backref=true 表示为各个参考文献的引用处、及定理、定义、例子等的引用处都添加上超链接；sorting=none 表示按照参考文献在论文中出现的先后顺序排序。在 enddocument 的前面添加指令：

```
1 \printbibliography
```

15.1.4 支持语言

Latex 原生不支持 JavaScript 语言，添加如下定义：

```
1 %\lstlisting 包 JavaScript 语言设置
2 \lstdefinlanguage{JavaScript}{
3     keywords={typeof, new, true, false, catch,
4         function, return, null, catch, switch, var,
5         if, in, while, do, else, case, break},
6     keywordstyle=\color{blue}\bfseries,
7     ndkeywords={class, export, boolean, throw,
8         implements, import, this},
9     ndkeywordstyle=\color{darkgray}\bfseries,
10    identifierstyle=\color{black},
11    sensitive=false,
12    comment=[1]{//},
13    morecomment=[s]{/*}{*/},
14    commentstyle=\color{purple}\ttfamily,
15    stringstyle=\color{red}\ttfamily,
16    morestring=[b]',
17    morestring=[b]"
18 }
```

即可在文档中定义 JavaScript，并支持关键字颜色区分。

15.1.5 常见问题

infoBulle.sty not found

一般的，.cls 和.sty 文件是用来提高 LaTeX 的排版效果的补充文件。它们分别用命令\documentclass{...} 和命令\usepackage{...} 来装载到 LaTeX 文件。.cls 文件通常被称为“classes”（也就是“类”），而.sty 文件被称为“style files”（样式文件），或者就简单地称其为“packages”（包）。

下载下来的宏包，有的直接包含.sty 文件（.sty 文件就是 package 的最终形式）；有的下载下来只包含了.ins, .dtx，以及一个 readme 文件，这时需要自己编译生成.sty 文件，以及生成相应的 manual。你可以在这个路径下建立一个文件夹，譬如 qrcode，然后把 qrcode.sty 放到这里，接着在你的电脑里找到 MikTeX 的 settings 这个程序，settings 有两个，选择后面括号里有 admin 的那个，打开以后，在 general 选项卡下有 Refresh FNDB 按钮，点击，过一会，这个 package 就会加入 MikTeX 的

路径中，然后在你的 `tex` 文件中就可以使用这个 `package` 了。

`infoBulle` 可能是作者自己写的一个宏包，系统里默认没有自带，此时就需要找到默认宏包的存放目录，将作者自己写的 `sty` 文件放进去。在 Mac OS X 下，路径为：

```
1 /usr/local/texlive/2016/texmf-dist/tex/latex
```

`sty` 文件拷贝到指定目录后，运行命令：

```
1 sudo texhash
```

即可使用命令引用包。有时会提示缺少 `sty` 文件，遇到一次是因为没有完全安装 `latex`，输入完整包安装命令：

```
1 sudo apt-get install texlive-full -y
```

中文支持

在 Ubuntu 14.04 LTS 下编译无法显示中文，在 \LaTeX 中指定中文字体即可：

```
1 % 设置缺省中文字体
2 \setCJKmainfont{WenQuanYi Micro Hei}
```


IV

DB

16	性能 (Performance)	187
16.1	索引 (Index)	
16.2	Redis	
17	PostgreSQL	193
17.1	基础操作	
18	MySQL	197
19	达梦	203
19.1	SQL	
19.2	通用知识	

16. 性能 (Performance)

16.1 索引 (Index)

16.1.1 索引常识

绝大多数情况下，数据库只使用一个索引。在开发中遇到有时根据时间查询很快：

```
1 --在列 org_id 和 jdrq 上都建立了索引
2 --同样都查询语句，不同都 org_id 值会有不同都查询性能
3 --当 org_id 结果集很少当时时候，数据库会使用 org_id 索引
4 --在使用了 org_id 索引后，不会使用 jdrq 索引
5 --同理，当 org_id 结果集更大，根据 jdrq 索引效率更高时
6 --数据库会使用 jdrq 列作为索引，而不会使用 org_id 索引
7 select *
8 from table
9 where org_id = '1000'
10 order by jdrq desc
```

所以需要同时利用多个列的索引，或者查询条件包含多列时，应该使用联合索引。

```
1 create index idx_ts_b_xzsk_jdrq_org_id on xzsk(jdrq
      desc,org_id);
```

B+ 树

IO 次数取决于 b+ 数的高度 h (height), 假设当前数据表的数据为 N , 每个磁盘块的数据项的数量是 m , 则有

$$h = \log_{(m+1)} N \quad (16.1)$$

当数据量 N 一定的情况下, m 越大, h 越小; 而 m = 磁盘块的大小 / 数据项的大小, 磁盘块的大小也就是一个数据页的大小, 是固定的, 如果数据项占的空间越小, 数据项的数量越多, 树的高度越低。这就是为什么每个数据项, 即索引字段要尽量的小, 比如 `int` 占 4 字节, 要比 `bigint8` 字节少一半。这也是为什么 b+ 树要求把真实的数据放到叶子节点而不是内层节点, 一旦放到内层节点, 磁盘块的数据项会大幅度下降, 导致树增高。当数据项等于 1 时将会退化线性表。

当 b+ 树的数据项是复合的数据结构, 比如 `(name,age,sex)` 的时候, b+ 数是按照从左到右的顺序来建立搜索树的, 比如当 `(张三,20,F)` 这样的数据来检索的时候, b+ 树会优先比较 `name` 来确定下一步的所搜方向, 如果 `name` 相同再依次比较 `age` 和 `sex`, 最后得到检索的数据; 但当 `(20,F)` 这样的没有 `name` 的数据来的时候, b+ 树就不知道下一步该查哪个节点, 因为建立搜索树的时候 `name` 就是第一个比较因子, 必须要先根据 `name` 来搜索才能知道下一步去哪里查询。比如当 `(张三,F)` 这样的数据来检索时, b+ 树可以用 `name` 来指定搜索方向, 但下一个字段 `age` 的缺失, 所以只能把名字等于张三的数据都找到, 然后再匹配性别是 `F` 的数据了, 这个是非常重要的性质, 即索引的最左匹配特性¹。

索引原则 (Index Principle)

最左前缀匹配原则 最左前缀匹配原则, 非常重要的原则, mysql 会一直向右匹配直到遇到范围查询 (`>`、`<`、`between`、`like`) 就停止匹配, 比如 `a = 1 and b = 2 and c > 3 and d = 4` 如果建立 `(a,b,c,d)` 顺序的索引, `d` 是用不到索引的, 如果建立 `(a,b,d,c)` 的索引则都可以用到, `a,b,d` 的顺序可以任意调整。组合索引的第一个字段必须出现在查询组句中, 这个索引才会被用到。举个例子:

```
1 select *
2 from DOLPHIN
3 where DFBM = '500101'
4 order by JDRQ DESC
5 limit 0, 10
```

¹:<http://tech.meituan.com/mysql-index.html>

在建立索引时, DFBM 列一定要排在前面, 建立 (DFBM,JDRQ) 这样的索引 DB 才会利用索引。如果顺序不对 (JDRQ,DFBM), 那么 DB 将不会采用索引。在 mysql 中执行查询时, 只能使用一个索引, 如果我们在 lname,fname,age 上分别建索引, 执行查询时, 只能使用一个索引, mysql 会选择一个最严格 (获得结果集记录数最少) 的索引。

= 和 in 可以乱序 = 和 in 可以乱序, 比如 `a = 1 and b = 2 and c = 3` 建立 (a,b,c) 索引可以任意顺序, mysql 的查询优化器会帮你优化成索引可以识别的形式

尽量选择区分度高的列作为索引 尽量选择区分度高的列作为索引, 区分度的公式是 $\text{count}(\text{distinct col})/\text{count}(*)$, 表示字段不重复的比例, 比例越大我们扫描的记录数越少, 唯一键的区分度是 1, 而一些状态、性别字段可能在大数据面前区分度就是 0, 那可能有人会问, 这个比例有什么经验值吗? 使用场景不同, 这个值也很难确定, 一般需要 join 的字段我们都要求是 0.1 以上, 即平均 1 条扫描 10 条记录

索引列不能参与计算 索引列不能参与计算, 保持列“干净”, 比如 `from_unixtime(create_time)` `= '2014-05-29'` 就不能使用到索引, 原因很简单, b+ 树中存的都是数据表中的字段值, 但进行检索时, 需要把所有元素都应用函数才能比较, 显然成本太大。所以语句应该写成 `create_time = unix_timestamp('2014-05-29')`;

尽量的扩展索引 尽量的扩展索引, 不要新建索引。比如表中已经有 a 的索引, 现在要加 (a,b) 的索引, 那么只需要修改原来的索引即可

16.1.2 索引类型

位图索引 (Bitmap Index) 相对于 B*Tree 索引, 占用的空间非常小, 创建和使用非常快。位图索引由于只存储键值的起止 Rowid 和位图, 占用的空间非常少。B*Tree 索引由于不记录空值, 当基于 is null 的查询时, 会使用全表扫描, 而对位图索引列进行 is null 查询时, 则可以使用索引。当 `select count(XX)` 时, 可以直接访问索引中一个位图就快速得出统计数据。当根据键值做 and,or 或 `in(x,y,...)` 查询时, 直接用索引的位图进行或运算, 快速得出结果行数据。

散列索引 Or 哈希索引 (Hash Index) 散列索引是根据 HASH 算法来构建的索引, 所以检索速度很快, 但不能范围查询。散列索引的特点, 只适合等值查询 (包括 `= <>` 和 `in`), 不适合模糊或范围查询

聚集索引 (Cluster Index) 聚集索引确定表中数据的物理顺序。聚集索引类似于电话簿，后者按姓氏排列数据。由于聚集索引规定数据在表中的物理存储顺序，因此一个表只能包含一个聚集索引。但该索引可以包含多个列（组合索引），就像电话簿按姓氏和名字进行组织一样。

非聚集索引 (Non-Cluster Index) 它并不决定数据在磁盘上的物理排序，索引上只包含被建立索引的数据，以及一个行定位符 row-locator，这个行定位符，可以理解为一个聚集索引物理排序的指针，通过这个指针，可以找到行数据。

全文索引 (Full-text Search) 全文索引是基于要编制索引的文本中的各个标记来生成倒排序、堆积且压缩的索引结构。每个表或索引视图只允许有一个全文索引。该索引最多可包含 1024 列。该对象中必须有一唯一并且非空的列。

联合索引 联合索引能够满足最左侧查询需求，例如 (a, b, c) 三列的联合索引，能够加速 a | (a, b) | (a, b, c) 三组查询需求。这也就是为何不建立 (passwd, loginName) 这样联合索引的原因，业务上几乎没有 passwd 的单条件查询需求，而有很多 loginName 的单条件查询需求。

16.1.3 like 查询优化

目前数据量将近千万，模糊查询效率比较低。模糊查询通常有下面三种，前匹配：like %abc，可以使用 btree 索引优化，后匹配：like abc%，可以使用 reverse 函数 btree 索引，全匹配：like %abc%，可以使用 pg_trgm 的 gin 索引。后面的 2 种优化方式是在 PostgreSQL 数据库中。还有全文索引技术也可以用来优化基于 like 查询的匹配。

采用函数 (无效)

采用函数来优化查询 (没有效果):

```
1  --120W 数据平均 2.3 秒左右，比原生查询稍慢
2  SELECT 'column' FROM 'table' WHERE LOCATE('keyword',
3                                     'field')>0
4
5  select *
6  from "table"
7  where locate('陶然居',mc)>0
8  and code is not null
9  limit 0,10;
```

```
9
10 --120W 数据平均 1.8 秒左右, 与原生查询时间相仿
11 SELECT 'column' FROM 'table' WHERE POSITION('keyword
    ' IN 'filed')
12
13 select *
14 from "table"
15 where position('西南铝业' in mc)
16 and code is not null
17 limit 0,10;
18
19 --120W 数据平均 1.8 秒左右, 与原生查询时间相仿
20 SELECT 'column' FROM 'table' WHERE INSTR('field', '
    keyword' )>0
21
22 select *
23 from "table"
24 where INSTR('西南铝业' in mc)>0
25 and code is not null
26 limit 0,10;
```

全文索引 (Full-text Search)

当使用 like 进行模糊搜索时, 如果是中间匹配, 那么无法使用索引, 需要全表扫描 (Full Table Scan)。优化? 没有办法优化, 所以不要花费时间在通过优化 SQL 语句来提升 like 中间匹配的查询效率。此时需要使用全文索引, 一种方案是使用全文索引查询出结果后, 再使用 like 进行匹配, 在达梦数据库中创建全文索引:

```
1 create context index qymcindex on user.
    TS_F_CORPORATION_TEST(qymc) lexer
    CHINESE_VGRAM_LEXER;
2 create context index xzcflltextindex on TS_B_XZCF(
    xdr) lexer CHINESE_VGRAM_LEXER;
3 alter context index xzcflltextindex on TS_B_XZCF
    rebuild;
```

填充索引:

```
1 alter context index qymcindex1 on user_xycq.  
   TS_F_CORPORATION rebuild;
```

删除索引:

```
1 drop context index qymcindex on user_xycq.  
   TS_F_CORPORATION_TEST;
```

达梦数据库中, 在写全文索引查询时, 全文索引的条件需要放到前面, 这样才能利用到索引。

16.2 Redis

16.2.1 安装

这里是在 Raspberry Pi 上安装 Redis:

```
1 # 下载源码  
2 wget -c http://download.redis.io/releases/redis  
   -3.2.9.tar.gz  
3 # 编译安装  
4 make  
5 make test
```


17. PostgreSQL

PostgreSQL 是自由的对象-关系型数据库服务器（数据库管理系统），在灵活的 BSD-风格许可证下发行。它在其他开放源代码数据库系统（比如 MySQL 和 Firebird），和专有系统比如 Oracle、Sybase、IBM 的 DB2 和 Microsoft SQL Server 之外，为用户又提供了一种选择。

17.1 基础操作

17.1.1 安装 (Install)

由于 OS 发行版本特殊，没有现成 Binary 包，所以从源码手动编译安装，选择的是 9.6.5 版本，运行 configure 的时候出现如下错误：

configure: error: readline library not found If you have readline already installed, see config.log for details on the failure. It is possible the compiler isn't looking in the proper directory. Use `--without-readline` to disable readline support.

提示缺失 readline¹包，GNU Readline 是一个开源的程序库，可以实现交互式的文本编辑功能。网上说安装 readline-devel 包和 libtermcap-devel 包可以解决这个问题²(Install the readline-devel and libtermcap-devel³ to solve the above issue).

由于是内网，无法连接互联网，手动安装依赖太复杂 (包的依赖复杂，A 包依

¹<https://cnswww.cns.cwru.edu/php/chet/readline/rltop.html>

²<http://www.thegeekstuff.com/2009/04/linux-postgresql-install-and-configure-from-source>

³https://centos.pkgs.org/5/centos-x86_64/libtermcap-devel-2.0.8-46.1.x86_64.rpm.html

赖 B 包, B 包依赖 C 包, 安装几个包之后整个人都不好了, 这才真的真的体会到包管理工具的方便)。所以考虑了 2 个方案, 一是在内网搭建一个离线 yum(Yellowdog Updater, Modified⁴) 源服务器, 二是在可以连接互联网的机器上采用 yum 下载好包之后, 在另一台内网机器上手动安装。使用如下命令下载依赖包:

```
1 sudo yum install --downloadonly <package-name>
2 yum install --downloadonly -y perl-ExtUtils-Embed
   readline-devel zlib-devel pam-devel libxml2-
   devel libxslt-devel openldap-devel python-
   devel gcc-c++ openssl-devel cmake
3 # Ubuntu 下仅仅下载而不安装软件包
4 # 下载的目录一般为/var/cache/apt/archives
5 sudo apt-get -d install wget
```

参数 d 表示仅仅在下载模式 (Download Mode), 并不会安装软件, 默认情况下, 一个下载的 RPM 包会保存在下面的目录中:

```
1 /var/cache/yum/x86_64/[centos/fedora-version]/[
   repository]/packages
```

如果想要将一个包下载到一个指定的目录 (如/tmp):

```
1 sudo yum install --downloadonly --downloadaddir=/tmp <
   package-name>
```

登录:

```
1 sudo -u postgres psql
```

这里是以超级用户的身份登录, 否则在修改密码时会提示: must be superuser to alter superusers。修改密码:

```
1 \password postgres
```

Gradle 引用驱动:

```
1 compile "org.postgresql:postgresql:42.1.1"
```

⁴[https://en.wikipedia.org/wiki/Yum_\(.rpm\)](https://en.wikipedia.org/wiki/Yum_(.rpm))

PostgreSQL 新建列时，列名推荐使用小写，列名使用小写时，`select` 里大小写都可以查询，如果列名为大写，查询语句中列名需要添加双引号才能查询。

18. MySQL

18.0.1 查询

MySQL 远程连接 (MySQL Remote Connection)

本来以为安装完毕 MySQL 就可以自动进行远程连接, 结果竟然花了半天时间才搞定。安装完毕后 MySQL 默认是不能远程连接的, 由于在其他主机上用 nmap 无法扫描到端口 3306, 开始还以为是防火墙的缘故, 在错误的道路上研究了好久, 最后竟然在本机也无法扫描到端口。才明白 MySQL 是与 localhost 绑定而不是与 IP 绑定。在目录/etc/mysql/目录中修改配置文件 my.cnf:

```
1 # Instead of skip-networking the default is now to  
   listen only on  
2 # localhost which is more compatible and is not less  
   secure.  
3 bind-address          = 192.168.31.25
```

默认是 127.0.0.1, 是一个回环地址, 只有本机请求本机, 目标地址才会是 127.0.0.1, 此时需要在其他主机请求, 需要将绑定 IP 改为网卡地址, 也就是外网 IP, 因为远程连接数据库目标 IP 肯定是公网 IP. 不过还可以改成 0.0.0.0, 这样就表示监听所有 IP, 只要端口一样, 网卡都会把请求传给进程。如果不在配置文件中修改地址, 则需要启动时指定绑定地址:

```
1 sudo mysqld --bind-address=192.168.31.25
```

绑定好地址后, 使用命令 `sudo lsof:3306` 查看。如果填写的绑定地址是 192 开

头,那么最终只有 192 开头的局域网里的机器能够远程连接 MySQL,有时一台设备可能会处于多个局域网中,那么此时另一个局域网的设备就无法远程连接 MySQL 了。此时可以将绑定地址修改为 0.0.0.0,即允许所有的 IP 连接到 MySQL,相应的 MySQL 的安全性相应的就降低了。

18.0.2 Mac 中 MySQL 初始密码

step1:

苹果-> 系统偏好设置-> 最下边点 mysql 在弹出页面中关闭 mysql 服务 (点击 stop mysql server)

step2:

进入终端输入: cd /usr/local/mysql/bin/ 回车后登录管理员权限 sudo su 回车后输入以下命令来禁止 mysql 验证功能./mysqld_safe --skip-grant-tables & 回车后 mysql 会自动重启 (偏好设置中 mysql 的状态会变成 running)

step3. 输入命令./mysql 回车后,输入命令 FLUSH PRIVILEGES; 回车后,输入命令 SET PASSWORD FOR 'root'@'localhost' = PASSWORD('你的新密码');

18.0.3 基础

启动 MySQL:

```
1  mysqld
2
3  # Mac 中启动 MySQL, 同理停止的参数为 stop, 重启的参数
   为 restart
4  sudo /usr/local/MySQL/support-files/mysql.server
   start
5
6  # 防火墙添加端口
7  sudo iptables -A INPUT -p tcp --dport 3306 -j ACCEPT
   /*允许包从3306端口进入*/
8  sudo iptables -A OUTPUT -p tcp --sport 3306 -m state
   --state ESTABLISHED -j ACCEPT /*允许从3306端
   口进入的包返回*/
9
10 # 查看 iptables
11 sudo iptables --list
12 SHOW DATABASES //列出
   MySQL Server 数据库。
```

```

13 SHOW TABLES [FROM db_name]           //列出
    数据库数据表。
14 SHOW TABLE STATUS [FROM db_name]      //列出
    数据表及表状态信息。
15 SHOW COLUMNS FROM tbl_name [FROM db_name] //列出
    资料表字段
16 SHOW FIELDS FROM tbl_name [FROM db_name], DESCRIBE
    tbl_name [col_name]。
17 SHOW FULL COLUMNS FROM tbl_name [FROM db_name] //列出
    字段及详情
18 SHOW FULL FIELDS FROM tbl_name [FROM db_name] //列出
    字段完整属性
19 SHOW INDEX FROM tbl_name [FROM db_name] //列出
    表索引。
20 SHOW STATUS                           //列出
    DB Server 状态。
21 SHOW VARIABLES                         //列出
    MySQL 系统环境变量。
22 SHOW PROCESSLIST                       //列出执
    行命令。
23 SHOW GRANTS FOR user                   //列出某
    用户权限
24 GRANT ALL PRIVILEGES ON *.* TO 'root'@'192.168.31.25'
    IDENTIFIED BY 'dolphin' WITH GRANT OPTION;

```

在使用 MySQL 会出现中文乱码问题，MySQL 的字符集支持 (Character Set Support) 有两个方面：字符集 (Character set) 和排序方式 (Collation)。对于字符集的支持细化到四个层次：服务器 (server)，数据库 (database)，数据表 (table) 和连接 (connection)。在 Mac 中，在 etc 目录下新建 my.cnf 文件，指定数据库编码：

```

1 [mysqld]
2 character-set-server=utf8
3 init_connect='SET NAMES utf8'
4 [mysql]
5 default-character-set=utf8

```

修改后，重新启动数据库。但是以前新建的表有可能还是采用的原来的编码，

使用如下语句进行修改：

```
1 --检查数据表所有字段的状态
2 show full columns from book;
3 --发现 address 字段的 Collation 项非 utf8, 改成 utf8
4 alter table book change publisher publisher varchar
    (512) character set utf8 collate
    utf8_unicode_ci not null;
```

18.0.4 备份

```
netstat -ln | grep mysql
```

```
mysqldump --sock=/var/run/mysqld/mysqld.sock -u root -p dolphin>dolphin.sql
```

18.0.5 存储过程

```
1 DELIMITER $$
2
3 CREATE DEFINER='root'@'%' PROCEDURE `
    insert_initial_id`()
4 BEGIN
5
6 DECLARE i INT DEFAULT 1;# can not be 0
7 DECLARE j INT DEFAULT 10000000;# can not be 0
8
9
10 WHILE i<99999999 && j < 10000050
11 DO
12 insert into douban_book_id(isscapy,douban_book_id)
    values (0,j);
13 SET i=i+1;
14 set j=j+1;
15 END WHILE ;
16 commit;
17
18 END$$
19 DELIMITER ;
```


18.0.6 mycli

MyCli 是一个 MySQL 命令行工具，支持自动补全和语法高亮。也可用于 MariaDB 和 Percona。安装 mycli：

```
1 pip install mycli
2 brew install mycli
```

使用 MyCli 连接数据库示例如下：

```
1 mycli -h 10.0.0.14 -u root -p dolphin
```

在 Ubuntu 下远程配置如图所示：

18.0.7 常见问题

ERROR 2002 (HY000): Can't connect to local MySQL server through socket '/tmp/mysql.sock'

mysql 使用 unix socket 或者 tcp 来连接数据库进行通讯，默认不加 -h 选项时使用的就是 localhost 即 unixsocket，此时会通过/tmp/mysql.sock 来通讯，但是在配置文件中默认生成的 socket 文件是在/var/lib/mysql/mysql.sock(不同安装可能不同，建议查看/etc/my.cnf 确认)，所以要想 mysql 使用这个文件通讯，最简单的方法就是建立软链接，一劳永逸，此为方法一

方法二就是强制 mysql 使用 tcp 通讯，因为 127.0.0.1 对于 mysql 来说走的是 tcp 协议而非 unixsocket，这种方法的弊端就是每次都要指明本地地址 127.0.0.1

18.0.8 中文查询

在用 MyBatis 查询中文时，出来的结果却不是中文匹配的结果，将 SQL 在客户端工具中执行却可以正确查询。在 MySQL 配置文件中修改相应的编码。在服务端/etc/mysql/my.cnf 修改相应编码：

```
1 [mysqld]
2
3 character-set-server=utf8
```


19. 达梦

启动达梦：

```
1 /home/hldev/dmdbms/bin/dmserver /home/hldev/dmdbms/  
data/DAMENG/dm.ini -noconsole
```

19.1 SQL

19.1.1 索引

全文索引未生效时，可以到 CTISYS 模式下查看系统所有的全文索引：

```
1 select *  
2 from "CTISYS"."SYSCONTEXTINDEXES";
```

新建索引时，选择独立的索引存储空间，最好是索引存储与数据存储独立，便于管理。每个用户建立一个数据表空间，一个索引表空间。采用函数的列应该保存在索引中。添加上 (a,b,c) 的组合索引相当于添加了，超过 3 个列的联合索引不合适，否则虽然减少了回表动作，但索引块过多，查询时就要遍历更多的索引块了，建索引动作应谨慎，因为建索引的过程会产生锁，不是行级锁，而是锁住整个表，任何该表的 DML 操作都将被阻止，在生产环境中的繁忙时段建索引是一件非常危险的事情。在达梦客户端中，按下 Ctrl + Alt + L 会出来所有快捷键的面板，不得不说这个设计还是非常贴心的，开始真的不敢相信自己的眼睛，反复确认了真的不是

IntelliJ Idea 调出的面板，是达梦调出来的，非常不错的设计。注意平台是 Ubuntu 16.04 LTS。

```
1 create index "TS_C500035_JBXX_DJXX_GTJBXX_ID" on "  
    USER_XYCQ_ZS"."TS_C500035_JBXX_DJXX_GTJBXX"("  
    ID") storage(initial 1,next 1,minextents 1);
```

19.1.2 函数

根据条件来计算：

```
1 SELECT ORG_ID,  
2 ORG_NAME,  
3 INTERFACE_NAME,  
4 COUNT(*) AS TOTAL,  
5 SUM(CASE WHEN RETURN_COUNT > 0 THEN 1 ELSE 0 END) AS  
    VALID_TIMES,  
6 ROUND(CONVERT(DOUBLE,SUM(CASE WHEN RETURN_COUNT > 0  
    THEN 1 ELSE 0 END))/CONVERT(DOUBLE,COUNT(*))  
    ,3) AS RATIO  
7 FROM TD_M_APPACCESS_LOG  
8 GROUP BY ORG_ID,ORG_NAME,INTERFACE_NAME
```

如上语句统计接口的有效调用次数，返回数量 (RETURN_COUNT) 大于 0 时，视为有效调用次数 (VALID_TIMES)。数量默认为整数，所以要得到小数的比率 (RATIO)，就需要使用函数 convert 将数据转换为 Double 类型。

开窗函数

使用简单函数进行统计的语句如下：

```
1 select org_id,  
2     max(org_name) org_name,  
3     sum(using_times) using_times,  
4     sum(valid_times) valid_times,  
5     sum(feedback_count) feedback_count,  
6     sum(handle_type_count) handle_type_count  
7 from (  
8     select
```

```

9      b.org_id,b.org_name,
10      SUM(count(b.responsekey)) over(partition by
      b.responsekey) using_times,--使用次
      数
11      SUM(CASE WHEN b.isokred > 0 OR b.isokblack >
      0 THEN 1 ELSE 0 END) AS VALID_TIMES,--触发反
      馈次数
12      sum(case when b.is_feedback = 1 then 1 else
      0 end) FEEDBACK_COUNT, --反馈次
      数
13      sum(case when b.handle_type = 1 then 1 else
      0 end) HANDLE_TYPE_COUNT --已实施奖惩次
      数
14      from (
15          --表联合基础信息
16          select  a.org_id,
17                  a.org_name,
18                  cxnr,--相对人
19                  a.responsekey responsekey,--responsekey
      一次查询标记, 统计查询次数
20                  a.ISOKRED isokred,--红名单个数, 统计触发
      次数
21                  a.ISOKBALCK isokblack,--黑名单个数, 统计
      触发次数
22                  a.is_feedback, --是否反馈标志, 统计反馈次
      数
23                  c.handle_type --已实施奖惩数
24      from TI_B_REDBALCKLOG a,TI_B_FEEDBACK_LOG
      c
25      where c.responsekey = a.responsekey
26      and a.cxnr = c.xdr --反馈根据相对人反馈,
      以相对人粒度进行统计
27      ) as b
28      group by b.org_id,b.org_name,b.responsekey,b.
      cxnr
29  ) d
30  group by org_id

```

19.1.3 常用 SQL 记录

嵌套查询：

```

1 select GNMK,
2 max(ORG_NAME) as ORG_NAME,
3 ORG_ID,
4 COUNT(*) AS USING_TIMES,
5 SUM(CASE WHEN ISOKRED > 0 OR ISBLACKRED > 0 THEN 1
        ELSE 0 END) AS VALID_TIMES,
6 SUM(CASE WHEN ISOKRED > 0 THEN 1 ELSE 0 END) AS
        VALID_COUNT_RED,
7 SUM(CASE WHEN ISBLACKRED > 0 THEN 1 ELSE 0 END) AS
        VALID_COUNT_BLACK
8 from (
9     SELECT GNMK,
10     MAX(ORG_NAME) AS ORG_NAME,
11     ORG_ID,
12     sum(ISOKRED) as ISOKRED,
13     sum(ISOKBALCK) AS ISBLACKRED,
14     RESPONSEKEY
15     FROM TI_B_REDBALCKLOG
16     GROUP BY RESPONSEKEY,ORG_ID,GNMK
17 )
18 GROUP BY ORG_ID,GNMK
19 order by ORG_ID DESC limit 0, 10
20
21 --修改参数
22 select *
23 from v$dm_ini
24 where para_name like '%ORDER%'
25
26 select * from v$sessions where state='ACTIVE';

```

删除数据库 ID 重复的记录，只保留其中一条记录：

```

1 delete from dolphin where rowid not in

```

```
2 (
3     select max(rowid)
4     from dolphin
5     group by id
6 )
```

拼接列：

19.2 通用知识

19.2.1 查询优化

最近使用数据库中，遇到查询性能低下的问题。数据量大概在 800W 左右，但是分页查询需要 40s 左右才能够查询出来结果，测试库 200W 数据量左右，分页查询也需要 10s 才能出结果。

使用 MySQL，有的人 BB 说 5.1 以前 300w 就会很明显，5.5-5.6 已经有很大优化，如果是 5.7 千万也不是问题。随着互联网的发展，数据的量级也是撑指数的增长，从 GB 到 TB 到 PB。对数据的各种操作也是愈加的困难，传统的关系性数据库已经无法满足快速查询与插入数据的需求。这个时候 NoSQL 的出现暂时解决了这一危机。它通过降低数据的安全性，减少对事务的支持，减少对复杂查询的支持，来获取性能上的提升。但是，在有些场合 NoSQL 一些折衷是无法满足使用场景的，就比如有些使用场景是绝对要有事务与安全指标的。这个时候 NoSQL 肯定是无法满足的，所以还是需要使用关系性数据库。

增加缓存

增加缓存可以提高查询效率，但是在对数据表做任何的更新操作 (update/insert/delete) 等操作，server 为了保证缓存与数据库的一致性，会强制刷新缓存数据，导致缓存数据全部失效。

分库分表

分库分表应该算是查询优化的杀手锏了。上述各种措施在数据量达到一定等级之后，能起到优化的作用已经不明显了。这个时候就必须对数据量进行分流。分流一般有分库与分表两种措施。而分表又有垂直切分与水平切分两种方式。下面我们就针对每一种方式简单介绍。

对于 mysql，其数据文件是以文件形式存储在磁盘上的。当一个数据文件过大的时候，操作系统对大文件的操作就会比较麻烦与耗时，而且有的操作系统就不支持大文件，所以这个时候就必须分表了。另外对于 mysql 常用的存储引擎是 InnoDB，它的底层数据结构是 B+ 树。当其数据文件过大的时候，B+ 树就会从层

次和节点上比较多，当查询一个节点的时候可能会查询很多层次，而这必定会导致多次 IO 操作进行装载进内存，肯定会耗时的。除此之外还有 Innodb 对于 B+ 树的锁机制。对每个节点进行加锁，那么当更改表结构的时候，这时候就会树进行加锁，当表文件大的时候，这可以认为是不可实现的。

分区表

分区表是 MySQL 5.1 引入的特性。根据官网 [alter-table-partition-operations](#) 的介绍，其本质是将分库分表直接集成到 MySQL 中。我们知道，传统的分库分表功能，存在业务层、中间件、数据库三层：业务层通过调用中间件的 API 访问数据库，不知道具体的物理存储细节；中间件将一张很大的逻辑表映射到数据库中多张较小的物理表，并对业务层的访问请求进行分解后分别放到对应物理库中执行，再将执行结果在中间件合并后返回给业务层，从而对业务层屏蔽物理存储细节；数据库则提供实际的物理存储。而 MySQL 的分区表，借助 MySQL 本身的逻辑架构，将分库分表功能进行了下沉。MySQL 逻辑架构中的客户端即对应业务层，Server 层对应中间件层，存储引擎层对应物理存储层。简单的说，分库表就是我们在数据库层面看到是一张表，但物理上是分成多个文件独立存储。逻辑上分析，分区表的优点很明显：既能解决大数据量的性能问题，又能对应用层无缝切换。

采用 NoSQL 数据库

Hbase 读的性能比较强，官方定义为适合 PB 级数据的秒级查询，就是说在巨海量的数据中查询，响应速度非常快。

19.2.2 系统架构

从系统架构来看，目前的商用服务器大体可以分为三类，即对称多处理器结构 (SMP : Symmetric Multi-Processor)，非一致存储访问结构 (NUMA : Non-Uniform Memory Access)，以及海量并行处理结构 (MPP : Massive Parallel Processing)

19.2.3 注意事项

不要 ORDER BY RAND()

MySQL 会不得不去执行 RAND() 函数（很耗 CPU 时间），而且这是为了每一行记录去记行，然后再对其排序。就算是你用了 Limit 1 也无济于事。自己在做爬虫时，写过这样的语句：

```
select * from douban_book_id where isscapy = 0 limit 1
```


难怪后来变的很慢，把中间停顿的 `sleep` 函数去掉之后还是很慢，开始还一位是 `sleep` 函数暂停的问题。



Network

20	HTTP	213
20.1	PAC(Proxy Auto-Configuration)	

20. HTTP

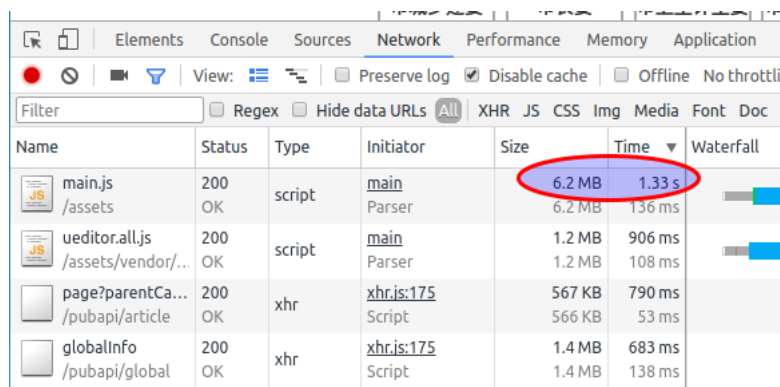
20.0.1 性能分析

慢的 SQL 查询

在项目中报表打印页面非常慢，需要 10 几秒才能出结果。打开 Network，找到所有请求按照时间排列发现耗时最多的一个请求，针对此请求进行优化。在 SQL 中添加相应的索引可以解决。

大的 Javascript 脚本

在项目中，发现有的 JavaScript 脚本特别大，有 6MB+ 之巨，明显是造成性能瓶颈的原因之一，如图20.1所示。



Name	Status	Type	Initiator	Size	Time	Waterfall
main.js /assets	200 OK	script	main Parser	6.2 MB	1.33 s	
ueditor.all.js /assets/vendor/...	200 OK	script	main Parser	1.2 MB	906 ms	
page?parentCa... /pubapi/article	200 OK	xhr	xhr.js:175 Script	567 KB	790 ms	
globalInfo /pubapi/global	200 OK	xhr	xhr.js:175 Script	1.4 MB	683 ms	

Figure 20.1: 较大的 Javascript 脚本

去掉 Javascript 注释 在 webpack.config.js 文件的 plugins 数组里面添加及配置插件即可。

```
1 new UglifyJsPlugin({
2   output: {
3     comments: false, //去掉 Javascript 中的注释
4   },
5   compress: {
6     warnings: false
7   }
8 })
```

去掉注释后小了 400KB 左右，但是还是有 6MB，没有从根本上解决问题。

添加插件 添加一些插件压缩包。

```
1 plugins: [
2   // <- key to reducing React's size
3   new webpack.DefinePlugin({
4     'process.env': {
5       'NODE_ENV': JSON.stringify('production')
6     }
7   }),
8   //dedupe similar code
9   new webpack.optimize.DedupePlugin(),
10  //minify everything
11  new webpack.optimize.UglifyJsPlugin(),
12  //Merge chunks
13  new webpack.optimize.AggressiveMergingPlugin()
14 ]
```

React 切换到产品环境

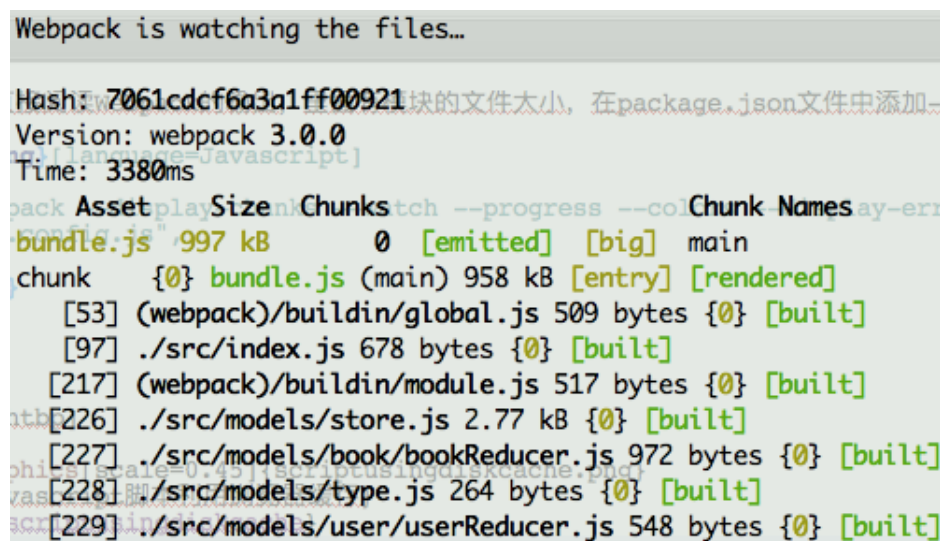
查看 Webpack 输出 最最直接的，可以直接阅读 Webpack 的输出，里面有模块的文件大小，在 package.json 文件中添加 `-display-chunks` 参数，配置如下：

```

1 "scripts": {
2   "dev": "webpack --display-chunks --watch --
      progress --colors --display-error-details --
      config webpack/dev.config.js",
3 }

```

输出如图20.2所示。也可以在输出中查找是否包含重复依赖。



```

Webpack is watching the files...

Hash: 7061cdcf6a3a1ff00921
Version: webpack 3.0.0
Time: 3380ms

Asset      Size  Chunks             Chunk Names
bundle.js  997 kB          0 [emitted] [big]  main
chunk      {0} bundle.js (main) 958 kB [entry] [rendered]
  [53] (webpack)/buildin/global.js 509 bytes {0} [built]
  [97] ./src/index.js 678 bytes {0} [built]
 [217] (webpack)/buildin/module.js 517 bytes {0} [built]
 [226] ./src/models/store.js 2.77 kB {0} [built]
 [227] ./src/models/book/bookReducer.js 972 bytes {0} [built]
 [228] ./src/models/type.js 264 bytes {0} [built]
 [229] ./src/models/user/userReducer.js 548 bytes {0} [built]

```

Figure 20.2: Webpack 查看输出文件大小

利用浏览器缓存 有时优化之后 Javascript 的大小还是不够理想，此时可以考虑将 Javascript 的文件名用 Hash 进行标识，Hash 是经过文件计算出来的唯一值，在文件有改动后，会自动更新，浏览器仅仅在首次请求时下载较大的 Javascript 脚本，以后就可以直接利用本地缓存。此种方式不需要频繁的访问服务器下载 Javascript 脚本，提升本地加载速度的同时也减轻了服务端的压力，如图20.3所示。

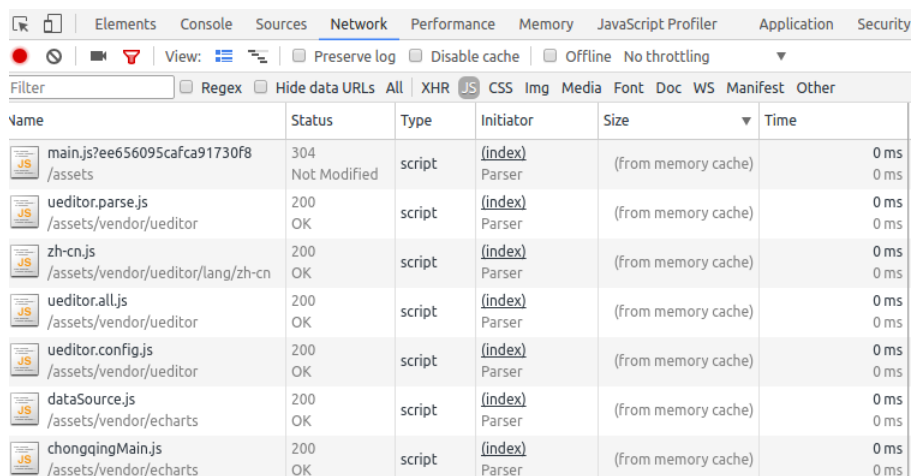
由于应用是 SPA(Single Page Application)，上图中单单一个 main.js 脚本都会有 5MB 之巨，加上其他的大大小的 Javascript 脚本，在未经过服务端压缩的情况下会有 10MB 以上的大小，在普通的网络环境下肯定是非常慢。如果下载速度时 1MB/s，打开应用至少需要 10s，将是用户无法忍受的。

开启 Gzip 压缩 同时可以考虑在服务端开启 Gzip 压缩，开启 Gzip 压缩配置如下：

```

1 #仅仅开启 Gzip 还不行，需要各项详细配置
2 gzip on;

```



Name	Status	Type	Initiator	Size	Time
main.js?ee656095cafca91730f8/assets	304 Not Modified	script	(index) Parser	(from memory cache)	0 ms
ueditor.parse.js/assets/vendor/ueditor	200 OK	script	(index) Parser	(from memory cache)	0 ms
zh-cn.js/assets/vendor/ueditor/lang/zh-cn	200 OK	script	(index) Parser	(from memory cache)	0 ms
ueditor.all.js/assets/vendor/ueditor	200 OK	script	(index) Parser	(from memory cache)	0 ms
ueditor.config.js/assets/vendor/ueditor	200 OK	script	(index) Parser	(from memory cache)	0 ms
dataSource.js/assets/vendor/echarts	200 OK	script	(index) Parser	(from memory cache)	0 ms
chongqingMain.js/assets/vendor/echarts	200 OK	script	(index) Parser	(from memory cache)	0 ms

Figure 20.3: Javascript 脚本利用浏览器缓存

```

3 gzip_min_length 1k;
4 gzip_buffers 4 16k;
5 gzip_comp_level 2;
6 gzip_types application/javascript application/json
    text/plain application/x-javascript text/css
    application/xml text/javascript application/x-
    -httpd-php image/jpeg image/gif image/png;
7 gzip_vary off;
8 gzip_disable "MSIE [1-6]\.";

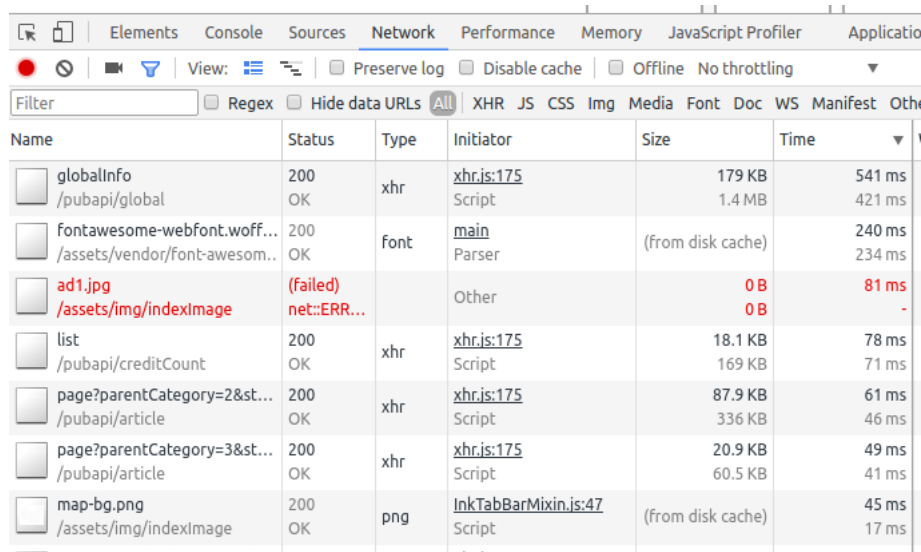
```

开启服务端压缩后效果非常显著，如图20.4所示。

图中，globalInfo 请求，压缩之前返回的 Json 有 1.4MB 大小，压缩之后只有 179KB，压缩之前传输时间在 1.5s 左右，压缩之后传输时间缩短了三分之二。另外比较大的 6MB 的 Javascript 脚本也变成了 1.2MB 大小。

字体文件加载

发现一个字体文件 (fontawesome-webfont.woff2) 在 Linux 下的 Chrome 中是从磁盘中加载 (from disk cache) 的，花费了 220ms 左右。在 Windows 平台下的 360 浏览器中是从内存中加载 (from cache) 的。diskCache 顾名思义，就是将资源缓存到磁盘中，等待下次访问时不需要重新下载资源，而直接从磁盘中获取，它的直接操作对象为 CurlCacheManager。它与 memoryCache 最大的区别在于，当退出进程时，内存中的数据会被清空，而磁盘的数据不会，所以，当下次再进入该进程时，该进程仍可以从 diskCache 中获得数据，而 memoryCache 则不行。diskCache 与 memoryCache 相似之处就是也只能存储一些派生类资源文件。它的存储形式为



Name	Status	Type	Initiator	Size	Time
globalInfo /pubapi/global	200 OK	xhr	xhr.js:175 Script	179 KB 1.4 MB	541 ms 421 ms
Fontawesome-webfont.woff... /assets/vendor/font-awesom...	200 OK	font	main Parser	(from disk cache)	240 ms 234 ms
ad1.jpg /assets/img/indexImage	(failed) net::ERR...		Other	0 B 0 B	81 ms -
list /pubapi/creditCount	200 OK	xhr	xhr.js:175 Script	18.1 KB 169 KB	78 ms 71 ms
page?parentCategory=2&st... /pubapi/article	200 OK	xhr	xhr.js:175 Script	87.9 KB 336 KB	61 ms 46 ms
page?parentCategory=3&st... /pubapi/article	200 OK	xhr	xhr.js:175 Script	20.9 KB 60.5 KB	49 ms 41 ms
map-bg.png /assets/img/indexImage	200 OK	png	InkTabBarMixin.js:47 Script	(from disk cache)	45 ms 17 ms

Figure 20.4: 开启服务端压缩效果

一个 index.dat 文件，记录存储数据的 url，然后再分别存储该 url 的 response 信息和 content 内容。Response 信息最大作用就是用于判断服务器上该 url 的 content 内容是否被修改。

耗时长请求

查看首页中一个耗时最长的请求，一般情况下耗时都稳定在 1s 左右。如图 20.5 所示。可以看出挂起 (stalled) 304ms，等待服务端响应 581ms，此两项的时间是最长的。Stalled 从 HTTP 连接建立到请求能够被发出送出去 (真正传输数据) 之间的时间花费。包含用于处理代理的时间，如果有已经建立好的连接，这个时间还包括等待已建立连接被复用的时间。挂起时间长是由于浏览器对同一个主机域名的并发连接数有限制，因此如果当前的连接数已经超过上限，那么其余请求就会被阻塞，等待新的可用连接；此外脚本也会阻塞其他组件的下载。另外在不同的平台上挂起时间时长也不一定，在 Windows 平台的 360 浏览器上挂起时长 (stalled) 仅仅为 0.458ms。

针对服务器端耗时长的请求，使用 VisualVM 查看服务端调用的耗时最长的方法。VisualVM 是一款免费的性能分析工具。它通过 jvmstat、JMX、SA (Serviceability Agent) 以及 Attach API 等多种方式从程序运行时获得实时数据，从而进行动态的性能分析。同时，它能自动选择更快更轻量级的技术尽量减少性能分析对应用程序造成的影响，提高性能分析的精度。

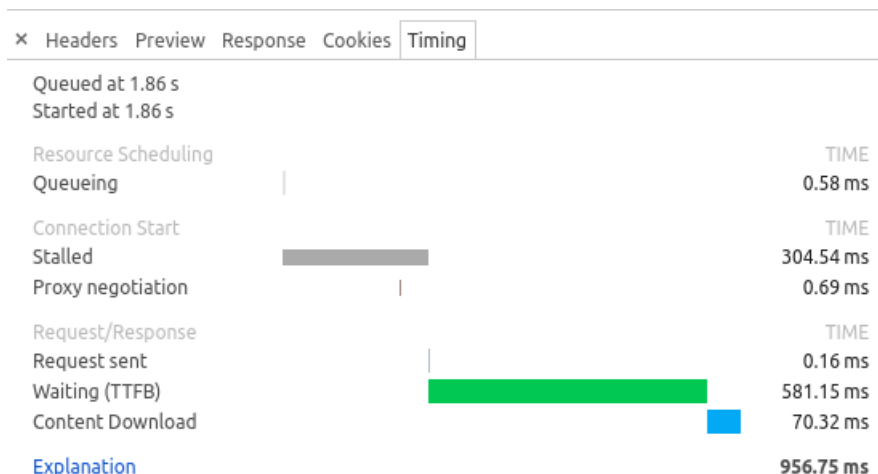


Figure 20.5: 耗时长请求

20.0.2 Session

当浏览器第一次请求时,服务器创建一个 session 对象,同时生成一个 sessionId,并在此次响应中将 sessionId 以响应报文的方式传回客户端浏览器内存或以重写 url 方式送回客户端,来保持整个会话。关闭此浏览器窗口,其内存中的 sessionId 也就随之销毁。session.invalidate() 是将 session 设置为失效,一般在退出时使用,但要注意: session 失效的同时浏览器会立即创建一个新的 session 的,你第一个 session 已经失效了所以调用它的 getAttribute 方法时候一定会抛出 NullPointerException。jsessionId 只是 tomcat 中对 session id 的叫法,在其它容器里面,不一定是叫 jsessionid。

20.1 PAC(Proxy Auto-Configuration)

20.1.1 PAC 简介 (Pac Introduce)

PAC 是 proxy auto-config 的缩写。PAC 文件是纯文本格式的,实际上就是 JavaScript 文件。PAC 最简单的格式就是包含一个叫 FindProxyForURL 的 JavaScript 函数,IE 通过传入两个变量来调用这个函数。一个是用户流量的地址的 URL 全路径,一个是这个 URL 中的主机名 host。在日常生活中浏览器中设置代理很简单,但是当你来回切换时总是觉得很烦,你可以使用 pac 脚本自动判断是否走代理,方便省去了来回手动切换的烦恼。

20.1.2 PAC 实例 (PAC Example)

一个最简单的 PAC 脚本:

```
function FindProxyForURL(url,host){
```

```
2     return "DIRECT";
3 }
```

参数 url 是用户输入的 url, 参数 host 是 url 中的主机名。PAC 文件返回值有三种类型:

- DIRECT 直连不通过代理
- PROXY www.lybbn.cn:8080 http 通过 8080 端口代理上网, 也可以使用 ip:port 的形式
- SOCKS5 www.lybbn.cn:8080 socks 通过 8080 端口代理上网, 可以使用 ip:port 形式

```
1 var FindProxyForURL = function(init, profiles) {
2     return function(url, host) {
3         "use strict";
4         var result = init, scheme = url.substr(0, url
5             .indexOf(":"));
6         do {
7             result = profiles[result];
8             if (typeof result === "function") result
9                 = result(url, host, scheme);
10        } while (typeof result !== "string" || result
11            .charAt(0) === 43);
12        return result;
13    };
14    }("+dolphin2", {
15        "+dolphin2": function(url, host, scheme) {
16            "use strict";
17            if (/^59\.214\.215\.6$/.test(host))
18                return "+dolphin-proxy";
19            if (/^10\.10\.1\.11$/.test(host)) return
20                "+dolphin-proxy";
21            return "DIRECT";
22        },
23        "+dolphin-proxy": function(url, host, scheme
24            ) {
25            "use strict";
```

```
20     if (/^127\.0\.0\.1$/.test(host) || /^::1$/.  
    test(host) || /^localhost$/.test(host))  
        return "DIRECT";  
21     return "PROXY 10.55.10.2:8888";  
22 }  
23 });
```

以上脚本说明, 当 IP 为 58.214.215.* 或 10.10.1.11 时, 使用代理 dolphin-proxy, 而代理 dolphin-proxy 设置的是代理机器的相关信息, 代理机器的 IP 为 10.55.10.2, 代理的端口是 8888。

20.1.3 同时连接内外网

有时需要同时连接内网和互联网 (内网与互联网不互通), 开发环境需要依赖内网, 与同事交流、查询资料等需要依赖互联网, 而内网与互联网切换是一大痛点, 不仅影响效率, 同时也影响心情, 宝贵的时间就在这样无意义的开关中白白浪费掉了。同时使用有线网卡和无线网卡, 既能保证内网访问的安全, 又能避免频繁切换网卡带来的时间开销, 可谓一举两得。一般的计算机有有线网卡, 也有无线网卡。一般情况下要么使用有线网卡, 要么使用无线网卡, 在使用无线时, 连接上了有线, 因为有线网卡的优先级高, 故此时仅有有线能够工作, 无线网卡可连接但是无法传送数据。实现双网卡的基本思路是删除默认网关, 配置各自的网关即可。

Mac 同时连内网外网

输入如下命令查看 Mac 的所有网络连接方式:

```
1 networksetup -listallnetworkservices
```

输出的结果如下:

```
1 An asterisk (*) denotes that a network service is  
   disabled.  
2 Apple USB Ethernet Adapter  
3 Wi-Fi  
4 Bluetooth PAN  
5 Thunderbolt Bridge
```

可以看出 Mac 可以通过 Wi-Fi 联网, 也可以通过 USB 有线联网。给指定的网络连接方式设定 DNS 服务器代码如下:

```
1 sudo networksetup -setdnsservers AirPort
    192.168.10.200
```

清空 DNS 缓存代码如下：

```
1 dscacheutil -flushcache
```

输入如下命令查看 MacBook 的路由表：

```
1 netstat -nr
```

Fedora、Ubuntu 同时连接内外网

给有线网卡配置地址信息时，内网网卡不要加默认网关，外网网卡加默认网关，并查看无线网卡分配到的地址信息中是否有默认网关。在这里内网通过网线连接，是有线网络，外网通过无线网卡连接路由器，是无线网络。使用 route 命令查看默认路由 (在 Mac 下使用 netstate -nr 命令)，输出如图20.6所示：

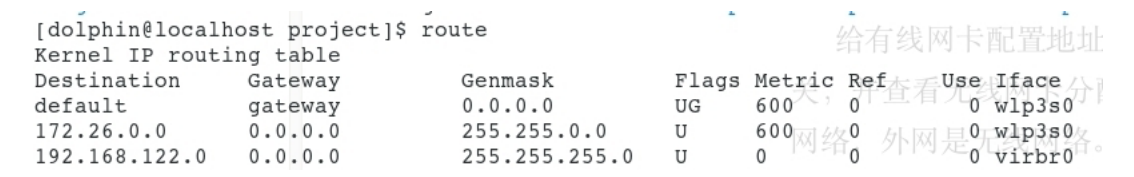


Figure 20.6: 查看 Linux Kernel 路由表

IP 路由选择主要完成以下这些步骤：1) 搜索路由表，寻找能与目的 IP 地址完全匹配的表目（网络号和主机号¹都要匹配）。如果找到，则把报文发送给该表目指定的下一站路由器或直接连接的网络接口（取决于标志字段的值）。2) 搜索路由表，寻找能与目的网络号相匹配的表目。如果找到，则把报文发送给该表目指定的下一站路由器或直接连接的网络接口（取决于标志字段的值）。目的网络上的所有主机都可以通过这个表目来处置。例如，一个以太网上的所有主机都是通过这种表目进行寻径的。这种搜索网络的匹配方法必须考虑可能的子网掩码。3) 搜索路由表，寻找标为“默认 (default)”的表目。如果找到，则把报文发送给该表目指定的下一站路由器。如果上面这些步骤都没有成功，那么该数据报就不能被传送。如果不能传送的数据报来自本机，那么一般会向生成数据报的应用程序返回一个“主机不可达”或“网络不可达”的错误。完整主机地址匹配在网络号匹配之前执行。

¹当前使用的 IP 地址有 4 个字节 (32bit) 组成，即 IPV4 编码方式。每个 IP 地址包括两部分：网络号和主机号。当分配给主机号的二进制位越多，则能标识的主机数就越多，相应地能标识的网络数就越少，反之同理。

Table 20.1: flag 含义

序号	名称
U	Up 表示此路由当前为启动状态
H	Host, 表示此网关为一主机
G	Gateway, 表示此网关为一路由器
R	Reinstate Route, 使用动态路由重新初始化的路由
D	Dynamically, 此路由是动态性地写入
M	Modified, 此路由是由路由守护程序或导向器动态修改
!	表示此路由当前为关闭状态

只有当它们都失败后才选择默认路由。默认路由, 以及下一站路由器发送的 ICMP 间接报文 (如果我们为数据报选择了错误的默认路由), 是 IP 路由选择机制中功能强大的特性。

其中 wlp3s0 是无线网卡的名称, 此命名规范为 v197², 说明此时的默认路由是无线网卡的路由。Destination 指定该路由属于哪个网络, Gateway 指定网络目标定义的地址集和子网掩码可以到达的前进或下一跃点 IP 地址, metric 设置路由跳数, metric 值越高, 优先级越低。添加 -p(permanent) 参数永久保存此条路由信息, 系统重启后路由也规则也不会丢失。flag 的含义如表20.1所示。

删除默认网关的命令如下:

```

1 # 在 Windows 下增加路由
2 route print
3 route -p add 10.55.10.0 mask 255.255.255.0 10.55.10.1
4 # 在 Fedora 里面执此命令删除默认网关
5 # Fedora 24
6 sudo route del default
7 # Ubuntu 16.04 LTS 中删除默认网关
8 # 先连接内网, 删除默认网关即可内外网同时连接
9 # 10.55.10.1 为内网默认网关
10 # enx000ec6c8c163 为本地有线网卡的名称
11 # dev 为随意的命名
12 sudo route del default gw 10.55.10.1 dev
    enx000ec6c8c163

```

²<https://www.freedesktop.org/wiki/Software/systemd/PredictableNetworkInterfaceNames/>

10.45.10.1 为内网的默认网关。Linux 系统的 `route` 命令用于显示和操作 IP 路由表 (show / manipulate the IP routing table)。要实现两个不同的子网之间的通信, 需要一台连接两个网络的路由器, 或者同时位于两个网络的网关来实现。删除一条路由:

```
1 sudo route del -net 192.168.122.0 netmask  
    255.255.255.0
```

如果默认路由是外网网关。那么只需要单独为内网设置转发特例, 所有 58.214.*.* 开头的, 全部走 `enp0s26u1u2`:

```
1 sudo route add -net 58.214.0.0 netmask 255.255.255.0  
    gw 10.55.10.1 dev enp0s26u1u2
```

其中 58.214.0.0 为内网 IP 的起始网段, 255.255.0.0 为内网的子网掩码, 内网网关是 10.36.40.12, `eth0` 为内网网卡的名称。路由添加最好是添加到开机启动中:

```
1 vim /etc/rc.local
```

```
1 #添加默认网关  
2 sudo route add default gw 10.55.10.1  
3 #删除默认网关  
4 sudo route del default gw 10.55.10.1
```

重启网络服务:

```
1 /etc/init.d/networking restart
```

此时可以同时访问互联网和局域网中的主机, 但是无法访问 A 网络。访问 A 网络可通过局域网代理的方式访问, 将局域网的一台主机作为代理服务器, 本机访问局域网中的代理服务器, 代理服务器再将请求转发给 A 网络。在本机只需要在浏览器中配置好代理服务器即可。以 FireFox 浏览器为例, 在 Preference->Advance->Network->Connection->Settings 中, 增加手动代理配置, 地址填写代理主机的地址, 端口填写 8888, 代理服务器运行的是 Fiddler。以后, 访问内网时就使用 FireFox 浏览器, 访问外网时就使用 Google Chrome 浏览器。

SSH 代理 (SSH Proxy)

数据库部署在 A 服务器 (58.214.215.*) 上, 而 A 服务器只能连接内网才能访问, 每次连接数据库都需要切换网络, 实在是非常不便。但是 localhost 可以在外网的情况下访问 B 机器, B 机器可以访问 A 机器。所以考虑通过 SSH 代理, 以 B 机器 (10.55.10.*) 为跳板, 直接访问数据库服务器 A。在服务器 B 上建立本地端口转发, 本地端口转发的格式是:

```
ssh -L <local port>:<remote host>:<remote port> <SSH  
hostname>
```

在 B 服务器 (10.55.10.77) 上建立一个本地端口转发:

```
ssh -L 5236:58.214.215.*:5236 10.55.10.77
```

-L 参数表示做本地映射端口。将本地机 (客户机) 的某个端口转发到远端指定机器的指定端口。工作原理是这样的, 本地机器上分配了一个 socket 侦听 port 端口, 一旦这个端口上有了连接, 该连接就经过安全通道转发出去, 同时远程主机和 host 的 hostport 端口建立连接。可以在配置文件中指定端口的转发。只有 root 才能转发特权端口。以上的语句只能在 B 服务器上使用, 如果其他机器想将流量发送到 B 服务器是无法利用隧道的。在主流 SSH 实现中, 本地端口转发绑定的是 loopback 接口, 这意味着只有 localhost 或者 127.0.0.1 才能使用本机的端口转发, 其他机器发起的连接只会得到 “connection refused.”。好在 SSH 同时提供了 GatewayPorts 关键字, 我们可以通过指定它与其他机器共享这个本地端口转发。

```
ssh -g -L 5236:58.214.215.*:5236 10.55.10.*
```

-g allows remote hosts to connect to local forwarded ports. If used on a multiplexed connection, then this option must be specified on the master process. 有时我们希望转发的不止一个端口, 那么可以用如下的命令多端口转发:

```
ssh -g -L 5236:58.214.215.*:5236 -L  
2222:58.214.215.*:22 -L  
3800:58.214.215.*:3800 10.55.10.77
```

在做端口转发时, 需要保证端口转发的机器 (10.55.10.77) 所使用的转发端口未被已经运行的程序占用。例如 10.55.10.77 已经运行了 sshd, 那么 22 端口就不能作为转发端口, 可以选择 1022 等任意未被使用的端口。在连接到 1022 端口时, 有可能提示 Connection Refused, 那么此时需要检查 1022 端口是否可以扫描到, 是否

打开，代理机器到防火墙是否允许 1022 端口到流量通过，以及代理机器 1022 的配置，是否已经做了到代理机器到免密登陆。最后发现改为 2222 端口即可。可以使用如下命令测试端口的联通：

```
1 # 采用 222 端口时，远程机器无法连接
2 # 暂时不清楚原因
3 telnet 10.77.10.55 2222
```

telnet 因为采用明文传送报文，安全性不好，很多 Linux 服务器都不开放 telnet 服务，而改用更安全的 ssh 方式。telnet 命令用于确定远程服务的状态，这里是确定远程服务器的某个端口是否能访问。以上配置在 Fedora 24 下工作良好。

Windows 代理设置

Windows 下还没有尝试直接通过 ssh 命令来设置代理转发，Windows 下的代理需要使用 putty 工具。假如在 A 机器代理，其他机器访问 A 机器的 2222 端口流量转发到 B 机器 (103.12.30.11) 的 22222 端口，那么 putty 可以作如图20.7所示的设置。

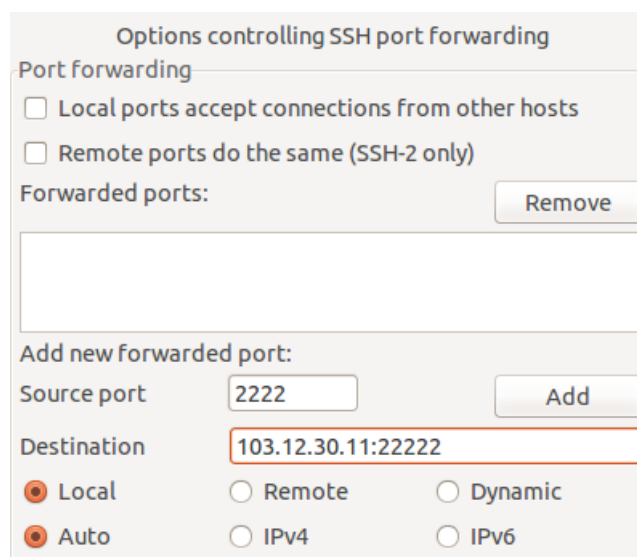


Figure 20.7: Windows 下设置 SSH 端口转发

当访问 A 机器的 2222 端口时，流量会自动转发到 B 机器的 22222 端口。在设置后，一段时间后会提示 network error software caused connection abort。在 putty 的 Connection 选项里，选择每隔一定事件发送空的数据包来保持连接 (Sending of null packets to keep session alive)。使用如下命令查看 Windows 端口是状态：

```
1 netstat -ano|findstr "3800"
```

设置好后，始终需要在前端运行一个界面。不是非常方便，所以可以让 putty 运行在后台。

SSH 反向代理 (Reverse Proxy)

A 网络有一台内网主机 AH(A Host)，可以连接互联网，有一台 A 网络的外网主机，无法连接互联网。此时我们可以使用 SSH 反向代理的方式，将 A 网络的外网主机端口映射到 AH 主机的端口。首先在 AH 主机上执行如下命令：

```
ssh -NfR 1234:localhost:2223 user1@123.123.123.123 -  
p2221
```

这句话的意思是将 AH 主机的 1234 端口和 B 主机的 2223 端口绑定，相当于远程端口映射 (Remote Port Forwarding)。

VI

Engineering

20.2 通用规范

Bibliography 235

Books

Articles

20.2 通用规范

20.2.1 版本管理

在项目的开发过程中，要求有明确的版本号，以前只是在正式发布版本的时候才会生成带版本号的文件，目前测试需要每次发布版本修改时定义版本号。以前直接使用脚本生成的带有年月日的版本标志，但是还是需要进一步规范化。由于发布采用的自动化脚本，在启动程序时需要指定明确的文件名称，文件名称与版本号有关联，所以考虑将版本号保存到一个文件里面，使用 Gradle 构建程序和发布程序时都读取此文件中的版本号。Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner
- PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format. 在项目的根目录下新建 version.properties 文件，文件中定义版本号：

```
1 VERSION_CODE=1.0.2
```

在 Gradle 中读取版本号：

```
1 version = getVersionCode()
2
3 def getVersionCode() {
4     def versionFile = file('/Users/dolphin/source/
5         credit-system/gradle/version.properties')
6     if (versionFile.canRead()) {
7         def Properties versionProps = new Properties
8         ()
9         versionProps.load(new FileInputStream(
10            versionFile))
11         def versionCode = versionProps['VERSION_CODE']
12         .toString()
13         /*def runTasks = gradle.startParameter.
14            taskNames //仅在assembleRelease任务是
15            增加版本号
16         if ('assembleRelease' in runTasks) {
```

```
11         versionProps['VERSION_CODE'] = (++
        versionCode).toString()
12         versionProps.store(versionFile.newWriter
        (), null)
13     }*/
14     return versionCode
15 } else {
16     throw new GradleException("Could not find
        version.properties!")
17 }
18 }
```

此处文件路径写的是绝对路径，如果仅仅在本机进行构建问题不大，但是一般都是团队协作，不同的成员使用的操作系统有区别，项目存放的路径也不一样，所以写成相对路径会比较妥当。可以写成如下：

```
1 File versionFile = file("$rootDir/cc-web-boot/src/
        main/resource/version.properties")
```

其中 `$rootDir` 表示项目的根目录，比如有一个叫做 `dolphin` 的项目，那么 `rootDir` 就是 `dolphin` 目录，另外还有一个 `projectDir` 的变量和 `buildDir` 的变量。`projectDir` 就是 `dolphin/src` 目录，`buildDir` 就是 `dolphin/src/build` 目录。如果是多个 `Project` 组成的项目，那么 `projectDir` 就是 `rootDir/projectName`，`buildDir` 就是 `dolphin/projectName/build`。

20.2.2 项目发布 (Publish)

在项目的版本管理上，可以引进 Alpha、Beta、Pre-release 和 Release 的机制。自动发布、灰度发布、自动回滚等。

撰写 Change Log

在项目管理平台 (禅道、Readmine 等) 上新建版本号，便于测试小组做测试。根据 `git log`，生成项目的修改记录，前提是 `git log` 需要有编写规范。或者手动编写项目的修改记录，一般为优化 (Optimize)、修改 (Modify)、新特性 (Feature)、bug 修复 (Bugfix) 等。

同步数据库 (Sync Database)

将涉及到本次发布的需要添加的字段、表等等在生产环境添加。这里就需要在开发时，记录修改数据库的 SQL 脚本，发布时执行脚本即可。

打标签 (Tag)

将此次发布的正式版本打上标记 (tag)，一般的命名是版本号加上特点，例如 1.1.35 版本的，便于以后的 Hotfix。

发布程序 (Publish)

需要注意的是，发布程序之前需要先备份原有的程序，便于回滚出现问题的版本。根据程序规模的大小，具体情况视主机的数量而定，使用脚本自动发布或者手动拷贝发布程序。

发布后确认

验证发布功能是否生效。在没有接口版本管理的情况下，告知调用接口方。

[2] [1]



Bibliography

Books

- [2] Michael F L'Annunziata. *Radioactivity: Introduction and History: Introduction and History*. Elsevier, 2007.

Articles

- [1] Dennis E. Discher et al. "Emerging applications of polymersomes in delivery: From molecular dynamics to shrinkage of tumors". In: *Progress in Polymer Science* 32.8–9 (2007). Polymers in Biomedical Applications, pp. 838 –857. ISSN: 0079-6700.

