码农 the code maker

20

云图

Docker 在 PHP 项目开发环境中的应用

曾金龙: 迅雷云的 Docker 开发实践

Kubernetes 的核心概念和核心组件

用 300 行代码创建一个分布式系统

Git 中级用户的 25 个提示

Apache Spark 入门攻略

优秀的叛逆者，卓越的工作

张磊: 关于 Docker、开源，以及教育的尝试

Kubernetes 负责人 Dawn Chen: 开源是唯一的路

iTuring.cn

目　录

编者的话

专题：Docker云图

1　Docker在PHP项目开发环境中的应用

13　《Docker开发实践》作者曾金龙：迅雷云的Docker开发实践

20　Kubernetes的核心概念和核心组件

37　Mesos+Docker+Go，用300行代码创建一个分布式系统

44　VLIS实验室云计算组张磊：关于Docker、开源，以及教育的尝试

53　阅读Docker源代码的神兵利器

人物

67　Kubernetes负责人Dawn Chen：开源是唯一的路

动手

77　Git中级用户的25个提示

践行

95　Apache Spark入门攻略

鲜阅

118　优秀的叛逆者，卓越的工作

书单

126　看看大家都在读什么？
129　电子书单

成书手记

130　《Clojure经典实例》：编程语言习得

妙评

132　《项目百态》：实话的力量

编者的话

开辟未来的Docker

编者 / 李盼

Docker刚发布时，作为一个单机版轻量级虚拟化工具，并没有像当前一样发达的生态系统。但是在随后的一年中，Docker从一个开发运维人员略有耳闻的工具成为一个技术圈里家喻户晓的名词。Docker对于IT行业的价值也从节省资源这一方面扩展到对整个软件开发运维生命周期的改造。

拿来主义对于懒惰而又聪明的程序员来说是件好事，这样他们就可以专注于本该干的活。对于测试人员和运维人员来说，也是如此，没人喜欢处理那些本不该重点关注、处理不好却会让人寸步难行的问题。Docker就像一个打包器，可以把你的应用及其环境整体打包，然后很方便地迁移到不同的平台，到处运行。如此接地气的技术，怪不得业内都惊呼Docker是下一个Hadoop。

如今，各大知名技术社区都为Docker开辟了专栏，甚至出现了专为讨论Docker而生的技术社区。基于Docker的中国本土化也开始萌芽，各类国内镜像托管和加速服务层出不穷。Docker生态系统的建立已经是不争的事实，越来越多的国内团队从Docker的使用者，成为了Docker、Kubernetes等开源项目的维护者和贡献者。

在本期《码农》中我们不仅讨论了常见而必备的Docker实践知识，还采访了Docker生态系统的深度使用者、参与者，甚至是创造者。这

里面有改造Docker，化为己用的迅雷云工程师曾金龙；有第一批使用Docker，如今成为了该项目维护者的科研人员张磊；最后，我们还请到了Kubernetes项目负责人之一Dawn Chen，她不仅向我们介绍了这个项目背后的设计理念，还讲述了她作为一位女工程师的精彩人生。

以Docker为代表的容器技术让所有人眼前一亮，它模糊了IaaS与PaaS之间的界限，为云计算的服务形式带来了一个开放的未来。正是因为有了这样的技术，才使得各种各样的探索成为可能。■

专题：Docker 云图

Docker在PHP项目开发环境中的应用

作者 / 徐谦

华尔街见闻CTO，开源爱好者，关注PHP及Web开发，曾向多个知名PHP开源项目提交代码。译有《自制编程语言》、《游戏开发的数学和物理》等。

环境部署是所有团队都必须面对的问题，随着系统越来越大，依赖的服务也越来越多，比如我们目前的一个项目就会用到：

- Web服务器：Nginx
- Web程序：PHP + Node
- 数据库：MySQL
- 搜索引擎：ElasticSearch
- 队列服务：Gearman
- 缓存服务：Redis + Memcache
- 前端构建工具：npm + bower + gulp
- PHP CLI工具：Composer + PHPUnit

因此团队的开发环境部署随之暴露出若干问题：

1. 依赖服务很多，本地搭建一套环境成本越来越高，初级人员很难解决环境部署中的一些问题
2. 服务的版本差异及OS的差异都可能导致线上环境BUG
3. 项目引入新的服务时所有人的环境需要重新配置

对于问题1，可以用Vagrant这样的基于虚拟机的项目来解决，团队成员共享一套开发环境镜像。对于问题2，可以引入类似PHPBrew这样的多版本PHP管理工具来解决。但两者都不能很好地解决问题3，因为虚

拟机镜像没有版本管理的概念，当多人维护一个镜像时，很容易出现配置遗漏或者冲突，一个很大的镜像传输起来也不方便。

Docker的出现让上面的问题有了更好的解决方案，虽然个人对于Docker大规模应用到生产环境还持谨慎态度，但如果仅仅考虑测试及开发，私以为Docker的容器化理念已经是能真正解决环境部署问题的银弹了。

下面介绍Docker构建PHP项目开发环境过程中的演进，本文中假设你的操作系统为Linux，已经安装了Docker，并且已经了解Docker是什么，以及Docker命令行的基础使用，如果没有这些背景知识建议先自行了解。

Hello World

首先还是从一个PHP在Docker容器下的Hello World实例开始。我们准备这样一个PHP文件`index.php`:

```
<?php
echo "PHP in Docker";
```

然后在同目录下创建文本文件并命名为`Dockerfile`，内容为:

```
# 从官方PHP镜像构建
FROM        php

# 将index.php复制到容器内的/var/www目录下
ADD         index.php /var/www

# 对外暴露8080端口
EXPOSE      8080

# 设置容器默认工作目录为/var/www
WORKDIR     /var/www

# 容器运行后默认执行的指令
ENTRYPOINT ["php", "-S", "0.0.0.0:8080"]
```

构建这个容器:

```
docker build -t allovince/php-helloworld .
```

运行这个容器

```
docker run -d -p 8080:8080 allovince/php-helloworld
```

查看结果:

```
curl localhost:8080
PHP in Docker
```

这样我们就创建了一个用于演示PHP程序的Docker容器，任何安装过Docker的机器都可以运行这个容器获得同样的结果。而任何有上面的php文件和Dockerfile的人都可以构建出相同的容器，从而完全消除了不同环境，不同版本可能引起的各种问题。

想象一下程序进一步复杂，我们应该如何扩展呢，很直接的想法是继续在容器内安装其他用到的服务，并将所有服务运行起来，那么我们的Dockerfile很可能发展成这个样子:

```
FROM        php
ADD         index.php /var/www

# 安装更多服务
RUN         apt-get install -y \
            mysql-server \
            nginx \
            php5-fpm \
            php5-mysql

# 编写一个启动脚本启动所有服务
ENTRYPOINT ["/opt/bin/php-nginx-mysql-start.sh"]
```

虽然我们通过Docker构建了一个开发环境，但觉不觉得有些似曾相识呢。没错，其实这种做法和制作一个虚拟机镜像是差不多的，这种方式存在几个问题：

- 如果需要验证某个服务的不同版本，比如测试PHP5.3/5.4/5.5/5.6，就必须准备4个镜像，但其实每个镜像只有很小的差异。
- 如果开始新的项目，那么容器内安装的服务会不断膨胀，最终无法弄清楚哪个服务是属于哪个项目的。

使用单一进程容器

上面这种将所有服务放在一个容器内的模式有个形象的非官方称呼：Fat Container。与之相对的是将服务分拆到容器的模式。从Docker的设计可以看到，构建镜像的过程中可以指定唯一一个容器启动的指令，因此Docker天然适合一个容器只运行一种服务，而这也是官方更推崇的。

分拆服务遇到的第一个问题就是，我们每一个服务的基础镜像从哪里来？这里有两个选项：

选项一、统一从标准的OS镜像扩展，比如下面分别是Nginx和MySQL镜像

```
FROM ubuntu:14.04
RUN  apt-get update -y && apt-get install -y nginx
```

```
FROM ubuntu:14.04
RUN  apt-get update -y && apt-get install -y mysql
```

这种方式的优点在于所有服务可以有一个统一的基础镜像，对镜像进行扩展和修改时可以使用同样的方式，比如选择了ubuntu，就可以使用apt-get指令安装服务。

问题在于大量的服务需要自己维护，特别是有时候需要某个服务的不同版本时，往往需要直接编译源码，调试维护成本都很高。

选项二、直接从Docker Hub继承官方镜像，下面同样是Nginx和MySQL镜像

```
FROM nginx:1.9.0
```

```
FROM mysql:5.6
```

Docker Hub可以看做是Docker的Github，Docker官方已经准备好了大量常用服务的镜像，同时也有非常多第三方提交的镜像。甚至可以基于Docker-Registry项目在短时间内自己搭建一个私有的Docker Hub。

基于某个服务的官方镜像去构建镜像，有非常丰富的选择，并且可以以很小的代价切换服务的版本。这种方式的问题在于官方镜像的构建方式多种多样，进行扩展时需要先了解原镜像的Dockerfile。

出于让服务搭建更灵活的考虑，我们选择后者构建镜像。

为了分拆服务，现在我们的目录变为如下所示结构：

```
~/Dockerfiles
├── mysql
│   └── Dockerfile
├── nginx
│   ├── Dockerfile
│   ├── nginx.conf
│   └── sites-enabled
│       ├── default.conf
│       └── evaengine.conf
```

```
├── php
│   ├── Dockerfile
│   ├── composer.phar
│   ├── php-fpm.conf
│   ├── php.ini
│   └── redis.tgz
└── redis
    └── Dockerfile
```

即为每个服务创建单独文件夹，并在每个服务文件夹下放一个Dockerfile。

MySQL容器

MySQL继承自官方的MySQL5.6镜像，Dockerfile仅有一行，无需做任何额外处理，因为普通需求官方都已经在镜像中实现了，因此Dockerfile的内容为：

```
FROM mysql:5.6
```

在项目根目录下运行

```
docker build -t eva/mysql ./mysql
```

会自动下载并构建镜像，这里我们将其命名为eva/mysql。

由于容器运行结束时会丢弃所有数据库数据，为了不用每次都要导入数据，我们将采用挂载的方式持久化MySQL数据库，官方镜像默认将数据库存放在/var/lib/mysql，同时要求容器运行时必须通过环境变量设置一个管理员密码，因此可以使用以下指令运行容器：

```
docker run -p 3306:3306 -v ~/opt/data/mysql:/var/lib/mysql -e MYSQL_ROOT_
PASSWORD=123456 -it eva/mysql
```

通过上面的指令，我们将本地的3306端口绑定到容器的3306端口，将容器内的数据库持久化到本地的~/opt/data/mysql，并且为MySQL设置了一个root密码123456。

Nginx容器

Nginx目录下提前准备了Nginx配置文件nginx.conf以及项目的配置文件default.conf等。Dockerfile内容为：

```
FROM nginx:1.9

ADD  nginx.conf      /etc/nginx/nginx.conf
ADD  sites-enabled/*    /etc/nginx/conf.d/
RUN  mkdir /opt/htdocs && mkdir /opt/log && mkdir /opt/log/nginx
RUN  chown -R www-data.www-data /opt/htdocs /opt/log

VOLUME ["/opt"]
```

由于官方的Nginx1.9是基于Debian Jessie的，因此首先将准备好的配置文件复制到指定位置，替换镜像内的配置，这里按照个人习惯，约定/opt/htdocs目录为Web服务器根目录，/opt/log/nginx目录为Nginx的Log目录。

同样构建一下镜像

```
docker build -t eva/nginx ./nginx
```

并运行容器

```
docker run -p 80:80 -v ~/opt:/opt -it eva/nginx
```

注意我们将本地的80端口绑定到容器的80端口，并将本地的~/opt目录挂载到容器的/opt目录，这样就可以将项目源代码放在~/opt目录下并通过容器访问了。

PHP容器

PHP容器是最复杂的一个，因为在实际项目中，我们很可能需要单独安装一些PHP扩展，并用到一些命令行工具，这里我们以Redis扩展以及Composer来举例。首先将项目需要的扩展等文件提前下载到php目录下，这样构建时就可以从本地复制而无需每次通过网络下载，大大加快镜像构建的速度：

```
wget https://getcomposer.org/composer.phar -O php/composer.phar
wget https://pecl.php.net/get/redis-2.2.7.tgz -O php/redis.tgz
```

php目录下还准备好了php配置文件php.ini以及php-fpm.conf，基础镜像我们选择的是PHP 5.6-FPM，这同样是一个Debian Jessie镜像。官方比较亲切的在镜像内部准备了一个docker-php-ext-install指令，可以快速安装如GD、PDO等常用扩展。所有支持的扩展名称可以通过在容器内运行docker-php-ext-install获得。

来看一下Dockerfile

```
FROM php:5.6-fpm

ADD php.ini    /usr/local/etc/php/php.ini
ADD php-fpm.conf    /usr/local/etc/php-fpm.conf

COPY redis.tgz /home/redis.tgz
RUN docker-php-ext-install gd \
    && docker-php-ext-install pdo_mysql \
    && pecl install /home/redis.tgz && echo "extension=redis.so" > /usr/local/etc/php/
conf.d/redis.ini
ADD composer.phar /usr/local/bin/composer
RUN chmod 755 /usr/local/bin/composer

WORKDIR /opt
RUN usermod -u 1000 www-data

VOLUME ["/opt"]
```

在构建过程中做了这样一些事情：

1. 复制php和php-fpm配置文件到相应目录
2. 复制redis扩展源代码到/home
3. 通过docker-php-ext-install安装GD和PDO扩展
4. 通过pecl安装Redis扩展
5. 复制composer到镜像作为全局指令

按照个人习惯，仍然设置/opt目录作为工作目录。

这里有一个细节，在复制tar包文件时，使用的Docker指令是COPY而不是ADD，这是由于ADD指令会自动解压tar文件。

现在终于可以构建+运行了：

```
docker build -t eva/php ./php
docker run -p 9000:9000 -v ~/opt:/opt -it eva/php
```

在大多数情况下，Nginx和PHP所读取的项目源代码都是同一份，因此这里同样挂载本地的~/opt目录，并且绑定9000端口。

PHP-CLI的实现

php容器除了运行php-fpm外，还应该作为项目的php cli使用，这样才能保证php版本、扩展以及配置文件保持一致。

例如在容器内运行Composer，可以通过下面的指令实现：

```
docker run -v $(pwd -P):/opt -it eva/php composer install --dev -vvv
```

这样在任意目录下运行这行指令，等于动态将当前目录挂载到容器的默认工作目录并运行，这也是PHP容器指定工作目录为/opt的原因。

同理还可以实现phpunit、npm、gulp等命令行工具在容器内运行。

Redis容器

为了方便演示，Redis仅仅作为缓存使用，没有持久化需求，因此Dockerfile仅有一行

```
FROM redis:3.0
```

容器的连接

上面已经将原本在一个容器中运行的服务分拆到多个容器，每个容器只运行单一服务。这样一来容器之间需要能互相通信。Docker容器间通讯的方法有两种，一种是像上文这样将容器端口绑定到一个本地端口，通过端口通讯。另一种则是通过Docker提供的Linking功能，在开发环境下，通过Linking通信更加灵活，也能避免端口占用引起的一些问题，比如可以通过下面的方式将Nginx和PHP链接起来：

```
docker run -p 9000:9000 -v ~/opt:/opt --name php -it eva/php
docker run -p 80:80 -v ~/opt:/opt -it --link php:php eva/nginx
```

在一般的PHP项目中，Nginx需要链接PHP，而PHP又需要链接MySQL，Redis等。为了让容器间互相链接更加容易管理，Docker官方推荐使用Docker-Compose完成这些操作。

用一行指令完成安装

```
pip install -U docker-compose
```

然后在Docker项目的根目录下准备一个docker-compose.yml文件，内容为：

```
nginx:
    build: ./nginx
    ports:
      - "80:80"
    links:
      - "php"
    volumes:
      - ~/opt:/opt

php:
    build: ./php
    ports:
      - "9000:9000"
    links:
      - "mysql"
      - "redis"
    volumes:
      - ~/opt:/opt

mysql:
    build: ./mysql
    ports:
      - "3306:3306"
    volumes:
      - ~/opt/data/mysql:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: 123456

redis:
    build: ./redis
    ports:
      - "6379:6379"
```

然后运行 docker-compose up，就完成了所有的端口绑定、挂载、链接操作。

更复杂的实例

上面是一个标准PHP项目在Docker环境下的演进过程，实际项目中一般会集成更多更复杂的服务，但上述基本步骤仍然可以通用。比如EvaEngine/Dockerfiles是为了运行我的开源项目EvaEngine准备的基于Docker的开发环境，EvaEngine依赖了队列服务Gearman，缓存服务Memcache、Redis，前端构建工具Gulp、Bower，后端Cli工具Composer、PHPUnit等。具体实现方式可以自行阅读代码。

经过团队实践，原本大概需要1天时间的环境安装，切换到Docker后只需要运行10余条指令，时间也大幅缩短到3小时以内（大部分时间是在等待下载），最重要的是Docker所构建的环境都是100%一致的，不会有人为失误引起的问题。未来我们会进一步将Docker应用到CI以及生产环境中。■

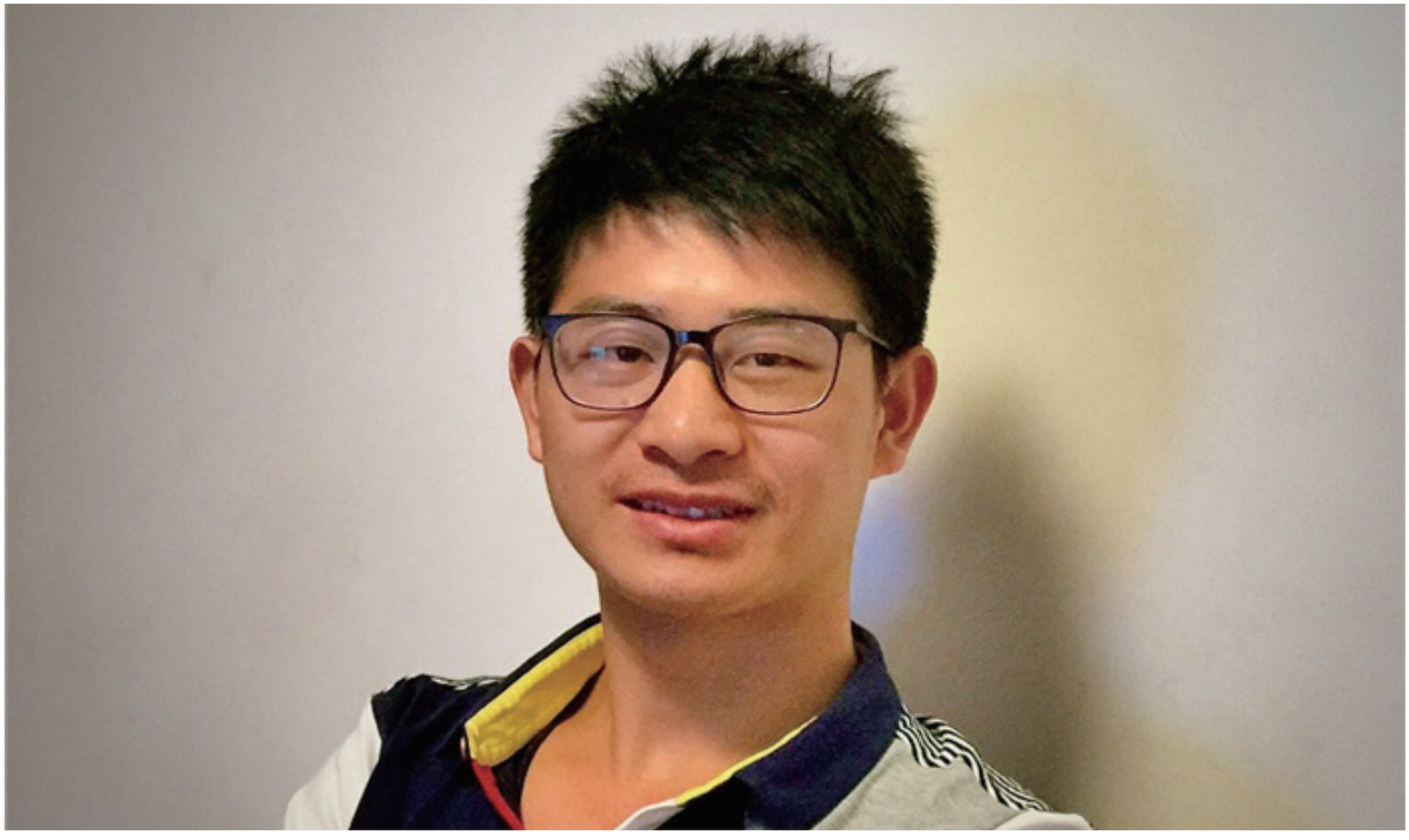
阅读卧龙阁专栏

阅读图灵社区原文

专题：Docker 云图

《Docker 开发实践》作者曾金龙：

迅雷云的 Docker 开发实践



曾金龙就职于迅雷网络，是国内覆盖面最广的“迅雷 P2P 引擎”核心研发成员。他毕业于中山大学，具有计算机科学硕士学位，他的研究方向是 P2P 网络、音视频传输和 CEP 系统。曾金龙对 Docker 技术有着深入的理解，是国内较早将 Docker 引入到实际软件开发、测试和部署中的人。他在 2015 年出版了《Docker 开发实践》一书。

问：作为迅雷P2P引擎的核心开发成员，能否谈一下当时开发的经历以及克服了哪些困难？

我一开始工作的时候就进入到了一家在P2P领域NO.1的公司，而且还是核心团队，所以压力非常大。我发现，现实的P2P依然处在第一代P2P的水平，和研究上的P2P有很大差异。我们在高校里面研究的是各种结构化的P2P，叫做第二代P2P。这就意味着即便我是以P2P为研究方向来到了迅雷，但一切都是归零的。这是理想与现实之间的差异，对我个人而言是个困难。

开发上的困难有两个方面，一是新做的项目往往配置基础环境会耗费大量的时间，而且开发不会碰到的问题在测试和运营那里会“莫名其妙”地出现，所以容器是解决程序员痛点的好技术。另外就是迅雷云是全国最大的P2P数据平台，所以里面有非常多分工不同的服务器，有打洞的Punch服务、有索引的Hub服务、跟踪的Tracker服务以及CDN管理服务等，而且处理的数据量非常大。让公司各个事业群的项目组合理管理和使用这些集群确实不容易，而大多数问题都来自不正确的使用，所以我们必须为各个事业群提供更为易用的服务和更加统一的接口。所以我们后面在迅雷云引入了Docker。

问：能否谈一下迅雷云使用Docker的过程？

其实最初的时候，迅雷团队对Docker是怀有谨慎的态度的。不能因为别人说好我们就用，需要先试，真的好我们才敢用，尝到甜头我们才愿意用。一开始我们用Docker来加快团队之间的协作效率，因为通过它部署的环境可以一次构建到处使用，所以开发、测试和部署都能够在一个非常一致的环境中进行，这是很初级的用法。随着对这个模式的认可，迅雷内部越来越多的团队接纳了我们的建议，于是迅雷、迅雷看看、迅雷游戏都采用了Docker，引入到了开发、测试和部署中。

后面我们的服务器项目基本就迁移到了Docker之上。当然，一开始大家对Docker的安全性都会有所顾虑，但其实大多数顾虑都是因为对

Docker不够理解而产生的。有了内部产品线的支持，我们后面把迅雷云部分服务器资源用Docker重新部署。迅雷云本身就是一个非常高效的集群系统，现在引入Docker是为了让资源使用更加便捷。因为研发人员的流动性比较高，不是所有的开发者对迅雷云内部资源的使用都熟悉，所以我们现在就是借助Docker去为各个业务的开发提供一套更普适的接口。

问：迅雷云在容器方面做了哪些优化和调整？

一是整合现有的集群，将部分集群更改为Docker集群，特别是新增部分。二、由于迅雷云是一个重数据、轻业务的云，所以在数据管理方面，像Docker的Data volumes和Data volume containers这种简单的方式还是不能够满足，所以我们主要还是以迅雷云自己的管理系统为主。为了能够让Docker之上的业务更好地使用我们的集群，我们添加了适配层。三、调度算法是迅雷云定制优化的。最后，我们有自建的私有仓库，提供常用镜像，让我们的开发更加快速。

问：Adrian Cockcroft是云技术社区中的思想家之一，他将微服务与Docker的结合视为一种颠覆，认为这种开发方式会极大提高开发团队敏捷性和适应性。请问迅雷云团队是否有这方面的实践？对于这方面的技术融合你有什么样的建议？

微服务是把一个原本很复杂的服务解耦拆解为一些相对功能单一的服务，这在迅雷云内部是一直都非常提倡的。我们认为一个服务专心做好一件事是最好的，这对解耦、易用、升级都非常有好处。迅雷内部把Hub服务和Tracker服务都进行了分解和抽象，提供给各个项目组。Hub本身是很复杂的，但我们的服务接口却是个位数，包括对CDN资源服务的使用方面我们也是重构了一套更加简单易用的服务。

有一点很遗憾的是我觉得迅雷云里面做得最好的微服务并没有使用到Docker，也就是我们的“水晶计划”。这主要是因为我们的节点是各种客户端设备，但我们的设计宗旨依然是遵循微服务的。水晶计划的节点是各种各样的设备，差异性非常大。但没有关系，在用户那里，它就是

可靠的CDN。我个人觉得微服务加上Docker是非常合适的，因为微服务很重要的一点就是解耦，而采用容器技术来解耦隔离是非常恰当的。再进一步考虑到服务的扩容和迁移的话，微服务设计加上Docker作为载体也是非常合适的。我的建议就是，一个服务一个容器，或者说一种服务，一类容器，这样一一对应地去部署。

问：Docker的image和安全问题一直都受到关注，特别在去年的出现的ShellShock和Heartbleed问题之后，请问在Docker的安全方面你是否有哪些经验可以分享？

其实，越多人使用的技术越安全，但一旦不安全造成的损失也非常大。迅雷在使用Docker这方面一直没有把数据层面交给迅雷云之外的系统。即便是Docker使用迅雷云的数据，也是通过我们的适配层，而通过适配层接进来的操作是要经过授权的，对使用能力也要进行限制，同时操作都是可跟踪和统计的。另外就是对用户的授权管理，严格规范的管理在安全方面很有必要。还有就是做好隔离工作，不仅仅是通过内核的Namespace/cgroup，同时也可以在不同层面隔离，比如VM和容器的隔离等。

问：在ClusterHQ最近的一项调查中显示，只有38%的IT专业人士在生产环境中使用容器，人们对于容器技术使用的顾虑包括安全、数据管理、IT人员技能、永久存储与可靠性等问题。你认为以Docker为代表的容器技术面临的最大阻碍是什么？是否在不远的未来有望跨越技术应用的鸿沟？

我觉得当前容器技术面临的最大阻力是安全问题和管理工具。安全性一直是企业使用容器所担心的问题，而另一个问题就是管理工具，特别是对集群的管理。现有的集群管理工具，比如Kubernetes，并不是很好上手，对于很多运维人员而言，特别是对一些小企业来说，运维人员可能并没有深入地去学习这些工具怎么使用。所以如果这些工具的部署配置和使用能够非常简单，那么容器在实际生产中可能会越来越广泛。

这种情况有些类似于ISO/OSI的参考模型和TCP/IP协议模型，前者从设计上而言是很优秀的，但后者却成了事实标准，为什么？因为简单。所以一门好技术，不但要技术本身好，而且要做到让别人更好地接纳也很重要。Docker相对于以前的诸如lxc的系统是一个进步，我希望Docker能继续做好整个工具链，让Docker集群更加平易近人。

问：Docker, CoreOS, 以及其他业界成员一起创立了“开放容器计划（OCP)”，你认为OCP的成立对于软件开发行业会造成什么影响?

Docker和CoreOS这对竞争对手握手言和，一起成立OCP，对于整个容器生态圈而言是一个非常重大的利好。因为OCP可以让容器技术更加统一化，如果所有的容器都遵循一套标准，那么使用容器的开发者就省了很多麻烦。而且有了统一的标准，就意味着在不同平台上的容器也可以相互迁移，Docker的镜像可以迁移到CoreOS上，反之亦然。破除各自为阵，容器技术会变得更加易用，同时也会加快其在产业上的应用。

问：Docker和CoreOS各自都具有哪些优势？它们的结合会带来哪些好处?

Docker更想去做一个以容器为中心的服务平台，所以它给用户提供了很多便利的操作，还包括一些集群管理工具链。CoreOS则更加侧重容器本身的标准化，CoreOS的开放标准更能够促进容器的开放，这也是OCP的宗旨，此外，CoreOS在安全性和可组合性上也做得更好。它们的结合可以取二者优势，弥补各自不足。

问：Docker最新发布的“Docker试验性发布”是一次有趣的尝试，请问你是否使用过上面的实验性功能？你觉得这个系统的发布对于Docker来说有哪些好处?

如果你指的是out-of-process volume plugins，我有了解但没有使用过。

在对数据方面采用插件的形式，让用户自己去适配不同的存储系统是非常有必要的，所以我觉得这个功能非常有用。我在上面也提到了，我们迅雷云就写了一套适配层去操作我们自己的数据系统。所以后面我们团队会去深入地理解这个功能，然后跟进。

问：学习 Docker 的进阶过程有哪几个阶段？你希望程序员们以怎样的方式使用《Docker 开发实践》这本书？

我觉得学习 Docker 有三个阶段。第一个阶段是对 Docker 本身功能的使用，也就是入门层，你要理解什么是镜像、容器、Registry、Dockerfile 等，了解这些对象的基本操作，能够构建出自己的应用环境来。第二个阶段是能够驾驭 Docker 集群。这个阶段你需要学习和使用 Docker 相关的工具，比如 Kubernetes、Shipyard、Machine+Swarm+Compose 等，根据不同的需求，需要使用不同的工具。第三个阶段就是发现和解决这些工具里面的问题，为自己的场景深度定制。

我觉得程序员可以从两个方面来使用好《Docker 开发实践》这本书，一方面可以把它当做开发手册，因为这本书的内容几乎涵盖了 Docker 所有的知识点，我们对其内容进行了很系统地梳理。所以当你对 Docker 有什么疑问的时候，把《Docker 开发实践》放在电脑前面，随手翻一翻基本就可以解决。另一方面，可以作为案例引导。这本书后面两篇甄选了当前经典的应用场景，一步步带领读者把服务搭建起来，并且把可能遇到的问题都标出来，教读者如何解决。所以，读者也可以参照本书的案例去更好更快地在实际生产中使用 Docker，提升效率、少走弯路。

问：《Docker 开发实践》的高级篇对 Docker 生态圈做了非常翔实的介绍和实践，为什么要对 Docker 生态圈着墨如此之多？对于这部分内容的遴选你有什么样的标准和原则？

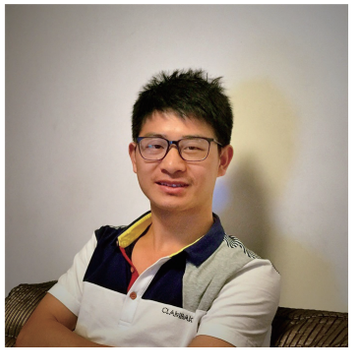
Docker 本身的操作其实很简单，如果复杂的话，Docker 就不会像今天这样成功。然而我所知道的市面上之前所有关于 Docker 的书籍，都是

把基础操作写了一大篇，然后堆砌一些案例，就成了一本书，我认为这是在浪费读者的钱和时间。基础的东西写成几章是合理的，写成一本书就是打肿脸充胖子了。我们在《Docker 开发实践》中把这些内容进行了压缩，不想去滥竽充数，也不会去浪费笔墨。

上面我提到的Docker面临的阻碍很大一点就是它的工具链不完善和不易用。你会发现在网络上程序员求助最多的也是Docker生态圈的实践，因为Docker做得还不够好，稍微不小心就犯错。所以，既然基础篇不该讲那么多，而且生态圈又是使用Docker的开发者心中的痛点，那么我就着重笔墨去写这部分，让大家少走弯路。这部分内容看上去有些散，每一章都是讲一个工具或者是一组工具，但是书中选的Kubernetes、Shipyard等都是我们最需要的和最热点的工具，而且它们的构建又相对复杂。所以，我遴选的原则就是，重要又是难点的，我就选。所以这就成了《Docker 开发实践》中的高级篇。■

专题：Docker 云图

Kubernetes的核心概念和核心组件



作者 / 曾金龙

曾金龙就职于迅雷网络，是国内覆盖面最广的“迅雷P2P引擎”核心研发成员。他毕业于中山大学，具有计算机科学硕士学位，他的研究方向是P2P网络、音视频传输和CEP系统。曾金龙对Docker技术有着深入的理解，是国内较早将Docker引入到实际软件开发、测试和部署中的人。他在2015年出版了《Docker开发实践》一书。

Kubernetes 简介

Kubernetes是Google公司开源的大规模容器集群管理系统。利用Kubernetes，能方便地管理跨 机器运行的容器化应用。它为容器化应用提供资源调度、部署、服务发现、扩展机制等功能，具 体如下：

- 使用Docker对应用进行打包、实例化和运行；
- 以集群的方式运行和管理跨主机的容器；
- 解决跨主机容器的通信问题；
- 提供自我修复功能，保证系统运行的健壮性。

Kubernetes目前处于快速迭代开发之中，几乎每周都会推出新的版本。作为一个容器管理框 架，Kubernetes可以被部署在物理集群和各类云环境中，例如GCE、vSphere、CoreOS、OpenShift、Azure等。

接下来，我们将会从Kubernetes的核心概念、设计框架等入手，让读者更好地理解它。等有 了相关基础之后，我们将会在CentOS 7上实验Kubernetes。

核心概念

Kubernetes的核心概念包含节点 (Node)、Pod、服务 (Service)、备份控制器 (Replication Controller)、卷 (Volume) 和标签 (Label)。

鉴于备份控制器和卷比较简要，这里不再详细介绍，后面介绍架构时一起说明。

节点

节点是Kubernetes系统中的一台工作机器，常被称为Minion，即从属主机。它可以是物理机，也可以是虚拟机。每一个节点都包含了Pod运行所需的必要服务，例如Docker、kubelet和网络代 理（proxy）。节点受Kubernetes系统中的主节点控制。和Pod、服务不一样，节点本身并不属于 Kubernetes的概念，它是云平台中的虚拟机或者实体机。所以，当一个节点加入到Kubernetes系统 中时，它将会创建一个数据结构来记录该节点的信息。另外，不是所有节点都能够加入到 Kubernetes系统中的，只有那些通过验证的节点才能够成为Kubernetes节点。

目前，节点的管理有两种方式：节点管理器（Node Controller）和通过命令手动管理。

- 节点管理器。它是Kubernetes主控节点上管理集群节点的组件，主要包含两个功能：集群 节点的同步和单个节点生命周期的管理。当有节点加入到Kubernetes中时，节点管理器将 会创建节点信息；当有节点需要从Kubernetes中删除时，节点管理器则会删除该节点的节 点信息。需要注意的是，节点管理器并不会真正创建节点本身，而仅仅创建节点的元数 据，用于跟踪节点的状态。所以，节点上的服务需要用户自己安装。单个节点的生命周 期的管理目前尚在开发之中。
- 手动管理节点。Kubernetes的管理员可以通过kubectl命令来管理节点。和节点管理器一样，使用kubectl命令创建和删除节点时，也只是删除节点的配置信息。

Pod

在Kubernetes中，Pod是最小的可创建、调度和管理的部署单元。它是容器化环境中的“逻辑 主机”，可以包含一个或多个有关联的容器，并

且容器之间可以共享数据卷。例如，一个Web站 点应用由前端、后端和数据库组成，这三个组件运行在各自的容器中，我们可以创建包含这三个 容器的Pod。

可以看出，容器存在于Pod之中，而Pod又存在于节点之中。那么，为什么需要抽象出Pod这 个概念呢？下面从资源共享和通信、管理这两个方面介绍一下。

- 资源共享和通信。同一Pod中的容器拥有相同的网络命名空间、IP地址和端口区间，它们 之间可以直接用localhost来发现和通信。在无层次的共享网络中，每个Pod都有一个IP地 址，用于跟其他物理主机和容器进行通信，Pod的名字也被用作主机名。此外，同一Pod 的容器可以共享数据卷。在将来，Pod内的容器还可以共享IPC命名空间、CPU和内存等。
- 管理。从管理的角度来看，Pod比容器站在更高的层面，它简化了应用的部署和管理。Pod 可以自动处理主机托管、资源共享、协调复制和依赖管理等问题。

虽然可以将Pod用在垂直依赖的应用栈中，但它更适合用在多个应用的横向协作部署中，为 多个容器提供集中的辅助功能。具体的使用用例有：

- 内容管理系统、文件和数据的装载和本地缓存管理等；
- 日志和检出点备份、压缩、轮换和快照；
- 数据变更监控、日志末端数据读取、日志和监控适配器和事件打印；
- 代理、桥接和适配器；
- 控制器、管理器、配置编辑和更新。

为什么不在一个容器中直接运行多个应用而采用在一个Pod中运行多个容器呢？主要原因如 下所示。

- 透明性。底层系统可以获取到Pod内的容器，这样底层系统就可以为容器提供诸如进程管 理和资源监控等服务，这会给用户带来不少便利。

- 解耦软件依赖。分为多个容器后，每个容器都可以单独存在，并且当其中某个应用需要 升级时，只影响到一个容器。后续Pod将会支持单个容器的在线升级。
- 易用性。用户不需要使用自己的进程管理程序，直接用Docker管理容器即可。此外，用 户也不用再担心信号量和退出码的传递等问题。
- 高效性。因为底层系统提供了更多的管理，这使得容器更加轻便。

服务

Kubernetes的服务是一系列Pod以及这些Pod的访问策略的抽象。

Kubernetes中的Pod是具有时效性的，它会随着时间而变化。虽然每个Pod都有一个单独的IP地 址，但是该IP地址却不是静态不变的。例如，当系统触发RepliController对Pod进行备份时，Pod的地 址很可能会发生改变。这将会导致一个问题。我们试想，在Kubernetes系统中，有一群Pod作为后端 给前端提供服务，如果后端的IP地址是变动的，那么前端又如何去发现和使用后端的服务呢?

Kubernetes的服务也叫作微服务 (micro-service)，它用于定义一系列Pod的逻辑关系以及它们 的访问规则。服务的目标是为了隔绝前端和后端的耦合性，让前端透明地使用该项服务，而不需 要知道该项服务具体由哪些后台机器提供。Kubernetes会为一个服务分配一对，该IP和端口并不 是真实的地址和端口，而是一个虚拟IP，当前端通过该对访问服务时，服务代理将会将请求重定 向到合适的后端机器。

1. 定义服务

Kubernetes 中 的 服 务 是 一 个 REST (Representational State Transfer，表述性状态转移) 对象。 下面演示了一个服务的定义:

```
{
    "id": "myapp",
```

```
    "selector": {
    "app": "MyApp"
    },
    "containerPort": 9376,
    "protocol": "TCP",
    "port": 8765
}
```

上述示例定义了一个id为myapp的服务，它通过选择器选择那些带有app=MyApp标签的Pod作为 服务的提供者。所有被选择的Pod都将9376端口暴露，用于监听该项服务的请求。前端客户可以 通过$MYAPP*SERVICE*PORT和$MYAPP*SERVICE*HOST来访问该项服务。

2. 工作原理

在Kubernetes中，每个节点都运行着一个服务代理 (service proxy)，它监控来自Kubernetes 主控节点的消息，主控节点会向其传递诸如添加和删除服务以及服务的端点列表等信息。服务代 理维护着一个映射表，表中每一项是服务和该服务的提供者列表的映射关系，服务代理还会为每 一个服务在本地开放一个端口，当节点需要使用某项服务时，将请求发送至该端口，然后由服务 代理通过某种策略 (例如轮换策略) 安排服务的具体提供节点。

当一个Pod加入到Kubernetes集群中时，主控节点会在它上面为每一个已经存在的服务分配一 系列环境变量。变量的命名形如SVCNAME*SERVICE*HOST，其中SVCNAME是服务名称的大写。例如，服务redis-master在端口6379上提供TCP服务，其虚拟IP地址为10.0.0.11，那么该项服务对应的环 境变量就有：

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
```

```
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

这些环境变量主要说明服务的地址、端口和协议。需要注意的是，既然一个Pod加入到 Kubernetes集群时，会被设置这些跟服务相关的环境变量，这意味着一个Pod只能发现在它加入之 前就已经存在的服务。当然，如果服务支持了DNS，该条限制将会失效。

如图1所示，在前端Pod加入到该Kubernetes集群时，主控节点的apiserver会将MyApp服务 的服务信息推送给该Pod，这些信息包含了服务及其对应的端点（Endpoint）列表。前端Pod需要 使用后端Pod提供的MyApp服务，它并不直接联系后端Pod，而是将请求发给本地的服务代理，服务代理会将该服务请求分配给这三个后端Pod中的一个。服务可以动态地增加和删除提供服务 的Pod，而前端Pod感知不到这些细节的变化，除非是正在为它提供服务的Pod状态发生了改变。 在Kubernetes服务的工作原理背后，还有两个细节需要注意：冲突避免和Portal。

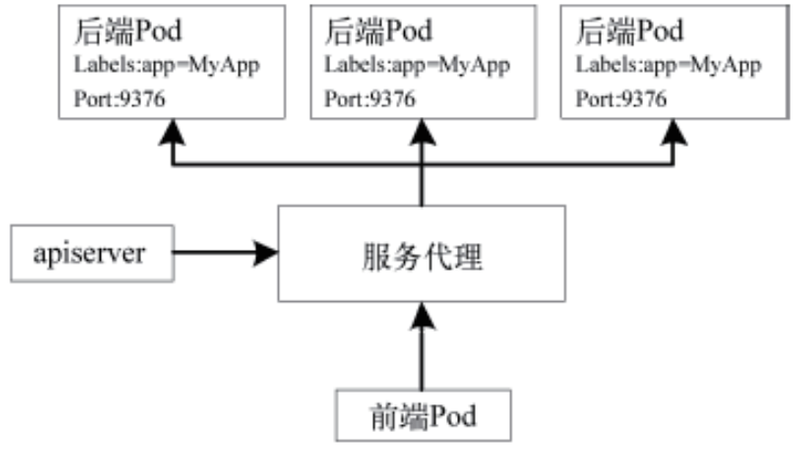


图1　前端通过服务代理来访问后端提供的服务

3. 冲突避免

Kubernetes的一个设计理念就是不能随便让用户去承受服务的失败，特别是用户根本就没有 犯错的情况下。这里我们考虑一下服务端口冲突的问题：假如第一项服务选取了80端口作为服务 端口，那么其他服务就不能再选该端口作为服务端口了。为了避免这种冲突问题，Kubernetes不 仅选择了为每一个服务分配端口，同时也配置了一个IP。这样每一个服务都拥有自己的IP和端口 对应，从而避免了服务端口冲突。

4. Portal

Portal就是前面提到的服务的二元组。这里的IP是虚拟IP，它并不是一台特定机器的真实IP。 用户访问某个服务，就是访问某个服务的Portal。当请求投递到该Portal之后，该请求会根据规则 重定向到某个特定的服务提供者Pod。下面我们举例说明该问题。如图2所示，每一个节点都 会有一个服务代理，当节点加入到Kubernetes集群中时，主

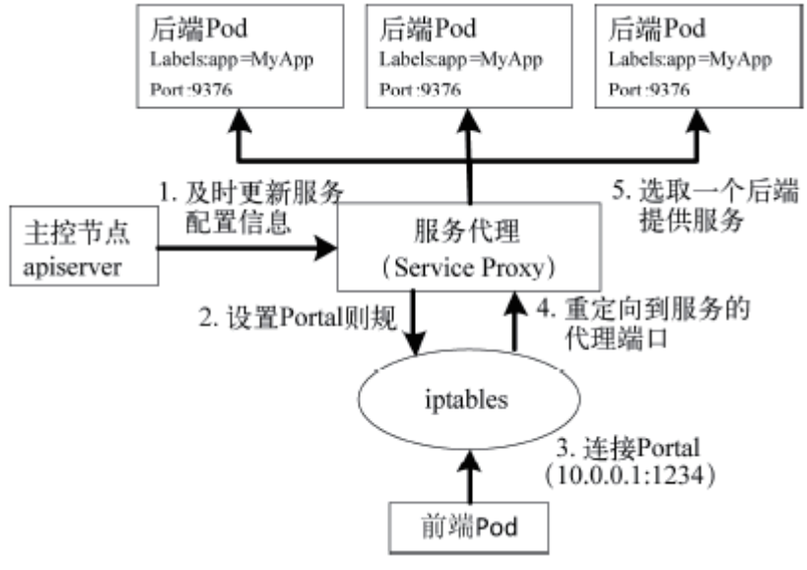


图2　服务代理响应服务的过程

控节点的apiserver程序会将服务的配 置信息投递给服务代理，服务代理保存服务配置信息，并根据Portal信息设置iptables的网络规则，这里我们假设Portal为10.0.0.1:1234。前端通过链接Portal来访问服务，iptables监听到该请求，将该请求重定向到配置好的代理端口上，服务代理从该端口接过服务请求，然后根据策略选取一个 后端Pod，最后由选中的后端Pod来给前端Pod提供服务。

标签

标签（Label）是一组附加在对象上的键值对。标签常用来从一组对象中选取符合条件的对 象，这也是Kubernetes中目前为止最重要的节点分组方法。标签的本质是附属在对象上的非系 统属性类的元数据，即它不是名字、Id以及对象的硬件属性，而是一些附加的键值对，这些键 值对对对象本身没什么影响，但对我们操作对象却极为有用。例如，下面是一个Pod节点的配 置文件：

```
{
    "id": "redis-master",
    "kind": "Pod",
    "apiVersion": "v1beta1",
    "desiredState": {
        "manifest": {
            "version": "v1beta1",
            "id": "redis-master",
            "containers": [{
                "name": "master",
                "image": "dockerfile/redis",
                "cpu": 100,
                "ports": [{
                    "containerPort": 6379,
                    "hostPort": 6379
                }]
            }]
        }
```

```
    },
    "labels": {
        "name": "redis-master"
    }
}
```

可以看到，除了id、kind等这些系统属性外，还有labels属性，它是这个Pod的标签，它的 键值对为"name":"redis-master"。有了这个标签，我们就可以在部署服务时，通过标签来指定只 有包含该标签的Pod才可以部署该服务。例如，我们定义如下服务：

```
{
    "id": "redis-master",
    "kind": "Service",
    "apiVersion": "v1beta1",
    "port": 6379,
    "containerPort": 6379,
    "selector": {
        "name": "redis-master"
    },
    "labels": {
        "name": "redis-master"
    }
}
```

这个redis-master服务通过selector（选择子）来选择标签为"name":"redis-master"的Pod 来部署服务。当然，该服务自己又定义了一个标签，其中的键值对为"name":"redis-master"，这 样需要使用该服务的应用又可以根据该标签来过滤服务。

架构和组件

Kubernetes的架构如图3所示。

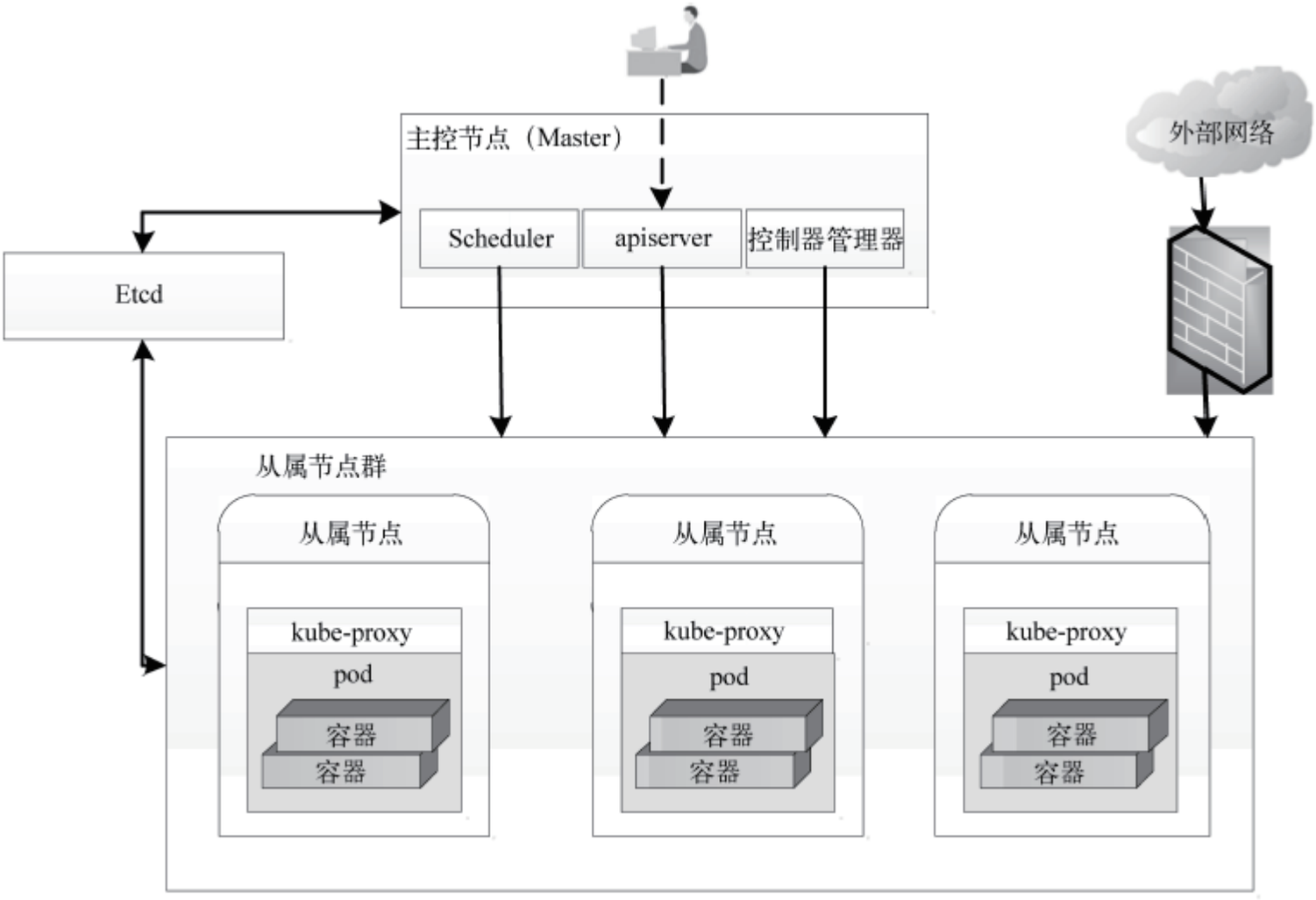


图3　Kubernetes的架构

在Kubernetes集群中，主要包含主控节点（Master）和从属节点（Minion），前者负责整个集 群的管理工作，后者是集群的工作节点群。

主控节点

主控节点包含apiserver、调度器（Scheduler）和控制器管理器（Controller Manager）。

1. apiserver

在apiserver中，我们定义了诸多Kubernetes的核心对象以及它们的操作。这些核心对象包含Pod注册表（Pod Registry）、控制器注册表（Controller Registry）、服务注册表（Service Registry）、端点注册表（Endpoint Registry）、从属注册表（Minion Registry）、绑定注册表（Binding Registry）。

在分别说明这些注册表之前，我们先来看看主控节点是如何处理客户端请求的，具体如图4 所示。

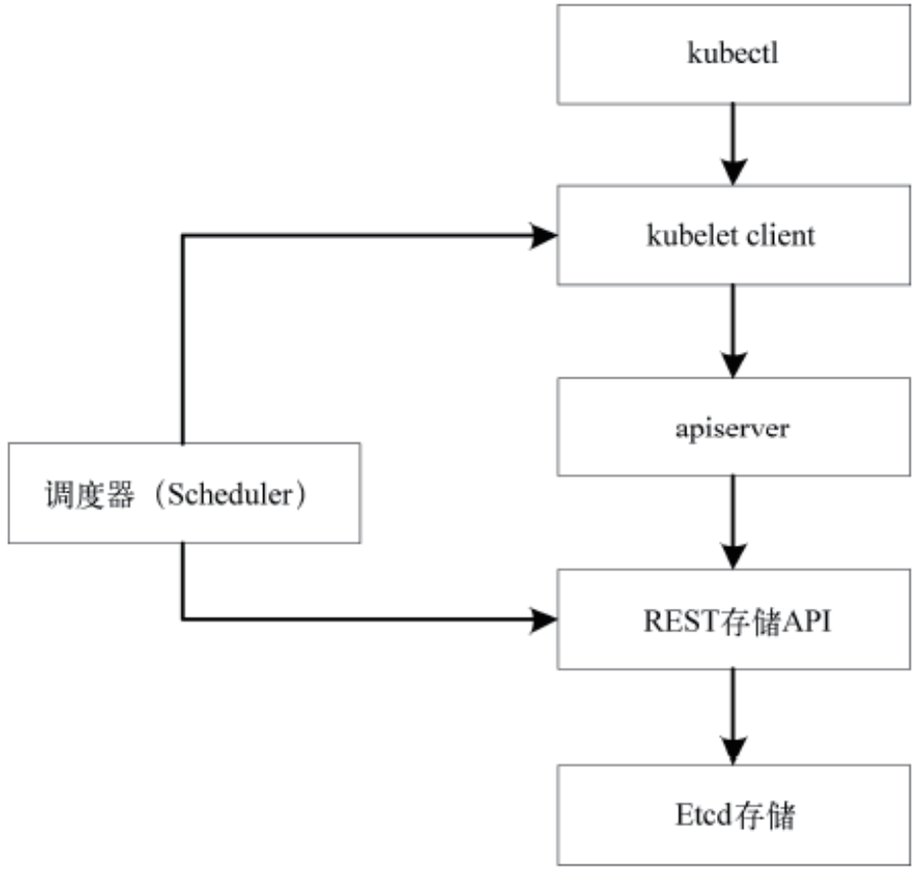


图4　主控节点处理客户端请求的流程

(1) kubectl将用户命令发送给Kubernetes客户端。
(2) Kubernetes客户端将请求发送给apiserver。
(3) apiserver根据请求类型，选择具体的REST存储API对请求进行处理。例如创建Pod，那么 则是Pod Registry存储，并将该值存储于Etcd中。
(4) 在apiserver响应请求之后，调度器会去Kubernetes客户端收集从属节点的节点信息以及Pod 信息。
(5) 根据所收集的信息，调度器将新建的Pod分发到合适的从属节点上。

在apiserver的存储库中，存储着Kubernetes的各种核心对象信息，它们是一个个注册表。下 面我们简要说明最重要的几个注册表。

- 从属注册表。负责跟踪集群中的从属节点信息。Kubernetes将节点注册表的信息封装成 RESTful形式并提供API，通过这些API可以创建和删除从属节点，目前不支持从属节点的 修改。Scheduler根据从属的节点信息决定是否将新的Pod分配到该节点。
- Pod注册表。记录集群中的Pod信息以及Pod和从属节点的映射关系。通过REST接口，可以对Pod进行创建（Create）、获取（Get）、列出（List）、更新（Update）和删除（Delete）等操作。此外，可以通过watch接口监听Pod的事件，例如创建、删除等。
- 服务注册表。负责跟踪集群中所有服务的信息。通过REST接口，可以对服务进行创建、获取、列出、更新和删除等操作。此外，也可以通过watch接口设置事件监听，监听服务 的变更事件。
- 控制器注册表。负责跟踪集群中所有的控制器，例如备份控制器（Replication Controller）的信息。通过REST接口，可以对控制器进行创建、获取、列出、更新和删除等操作。此 外，还可以通过watch接口监听控制器的事件。
- 端点注册表。负责收集服务的端点信息。一个服务可以和多个端点关联，每一个端点就 是集群中提供该服务的从属节点。通过REST接口，可以对该表项进行创建、获取、列出、更新、删除和监视操作。
- 绑定注册表。它是记录Pod和节点之间绑定关系的表，只有创建操作。

2. 调度器

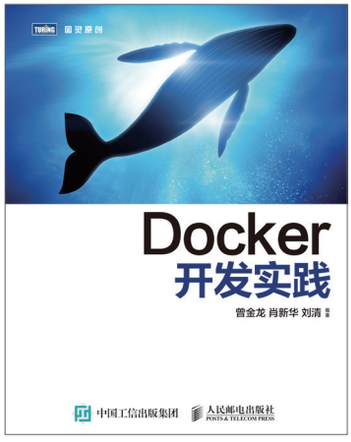
它负责收集和分析当前Kubernetes集群中所有从属节点的资源使用情况，并为新建的Pod分配 资源。调度器会实时监控Kubernetes集群中未分发的Pod，同时实时追踪集群中所有运行的Pod，根据这些Pod的资源使用状况，为尚未分发的Pod选择一个最为合适的从属节点。一旦将某个Pod 分配到某个从属节点，那就意味着这个节点的资源被占用，这部分资源就不可能再分配给其他 Pod，除非该Pod被回收。在将Pod分发到指定从属节点后，调度器会把Pod的相关信息写回到 apiserver。

3. 控制器管理器

目前，Kubernetes的控制器有端点控制器（Endpoint Controller）和备份控制器（Replication Controller）。端点控制器主要保证服务和Pod之间的映射关系是最新的。例如当Pod失效时，则应 该及时更新服务与它的映射，将它从服务的映射列表中去除。当有新的符合条件的Pod加入到一 项服务时，需要更新服务的端点信息。备份控制器用于控制Pod的备份，解决Pod的扩容缩容问题。 分布式应用出于提升服务性能和容错性等考虑，需要复制多份资源并根据负载而动态增加或者减 少。当某项服务的某个Pod因为异常而宕机时，备份控制器在检测到该事件发生后，会立即新建 一个该服务的Pod，以保证服务质量。

备份控制器的主要用法如下所示。

- 调度。备份控制器会保证指定Pod的指定副本数量在运行，当节点异常退出时，立即新建 Pod副本进行替换。
- 扩容缩容。根据需要动态增加或减少Pod数量。
- 逐步更新。某项服务需要更新时，可以对一个个的Pod进行升级更新。
- 应用的多分支跟踪。一个应用可以有多个分支，并且在集群中可以同时运行多个分支，而分支之间通过标签来识别。



《Docker 开发实践》由浅入深地介绍了Docker的实践之道，首先讲解Docker的概念、容器和镜像的相关操作、容器的数据管理等内容，接着通过不同类型的应用说明Docker的实际应用，然后介绍了网络、安全、API、管理工具Fig、Kubernetes、shipyard以及Docker三件套（Machine + Swarm + Compose）等，最后列举了常见镜像、Docker API等内容。本文节选自《Docker开发实践》。

从属节点

从属节点是Kubernetes集群中真正工作的节点。除了包含Pod外，还有用于管理和通信的基础 设施，主要是kubelet组件和服务代理。

1. kubelet

kubelet主要负责管理Pod及容器，以及与apiserver通信。它接收来自主控节点的apiserver组件 发来的命令和任务，并与Etcd、http等服务交互。kubelet包含Docker客户端、根目录、Pod Worker、Etcd客户端、Cadvisor客户端以及Health Checker组件。它的工作具体包括：

- 通过Pod Worker给Pod分配任务；
- 同步Pod的状态；
- 从Cadvisor获取容器、Pod、宿主主机信息；
- 管理Pod容器，包括运行一个指定的容器，创建网络容器，给容器绑定数据卷和端口，杀 死和删除容器以及在容器中运行命令。

2. 服务代理

我们这里说到的代理即为服务代理。每生成 一种服务，代理都会从Etcd中获取该服务的端点列表信息，然后根据配置设置iptables规则，对服务请求进行重定向。

组件交互流程

在介绍完核心概念、架构和组件之后，接下来说明这些组件是如何串联起来为我们服务的。 这里我们介绍一下创建Pod、备份控制器和服务的流程。

1. 创建Pod的流程

图5展示的是创建Pod的时序图。首先用户发起一个创建Pod的请求，

kubectl会将该请求发 送给主控节点的apiserver组件；apiserver收到请求后，会向Etcd服务器请求添加一个Pod对象。调 度器定时获取整个集群中从属节点和Pod的状态，收集其中的资源利用状况，然后决定将本次新 建的Pod分配到哪个从属节点。当从属节点和Pod绑定后，调度器会将该绑定信息返回给apiserver，然后apiserver会将该绑定持久化到Etcd中。此后，从属节点上的kubelet会定时地向Etcd汇报该绑定的状态。kubelet会根据Pod配置信息创建并启动容器。

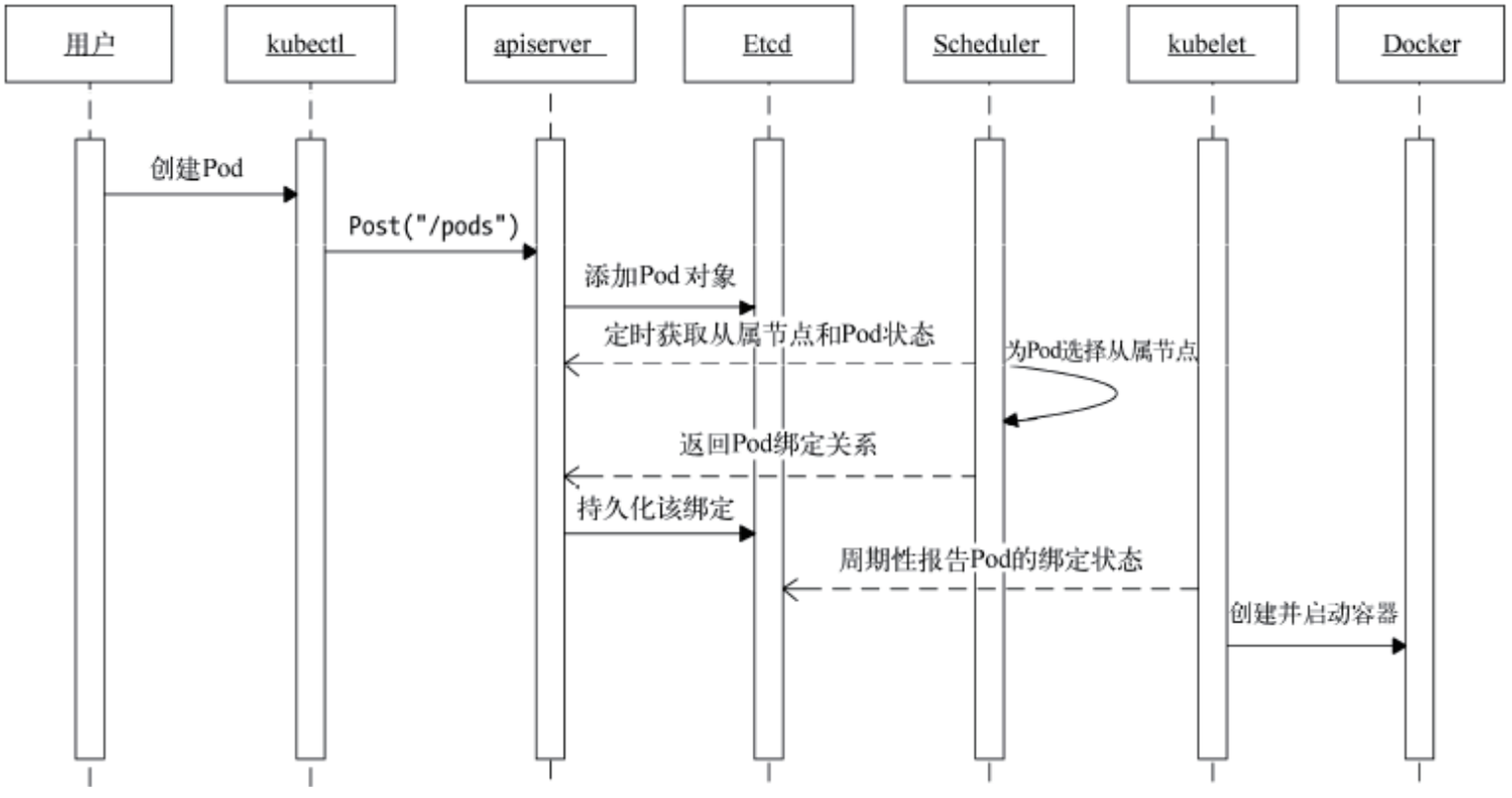


图5　创建Pod的时序图

2. 创建备份控制器的流程

图6是创建备份控制器的时序图。用户通过kubectl创建控制器，kubectl会将该请求投递到 主控节点的apiserver组件上，apiserver会

在Etcd上新建controller对象。控制器管理器定时从 apiserver查询控制器的状态，获取控制器相关Pod的状态。当apiserver收到查询Pod状态的请求后，它需要向相关的从属节点发出Pod状态查询，从属节点上的kubelet组件收到该请求后，会进一步 向Pod中的容器发起查询，然后将状态返回给apiserver，进一步返回给控制器管理器。控制器管理器同步好Pod状态信息后，如有需要，例如发现有些Pod已经异常退出了，那么它就需要创建Pod 的副本，以保证集群中始终保持设定数量的Pod在运行。

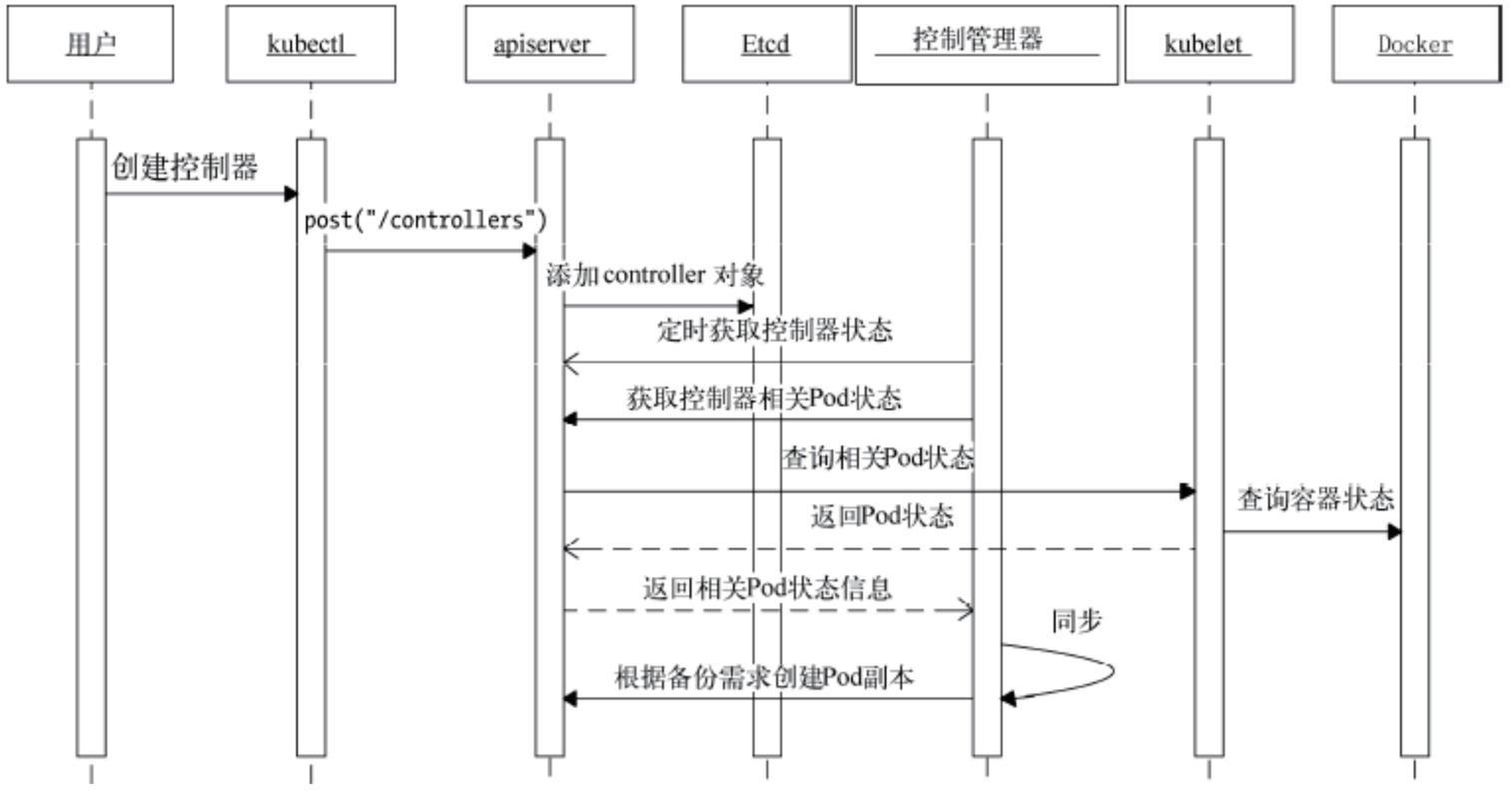


图6　创建备份控制器的流程

3. 创建服务的流程

图7是创建服务的时序图。用户先创建一个新的应用服务，apiserver会通知Etcd建立相应 的service对象。控制器管理器获取该Pod列表后，将这些Pod都设定为该服务 的端点，这些端点信息会保存在Etcd中。从属节点上的服务代理会定期从apiserver那里获取所有 服务的信息，发现有新的服务并确认自己属于服务的提供者之后，接着在本地创建套接字用于监 听该服务的请求，设置iptables的网络规则，用于服务的重定向。此外，代理会及时更新服务的端 口列表，以供服务响应。

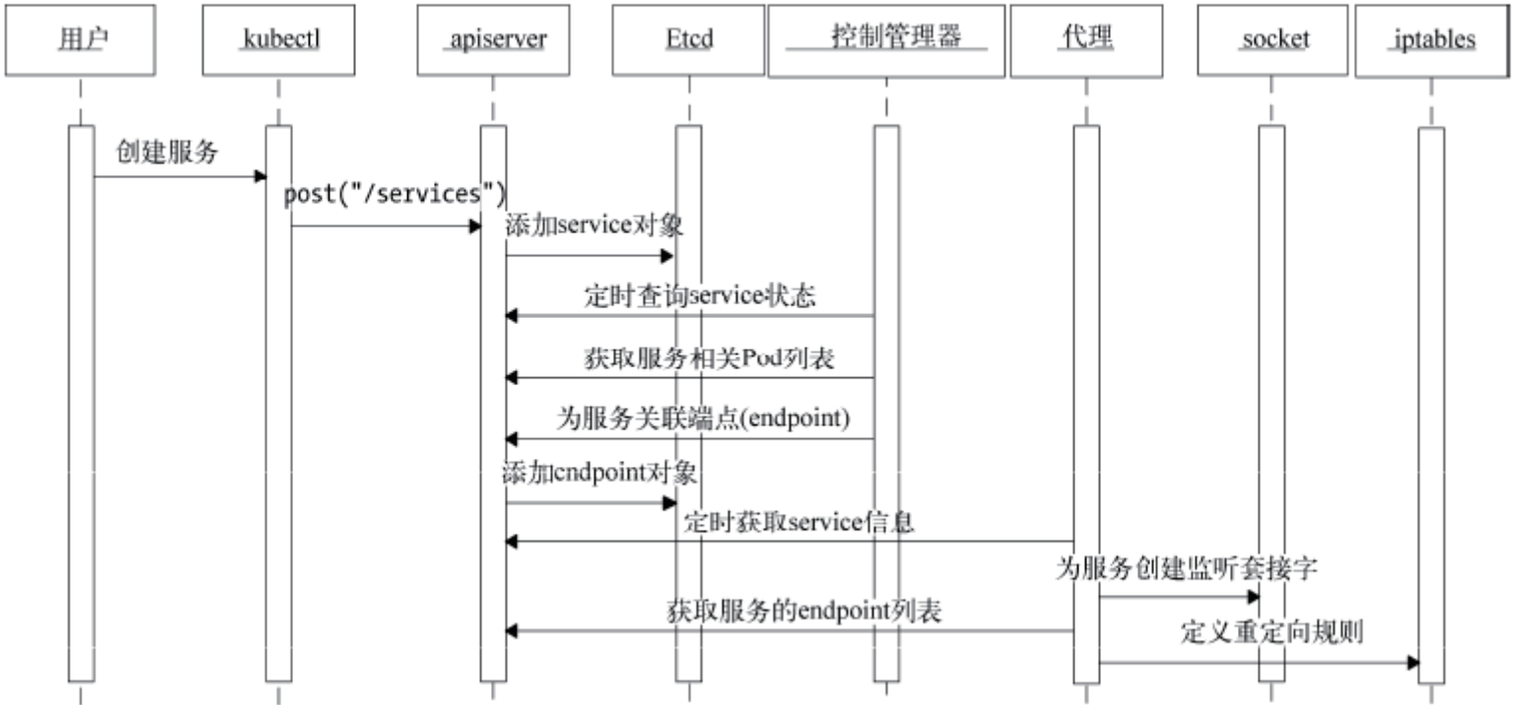


图7　创建服务的时序图

专题：Docker 云图

Mesos+Docker+Go，用300行代码创建一个分布式系统

作者 / John Walter

John毕业于北卡罗来纳大学，他在那里学习的是电影。在DZone，他专注于物联网和移动开发，他是物联网Zone和移动Zone的引导者，他对社区充满热情。他的爱好包括制作电影和打篮球。

虽然Docker和Mesos已成为不折不扣的Buzzwords，但是对于大部分人来说它们仍然是陌生的，下面我们就一起领略Mesos 、Docker和Go 配合带来的强大破坏力，如何通过300行代码打造一个比特币开采系统。

时下，对于大部分IT玩家来说，Docker和Mesos都是熟悉和陌生的：熟悉在于这两个词无疑已成为大家讨论的焦点，而陌生在于这两个技术并未在生产环境得到广泛使用，因此很多人仍然不知道它们究竟有什么优势，或者能干什么。近日，John Walter在 Dzone 上 撰 文 Creating a Distributed System in 300 Lines With Mesos, Docker, and Go，讲述了Mesos、Docker和Go配合带来的强大破坏力，本文由OneAPM工程师编译整理。

诚然，构建一个分布式系统是很困难的，它需要可扩展性、容错性、高可用性、一致性、可伸缩以及高效。为了达到这些目的，分布式系统需要很多复杂的组件以一种复杂的方式协同工作。例如，Apache Hadoop 在大型集群上并行处理 TB 级别的数据集时，需要依赖有着高容错的文件系统 (HDFS) 来达到高吞吐量。

在之前，每一个新的分布式系统，例如Hadoop和Cassandra ，都需要构建自己的底层架构，包括消息处理、存储、网络、容错性和可伸缩性。庆幸的是，像Apache Mesos这样的系统，通过给分布式系统的关键构建模块提供类似操作系统的管理服务，简化了构建和管理分布式系统的任务。

Mesos抽离了CPU 、存储和其它计算资源，因此开发者开发分布式应用程序时能够将整个数据中心集群当做一台巨型机对待。

构建在Mesos上的应用程序被称为框架，它们能解决很多问题：Apache Spark，一种流行的集群式数据分析工具；Chronos，一个类似cron的具有容错性的分布式scheduler，这是两个构建在Mesos上的框架的例子。构建框架可以使用多种语言，包括C++，Go，Python，Java，Haskell和Scala。

在分布式系统用例上，比特币开采就是一个很好的例子。比特币将为生成acceptable hash的挑战转为验证一块事务的可靠性。可能需要几十年，单台笔记本电脑挖一块可能需要花费超过 150 年。结果是，有许多的“采矿池”允许采矿者将他们的计算资源联合起来以加快挖矿速度。Mesosphere 的一个实习生，Derek，写了一个比特币开采框架（https://github.com/derekchiang/Mesos-Bitcoin-Miner），利用集群资源的优势来做同样的事情。在接下来的内容中，会以他的代码为例。

1个Mesos框架有1个scheduler和1个executor组成。schedule和Mesos master通信并决定运行什么任务，而executor运行在slaves上面，执行实际任务。大多数的框架实现了自己的scheduler，并使用1个由Mesos提供的标准executors。当然，框架也可以自己定制executor。在这个例子中即会编写定制的scheduler，并使用标准命令执行器（executor）运行包含我们比特币服务的Docker镜像。

对这里的scheduler来说，需要运行的有两种任务——one miner server task and multiple miner worker tasks。server会和一个比特币采矿池通信，并给每个worker分配blocks 。Worker会努力工作，即开采比特币。

任务实际上被封装在executor框架中，因此任务运行意味着告诉Mesos master在其中一个slave上面启动一个executor。由于这里使

用的是标准命令执行器（executor），因此可以指定任务是二进制可执行文件、bash脚本或者其他命令。由于Mesos支持Docker，因此在本例中将使用可执行的Docker镜像。Docker是这样一种技术，它允许你将应用程序和它运行时需要的依赖一起打包。

为了在Mesos中使用Docker镜像，这里需要在Docker registry中注册它们的名称：

```
const (
    MinerServerDockerImage = "derekchiang/p2pool"
    MinerDaemonDockerImage = "derekchiang/cpuminer"
)
```

然后定义一个常量，指定每个任务所需资源：

```
const (
    MemPerDaemonTask = 128  // mining shouldn't be    memory-intensive
    MemPerServerTask = 256
    CPUPerServerTask = 1    // a miner server does not use much     CPU
)
```

现在定义一个真正的scheduler，对其跟踪，并确保其正确运行需要的状态：

```
type MinerScheduler struct {
    // bitcoind RPC credentials
    bitcoindAddr string
    rpcUser      string
    rpcPass      string
    // mutable state
    minerServerRunning  bool
    minerServerHostname string
    minerServerPort     int    // the port that miner daemons
                              // connect to
    // unique task ids
    tasksLaunched        int
    currentDaemonTaskIDs []*mesos.TaskID
}
```

这个scheduler必须实现下面的接口：

```
type Scheduler interface {
    Registered(SchedulerDriver, *mesos.FrameworkID,     *mesos.MasterInfo)
    Reregistered(SchedulerDriver, *mesos.MasterInfo)
    Disconnected(SchedulerDriver)
    ResourceOffers(SchedulerDriver, []*mesos.Offer)
    OfferRescinded(SchedulerDriver, *mesos.OfferID)
    StatusUpdate(SchedulerDriver, *mesos.TaskStatus)
    FrameworkMessage(SchedulerDriver, *mesos.ExecutorID,
                     *mesos.SlaveID, string)
    SlaveLost(SchedulerDriver, *mesos.SlaveID)
    ExecutorLost(SchedulerDriver, *mesos.ExecutorID,   *mesos.SlaveID,
                 int)
    Error(SchedulerDriver, string)
}
```

现在一起看一个回调函数：

```
func (s *MinerScheduler) Registered(_ sched.SchedulerDriver,
      frameworkId *mesos.FrameworkID, masterInfo *mesos.MasterInfo) {
    log.Infoln("Framework registered with Master ", masterInfo)
}
func (s *MinerScheduler) Reregistered(_ sched.SchedulerDriver,
      masterInfo *mesos.MasterInfo) {
    log.Infoln("Framework Re-Registered with Master ",  masterInfo)
}
func (s *MinerScheduler) Disconnected(sched.SchedulerDriver) {
    log.Infoln("Framework disconnected with Master")
}
```

Registered 在scheduler成功向Mesos master注册之后被调用。

Reregistered 在scheduler与Mesos master断开连接并且再次注册时被调用，例如，在 master 重启的时候。

Disconnected 在scheduler与Mesos master断开连接时被调用。这个在master挂了的时候会发生。

目前为止，这里仅仅在回调函数中打印了日志信息，因为对于一个像这样的简单框架，大多数回调函数可以空在那里。然而，下一个回调函数就是每一个框架的核心，必须要认真的编写。

ResourceOffers 在scheduler从master那里得到一个offer的时候被调用。每一个offer包含一个集群上可以给框架使用的资源列表。资源通常包括 CPU 、内存、端口和磁盘。一个框架可以使用它提供的一些资源、所有资源或者一点资源都不给用。

针对每一个offer，现在期望聚集所有的提供的资源并决定是否需要发布一个新的server任务或者一个新的worker任务。这里可以向每个offer发送尽可能多的任务以测试最大容量，但是由于开采比特币是依赖CPU的，所以这里每个offer运行一个开采者任务并使用所有可用的CPU资源。

```
for i, offer := range offers {
    // … Gather resource being offered and do setup
    if !s.minerServerRunning && mems >= MemPerServerTask &&
            cpus >= CPUPerServerTask && ports >= 2 {
        // … Launch a server task since no server is running and     we
        // have resources to launch it.
    } else if s.minerServerRunning && mems >= MemPerDaemonTask {
        // … Launch a miner since a server is running and we have     mem
        // to launch one.
    }
}
```

针对每个任务都需要创建一个对应的TaskInfo message，它包含了运行这个任务需要的信息。

```
s.tasksLaunched++
taskID = &mesos.TaskID {
    Value: proto.String("miner-server-" +
                        strconv.Itoa(s.tasksLaunched)),
}
```

Task IDs由框架决定，并且每个框架必须是唯一的。

```
containerType := mesos.ContainerInfo_DOCKER
task = &mesos.TaskInfo {
    Name: proto.String("task-" + taskID.GetValue()),
    TaskId: taskID,
    SlaveId: offer.SlaveId,
    Container: &mesos.ContainerInfo {
        Type: &containerType,
        Docker: &mesos.ContainerInfo_DockerInfo {
            Image: proto.String(MinerServerDockerImage),
        },
    },
    Command: &mesos.CommandInfo {
        Shell: proto.Bool(false),
        Arguments: []string {
            // these arguments will be passed to run_p2pool.py
            "--bitcoind-address", s.bitcoindAddr,
            "--p2pool-port", strconv.Itoa(int(p2poolPort)),
            "-w", strconv.Itoa(int(workerPort)),
            s.rpcUser, s.rpcPass,
        },
    },
    Resources: []*mesos.Resource {
        util.NewScalarResource("cpus", CPUPerServerTask),
        util.NewScalarResource("mem", MemPerServerTask),
    },
}
```

TaskInfo message指定了一些关于任务的重要元数据信息，它允许Mesos节点运行Docker容器，特别会指定name、task ID、container information以及一些需要给容器传递的参数。这里也会指定任务需要的资源。

现在TaskInfo已经被构建好，因此任务可以这样运行：

```
driver.LaunchTasks([]*mesos.OfferID{offer.Id}, tasks,       &mesos.
Filters{RefuseSeconds: proto.Float64(1)})
```

在框架中，需要处理的最后一件事情是当开采者 server 关闭时会发生什么。这里可以利用 StatusUpdate 函数来处理。

在一个任务的生命周期中，针对不同的阶段有不同类型的状态更新。对这个框架来说，想要确保的是如果开采者 server 由于某种原因失败，系统会 Kill 所有开采者 worker 以避免浪费资源。这里是相关的代码：

```
if strings.Contains(status.GetTaskId().GetValue(), "server") &&
    (status.GetState() == mesos.TaskState_TASK_LOST ||
        status.GetState() == mesos.TaskState_TASK_KILLED ||
        status.GetState() == mesos.TaskState_TASK_FINISHED ||
        status.GetState() == mesos.TaskState_TASK_ERROR ||
        status.GetState() == mesos.TaskState_TASK_FAILED) {
    s.minerServerRunning = false
    // kill all tasks
    for _, taskID := range s.currentDaemonTaskIDs {
        _, err := driver.KillTask(taskID)
        if err != nil {
            log.Errorf("Failed to kill task %s", taskID)
        }
    }
    s.currentDaemonTaskIDs = make([]*mesos.TaskID, 0)
}
```

万事大吉！通过努力，这里在 Apache Mesos 上建立一个正常工作的分布式比特币开采框架，它只用了大约 300 行 GO 代码。这证明了使用 Mesos 框架的 API 编写分布式系统是多么快速和简单。■

阅读英文原文

阅读图灵社区原文

专题：Docker 云图

VLIS 实验室云计算组张磊：

关于 Docker、开源，以及教育的尝试

张磊，浙江大学计算机学院博士生，科研人员，VLIS 实验室云计算组技术负责人、策划人。Kubernetes 项目贡献者和维护者，Docker 项目贡献者。前 Cloud Foundry 中国团队和百度私有云项目组成员。InfoQ、CSDN、《程序员》杂志等多篇浙大系技术文章的贡献者和策划人，他还是《Docker——容器与容器云》一书的主要作者之一。

问：你现在的目标是成为一位计算机科学家吗？

是。

问：你是从什么时候开始想成为一位计算机科学家的？

大概是由于中学时候就开始玩电脑，然后自然而然就进入了计算机专业。由于我的兴趣主要在云计算以及Linux操作系统上，所以我的导师跟课题都是跟这些相关的。

2011年我们开始专注于开源云计算技术，当时开源的力量正在逐渐浮现。后来事实证明我们对趋势的判断是对的，因为从那个时候开始，软件的开发和发布以及整个生命周期就发生了改变。从此以后，可以说开源技术掌握了云计算行业，甚至是整个计算机行业的主流发展态势。于是，我们开始更深入地钻研Cloud Foundry, Docker, Kubernetes这样的技术，并且作为这些项目的贡献者成为了社区中的重要成员。

问：你认为在云计算领域，学术跟产业之间有没有明确的界限？

首先，云计算这个领域本身就比较特殊，它其实没有很多基础性研究，所以这个领域的学术跟工业是分不开的，中间没有一个明显的界限。比如，伯克利的AMP实验室做的一套高性能数据分析系统，最终开源出去就变成现在的明星项目Spark，成为工业界大数据的事实标准。在这些领域中，没有任何一个界限能够划定哪些技术是学术的，而工业界不能用。因为云计算本来就是“站在巨人肩膀上”的一种技术，它基于已有的分布式系统来做进一步的创新和整合。

问：你现在在SEL实验室的工作是什么？

我主要负责实验室云计算团队的技术工作，以及与技术相关的其他事宜，包括开源以及一些商业上的技术合作。

问：SEL实验室的前身是VLIS实验室，当初建立VLIS实验室的目的是什么？现在SEL实验室的关注点有没有变化？

我们实验室最开始的研究方向就是软件工程以及计算机软件。实验室一开始就注重从学校的角度跟知名企业建立强强联合的关系，致力于为工业界提供最好的大规模信息系统的开发技术和能力。在那个时候，我们的主要关注点是金融信息系统的开发，并且同全球最大的资金托管机构美国道富银行建立了紧密的合作研发关系。从这个时候开始，我们向工业界输出了大量的技术能力，整个北美市场的股票交易系统的后台都是我们实验室师生参与重新开发的。而软件技术发展到现在，新一代的大规模分布式系统开始更多地以知名开源项目作为表现形式，这些技术也就自然成为我们新的关注点。其中最重要的还是云计算技术，但是我们略有侧重，更关注轻量化的云计算技术，我们认为这将是一个新的变革。

问：你们实验室跟Cloud Foundry还有百度、思科等知名企业都是以什么形式合作的？

首先肯定有人才上的合作，因为学校本来是人才，我们会选择一些从事这个方向的优秀的学生，联合工业界的公司，比如去道富，VMware，思科总部或者百度等等，完成一个以技术研究为核心，以实际开发为途径的长期合作学习的过程。学生不是实习几个月然后回来，而是从开始到毕业，从研究的方向到最后的毕业论文都跟这些公司的真实技术场景紧密相关，并且专注于这些IT巨头的核心技术以及云计算平台的开发和研究工作。

问：你们实验室为什么在Docker还不太完善的时候就敢去尝试这种技术？

容器技术不是一种新技术，它其实很早就存在了。在此之前，我们搞Linux内核的时候已经用过类似的技术，而且做PaaS用到的技术也是基于Linux容器的，所以我们团队很久以前就对这种技术有过很多接触和研究。2013年的时候，我们同学主要在从事的是VMware的Warden容器的研发，紧接着Docker就出现了，并且比前一代容器技术要完善很多，

所以我们就自然而然地转到了Docker上。这就是为什么Docker一出现就会引起我们的关注。轻量化的云计算技术一定会以这样的方式实现，只不过实现不同，而我们肯定会选择更好的实现。

问：你本人是怎么成为Kubernetes和Docker项目的贡献者的？

我是从Cloud Foundry团队出来的，而Cloud Foundry是一个纯粹的开源项目，它没有市场人员和项目经理这样的角色来干扰工程师的工作，所有贡献者都是通过远程协作和结对编程来贡献代码的。我，以及我们实验室的大多数同学从一开始就是这样一种工作模式，所以对于我们来说，参与开源项目是很自然的，而且我们也不像其他公司那样寻求互等的商业利益。

我们认为开源项目是一定要参与的，不仅要参与，还要学会主导项目的方向，成为维护者和更重要的核心代码贡献者。我刚才讲过，我们的愿景就是要基于开源软件做事情，做研究也好，做进一步的商业活动也好，一定要进入到社区里面，而不仅仅是一个使用者。

问：Docker一直存在安全方面的问题，在这方面你有哪些经验可以和大家分享？

我也在《Docker——容器和容器云》里提到过，Docker本身确实有安全问题，但是一定要分场景讨论。比如，什么样的场景下我会在一台裸机上部署Docker；什么样的场景下我会让Docker容器跑在虚拟机里面。在目前这种情况下，如果你是一个公有云提供商，我认为你还是要把容器拷在你的虚拟机里面，防止出现逃逸状况。

我们在书里讲过，你的Docker容器和整个Docker Daemon环境最好做安全加固，在操作系统层面做很多加固，设置权限，并且在整个系统的设计上把权限设计和授权设计摆在第一位，逐层来把不正确的行为过滤掉。另外，一定要区分场景。对于私有云的话，安全要求满足第一点就可以了。但是对于公有云来说，一定要做最高等级的安全预案。

以上是从业务方面讲，但是从技术方面讲，容器本身的安全问题是很难解决的，但是有一些努力的方向非常值得我们关注。比如最新的Rocket集成了英特尔在CPU上的一些虚拟化技术来做到硬件加固，这就是等级很高的安全技术。另外还有像国内赵鹏他们做的Hyper，是一个基于虚拟化技术的容器，它跟虚拟机的安全系数几乎是一样的。所以我觉得从另一方面说，这些技术应该得到大家的重视，并且集成到我们现有的解决方案里面。

问：现在在生产环境中使用Docker的人还是不太多，其背后的原因多种多样，你认为现在Docker面临的最大的阻碍是什么？

第一个问题在于Docker所属公司本身的强势，以及他们自己想做一揽子事情的态度。这个问题使得Docker现在变得非常臃肿，而且导致本来应该专注解决的问题没有解决掉。如果他们投入主要力量来解决容器的安全问题，我觉得反而要比现在的态势好。

第二，我们使用Docker也好，对它做二次开发也好，其实我们不要把自己的眼光局限在Docker上面。让Docker只做容器的事情其实是最好的选择，其余的事情交给更专业的人。我们不要过分相信来自某些商业化的宣传，比如一个人或者一个团队就能完成从开发到部署到运维的所有流程。哪怕你一开始可以，但是随着业务的正常增长，技术发展到一定程度之后你一定是做不到的。所以一定是专业的人做专业的事。

问：Docker和CoreOS一起创立了一个开放容器计划（OCP）。你认为OCP的成立对于软件开发行业会造成什么影响？

容器镜像不单指Docker镜像，它很久之前就已经存在了。容器镜像作为一种软件的发布方式，现在已经得到了大家的认可，成为了行业的事实标准。并且由于容器镜像本身已经存在了很久，所以它本身标准的普及是比较容易的。所以OCP的成立其实是为这种镜像发布的方式提供了一个技术上的标准。

以前，这一套东西虽然可以做标准，但是没有技术来支撑它，现在有了。我们通过标准容器来支撑标准镜像。所以，这两个标准如果能够在OCP里面得到统一，我相信它对整个软件工程将来的发展都是有很大影响的。我们现在学校的课程里就已经引入了完全基于容器的软件工程设计模式，谷歌也提出了基于容器的编程模型，将来的软件工程一定会向这方向发展。

问：你有一篇文章叫做《从Borg到Kubernetes》。你觉得Kubernetes今后的发展会怎么样？它和Mesos分别会向什么方向发展？

我们最近也跟谷歌的人一起交流了很久，首先，Kubernetes确实背负了很多Borg之前的优秀的设计理念，其中包括Borg在谷歌内部大规模集群业务的应用。虽然现在还有一些应用我们看不到，但是Kubernetes将来的发展目标一定是用来解决这些问题。

Mesos和Kubernetes在一开始发展时其实是非常直接的竞争对手，因为这两者关注的事情有很多是一致的。但是随着这两个项目的继续发展，它们已经形成了合作关系。比如，Mesos本来就是一个优秀的调度器，那么接下来Mesos会更关注这个业务。并且Mesos可以被更方便地集成到Kubernetes里，作为Kubernetes的一个核心调度器来工作。

这两个项目现在的关注点其实是不一样的，使用的场景也不一样。在今后的发展中，它们会逐渐融合对方的优点。互相之间的集成会越来越多，互相之间的重合会越来越少。

问：学术跟产业之间的脱轨问题一直以来经常被人们所诟病，浙大在这方面做得很不错，有没有什么经验可以分享？

首先，作为一所学校，要学会如何在我们国家的体制下，在不影响正常的教学、科研的前提下，获得企业的支持。学校需要鼓励学生在看似枯燥的学业中找到真正的个人兴趣点。比如我们实验室就涌现出了非常多

的代码贡献者、作者、领域内的小专家。因为我们实验室从一开始就鼓励个人发展自己的兴趣，并且鼓励同学向大家分享你的工作成果。这样，工业界会自然而然地关注过来，合作也接踵而至。

另一方面，如果学校自身的硬件条件很强，技术水平很高的话，学校就可以为工业界做出很多贡献，无论是开源贡献，还是参与到工业的开发。并且还有一点很重要，就是学校要想办法把实验室的技术和研究经验转化成工程上可以应用到增值需求里的东西，而不只是埋头写论文。所以，我们实验室在硕士生阶段，不会提出苛刻的论文要求。我们更希望你的论文是对一个开源项目的贡献，或者是我们和产业界合作的课题的相关实际工程经验。

同时，我们也在浙大试点了一个学院，整个软件学院在以更加工程的方式推动科研的走向。我们课题组从十年前开始做这样的事情，所以学术跟产业之间的脱轨问题我们这边几乎是没有的。

问：浙大有这么好的环境，但是很多其他大学没有能力提供这样的环境。你建议在一般大学学习的学生怎样来丰富自己的专业知识？

首先对于一个学生，尤其是CS专业的学生，我认为实习是最重要的。你一定要想办法在跟导师融洽相处的前提下，寻求到与自己专业相关的实习机会，并且珍惜这些机会，因为实习能够使你的专业技能得到锻炼。并且你应该想办法把实习转化成毕业论文，或者是学校要求的课程设计。这样的经历会对你在工业界的影响力也好，工作也好，起到很大的积极作用。

问：你认为学习Docker需要几个阶段？

我们在书的后记里面讲过，不止是Docker，对于任何一个开源项目来说，都有这样的三个或四个阶段。 首先，你要去用，而且不只要用，还要变

成一个优秀的玩家。对于开源项目的所有指令、所有设计，你应该有一个感性认识。

在这个基础之上就是源码，要读源码。读源码是一件非常有意思的事，但是在这个过程中，你要学会提问，带着问题去读源码，才会有收获。

然后就是转化，转化包括几种情况。比如，你可以将容器技术转化成你们实验室的某个项目的基础或者工作中的整个项目。另外，你要学会对项目做贡献。从最开始的找bug、解决bug、修改文档，到最后提出自己的特性、融入到社区里，只有这样你才能够获得最多的知识，以最快的速度提高自己在这个领域中的能力。

这三步之后，如果你在这个方面做得更多，可以考虑一些商业化的事情。比如你可以做一些相关的买卖，或者在你的公司里推广这些开源技术。

问：你觉得读者应该怎样使用《Docker——容器和容器云》这本书，读者在哪个阶段需要用到这本书？

这本书应该更适合在第二或第三阶段阅读。

这本书的一个特点就是它倾向于把原理帮你从源码中抽象出来，而不仅仅带你走读代码。因为代码很快会过时，所以我们特别注重抽象原理。在第三阶段的时候，你要去做开发，所以你对技术的熟悉程度和原理必然要有一个深刻的认识。而这本书的很多实践章节，包括我们对代码的整个框架结构的分析，会对你有很大的帮助。

问：《Docker——容器和容器云》的关注点和其他类似的书有什么不同？

首先，我们不认为容器就是Docker，我们认为它只是容器技术一种优秀的实现。所以我们的书叫《Docker——容器与容器云》，我们更关注所

有基于容器的云平台的实现方式。你可以把容器理解为我们现在的虚拟机，把容器云理解为 OpenStack，所以我们这本书肯定要先讲虚拟机原理，以此为基础我们才能讲清楚容器云，也就是容器的大规模管理方式。

很多目前市面上的书只关注于 Docker 本身，而我们更关注 Docker 背后的 `libcontainer` 也就是 runc 的工作原理，于此同时，我们还非常关注所谓的容器技术与大规模容器集群管理的结合方式。我们非常想为大家解决的一个问题就是，真正的大规模容器集群管理应该是什么样的。我们认为 Kubernetes 现在的方向是非常好的，所以在书中我们对它做了一个非常详细深入的解读，这在国内外应该是首次，而且从谷歌和 CoreOS 工程师的反馈来看，甚至在国外可能也是第一次。■

专题：Docker 云图

阅读Docker源代码的神兵利器

作者 / 张磊

浙江大学计算机学院博士生，科研人员，VLIS实验室云计算组技术负责人、策划人。Kubernetes项目贡献者和maintainer，Docker项目贡献者。前Cloud Foundry中国团队和百度私有云项目组成员。InfoQ、CSDN、《程序员》杂志等多篇浙大系技术文章的贡献者和策划人。

正所谓“磨刀不误砍柴工”，本文我们将介绍几个阅读源码所要用到的神兵利器。其中，LiteIDE是国人开发的专为Go语言而生的集成开发环境（IDE），它可以用以简洁和美观著称的 Sublime Text 2编辑器进行源码阅读。当然，作为一代Geek，我们还可以选择使用长盛不衰的Vim 和Emacs。相信通过本节的阅读，一定能让读者在阅读源码时更加得心应手。

Golang开发环境的安装

阅读Go源码之前，安装Go语言的开发环境是必不可少的。下面我们介绍下载和安装的步骤。

1. 下载官方的Go语言安装包

请根据操作系统的版本（FreeBSD、Linux、Mac OS X或者Windows）以及处理器的架构（386、amd64或者arm）进行选择。下载地址为：https://golang.org/dl/ ，是Google提供的服务，可能需要 使用VPN才能访问。

2. 安装Go语言安装包

选择合适的版本下载完成后，就可以开始进行Go语言安装包的安装了，过程如下。

FreeBSD、Linux以及Mac OS X之tar安装

对于FreeBSD、Linux以及Mac OS X用户来说，下载好的tar压缩文件需要再执行以下步骤才 算是安装完成。

把压缩包解压至/usr/local目录下，命令如下：

```
tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
```

选择适合的压缩包进行安装，例如，如果在64位Linux系统上安装Go 1.2.1版本，那么对应的 压缩包就是go1.2.1.linux-amd64.tar.gz。

把/usr/local/go/bin添加到系统的环境变量中，可以通过把下面这行命令加入到/etc/profile（系统所有用户都受影响）或者$HOME/.profile（当前用户受影响）文件中来完成。

```
export PATH=$PATH:/usr/local/go/bin
```

提示 >>>

Go的安装环境默认安装在/usr/local（Windows系统是C:\）路径下。如果指定某个本地目 录为安装路径，就必须设置$GOROOT环境变量。如果要把安装包解压至$HOME目录下，就需 要把下面两行代码加入到$HOME/.profile文件中。

```
export GOROOT=$HOME/go
export PATH=$PATH:$GOROOT/bin
```

Mac OS X之pkg快速安装

下载后缀为.pkg的相关安装包，打开后按照图形界面的指引操作即可顺利安装。使用这种方 法时，安装包默认会安装到/usr/local目录下，/usr/local/go/bin也会被加入到环境变量。安装 完成后，在终端使用时需要重新开启一个会话，以便生效。

Windows安装

除了源码安装以外，Go官方给用户提供了两种安装开发环境的方法：一种是手动解压缩Zip 包安装，这需要设置环境变量；另一种是全自动安装。

MSI安装：打开MSI文件，按照指引界面一步步操作即可，默认安装在C:\Go路径下。安装 器会自动把C:\Go\bin目录加入到环境变量中。同样，需要重启命令行使之生效。

Zip安装：把Zip文件下载并解压缩到自己选择的目录，推荐C:\Go。如果放到C:\Go以外的目 录，需要设置GOROOT变量。然后把解压缩后Go目录下的bin目录（如C:\Go\bin）加入到PATH环境 变量中。

Windows下设置环境变量：在Windows系统下，可以通过“计算机”→“系统属性”→“高 级”→“PATH”来设置环境变量。

3. 测试Go语言环境

完成以上步骤后，Go语言环境便安装完成了，最后我们来测试一下。

首先，创建一个hellow.go的空白文件，输入以下代码：

```
package main
import "fmt"
func main() {
    fmt.Printf("hello, world\n")
}
```

然后通过Go语言工具编译运行，示例如下：

```
$ go run hello.go
hello, world
```

如果看到了hello world，那么一切便大功告成了。

工具的配置与技巧

遗憾的是，Google官方并没有开发出一款专门为Golang打造的IDE，但开源社区为此作出了 巨大的贡献。本节将介绍几种常见的IDE与编辑器的配置和使用技巧。

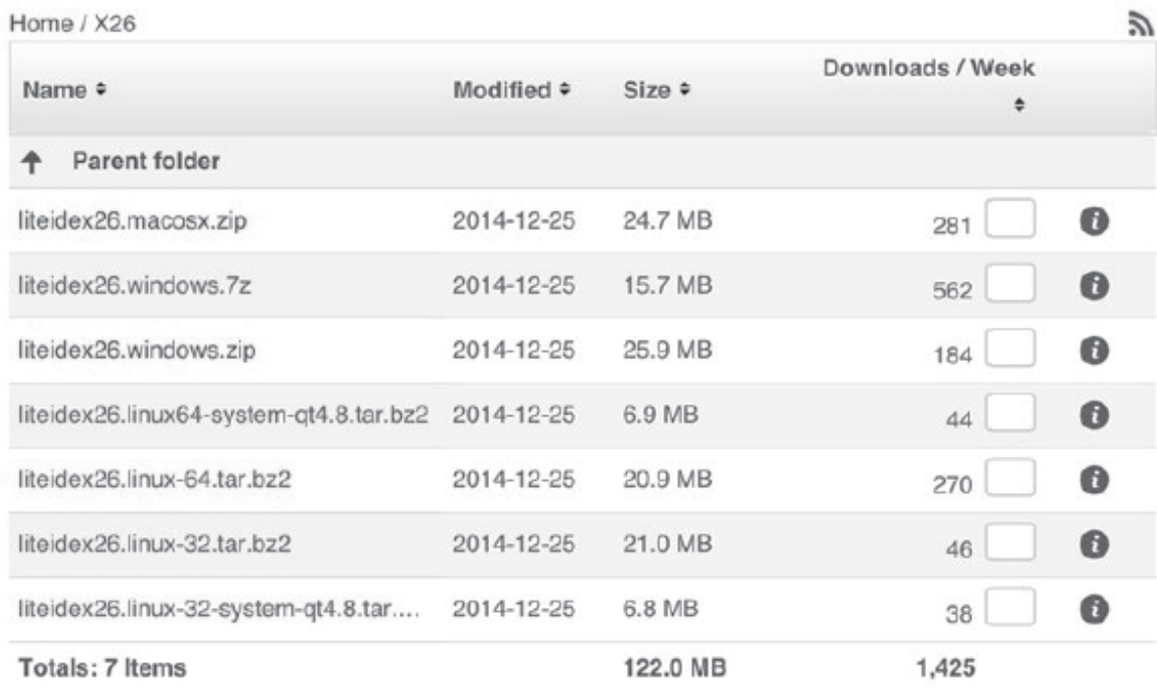
1. LiteIDE

LiteIDE是国人开发的一款专门为Go语言量身定做的IDE，它简单实用、开源并且可跨平台。

下载与安装

LiteIDE的安装文件托管在sourceforge平台上，因为项目是开源的，我们也可以在GitHub上下 载项目源代码进行安装。下载地址为：http://sourceforge.net/projects/liteide/files。

打开下载目录，如图1所示，会看到各个版本的下载目录，推荐下载最新的版本。有了Go 开发环境的安装经验，安装LiteIDE就会简单很多。



Home / X26

Name	Modified	Size	Downloads / Week
Parent folder			
liteidex26.macosx.zip	2014-12-25	24.7 MB	281
liteidex26.windows.7z	2014-12-25	15.7 MB	562
liteidex26.windows.zip	2014-12-25	25.9 MB	184
liteidex26.linux64-system-qt4.8.tar.bz2	2014-12-25	6.9 MB	44
liteidex26.linux-64.tar.bz2	2014-12-25	20.9 MB	270
liteidex26.linux-32.tar.bz2	2014-12-25	21.0 MB	46
liteidex26.linux-32-system-qt4.8.tar....	2014-12-25	6.8 MB	38
Totals: 7 Items		**122.0 MB**	**1,425**

图1　LiteIDE下载列表

下载相应版本的LiteIDE后进行解压缩。不同操作系统的用户，LiteIDE对应的安装方法如下。

- Mac OS X：把解压后的LiteIDE.app直接拖动到Application文件夹下，以后即可在 Launchpad中方便地找到并打开它。
- Windows：解压缩后，在liteide/bin目录下，双击liteide.exe即可打开运行。
- Linux：解压缩后，在liteide/bin/目录下，双击liteide即可打开运行。
- LiteIDE支持Go语言阅读的功能简介

LiteIDE支持以下Go语言阅读功能。

- Go语言包浏览器（Package browser）
- 类视图和大纲（Class view and outline）

- 文档浏览（Document browser）
- Go语言Api函数检索（GOPATH API index）
- 代码跳转（Jump to Declaration）
- 代码表达式信息显示（Find Usages）

LiteIDE使用的图标是太极两仪的样式，有着浓浓的中国风，打开它就会看到介绍。可以使 用打开文件夹功能，直接打开Docker源代码文件夹。如图2所示，在左边可以看到文件浏览目 录、类试图、文件夹列表、大纲以及包浏览器这几个功能；右边是打开的Docker官方的 README.md文件，可以看到LiteIDE支持预览Markdown格式，这样可以在这个IDE上面方便地读 文档。

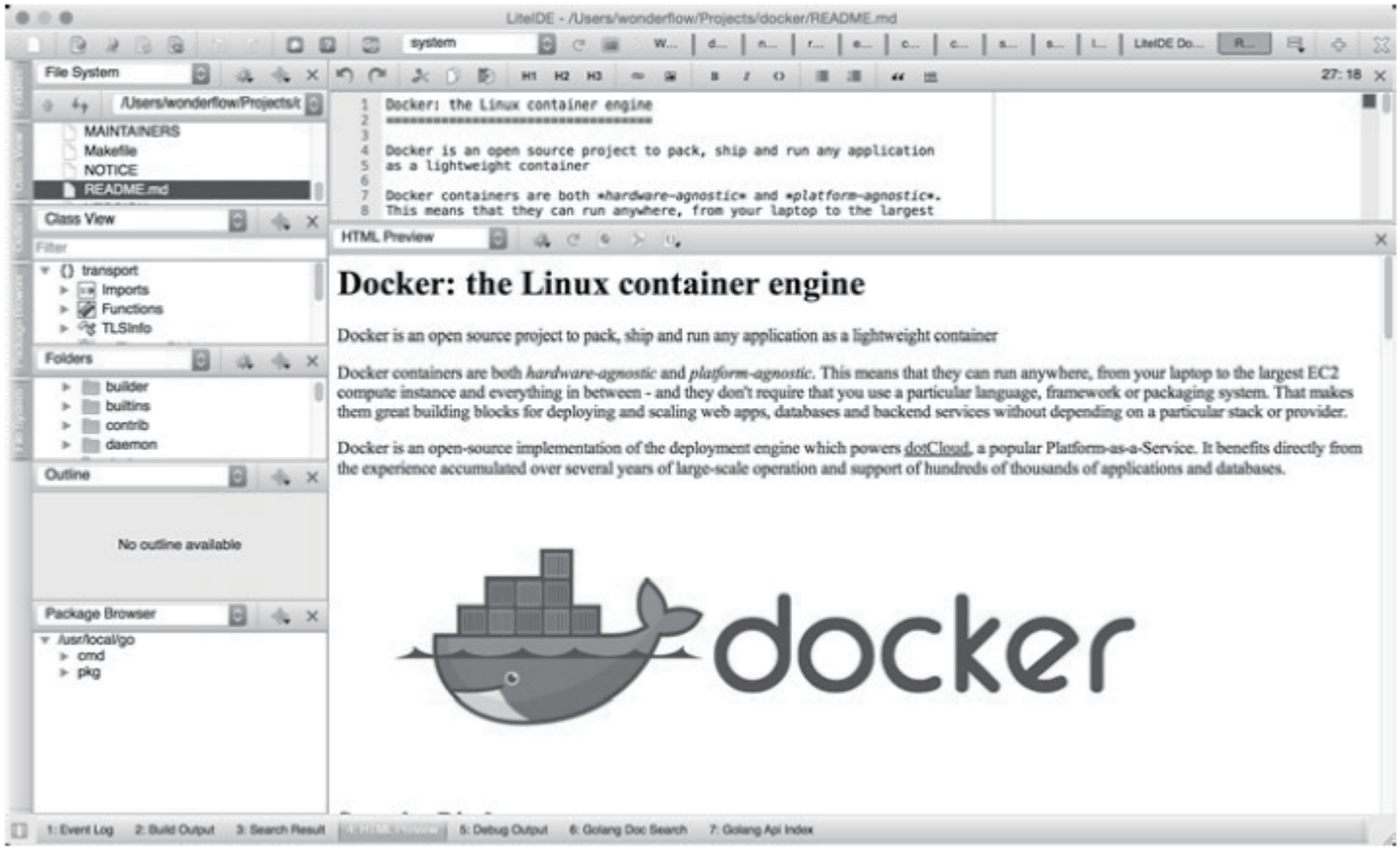


图2　LiteIDE功能界面展示

打开IDE以后，大家可以在设置中查看和修改快捷键设置，如图3所示。

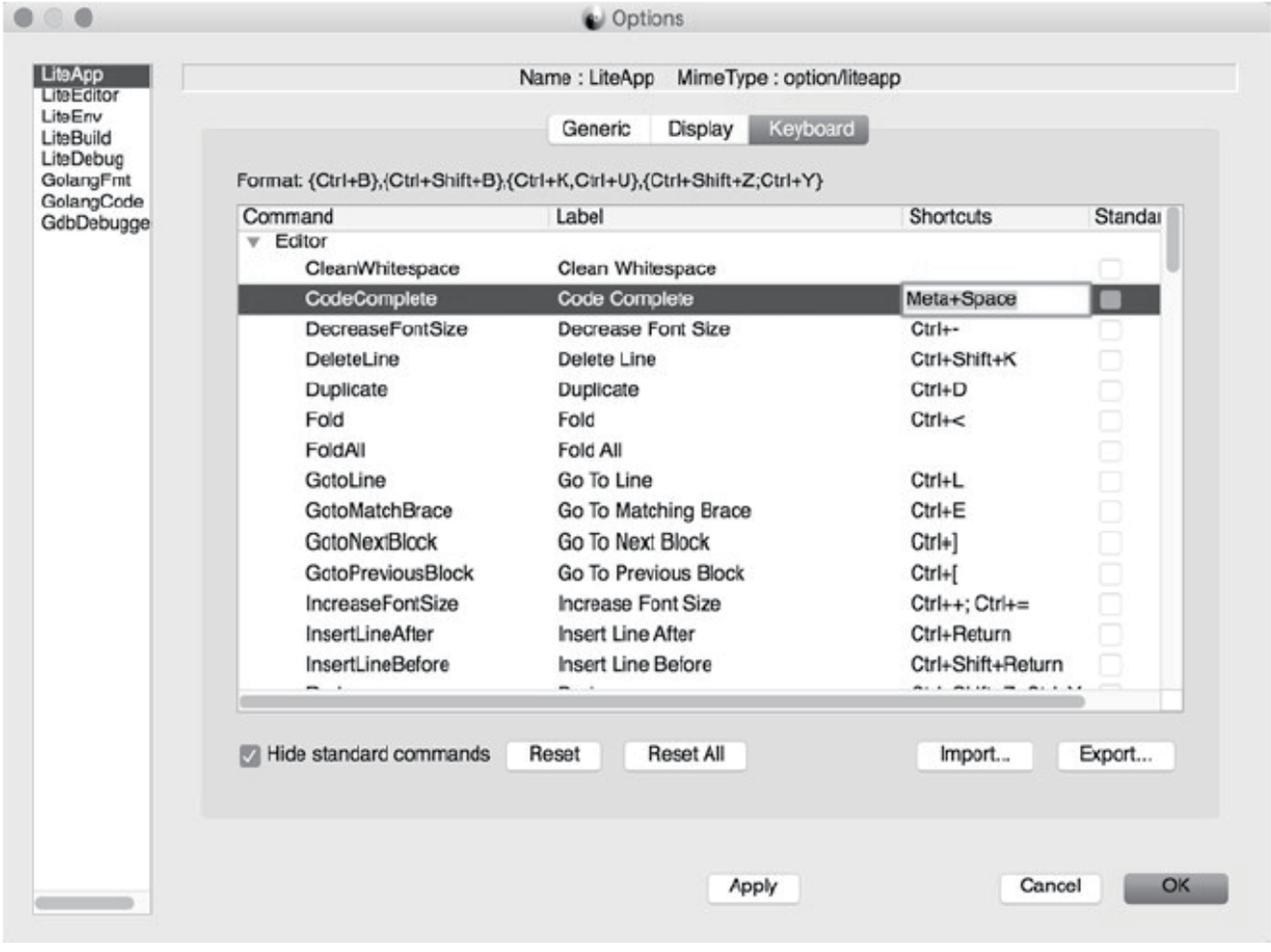


图3　LiteIDE快捷键设置

需要注意的是，LiteIDE的代码跳转依旧不够完善，只有当跳转到的代码所在的文件处于打 开（已经浏览过）状态时，才能正确地跳转。

Linux或者Mac OS X用户可以配合grep命令进行查看。如果我们要查找的函数是get_ instance，则命令如下：

```
grep -rl get_instance docker/api
grep -rl [grep_pattern] [file_diretory]
```

Windows用户可以在文件中使用查找功能。

相信现在你已经能使用LiteIDE比较方便地阅读源码了。

2. Sublime Text 2

Sublime Text 2（简称ST2）自2012年发布以来，便以其简洁华丽的外观、多平台的支持、多 语言支持以及超强的可扩展性而风靡起来。同样，作为一款文本编辑器，与VIM相比，它自带有 目录树等功能，省去了一些配置上的麻烦，易于上手。

下载

打开Sublime Text的官方网站，页面下方映入眼帘的就是一个大大的Download图标，它会根 据操作系统的平台自动选择适合的版本，只需直接点击Download即可下载。当然我们也可以打开下载列表页面（http://www.sublimetext.com/2），根据平台选择合适的版本进行下载。

安装

不同系统的用户可以参照不同的方式下载安装Sublime Text 2，方法如下。

- Windows：下载后打开exe文件，按照引导界面进行安装即可。
- Mac OS X：下载后打开dmg文件，按照引导界面进行安装即可。
- Linux：下载后进行解压缩，解压缩后进入目录点击sublime_text即可使用。但这样不便 于使用，可以按照以下步骤进行配置，以便在终端也可方便地使用Sublime（假设已经解 压缩到HOME目录）。

  ```
  mv $HOME/Sublime\ Text\ 2/ /opt/ sudo ln -s /opt/Sublime\ Text\ 2/sublime_text /usr/local/bin/subl
  ```

以上配置完成后，我们便可以方便地在终端通过subl命令打开Sublime Text 2了。

插件安装

Package Control

- 点击菜单栏偏好设置（Preferences）->包浏览（Browse Packages）。
- 打开它的上一级目录Sublime-Text-2/，可以看到“Installed Packages”、“Packages”等文 件夹。
- 下载Control.sublime-package包，并且复制到Installed Packages/目录中。
- 重启Sublime Text。

GoSublime

- 安装完Package Control后，默认情况下，通过快捷键Ctrl+Shift+P①即可调出命令执行模块，输入Package Control:Install Package即可激活包安装。
- 在跳出的可用包列表中，键入GoSublime后按回车键，即可下载并自动安装。
- 安装完GoSublime后，即可使用诸如代码补全、格式调整等功能。

CTags

- 与GoSublime相同，通过快捷键Ctrl+Shift+P调出命令执行模块，输入Package Control:Install Package，在跳出的可用包列表中键入CTags，再按回车键即可自动安装。
- 安装完成后，在Docker项目的根目录下点击鼠标右键，执行CTags：Rebuild Tags，对项 目进行CTags检索。
- 通过快捷键Ctrl+Shift+ . 即可进行代码跳转（跳转到定义），通过快捷键Ctrl+Shift+,即可 跳回。

至此，Sublime的介绍就完成了，熟练运用以上3个插件，不仅便于阅读Docker源码，对于Go 语言项目的编写也会有非常大的帮助。

3. Vim

相信习惯使用文本编辑器的读者，一定对开源软件Vim相当熟悉和亲切。Vim被誉为“编辑 器之神”，学习曲线极为陡峭，但是，一旦熟练掌握了Vim自成体系的一套快捷键，代码编辑速 度将快速提升，你也会从此对Vim爱不释手。本节只作为对Vim老用户的抛砖引玉，不适于Vim 的初学者使用。

插件的安装

go-vim是一款让Vim可以高度支持Go语言的Vim插件，所以也是要使用Vim作为Go IDE的必 装插件。

如果使用pathogen对插件进行管理，那么只要执行如下步骤即可。

```
cd ~/.vim/bundle
git clone https://github.com/fatih/vim-go.git
```

对于Vundle用户，需要在.vimrc文件中加入下面这行：

```
Plugin 'fatih/vim-go'
```

并且打开Vim，在命令模式下执行:PluginInstall①)。

安装完插件后，为确保所有依赖的二进制文件（如gocode、godef、goimports等），可以在 命令模式下执行如下命令进行自动安装：

```
:GoInstallBinaries
```

除go-vim外，还有如下可选插件。

- 实时代码自动补全插件：YCM 或 neocomplete；
- 侧边栏类试图插件：tagbar；

- 代码段自动生成插件：ultisnips 或 neosnippet；
- 目录浏览器插件：nerdtree。

常用命令

- :GoDef [identifier]：用于代码跳转，默认会跳转的项为鼠标定位的函数或变量，后面 可以跟标识符进行跳转。
- :GoDoc：用于打开相关Go文档。
- :GoInfo：用于显示鼠标定位的标识符的变量类型。
- :GoFmt：用于对选中的代码进行Go格式化。
- :GoDeps：用于显示当前包依赖的其他包。
- :GoFiles：用于显示依赖当前包的其他包。
- :GoInstallBinaries：用于安装Go语言的Vim依赖项。
- :GoUpdateBinaries：用于更新Go语言的Vim依赖项。

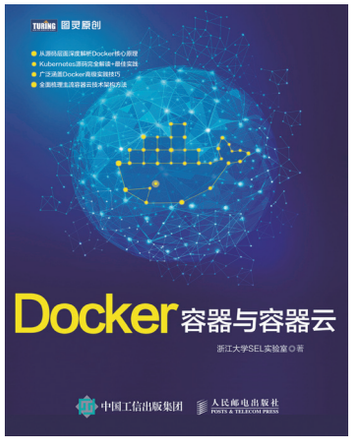
关于Vim的介绍到此结束，但相信对于Vim老用户来说，探索才刚刚开始。Vim作为一个经久 不衰、广受赞誉的好工具，一直都是每一个Geek心中最好的神兵利器！

4. Emacs

Emacs作为与Vim齐名的文本编辑器，号称“神之编辑器”，用来浏览和编写Go代码也是非常 方便的。本节也只作为Emacs老用户的抛砖引玉，在此之前，用户需要做好适合自己的配置。下 面我们就以Emacs24为例，简单介绍几个实用的插件，用户需要先使用go get命令安装好gofmt、godef、godoc、gocode等工具。

go-mode

go-mode在提供了自动缩进和语法高亮功能的基础上，还整合了Go语言自带的工具，如gofmt、godoc、godef等。在Emacs24以后



《Docker——容器与容器云》从实践者的角度出发，基于Docker 和 Kubernetes 最新源码，系统梳理了Docker容器技术和Kubernetes项目的实现原理和设计思路，有助于读者在实际场景中利用Docker容器和容器云解决问题并启发新的思考。本书节选自《Docker——容器与容器云》。

的版本中，可以使用自带的Package工具进行安装，命令如下：M-x package-install go-mode 下面我们主要介绍格式整理以及定义跳转两项功能的配置。

格式整理

格式整理功能直接调用了gofmt工具，该工具能使用户的代码风格与其他开发人员保持一致。 在Emacs中，用户可以直接调用gofmt命令，对当前窗口的代码进行格式整理。另一种方式是为 before-save-hook 添加函数，示例如下：

```
(add-hook 'before-save-hook 'gofmt-before-save)
```

这样在用户每次存档时就会自动进行格式整理。

定义跳转

定义跳转使用了godef工具，该工具能分析用户的代码、其他包内的代码以及Go标准库，实 现在这三者间的定义跳转。Emacs提供了godef-jump命令实现跳转，默认绑定键为C-c C-j，用户 也可以自己定义按键绑定，如绑定到F3键：

```
(add-hook 'go-mode-hook
    '(lambda () (local-set-key (kbd "<f3>") 'godef-jump)))
```

为了在跳转之后能跳转回来，用户可以添加如下配置，这样可以使用F2键回到原先的位置。

```
(add-hook 'go-mode-hook
    '(lambda () (local-set-key (kbd "<f2>") 'pop-tag-mark)))
```

此外，go-mode还提供了管理imports、使用godoc等工具，这里不再一一赘述。

company-go

company-go 调用 gocode 工具提供自动完成功能，用户可以直接使用 Package 工具安装 company-mode 和 company-go，并进行如下配置：

```
(add-hook 'go-mode-hook 'company-mode)
(add-hook 'go-mode-hook
    (lambda ()
    (set (make-local-variable 'company-backends) '(company-go))
    (company-mode)))
```

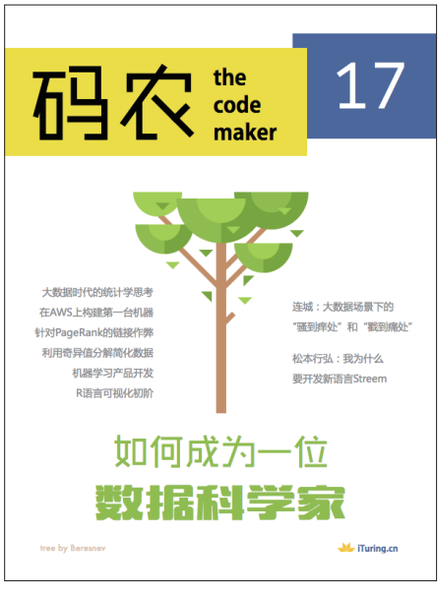
Emacs 还为用户提供了极大的自由度，建议用户使用最新版本的 Emacs 和插件，善用 Package 功能和网上贡献的工具，这样能获得最新的功能和更好的体验。■

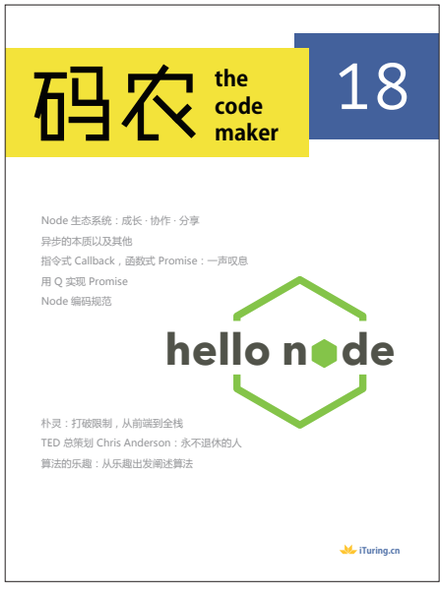
读《码农》吐吐槽 还能赚银子！

《码农》电子刊如今慢慢悠悠已经出版了20期，从一开始的好评如潮，到现在的“怎么这么抽象啊”“赶脚内容没有以前好了？”“一下就翻完了，没什么内容啊”……小编表示压力很大，现在的码农读者太难伺候了，明明是本免费杂志，￥%#%&*%@#￥*@￥#@！

但是，一看到好评，盼盼姐还是会热泪盈眶，热血沸腾，各种正能量啊“这么好质量的杂志还免费，天上掉馅饼呀”“希望一直出！每一版都读了，很值得推荐！！”……

所以《码农》还是会一直做下去，但是，是好是坏，您得给个话儿啊！





活动规则：在吐槽贴中留言，选出任何一期中你最喜欢的文章和最不喜欢的文章，即可获得图灵社区银子2两！凡吐槽吐得掷地有声者，加赠银子3两（共5两）！（怎样使用银子兑换图书）

活动时间：本活动长期有效。

Kubernetes负责人Dawn Chen：

开源是唯一的路

Dawn Chen是谷歌云平台软件工程师，目前负责Kubernetes项目。Dawn有超过8年的Google工作经验，负责过多层堆栈，包括内核，机管理，群集节点，管理，调度。供职谷歌前，她曾任职于 Veritas 和 Symantec。从大学开始在美国学习计算机的她，用中文讲解计算机术语有些困难。然而在中国和美国，不同的语言和文化共通的却是对女工程师的偏见和挑战。勇敢叛逆如她，Dawn从来没有质疑过自己作为一位工程师的资格和能力。她相信“偏见都是可以克服的，时代都是往前走的”，除了你自己没人能挡住你的路。

叛逆如她

除非你不想做，不想做就算了，想做好的话就不可能不劳而获。

问：您是从什么时候开始编程的？

我其实从小就很反对编程。为什么？可能因为我从小就比较叛逆。我父母是大学教授，因为我理工科特别好，所以他们就希望我去学计算机，他们觉得这是件比较容易的事。但是他们越想让我学我就越不想学。

我原来是学数理统计的，是数学系的。到了美国之后我申请统计系没有拿到奖学金，但是计算机给了我奖学金。那个时候就不需要叛逆了，于是我就去学了计算机。

这段经历现在对我的影响可能就是，我在头脑中都是用英文来思考计算机问题的，但是对于生活中任何别的事，我都是用中文思考的，分得很清楚。

问：能谈一下您在美国学习计算机的经历吗？

我后来去美国学了计算机，那个时候因为我数学很好（我本来是学数学的），所以就觉得计算机真的很容易。后来随着学习越来越深入，我发现计算机也变得越学越有意思了，尤其当我能用计算机解决生活中的实际问题时。那时候统计在各个领域还没有应用，因为还没有大数据处理，所以统计学毕业的话，可能只能去气象、制药厂，或者是金融机构。因为用计算机写程序解决的都是生活中的问题，我可以把问题自动化、模拟化，所以当时我也很高兴自己学了计算机。

我在美国研究生毕业的时候，正好赶上2000年第一次互联网泡沫。那个时候我也不知道为什么，很多人都去做web，可能是因为既简单，来钱又快。虽然我也去了硅谷，但是因为我比较叛逆的性格，我就不太想做web，我不太愿意做大家觉得接下来会很火的东西。

所以我当时就去了Veritas的一个研究实验室，Veritas后来被Symantec买进去了。这个实验室实际上主要是做存储的，就是给企业做集群管理。但是Veritas本身没有数据中心，它实际上还是传统的软件公司，把自己的东西打包卖出去。

我们当时的CTO非常有想法，我们做的项目叫效用计算，直到今天，计算机都没有办法实现效用计算。但是在15年前，我们在Veritas组建了用来做效用计算的很大的集群，那个时候我主要做算法。我们用的都是当时最先进的技术，比如端对端，我们想要通过端对端网络去进行更有效地连接。我们做了一些很有趣的东西，这些技术对我今天的工作来说都很有价值。

我在Veritas做过好多项目，但是当时让我印象最深的是，我们每一个项目做完，都没有办法部署出去。因为Veritas是以研究为导向的软件公司，而不是谷歌那样的服务型公司，所以这些项目既没有办法跟公司已有的软件结合在一起，也因为Veritas没有面向外界的基础设施，所以不能跟公司的基础设施结合。最后造成我们很多有意思的研究都没有办法应用。后来研究实验室也越来越不景气，因为CTO走了，然后公司被Symantec买了，也变得越来越以商业为导向。

于是我就离开了。我面试了几家公司，拿到了三个offer，一个是VMware，一个是谷歌，还有一个就是Facebook。当时Facebook还很早，还没有IPO，我同事介绍我去的是研究实验室。我的同事离开比

较早，他是Facebook里面的一个主架构师。他介绍我去的时候，就跟我说，他觉得Facebook的工作肯定能赚钱，但是喜不喜欢不一定，他后来也确实发财了。

我自己虽然有Facebook账户，但是我从来不用。然而从我读书第一天听说有搜索的时候（那时还没有谷歌），我就使用搜索，对我来说搜索改变了我很多。我学计算机比较晚，可能并不算一个典型的工程师，我对很多其他议题都很感兴趣，尤其是一些人文、社会的话题。除了工作，我只要下了班就不会再看计算机方面的东西。所以对我来说搜索是非常有用的，我对谷歌的印象特别好。

而VMware对我来说，是一个非常技术的公司，那时候虚拟技术和虚拟机其实都很火。而Facebook从钱的角度上说肯定也很好，我也相信它会成功。但是这两家公司对我来说都不像谷歌那样有吸引力。我觉得我需要做一个让我感兴趣的东西，然后找一家对我来说有触动的公司。我相信谷歌的理念，所以就去了谷歌。

问：去谷歌做的是什么工作？

我去谷歌的时候本来是做算法的，同一个组里有人在做系统网络，有人在做算法，还有人在做调度。大家都觉得算法更酷，更有意思。我第一次听一位技术主管讲容器技术时，我心里想：好酷，未来为什么不能用容器来取代虚拟机？于是我联系到了那位技术主管，他虽然是技术主管，但是那时候只有他一个人在干。他问我愿不愿意帮他做这个，我说好，我正好也想学。我当时其实也没有觉得我会一直做这个技术，我想有一天我还是会回去的吧。

随着工作展开，我感觉越做就越有意思，总有新的东西进来。因为这种技术是从无到有发展的，所以我们慢慢加入了越来越多的容器管理特性。很快我自己也成了技术主管。因为谷歌是一家技术驱动的公司，所以我

可以做很多决定。这些决定会造成很多影响，对公司其他人做的很多东西也会有影响。这份工作不仅让我感觉很有意思，还很有成就感，于是我就这样一直做了下去。

问：您认为女性程序员数量比较少的原因是什么？

我早晨起来穿衣服的时候会注意让自己穿得好看一点，我相信我老公也会注意到。但是我在工作中实际上会忘掉自己的性别。

话说回来，我相信无论在中国还是在美国，女工程师都会相对少一点。我认为这是一个传递途径（pipeline）的问题，最起码在美国是这样。读书的时候，尤其在大学，虽然女性读计算机科学的也没有男性多，但数量还可以。然而到了工作中、产业中之后，女性就会少很多。

总体上对于计算机产业里面女性来说，她们的努力（尤其在年轻的时候）不太容易被别人重视。比如当她提出一个观点的时候，无论是在美国还是在中国，别人都更容易忽略她的观点，就算她是对的，然后当另外一个男性工程师把同一个观点说出来时，就算他不一定有她说的好，大家还是会附和。

在我做第一份工作的时候，碰见了一个中国男性工程师，他是一个QA，而我是个开发者。他看见我非常不爽，他说一个年轻漂亮的女生，做得了程序员吗？我当时觉得，怎么会这样？他为什么会这样说？虽然我今天已经忘掉这些事情了，有些时候可能就是百毒不侵了，但当时我还是会被这样的事所影响。我虽然心里有些不高兴，但是我不会质疑自己，因为我从小到大数理化都是年级的第一名，我从来不会觉得我不能做程序员。我相信我可以的。

从另一个方面来说，我认为是家庭教育的问题。从读书的时候开始，家长可能就会说女孩子做这种工作很辛苦，女孩子要更重视家庭。其实，

虽然程序员工作时间长一些，但是也很灵活，你完全可以在完成工作的同时享受生活。我个人觉得，做任何事的核心理念就是不劳而获是永远不会成功的。除非你不想做，不想做就算了，想做好的话就不可能不劳而获。而父辈们一直在灌输给我们一种思想，女性要找工作，要照顾家庭。既要离家近，挣钱多，还要事少，在我看来这是根本不可能的。有得就要有失，我一直相信这一点。

所以我觉得做任何工作，如果想要自己感觉到很满意，有成就感和满足感，你都需要投入心力，无论是做工程师还是其他工作。我觉得是偏见让很多女性不愿意去做软件工程师，反过来在工作中也确实因为女性相对较少，从而加深了这些偏见。但是我觉得时间久了，这些偏见都是可以克服的。时代都是往前走的，不受影响了就好了。

谷歌本身是很重视这一点的。我刚加入谷歌的时候开年会，我印象很深，可能在100多人里面就我一位女性。但是现在就多了，同样在一百个工程师里面可能有十几位女性。

问：作为一位Google的员工，您怎么看Alphabet？

这完全是我个人的看法，我觉得这是一个很聪明的做法。我刚才也提到了谷歌是一家技术驱动的公司，谷歌很鼓励创新，但是同时谷歌也是一家上市公司，它需要对投资人负责。投资有很多种类，有些投资人看中了谷歌的主流业务，比如说广告、搜索。当他去买谷歌股票的时候，看到财政报告上说谷歌花了一堆钱在技术上，他会问：谷歌为什么要花那么多钱？谷歌在做无人驾驶汽车，做气球，做生命科学，我都觉得谷歌很棒。虽然这些东西我不懂，但是作为我一个工程师，我非常享受在谷歌工作。但反过来说，我也很支持也能理解华尔街，当投资人做投资的时候，他看重的是谷歌的主流业务和赚钱的能力，而不是谷歌做创新的能力。同时对于创新领域的VC来说，他们也可以选择不把投资绑在主流业务上。

另外我觉得这也是一种宣传，让大家觉得谷歌是一家创新的公司。Alphabet下面的某些子公司是非常创新的，在未来领导技术前沿上也是非常活跃的。同时，对于那些更看重主流业务的人，他们会觉得，谷歌非常专注于自己的核心业务。从另一方面来说，这也符合谷歌对于管理透明度的要求，对于外界来说钱投在哪里都是透明的。

开源是唯一的路

我们相信容器既然能让谷歌受益，也就能让别人受益。

问：您现在负责的是Kubernetes哪个部分？

Kubernetes是一个很大的开源项目，有好几个负责人，我是其中一个。我负责的主要就是容器技术这部分。除此之外，我们有负责调度的，负责API的，还有负责整个集群以及网络的。我负责的这部分包括管理节点在某台机器上的实现，以及保障每个组件的顺利运行。对于虚拟机，甚至是裸机来说，我们需要确定内核，库，还有容器技术的调度方式和应用方式。

问：您在全球容器技术大会的一个关门会议上讲到了谷歌为什么要开源Kubernetes，能不能把大致内容跟我们分享一下？

很多谷歌员工在离开了谷歌之后去了其他公司，比如Facebook、Twitter，或者腾讯。他们中的很多人都会遇到一个问题，新的公司为什么没有Borg？很多人都问过我：你们的Kubernetes为什么没有Borg的这个功能、那个功能？可见他们有多么喜欢Borg，所以有很多人在不同的公司去重新做Borg。比如Facebook的集群管理系统，实际是从谷歌开始的，这个系统很像Borg，包括里面有些内部名词都是一样的。

我们在几年前发表了一篇论文，是关于Omega的。Mesos就是基于这篇论文的一个开源项目，这个项目含有一些类似于集群管理的东西，但是更多着重在调度上，所以Mesos并不是一个完整的生态系统。Twitter自己的集群管理实际就是用的Mesos，再加上一些别的东西做成的一个生态管理系统。对于很多其他公司来说，也有类似的情况。

从经验上看，谷歌过去总体上不怎么开源，但是却发表了很多论文，比如说对于业界很重要的MapReduce、BigTable论文。有很多开源的人或者公司觉得论文很有价值，然后就想要重新装备这些系统。后来谷歌自己也发现了这点，但是我们还发现重新组装的那些系统虽然很像我们的系统，但是又不像，这些系统其实跟我们的系统并不兼容，尤其在API方面。所以作为一家越来越重视云的公司，谷歌也希望能在云方面有所作为。

很多开源项目都建立在谷歌的论文上，但是却又跟我们本身的系统不兼容。谷歌觉得这一点很难办，如果只是发表论文，而不去做一个真实的系统的话，问题是很难解决的。就算我们开源API，但是没有实现，甚至没有示例实现，那么API就是不成立的。综上所述，我们觉得谷歌应该重新检视下关于是否开源的决定。并不是说谷歌不愿意开源，否则它也不会去发表论文，最重要的问题在于开源需要太多的人力和物力了。

但是反过来说，如果真的要在云上面做文章，我们是没有办法不开源的。容器技术如果不开源的话，我们就应该做到让用户完全信任容器技术，能够不担心安全问题。如果用户能够完全信赖容器技术，我们就不需要提供虚拟机了，也就是说用户可以不需要SSH这样的机器了。但是我们做不到，现在的容器技术做不到。而且就算能做到，我们也没有办法去说服使用者，要他们接受纯粹的服务而不是一台机器。用户会很担心：程序到底在哪里运行？他们会不会偷了我的东西？因为容器技术还没有得到大家的广泛认可，尤其在安全性能方面。所以我们还是需要卖给用户虚拟机，那么用户就有可能会SSH到一个机器。如果用户看到一个隐藏的程序在运行，他会很惊慌，因为他不知道我是不是要偷他的商业机密。

那我们唯一能做的事情是什么？就是开源。我们大家都相信无论是谷歌的云还是别人的云，都需要有一个集群管理。我们也相信容器既然能让谷歌受益，也能让别人受益。

问：Kubernetes跟Borg有很多相同的地方，但是您能说一下它们在设计理念和设计目的上的主要不同吗？

在我看来最主要不同就是API。我们可以说Borg的高层是描述性的，但是在Borg真正实现的组件之间，实际上是命令性的API。在我们意识到这个问题之后，我们在设计Kubernetes时就坚持用描述性的API。

还有一点不一样的就是，因为使用了描述性API，所以内部实现不需要有非常复杂的状态机，我们使用了一个比较简单的调和控制回路。所谓描述性就是形容你想要的是什么状态，最终要的是什么结果，然后你的调和控制回路（也就是控制系统）就知道了这个目标，它会根据现在状况调整，一直驱动达到理想的状态。比如你在调度的时候，需要考虑有多少个job在运行，job是在什么情况下运行，有多少个copy在运行，少了一个copy你就多加一个，多了一个你就杀死一个。这两点就是主要贯彻在整个设计里面的原则。

问：Kubernetes开源之后会有什么变化？它跟Mesos是不是会越来越不一样？

我们其实不太关心Kubernetes跟Mesos是不是会越来越不一样。因为Mesos的调度今天已经做得非常成熟和完善了。与此相对的是，Kubernetes从一开始就是以一个生态系统为目的而设计的，它有正常检查、监控、记录，它有所有的东西。Mesos是基于Omega那篇论文构建的，而Omega的论文的重点就是调度的算法。所以Mesos自己不是一个完整的生态系统，也不是一个集群管理系统。Mesos能把你要做的东西建立在很多要求上，部署到机器上，但是当你要做升级，需要扩

大规模或减小规模时，Mesos是做不到的。Mesos需要跟很多别的东西结合，比如现在的Mesos实际上是跟Kubernetes结合的，你可以使用Mesos非常酷炫的调度功能，同时也可以用到Kubernetes自己的集群管理功能。除此之外，Mesos也跟马拉松结合。

当然，我们的Kubernetes也会越来越完善，因为算法我们都知道，我们自己内部的算法也很复杂。我相信Mesos也会逐渐提升它的东西。我们都是开源的项目，其实是种互补的关系，或者说是一种良性竞争的关系。所以未来会怎么样？没有人知道。■

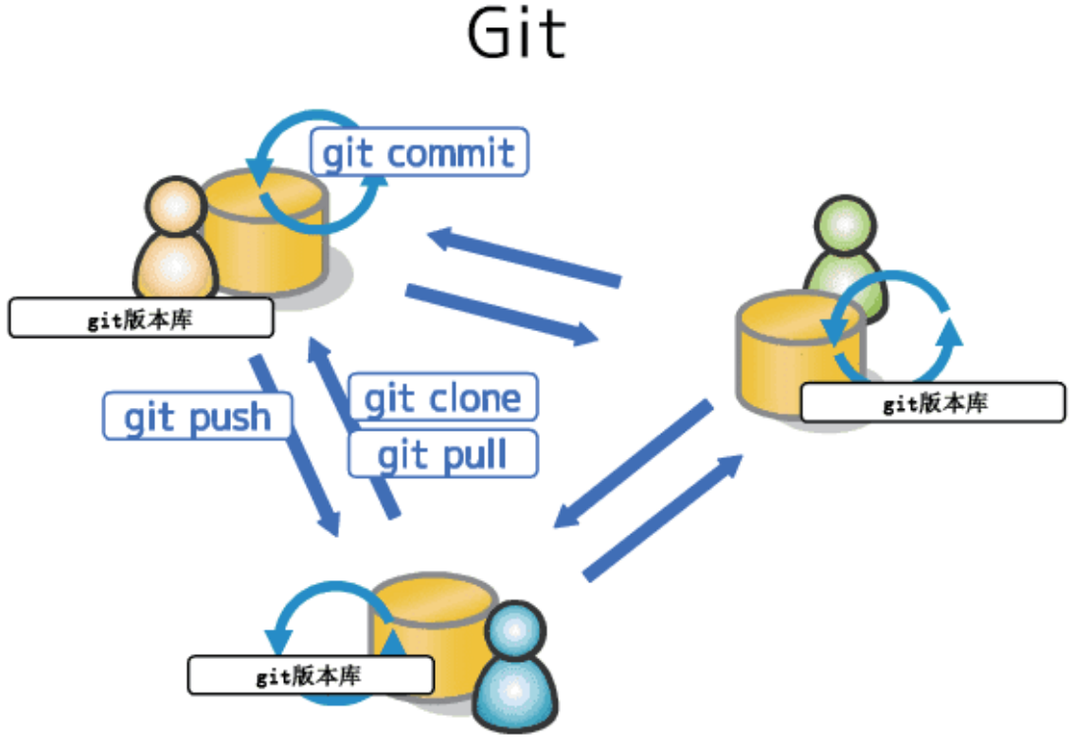
Git中级用户的25个提示

作者 / Andy Jeffries

Ruby on Rails 开发者 & 跆拳道教练，生活在英国伦敦。

我使用 Git 大约已经有 18 个月时间，自认为能很好地驾驭它了。但是当我们请到 GitHub 的 Scott Chacon 来到 LVS 公司（一个博彩 / 游戏软件供应商 / 开发商）做专场培训时，我在第一天就学到了大量的东西。

由于有些人总是对使用 Git 自我感觉良好，因此，我想分享一些我从社区获取到的 Git 精品，这样就可能会帮助那些人无需浪费大量研究时间而直接找到答案。



基本提示

1. 安装之后第一步

安装 Git 之后，你要做的第一件事情就是去配置你的名字和邮箱，因为每一次提交都需要这些信息：

```
$ git config --global user.name "Some One"
$ git config --global user.email "someone@gmail.com"
```

2. Git 是以指针为基础

存储在 git 中的所有东西都包含在一个文件中。当你提交的时候，git 会创建一个包含提交消息和相关数据的文件（名称、邮件、日期/时间、上一次提交等等），并将其链接到一个树形文件。树形文件包含一个对象列表或者其它树。对象或二进制大数据对象（BLOB）是提交的真正内容（一个文件，如果你愿意，虽然文件名没有存储在对象中，但会存储在树中）。所有这些文件都以对象的 SHA-1 哈希为文件名进行存储。

分支和标签只是一些文件，这些文件包含（基本上）一个指向提交的 SHA-1 哈希值。使用这些引用在灵活性和速度上均有大幅提升，创建一个新的分支就和创建一个文件一样简单，只是这个文件带有分支名称和一个包含指向提交（你从这个提交建立分支）的SHA-1 哈希值。当然，当你使用 Git 命令行工具（或一个图形用户界面）时，你永远也不会这么做，但它就是这么简单。

你可能已经听说过对 HEAD 的引用。它只是一个包含 SHA-1 引用的文件，这个引用指向你当前的提交。如果你正在解决一个合并冲突问题，查看一下 HEAD，你会发现，它与一个特定的分支或分支上的特定点无关，只和你现在的位置有关。

所有的分支指针保存在 .git/refs/heads 目录下，HEAD 在 .git/HEAD 目录下，标签在 .git/refs/tags 目录下 - 你可以随意看看。

3. 两个 Parents - 当然！

当在日志文件中查看一个合并提交的消息时，你会看到两个 parents（与正常提交相比）。第一个parent 是你所在的分支，第二个 parents 是你并入的分支。

4. 合并冲突

到目前为止，我确信你一定有一个合并冲突需要解决。通常情况下，通过编辑该文件，删除文件中的<<<<, ====, >>>>标记，然后保存你需要保留的代码就可以了。有时候，在任何变更之前查看代码都是一个值得推荐的做法，比如，在你对两个有冲突的分支采取行动之前。这是又一个命令：

```
$ git diff --merge
diff --cc dummy.rb
index 5175dde,0c65895..4a00477
--- a/dummy.rb
+++ b/dummy.rb
@@@ -1,5 -1,5 +1,5 @@@
  class MyFoo
    def say
-     puts "Bonjour"
 -    puts "Hello world"
++    puts "Annyong Haseyo"
    end
  end
```

如果文件是二进制的，文件比较就不是那么容易了...你通常要做的是尝试每个版本的二进制文件，并决定使用哪一个（或者在二进制文件编辑器

手动复制部分内容)。从一个特定分支下 pull 一个文件副本 (如果你要合并主分支和分支132的话):

```
$ git checkout master flash/foo.fla # or...
$ git checkout feature132 flash/foo.fla
$ # Then...
$ git add flash/foo.fla
```

另一种方法是从 git 中查看这个文件 - 你能够以其他文件名的方式进行查看, 然后将正确的文件 (当你确定它是哪一个时) 复制到正常的文件名中:

```
$ git show master:flash/foo.fla > master-foo.fla
$ git show feature132:flash/foo.fla > feature132-foo.fla
$ # Check out master-foo.fla and feature132-foo.fla
$ # Let's say we decide that feature132's is correct
$ rm flash/foo.fla
$ mv feature132-foo.fla flash/foo.fla
$ rm master-foo.fla
$ git add flash/foo.fla
```

更新: 感谢 Carl 在早先的的博客评论中给与的提醒, 你实际上能使用“git checkout —ours flash/foo.fla”和“git checkout —theirs flash/foo.fla”检出一个特定的版本而不需要记住你要合并到哪一个分支。我个人更喜欢更明确些, 但是你可以随便选择...

在解决了合并冲突问题之后 (就像我上面所做的那样), 请记得将这个文件添加给索引。

服务器、分支和标签

5. 远程服务器

Git 最强大的功能之一是可以有一个以上的远程服务器 (另一个事实, 你总是可以运行一个本地仓库)。你不一定总是需要写访问权限, 你可以从

多个服务器中读取 (用于合并)，然后写到另一个服务器中。添加一个远程服务器很简单：

```
$ git remote add john git@github.com:johnsomeone/someproject.git
```

如果你想查看远程服务器的相关信息，你可以这样做：

```
# shows URLs of each remote server
$ git remote -v

# gives more details about each
$ git remote show name
```

你可以查看本地分支和远程分支之间的差别：

```
$ git diff master..john/master
```

你也能查看不在远程分支上的 HEAD 的变化：

```
$ git log remote/branch..
# Note: no final refspec after ..
```

6. 标签

在 Git 中存在两种类型的标签 - 一个轻量级标签和一个注解标签。记着第二个提示中说过 Git 是基于指针的，二者的区别很简单。一个轻量级标签无非是一个指向提交的具名指针。你可以改变它并指向另一个提交。一个注解标签是一个指向标签对象的具名指针，这个标签对象拥有自己的消息和历史。如果有需要，标签对象的消息可以采用 GPG 加密签名。

创建两种类型的标签其实很容易 (只是一个命令行选项的差异)

```
$ git tag to-be-tested
$ git tag -a v1.1.0 # Prompts for a tag message
```

7. 创建分支

在 Git 中创建分支非常容易（闪电般的速度，因为它仅仅需要创建一个不到100字节的文件）。创建一个新分支并切换过去的通用写法是：

```
$ git branch feature132
$ git checkout feature132
```

当然，如果你知道你要马上切换过去，你可以使用一条命令就能做到：

```
$ git checkout -b feature132
```

如果你要重命名一个本地分支，同样是件容易的事（长命令方式用来显示具体执行过程）：

```
$ git checkout -b twitter-experiment feature132
$ git branch -d feature132
```

更新：或者你（就像 Brian Palmer 在博客文章评论中指出的那样）只使用“git branch”和 -m 选项就可以一步到位：

```
$ git branch -m twitter-experiment
$ git branch -m feature132 twitter-experiment
```

8. 合并分支

在将来某个时候，你想要合并你的变更。有两种方式可以实现：

```
$ git checkout master
$ git merge feature83 # Or...
$ git rebase feature83
```

merge 和 rebase 的区别在于，merge 试图解决变更而且创建一个融合后的新提交，而 rebase 则试图把自你上次在其他分支上的变化，在另一个分支的 HAED 上重现。但是，在你向远程服务器推送一个分支之后，不要进行 rebase 操作 - 这会引发混淆 / 问题。

如果你不能确定哪些分支仍然有独立的工作在进行 - 以便你能知道你需要合并哪一个分支以及删除哪些分支，git branch 命令有两个选项可以帮助实现这一点：

```
# Shows branches that are all merged in to your current branch
$ git branch --merged

# Shows branches that are not merged in to your current branch
$ git branch --no-merged
```

9. 远程分支

如果你有一个本地分支，你想让它出现在远程服务器上，你可以使用一个推送命令：

```
$ git push origin twitter-experiment:refs/heads/twitter-experiment
# Where origin is our server name and twitter-experiment is the branch
```

更新：感谢 Erlend 在博客文章评论中提到的 - 这实际上和 git push origin twitter-experiment 达到的效果的一样，但是通过使用全部语法，你能看到你实际上在两端使用了不同的名字（你的本地名字可能是 add-ssl-support，而远程名字可能是 issue-1723）。

如果你想删除一个远程服务器上的分支（请注意分支名称之前的冒号）：

```
$ git push origin :twitter-experiment
```

如果你想显示所有远程分支的状态，你能像这样查看它们：

```
$ git remote show origin
```

这可能会列出一些服务器上曾经有过但现在已不存在的分支。如果碰到这种情况，你可以很轻松地使用如下命令从本地检出并将其删除：

```
$ git remote prune
```

最后，如果你有一个远程分支，你想在本地进行跟踪它，通常的做法是：

```
$ git branch --track myfeature origin/myfeature
$ git checkout myfeature
```

然而，如果你使用 -b 标识符去检出的话，新版的 Git 会自动建立跟踪：

```
$ git checkout -b myfeature origin/myfeature
```

在临时存放区、索引和文件系统中保存内容

10. 临时存放 (Stashing)

在Git中，你可以把当前的工作状态储存在一个临时的存储区域堆栈，然后重新加以利用。简单的案例如下：

```
$ git stash # Do something...
$ git stash pop
```

很多人推荐使用 git stash apply 来代替“pop”，然而如果你真这么做的话，你最终得到一个长长的毫无用处的储藏清单。如果对它进行清理，“pop”只会把它从堆栈中删除。如果你已经使用了 git stash apply，你可以使用如下命令从堆栈中删除最后一项：

```
$ git stash drop
```

Git 会基于当前的提交消息自动创建一个注释信息。如果你更喜欢使用一个自定义的消息（因为它可能和之前的提交无关）：

```
$ git stash save "My stash message"
```

如果你想从你的列表中（不必是最后一个）对一个特定的 stash 加以利用，你可以列出它们并像这样来使用它：

```
$ git stash list
  stash@{0}: On master: Changed to German
  stash@{1}: On master: Language is now Italian
$ git stash apply stash@{1}
```

11. 交互式添加

在 Subversion 的世界里，你修改文件然后只是提交有变化的文件。而在 Git 的世界里，你在提交某些文件甚至某些补丁上有更多的控制权。为了提交某些文件或者文件的某些部分，你必须进入交互模式。

```
$ git add -i
           staged     unstaged path


*** Commands ***
  1: status      2: update   3: revert   4: add untracked
  5: patch      6: diff     7: quit     8: help
What now>
```

这会让你进入一个基于交互式命令的菜单模式。你可以使用命令的数字符号或者加亮字符（如果你开启颜色高亮显示功能的话）进入对应模式，然后就是正常输入文件数的问题了（你可以使用像 1 或 1-4 或 2,4,7 这样的格式）。

如果你想进入修补模式（交互模式下输入‘p’或‘5’），你也可以直接进入那个模式：

```
$ git add -p
diff --git a/dummy.rb b/dummy.rb
index 4a00477..f856fb0 100644
--- a/dummy.rb
+++ b/dummy.rb
@@ -1,5 +1,5 @@
 class MyFoo
   def say
-    puts "Annyong Haseyo"
+    puts "Guten Tag"
   end
 end
Stage this hunk [y,n,q,a,d,/,e,?]?
```

如你所见，在底部你得到一系列选项为选择去添加文件改变的部分，这个文件的所有变化等等。使用‘?’命令可以了解选不同选项的解释。

12. 存储 / 从文件系统检索

一些项目（例如 Git 项目自身）直接在 Git 文件系统中存储额外的文件而不必是检入文件。

让我们开始在 Git 中存储一个任意文件：

```
$ echo "Foo" | git hash-object -w --stdin
51fc03a9bb365fae74fd2bf66517b30bf48020cb
```

此时，该文件对象已在数据库中，但是如果你不设置（一些东西）指向那个文件对象，它将被作为垃圾而回收。最简单的方法是标记它：

```
$ git tag myfile 51fc03a9bb365fae74fd2bf66517b30bf48020cb
```

既然在这里我们已经标记了 myfile。当我们需要获取该文件时，我们可以这样做：

```
$ git cat-file blob myfile
```

程序员可能经常用到的工具文件（密码、GPG 密钥、等等），不需要每次都检出到磁盘上（特别是在生产环境下），这种方法非常有效。

日志记录

13. 查看日志

如果你不使用‘git log’查看最近提交历史的话，你就不能长时间顺利地使用 Git。但是，也存在一些如何更好使用它的建议。例如，你可以查看每次提交中改变的一个补丁：

```
$ git log -p
```

或者你可以只是查看一个哪些文件有所更改的概述：

```
$ git log --stat
```

你可以在一行中设置一个不错的别名，用于显示简短的提交和漂亮的带有消息的分支图（像 gitk，但在命令行上）：

```
$ git config --global alias.lol "log --pretty=oneline --abbrev-commit --graph
--decorate"
$ git lol
* 4d2409a (master) Oops, meant that to be in Korean
* 169b845 Hello world
```

14. 检索日志

如果你想在日志中查询一个特定作者，你可以这样指定：

```
$ git log --author=Andy
```

更新：感谢 Johannes 的评论，我终于化解了一部分困惑。

或者如果你有一个搜索词出现在提交消息中：

```
$ git log --grep="Something in the message"
```

还有一个功能更强大的叫 pickaxe 的命令，它可以查找条目用来添加或删除一个特定的内容（也就是，当它第一次出现或被删除的时候）。这样你就可以知道何时增加了一行（但是如果那一行中的字符随后被改变，你将无从得知）：

```
$ git log -S "TODO: Check for admin status"
```

如果你改变一个特定的文件会怎么样呢，例如 lib/foo.rb

```
$ git log lib/foo.rb
```

比如说你有一个 feature/132 分支和一个 feature/145 分支，你想查看在这些分支但却不在主分支上的提交（备注：^ 代表非）：

```
$ git log feature/132 feature/145 ^master
```

你也可以使用 ActiveSupport 风格的日期缩小日期范围：

```
$ git log --since=2.months.ago --until=1.day.ago
```

它默认使用 OR 模式来组合查询，但是你也可以很轻松地改为 AND 模式（如果你的查询项不止一个的话）

```
$ git log --since=2.months.ago --until=1.day.ago --author=andy -S "something" --all-match
```

15. 选择查看 / 修改的版本

当引用一个修订版本时，你有许多选项可以选择，当然，这取决于你对此功能的了解程度：

```
$ git show 12a86bc38 # By revision
$ git show v1.0.1 # By tag
$ git show feature132 # By branch name
$ git show 12a86bc38^ # Parent of a commit
$ git show 12a86bc38~2 # Grandparent of a commit
$ git show feature132@{yesterday} # Time relative
$ git show feature132@{2.hours.ago} # Time relative
```

请注意，和上一节有所不同，在行尾的脱字符表示提交的 parent - 行首的脱字符则表示不在这个分支上。

16. 选择一个范围

最简单的方法是这样来用：

```
$ git log origin/master..new
# [old]..[new] - everything you haven't pushed yet
```

你也可以删除 [new]，这将使用当前的 HEAD。

时间回退和错误修复

17. 重置更改

如果你还没有提交一个更改，你可以很容易地重置它：

```
$ git reset HEAD lib/foo.rb
```

通常使用‘unstage’作为别名比较好，因为它不是那么显而易见。

```
$ git config --global alias.unstage "reset HEAD"
$ git unstage lib/foo.rb
```

如果你已经提交了文件，你可以做两件事情 - 如果是最后一次提交，你可以这样来修改：

```
$ git commit --amend
```

这将回滚到最后一次提交，让你的工作副本回到变化存储在暂存区的状态，你可以编辑提交消息准备下一次提交。

如果你的提交不止一次，并且只想完全回滚它们，你可以重置分支回到之前的时间点。

```
$ git checkout feature132
$ git reset --hard HEAD~2
```

如果你真的想把分支指向一个完全不同的 SHA-1（也许你把一个分支的 HEAD 指向另一个分支，或者进一步提交），你可以按照以下方式去做：

```
$ git checkout FOO
$ git reset --hard SHA
```

实际上还有一种更便捷的方式（因为它不会先将你的工作副本变回最初 FOO 状态，然后再指向 SHA）：

```
$ git update-ref refs/heads/FOO SHA
```

18. 提交到错误的分支

好吧，让我们假设你提交到主分支，但应该已经创建了一个叫做 experimental 的主题分支。为了移除这些变化，你可以在当前点创建一个分支，回退 HEAD，然后检出新的分支：

```
$ git branch experimental   # Creates a pointer to the current master state
$ git reset --hard master~3 # Moves the master branch pointer back to 3 revisions ago
$ git checkout experimental
```

如果你已经在一个分支的一个分支的一个分支等上面做了些变更，这将会更复杂。然后你需要做的就是在这个分支上将其变更 rebase 到另一个的地方：

```
$ git branch newtopic STARTPOINT
$ git rebase oldtopic --onto newtopic
```

19. 交互式 rebasing

这是一个很酷的特性，我之前已看过演示，但当时没有真正搞明白，现在来看其实很简单。比方说，你已做了3次提交，但是你想对它们进行重新排序或者编辑（或者合并它们）：

```
$ git rebase -i master~3
```

然后你将编辑器打开。你所要做的就是修改 “pick/squash/edit 的指令来进行如何提交，然后保存 / 退出。在编辑之后，你可以使用 git rebase —continue 让你的每一个指令一个一个进行。

如果你选择编辑一个文件，这会让你停留在你提交时的状态，因此你需要使用 git commit —amend 对它进行编辑。

备注：在 REBASE 过程中不要进行提交工作 - 只能添加然后使用 —continue, —skip or —abort 选项。

20. 清理

如果你已经提交了一些内容到你的分支中（也许你是从SVN中的旧代码库导入的），你想从历史中删除掉所有的已提交内容：

```
$ git filter-branch --tree-filter 'rm -f *.class' HEAD
```

如果你已经向远程服务器推送过代码，但自那之后提交的都是一些垃圾，在推送之前你可以在本地系统上执行这样的操作：

```
$ git filter-branch --tree-filter 'rm -f *.class' origin/master..HEAD
```

各种各样的提示

21. 之前你看过的引用

如果你知道你之前已经查看过一个 SHA-1，但是你已经做了一些重置/回退工作，你可以使用 reflog 命令去查看你最近看过的 SHA-1：

```
$ git reflog
$ git log -g # Same as above, but shows in 'log' format
```

22. 分支命名

一个可爱的小提示 - 请记住，分支的名字并不局限于 a-z 和 0-9 这些字符。名字中可以使用 / 和 . 来伪装命名空间或者版本号，例如：

```
$ # Generate a changelog of Release 132
$ git shortlog release/132 ^release/131
$ # Tag this as v1.0.1
$ git tag v1.0.1 release/132
```

23. 寻找谁是始作俑者

寻找谁更改了一个文件中的一行代码经常会用到。简单命令如下：

```
$ git blame FILE
```

有时更改来自于前一个文件（如果你已经合并了两个文件，或者你已经移动了一个函数），因此你可以这样用：

```
$ # shows which file names the content came from
$ git blame -C FILE
```

有时通过向前或向后点击来进行变化跟踪，这是很好的方法。有一个内置的 GUI 程序专门为此设计：

```
$ git gui blame FILE
```

24. 数据库维护

Git 通常不需要大量维护，它基本上可以自我维护。然而，你可以使用如下命令查看数据库统计信息：

```
$ git count-objects -v
```

如果数值很高，你可以选择使用垃圾回收你的重复内容。这不会影响推送或者其它用户，但却可以让你的命令运行更快且占用更少空间：

```
$ git gc
```

译者 / 傅俊杰

Qingniu/ 青牛，码农，复唧唧联合创始人，业余时间喜欢研究中国传统文化。博客@明珠夜话。

经常运行一致性检查也是值得推荐的做法：

```
$ git fsck --full
```

你也可以在行尾添加一个 —auto 参数（如果你频繁运行它，或者在你的服务器上每日从 crontab 中运行它），如果统计数据表明必须进行要进行一致性检查，只要 fsck 命令就行。

如果检查“dangling”或“unreachable”的结果一切正常，这经常是由于回退 HEAD 或 rebasing 的结果。如果检查“missing”或“sha1 mismatch”出了问题...寻求专业帮助吧！

25. 恢复一个丢失的分支

如果你使用 -D 选项删除了一个分支 experimental，你可以重新创建它：

```
$ git branch experimental SHA1_OF_HASH
```

你可以使用 git reflog 来发现一个 SHA-1 哈希值，如果你近期访问过它的话。

另一种方法是使用 git fsck —lost-found。一个悬空的提交就是一个 lost HEAD（它只会是一个已删除分支的 HEAD，因为当一个 HEAD^ 被 HEAD 引用时，它就没有悬空）。■

阅读英文原文

阅读图灵社区原文

Apache Spark 入门攻略

作者 / Ashwini Kuntamukkala

Ashwini 是一位开源和云技术爱好者。在构建企业级解决方案方面，他有 10 年以上的经验，涉及的领域包括旅游业、制药业、医疗业。他现在在 SciSpike 任软件架构师。工作之余，他经常在博客上分享他在解决挑战性问题的过程中获得的新发现。

时至今日，Spark 已成为大数据领域最火的一个开源项目，具备高性能、易于使用等特性。然而作为一个年轻的开源项目，其使用上存在的挑战亦不可为不大，这里为大家分享 SciSpike 软件架构师 Ashwini Kuntamukkala 在 Dzone 上进行的 Spark 入门总结（虽然有些地方基于的是 Spark 1.0 版本，但仍然值得阅读）—— Apache Spark: An Engine for Large-Scale Data Processing，由 OneAPM 工程师翻译。

本文聚焦 Apache Spark 入门，了解其在大数据领域的地位，覆盖 Apache Spark 的安装及应用程序的建立，并解释一些常见的行为和操作。

一、为什么要使用 Apache Spark

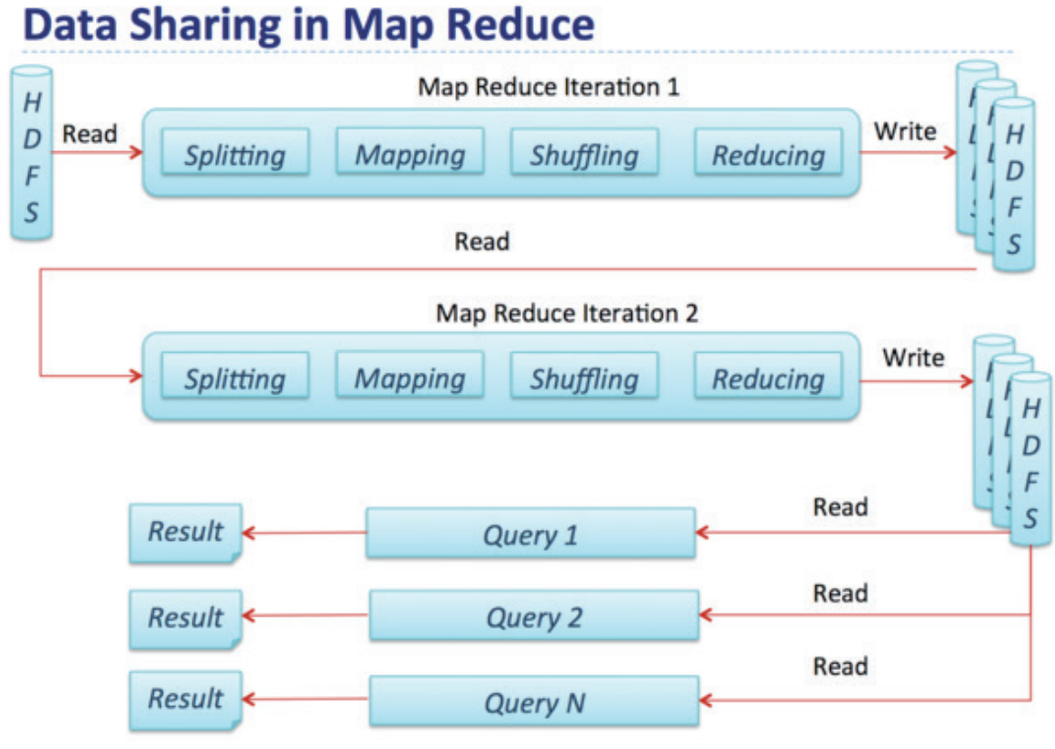
时下，我们正处在一个“大数据”的时代，每时每刻，都有各种类型的数据被生产。而在此紫外，数据增幅的速度也在显著增加。从广义上看，这些数据包含交易数据、社交媒体内容（比如文本、图像和视频）以及传感器数据。那么，为什么要在这些内容上投入如此多精力，其原因无非就是从海量数据中提取洞见可以对生活和生产实践进行很好的指导。

在几年前，只有少部分公司拥有足够的技术力量和资金去储存和挖掘大量数据，并对其挖掘从而获得洞见。然而，被雅虎 2009 年开源的 Apache Hadoop 对这一状况产生了颠覆性的冲击——通过使用商用服

务器组成的集群大幅度地降低了海量数据处理的门槛。因此，许多行业（比如医疗、基础设施、金融、保险、远程信息处理、消费者、零售、营销、电子商务、媒体、制造和娱乐）开始了 Hadoop 的征程，走上了海量数据提取价值的道路。着眼 Hadoop ， 其主要提供了两个方面的功能：

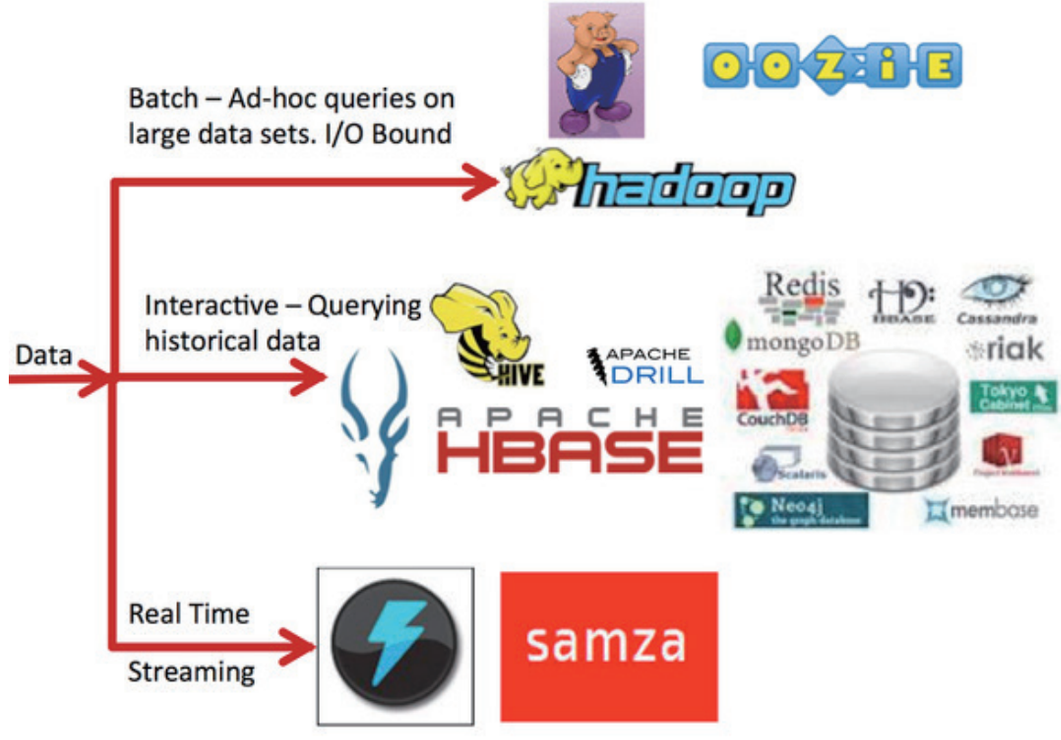
- 通过水平扩展商用主机，HDFS 提供了一个廉价的方式对海量数据进行容错存储。
- MapReduce 计算范例，提供了一个简单的编程模型来挖掘数据并获得洞见。

下图展示了 MapReduce 的数据处理流程，其中一个 Map-Reduce step 的输出将作为下一个典型 Hadoop job 的输入结果。



在整个过程中，中间结果会借助磁盘传递，因此对比计算，大量的 Map-Reduced 作业都受限于 IO 。然而对于 ETL 、数据整合和清理这样的用例来说，IO 约束并不会产生很大的影响，因为这些场景对数据处理时间往往不会有较高的需求。然而，在现实世界中，同样存在许多对延时要求较为苛刻的用例，比如：

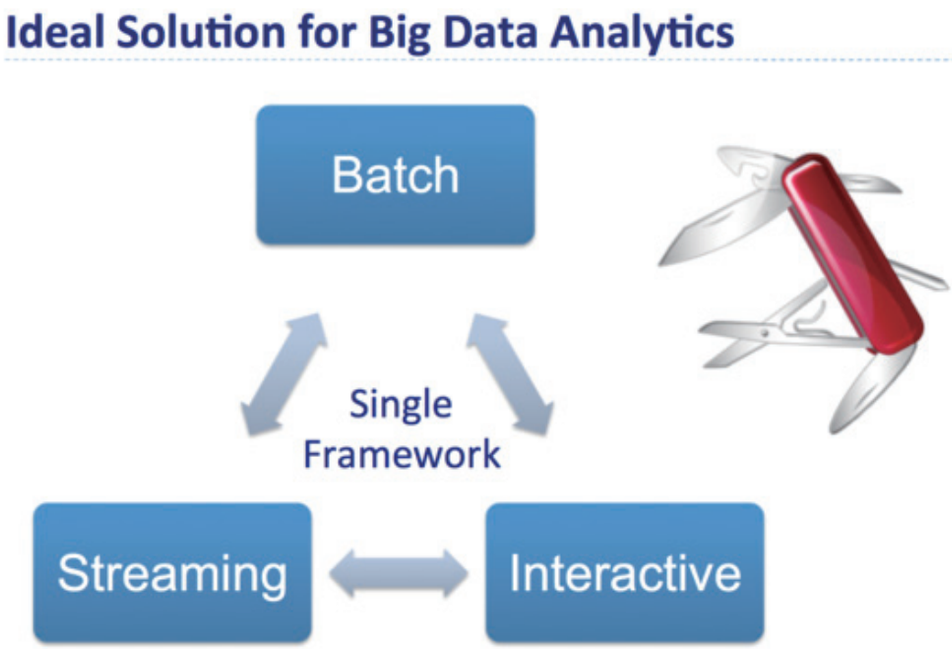
1. 对流数据进行处理来做近实时分析。举个例子，通过分析点击流数据做视频推荐，从而提高用户的参与度。在这个用例中，开发者必须在精度和延时之间做平衡。
2. 在大型数据集上进行交互式分析，数据科学家可以在数据集上做 ad-hoc 查询。



毫无疑问，历经数年发展，Hadoop 生态圈中的丰富工具已深受用户喜爱，然而这里仍然存在众多问题给使用带来了挑战：

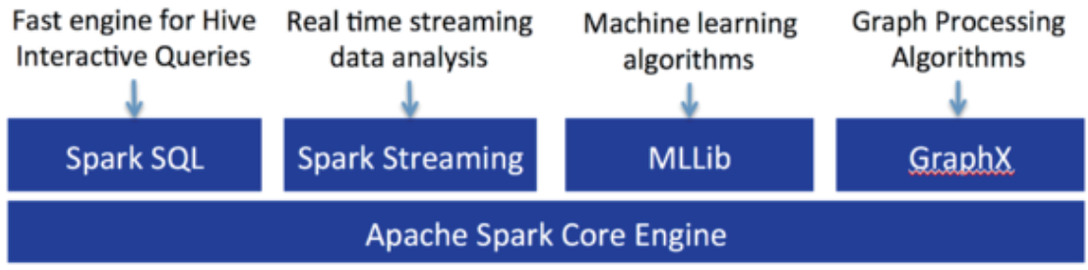
1. 每个用例都需要多个不同的技术堆栈来支撑，在不同使用场景下，大量的解决方案往往捉襟见肘。
2. 在生产环境中机构往往需要精通数门技术。
3. 许多技术存在版本兼容性问题。
4. 无法在并行 job 中更快地共享数据。

而通过 Apache Spark，上述问题迎刃而解！Apache Spark 是一个轻量级的内存集群计算平台，通过不同的组件来支撑批、流和交互式用例，如下图。



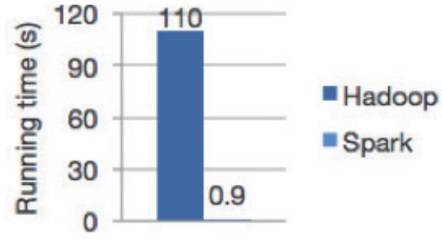
二、关于 Apache Spark

Apache Spark 是个开源和兼容 Hadoop 的集群计算平台。由加州大学伯克利分校的 AMPLabs 开发，作为 Berkeley Data Analytics Stack（BDAS）的一部分，当下由大数据公司 Databricks 保驾护航，更是 Apache 旗下的顶级项目，下图显示了 Apache Spark 堆栈中的不同组件。

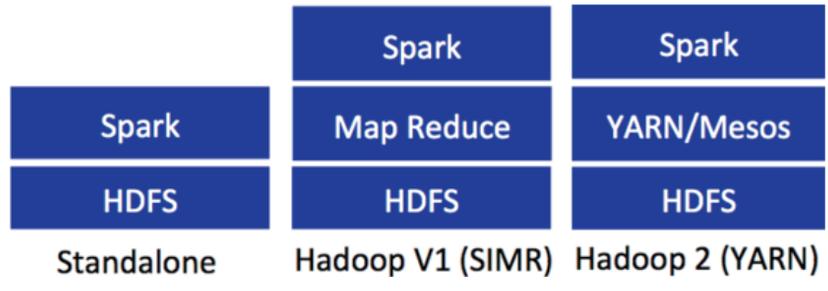


Apache Spark 的5大优势:

1. 更高的性能, 因为数据被加载到集群主机的分布式内存中。数据可以被快速的转换迭代, 并缓存用以后续的频繁访问需求。很多对 Spark 感兴趣的朋友可能也会听过这样一句话——在数据全部加载到内存的情况下，Spark 可以比 Hadoop 快 100 倍，在内存不够存放所有数据的情况下快 Hadoop 10 倍。
2. 通过建立在 Java、Scala、Python、SQL（应对交互式查询）的标准 API 以方便各行各业使用，同时还含有大量开箱即用的机器学习库。



3. 与现有 Hadoop v1（SIMR）和 2.x (YARN) 生态兼容，因此机构可以进行无缝迁移。



4. 方便下载和安装。方便的 shell（REPL: Read-Eval-Print-Loop）可以对 API 进行交互式的学习。
5. 借助高等级的架构提高生产力，从而可以讲精力放到计算上。

同时，Apache Spark 由 Scala 实现，代码非常简洁。

三、安装Apache Spark

下表列出了一些重要链接和先决条件：

Current Release	**1.0.1 @http://d3kbcqa49mib13.cloudfront.net/spark-1.0.1.tgz**
Downloads Page	https://spark.apache.org/downloads.html
JDK Version (Required)	1.6 or higher
Scala Version (Required)	2.10 or higher
Python (Optional)	[2.6, 3.0)
Simple Build Tool (Required)	http://www.scala-sbt.org
Development Version	git clone git://github.com/apache/spark.git
Building Instructions	https://spark.apache.org/docs/latest/building-with-maven.html
Maven	3.0 or higher

如上图所示，Apache Spark 的部署方式包括 standalone、Hadoop V1 SIMR、Hadoop 2 YARN/Mesos 。Apache Spark 需求一定的 Java、Scala 或 Python 知识。这里，我们将专注 standalone 配置下的安装和运行。

1. 安装 JDK 1.6+、Scala 2.10+、Python [2.6,3] 和 sbt
2. 下载 Apache Spark 1.0.1 Release
3. 在指定目录下 Untar 和 Unzip spark-1.0.1.tgz

```
akuntamukkala@localhost~/Downloads$ pwd /Users/akuntamukkala/Downloads akuntamukkala@
localhost~/Downloads$ tar -zxvf spark- 1.0.1.tgz -C /Users/akuntamukkala/spark
```

4. 运行 sbt 建立 Apache Spark

```
akuntamukkala@localhost~/spark/spark-1.0.1$ pwd /Users/akuntamukkala/spark/spark-1.0.1
akuntamukkala@localhost~/spark/spark-1.0.1$ sbt/sbt assembly
```

5. 发布 Scala 的 Apache Spark standalone REPL

```
/Users/akuntamukkala/spark/spark-1.0.1/bin/spark-shell
```

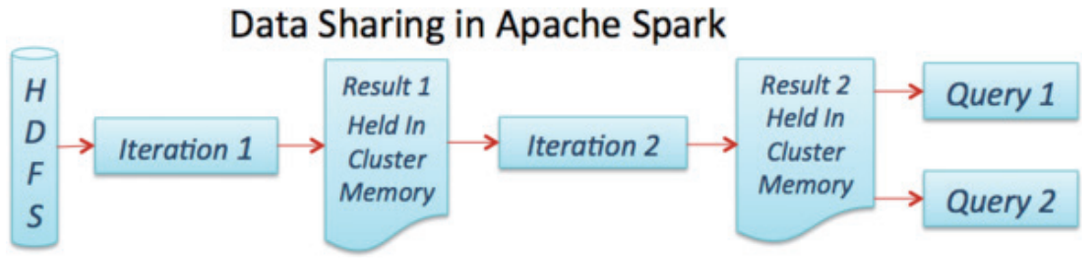
如果是 Python

```
/Users/akuntamukkala/spark/spark-1.0.1/bin/ pyspark
```

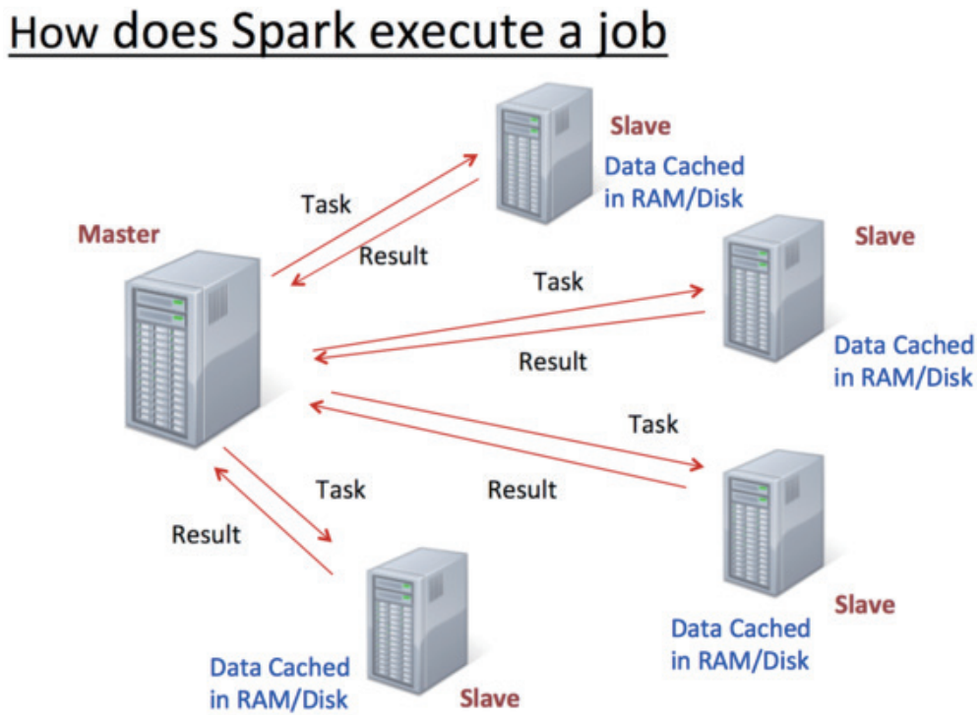
6. 查看 SparkUI @ http://localhost:4040

四、Apache Spark 的工作模式

Spark 引擎提供了在集群中所有主机上进行分布式内存数据处理的能力，下图显示了一个典型 Spark job 的处理流程。



下图显示了 Apache Spark 如何在集群中执行一个作业。



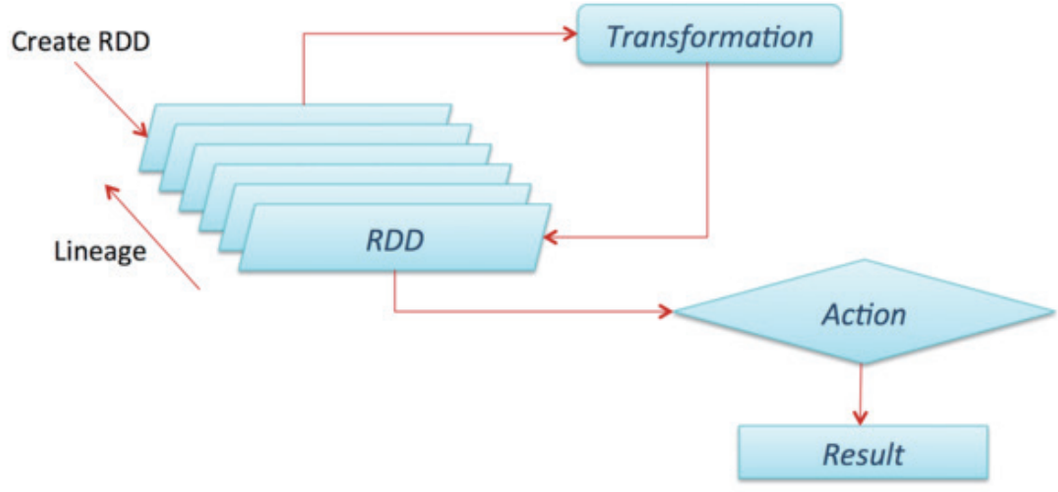
Master 控制数据如何被分割，利用了数据本地性，并在 Slaves 上跟踪所有分布式计算。在某个Slave不可用时，其存储的数据会分配给其他可用的 Slaves 。虽然当下（ 1.0.1 版本）Master 还存在单点故障，但后期必然会被修复。

五、弹性分布式数据集（Resilient Distributed Dataset，RDD）

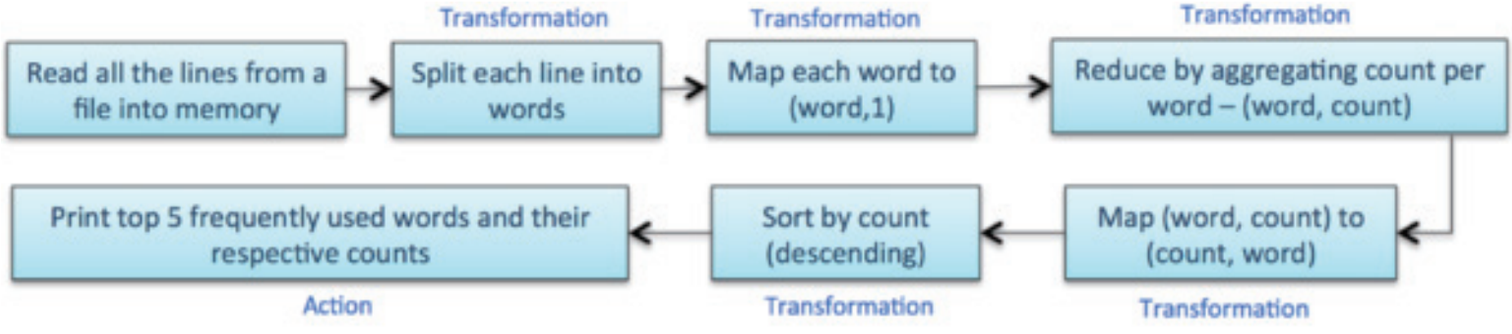
弹性分布式数据集（RDD，从 Spark 1.3 版本开始已被 DataFrame 替代）是 Apache Spark 的核心理念。它是由数据组成的不可变分布式集合，其主要进行两个操作：transformation 和 action。Transformation 是类似在 RDD 上做 filter()、map() 或 union() 以生成另一个 RDD 的操

作，而 action 则是 count()、first()、take(n)、collect() 等促发一个计算并返回值到 Master 或者稳定存储系统的操作。Transformations 一般都是 lazy 的，直到 action 执行后才会被执行。Spark Master/Driver 会保存 RDD 上的 Transformations 。这样一来，如果某个 RDD 丢失（也就是 salves 宕掉），它可以快速和便捷地转换到集群中存活的主机上。这也就是 RDD 的弹性所在。

下图展示了 Transformation 的 lazy：



我们可以通过下面示例来理解这个概念：从文本中发现 5 个最常用的 word 。下图显示了一个可能的解决方案。



在上面命令中，我们对文本进行读取并且建立字符串的 RDD 。每个条目代表了文本中的 1 行。

```
scala> val hamlet = sc.textFile(“/Users/akuntamukkala/temp/gutenburg.txt”)
hamlet: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at textFile at <console>:12
```

```
scala> val topWordCount = hamlet.flatMap(str=>str.split(“ “)). filter(!_.isEmpty).
map(word=>(word,1)).reduceByKey(_+_).map{case (word, count) => (count, word)}.
sortByKey(false)
topWordCount: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[10] at
sortByKey at <console>:14
```

1. 通过上述命令我们可以发现这个操作非常简单——通过简单的 Scala API 来连接 transformations 和 actions 。
2. 可能存在某些 words 被 1 个以上空格分隔的情况，导致有些 words 是空字符串，因此需要使用 filter(!_.isEmpty) 将它们过滤掉。
3. 每个 word 都被映射成一个键值对：map(word=>(word,1))。
4. 为了合计所有计数，这里需要调用一个 reduce 步骤—— reduceByKey(+) 。 + 可以非常便捷地为每个 key 赋值。
5. 我们得到了 words 以及各自的 counts，下一步需要做的是根据 counts 排序。在 Apache Spark ，用户只能根据 key 排序，而不是值。因此，这里需要使用 map{case (word, count) => (count, word)} 将 (word, count) 流转到 (count, word)。
6. 需要计算最常用的 5 个 words ，因此需要使用 sortByKey(false) 做一个计数的递减排序。

上述命令包含了一个 .take(5) (an action operation, which triggers computation) 和 在 /Users/akuntamukkala/temp/gutenburg.txt 文本中输出 10 个最常用的 words 。在 Python shell 中用户可以实现同样的功能。

RDD lineage 可以通过 toDebugString（一个值得记住的操作）来跟踪。

```
scala> topWordCount.take(5).foreach(x=>println(x)) (1044,the) (730,and) (679,of)
(648,to) (511,I)
```

常用的 Transformations：

Transformation & Purpose	Example & Result
filter(func) **Purpose**: new RDD by selecting those data elements on which func returns true	scala> val rdd = sc.parallelize(List("ABC" ," BCD" ," DEF")) scala> val filtered = rdd.filter(_.contains("C")) scala> filtered.collect() **Result**: Array[String] = Array(ABC, BCD)
map(func) **Purpose**: return new RDD by applying func on each data element	scala> val rdd=sc.parallelize(List(1,2,3,4,5)) scala> val times2 = rdd.map(_*2) scala> times2.collect() **Result**: Array[Int] = Array(2, 4, 6, 8, 10)
flatMap(func) **Purpose**: Similar to map but func returns a Seq instead of a value. For example, mapping a sentence into a Seq of words	scala> val rdd=sc.parallelize(List("Spark is awesome" ," It is fun")) scala> val fm=rdd.flatMap(str=>str.split(" ")) scala> fm.collect() **Result**: Array[String] = Array(Spark, is, awesome, It, is, fun)
reduceByKey(func,[numTasks]) **Purpose**: To aggregate values of a key using a function. "numTasks" is an optional parameter to specify number of reduce tasks scala> val word1=fm.map(word=>(word,1))	scala> val wrdCnt=word1.reduceByKey(+) scala> wrdCnt.collect() **Result**: Array[(String, Int)] = Array((is,2), (It,1), (awesome,1), (Spark,1), (fun,1))

常见集合操作

Transformation and Purpose	Example and Result
union() **Purpose**: new RDD containing all elements from source RDD and argument.	Scala> val rdd1=sc.parallelize(List('A' ,' B')) scala> val rdd2=sc.parallelize(List('B' ,' C')) scala> rdd1.union(rdd2).collect() **Result**: Array[Char] = Array(A, B, B, C)
intersection() **Purpose**: new RDD containing only common elements from source RDD and argument.	Scala> rdd1.intersection(rdd2).collect() **Result**: Array[Char] = Array(B)
cartesian() **Purpose**: new RDD cross product of all elements from source RDD and argument	Scala> rdd1.cartesian(rdd2).collect() **Result**: Array[(Char, Char)] = Array((A,B), (A,C), (B,B), (B,C))
subtract() **Purpose**: new RDD created by removing data elements in source RDD in common with argument	scala> rdd1.subtract(rdd2).collect() **Result**: Array[Char] = Array(A)
join(RDD,[numTasks]) **Purpose**: When invoked on (K,V) and (K,W), this operation creates a new RDD of (K, (V,W))	scala> val personFruit = sc.parallelize(Seq(("Andy" , "Apple"), ("Bob" , "Banana"), ("Charlie" , "Cherry"), ("Andy" ," Apricot"))) scala> val personSE = sc.parallelize(Seq(("Andy" , "Google"), ("Bob" , "Bing"), ("Charlie" , "Yahoo"), ("Bob" ," AltaVista"))) scala> personFruit.join(personSE).collect() **Result**: Array[(String, (String, String))] = Array((Andy,(Apple,Google)), (Andy, (Apricot,Google)), (Charlie,(Cherry,Yahoo)), (Bob,(Banana,Bing)), (Bob, (Banana,AltaVista)))
cogroup(RDD, [numTasks])**Purpose**: To convert (K,V) to (K,Iterable)	scala> personFruit.cogroup(personSe).collect() **Result**: Array[(String, (Iterable[String], Iterable[String]))] = Array((Andy, (ArrayBuffer(Apple, Apricot),ArrayBuffer(google))), (Charlie, (ArrayBuffer(Cherry),ArrayBuffer(Yahoo))), (Bob, (ArrayBuffer(Banana),ArrayBuffer(Bing, AltaVista))))

更 多 transformations 信 息，请 查 看 http://spark.apache.org/docs/latest/programming-guide.html#transformations

常用的 actions enter image description here

Action & Purpose	Example & Result
count() **Purpose**: get the number of data elements in the RDD	scala> val rdd = sc.parallelize(list('A' ,' B' ,' c')) scala> rdd.count() **Result**: long = 3
collect()**Purpose**: get all the data elements in an RDD as an array	scala> val rdd = sc.parallelize(list('A' ,' B' ,' c')) scala> rdd.collect() **Result**: Array[char] = Array(A, B, c)
reduce(func) **Purpose**: Aggregate the data elements in an RDD using this function which takes two arguments and returns one	scala> val rdd = sc.parallelize(list(1,2,3,4)) scala> rdd.reduce(+) **Result**: Int = 10
take (n) **Purpose**: fetch first n data elements in an RDD. computed by driver program.	Scala> val rdd = sc.parallelize(list(1,2,3,4)) scala> rdd.take(2) **Result:** Array[Int] = Array(1, 2)
foreach(func) **Purpose**: execute function for each data element in RDD. usually used to update an accumulator(discussed later) or interacting with external systems.	Scala> val rdd = sc.parallelize(list(1,2,3,4)) scala> rdd.foreach(x=>println("%s*10=%s" . format(x,x*10))) **Result:** 1*10=10 4*10=40 3*10=30 2*10=20
first() **Purpose**: retrieves the first data element in RDD.	Similar to take(1) scala> val rdd = sc.parallelize(list(1,2,3,4)) scala> rdd.first() **Result:** Int = 1
saveAsTextFile(path) **Purpose**: Writes the content of RDD to a text file or a set of text files to local file system/ HDFS	scala> val hamlet = sc.textFile("/users/akuntamukkala/temp/gutenburg.txt") scala>hamlet.filter(_.contains("Shakespeare")). saveAsTextFile("/users/akuntamukkala/temp/filtered") **Result:** akuntamukkala@localhost~/temp/filtered$ ls _SUCCESS part-00000 part-00001

更多actions参见http://spark.apache.org/docs/latest/programming-guide.html#actions

六、RDD持久性

Apache Spark 中一个主要的能力就是在集群内存中持久化/缓存 RDD。这将显著地提升交互速度。下表显示了 Spark 中各种选项。

Storage Level	Purpose
MEMORY_ONLY (Default level)	This option stores RDD in available cluster memory as deserialized Java objects. Some partitions may not be cached if there is not enough cluster memory. Those partitions will be recalculated on the fly as needed.
MEMORY_AND_DISK	This option stores RDD as deserialized Java objects. If RDD does not fit in cluster memory, then store those partitions on the disk and read them as needed.
MEMORY_ONLY_SER	This options stores RDD as serialized Java objects (One byte array per partition). This is more CPU intensive but saves memory as it is more space efficient. Some partitions may not be cached. Those will be recalculated on the fly as needed.
MEMORY_ONLY_DISK_SER	This option is same as above except that disk is used when memory is not sufficient.
DISC_ONLY	This option stores the RDD only on the disk
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as other levels but partitions are replicated on 2 slave nodes

上面的存储等级可以通过 RDD. cache() 操作上的 persist () 操作访问，可以方便地指定 MEMORY_ONLY 选项。关于持久化等级的更多信息，可以访问这里 http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence。

Spark 使用 Least Recently Used (LRU) 算法来移除缓存中旧的、不常用的 RDD，从而释放出更多可用内存。同样还提供了一个 unpersist() 操作来强制移除缓存/持久化的 RDD 。

七、变量共享

Accumulators。Spark 提供了一个非常便捷地途径来避免可变的计数器和计数器同步问题—— Accumulators 。Accumulators 在一个 Spark context 中通过默认值初始化，这些计数器在 Slaves 节点上可用，但是 Slaves 节点不能对其进行读取。它们的作用就是来获取原子更新，并将其转发到 Master 。 Master 是唯一可以读取和计算所有更新合集的节点。举个例子：

```
akuntamukkala@localhost~/temp$ cat output.log error warning info trace error info info scala> val nErrors=sc.accumulator(0.0) scala> val logs = sc.textFile(“/Users/akuntamukkala/temp/output.log”) scala> logs.filter(_.contains(“error”)).foreach(x=>nErrors+=1) scala> nErrors.value Result:Int = 2
```

Broadcast Variables。实际生产中，通过指定 key 在 RDDs 上对数据进行合并的场景非常常见。在这种情况下，很可能会出现给 slave nodes 发送大体积数据集的情况，让其负责托管需要做 join 的数据。因此，这里很可能存在巨大的性能瓶颈，因为网络 IO 比内存访问速度慢 100 倍。为了解决这个问题，Spark 提供了 Broadcast Variables，如其名称一样，它会向 slave nodes 进行广播。因此，节点上的 RDD 操作可以快速访问 Broadcast Variables 值。举个例子，期望计算一个文件中所有路线项的运输成本。通过一个 look-up table指定每种运输类型的成本，这个look-up table 就可以作为 Broadcast Variables 。

```
akuntamukkala@localhost~/temp$ cat packagesToShip.txt ground express media priority priority ground express media scala> val map = sc.parallelize(Seq((“ground”,1),(“med”,2), (“priority”,5),(“express”,10))).collect().toMap map: scala.collection.immutable.Map[String,Int] = Map(ground -> 1, media -> 2, priority -> 5, express -> 10) scala> val bcMailRates = sc.broadcast(map)
```

上述命令中，我们建立了一个 broadcast variable，基于服务类别成本的 map 。

```
scala> val pts = sc.textFile(“/Users/akuntamukkala/temp/packagesToShip.txt”)
```

在上述命令中，我们通过 broadcast variable 的 mailing rates 来计算运输成本。

```
scala> pts.map(shipType=>(shipType,1)).reduceByKey(+). map{case (shipType,nPackages)=
>(shipType,nPackages*bcMailRates. value(shipType))}.collect()
```

通过上述命令，我们使用 accumulator 来累加所有运输的成本。详细信息可通过下面的 PDF 查看 http://ampcamp.berkeley.edu/wp-content/uploads/2012/06/matei-zaharia-amp-camp-2012-advanced-spark.pdf。

八、Spark SQL

通过 Spark Engine，Spark SQL 提供了一个便捷的途径来进行交互式分析，使用一个被称为 SchemaRDD 类型的 RDD 。SchemaRDD 可以通过已有 RDDs 建立，或者其他外部数据格式，比如 Parquet files、JSON 数据，或者在 Hive 上运行 HQL。SchemaRDD 非常类似于 RDBMS 中的表格。一旦数据被导入 SchemaRDD，Spark 引擎就可以对它进行批或流处理。Spark SQL 提供了两种类型的 Contexts——SQLContext 和 HiveContext，扩展了 SparkContext 的功能。

SparkContext 提供了到简单 SQL parser 的访问，而 HiveContext 则提供了到 HiveQL parser 的访问。HiveContext 允许企业利用已有的 Hive 基础设施。

这里看一个简单的 SQLContext 示例。

下面文本中的用户数据通过“|”来分割。

```
John Smith|38|M|201 East Heading Way #2203,Irving, TX,75063 Liana Dole|22|F|1023 West
Feeder Rd, Plano,TX,75093 Craig Wolf|34|M|75942 Border Trail,Fort Worth,TX,75108
John Ledger|28|M|203 Galaxy Way,Paris, TX,75461 Joe Graham|40|M|5023 Silicon
Rd,London,TX,76854
```

定义 Scala case class 来表示每一行：

```
case class Customer(name:String,age:Int,gender:String,address: String)
```

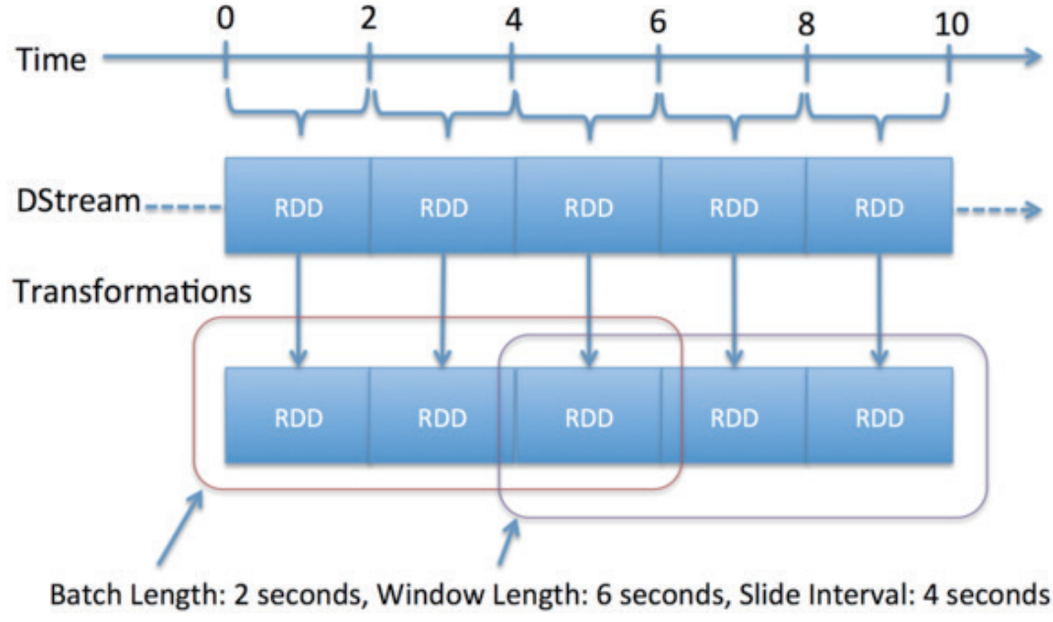
下面的代码片段体现了如何使用 SparkContext 来建立 SQLContext，读取输入文件，将每一行都转换成 SparkContext 中的一条记录，并通过简单的 SQL 语句来查询 30 岁以下的男性用户。

```
val sparkConf = new SparkConf().setAppName(“Customers”)
val sc = new SparkContext(sparkConf)
val sqlContext = new SQLContext(sc)
val r = sc.textFile(“/Users/akuntamukkala/temp/customers.txt”) val records =
r.map(_.split(‘|’))
val c = records.map(r=>Customer(r(0),r(1).trim.toInt,r(2),r(3)))
c.registerAsTable(“customers”)
sqlContext.sql(“select * from customers where gender=’M’ and age <
                30”).collect().foreach(println) Result:[John Ledger,28,M,203 Galaxy
Way,Paris,
            TX,75461]
```

更多使用 SQL 和 HiveQL 的示例请访问下面链接 https://spark.apache.org/docs/latest/sql-programming-guide.html 、https://databricks-training.s3.amazonaws.com/data-exploration-using-spark-sql.html。

九、Spark Streaming

Spark Streaming 提供了一个可扩展、容错、高效的途径来处理流数据，同时还利用了 Spark 的简易编程模型。从真正意义上讲，Spark Streaming 会将流数据转换成 micro batches，从而将 Spark 批处理编程模型应用到流用例中。这种统一的编程模型让 Spark 可以很好地整合批量处理和交互式流分析。下图显示了 Spark Streaming 可以从不同数据源中读取数据进行分析。 p12



Spark Streaming 中的核心抽象是 Discretized Stream (DStream)。DStream 由一组 RDD 组成，每个 RDD 都包含了规定时间 (可配置) 流入的数据。上图很好地展示了 Spark Streaming 如何通过将流入数据转换成一系列的 RDDs，再转换成 DStream 。每个 RDD 都包含两秒 (设定的区间长度) 的数据。在 Spark Streaming 中，最小长度可以设置为 0.5 秒，因此处理延时可以达到 1 秒以下。

Spark Streaming 同样提供了 window operators，它有助于更有效率在一组 RDD（a rolling window of time）上进行计算。同时，DStream 还提供了一个 API，其操作符（transformations 和 output operators）可以帮助用户直接操作 RDD。下面不妨看向包含在 Spark Streaming 下载中的一个简单示例。示例是在 Twitter 流中找出趋势 hashtags，详见下面代码。

```
 spark-    1.0.1/examples/src/main/scala/org/apache/spark/examples/streaming/
TwitterPopularTags.scala
val sparkConf = new SparkConf().setAppName(“TwitterPopularTags”)
val ssc = new StreamingContext(sparkConf, Seconds(2))
val stream = TwitterUtils.createStream(ssc, None, filters)
```

上述代码用于建立 Spark Streaming Context。Spark Streaming 将在 DStream 中建立一个 RDD，包含了每 2 秒流入的 tweets。

```
val hashTags = stream.flatMap(status => status.getText.split(“ “).filter(_.
startsWith(“#”)))
```

上述代码片段将 Tweet 转换成一组 words，并过滤出所有以 a# 开头的。

```
val topCounts60 = hashTags.map((_, 1)).reduceByKeyAndWindow(_ + _, Seconds(60)).
map{case (topic, count) => (count, topic)}. transform(_.sortByKey(false))
```

上述代码展示了如何整合计算 60 秒内一个 hashtag 流入的总次数。

```
topCounts60.foreachRDD(rdd => {
val topList = rdd.take(10)
println(“\nPopular topics in last 60 seconds (%s
total):”.format(rdd.count())) topList.foreach{case (count, tag) => println(“%s (%s
tweets)”.format(tag, count))} })
```

上面代码将找出 top 10 趋势 tweets，然后将其打印。

```
ssc.start()
```

上述代码让 Spark Streaming Context 开始检索 tweets 。一起聚焦一些常用操作，假设我们正在从一个 socket 中读入流文本。

```
al lines = ssc.socketTextStream(“localhost”, 9999, StorageLevel.MEMORY_AND_DISK_SER)
```

TRANFORMATION & PURPOSE	EXAMPLE & RESULT
map(func) ***Purpose:*** Create new DStream by applying this function to tall constituent RDDS in DStream	lines.map(x=>x.toInt*10).print() prompt>nc –lk 9999 12 34 Output: 120 340
flatMap(func) ***Purpose:*** This is same as map but mapping function can output 0 or more items	lines.flatMap(_.split(“ “)).print() prompt>nc –lk 9999 Spark is fun Output: Spark is fun
count() ***Purpose:*** create a DStream of RDDs containing count of number of data elements	lines.flatMap(_.split(“ “)).print() prompt>nc –lk 9999 say hello to spark Output: 4
reduce(func) ***Purpose:*** Same as count but instead of count, the value is derived by applying the function	lines.map(x=>x.toInt).reduce(_+_).print() prompt>nc –lk 9999 1 3 5 7 Output: 16
countByValue() ***Purpose:*** This is same as map but mapping function can output 0 or more items	lines.map(x=>x.toInt).reduce(_+_).print() prompt>nc –lk 9999 9999 spark spark is fun fun Output: (is,1) (spark,2) (fun,2)

<table>
<tr><td>reduceByKey(func,[numTasks])</td><td>lines.map(x=>x.toInt).reduce(_+_).print()
<table><tr><td>prompt>nc –lk 9999
9999
spark
spark
is
fun
fun</td><td>Output:
(is,1)
(spark,2)
(fun,2)</td></tr></table></td></tr>
<tr><td>reduceByKey(func,[numTasks])</td><td>val words = lines.flatMap(_.split(“ “))
val wordCounts = words.map(x => (x, 1)).
reduceByKey(_+_)
wordCounts.print()
<table><tr><td>prompt>nc –lk 9999
spark is fun
fun
fun</td><td>Output:
(is,1)
(spark,1)
(fun,3)</td></tr></table></td></tr>
<tr><td colspan="2">The following example shows how Apache Spark combines Spark batch with Spark Streaming. This is a powerful capability for an all-in-one technology stack. In this example, we read a file containing brand names and filter those streaming data sets that contain any of the brand names in the file.</td></tr>
<tr><td>transform(func)
Purpose: Creates a new DStream by applying RDD->RDD transformation to all RDDs in DStream.

brandNames.txt
coke
nike
sprite
reebok</td><td>val sparkConf = new SparkConf()
.setAppName(“NetworkWordCount”)
val sc = new SparkContext(sparkConf)
val ssc = new StreamingContext(sc,
Seconds(10))
val lines = ssc.
socketTextStream(“localhost” 9999,
StorageLevel.MEMORY_AND_DISK_SER)
val brands = sc.textFile(“/Users/
akuntamukkala/temp/brandNames.txt”)
lines.transform(rdd=> {
rdd.intersection(brands)
}).print()
<table><tr><td>prompt>nc –lk 9999
msft
apple
nike
sprite
ibm</td><td>Output:
sprite
nike</td></tr></table></td></tr>
<tr><td>updateStateByKey(func)
Purpose: creates a new DStream where the value of each key is updated by applying given function.</td><td>Please refer to the StatefulNetworkCount example in Spark Streaming.

This helps in computing a running aggregate of total number of times a word has occurred</td></tr>
</table>

Common Window Operations

TRANFORMATION & PURPOSE	EXAMPLE & RESULT

window(windowLength, slideInterval)
Purpose: Returns a new DStream computed from windowed batches of source DStream
	val win = lines.window(Seconds(30),Seconds(10));
	win.foreachRDD(rdd => {
	rdd.foreach(x=>println(x+ " "))
	})
	prompt>nc –lk 9999 / Output:
	10 (0th second) / 10
	20 (10 seconds later) / 10 20
	30 (20 seconds later) / 20 10 30
	40 (30 seconds later) / 20 30 40 (drops 10)

countByWindow(windowLength, slideInterval) ***Purpose:*** Returns a new sliding window count of elements in a steam	lines.countByWindow(Seconds(30),Seconds(10)).print() prompt>nc –lk 9999 / Output: 10 (0th second) / 1 20 (10 seconds later) / 2 30 (20 seconds later) / 3 40 (30 seconds later) / 3

更 多 operators 请 访 问 http://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations。

Spark Streaming 拥有大量强大的 output operators，比如上文提到的 foreachRDD()，了解更多可访问 http://spark.apache.org/docs/latest/streaming-programming-guide.html#output-operations。

十、附加学习资源

- Wikipedia article (good): http://en.wikipedia.org/wiki/Apache_Spark
- Launching a Spark cluster on EC2: http://ampcamp.berkeley.edu/exercises-strata-conf-2013/launching-a-cluster.html
- Quick start: https://spark.apache.org/docs/1.0.1/quick-start.html

- The Spark platform provides MLLib(machine learning) and GraphX(graph algorithms). The following links provide more information:https://spark.apache.org/docs/latest/mllib-guide.html 、https://spark.apache.org/docs/1.0.1/graphx-programming-guide.html 、https://dzone.com/refcardz/apache-spark ■

阅读英文原文

阅读图灵社区原文

优秀的叛逆者，卓越的工作

作者 / Carmen Medina

Carmen 在美国中央情报局工作了 32 年，她曾是中情局职位最高的女性之一。她现在的工作是针对认知多样性、如何更好地思考以及如何成为工作中的叛逆者进行写作和演说。

优秀的叛逆者只想做卓越的工作。

我们想要改进工作方式，解决那些让我们的组织陷入危险境地的问题。我们不是为了获得个人荣誉，而是为了引进新的想法，惠及同事、客户或团队成员。叛逆者最渴求的就是帮助组织进化，找到缜密的方法来检验新想法，确认何时以及如何实践这些想法，迈出走向更好未来的第一步。

“叛逆者”这个词被提及了很多次，因此我们会解释一下它的具体意思。从最基本的层面来看，优秀的叛逆者会创造更新更好的做事方法，而差劲的叛逆者只会抱怨工作中出现的那些问题。抱怨问题很简单，困难的是弄清如何解决问题。

几年前，我们制作了一张表，来对比优秀的叛逆者和差劲的叛逆者（见表 1-1）。这张表被下载了十万多次，并出现在推文和世界各地的演讲中。

我们认为这张表受到欢迎主要出于以下三个原因。第一，它总结了叛逆者的常见行为。第二，管理者有时会给想做出积极改变的有志之士强加上“麻烦制造者”的标签，而这张表对此进行了驳斥。第三点可能更为复杂，它指出了许多优秀的叛逆者会幻想破灭，最终成为差劲的叛逆者，虽然这些人最初的想法都是好的。试图让同事倾听和认同合理的新想法，会让优秀的叛逆者受到挫折，而这些挫折会严重到让优秀的叛逆者感到悲观，受到指责，或者变得愤怒和心塞。

表 1-1　差劲的叛逆者与优秀的叛逆者

差劲的叛逆者	优秀的叛逆者
抱怨	创造
断言	质疑
关注个人	关注任务
悲观	乐观
愤怒	热情
耗费能量	生成能量
疏离	吸引
问题	可能性
抱怨问题	寻找机会
担心	希冀
指责他人	指出原因
困扰	反抗
说教	倾听

关于这张表，需要注意的一点是：虽然它很有用，但它把一些事项过于简化了。表中提到的很多特质是一个连续过程的不同阶段。有时在叛逆者的征程上，坚持自己的想法可能是必要的，但我们并不认为固执的行为对任何叛逆者而言是一项好的长期策略。为了获得管理层的注意，有时候，比起单独和每一个人沟通想法以慢慢获得支持，在公开论坛中指出问题要更加有效。

哪些人是优秀的叛逆者

哪些人是优秀的叛逆者？他们想作出哪些改变？

在过去三年里，我们和数百位叛逆者进行过对话。他们开始成为叛逆者都是因为他们关心的问题变得非常严重，以至于他们觉得自己必须有所行动。他们大多数人从不认为自己是变革的推动者、创新者，当然更不会觉得自己是英雄。但他们认为自己非常关心自己所在的组织、同事，以及组织所服务的对象，不论是顾客还是学生，病人还是普通人。

虽然有些叛逆者能作出巨大到影响整个公司的改变，但更多的叛逆者作出的改变都是小范围的。以下是一些在不同工作场景下担任不同角色的叛逆者案例。

- 一名行政助理，因在协调各位副总裁的会议和决议时感到挫败，邀请了其他行政人员参加一次月度简易午餐会议。他们坐下来一起探讨如何更好地管理老板的时间和尽自己所能引起组织变革。
- 因一项创新方案需经过层层审批，一位政府官员感到精疲力竭，于是他发送了一条推文，请他们机构的135万名员工一起承诺做一件小事来提高机构的效率。这些员工在第一届年度英国国家医疗服务体系（NHS）改变日上做出了189 000个承诺，发掘出了全英国的创造力和改进想法。
- 一名25岁的公关经理，有幸为他钦佩的一家执法机构工作，在工作期间他发现了用社交媒体帮助该机构履行职责的新方法。当他提出这个新想法时，他的老板们和法律部门用复杂且无意义的规章制度否决了他。他开始怀疑政府到底是想要人才还是想要奴才。他给他的老板们展示了不采用新的公关方式会带来的危险，并由此慢慢获得了他们的支持。
- 一名政策分析师分享了她关于改善政府机构履行职能的方式的想法。她的老板在静静听完她的想法后说道：“闭紧你的嘴，等到20年以后你身居要位时，再来进行这些你认为重要的改变吧。”听完后，这位分析师就辞职了，因为她确信这家机构不可能有任何进步。

- 一位污水工程公司的销售总监连续三年向执行团队展示能改变行业的技术理念，之后再也不在公司内部倡导该理念了。在接下来的六年里，他继续与客户交谈，收集意见以改进这个理念。十年后，公司引入了他所提出的新系统，并获得了巨大的成功。
- 一名一直在晋升的军官，提出了现代化的方式来改进领导力和专业化发展。但在意识到上级只会提拔那些让他们感到舒心的人之后，他提早退休了。那些想要改变现状的人让这些高层感到不安，因而不会被提拔到最高职位。

作为叛逆者，我们的故事都不一样，但我们有许多共同点。表 1-2 展示了成功的叛逆者会使用的一些策略以及会培养的一些行为。

表 1-2　成功叛逆者的秘密

策略：实现特定目标的行动	行为：你的行为举止，特别是在对待他人时
利用他人的才华，明白没人能独自完成意义重大的改变	保持乐观：乐观的态度能鼓舞他人加入解决问题的行列
将自己的想法和组织的目标统一起来。一个想法对于组织越重要，就越有可能被采用	评论想法，不评论人。讨论想法和想法的价值，而非讨论人的性格
展现改变带来的好处与代价相称	从愤怒中学习：思考是什么引发了愤怒的情绪，并避免陷入情绪闹剧
有效利用冲突：探究分歧和冲突，学习如何完善和推进一个想法	尊重他人并考虑不同的观点
不操之过急，给他人时间来接受新想法并思考它的影响	知道何时该放弃坚持，懂得权衡想法的重要性和固执的代价

一个没有叛逆者的世界

有些人可能会怀疑叛逆者在社会中起到的重要作用，那我们就来看看一个没有叛逆者的世界会是什么样的。

我们身边就有这样的例子。

我们教导孩子言论自由的重要性和“群体思维”的危险性，鼓励他们阅读像乔治·奥威尔的《1984》那样的小说，《1984》中真理局的真正使命是伪造历史事件。而在洛伊丝·洛利的《授者》一书描绘的世界里，痛苦、恐惧、强烈的爱和仇恨都已被完全消除。不存在偏见，因为人们的所想所见都是一样的。然而，正因为如此，罪恶也暗藏其中。

尽管如此，不论是在学校里还是在工作中，群体思维都是被推崇的。那些提出质疑和主张不同方式的人通常都会被忽视、排斥或解雇。（最受欢迎的两篇博客文章，主题都是关于被推下水的员工。）然而，如果没有了叛逆者，我们的系统、公司、学校、教堂、政府部门以及医疗机构都会变得僵化，甚至岌岌可危。

孩子们会因为与众不同而被嘲笑，甚至被欺辱。政府官员会为了保护他们的预算和员工编制而忽视民众的需求。公司无视或者忽视新兴趋势，以至于每天都在亏损，只能不停地裁员。

在没有叛逆者的世界中，危险有时也会为更具体。

1986年，美国国家航空航天局（NASA）的工程师警告说，“挑战者”号航天飞机的关键部件有一个潜在的致命缺陷，会导致该部件在低温下无法正常运转。在一月份一个寒冷的早晨，NASA的官员决定忽视工程师的警告，并批准了“挑战者”号的发射。在起飞73秒后，“挑战者”号爆炸解体，七名机组人员身亡，而约17%的美国人在电视上目睹了这一幕。随后由罗纳德·里根总统指派的调查委员会发现，NASA的组织文化和决策程序是导致这一灾难的关键因素。优秀叛逆者的意见没有被重视。

对于导致数百万辆汽车被召回的点火开关危机，通用汽车公司的首席执行官Mary Barra曾公开表示，公司的企业文化使得担心安全问题的员

工闭口不言。在会议上坦言问题并不安全。2014年，这家汽车厂商承认，他们在进行第一次汽车召回的十多年前就知道点火开关存在安全问题。在这期间，至少发生了54次事故，多达100人死亡。当这个问题于2014年被披露后，通用汽车又进行了47次召回，召回车辆超过2000万辆。

伊士曼柯达公司的叛逆者们曾预见了胶片业务的灭亡，并建议主动转战数码摄影业务。然而柯达公司的领导者对胶片业务带来的利润感到很满意，安于现状，并未给这些创新者提供足够的支持。最终，不仅数千名员工丢掉了工作，这个曾经充满活力的公司也崩溃瓦解。

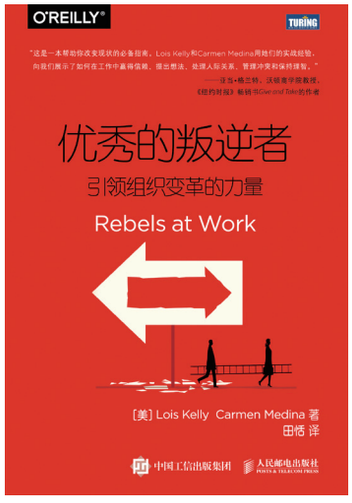
既然组织里的人们已意识到这些风险的存在，为什么这种悲剧性的结局还会一再上演呢？这正是因为在这个世界里，叛逆者们总是被孤立，而当权者却墨守成规，这就导致了不理性的决定和悲剧性的后果。

在一次内部调查后，通用汽车的首席执行官Mary Barra表示："广泛存在的官僚问题和数个部门中的员工无法汇报这个安全问题，导致我们没有及时采取行动……员工在揭露这个关键信息时一次次地失败，因而受到有问题的点火开关的影响，许多人的生活就此改写。"

通用汽车并非个例，否则你现在也不会在读这本书。拒绝改变、墨守成规、官僚主义仍然普遍存在于高速变化的现代世界中。

组织沉默：我们并不是真的在乎你的意见

纽约大学的研究员Elizabeth Morrison和Frances Milliken把这种现象称为"组织沉默"文化。在文章"组织沉默：一个多元化世界中改变和发展的障碍"(Organizational Silence: A Barrier to Change and Development in a Pluralistic World) 中，Morrison和Milliken表明，



工作中的“叛逆者”是想要改变组织现状的积极分子，他们想引入新的想法，改进组织的工作方式，以帮助组织更好地运转和发展。然而叛逆者时常会有所顾忌，不敢表达自己的想法，担心得不到领导的认可和支持，甚至被视为麻烦制造者。《优秀的叛逆者：引领组织变革的力量》针对这个普遍存在的职场难题给出了解决之道。本文节选自《优秀的叛逆者：引领组织变革的力量》。

尽管组织可能会表示他们愿意欢迎新想法，但大多数组织文化都明示或暗示员工对所担心的事情保持沉默。

“因为管理者可能强烈地需要避免尴尬和脆弱无能的感觉，所以他们可能常常回避会揭示其弱点和错误的信息，或是挑战了其当前行为的信息。而研究表明，当负面的反馈来自下级而非上级（来自自己的下属而非领导）时，他们会倾向于认为这些反馈是不准确的和不合理的，并会威胁到自己的权威与信誉。因此，对‘坏消息’或负面反馈的恐惧或抵抗，会让他们有所行动，设置一套组织框架和规定，以阻碍信息的上行沟通。”Morrison和Milliken这样解释道。

不仅管理者可能会压制好想法，所有的工作委员会、治理委员会、专责小组以及对等协同倡导者也会这么做，随着决策权变得更加分散，不再自上而下，它们的数量也在不断激增。若想融合所有人的观点，就会让所有的好想法失去意义。一个原本很好的想法会被“打折”，以至于在通过委员会审核后已不再有价值。更糟糕的是，因为委员会会议拖了几个月甚至几年，一个有价值的想法未能及时推出。一个过时的好想法就不再那么好了。

有能力者改变世界

如果我们目前的工作场所是一本小说，我们可能会停止阅读。“好悲伤！人的灵魂都被抽干了，却似乎没有人在意。我再也无法忍受了。”当试图继续阅读时，我们会迫切希望有人能扭转乾坤。“拜托了，快把写在每个人脸上的问题都一一解决吧。 有能力的人快来做点什么吧。”

幸运的是，我们的世界里从不缺少叛逆者。每个组织中都存在担任不同职位的叛逆者，并且都在学习如何扭转乾坤。

每一天，人们在工作中都会达到一个临界点，让他们说出“我受够了”。

虽然让每一个人加速成为叛逆者的原因都不一样，但大家几乎都意识到：“我需要做点什么。”这样的叛逆者在工作中扮演着很重要的角色。不论头衔、资历还是经验，他们是发现问题并推动解决问题的人。

在组织中，不需要每个人都成为叛逆者，也不需要每个叛逆者不断地在工作中发起变革，但所有的组织都需要有勇气、想法和坚定决心的叛逆者，来使情况变得更好。■

书　单

第一行代码——Android

作者：郭霖

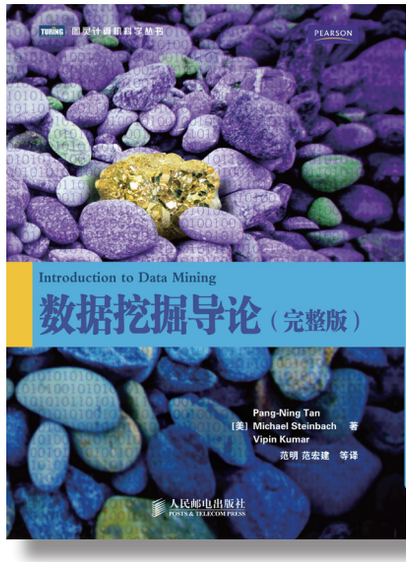
书号：978-7-115-36286-5

图灵社区推荐：16

CSDN超人气博主、CSDN 2013年度十大博客之星、资深Android开发者郭霖力作！

第一本引入经验值、升级与宝物系统的计算机书！见证自己从菜鸟到鹰的成长！

全球最大中文Android开发者社区（eoe.cn）、安卓巴士（www.apkbus.com）联袂推荐。



数据挖掘导论（完整版）

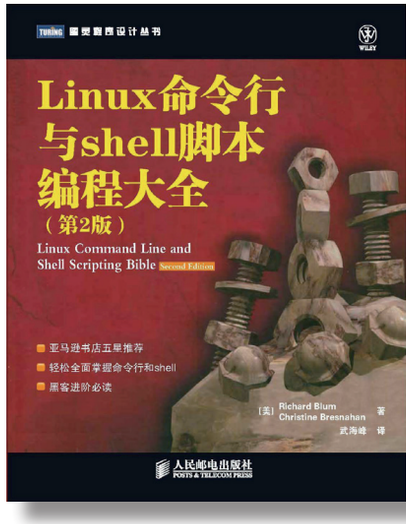
作者：Pang-Ning Tan，Michael Steinbach，Vipin Kumar

译者：范明　范宏建

书号：978-7-115-24100-9

图灵社区推荐：8

与许多其他同类图书不同，本书将重点放在如何用数据挖掘知识解决各种实际问题。只要求具备很少的预备知识——不需要数据库背景，只需要很少的统计学或数学背景知识。



Linux命令行与Shell脚本编程大全（第2版）

作者：Richard Blum，Christine Bresnahan

译者：武海峰

书号：978-7-115-28889-9

图灵社区推荐：12

本书不仅涵盖了详尽的动手教程和现实世界中的实用信息，还提供了与所学内容相关的参考信息和背景资料。亚马逊书店五星推荐。

GitHub入门与实践

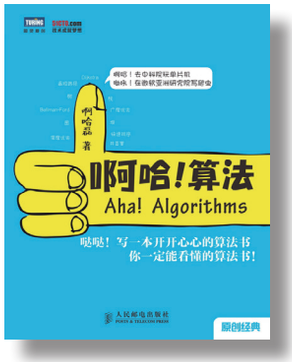
作者：大塚弘记

译者：支鹏浩　刘斌

书号：978-7-115-39409-5

图灵社区推荐：11

本书从Git的基本知识和操作方法入手，详细介绍了GitHub的各种功能，GitHub与其他工具或服务的协作，使用GitHub的开发流程以及如何将GitHub引入到企业中。

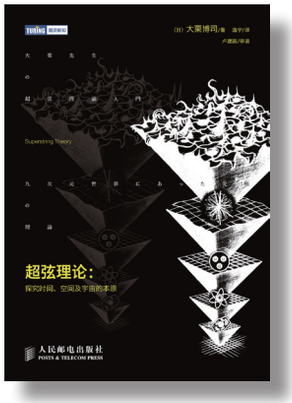


5 啊哈！算法

作者：啊哈磊
书号：978-7-115-35459-4
图灵社区推荐：12

6 程序员面试金典（第5版）

作者：Gayle Laakmann McDowell
译者：李琳骁　漆犇
书号：978-7-115-33291-2
图灵社区推荐：19

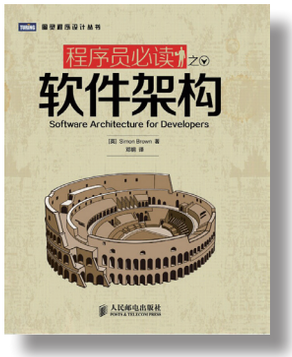


7 超弦理论：探究时间、空间及宇宙的本原

作者：大栗博司
译者：逸宁
书号：978-7-115-37386-1
图灵社区推荐：6

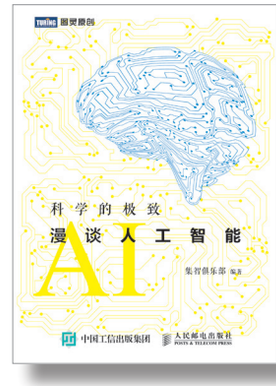
8 家用游戏机简史

作者：前田寻之
译者：周自恒
书号：978-7-115-39259-6
图灵社区推荐：6



9 程序员必读之软件架构

作者：Simon Brown
译者：邓钢
书号：978-7-115-37107-2
图灵社区推荐：14



10 科学的极致：漫谈人工智能

作者：集智俱乐部
书号：978-7-1115-39662-4
图灵社区推荐：8

电子书单

1. 命令行中的数据科学
   ——本书独树一帜，教你利用灵活的命令行工具成为高效多产的数据科学家。

2. 学习响应式设计
   ——最新最全面的响应式设计原理、技术、策略与流程。

3. 图解网站分析（修订版）：让流量倍增的网站优化方法
   ——日本亚马逊首席分析师写给站长的书。

4. Spark 机器学习
   ——适合对机器学习和数据分析感兴趣的Scala、Java或Python开发者。

5. 图解HTTP
   ——Web前端开发者必备，从基础知识到最新动向一网打尽。

6. R语言实战
   ——全面掌握使用R语言进行数据分析、数据挖掘的技巧。

7. Unity游戏设计与实现：南梦宫一线程序员的开发实例
   ——南梦宫资深开发者执笔，重点讲解设计思路和实现细节。

8. 机器学习实践：测试驱动的开发方法
   ——“这本书非常有趣。对于想深入了解机器学习的开发者来说不可多得！”

9. 函数式编程思维
   ——“这是一本非常重要的书，而说到写这本书，没有人比Neal更合适了。”

10. PHP快速参考手册
    ——从功能开发的角度来介绍技术的实践过程。

《Clojure经典实例》：编程语言习得

作者 / 王海鹏

王海鹏是一位独立的咨询顾问、培训讲师、译者和软件开发者，1994年毕业于华东师范大学。他拥有20余年编程经验，目前主要的研究领域是算法交易，对各种新技术都有兴趣，已翻译20余本软件开发类图书。

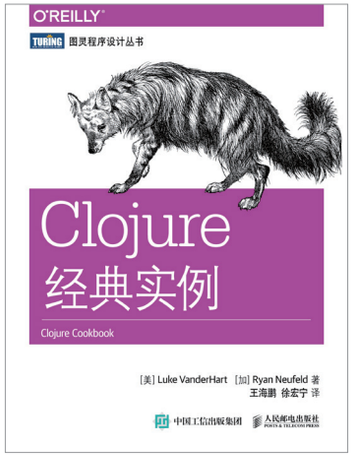
> “熟悉与优雅正交。”
>
> ——Rich Hickey

约二十年前，我买过一本高等教育出版社出版的《LISP 语言》，作者是马希文、宋柔。可惜当年没有老师指导，自己水平不够，未能深入下去，只留下了一点模糊的印象：LISP 语言适用于人工智能，括号很多。

几年前，图灵公司的朋友送我一本《黑客与画家》，我连夜看完，重新燃起了对 LISP 的兴趣。我在书评中写道：“读完之后有一种想去学习 LISP 语言的冲动。一个不懂 LISP 的 Java 程序员，不是一个好的 C++ 程序员。”

现在，我终于找到了机会，开始学习 Clojure 这种运行在 JVM 上的 LISP 方言。经过一段时间的学习，我完全被它迷住了！

首先吸引我的是它的函数式编程特性。作为一个学习 C++ 和 Java 多年的程序员，我已习惯在程序中使用各种名词抽象，也就是领域术语，希望在程序中体现领域专家的思想和认识水平。而在 Clojure 编程中，虽然它也很适合领域抽象，但它的抽象程度更高，它希望达到数学家认识世界的水平。问题的开头通常是 “给定一个无限序列……”，而常见的例子是如何实现斐波那契数列。



《Clojure 经典实例》涵盖150多个具体实例，展示了有经验的Clojure开发者如何用这门JVM语言完成各种编程任务。解决方案全面广泛：从构建动态网站和应用数据库到网络通信、云计算、高级测试策略等，面面俱到。这些实例源于全球60多名顶级Clojure开发者。

Leslie Lamport 说过，要将事情描述得清晰准确，人类发明的最好语言就是数学。这种对“表达的经济性”的追求，对于中国人是不陌生的。中国是诗歌的国度，而且古人对言简意赅的追求也有许多例子，比如“逸马杀犬于道”的故事。所以我觉得，LISP/Clojure 在精神上与有追求的中国程序员是契合的。

其次，它特别适合开发领域特定语言（DSL）。在 LISP 社区中流传着一个笑话，可以说明这一点：任何足够大的软件，最后都会实现一个半调子 LISP 解析器。LISP 的底层抽象极其简单，允许程序员设计更多的抽象，来描述这个世界。

学习一门新的语言，会改变学习者的思维方式。在面向对象编程时，我们更多关注单个对象。在函数式编程中，我们更多关注函数和集合。在工作中，不一定马上有机会使用 Clojure，但其中学到的思维方式，将对编程产生立竿见影的影响。在翻译本书时，我同时在用 Lua 开发项目，学习了 Clojure，让我能写出更简洁、更优雅的 Lua 代码。

学习新语言有这样一些原则：(1) 专注于与你相关的内容；(2) 从学习这门语言的第一天起，就把它当作你的交流方式；(3) 当你听得懂别人在说什么时，就会不知不觉慢慢习得这门语言；(4) 语言不是大量的知识积累，而更像一种生理训练；(5) 心理状态和生理状态都很重要，要愉快和放松。对于模棱两可要有一定的容忍性，对于细枝末节不要过于纠结，因为那会把你逼疯。

本书提供了大量的例子，覆盖了日常编程领域的方方面面，正是学习 Clojure 的好读物。在翻译本书的过程中，我学到了很多，在此郑重推荐给大家。不足之处，还望大家指正。■

妙　评

《项目百态》：实话的力量

作者 / 熊节

ThoughtWorks 中国公司首席咨询师，《重构》译者。

我对我的客户、一支百人团队的领导说：“你们并不是没有优先级。你们的优先级策略是最后来的事情优先。”

他苦笑不语。因为他知道，这就等于在说，他手下的百十来号人基本是在做布朗运动；更因为他知道，这是实话。只是他自己不能说，他的手下人也不能说。

事情经常是这样：尽管所有人都知道，但谁也不会把它说出口，因为说那样的话是政治不正确的。比如说吧，也许你早就知道你手上那个项目注定是条死鱼，但你敢说出口吗？更多时候，你会让自己变得乐观——乐观程度与最后期限的紧迫程度成正比，直到时间夺走你所有的手牌。

另一些时候，你知道自己碰巧做对了某些事，你不愿意新来的经理改变它：取消每周四晚上的三国杀，把团队从大会议室里搬回格子间以便“释放资源”，把几个模块外包到一千公里外的另一个城市。可你没办法说服领导。“你有度量证据吗？”当然，你没有，而削减成本总是政治正确的。

情况不会自己变好的，如果人们连真实情况都不敢说出来的话。

幸运的是，像 Tom DeMarco 这样名声和年纪（这很重要）都足够大的人可以不在乎政治正确性——换句话说，他们可以说实话。这本《项目百态》就是他们关于项目管理的实话集。

《项目百态：软件项目管理面面观（修订版）》介绍了软件项目行为的86个模式，基本上概括了软件项目生命周期的方方面面，揭示了软件项目最常遇到的困境，反省了行业内种种不良习惯和做法。六位作者均来自一个开发咨询的管理团队大西洋系统行会，长期以来为众多软件公司的经理人提供专业的咨询服务。他们浓缩了成百上千个项目管理的案例，通过本书中一个个模式展现出来。每个模式都以生动形象的插图开始，另外还加上一些趣闻和真实事件。

在这本实话集里，作者们挑选了86个项目管理中常见的模式，从“肾上腺素成瘾”直到“模板僵尸”——从英文书名不难看出，这无非是“从A到Z”的又一个变体。这些模式，诚如一位评论者所说，有良性的，更多的不仅恶性，而且丑陋可笑。读这样一本书，你会笑，更多的时候你会摇头苦笑，甚至如芒在背。“我的团队没有肾上腺素成瘾吗？”很多读者将很难面对这个问题。

那就对了。

我要感谢我的同事金明向我推荐这本书：几位作者帮我们消解了政治正确性的风险，我们就可以放心地说出实话，然后努力进步。下次当同样的模式出现时，我就可以大声地说：“这就是《项目百态》中的××模式。”或者当我再次面对前面提到的那位客户时，我可以把书递给他，对他说：“也许你该看看××模式。”

也许你也应该看看。■

图灵社区 出品

出版人：武卫东
编辑：李盼
设计：大胖

地址：北京市朝阳区北苑路13号院领地OFFICE C座603室
电话：010-51095181
微博：weibo.com/ituring
Email: ebook@turingbook.com