fpchen

发布 *technology*

方平

2021 年 01 月 29 日

Contents

CHAPTER 1

常用命令脚本

1.1 Linux 命令

1.1.1 进程常用命令

```
# 后台不停止运行 test 脚本，并将脚本日志写到 test.log
nohup ./test.sh  > test.log &

# 搜索当前运行的进程
ps aux | grep start

# 查看端口占用
lsof -i: 端口号

# 列出所有这个进程关联的文件句柄
lsof -p $pid


# 杀死进程
kill -9 ppid

# 进程监控工具
top
```

1.1.2 文件常用命令

```
# 查看日志中最近出现的字符 str
tail -f test.log | grep str

# 使用通配符 *(0 或者任意多个)。表示在/etc 目录下查找文件名中含有字符串‘srm’的文件
find /etc -name '*srm*'

# 修改文件目录权限
chmod -R 755 /upload
 # -rwxr-xr-x (755) 只有所有者才有读，写，执行的权限，组群和其他人只有读和执行的权限

# 更改文件目录所属用户
chown - R $USER /tmp/

# 查看系统用户:
cat /etc/passwd

#  文件比较异常
cat orgin_text_file1 | sort > file1.out
cat orgin_text_file2 | sort > file2.out
vimdiff file2.out file2.out
```

1.1.3 网络相关命令

```
# 查询、配置网络卡与 IP 网域等相关参数
ifconfig

# 域名解析命令，进行域名与 IP 地址解析
nslookup [主机名或者 IP]
> server

# 查看连接的网络服务
netstat -an | grep ESTABLISHED

# 查看路由表（网关）
netstat -rn


cat /etc/hosts:        域名到 IP 地址的映射。
cat /etc/networks:     网络名称到 IP 地址的映射。
cat /etc/resolv.conf: DNS 域名地址
cat /etc/protocols:    协议名称到协议编号的映射。
cat /etc/services:     TCP/UDP 服务名称到端口号的映射。
```

1.2 git 操作

1.2.1 本地分支回退版本

```
 git reset --hard   22f8aae 。22f8aae 为某次提交的提交号。
 git reset --hard HEAD~3 (回退 3 次提交)

 --hard: 本地的源码和本地未提交的源码都会回退到某个版本，包括 commit 内容，和 git 自己对代码的索引都会回退到某个版本.any local changes will be lost。
 --soft: 保留源码，只能回退到 commit 信息到某个版本，不涉及到 index 的回退，如果还需要提交，直接 commit 即可。比如我选择 soft 方式来进行回退，我的本地代码和本地新添加的尚未 commit 的代码都没有改变。
 --mixed: 会保留源码，只是将 git commit 和 index 信息回退到某个版本。
```

1.2.2 远程分支回退版本

方法一，先 git reset 回滚到本地，然后再强制 push 到远程。

```
git push -u origin master -f        origin: 远程仓库名  master: 分支名称  -f: force，意为强制、强行
```

1.2.3 删除提交分支

```
# 删除当前提交分支
git remote remove origin

# 添加新的分支
git remote add origin https://gitee.com/kingCould/HelloWord.git
```

1.2.4 分支文件冲突

```
# 保存未修改文件，添加备注，方便查找
git stash save "save message"

# 应用并删除第二个存储
git stash pop stash@{1}

# 应用第二个存储，但不会把存储从存储列表中删除
git stash apply stash@{1}

# 从列表中删除第二个存储
git stash drop stash@{1}

# 先切换到主分支（或即将被覆盖的分支）
$ git checkout master

# 将 test 分支覆盖到当前分支
$ git merge test

# 删除之前的分支（已经没用了）
$ git branch -d <branch-name>
```

1.3 SSH 远程操作

1.3.1 远程连接

连接远程服务器

```
ssh root@192.168.163.129
```

iterm 添加连接文件

- 可以在 ~ /.ssh/ 下面写一个脚本，~/.ssh/server-test

```
#!/usr/bin/expect -f
set user root
set host 192.168.1.110
set password 123456
set timeout -1

spawn ssh $user@$host
expect "*assword:*"
send "$password\r"
interact
expect eof
```

- iTerm2 的 Profiles 新建一个 profile，Command 里填入 expect ~/.ssh/server-test

1.3.2 文件传输

1、上传本地文件到服务器

```
scp /path/filename username@servername:/path/
```

例如 scp /var/www/test.php root@192.168.0.101:/var/www/ 把本机/var/www/目录下的 test.php 文件上传到 192.168.0.101 这台服务器上的/var/www/目录中

2、从服务器上下载文件

下载文件我们经常使用 wget，但是如果没有 http 服务，如何从服务器上下载文件呢?

```
scp username@servername:/path/filename /var/www/local_dir（本地目录）
```

例如 scp root@192.168.0.101:/var/www/test.txt 把 192.168.0.101 上的/var/www/test.txt 的文件下载到/var/www/local_dir（本地目录）

3、从服务器下载整个目录

```
scp -r username@servername:/var/www/remote_dir/（远程目录） /var/www/local_dir（本地目录）
```

例如:scp -r root@192.168.0.101:/var/www/test /var/www/

4、上传目录到服务器

```
scp -r local_dir username@servername:remote_dir
```

例如：scp -r test root@192.168.0.101:/var/www/ 把当前目录下的 test 目录上传到服务器的/var/www/ 目录

1.3.3 Bash 脚本连接

- 执行多条命令

```
#!/bin/bash
ssh user@remoteNode << eeooff
cd /home
touch abcdefg.txt
exit
eeooff
echo done!
```

```
#!/bin/bash

# 变量定义
ip_array=("192.168.1.1" "192.168.1.2" "192.168.1.3")
user="test1"
remote_cmd="/home/test/1.sh"

# 本地通过 ssh 执行远程服务器的脚本
for ip in ${ip_array[*]}
do
    if [ $ip = "192.168.1.1" ]; then
        port="7777"
    else
        port="22"
    fi
    ssh -t -p $port $user@$ip "remote_cmd"
done
```

CHAPTER 2

数据存储与消费

2.1 Mysql

2.1.1 索引

索引用于：

- 快速找出匹配一个 WHERE 子句的行；
- 当执行联结时，从其他表检索行；
- 对特定的索引列找出 MAX() 或 MIN() 值；
- 如果排序或分组在一个可用键的最左面前缀上进行 (例如，ORDER BY key_part_1,key_part_2)，排序或分组一个表。
- 如果所有键值部分跟随 DESC，键以倒序被读取。
- 在一些情况中，一个查询能被优化来检索值，不用咨询数据文件。

如果对某些表的所有使用的列是数字型的并且构成某些键的最左面前缀，为了更快，值可以从索引树被检索出来。

索引原理

索引要求做到减少 IO，加速查询

- **在表中有大量数据的前提下，创建索引速度会很慢**

  在没有数据的情况下先建索引或者说目录快，还是已经存在好多的数据了，然后再去建索引，哪个快，肯定是没有数据的时候快，因为如果已经有了很多数据了，你再去根据这些数据建索引，是不是要将数据全部遍历一遍，然后根据数据建立索引。

- **在索引创建完毕后，对表的查询性能会幅度提升，但是写性能会降低**

索引建立好之后再添加数据快，还是没有索引的时候添加数据快，索引是用来干什么的，是用来加速查询的，那对你写入数据会有什么影响，肯定是慢一些了，因为你但凡加入一些新的数据，都需要把索引或者说书的目录重新做一个，**所以索引虽然会加快查询，但是会降低写入的效率**。

磁盘 IO 与预读

磁盘读取数据靠的是机械运动，每次读取数据花费的时间可以分为**寻道时间**、**旋转延迟**、**传输时间**三个部分。

- 寻道时间指的是磁臂移动到指定磁道所需要的时间，主流磁盘一般在 5ms 以下；
- 旋转延迟就是我们经常听说的磁盘转速，比如一个磁盘 7200 转/min，表示每分钟能转 7200 次，也就是说 1 秒钟能转 120 次，旋转延迟就是 1/120/2 = 4.17ms，也就是半圈的时间（这里有两个时间：平均寻道时间，受限于目前的物理水平，大概是 5ms 的时间，找到磁道了，还需要找到你数据存在的那个点，寻点时间，这寻点时间的一个平均值就是半圈的时间，这个半圈时间叫做平均延迟时间，那么平均延迟时间加上平均寻道时间就是你找到一个数据所消耗的平均时间，大概 9ms，其实机械硬盘慢主要是慢在这两个时间上了，当找到数据然后把数据拷贝到内存的时间是非常短暂的，和光速差不多了）；
- 传输时间指的是从磁盘读出或将数据写入磁盘的时间，一般在零点几毫秒，相对于前两个时间可以忽略不计

那么访问一次磁盘的时间，即一次磁盘 IO 的时间约等于 5+4.17 = 9ms 左右。

计算机硬件延迟的对比图：

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

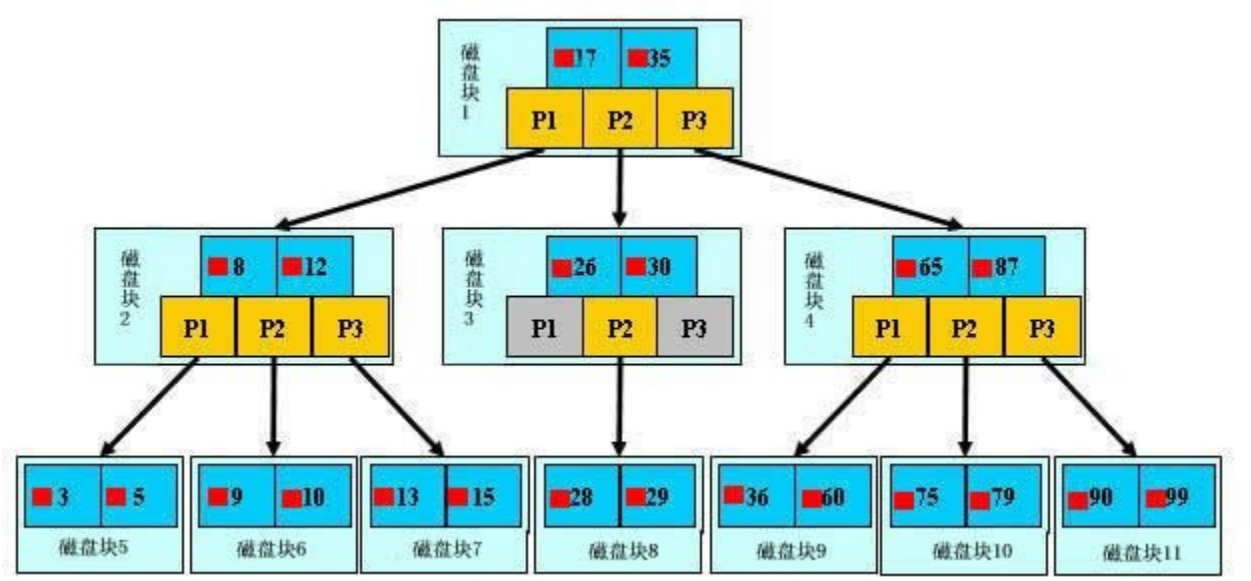
考虑到磁盘 IO 是非常高昂的操作，计算机操作系统做了一些优化，**当一次 IO 时，不光把当前磁盘地址的数据，而是把相邻的数据也都读取到内存缓冲区内**，因为局部预读性原理告诉我们，当计算机访问一个地址的数据的时候，与其相邻的数据也会很快被访问到。每一次 IO 读取的数据我们称之为一页 (page)。具体一页有多大数据跟操作系统有关，一般为 4k 或 8k，也就是我们读取一页内的数据时候，实际上才发生了一次 IO，这个理论对于索引的数据结构设计非常有帮助。

索引的数据结构

红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用 B-/+Tree 作为索引结构.

一般来说，索引本身也很大，不可能全部存储在内存中，因此**索引往往以索引文件的形式存储的磁盘上**。这样的话，索引查找过程中就要产生磁盘 I/O 消耗，索引的结构组织要尽量减少查找过程中磁盘 I/O 的存取次数，最好是常数数量级。

详解 b+ 树



如上图，是一颗 b+ 树，关于 b+ 树的定义可以参见B+ 树，这里只说一些重点，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块 1 包含数据项 17 和 35，包含指针 P1、P2、P3，P1 表示小于 17 的磁盘块，P2 表示在 17 和 35 之间的磁盘块，P3 表示大于 35 的磁盘块。真实的数据存在于叶子节点即 3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如 17、35 并不真实存在于数据表中。

b+ 树的查找过程

如图所示，如果要查找数据项 29，那么首先会把磁盘块 1 由磁盘加载到内存，此时发生一次 IO，在内存中用二分查找确定 29 在 17 和 35 之间，锁定磁盘块 1 的 P2 指针，内存时间因为非常短（相比磁盘的 IO）可以忽略不计，通过磁盘块 1 的 P2 指针的磁盘地址把磁盘块 3 由磁盘加载到内存，发生第二次 IO，29 在 26 和 30 之间，锁定磁盘块 3 的 P2 指针，通过指针加载磁盘块 8 到内存，发生第三次 IO，同时内存中做二分查找找到 29，结束查询，总计三次 IO。真实的情况是，3 层的 b+ 树可以表示上百万的数据，如果上百万的数据查找只需要三次 IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次 IO，那么总共需要百万次的 IO，显然成本非常非常高。

b+ 树性质

1**. 索引字段要尽量的小 **：通过上面的分析，我们知道 IO 次数取决于 b+ 数的高度 h 或者说层级，这个高度或者层级就是你每次查询数据的 IO 次数，**假设当前数据表的数据为 N，每个磁盘块的数据项的数量是 m，则有 h= 🄴 (m+1)N，当数据量 N 一定的情况下，m 越大，h 越小；而 m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低**。这就是为什么每个数据项，即索引字段要尽量的小，比如 int 占 4 字节，要比 bigint8 字节少一半。这也是为什么 b+ 树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于 1 时将会退化成线性表。

2. **索引的最左匹配特性**：简单来说就是你的数据来了以后，从数据块的左边开始匹配，在匹配右边的.

我们继续学下面的内容。当 b+ 树的数据项是复合的数据结构，比如 (name,age,sex) 的时候，b+ 数是按照从左到右的顺序来建立搜索树的，比如当 (张三,20,F) 这样的数据来检索的时候，b+ 树会优先比较 name 来确定下一步的所搜方向，如果 name 相同再依次比较 age 和 sex，最后得到检索的数据；但当 (20,F) 这样的没有 name 的数据来的时候，b+ 树就不知道下一步该查哪个节点，因为建立搜索树的时候 name 就是第一个比较因子，必须要先根据 name 来搜索才能知道下一步去哪里查询。比如当 (张三,F) 这样的数据来检索时，b+ 树可以用 name 来指定搜索方向，但下一个字段 age 的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是 F 的数据了，这个是非常重要的性质，即索引的最左匹配特性。

建索引的几大原则

1. 最左前缀匹配原则，非常重要的原则，mysql 会一直向右匹配直到遇到范围查询 (>、<、between、like) 就停止匹配，比如 a = 1 and b = 2 and c > 3 and d = 4 如果建立 (a,b,c,d) 顺序的索引，d 是用不到索引的，如果建立 (a,b,d,c) 的索引则都可以用到，a,b,d 的顺序可以任意调整。

2.= 和 in 可以乱序，比如 a = 1 and b = 2 and c = 3 建立 (a,b,c) 索引可以任意顺序，mysql 的查询优化器会帮你优化成索引可以识别的形式。

3. 尽量选择区分度高的列作为索引，区分度的公式是 count(distinct col)/count(*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是 1，而一些状态、性别字段可能在大数据面前区分度就是 0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要 join 的字段我们都要求是 0.1 以上，即平均 1 条扫描 10 条记录。

4. 索引列不能参与计算，保持列“干净”，比如 from_unixtime(create_time) = ’2014-05-29’ 就不能使用到索引，原因很简单，b+ 树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成 create_time = unix_timestamp(’2014-05-29’)。

5. 尽量的扩展索引，不要新建索引。比如表中已经有 a 的索引，现在要加 (a,b) 的索引，那么只需要修改原来的索引即可。

MySQL 索引管理

功能

```
#1. 索引的功能就是加速查找
#2. mysql 中的 primary key, unique, 联合唯一也都是索引, 这些索引除了加速查找以外, 还有约束的功能
```

MySQL 常用的索引

```
普通索引 INDEX：加速查找

唯一索引：
    -主键索引 PRIMARY KEY：加速查找 + 约束（不为空、不能重复）
    -唯一索引 UNIQUE：加速查找 + 约束（不能重复）

联合索引：
    -PRIMARY KEY(id,name)：联合主键索引
    -UNIQUE(id,name)：联合唯一索引
    -INDEX(id,name)：联合普通索引
```

参考如下：

MySQL 索引原理及慢查询优化

MySQL 索引

2.1.2 gorm

创建连接

```
 db, err := gorm.Open("mysql", "user:password@/dbname?charset=utf8&parseTime=True&
↪loc=Local")
```

数据模型定义

表名，列名如何对应结构体

在 Gorm 中，表名是结构体名的复数形式，列名是字段名的蛇形小写。

即，如果有一个 Order 表，那么如果你定义的结构体名为：User，gorm 会默认表名为 orders 而不是 order。

```
CREATE TABLE `orders` (
  `order_id` varchar(3) NOT NULL,
  `status` bigint(20) DEFAULT NULL,
  PRIMARY KEY (`order_id`),
  KEY `idx_orders_status` (`status`),
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

那么对应的结构体定义如下：

```
type Order struct {
        OrderId         string `gorm:"PRIMARY_KEY;type:varchar(3)" json:"orderId"`
        Status          int64  `gorm:"index;" json:"status"` //索引
}
```

如何全局禁用表名复数呢?

可以在创建数据库连接的时候设置如下参数：

```
// 全局禁用表名复数
db.SingularTable(true) // 如果设置为 true,`User`的默认表名为 `user`, 使用 `TableName`设置的表名不受影响
```

这样的话，表名默认即为结构体的首字母小写形式。

生成表

当数据库需要频繁更新结构时, 代码与数据库难以保持一致是烦人的问题. 而 golang 的 gorm 库, 有 Auto Migration 功能, 可以根据 go 里的 struct tag 自动更新数据库结构, 非常方便。

```
//自动生成表
db.AutoMigrate(&Order{})
```

查询

```
query := db.Model(Order{}).Where("order_id = ? AND status = ?", order_id, status)
var count int
err := db.Model(&Order{}).Where(&Order{order_id: id, status: status}).Count(&count).
↪Error
```

先用 `db.Model()` 选择一个表，再用 `db.Where()` 构造查询条件，后面可以使用 `db.Count()` 计算数量，如果要获取对象，可以使用 `db.Find(&Likes)` 或者只需要查一条记录 `db.First(&Order)`

事务

要在事务中执行一组操作，一般流程如下。

```
func CreateAnimals(db *gorm.DB) err {
  tx := db.Begin()
  // 注意，一旦你在一个事务中，使用 tx 作为数据库句柄

  if err := tx.Create(&Animal{Name: "Giraffe"}).Error; err != nil {
     tx.Rollback()
     return err
  }
  // 在事务中做一些数据库操作（从这一点使用'tx'，而不是'db'）
  if err := tx.Create(&Animal{Name: "Lion"}).Error; err != nil {
     tx.Rollback()
     return err
  }

  tx.Commit()
  return nil
}
```

日志

Gorm 有内置的日志记录器支持，默认情况下，它会打印发生的错误。

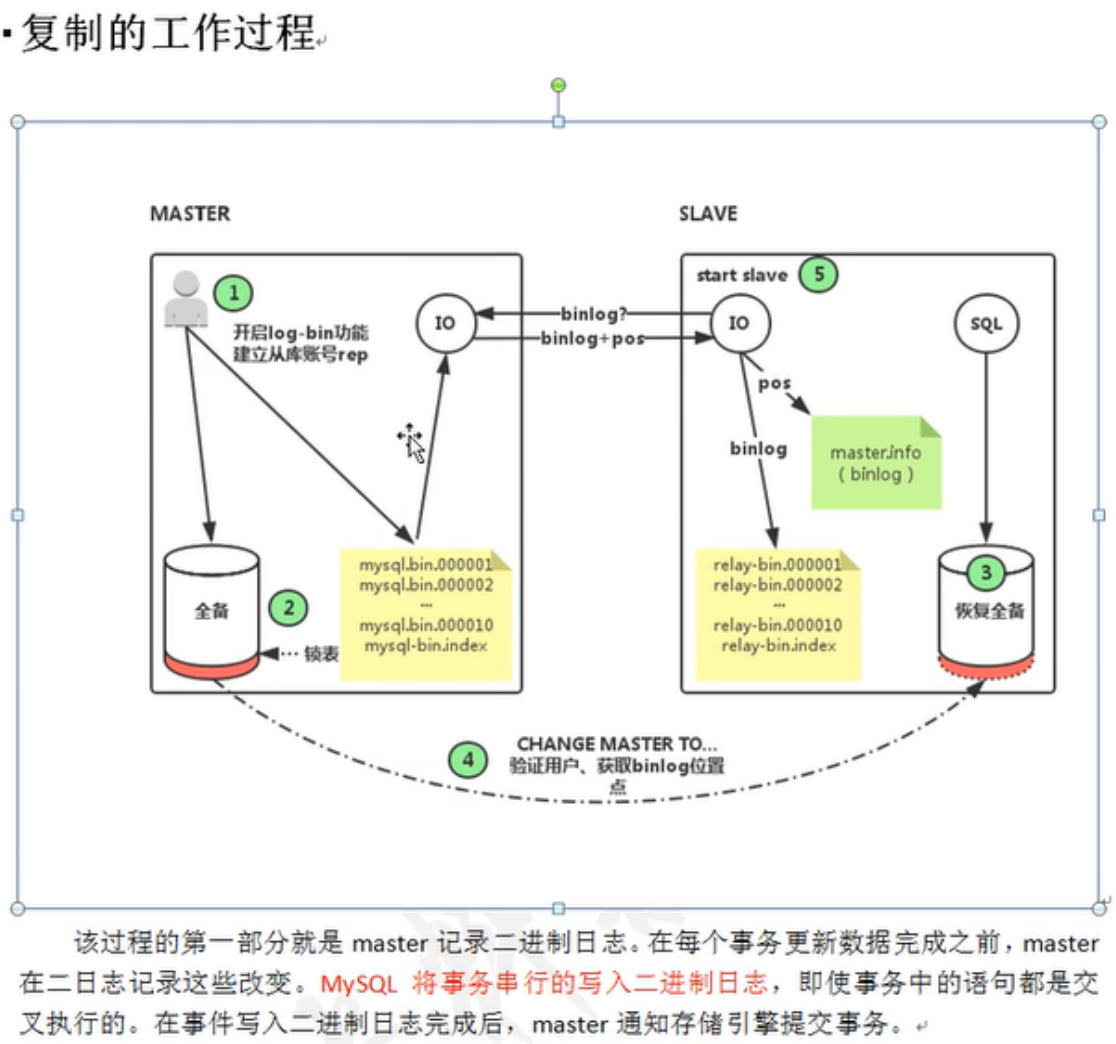
```
// 启用 Logger，显示详细日志
db.LogMode(true)

// 禁用日志记录器，不显示任何日志
db.LogMode(false)

// 调试单个操作，显示此操作的详细日志
db.Debug().Where("name = ?", "xiaoming").First(&User{})
```

参考如下：

Go orm 框架 gorm 学习

2.1.3 使用 Docker 搭建 MySQL 主从复制

主从复制原理



复制原理 mysql 的复制原理大致如下。

- 主库开启 bin-log 二进制日志功能，并建立 slave 账号，并授权从库连接主库，从库通过 change master 得到主库的相关同步信息。mysql 主库在事务提交时会把数据库变更作为事件 Events 记录在二进制文件 binlog 中；mysql 主库上的 sys_binlog 控制 binlog 日志刷新到磁盘。
- 从库然后连接主库进行验证，主库产生的新数据会导入到 bin-log 二进制文件中，同时主库会开启 IO 线程，从库也会开启 IO 线程以及 SQL 线程，从库中的 IO 线程与主库的 IO 线程连接一旦主库库数据有所变更则从库将变更的数据复制到 relary-bin（中继日志）中。SQL 线程会读取 relay log（中继日志）文件中的日志，并解析成具体操作，至此整个同步过程完成。

主从复制操作

流程命令集

```
# 1、搭建部署 mysql
docker run --name mysql1 -p 33061:3306 -e MYSQL_ROOT_PASSWORD=123 -d mysql:5.7 --
↪character-set-server=utf8mb4 --collation-server=utf8mb4_unicode_ci
docker run --name mysql2 -p 33062:3306 -e MYSQL_ROOT_PASSWORD=123 -d mysql:5.7 --
↪character-set-server=utf8mb4 --collation-server=utf8mb4_unicode_ci

# 2、主机配置
## 2.1、进入 mysql
mysql -u root -h 127.0.0.1 -P 33061 -p

## 2.2、 主机配置 配置一个从机登录用户
GRANT REPLICATION SLAVE ON *.* TO 'user'@'%' IDENTIFIED BY '123';

## 2.3、进入 docker 容器
docker exec -it mysql1 bash

## 2.4、需要修改 MySQL 的配置，开启 binlog
vi /etc/mysql/mysql.conf.d/mysqld.cnf


# 3、从机配置
## 3.1、进入 mysql
mysql -u root -h 127.0.0.1 -P 33061 -p


## 3.2、从配置 配置一个从机登录用户
change master to master_host='192.168.66.114',master_port=33061,master_user='user',
↪master_password='123',master_log_file='binlog.000001',master_log_pos=154;

## 3.3、 slave 启动或停止
start slave;
stop slave;

## 3.4、 查看 slave 配置正确与否
show slave status\G;

####Slave_IO_Running 和 Slave_SQL_Running ，这两个的值必须为 Yes。
```

配置相关文件

主机 MySQL 配置文件 mysqld.cnf

- binlog 的开启，需要修改 MySQL 的配置，因此，我们需要进入到容器内部去执行。

  `docker exec -it mysql1 /bin/bash`

- MySQL 配置文件`/etc/mysql/mysql.conf.d/mysqld.cnf` 追加以下内容

```
# 第一行表示配置 binlog 的位置，理论上 binlog 可以放在任意位置，但是该位置，MySQL 一定要有操作权限。
log-bin=/var/lib/mysql/binlog
```

(下页继续)

(续上页)

```
#server-id 表示集群中，每个实例的一标识符
server-id=1

# 不同步哪些数据库
binlog-ignore-db = mysql
binlog-ignore-db = test
binlog-ignore-db = information_schema

# 只同步哪些数据库，除此之外，其他不同步
binlog-do-db = game
```

```
mysql> show master status;
+---------------+----------+-----------------+------------------+-------------------+
| File          | Position | Binlog_Do_DB    | Binlog_Ignore_DB | Executed_Gtid_Set |
+---------------+----------+-----------------+------------------+-------------------+
| binlog.000001 |      154 | cmdb,db1,db2,db3 |                  |                   |
+---------------+----------+-----------------+------------------+-------------------+
1 row in set (0.00 sec)
```

File 和 Position 需要记着，这两个标记了二进制日志的起点位置，在从机的配置中将使用到这两个参数。

从机配置

- `show slave status;`
  - `Slave_IO_State`: 当前 `I/O` 线程的状态
  - `Master_Log_File`: 当前同步的主服务器的二进制文件
  - `Read_Master_Log_Pos`: 当前同步的主服务器的二进制文件的偏移量，单位为字节，如图中为已经同步了 `12.9M(13630580/1024/1024)` 的内容
  - `Relay_Master_Log_File`: 当前中继日志同步的二进制文件
  - `Slave_IO_Running`: 从服务器中 `I/O` 线程的运行状态，`YES` 为运行正常
  - `Slave_SQL_Running`: 从服务器中 `SQL` 线程的运行状态，`YES` 为运行正常
  - `Exec_Master_Log_Pos`: 表示同步完成的主服务器的二进制日志偏移量
  - `Seconds_Behind_Master`: 表示从服务器数据比主服务器落后的持续时长

参考链接

使用 Docker 搭建 MySQL 主从复制

2.2 Redis

2.2.1 redis 使用规范

1、核心业务数据和边缘业务数据分离

这两类数据分开为不同的缓存实例，避免边缘业务影响核心业务服务；

2、数据缓存按业务功能分离

不能将众多的业务服务依赖同一缓存实例，而应该按业务进行拆分 (尤其是核心的高并发的数据应单独的缓存服务实例)。

3、缓存 key 命名规范

包括三个维度来命名（命名空间、数据分类名、变量），三者之间采用冒号“:”隔开

4、提升访问性能（pipline）

多个操作命令发送到服务端，客户端不依赖每个操作命令返回的结果（即可批量化）时，可以使用 pipline 机制减少 tcp 往返次数，提升 redis 访问性能；

5、实现优雅删除（unlink）

del 指令会同步释放对象内存，大部分情况下没有问题，当遇到比较大的 key 时（如上千万记录的 list、hash 等），则该指令会造成服务卡顿；因此 redis 从 4.0 版本开始发布了 unlink 指令来解决该问题。unlink 用于替代 del 指令，解决大 key 删除卡顿问题。在执行 unlink 指令时 redis 第一步将 key 从 redis 空间摘除此 key 的引用，第二步会在一个异步线程中释放此 key 的内存（因是异步的，不会对工作线程造成影响）。

6、安全遍历（scan）

线上应避免使用 keys 指令，keys 会遍历 redis 中所有的 key，当 key 的数据达到千万时，执行该指令会导致服务卡顿；可以采用 scan 指令带上 limit 参数来替代 keys。

redis 是单线程的。keys 指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用 **scan** 指令，**scan** 指令可以无阻塞的提取出指定模式的 key 列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用 keys 指令长。

7、预防缓存击穿（缓存默认值）

当查询的数据在 Redis 数据库本身不存在时，每次请求一般都会查询数据库。恶意用户会利用此场景对数据库进行攻击从而影响服务。可以采用下发的方式进行预防：

从数据库查询不到数据时，在缓存中设置一个特殊的默认值并设置一个较短的过期时间。

常见攻击

- 缓存击穿当我们缓存 key 设置过期时间，恰巧在这一刻这个 key 在某一刻被高并发的访问，把所有的请求都打到了 DB 中这就可能会导致 DB 挂了。

  缓存击穿的话，设置热点数据永远不过期。或者加上互斥锁就能搞定了

  或者 **使用分布锁，使一个线程去重新回原数据，其他请求返回空值，UI 友好提示**。

- 缓存穿透使用大量不存在的 key 发布访问，由于缓存中找不到，故把所有请求全都打到 DB 中，使用 DB 压力巨增，导致 DB 挂掉。

  缓存穿透我会在接口层增加校验，比如用户鉴权校验，参数做校验，不合法的参数直接代码 Return，比如：id 做基础校验，id <=0 的直接拦截等。

  1. 使用 Boolm 过滤器，2. 使用 Hystrix 第三方组件进行限流、溶断，从而保护服务器。

- 缓存雪崩雪崩指的是多个 key 同时失效，当高并发的请求过来所有请求都打到数据库导致挂了

  处理缓存雪崩简单，在批量往 Redis 存数据的时候，把每个 Key 的失效时间都加个随机值就好了，这样可以保证数据不会在同一时间大面积失效

  ```
  setRedis (Key, value, time + Math.random() * 10000);
  ```

  或者设置热点数据永远不过期，有更新操作就更新缓存就好了

8、缓存失效（加锁）

高并发时，如果缓存失效，会有大量的请求打到数据库，对数据库造成瞬间冲击，影响服务质量，因此在查询数据库加载到缓存时需要加锁，以预防数据库瞬间流量冲击。

2.2.2 redis 应用场景

一、分布式锁

二、数据存储

redis 可用作数据缓存，包括五种类型：

- **String:**

  Strings 数据结构是简单的 key-value 类型，value 其实不仅是 String，也可以是数字。

  常用命令: set,get,decr,incr,mget 等。

  应用场景：String 是最常用的一种数据类型，普通的 key/ value 存储都可以归为此类. 即可以完全实现目前 Memcached 的功能，并且效率更高。还可以享受 Redis 的定时持久化，操作日志及 Replication 等功能。除了提供与 Memcached 一样的 get、set、incr、decr 等操作外，Redis 还提供了下面一些操作：

  获取字符串长度

  往字符串 append 内容

  设置和获取字符串的某一段内容

  设置及获取字符串的某一位（bit）

  批量设置一系列字符串的内容

  实现方式：String 在 redis 内部存储默认就是一个字符串，被 redisObject 所引用，当遇到 incr,decr 等操作时会转成数值型进行计算，此时 redisObject 的 encoding 字段为 int。

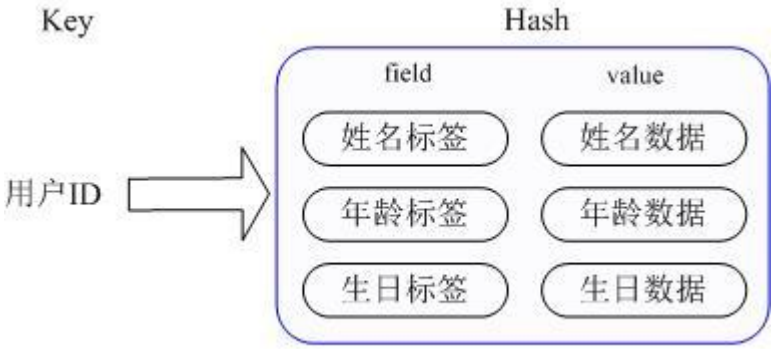
扩容过程:

当 len 长度小于 1M 时，扩容时双倍扩容

当 len 长度大于 1M 时，每次扩容 1M 容量

- **Hash:**

  Redis 的 Hash 实际是内部存储的 Value 为一个 HashMap，并提供了直接存取这个 Map 成员的接口，如下图:



  也就是说，Key 仍然是用户 ID, value 是一个 Map，这个 Map 的 key 是成员的属性名，value 是属性值，这样对数据的修改和存取都可以直接通过其内部 Map 的 Key(Redis 里称内部 Map 的 key 为 field), 也就是通过 key(用户 ID) + field(属性标签) 就可以操作对应属性数据了，既不需要重复存储数据，也不会带来序列化和并发修改控制的问题。很好的解决了问题。

  常用命令：hget,hset,hgetall 等。

  HSET 命令用来给字段赋值，而 HGET 命令用来获得字段的值。用法如下:

```
HSET key field value
HGET key field
redis > HSET car price 500
(integer) 1
redis > HSET car name BMW
(integer) 1
redis > HGET car name
"BMW"

HMSET key field1 value1 field2 value2
redis > HMGET car price name
1) "500"
2) "BMW"
```

  HSET 命令的方便之处在于不区分插入和更新操作，这意味着修改数据时不用事先判断字段是否存在来决定要执行的是插入操作（update）还是更新操作（insert）。当执行的是插入操作时（即之前字段不存在）HSET 命令会返回 1，当执行的是更新操作时（即之前字段已经存在）HSET 命令会返回 0。更进一步，当键本身不存在时，HSET 命令还会自动建立它。

  Redis 提供了接口 (hgetall) 可以直接取到全部的属性数据, 但是如果内部 Map 的成员很多，那么涉及到遍历整个内部 Map 的操作，由于 Redis 单线程模型的缘故，这个遍历操作可能会比较耗时，而另其它客户端的请求完全不响应，这点需要格外注意。

  实现方式:

  上面已经说到 Redis Hash 对应 Value 内部实际就是一个 HashMap，实际这里会有 2 种不同实现，这个 Hash 的成员比较少时 Redis 为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正

的 HashMap 结构，对应的 value redisObject 的 encoding 为 zipmap, 当成员数量增大时会自动转成真正的 HashMap, 此时 encoding 为 ht。

- **List：**

  常用命令：lpush,rpush,lpop,rpop,lrange 等。

  应用场景：

  Redis list 的应用场景非常多，也是 Redis 最重要的数据结构之一，比如 twitter 的关注列表，粉丝列表等都可以用 Redis 的 list 结构来实现。

  Lists 就是链表，相信略有数据结构知识的人都应该能理解其结构。使用 Lists 结构，我们可以轻松地实现最新消息排行等功能。Lists 的另一个应用就是消息队列，可以利用 Lists 的 PUSH 操作，将任务存在 Lists 中，然后工作线程再用 POP 操作将任务取出进行执行。Redis 还提供了操作 Lists 中某一段的 api，你可以直接查询，删除 Lists 中某一段的元素。

  实现方式：

  Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis 内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。

- **Set**

  常用命令：sadd,spop,smembers,sunion 等。

  应用场景：

  Redis set 对外提供的功能与 list 类似是一个列表的功能，特殊之处在于 set 是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择，并且 set 提供了判断某个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供的。

  Sets 集合的概念就是一堆不重复值的组合。利用 Redis 提供的 Sets 数据结构，可以存储一些集合性的数据，比如在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis 还为集合提供了求交集、并集、差集等操作，可以非常方便的实现如共同关注、共同喜好、二度好友等功能，对上面的所有集合操作，你还可以使用不同的命令选择将结果返回给客户端还是存集到一个新的集合中。

  实现方式：

  set 的内部实现是一个 value 永远为 null 的 HashMap，实际就是通过计算 hash 的方式来快速排重的，这也是 set 能提供判断一个成员是否在集合内的原因。

- **Sorted Set**

  常用命令：zadd,zrange,zrem,zcard 等

  使用场景：

  Redis sorted set 的使用场景与 set 类似，区别是 set 不是自动有序的，而 sorted set 可以通过用户额外提供一个优先级 (score) 的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择 sorted set 数据结构，比如 twitter 的 public timeline 可以以发表时间作为 score 来存储，这样获取时就是自动按时间排好序的。

  另外还可以用 Sorted Sets 来做带权重的队列，比如普通消息的 score 为 1，重要消息的 score 为 2，然后工作线程可以选择按 score 的倒序来获取工作任务。让重要的任务优先执行。

  实现方式：

  Redis sorted set 的内部使用 HashMap 和跳跃表 (SkipList) 来保证数据的存储和有序，HashMap 里放的是成员到 score 的映射，而跳跃表里存放的是所有的成员，排序依据是 HashMap 里存的 score, 使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

参考如下:

Redis 之数据存储结构

Redis 面试题

三、消息队列

一般使用 list 结构作为队列，rpush 生产消息，lpop 消费消息。当 lpop 没有消息的时候，要适当 sleep 一会再重试。

```
# 无限循环读取任务队列中的内容
loop
task=RPOR queue
if task
# 如果任务队列中有任务则执行它
execute( task)
else
# 如果没有则等待 1 秒以免过于频繁地请求数据
wait 1 second
```

当任务队列中没有任务时消费者每秒都会调用一次 RPOP 命令查看是否有新任务。如果可以实现一旦有新任务加入任务队列就通知消费者就好了。其实借助 BRPOP 命令就可以实现这样的需求。

BRPOP 命令和 RPOP 命令相似，唯一的区别是当列表中没有元素时 BRPOP 命令会一直阻塞住连接，在没有消息的时候，它会阻塞住直到有新元素加入。

```
loop
# 如果任务队列中没有新任务，BRPOP 命令会一直阻塞，不会执行 execute()。
task=BRPOP queue, 0
# 返回值是一个数组（见下介绍），数组第二个元素是我们需要的任务。
execute( task[1])
```

BRPOP 命令接收两个参数，第一个是键名，第二个是超时时间，单位是秒。当超过了此时间仍然没有获得新元素的话就会返回 nil。上例中超时时间为 “0”，表示不限制等待的时间，即如果没有新元素加入列表就会永远阻塞下去。

四、订阅推送

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在 Redis 中，你可以设定对某一个 key 值进行消息发布及消息订阅，当一个 key 值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

“发布/订阅” 模式中包含两种角色，分别是发布者和订阅者。订阅者可以订阅一个或若干个频道（channel），而发布者可以向指定的频道发送消息，所有订阅此频道的订阅者都会收到此消息。

pub/sub 在消费者下线的情况下，生产的消息会丢失。

2.2.3 redis 面试

Redis 采用的是基于内存的采用的是单进程单线程模型的 KV 数据库，由 C 语言编写，官方提供的数据是可以达到 100000+ 的 **QPS（每秒内查询次数）**。

- 完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。它的数据存在内存中，类似于 **HashMap**，**HashMap** 的优势就是查找和操作的时间复杂度都是 O(1)；
- 数据结构简单，对数据操作也简单，**Redis** 中的数据结构是专门进行设计的；
- 采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 **CPU**，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
- 使用多路 I/O 复用模型，非阻塞 IO，将速度优势发挥到最大，也提供了较简单的计算功能；
- 使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，**Redis** 直接自己构建了 VM 机制，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；
- Redis 提供了事务的功能，可以保证一串命令的原子性，中间不会被任何操作打断

Redis 虽然是单线程的，但是可以通过开多个 Redis 实例利用多核机器

Redis 持久化数据和缓存怎么做扩容？

- 如果 Redis 被当做缓存使用，使用一致性哈希实现动态扩容缩容。

  shared 一致性哈希采用以下方案：

  1. Redis 服务器节点划分：将每台服务器节点采用 hash 算法划分为 160 个虚拟节点 (可以配置划分权重)
  2. 将划分虚拟节点采用 TreeMap 存储
  3. 对每个 Redis 服务器的物理连接采用 LinkedHashMap 存储
  4. 对 Key or KeyTag 采用同样的 hash 算法，然后从 TreeMap 获取大于等于键 hash 值得节点，取最邻近节点存储；当 key 的 hash 值大于虚拟节点 hash 值得最大值时，存入第一个虚拟节点

  sharded 采用的 hash 算法：MD5 和 MurmurHash 两种；默认采用 64 位的 MurmurHash 算法；有兴趣的可以研究下，MurmurHash 是一种高效，低碰撞的 hash 算法；参考地址：

  http://blog.csdn.net/yfkiss/article/details/7337382

  https://sites.google.com/site/murmurhash/

- 如果 Redis 被当做一个持久化存储使用，必须使用固定的 keys-to-nodes 映射关系，节点的数量一旦确定不能变化。否则的话 (即 Redis 节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有 Redis 集群可以做到这样。

redis 的一些其他特点

- Redis 是单进程单线程的 redis 利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销
- 读写分离模型通过增加 Slave DB 的数量，读的性能可以线性增长。为了避免 Master DB 的单点故障，集群一般都会采用两台 Master DB 做双机热备，所以整个集群的读和写的可用性都非常高。读写分离架构的缺陷在于，不管是 Master 还是 Slave，每个节点都必须保存完整的数据，如果在数据量很大的情况下，集群的扩展能力还是受限于单个节点的存储能力，而且对于 Write-intensive 类型的应用，读写分离架构并不适合
- 数据分片模型为了解决读写分离模型的缺陷，可以将数据分片模型应用进来。可以将每个节点看成都是独立的 master，然后通过业务实现数据分片。结合上面两种模型，可以将每个 master 设计成由一个 master 和多个 slave 组成的模型。
- Redis 的回收策略
  - volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘
  - volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
  - volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
  - allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
  - allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
  - no-enviction（驱逐）：禁止驱逐数据

注意这里的 6 种机制，volatile 和 allkeys 规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据，后面的 lru、ttl 以及 random 是三种不同的淘汰策略，再加上一种 no-enviction 永不回收的策略。使用策略规则： 如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率低，则使用 allkeys-lru 如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用 allkeys-random

Redis 虽然是一种内存型数据库，一旦服务器进程退出，数据库的数据就会丢失，为了解决这个问题 Redis 提供了两种持久化的方案，将内存中的数据保存到磁盘中，避免数据的丢失。

RDB 持久化

redis 提供了 RDB 持久化的功能，这个功能可以将 redis 在内存中的的状态保存到硬盘中，它可以**手动执行**，也可以再 redis.conf 中配置，**定期执行**。

RDB 持久化产生的 RDB 文件是一个**经过压缩**的二进制文件，这个文件被保存在硬盘中，redis 可以通过这个文件还原数据库当时的状态。

RDB 的创建与载入

RDB 文件可以通过两个命令来生成：

- SAVE：阻塞 redis 的**服务器进程**，直到 RDB 文件被创建完毕。
- BGSAVE：派生 (fork) 一个子进程来创建新的 RDB 文件，记录接收到 BGSAVE 当时的数据库状态，父进程继续处理接收到的命令，子进程完成文件的创建之后，会**发送信号**给父进程，而与此同时，父进程处理命令的同时，通过**轮询**来接收子进程的信号。

而 RDB 文件的载入一般情况是自动的，redis 服务器启动的时候，redis 服务器再启动的时候如果检测到 RDB 文件的存在，那么 redis 会自动载入这个文件。

如果服务器开启了 AOF 持久化，那么服务器会优先使用 AOF 文件来还原数据库状态。

RDB 是通过保存键值对来记录数据库状态的，采用 copy on write 的模式，每次都是全量的备份。

自动保存间隔

BGSAVE 可以在不阻塞主进程的情况下完成数据的备份。可以通过 redis.conf 中设置多个自动保存条件，只要有一个条件被满足，服务器就会执行 BGSAVE 命令。

```
# 以下配置表示的条件:
# 服务器在 900 秒之内被修改了 1 次
save 900 1
# 服务器在 300 秒之内被修改了 10 次
save 300 10
# 服务器在 60 秒之内被修改了 10000 次
save 60 10000
```

AOF 持久化

AOF 持久化（Append-Only-File），与 RDB 持久化不同，AOF 持久化是通过保存 Redis 服务器锁执行的写状态来记录数据库的。

具体来说，RDB 持久化相当于备份数据库状态，而 AOF 持久化是备份数据库接收到的**命令**，所有被写入 AOF 的命令都是以 redis 的协议格式来保存的。

在 AOF 持久化的文件中，数据库会记录下所有变更数据库状态的命令，除了指定数据库的 select 命令，其他的命令都是来自 client 的，这些命令会以追加 (append) 的形式保存到文件中。

服务器配置中有一项 appendfsync，这个配置会影响服务器多久完成一次命令的记录：

- always：将缓存区的内容总是即时写到 AOF 文件中。
- everysec：将缓存区的内容每隔一秒写入 AOF 文件中。
- no：写入 AOF 文件中的操作由操作系统决定，一般而言为了提高效率，操作系统会等待缓存区被填满，才会开始同步数据到磁盘。

redis 默认实用的是 everysec。

redis 在载入 AOF 文件的时候，会创建一个虚拟的 client，把 AOF 中每一条命令都执行一遍，最终还原回数据库的状态，它的载入也是自动的。在 RDB 和 AOF 备份文件都有的情况下，redis 会优先载入 AOF 备份文件

AOF 文件可能会随着服务器运行的时间越来越大，可以利用 AOF 重写的功能，来控制 AOF 文件的大小。AOF 重写功能会首先读取数据库中现有的键值对状态，然后根据类型使用一条命令来替代前的键值对多条命令。

AOF 重写功能有大量写入操作，所以 redis 才用子进程来处理 AOF 重写。这里带来一个新的问题，由于处理重新的是子进程，这样意味着如果主线程的数据在此时被修改，备份的数据和主库的数据将会有不一致的情况发生。因此 redis 还设置了一个 AOF 重写缓冲区，这个缓冲区在子进程被创建开始之后开始使用，这个期间，所有的命令会被存两份，一份在 AOF 缓存空间，一份在 AOF 重写缓冲区，当 AOF 重写完成之后，子进程发送信号给主进程，通知主进程将 AOF 重写缓冲区的内容添加到 AOF 文件中。

相关配置

```
#AOF 和 RDB 持久化方式可以同时启动并且无冲突。
# 如果 AOF 开启，启动 redis 时会加载 aof 文件，这些文件能够提供更好的保证。
appendonly yes

# 只增文件的文件名称。(默认是 appendonly.aof)
# appendfilename appendonly.aof
#redis 支持三种不同的写入方式:
#
# no: 不调用，之等待操作系统来清空缓冲区当操作系统要输出数据时。很快。
# always: 每次更新数据都写入仅增日志文件。慢，但是最安全。
# everysec: 每秒调用一次。折中。
appendfsync everysec

# 设置为 yes 表示 rewrite 期间对新写操作不 fsync，暂时存在内存中，等 rewrite 完成后再写入. 官方
文档建议如果你有特殊的情况可以配置为'yes'。但是配置为'no' 是最为安全的选择。
no-appendfsync-on-rewrite no

# 自动重写只增文件。
# redis 可以自动盲从的调用 'BGREWRITEAOF' 来重写日志文件，如果日志文件增长了指定的百分比。
# 当前 AOF 文件大小是上次日志重写得到 AOF 文件大小的二倍时，自动启动新的日志重写过程。
auto-aof-rewrite-percentage 100
# 当前 AOF 文件启动新的日志重写过程的最小值，避免刚刚启动 Reids 时由于文件尺寸较小导致频繁的重写。
auto-aof-rewrite-min-size 64mb
```

对比

- AOF 更安全，可将数据及时同步到文件中，但需要较多的磁盘 IO，AOF 文件尺寸较大，文件内容恢复相对较慢，也更完整。
- RDB 持久化，安全性较差，它是正常时期数据备份及 master-slave 数据同步的最佳手段，文件尺寸较小，恢复数度较快。

作者：whthomas 链接：https://www.jianshu.com/p/bedec93e5a7b 来源：简书著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

2.2.4 redis 分布式锁

分布式锁

分布式锁一般有三种实现方式：1. 数据库乐观锁；2. 基于 Redis 的分布式锁；3. 基于 ZooKeeper 的分布式锁。本篇博客将介绍第二种方式，基于 Redis 实现分布式锁。虽然网上已经有各种介绍 Redis 分布式锁实现的博客，然而他们的实现却有着各种各样的问题，为了避免误人子弟，本篇博客将详细介绍如何正确地实现 Redis 分布式锁。

可靠性

首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

1. **互斥性**在任意时刻，只有一个客户端能持有锁。
2. **不会发生死锁**即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
3. **具有容错性**只要大部分的 Redis 节点正常运行，客户端就可以加锁和解锁。
4. **解铃还须系铃人**加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

代码实现

组件依赖

首先我们要通过 Maven 引入 `Jedis` 开源组件，在 `pom.xml` 文件加入下面的代码：

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
```

加锁代码

Talk is cheap, show me the code。先展示代码，再带大家慢慢解释为什么这样实现：

```
public class RedisTool {

    private static final String LOCK_SUCCESS = "OK";
    private static final String SET_IF_NOT_EXIST = "NX";
    private static final String SET_WITH_EXPIRE_TIME = "PX";

    /**
     * 尝试获取分布式锁
     * @param jedis Redis 客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @param expireTime 超期时间
     * @return 是否获取成功
     */
    public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String␣
↪requestId, int expireTime) {

        String result = jedis.set(lockKey, requestId, SET_IF_NOT_EXIST, SET_WITH_
↪EXPIRE_TIME, expireTime);

        if (LOCK_SUCCESS.equals(result)) {
            return true;
        }
        return false;
    }
}
```

可以看到，我们加锁就一行代码：`jedis.set(String key, String value, String nxxx, String expx, int time)`，这个 set() 方法一共有五个形参：

- 第一个为 key，我们使用 key 来当锁，因为 key 是唯一的。
- 第二个为 value，我们传的是 requestId，很多童鞋可能不明白，有 key 作为锁不就够了吗，为什么还要用到 value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件**解铃还须系铃人**，通过给 value 赋值为 requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId 可以使用 `UUID.randomUUID().toString()` 方法生成。
- 第三个为 nxxx，这个参数我们填的是 NX，意思是 SET IF NOT EXIST，即当 key 不存在时，我们进行 set 操作；若 key 已经存在，则不做任何操作；
- 第四个为 expx，这个参数我们传的是 PX，意思是我们要给这个 key 加一个过期的设置，具体时间由第五个参数决定。
- 第五个为 time，与第四个参数相呼应，代表 key 的过期时间。

总的来说，执行上面的 set() 方法就只会导致两种结果：1. 当前没有锁（key 不存在），那么就进行加锁操作，并对锁设置个有效期，同时 value 表示加锁的客户端。2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足我们**可靠性**里描述的三个条件。首先，set() 加入了 NX 参数，可以保证如果已有 key 存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即 key 被删除），不会发生死锁。最后，因为我们将 value 赋值为 requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑 Redis 单机部署的场景，所以容错性我们暂不考虑。

解锁代码

还是先展示代码，再带大家慢慢解释为什么这样实现：

```
public class RedisTool {

    private static final Long RELEASE_SUCCESS = 1L;

    /**
     * 释放分布式锁
     * @param jedis Redis 客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @return 是否释放成功
     */
    public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String␣
↪requestId) {

        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.
↪call('del', KEYS[1]) else return 0 end";
        Object result = jedis.eval(script, Collections.singletonList(lockKey),␣
↪Collections.singletonList(requestId));

        if (RELEASE_SUCCESS.equals(result)) {
            return true;
        }
        return false;
    }
}
```

可以看到，我们解锁只需要两行代码就搞定了！第一行代码，我们写了一个简单的 Lua 脚本代码，上一次见到这个编程语言还是在《黑客与画家》里，没想到这次居然用上了。第二行代码，我们将 Lua 代码传到 `jedis.eval()` 方法里，并使参数 KEYS[1] 赋值为 lockKey，ARGV[1] 赋值为 requestId。eval() 方法是将 Lua 代码交给 Redis 服务端执行。

那么这段 Lua 代码的功能是什么呢？其实很简单，首先获取锁对应的 value 值，检查是否与 requestId 相等，如果相等则删除锁（解锁）。那么为什么要使用 Lua 语言来实现呢？因为要确保上述操作是原子性的。关于非原子性会带来什么问题，会在锁已经不属于当前客户端的时候会解除他人加的锁。那么为什么执行 eval() 方法可以确保原子性，源于 Redis 的特性：

简单来说，就是在 eval 命令执行 Lua 代码的时候，Lua 代码将被当成一个命令去执行，并且直到 eval 命令执行完成，Redis 才会执行其他命令。

脚本

```
//加锁
conn.Do("SET", lockKey, locker, "PX", expiredInMS, "NX")

// 解锁脚本
var unlockScript = redis.NewScript(1, `
   if redis.call("get", KEYS[1]) == ARGV[1]
   then
      return redis.call("del", KEYS[1])
   else
      return 0
   end
`)

var unlockScriptWithState = redis.NewScript(1, `
   if redis.call("get", KEYS[1]) == ARGV[1]
   then
      redis.call("set", ARGV[2], ARGV[3])
      return redis.call("del", KEYS[1])
   else
      return -1
   end
`)
```

原文链接

Redis 分布式锁的正确实现方式（Java 版）

2.3 Pulsar

2.3.1 pulsar-go

客户端

Pulsar 协议 url 使用 `pulsar` scheme 来指定被连接的集群，默认端口为 6650。以下是 `localhost` 的示例:

```
pulsar://localhost:6650
```

生产环境的 Pulsar 集群 URL 类似这样：

```
pulsar://pulsar.us-west.example.com:6650
```

创建客户端

```
import (
    "log"
    "runtime"

    "github.com/apache/pulsar/pulsar-client-go/pulsar"
)

func main() {
    client, err := pulsar.NewClient(pulsar.ClientOptions{
        URL: "pulsar://localhost:6650",
        OperationTimeoutSeconds: 5,
        MessageListenerThreads: runtime.NumCPU(),
    })

    if err != nil {
        log.Fatalf("Could not instantiate Pulsar client: %v", err)
    }
}
```

Producers

```
producer, err := client.CreateProducer(pulsar.ProducerOptions{
    Topic: "my-topic",
})

if err != nil {
    log.Fatalf("Could not instantiate Pulsar producer: %v", err)
}

defer producer.Close()

msg := pulsar.ProducerMessage{
    Payload: []byte("Hello, Pulsar"),
}

if err := producer.Send(msg); err != nil {
    log.Fatalf("Producer could not send message: %v", err)
}
```

Producer operations

Consumers

消费者订阅一个或多个主题，并听取在该主题/这些主题上产生的传入消息。

```
msgChannel := make(chan pulsar.ConsumerMessage)

consumerOpts := pulsar.ConsumerOptions{
    Topic:            "my-topic",
    SubscriptionName: "my-subscription-1",
    Type:             pulsar.Exclusive,
    MessageChannel:   msgChannel,
}

consumer, err := client.Subscribe(consumerOpts)

if err != nil {
    log.Fatalf("Could not establish subscription: %v", err)
}

defer consumer.Close()

for cm := range msgChannel {
    msg := cm.Message

    fmt.Printf("Message ID: %s", msg.ID())
    fmt.Printf("Message value: %s", string(msg.Payload()))

    consumer.Ack(msg)
}
```

Consumer operations

Readers

Pulsar readers process messages from Pulsar topics

readers 与 consumers 不同，因为使用读取器，您需要明确指定要从流中开始的消息（另一方面，consumers 会自动以最新的未确认消息开始）。

```
reader, err := client.CreateReader(pulsar.ReaderOptions{
    Topic: "my-golang-topic",
    StartMessageId: pulsar.LatestMessage,
})
```

CHAPTER 3

BlockChain

3.1 Consensus

3.1.1 ACID 原则与多阶段算法

ACID 原则

具体来说，ACID 原则描述了分布式数据库需要满足的一致性需求，同时允许付出可用性的代价。

- Atomicity(原子性)：每次事务是原子的，事务包含的所有操作要么全部成功，要么全部不执行。一旦有操作失败，则需要回退状态到执行事务之前；
- Consistency（一致性）：数据库的状态在事务执行前后的状态是一致的和完整的，无中间状态。即只能处于成功事务提交后的状态；
- Isolation（隔离性）：各种事务可以并发执行，但彼此之间互相不影响。按照标准 SQL 规范，从弱到强可以分为未授权读取、授权读取、可重复读取和串行化四种隔离等级；
- Durability（持久性）：状态的改变是持久的，不会失效。一旦某个事务提交，则它造成的状态变更就是永久性的。

两阶段提交算法（Two-phase Commit，2PC）

其基本思想十分简单，既然在分布式场景下，直接提交事务可能出现各种故障和冲突，那么可将其分解为预提交和正式提交两个阶段，规避冲突的风险。

- 预提交：协调者（Coordinator）发起提交某个事务的申请，各参与执行者（Participant）需要尝试进行提交并反馈是否能完成；
- 正式提交：协调者如果得到所有执行者的成功答复，则发出正式提交请求。如果成功完成，则算法执行成功。

在此过程中任何步骤出现问题（例如预提交阶段有执行者回复预计无法完成提交），则需要回退。

两阶段提交算法因为其简单容易实现的优点，在关系型数据库等系统中被广泛应用。当然，其缺点也很明显。整个过程需要同步阻塞导致性能一般较差；同时存在单点问题，较坏情况下可能一直无法完成提交；另外可能产生数据不一致的情况（例如协调者和执行者在第二个阶段出现故障）。

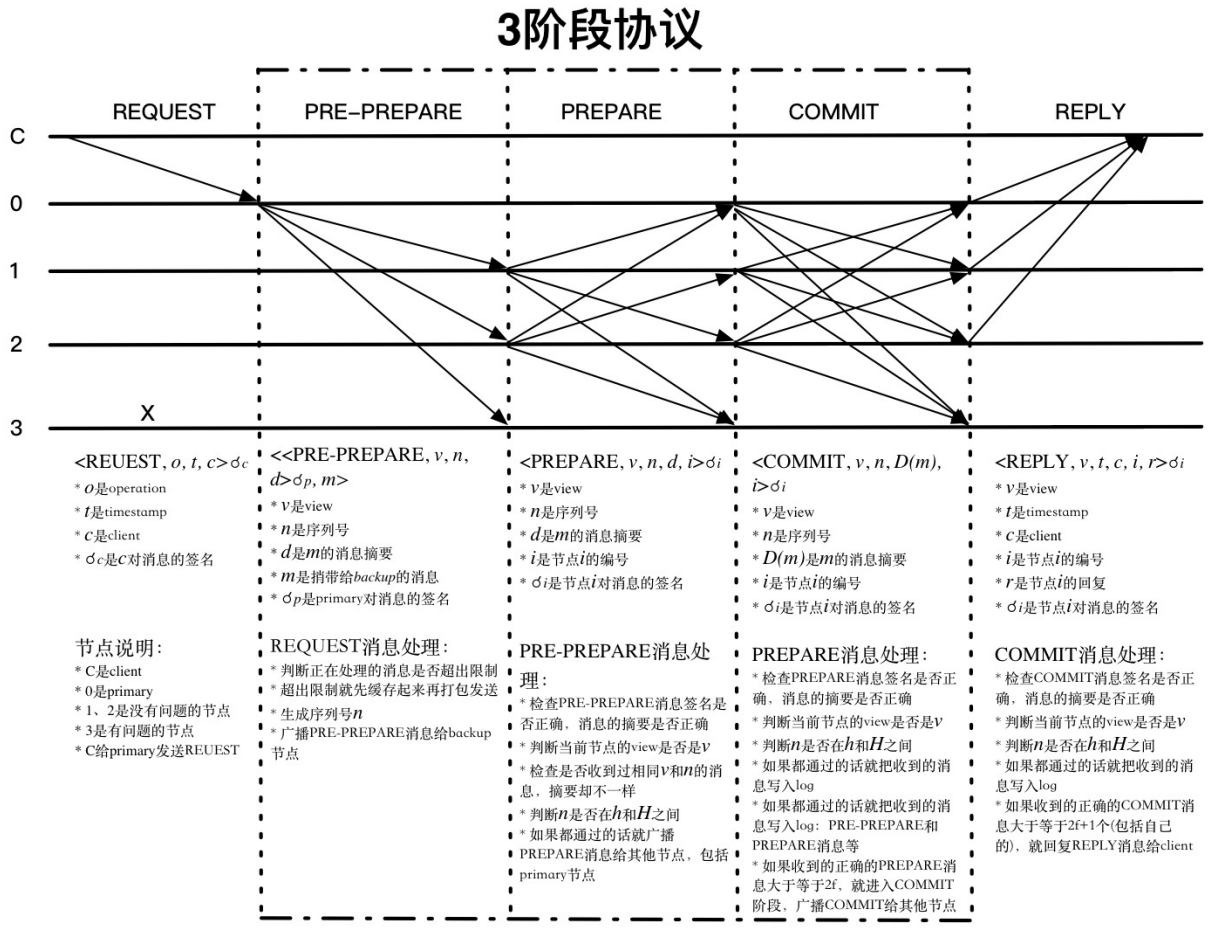
三阶段提交算法（Three-phase Commit，3PC）。

三阶段提交针对两阶段提交算法第一阶段中可能阻塞部分执行者的情况进行了优化。具体来说，将预提交阶段进一步拆成两个步骤：尝试预提交和预提交。

完整过程如下：

- 尝试预提交：协调者询问执行者是否能进行某个事务的提交。执行者需要返回答复，但无需执行提交。这就避免出现部分执行者被无效阻塞住的情况。
- 预提交：协调者检查收集到的答复，如果全部为真，则发起提交事务请求。各参与执行者（Participant）需要尝试进行提交并反馈是否能完成；
- 正式提交：协调者如果得到所有执行者的成功答复，则发出正式提交请求。如果成功完成，则算法执行成功。

其实，无论两阶段还是三阶段提交，都只是一定程度上缓解了提交冲突的问题，并无法一定保证系统的一致性。首个有效的算法是后来提出的 Paxos 算法。



3.1.2 FLP and CAP

分布式系统的核心问题：**一致性问题共识问题**引起不一致的因素:

- 节点间网络通信的不可靠, 消息延迟, 消息乱序, 内容错误.
- 节点处理时间无法保证: 结果可能错误, 或者节点宕机.

FLP 不可能原理

FLP 不可能原理：在网络可靠，但允许节点失效（即便只有一个）的最小化异步模型系统中，不存在一个可以解决一致性问题的确定性共识算法。（节点失效不可避免，所以不能共识）

FLP 不可能原理告诉大家不必浪费时间去追求完美的共识方案，而要根据实际情况设计可行的工程方案。

那么，退一步讲，在付出一些代价的情况下，共识能做到多好?

回答这一问题的是另一个很出名的原理：CAP 原理。

CAP 原理

CAP 原理：分布式计算系统不可能同时确保以下三个特性：强一致性（Consistency）、可用性（Availability）和分区容忍性（Partition），设计中往往需要弱化对某个特性的保证。

- **一致性**：所有节点在同一时刻能够看到相同的数据。
- **可用性**：在有限时间内，任何非失败节点都能应答请求；
- **分区容忍性**：分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性或可用性的服务。

一致性

- 对于客户端来说，一致性指的是并发访问时更新过的数据如何获取的问题。
- 从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。

规范来看，分布式系统达成一致的过程，应该满足:

- 可终止性（Termination）：一致的结果在有限时间内能完成；意味着可以保障提供服务（Liveness）。
- 约同性（Agreement）：不同节点最终完成决策的结果是相同的；
- 合法性（Validity）：决策的结果必须是某个节点提出的提案。

CAP 原理认为，分布式系统最多只能保证三项特性中的两项特性。

比较直观地理解，当网络可能出现分区时候，系统是无法同时保证一致性和可用性的。要么，节点收到请求后因为没有得到其它节点的确认而不应答（牺牲可用性），要么节点只能应答非一致的结果（牺牲一致性）。

CAP 应用场景

既然 CAP 三种特性不可同时得到保障，则设计系统时候必然要弱化对某个特性的支持。

弱化一致性

对结果一致性不敏感的应用，可以允许在新版本上线后过一段时间才最终更新成功，期间不保证一致性。

例如网站静态页面内容、实时性较弱的查询类数据库等，简单分布式同步协议如 Gossip，以及 CouchDB、Cassandra 数据库等，都为此设计。

弱化可用性

对结果一致性很敏感的应用，例如银行取款机，当系统故障时候会拒绝服务。MongoDB、Redis、MapReduce 等为此设计。

Paxos、Raft 等共识算法，主要处理这种情况。在 Paxos 类算法中，可能存在着无法提供可用结果的情形，同时允许少数节点离线。

弱化分区容忍性

现实中，网络分区出现概率较小，但很难完全避免。

两阶段的提交算法，某些关系型数据库以及 ZooKeeper 主要考虑了这种设计。

实践中，网络可以通过双通道等机制增强可靠性，实现高稳定的网络通信。

共识算法

针对如上不稳定的情况，把出现故障（Crash 或 Fail-stop，即不响应）但不会伪造信息的情况称为“非拜占庭错误（Non-Byzantine Fault）”或“故障错误（Crash Fault）”；伪造信息恶意响应的情况称为“拜占庭错误”（Byzantine Fault），对应节点为拜占庭节点。显然，后者场景中因为存在“捣乱者”更难达成共识。

共识算法:

- BFT 类针对拜占庭错误有: PBFT 确定性算法、POW 工作量证明的概率算法等.
- CFT 类针对非拜占庭错误有: PAXOS 算法、Raft 算法（容错较好, 处理快, 保证不超过一半节点故障）

参考如下:

常用共识算法总结（Paxos，Raft，PBFT，PoW，PoS，DPoS，Ripple）

blockchain_guide

3.1.3 Raft 算法

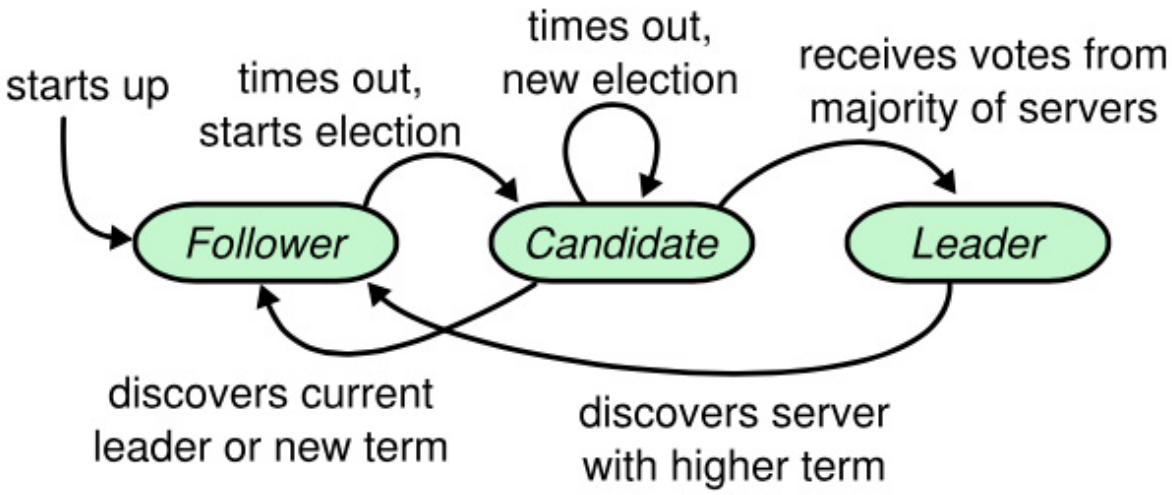
Raft 协议分为两个阶段

一、节点角色

角色定义

- **领导者 Leader**
- **候选者 Candidate**
- **追随者 Follower**

角色变化



服务器状态。**Follower** 只响应来自其他服务器的请求。如果 **Follower** 接收不到消息，那么他就会变成 **Candidate** 并发起一次选举。获得集群中大多数选票的 **Candidate** 将成为 **Leader**。在一个任期内，**Leader** 一直都会是 **Leader** 直到自己宕机了。

二、RPC 调用

AppendEntries RPC

由 Leader 负责调用来复制日志指令；也会用作 heartbeat

接收服务器需实现以下处理逻辑：

- 1、如果任期编号比本地小 `term < currentTerm`，则返回 `false`(5.1 节讨论)；
- 2、如果 **Follower** 在 prevLogIndex 索引处不包含 prevLogTerm 任期条目，则返回 `false`(5.3 节讨论)；
- 3、如果 **Follower** 有条目与新条目冲突 (索引相同任期不同)，则删除该条目以及所有后续条目 (5.3 节讨论)；

- 4、追加所有未在日志中的新条目；
- 5、如果 `leaderCommit > commitIndex`, 则将 commitIndex 更新为 leaderCommit 以及最后新条目索引两者的较小值；

RequestVote RPC

由候选人负责调用用来征集选票

接收服务器需实现以下处理逻辑：

- 1、如果 `term < currentTerm` 返回 false （5.2 节）
- 2、如果 votedFor 为空或者为 candidateId，并且 **Candidate** 的日志至少和自己一样新，那么就投票给他（5.2 节，5.4 节）

所有服务器需遵守的规则：

所有服务器：

- 如果 `commitIndex > lastApplied`，那么就 lastApplied 加一，并把 `log[lastApplied]` 条目应用到状态机中（5.3 节）
- 如果接收到的 RPC 请求或响应中，任期号 `T > currentTerm`，那么就令 currentTerm 等于 T，并切换状态为 **Follwer**（5.1 节）

Follwer（5.2 节）：

- 响应来自 **Candidate** 和 **Leader** 的请求
- 如果在超过选举超时时间的情况之前都没有收到 **Leader** 的心跳，或者是 **Candidate** 请求投票的，就自己变成 **Candidate**

Candidate（5.2 节）：

- 在转变成 **Candidate** 后就立即开始选举过程
  - 自增当前的任期号（currentTerm）
  - 给自己投票
  - 重置选举超时计时器
  - 发送 **RequestVote RPC** 给其他所有服务器
- 如果接收到大多数服务器的选票，那么就变成 **Leader**
- 如果接收到来自新的 **Leader** 的 **AppendEntries RPC**，转变成 **Follwer**
- 如果选举过程超时，再次发起一轮选举

Leader:

- 一旦成为 **Leader**：发送空的 **AppendEntries RPC** （心跳）给其他所有的服务器；在一定的空余时间之后不停的重复发送，以阻止 **Follwer** 超时（5.2 节）
- 如果接收到来自客户端的请求：附加条目到本地日志中，在条目被应用到状态机后响应客户端（5.3 节）
- 如果对于一个 **Follwer**，最后日志条目的索引值大于等于 nextIndex，即 `lastIndex >= nextIndex`，则发起 **AppendEntries RPC** 追加从 nextIndex 开始的日志条目：
  - 如果成功：更新相应 **Follwer** 的 nextIndex 和 matchIndex
  - 如果因为日志不一致而失败，减少 nextIndex 重试, 直到成功。
- 如果存在一个满足 `N > commitIndex` 的 N，并且大多数 **Follwer** 的 `matchIndex[i] ≥ N` 成立，并且 `log[N].term == currentTerm` 成立，那么令 commitIndex 等于这个 N （5.3 和 5.4 节）

阶段操作

选举阶段

Raft 采用 **心跳机制**来触发 **领袖选举**。服务器启动后，开始以 **Follwer** 角色运行。只要它不断收到来自 **Leader** 或者 **Candidate** 的 RPC 请求，便保持 **Follwer** 状态不变。**Leader** 发送周期性心跳 (不带任何条目的 **AppendEntries RPC** 请求) 给所有 **Follwer** ，以保持领导权。如果 **Follwer** 超过一定时间 (选举超时时间) 没有收到任何通讯，它便假设当前没有可见的的 **Leader**，进而发起新选举。

为发起选举，**Follwer** 自增自身任期，并转换为 **Candidate**。随后它为自己投票，同时向其他服务器发起 RequestVote RPC 请求。**Candidate** 持续这个状态直到发生以下三种情形之一：

- **Candidate** 赢得选举, 成为 **Leader**, 随后它向其他所有服务器发送心跳信息，建立领导权并阻止新的选举。
- 另一台机器以 **Leader** 身份连接 **Candidate** , 如果 **Leader** 领袖任期 `term` 至少与 **Candidate** 一样大，**Candidate** 便认为 **Leader** 合法，并重回 **Follwer** 状态。相反，**Candidate** 将拒绝 RPC 请求并继续保持 **Candidate** 状态。
- 超过设定时间但未产生获胜者, 开启一次新的选举——自增任期并开始下一轮 RequestVote 请求。

日志复制

Leader 被选举出来之后，开始服务客户端请求。每个客户端请求包含一个可以被复制状态机执行的命令。**Leader** 将命令最为新条目追加到本地日志，然后并行发起 **AppendEntries RPC** 请求往其他机器复制该条目。一旦条目安全完成复制 (已复制到过半数机器)，**Leader** 将把条目应用到自己的状态机并将执行结果告知客户端。如果 **Follwer** 节点宕机、响应缓慢或者网络丢包，**Leader** 将不断重试 AppendEntries RPC 请求 (就算已经响应客户端)，直到该节点日志同步。

解决问题

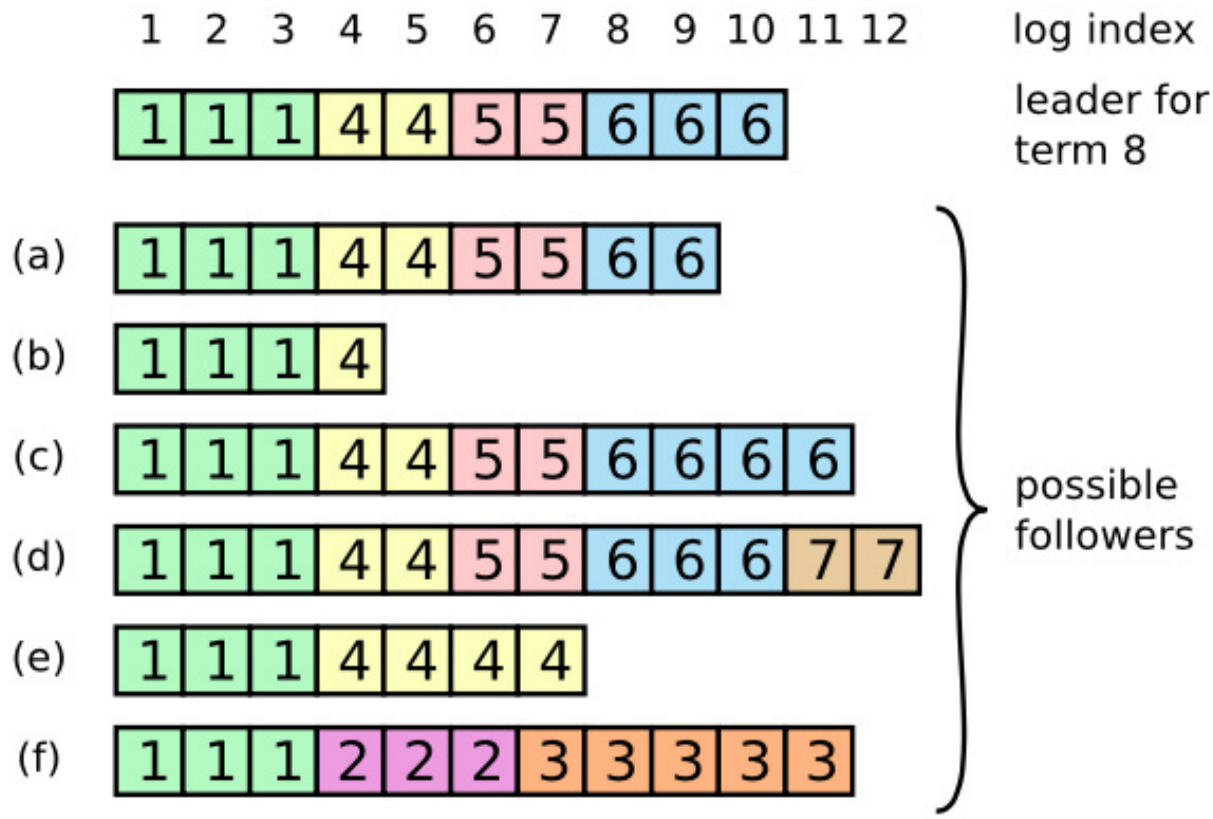
日志冲突解决

一致性检查

在发送 **AppendEntries RPC** 的时候，**Leader** 顺便带上前一日志条目的索引以及任期编号。**Follwer** 节点如果找不到对应的条目，它将拒绝新条目。

一致性检查就像一个归纳步骤：一开始空的日志状态肯定是满足日志匹配特性的，然后一致性检查保护了日志匹配特性当日志扩展的时候。因此，每当 **AppendEntries RPC** 返回成功时，领导人就知道跟随者的日志一定是和自己相同的了。

不一致，Leader 强制 Follwer 直接复制自己的日志



当顶部 **Leader** 节点启动后，从属节点日志状态存在各种可能：a-f 。每个格子表示一个日志条目；数字代表任期。**Follwer** 节点可能缺失部分日志条目 (a-b)；也可能包含多余的未提交日志；或者同时出现 (e-f)。举个例子，情景 f 可能是这样产生的：该服务器是任期 2 的 **Leader**，追加了几个条目，但是还没提交就宕机了；它迅速重启，在任期 3 继续担任 **Leader**，又追加了若干条目；在新条目提交前，服务器再次宕机，错过了接下来几个任期。

为了让 **Follwer** 日志与自己保持一致，**Leader** 需要找到双方最后一个匹配的条目，从该点开始删除 **Follwer** 所有不一致条目，并发送所有后续条目。这些操作根据 **AppendEntries RPC** 返回的一致性检查结果视情况执行。**Leader** 为每台 **Follwer** 维护一个 nextIndex 变量，保存下一个发送给 **Follwer** 的日志条目编号。当 **Leader** 刚开始服务时，将 nextIndex 设置为自己日志中下一个条目的索引 (索引 11)。如果 **Follwer** 日志与领袖不

一致，下一次 **AppendEntries RPC** 请求一致性检查将失败。请求被拒绝后，领袖将降低 nextIndex 并重试 **AppendEntries RPC** 请求。最终 nextIndex 将到达一个双方日志都匹配的点 (还可以用二分查找加快这个过程)。这时，**AppendEntries** 请求将成功，请求将删除属下所有有冲突条目并从领袖日志追加新条目 (如有)。一旦 **AppendEntries RPC** 请求成功，意味着属下日志与领袖一致，任期内后续时间也是如此。

选举约束

任何依赖领袖的共识算法，**Leader** 必须拥有所有已提交日志条目。

Raft 采用一种更简化的做法，选举保证新 **Leader** 必须包含先前任期所有已提交条目，无须传输缺失条目。这意味着，日志条目只有一个流向，从 **Leader** 流向 **Follwer**，**Leader** 不会覆盖日志中已存在的条目。

Raft 通过选举过程阻止 **Candidate** 赢得选举，除非它拥有所有已提交日志。**Candidate** 想赢得选举必须与集群内大多数机器通讯，这意味着每个已提交条目必须出现在这些机器中至少一台。只要 **Candidate** 日志至少与其他大多数机器一样新 (up-to-date)，它便一定拥有所有已提交条目。**RequestVote RPC** 请求实现这个约束：该请求带有候选人日志信息，其他节点发现自己日志更新 (即更加新，more up-to-date) 则拒绝投票。

Raft 以日志最后条目的索引以及任期判断哪个日志更新。如果最后条目任期不同，则任期靠后的更新；如果最后条目任期相同索引不同，则索引大 (日志更长) 的更新。

3.1.4 PBFT 算法

raft 和 pbft 的最大容错节点数

pbft 算法的最大容错节点数量是（n-1）/3

raft 算法的最大容错节点数量是（n-1）/2

对于 pbft 算法，因为 pbft 算法的除了需要支持容错故障节点之外，还需要支持容错作恶节点。假设集群节点数为 N，有问题的节点为 f。有问题的节点中，可以既是故障节点，也可以是作恶节点，或者只是故障节点或者只是作恶节点.

- 第一种情况，f 个有问题节点既是故障节点，又是作恶节点，那么根据小数服从多数的原则，集群里正常节点只需要比 f 个节点再多一个节点，即 f+1 个节点，确节点的数量就会比故障节点数量多，那么集群就能达成共识。也就是说这种情况支持的最大容错节点数量是（n-1）/2。
- 第二种情况，故障节点和作恶节点都是不同的节点。那么就会有 f 个问题节点和 f 个故障节点，当发现节点是问题节点后，会被集群排除在外，剩下 f 个故障节点，那么根据小数服从多数的原则，集群里正常节点只需要比 f 个节点再多一个节点，即 f+1 个节点，确节点的数量就会比故障节点数量多，那么集群就能达成共识。所以，所有类型的节点数量加起来就是 f+1 个正确节点，f 个故障节点和 f 个问题节点，即 3f+1=n。

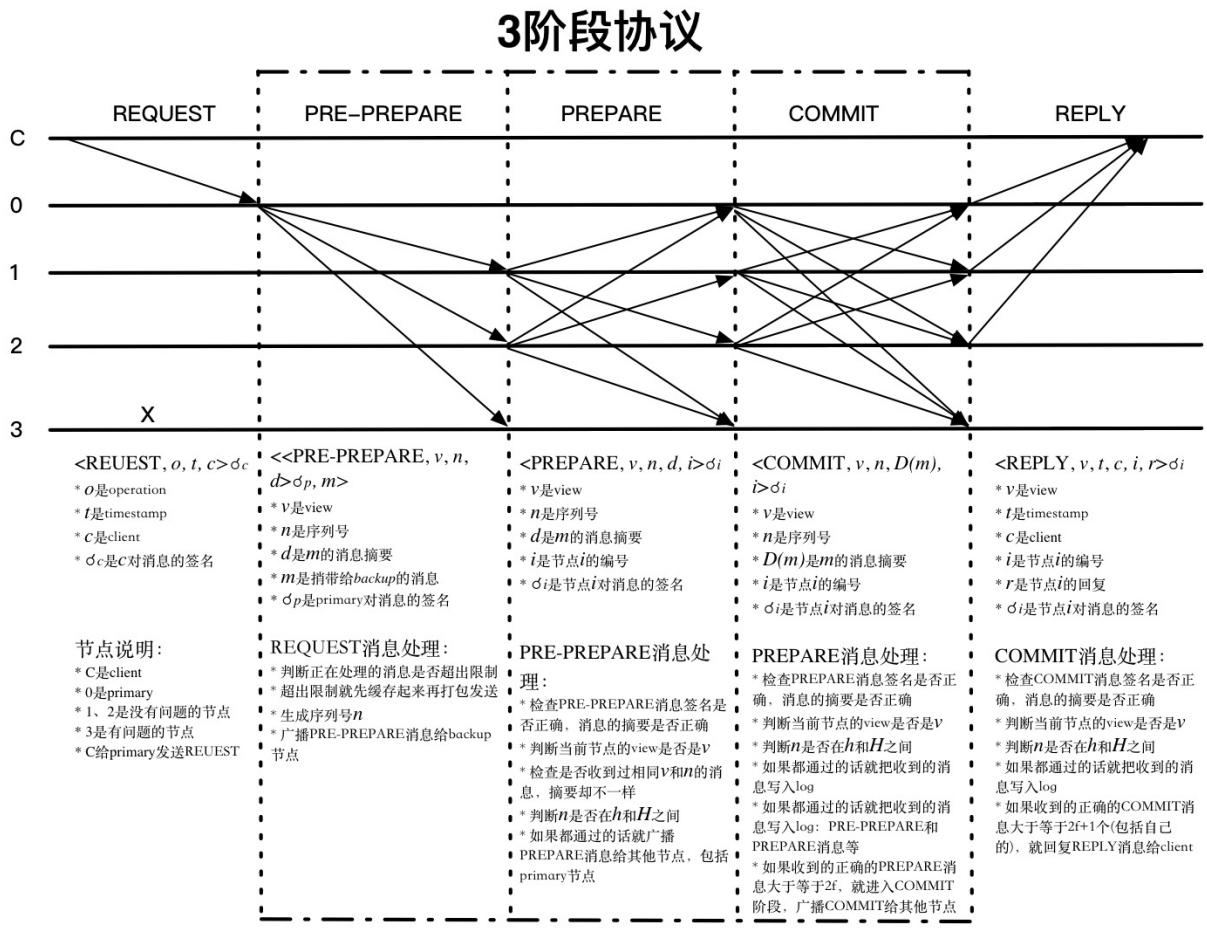
算法基本流程

流程

pbft 算法的基本流程主要有以下四步：

- 客户端发送请求给主节点
- 主节点广播请求给其它节点，节点执行 pbft 算法的三阶段共识流程。
- 节点处理完三阶段流程后，返回消息给客户端。
- 客户端收到来自 f+1 个节点的相同消息后，代表共识已经正确完成。

三阶段流程



- client：发出调用请求的实体
- view：连续的编号
- replica：网络节点
- primary：主节点，负责生成消息序列号
- backup：支撑节点
- state：节点状态

从 primary 收到消息开始，每个消息都会有 view 的编号，每个节点都会检查是否和自己的 view 是相同的，代表是哪个节点发送出来的消息，源头在哪里，client 收到消息也会检查该请求返回的所有消息是否是相同的 view。如果过程中发现 view 不相同，消息就不会被处理。除了检查 view 之外，每个节点收到消息的时候都会检查对应的序列号 n 是否匹配，还会检查相同 view 和 n 的 PRE-PREPARE、PREPARE 消息是否匹配，从协议的连续性上提供了一定程度的安全。

每个节点收到其他节点发送的消息，能够验证其签名确认发送来源，但并不能确认发送节点是否伪造了消息，PBFT 采用的办法就是数数，看有多少节点发送了相同的消息，在有问题的节点数有限的情况下，就能判断哪些节点发送的消息是真实的。**REQUEST 和 PRE-PREPARE 阶段还不涉及到消息的真实性，只是独立的生成或者确认 view 和序列号 n，所以收到消息判断来源后就广播出去了。PREPARE 阶段开始会汇总消息，通过数数判断消息的真实性**。

PREPARE 消息是收到 PRE-PREPARE 消息的节点发送出来的，primary 收到 REQUEST 消息后不会给自己发送 PRE-PREPARE 消息，也不会发送 PREPARE 消息，所以一个节点收到的消息数满足 **2f+1-1=2f** 个就能满足没问题的节点数比有问题节点多了（包括自身节点）。COMMIT 阶段 primary 节点也会在收到 PREPARE 消息后发送 COMMIT 消息，所以收到的消息数满足 2f+1 个就能满足没问题的节点数比有问题节点多了（包括自身节点）。

PRE-PREPARE 和 PREPARE 阶段保证了所有正常的节点对请求的处理顺序达成一致，它能够保证如果 `PREPARE(m, v, n, i)` 是真的话，`PREPARE(m', v, n, j)` 就一定是假的，其中 j 是任意一个正常节点的编号，只要 `D(m) != D(m')`。因为如果有 3f+1 个节点，至少有 f+1 个正常的节点发送了 PRE-PREPARE 和 PREPARE 消息，所以如果 `PREPARE(m', v, n, j)` 是真的话，这些节点中就至少有一个节点发了不同的 PRE-PREPARE 或者 PREPARE 消息，这和它是正常的节点不一致。当然，还有一个假设是安全强度是足够的，能够保证 m != m' 时，D(m) != D(m')，D(m) 是消息 m 的摘要。

确定好了每个请求的处理顺序，怎么能保证按照顺序执行呢？网络消息都是无序到达的，每个节点达成一致的顺序也是不一样的，有可能在某个节点上 n 比 n-1 先达成一致。其实每个节点都会把 PRE-PREPARE、PREPARE 和 COMMIT 消息缓存起来，它们都会有一个状态来标识现在处理的情况，然后再按顺序处理。而且序列号 n 在不同 view 中也是连续的，所以 n-1 处理完了，处理 n 就好了。

客户端 C 向主节点发起请求，主节点 0 收到客户端请求

```
<Request,o,t,c>σc
```

- σc 是 client 对请求消息的签名
- o 是 operation
- c 是 client

Pre-prepare 阶段

主节点收到`<Request,o,t,c>σc` 之后，生成序列号 n，广播 `Pre-prepare` 消息

```
<<Pre-Prepare,v,n,d>σp,m>
```

- v 是 view
- d 是消息的摘要
- n 是序列号
- σp 是 primary 对消息的签名
- m 是发给 backup 节点的消息

节点收到 pre-prepare 消息后，会有两种选择，一种是接受，一种是不接受。什么时候才不接受主节点发来的 pre-prepare 消息呢？一种典型的情况就是如果一个节点接受到了一条 pre-prepare 消息，消息里的 v 和 n 在之前收到里的消息是曾经出现过的，但是 d 和 m 却和之前的消息不一致，或者请求编号不在高低水位之间，这时候就会拒绝请求。**拒绝的逻辑就是主节点不会发送两条具有相同的 v 和 n，但 d 和 m 却不同的消息**。

Prepare 阶段

节点验证只要没有问题，就发送 prepare 消息（不涉及消息的真实性）

节点同意请求后会向其它节点发送 prepare 消息。这里要注意一点，同一时刻不是只有一个节点在进行这个过程，可能有 n 个节点也在进行这个过程。因此节点是有可能收到其它节点发送的 prepare 消息的。

```
<Prepare,v,n,d,i>σi
```

- i 是节点 i 的编号
- σi 是节点 i 对 prepare 消息的签名

节点收到了 prepare 消息之后，先判断 v,n 等数据签名没问题，如果没问题，就把收到的消息写入 log, 包括（pre-prepare,prepare 消息）

在一定时间范围内，如果收到超过 2f 个不同节点的 prepare 消息，就代表 prepare 阶段已经完成。

Commit 阶段

节点进入 commit 阶段。向其它节点广播 commit 消息，同理，这个过程可能是有 n 个节点也在进行的。因此可能会收到其它节点发过来的 commit 消息。

```
<Commit,v,n,D(m),i>σi
```

- D(m) 是 m 的信息摘要

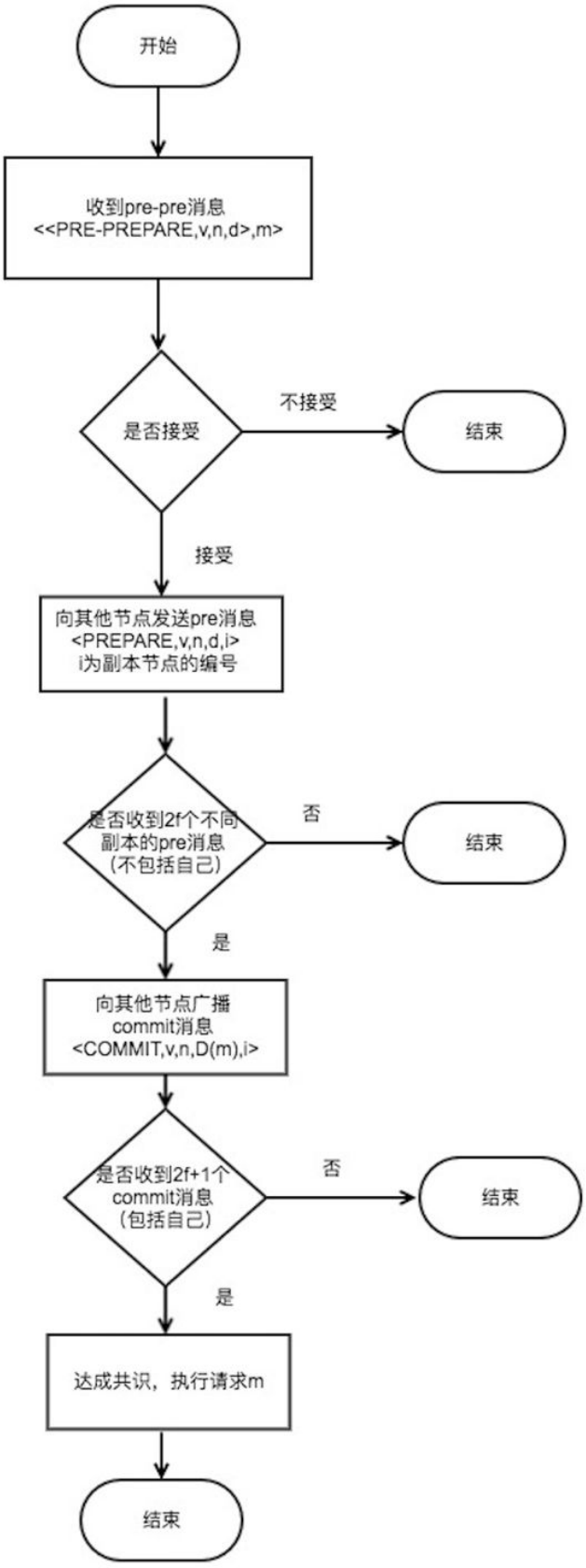
当收到 2f+1 个 commit 消息后（包括自己），代表大多数节点已经进入 commit 阶段，这一阶段已经达成共识，于是节点就会执行请求，写入数据。

Reply

节点达成共识，会回复 Reply 消息给 client

```
<Reply,v,t,c,i,r>σi
```

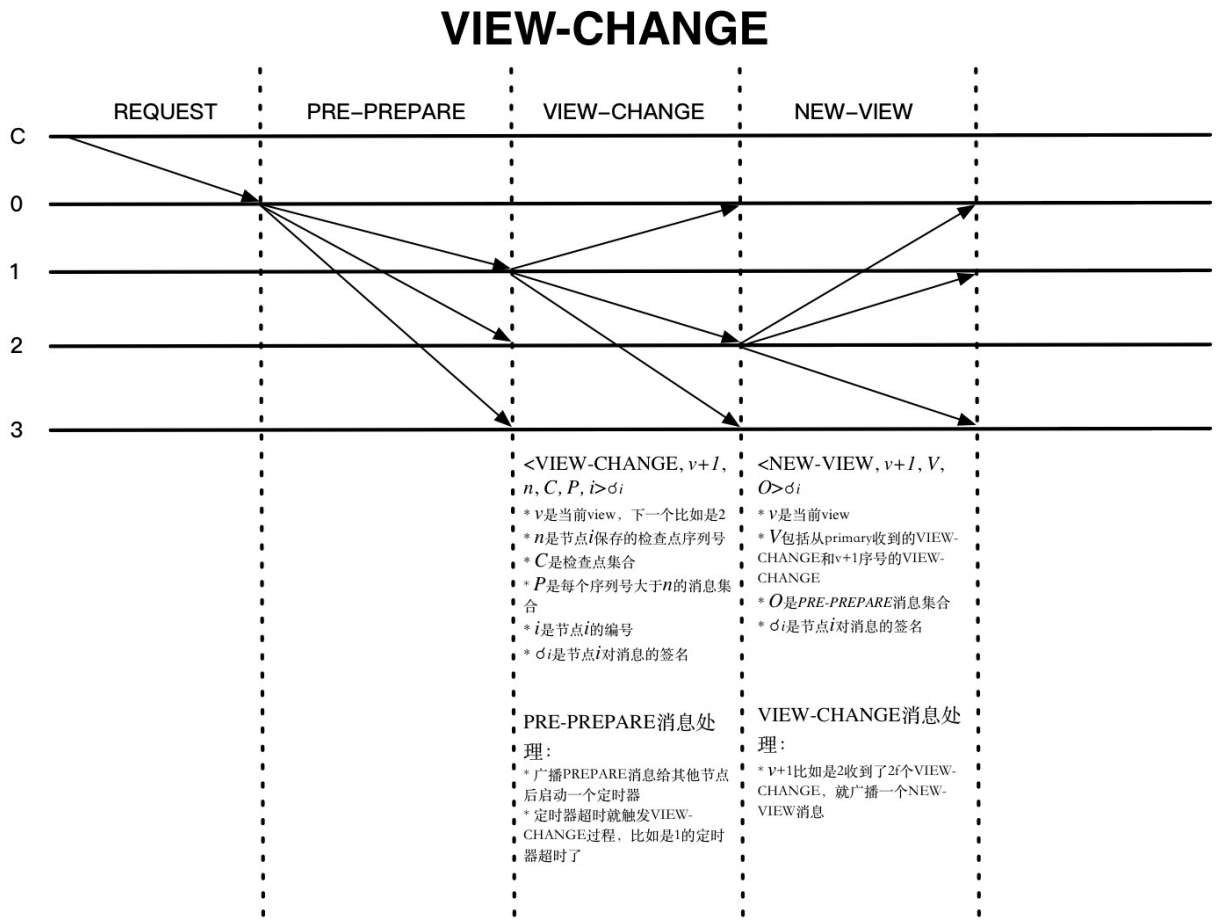
- v 是 view
- t 是时间戳
- r 是节点 i 的回复
- σi 是节点对 Reply 消息的签名


开始
收到pre-pre消息
<<PRE-PREPARE,v,n,d>,m>
是否接受
不接受
结束
接受
向其他节点发送pre消息
<PREPARE,v,n,d,i>
i为副本节点的编号
是否收到2f个不同
副本的pre消息
（不包括自己）
否
结束
是
向其他节点广播
commit消息
<COMMIT,v,n,D(m),i>
是否收到2f+1个
commit消息
（包括自己）
否
结束
是
达成共识，执行请求m
结束

所有的流程如上图所示

ViewChange（视图更改）事件

当主节点挂了（超时无响应）或者从节点集体认为主节点是问题节点时，就会触发 ViewChange 事件，ViewChange 完成后，视图编号将会加 1 。



viewchange 会有三个阶段，分别是 view-change ， view-change-ack 和 new-view 阶段。从节点认为主节点有问题时，会向其它节点发送 view-change 消息，当前存活的节点编号最小的节点将成为新的主节点。

```
<View-Change,v+1,C,P,Q,i>
```

- h 是节点 i 的最新 stable checkpoin
- C 是每个 checkpoint < 序号, digest> 的集合
- 集合 P 存储了之前 view 中到达 prepared 状态的请求。数据结构为 `<n,d,v>` 意为节点 i 收集到 view v 中序号为 n，digest 为 d 的请求的 prepared certificate，并且节点 i 上的下一个 view 中就不能有相同序号的 prepare 请求。
- 集合 Q 存储了之前 view 中到达 pre-prepared 状态，数据结构为 <n,d,v> 意为 i 已经 pre-prepare 了一个请求并且在下一 view 中这个请求没有相同序号的 pre-prepare 请求。

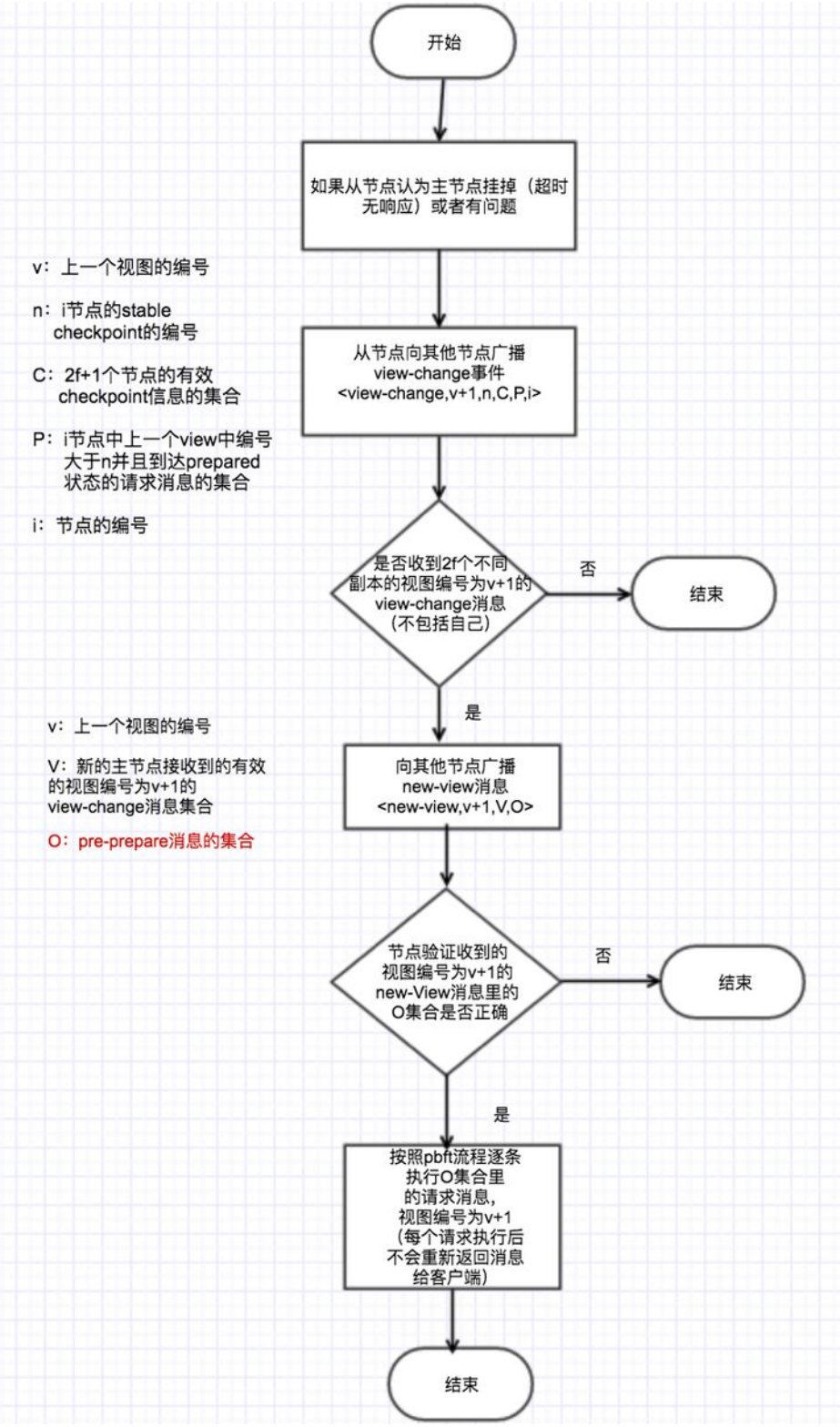
副本收集 VIEW-CHANGE 消息后发送 acknowledgments 以达成切换到 v+1 的共识

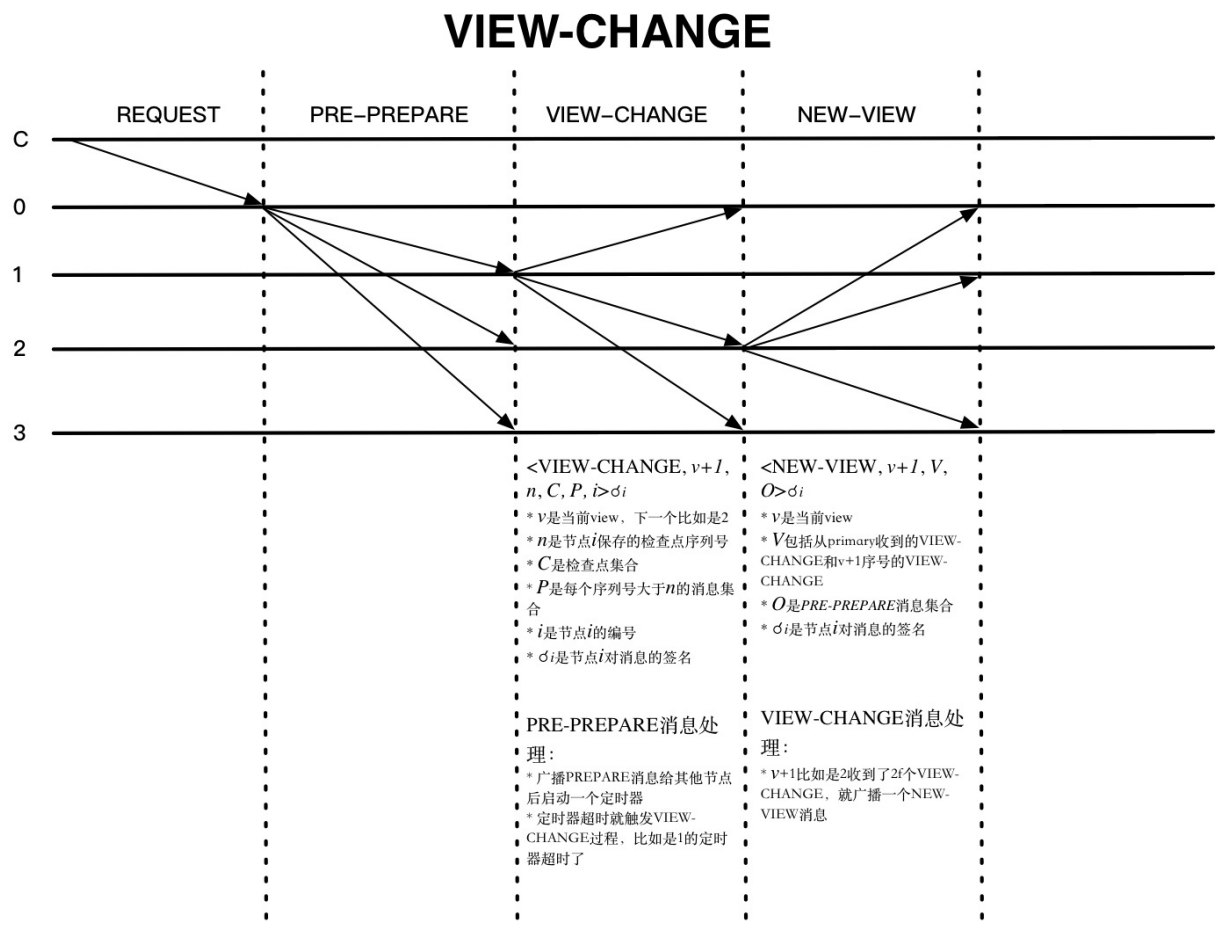
```
<View-Change-ACK,v+1,i,j,d>
```

其中 i 是发送者的标识，d 是 acknowledged VIEW-CHANGE message 的 digest，j 是发送 VIEW-CHANGE 消息的节点。

当新的主节点收到 2f 个其它节点的 view-change 消息，则证明有足够多人的节点认为主节点有问题，于是就会向其它节点广播 New-view 消息

对于主节点，发送 new-view 消息后会继续执行上个视图未处理完的请求，从 pre-prepare 阶段开始。其它节点验证 new-view 消息通过后，就会处理主节点发来的 pre-prepare 消息，这时执行的过程就是前面描述的 pbft 过程。到这时，正式进入 v+1 （视图编号加 1）的时代了。


开始
如果从节点认为主节点挂掉（超时无响应）或者有问题
v：上一个视图的编号
n：i节点的stable checkpoint的编号
C：2f+1个节点的有效checkpoint信息的集合
P：i节点中上一个view中编号大于n并且到达prepared状态的请求消息的集合
i：节点的编号
从节点向其他节点广播view-change事件<view-change,v+1,n,C,P,i>
是否收到2f个不同副本的视图编号为v+1的view-change消息（不包括自己）
否
结束
是
v：上一个视图的编号
V：新的主节点接收到的有效的视图编号为v+1的view-change消息集合
O：pre-prepare消息的集合
向其他节点广播new-view消息<new-view,v+1,V,O>
节点验证收到的视图编号为v+1的new-View消息里的O集合是否正确
否
结束
是
按照pbft流程逐条执行O集合里的请求消息，视图编号为v+1（每个请求执行后不会重新返回消息给客户端）
结束



上图是发生 VIEW-CHANGE 的一种情况，就是节点正常收到 PRE-PREPARE 消息以后都会启动一个定时器，如果在设置的时间内都没有收到回复，就会触发 VIEW-CHANGE，该节点就不会再接收除 CHECKPOINT 、VIEW-CHANGE 和 NEW-VIEW 等消息外的其他消息了。NEW-VIEW 是由新一轮的 primary 节点发送的，O 是不包含捎带的 REQUEST 的 PRE-PREPARE 消息集合，计算方法如下：

primary 节点确定 V 中最新的稳定检查点序列号 min-s 和 PRE-PREPARE 消息中最大的序列号 max-s 对 min-s 和 max-s 之间每个序列号 n 都生成一个 PRE-PREPARE 消息。这可能有两种情况：P 的 VIEW-CHANGE 消息中至少存在一个集合，序列号是 n 不存在上面的集合第一种情况，会生成新的 PRE-PREPARE 消息

```
<PRE-PREPARE, v+1, n, d>
```

高低水位 (处理垃圾回收)

- checkpoint 就是当前节点处理的最新请求序号。
- stable checkpoint 就是大部分节点（2f+1）已经共识完成的最大请求序号。比如系统有 4 个节点，三个节点都已经共识完了的请求编号是 213 ，那么这个 213 就是 stable checkpoint 了。

为了不保留过去的所有 preprepare, prepare 以及 commit 等 log 信息，所以我们每个一段时间设立一个检查点，这个检查点需要得到全网的共识。这个经过全网共识的检查点被称为 stable checkpoint。在 stable checkpoint 前面的所有 log 信息就可以删除。

stable checkpoint 最大的目的就是减少内存的占用。

对于每个共识结点每 K 个请求，就设立一个 checkpoint，但由于每个共识结点共识过程是异步的，所以可能结点 1 刚刚处理了请求 1，但结点 2 已经处理了请求 k，此时结点 2 设立 `checkpoint_1`，并广播出去（告诉大家我已经到 K 了，K 时候的状态是这样的）。此时，结点 2 也不会等其他结点（如结点 1）返回同样的 `checkpoint_1` 信息再继续处理请求。结点 2 一直向前处理交易，比如处理了 2K 个交易后，他设立另一个 `checkpoint_2` 并广播出去。

结点 2 是不能一直狂奔的，这涉及到高低水位。因为如果主节点是坏的，它在给请求编号时故意选择了一个很大的编号，以至于超出了序号的范围，所以我们需要设置一个低水位（low water mark）h 和高水位（high water mark）H，让主节点分配的编号在 h 和 H 之间，不能肆意分配。

```
低水位：h
高水位：H
间隔：L
即：H=h+L
```

如果我们的间隔设为 2 倍的 K，低水位为最新的 `stable checkpoint_0`，其为 0；高水位为 `0+2K=2K`；现在结点 2 已经狂奔到 2K 了，他不能向前了，所以他只能等待低水位 L 的前进，即他等到了 2f 个 `checkpoint_1` 的确认信息。此时，L 会先前移动为 `stable_checkpoint_1`，其为 K；高水位变成 3K，结点 2 可以继续前进了。

参考如下：

PBFT

共识算法系列之一：raft 和 pbft 算法

3.1.5 POW 算法

掷骰子游戏

矿工节点挖矿

监控网络新的区块

所有的节点时刻监听着传播到比特币网络的新区块。而这些新加入的区块对挖矿节点有着特殊的意义。矿工间的竞争以新区块的传播而结束，如同宣布谁是最后的赢家。对于矿工们来说，收到一个新区块进行验证意味着别人已经赢了，而自己则输了这场竞争。

节点同步区块后，去除交易池中的已成交交易，准备挖下一个块

接收并验证区块 277,315 后，节点会检查内存池中的全部交易，并移除已经在区块 277,315 中出现过的交易记录，确保任何留在内存池中的交易都是未确认的，等待被记录到新区块中。

收集交易，准备出块

收到新的交易后，验证交易，添加到自己的内存池（交易池），取出交易整合到一个候选区块中

创币交易

区块中的第一笔交易是笔特殊交易，称为创币交易或者 coinbase 交易。这个交易是出块节点 A 构造并用来奖励矿工节点 A 所做的贡献的。挖出区块获得的奖励金额是 coinbase 奖励（12.5 个全新的比特币）和区块中全部交易矿工费的总和。

与常规交易不同，创币交易没有输入，不消耗 UTXO。它只包含一个被称作 coinbase 的输入，仅仅用来创建新的比特币。创币交易有一个输出，支付到这个矿工的比特币地址。

创币交易需要 100 个区块确认高度，一般的需要 6 个区块确认高度

构建区块头

最后一个字段是 nonce，初始值为 0。

区块头完成全部的字段填充后，挖矿就可以开始进行了。挖矿的目标是找到一个使区块头哈希值小于难度目标的 nonce。挖矿节点通常需要尝试数十亿甚至数万亿个不同的 nonce 取值，直到找到一个满足条件的 nonce 值。

构建区块

比特币挖矿过程使用的是 SHA256 哈希函数。

用最简单的术语来说，挖矿就是重复计算区块头的哈希值，不断修改该参数，直到与哈希值匹配的一个过程。哈希函数的结果无法提前得知，也没有能得到一个特定哈希值的模式。哈希函数的这个特性意味着：得到哈希值的唯一方法是不断的尝试，每次随机修改输入，直到出现适当的哈希值。

POW 算法

比特币的共识算法不适合于私有链和联盟链。其原因首先是它是一个最终一致性共识算法，不是一个强一致性共识算法。第二个原因是其共识效率低。提供共识效率又会牺牲共识协议的安全性。

工作证明必须产生小于目标的哈希值。更高的目标意味着找到低于目标的哈希是不太困难的。较低的目标意味着在目标下方找到哈希更难。目标和难度是成反比。

3.1.6 寻找一种易于理解的一致性算法（扩展版）

摘要

Raft 是一种为了管理复制日志的一致性算法。它提供了和 Paxos 算法相同的功能和性能，但是它的算法结构和 Paxos 不同，使得 Raft 算法更加容易理解并且更容易构建实际的系统。为了提升可理解性，Raft 将一致性算法分解成了几个关键模块，例如领导人选举、日志复制和安全性。同时它通过实施一个更强的一致性来减少需要考虑的状态的数量。从一个用户研究的结果可以证明，对于学生而言，Raft 算法比 Paxos 算法更加容易学习。Raft 算法还包括一个新的机制来允许集群成员的动态改变，它利用重叠的大多数来保证安全性。

1 介绍

一致性算法允许一组机器像一个整体一样工作，即使其中一些机器出现故障也能够继续工作下去。正因为如此，一致性算法在构建可信赖的大规模软件系统中扮演着重要的角色。在过去的 10 年里，Paxos 算法统治着一致性算法这一领域：绝大多数的实现都是基于 Paxos 或者受其影响。同时 Paxos 也成为了教学领域里讲解一致性问题时的示例。

但是不幸的是，尽管有很多工作都在尝试降低它的复杂性，但是 Paxos 算法依然十分难以理解。并且，Paxos 自身的算法结构需要进行大幅的修改才能够应用到实际的系统中。这些都导致了工业界和学术界都对 Paxos 算法感到十分头疼。

和 Paxos 算法进行过努力之后，我们开始寻找一种新的一致性算法，可以为构建实际的系统和教学提供更好的基础。我们的做法是不寻常的，我们的首要目标是可理解性：我们是否可以在实际系统中定义一个一致性算法，并且能够比 Paxos 算法以一种更加容易的方式来学习。此外，我们希望该算法方便系统构建者的直觉的发展。不仅一个算法能够工作很重要，而且能够显而易见的知道为什么能工作也很重要。

Raft 一致性算法就是这些工作的结果。在设计 Raft 算法的时候，我们使用一些特别的技巧来提升它的可理解性，包括算法分解（Raft 主要被分成了领导人选举，日志复制和安全三个模块）和减少状态机的状态（相对于 Paxos，Raft 减少了非确定性和服务器互相处于非一致性的方式）。一份针对两所大学 43 个学生的研究表明 Raft 明显比 Paxos 算法更加容易理解。在这些学生同时学习了这两种算法之后，和 Paxos 比起来，其中 33 个学生能够回答有关于 Raft 的问题。

Raft 算法在许多方面和现有的一致性算法都很相似（主要是 Oki 和 Liskov 的 Viewstamped Replication），但是它也有一些独特的特性：

- **强领导者**：和其他一致性算法相比，Raft 使用一种更强的领导能力形式。比如，日志条目只从领导者发送给其他的服务器。这种方式简化了对复制日志的管理并且使得 Raft 算法更加易于理解。
- **领导选举**：Raft 算法使用一个随机计时器来选举领导者。这种方式只是在任何一致性算法都必须实现的心跳机制上增加了一点机制。在解决冲突的时候会更加简单快捷。
- **成员关系调整**：Raft 使用一种共同一致的方法来处理集群成员变换的问题，在这种方法下，处于调整过程中的两种不同的配置集群中大多数机器会有重叠，这就使得集群在成员变换的时候依然可以继续工作。

我们相信，Raft 算法不论出于教学目的还是作为实践项目的基础都是要比 Paxos 或者其他一致性算法要优异的。它比其他算法更加简单，更加容易理解；它的算法描述足以实现一个现实的系统；它有好多开源的实现并且在很多公司里使用；它的安全性已经被证明；它的效率和其他算法比起来也不相上下。

接下来，这篇论文会介绍以下内容：复制状态机问题（第 2 节），讨论 Paxos 的优点和缺点（第 3 节），讨论我们为了可理解性而采取的方法（第 4 节），阐述 Raft 一致性算法（第 5-8 节），评价 Raft 算法（第 9 节），以及一些相关的工作（第 10 节）。

2 复制状态机

一致性算法是从复制状态机的背景下提出的（参考英文原文引用 37）。在这种方法中，一组服务器上的状态机产生相同状态的副本，并且在一些机器宕掉的情况下也可以继续运行。复制状态机在分布式系统中被用于解决很多容错的问题。例如，大规模的系统中通常都有一个集群领导者，像 GFS、HDFS 和 RAMCloud，典型应用就是一个独立的的复制状态机去管理领导选举和存储配置信息并且在领导人宕机的情况下也要存活下来。比如 Chubby 和 ZooKeeper。

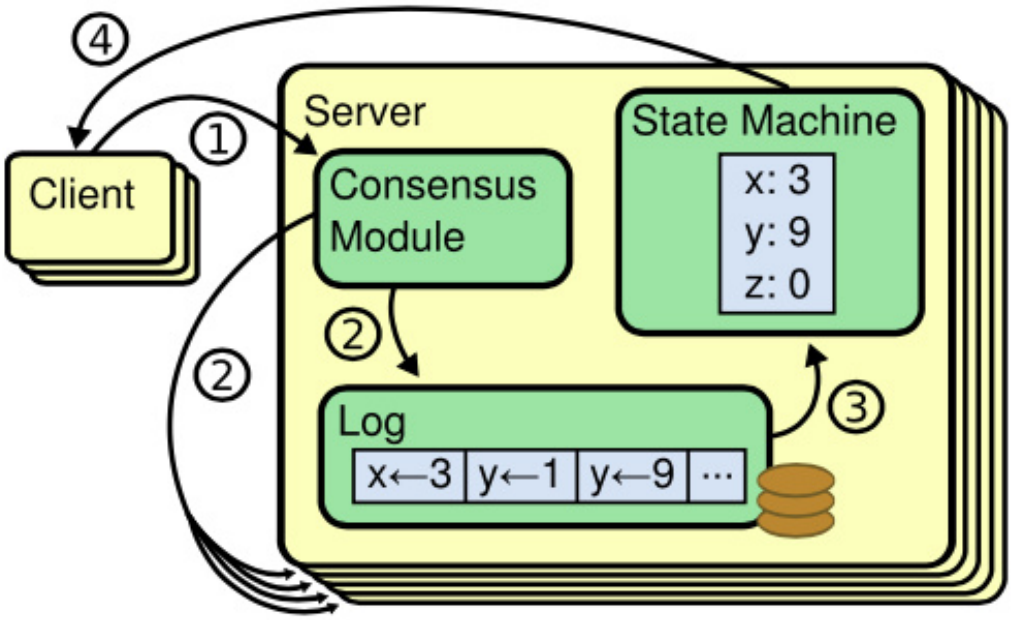


图 1： 复制状态机的结构。一致性算法管理着来自客户端指令的复制日志。状态机从日志中处理相同顺序的相同指令，所以产生的结果也是相同的。

复制状态机通常都是基于复制日志实现的，如图 1。每一个服务器存储一个包含一系列指令的日志，并且按照日志的顺序进行执行。每一个日志都按照相同的顺序包含相同的指令，所以每一个服务器都执行相同的指令序列。因为每个状态机都是确定的，每一次执行操作都产生相同的状态和同样的序列。

保证复制日志相同就是一致性算法的工作了。在一台服务器上，一致性模块接收客户端发送来的指令然后增加到自己的日志中去。它和其他服务器上的一致性模块进行通信来保证每一个服务器上的日志最终都以相同的顺序包含相同的请求，尽管有些服务器会宕机。一旦指令被正确的复制，每一个服务器的状态机按照日志顺序处理他们，然后输出结果被返回给客户端。因此，服务器集群看起来形成一个高可靠的状态机。

实际系统中使用的一致性算法通常含有以下特性：

- 安全性保证（绝对不会返回一个错误的结果）：在非拜占庭错误情况下，包括网络延迟、分区、丢包、冗余和乱序等错误都可以保证正确。
- 可用性：集群中只要有大多数的机器可运行并且能够相互通信、和客户端通信，就可以保证可用。因此，一个典型的包含 5 个节点的集群可以容忍两个节点的失败。服务器被停止就认为是失败。他们当有稳定的存储的时候可以从状态中恢复回来并重新加入集群。
- 不依赖时序来保证一致性：物理时钟错误或者极端的消息延迟只有在最坏情况下才会导致可用性问题。
- 通常情况下，一条指令可以尽可能快的在集群中大多数节点响应一轮远程过程调用时完成。小部分比较慢的节点不会影响系统整体的性能。

3 Paxos 算法的问题

在过去的 10 年里，Leslie Lamport 的 Paxos 算法几乎已经成为一致性的代名词：Paxos 是在课程教学中最经常使用的算法，同时也是大多数一致性算法实现的起点。Paxos 首先定义了一个能够达成单一决策一致的协议，比如单条的复制日志项。我们把这一子集叫做单决策 Paxos。然后通过组合多个 Paxos 协议的实例来促进一系列决策的达成。Paxos 保证安全性和活性，同时也支持集群成员关系的变更。Paxos 的正确性已经被证明，在通常情况下也很高效。

不幸的是，Paxos 有两个明显的缺点。第一个缺点是 Paxos 算法特别的难以理解。完整的解释是出了名的不透明；通过极大的努力之后，也只有少数人成功理解了这个算法。因此，有了几次用更简单的术语来解释 Paxos 的尝试。尽管这些解释都只关注了单决策的子集问题，但依然很具有挑战性。在 2012 年 NSDI 的会议中的一次调查显示，很少有人对 Paxos 算法感到满意，甚至在经验老道的研究者中也是如此。我们自己也尝试去理解 Paxos；我们一直没能理解 Paxos 直到我们读了很多对 Paxos 的简化解释并且设计了我们自己的算法之后，这一过程花了近一年时间。

我们假设 Paxos 的不透明性来自它选择单决策问题作为它的基础。单决策 Paxos 是晦涩微妙的，它被划分成了两种没有简单直观解释和无法独立理解的情景。因此，这导致了很难建立起直观的感受为什么单决策 Paxos 算法能够工作。构成多决策 Paxos 增加了很多错综复杂的规则。我们相信，在多决策上达成一致性的问题（一份日志而不是单一的日志记录）能够被分解成其他的方式并且更加直接和明显。

Paxos 算法的第二个问题就是它没有提供一个足够好的用来构建一个现实系统的基础。一个原因是还没有一种被广泛认同的多决策问题的算法。Lamport 的描述基本上都是关于单决策 Paxos 的；他简要描述了实施多决策 Paxos 的方法，但是缺乏很多细节。当然也有很多具体化 Paxos 的尝试，但是他们都互相不一样，和 Paxos 的概述也不同。例如 Chubby 这样的系统实现了一个类似于 Paxos 的算法，但是大多数的细节并没有被公开。

而且，Paxos 算法的结构也不是十分易于构建实践的系统；单决策分解也会产生其他的结果。例如，独立的选择一组日志条目然后合并成一个序列化的日志并没有带来太多的好处，仅仅增加了不少复杂性。围绕着日志来设计一个系统是更加简单高效的；新日志条目以严格限制的顺序增添到日志中去。另一个问题是，Paxos 使用了一种对等的点对点的方式作为它的核心（尽管它最终提议了一种弱领导人的方法来优化性能）。在只有一个决策会被制定的简化世界中是很有意义的，但是很少有现实的系统使用这种方式。如果有一系列的决策需要被制定，首先选择一个领导人，然后让他去协调所有的决议，会更加简单快速。

因此，实际的系统中很少有和 Paxos 相似的实践。每一种实现都是从 Paxos 开始研究，然后发现很多实现上的难题，再然后开发了一种和 Paxos 明显不一样的结构。这样是非常费时和容易出错的，并且理解 Paxos 的难度使得这个问题更加糟糕。Paxos 算法在理论上被证明是正确可行的，但是现实的系统和 Paxos 差别是如此的大，以至于这些证明没有什么太大的价值。下面来自 Chubby 实现非常典型：

> 在 Paxos 算法描述和实现现实系统中间有着巨大的鸿沟。最终的系统建立在一种没有经过证明的算法之上。

由于以上问题，我们认为 Paxos 算法既没有提供一个良好的基础给实践的系统，也没有给教学很好的帮助。基于一致性问题在大规模软件系统中的重要性，我们决定看看我们是否可以设计一个拥有更好特性的替代 Paxos 的一致性算法。Raft 算法就是这次实验的结果。

4 为了可理解性的设计

设计 Raft 算法我们有几个初衷：它必须提供一个完整的实际的系统实现基础，这样才能大大减少开发者的工作；它必须在任何情况下都是安全的并且在大多数的情况下都是可用的；并且它的大部分操作必须是高效的。但是我们最重要也是最大的挑战是可理解性。它必须保证对于普遍的人群都可以十分容易的去理解。另外，它必须能够让人形成直观的认识，这样系统的构建者才能够在现实中进行必然的扩展。

在设计 Raft 算法的时候，有很多的点需要我们在各种备选方案中进行选择。在这种情况下，我们评估备选方案基于可理解性原则：解释各个备选方案有多大的难度（例如，Raft 的状态空间有多复杂，是否有微妙的暗示）？对于一个读者而言，完全理解这个方案和暗示是否容易？

我们意识到对这种可理解性分析上具有高度的主观性；尽管如此，我们使用了两种通常适用的技术来解决这个问题。第一个技术就是众所周知的问题分解：只要有可能，我们就将问题分解成几个相对独立的，可被解

决的、可解释的和可理解的子问题。例如，Raft 算法被我们分成领导人选举，日志复制，安全性和角色改变几个部分。

我们使用的第二个方法是通过减少状态的数量来简化需要考虑的状态空间，使得系统更加连贯并且在可能的时候消除不确定性。特别的，所有的日志是不允许有空洞的，并且 Raft 限制了日志之间变成不一致状态的可能。尽管在大多数情况下我们都试图去消除不确定性，但是也有一些情况下不确定性可以提升可理解性。尤其是，随机化方法增加了不确定性，但是他们有利于减少状态空间数量，通过处理所有可能选择时使用相似的方法。我们使用随机化去简化 Raft 中领导人选举算法。

5 Raft 一致性算法

Raft 是一种用来管理章节 2 中描述的复制日志的算法。图 2 为了参考之用，总结这个算法的简略版本，图 3 列举了这个算法的一些关键特性。图中的这些元素会在剩下的章节逐一介绍。

Raft 通过选举一个高贵的领导人，然后给予他全部的管理复制日志的责任来实现一致性。领导人从客户端接收日志条目，把日志条目复制到其他服务器上，并且当保证安全性的时候告诉其他的服务器应用日志条目到他们的状态机中。拥有一个领导人大大简化了对复制日志的管理。例如，领导人可以决定新的日志条目需要放在日志中的什么位置而不需要和其他服务器商议，并且数据都从领导人流向其他服务器。一个领导人可以宕机，可以和其他服务器失去连接，这时一个新的领导人会被选举出来。

通过领导人的方式，Raft 将一致性问题分解成了三个相对独立的子问题，这些问题会在接下来的子章节中进行讨论：

- **领导选举**：一个新的领导人需要被选举出来，当现存的领导人宕机的时候（章节 5.2）
- **日志复制**：领导人必须从客户端接收日志然后复制到集群中的其他节点，并且强制要求其他节点的日志保持和自己相同。
- **安全性**：在 Raft 中安全性的关键是在图 3 中展示的状态机安全：如果有任何的服务器节点已经应用了一个确定的日志条目到它的状态机中，那么其他服务器节点不能在同一个日志索引位置应用一个不同的指令。章节 5.4 阐述了 Raft 算法是如何保证这个特性的；这个解决方案涉及到一个额外的选举机制（5.2 节）上的限制。

在展示一致性算法之后，这一章节会讨论可用性的一些问题和计时在系统的作用。

状态：

附加日志 RPC：

由领导人负责调用来复制日志指令；也会用作 heartbeat

接收者实现：

1. 如果 `term < currentTerm` 就返回 false （5.1 节）
2. 如果日志在 prevLogIndex 位置处的日志条目的任期号和 prevLogTerm 不匹配，则返回 false （5.3 节）
3. 如果已经存在的日志条目和新的产生冲突（索引值相同但是任期号不同），删除这一条和之后所有的（5.3 节）
4. 附加日志中尚未存在的任何新条目
5. 如果 `leaderCommit > commitIndex`，令 commitIndex 等于 leaderCommit 和新日志条目索引值中较小的一个

请求投票 RPC：

由候选人负责调用用来征集选票（5.2 节）

接收者实现：

1. 如果 `term < currentTerm` 返回 false （5.2 节）

2. 如果 votedFor 为空或者为 candidateId，并且候选人的日志至少和自己一样新，那么就投票给他（5.2 节，5.4 节）

所有服务器需遵守的规则：

所有服务器：

- 如果 `commitIndex > lastApplied`，那么就 lastApplied 加一，并把 `log[lastApplied]` 应用到状态机中（5.3 节）
- 如果接收到的 RPC 请求或响应中，任期号 `T > currentTerm`，那么就令 currentTerm 等于 T，并切换状态为跟随者（5.1 节）

跟随者（5.2 节）：

- 响应来自候选人和领导者的请求
- 如果在超过选举超时时间的情况之前都没有收到领导人的心跳，或者是候选人请求投票的，就自己变成候选人

候选人（5.2 节）：

- 在转变成候选人后就立即开始选举过程
  - 自增当前的任期号（currentTerm）
  - 给自己投票
  - 重置选举超时计时器
  - 发送请求投票的 RPC 给其他所有服务器
- 如果接收到大多数服务器的选票，那么就变成领导人
- 如果接收到来自新的领导人的附加日志 RPC，转变成跟随者
- 如果选举过程超时，再次发起一轮选举

领导人：

- 一旦成为领导人：发送空的附加日志 RPC（心跳）给其他所有的服务器；在一定的空余时间之后不停的重复发送，以阻止跟随者超时（5.2 节）
- 如果接收到来自客户端的请求：附加条目到本地日志中，在条目被应用到状态机后响应客户端（5.3 节）
- 如果对于一个跟随者，最后日志条目的索引值大于等于 nextIndex，那么：发送从 nextIndex 开始的所有日志条目：
  - 如果成功：更新相应跟随者的 nextIndex 和 matchIndex
  - 如果因为日志不一致而失败，减少 nextIndex 重试
- 如果存在一个满足 `N > commitIndex` 的 N，并且大多数的 `matchIndex[i] ≥ N` 成立，并且 `log[N].term == currentTerm` 成立，那么令 commitIndex 等于这个 N （5.3 和 5.4 节）

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

图 2：一个关于 Raft 一致性算法的浓缩总结（不包括成员变换和日志压缩）。

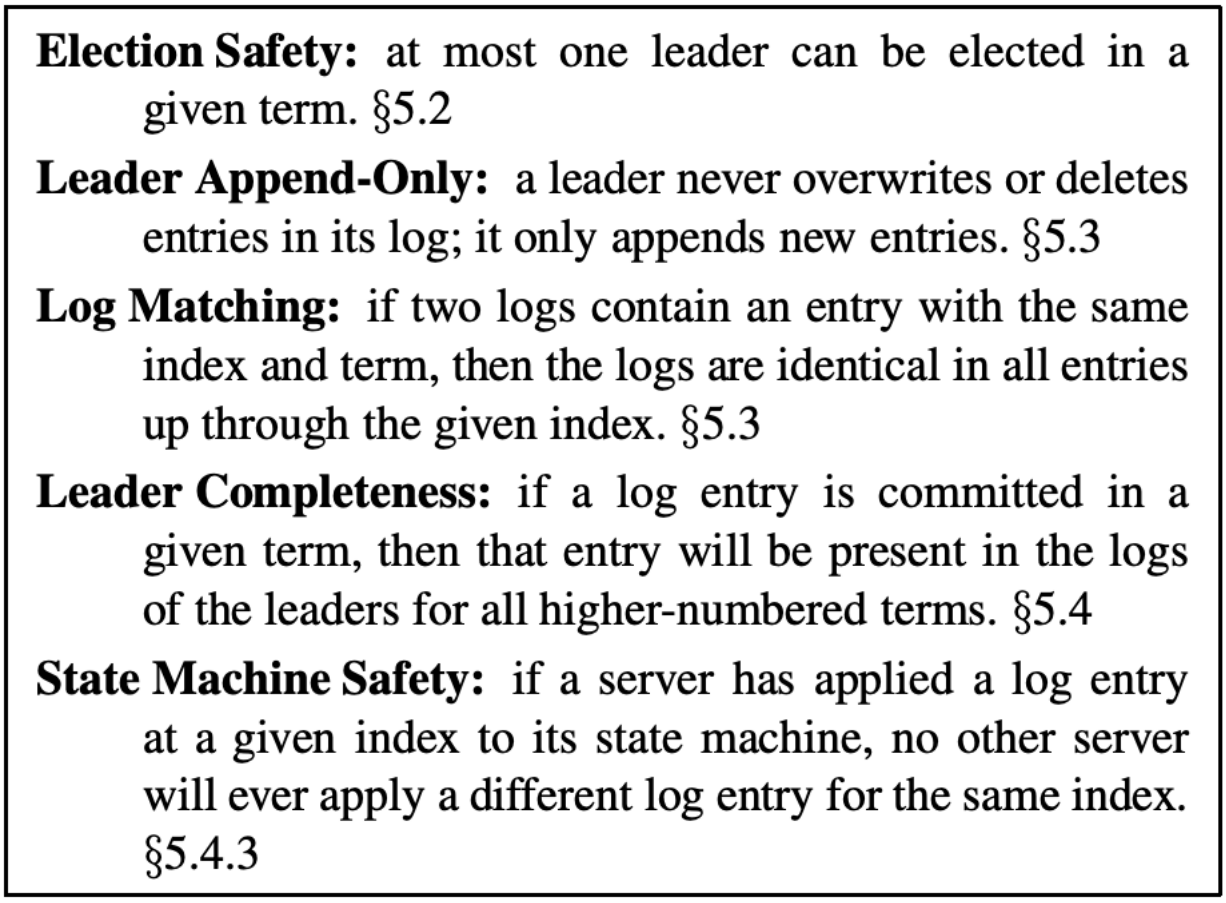


图 3：Raft 在任何时候都保证以上的各个特性。

5.1 Raft 基础

一个 Raft 集群包含若干个服务器节点；通常是 5 个，这允许整个系统容忍 2 个节点的失效。在任何时刻，每一个服务器节点都处于这三个状态之一：领导人、跟随者或者候选人。在通常情况下，系统中只有一个领导人并且其他的节点全部都是跟随者。跟随者都是被动的：他们不会发送任何请求，只是简单的响应来自领导者或者候选人的请求。领导人处理所有的客户端请求（如果一个客户端和跟随者联系，那么跟随者会把请求重定向给领导人）。第三种状态，候选人，是用来在 5.2 节描述的选举新领导人时使用。图 4 展示了这些状态和他们之间的转换关系；这些转换关系会在接下来进行讨论。

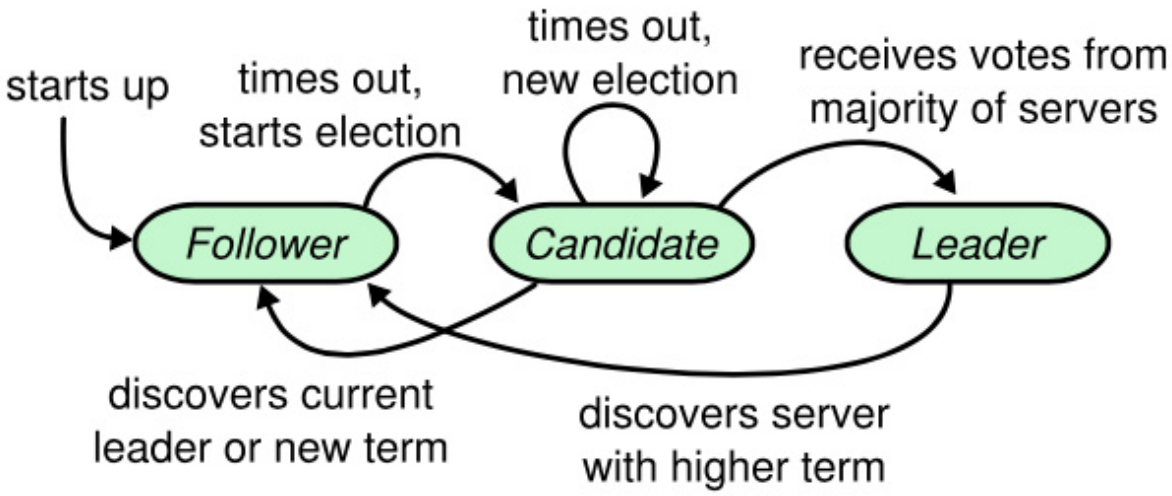


图 4：服务器状态。跟随者只响应来自其他服务器的请求。如果跟随者接收不到消息，那么他就会变成候选人并发起一次选举。获得集群中大多数选票的候选人将成为领导者。在一个任期内，领导人一直都会是领导人直到自己宕机了。

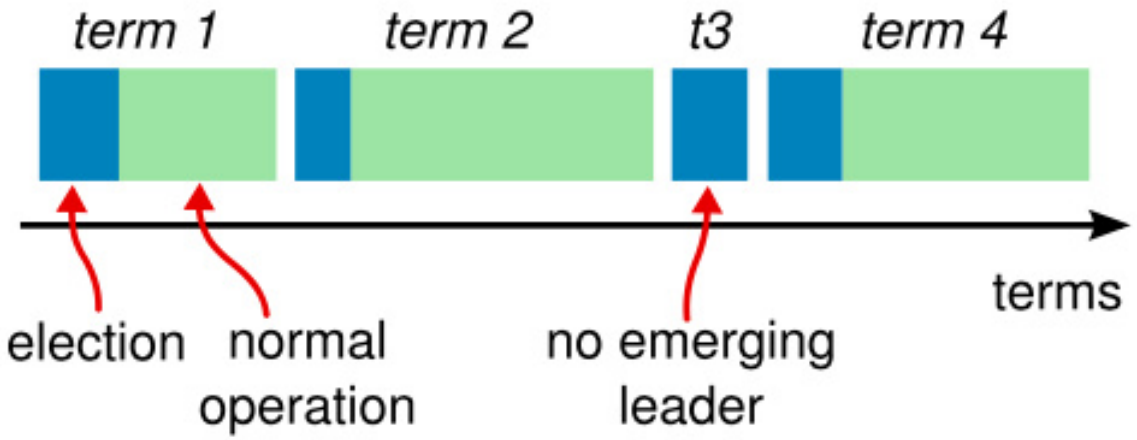


图 5：时间被划分成一个个的任期，每个任期开始都是一次选举。在选举成功后，领导人会管理整个集群直到任期结束。有时候选举会失败，那么这个任期就会没有领导人而结束。任期之间的切换可以在不同的时间不同的服务器上观察到。

Raft 把时间分割成任意长度的**任期**，如图 5。任期用连续的整数标记。每一段任期从一次**选举**开始，就像章节 5.2 描述的一样，一个或者多个候选人尝试成为领导者。如果一个候选人赢得选举，然后他就在接下来的任期内充当领导人的职责。在某些情况下，一次选举过程会造成选票的瓜分。在这种情况下，这一任期会以没有领导人结束；一个新的任期（和一次新的选举）会很快重新开始。Raft 保证了在一个给定的任期内，最多只有一个领导者。

不同的服务器节点可能多次观察到任期之间的转换，但在某些情况下，一个节点也可能观察不到任何一次选举或者整个任期全程。任期在 Raft 算法中充当逻辑时钟的作用，这会允许服务器节点查明一些过期的信息比如陈旧的领导者。每一个节点存储一个当前任期号，这一编号在整个时期内单调的增长。当服务器之间通信的时候会交换当前任期号；如果一个服务器的当前任期号比其他人小，那么他会更新自己的编号到较大的编号值。如果一个候选人或者领导者发现自己的任期号过期了，那么他会立即恢复成跟随者状态。如果一个节点接收到一个包含过期的任期号的请求，那么他会直接拒绝这个请求。

Raft 算法中服务器节点之间通信使用远程过程调用（RPCs），并且基本的一致性算法只需要两种类型的 RPCs。

请求投票（RequestVote）RPCs 由候选人在选举期间发起（章节 5.2），然后附加条目（AppendEntries）RPCs 由领导人发起，用来复制日志和提供一种心跳机制（章节 5.3）。第 7 节为了在服务器之间传输快照增加了第三种 RPC。当服务器没有及时的收到 RPC 的响应时，会进行重试，并且他们能够并行的发起 RPCs 来获得最佳的性能。

5.2 领导人选举

Raft 使用一种心跳机制来触发领导人选举。当服务器程序启动时，他们都是跟随者身份。一个服务器节点继续保持着跟随者状态只要他从领导人或者候选者处接收到有效的 RPCs。领导者周期性的向所有跟随者发送心跳包（即不包含日志项内容的附加日志项 RPCs）来维持自己的权威。如果一个跟随者在一段时间里没有接收到任何消息，也就是**选举超时**，那么他就会认为系统中没有可用的领导者, 并且发起选举以选出新的领导者。

要开始一次选举过程，跟随者先要增加自己的当前任期号并且转换到候选人状态。然后他会并行的向集群中的其他服务器节点发送请求投票的 RPCs 来给自己投票。候选人会继续保持着当前状态直到以下三件事情之一发生：(a) 他自己赢得了这次的选举，(b) 其他的服务器成为领导者，(c) 一段时间之后没有任何一个获胜的人。这些结果会分别的在下面的段落里进行讨论。

当一个候选人从整个集群的大多数服务器节点获得了针对同一个任期号的选票，那么他就赢得了这次选举并成为领导人。每一个服务器最多会对一个任期号投出一张选票，按照先来先服务的原则（注意：5.4 节在投票上增加了一点额外的限制）。要求大多数选票的规则确保了最多只会有一个候选人赢得此次选举（图 3 中的选举安全性）。一旦候选人赢得选举，他就立即成为领导人。然后他会向其他的服务器发送心跳消息来建立自己的权威并且阻止新的领导人的产生。

在等待投票的时候，候选人可能会从其他的服务器接收到声明它是领导人的附加日志项 RPC。如果这个领导人的任期号（包含在此次的 RPC 中）不小于候选人当前的任期号，那么候选人会承认领导人合法并回到跟随者状态。如果此次 RPC 中的任期号比自己小，那么候选人就会拒绝这次的 RPC 并且继续保持候选人状态。

第三种可能的结果是候选人既没有赢得选举也没有输：如果有多个跟随者同时成为候选人，那么选票可能会被瓜分以至于没有候选人可以赢得大多数人的支持。当这种情况发生的时候，每一个候选人都会超时，然后通过增加当前任期号来开始一轮新的选举。然而，没有其他机制的话，选票可能会被无限的重复瓜分。

Raft 算法使用随机选举超时时间的方法来确保很少会发生选票瓜分的情况，就算发生也能很快的解决。为了阻止选票起初就被瓜分，选举超时时间是从一个固定的区间（例如 150-300 毫秒）随机选择。这样可以把服务器都分散开以至于在大多数情况下只有一个服务器会选举超时；然后他赢得选举并在其他服务器超时之前发送心跳包。同样的机制被用在选票瓜分的情况下。每一个候选人在开始一次选举的时候会重置一个随机的选举超时时间，然后在超时时间内等待投票的结果；这样减少了在新的选举中另外的选票瓜分的可能性。9.3 节展示了这种方案能够快速的选出一个领导人。

领导人选举这个例子，体现了可理解性原则是如何指导我们进行方案设计的。起初我们计划使用一种排名系统：每一个候选人都被赋予一个唯一的排名，供候选人之间竞争时进行选择。如果一个候选人发现另一个候选人拥有更高的排名，那么他就会回到跟随者状态，这样高排名的候选人能够更加容易的赢得下一次选举。但是我们发现这种方法在可用性方面会有一点问题（如果高排名的服务器宕机了，那么低排名的服务器可能会超时并再次进入候选人状态。而且如果这个行为发生得足够快，则可能会导致整个选举过程都被重置掉）。我们针对算法进行了多次调整，但是每次调整之后都会有新的问题。最终我们认为随机重试的方法是更加明显和易于理解的。

5.3 日志复制

一旦一个领导人被选举出来，他就开始为客户端提供服务。客户端的每一个请求都包含一条被复制状态机执行的指令。领导人把这条指令作为一条新的日志条目附加到日志中去，然后并行的发起附加条目 RPCs 给其他的服务器，让他们复制这条日志条目。当这条日志条目被安全的复制（下面会介绍），领导人会应用这条日志条目到它的状态机中然后把执行的结果返回给客户端。如果跟随者崩溃或者运行缓慢，再或者网络丢包，领导人会不断的重复尝试附加日志条目 RPCs （尽管已经回复了客户端）直到所有的跟随者都最终存储了所有的日志条目。

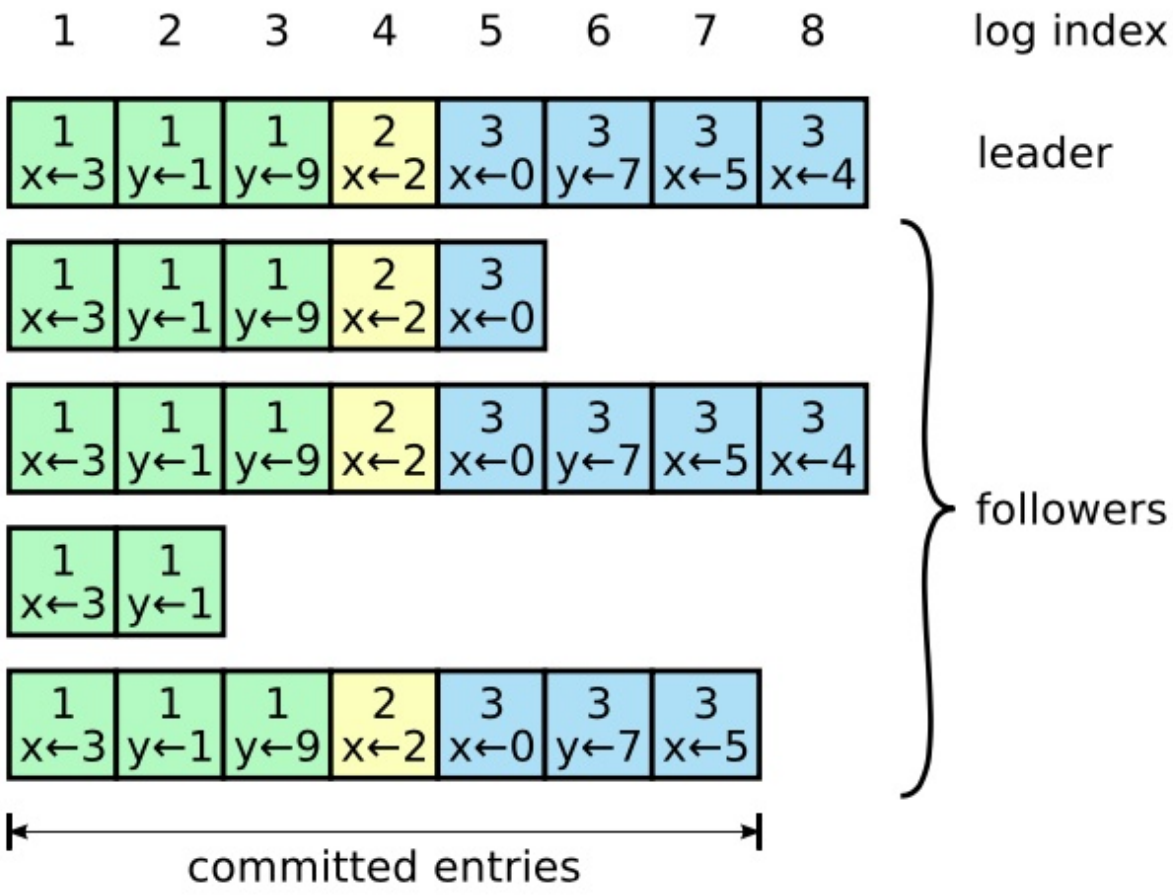


图 6：日志由有序序号标记的条目组成。每个条目都包含创建时的任期号（图中框中的数字），和一个状态机需要执行的指令。一个条目当可以安全的被应用到状态机中去的时候，就认为是可以提交了。

日志以图 6 展示的方式组织。每一个日志条目存储一条状态机指令和从领导人收到这条指令时的任期号。日志中的任期号用来检查是否出现不一致的情况，同时也用来保证图 3 中的某些性质。每一条日志条目同时也都有一个整数索引值来表明它在日志中的位置。

领导人来决定什么时候把日志条目应用到状态机中是安全的；这种日志条目被称为**已提交**。Raft 算法保证所有已提交的日志条目都是持久化的并且最终会被所有可用的状态机执行。在领导人将创建的日志条目复制到大多数的服务器上的时候，日志条目就会被提交（例如在图 6 中的条目 7）。同时，领导人的日志中之前的所有日志条目也都会被提交，包括由其他领导人创建的条目。5.4 节会讨论某些当在领导人改变之后应用这条规则的隐晦内容，同时他也展示了这种提交的定义是安全的。领导人跟踪了最大的将会被提交的日志项的索引，并且索引值会被包含在未来的所有附加日志 RPCs（包括心跳包），这样其他的服务器才能最终知道领导人的提交位置。一旦跟随者知道一条日志条目已经被提交，那么他也会将这个日志条目应用到本地的状态机中（按照日志的顺序）。

我们设计了 Raft 的日志机制来维护一个不同服务器的日志之间的高层次的一致性。这么做不仅简化了系统的行为也使得更加可预计，同时他也是安全性保证的一个重要组件。Raft 维护着以下的特性，这些同时也组成了图 3 中的日志匹配特性：

- 如果在不同的日志中的两个条目拥有相同的索引和任期号，那么他们存储了相同的指令。
- 如果在不同的日志中的两个条目拥有相同的索引和任期号，那么他们之前的所有日志条目也全部相同。

第一个特性来自这样的一个事实，领导人最多在一个任期里在指定的一个日志索引位置创建一条日志条目，同时日志条目在日志中的位置也从来不会改变。第二个特性由附加日志 RPC 的一个简单的一致性检查所保证。在发送附加日志 RPC 的时候，领导人会把新的日志条目紧接着之前的条目的索引位置和任期号包含在里面。如果跟随者在它的日志中找不到包含相同索引位置和任期号的条目，那么他就会拒绝接收新的日志条目。一致性检查就像一个归纳步骤：一开始空的日志状态肯定是满足日志匹配特性的，然后一致性检查保护了日志匹配特性当日志扩展的时候。因此，每当附加日志 RPC 返回成功时，领导人就知道跟随者的日志一定是和自己相同的了。

在正常的操作中，领导人和跟随者的日志保持一致性，所以附加日志 RPC 的一致性检查从来不会失败。然而，领导人崩溃的情况会使得日志处于不一致的状态（老的领导人可能还没有完全复制所有的日志条目）。这种不一致问题会在领导人和跟随者的一系列崩溃下加剧。图 7 展示了跟随者的日志可能和新的领导人不同的方式。跟随者可能会丢失一些在新的领导人中有的日志条目，他也可能拥有一些领导人没有的日志条目，或者两者都发生。丢失或者多出日志条目可能会持续多个任期。

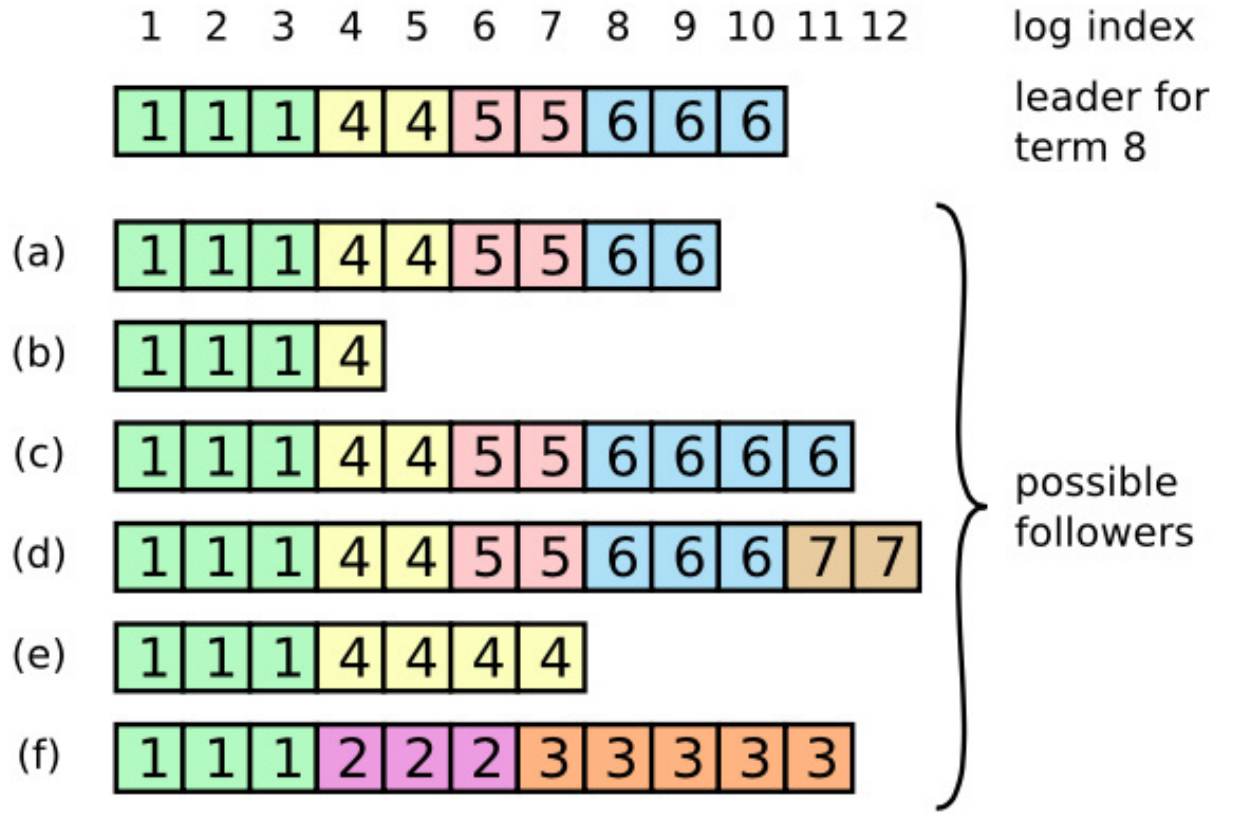


图 7：当一个领导人成功当选时，跟随者可能是任何情况（a-f）。每一个盒子表示是一个日志条目；里面的数字表示任期号。跟随者可能会缺少一些日志条目（a-b），可能会有一些未被提交的日志条目（c-d），或者两种情况都存在（e-f）。例如，场景 f 可能会这样发生，某服务器在任期 2 的时候是领导人，已附加了一些日志条目到自己的日志中，但在提交之前就崩溃了；很快这个机器就被重启了，在任期 3 重新被选为领导人，并且又增加了一些日志条目到自己的日志中；在任期 2 和任期 3 的日志被提交之前，这个服务器又宕机了，并且在接下来的几个任期里一直处于宕机状态。

在 Raft 算法中，领导人处理不一致是通过强制跟随者直接复制自己的日志来解决了。这意味着在跟随者中的冲突的日志条目会被领导人的日志覆盖。5.4 节会阐述如何通过增加一些限制来使得这样的操作是安全的。

要使得跟随者的日志进入和自己一致的状态，领导人必须找到最后两者达成一致的地方，然后删除从那个点之后的所有日志条目，发送自己的日志给跟随者。所有的这些操作都在进行附加日志 RPCs 的一致性检查时完成。领导人针对每一个跟随者维护了一个 **nextIndex**，这表示下一个需要发送给跟随者的日志条目的索引地址。当一个领导人刚获得权力的时候，他初始化所有的 nextIndex 值为自己的最后一条日志的 index 加 1（图 7 中的 11）。如果一个跟随者的日志和领导人不一致，那么在下一次的附加日志 RPC 时的一致性检查就会失败。在被跟随者拒绝之后，领导人就会减小 nextIndex 值并进行重试。最终 nextIndex 会在某个位置使得领导人和跟随者的日志达成一致。当这种情况发生，附加日志 RPC 就会成功，这时就会把跟随者冲突的日志条目全部删除并且加上领导人的日志。一旦附加日志 RPC 成功，那么跟随者的日志就会和领导人保持一致，并且在接下来的任期里一直继续保持。

如果需要的话，算法可以通过减少被拒绝的附加日志 RPCs 的次数来优化。例如，当附加日志 RPC 的请求被拒绝的时候，跟随者可以包含冲突的条目的任期号和自己存储的那个任期的最早的索引地址。借助这些信息，领导人可以减小 nextIndex 越过所有那个任期冲突的所有日志条目；这样就变成每个任期需要一次附加条目 RPC 而不是每个条目一次。在实践中，我们十分怀疑这种优化是否是必要的，因为失败是很少发生的并且也不大可能会有这么多不一致的日志。

通过这种机制，领导人在获得权力的时候就不需要任何特殊的操作来恢复一致性。他只需要进行正常的操作，然后日志就能自动的在回复附加日志 RPC 的一致性检查失败的时候自动趋于一致。领导人从来不会覆盖或者删除自己的日志（图 3 的领导人只附加特性）。

日志复制机制展示出了第 2 节中形容的一致性特性：Raft 能够接受，复制并应用新的日志条目只要大部分的机器是工作的；在通常的情况下，新的日志条目可以在一次 RPC 中被复制给集群中的大多数机器；并且单个的缓慢的跟随者不会影响整体的性能。

5.4 安全性

前面的章节里描述了 Raft 算法是如何选举和复制日志的。然而，到目前为止描述的机制并不能充分的保证每一个状态机会按照相同的顺序执行相同的指令。例如，一个跟随者可能会进入不可用状态同时领导人已经提交了若干的日志条目，然后这个跟随者可能会被选举为领导人并且覆盖这些日志条目；因此，不同的状态机可能会执行不同的指令序列。

这一节通过在领导选举的时候增加一些限制来完善 Raft 算法。这一限制保证了任何的领导人对于给定的任期号，都拥有了之前任期的所有被提交的日志条目（图 3 中的领导人完整特性）。增加这一选举时的限制，我们对于提交时的规则也更加清晰。最终，我们将展示对于领导人完整特性的简要证明，并且说明领导人是如何领导复制状态机的做出正确行为的。

5.4.1 选举限制

在任何基于领导人的一致性算法中，领导人都必须存储所有已经提交的日志条目。在某些一致性算法中，例如 Viewstamped Replication，某个节点即使是一开始并没有包含所有已经提交的日志条目，它也能被选为领导者。这些算法都包含一些额外的机制来识别丢失的日志条目并把他们传送给新的领导人，要么是在选举阶段要么在之后很快进行。不幸的是，这种方法会导致相当大的额外的机制和复杂性。Raft 使用了一种更加简单的方法，它可以保证所有之前的任期号中已经提交的日志条目在选举的时候都会出现在新的领导人中，不需要传送这些日志条目给领导人。这意味着日志条目的传送是单向的，只从领导人传给跟随者，并且领导人从不会覆盖自身本地日志中已经存在的条目。

Raft 使用投票的方式来阻止一个候选人赢得选举除非这个候选人包含了所有已经提交的日志条目。候选人为了赢得选举必须联系集群中的大部分节点，这意味着每一个已经提交的日志条目在这些服务器节点中肯定存在于至少一个节点上。如果候选人的日志至少和大多数的服务器节点一样新（这个新的定义会在下面讨论），那么他一定持有了所有已经提交的日志条目。请求投票 RPC 实现了这样的限制：RPC 中包含了候选人的日志信息，然后投票人会拒绝掉那些日志没有自己新的投票请求。

Raft 通过比较两份日志中最后一条日志条目的索引值和任期号定义谁的日志比较新。如果两份日志最后的条目的任期号不同，那么任期号大的日志更加新。如果两份日志最后的条目任期号相同，那么日志比较长的那个就更加新。

5.4.2 提交之前任期内的日志条目

如同 5.3 节介绍的那样，领导人知道一条当前任期内的日志记录是可以被提交的，只要它被存储到了大多数的服务器上。如果一个领导人在提交日志条目之前崩溃了，未来后续的领导人会继续尝试复制这条日志记录。然而，一个领导人不能断定一个之前任期里的日志条目被保存到大多数服务器上的时候就一定已经提交了。图 8 展示了一种情况，一条已经被存储到大多数节点上的老日志条目，也依然有可能会被未来的领导人覆盖掉。

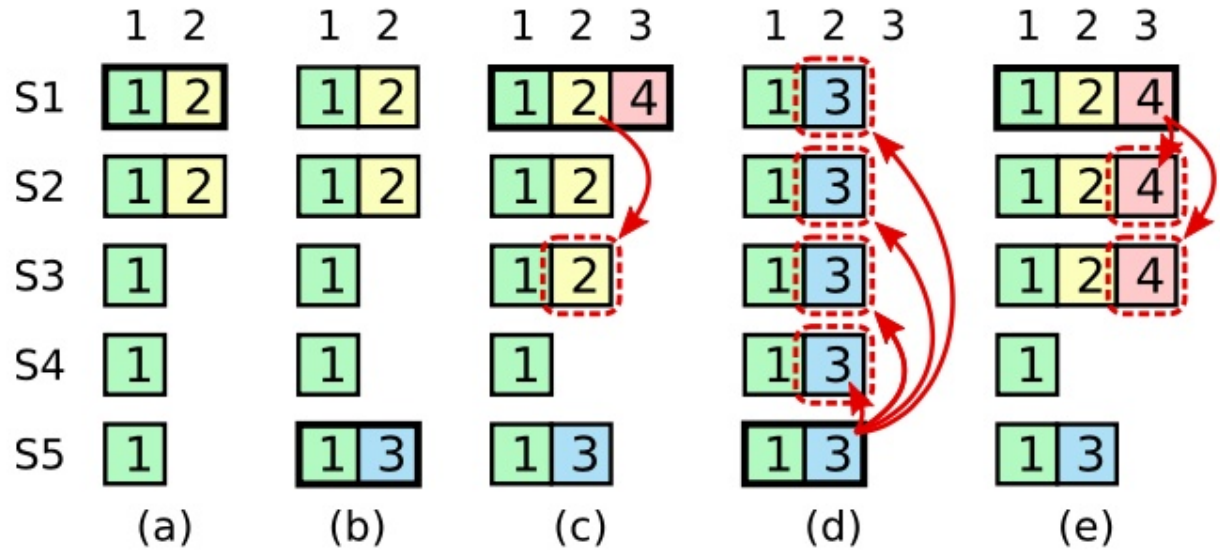


图 8：如图的时间序列展示了为什么领导人无法决定对老任期号的日志条目进行提交。在 (a) 中，S1 是领导者，部分的复制了索引位置 2 的日志条目。在 (b) 中，S1 崩溃了，然后 S5 在任期 3 里通过 S3、S4 和自己的选票赢得选举，然后从客户端接收了一条不一样的日志条目放在了索引 2 处。然后到 (c)，S5 又崩溃了；S1 重新启动，选举成功，开始复制日志。在这时，来自任期 2 的那条日志已经被复制到了集群中的大多数机器上，但是还没有被提交。如果 S1 在 (d) 中又崩溃了，S5 可以重新被选举成功（通过来自 S2，S3 和 S4 的选票），然后覆盖了他们在索引 2 处的日志。反之，如果在崩溃之前，S1 把自己主导的新任期里产生的日志条目复制到了大多数机器上，就如 (e) 中那样，那么在后面任期里面这些新的日志条目就会被提交（因为 S5 就不可能选举成功）。这样在同一时刻就同时保证了，之前的所有老的日志条目就会被提交。

为了消除图 8 里描述的情况，Raft 永远不会通过计算副本数目的方式去提交一个之前任期内的日志条目。只有领导人当前任期里的日志条目通过计算副本数目可以被提交；一旦当前任期的日志条目以这种方式被提交，那么由于日志匹配特性，之前的日志条目也都会被间接的提交。在某些情况下，领导人可以安全的知道一个老的日志条目是否已经被提交（例如，该条目是否存储到所有服务器上），但是 Raft 为了简化问题使用一种更加保守的方法。

当领导人复制之前任期里的日志时，Raft 会为所有日志保留原始的任期号，这在提交规则上产生了额外的复杂性。在其他的一致性算法中，如果一个新的领导人要重新复制之前的任期里的日志时，它必须使用当前新的任期号。Raft 使用的方法更加容易辨别出日志，因为它可以随着时间和日志的变化对日志维护着同一个任期编号。另外，和其他的算法相比，Raft 中的新领导人只需要发送更少日志条目（其他算法中必须在他们被提交之前发送更多的冗余日志条目来为他们重新编号）。

5.4.3 安全性论证

在给定了完整的 Raft 算法之后，我们现在可以更加精确的讨论领导人完整性特性（这一讨论基于 9.2 节的安全性证明）。我们假设领导人完全性特性是不存在的，然后我们推出矛盾来。假设任期 T 的领导人（领导人 T）在任期内提交了一条日志条目，但是这条日志条目没有被存储到未来某个任期的领导人的日志中。设大于 T 的最小任期 U 的领导人 U 没有这条日志条目。

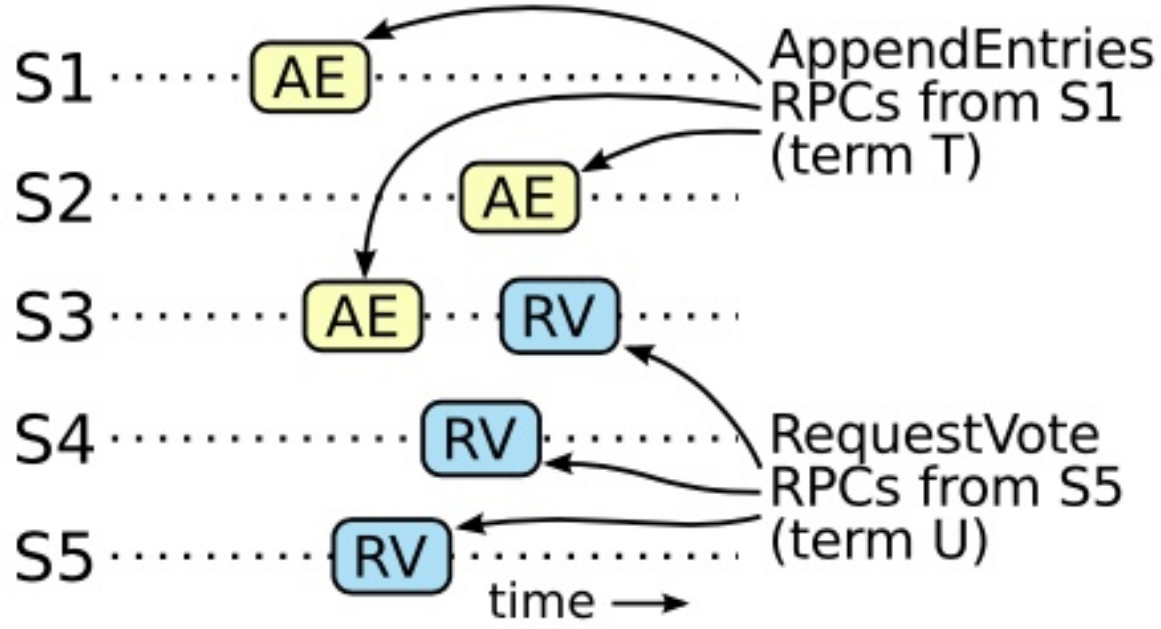


图 9：如果 S1 （任期 T 的领导者）提交了一条新的日志在它的任期里，然后 S5 在之后的任期 U 里被选举为领导人，然后至少会有一个机器，如 S3，既拥有来自 S1 的日志，也给 S5 投票了。

1. 在领导人 U 选举的时候一定没有那条被提交的日志条目（领导人从不会删除或者覆盖任何条目）。
2. 领导人 T 复制这条日志条目给集群中的大多数节点，同时，领导人 U 从集群中的大多数节点赢得了选票。因此，至少有一个节点（投票者、选民）同时接受了来自领导人 T 的日志条目，并且给领导人 U 投票了，如图 9。这个投票者是产生这个矛盾的关键。
3. 这个投票者必须在给领导人 U 投票之前先接受了从领导人 T 发来的已经被提交的日志条目；否则他就会拒绝来自领导人 T 的附加日志请求（因为此时他的任期号会比 T 大）。
4. 投票者在给领导人 U 投票时依然保存有这条日志条目，因为任何中间的领导人都包含该日志条目（根据上述的假设），领导人从不会删除条目，并且跟随者只有在和领导人冲突的时候才会删除条目。
5. 投票者把自己选票投给领导人 U 时，领导人 U 的日志必须和投票者自己一样新。这就导致了两者矛盾之一。
6. 首先，如果投票者和领导人 U 的最后一条日志的任期号相同，那么领导人 U 的日志至少和投票者一样长，所以领导人 U 的日志一定包含所有投票者的日志。这是另一处矛盾，因为投票者包含了那条已经被提交的日志条目，但是在上述的假设里，领导人 U 是不包含的。
7. 除此之外，领导人 U 的最后一条日志的任期号就必须比投票人大了。此外，他也比 T 大，因为投票人的最后一条日志的任期号至少和 T 一样大（他包含了来自任期 T 的已提交的日志）。创建了领导人 U 最后一条日志的之前领导人一定已经包含了那条被提交的日志（根据上述假设，领导人 U 是第一个不包含该日志条目的领导人）。所以，根据日志匹配特性，领导人 U 一定也包含那条被提交的日志，这里产生矛盾。
8. 这里完成了矛盾。因此，所有比 T 大的领导人一定包含了所有来自 T 的已经被提交的日志。

9. 日志匹配原则保证了未来的领导人也同时会包含被间接提交的条目，例如图 8 (d) 中的索引 2。

通过领导人完全特性，我们就能证明图 3 中的状态机安全特性，即如果服务器已经在某个给定的索引值应用了日志条目到自己的状态机里，那么其他的服务器不会应用一个不一样的日志到同一个索引值上。在一个服务器应用一条日志条目到他自己的状态机中时，他的日志必须和领导人的日志，在该条目和之前的条目上相同，并且已经被提交。现在我们来考虑在任何一个服务器应用一个指定索引位置的日志的最小任期；日志完全特性保证拥有更高任期号的领导人会存储相同的日志条目，所以之后的任期里应用某个索引位置的日志条目也会是相同的值。因此，状态机安全特性是成立的。

最后，Raft 要求服务器按照日志中索引位置顺序应用日志条目。和状态机安全特性结合起来看，这就意味着所有的服务器会应用相同的日志序列集到自己的状态机中，并且是按照相同的顺序。

5.5 跟随者和候选人崩溃

到目前为止，我们都只关注了领导人崩溃的情况。跟随者和候选人崩溃后的处理方式比领导人要简单的多，并且他们的处理方式是相同的。如果跟随者或者候选人崩溃了，那么后续发送给他们的 RPCs 都会失败。Raft 中处理这种失败就是简单的通过无限的重试；如果崩溃的机器重启了，那么这些 RPC 就会完整的成功。如果一个服务器在完成了一个 RPC，但是还没有响应的时候崩溃了，那么在他重新启动之后就会再次收到同样的请求。Raft 的 RPCs 都是幂等的，所以这样重试不会造成任何问题。例如一个跟随者如果收到附加日志请求但是他已经包含了这一日志，那么他就会直接忽略这个新的请求。

5.6 时间和可用性

Raft 的要求之一就是安全性不能依赖时间：整个系统不能因为某些事件运行的比预期快一点或者慢一点就产生了错误的结果。但是，可用性（系统可以及时的响应客户端）不可避免的要依赖于时间。例如，如果消息交换比服务器故障间隔时间长，候选人将没有足够长的时间来赢得选举；没有一个稳定的领导人，Raft 将无法工作。

领导人选举是 Raft 中对时间要求最为关键的方面。Raft 可以选举并维持一个稳定的领导人, 只要系统满足下面的时间要求：

广播时间（broadcastTime）« 选举超时时间（electionTimeout）« 平均故障间隔时间（MTBF）

在这个不等式中，广播时间指的是从一个服务器并行的发送 RPCs 给集群中的其他服务器并接收响应的平均时间；选举超时时间就是在 5.2 节中介绍的选举的超时时间限制；然后平均故障间隔时间就是对于一台服务器而言，两次故障之间的平均时间。广播时间必须比选举超时时间小一个量级，这样领导人才能够发送稳定的心跳消息来阻止跟随者开始进入选举状态；通过随机化选举超时时间的方法，这个不等式也使得选票瓜分的情况变得不可能。选举超时时间应该要比平均故障间隔时间小上几个数量级，这样整个系统才能稳定的运行。当领导人崩溃后，整个系统会大约相当于选举超时的时间里不可用；我们希望这种情况在整个系统的运行中很少出现。

广播时间和平均故障间隔时间是由系统决定的，但是选举超时时间是我们自己选择的。Raft 的 RPCs 需要接收方将信息持久化的保存到稳定存储中去，所以广播时间大约是 0.5 毫秒到 20 毫秒，取决于存储的技术。因此，选举超时时间可能需要在 10 毫秒到 500 毫秒之间。大多数的服务器的平均故障间隔时间都在几个月甚至更长，很容易满足时间的需求。

6 集群成员变化

到目前为止，我们都假设集群的配置（加入到一致性算法的服务器集合）是固定不变的。但是在实践中，偶尔是会改变集群的配置的，例如替换那些宕机的机器或者改变复制级别。尽管可以通过暂停整个集群，更新所有配置，然后重启整个集群的方式来实现，但是在更改的时候集群会不可用。另外，如果存在手工操作步骤，那么就会有操作失误的风险。为了避免这样的问题，我们决定自动化配置改变并且将其纳入到 Raft 一致性算法中来。

为了让配置修改机制能够安全，那么在转换的过程中不能够存在任何时间点使得两个领导人同时被选举成功在同一个任期里。不幸的是，任何服务器直接从旧的配置直接转换到新的配置的方案都是不安全的。一次性自动的转换所有服务器是不可能的，所以在转换期间整个集群存在划分成两个独立的大多数群体的可能性（见图 10）。

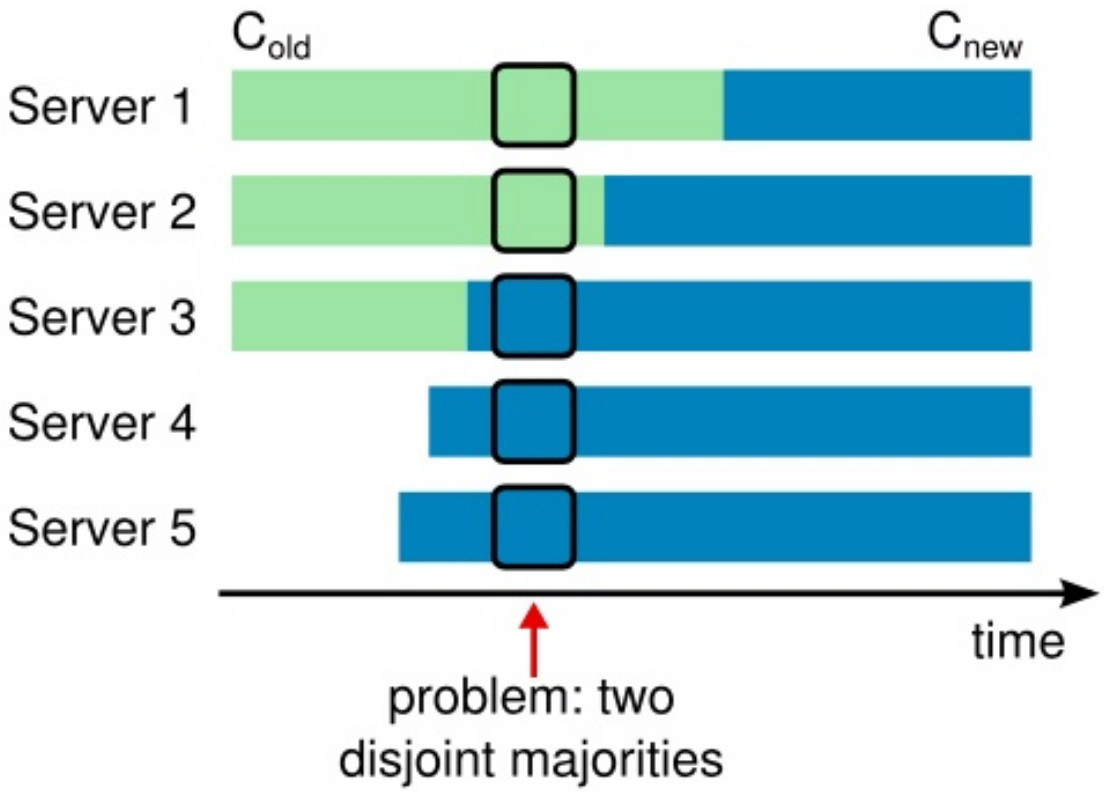


图 10：直接从一种配置转到新的配置是十分不安全的，因为各个机器可能在任何的时候进行转换。在这个例子中，集群配额从 3 台机器变成了 5 台。不幸的是，存在这样的一个时间点，两个不同的领导人在同一个任期里都可以被选举成功。一个是通过旧的配置，一个通过新的配置。

为了保证安全性，配置更改必须使用两阶段方法。目前有很多种两阶段的实现。例如，有些系统在第一阶段停掉旧的配置所以集群就不能处理客户端请求；然后在第二阶段在启用新的配置。在 Raft 中，集群先切换到一个过渡的配置，我们称之为共同一致；一旦共同一致已经被提交了，那么系统就切换到新的配置上。共同一致是老配置和新配置的结合：

- 日志条目被复制给集群中新、老配置的所有服务器。
- 新、旧配置的服务器都可以成为领导人。
- 达成一致（针对选举和提交）需要分别在两种配置上获得大多数的支持。

共同一致允许独立的服务器在不影响安全性的前提下，在不同的时间进行配置转换过程。此外，共同一致可以让集群在配置转换的过程人依然响应客户端的请求。

集群配置在复制日志中以特殊的日志条目来存储和通信；图 11 展示了配置转换的过程。当一个领导人接收到一个改变配置从 C-old 到 C-new 的请求，他会为了共同一致存储配置（图中的 C-old,new），以前面描述的日志条目和副本的形式。一旦一个服务器将新的配置日志条目增加到它的日志中，他就会用这个配置来做出未来所有的决定（服务器总是使用最新的配置，无论他是否已经被提交）。这意味着领导人要使用 C-old,new 的规则来决定日志条目 C-old,new 什么时候需要被提交。如果领导人崩溃了，被选出来的新领导人可能是使用 C-old 配置也可能是 C-old,new 配置，这取决于赢得选举的候选人是否已经接收到了 C-old,new 配置。在任何情况下，C-new 配置在这一时期都不会单方面的做出决定。

一旦 C-old,new 被提交，那么无论是 C-old 还是 C-new，在没有经过他人批准的情况下都不可能做出决定，并且领导人完全特性保证了只有拥有 C-old,new 日志条目的服务器才有可能被选举为领导人。这个时候，领导人创建一条关于 C-new 配置的日志条目并复制给集群就是安全的了。再者，每个服务器在见到新的配置的时候就会立即生效。当新的配置在 C-new 的规则下被提交，旧的配置就变得无关紧要，同时不使用新的配置的服务器就可以被关闭了。如图 11，C-old 和 C-new 没有任何机会同时做出单方面的决定；这保证了安全性。

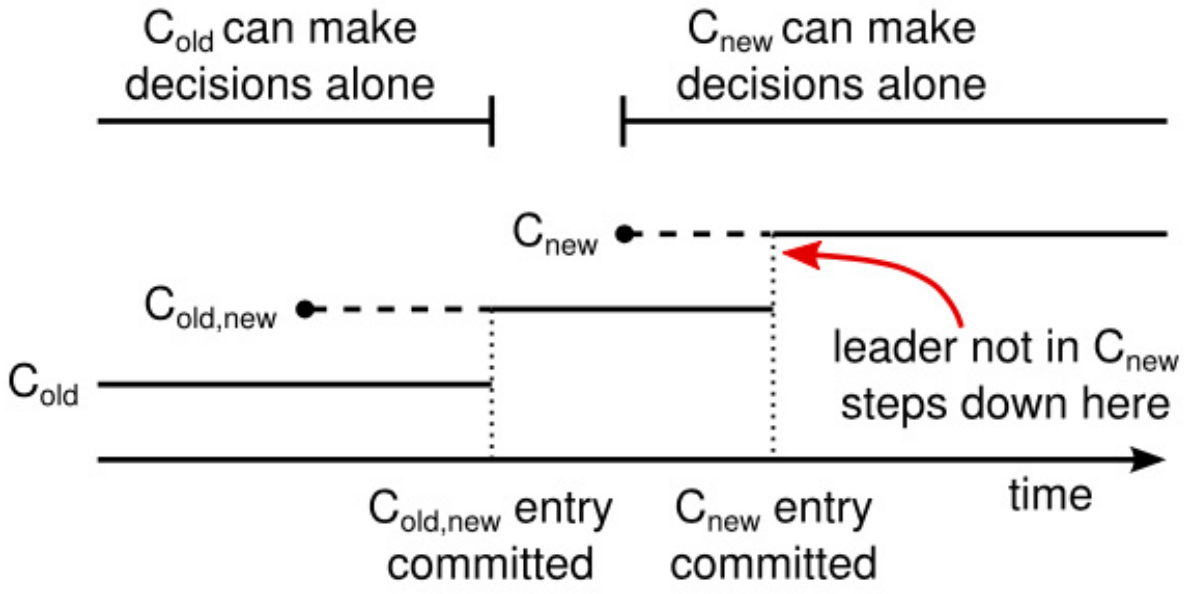


图 11：一个配置切换的时间线。虚线表示已经被创建但是还没有被提交的条目，实线表示最后被提交的日志条目。领导人首先创建了 C-old,new 的配置条目在自己的日志中，并提交到 C-old,new 中（C-old 的大多数和 C-new 的大多数）。然后他创建 C-new 条目并提交到 C-new 中的大多数。这样就不存在 C-new 和 C-old 可以同时做出决定的时间点。

在关于重新配置还有三个问题需要提出。第一个问题是，新的服务器可能初始化没有存储任何的日志条目。当这些服务器以这种状态加入到集群中，那么他们需要一段时间来更新追赶，这时还不能提交新的日志条目。为了避免这种可用性的间隔时间，Raft 在配置更新的时候使用了一种额外的阶段，在这个阶段，新的服务器以没有投票权身份加入到集群中来（领导人复制日志给他们，但是不考虑他们是大多数）。一旦新的服务器追赶上了集群中的其他机器，重新配置可以像上面描述的一样处理。

第二个问题是，集群的领导人可能不是新配置的一员。在这种情况下，领导人就会在提交了 C-new 日志之后退位（回到跟随者状态）。这意味着有这样的一段时间，领导人管理着集群，但是不包括他自己；他复制日志但是不把他自己算作是大多数之一。当 C-new 被提交时，会发生领导人过渡，因为这时是最早新的配置可以独立工作的时间点（将总是能够在 C-new 配置下选出新的领导人）。在此之前，可能只能从 C-old 中选出领导人。

第三个问题是，移除不在 C-new 中的服务器可能会扰乱集群。这些服务器将不会再接收到心跳，所以当选举超时，他们就会进行新的选举过程。他们会发送拥有新的任期号的请求投票 RPCs，这样会导致当前的领导人回退成跟随者状态。新的领导人最终会被选出来，但是被移除的服务器将会再次超时，然后这个过程会再次重复，导致整体可用性大幅降低。

为了避免这个问题，当服务器确认当前领导人存在时，服务器会忽略请求投票 RPCs。特别的，当服务器在

当前最小选举超时时间内收到一个请求投票 RPC，他不会更新当前的任期号或者投出选票。这不会影响正常的选举，每个服务器在开始一次选举之前，至少等待一个最小选举超时时间。然而，这有利于避免被移除的服务器扰乱：如果领导人能够发送心跳给集群，那么他就不会被更大的任期号废黜。

7 日志压缩

Raft 的日志在正常操作中不断的增长，但是在实际的系统中，日志不能无限制的增长。随着日志不断增长，他会占用越来越多的空间，花费越来越多的时间来重置。如果没有一定的机制去清除日志里积累的陈旧的信息，那么会带来可用性问题。

快照是最简单的压缩方法。在快照系统中，整个系统的状态都以快照的形式写入到稳定的持久化存储中，然后到那个时间点之前的日志全部丢弃。快照技术被使用在 Chubby 和 ZooKeeper 中，接下来的章节会介绍 Raft 中的快照技术。

增量压缩的方法，例如日志清理或者日志结构合并树，都是可行的。这些方法每次只对一小部分数据进行操作，这样就分散了压缩的负载压力。首先，他们先选择一个已经积累的大量已经被删除或者被覆盖对象的区域，然后重写那个区域还活跃的对象，之后释放那个区域。和简单操作整个数据集合的快照相比，需要增加复杂的机制来实现。状态机可以实现 LSM tree 使用和快照相同的接口，但是日志清除方法就需要修改 Raft 了。

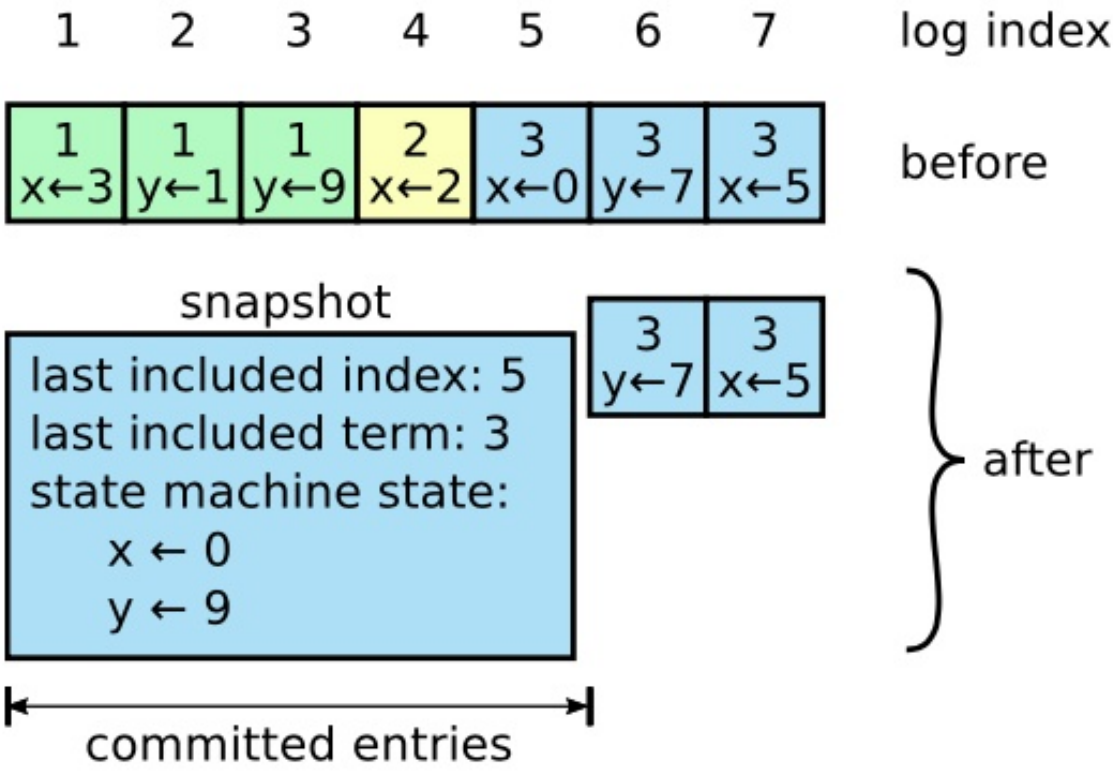


图 12：一个服务器用新的快照替换了从 1 到 5 的条目，快照值存储了当前的状态。快照中包含了最后的索引位置和任期号。

图 12 展示了 Raft 中快照的基础思想。每个服务器独立的创建快照，只包括已经被提交的日志。主要的工作包括将状态机的状态写入到快照中。Raft 也包含一些少量的元数据到快照中：**最后被包含索引**指的是被快照取代的最后的条目在日志中的索引值（状态机最后应用的日志），**最后被包含的任期**指的是该条目的任期号。保留这些数据是为了支持快照后紧接着的第一个条目的附加日志请求时的一致性检查，因为这个条目需要前一日志条目的索引值和任期号。为了支持集群成员更新（第 6 节），快照中也将最后的一次配置作为最后一个条目存下来。一旦服务器完成一次快照，他就可以删除最后索引位置之前的所有日志和快照了。

尽管通常服务器都是独立的创建快照，但是领导人必须偶尔的发送快照给一些落后的跟随者。这通常发生在当领导人已经丢弃了下一条需要发送给跟随者的日志条目的时候。幸运的是这种情况不是常规操作：一个与领导人保持同步的跟随者通常都会有这个条目。然而一个运行非常缓慢的跟随者或者新加入集群的服务器（第 6 节）将不会有这个条目。这时让这个跟随者更新到最新的状态的方式就是通过网络把快照发送给他们。

安装快照 RPC：

由领导人调用以将快照的分块发送给跟随者。领导者总是按顺序发送分块。

接收者实现：

1. 如果 `term < currentTerm` 就立即回复
2. 如果是第一个分块（offset 为 0）就创建一个新的快照
3. 在指定偏移量写入数据
4. 如果 done 是 false，则继续等待更多的数据
5. 保存快照文件，丢弃具有较小索引的任何现有或部分快照
6. 如果现存的日志条目与快照中最后包含的日志条目具有相同的索引值和任期号，则保留其后的日志条目并进行回复
7. 丢弃整个日志
8. 使用快照重置状态机（并加载快照的集群配置）

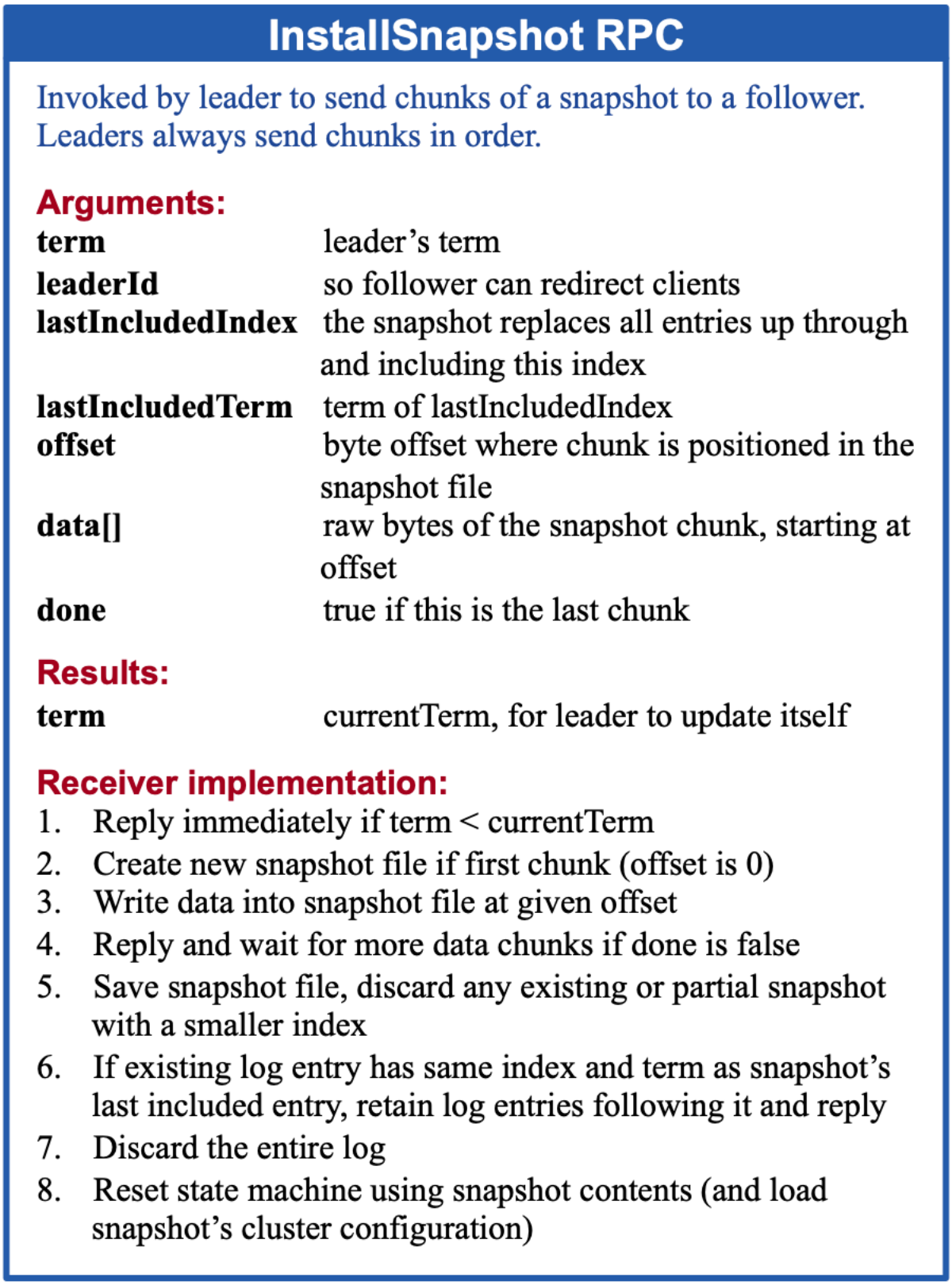


图 13：一个关于安装快照的简要概述。为了便于传输，快照都是被分成分块的；每个分块都给了跟随者生命的迹象，所以跟随者可以重置选举超时计时器。

在这种情况下领导人使用一种叫做安装快照的新的 RPC 来发送快照给太落后的跟随者；见图 13。当跟随者通过这种 RPC 接收到快照时，他必须自己决定对于已经存在的日志该如何处理。通常快照会包含没有在接收者日志中存在的信息。在这种情况下，跟随者丢弃其整个日志；它全部被快照取代，并且可能包含与快照冲突的未提交条目。如果接收到的快照是自己日志的前面部分（由于网络重传或者错误），那么被快照包含的条目将会被全部删除，但是快照后面的条目仍然有效，必须保留。

这种快照的方式背离了 Raft 的强领导人原则，因为跟随者可以在不知道领导人情况下创建快照。但是我们认为这种背离是值得的。领导人的存在，是为了解决在达成一致性的时候的冲突，但是在创建快照的时候，一致性已经达成，这时不存在冲突了，所以没有领导人也是可以的。数据依然是从领导人传给跟随者，只是跟随者可以重新组织他们的数据了。

我们考虑过一种替代的基于领导人的快照方案，即只有领导人创建快照，然后发送给所有的跟随者。但是这样做有两个缺点。第一，发送快照会浪费网络带宽并且延缓了快照处理的时间。每个跟随者都已经拥有了所有产生快照需要的信息，而且很显然，自己从本地的状态中创建快照比通过网络接收别人发来的要经济。第二，领导人的实现会更加复杂。例如，领导人需要发送快照的同时并行的将新的日志条目发送给跟随者，这样才不会阻塞新的客户端请求。

还有两个问题影响了快照的性能。首先，服务器必须决定什么时候应该创建快照。如果快照创建的过于频繁，那么就会浪费大量的磁盘带宽和其他资源；如果创建快照频率太低，他就要承受耗尽存储容量的风险，同时也增加了从日志重建的时间。一个简单的策略就是当日志大小达到一个固定大小的时候就创建一次快照。如果这个阈值设置的显著大于期望的快照的大小，那么快照对磁盘压力的影响就会很小了。

第二个影响性能的问题就是写入快照需要花费显著的一段时间，并且我们还不希望影响到正常操作。解决方案是通过写时复制的技术，这样新的更新就可以被接收而不影响到快照。例如，具有函数式数据结构的状态机天然支持这样的功能。另外，操作系统的写时复制技术的支持（如 Linux 上的 fork）可以被用来创建完整的状态机的内存快照（我们的实现就是这样的）。

8 客户端交互

这一节将介绍客户端是如何和 Raft 进行交互的，包括客户端如何发现领导人和 Raft 是如何支持线性化语义的。这些问题对于所有基于一致性的系统都存在，并且 Raft 的解决方案和其他的也差不多。

Raft 中的客户端发送所有请求给领导人。当客户端启动的时候，他会随机挑选一个服务器进行通信。如果客户端第一次挑选的服务器不是领导人，那么那个服务器会拒绝客户端的请求并且提供他最近接收到的领导人的信息（附加条目请求包含了领导人的网络地址）。如果领导人已经崩溃了，那么客户端的请求就会超时；客户端之后会再次重试随机挑选服务器的过程。

我们 Raft 的目标是要实现线性化语义（每一次操作立即执行，只执行一次，在他调用和收到回复之间）。但是，如上述，Raft 是可以执行同一条命令多次的：例如，如果领导人在提交了这条日志之后，但是在响应客户端之前崩溃了，那么客户端会和新的领导人重试这条指令，导致这条命令就被再次执行了。解决方案就是客户端对于每一条指令都赋予一个唯一的序列号。然后，状态机跟踪每条指令最新的序列号和相应的响应。如果接收到一条指令，它的序列号已经被执行了，那么就立即返回结果，而不重新执行指令。

只读的操作可以直接处理而不需要记录日志。但是，在不增加任何限制的情况下，这么做可能会冒着返回脏数据的风险，因为领导人响应客户端请求时可能已经被新的领导人作废了，但是他还不知道。线性化的读操作必须不能返回脏数据，Raft 需要使用两个额外的措施在不使用日志的情况下保证这一点。首先，领导人必须有关于被提交日志的最新信息。领导人完全特性保证了领导人一定拥有所有已经被提交的日志条目，但是在他任期开始的时候，他可能不知道那些是已经被提交的。为了知道这些信息，他需要在他的任期里提交一条日志条目。Raft 中通过领导人在任期开始的时候提交一个空白的没有任何操作的日志条目到日志中去来实现。第二，领导人在处理只读的请求之前必须检查自己是否已经被废黜了（他自己的信息已经变脏了如果一个更新的领导人被选举出来）。Raft 中通过让领导人在响应只读请求之前，先和集群中的大多数节点交换一次心跳信息来处理这个问题。可选的，领导人可以依赖心跳机制来实现一种租约的机制，但是这种方法依赖时间来保证安全性（假设时间误差是有界的）。

9 算法实现和评估

我们已经为 RAMCloud 实现了 Raft 算法作为存储配置信息的复制状态机的一部分，并且帮助 RAMCloud 协调故障转移。这个 Raft 实现包含大约 2000 行 C++ 代码，其中不包括测试、注释和空行。这些代码是开源的。同时也有大约 25 个其他独立的第三方的基于这篇论文草稿的开源实现，针对不同的开发场景。同时，很多公司已经部署了基于 Raft 的系统。

这一节会从三个方面来评估 Raft 算法：可理解性、正确性和性能。

9.1 可理解性

为了和 Paxos 比较 Raft 算法的可理解能力，我们针对高层次的本科生和研究生，在斯坦福大学的高级操作系统课程和加州大学伯克利分校的分布式计算课程上，进行了一次学习的实验。我们分别拍了针对 Raft 和 Paxos 的视频课程，并准备了相应的小测验。Raft 的视频讲课覆盖了这篇论文的所有内容除了日志压缩；Paxos 讲课包含了足够的资料来创建一个等价的复制状态机，包括单决策 Paxos，多决策 Paxos，重新配置和一些实际系统需要的性能优化（例如领导人选举）。小测验测试一些对算法的基本理解和解释一些边角的示例。每个学生都是看完第一个视频，回答相应的测试，再看第二个视频，回答相应的测试。大约有一半的学生先进行 Paxos 部分，然后另一半先进行 Raft 部分，这是为了说明两者从第一部分的算法学习中获得的表现和经验的差异。我们计算参加人员的每一个小测验的得分来看参与者是否在 Raft 算法上更加容易理解。

我们尽可能的使得 Paxos 和 Raft 的比较更加公平。这个实验偏爱 Paxos 表现在两个方面：43 个参加者中有 15 个人在之前有一些 Paxos 的经验，并且 Paxos 的视频要长 14%。如表格 1 总结的那样，我们采取了一些措施来减轻这种潜在的偏见。我们所有的材料都可供审查。

表 1：考虑到可能会存在的偏见，对于每种情况的解决方法，和相应的材料。

参加者平均在 Raft 的测验中比 Paxos 高 4.9 分（总分 60，那么 Raft 的平均得分是 25.7，而 Paxos 是 20.8）；图 14 展示了每个参与者的得分。配置 t-检验（又称 student‘s t-test）表明，在 95% 的可信度下，真实的 Raft 分数分布至少比 Paxos 高 2.5 分。

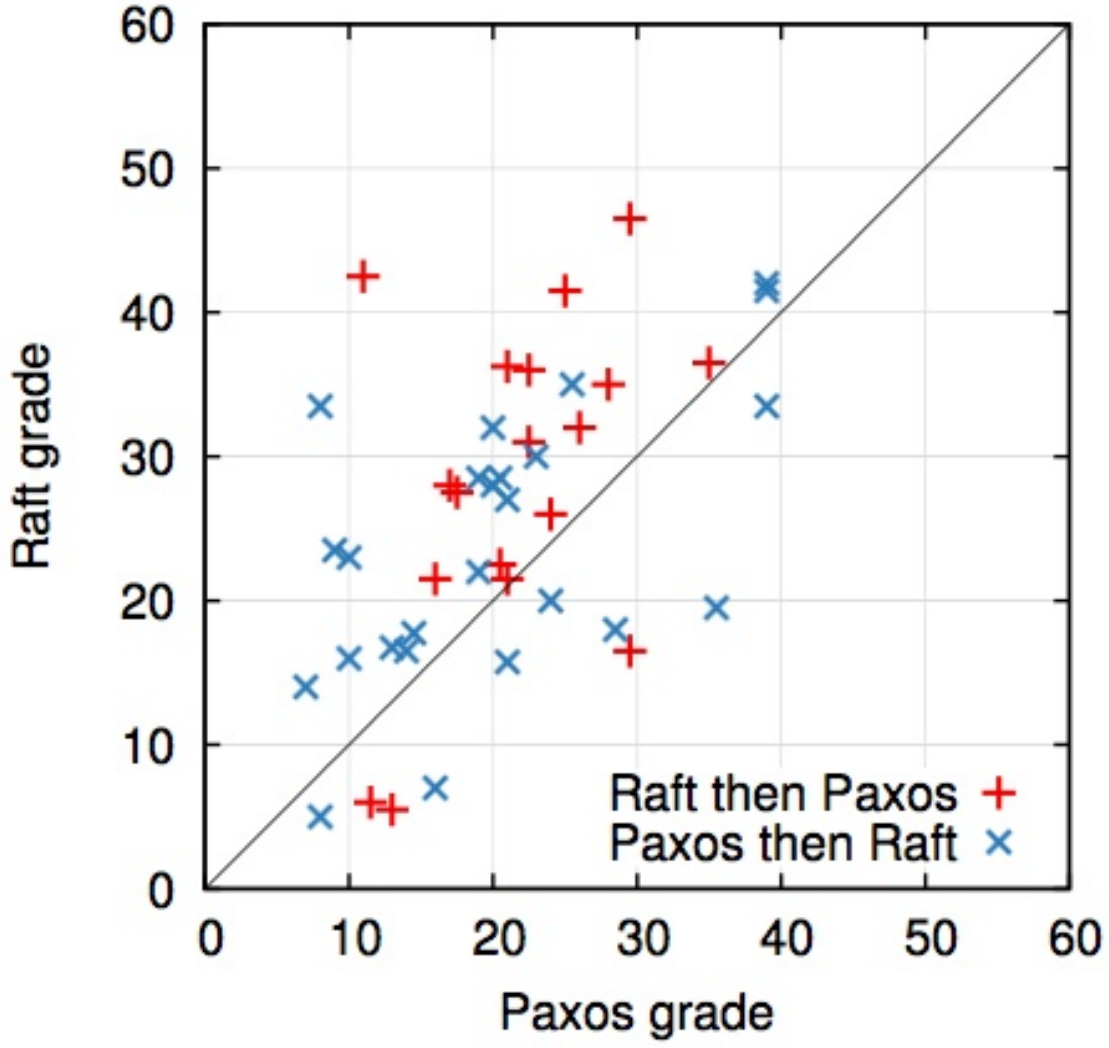


图 14：一个散点图表示了 43 个学生在 Paxos 和 Raft 的小测验中的成绩。在对角线之上的点表示在 Raft 获得了更高分数的学生。

我们也建立了一个线性回归模型来预测一个新的学生的测验成绩，基于以下三个因素：他们使用的是哪个小测验，之前对 Paxos 的经验，和学习算法的顺序。模型预测，对小测验的选择会产生 12.5 分的差别。这显著的高于之前的 4.9 分，因为很多学生在之前都已经有了对于 Paxos 的经验，这相当明显的帮助 Paxos，对 Raft 就没什么太大影响了。但是奇怪的是，模型预测对于先进行 Paxos 小测验的人而言，Raft 的得分低了 6.3 分；虽然我们不知道为什么，这似乎在统计上是有意义的。

我们同时也在测验之后调查了参与者，他们认为哪个算法更加容易实现和解释；这个的结果在图 15 上。压倒性的结果表明 Raft 算法更加容易实现和解释（41 人中的 33 个）。但是，这种自己报告的结果不如参与者的成绩更加可信，并且参与者可能因为我们的 Raft 更加易于理解的假说而产生偏见。

图 15：通过一个 5 分制的问题，参与者（左边）被问哪个算法他们觉得在一个高效正确的系统里更容易实现，右边被问哪个更容易向学生解释。

关于 Raft 用户学习有一个更加详细的讨论。

9.2 正确性

在第 5 节，我们已经制定了正式的规范，和对一致性机制的安全性证明。这个正式规范使用 TLA+ 规范语言使图 2 中总结的信息非常清晰。它长约 400 行，并作为证明的主题。同时对于任何想实现 Raft 的人也是十分有用的。我们通过 TLA 证明系统非常机械的证明了日志完全特性。然而，这个证明依赖的约束前提还没有被机械证明（例如，我们还没有证明规范的类型安全）。而且，我们已经写了一个非正式的证明关于状态机安全性是完备的，并且是相当清晰的（大约 3500 个词）。

9.3 性能

Raft 和其他一致性算法例如 Paxos 有着差不多的性能。在性能方面，最重要的关注点是，当领导人被选举成功时，什么时候复制新的日志条目。Raft 通过很少数量的消息包（一轮从领导人到集群大多数机器的消息）就达成了这个目的。同时，进一步提升 Raft 的性能也是可行的。例如，很容易通过支持批量操作和管道操作来提高吞吐量和降低延迟。对于其他一致性算法已经提出过很多性能优化方案；其中有很多也可以应用到 Raft 中来，但是我们暂时把这个问题放到未来的工作中去。

我们使用我们自己的 Raft 实现来衡量 Raft 领导人选举的性能并且回答两个问题。首先，领导人选举的过程收敛是否快速？第二，在领导人宕机之后，最小的系统宕机时间是多久？

> 图 16：发现并替换一个已经崩溃的领导人的时间。上面的图考察了在选举超时时间上的随机化程度，下面的图考察了最小选举超时时间。每条线代表了 1000 次实验（除了 150-150 毫秒只试了 100 次），和相应的确定的选举超时时间。例如，150-155 毫秒意思是，选举超时时间从这个区间范围内随机选择并确定下来。这个实验在一个拥有 5 个节点的集群上进行，其广播时延大约是 15 毫秒。对于 9 个节点的集群，结果也差不多。

为了衡量领导人选举，我们反复的使一个拥有五个节点的服务器集群的领导人宕机，并计算需要多久才能发现领导人已经宕机并选出一个新的领导人（见图 16）。为了构建一个最坏的场景，在每一的尝试里，服务器都有不同长度的日志，意味着有些候选人是没有成为领导人的资格的。另外，为了促成选票瓜分的情况，我们的测试脚本在终止领导人之前同步的发送了一次心跳广播（这大约和领导人在崩溃前复制一个新的日志给其他机器很像）。领导人均匀的随机的在心跳间隔里宕机，也就是最小选举超时时间的一半。因此，最小宕机时间大约就是最小选举超时时间的一半。

图 16 中上面的图表明，只需要在选举超时时间上使用很少的随机化就可以大大避免选票被瓜分的情况。在没有随机化的情况下，在我们的测试里，选举过程往往都需要花费超过 10 秒钟由于太多的选票瓜分的情况。仅仅增加 5 毫秒的随机化时间，就大大的改善了选举过程，现在平均的宕机时间只有 287 毫秒。增加更多的随机化时间可以大大改善最坏情况：通过增加 50 毫秒的随机化时间，最坏的完成情况（1000 次尝试）只要 513 毫秒。

图 16 中下面的图显示，通过减少选举超时时间可以减少系统的宕机时间。在选举超时时间为 12-24 毫秒的情况下，只需要平均 35 毫秒就可以选举出新的领导人（最长的一次花费了 152 毫秒）。然而，进一步降低选举超时时间的话就会违反 Raft 的时间不等式需求：在选举新领导人之前，领导人就很难发送完心跳包。这会导致没有意义的领导人改变并降低了系统整体的可用性。我们建议使用更为保守的选举超时时间，比如 150-300 毫秒；这样的时间不大可能导致没有意义的领导人改变，而且依然提供不错的可用性。

10 相关工作

已经有很多关于一致性算法的工作被发表出来，其中很多都可以归到下面的类别中：

- Lamport 关于 Paxos 的原始描述，和尝试描述的更清晰。
- 关于 Paxos 的更详尽的描述，补充遗漏的细节并修改算法，使得可以提供更加容易的实现基础。
- 实现一致性算法的系统，例如 Chubby，ZooKeeper 和 Spanner。对于 Chubby 和 Spanner 的算法并没有公开发表其技术细节，尽管他们都声称是基于 Paxos 的。ZooKeeper 的算法细节已经发表，但是和 Paxos 着实有着很大的差别。
- Paxos 可以应用的性能优化。
- Oki 和 Liskov 的 Viewstamped Replication（VR），一种和 Paxos 差不多的替代算法。原始的算法描述和分布式传输协议耦合在了一起，但是核心的一致性算法在最近的更新里被分离了出来。VR 使用了一种基于领导人的方法，和 Raft 有很多相似之处。

Raft 和 Paxos 最大的不同之处就在于 Raft 的强领导特性：Raft 使用领导人选举作为一致性协议里必不可少的部分，并且将尽可能多的功能集中到了领导人身上。这样就可以使得算法更加容易理解。例如，在 Paxos 中，领导人选举和基本的一致性协议是正交的：领导人选举仅仅是性能优化的手段，而且不是一致性所必须要求的。但是，这样就增加了多余的机制：Paxos 同时包含了针对基本一致性要求的两阶段提交协议和针对领导人选举的独立的机制。相比较而言，Raft 就直接将领导人选举纳入到一致性算法中，并作为两阶段一致性的第一步。这样就减少了很多机制。

像 Raft 一样，VR 和 ZooKeeper 也是基于领导人的，因此他们也拥有一些 Raft 的优点。但是，Raft 比 VR 和 ZooKeeper 拥有更少的机制因为 Raft 尽可能的减少了非领导人的功能。例如，Raft 中日志条目都遵循着从领导人发送给其他人这一个方向：附加条目 RPC 是向外发送的。在 VR 中，日志条目的流动是双向的（领导人可以在选举过程中接收日志）；这就导致了额外的机制和复杂性。根据 ZooKeeper 公开的资料看，它的日志条目也是双向传输的，但是它的实现更像 Raft。

和上述我们提及的其他基于一致性的日志复制算法中，Raft 的消息类型更少。例如，我们数了一下 VR 和 ZooKeeper 使用的用来基本一致性需要和成员改变的消息数（排除了日志压缩和客户端交互，因为这些都比较独立且和算法关系不大）。VR 和 ZooKeeper 都分别定义了 10 中不同的消息类型，相对的，Raft 只有 4 中消息类型（两种 RPC 请求和对应的响应）。Raft 的消息都稍微比其他算法的要信息量大，但是都很简单。另外，VR 和 ZooKeeper 都在领导人改变时传输了整个日志；所以为了能够实践中使用，额外的消息类型就很必要了。

Raft 的强领导人模型简化了整个算法，但是同时也排斥了一些性能优化的方法。例如，平等主义 Paxos（EPaxos）在某些没有领导人的情况下可以达到很高的性能。平等主义 Paxos 充分发挥了在状态机指令中的交换性。任何服务器都可以在一轮通信下就提交指令，除非其他指令同时被提出了。然而，如果指令都是并发的被提出，并且互相之间不通信沟通，那么 EPaxos 就需要额外的一轮通信。因为任何服务器都可以提交指令，所以 EPaxos 在服务器之间的负载均衡做的很好，并且很容易在 WAN 网络环境下获得很低的延迟。但是，他在 Paxos 上增加了非常明显的复杂性。

一些集群成员变换的方法已经被提出或者在其他的工作中被实现，包括 Lamport 的原始的讨论，VR 和 SMART。我们选择使用共同一致的方法因为他对一致性协议的其他部分影响很小，这样我们只需要很少的一些机制就可以实现成员变换。Lamport 的基于 α 的方法之所以没有被 Raft 选择是因为它假设在没有领导人的情况下也可以达到一致性。和 VR 和 SMART 相比较，Raft 的重新配置算法可以在不限制正常请求处理的情况下进行；相比较的，VR 需要停止所有的处理过程，SMART 引入了一个和 α 类似的方法，限制了请求处理的数量。Raft 的方法同时也需要更少的额外机制来实现，和 VR、SMART 比较而言。

11 结论

算法的设计通常会把正确性，效率或者简洁作为主要的目标。尽管这些都是很有意义的目标，但是我们相信，可理解性也是一样的重要。在开发者把算法应用到实际的系统中之前，这些目标没有一个会被实现，这些都会必然的偏离发表时的形式。除非开发人员对这个算法有着很深的理解并且有着直观的感觉，否则将会对他们而言很难在实现的时候保持原有期望的特性。

在这篇论文中，我们尝试解决分布式一致性问题，但是一个广为接受但是十分令人费解的算法 Paxos 已经困扰了无数学生和开发者很多年了。我们创造了一种新的算法 Raft，显而易见的比 Paxos 要容易理解。我们同时也相信，Raft 也可以为实际的实现提供坚实的基础。把可理解性作为设计的目标改变了我们设计 Raft 的方式；随着设计的进展，我们发现自己重复使用了一些技术，比如分解问题和简化状态空间。这些技术不仅提升了 Raft 的可理解性，同时也使我们坚信其正确性。

12 感谢

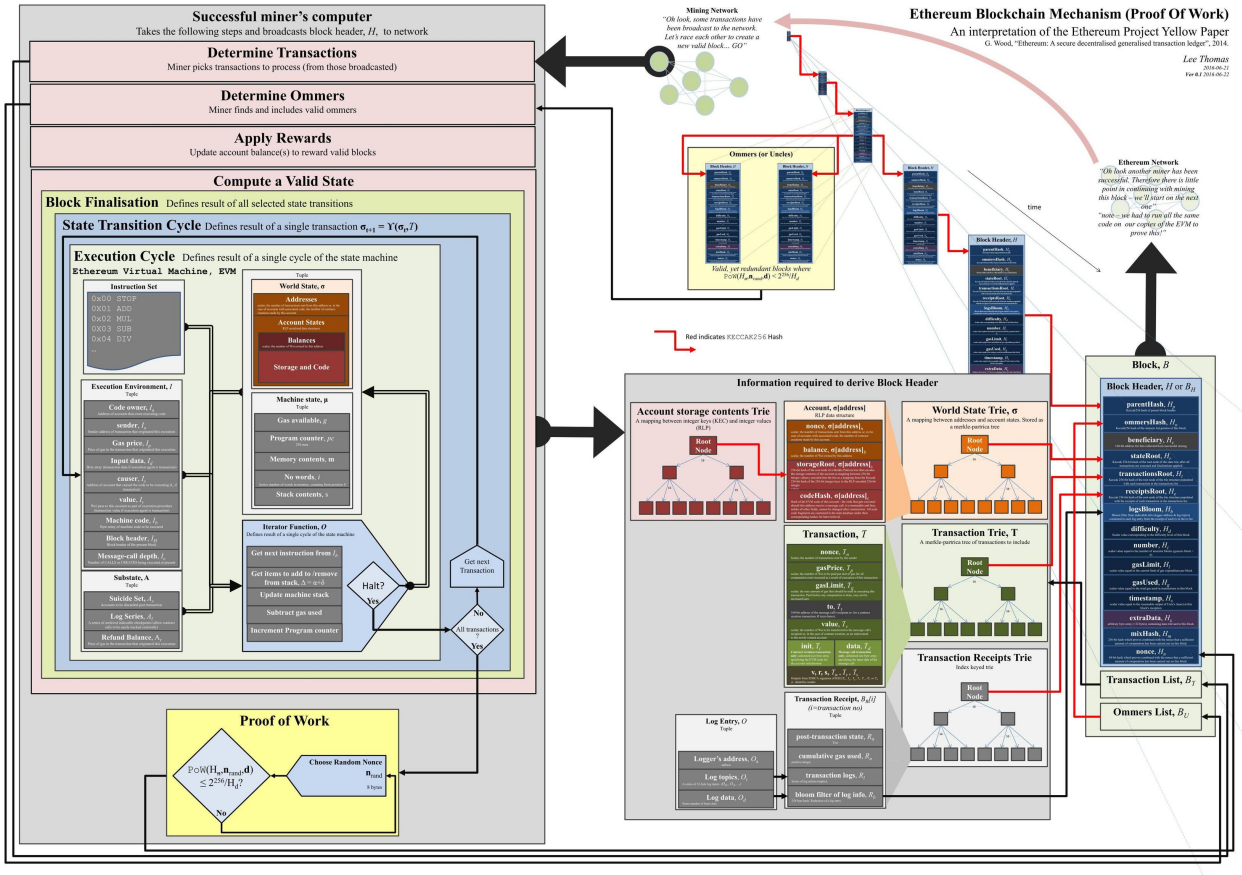
这项研究必须感谢以下人员的支持：Ali Ghodsi，David Mazie‘res，和伯克利 CS 294-91 课程、斯坦福 CS 240 课程的学生。Scott Klemmer 帮我们设计了用户调查，Nelson Ray 建议我们进行统计学的分析。在用户调查时使用的关于 Paxos 的幻灯片很大一部分是从 Lorenzo Alvisi 的幻灯片上借鉴过来的。特别的，非常感谢 DavidMazieres 和 Ezra Hoch，他们找到了 Raft 中一些难以发现的漏洞。许多人提供了关于这篇论文十分有用的反馈和用户调查材料，包括 Ed Bugnion，Michael Chan，Hugues Evrard，Daniel Giffin，Arjun Gopalan，Jon Howell，Vimalkumar Jeyakumar，Ankita Kejriwal，Aleksandar Kracun，Amit Levy，Joel Martin，Satoshi Matsushita，Oleg Pesok，David Ramos，Robbert van Renesse，Mendel Rosenblum，Nicolas Schiper，Deian Stefan，Andrew Stone，Ryan Stutsman，David Terei，Stephen Yang，Matei Zaharia 以及 24 位匿名的会议审查人员（可能有重复），并且特别感谢我们的领导人 Eddie Kohler。Werner Vogels 发了一条早期草稿链接的推特，给 Raft 带来了极大的关注。我们的工作由 Gigascale 系统研究中心和 Multiscale 系统研究中心给予支持，这两个研究中心由关注中心研究程序资金支持，一个是半导体研究公司的程序，由 STARnet 支持，一个半导体研究公司的程序由 MARCO 和 DARPA 支持，在国家科学基金会的 0963859 号批准，并且获得了来自 Facebook，Google，Mellanox，NEC，NetApp，SAP 和 Samsung 的支持。Diego Ongaro 由 Junglee 公司，斯坦福的毕业团体支持。

原文链接

https://github.com/maemual/raft-zh_cn/blob/master/raft-zh_cn.md

3.2 OverView

3.2.1 ETH 结构



3.3 Wallet

3.3.1 多重签名

多重签名脚本设置了一个条件，其中 N 个公钥被记录在脚本中，并且至少有 M 个必须提供签名来解锁资金。这也称为 M-N 方案，其中 N 是密钥的总数，M 是验证所需的签名的数量。例如，2/3 的多重签名是三个公钥被列为潜在签名人，至少有 2 个有效的签名才能花费资金。

设置 M-N 多重签名条件的锁定脚本的一般形式是：

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N CHECKMULTISIG
```

M 是花费输出所需的签名的数量，N 是列出的公钥的总数。设置 2 到 3 多重签名条件的锁定脚本如下所示：

```
2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG
```

上述锁定脚本可由含有签名和公钥的脚本予以解锁：或者由 3 个存档公钥中的任意 2 个相一致的私钥签名组合予以解锁。两个脚本组合将形成一个验证脚本：

```
<Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3␣
↪CHECKMULTISIG
```

上述例子中相应的设置条件即为：解锁脚本是否含有 3 个公钥中的任意 2 个相对应的私钥的有效签名。

3.3.2 加密与签名

加密算法

椭圆曲线加密（ECC：Elliptic Curve Cryptography）

1. 原理性质

椭圆曲线加密法是一种基于离散对数问题的非对称加密法，可以用对椭圆曲线上的点进行加法或乘法运算来表达。下图是一个椭圆曲线的示例，类似于比特币所用的曲线。

secp256k1

```
y^2 = (x^3 + 7) \over (F_p)

 y^2 \bmod p = (x^3 + 7) \bmod p
```

上述 mod p（素数 p 取模）表明该曲线是在素数阶 p 的有限域内，也写作 Fp，其中 p = 2^256 –2^32 –2^9 –2^8 –2^7 –2^6 –2^4 –1，这是个非常大的素数

数字签名（ECDSA）

BTC 数字签名算法是椭圆曲线数字签名算法（Elliptic Curve Digital Signature Algorithm）或 ECDSA。

ECDSA 用于脚本函数 `OP_CHECKSIG`，`OP_CHECKSIGVERIFY`，`OP_CHECKMULTISIG` 和 `OP_CHECKMULTISIGVERIFY`。

数字签名在比特币中有三种用途。

- 第一，签名证明私钥的所有者，即资金所有者，已经授权支出这些资金
- 第二，授权证明是不可否认的（不可否认性）
- 第三，签名证明交易（或交易的具体部分）在签字之后没有也不能被任何人修改。

通常使用椭圆曲线算法生成密钥对

- 比特币密钥长度：256 位
- 公钥哈希值 =RIMPED160(SHA256(公钥))
- 比特币地址 =*1*+Base58(0+ 公钥哈希值 + 校验码)
- 校验码 = 前四字节 (SHA256(SHA256(0+ 公钥哈希值)))

签名

- 发送方使用 HASH 算法计算数据的 HASH 值
- 发送方使用本方的私钥加密 HASH 值，得到签名

- 接收方使用 HASH 算法计算数据的 HASH 值
- 接收方使用发送方的公钥解密签名得到发送的 HASH 值
- 比较两个 HASH 值的一致性

数字签名如何工作

数字签名是一种由两部分组成的数学方案：

- 第一部分是使用私钥（签名密钥）从消息（交易）创建签名的算法；
- 第二部分是允许任何人验证签名的算法，给定消息和公钥。

创建签名

在比特币的 ECDSA 算法的实现中，被签名的“消息”是交易，或更确切地说是交易中特定数据子集的哈希值（参见签名哈希类型（SIGHASH））。签名密钥是用户的私钥，结果是签名：

```
Sig = F{sig}(F{hash}(m),   dA)
```

- dA 是签名私钥
- m 是交易（或其部分）
- Fhash 是散列函数
- Fsig 是签名算法
- Sig 是结果签名

函数 `Fsig` 产生由两个值组成的签名 Sig，通常称为 R 和 S：

验证签名

要验证签名，必须有签名（R 和 S）、序列化交易和公钥（对应于用于创建签名的私钥）。本质上，签名的验证意味着“只有生成此公钥的私钥的所有者，才能在此交易上产生此签名。”

签名验证算法采用消息（交易或其部分的哈希值）、签名者的公钥和签名（R 和 S 值），如果签名对该消息和公钥有效，则返回 TRUE 值。

ECDSA 数学

签名由产生由两个值 R 和 S 组成的签名的数学函数 `Fsig` 创建。

签名算法首先生成一个 ephemeral（临时）私公钥对。在涉及签名私钥和交易哈希的变换之后，该临时密钥对用于计算 R 和 S 值。

临时密钥对基于随机数 k，用作临时私钥。从 k，我们生成相应的临时公钥 P（以 P = k * G 计算，与派生比特币公钥相同）；参见 [pubkey] 部分）。数字签名的 R 值则是临时公钥 P 的 x 坐标。

2. 常用库

JavaScript :

Elliptic

支持以下曲线

- secp256k1 **==ECDSA==**
- p192
- p224
- p256
- p384
- p521
- curve25519 **==ECDH==**
- ed25519 **==EdDSA==**

Eccrypto

Only secp256k1 curve **==ECDSA== ==ECDH== ==ECIES==**

tiny-secp256k1

crypto-browserify

实现以下:

- createHash (sha1, sha224, sha256, sha384, sha512, md5, rmd160)
- createHmac (sha1, sha224, sha256, sha384, sha512, md5, rmd160)
- pbkdf2
- pbkdf2Sync
- randomBytes
- pseudoRandomBytes
- createCipher (aes)
- createDecipher (aes)
- createDiffieHellman
- createSign (rsa, ecdsa)
- createVerify (rsa, ecdsa)
- createECDH (secp256k1)
- publicEncrypt/privateDecrypt (rsa)
- privateEncrypt/publicDecrypt (rsa)

参考如下:

椭圆曲线密码学相关概念与开源实现精通比特币 - 密钥和地址

3.3.3 钱包常用知识

钱包登录

- 单链钱包：只支持一条链的钱包，比方说比特币钱包。
- 多链钱包：支持所有币种的钱包。
- 单账户钱包：每种币只支持一个账户。
- 多账户钱包：每种币支持多个账户

钱包登录的方案三种：

钱包密码

可用于转账确认，调用合约等，以及辅助 keystore 登录

助记词

私钥是 64 位长度的十六进制的字符，不利于记录且容易记错，且每个账号对应一个私钥，多个账号就有多个私钥，不易管理，所以用算法将一串随机数转化为了一串 12 ~ 24 个容易记住的单词，方便保存记录。

通过助记词可以获取相关联的多个私钥，但是通过其中一个私钥是不能获取助记词的，因此助记词 ≠ 私钥。

根据助记词推算种子的算法是 **PBKDF2**，使用的哈希函数是 Hmac-SHA512，其中，输入是助记词的 UTF-8 编码，并设置 Key 为 mnemonic+password，循环 2048 次，得到最终的 64 字节种子。

KeyStore

```
{
   "address":"856e604698f79cef417aab...",
   "crypto":{
      "cipher":"aes-128-ctr",
      "ciphertext":"13a3ad2135bef1ff228e399dfc8d7757eb4bb1a81d1b31....",
      "cipherparams":{
         "iv":"92e7468e8625653f85322fb3c..."
      },
      "kdf":"scrypt",
      "kdfparams":{
         "dklen":32,
         "n":262144,
         "p":1,
         "r":8,
         "salt":"3ca198ce53513ce01bd651aee54b16b6a...."
      },
      "mac":"10423d837830594c18a91097d09b7f2316..."
   },
   "id":"5346bac5-0a6f-4ac6-baba-e2f3ad464f3f",
```

(下页继续)

(续上页)

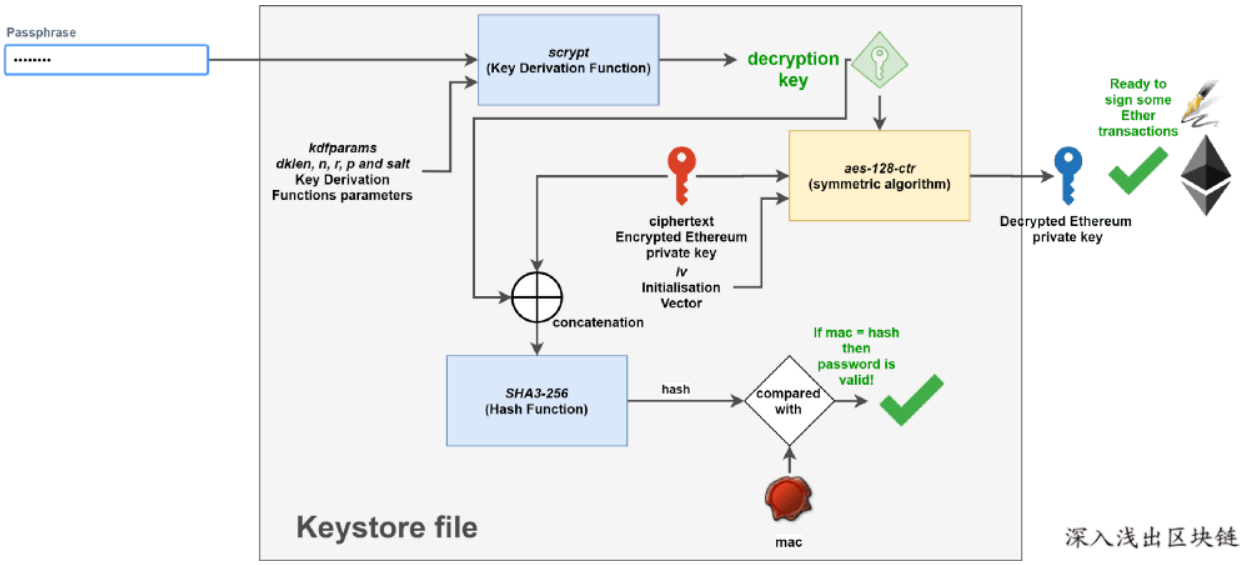
```
    "version":3
}
```

- address: 账号地址
- version: Keystore 文件的版本，目前为第 3 版，也称为 V3 KeyStore。
- salt 是一段随机的盐
- dk_len 是输出的哈希值的长度
- n 是 CPU/Memory 开销值，越高的开销值，计算就越困难。
- r 表示块大小
- p 表示并行度。
- id : uuid
- crypto: 加密推倒的相关配置:
  1. cipher 是用于加密以太坊私钥的对称加密算法。用的是 aes-128-ctr 。
  2. cipherparams 是 aes-128-ctr 加密算法需要的参数。在这里，用到的唯一的参数 iv。
  3. ciphertext 是加密算法输出的密文，也是将来解密时的需要的输入。
  4. kdf: 指定使用哪一个算法，这里使用的是 scrypt。
  5. kdfparams: scrypt 函数需要的参数
  6. mac: 用来校验密码的正确性，mac= sha3 (DK [16:32], ciphertext)

Keystore 文件的产生:

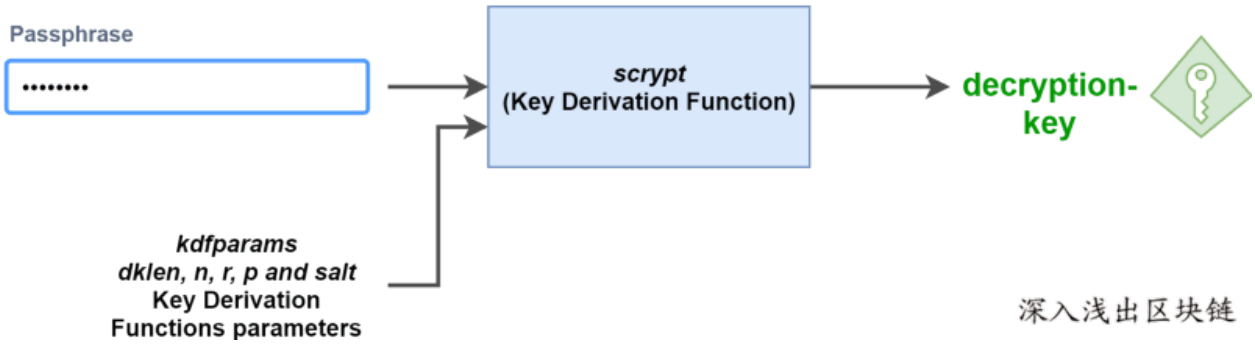
- 1、使用 scrypt 函数（根据密码和相应的参数）生成秘钥
- 2、使用上一步生成的秘钥 + 账号私钥 + 参数进行对称加密。
- 3、把相关的参数和输出的密文保存为以上格式的 JSON 文件

整体流程



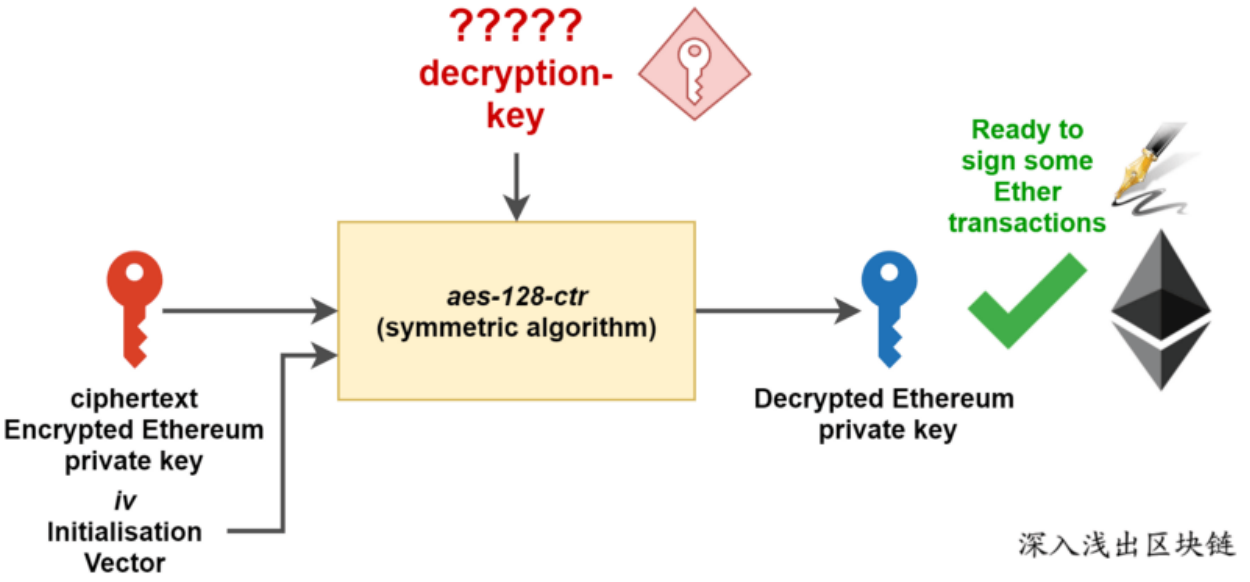
在 Keystore 中，是用的是 Scrypt 算法，用一个公式来表示的话，派生的 Key 生成方程为：

```
DK = Scrypt(salt, dk_len, n, r, p)
```



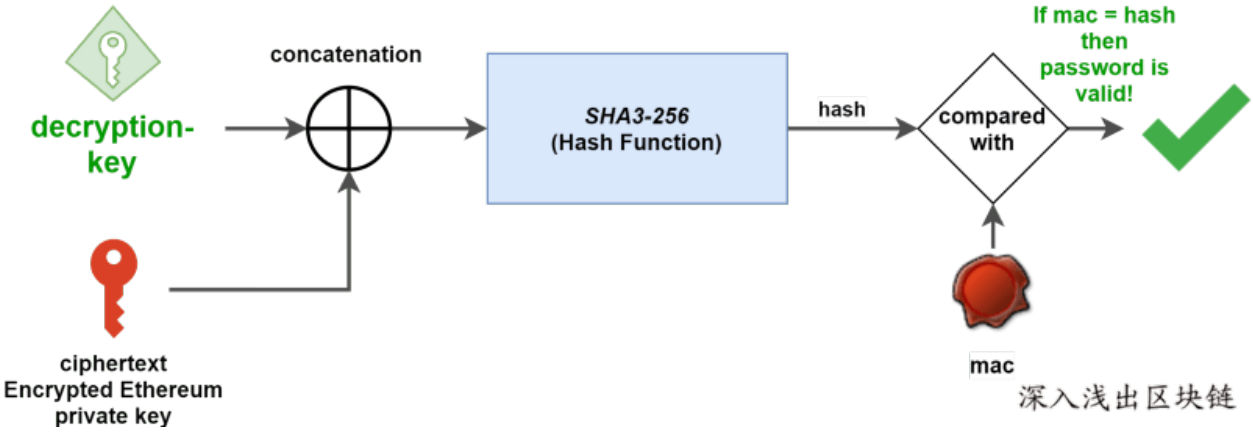
校验 Keystore

当我们在使用 Keystore 文件来还原私钥时，依然是使用 kdf 生成一个秘钥，然后用秘钥对 ciphertext 进行解密，其过程如下：



无论使用说明密码，来进行这个操作，都会生成一个私钥，但是最终计算的以太坊私钥到底是不是正确的，却不得而知。

这就是 keystore 文件中 mac 值的作用。**mac 值是 kdf 输出和 ciphertext 密文进行 SHA3-256 运算的结果，显然密码不同，计算的 mac 值也不同**，因此可以用来检验密码的正确性。



参考信息

账号 Keystore 文件导入导出精通比特币-钱包

3.3.4 UXTO

比特币完整节点跟踪所有可找到的和可使用的输出，称为“未花费的交易输出”（unspent transaction outputs），即 UTXO。所有 UTXO 的集合被称为 UTXO 集，目前有数百万个 UTXO。当新的 UTXO 被创建，UTXO 集就会变大，当 UTXO 被消耗时，UTXO 集会随着缩小。每一个交易都代表 UTXO 集的变化（状态转换）

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid":"7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig":
↪"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9
↪0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04
↪",
      "sequence": 4294967295
    }
 ],
  "vout": [
    {
      "value": 0.01500000,
      "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_
↪EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": 0.08450000,
      "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_
↪EQUALVERIFY OP_CHECKSIG",
    }
  ]
}
```

交易输出 (创世交易先有输出)

```
"vout": [
  {
    "value": 0.01500000,
    "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_
↪EQUALVERIFY
OP_CHECKSIG"
  },
  {
    "value": 0.08450000,
    "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_
↪EQUALVERIFY OP_CHECKSIG",
  }
]
```

交易输入

交易输入将 UTXO（通过引用）标记为将被消费，并通过解锁脚本提供所有权证明。

输入的第一部分是一个指向 UTXO 的指针，通过指向 UTXO 被记录在区块链中所在的交易的哈希值和序列号来实现

```
"vin": [
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" :
↪"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9
↪0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c0
↪",
    "sequence": 4294967295
  }
]
```

- 一个交易 ID，引用包含正在使用的 UTXO 的交易
- 一个输出索引（vout），用于标识来自该交易的哪个 UTXO 被引用（第一个为零）
- 一个 scriptSig（解锁脚本），满足放置在 UTXO 上的条件，解锁它用于支出
- 一个序列号（稍后讨论）

在 Alice 的交易中，输入指向的交易 ID 是：

```
7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18
```

输出索引是 0（即由该交易创建的第一个 UTXO），那个输出（如上：value）就可以被引用了。

如果不检索输入中引用的 UTXO，则不知道它们的值。因此，在单个交易中计算交易费用的简单操作，实际上涉及多个交易的多个步骤和数据。

比特币交易脚本和脚本语言

脚本构建（锁定与解锁）

- **锁定脚本**是一个放置在输出上面的花费条件：它指定了今后花费这笔输出必须要满足的条件。由于锁定脚本往往含有一个公钥或比特币地址（公钥哈希值），在历史上它曾被称为脚本公钥（scriptPubKey）。
- **解锁脚本**是一个“解决”或满足被锁定脚本在一个输出上设定的花费条件的脚本，它将允许输出被消费。解锁脚本是每一笔比特币交易输入的一部分，而且往往含有一个由用户的比特币钱包（通过用户的私钥）生成的数字签名。由于解锁脚本常常包含一个数字签名，因此它曾被称作 ScriptSig。

每一个比特币验证节点会通过同时执行锁定和解锁脚本来验证一笔交易。每个输入都包含一个解锁脚本，并引用了之前存在的 UTXO。

最常见类型的比特币交易（P2PKH: 对公钥哈希的付款）的解锁和锁定脚本的示例

P2PKH（Pay-to-Public-Key-Hash）

```
OP_DUP OP_HASH160 <Cafe Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

脚本中的 Cafe Public Key Hash 即为咖啡馆的比特币地址，但该地址不是基于 Base58Check 编码上述锁定脚本相应的解锁脚本是：

```
<Cafe Signature> <Cafe Public Key>
```

将两个脚本结合起来可以形成如下组合验证脚本：

```
<Cafe Signature> <Cafe Public Key> OP_DUP OP_HASH160
<Cafe Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

只有当解锁脚本得到了咖啡馆的有效签名，交易执行结果才会被通过（结果为真）

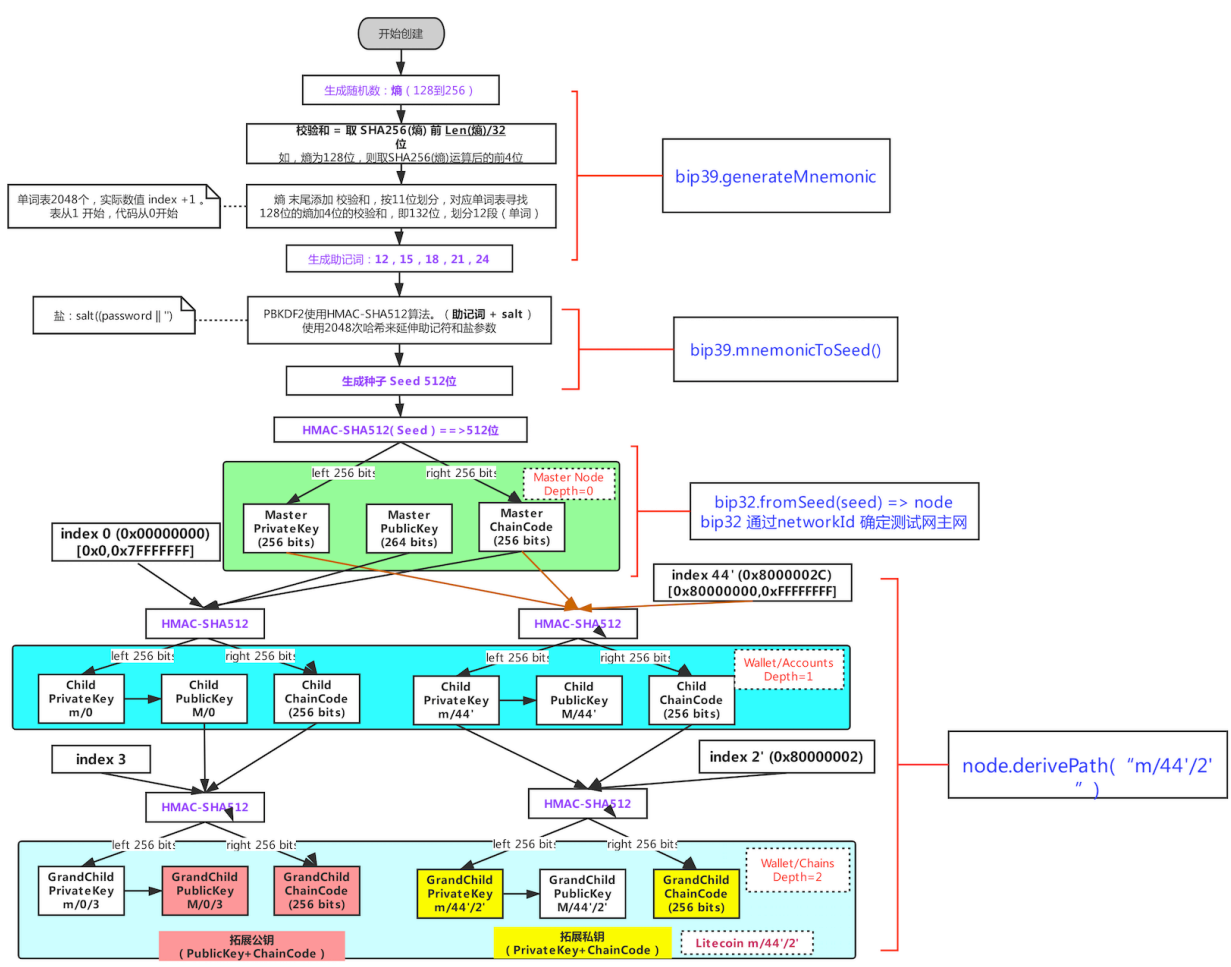
余额计算

为了构建“总接收”数量，区块链浏览器首先解码比特币地址的 Base58Check 编码，以检索编码在地址中的 Bob 的公钥的 160 位哈希值。然后，区块链浏览器搜索交易数据库，使用包含 Bob 公钥哈希的 P2PKH 锁定脚本寻找输出。通过总结所有输出的值，浏览器可以产生接收的总值。

区块链接浏览器将当前未被使用的输入保存为一个分离的数据库——UTXO 集。为了维护这个数据库，区块链浏览器必须监视比特币网络，添加新创建的 UTXO，并在已被使用的 UTXO 出现在未经确认的交易中时，实时地删除它们。这是一个复杂的过程，不但要实时地跟踪交易在网络上的传播，同时还要保持与比特币网络的共识，确保在正确的链上。有时区块链浏览器未能保持同步，导致其对 UTXO 集的跟踪扫描不完整或不正确。

3.3.5 BIP39、BIP44、BIP32 协议

流程



BIP39

熵每次都可以得到不同的助记词

```
CS = ENT /32

checksum = SHA256(entropy)

MS = (ENT + CS) / 11
```

ENT：熵长度。CS：校验长度。MS：助记词长度。checksum：校验和校验和：对熵取 SHA256 哈希结果前 CS 位

创建助记词

1. 创建一个 128 到 256 位的随机序列（熵）。
2. 提出 SHA256 哈希前几位（熵长/ 32），就可以创造一个随机序列的校验和。
3. 将校验和添加到随机序列的末尾。
4. 将序列划分为包含 11 位的不同部分。
5. 将每个包含 11 位部分的值与一个已经预先定义 2048 个单词的字典做对应。
6. 生成的有顺序的单词组就是助记码。

助记词生成种子（seed）

创建助记词之后的 7-9 步是：

1. PBKDF2 密钥延伸函数的第一个参数是从步骤 6 生成的助记符。
2. PBKDF2 密钥延伸函数的第二个参数是盐。由字符串常数“助记词”与可选的用户提供的密码字符串连接组成。
3. PBKDF2 使用 HMAC-SHA512 算法，使用 2048 次哈希来延伸助记符和盐参数，产生一个 512 位的值作为其最终输出。这个 512 位的值就是种子。

图 5-7 显示了从助记词如何生成种子

BIP32

分层路径

从 Seed 创建 HDWallet

HD 钱包从单个根种子（root seed）中创建，为 128 到 256 位的随机数。常见的通过助记词可以确定种子 Root Seed。

HD 钱包的所有的确定性都衍生自这个根种子。任何兼容 HD 钱包的根种子也可重新创造整个 HD 钱包。

- 父密钥 (没有压缩过的椭圆曲线推导的私钥或公钥 ECDSA uncompressed key)
- 链码作为熵 (chain code 256 bits)
- 子代索引序号 (index 32 bits)

钱包安全的核心在私钥，而公钥则比较容易被找到，如果子节点生成过程只依赖父节点公钥和子节点序号，那么黑客拿到父节点公钥之后就能复原出所有子节点、孙节点的公钥，这样就会破坏隐私性，CKD 里面引入的 Chain Code 则是在整个子节点派生过程中引入确定的随机数，为 HD 钱包的隐私性增加了一重保障。

正常衍生子密钥

强化衍生子密钥

- 图上标识了索引号码根据正常衍生和硬化衍生不同，索引的范围不同，对于正常衍生的索引号范围为 [0x0, 0x7FFFFFFF]，而硬化衍生的索引号范围为 [0x80000000, 0xFFFFFFFF]
- 硬化衍生的索引号太长，一般为了便于阅读，都是会将索引号右上角加上撇号，譬如：0x80000000 记为 0’，0x80000001 记为 1’，以此类推.

针对扩展密钥的学习，可以看到有三种生成规则，分别如下：

- Private parent key -> private child key 即，从父私钥和父链码计算生成子私钥和子链码。用公式表示就是：

```
CKDpriv((kpar, cpar), i) → (ki, ci)
```

- 检查是否 i ≥ 2^31(子私钥)。如果是（硬化的子密钥）：让 I= HMAC-SHA512（Key = cpar，Data = 0x00 ‖ ser256（kpar）‖ ser32（i））（注意：0x00 将私钥补齐到 33 字节长。）如果不是（普通的普通子密钥）：让 I= HMAC-SHA512（Key = cpar，Data = serP（point（kpar））‖ ser32（i））。
- 将 I 分为两个 32 字节序列，IL 和 IR。
- 返回的子密钥 ki 是 parse256（IL）+ kpar（mod n）。
- 返回的链码 ci 是 IR。
- 如果 parse256（IL）≥n 或 ki = 0，则生成的密钥无效，并且应继续下一个 i 值。（注：概率低于 1/2127）
- Public parent key -> public child key 即，从父公钥和父链码计算生成子公钥和子链码。公式表示如下：

```
CKDpub((Kpar, cpar), i) → (Ki, ci)
```

- 检查是否 i ≥ 2^31 (子密钥是否是硬化密钥) 如果是 (硬化子密钥)：返回失败如果不是 (普通子密钥): 让 I= HMAC-SHA512(Key = cpar, Data = serP(Kpar) ‖ ser32(i)).
- 将 I 分为两个 32 字节序列，IL 和 IR。
- 返回的子密钥 Ki 是 point（parse256（IL））+ Kpar。
- 返回的链码 ci 是 IR。
- 如果 parse256（IL）≥n 或 Ki 是无限远的点，则生成的密钥无效，并且应继续下一个 i 值。
- Private parent key -> public child key

  即，从父私钥和父链码计算生成子公钥和子链码。公式表示如下：

```
N((k, c)) → (K, c)
```

- 返回的密钥 K 是 point（k）。
- 返回的链码 c 只是传递的链码。

要计算父私钥的公用子密钥：

- N（CKDpriv（(kpar，cpar)，i)）（总是工作）。
- CKDpub（N（kpar，cpar），i）（仅适用于非硬化子密钥）。

它们等价的事实是使非硬化密钥有用（可以在不知道任何私钥的情况下导出给定父密钥的子公钥），以及它们与硬密钥的区别。不总是使用非硬化键（更有用）的原因是安全性; 后面可以了解更详细的信息。

参考如下:

基于 BIP-32 和 BIP-39 规范生成 HD 钱包（分层确定性钱包）分层确定性钱包 HD Wallet 剖析：设计和实现

BIP44

BIP32 路径中的 5 个层次定义:

```
m / purpose' / coin_type' / account' / change / address_index
```

路径中的撇号表示使用了经过硬化（hardern）处理的 BIP32 派生。

1. purporse': 固定值 44'(或 0x8000002C), 代表是 BIP44
2. coin_type': 这个代表的是币种, 可以兼容很多种币, 比如 BTC 是 0', ETH 是 60' 硬化的派生在这个级别上使用。(所以右上角有' 标记）币种，代表一个主节点（种子）可用于无限数量的独立加密币，如比特币等。此级别为每个加密币创建一个单独的子树，避免重用已经在其它链上存在的地址。开发人员可以为他们的项目注册未使用的号码
3. Account: 一类 coin 下能够有多个账户，就相当于你的人民币会存在多张银行卡以顺序递增的方式从索引 0 开始编号。这个数字在 BIP32 派生中用作子索引。
4. Change: 表明为外部链 0 或内部链 1：常量 0 用于外部链，常量 1 用于内部链 (也称为更改地址)。外部链用于在钱包外部可见的地址 (例如用于接收付款)。内部链用于不在钱包外部可见的地址，用于返回交易更改. **一个是用来接收地址一个是用来创造找零地址**
5. Index: 被 HD 钱包衍生的真正可用的地址是第四层级的子级，就是第五层级的树的 “address_index”。比如，第三个层级的主账户如 M/44'/0'/1' 收到比特币真正支付的地址是 M/44'/0'/0'/0/2

最后我们简要介绍一下 BIP44 中的账户发现算法:

1. 推导出第一个账户节点（生成编号初始值设为 0）
2. 推导出这个账户的外部链
3. 依次扫描外部链上的地址，如果连续二十个地址都没有交易记录，停止扫描
4. 如果外部链上没有发现交易，退出
5. 如果存在交易，将生成编号的值增加 1，跳转到第一步

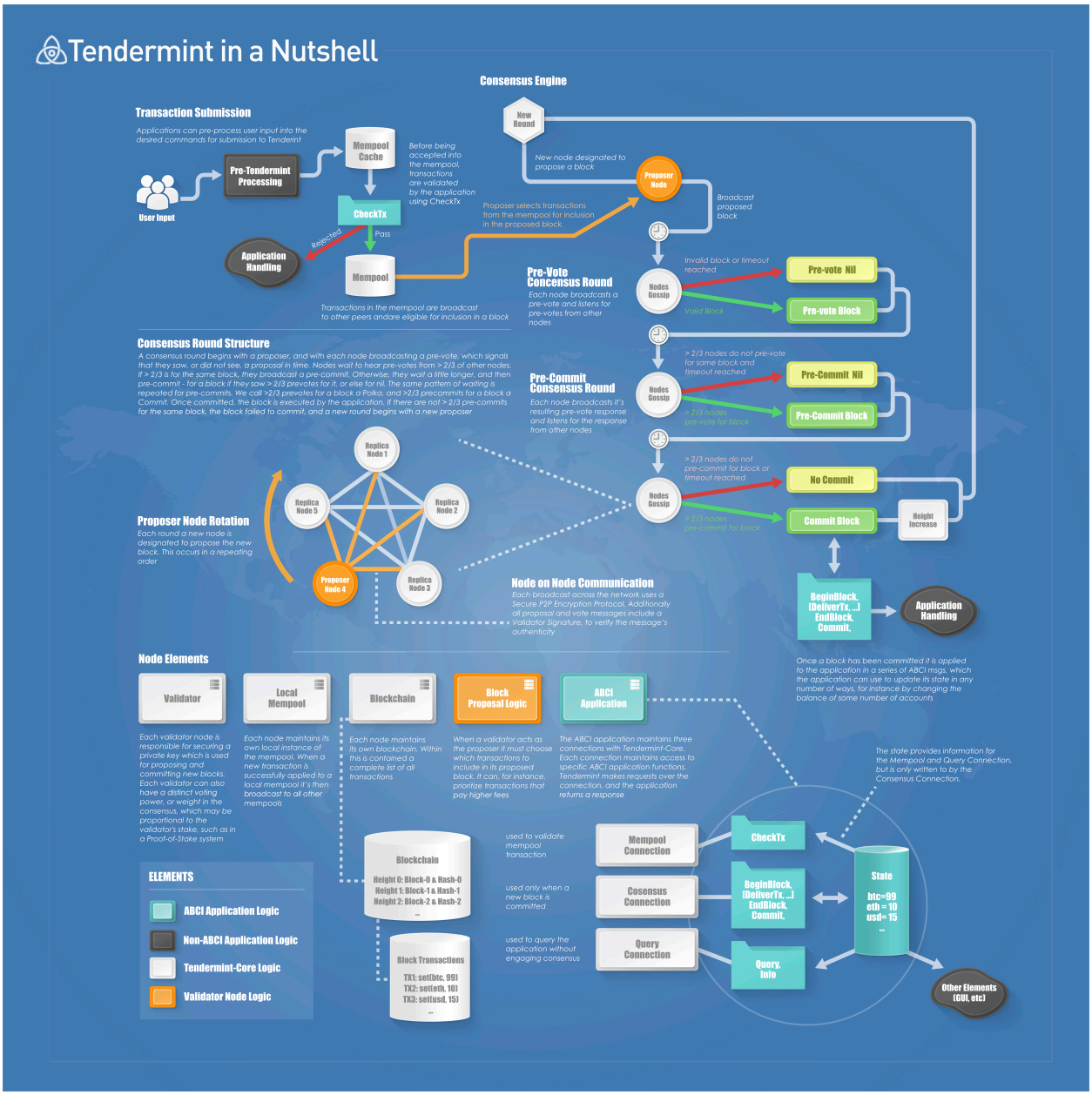
一句话概括下 BIP44 就是：给 BIP32 的分层路径定义规范

参考如下：Ethereum HD W allet（虚拟货币钱包）-BIP32、BIP39、BIP44理解开发 HD 钱包涉及的 BIP32、BIP44、BIP39

3.4 Cosmos

3.4.1 Tendermint

架构流程



client 通过 RPC 接口 broadcast_tx_commit 提交交易；

mempool 调用 ABCI 接口 CheckTx 用于校验交易的有效性，比如交易序号、发送者余额等，同时订阅交易执行后的事件并等待监听。

共识从 mempool 中获取交易开始共识排序，

协议中有两个角色：

- 验证人：协议中的角色或者节点，不同的验证者在投票过程中具备不同的权力（vote power）。
- 提议人：由验证人轮流产生。

验证人轮流对交易的区块提议并对提议的区块投票。区块被提交到链上，且每个区块就是一个区块高度。但区块也有可能提交失败，这种情况下协议将选择下一个验证人在相同高度上提议一个新块，重新开始投票。

从图中可以看到，成功提交一个区块，必须经过两阶段的投票，称为 pre-vote 和 pre-commit。当超过 2/3 的验证人在同一轮提议中对同一个块进行了 pre-commit 投票，那么这个区块才会被提交。

由于离线或者网络延迟等原因，可能造成提议人提议区块失败。这种情况在 Tendermint 中也是允许的，因为验证人会在进入下一轮提议之前等待一定时间，用于接收提议人提议的区块。

假设少于三分之一的验证人是拜占庭节点，Tendermint 能够保证验证人永远不会在同一高度重复提交区块而造成冲突。为了做到这一点，Tendermint 引入了锁定机制，一旦验证人预投票了一个区块，那么该验证人就会被锁定在这个区块。然后：

1. 该验证人必须在预提交的区块进行预投票。
2. 当前一轮预提议和预投票没成功提交区块时，该验证人就会被解锁，然后进行对新块的下一轮预提交。

Commit 阶段先执行 BeginBlock，DeliverTxAsync，EndBlock，然后再执行 Commit

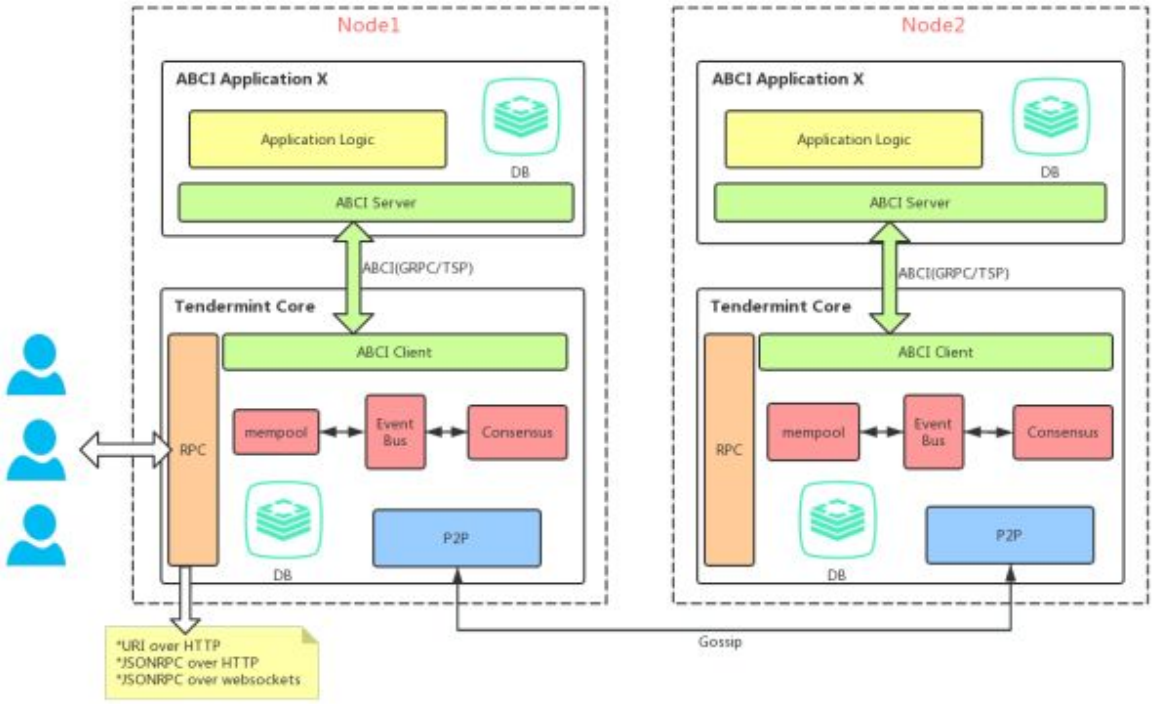
PBFT 相同点：

1. 同属 BFT 体系。
2. 抗 1/3 拜占庭节点攻击。
3. 三阶段提交，第一阶段广播交易（区块），后两阶段广播签名（确认）。
4. 两者都需要达到法定人数才能提交块。

与 PBFT 不同点：

1. Tendermint 与 PBFT 的区别主要是在超过 1/3 节点为拜占庭节点的情况下。当拜占庭节点数量在验证者数量的 1/3 和 2/3 之间时，PBFT 算法无法提供保证，使得攻击者可以将任意结果返回给客户端。而 Tendermint 共识模型认为必须超过 2/3 数量的 precommit 确认才能提交块。举个例子，如果 1/2 的验证者是拜占庭节点，Tendermint 中这些拜占庭节点能够阻止区块的提交，但他们自己也无法提交恶意块。而在 PBFT 中拜占庭节点却是可以提交块给客户端。
2. **另一个不同点在于拜占庭节点概念不同，PBFT 指的是节点数，而 Tendermint 代表的是节点的权益数，也就是投票权力。**
3. 最后一点，PBFT 需要预设一组固定的验证人，而 Tendermint 是通过要求超过 2/3 法定人数的验证人员批准会员变更，从而支持验证人的动态变化。

关于 P2P 网络



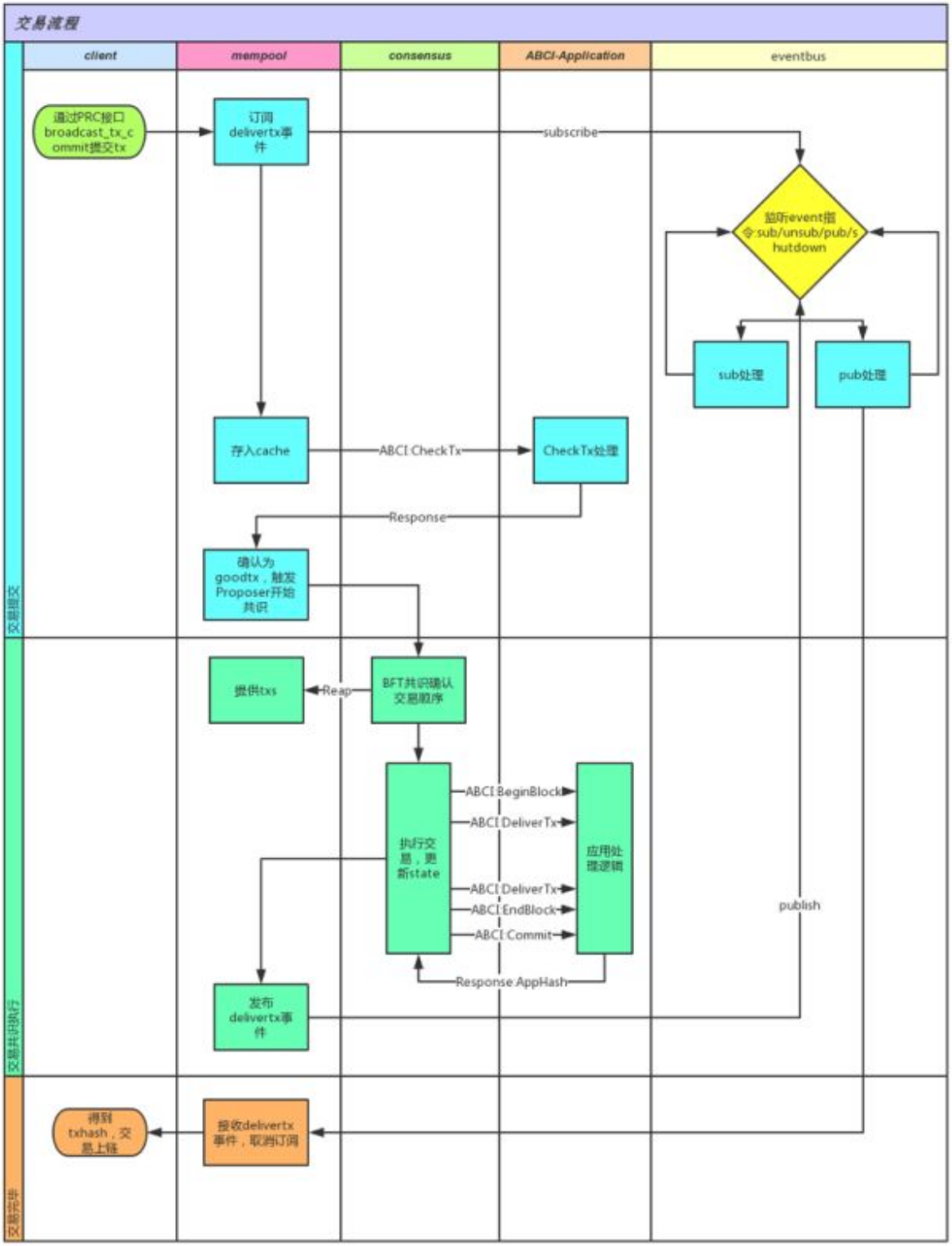
- P2P：采用 gossip 算法，默认端口是 46656。
- RPC：区块链对外接口，默认端口是 46657。支持三种访问方式：`URI over HTTP`、`JSONRPC over HTTP`、`JSONRPC over websockets`。

Tendermint 的 P2P 网络协议借鉴了比特币的对等发现协议，更准确地说，Tendermint 是采用了 BTCD 的 P2P 地址簿（Address Book）机制。当连接建立后，新节点将自身的 Address 信息（包含 IP、Port、ID 等）发送给相邻节点，相邻节点接收到信息后加入到自己的地址薄，再将此条 Address 信息，转播给它的相邻节点。

此外为了保证节点之间数据传输的安全性，Tendermint 采用了基于 Station-to-Station 协议的认证加密方案，此协议是一种密钥协商方案，基于经典的 DH 算法，并提供相互密钥和实体认证。大致的流程如下：

1. 每一个节点都必须生成一对 ED25519 密钥对作为自己的 ID。
2. 当两个节点建立起 TCP 连接时，两者都会生成一个临时的 ED25519 密钥对，并把临时公钥发给对方。
3. 两个节点分别将自己的私钥和对方的临时公钥相乘，得到共享密钥。这个共享密钥对称加密密钥。
4. 将两个临时公钥以一定规则进行排序，并将两个临时公钥拼接起来后使用 Ripemd160 进行哈希处理，后面填充 4 个 0，这样可以得到一个 24 字节的随机数。
5. 得到的随机数作为加密种子，但为了保证相同的随机数不会被相同的私钥使用两次，我们将随机数最后一个 bit 置为 1，这样就得到了两个随机数，同时约定排序更高的公钥使用反转过的随机数来加密自己的消息，而另外一个用于解密对方节点的消息。
6. 使用排序的临时公钥拼接起来，并进行 SHA256 哈希，得到一个挑战码。
7. 每个节点都使用自己的私钥对挑战码进行签名，并将自己的公钥和签名发给其它节点校验。
8. 校验通过之后，双方的认证就验证成功了。后续的通信就使用共享密钥和随机数进行加密，保护数据的安全。

Tendermint 工作流



上图简单描述了 Tenermint 的工作流。大致为：

1. client 通过 RPC 接口 broadcast_tx_commit 提交交易；
2. mempool 调用 ABCI 接口 CheckTx 用于校验交易的有效性，比如交易序号、发送者余额等，同时订阅交易执行后的事件并等待监听。
3. 共识从 mempool 中获取交易开始共识排序，打包区块，确定之后依次调用 ABCI 相关接口更新当前的世界状态，并触发事件。
4. 最终将交易信息返回 client。

ABCI 接口

ABCI 接口可以分为三类：信息查询、交易校验以及共识相关处理。而 Tendermint Core 作为 ABCI Client 在启动时，会与 ABCI Server 建立三个连接，分别用于这三类接口消息的处理。

在 Tendermint Core 与 Application 交互的所有消息类型中，有 3 种主要的消息类型：

1. CheckTx 消息用于验证交易。Tendermint Core 中的 mempool 通过此消息校验交易的合法性，通过之后才会将交易广播给其它节点。
2. DeliverTx 消息是应用的主要工作流程，通过此消息真正执行交易，包括验证交易、更新应用程序的状态。
3. Commit 消息通知应用程序计算当前的世界状态，并存在下一区块头中。

3.5 Rust

3.5.1 基础语法

实例声明

变量

- 不可变变量 `let`： 代码可读性强；每次需要创建实例，开销较大；
- 可变变量 `let mut`： 复用变量/地址空间；

常量

- 声明方式 `const`： 不允许用 `mut`；必须注明值的类型；常量只能被设置为常量表达式；

  `const MAX_POINTS: u32 = 100_000;`

```
let mut guess = String::new();
// 创建了一个可变变量，当前它绑定到一个新的 String 空实例上
```

`::new` 那一行的 `::` 语法表明 `new` 是 `String` 类型的一个 **关联函数**（*associated function*）。关联函数是针对类型实现的，在这个例子中是 `String`，而不是 `String` 的某个特定实例。一些语言中把它称为 **静态方法**（*static method*）。

`new` 函数创建了一个新的空字符串（String 类的实例），很多类型上有 `new` 函数

数据类型

标量（scalar）

复合（compound）

元组（tuple）

- 元组长度固定：一旦声明，其长度不会增大或缩小。

```
fn main() {
    //创建了一个包含多个类型元素的元组，并绑定到 tup 变量上。
    let tup = (500, 6.4, 1);

    // 接着使用了 let 和一个模式将 tup 分成了三个不同的变量，x、y 和 z。这叫做 解构
(destructuring)
    let (x, y, z) = tup;

    println!("The value of y is: {}", y);

    let tuple: (i32, f64, u8) = (500, 6.4, 1);

    // 也可以使用点号 (.) 后跟值的索引来直接访问它们
    println!("The value of x is: {}", tuple.0);
}
```

数组（array）

- 数组固定长度：一旦声明，它们的长度不能增长或缩小。

```
fn main() {
    // [元素类型; 元素数量] i32: 标量类型;  5 个元素
    let a: [i32; 5] = [1, 2, 3, 4, 5];

    // 数组是一整块分配在栈上的内存。可以使用索引来访问数组的元素
    let first = a[0];
    println!("first = {} ", first);
}
```

函数调用

- 每个参数需要注明类型，返回值用 -> 表示

```
fn main() {
    let x = plus_one(5);
    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    // return x + 1; 显式的 x + 1
    x + 1
    // x + 1; 会报错，相当于执行 return 空
}
```

循环控制

- Rust 有三种循环：loop、while 和 for

```
fn main() {
    let mut a: [i32; 4] = [0;4];
    let mut i: i32  = 1; let mut count = 0;

    // result 表达式: 会返回 loop 过程中的返回值
    let result = loop {
        if i > 10 {
            break;
        }
        i += i; a[count] = i; count += 1;
    };
    result;    // 执行刚刚的代码块

    // while 循环
    while i < 100 {
        i += i;
    }
    println!("i is {}", i);

    // for 循环遍历 :
    for element in a.iter() {
        println!("iter value is: {}", element);
    }

    // 循环遍历 rev: 反转 range
    for number in (1..4).rev() {
        println!("rev  {}!", number);
    }
}
```

3.5.2 变量与数据

所有权

所有权规则

- 1、rust 中的每个值都有一个被称为**所有者**（owner）的变量
- 2、值在任意时刻有且只有一个所有者
- 3、当**所有者**（变量）离开作用域，这个值会被丢弃

内存与分配

- `String` 类型，为了支持一个可变，可增长的文本片段，需要在堆上分配一块在编译时未知大小的内存来存放内容。这意味着：
  - 必须在运行时向操作系统请求内存。
  - 需要一个当我们处理完 `String` 时将内存返回给操作系统的方法。
- 当变量离开作用域，Rust 为我们调用一个特殊的函数。这个函数叫做 `drop`，在这里 `String` 的作者可以放置释放内存的代码。Rust 在结尾的 `}` 处自动调用 `drop`。

变量与数据交互的方式（一）：移动

- Rust 永远也不会自动创建数据的“深拷贝”。因此，任何 **自动**的复制可以被认为对运行时性能影响较小。
- `String` 会申请内存资源，复制时涉及到内存（值）的所有权转移，参见所有权规则 1

```
fn move_test(){
    // 像整型这样的在编译时已知大小的类型被整个存储在栈上
    let stack_str = "stack";
    let copy_stack = stack_str;
    println!("{}, hello!, copy {} ", stack_str, copy_stack);

    // String 类型被分配到堆上，所以能够存储在编译时未知大小的文本。
    let heap_str = String::from("heap");

    // String 实例 heap_str 中指向堆存储的指针作废，move_pointer 复制或者说移动刚刚的指针
    // 拷贝指针、长度和容量而不拷贝数据
    let move_pointer = heap_str;
    println!("{0}, hello!; move stack pointer, {0} data no change", move_pointer);
↪
}
```

变量与数据交互的方式（二）：克隆

- 会重新生成一个

```
fn clone_test(){
    let heap_str = String::from("hello");

    // clone 会深度复制 String 中堆上的数据，而不仅仅是栈上的数据
    let heap_copy = heap_str.clone();
    println!("heap_str = {}, copy = {}", heap_str, heap_copy);
}
```

只在栈上的数据：拷贝

- 任何简单标量值的组合可以是 Copy 的，不需要分配内存或某种形式资源的类型是 Copy 的
  - 所有整数类型，比如 u32。
  - 布尔类型，bool，它的值是 true 和 false。
  - 所有浮点数类型，比如 f64。
  - 字符类型，char。
  - 元组，当且仅当其包含的类型也都是 Copy 的时候。比如，(i32, i32) 是 Copy 的，但 (i32, String) 就不是。

所有权转移

- 将值赋给另一个变量时移动它。当持有堆中数据值的变量离开作用域时，其值将通过 drop 被清理掉，除非数据被移动为另一个变量所有。
- 函数参数可转移所有权：(copy) / (move)。目前来看取决于传递的类型是标量值还是占据内存资源的值
- 返回值可转移所有权

```
 // gives_ownership 将返回值 move 给调用它的函数
fn gives_ownership() -> String {
    let some_string = String::from("hello"); // some_string 进入作用域.
    some_string                              // 返回 some_string 并移出给调用的函数
}

// takes_and_gives_back 将传入字符串并返回该值
fn takes_and_gives_back(a_string: String) -> String { // a_string 进入作用域
    "change".to_owned()  // 返回并 move 给调用的函数
} // a_string 移出作用域并被丢弃。

fn makes_copy(some_integer: i32) { // some_integer 进入作用域
    println!("{}", some_integer);
} // 这里，some_integer 移出作用域。不会有特殊操作

fn ownership_test() {
    let s1 = gives_ownership();          // gives_ownership 将返回值 move 给 s1

    let s2 = String::from("hello");      // s2 进入作用域

    // s2 被 move 到函数中，它也将返回值移给 s3。
    // 根据规则二：s2 已经没有所有权，不能使用
    let s3 = takes_and_gives_back(s2);

    println!("s1 = {}, s3 = {}",s1, s3);

    let x = 5;                           // x 进入作用域

    makes_copy(x);                       // x 应该移动函数里，但 i32 是 Copy 的，所以在后面可
继续使用 x

} // 根据规则三：这里，s3 移出作用域并被丢弃。s2 也移出作用域，但已被移走，
  // 所以什么也不会发生。s1 移出作用域并被丢弃
```

引用

规则

- 在任意给定时间，**要么**只能有一个可变引用，**要么**只能有多个不可变引用。
- 引用必须总是有效的。

实例

- 可变引用/

```
fn reference_test() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);

    // 在特定作用域中的特定数据只能有一个可变引用。可以通过代码块来变通约束条件
    {
        let r1 = &mut s1;

    } // r1 在这里离开了作用域，所以我们完全可以创建一个新的引用

    let r2 = &mut s1;
}

fn calculate_length(s: &String) -> usize { // s 是对 String 的引用; 获取引用作为函数参数称为 借用 (borrowing)
    s.len()
}// 这里，s 离开了作用域。但因为它并不拥有引用值的所有权，所以什么也不会发生
```

CHAPTER 4

算法与数据结构

4.1 链表

4.1.1 实现链表的排序

使用 O(n logn) 的时间复杂度对链表进行排序

```
/* 链表排序
 * 1、每一个节点是一个有序链表，当前节点与排好序的链表进行合并
 * 2、合并两个有序链表，只需要两个当前指针进行比大小即可
 * 3、O(n logn) 需要进行分治，合并是 O(n)，分治才能 O(log n)
 * 4、两个快慢指针，快指针跳两下，慢指针跳一下，达到对半分治的效果
 */
func sortList(head *ListNode) *ListNode {
    if head == nil || head.Next == nil { return head }

        slow, fast := head, head
        var prev *ListNode

        for fast != nil && fast.Next != nil {  // fast 比 slow 快一倍，即 slow 是半点
                prev = slow
                slow = slow.Next
                fast = fast.Next.Next
        }
        prev.Next = nil

        l1 := sortList(head)
        l2 := sortList(slow)
    return mergeTwoLists(l1,l2)
}

func mergeTwoLists(l1 *ListNode, l2 *ListNode) *ListNode {
    var tmp = &ListNode{}
```

(下页继续)

(续上页)

```
    var head = tmp
    for l1!=nil && l2!=nil{
        if(l1.Val < l2.Val){
            tmp.Next = l1
            l1 = l1.Next
            tmp = tmp.Next
        }else{
            tmp.Next = l2
            l2 = l2.Next
            tmp = tmp.Next
        }
    }
    if(l1 != nil){
        tmp.Next = l1
    }else if(l2 != nil){
        tmp.Next = l2
    }
    return head.Next
}
```

4.1.2 实现高效的单向链表逆序输出

```
// 如何实现一个高效的单向链表逆序输出？
import "fmt"

type node struct {
        data int
        next *node
}

func reverse(head *node) {
        if head == nil || head.next == nil {
                return
        }
        var prev *node
        var next *node
        pcur := head
        for pcur != nil {
                if pcur.next == nil { //此时为最后一个节点
                        pcur.next = prev
                        break
                }
                next = pcur.next //取下一个节点
                pcur.next = prev // 将当前节点挂在下一个节点 (pcur) 的后面
                prev = pcur      // 当前节点
                pcur = next      //将下一个节点指给 pcur
        }
        tmp := pcur
        for tmp != nil {
                fmt.Print(tmp.data, " ")
                tmp = tmp.next
        }
}
func main() {
```

(下页继续)

(续上页)

```
        var head *node = &node{data: 0, next: nil} //指针指向资源区
        var varHead *node = head
        fmt.Println("before:")
        for i := 1; i < 6; i++ { // 构建链表
                var tmp node = node{data: i, next: nil}
                fmt.Print(tmp.data, " ")
                varHead.next = &tmp
                varHead = varHead.next
        }
        fmt.Println()
        fmt.Println("after:")
        reverse(head)
}
```

4.2 Tree

4.2.1 验证一棵树是否为二叉搜索树

```
func isValidBST(root *TreeNode) bool {
    if(root == nil) {return true}
    if(root.Left !=nil && (getRight(root.Left) >= root.Val)) {return false}
    if(root.Right !=nil && (getLeft(root.Right) <= root.Val)) {return false}
    return (isValidBST(root.Left) && isValidBST(root.Right))
}
func getRight(root *TreeNode) int {
    if(root.Right == nil){return root.Val }
    return getRight(root.Right)
}
func getLeft(root *TreeNode) int {
    if(root.Left == nil){return root.Val }
    return getLeft(root.Left)
}
```

4.2.2 平衡二叉树

```
//二叉树，其中每个节点的左和右子树的高度相差不超过 1。
func isBalanced(root *TreeNode) bool {
    if(root == nil){ return true }
    if(isBalanced(root.Left) && isBalanced(root.Right)){
        return WithBranch(getHeight(root.Left) - getHeight(root.Right)) <=1
    }
    return false
}
func getHeight(root *TreeNode) int{
    if(root == nil) { return 0 }
    l := getHeight(root.Left) +1
    r := getHeight(root.Right) +1
    if(l <= r){return r}
    return l
}
```

(下页继续)

(续上页)

```
func WithBranch(n int) int {
    if n < 0 {
        return -n
    }
    return n
}
```

CHAPTER 5

Golang 开发

5.1 基础知识

5.1.1 数组

Go 语言中数组是值语义。一个数组变量即表示整个数组，它并不是隐式的指向第一个元素的指针（比如 C 语言的数组），而是一个完整的值。当一个数组变量被赋值或者被传递的时候，实际上会复制整个数组。如果数组较大的话，数组的赋值也会有较大的开销。为了避免复制数组带来的开销，可以传递一个指向数组的指针，但是数组指针并不是数组。

```
var a = [...]int{1, 2, 3} // a 是一个数组
var b = &a                // b 是指向数组的指针

fmt.Println(a[0], a[1])   // 打印数组的前 2 个元素
fmt.Println(b[0], b[1])   // 通过数组指针访问数组元素的方式和数组类似

for i, v := range b {     // 通过数组指针迭代数组的元素
    fmt.Println(i, v)
}
```

其实数组指针类型除了类型和数组不同之外，通过数组指针操作数组的方式和通过数组本身的操作类似，而且数组指针赋值时只会拷贝一个指针。

5.1.2 字符串

Go 语言字符串的底层结构在 reflect.StringHeader 中定义：

```
type StringHeader struct {
    Data uintptr
    Len  int
}

uintptr 是 golang 的内置类型，是能存储指针的整型
```

字符串结构由两个信息组成：第一个是字符串指向的底层字节数组，第二个是字符串的字节的长度。字符串其实是一个结构体，因此字符串的赋值操作也就是 `reflect.StringHeader` 结构体的复制过程.

字符串虽然不是切片，但是支持切片操作，不同位置的切片底层也访问的同一块内存数据（因为字符串是只读的，相同的字符串面值常量通常是对应同一个字符串常量）：

5.1.3 切片

reflect.SliceHeader：

```
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}

uintptr 是 golang 的内置类型，是能存储指针的整型
```

只要是切片的底层数据指针、长度和容量没有发生变化的话，对切片的遍历、元素的读取和修改都和数组是一样的。在对切片本身赋值或参数传递时，和数组指针的操作方式类似，只是复制切片头信息（`reflect.SliceHeader`），并不会复制底层的数据。对于类型，和数组的最大不同是，切片的类型和长度信息无关，只要是相同类型元素构成的切片均对应相同的切片类型。

切片操作并不会复制底层的数据。底层的数组会被保存在内存中，直到它不再被引用。但是有时候可能会因为一个小的内存引用而导致底层整个数组处于被使用的状态，这会延迟自动内存回收器对底层数组的回收。

切片中的底层数组部分是通过隐式指针传递 (指针本身依然是传值的，但是指针指向的却是同一份的数据) 所以被调用函数是可以通过指针修改掉调用参数切片中的数据。

除了数据之外，切片结构还包含了切片长度和切片容量信息，这 2 个信息也是传值的。

如果被调用函数中修改了 Len 或 Cap 信息的话，就无法反映到调用参数的切片中，这时候我们一般会通过返回修改后的切片来更新之前的切片。这也是为何内置的 append 必须要返回一个切片的原因。

5.1.4 Make 和 New

虽然 `make` 和 `new` 都是能够用于初始化数据结构，但是它们两者能够初始化的结构类型却有着较大的不同；`make` 在 Go 语言中只能用于初始化语言中的基本类型：

```
slice := make([]int, 0, 100)
hash := make(map[int]bool, 10)
ch := make(chan int, 5)
```

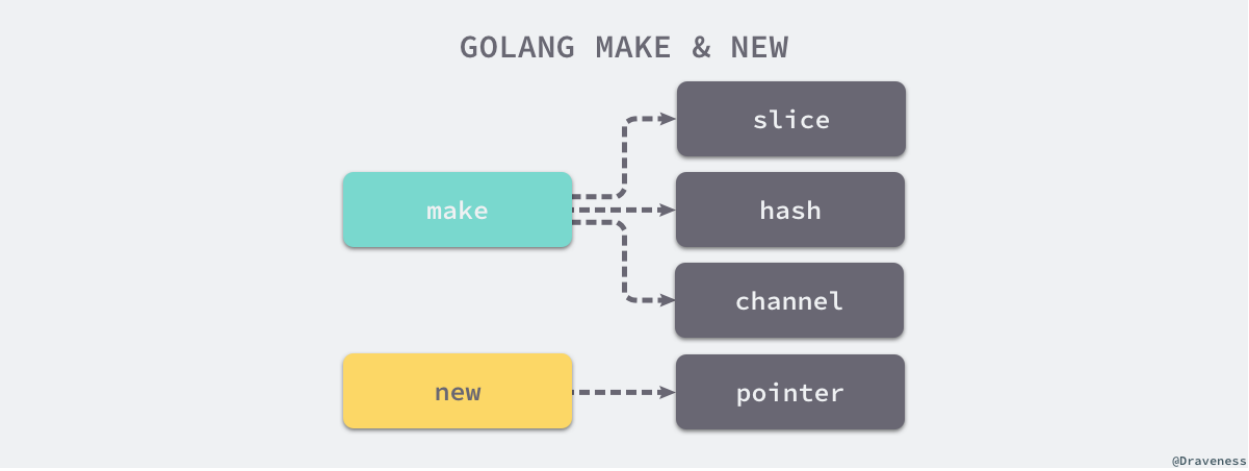
- `slice` 是一个包含 `data`、`cap` 和 `len` 的**结构体**；
- `hash` 是一个指向 `hmap` 结构体的**指针**；

- ch 是一个指向 hchan 结构体的**指针**；

而另一个用于初始化数据结构的关键字 new 的作用其实就非常简单了，它只是接收一个类型作为参数然后返回一个指向这个类型的指针：

```
i := new(int)
var v int
i := &v
```

上述代码片段中的两种不同初始化方法其实是等价的，它们都会创建一个指向 int 零值的指针。

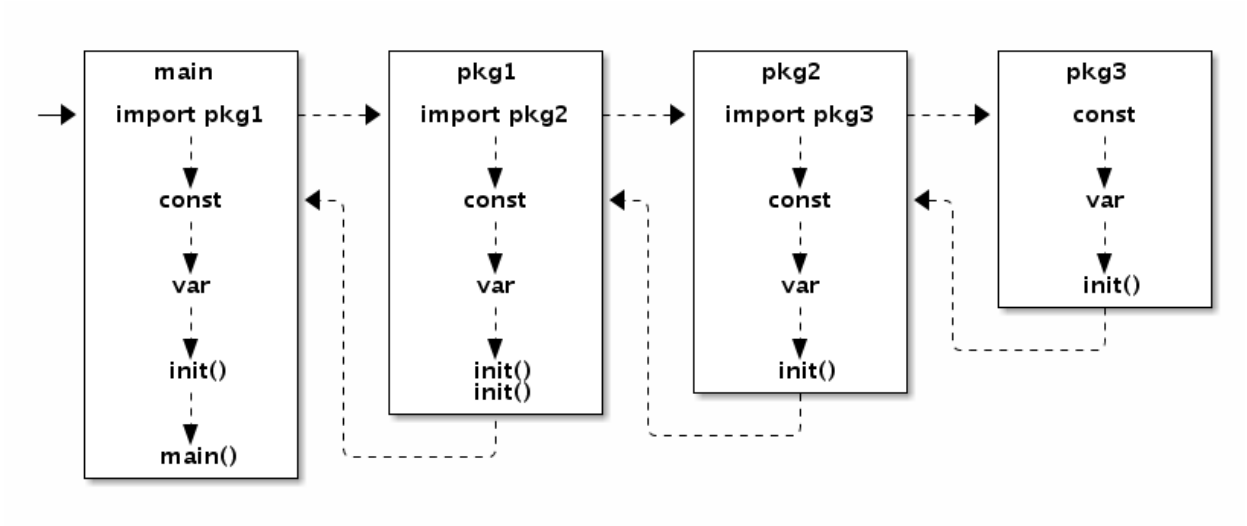


到了这里我们对 Go 语言中这两种不同关键字的使用也有了一定的了解：make 用于创建切片、哈希表和管道等内置数据结构，new 用于分配并创建一个指向对应类型的指针。

5.1.5 函数初始化

Go 语言中的函数有具名和匿名之分：具名函数一般对应于包级的函数，是匿名函数的一种特例，当匿名函数引用了外部作用域中的变量时就成了闭包函数，闭包函数是函数式编程语言的核心。

Go 语言程序的初始化和执行总是从 main.main 函数开始的。但是如果 main 包导入了其它的包，则会按照顺序将它们包含进 main 包里（这里的导入顺序依赖具体实现，一般可能是以文件名或包路径名的字符串顺序导入）。如果某个包被多次导入的话，在执行的时候只会导入一次。当一个包被导入时，如果它还导入了其它的包，则先将其它的包包含进来，然后创建和初始化这个包的常量和变量，再调用包里的 init 函数，如果一个包有多个 init 函数的话，调用顺序未定义 (实现可能是以文件名的顺序调用)，同一个文件内的多个 init 则是以出现的顺序依次调用（init 不是普通函数，可以定义有多个，所以也不能被其它函数调用）。最后，当 main 包的所有包级常量、变量被创建和初始化完成，并且 init 函数被执行后，才会进入 main.main 函数，程序开始正常执行。



5.1.6 函数调用

Go 语言中对于整型和数组类型的参数都是值传递的，也就是在调用函数时会对内容进行拷贝，需要注意的是如果当前数组的大小非常的大，这种直接复制传值的方式就会对性能造成比较大的影响。

函数可以直接或间接地调用自己，也就是支持递归调用。Go 语言函数的递归调用深度逻辑上没有限制，函数调用的栈是不会出现溢出错误的，因为 Go 语言运行时会根据需要动态地调整函数栈的大小。每个 goroutine 刚启动时只会分配很小的栈（4 或 8KB，具体依赖实现），根据需要动态调整栈的大小，栈最大可以达到 GB 级（依赖具体实现，在目前的实现中，32 位体系结构为 250MB,64 位体系结构为 1GB）

```
func f(x int) *int {
    return &x
}

func g() int {
    x = new(int)
    return *x
}
```

第一个函数直接返回了函数参数变量的地址——这似乎是不可以的，因为如果参数变量在栈上的话，函数返回之后栈变量就失效了，返回的地址自然也应该失效了。但是 Go 语言的编译器和运行时比我们聪明的多，它会保证指针指向的变量在合适的地方。第二个函数，内部虽然调用 new 函数创建了 *int 类型的指针对象，但是依然不知道它具体保存在哪里。

任何可以通过函数参数修改调用参数的情形，都是因为函数参数中显式或隐式传入了指针参数。

5.1.7 方法

Go 语言的方法却是关联到类型的，这样可以在编译阶段完成方法的静态绑定。一个面向对象的程序会用方法来表达其属性对应的操作，这样使用这个对象的用户就不需要直接去操作对象，而是借助方法来做这些事情。

对于给定的类型，每个方法的名字必须是唯一的，同时方法和函数一样也不支持重载。

5.1.8 接口

Go 的接口类型是对其它类型行为的抽象和概括；因为接口类型不会和特定的实现细节绑定在一起。Go 语言的接口类型是延迟绑定，可以实现类似虚函数的多态功能。

Go 语言对于接口类型的转换则非常的灵活。对象和接口之间的转换、接口和接口之间的转换都可能是隐式的转换。

```
var (
    a io.ReadCloser = (*os.File)(f) // 隐式转换, *os.File 满足 io.ReadCloser 接口
    b io.Reader     = a             // 隐式转换, io.ReadCloser 满足 io.Reader 接口
    c io.Closer     = a             // 隐式转换, io.ReadCloser 满足 io.Closer 接口
    d io.Reader     = c.(io.Reader) // 显式转换, io.Closer 不满足 io.Reader 接口
)
```

interface 应用场景

类型转换

类型推断可将接口变量还原为原始类型，或用来判断是否实现了某个更具体的接口类型。

```
type data int

func(d data)String()string{
   return fmt.Sprintf("data:%d",d)
}

func main() {
   var d data=15
   var x interface{} =d

   if n,ok:=x.(fmt.Stringer);ok{  // 转换为更具体的接口类型
       fmt.Println(n)
    }

   if d2,ok:=x.(data);ok{         // 转换回原始类型
       fmt.Println(d2)
    }

   e:=x.(error)            // 错误:main.data is not error
   fmt.Println(e)
}

data:15
data:15
panic:interface conversion:main.data is not error:missing method Error
```

实现多态功能

多态功能是 interface 实现的重要功能，也是 Golang 中的一大行为特色，其多态功能一般要结合 Go method 实现，作为函数参数可以容易的实现多台功能。

5.1.9 指针和引用

结构体和指针在参数的传递过程中：如果传递的是结构体，那么在传递参数时依然会对结构体中的全部内容进行拷贝，而传递指针时复制的其实也是指针的内容 - 地址

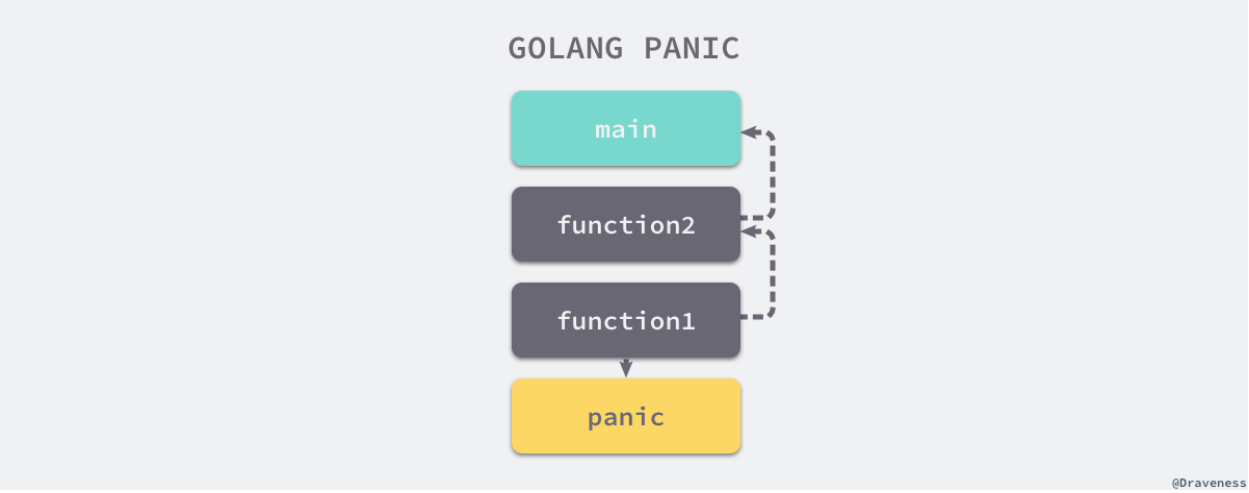
5.1.10 函数调用

Go 通过堆栈的方式对函数的参数和返回值进行传递和接受，在调用函数之前会在栈上为返回值分配合适的内存空间，随后按照入参从右到左按顺序压栈，被调用方接受参数时会对参数进行拷贝后再进行计算，返回值最终会被放置到调用者预留好的栈空间上，Go 语言函数调用的原理可以总结成以下的几条规则：

- 1、通过堆栈传递参数，入栈的顺序是从右到左；
- 2、函数返回值通过堆栈传递并由调用者预先分配内存空间；
- 3、调用函数时都是传值，接收方会对入参进行复制再计算；

5.1.11 panic 和 recover

`panic` 和 `recover` 两个关键字其实都是 Go 语言中的内置函数，`panic` 能够改变程序的控制流，当一个函数调用执行 `panic` 时，它会立刻停止执行函数中其他的代码，而是会运行其中的 `defer` 函数，执行成功后会返回到调用方。



对于上层调用方来说，调用导致 `panic` 的函数其实与直接调用 `panic` 类似，所以也会执行所有的 `defer` 函数并返回到它的调用方，这个过程会一直进行直到当前 Goroutine 的调用栈中不包含任何的函数，这时整个程序才会崩溃，这个『恐慌过程』不仅会被显式的调用触发，还会由于运行期间发生错误而触发。

然而 `panic` 导致的『恐慌』状态其实可以被 **defer 中的** `recover` 中止，`recover` 是一个只在 `defer` 中能够发挥作用的函数，在正常的控制流程中，调用 `recover` 会直接返回 `nil` 并且没有任何的作用，但是如果当前的 Goroutine 发生了『恐慌』，`recover` 其实就能够捕获到 `panic` 抛出的错误并阻止『恐慌』的继续传播

5.1.12 参考资料:

golang 语言实现

5.2 开发常用

5.2.1 context 包

Go 语言中的每一个请求的都是通过一个单独的 Goroutine 进行处理的，HTTP/RPC 请求的处理器往往都会启动新的 Goroutine 访问数据库和 RPC 服务，我们可能会创建多个 Goroutine 来处理一次请求，而 Context 的主要作用就是在不同的 Goroutine 之间同步请求特定的数据、取消信号以及处理请求的截止日期。

Context 结构

```
// A Context carries a deadline, cancelation signal, and request-scoped values
// across API boundaries. Its methods are safe for simultaneous use by multiple
// goroutines.
type Context interface {
    // Done returns a channel that is closed when this Context is canceled
    // or times out.
    Done() <-chan struct{}

    // Err indicates why this context was canceled, after the Done channel
    // is closed.
    Err() error

    // Deadline returns the time when this Context will be canceled, if any.
    Deadline() (deadline time.Time, ok bool)

    // Value returns the value associated with key or nil if none.
    Value(key interface{}) interface{}
}
```

- Deadline 方法需要返回当前 Context 被取消的时间，也就是完成工作的截止日期；
- Done 方法需要返回一个 Channel，这个 Channel 会在当前工作完成或者上下文被取消之后关闭，多次调用 Done 方法会返回同一个 Channel；
- Err 方法会返回当前 Context 结束的原因，它只会在 Done 返回的 Channel 被关闭时才会返回非空的值；
  - 如果当前 Context 被取消就会返回 Canceled 错误；
  - 如果当前 Context 超时就会返回 DeadlineExceeded 错误；
- Value 方法会从 Context 中返回键对应的值，对于同一个上下文来说，多次调用 Value 并传入相同的 Key 会返回相同的结果，这个功能可以用来传递请求特定的数据；

Context 的实现方法

根 context

Context 虽然是个接口，但是并不需要使用方实现：Background() TODO

```
var (
        background = new(emptyCtx)
        todo = new(emptyCtx)
)
func Background() Context {
  //主要用于 main 函数、初始化以及测试代码中，作为 Context 这个树结构的最顶层的 Context，也就是根
Context，它不能被取消
        return background
}
func TODO() Context { //只在不确定时使用 context.TODO()
        return todo
}
```

一般在代码中，开始上下文的时候都是以这两个作为最顶层的 parent context，然后再衍生出子 context。这些 Context 对象形成一棵树：当一个 Context 对象被取消时，继承自它的所有 Context 都会被取消。

继承 context

一般情况下是以 context.Background() 做为根节点

```
//传递一个父 Context 作为参数，返回子 Context，以及一个取消函数用来取消 Context。
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)

//会多传递一个截止时间参数，意味着到了这个时间点，会自动取消 Context，也可以提前通过取消函数进行取消。
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)

//超时自动取消，是多少时间后自动取消 Context 的意思。也可以提前通过取消函数进行取消。
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)

// 为了生成一个绑定了一个键值对数据的 Context，这个绑定的数据可以通过 Context.Value 方法访问到，这
是我们实际用经常要用到的技巧，一般我们想要通过上下文来传递数据时，可以通过这个方法，如我们需要 tarce 追
踪系统调用栈的时候
func WithValue(parent Context, key, val interface{}) Context
```

参考如下

golang 服务器开发利器 context 用法详解

Golang Context 深入理解

5.2.2 定时器

time.Timer

Timer 的结构定义

```
type Timer struct {
    C <-chan Time
    r runtimeTimer
}
```

其他方法

- func After(d Duration) <-chan Time { return NewTimer(d).C }

根据源码可以看到 After 直接是返回了 Timer 的 channel，这种就可以做超时处理。比如我们有这样一个需求：我们写了一个爬虫，爬虫在 HTTP GET 一个网页的时候可能因为网络的原因就一只等待着，这时候就需要做超时处理，比如只请求五秒，五秒以后直接丢掉不请求这个网页了，或者重新发起请求。

```
Get("http://baidu.com/")
func Get(url string) {
    response := make(chan string)
    response = http.Request(url)
    select {
    case html :=<- response:
        println(html)
    case <-time.After(time.Second * 5):
        println("超时处理")
    }
}
```

可以从代码中体现出来，如果五秒到了，网页的请求还没有下来就是执行超时处理，因为 Timer 的内部会是帮你在你设置的时间长度后自动向 Timer.C 中写入当前时间。

其实也可以写成这样：

```
func Get(url string) {
    response := make(chan string)
    response = http.Request(url)
    timeOut := time.NewTimer(time.Second * 3)
    select {
    case html :=<- response:
        println(html)
    case <-timeOut.C:
        println("超时处理")
    }
}
```

- func (t *Timer) Reset(d Duration) bool//强制的修改 timer 中规定的时间，Reset 会先调用 stopTimer 再调用 startTimer，类似于废弃之前的定时器，重新启动一个定时器，Reset 在 Timer 还未触发时返回 true；触发了或 Stop 了，返回 false。
- func (t *Timer) Stop() bool// 如果定时器还未触发，Stop 会将其移除，并返回 true；否则返回 false；后续再对该 Timer 调用 Stop，直接返回 false。
- func AfterFunc(d Duration, f func()) *Timer // 在时间 d 后自动执行函数 f

```
func main() {
    f := func(){fmt.Println("I Love You!")}
    time.AfterFunc(time.Second*2, f)
    time.Sleep(time.Second * 4)

}
```

time.Ticker

其实 Ticker 就是一个重复版本的 Timer，它会重复的在时间 d 后向 Ticker 中写数据

- func NewTicker(d Duration) *Ticker // 新建一个 Ticker
- func (t *Ticker) Stop() // 停止 Ticker
- func Tick(d Duration) <-chan Time // Ticker.C 的封装

参考

Golang time.Timer and time.Ticker

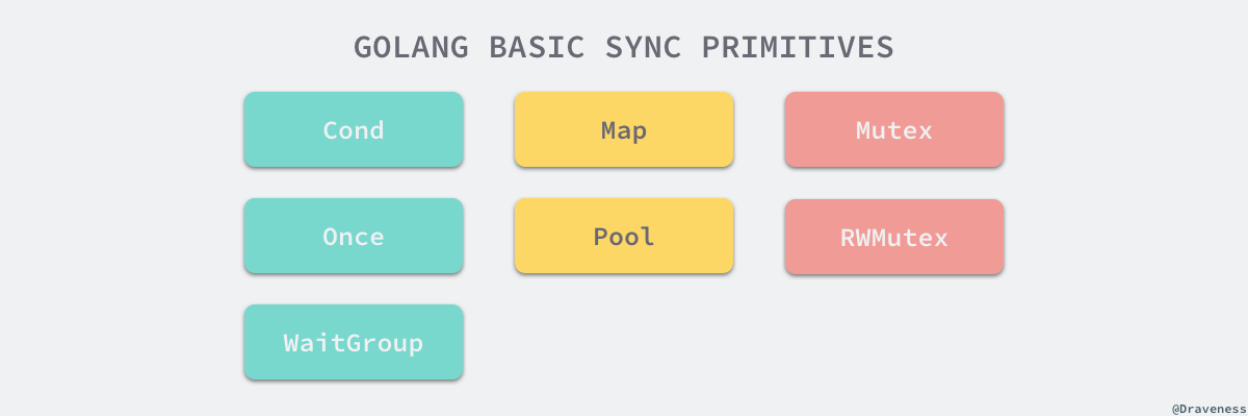
5.3 同步原语与锁

当提到并发编程、多线程编程时，我们往往都离不开『锁』这一概念，Go 语言作为一个原生支持用户态进程 Goroutine 的语言，也一定会为开发者提供这一功能，锁的主要作用就是保证多个线程或者 Goroutine 在访问同一片内存时不会出现混乱的问题，锁其实是一种并发编程中的同步原语（Synchronization Primitives）。

在这一节中我们就会介绍 Go 语言中常见的同步原语 Mutex、RWMutex、WaitGroup、Once 和 Cond 以及扩展原语 ErrGroup、Semaphore 和 SingleFlight 的实现原理，同时也会涉及互斥锁、信号量等并发编程中的常见概念。

5.3.1 1. 基本原语

Go 语言在 sync 包中提供了用于同步的一些基本原语，包括常见的互斥锁 Mutex 与读写互斥锁 RWMutex 以及 Once、WaitGroup。



这些基本原语的主要作用是提供较为基础的同步功能，我们应该使用 Channel 和通信来实现更加高级的同步机制，我们在这一节中并不会介绍标准库中全部的原语，而是会介绍其中比较常见的 Mutex、RWMutex、Once、WaitGroup 和 Cond，我们并不会涉及剩下两个用于存取数据的结构体 Map 和 Pool

1.1. Mutex

Go 语言中的互斥锁在 sync 包中，它由两个字段 state 和 sema 组成，state 表示当前互斥锁的状态，而 sema 真正用于控制锁状态的信号量，这两个加起来只占 8 个字节空间的结构体就表示了 Go 语言中的互斥锁。

```
type Mutex struct {
    state int32
    sema  uint32
}
```

状态

互斥锁的状态是用 int32 来表示的，但是锁的 state 并不是互斥的，它的最低三位分别表示 mutexLocked、mutexWoken 和 mutexStarving，剩下的位置都用来表示当前有多少个 Goroutine 等待互斥锁被释放：

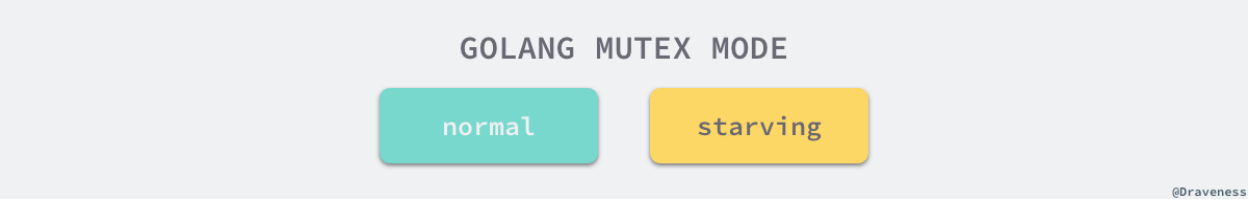

```
|-----------------|---------------| -----------| -----------|
| waitGoroutines  | mutexStarving | mutexWoken | mutexLocked|
```

互斥锁在被创建出来时，所有的状态位的默认值都是 0，当互斥锁被锁定时 mutexLocked 就会被置成 1、当互斥锁被在正常模式下被唤醒时 mutexWoken 就会被被置成 1、mutexStarving 用于表示当前的互斥锁进入了状态，最后的几位是在当前互斥锁上等待的 Goroutine 个数。

饥饿模式

Mutex 在可能会进入饥饿模式，饥饿模式主要功能就是保证互斥锁的获取的『公平性』(Fairness)。

互斥锁可以同时处于两种不同的模式，也就是正常模式和饥饿模式，在正常模式下，所有锁的等待者都会按照先进先出的顺序获取锁，但是如果一个刚刚被唤起的 Goroutine 遇到了新的 Goroutine 进程也调用了 Lock 方法时，大概率会获取不到锁，为了减少这种情况的出现，防止 Goroutine 被『饿死』，一旦 Goroutine 超过 1ms 没有获取到锁，它就会将当前互斥锁切换饥饿模式。



在饥饿模式中，互斥锁会被直接交给等待队列最前面的 Goroutine，新的 Goroutine 在这时不能获取锁、也不会进入自旋的状态，它们只会在队列的末尾等待，如果一个 Goroutine 获得了互斥锁并且它是队列中最末尾的协程或者它等待的时间少于 1ms，那么当前的互斥锁就会被切换回正常模式。

相比于饥饿模式，正常模式下的互斥锁能够提供更好地性能，饥饿模式的主要作用就是避免一些 Goroutine 由于陷入等待无法获取锁而造成较高的尾延时，这也是对 `Mutex` 的一个优化。

加锁

互斥锁 `Mutex` 的加锁是靠 `Lock` 方法完成的，最新的 Go 语言源代码中已经将 `Lock` 方法进行了简化，方法的主干只保留了最常见、简单并且快速的情况；当锁的状态是 `0` 时直接将 `mutexLocked` 位置成 `1`：

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        return
    }
    m.lockSlow()
}
```

5.4 RPC

RPC（Remote Procedure Call），即远程过程调用，是一个分布式系统间通信的必备技术。RPC 最核心要解决的问题就是在**分布式系统间，如何执行另外一个地址空间上的函数、方法**，就仿佛在本地调用一样。

rpc 是远端过程调用，其调用协议通常包含传输协议和编码协议

5.4.1 传输（Transport）

TCP 协议是 RPC 的基石，一般来说通信是建立在 TCP 协议之上的，而且 RPC 往往需要可靠的通信，因此不采用 UDP。**RPC 传输的 message 也就是 TCP body 中的数据**，这个 message 也同样可以包含 header+body。body 也经常叫做 payload。TCP 协议栈存在端口的概念，端口是进程获取数据的渠道。

5.4.2 I/O 模型（I/O Model）

做一个高性能 /scalable 的 RPC，需要能够满足：

- 服务端尽可能多的处理并发请求
- 同时尽可能短的处理完毕。

Socket I/O 可以看做是二者之间的桥梁，如何更好地协调二者，去满足前面说的两点要求，有一些模式（pattern）是可以应用的。RPC 框架可选择的 I/O 模型严格意义上有 5 种，这里不讨论基于信号驱动的 I/O（Signal Driven I/O）。它们分别是：

- 传统的阻塞 I/O（Blocking I/O）
- 非阻塞 I/O（Non-blocking I/O）
- I/O 多路复用（I/O multiplexing）
- 异步 I/O（Asynchronous I/O）

5.4.3 参考如下:

理解 REST 和 RPC

RPC

Go 语言的 RPC 框架有两个比较有特色的设计:

- 一个是 RPC 数据打包时可以通过插件实现自定义的编码和解码;
- 另一个是 RPC 建立在抽象的 io.ReadWriteCloser 接口之上的

5.5 Protobuf3

5.5.1 [转]Protobuf3 语法指南](https://colobu.com/2017/03/16/Protobuf3-language-guide/)

> 语言指南 [译]Protobuf 语法指南 中文出处是 proto2 的译文，proto3 的英文出现后在原来基础上增改了，水平有限，还请指正

这个指南描述了如何使用 Protocol buffer 语言去描述你的 protocol buffer 数据，包括.proto 文件符号和如何从.proto 文件生成类。包含了 proto2 版本的 protocol buffer 语言：对于老版本的 proto3 符号，请见Proto2 Language Guide（以及中文译本，抄了很多这里的感谢下老版本的翻译者）

本文是一个参考指南——如果要查看如何使用本文中描述的多个特性的循序渐进的例子，请在教程中查找需要的语言的教程。

5.5.2 定义一个消息类型

先来看一个非常简单的例子。假设你想定义一个“搜索请求”的消息格式，每一个请求含有一个查询字符串、你感兴趣的查询结果所在的页数，以及每一页多少条查询结果。可以采用如下的方式来定义消息类型的.proto 文件了:

```
syntax = "proto3";
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

- 文件的第一行指定了你正在使用 proto3 语法：如果你没有指定这个，编译器会使用 proto2。这个指定语法行必须是文件的非空非注释的第一个行。
- SearchRequest 消息格式有 3 个字段，在消息中承载的数据分别对应于每一个字段。其中每个字段都有一个名字和一种类型。

指定字段类型

在上面的例子中，所有字段都是标量类型：两个整型（page_number 和 result_per_page），一个 string 类型（query）。当然，你也可以为字段指定其他的合成类型，包括枚举（enumerations）或其他消息类型。

分配标识号

正如你所见，在消息定义中，每个字段都有唯一的一个数字标识符。这些标识符是用来在消息的二进制格式中识别各个字段的，一旦开始使用就不能够再改变。注：[1,15] 之内的标识号在编码的时候会占用一个字节。[16,2047] 之内的标识号则占用 2 个字节。所以应该为那些频繁出现的消息元素保留 [1,15] 之内的标识号。切记：要为将来有可能添加的、频繁出现的标识号预留一些标识号。

最小的标识号可以从 1 开始，最大到 2^29 - 1, or 536,870,911。不可以使用其中的 [19000 - 19999]（(从 FieldDescriptor::kFirstReservedNumber 到 FieldDescriptor::kLastReservedNumber)）的标识号，Protobuf 协议实现中对这些进行了预留。如果非要在.proto 文件中使用这些预留标识号，编译时就会报警。同样你也不能使用早期保留的标识号。

指定字段规则

所指定的消息字段修饰符必须是如下之一：

- singular：一个格式良好的消息应该有 0 个或者 1 个这种字段（但是不能超过 1 个）。
- repeated：在一个格式良好的消息中，这种字段可以重复任意多次（包括 0 次）。重复的值的顺序会被保留。

在 proto3 中，repeated 的标量域默认情况虾使用 packed。

你可以了解更多的 pakced 属性在Protocol Buffer 编码

添加更多消息类型

在一个.proto 文件中可以定义多个消息类型。在定义多个相关的消息的时候，这一点特别有用——例如，如果想定义与 SearchResponse 消息类型对应的回复消息格式的话，你可以将它添加到相同的.proto 文件中，如：

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
message SearchResponse {
 ...
}
```

添加注释

向.proto 文件添加注释，可以使用 C/C++/Java 风格的双斜杠（//）语法格式，如：

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;  // Which page number do we want?
  int32 result_per_page = 3;  // Number of results to return per page.
}
```

保留标识符（Reserved）

如果你通过删除或者注释所有域，以后的用户在更新这个类型的时候可能重用这些标识号。如果你使用旧版本加载相同的.proto 文件会导致严重的问题，包括数据损坏、隐私错误等等。现在有一种确保不会发生这种情况的方法就是为字段 tag（reserved name 可能会 JSON 序列化的问题）指定 `reserved` 标识符，protocol buffer 的编译器会警告未来尝试使用这些域标识符的用户。

```
message Foo {
  reserved 2, 15, 9 to 11;
  reserved "foo", "bar";
}
```

注：不要在同一行 reserved 声明中同时声明域名字和 tag number。

从.proto 文件生成了什么？

当用 protocol buffer 编译器来运行.proto 文件时，编译器将生成所选择语言的代码，这些代码可以操作在.proto 文件中定义的消息类型，包括获取、设置字段值，将消息序列化到一个输出流中，以及从一个输入流中解析消息。

- 对 C++ 来说，编译器会为每个.proto 文件生成一个.h 文件和一个.cc 文件，.proto 文件中的每一个消息有一个对应的类。
- 对 Java 来说，编译器为每一个消息类型生成了一个.java 文件，以及一个特殊的 Builder 类（该类是用来创建消息类接口的）。
- 对 Python 来说，有点不太一样——Python 编译器为.proto 文件中的每个消息类型生成一个含有静态描述符的模块，，该模块与一个元类（metaclass）在运行时（runtime）被用来创建所需的 Python 数据访问类。
- 对 go 来说，编译器会位每个消息类型生成了一个.pd.go 文件。
- 对于 Ruby 来说，编译器会为每个消息类型生成了一个.rb 文件。
- javaNano 来说，编译器输出类似域 java 但是没有 Builder 类
- 对于 Objective-C 来说，编译器会为每个消息类型生成了一个 pbobjc.h 文件和 pbobjcm 文件，.proto 文件中的每一个消息有一个对应的类。
- 对于 C# 来说，编译器会为每个消息类型生成了一个.cs 文件，.proto 文件中的每一个消息有一个对应的类。你可以从如下的文档链接中获取每种语言更多 API(proto3 版本的内容很快就公布)。API Reference

5.5.3 标量数值类型

一个标量消息字段可以含有一个如下的类型——该表格展示了定义于.proto 文件中的类型，以及与之对应的、在自动生成的访问类中定义的类型：

你可以在文章Protocol Buffer 编码中，找到更多“序列化消息时各种类型如何编码”的信息。

1. 在 java 中，无符号 32 位和 64 位整型被表示成他们的整型对应形式，最高位被储存在标志位中。
2. 对于所有的情况，设定值会执行类型检查以确保此值是有效。
3. 64 位或者无符号 32 位整型在解码时被表示成为 ilong，但是在设置时可以使用 int 型值设定，在所有的情况下，值必须符合其设置其类型的要求。
4. python 中 string 被表示成在解码时表示成 unicode。但是一个 ASCIIstring 可以被表示成 str 类型。
5. Integer 在 64 位的机器上使用，string 在 32 位机器上使用

5.5.4 默认值

当一个消息被解析的时候，如果被编码的信息不包含一个特定的 singular 元素，被解析的对象锁对应的域被设置位一个默认值，对于不同类型指定如下：

- 对于 string，默认是一个空 string
- 对于 bytes，默认是一个空的 bytes
- 对于 bool，默认是 false
- 对于数值类型，默认是 0
- 对于枚举，默认是第一个定义的枚举值，必须为 0;
- 对于消息类型（message），域没有被设置，确切的消息是根据语言确定的，详见 generated code guide

对于可重复域的默认值是空（通常情况下是对应语言中空列表）。

注：对于标量消息域，一旦消息被解析，就无法判断域释放被设置为默认值（例如，例如 *boolean* 值是否被设置为 *false*）还是根本没有被设置。你应该在定义你的消息类型时非常注意。例如，比如你不应该定义 *boolean* 的默认值 *false* 作为任何行为的触发方式。也应该注意如果一个标量消息域被设置为标志位，这个值不应该被序列化传输。

查看generated code guide选择你的语言的默认值的工作细节。

5.5.5 枚举

当需要定义一个消息类型的时候，可能想为一个字段指定某 “预定义值序列” 中的一个值。例如，假设要为每一个 SearchRequest 消息添加一个 corpus 字段，而 corpus 的值可能是 UNIVERSAL，WEB，IMAGES，LOCAL，NEWS，PRODUCTS 或 VIDEO 中的一个。其实可以很容易地实现这一点：通过向消息定义中添加一个枚举（enum）并且为每个可能的值定义一个常量就可以了。

在下面的例子中，在消息格式中添加了一个叫做 Corpus 的枚举类型——它含有所有可能的值——以及一个类型为 Corpus 的字段：

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
  enum Corpus {
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 6;
  }
  Corpus corpus = 4;
}
```

如你所见，Corpus 枚举的第一个常量映射为 0：每个枚举类型必须将其第一个类型映射为 0，这是因为：

- 必须有有一个 0 值，我们可以用这个 0 值作为默认值。
- 这个零值必须为第一个元素，为了兼容 proto2 语义，枚举类的第一个值总是默认值。

你可以通过将不同的枚举常量指定位相同的值。如果这样做你需要将 allow_alias 设定位 true，否则编译器会在别名的地方产生一个错误信息。

```
enum EnumAllowingAlias {
  option allow_alias = true;
  UNKNOWN = 0;
  STARTED = 1;
  RUNNING = 1;
}
enum EnumNotAllowingAlias {
  UNKNOWN = 0;
  STARTED = 1;
  // RUNNING = 1;  // Uncommenting this line will cause a compile error inside Google
↪and a warning message outside.
}
```

枚举常量必须在 32 位整型值的范围内。因为 enum 值是使用可变编码方式的，对负数不够高效，因此不推荐在 enum 中使用负数。如上例所示，可以在一个消息定义的内部或外部定义枚举——这些枚举可以在.proto 文件中的任何消息定义里重用。当然也可以在一个消息中声明一个枚举类型，而在另一个不同的消息中使用它——采用 MessageType.EnumType 的语法格式。

当对一个使用了枚举的.proto 文件运行 protocol buffer 编译器的时候，生成的代码中将有一个对应的 enum（对 Java 或 C++ 来说），或者一个特殊的 EnumDescriptor 类（对 Python 来说），它被用来在运行时生成的类中创建一系列的整型值符号常量（symbolic constants）。

在反序列化的过程中，无法识别的枚举值会被保存在消息中，虽然这种表示方式需要依据所使用语言而定。在那些支持开放枚举类型超出指定范围之外的语言中（例如 C++ 和 Go），为识别的值会被表示成所支持的整型。在使用封闭枚举类型的语言中（Java），使用枚举中的一个类型来表示未识别的值，并且可以使用所支持整型来访问。在其他情况下，如果解析的消息被序列号，未识别的值将保持原样。

关于如何在你的应用程序的消息中使用枚举的更多信息，请查看所选择的语言generated code guide。

5.5.6 使用其他消息类型

你可以将其他消息类型用作字段类型。例如，假设在每一个 SearchResponse 消息中包含 Result 消息，此时可以在相同的.proto 文件中定义一个 Result 消息类型，然后在 SearchResponse 消息中指定一个 Result 类型的字段，如：

```
message SearchResponse {
  repeated Result results = 1;
}
message Result {
  string url = 1;
  string title = 2;
  repeated string snippets = 3;
}
```

导入定义

在上面的例子中，Result 消息类型与 SearchResponse 是定义在同一文件中的。如果想要使用的消息类型已经在其他.proto 文件中已经定义过了呢？你可以通过导入（importing）其他.proto 文件中的定义来使用它们。要导入其他.proto 文件的定义，你需要在你的文件中添加一个导入声明，如：

```
import "myproject/other_protos.proto";
```

默认情况下你只能使用直接导入的.proto 文件中的定义. 然而，有时候你需要移动一个.proto 文件到一个新的位置，可以不直接移动.proto 文件，只需放入一个伪.proto 文件在老的位置，然后使用 import public 转向新的位置。import public 依赖性会通过任意导入包含 import public 声明的 proto 文件传递。例如：

```
// 这是新的 proto
// All definitions are moved here

// 这是久的 proto
// 这是所有客户端正在导入的包
import public "new.proto";
import "other.proto";

// 客户端 proto
import "old.proto";
// 现在你可以使用新旧两种包的 proto 定义了。
```

通过在编译器命令行参数中使用-I/--proto_pathprotocal 编译器会在指定目录搜索要导入的文件。如果没有给出标志，编译器会搜索编译命令被调用的目录。通常你只要指定 proto_path 标志为你的工程根目录就好。并且指定好导入的正确名称就好。

使用 proto2 消息类型

在你的 proto3 消息中导入 proto2 的消息类型也是可以的，反之亦然，然后 proto2 枚举不可以直接在 proto3 的标识符中使用（如果仅仅在 proto2 消息中使用是可以的）。

5.5.7 嵌套类型

你可以在其他消息类型中定义、使用消息类型，在下面的例子中，Result 消息就定义在 SearchResponse 消息内，如：

```
message SearchResponse {
  message Result {
    string url = 1;
    string title = 2;
    repeated string snippets = 3;
  }
  repeated Result results = 1;
}
```

如果你想在它的父消息类型的外部重用这个消息类型，你需要以 Parent.Type 的形式使用它，如：

```
message SomeOtherMessage {
  SearchResponse.Result result = 1;
}
```

当然，你也可以将消息嵌套任意多层，如：

```
message Outer {                  // Level 0
  message MiddleAA {  // Level 1
    message Inner {   // Level 2
      int64 ival = 1;
      bool  booly = 2;
    }
  }
  message MiddleBB {  // Level 1
    message Inner {   // Level 2
      int32 ival = 1;
      bool  booly = 2;
```

(下页继续)

(续上页)

```
    }
  }
}
```

5.5.8 更新一个消息类型

如果一个已有的消息格式已无法满足新的需求——如，要在消息中添加一个额外的字段——但是同时旧版本写的代码仍然可用。不用担心！更新消息而不破坏已有代码是非常简单的。在更新时只要记住以下的规则即可。

- 不要更改任何已有的字段的数值标识。
- 如果你增加新的字段，使用旧格式的字段仍然可以被你新产生的代码所解析。你应该记住这些元素的默认值这样你的新代码就可以以适当的方式和旧代码产生的数据交互。相似的，通过新代码产生的消息也可以被旧代码解析：只不过新的字段会被忽视掉。注意，未被识别的字段会在反序列化的过程中丢弃掉，所以如果消息再被传递给新的代码，新的字段依然是不可用的（这和 proto2 中的行为是不同的，在 proto2 中未定义的域依然会随着消息被序列化）
- 非 required 的字段可以移除——只要它们的标识号在新的消息类型中不再使用（更好的做法可能是重命名那个字段，例如在字段前添加“OBSOLETE_”前缀，那样的话，使用的.proto 文件的用户将来就不会无意中重新使用了那些不该使用的标识号）。
- int32, uint32, int64, uint64, 和 bool 是全部兼容的，这意味着可以将这些类型中的一个转换为另外一个，而不会破坏向前、向后的兼容性。如果解析出来的数字与对应的类型不相符，那么结果就像在 C++ 中对它进行了强制类型转换一样（例如，如果把一个 64 位数字当作 int32 来读取，那么它就会被截断为 32 位的数字）。
- sint32 和 sint64 是互相兼容的，但是它们与其他整数类型不兼容。
- string 和 bytes 是兼容的——只要 bytes 是有效的 UTF-8 编码。
- 嵌套消息与 bytes 是兼容的——只要 bytes 包含该消息的一个编码过的版本。
- fixed32 与 sfixed32 是兼容的，fixed64 与 sfixed64 是兼容的。
- 枚举类型与 int32，uint32，int64 和 uint64 相兼容（注意如果值不相兼容则会被截断），然而在客户端反序列化之后他们可能会有不同的处理方式，例如，未识别的 proto3 枚举类型会被保留在消息中，但是他的表示方式会依照语言而定。int 类型的字段总会保留他们的

5.5.9 Any

Any 类型消息允许你在没有指定他们的.proto 定义的情况下使用消息作为一个嵌套类型。一个 Any 类型包括一个可以被序列化 bytes 类型的任意消息，以及一个 URL 作为一个全局标识符和解析消息类型。为了使用 Any 类型，你需要导入 `import google/protobuf/any.proto`。

```
import "google/protobuf/any.proto";
message ErrorStatus {
  string message = 1;
  repeated google.protobuf.Any details = 2;
}
```

对于给定的消息类型的默认类型 URL 是 type.googleapis.com/packagename.messagename。

不同语言的实现会支持动态库以线程安全的方式去帮助封装或者解封装 Any 值。例如在 java 中，Any 类型会有特殊的 `pack()` 和 `unpack()` 访问器，在 C++ 中会有 `PackFrom()` 和 `UnpackTo()` 方法。

```
// Storing an arbitrary message type in Any.
NetworkErrorDetails details = ...;
ErrorStatus status;
status.add_details()->PackFrom(details);
// Reading an arbitrary message from Any.
ErrorStatus status = ...;
for (const Any& detail : status.details()) {
  if (detail.Is<NetworkErrorDetails>()) {
    NetworkErrorDetails network_error;
    detail.UnpackTo(&network_error);
    ... processing network_error ...
  }
}
```

目前，用于 Any 类型的动态库仍在开发之中如果你已经很熟悉proto2 语法，使用 Any 替换扩展。

5.5.10 Oneof

如果你的消息中有很多可选字段，并且同时至多一个字段会被设置，你可以加强这个行为，使用 oneof 特性节省内存.

Oneof 字段就像可选字段，除了它们会共享内存，至多一个字段会被设置。设置其中一个字段会清除其它字段。你可以使用 `case()` 或者 `WhichOneof()` 方法检查哪个 oneof 字段被设置，看你使用什么语言了.

使用 Oneof

为了在.proto 定义 Oneof 字段，你需要在名字前面加上 oneof 关键字, 比如下面例子的 test_oneof:

```
message SampleMessage {
  oneof test_oneof {
    string name = 4;
    SubMessage sub_message = 9;
  }
}
```

然后你可以增加 oneof 字段到 oneof 定义中. 你可以增加任意类型的字段, 但是不能使用 repeated 关键字.

在产生的代码中, oneof 字段拥有同样的 getters 和 setters，就像正常的可选字段一样. 也有一个特殊的方法来检查到底那个字段被设置. 你可以在相应的语言API 指南中找到 oneof API 介绍.

Oneof 特性

- 设置 oneof 会自动清楚其它 oneof 字段的值. 所以设置多次后，只有最后一次设置的字段有值.

```
SampleMessage message;
message.set_name("name");
CHECK(message.has_name());
message.mutable_sub_message();   // Will clear name field.
CHECK(!message.has_name());
```

- 如果解析器遇到同一个 oneof 中有多个成员，只有最会一个会被解析成消息。
- oneof 不支持 repeated.
- 反射 API 对 oneof 字段有效.

- 如果使用 C++, 需确保代码不会导致内存泄漏. 下面的代码会崩溃, 因为 sub_message 已经通过 set_name() 删除了

```
SampleMessage message;
SubMessage* sub_message = message.mutable_sub_message();
message.set_name("name");      // Will delete sub_message
sub_message->set_...            // Crashes here
```

- 在 C++ 中，如果你使用 Swap() 两个 oneof 消息，每个消息，两个消息将拥有对方的值，例如在下面的例子中，msg1 会拥有 sub_message 并且 msg2 会有 name。

```
SampleMessage msg1;
msg1.set_name("name");
SampleMessage msg2;
msg2.mutable_sub_message();
msg1.swap(&msg2);
CHECK(msg1.has_sub_message());
CHECK(msg2.has_name());
```

向后兼容性问题

当增加或者删除 oneof 字段时一定要小心. 如果检查 oneof 的值返回 None/NOT_SET, 它意味着 oneof 字段没有被赋值或者在一个不同的版本中赋值了。你不会知道是哪种情况，因为没有办法判断如果未识别的字段是一个 oneof 字段。

Tag 重用问题:

- **将字段移入或移除 oneof**：在消息被序列号或者解析后，你也许会失去一些信息（有些字段也许会被清除）
- **删除一个字段或者加入一个字段**：在消息被序列号或者解析后，这也许会清除你现在设置的 oneof 字段
- **分离或者融合 oneof**：行为与移动常规字段相似。

5.5.11 Map

如果你希望创建一个关联映射，protocol buffer 提供了一种快捷的语法:

```
map<key_type, value_type> map_field = N;
```

其中 key_type 可以是任意 Integer 或者 string 类型（所以，除了 floating 和 bytes 的任意标量类型都是可以的）value_type 可以是任意类型。

例如，如果你希望创建一个 project 的映射，每个 Projecct 使用一个 string 作为 key，你可以像下面这样定义:

```
map<string, Project> projects = 3;
```

- Map 的字段可以是 repeated。
- 序列化后的顺序和 map 迭代器的顺序是不确定的，所以你不要期望以固定顺序处理 Map
- 当为.proto 文件产生生成文本格式的时候，map 会按照 key 的顺序排序，数值化的 key 会按照数值排序。
- 从序列化中解析或者融合时，如果有重复的 key 则后一个 key 不会被使用，当从文本格式中解析 map 时，如果存在重复的 key。

生成 map 的 API 现在对于所有 proto3 支持的语言都可用了，你可以从API 指南找到更多信息。

向后兼容性问题

map 语法序列化后等同于如下内容，因此即使是不支持 map 语法的 protocol buffer 实现也是可以处理你的数据的：

```
message MapFieldEntry {
  key_type key = 1;
  value_type value = 2;
}
repeated MapFieldEntry map_field = N;
```

5.5.12 Package

当然可以为.proto 文件新增一个可选的 package 声明符，用来防止不同的消息类型有命名冲突。如：

```
package foo.bar;
message Open { ... }
```

在其他的消息格式定义中可以使用包名 + 消息名的方式来定义域的类型，如：

```
message Foo {
  ...
  required foo.bar.Open open = 1;
  ...
}
```

包的声明符会根据使用语言的不同影响生成的代码。

- 对于 C++，产生的类会被包装在 C++ 的命名空间中，如上例中的 Open 会被封装在 foo::bar 空间中；- 对于 Java，包声明符会变为 java 的一个包，除非在.proto 文件中提供了一个明确有 java_package；
- 对于 Python，这个包声明符是被忽略的，因为 Python 模块是按照其在文件系统中的位置进行组织的。
- 对于 Go，包可以被用做 Go 包名称，除非你显式的提供一个 option go_package 在你的.proto 文件中。
- 对于 Ruby，生成的类可以被包装在内置的 Ruby 名称空间中，转换成 Ruby 所需的大小写样式（首字母大写；如果第一个符号不是一个字母，则使用 PB_ 前缀），例如 Open 会在 Foo::Bar 名称空间中。
- 对于 javaNano 包会使用 Java 包，除非你在你的文件中显式的提供一个 option java_package。
- 对于 C# 包可以转换为 PascalCase 后作为名称空间，除非你在你的文件中显式的提供一个 option csharp_namespace，例如，Open 会在 Foo.Bar 名称空间中

包及名称的解析

Protocol buffer 语言中类型名称的解析与 C++ 是一致的：首先从最内部开始查找，依次向外进行，每个包会被看作是其父类包的内部类。当然对于（foo.bar.Baz）这样以“.”分隔的意味着是从最外围开始的。

ProtocolBuffer 编译器会解析.proto 文件中定义的所有类型名。对于不同语言的代码生成器会知道如何来指向每个具体的类型，即使它们使用了不同的规则。

5.5.13 定义服务 (Service)

如果想要将消息类型用在 RPC(远程方法调用) 系统中，可以在.proto 文件中定义一个 RPC 服务接口，protocol buffer 编译器将会根据所选择的不同语言生成服务接口代码及存根。如，想要定义一个 RPC 服务并具有一个方法，该方法能够接收 SearchRequest 并返回一个 SearchResponse，此时可以在.proto 文件中进行如下定义：

```
service SearchService {
  rpc Search (SearchRequest) returns (SearchResponse);
}
```

最直观的使用 protocol buffer 的 RPC 系统是gRPC, 一个由谷歌开发的语言和平台中的开源的 PRC 系统，gRPC 在使用 protocl buffer 时非常有效，如果使用特殊的 protocol buffer 插件可以直接为您从.proto 文件中产生相关的 RPC 代码。

如果你不想使用 gRPC，也可以使用 protocol buffer 用于自己的 RPC 实现，你可以从proto2 语言指南中找到更多信息

还有一些第三方开发的 PRC 实现使用 Protocol Buffer。参考第三方插件 wiki查看这些实现的列表。

5.5.14 JSON 映射

Proto3 支持 JSON 的编码规范，使他更容易在不同系统之间共享数据，在下表中逐个描述类型。

如果 JSON 编码的数据丢失或者其本身就是 null，这个数据会在解析成 protocol buffer 的时候被表示成默认值。如果一个字段在 protocol buffer 中表示为默认值，体会在转化成 JSON 的时候编码的时候忽略掉以节省空间。具体实现可以提供在 JSON 编码中可选的默认值。

5.5.15 选项

定义.proto 文件时能够标注一系列的 option。Option 并不改变整个文件声明的含义，但却能够影响特定环境下处理方式。完整的可用选项可以在 `google/protobuf/descriptor.proto` 找到。

一些选项是文件级别的，意味着它可以作用于最外范围，不包含在任何消息内部、enum 或服务定义中。一些选项是消息级别的，意味着它可以用在消息定义的内部。当然有些选项可以作用在域、enum 类型、enum 值、服务类型及服务方法中。到目前为止，并没有一种有效的选项能作用于所有的类型。

如下就是一些常用的选项：

- java_package (文件选项) : 这个选项表明生成 java 类所在的包。如果在.proto 文件中没有明确的声明 java_package，就采用默认的包名。当然了，默认方式产生的 java 包名并不是最好的方式，按照应用名称倒序方式进行排序的。如果不需要产生 java 代码，则该选项将不起任何作用。如：

```
option java_package = "com.example.foo";
```

- java_outer_classname (文件选项): 该选项表明想要生成 Java 类的名称。如果在.proto 文件中没有明确的 java_outer_classname 定义，生成的 class 名称将会根据.proto 文件的名称采用驼峰式的命名方式进行生成。如（foo_bar.proto 生成的 java 类名为 FooBar.java）, 如果不生成 java 代码，则该选项不起任何作用。如：

```
option java_outer_classname = "Ponycopter";
```

- optimize_for(文件选项): 可以被设置为 SPEED, CODE_SIZE, 或者 LITE_RUNTIME。这些值将通过如下的方式影响 C++ 及 java 代码的生成：
  - SPEED (default): protocol buffer 编译器将通过在消息类型上执行序列化、语法分析及其他通用的操作。这种代码是最优的。

  - CODE_SIZE: protocol buffer 编译器将会产生最少量的类，通过共享或基于反射的代码来实现序列化、语法分析及各种其它操作。采用该方式产生的代码将比 SPEED 要少得多，但是操作要相对慢些。当然实现的类及其对外的 API 与 SPEED 模式都是一样的。这种方式经常用在一些包含大量的.proto 文件而且并不盲目追求速度的应用中。
  - LITE_RUNTIME: protocol buffer 编译器依赖于运行时核心类库来生成代码（即采用 libprotobuf-lite 替代 libprotobuf）。这种核心类库由于忽略了一些描述符及反射，要比全类库小得多。这种模式经常在移动手机平台应用多一些。编译器采用该模式产生的方法实现与 SPEED 模式不相上下，产生的类通过实现 MessageLite 接口，但它仅仅是 Messager 接口的一个子集。

```
option optimize_for = CODE_SIZE;
```

- cc_enable_arenas(文件选项): 对于 C++ 产生的代码启用 arena allocation
- objc_class_prefix(文件选项): 设置 Objective-C 类的前缀，添加到所有 Objective-C 从此.proto 文件产生的类和枚举类型。没有默认值，所使用的前缀应该是苹果推荐的 3-5 个大写字符，注意 2 个字节的前缀是苹果所保留的。
- deprecated(字段选项): 如果设置为 true 则表示该字段已经被废弃，并且不应该在新的代码中使用。在大多数语言中没有实际的意义。在 java 中，这回变成 @Deprecated 注释，在未来，其他语言的代码生成器也许会在字标识符中产生废弃注释，废弃注释会在编译器尝试使用该字段时发出警告。如果字段没有被使用你也不希望有新用户使用它，尝试使用保留语句替换字段声明。

```
int32 old_field = 6 [deprecated=true];
```

自定义选项

ProtocolBuffers 允许自定义并使用选项。该功能应该属于一个高级特性，对于大部分人是用不到的。如果你的确希望创建自己的选项，请参看 Proto2 Language Guide。注意创建自定义选项使用了拓展，拓展只在 proto3 中可用。

5.5.16 生成访问类

可以通过定义好的.proto 文件来生成 Java,Python,C++, Ruby, JavaNano, Objective-C, 或者 C# 代码，需要基于.proto 文件运行 protocol buffer 编译器 protoc。如果你没有安装编译器，下载安装包并遵照 README 安装。对于 Go, 你还需要安装一个特殊的代码生成器插件。你可以通过 GitHub 上的 protobuf 库找到安装过程

通过如下方式调用 protocol 编译器：

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --python_out=DST_
↪DIR --go_out=DST_DIR --ruby_out=DST_DIR --javanano_out=DST_DIR --objc_out=DST_DIR --
↪csharp_out=DST_DIR path/to/file.proto
```

- IMPORT_PATH 声明了一个.proto 文件所在的解析 import 具体目录。如果忽略该值，则使用当前目录。如果有多个目录则可以多次调用--proto_path，它们将会顺序的被访问并执行导入。-I=IMPORT_PATH 是--proto_path 的简化形式。
- 当然也可以提供一个或多个输出路径：
  - --cpp_out 在目标目录 DST_DIR 中产生 C++ 代码，可以在C++ 代码生成参考中查看更多。
  - --java_out 在目标目录 DST_DIR 中产生 Java 代码，可以在 Java 代码生成参考中查看更多。
  - --python_out 在目标目录 DST_DIR 中产生 Python 代码，可以在Python 代码生成参考中查看更多。
  - --go_out 在目标目录 DST_DIR 中产生 Go 代码，可以在GO 代码生成参考中查看更多。
  - --ruby_out 在目标目录 DST_DIR 中产生 Ruby 代码，参考正在制作中。

  - --javanano_out 在目标目录 DST_DIR 中生成 JavaNano，JavaNano 代码生成器有一系列的选项用于定制自定义生成器的输出：你可以通过生成器的README查找更多信息，JavaNano 参考正在制作中。
  - --objc_out 在目标目录 DST_DIR 中产生 Object 代码，可以在Objective-C 代码生成参考中查看更多。
  - --csharp_out 在目标目录 DST_DIR 中产生 Object 代码，可以在C# 代码生成参考中查看更多。
  - --php_out 在目标目录 DST_DIR 中产生 Object 代码，可以在PHP 代码生成参考中查看更多。

作为一个方便的拓展，如果 DST_DIR 以.zip 或者.jar 结尾，编译器会将输出写到一个 ZIP 格式文件或者符合 JAR 标准的.jar 文件中。注意如果输出已经存在则会被覆盖，编译器还没有智能到可以追加文件。

- 你必须提议一个或多个.proto 文件作为输入，多个.proto 文件可以只指定一次。虽然文件路径是相对于当前目录的，每个文件必须位于其 IMPORT_PATH 下，以便每个文件可以确定其规范的名称。

CHAPTER 6

服务部署运行

6.1 Docker

6.1.1 Docker 基础

Docker 镜像: image

Docker 镜像是一个特殊的文件系统，它提供容器运行时所需的程序、库、资源、配置等文件，还包含了一些为运行时准备的一些配置参数（如匿名卷、环境变量、用户等）。镜像中不包含任何动态数据，其内容在构建之后也不会被改变，Docker 引擎基于 Docker 镜像、提供运行时所需的配置参数就可以运行一个 Docker 容器。Docker 镜像是一个虚拟的概念，设计时采用了分层存储的架构，由多层文件系统（如基础镜像）联合组成。

镜像构建时，会一层层构建，前一层是后一层的基础。每一层构建完就不会再发生改变，后一层上的任何改变只发生在自己这一层。比如，删除前一层文件的操作，实际不是真的删除前一层的文件，而是仅在当前层标记为该文件已删除。在最终容器运行的时候，虽然不会看到这个文件，但是实际上该文件会一直跟随镜像。因此，在构建镜像的时候，需要额外小心，每一层尽量只包含该层需要添加的东西，任何额外的东西应该在该层构建结束前清理掉。

分层存储的特征还使得镜像的复用、定制变的更为容易。甚至可以用之前构建好的镜像作为基础层，然后进一步添加新的层，以定制自己所需的内容，构建新的镜像。

容器运行时，会在镜像文件系统层上方再添加一个读写层，容器运行时在容器读写层进行文件读写操作，当容器停止时，docker 引擎会删除容器读写层。

查看一个镜像的层次结构，可通过 `docker history <image-id>` 查看：

容器: container

容器是镜像运行时的实体 (镜像实例化)。容器可以被创建、启动、停止、删除、暂停等。

容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于自己的独立的命名空间。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。

每一个容器运行时，是以镜像为基础层，在其上创建一个当前容器的存储层。容器存储层的生存周期和容器一样，因此，容器不应该向其存储层内写入任何数据，容器存储层要保持无状态化。所有的文件写入操作，都应该使用数据卷 (Volume)、或者绑定宿主目录，在这些位置的读写会跳过容器存储层，直接对宿主（或网络存储）发生读写，其性能和稳定性更高。

常用命令

```
# 显示本机镜像
docker images
docker image ls
# 编译镜像
docker build -t cfp/node
# 拉取镜像
docekr pull cfp/node-1:latest
# 推送镜像
docekr push cfp/node-1:latest
# 删除本地镜像
docker rmi cfp/node-1:latest
# 镜像 tag
docker tag cfp/node-1:latest  cfp/node-1:v1.0
# 显示本地容器
docker ps -a
# 删除本地容器
docker rm full0
# 进入容器
docker exec -it full0 /bin/sh
# 进入容器
docker attach full0
# 启动、停止、重启容器
docker start/stop/restart
# 查看容器 log
docker logs -f full0
# 启动容器
docker run cfp/node ls /go/src/github.com/cosmos/launch
# 使用容器生成镜像
docker commit -a "author" -m "message" full0 image:new
```

- docker exec 与 attache 的区别
  - `docker exec -it` 分配伪终端和 stdin，跟正常的 console 一样执行命令
  - `docker attach` attach 到一个已经运行的容器的 stdin，然后进行命令执行。但是，**如果从这个 stdin 中 exit(如 ctl+c)，会导致容器的停止**
- docker commit 生成镜像
  - 将容器的存储层保存下来成为镜像的一层。就是在原有镜像的基础上，再叠加上容器的存储层，并构成新的镜像。以后我们运行这个新镜像的时候，就会拥有原有容器最后的文件变化。
  - `docker commit` 生成镜像时黑箱操作，除了制作镜像的人知道执行过什么命令、怎么生成的镜像，别人根本无从得知。难用且不易维护。

- docker commit 生成镜像易包含其他无用修改，使镜像臃肿
- **通常不建议使用此方式构建镜像，推荐使用 Dockerfile**
- 可用于特殊场景，如被入侵后保存现场等

6.1.2 Docker 应用

docker 运行 mysql

- 获取镜像

```
# 从 docker hub 的仓库中拉取 mysql 镜像
sudo docker pull mysql
```

- 运行一个 mysql 实例

```
docker run --name fp-mysql -p 23306:3306 -e MYSQL_ROOT_PASSWORD=123456 -d mysql:5.
↪6

5b6bf6f629bfe46b4c8786b555d8db1947680138b2de1f268f310a15ced7247a


run             运行一个容器
--name          后面是这个镜像的名称
-p 3306:3306    表示在这个容器中使用 3306 端口(第二个)映射到本机的端口号也为 3306(第一个)
-d              表示使用守护进程运行，即服务挂在后台
```

- 装一个 mysql-client，访问 mysql 数据库

```
sudo apt-get install mysql-client-core-5.6

# mysql 命令访问服务器
mysql -h 192.168.95.4 -P 3306 -uroot -p 123456
```

-

6.1.3 Docker file

Docker file 分析

Dockerfile 是一个文本文件，用于构建生成 Docker 镜像，其内**以一个镜像为基础**，在基础镜像之上通过一条条的指令 (Instruction) 加工、定制，最终产出自己想要的镜像。每一条指令都基于前一层依赖的镜像层构建一层新的镜像层，因此每一条指令的内容，就是描述该层应当如何构建。

```
FROM cfp/build as builder

# Set working directory for the build
WORKDIR $GOPATH/src/github.com/fpchan/cfp

# Add cfp source files
COPY . .

# build cfp
RUN make install
```

(下页继续)

(续上页)

```
RUN make launchcmd

# FROM cfp/node
FROM cfp/build

# Set working directory for the build
WORKDIR $GOPATH/src/github.com/cosmos/launch

ARG LAUNCH_REPOSITORY=https://github.com/cfp/launch
ARG LAUNCH_BRANCH=dev
ARG SEED_NODE_NUM=1
ARG VAL_NODE_NUM=1
ARG FULL_NODE_NUM=1

RUN mkdir -p $GOPATH/src/github.com/cosmos \
    && cd $GOPATH/src/github.com/cosmos \
    && git clone $LAUNCH_REPOSITORY -b $LAUNCH_BRANCH

COPY --from=builder $GOPATH/bin $GOPATH/bin
COPY docker/launch/genfile.sh .
COPY docker/launch/start.sh .

EXPOSE 26656 26657 26658 26659 6060

# generate genesis file
RUN $GOPATH/src/github.com/cosmos/launch/genfile.sh  ${SEED_NODE_NUM} ${VAL_NODE_NUM}
↪${FULL_NODE_NUM}
ENTRYPOINT $GOPATH/src/github.com/cosmos/launch/start.sh
docker build \
--build-arg SEED_NODE_NUM=1 \
--build-arg VAL_NODE_NUM=4 \
--build-arg FULL_NODE_NUM=10 \
-f Dockerfile -t cfp/node-1:rr .
```

- FROM 指定基础镜像
- COPY 从构建上下文目录中 < 源路径 > 的文件/目录复制到新的一层的镜像内的 < 目标路径 > 位置
  - **构建上下文目录**是指 docker build 指定的目录，通常使用 . 即当前目录，操作文件不可以超出上下文目录
  - COPY [--chown=<user>:<group>] < 源路径>... < 目标路径>
  - COPY [--chown=<user>:<group>] ["< 源路径 1>",... "< 目标路径>"]
- ADD 与 COPY 基本一致，但如果 < 源路径 > 是 URL 或者压缩文件，ADD 可以自动下载和解压
- RUN 指定要执行的命令
  - *shell* 格式：RUN < 命令>
  - *exec* 格式：RUN [" 可执行文件", " 参数 1", " 参数 2"]
- CMD 启动容器时执行的命令，格式同 RUN
  - 在运行时可以替代，docker run 中执行指定的指令或 docker-compose 中的 command
  - 在指令格式上，一般推荐使用 exec 格式，这类格式在解析时会被解析为 JSON 数组，因此一定要使用双引号 "，而不要使用单引号
- ENTRYPOINT 同 CMD 含义相同，格式同 RUN

  - 当同时执行 CMD 和 ENTRYPOINT 后，CMD 不再是直接的运行命令，二是将其作为参数传给 ENTRYPOINT，变成<ENTRYPOINT> "<CMD>"
  - ENTRYPOINT 比 CMD 的优势：
    * 如果指定 CMD ["curl", "-s", "https://ip.cn"]，执行 docker run {images} 会输出 IP，如果想加入参数-i，则执行 docker run {images} -i 会报错
    * 如果指定 ENTRYPOINT ["curl", "-s", "https://ip.cn"]，同样的执行 docker run {images} -i 则没问题，相当于执行 curl -s https://ip.cn -i
- ENV 环境变量
- ARG 构建参数
- EXPOSE 声明容器提供的服务端口
- WORKDIR 工作目录
- USER 指定当前用户

Docker file 最佳实践

- 使用.dockerignore 文件
- 使用多阶段构建
  - 多阶段构建，允许在一个 Dockerfile 中使用多个 FROM 指令，可直接将前置阶段、或其他镜像中的资源文件复制到当前构建阶段，并最终只产生最后一个镜像，大大减少了镜像大小及个数
- 避免安装不必要的包
- 一个容器只运行一个进程
- 镜像层数尽可能少
- 构建缓存
  - 如果你不想在构建过程中使用缓存，你可以在 docker build 命令中使用 --no-cache=true 选项
- 将多个 RUN 指令合并为一个
  - Dockerfile 中的每个指令都会创建一个新的镜像层。
  - 镜像层将被缓存和复用
  - 当 Dockerfile 的指令修改了，复制的文件变化了，或者构建镜像时指定的变量不同了，对应的镜像层缓存就会失效
  - 某一层的镜像缓存失效之后，它之后的镜像层缓存都会失效
  - 镜像层是不可变的，如果我们再某一层中添加一个文件，然后在下一层中删除它，则镜像中依然会包含该文件 (只是这个文件在 Docker 容器中不可见了)
- 基础镜像的标签不要用 latest
  - 当镜像更新时，latest 标签会指向不同的镜像，这时构建镜像有可能失败
- 每个 RUN 指令后删除多余文件
- 选择合适的基础镜像 (alpine 版本最好)
  - alpine 是一个极小化的 Linux 发行版，只有 4MB，这让它非常适合作为基础镜像
- COPY 与 ADD 优先使用前者

  - COPY 指令非常简单，仅用于将文件拷贝到镜像中。ADD 相对来讲复杂，可以用于下载远程文件以及解压压缩包
- 合理调整 COPY 与 RUN 的顺序
  - 应该**把变化最少的部分放在 Dockerfile 的前面**，这样可以充分利用镜像缓存。

6.1.4 Docker Compose

Docker Compose 介绍

Docker Compose 是 Docker 官方编排（orchestration）项目之一，负责快速的部署分布式应用。

它允许用户通过一个 Docker-compose.yml 模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。

- 服务 (Service)：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。
- 项目 (Project)：由一组关联的应用容器组成的一个完整业务单元，在 Docker-compose.yml 文件中定义。
- Compose 的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷地生命周期管理。
- Compose 项目由 Python 编写，实现上调用了 Docker 服务提供的 API 来对容器进行管理。

常用命令

大部分命令跟 `docker` 下的命令功能相同

up

尝试自动完成包括构建镜像，（重新）创建服务，启动服务，并关联服务相关容器

```
docker-compose -f localhost.yml up -d
```

- `-f` 指定使用的 Compose 模板文件，默认为 `docker-compose.yml`，可以多次指定
- `-d` 后台启动并运行所有的容器
- 默认网络为 `bridge`，可通过`--x-network-driver DRIVER` 设置
- 默认，如果服务容器已经存在，`docker-compose up` 将会尝试停止容器，然后重新创建（保持使用 `volumes-from` 挂载的卷），以保证新启动的服务匹配 `docker-compose.yml` 文件的最新内容。
- 如果用户不希望容器被停止并重新创建，可以使用 `docker-compose up --no-recreate`。这样将只会启动处于停止状态的容器，而忽略已经运行的服务。
- 如果用户只想重新部署某个服务，可以使用 `docker-compose up --no-deps -d <SERVICE_NAME>` 来重新创建服务并后台停止旧服务，启动新服务，并不会影响到其所依赖的服务。

down

停止 up 命令所启动的容器，并移除网络

```
docker-compose -f localhost.yml down
```

stop

停止已经处于运行状态的容器，但不删除。通过 docker-compose start 可以再次启动这些容器

```
docker-compose -f localhost.yml stop nginx
```

- 如果不指定服务名，即停止 yml 文件中的所有服务容器

start

启动已经存在的服务容器

```
docker-compose -f localhost.yml start nginx
```

- 如果不指定服务名，即启动 yml 文件中的所有服务容器

其余脚本命令

```
# 登录到 nginx 容器中
docker-compose exec nginx bash

# 重新启动 nginx 容器
docker-compose restart nginx

# 暂停 nignx 容器
docker-compose pause nginx

#  恢复 ningx 容器
docker-compose unpause nginx

# 删除容器（删除前必须关闭容器）
docker-compose rm nginx

# 停止 nignx 容器
docker-compose stop nginx

# 启动 nignx 容器
docker-compose start nginx

# 在 php-fpm 中不启动关联容器，并容器执行 php -v 执行完成后删除容器
docker-compose run --no-deps --rm php-fpm php -v

# 构建镜像
docker-compose build nginx

# 不带缓存的构建。
```

(下页继续)

(续上页)

```
docker-compose build --no-cache nginx

# 验证 (docker-compose.yml) 文件配置，当配置正确时，不输出任何内容，当文件配置错误，输出错误信息。
docker-compose config  -q

# 查看 nginx 的日志
docker-compose logs  nginx

# 查看 nginx 的实时日志
docker-compose logs -f nginx

# 以 json 的形式输出 nginx 的 docker 日志
docker-compose events --json nginx
```

docker-compose file

分解案例

```
## 服务基于已经存在的镜像
services:
  web:
    image: hello-world


## 服务基于 dockerfile
build: /path/to/build/dir
build: ./dir
build:
  context: ../
  dockerfile: path/of/Dockerfile

build: ./dir
image: webapp:tag


##command command 命令可以覆盖容器启动后默认执行的命令
command: bundle exec thin -p 3000
command: [bundle, exec, thin, -p, 3000]

##container_name
## Compose 的容器名称格式是: < 项目名称>< 服务名称>< 序号>
## 虽然可以自定义项目名称、服务名称，但是如果你想完全控制容器的命名，可以使用这个标签指定
container_name: app

##depends_on depends_on 解决了容器的依赖、启动先后的问题
version: '2'
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
```

(下页继续)

(续上页)

```
  db:


##dns
dns: 8.8.8.8
dns:
  - 8.8.8.8
  - 9.9.9.9

dns_search: example.com
dns_search:
  - dc1.example.com
  - dc2.example.com

##tmfs 挂载临时目录到容器内部，与 run 的参数一样效果
tmpfs: /run
tmpfs:
  - /run
  - /tmp

##environment 设置镜像变量，它可以保存变量到镜像里，也就是说启动的容器也会包含这些变量设置
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SHOW=true
  - SESSION_SECRET

##expose  用于指定暴露的端口，但是只是作为参考，端口映射的话还得 ports 标签
expose:
 - "3000"
 - "8000"

##external_links
# 在使用 Docker 的过程中，我们会有许多单独使用 docker run 启动的容器，为了使 Compose 能够连接这些不在 docker-compose.yml 中定义的容器，我们需要一个特殊的标签，就是 external_links，它可以让 Compose 项目里面的容器连接到那些项目配置外部的容器（前提是外部容器中必须至少有一个容器是连接到与项目内的服务的同一个网络里面）

external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql

##extra_hosts  添加主机名的标签，就是往容器内部/etc/hosts 文件中添加一些记录
extra_hosts:
 - "somehost:162.242.195.82"
 - "otherhost:50.31.209.229"

##labels 向容器添加元数据，和 Dockerfile 的 lable 指令一个意思

labels:
  com.example.description: "Accounting webapp"
```

(下页继续)

(续上页)

```
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""
labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"

##links
# 解决容器连接问题，与 docker 的-link 一样的效果，会连接到其他服务中的容器，使用的别名将会自动在服务
容器中的/etc/hosts 里创建
links:
 - db
 - db:database
 - redis


##ports
# 用作端口映射
# 使用 HOST:CONTAINER 格式或者只是指定容器的端口，宿主机会随机映射端口
ports:
 - "3000"
 - "8000:8000"
 - "49100:22"
 - "127.0.0.1:8001:8001"
# 当使用 HOST:CONTAINER 格式来映射端口时，如果你使用的容器端口小于 60 你可能会得到错误得结果，因为
YAML 将会解析 xx:yy 这种数字格式为 60 进制。所以建议采用字符串格式

##security_opt
# 为每个容器覆盖默认的标签。简单说来就是管理全部服务的标签，比如设置全部服务的 user 标签值为 USER
security_opt:
  - label:user:USER
  - label:role:ROLE

##volumes
# 挂载一个目录或者一个已经存在的数据卷容器，可以直接使用 [HOST:CONTAINER] 这样的格式，或者使用
[HOST:CONTAINER:ro] 这样的格式，或者对于容器来说，数据卷是只读的，这样可以有效保护宿主机的文件系统。
# compose 的数据卷指定路径可以是相对路径，使用 . 或者 … 来指定性对目录

volumes:
  // 只是指定一个路径，Docker 会自动在创建一个数据卷（这个路径是容器内部的）。
  - /var/lib/mysql

  // 使用绝对路径挂载数据卷
  - /opt/data:/var/lib/mysql

  // 以 Compose 配置文件为中心的相对路径作为数据卷挂载到容器。
  - ./cache:/tmp/cache

  // 使用用户的相对路径（~/ 表示的目录是 /home/< 用户目录>/ 或者 /root/）。
  - ~/configs:/etc/configs/:ro

  // 已经存在的命名的数据卷。
  - datavolume:/var/lib/mysql

# 如果你不使用宿主机的路径，你可以指定一个 volume_driver。
volume_driver: mydriver
```

(下页继续)

(续上页)

```
##volumes_from
# 从其它容器或者服务挂载数据卷，可选的参数是:ro 或者:rw，前者表示容器只读，后者表示容器对数据卷是可读可
写的，默认是可读可写的

volumes_from:
  - service_name
  - service_name:ro
  - container:container_name
  - container:container_name:rw

##network_mode
# 网络模式，与 docker client 的-net 参数类似，只是相对多了一个 service:[sevice name] 的格式
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"

##networks 加入指定网络
services:
  some-service:
    networks:
     - some-network
     - other-network
```

Compose file 版本和兼容性

6.2 Apllo

6.2.1 Apollo 配置中心介绍

Jason Song edited this page on 13 Jul · 34 revisions

- 1、What is Apollo
- 2、Why Apollo
- 3、Apollo at a glance
- 4、Apollo in depth
- 5、Contribute to Apollo

6.2.2 1、What is Apollo

1.1 背景

随着程序功能的日益复杂，程序的配置日益增多：各种功能的开关、参数的配置、服务器的地址……

对程序配置的期望值也越来越高：配置修改后实时生效，灰度发布，分环境、分集群管理配置，完善的权限、审核机制……

在这样的大环境下，传统的通过配置文件、数据库等方式已经越来越无法满足开发人员对配置管理的需求。

Apollo 配置中心应运而生!

1.2 Apollo 简介

Apollo（阿波罗）是携程框架部门研发的开源配置管理中心，能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性。

Apollo 支持 4 个维度管理 Key-Value 格式的配置：

1. application (应用)
2. environment (环境)
3. cluster (集群)
4. namespace (命名空间)

同时，Apollo 基于开源模式开发，开源地址：https://github.com/ctripcorp/apollo

1.2 配置基本概念

既然 Apollo 定位于配置中心，那么在这里有必要先简单介绍一下什么是配置。

按照我们的理解，配置有以下几个属性：

- **配置是独立于程序的只读变量**
  - 配置首先是独立于程序的，同一份程序在不同的配置下会有不同的行为。
  - 其次，配置对于程序是只读的，程序通过读取配置来改变自己的行为，但是程序不应该去改变配置。
  - 常见的配置有：DB Connection Str、Thread Pool Size、Buffer Size、Request Timeout、Feature Switch、Server Urls 等。
- **配置伴随应用的整个生命周期**
  - 配置贯穿于应用的整个生命周期，应用在启动时通过读取配置来初始化，在运行时根据配置调整行为。
- **配置可以有多种加载方式**
  - 配置也有很多种加载方式，常见的有程序内部 hard code，配置文件，环境变量，启动参数，基于数据库等
- **配置需要治理**
  - 权限控制
    * 由于配置能改变程序的行为，不正确的配置甚至能引起灾难，所以对配置的修改必须有比较完善的权限控制
  - 不同环境、集群配置管理
    * 同一份程序在不同的环境（开发，测试，生产）、不同的集群（如不同的数据中心）经常需要有不同的配置，所以需要有完善的环境、集群配置管理
  - 框架类组件配置管理
    * 还有一类比较特殊的配置 - 框架类组件配置，比如 CAT 客户端的配置。
    * 虽然这类框架类组件是由其他团队开发、维护，但是运行时是在业务实际应用内的，所以本质上可以认为框架类组件也是应用的一部分。

* 这类组件对应的配置也需要有比较完善的管理方式。

6.2.3 2、Why Apollo

正是基于配置的特殊性，所以 Apollo 从设计之初就立志于成为一个有治理能力的配置发布平台，目前提供了以下的特性：

- **统一管理不同环境、不同集群的配置**
  - Apollo 提供了一个统一界面集中式管理不同环境（environment）、不同集群（cluster）、不同命名空间（namespace）的配置。
  - 同一份代码部署在不同的集群，可以有不同的配置，比如 zookeeper 的地址等
  - 通过命名空间（namespace）可以很方便地支持多个不同应用共享同一份配置，同时还允许应用对共享的配置进行覆盖
- **配置修改实时生效（热发布）**
  - 用户在 Apollo 修改完配置并发布后，客户端能实时（1 秒）接收到最新的配置，并通知到应用程序
- **版本发布管理**
  - 所有的配置发布都有版本概念，从而可以方便地支持配置的回滚
- **灰度发布**
  - 支持配置的灰度发布，比如点了发布后，只对部分应用实例生效，等观察一段时间没问题后再推给所有应用实例
- **权限管理、发布审核、操作审计**
  - 应用和配置的管理都有完善的权限管理机制，对配置的管理还分为了编辑和发布两个环节，从而减少人为的错误。
  - 所有的操作都有审计日志，可以方便地追踪问题
- **客户端配置信息监控**
  - 可以在界面上方便地看到配置在被哪些实例使用
- **提供 Java 和.Net 原生客户端**
  - 提供了 Java 和.Net 的原生客户端，方便应用集成
  - 支持 Spring Placeholder, Annotation 和 Spring Boot 的 ConfigurationProperties，方便应用使用（需要 Spring 3.1.1+）
  - 同时提供了 Http 接口，非 Java 和.Net 应用也可以方便地使用
- **提供开放平台 API**
  - Apollo 自身提供了比较完善的统一配置管理界面，支持多环境、多数据中心配置管理、权限、流程治理等特性。不过 Apollo 出于通用性考虑，不会对配置的修改做过多限制，只要符合基本的格式就能保存，不会针对不同的配置值进行针对性的校验，如数据库用户名、密码，Redis 服务地址等
  - 对于这类应用配置，Apollo 支持应用方通过开放平台 API 在 Apollo 进行配置的修改和发布，并且具备完善的授权和权限控制
- **部署简单**
  - 配置中心作为基础服务，可用性要求非常高，这就要求 Apollo 对外部依赖尽可能地少
  - 目前唯一的外部依赖是 MySQL，所以部署非常简单，只要安装好 Java 和 MySQL 就可以让 Apollo 跑起来

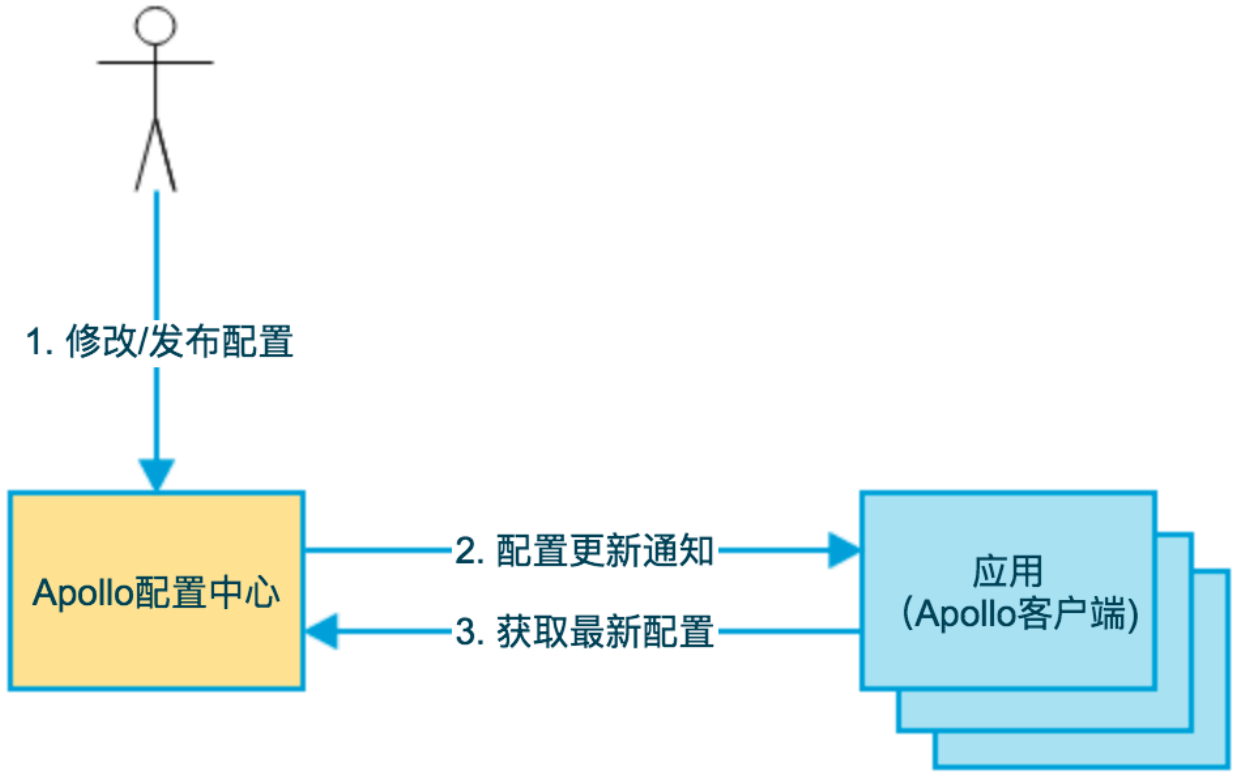
- Apollo 还提供了打包脚本，一键就可以生成所有需要的安装包，并且支持自定义运行时参数

6.2.4 3、Apollo at a glance

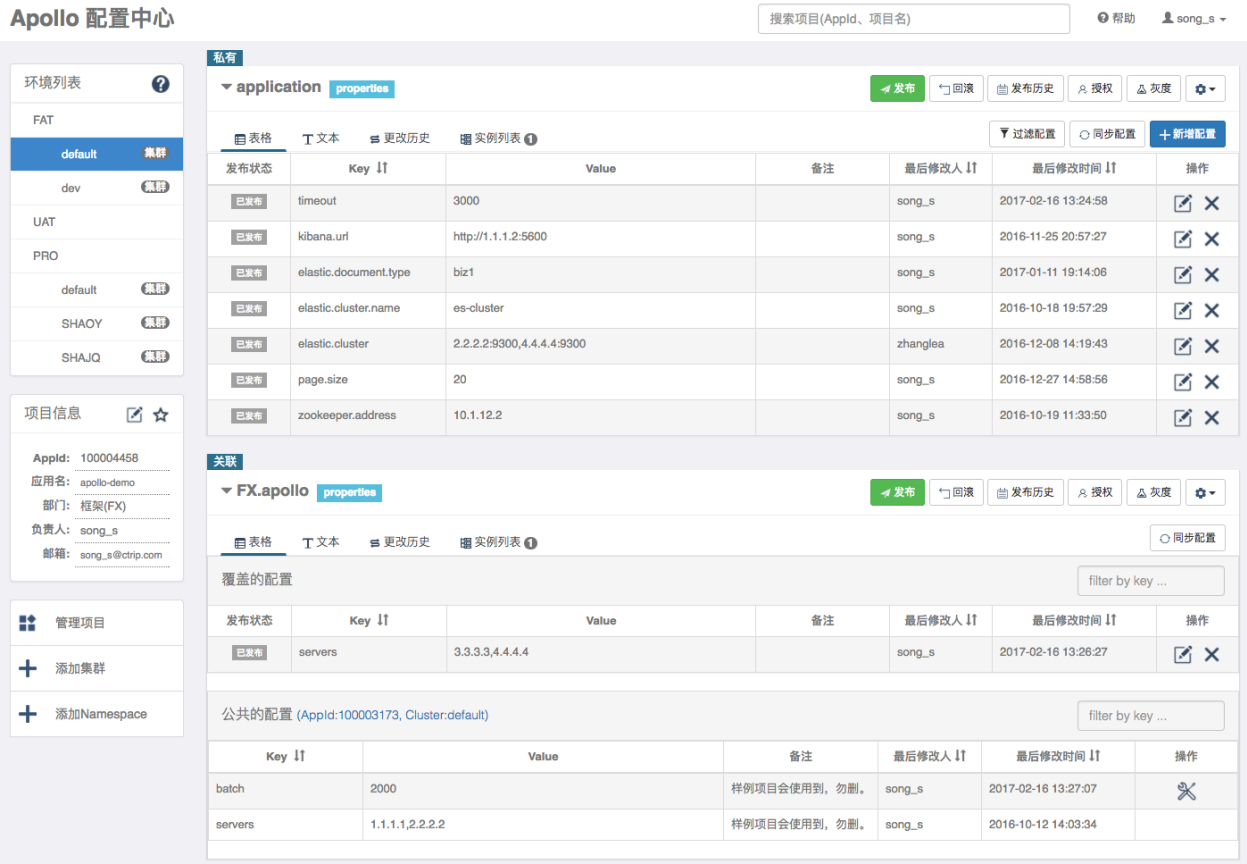
3.1 基础模型

如下即是 Apollo 的基础模型：

1. 用户在配置中心对配置进行修改并发布
2. 配置中心通知 Apollo 客户端有配置更新
3. Apollo 客户端从配置中心拉取最新的配置、更新本地配置并通知到应用



3.2 界面概览

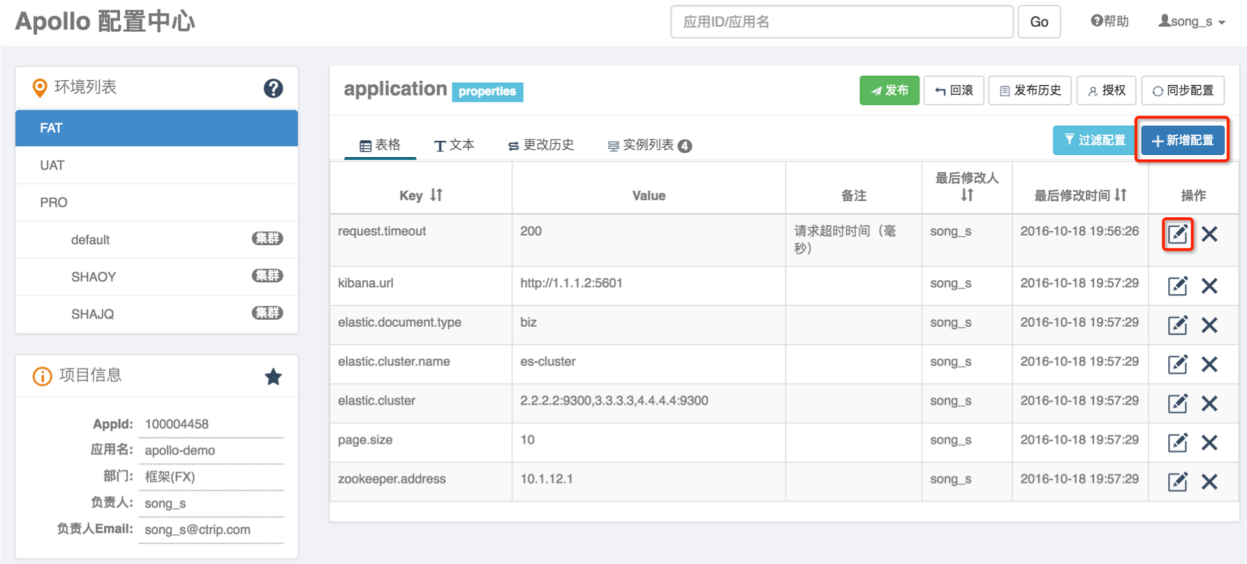


上图是 Apollo 配置中心中一个项目的配置首页

- 在页面左上方的环境列表模块展示了所有的环境和集群，用户可以随时切换。
- 页面中央展示了两个 namespace(application 和 FX.apollo) 的配置信息，默认按照表格模式展示、编辑。用户也可以切换到文本模式，以文件形式查看、编辑。
- 页面上可以方便地进行发布、回滚、灰度、授权、查看更改历史和发布历史等操作

3.3 添加/修改配置项

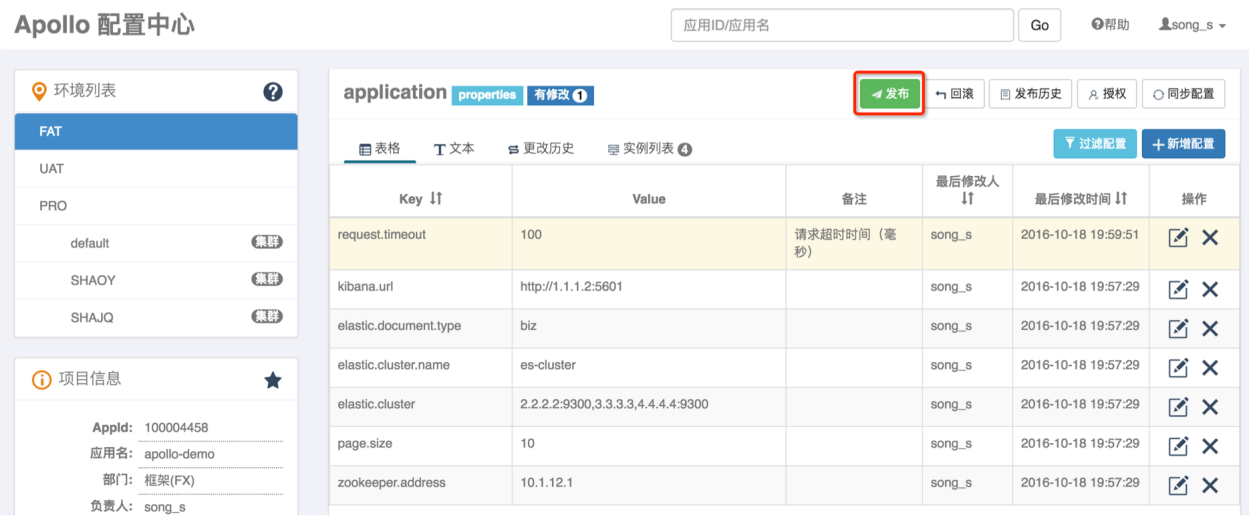
用户可以通过配置中心界面方便的添加/修改配置项，更多使用说明请参见应用接入指南



输入配置信息：

3.4 发布配置

通过配置中心发布配置：



填写发布信息：

3.5 客户端获取配置（Java API 样例）

配置发布后，就能在客户端获取到了，以 Java 为例，获取配置的示例代码如下。Apollo 客户端还支持和 Spring 整合，更多客户端使用说明请参见Java 客户端使用指南和.Net 客户端使用指南。

```
Config config = ConfigService.getAppConfig();
Integer defaultRequestTimeout = 200;
Integer requestTimeout = config.getIntProperty("requestTimeout",␣
↪defaultRequestTimeout);
```

3.6 客户端监听配置变化

通过上述获取配置代码，应用就能实时获取到最新的配置了。

不过在某些场景下，应用还需要在配置变化时获得通知，比如数据库连接的切换等，所以 Apollo 还提供了监听配置变化的功能，Java 示例如下：

```
Config config = ConfigService.getAppConfig();
config.addChangeListener(new ConfigChangeListener() {
  @Override
  public void onChange(ConfigChangeEvent changeEvent) {
    for (String key : changeEvent.changedKeys()) {
      ConfigChange change = changeEvent.getChange(key);
      System.out.println(String.format(
        "Found change - key: %s, oldValue: %s, newValue: %s, changeType: %s",
        change.getPropertyName(), change.getOldValue(),
        change.getNewValue(), change.getChangeType()));
    }
  }
});
```

3.7 Spring 集成样例

Apollo 和 Spring 也可以很方便地集成，只需要标注 `@EnableApolloConfig` 后就可以通过 `@Value` 获取配置信息：

```
@Configuration
@EnableApolloConfig
public class AppConfig {}
```

```
@Component
public class SomeBean {
    //timeout 的值会自动更新
    @Value("${request.timeout:200}")
    private int timeout;
}
```

6.2.5 4、Apollo in depth

通过上面的介绍，相信大家已经对 Apollo 有了一个初步的了解，并且相信已经覆盖到了大部分的使用场景。

接下来会主要介绍 Apollo 的 cluster 管理（集群）、namespace 管理（命名空间）和对应的配置获取规则。

4.1 Core Concepts

在介绍高级特性前，我们有必要先来了解一下 Apollo 中的几个核心概念：

1. **application (应用)**
    - 这个很好理解，就是实际使用配置的应用，Apollo 客户端在运行时需要知道当前应用是谁，从而可以去获取对应的配置
    - 每个应用都需要有唯一的身份标识 -- appId，我们认为应用身份是跟着代码走的，所以需要在代码中配置，具体信息请参见Java 客户端使用指南。

2. **environment (环境)**
    - 配置对应的环境，Apollo 客户端在运行时需要知道当前应用处于哪个环境，从而可以去获取应用的配置
    - 我们认为环境和代码无关，同一份代码部署在不同的环境就应该能够获取到不同环境的配置
    - 所以环境默认是通过读取机器上的配置（server.properties 中的 env 属性）指定的，不过为了开发方便，我们也支持运行时通过 System Property 等指定，具体信息请参见Java 客户端使用指南。
3. **cluster (集群)**
    - 一个应用下不同实例的分组，比如典型的可以按照数据中心分，把上海机房的应用实例分为一个集群，把北京机房的应用实例分为另一个集群。
    - 对不同的 cluster，同一个配置可以有不一样的值，如 zookeeper 地址。
    - 集群默认是通过读取机器上的配置（server.properties 中的 idc 属性）指定的，不过也支持运行时通过 System Property 指定，具体信息请参见Java 客户端使用指南。
4. **namespace (命名空间)**
    - 一个应用下不同配置的分组，可以简单地把 namespace 类比为文件，不同类型的配置存放在不同的文件中，如数据库配置文件，RPC 配置文件，应用自身的配置文件等
    - 应用可以直接读取到公共组件的配置 namespace，如 DAL，RPC 等
    - 应用也可以通过继承公共组件的配置 namespace 来对公共组件的配置做调整，如 DAL 的初始数据库连接数

4.2 自定义 Cluster

【本节内容仅对应用需要对不同集群应用不同配置才需要，如没有相关需求，可以跳过本节】

比如我们有应用在 A 数据中心和 B 数据中心都有部署，那么如果希望两个数据中心的配置不一样的话，我们可以通过新建 cluster 来解决。

4.2.1 新建 Cluster

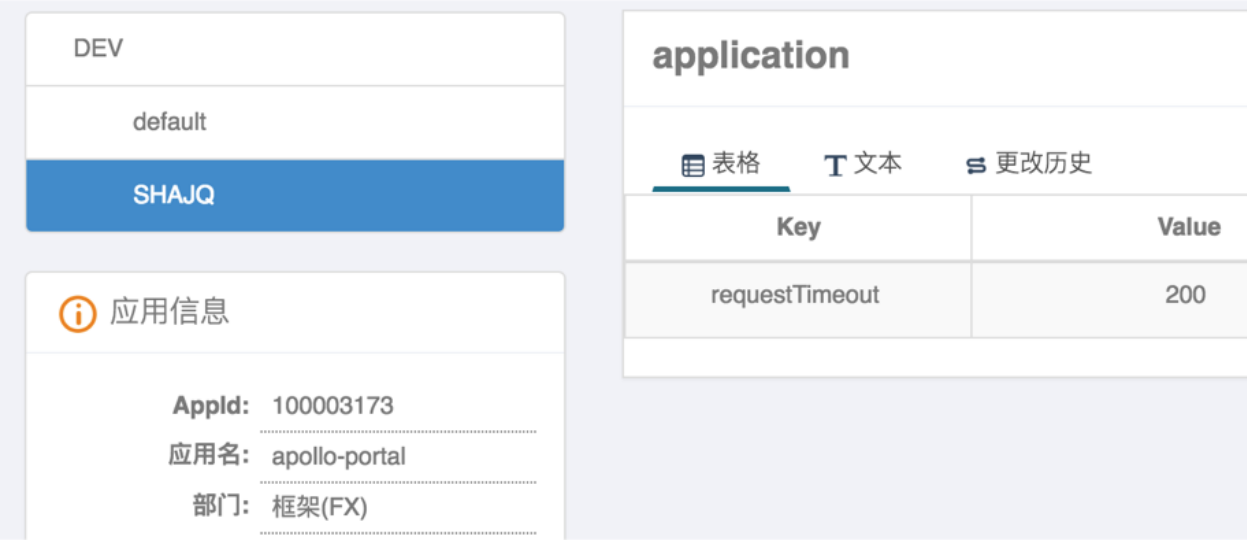
新建 Cluster 只有项目的管理员才有权限，管理员可以在页面左侧看到“添加集群”按钮。

点击后就进入到集群添加页面，一般情况下可以按照数据中心来划分集群，如 SHAJQ、SHAOY 等。

不过也支持自定义集群，比如可以为 A 机房的某一台机器和 B 机房的某一台机创建一个集群，使用一套配置。

4.2.2 在 Cluster 中添加配置并发布

集群添加成功后，就可以为该集群添加配置了，首先需要按照下图所示切换到 SHAJQ 集群，之后配置添加流程和3.2 添加/修改配置项一样，这里就不再赘述了。



4.2.3 指定应用实例所属的 Cluster

Apollo 会默认使用应用实例所在的数据中心作为 cluster，所以如果两者一致的话，不需要额外配置。

如果 cluster 和数据中心不一致的话，那么就需要通过 System Property 方式来指定运行时 cluster：

- -Dapollo.cluster=SomeCluster
- 这里注意 `apollo.cluster` 为全小写

4.3 自定义 Namespace

【本节仅对公共组件配置或需要多个应用共享配置才需要，如没有相关需求，可以跳过本节】

如果应用有公共组件（如 hermes-producer，cat-client 等）供其它应用使用，就需要通过自定义 namespace 来实现公共组件的配置。

4.3.1 新建 Namespace

以 hermes-producer 为例，需要先新建一个 namespace，新建 namespace 只有项目的管理员才有权限，管理员可以在页面左侧看到“添加 Namespace”按钮。

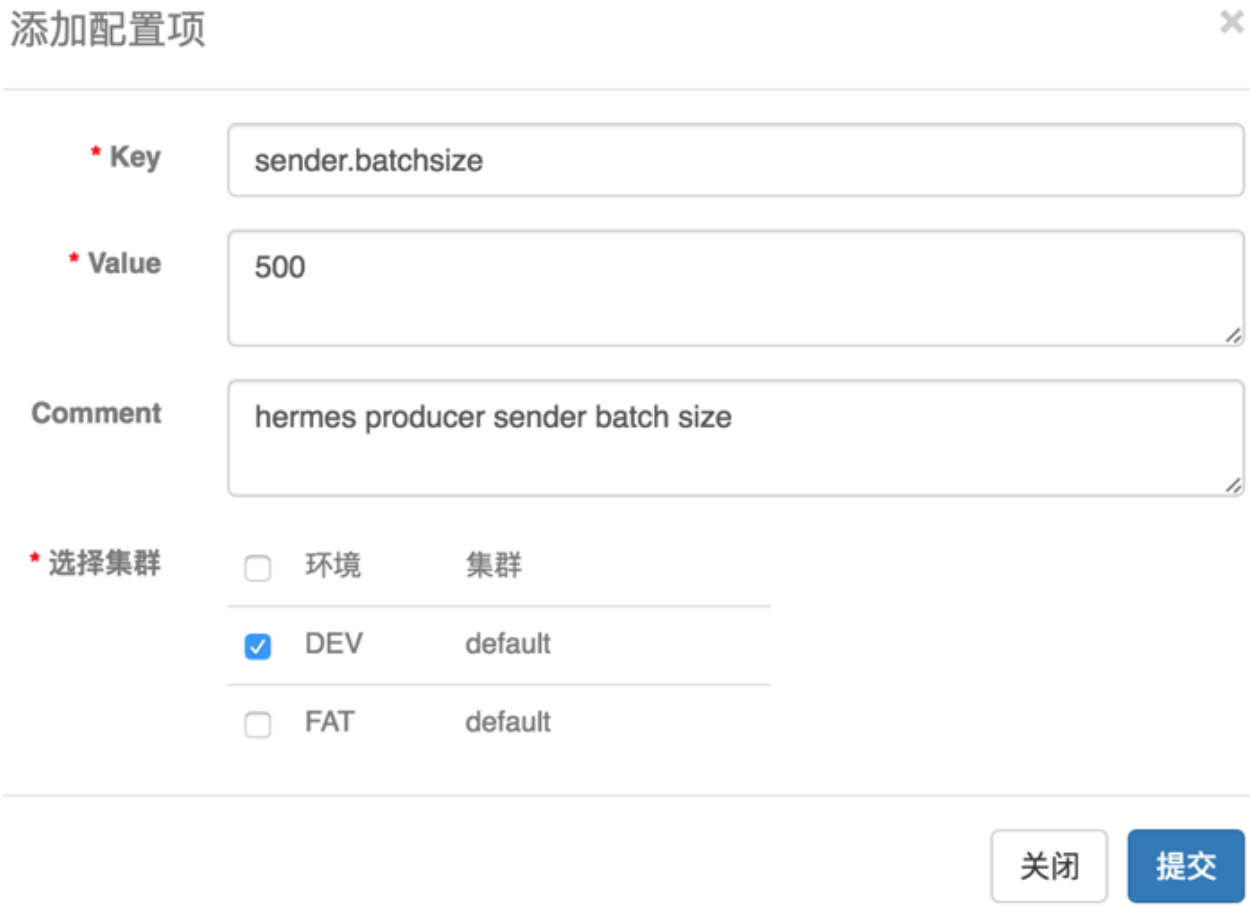
点击后就进入 namespace 添加页面，Apollo 会把应用所属的部门作为 namespace 的前缀，如 FX。

4.3.2 关联到环境和集群

Namespace 创建完，需要选择在哪些环境和集群下使用

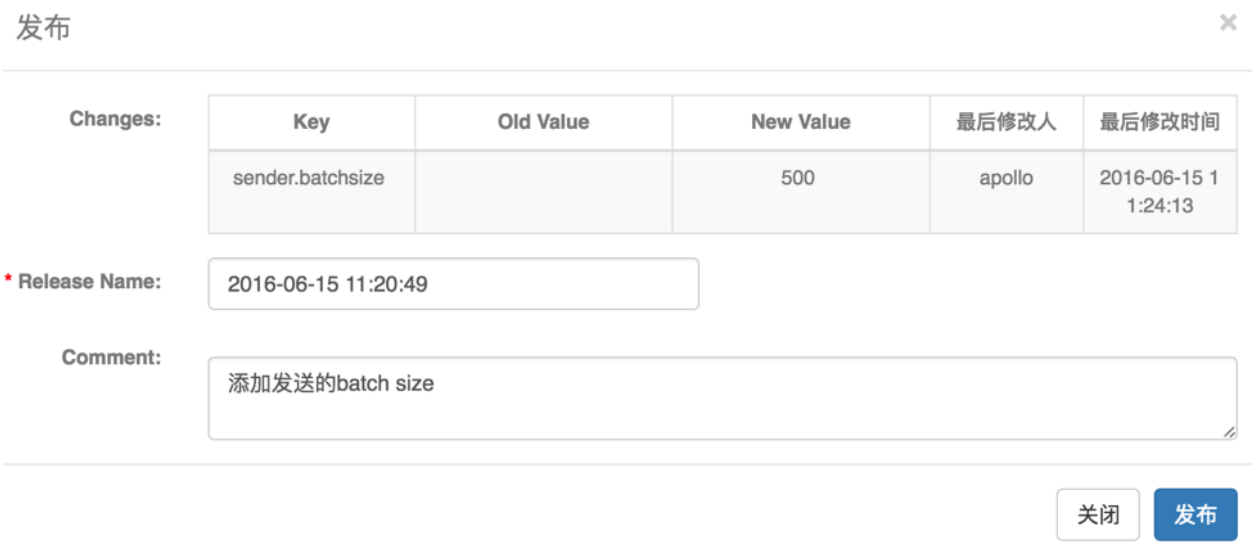
4.3.3 在 Namespace 中添加配置项

接下来在这个新建的 namespace 下添加配置项



添加完成后就能在 FX.Hermes.Producer 的 namespace 中看到配置。

4.3.4 发布 namespace 的配置



4.3.5 客户端获取 Namespace 配置

对自定义 namespace 的配置获取，稍有不同，需要程序传入 namespace 的名字。Apollo 客户端还支持和 Spring 整合，更多客户端使用说明请参见Java 客户端使用指南和.Net 客户端使用指南。

```
Config config = ConfigService.getConfig("FX.Hermes.Producer");
Integer defaultSenderBatchSize = 200;
Integer senderBatchSize = config.getIntProperty("sender.batchsize",␣
↪defaultSenderBatchSize);
```

4.3.6 客户端监听 Namespace 配置变化

```
Config config = ConfigService.getConfig("FX.Hermes.Producer");
config.addChangeListener(new ConfigChangeListener() {
  @Override
  public void onChange(ConfigChangeEvent changeEvent) {
    System.out.println("Changes for namespace " + changeEvent.getNamespace());
    for (String key : changeEvent.changedKeys()) {
      ConfigChange change = changeEvent.getChange(key);
      System.out.println(String.format(
        "Found change - key: %s, oldValue: %s, newValue: %s, changeType: %s",
        change.getPropertyName(), change.getOldValue(),
        change.getNewValue(), change.getChangeType()));
    }
  }
});
```

4.3.7 Spring 集成样例

```
@Configuration
@EnableApolloConfig("FX.Hermes.Producer")
public class AppConfig {}
```

```
@Component
public class SomeBean {
    //timeout 的值会自动更新
    @Value("${request.timeout:200}")
    private int timeout;
}
```

4.4 配置获取规则

【本节仅当应用自定义了集群或 namespace 才需要，如无相关需求，可以跳过本节】

在有了 cluster 概念后，配置的规则就显得重要了。

比如应用部署在 A 机房，但是并没有在 Apollo 新建 cluster，这个时候 Apollo 的行为是怎样的?

或者在运行时指定了 cluster=SomeCluster，但是并没有在 Apollo 新建 cluster，这个时候 Apollo 的行为是怎样的?

接下来就来介绍一下配置获取的规则。

4.4.1 应用自身配置的获取规则

当应用使用下面的语句获取配置时,我们称之为获取应用自身的配置,也就是应用自身的 application namespace 的配置。

```
Config coig = ConfigService.getAppConfig();
```

对这种情况的配置获取规则，简而言之如下:

1. 首先查找运行时 cluster 的配置（通过 apollo.cluster 指定）
2. 如果没有找到，则查找数据中心 cluster 的配置
3. 如果还是没有找到，则返回默认 cluster 的配置

图示如下:

所以如果应用部署在 A 数据中心，但是用户没有在 Apollo 创建 cluster，那么获取的配置就是默认 cluster（default）的。

如果应用部署在 A 数据中心，同时在运行时指定了 SomeCluster，但是没有在 Apollo 创建 cluster，那么获取的配置就是 A 数据中心 cluster 的配置，如果 A 数据中心 cluster 没有配置的话，那么获取的配置就是默认 cluster（default）的。

4.4.2 公共组件配置的获取规则

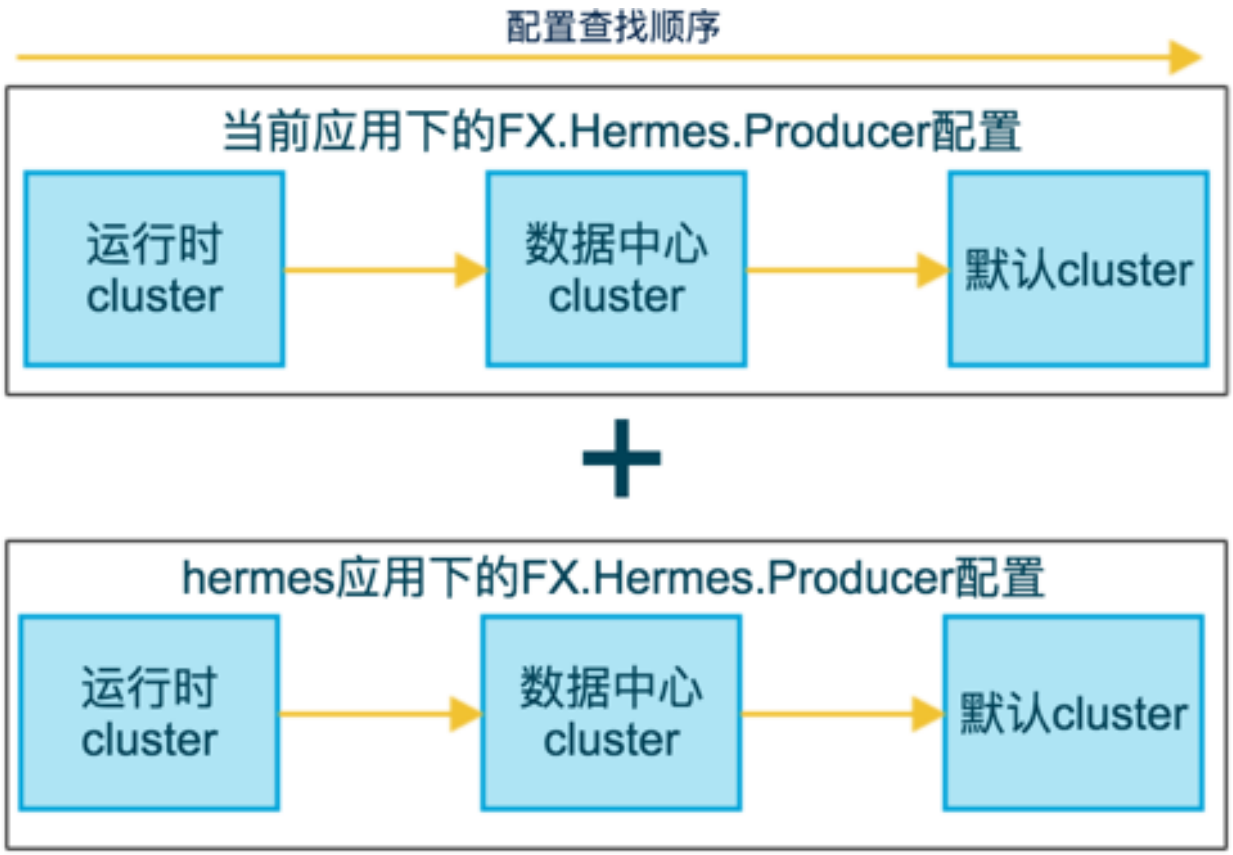
以 `FX.Hermes.Producer` 为例，hermes producer 是 hermes 发布的公共组件。当使用下面的语句获取配置时，我们称之为获取公共组件的配置。

```
Config config = ConfigService.getConfig("FX.Hermes.Producer");
```

对这种情况的配置获取规则，简而言之如下：

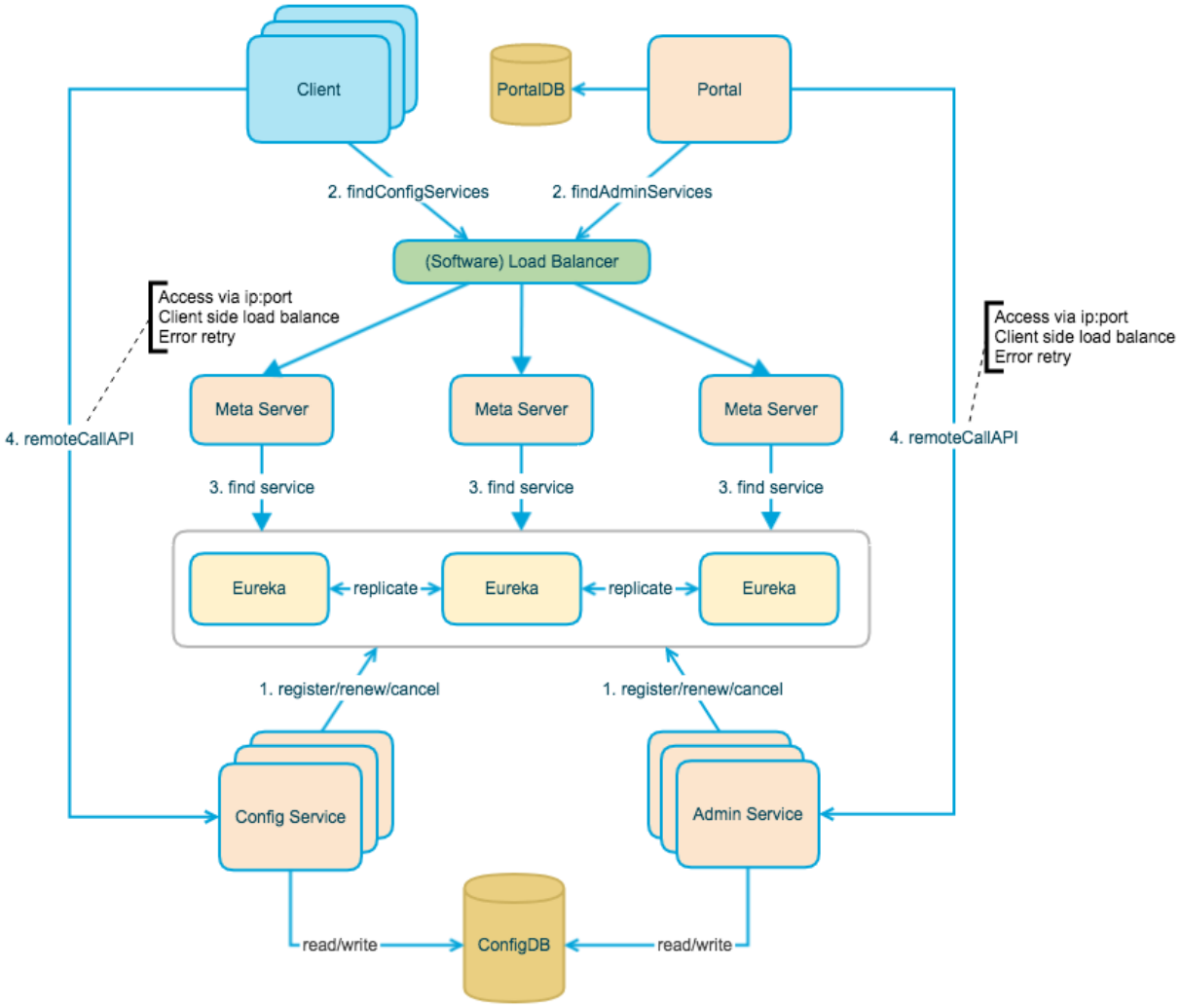
1. 首先获取当前应用下的 `FX.Hermes.Producer` namespace 的配置
2. 然后获取 hermes 应用下 `FX.Hermes.Producer` namespace 的配置
3. 上面两部分配置的并集就是最终使用的配置，如有 key 一样的部分，以当前应用优先

图示如下：



通过这种方式，就实现了对框架类组件的配置管理，框架组件提供方提供配置的默认值，应用如果有特殊需求，可以自行覆盖。

4.5 总体设计



上图简要描述了 Apollo 的总体设计，我们可以从下往上看：

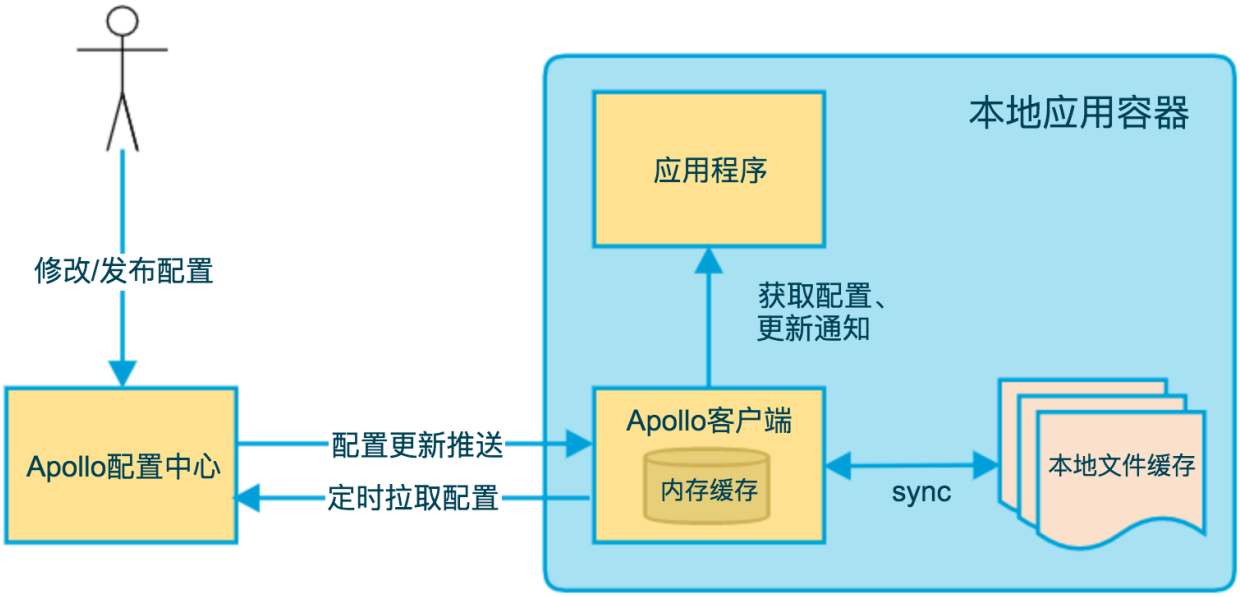
- Config Service 提供配置的读取、推送等功能，服务对象是 Apollo 客户端
- Admin Service 提供配置的修改、发布等功能，服务对象是 Apollo Portal（管理界面）
- Config Service 和 Admin Service 都是多实例、无状态部署，所以需要将自己注册到 Eureka 中并保持心跳
- 在 Eureka 之上我们架了一层 Meta Server 用于封装 Eureka 的服务发现接口
- Client 通过域名访问 Meta Server 获取 Config Service 服务列表（IP+Port），而后直接通过 IP+Port 访问服务，同时在 Client 侧会做 load balance、错误重试
- Portal 通过域名访问 Meta Server 获取 Admin Service 服务列表（IP+Port），而后直接通过 IP+Port 访问服务，同时在 Portal 侧会做 load balance、错误重试
- 为了简化部署，我们实际上会把 Config Service、Eureka 和 Meta Server 三个逻辑角色部署在同一个 JVM 进程中

4.5.1 Why Eureka

为什么我们采用 Eureka 作为服务注册中心，而不是使用传统的 zk、etcd 呢？我大致总结了一下，有以下几方面的原因：

- 它提供了完整的 Service Registry 和 Service Discovery 实现
  - 首先是提供了完整的实现，并且也经受住了 Netflix 自己的生产环境考验，相对使用起来会比较省心。
- 和 Spring Cloud 无缝集成
  - 我们的项目本身就使用了 Spring Cloud 和 Spring Boot，同时 Spring Cloud 还有一套非常完善的开源代码来整合 Eureka，所以使用起来非常方便。
  - 另外，Eureka 还支持在我们应用自身的容器中启动，也就是说我们的应用启动完之后，既充当了 Eureka 的角色，同时也是服务的提供者。这样就极大的提高了服务的可用性。
  - **这一点是我们选择 Eureka 而不是 zk、etcd 等的主要原因，为了提高配置中心的可用性和降低部署复杂度，我们需要尽可能地减少外部依赖**。
- Open Source
  - 最后一点是开源，由于代码是开源的，所以非常便于我们了解它的实现原理和排查问题。

4.6 客户端设计



上图简要描述了 Apollo 客户端的实现原理：

1. 客户端和服务端保持了一个长连接，从而能第一时间获得配置更新的推送。
2. 客户端还会定时从 Apollo 配置中心服务端拉取应用的最新配置。
   - 这是一个 fallback 机制，为了防止推送机制失效导致配置不更新
   - 客户端定时拉取会上报本地版本，所以一般情况下，对于定时拉取的操作，服务端都会返回 304 - Not Modified

- 定时频率默认为每 5 分钟拉取一次，客户端也可以通过在运行时指定 System Property: `apollo.refreshInterval` 来覆盖，单位为分钟。

3. 客户端从 Apollo 配置中心服务端获取到应用的最新配置后，会保存在内存中
4. 客户端会把从服务端获取到的配置在本地文件系统缓存一份
   - 在遇到服务不可用，或网络不通的时候，依然能从本地恢复配置
5. 应用程序从 Apollo 客户端获取最新的配置、订阅配置更新通知

4.6.1 配置更新推送实现

前面提到了 Apollo 客户端和服务端保持了一个长连接，从而能第一时间获得配置更新的推送。

长连接实际上我们是通过 Http Long Polling 实现的，具体而言：

- 客户端发起一个 Http 请求到服务端
- 服务端会保持住这个连接 60 秒
  - 如果在 60 秒内有客户端关心的配置变化，被保持住的客户端请求会立即返回，并告知客户端有配置变化的 namespace 信息，客户端会据此拉取对应 namespace 的最新配置
  - 如果在 60 秒内没有客户端关心的配置变化，那么会返回 Http 状态码 304 给客户端
- 客户端在收到服务端请求后会立即重新发起连接，回到第一步

考虑到会有数万客户端向服务端发起长连，在服务端我们使用了 async servlet(Spring DeferredResult) 来服务 Http Long Polling 请求。

4.7 可用性考虑

配置中心作为基础服务，可用性要求非常高，下面的表格描述了不同场景下 Apollo 的可用性：

6.2.6 5、Contribute to Apollo

Apollo 从开发之初就是以开源模式开发的，所以也非常欢迎有兴趣、有余力的朋友一起加入进来。

服务端开发使用的是 Java，基于 Spring Cloud 和 Spring Boot 框架。客户端目前提供了 Java 和.Net 两种实现。

Github 地址：https://github.com/ctripcorp/apollo

欢迎大家发起 Pull Request！

- 设计文档
  - Apollo 配置中心介绍
  - Apollo 配置中心设计
  - Apollo 核心概念之“Namespace”
- 部署文档
  - Quick Start
  - Docker 方式部署 Quick Start
  - 分布式部署指南
  - Apollo 源码解析（全）

- 开发文档
  - Apollo 开发指南
  - Code Styles
    - Eclipse Code Style
    - Intellij Code Style
  - Portal 实现用户登录功能
  - 邮件模板样例
- 系统使用文档
  - Apollo 使用指南
  - Java 客户端使用指南
  - .Net 客户端使用指南
  - Go、Python、NodeJS、PHP 等客户端使用指南
  - 其它语言客户端接入指南
  - Apollo 开放平台接入指南
  - Apollo 使用场景和示例代码
- FAQ
  - 常见问题回答
  - 部署 & 开发遇到的常见问题

6.3 Eureka

6.3.1 1、背景介绍

Eureka是 Netflix 开源的一款提供服务注册和发现的产品。

其官方文档中对自己的定义是：

> Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers. We call this service, the Eureka Server. Eureka also comes with a Java-based client component,the Eureka Client, which makes interactions with the service much easier. The client also has a built-in load balancer that does basic round-robin load balancing.

我们在研发 Apollo 配置中心时 (ctripcorp/apollo)，考虑到配置中心是基础服务，有非常高的可用性要求，为了更好的支持服务动态扩容、缩容、失效剔除等特性，所以就选择了使用 Eureka 来提供服务注册和发现功能。

本着“当你选择一款开源产品后，你就应当对它负责，既要信任它又要挑战它”的原则，我花了一些时间比较深入的研究了 Eureka 的实现细节（好在 Eureka 的实现短小精悍，通读源码也没花太多时间），今天就来详细介绍一下。

6.3.2 2、Why Eureka?

那么为什么我们在项目中使用了 Eureka 呢？我大致总结了一下，有以下几方面的原因：

1）它提供了完整的 **Service Registry** 和 **Service Discovery** 实现

首先是提供了完整的实现，并且也经受住了 Netflix 自己的生产环境考验，相对使用起来会比较省心。

2）和 **Spring Cloud** 无缝集成

我们的项目本身就使用了 Spring Cloud 和 Spring Boot，同时 Spring Cloud 还有一套非常完善的开源代码来整合 Eureka，所以使用起来非常方便。

另外，Eureka 还支持在我们应用自身的容器中启动，也就是说我们的应用启动完之后，既充当了 Eureka 的角色，同时也是服务的提供者。这样就极大的提高了服务的可用性。

这一点是我们选择 **Eureka** 而不是 **zk**、**etcd** 等的主要原因，为了提高配置中心的可用性和降低部署复杂度，我们需要尽可能地减少外部依赖。

3）Open Source

最后一点是开源，由于代码是开源的，所以非常便于我们了解它的实现原理和排查问题。

6.3.3 3、Dive into Eureka

相信大家看到这里，已经对 Eureka 有了一个初步的认识，接下来我们就来深入了解下它吧 ~

3.1 Overview

3.1.1 Basic Architecture

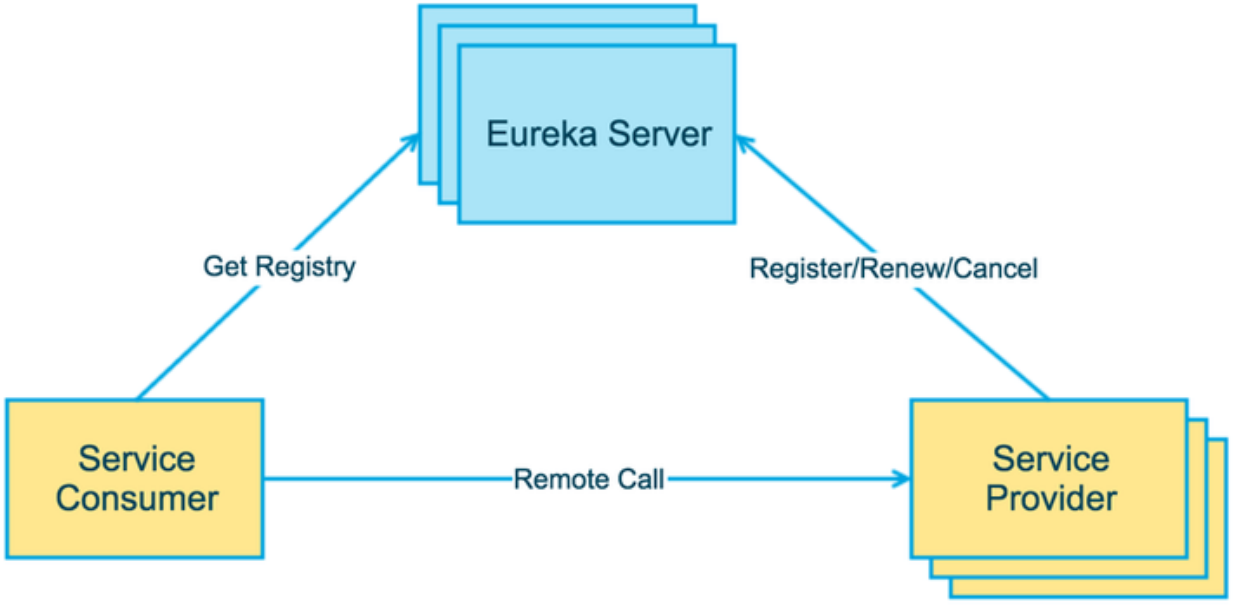


图 1

上图简要描述了 Eureka 的基本架构，由 3 个角色组成：

Eureka Server

提供服务注册和发现

Service Provider

服务提供方

将自身服务注册到 Eureka，从而使服务消费方能够找到

Service Consumer

服务消费方

从 Eureka 获取注册服务列表，从而能够消费服务

需要注意的是，上图中的 3 个角色都是逻辑角色。在实际运行中，这几个角色甚至可以是同一个实例，比如在我们项目中，Eureka Server 和 Service Provider 就是同一个 JVM 进程。

3.1.2 More in depth

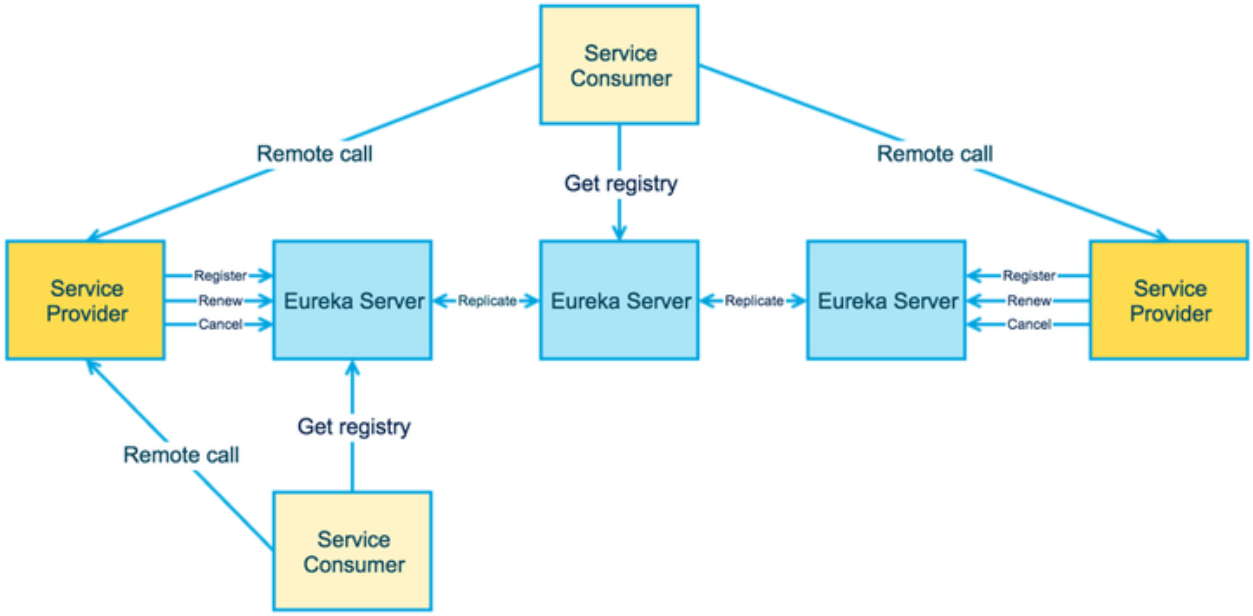


图 2

上图更进一步的展示了 3 个角色之间的交互。

1. Service Provider 会向 Eureka Server 做 Register（服务注册）、Renew（服务续约）、Cancel（服务下线）等操作。
2. Eureka Server 之间会做注册服务的同步，从而保证状态一致
3. Service Consumer 会向 Eureka Server 获取注册服务列表，并消费服务

3.2 Demo

为了给大家一个更直观的印象，我们可以通过一个简单的 demo 来实际运行一下，从而对 Eureka 有更好的了解。

3.2.1 Git Repository

Git 仓库：nobodyiam/spring-cloud-in-action

这个项目使用了 Spring Cloud 相关类库，包括：

- Spring Cloud Config
- Spring Cloud Eureka (Netflix)

3.2.2 准备工作

Demo 项目使用了 Spring Cloud Config 做配置，所以第一步先在本地启动 Config Server。

由于项目基于 Spring Boot 开发，所以直接运行com.nobodyiam.spring.cloud.in.action.config.ConfigServerApplication 即可。

3.2.3 Eureka Server Demo

Eureka Server 的 Demo 模块名是：eureka-server。

3.2.3.1 Maven 依赖

eureka-server 是一个基于 Spring Boot 的 Web 应用，我们首先需要做的就是在 pom 中引入 Spring Cloud Eureka Server 的依赖。

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-eureka-server</artifactId><version>1.2.0.
↪RELEASE</version>
</dependency>
```

3.2.3.2 启用 Eureka Server

启用 Eureka Server 非常简单，只需要加上 @EnableEurekaServer 即可。

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServiceApplication {
public static void main(String[] args{
SpringApplication.run(EurekaServiceApplication.class, args);
}
}
```

做完以上配置后，启动应用，Eureka Server 就开始工作了！

启动完，打开http://localhost:8761，就能看到启动成功的画面了。

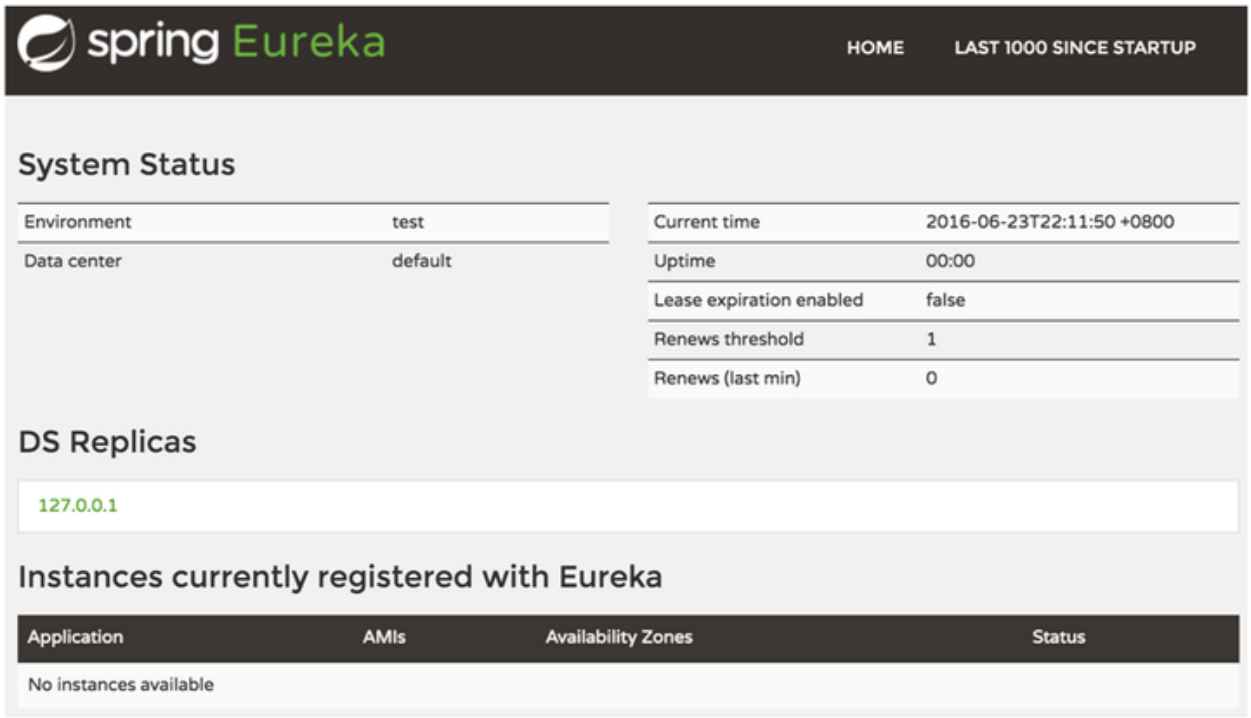


图3

3.2.4 Service Provider and Service Consumer Demo

Service Provider 的 Demo 模块名是：reservation-service。

Service Consumer 的 Demo 模块名是：reservation-client。

3.2.4.1 Maven 依赖

reservation-service 和 reservation-client 都是基于 Spring Boot 的 Web 应用，我们首先需要做的就是在 pom 中引入 Spring Cloud Eureka 的依赖。

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-eureka</artifactId>    <version>1.2.0.
↪RELEASE</version>
</dependency>
```

3.2.4.2 启动 Service Provider

启用 Service Provider 非常简单，只需要加上 @EnableDiscoveryClient 即可。

```
@EnableDiscoveryClient
@SpringBootApplication
public class ReservationServiceApplication {
public static void main(String[] args) {
↪newSpringApplicationBuilder(ReservationServiceApplication.class)
.run(args);
}
}
```

做完以上配置后，启动应用，Server Provider 就开始工作了！

启动完，打开http://localhost:8761，就能看到服务已经注册到 Eureka Server 了。

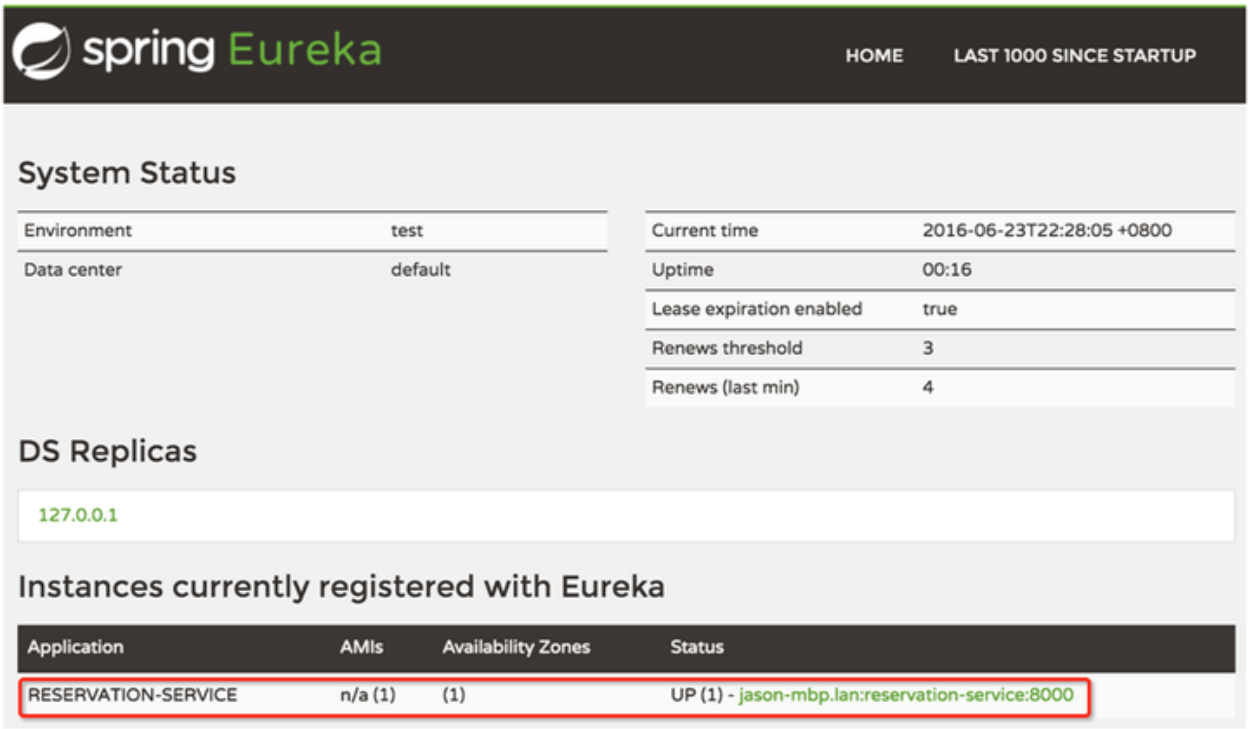


图 4

3.2.4.3 启动 Service Consumer

启动 Service Consumer 其实和 Service Provider 一样，因为本质上 Eureka 提供的客户端是不区分 Provider 和 Consumer 的，一般情况下，Provider 同时也会是 Consumer。

```
@EnableDiscoveryClient
@SpringBootApplication
public class ReservationClientApplication {
@Bean
CommandLineRunner runner(DiscoveryClient dc) {
return args -> {
dc.getInstances("reservation-service")
```

(下页继续)

(续上页)

```
.forEach(si -> System.out.println(String.format(
"Found %s %s:%s", si.getServiceId(), si.getHost(), si.getPort())));
};
}
public static void main(String[] args) {    SpringApplication.
↪run(ReservationClientApplication.class, args);
}
}
```

上述代码中通过 dc.getInstances(“reservation-service”) 就能获取到当前所有注册的 reservation-service 服务。

3.3 Eureka Server 实现细节

看了前面的 demo，我们已经初步领略到了 Spring Cloud 和 Eureka 的强大之处，通过短短几行配置就实现了服务注册和发现！

相信大家一定想了解 Eureka 是如何实现的吧，所以接下来我们继续 Dive！首先来看下 Eureka Server 的几个对外接口实现。

3.3.1 Register

首先来看 Register（服务注册），这个接口会在 Service Provider 启动时被调用来实现服务注册。同时，当 Service Provider 的服务状态发生变化时（如自身检测认为 Down 的时候），也会调用来更新服务状态。

接口实现比较简单，如下图所示。

1. ApplicationResource 类接收 Http 服务请求，调用 PeerAwareInstanceRegistryImpl 的 register 方法
2. PeerAwareInstanceRegistryImpl 完成服务注册后，调用 replicateToPeers 向其它 Eureka Server 节点（Peer）做状态同步（异步操作）

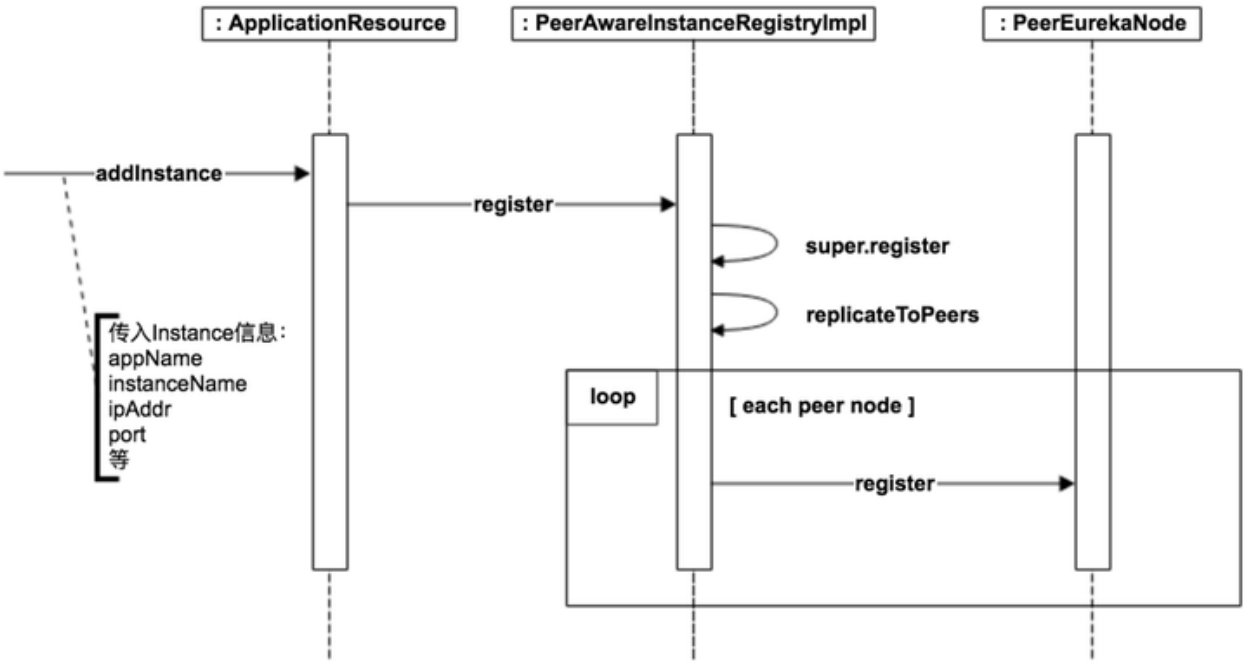


图 5

注册的服务列表保存在一个嵌套的 hash map 中：

- 第一层 hash map 的 key 是 app name，也就是应用名字
- 第二层 hash map 的 key 是 instance name，也就是实例名字

以3.2.4.2中的截图为例，RESERVATION-SERVICE 就是 app name，jason-mbp.lan:reservation-service:8000 就是 instance name。

Hash map 定义如下：

private final ConcurrentHashMap<String, Map<String, Lease» registry =new ConcurrentHashMap<String, Map<String, Lease»();

3.3.2 Renew

Renew（服务续约）操作由 Service Provider 定期调用，类似于 heartbeat。主要是用来告诉 Eureka Server Service Provider 还活着，避免服务被剔除掉。接口实现如下图所示。

可以看到，接口实现方式和 register 基本一致：首先更新自身状态，再同步到其它 Peer。

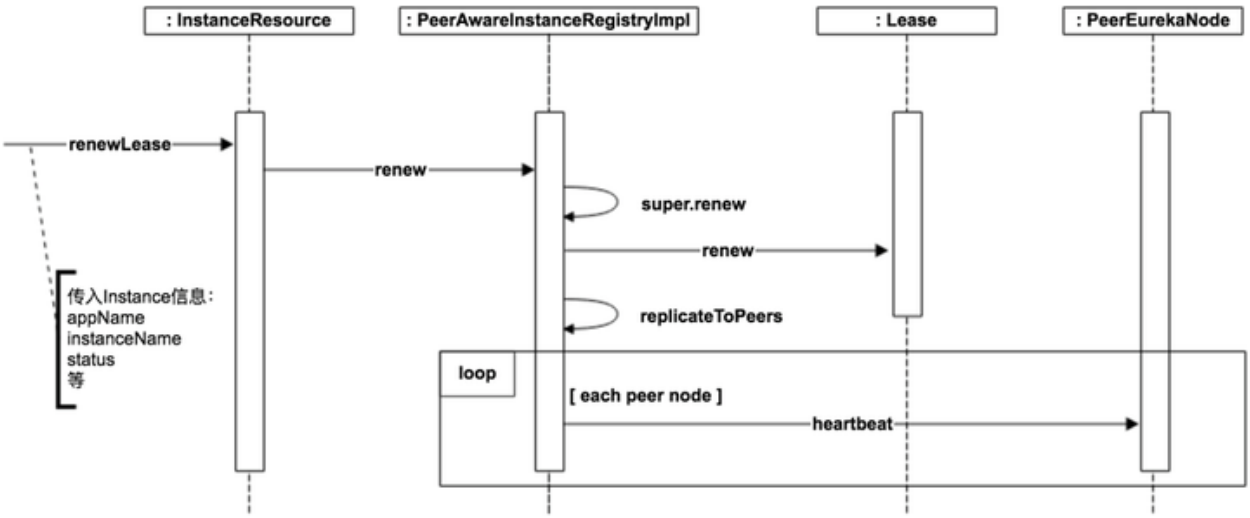


图 6

3.3.3 Cancel

Cancel（服务下线）一般在 Service Provider shut down 的时候调用，用来把自身的服务从 Eureka Server 中删除，以防客户端调用不存在的服务。接口实现如下图所示。

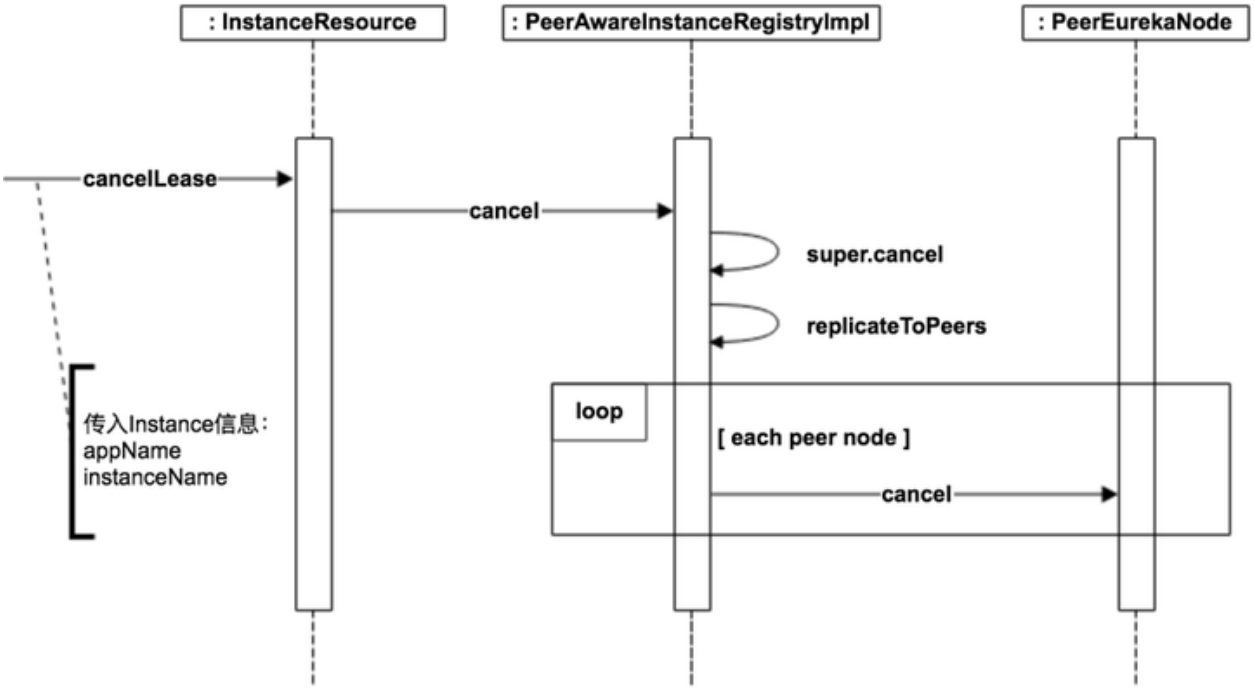


图 7

3.3.4 Fetch Registries

Fetch Registries 由 Service Consumer 调用，用来获取 Eureka Server 上注册的服务。

为了提高性能，服务列表在 Eureka Server 会缓存一份，同时每 30 秒更新一次。

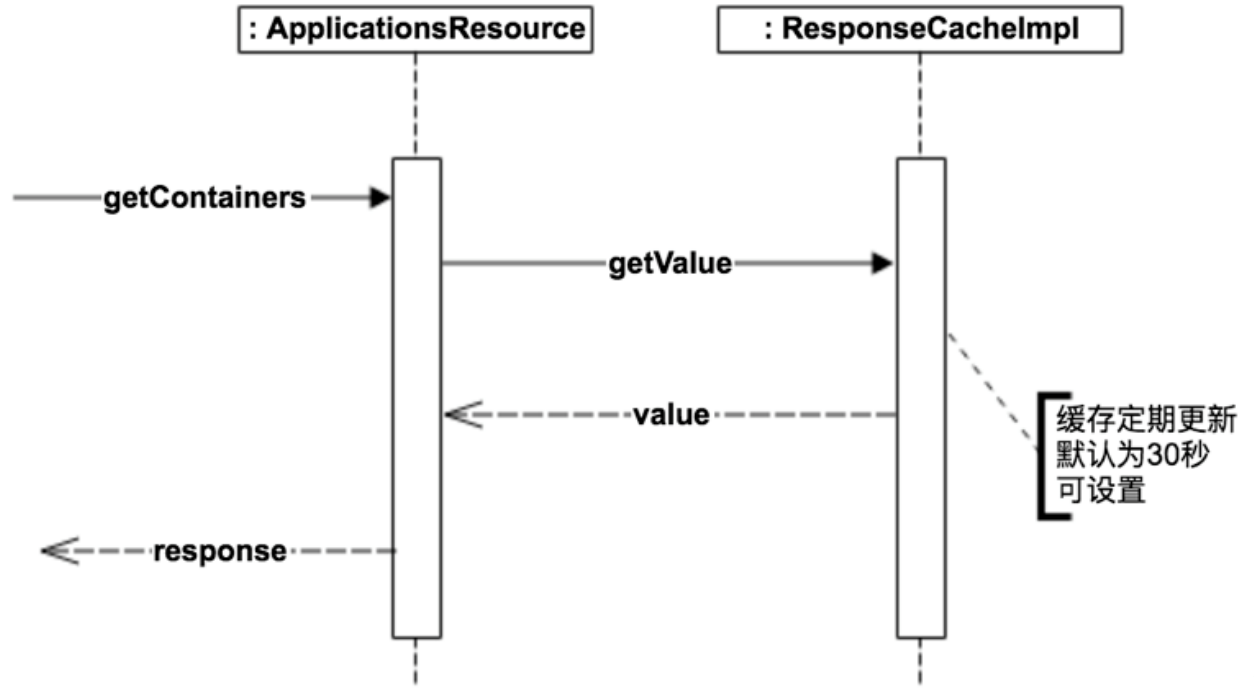


图 8

3.3.5 Eviction

Eviction（失效服务剔除）用来定期（默认为每 60 秒）在 Eureka Server 检测失效的服务，检测标准就是超过一定时间没有 Renew 的服务。

默认失效时间为 90 秒，也就是如果有服务超过 90 秒没有向 Eureka Server 发起 Renew 请求的话，就会被当做失效服务剔除掉。

失效时间可以通过 eureka.instance.leaseExpirationDurationInSeconds 进行配置，定期扫描时间可以通过 eureka.server.evictionIntervalTimerInMs 进行配置。

接口实现逻辑见下图：

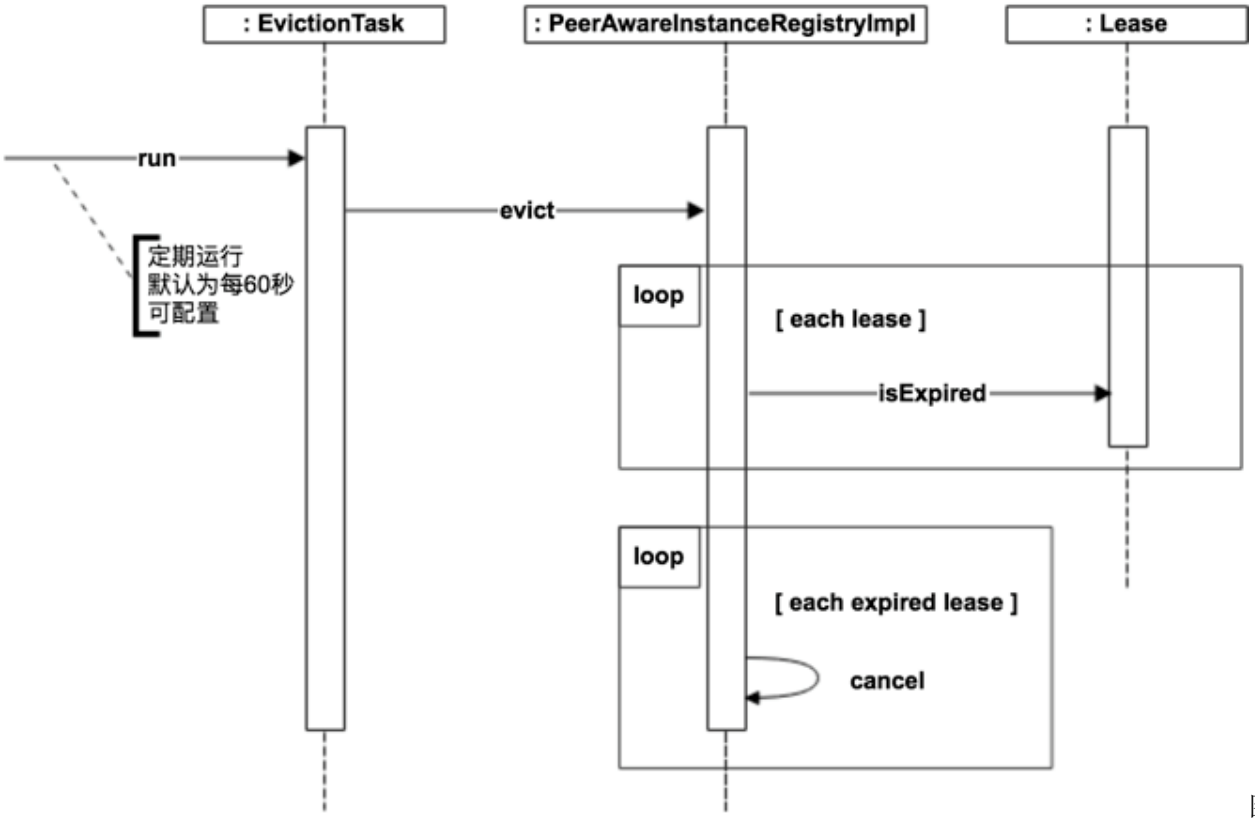


图 9

3.3.6 How Peer Replicates

在前面的 Register、Renew、Cancel 接口实现中，我们看到了都会有 replicateToPeers 操作，这个就是用来做 Peer 之间的状态同步。

通过这种方式，Service Provider 只需要通知到任意一个 Eureka Server 后就能保证状态会在所有的 Eureka Server 中得到更新。

具体实现方式其实很简单，就是接收到 Service Provider 请求的 Eureka Server，把请求再次转发到其它的 Eureka Server，调用同样的接口，传入同样的参数，除了会在 header 中标记 isReplication=true，从而避免重复的 replicate。

Peer 之间的状态是采用异步的方式同步的，所以不保证节点间的状态一定是一致的，不过基本能保证最终状态是一致的。

结合服务发现的场景，实际上也并不需要节点间的状态强一致。在一段时间内（比如 30 秒），节点 A 比节点 B 多一个服务实例或少一个服务实例，在业务上也是完全可以接受的（Service Consumer 侧一般也会实现错误重试和负载均衡机制）。

所以按照 CAP 理论，Eureka 的选择就是放弃 C，选择 AP。

3.3.7 How Peer Nodes are Discovered

那大家可能会有疑问，Eureka Server 是怎么知道有多少 Peer 的呢?

Eureka Server 在启动后会调用 EurekaClientConfig.getEurekaServerServiceUrls 来获取所有的 Peer 节点，并且会定期更新。定期更新频率可以通过 eureka.server.peerEurekaNodesUpdateIntervalMs 配置。

这个方法的默认实现是从配置文件读取，所以如果 Eureka Server 节点相对固定的话，可以通过在配置文件中配置来实现。

如果希望能更灵活的控制 Eureka Server 节点，比如动态扩容/缩容，那么可以 override getEurekaServerServiceUrls 方法，提供自己的实现，比如我们的项目中会通过数据库读取 Eureka Server 列表。

具体实现如下图所示：

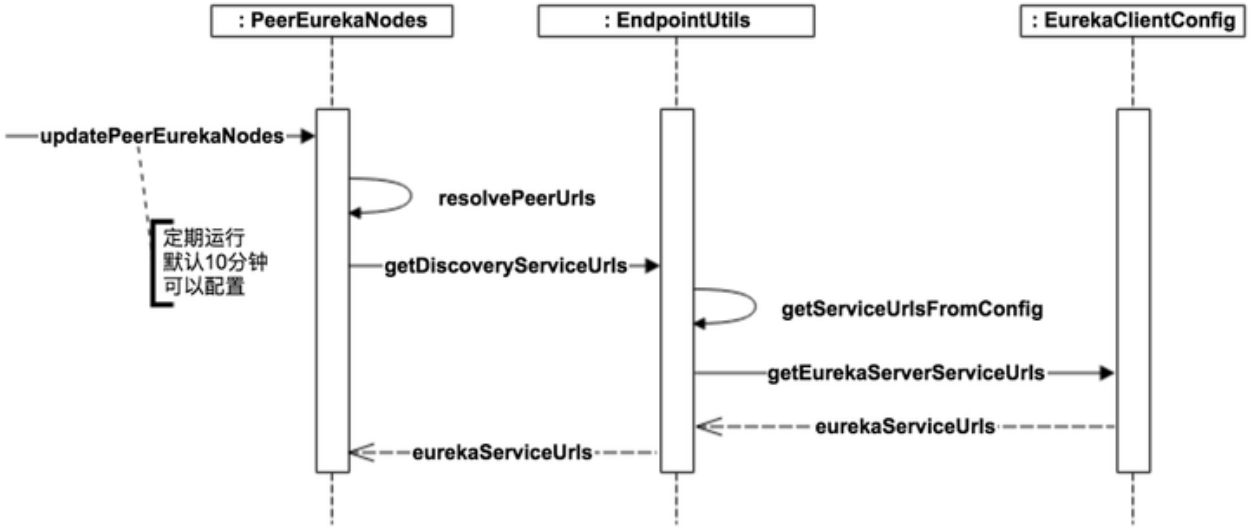


图 10

3.3.8 How New Peer Initializes

最后再来看一下一个新的 Eureka Server 节点加进来，或者 Eureka Server 重启后，如何来做初始化，从而能够正常提供服务。

具体实现如下图所示，简而言之就是启动时把自己当做是 Service Consumer 从其它 Peer Eureka 获取所有服务的注册信息。然后对每个服务，在自己这里执行 Register，isReplication=true，从而完成初始化。

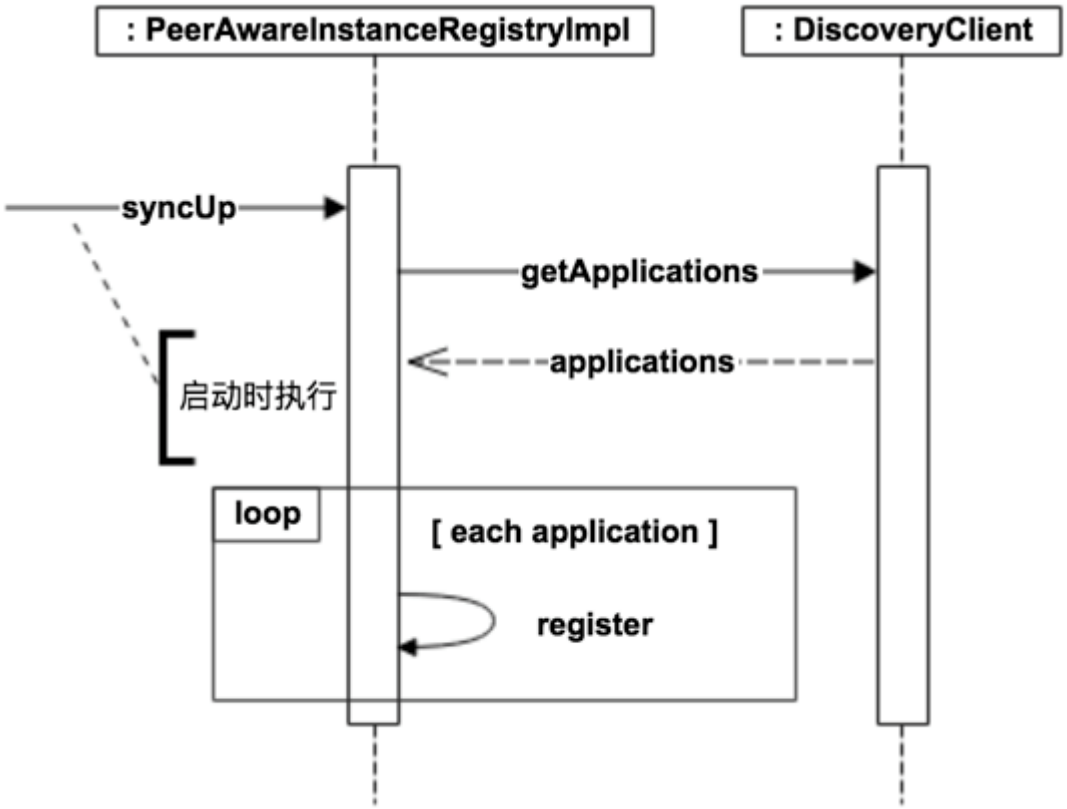


图 11

3.4 Service Provider 实现细节

现在来看下 Service Provider 的实现细节，主要就是 Register、Renew、Cancel 这 3 个操作。

3.4.1 Register

Service Provider 要对外提供服务，一个很重要的步骤就是把自己注册到 Eureka Server 上。

这部分的实现比较简单，只需要在启动时和实例状态变化时调用 Eureka Server 的接口注册即可。需要注意的是，需要确保配置 eureka.client.registerWithEureka=true。

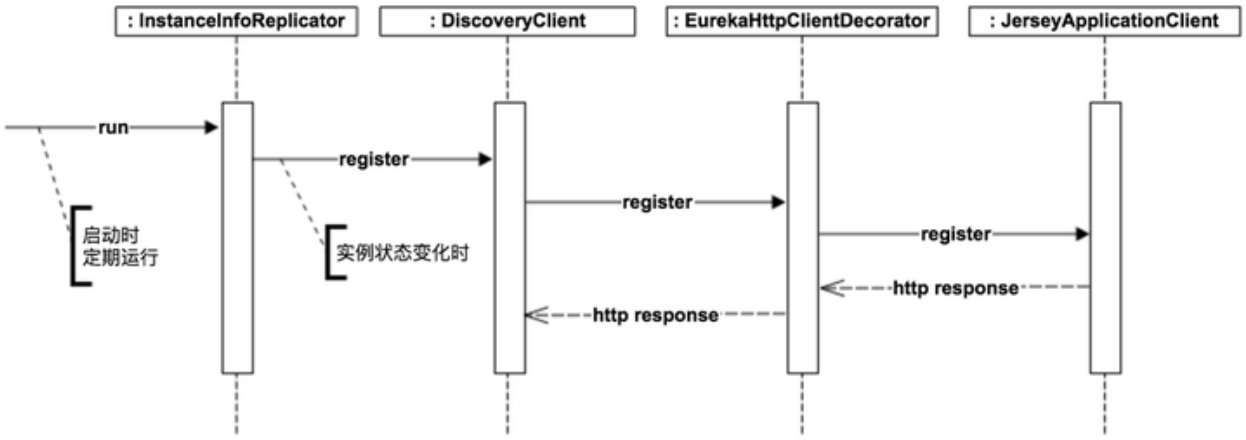


图 12

3.4.2 Renew

Renew 操作会在 Service Provider 端定期发起，用来通知 Eureka Server 自己还活着。这里有两个比较重要的配置需要注意一下：

1. instance.leaseRenewalIntervalInSeconds

Renew 频率。默认是 30 秒，也就是每 30 秒会向 Eureka Server 发起 Renew 操作。

1. instance.leaseExpirationDurationInSeconds

服务失效时间。默认是 90 秒，也就是如果 Eureka Server 在 90 秒内没有接收到来自 Service Provider 的 Renew 操作，就会把 Service Provider 剔除。

具体实现如下：

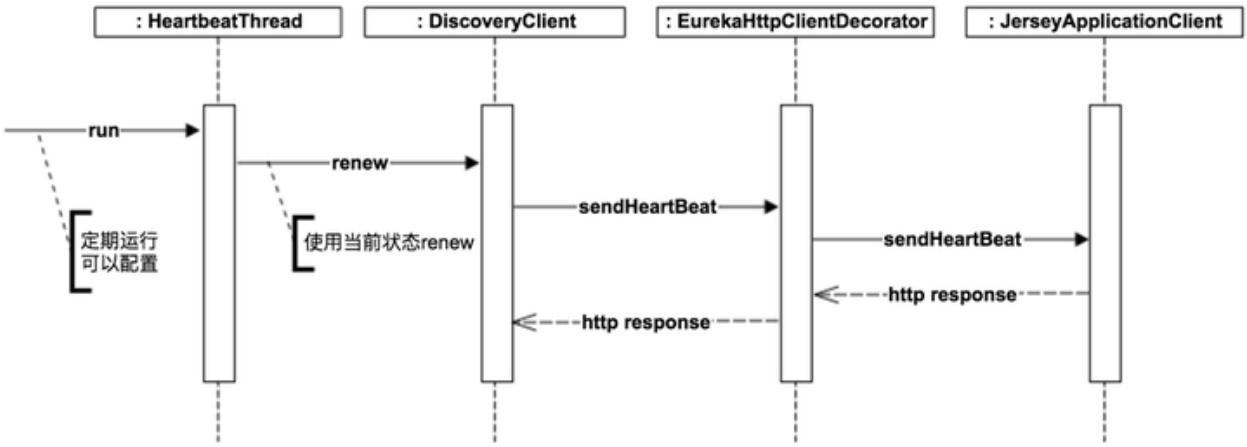
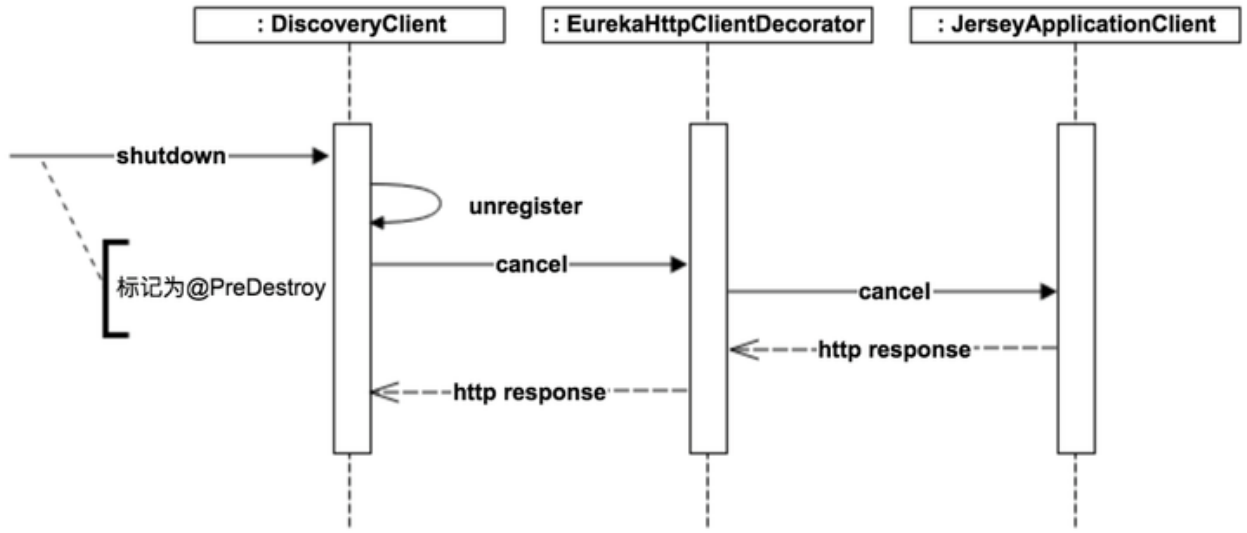


图 13

3.4.3 Cancel

在 Service Provider 服务 shut down 的时候，需要及时通知 Eureka Server 把自己剔除，从而避免客户端调用已经下线的服务。

逻辑本身比较简单，通过对方法标记 @PreDestroy，从而在服务 shut down 的时候会被触发。



图

14

3.4.4 How Eureka Servers are Discovered

这里大家疑问又来了，Service Provider 是怎么知道 Eureka Server 的地址呢?

其实这部分的主体逻辑和 3.3.7 How Peer Nodes are Discovered 几乎是一样的。

也是默认从配置文件读取，如果需要更灵活的控制，可以通过 override getEurekaServerServiceUrls 方法来提供自己的实现。定期更新频率可以通过 eureka.client.eurekaServiceUrlPollIntervalSeconds 配置。

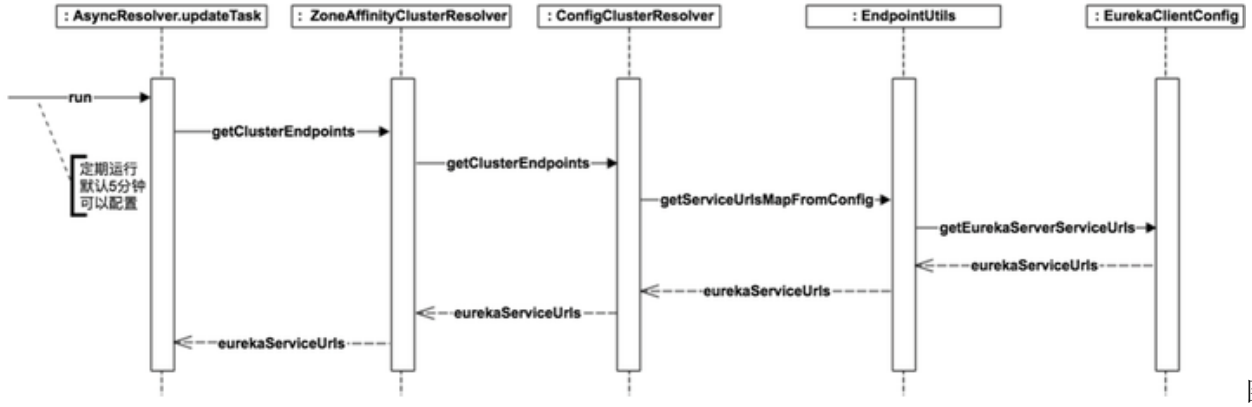


图 15

3.5 Service Consumer 实现细节

Service Consumer 这块的实现相对就简单一些，因为它只涉及到从 Eureka Server 获取服务列表和更新服务列表。

3.5.1 Fetch Service Registries

Service Consumer 在启动时会从 Eureka Server 获取所有服务列表，并在本地缓存。需要注意的是，需要确保配置 eureka.client.shouldFetchRegistry=true。

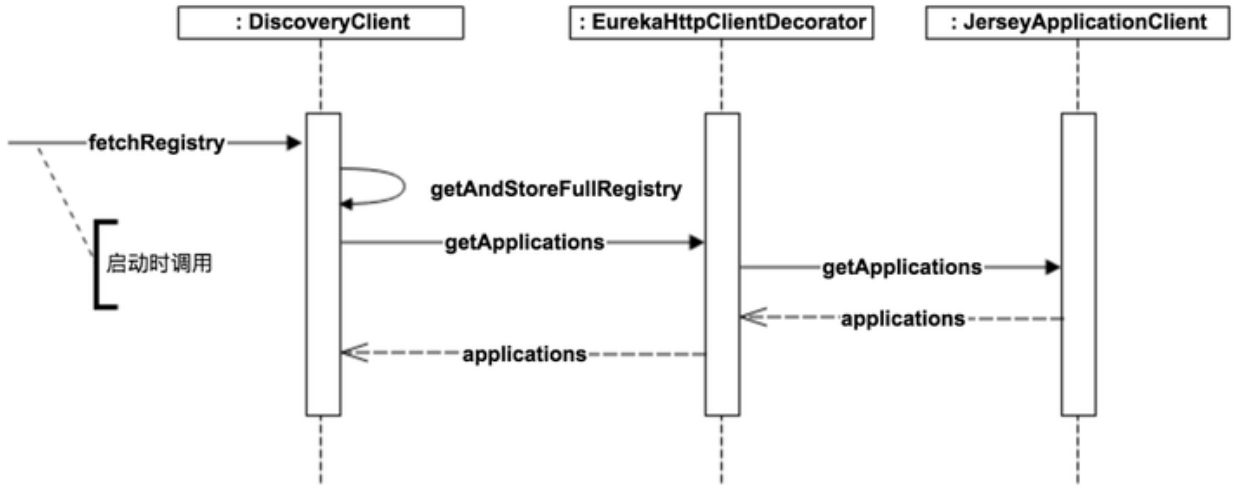


图 16

3.5.2 Update Service Registries

由于在本地有一份缓存，所以需要定期更新，定期更新频率可以通过 eureka.client.registryFetchIntervalSeconds 配置。

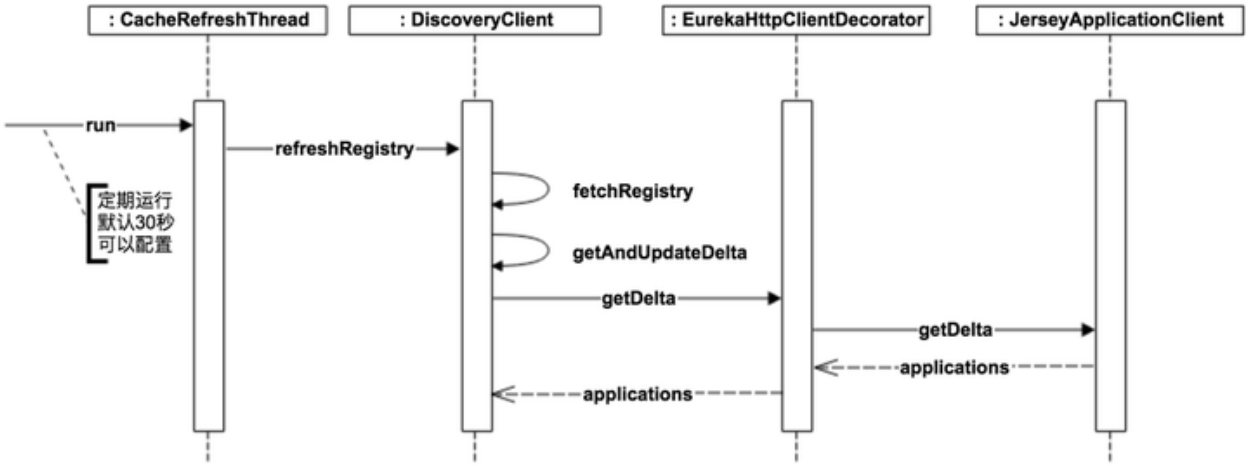


图 17

3.5.3 How Eureka Servers are Discovered

Service Consumer 和 Service Provider 一样，也有一个如何知道 Eureka Server 地址的问题。

其实由于 Service Consumer 和 Service Provider 本质上是同一个 Eureka 客户端，所以这部分逻辑是一样的，这里就不再赘述了。详细信息见 3.4.4 节。

6.4 Log

6.4.1 日志查看

查找关键日志 grep

Linux 查找文件内容的常用命令方法

```
从文件内容查找匹配指定字符串的行： $ grep " 被查找的字符串" 文件名

从文件内容查找与正则表达式匹配的行： $ grep -e “正则表达式” 文件名

查找时不区分大小写： $ grep -i " 被查找的字符串" 文件名

查找匹配的行数： $ grep -c " 被查找的字符串" 文件名

从文件内容查找不匹配指定字符串的行： $ grep -v " 被查找的字符串" 文件名

从根目录开始查找所有扩展名为.log 的文本文件，并找出包含”ERROR”的行 find / -type f -name "*.log
↪" | xargs grep "ERROR"

如果需要查找的内容包含特殊符号，比如$ 等等，grep 要加参数 find ./ -name "*.php" | xargs grep -
↪F ' 要查找的内容'
```

linux 查看日志文件内容命令 tail、cat、tac、head、echo

```
tail -f test.log
```

你会看到屏幕不断有内容被打印出来. 这时候中断第一个进程 Ctrl-C,

linux 如何显示一个文件的某几行 (中间几行)

从第 3000 行开始，显示 1000 行。即显示 3000~3999 行

```
cat filename | tail -n +3000 | head -n 1000
```

显示 1000 行到 3000 行

```
cat filename| head -n 3000 | tail -n +1000
```

注意两种方法的顺序

tail -n 1000：显示最后 1000 行 tail -n +1000：从 1000 行开始显示，显示 1000 行以后的 head -n 1000：显示前面 1000 行

用 sed 命令

```
sed -n '5,10p' filename
```

这样你就可以只查看文件的第 5 行到第 10 行。

例：`cat mylog.log | tail -n 1000` 输出 mylog.log 文件最后一千行

cat 主要有三大功能：

- 1. 一次显示整个文件。$ cat filename
- 2. 从键盘创建一个文件。$ cat > filename 只能创建新文件, 不能编辑已有文件.
- 3. 将几个文件合并为一个文件：$cat file1 file2 > file 参数：-n 或 --number 由 1 开始对所有输出的行数编号 -b 或 --number-nonblank 和 -n 相似，只不过对于空白行不编号 -s 或 --squeeze-blank 当遇到有连续两行以上的空白行，就代换为一行的空白行 -v 或 --show-nonprinting 例：

```
把 textfile1 的档案内容加上行号后输入 textfile2 这个档案里
cat -n textfile1 > textfile2

把 textfile1 和 textfile2 的档案内容加上行号（空白行不加）之后将内容附加到 textfile3 里。
cat -b textfile1 textfile2 >> textfile3

把 test.txt 文件扔进垃圾箱，赋空值 test.txt
cat /dev/null > /etc/test.txt
注意：> 意思是创建，>> 是追加。千万不要弄混了。
```

tac (反向列示)

tac 是将 cat 反写过来，所以他的功能就跟 cat 相反，cat 是由第一行到最后一行连续显示在萤幕上，而 tac 则是由最后一行到第一行反向在萤幕上显示出来！

在 Linux 中 echo 命令用来在标准输出上显示一段字符，比如：echo ”the echo command test!”

这个就会输出“the echo command test!”这一行文字！

echo ”the echo command test!”>a.sh 这个就会在 a.sh 文件中输出“the echo command test!”这一行文字！该命令的一般格式为：echo [-n] 字符串其中选项 n 表示输出文字后不换行；字符串能加引号，也能不加引号。用 echo 命令输出加引号的字符串时，将字符串原样输出；用 echo 命令输出不加引号的字符串时，将字符串中的各个单词作为字符串输出，各字符串之间用一个空格分割。

6.5 开发踩坑

1、程序的可用性：

程序的设计很多常见的算法都是空间换时间。增加空间复杂度，减少时间复杂度，提高用户体验度。比如参数过滤，使用 bloom 过滤器，底层需要增加一个非常大的数组进行存储错误的参数值和可能错误的参数值。但是能够过滤大部分的错误参数。

2、错误处理：

程序设计需要对一些考虑到的边界问题进行处理，需要尽可能减少错误率。

例如数据库键重复，数据库底层会爆出键重复的异常，上层虽然有捕捉，但反馈结果是报错。这种设计是不合理的，这种应该在数据的入口就进行排查。程序设计的错误不仅仅是用户级别，也在监控级别。不仅要尽量减少用户反馈的错误，也要尽量减少监控能检测的错误。

真的错误是指程序无法处理的错误，比如系统奔溃。程序的错误一定需要提前考虑并进行处理。

3、技术选型

- 场景适配

4、方案设计

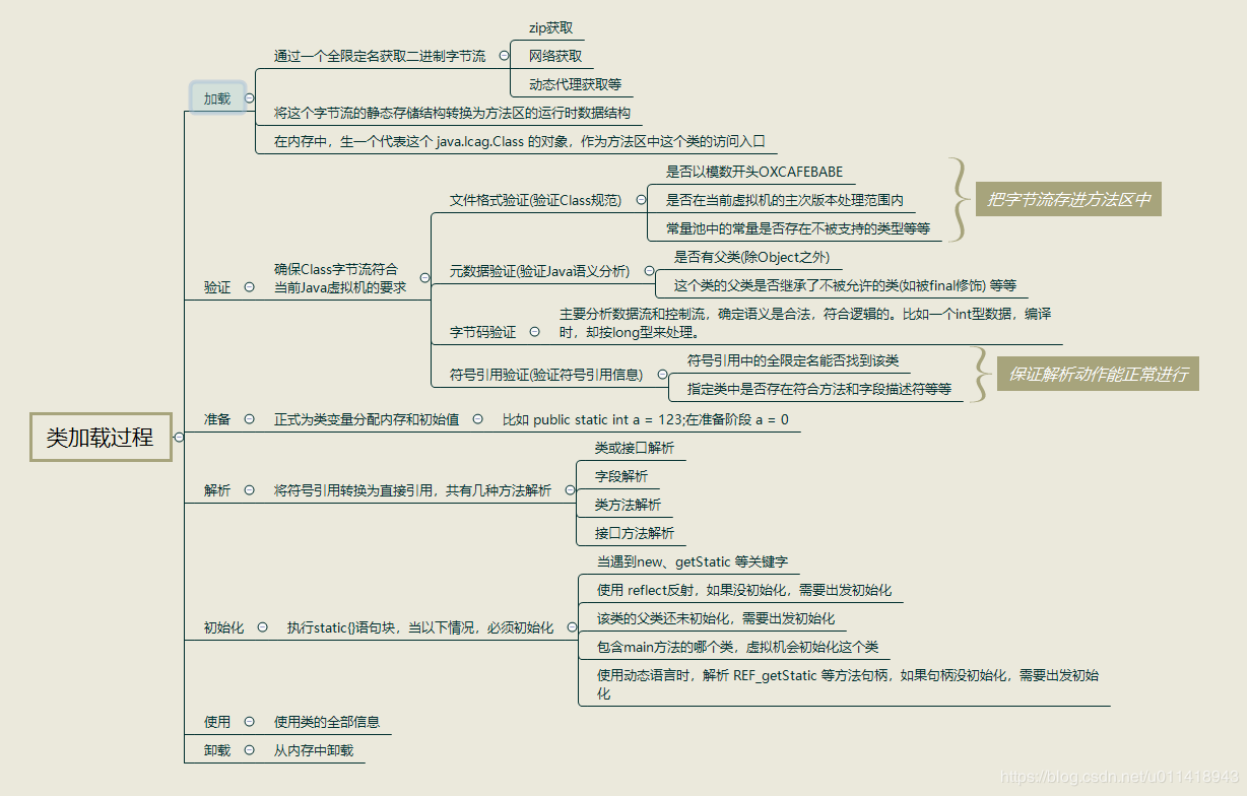
- 需求切入点
- 需求切割

CHAPTER 7

Java 开发

7.1 JVM

7.1.1 类加载过程



类的初始化

初始化，为类的静态变量赋予正确的初始值，JVM 负责对类进行初始化，主要对类变量进行初始化。在 Java 中对类变量进行初始值设定有两种方式：

- ① 声明类变量时指定初始值
- ② 使用静态代码块为类变量指定初始值

JVM 初始化步骤

- 1、假如这个类还没有被加载和连接，则程序先加载并连接该类
- 2、假如该类的直接父类还没有被初始化，则先初始化其直接父类
- 3、假如类中有初始化语句，则系统依次执行这些初始化语句

初始化阶段时执行类构造器方法 () 的过程。

- 1、类构造器方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{} 块）中的语句合并产生的，编译器收集的顺序由语句在源文件中出现的顺序所决定。
- 2、类构造器方法与类的构造函数不同，它不需要显式地调用父类构造器，虚拟机会保证在子类的类构造器方法执行之前，父类的类构造器方法已经执行完毕，因此在虚拟机中第一个执行的类构造器方法的类一定是 java.lang.Object。
- 3、由于父类的类构造器方法方法先执行，也就意味着**父类中定义的静态语句块要优先于子类的变量赋值操作**。
- 4、类构造器方法对于类或者接口来说并不是必需的，如果一个类中没有静态语句块也没有对变量的赋值操作，那么编译器可以不为这个类生成类构造器方法。
- 5、接口中可能会有变量赋值操作，因此接口也会生成类构造器方法。但是接口与类不同，执行接口的类构造器方法不需要先执行父接口的类构造器方法。只有当父接口中定义的变量被使用时，父接口才会被初始化。另外，接口的实现类在初始化时也不会执行接口的类构造器方法。
- 6、虚拟机会保证一个类的类构造器方法在多线程环境中被正确地加锁和同步。如果有多个线程去同时初始化一个类，那么只会有一个线程去执行这个类的类构造器方法，其它线程都需要阻塞等待，直到活动线程执行类构造器方法完毕。如果在一个类的类构造器方法中有耗时很长的操作，那么就可能造成多个进程阻塞。

于初始化阶段虚拟机规范是严格规定了如下几种情况，如果类未初始化会对类进行初始化。

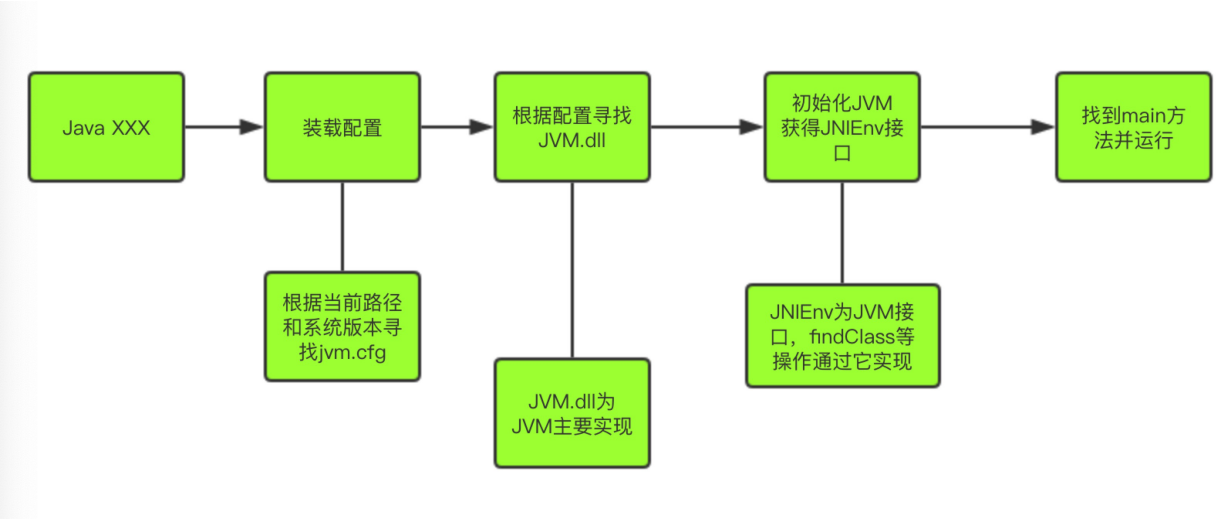
参考

深入 Java 虚拟机

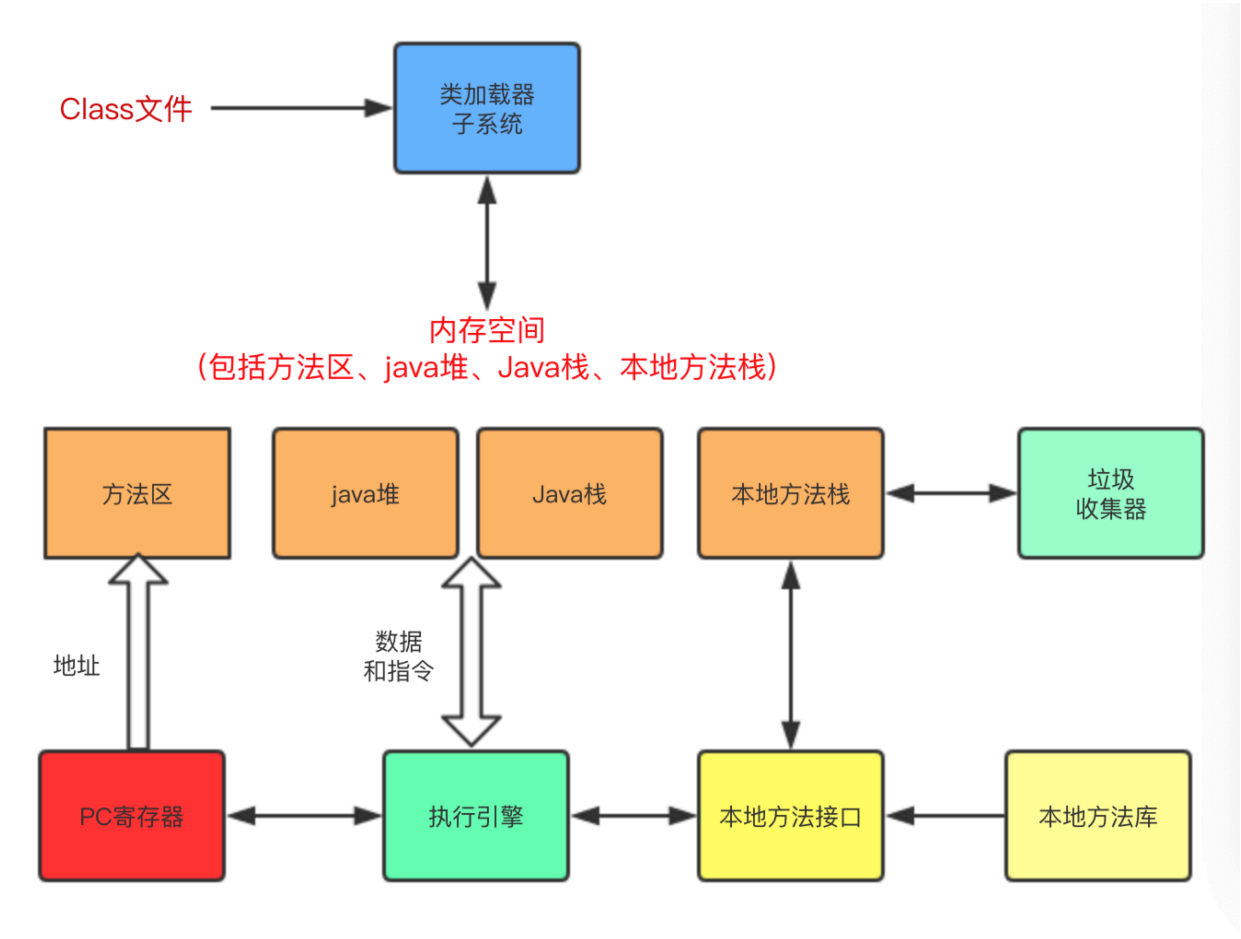
Java 类加载机制（全套）

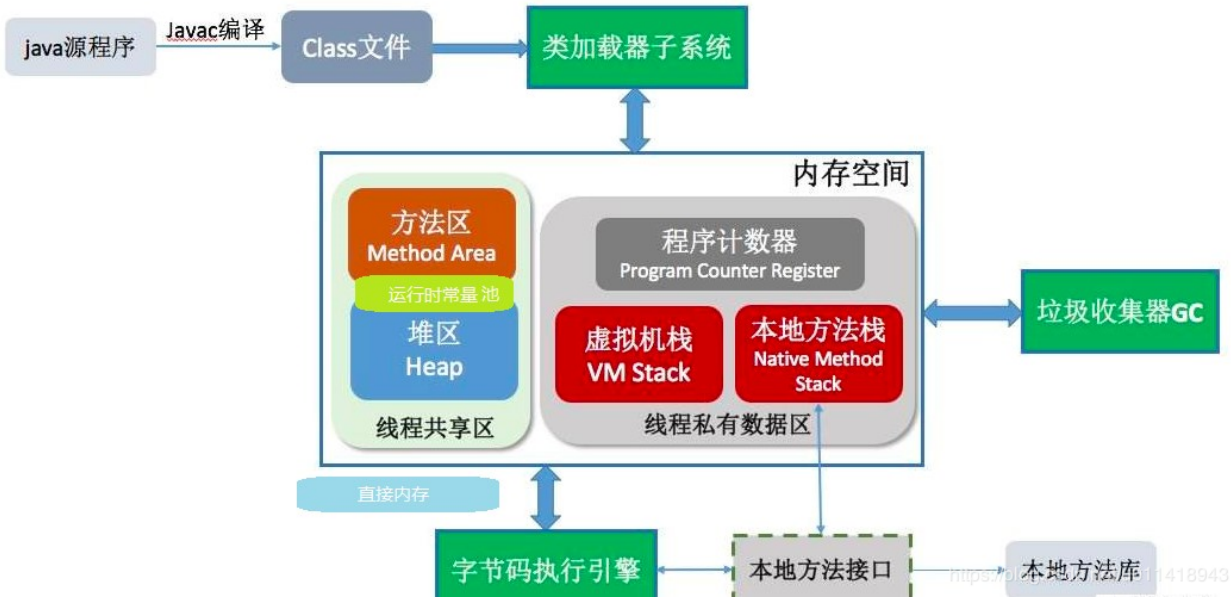
7.1.2 JVM 运行机制

1、启动流程



2、JVM 基本结构


Class文件
类加载器
子系统
内存空间
（包括方法区、java堆、Java栈、本地方法栈）
方法区
java堆
Java栈
本地方法栈
垃圾
收集器
地址
数据
和指令
PC寄存器
执行引擎
本地方法接口
本地方法库


java源程序
Javac编译
Class文件
类加载器子系统
内存空间
方法区
Method Area
运行时常量池
堆区
Heap
线程共享区
程序计数器
Program Counter Register
虚拟机栈
VM Stack
本地方法栈
Native Method
Stack
线程私有数据区
垃圾收集器GC
直接内存
字节码执行引擎
本地方法接口
本地方法库

2 .1、PC 寄存器

- 每个线程拥有一个 PC 寄存器
- 在线程创建时创建
- 指向下一条指令的地址
- 执行本地方法时，PC 的值为 undefine

2 .2、方法区

- 保存装载的类信息:
  - 类型的常量池
  - 字段、方法信息
  - 方法字节码
- 通常和永久区 (Perm) 关联在一起

2.3、Java 堆

- 和程序开发密切相关
- 应用系统对象都保存在 Java 堆中
- 所有线程共享 Java 堆
- 对分代 GC 来说，堆也是分代的
- GC 的主要工作区间

2.4 、Java 栈

- 线程私有
- 栈由一系列帧组成 (因此 Java 栈也叫 Java 帧栈)
- 帧保存一个方法的局部变量、操作数帧、常量池指针
- 每一次方法调用创建一个帧，并压栈

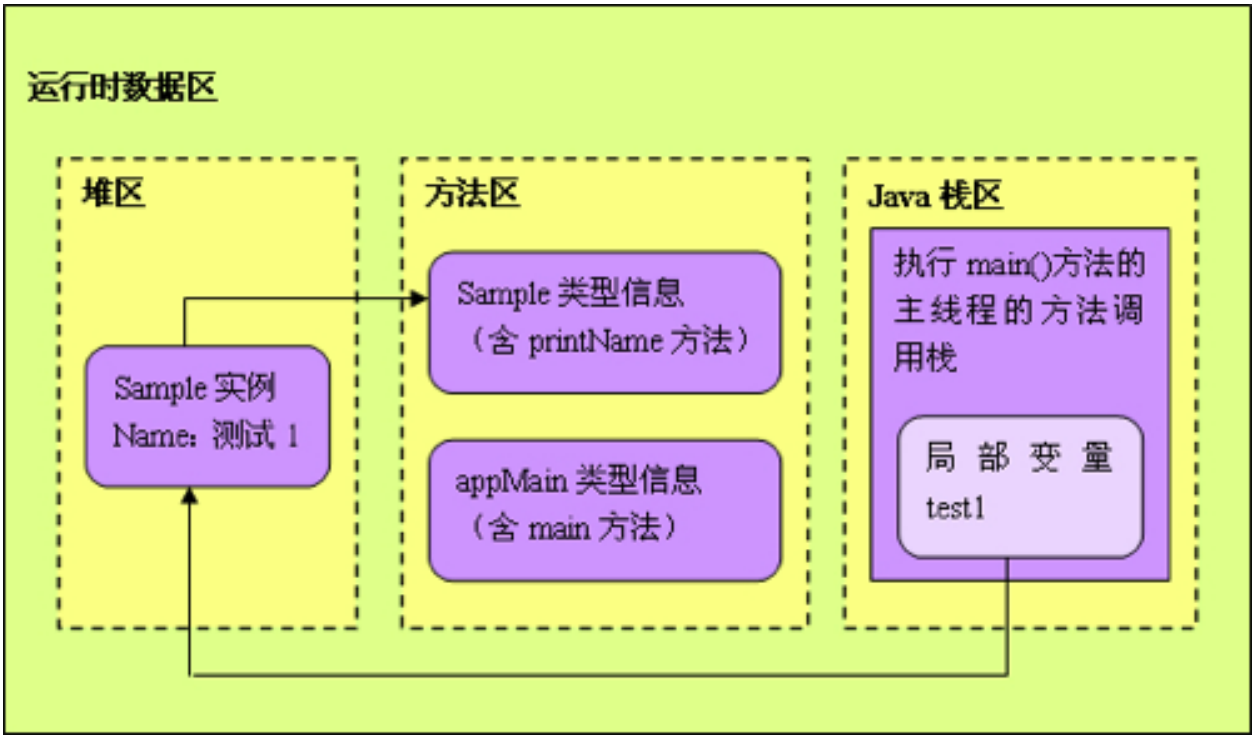
2.4 、栈、堆、方法区交互

```
public class AppMain {
    //运行时, jvm 把 appmain 的信息都放入方法区

    //main 方法本身放入方法区。
    public static void main(String[] args) {
        //test1 是引用, 所以放到栈区里， Sample 是自定义对象应该放到堆里面

        Sample test1 = new Sample(" 测试 1 ");
        Sample test2 = new Sample(" 测试 2 ");
        test1.printName();
```

(下页继续)

(续上页)

```
        test2.printName();
    }
}
//运行时, jvm 把 appmain 的信息都放入方法区
class Sample{
    private String name;

    //new Sample 实例后, name 引用放入栈区里, name 对象放入堆里
    public Sample(String name) {
        this.name = name;
    }
    //print 方法本身放入方法区里。
    public void printName() { System.out.println(name);
    }
}
```



7.2 Spring

7.2.1 Spring 工程代码结构

代码结构

根目录：net.csdn

- 启动类（CsdnApplication.java）推荐放在根目录 net.csdn 包下
- 实体类（domain）

  - A：net.csdn.domain（jpa 项目）
  - B：net.csdn.pojo（mybatis 项目）
- 数据接口访问层 (Dao)
  - A：net.csdn.repository（jpa 项目）
  - B：net.csdn.mapper（mybatis 项目）
- 数据服务接口层（Service）推荐：net.csdn.service
- 数据服务实现层（Service Implements）推荐：net.csdn.service.impl **使用 idea 的同学推荐使用 net.csdn.serviceImpl 目录**
- 前端控制器层（Controller）推荐：net.csdn.controller
- 工具类库（utils）推荐：net.csdn.utils
- 配置类（config）推荐：net.csdn.config
- 数据传输对象（dto）推荐：net.csdn.dto **数据传输对象（Data Transfer Object）用于封装多个实体类（domain）之间的关系，不破坏原有的实体类结构**
- 视图包装对象（vo）推荐：net.csdn.vo **视图包装对象（View Object）用于封装客户端请求的数据，防止部分数据泄露（如：管理员 ID），保证数据安全，不破坏原有的实体类结构**

资源目录结构

根目录：resources

- 项目配置文件：resources/application.yml
- 静态资源目录：resources/static/**用于存放 html、css、js、图片等资源**
- 视图模板目录：resources/templates/**用于存放 jsp、thymeleaf 等模板文件**
- mybatis 映射文件：resources/mapper/（mybatis 项目）
- mybatis 配置文件：resources/mapper/config/（mybatis 项目）

原文链接：spring boot 项目开发常用目录结构

7.2.2 Spring 常用注解

```
@Component
public class BookDao {
  ...
}
```

使用 @Component 注解在 BookDao 类声明处对它进行标注，这样它就可以被 Spring 容器识别，并把这个类转换为容器管理的 Bean

@PostConstruct

被 `@PostConstruct` 修饰的方法会在服务器加载 Servlet 的时候运行，并且只会被服务器执行一次。`PostConstruct` 在构造函数之后执行，init（）方法之前执行。PreDestroy（）方法在 destroy（）方法执行之后执行

@PostConstruct 应用场景

在项目中 @PostConstruct 主要应用场景是在初始化 Servlet 时加载一些缓存数据等。

@Scheduled

- 需要在定时任务的类上加上注释：@Component，在具体的定时任务方法上加上注释 @Scheduled,@EnableScheduling 即可启动该定时任务
- @Scheduled(fixedRate=3000)：上一次开始执行时间点后 3 秒再次执行；
- @Scheduled(fixedDelay=3000)：上一次执行完毕时间点后 3 秒再次执行；
- @Scheduled(initialDelay=1000, fixedDelay=3000)：第一次延迟 1 秒执行，然后在上一次执行完毕时间点后 3 秒再次执行；
- @Scheduled(cron="* * * * * ?")：按 cron 规则执行。

@FeignClient

7.3 Mybatis

7.3.1 Spring boot 工程应用 mybatis

引入依赖，完成配置

依赖包

mybatis 开发团队为 Spring Boot 提供了 `mybatis-spring-boot-starter`。你需要引入如下依赖：

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.1.1</version>
</dependency>
```

使用了该 starter 之后，只需要定义一个 `DataSource` 即可，它会自动利用该 `DataSource` 创建需要使用到的 `SqlSessionFactoryBean`、`SqlSessionTemplate`、以及 `ClassPathMapperScanner` 来自动扫描你的映射器接口，并针对每个接口都创建一个 `MapperFactoryBean`，注册到 Spring 上下文中。

关于 `mybatis-spring-boot-starter` 如何实现自动配置的相关源码，参见：

`org.mybatis.spring.boot.autoconfigure.MybatisAutoConfiguration` 类。

默认情况下，扫描的 `basePackage` 是 `spring boot` 的根目录 (这里指的是应用启动类 Application.java 类所在的目录)，且只会对添加了 @Mapper 注解的映射器接口进行注册。

连接信息配置

application.yml 中进行数据源的相关配置即可，以下配置依赖于 spring-boot-starter-jdbc:

```
server:
  port: 9000

spring:
    datasource:
       url: jdbc:mysql://localhost:3306/demo?allowMultiQueries=true&amp;
↪characterEncoding=UTF-8&amp;autoReconnect=true&amp;rewriteBatchedStatements=true
       username: root
       password: root
       driver-class-name: com.mysql.jdbc.Driver
```

MybatisConfiguration

```
package springboot.tutorials.mybatis.annotation.config;

import com.alibaba.druid.pool.DruidDataSource;
import org.apache.ibatis.session.SqlSessionFactory;
import org.mybatis.spring.SqlSessionFactoryBean;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import javax.sql.DataSource;

/**
 * @author Ricky Fung
 */
@Configuration
@MapperScan(basePackages = "springboot.tutorials.mybatis.annotation.mapper",
        sqlSessionFactoryRef = "sqlSessionFactory")
public class MybatisConfiguration {

    static final String MAPPER_LOCATION = "classpath:mapper/*.xml";

    @Value("${spring.datasource.url}")
    private String url;

    @Value("${spring.datasource.username}")
    private String user;

    @Value("${spring.datasource.password}")
    private String password;

    @Value("${spring.datasource.driver-class-name}")
    private String driverClass;

    @Bean(name = "sqlSessionFactory")
    public SqlSessionFactory sqlSessionFactory(@Qualifier("dataSource") DataSource␣
↪dataSource)
```

(下页继续)

(续上页)

```
            throws Exception {
        final SqlSessionFactoryBean sqlSessionFactoryBean = new␣
↪SqlSessionFactoryBean();
        sqlSessionFactoryBean.setDataSource(dataSource);
        sqlSessionFactoryBean.setMapperLocations(new␣
↪PathMatchingResourcePatternResolver()
                .getResources(MAPPER_LOCATION));
        return sqlSessionFactoryBean.getObject();
    }

    @Bean(name = "dataSource")
    public DataSource dataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName(driverClass);
        dataSource.setUrl(url);
        dataSource.setUsername(user);
        dataSource.setPassword(password);
        return dataSource;
    }

    @Bean(name = "transactionManager")
    public DataSourceTransactionManager transactionManager() {
        return new DataSourceTransactionManager(dataSource());
    }
}
```

数据库

```
CREATE TABLE `user` (
  `id` int(13) NOT NULL AUTO_INCREMENT COMMENT ' 主键',
  `name` varchar(33) DEFAULT NULL COMMENT ' 姓名',
  `ctime` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8
```

IDEA 自动生成相应文件

- 1、安装插件 better-mybatis-generator
- 2、功能 database tool 配置连接数据库，选择 table，右键，选择 mybatis generate，打开预览界面，填好相应配置信息，即可。
  - model folder：xxx.java - java 表的对象、xxxExample.java - Criteria 过滤条件。
  - dao folder：xxxDao.java - java 对象方法与 xml sql 语句之间的关系，增删改查的方法。
  - xml folder：xxxDao.xml - 数据库语句与对象之间的映射关系，可修改 SQL 语句。

业务实现，增删改查

```
// 实例化操作对象
@Autowired
private xxxDao xxxDao;

//  select
xxxExample xxxExample = new xxxExample();
xxxExample.Criteria criteria = xxxExample.createCriteria();
criteria.andNameEqualTo(name); // 设置筛选条件
List<xxx> xxxs = this.xxxDao.selectByExample(xxxExample);

// insert
xxx xxxInsert2DB = new xxx();
xxxInsert2DB.setName(name);
this.xxxDao.insert(xxxInsert2DB);

// update 多种选择
xxx xxxUpdate2DB = new xxx();
xxxUpdate2DB.setName(name);
this.xxxDao.updateByPrimaryKeySelective(xxxUpdate2DB);


// delete
xxxExample xxxExample = new xxxExample();
xxxExample.Criteria criteria = xxxExample.createCriteria();
criteria.andNameEqualTo(name); // 设置筛选条件
this.xxxDao.deleteByExample(xxxExample);
```

参考如下

MYBATIS 中文教程

Spring Boot 教程 - 整合 Mybatis（注解方式）

idea 插件 better mybatis generator 自动生成代码

7.3.2 Mybatis 深入学习

Mybatis 执行步骤

Mybatis 只做了两件事情：

- 根据 JDBC 规范建立与数据库的连接。
- 通过反射打通 Java 对象和数据库参数和返回值之间相互转化的关系。

Mybatis 的运行过程

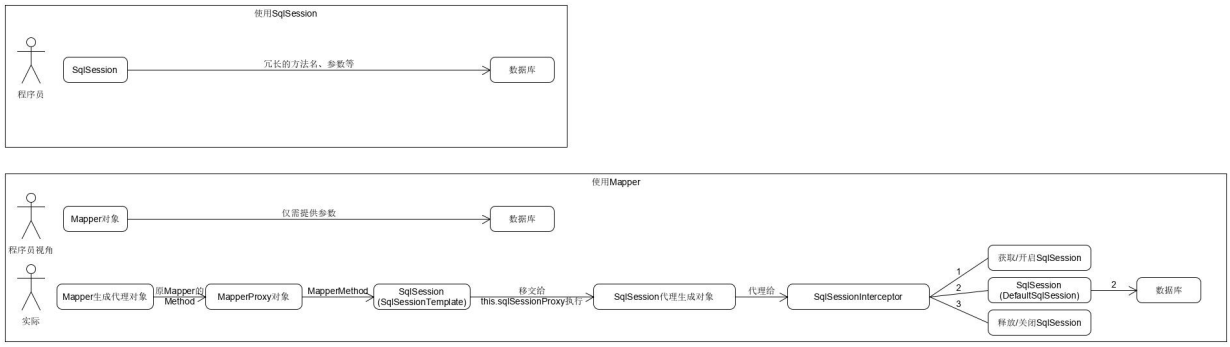
- 第一部分是读取配置文件创建 Configuration 对象, 用以创建 SqlSessionFactroy
- 第二部分是 SQLSession 的执行过程.

常见类

- 1、SqlSessionaFactoryBuilder 该类主要用于创建 SqlSessionFactory, 并给与一个流对象, 该类使用了创建者模式, 如果是手动创建该类 (这种方式很少了), 那么建议在创建完毕之后立即销毁.
- 2、SqlSessionFactory 该类的作用了创建 SqlSession, 从名字上我们也能看出, 该类使用了工厂模式, 每次应用程序访问数据库, 我们就要通过 SqlSessionFactory 创建 SqlSession, 所以 SqlSessionFactory 和整个 Mybatis 的生命周期是相同的. 这也告诉我们不要创建多个同一个数据的 SqlSessionFactory, 如果创建多个, 会消耗尽数据库的连接资源, 导致服务器夯机. 应当使用单例模式. 避免过多的连接被消耗, 也方便管理.
- 3、SqlSession 那么是什么 SqlSession 呢? SqlSession 相当于一个会话, 就像 HTTP 请求中的会话一样, 每次访问数据库都需要这样一个会话, 大家可能会想起了 JDBC 中的 Connection, 很类似, 但还是有区别的, 何况现在几乎所有的连接都是使用的连接池技术, 用完后直接归还而不会像 Session 一样销毁. 注意: 他是一个线程不安全的对象, 在设计多线程的时候我们需要特别的当心, 操作数据库需要注意其隔离级别, 数据库锁等高级特性, 此外, 每次创建的 SqlSession 都必须及时关闭它, 它长期存在就会使数据库连接池的活动资源减少, 对系统性能的影响很大, 我们一般在 finally 块中将其关闭. 还有, SqlSession 存活于一个应用的请求和操作, 可以执行多条 Sql, 保证事务的一致性.
- 4、Mapper 映射器，正如我们编写的那样, Mapper 是一个接口, 没有任何实现类, 他的作用是发送 SQL, 然后返回我们需要的结果. 或者执行 SQL 从而更改数据库的数据, 因此它应该在 SqlSession 的事务方法之内, 在 Spring 管理的 Bean 中, Mapper 是单例的。

@mapperScan

- 1、@MapperScan 扫描指定的包，对每个 Mapper，以它的名字注册了实际类型是 MapperFactoryBean 的 Bean 定义。
- 2、有了这些 Bean 定义，在 spring 实例化 Bean 时，这些 MapperFactoryBean 会被实例化、初始化，对应的方法也会设置
- 3、在处理 @autowired 标注的 Mapper 时，会返回 MapperFactoryBean.getObject 的调用结果, 也就是 getSqlSession().getMapper(this.mapperInterface); 了。
- 4、上一步的结果会导致 @Autowired SomeMapper mapper; 上注入了一个 Mapper 代理类，该代理类会将所有数据库请求都移交给底层的 SqlSession 操作。
- 5、上一步中，Mapper 移交到的 sqlSession 其实是个 SqlSessionTemplate，SqlSessionTemplate 又将一切数据库操作移交给 sqlSessionProxy，而后者是基于 SqlSessionInterceptor 创建的代理类。也就是说，SqlSessionTemplate 的数据库操作会被 SqlSessionInterceptor.invoke 所拦截。
- 6、SqlSessionInterceptor.invoke 中调用 getSqlSession 方法，其内部在需要的时候调用 session = sessionFactory.openSession(executorType); 获取新的 session，其实也就在开启新的连接。也就是说 SqlSessionTemplate 的数据库操作会被 SqlSessionInterceptor.invoke 所拦截，每次操作前都要获取到 SqlSession(实际类型是 DefaultSqlSession)，这个 SqlSession: 要么是复用现有的 (比如复用当前事务使用的)，要么是新建的。



深入 Mybatis

Executor 执行器

Mybatis 有三种基本的 Executor 执行器，SimpleExecutor、ReuseExecutor、BatchExecutor。

- SimpleExecutor：每执行一次 update 或 select，就开启一个 Statement 对象，用完立刻关闭 Statement 对象。
- ReuseExecutor：执行 update 或 select，以 sql 作为 key 查找 Statement 对象，存在就使用，不存在就创建，用完后，不关闭 Statement 对象，而是放置于 Map<String, Statement> 内，供下一次使用。简言之，就是重复使用 Statement 对象。
- BatchExecutor：执行 update（没有 select，JDBC 批处理不支持 select），将所有 sql 都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个 Statement 对象，每个 Statement 对象都是 addBatch() 完毕后，等待逐一执行 executeBatch() 批处理。与 JDBC 批处理相同。

Executor 的这些特点，都严格限制在 SqlSession 生命周期范围内。

参考如下：

mybatis mapper 解析 (下) @mapperScan

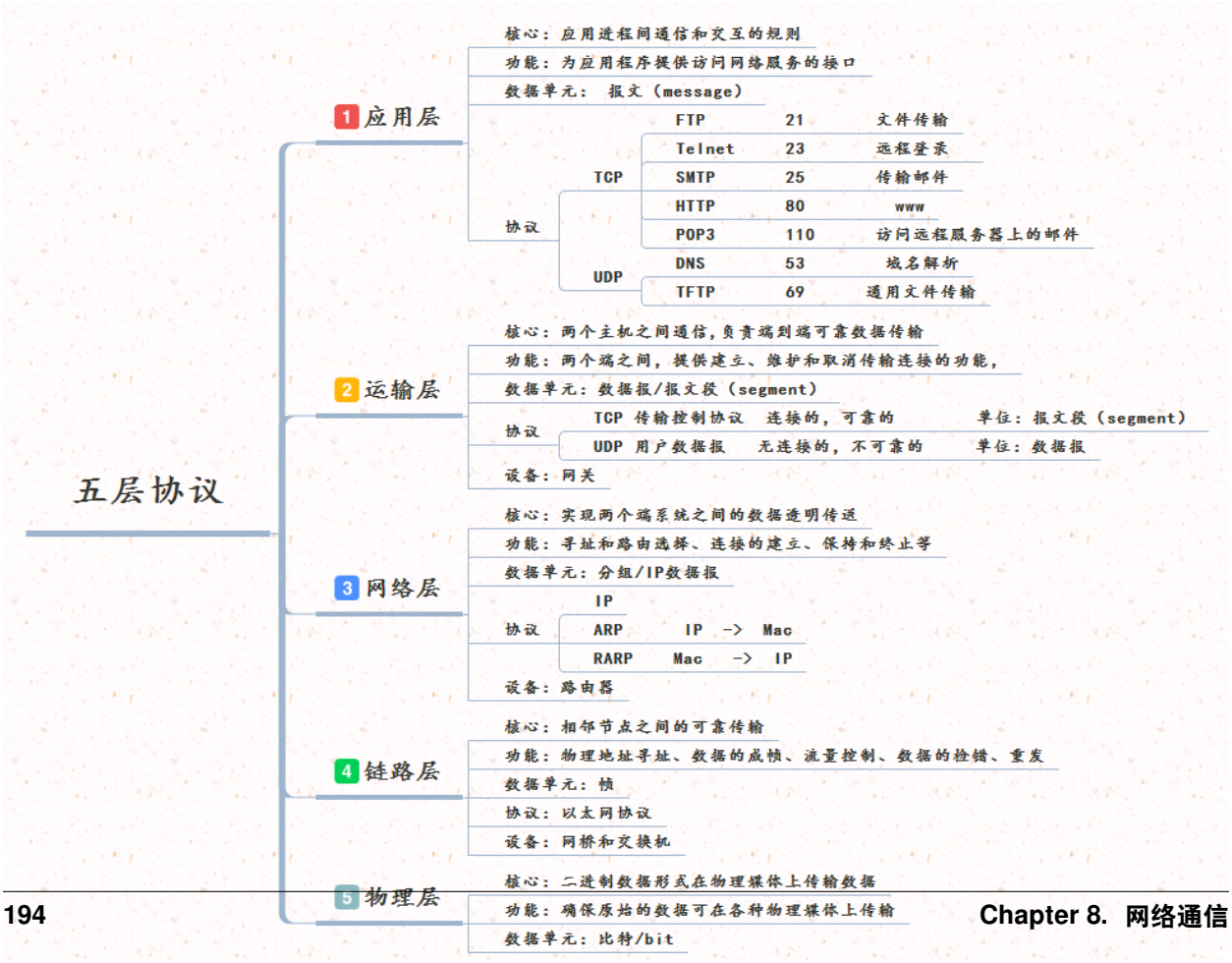
[深入剖析 mybatis 原理（一）]

CHAPTER 8

网络通信

8.1 网络协议

8.1.1 计算机网络



8.1.2 HTTP 协议

请求数据

请求码

- 1xx：指示信息--表示请求已接收，继续处理
- 2xx：成功--表示请求已被成功接收、理解、接受
- 3xx：重定向--要完成请求必须进行更进一步的操作
- 4xx：客户端错误--请求有语法错误或请求无法实现
- 5xx：服务器端错误--服务器未能实现合法的请求

常见请求码

```
200 OK                          //客户端请求成功
301 Moved Permanently           //请求永久重定向
302 Moved Temporarily           //请求临时重定向
400 Bad Request                 //客户端请求有语法错误，不能被服务器所理解
401 Unauthorized                //请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使
用
403 Forbidden                   //服务器收到请求，但是拒绝提供服务
404 Not Found                   //请求资源不存在，eg：输入了错误的 URL
500 Internal Server Error       //服务器发生不可预期的错误
503 Server Unavailable          //服务器当前不能处理客户端的请求，一段时间后可能恢复正常
```

请求方法

```
GET       请求指定的页面信息，并返回实体主体。
HEAD      类似于 get 请求，只不过返回的响应中没有具体的内容，用于获取报头
POST      向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可
能会导致新的资源的建立和/或已有资源的修改。
PUT       从客户端向服务器传送的数据取代指定的文档的内容。
DELETE    请求服务器删除指定的页面。
CONNECT   HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。
OPTIONS   允许客户端查看服务器的性能。
TRACE     回显服务器收到的请求，主要用于测试或诊断。
```

HTTP 工作原理

HTTP 协议定义 Web 客户端如何从 Web 服务器请求 Web 页面，以及服务器如何把 Web 页面传送给客户端。HTTP 协议采用了请求/响应模型。客户端向服务器发送一个请求报文，请求报文包含请求的方法、URL、协议版本、请求头部和请求数据。服务器以一个状态行作为响应，响应的内容包括协议的版本、成功或者错误代码、服务器信息、响应头部和响应数据。

以下是 HTTP 请求/响应的步骤：

1. 客户端连接到 Web 服务器一个 HTTP 客户端，通常是浏览器，与 Web 服务器的 HTTP 端口（默认为 80）建立一个 TCP 套接字连接。例如，http://www.luffycity.com。

2. 发送 HTTP 请求通过 TCP 套接字，客户端向 Web 服务器发送一个文本的请求报文，一个请求报文由请求行、请求头部、空行和请求数据 4 部分组成。
3. 服务器接受请求并返回 HTTP 响应 Web 服务器解析请求，定位请求资源。服务器将资源复本写到 TCP 套接字，由客户端读取。一个响应由状态行、响应头部、空行和响应数据 4 部分组成。
4. 释放连接 TCP 连接若 connection 模式为 close，则服务器主动关闭 TCP 连接，客户端被动关闭连接，释放 TCP 连接; 若 connection 模式为 keepalive，则该连接会保持一段时间，在该时间内可以继续接收请求;
5. 客户端浏览器解析 HTML 内容客户端浏览器首先解析状态行，查看表明请求是否成功的状态代码。然后解析每一个响应头，响应头告知以下为若干字节的 HTML 文档和文档的字符集。客户端浏览器读取响应数据 HTML，根据 HTML 的语法对其进行格式化，并在浏览器窗口中显示。

例如：在浏览器地址栏键入 URL，按下回车之后会经历以下流程：

1. 浏览器向 DNS 服务器请求解析该 URL 中的域名所对应的 IP 地址;
2. 解析出 IP 地址后，根据该 IP 地址和默认端口 80，和服务器建立 TCP 连接;
3. 浏览器发出读取文件 (URL 中域名后面部分对应的文件) 的 HTTP 请求，该请求报文作为 TCP 三次握手的第三个报文的数据发送给服务器;
4. 服务器对浏览器请求作出响应，并把对应的 html 文本发送给浏览器;
5. 释放 TCP 连接;
6. 浏览器将该 html 文本并显示内容;

参考如下：

HTTP 协议超级详解

HTTP 详解

8.1.3 websocket

建立在 TCP 协议之上

WebSocket 协议建立在 http 协议的基础之上，因此二者有很多的类似之处。事实上，在使用 websocket 协议时，浏览器与服务端最开始建立的还是 http 连接，之后再将协议从 http 转换成 websocket，协议转换的过程称之为握手 (handshake)，表示服务端与客户端都同意建立 websocket 协议。

请求

```
GET ws://server.example.com/ws HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

`Upgrade` 字段表示将通信协议从 `HTTP/1.1` 转向该字段指定的协议。`Connection` 字段表示浏览器通知服务器，如果可以的话，就升级到 `WebSocket` 协议。`Origin` 字段用于提供请求发出的域名，供服务器验证是否许可的范围内（服务器也可以不验证）。`Sec-WebSocket-Key` 则是用于握手协议的密钥，是 Base64 编码的 16 字节随机字符串。

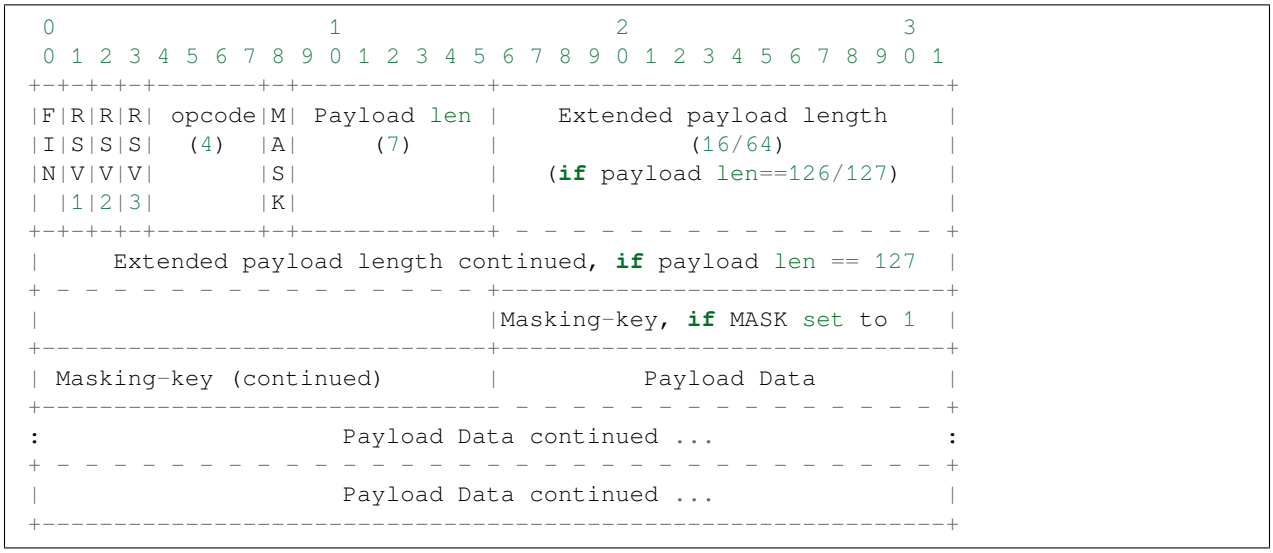
注意 GET 的路径是以 ws 开头，这是因为 WebSocket 是一种全新的协议，不属于 http 无状态协议，协议名为”ws”，与 http 协议使用相同的 80 端口，类似的，”wss” 和 https 协议使用相同的 443 端口。

响应

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: fFBooB7FAkLlXgRSz0BT3v4hq5s=
Sec-WebSocket-Origin: null
Sec-WebSocket-Location: ws://example.com/
```

服务器同样用 `Connection` 字段通知浏览器，需要改变协议。`Sec-WebSocket-Accept` 字段是服务器在浏览器提供的 `Sec-WebSocket-Key` 字符串后面，添加 RFC6456 标准规定的 “258EAFA5-E914-47DA-95CA-C5AB0DC85B11” 字符串，然后再取 SHA-1 的哈希值。浏览器将对这个值进行验证，以证明确实是目标服务器回应了 WebSocket 请求。`Sec-WebSocket-Location` 字段表示进行通信的 `WebSocket` 网址。

格式



- FIN：1 bit 。表示此帧是否是消息的最后帧，第一帧也可能是最后帧。
- RSV1，RSV2，RSV3：各 1 bit 。必须是 0，除非协商了扩展定义了非 0 的意义。
- opcode：4 bit。表示被传输帧的类型：x0 表示一个后续帧；x1 表示一个文本帧；x2 表示一个二进制帧；x3-7 为以后的非控制帧保留；x8 表示一个连接关闭；x9 表示一个 ping；xA 表示一个 pong；xB-F 为以后的控制帧保留。
- Mask：1 bit。表示净荷是否有掩码（只适用于客户端发送给服务器的消息）。
- Payload length：7 bit, 7 + 16 bit, 7 + 64 bit。净荷长度由可变长度字段表示：如果是 0~125，就是净荷长度；如果是 126，则接下来 2 字节表示的 16 位无符号整数才是这一帧的长度；如果是 127，则接下来 8 字节表示的 64 位无符号整数才是这一帧的长度。
- Masking-key：0 或 4 Byte。用于给净荷加掩护，客户端到服务器标记。
- Extension data：x Byte。默认为 0 Byte，除非协商了扩展。
- Application data：y Byte。在”Extension data”之后，占据了帧的剩余部分。

- Payload data：(x + y) Byte。”extension data”后接“application data”。

参考如下：

WebSocket 浅析

Netty-Websocket 根据 URL 路由，分发机制的实现

netty

WebSocket 协议深入探究

学习 WebSocket 协议—从顶层到底层的实现原理（修订版）

基于 Redis 以及 WebSocket 的一个实时消息推送系统

8.2 IO 通信

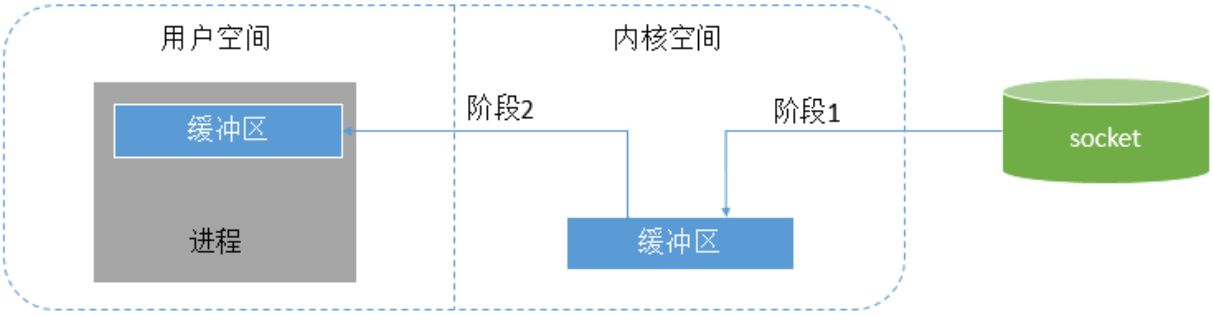
8.2.1 网络 I/O 模型

数据操作

对于一个 network IO (以 read 举例)，它会涉及到两个系统对象：一个是调用这个 IO 的进程，另一个就是系统内核 (kernel)。当一个 read 操作发生时，它会经历两个阶段：

阶段 1: 等待数据准备 (Waiting for the data to be ready)

阶段 2: 将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)



用户空间是常规进程所在区域。JVM 就是常规进程，驻守于用户空间。用户空间是非特权区域：比如，在该区域执行的代码就不能直接访问硬件设备。

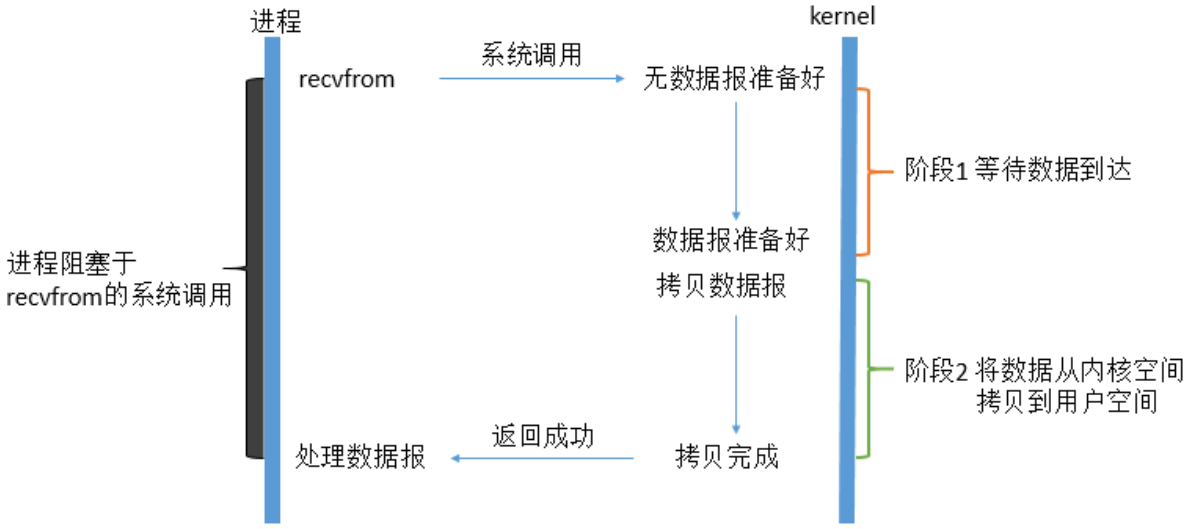
内核空间是操作系统所在区域。内核代码有特别的权力：它能与设备控制器通讯，控制着用户区域进程的运行状态，等等。最重要的是，所有 I/O 都直接（如这里所述）或间接通过内核空间。

当进程请求 I/O 操作的时候，它执行一个系统调用将控制权移交给内核。C/C++ 程序员所熟知的底层函数 open( )、read( )、write( ) 和 close( ) 要做的无非就是建立和执行适当的系统调用。当内核以这种方式被调用，它随即采取任何必要步骤，找到进程所需数据，并把数据传送到用户空间内的指定缓冲区。内核试图对数据进行高速缓存或预读取，因此进程所需数据可能已经在内核空间里了。如果是这样，该数据只需简单地拷贝出来即可。如果数据不在内核空间，则进程被挂起，内核着手把数据读进内存。

1、阻塞式 I/O：blocking IO

在 linux 中，默认情况下所有的 socket 都是 blocking，一个典型的读操作流程大概是这样：

- 第一步通常涉及等待数据从网络中到达。当所有等待数据到达时，它被复制到内核中的某个缓冲区。
- 第二步就是把数据从内核缓冲区复制到应用程序缓冲区。

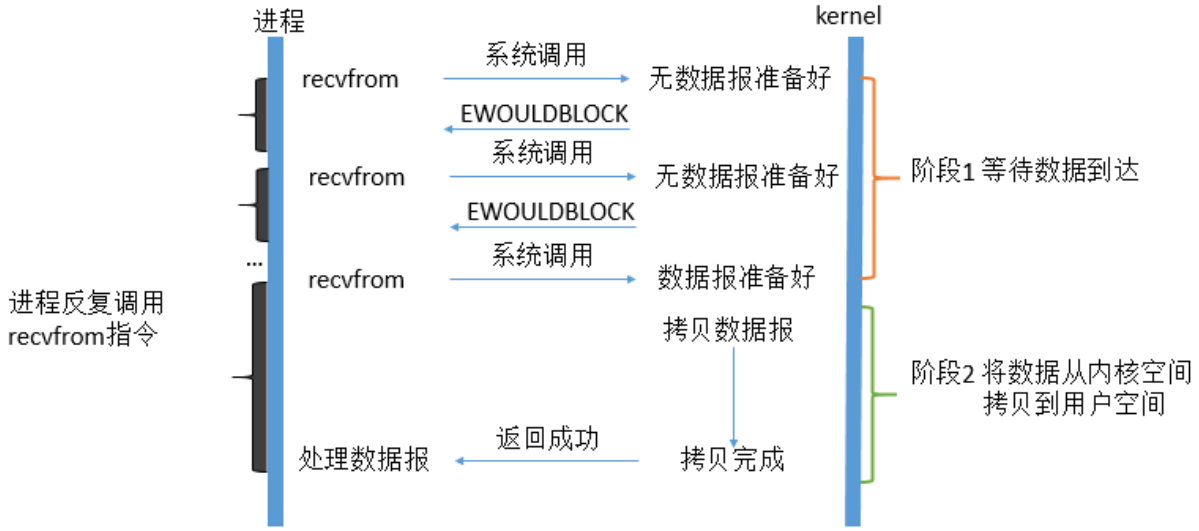


当用户进程调用了 recvfrom 这个系统调用，kernel 就开始了 IO 的第一个阶段：准备数据。对于 network io 来说，很多时候数据在一开始还没有到达（比如，还没有收到一个完整的 UDP 包），这个时候 kernel 就要等待足够的数据到来。而在用户进程这边，整个进程会被阻塞。当 kernel 一直等到数据准备好了，它就会将数据从 kernel 中拷贝到用户内存，然后 kernel 返回结果，用户进程才解除 block 的状态，重新运行起来。

所以，blocking IO 的特点就是在 IO 执行的两个阶段都被 block 了。

2、非阻塞式 I/O：nonblocking IO

linux 下，可以通过设置 socket 使其变为 non-blocking。当对一个 non-blocking socket 执行读操作时，流程是这个样子：



从图中可以看出，当用户进程发出 read 操作时，如果 kernel 中的数据还没有准备好，那么它并不会 block 用户进程，而是立刻返回一个 error。从用户进程角度讲，它发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个 error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦 kernel 中的数据准备好了，并且又再次收到了用户进程的 system call，那么它马上就将数据拷贝到了用户内存，然后返回。

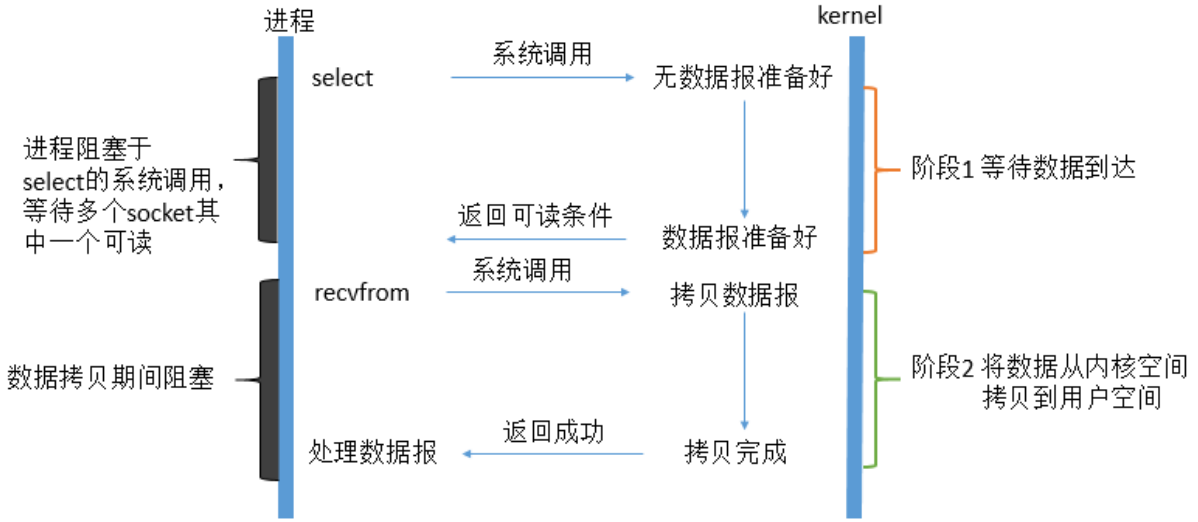
所以，用户进程第一个阶段不是阻塞的，需要不断的主动询问 kernel 数据好了没有；第二个阶段依然总是阻塞的。

3、I/O 复用（select，poll，epoll...）：IO multiplexing

IO 复用同非阻塞 IO 本质一样，不过利用了新的 select 系统调用，由内核来负责本来是请求进程该做的轮询操作。看似比非阻塞 IO 还多了一个系统调用开销，不过因为可以支持多路 IO，才算提高了效率。

基于内核，建立在 epoll 或者 kqueue 上实现，I/O 多路复用最大的优势是用户可以在一个线程内同时处理多个 Socket 的 I/O 请求。通过一个线程监听全部的 TCP 连接，有任何事件发生就通知用户态处理即可。

它的基本原理就是 select /epoll 这个 function 会不断的轮询所负责的所有 socket，当某个 socket 有数据到达了，就通知用户进程。它的流程如图：



当用户进程调用了 `select`，那么整个进程会被 block，而同时，kernel 会“监视”所有 select 负责的 socket，当任何一个 socket 中的数据准备好了，select 就会返回。这个时候用户进程再调用 read 操作，将数据从 kernel 拷贝到用户进程

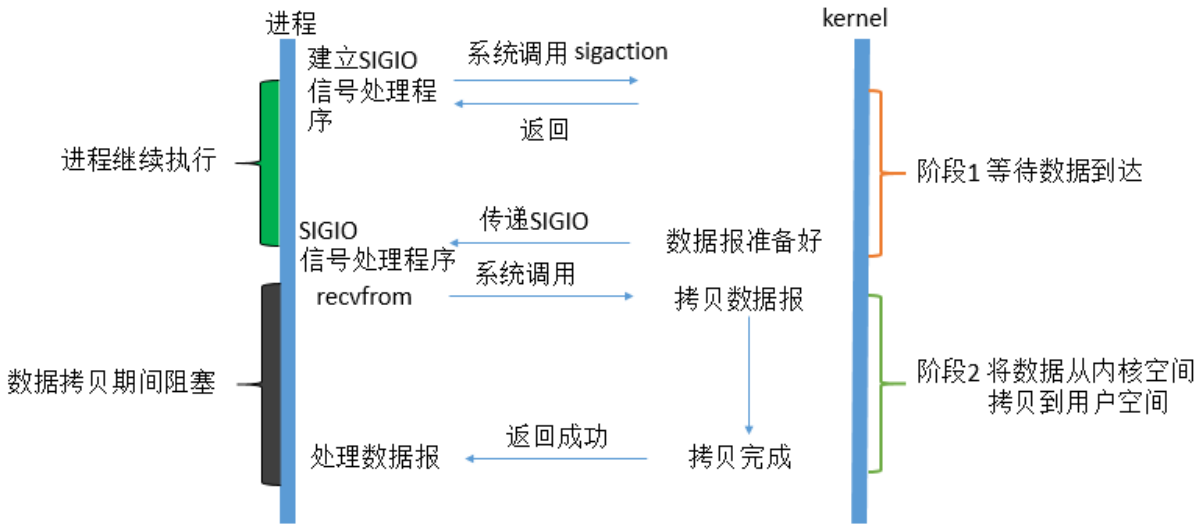
这里需要使用两个 system call (select 和 recvfrom)，而 blocking IO 只调用了一个 system call (recvfrom)。但是，用 select 的优势在于它可以同时处理多个 connection。（多说一句。所以，如果处理的连接数不是很高的话，使用 select/epoll 的 web server 不一定比使用 multi-threading + blocking IO 的 web server 性能更好，可能延迟还更大。

select/epoll 的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。

实际中，对于每一个 socket，一般都设置成为 non-blocking，但是，如上图所示，整个用户的 process 其实是一直被 block 的。只不过 process 是被 select 这个函数 block，而不是被 socket IO 给 block。

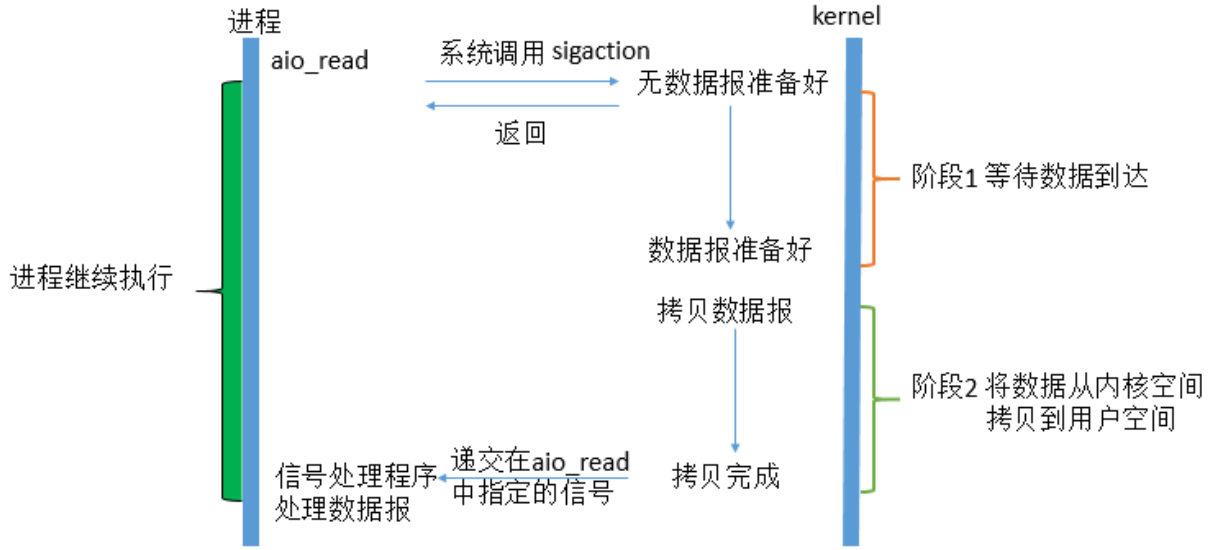
4、信号驱动式 I/O（SIGIO）：signal driven IO

用的很少



5、异步 I/O（POSIX 的 aio_ 系列函数）：asynchronous IO

这类函数的工作机制是告知内核启动某个操作，并让内核在整个操作（包括将数据从内核拷贝到用户空间）完成后通知我们。如图：



用户进程发起 read 操作之后，立刻就可以开始去做其它的事。而另一方面，从 kernel 的角度，当它受到一个 asynchronous read 之后，首先它会立刻返回，所以不会对用户进程产生任何 block。然后，kernel 会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel 会给用户进程发送一个 signal，告诉它 read 操作完成了。在这整个过程中，进程完全没有被 block。

I/O 多路复用往往对应 Reactor 模式，异步 I/O 往往对应 Proactor。

总结

前四种 I/O 模型都是同步 I/O 操作，他们的区别在于第一阶段，而他们的第二阶段是一样的：在数据从内核复制到应用缓冲区期间（用户空间），进程阻塞于 recvfrom 调用。

参考

Unix 五种 IO 模型

8.3 通信框架

8.3.1 Netty 实现 webSocket

处理 WebSocket frame

WebSockets 在“帧”里面来发送数据，其中每一个都代表了一个消息的一部分。一个完整的消息可以利用了多个帧。WebSocket “Request for Comments”(RFC) 定义了六中不同的 frame; Netty 给他们每个都提供了一个 POJO 实现

- CloseWebSocketFrame

- PingWebSocketFrame
- PongWebSocketFrame
- TextWebSocketFrame
- BinaryWebSocketFrame

其中 Close、Ping 以及 Pong 称之为控制帧，Close 关闭帧很容易理解，客户端如果接受到了就关闭连接，客户端也可以发送关闭帧给服务端。Ping 和 Pong 是 websocket 里的心跳，用来保证客户端是在线的。Text 和 Binary 表示这个传输的帧是文本类型还是二进制类型，二进制类型传输的数据可以是图片或者语音之类的。

如何实现消息推送

之前的案例，还是通过客户端发送请求，服务端进行响应的方式进行交互。WebSocket 协议还可以实现服务端主动向客户端推送消息的功能。

主动推送需要考虑的问题包括：

1、服务端要保存用户唯一标记 (假设为 userId) 与其对应的 SocketChannel 的对应关系，以便之后根据这个唯一标记给用户推送消息。

对应关系的保存很简单，用一个 Map 来记录即可。保存这种对应关系的最佳时机是 websocket 连接刚刚建立的时候，因为这种对应关系只需要保存一次，之后就可以拿着这个 userId 找到对应 SocketChannel 给用户推送消息。以之前的代码为例，很明显这个操作，应该放在 handleHttpRequest 方法中处理，因为这个方法在一个 websocket 连接的生命周期只会调用一次。

我们知道，当连接建立时，我们在服务端就能获取到客户端对应的 SocketChannel，我们如果获取到 userId 呢？WebSocket 协议与 HTTP 协议一样，都支持在 url 中添加参数。如：

```
String userId=params.get("userId").get(0);
//保存 userId 和 Channel 的对应关系
map.put(userId,ctx.channel());
```

2、通过 userId 给用户推送消息

通常消息推送是由运营人员根据业务需要，从一个后台管理界面，过滤出符合特定条件的用户来推送消息，因此需要推送的 userId 和推送内容都是运营指定的。推送服务端只需要根据 userId 找到对应的 Channel，将推送消息推送给浏览器即可。代码片段如下：

```
//运营人员制定要推送的 userId 和消息
String userId="123456";
String msg="xxxxxxxxxxxxxxxxxxxxx";

//根据 userId 找到对应的 channel，并将消息写出
Channel channel = map.get(userId);
channel.writeAndFlush(new TextWebSocketFrame(msg));
```

一般公司的推送的消息量都比较大，所以需要推送的消息一般会放到一个外部消息队列中，如 rocketmq，kafka。推送服务端通过消费这些队列，来给用户推送消息。

3、客户端 ack 接受到的消息

一般情况下，我们会每一个推送的消息都设置一个唯一的 msgId。客户端在接受推送的消息后，需要对这条消息进行 ack，也就是告诉服务端自己接收到了这条消息，ack 时需要将 msgId 带回来。

如果我们只使用 websocket 做推送服务，那么之前代码中的 handleWebSocketFrame 方法，要处理的主要就是这类 ack 消息。

4、离线消息

如果当给某个用户推送消息的时候，其并不在线，可以将消息保存下来，一般都是保存到一个缓存服务器中(如 redis)，每次当一个用户与服务端建立连接的时候，首先检查缓存服务器中有没有其对应的离线消息，如果有直接取出来发送给用户。

5、消息推送记录的存储

服务端推送完成消息记录之后，还需要将这些消息存储下来，以支持一个用户查看自己曾经受到过的推送消息。

6、疲劳度控制

在实现推送功能时，我们通常还会做一个疲劳度控制的功能，也就是限制给一个用户每天推送消息的数量，以免频繁的推送消息给用户造成不好的体验。疲劳度控制可以从多个维度进行设计。例如用户一天可以接受到消息总数，根据业务划分的某种特定类型的消息的一天可以接受到的消息总数等。

参考如下

Netty 实现 WebSocket 聊天功能

WebSocket 协议

8.4 Nginx

8.4.1 Nginx

Nginx 的模块根据其功能基本上可以分为以下几种类型：

- event module: 搭建了独立于操作系统的事件处理机制的框架，及提供了各具体事件的处理。包括 ngx_events_module，ngx_event_core_module 和 ngx_epoll_module 等。Nginx 具体使用何种事件处理模块，这依赖于具体的操作系统和编译选项。
- phase handler: 此类型的模块也被直接称为 handler 模块。主要负责处理客户端请求并产生待响应内容，比如 ngx_http_static_module 模块，负责客户端的静态页面请求处理并将对应的磁盘文件准备为响应内容输出。
- output filter: 也称为 filter 模块，主要是负责对输出的内容进行处理，可以对输出进行修改。例如，可以实现对输出的所有 html 页面增加预定义的 footbar 一类的工作，或者对输出的图片的 URL 进行替换之类的工作。
- upstream: upstream 模块实现反向代理的功能，将真正的请求转发到后端服务器上，并从后端服务器上读取响应，发回客户端。upstream 模块是一种特殊的 handler，只不过响应内容不是真正由自己产生的，而是从后端服务器上读取的。
- load-balancer: 负载均衡模块，实现特定的算法，在众多的后端服务器中，选择一个服务器出来作为某个请求的转发服务器。

example:

```
upstream getsvr {
    #server 13.231.34.83:1317;# 请求 rest 地址
    server 192.168.13.125:26659;# 请求 rest 地址
}

    server {
        listen 7777;
        location / {
            add_header Access-Control-Allow-Origin 'http://192.168.71.154:5200';# 需要
设置前端部署的地址
```

(下页继续)

(续上页)

```
                add_header Access-Control-Allow-Methods 'POST, GET, OPTIONS';
                add_header Access-Control-Allow-Headers 'X-Requested-With,Content-
↪Type,origin, content-type, accept, authorization,Action, Module, access-control-
↪allow-origin,app-type,timeout,devid';
                add_header Access-Control-Allow-Credentials true;
                if ($request_method = 'OPTIONS') {
                        return 204;
                }
                proxy_pass http://getsvr;
        }
    }
```

Nginx 使用一个多进程模型来对外提供服务，其中一个 master 进程，多个 worker 进程。master 进程负责管理 Nginx 本身和其他 worker 进程。

所有实际上的业务处理逻辑都在 worker 进程。worker 进程中有一个函数，执行无限循环，不断处理收到的来自客户端的请求，并进行处理，直到整个 Nginx 服务被停止。

worker 进程中，ngx_worker_process_cycle() 函数就是这个无限循环的处理函数。在这个函数中，一个请求的简单处理流程如下：

- 操作系统提供的机制（例如 epoll, kqueue 等）产生相关的事件。
- 接收和处理这些事件，如是接受到数据，则产生更高层的 request 对象。
- 处理 request 的 header 和 body。
- 产生响应，并发送回客户端。
- 完成 request 的处理。
- 重新初始化定时器及其他事件。

请求的处理流程

为了让大家更好的了解 Nginx 中请求处理过程，我们以 HTTP Request 为例，来做一下详细地说明。

从 Nginx 的内部来看，一个 HTTP Request 的处理过程涉及到以下几个阶段。

- 初始化 HTTP Request（读取来自客户端的数据，生成 HTTP Request 对象，该对象含有该请求所有的信息）。
- 处理请求头。
- 处理请求体。
- 如果有的话，调用与此请求（URL 或者 Location）关联的 handler。
- 依次调用各 phase handler 进行处理。

在这里，我们需要了解一下 phase handler 这个概念。phase 字面的意思，就是阶段。所以 phase handlers 也就好理解了，就是包含若干个处理阶段的一些 handler。

在每一个阶段，包含有若干个 handler，再处理到某个阶段的时候，依次调用该阶段的 handler 对 HTTP Request 进行处理。

通常情况下，一个 phase handler 对这个 request 进行处理，并产生一些输出。通常 phase handler 是与定义在配置文件中的某个 location 相关联的。

一个 phase handler 通常执行以下几项任务：

- 获取 location 配置。

- 产生适当的响应。
- 发送 response header。
- 发送 response body。

当 Nginx 读取到一个 HTTP Request 的 header 的时候，Nginx 首先查找与这个请求关联的虚拟主机的配置。如果找到了这个虚拟主机的配置，那么通常情况下，这个 HTTP Request 将会经过以下几个阶段的处理（phase handlers）：

- NGX_HTTP_POST_READ_PHASE: 读取请求内容阶段
- NGX_HTTP_SERVER_REWRITE_PHASE: Server 请求地址重写阶段
- NGX_HTTP_FIND_CONFIG_PHASE: 配置查找阶段:
- NGX_HTTP_REWRITE_PHASE: Location 请求地址重写阶段
- NGX_HTTP_POST_REWRITE_PHASE: 请求地址重写提交阶段
- NGX_HTTP_PREACCESS_PHASE: 访问权限检查准备阶段
- NGX_HTTP_ACCESS_PHASE: 访问权限检查阶段
- NGX_HTTP_POST_ACCESS_PHASE: 访问权限检查提交阶段
- NGX_HTTP_TRY_FILES_PHASE: 配置项 try_files 处理阶段
- NGX_HTTP_CONTENT_PHASE: 内容产生阶段
- NGX_HTTP_LOG_PHASE: 日志模块处理阶段

在内容产生阶段，为了给一个 request 产生正确的响应，Nginx 必须把这个 request 交给一个合适的 content handler 去处理。如果这个 request 对应的 location 在配置文件中被明确指定了一个 content handler，那么 Nginx 就可以通过对 location 的匹配，直接找到这个对应的 handler，并把这个 request 交给这个 content handler 去处理。这样的配置指令包括像，perl，flv，proxy_pass，mp4 等。

如果一个 request 对应的 location 并没有直接有配置的 content handler，那么 Nginx 依次尝试:

- 如果一个 location 里面有配置 random_index on，那么随机选择一个文件，发送给客户端。
- 如果一个 location 里面有配置 index 指令，那么发送 index 指令指明的文件，给客户端。
- 如果一个 location 里面有配置 autoindex on，那么就发送请求地址对应的服务端路径下的文件列表给客户端。
- 如果这个 request 对应的 location 上有设置 gzip_static on，那么就查找是否有对应的.gz 文件存在，有的话，就发送这个给客户端（客户端支持 gzip 的情况下）。
- 请求的 URI 如果对应一个静态文件，static module 就发送静态文件的内容到客户端。内容产生阶段完成以后，生成的输出会被传递到 filter 模块去进行处理。filter 模块也是与 location 相关的。所有的 fiter 模块都被组织成一条链。输出会依次穿越所有的 filter，直到有一个 filter 模块的返回值表明已经处理完成。

这里列举几个常见的 filter 模块，例如：

- server-side includes。
- XSLT filtering。
- 图像缩放之类的。
- gzip 压缩。

在所有的 filter 中，有几个 filter 模块需要关注一下。按照调用的顺序依次说明如下：

- write: 写输出到客户端，实际上是写到连接对应的 socket 上。
- postpone: 这个 filter 是负责 subrequest 的，也就是子请求的。

- copy: 将一些需要复制的 buf(文件或者内存) 重新复制一份然后交给剩余的 body filter 处理。

参考如下：深入浅出 NginxNginx 入门指南

8.4.2 nginx 代理跨域

nginx 反向代理接口跨域

跨域原理：同源策略是浏览器的安全策略，不是 HTTP 协议的一部分。服务器端调用 HTTP 接口只是使用 HTTP 协议，不会执行 JS 脚本，不需要同源策略，也就不存在跨越问题。

实现思路：通过 nginx 配置一个代理服务器（域名与 domain1 相同，端口不同）做跳板机，反向代理访问 domain2 接口，并且可以顺便修改 cookie 中 domain 信息，方便当前域 cookie 写入，实现跨域登录。

nginx 具体配置：

```
#proxy 服务器
server {
    listen       81;
    server_name  www.domain1.com;

    location / {
        proxy_pass   http://www.domain2.com:8080;  # 反向代理
        proxy_cookie_domain www.domain2.com www.domain1.com; # 修改 cookie 里域名
        index  index.html index.htm;

        # 当用 webpack-dev-server 等中间件代理接口访问 nignx 时，此时无浏览器参与，故没有同源限
制，下面的跨域配置可不启用
        add_header Access-Control-Allow-Origin http://www.domain1.com;  # 当前端只跨域不
带 cookie 时，可为 *
        add_header Access-Control-Allow-Credentials true;
    }
}
```

配置

```
location / {
  add_header Access-Control-Allow-Origin *;
  add_header Access-Control-Allow-Headers "Origin, X-Requested-With, Content-Type,␣
↪Accept";
  add_header Access-Control-Allow-Methods "GET, POST, OPTIONS";
}
```

1. Access-Control-Allow-Origin

服务器默认是不被允许跨域的。给 Nginx 服务器配置 Access-Control-Allow-Origin * 后，表示服务器可以接受所有的请求源（Origin），即接受所有跨域的请求。

2. Access-Control-Allow-Headers

是为了防止出现以下错误:

```
Request header field Content-Type is not allowed by Access-Control-Allow-Headers in␣
↪preflight response.
```

这个错误表示当前请求 Content-Type 的值不被支持。其实是我们发起了"application/json" 的类型请求导致的。

3. Access-Control-Allow-Methods

是为了防止出现以下错误:

```
Content-Type is not allowed by Access-Control-Allow-Headers in preflight response.
```

发送" 预检请求" 时，需要用到方法 OPTIONS , 所以服务器需要允许该方法。

预检请求（preflight request）

> 跨域资源共享 (CORS) 标准新增了一组 HTTP 首部字段，允许服务器声明哪些源站有权限访问哪些资源。另外，规范要求，对那些可能对服务器数据产生副作用的 HTTP 请求方法（特别是 GET 以外的 HTTP 请求，或者搭配某些 MIME 类型的 POST 请求），浏览器必须首先使用 OPTIONS 方法发起一个预检请求（preflight request），从而获知服务端是否允许该跨域请求。服务器确认允许之后，才发起实际的 HTTP 请求。在预检请求的返回中，服务器端也可以通知客户端，是否需要携带身份凭证（包括 Cookies 和 HTTP 认证相关数据）。

Content-Type 不属于以下 MIME 类型的，都属于预检请求:

```
application/x-www-form-urlencoded
multipart/form-data
text/plain
```

所以 application/json 的请求会在正式通信之前，增加一次" 预检" 请求，这次" 预检" 请求会带上头部信息 Access-Control-Request-Headers: Content-Type

CHAPTER 9

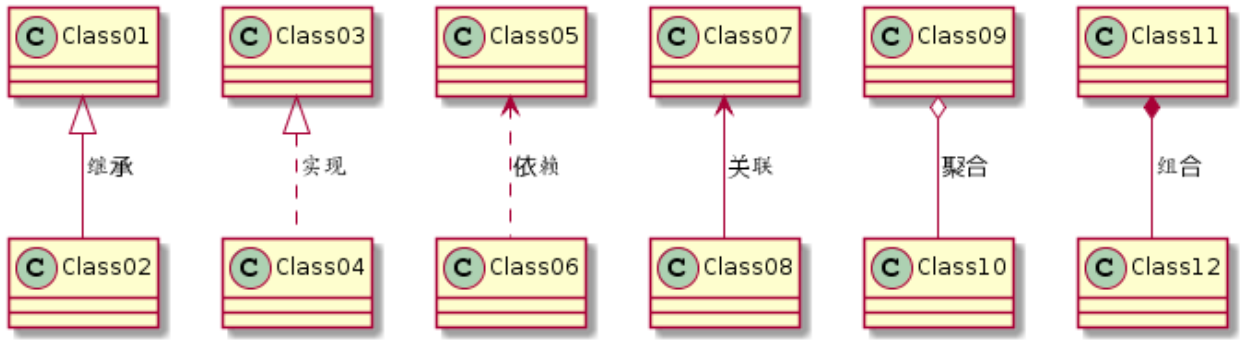
设计模式

9.1 类图关系

- **泛化 (generalization)**：表示 is-a 的关系，是对象之间耦合度最大的一种关系，子类继承父类的所有细节。直接使用语言中的继承表达。**带三角箭头的实线表示，箭头从子类指向父类**
- **实现 (Realization)**: 在类图中就是接口和实现的关系。**三角箭头的虚线表示，箭头从实现类指向接口**。
- **依赖 (Dependency)**：是临时性的关联。代码中一般指由局部变量、函数参数、返回值建立的对于其他对象的调用关系。一个类调用被依赖类中的某些方法而得以完成这个类的一些职责。**带箭头的虚线表示，箭头从使用类指向被依赖的类**
- ** 关联 (Association) **: 对象之间一种引用关系，比如客户类与订单类之间的关系。这种关系通常使用类的属性表达。关联又分为一般关联、聚合关联与组合关联。后两种在后面分析。**使用带箭头的实线表示，箭头从使用类指向被关联的类。可以是单向和双向**。
- **聚合 (Aggregation)**：表示 has-a 的关系，是一种不稳定的包含关系。较强于一般关联, 有整体与局部的关系, 并且没有了整体, 局部也可单独存在。如公司和员工的关系，公司包含员工，但如果公司倒闭，员工依然可以换公司。在**类图使用空心的菱形表示，菱形从局部指向整体**
- **组合 (Composition)**：表示 contains-a 的关系，是一种强烈的包含关系。组合类负责被组合类的生命周期。是一种更强的聚合关系。部分不能脱离整体存在。如公司和部门的关系，没有了公司，部门也不能存在了；**在类图使用实心的菱形表示，菱形从局部指向整体**。

9.2 PUML 图形

```
@startuml
Class01  <|--  Class02  : 继承
Class03  <|..  Class04  : 实现
Class05  <..   Class06  : 依赖
Class07  <--   Class08  : 关联
Class09  o--   Class10  : 聚合
Class11  *--   Class12  : 组合
@enduml
```



9.2.1 PUML 图像网站

9.3 单例模式

9.3.1 涉及的 JVM 与 CPU

Java 虚拟机在执行该对象创建语句时具体做了哪些事情，我们简单概括为 3 步：

- 1 在栈内存中创建 singleton 变量，在堆内存中开辟一个空间用来存放 Singleton 实例对象，该空间会得到一个随机地址，假设为 0x0001。
- 2 对 Singleton 实例对象初始化。
- 3 将 singleton 变量指向该对象，也就是将该对象的地址 0x0001 赋值给 singleton 变量，此时 singleton 就不为 null 了。

CPU 在执行指令时并不是无节操随意打乱顺序，而是有一定的原则可寻的，这个原则也叫先行发生原则（happens-before）

先行发生原则（happens-before）：

- 1 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作
- 2 锁定规则：一个 unLock 操作先行发生于后面对同一个锁额 lock 操作
- 3 volatile 变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作
- 4 传递规则：如果操作 A 先行发生于操作 B，而操作 B 又先行发生于操作 C，则可以得出操作 A 先行发生于操作 C
- 5 线程启动规则：Thread 对象的 start() 方法先行发生于此线程的每个一个动作
- 6 线程中断规则：对线程 interrupt() 方法的调用先行发生于被中断线程的代码检测到中断事件的发生

- 7 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过 Thread.join() 方法结束、Thread.isAlive() 的返回值手段检测到线程已经终止执行
- 8 对象终结规则：一个对象的初始化完成先行发生于他的 finalize() 方法的开始

9.3.2 单例模式介绍

所谓单例，就是整个程序有且仅有一个实例。该类负责创建自己的对象，同时确保只有一个对象被创建。在 Java，一般常用在工具类的实现或创建对象需要消耗资源。

特点

- 类构造器私有
- 持有自己类型的属性
- 对外提供获取实例的静态方法

饿汉模式

线程安全，比较常用，但容易产生垃圾，因为一开始就初始化

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
       return instance;
    }
}

直接调用 Singleton.getInstance() 来创建对象
```

只要这个类一加载完，就会创建实例对象，显得很着急，像一个恶汉一样，所以称之为恶汉式单例模式。

由于饿汉模式会带来一个问题，类加载的时候静态变量便申请了空间，如果没有用到的话，而且复杂的话，会造成大量的资源浪费，所以用一种懒加载的方式来创建对象，等需要用到这个对象的时候再创建。

懒汉模式

线程不安全，延迟初始化，严格意义上不是不是单例模式

```
public class Singleton {
    private static Singleton instance; // 静态类型
    private Singleton (){}

    public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}

直接调用 Singleton.getInstance() 来创建对象
```

由于只有当用到时才创建对象，比较懒，我们称之为懒汉式单例模式。

双重锁模式

线程安全，延迟初始化。这种方式采用双锁机制，安全且在多线程情况下能保持高性能。

```
public class Singleton {

    private volatile static Singleton singleton;

    private Singleton() {
        System.out.print("实例化对象\n");
    }

    public static Singleton getInsatnce() {
        if (singleton == null) {//-----------------------------1
            synchronized (Singleton.class) {//------------2
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

这里 synchronized 的用法和上面的有所不同，上面我们用 synchronized 来修饰方法，表示给整个方法上了把锁，我们称之为同步方法。这里我们只给语句 1 和语句 2 加了把锁，我们称这种结构为同步代码块，同步代码块 synchronized 后面的括号中需要一个对象，可以任意，这里我们用了 Singleton 的类对象 Singleton.class。可以看到我们在方法中进行了两次对象是否为空的判断，一次在同步代码块外面，一次在里面。因此称之为双重检验锁模式（Double Checked Locking Pattern）。

双重检查模式，进行了两次的判断，第一次是为了避免不要的实例，第二次是为了进行同步，避免多线程问题。由于 singleton=new Singleton() 对象的创建在 JVM 中可能会进行重排序，在多线程访问下存在风险，使用 volatile 修饰 signleton 实例变量有效，解决该问题。

静态内部类单例模式

```
public class Singleton {
    private Singleton(){}
    public static Singleton getInstance(){
        return Inner.instance;
    }
    private static class Inner {
        private static final Singleton instance = new Singleton();
    }
}
```

只有第一次调用 getInstance 方法时，虚拟机才加载 Inner 并初始化 instance ，只有一个线程可以获得对象的初始化锁，其他线程无法进行初始化，保证对象的唯一性。目前此方式是所有单例模式中最推荐的模式，但具体还是根据项目选择。

我们在 Singleton 类内部定义一个 Inner 静态内部类，里面有一个对象实例，当然由于 Java 虚拟机自身的特性，只有调用该静态内部类时才会创建该对象，从而实现了单例的延迟加载，同样由于虚拟机的特性，该模式是线程安全的，并且后续读取该单例对象时不会进行同步加锁的操作，提高了性能。

枚举单例模式

- 枚举类隐藏了私有的构造器。
- 枚举类的域是相应类型的一个实例对象

```
public enum Singleton  {
    INSTANCE

    // doSomething 该实例支持的行为

    // 可以省略此方法，通过 Singleton.INSTANCE 进行操作
    public static Singleton get Instance() {
        return Singleton.INSTANCE;
    }
}
```

参考如下:

Java 大白话讲解设计模式之 -- 单例模式