

## 课程介绍

- MongoDB集群之复制集
- MongoDB集群之分片集群
- 日志规范
- 异常规范
- 其它规范

## 1、MongoDB集群之复制集

### 1.1、简介

一组Mongodb复制集，就是一组mongod进程，这些进程维护同一个数据集合。复制集提供了数据冗余和高等级的可靠性，这是生产部署的基础。

#### 目的

- 保证数据在生产部署时的冗余和可靠性，通过在不同的机器上保存副本来保证数据的不会因为单点损坏而丢失。能够随时应对数据丢失、机器损坏带来的风险。
- 还能提高读取能力，用户的读取服务器和写入服务器在不同的地方，而且，由不同的服务器为不同的用户提供服务，提高整个系统的负载。

#### 机制

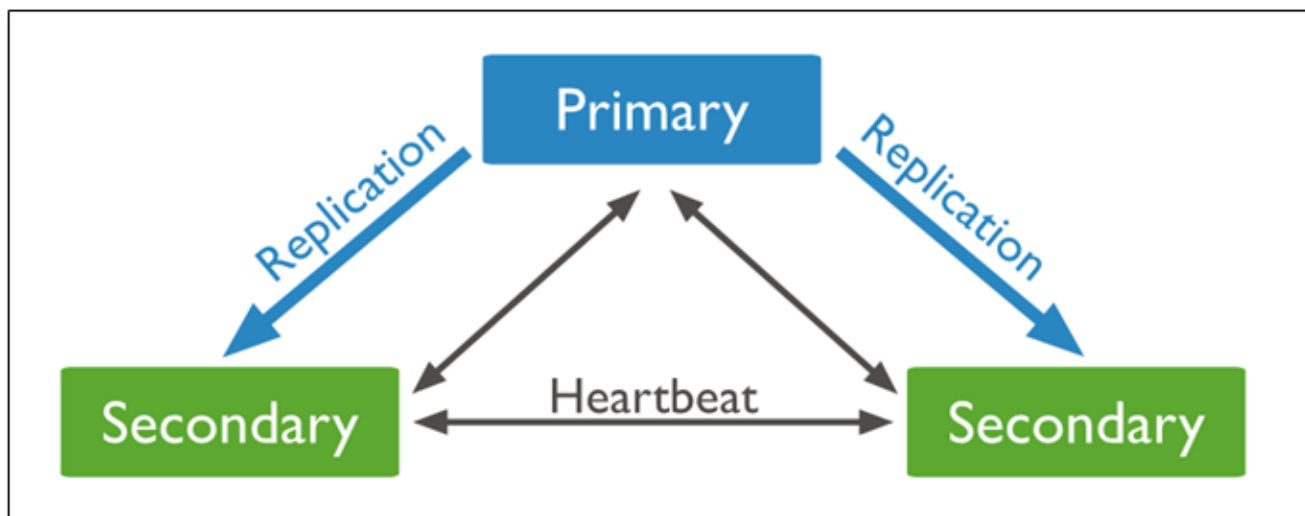
- 一组复制集就是一组mongod实例掌管同一个数据集，实例可以在不同的机器上面。实例中包含一个主导（Primary），接受客户端所有的写入操作，其他都是副本实例（Secondary），从主服务器上获得数据并保持同步。
- 主服务器很重要，包含了所有的改变操作（写）的日志。但是副本服务器集群包含有所有的主服务器数据，因此当主服务器挂掉了，就会在副本服务器上重新选取一个成为主服务器。
- 每个复制集还有一个仲裁者（Arbiter），仲裁者不存储数据，只是负责通过心跳包来确认集群中集合的数量，并在主服务器选举的时候作为仲裁决定结果。

### 1.2、架构

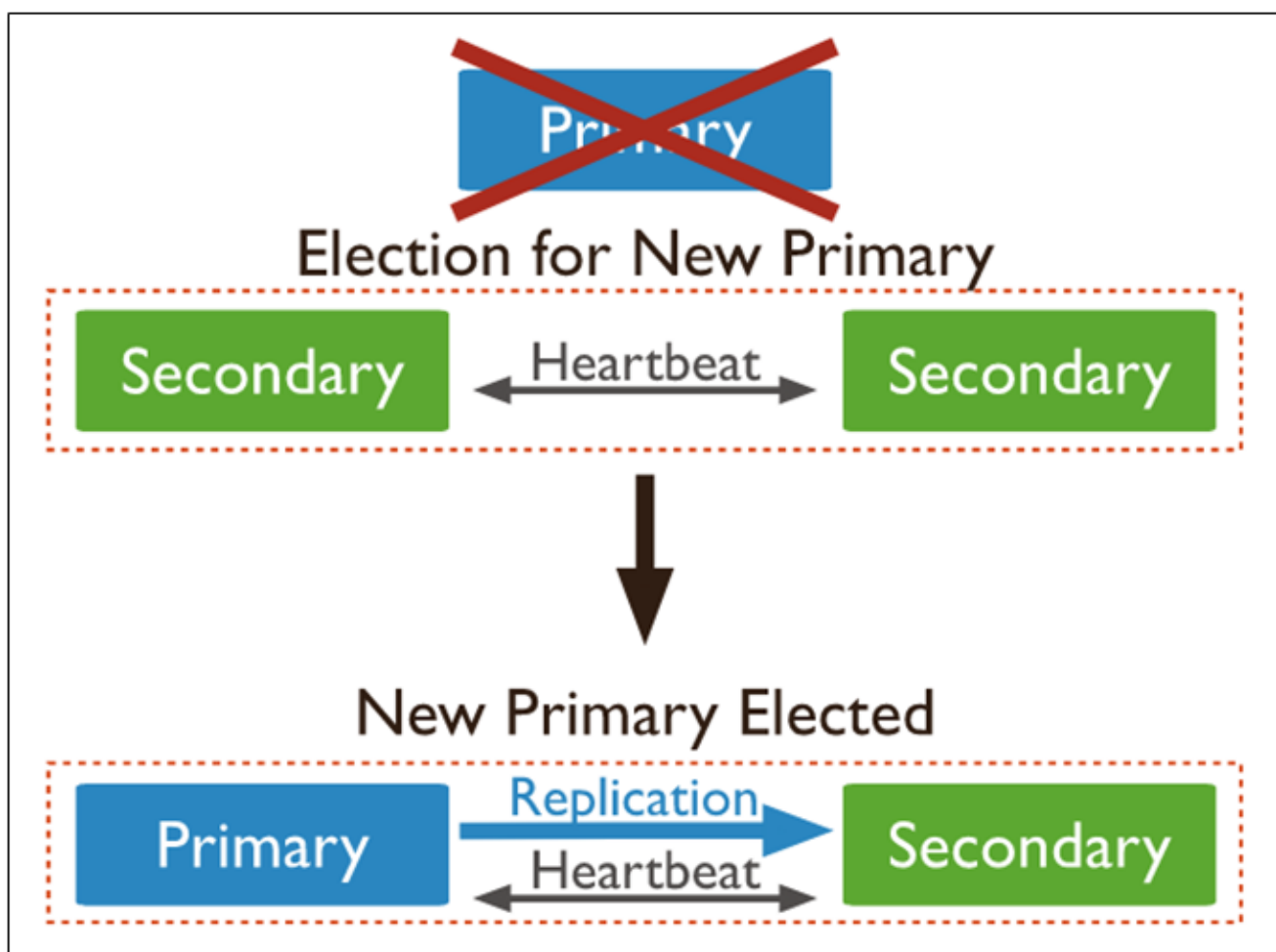
基本的架构由3台服务器组成，一个三成员的复制集，由三个有数据，或者两个有数据，一个作为仲裁者。

#### 1.2.1、三个存储数据的复制集

一个主，两个从库组成，主库宕机时，这两个从库都可以被选为主库。

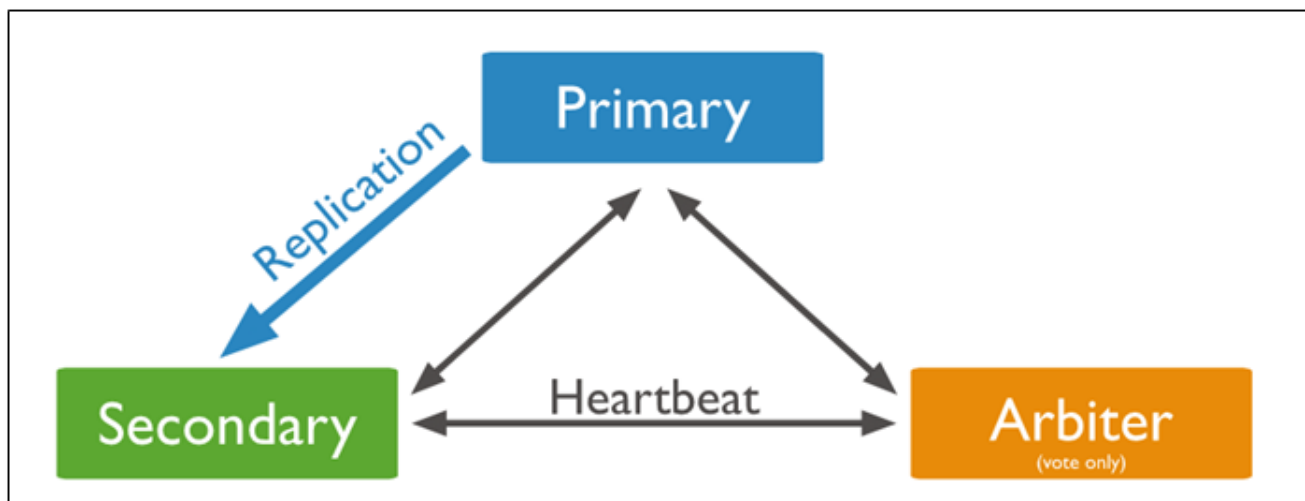


当主库宕机后,两个从库都会进行竞选,其中一个变为主库,当原主库恢复后,作为从库加入当前的复制集群即可。



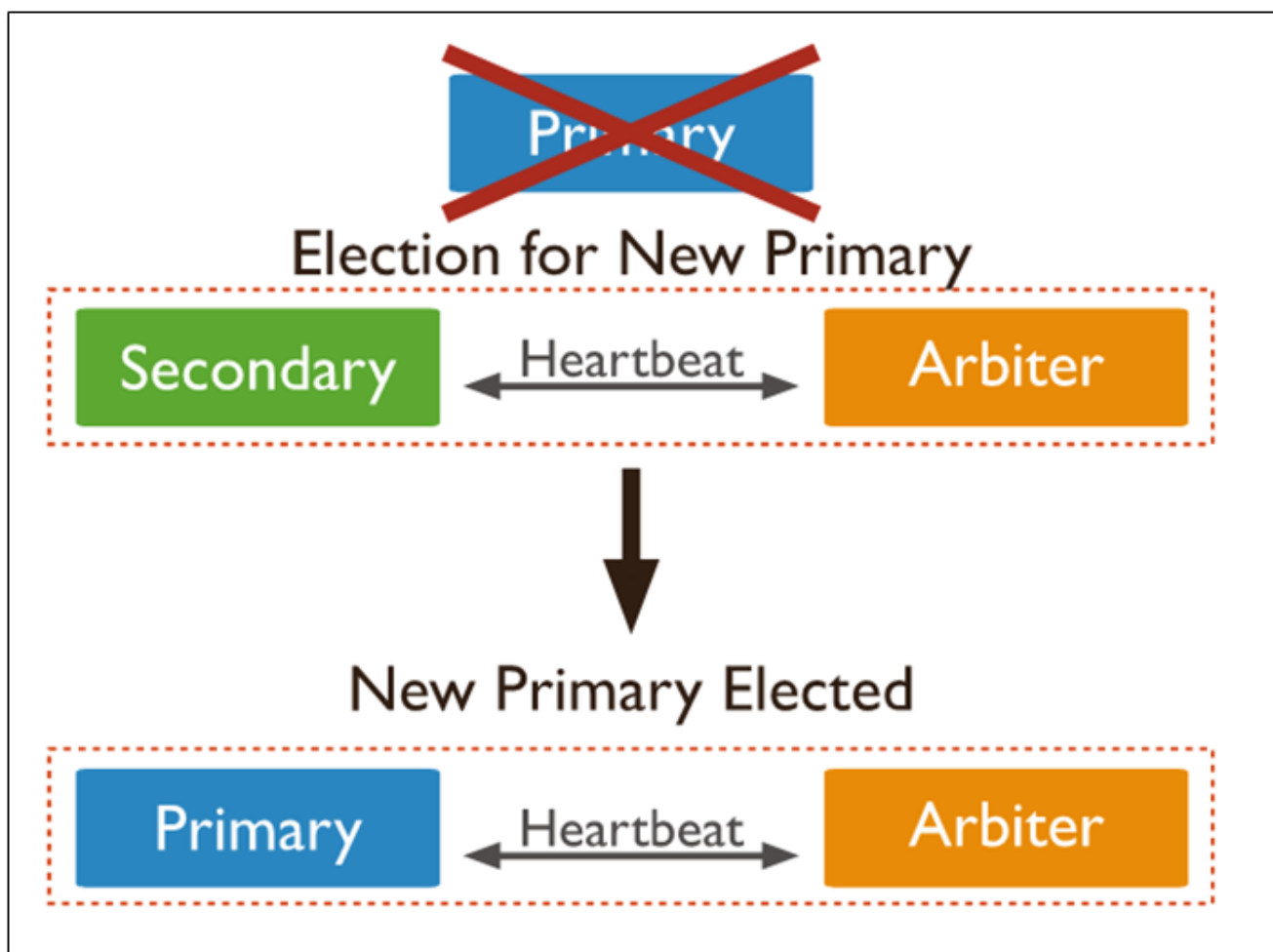
### 1.2.2、存在arbiter节点的复制集

一个主库，一个从库，可以在选举中成为主库，一个arbiter节点，在选举中，只进行投票，不能成为主库。



说明：由于arbiter节点没有复制数据，因此这个架构中仅提供一个完整的数据副本。arbiter节点只需要更少的资源，代价是更有限的冗余和容错。

当主库宕机时，将会选择从库成为主，主库修复后，将其加入到现有的复制集群中即可。



### 1.3、Primary选举

复制集通过replSetInitiate命令（或mongo shell的rs.initiate()）进行初始化，初始化后各个成员间开始发送心跳消息，并发起Primary选举操作，获得『大多数』成员投票支持的节点，会成为Primary，其余节点成为Secondary。

## 『大多数』的定义

假设复制集内投票成员数量为N，则大多数为  $N/2 + 1$ ，当复制集内存活成员数量不足大多数时，整个复制集将无法选举出Primary，复制集将无法提供写服务，处于只读状态。

## 1.4、成员说明

成员	说明
Primary	Primary的作用是接收用户的写入操作，将自己的数据同步给其他的Secondary。
Secondary	正常情况下，复制集的Secondary会参与Primary选举（自身也可能被选为Primary），并从Primary同步最新写入的数据，以保证与Primary存储相同的数据。Secondary可以提供读服务，增加Secondary节点可以提供复制集的读服务能力，同时提升复制集的可用性。另外，Mongodb支持对复制集的Secondary节点进行灵活的配置，以适应多种场景的需求。
Arbiter	Arbiter节点只参与投票，不能被选为Primary，并且不从Primary同步数据。比如你部署了一个2个节点的复制集，1个Primary，1个Secondary，任一节点宕机，复制集将不能提供服务了（无法选出Primary），这时可以给复制集添加一个Arbiter节点，即使有节点宕机，仍能选出Primary。Arbiter本身不存储数据，是非常轻量级的服务，当复制集成员为偶数时，最好加入一个Arbiter节点，以提升复制集可用性。
Priority0	Priority0节点的选举优先级为0，不会被选举为Primary。比如你跨机房A、B部署了一个复制集，并且想指定Primary必须在A机房，这时可以将B机房的复制集成员Priority设置为0，这样Primary就一定会是A机房的成员。（注意：如果这样部署，最好将『大多数』节点部署在A机房，否则网络分区时可能无法选出Primary）
Vote0	Mongodb 3.0里，复制集成员最多50个，参与Primary选举投票的成员最多7个，其他成员（Vote0）的vote属性必须设置为0，即不参与投票。
Hidden	Hidden节点不能被选为主（Priority为0），并且对Driver不可见。因Hidden节点不会接受Driver的请求，可使用Hidden节点做一些数据备份、离线计算的任务，不会影响复制集的服务。
Delayed	Delayed节点必须是Hidden节点，并且其数据落后与Primary一段时间（可配置，比如1个小时）。因Delayed节点的数据比Primary落后一段时间，当错误或者无效的数据写入Primary时，可通过Delayed节点的数据来恢复到之前的时间点。

## 1.5、搭建复制集

```

1  #创建3个mongo容器
2  docker create --name mongo01 -p 27017:27017 -v mongo-data-01:/data/db mongo:4.0.3 --
   replset "rs0" --bind_ip_all
3  docker create --name mongo02 -p 27018:27017 -v mongo-data-02:/data/db mongo:4.0.3 --
   replset "rs0" --bind_ip_all
4  docker create --name mongo03 -p 27019:27017 -v mongo-data-03:/data/db mongo:4.0.3 --
   replset "rs0" --bind_ip_all
5
6  #启动容器
7  docker start mongo01 mongo02 mongo03
8
    
```



```
9  #进入容器操作
10 docker exec -it mongo01 /bin/bash
11
12  #登录到mongo服务
13 mongo 172.16.55.185:27017
14
15  #初始化复制集集群
16 rs.initiate( {
17     _id : "rs0",
18     members: [
19         { _id: 0, host: "172.16.55.185:27017" },
20         { _id: 1, host: "172.16.55.185:27018" },
21         { _id: 2, host: "172.16.55.185:27019" }
22     ]
23 })
24
25  #响应
26 {
27     "ok" : 1, #成功
28     "operationTime" : Timestamp(1551619334, 1),
29     "$clusterTime" : {
30         "clusterTime" : Timestamp(1551619334, 1),
31         "signature" : {
32             "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAAA="),
33             "keyId" : NumberLong(0)
34         }
35     }
36 }
```

测试复制集群：

```
1  #在主库插入数据
2  rs0:PRIMARY> use test
3  rs0:PRIMARY> db.user.insert({"id":1001,"name":"zhangsan"})
4  WriteResult({ "nInserted" : 1 })
5  rs0:PRIMARY> db.user.find()
6  { "_id" : ObjectId("5c7bd5965504bcd309686907"), "id" : 1001, "name" : "zhangsan" }
7
8  #在复制库查询数据
9  mongo 172.16.55.185:27018
10
11 rs0:SECONDARY> use test
12 rs0:SECONDARY> db.user.find()
13 Error: error: { #出错，默认情况下从库是不允许读写操作的
14     "operationTime" : Timestamp(1551619556, 1),
15     "ok" : 0,
16     "errmsg" : "not master and slaveOk=false",
17     "code" : 13435,
18     "codeName" : "NotMasterNoSlaveOk",
19     "$clusterTime" : {
20         "clusterTime" : Timestamp(1551619556, 1),
21         "signature" : {
22             "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAAA="),
```

```
23         "keyId" : NumberLong(0)
24     }
25 }
26 }
27
28 rs0:SECONDARY> rs.slaveOk() #设置允许从库读取数据
29 rs0:SECONDARY> db.user.find()
30 { "_id" : ObjectId("5c7bd5965504bcd309686907"), "id" : 1001, "name" : "zhangsan" }
31
```

## 1.6、故障转移

- 测试一：从节点宕机
  - 集群依然可以正常使用，可以读写操作。
- 测试二：主节点宕机
  - 选举出新的主节点继续提供服务
- 测试三：停止集群中的2个节点
  - 当前集群无法选举出Primary，无法提供写操作，只能进行读操作

## 1.7、增加arbiter节点

当集群中的节点数为偶数时，如一主一从情况下，任意一节点宕机都无法选举出Primary，无法提供写操作，加入arbiter节点后即可解决该问题。

```
1 docker create --name mongo04 -p 27020:27017 -v mongo-data-04:/data/db mongo:4.0.3 --
  replset "rs0" --bind_ip_all
2
3 docker start mongo04
4
5 #在主节点执行
6 rs0:PRIMARY> rs.addArb("172.16.55.185:27020")
7 {
8     "ok" : 1,
9     "operationTime" : Timestamp(1551627454, 1),
10    "$clusterTime" : {
11        "clusterTime" : Timestamp(1551627454, 1),
12        "signature" : {
13            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAA="),
14            "keyId" : NumberLong(0)
15        }
16    }
17 }
18
19 #查询集群状态
20 rs.status()
21
```

通过测试，添加arbiter节点后，如果集群节点数不满足 $N/2+1$ 时，arbiter节点可作为“凑数”节点，可以选出主节点，继续提供服务。

## 2、MongoDB集群之分片集群

分片（sharding）是MongoDB用来将大型集合分割到不同服务器（或者说一个集群）上所采用的方法。尽管分片起源于关系型数据库分区，但MongoDB分片完全又是另一回事。

和MySQL分区方案相比，MongoDB的最大区别在于它几乎能自动完成所有事情，只要告诉MongoDB要分配数据，它就能自动维护数据在不同服务器之间的均衡。

### 2.1、简介

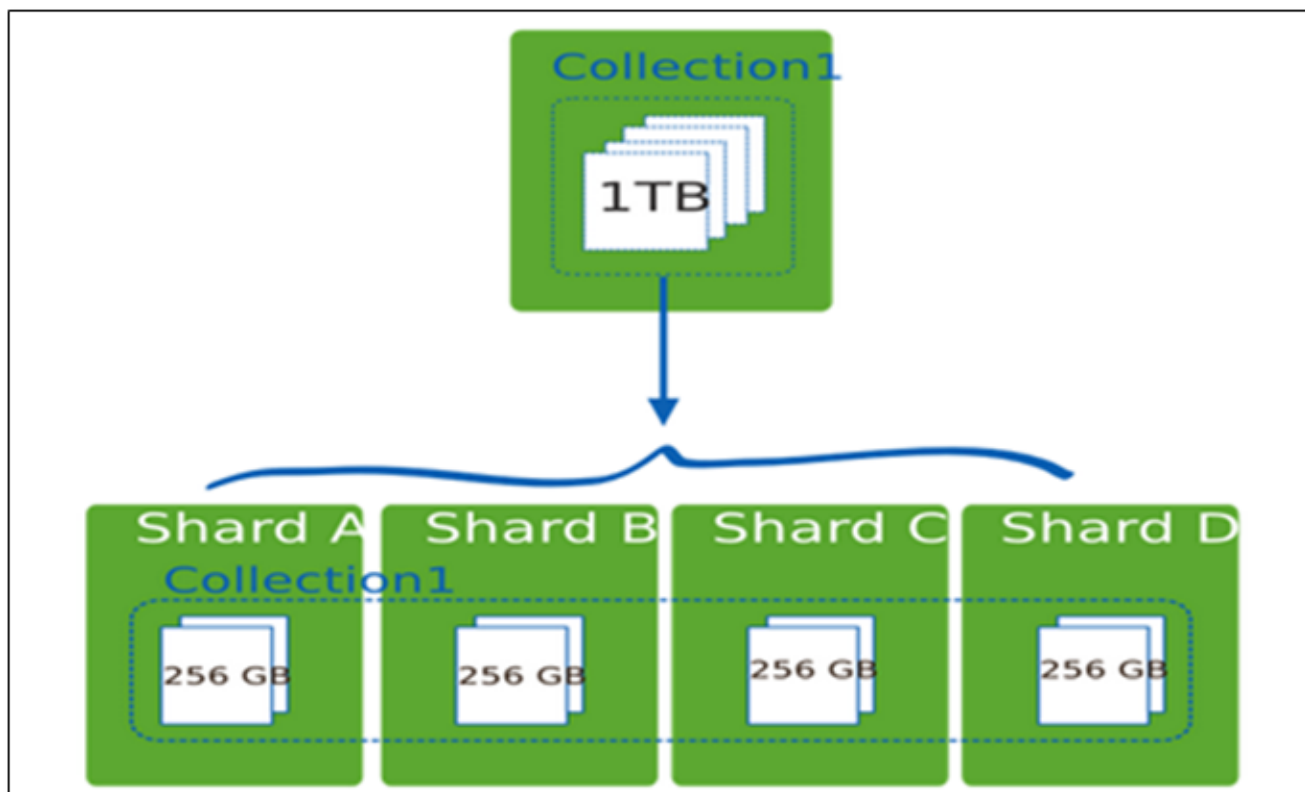
高数据量和吞吐量的数据库应用会对单机的性能造成较大压力,大的查询量会将单机的CPU耗尽,大的数据量对单机的存储压力较大,最终会耗尽系统的内存而将压力转移到磁盘IO上。

为了解决这些问题,有两个基本的方法: 垂直扩展和水平扩展。

- 垂直扩展：增加更多的CPU和存储资源来扩展容量。
- 水平扩展：将数据集分布在多个服务器上。水平扩展即分片。

分片为应对高吞吐量与大数据量提供了方法。使用分片减少了每个分片需要处理的请求数，因此，通过水平扩展，集群可以提高自己的存储容量和吞吐量。举例来说，当插入一条数据时，应用只需要访问存储这条数据的分片。

使用分片减少了每个分片存储的数据。例如，如果数据库1tb的数据集，并有4个分片，然后每个分片可能仅持有256 GB的数据。如果有40个分片，那么每个切分可能只有25GB的数据。



### 2.2、优势

- 对集群进行抽象，让集群“不可见”
  - MongoDB自带了一个叫做mongos的专有路由进程。mongos就是掌握统一路口的路由器，其会将客户端发来的请求准确无误的路由到集群中的一个或者一组服务器上，同时会把接收到的响应拼装起来发回到客户端。

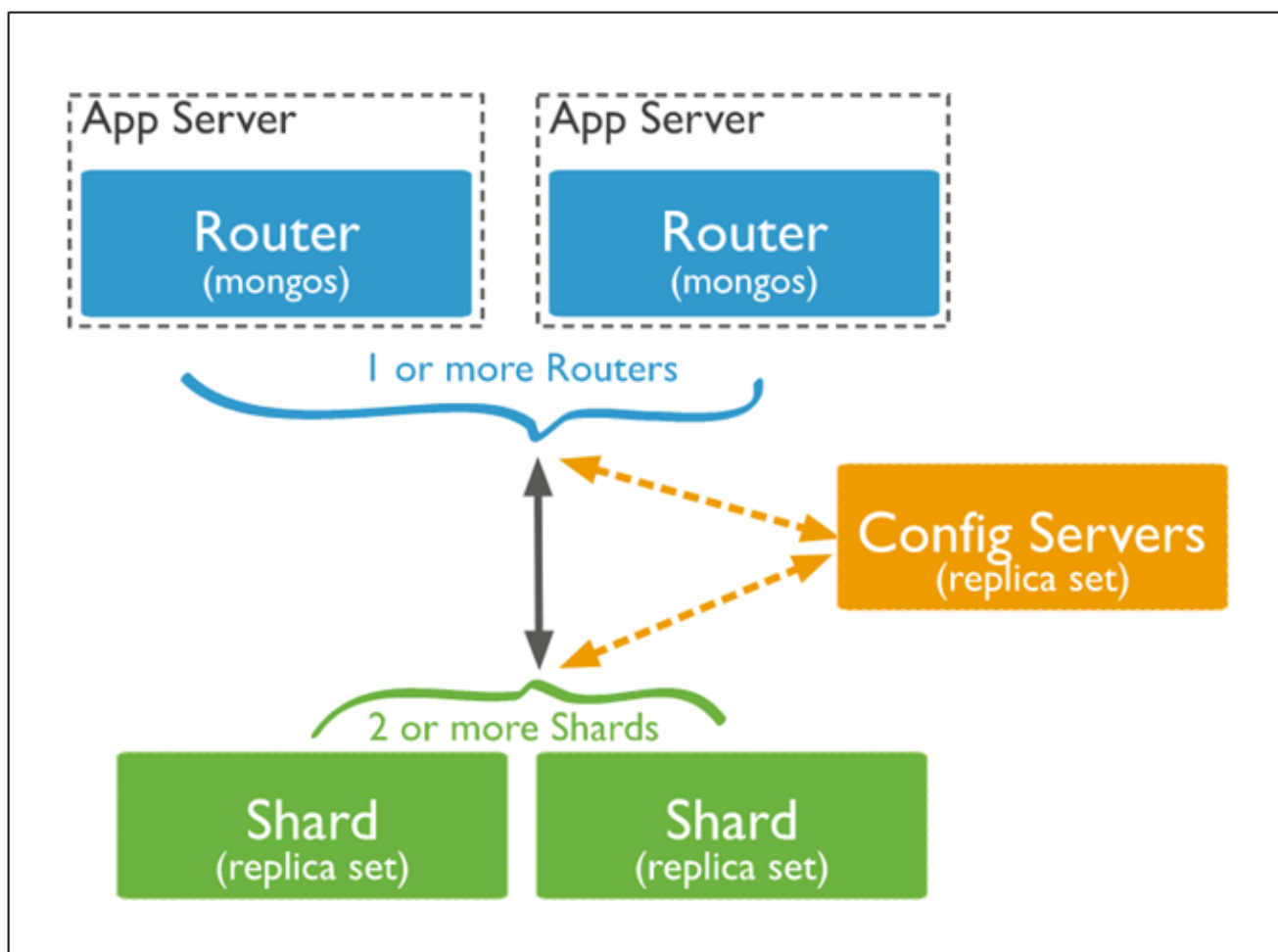


- 保证集群总是可读写
  - MongoDB通过多种途径来确保集群的可用性和可靠性。
  - 将MongoDB的分片和复制功能结合使用，在确保数据分片到多台服务器的同时，也确保了每份数据都有相应的备份，这样就可以确保有服务器换掉时，其他的从库可以立即接替坏掉的部分继续工作。
- 使集群易于扩展

当系统需要更多的空间和资源的时候，MongoDB使我们可以按需方便的扩充系统容量。

## 2.3、架构

组件	说明
Config Server	存储集群所有节点、分片数据路由信息。默认需要配置3个Config Server节点。
Mongos	提供对外应用访问，所有操作均通过mongos执行。一般有多个mongos节点。数据迁移和数据自动平衡。
Mongod	存储应用数据记录。一般有多个Mongod节点，达到数据分片目的。



Mongos本身并不持久化数据，Sharded cluster所有的元数据都会存储到Config Server，而用户的数据会分散存储到各个shard。Mongos启动后，会从配置服务器加载元数据，开始提供服务，将用户的请求正确路由到对应的分片。



当数据写入时，MongoDB Cluster根据分片键设计写入数据。当外部语句发起数据查询时，MongoDB根据数据分布自动路由至指定节点返回数据。

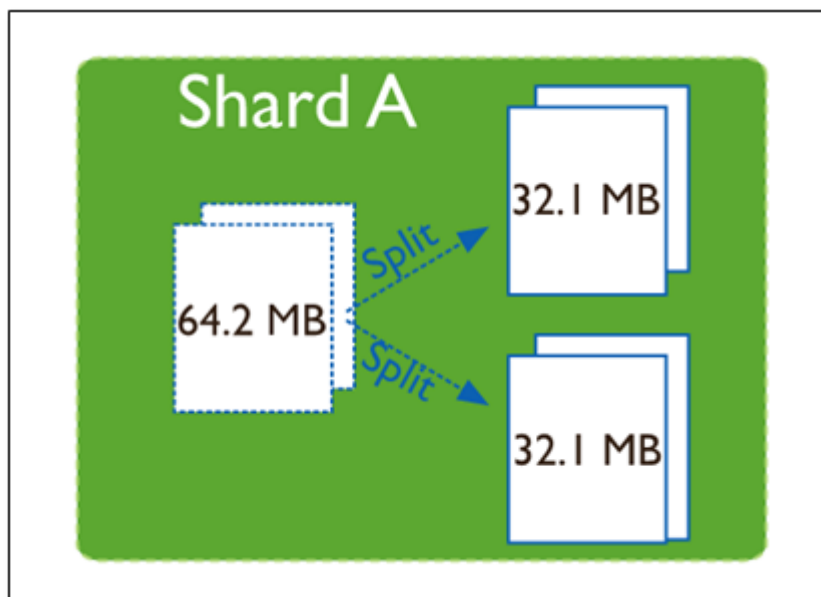
## 2.4、集群中数据分布

在一个shard server内部，MongoDB会把数据分为chunks，每个chunk代表这个shard server内部一部分数据。chunk的产生，会有以下两个用途：

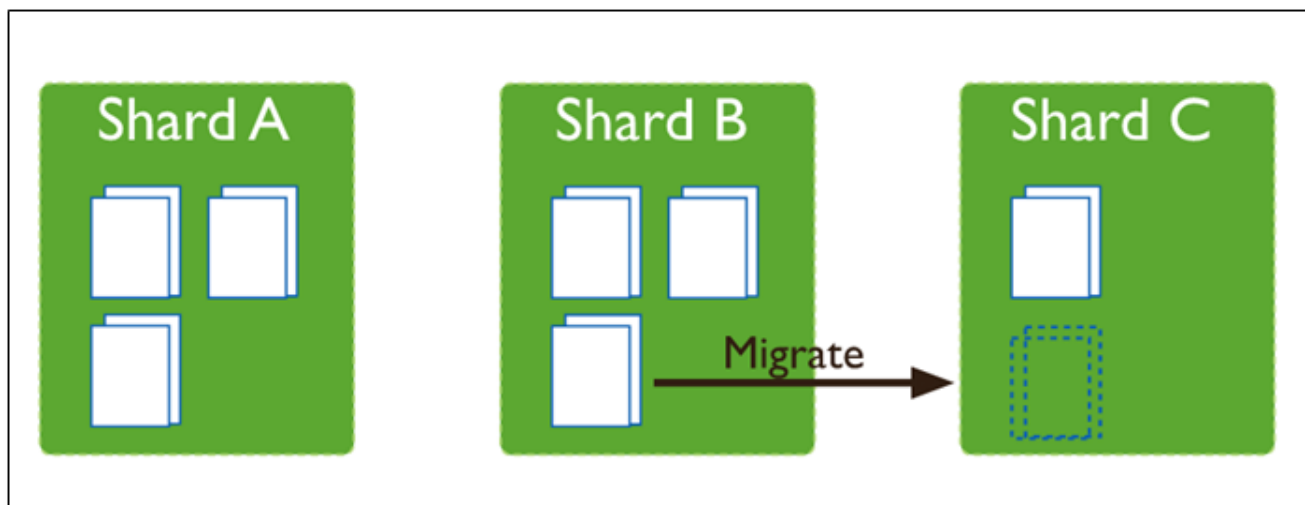
- **Splitting**：当一个chunk的大小超过配置中的chunk size时，MongoDB的后台进程会把这个chunk切分成更小的chunk，从而避免chunk过大的情况
- **Balancing**：在MongoDB中，balancer是一个后台进程，负责chunk的迁移，从而均衡各个shard server的负载，系统初始1个chunk，chunk size默认值64M,生产库上选择适合业务的chunk size是最好的。mongoDB会自动拆分和迁移chunks。

### 2.4.1、chunk分裂及迁移

随着数据的增长，其中的数据大小超过了配置的chunk size，默认是64M，则这个chunk就会分裂成两个。数据的增长会让chunk分裂得越来越多。



这时候，各个shard 上的chunk数量就会不平衡。mongos中的一个组件balancer 就会执行自动平衡。把chunk从chunk数量最多的shard节点挪动到数量最少的节点。



### 2.4.2、chunksize

chunk的分裂和迁移非常消耗IO资源；chunk分裂的时机：在插入和更新，读数据不会分裂。

- 小的chunksize：数据均衡是迁移速度快，数据分布更均匀。数据分裂频繁，路由节点消耗更多资源。
- 大的chunksize：数据分裂少。数据块移动集中消耗IO资源。

适合业务的chunksize是最好的。

#### chunkSize 对分裂及迁移的影响

- MongoDB 默认的 chunkSize 为64MB，如无特殊需求，建议保持默认值；chunkSize 会直接影响到 chunk 分裂、迁移的行为。
- chunkSize 越小，chunk 分裂及迁移越多，数据分布越均衡；反之，chunkSize 越大，chunk 分裂及迁移会更少，但可能导致数据分布不均。
- chunk 自动分裂只会在数据写入时触发，所以如果将 chunkSize 改小，系统需要一定的时间来将 chunk 分裂到指定的大小。
- chunk 只会分裂，不会合并，所以即使将 chunkSize 改大，现有的 chunk 数量不会减少，但 chunk 大小会随着写入不断增长，直到达到目标大小。

## 2.5、搭建集群

```
1  #创建3个config节点
2  docker create --name configsvr01 -p 17000:27019 -v mongoconfigsvr-data-01:/data/configdb mongo:4.0.3 --configsvr --replSet "rs_configsvr" --bind_ip_all
3
4  docker create --name configsvr02 -p 17001:27019 -v mongoconfigsvr-data-02:/data/configdb mongo:4.0.3 --configsvr --replSet "rs_configsvr" --bind_ip_all
5
6  docker create --name configsvr03 -p 17002:27019 -v mongoconfigsvr-data-03:/data/configdb mongo:4.0.3 --configsvr --replSet "rs_configsvr" --bind_ip_all
7
8  #启动服务
9  docker start configsvr01 configsvr02 configsvr03
10
11 #进去容器进行操作
12 docker exec -it configsvr01 /bin/bash
13 mongo 172.16.55.185:17000
```



```
14
15 #集群初始化
16 rs.initiate(
17     {
18         _id: "rs_configsvr",
19         configsvr: true,
20         members: [
21             { _id : 0, host : "172.16.55.185:17000" },
22             { _id : 1, host : "172.16.55.185:17001" },
23             { _id : 2, host : "172.16.55.185:17002" }
24         ]
25     }
26 )
27
28 #创建2个shard集群，每个集群都有3个数据节点
29
30 #集群一
31 docker create --name shardsvr01 -p 37000:27018 -v mongoshardsvr-data-01:/data/db
  mongo:4.0.3 --replSet "rs_shardsvr1" --bind_ip_all --shardsvr
32 docker create --name shardsvr02 -p 37001:27018 -v mongoshardsvr-data-02:/data/db
  mongo:4.0.3 --replSet "rs_shardsvr1" --bind_ip_all --shardsvr
33 docker create --name shardsvr03 -p 37002:27018 -v mongoshardsvr-data-03:/data/db
  mongo:4.0.3 --replSet "rs_shardsvr1" --bind_ip_all --shardsvr
34
35 #集群二
36 docker create --name shardsvr04 -p 37003:27018 -v mongoshardsvr-data-04:/data/db
  mongo:4.0.3 --replSet "rs_shardsvr2" --bind_ip_all --shardsvr
37 docker create --name shardsvr05 -p 37004:27018 -v mongoshardsvr-data-05:/data/db
  mongo:4.0.3 --replSet "rs_shardsvr2" --bind_ip_all --shardsvr
38 docker create --name shardsvr06 -p 37005:27018 -v mongoshardsvr-data-06:/data/db
  mongo:4.0.3 --replSet "rs_shardsvr2" --bind_ip_all --shardsvr
39
40 #启动容器
41 docker start shardsvr01 shardsvr02 shardsvr03
42 docker start shardsvr04 shardsvr05 shardsvr06
43
44 #进去容器执行
45 docker exec -it shardsvr01 /bin/bash
46 mongo 172.16.55.185:37000
47
48 #初始化集群
49 rs.initiate(
50     {
51         _id: "rs_shardsvr1",
52         members: [
53             { _id : 0, host : "172.16.55.185:37000" },
54             { _id : 1, host : "172.16.55.185:37001" },
55             { _id : 2, host : "172.16.55.185:37002" }
56         ]
57     }
58 )
59
60 #初始化集群二
```



```
61 mongo 172.16.55.185:37003
62
63 rs.initiate(
64   {
65     _id: "rs_shardsvr2",
66     members: [
67       { _id : 0, host : "172.16.55.185:37003" },
68       { _id : 1, host : "172.16.55.185:37004" },
69       { _id : 2, host : "172.16.55.185:37005" }
70     ]
71   }
72 )
73
74 #创建mongos节点容器，需要指定config服务
75 docker create --name mongos -p 6666:27017 --entrypoint "mongos" mongo:4.0.3 --
configdb rs_configsvr/172.16.55.185:17000,172.16.55.185:17001,172.16.55.185:17002 --
bind_ip_all
76
77 docker start mongos
78
79 #进入容器执行
80 docker exec -it mongos bash
81 mongo 172.16.55.185:6666
82
83 #添加shard节点
84 sh.addShard("rs_shardsvr1/172.16.55.185:37000,172.16.55.185:37001,172.16.55.185:37002
")
85 sh.addShard("rs_shardsvr2/172.16.55.185:37003,172.16.55.185:37004,172.16.55.185:37005
")
86
87 #启用分片
88 sh.enableSharding("test")
89
90 #设置分片规则，按照_id的hash进行区分
91 sh.shardCollection("test.order", {"_id": "hashed" })
92
93 #插入测试数据
94 use test
95
96 for (i = 1; i <= 1000; i=i+1){
97   db.order.insert({'id':i , 'price': 100+i})
98 }
99
100 #分别在2个shard集群中查询数据进行测试
101 db.order.count()
102
103
104 #集群操作（在mongos中执行）
105 use config
106 db.databases.find() #列出所有数据库分片情况
107 db.collections.find() #查看分片的片键
108 sh.status() #查询分片集群的状态信息
109
```

## 3、日志规范

日志对项目而言，其重要性不言而喻，**如果没有日志，生成环境的问题就无法定位**，面对多种日志框架，如：JDK14、simple、Log4j、Logback等，视乎我们有开始纠结了，到底使用哪个呢？

### 3.1、阿里巴巴开发手册

首先从《阿里巴巴Java开发手册》说起，在手册中提到：

1. **【强制】**应用中不可直接使用日志系统（Log4j、Logback）中的 API，而应依赖使用日志框架 SLF4J 中的 API，使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。

```
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
private static final Logger logger = LoggerFactory.getLogger(ABC.class);
```

为什么是强制？为什么要求必须使用SLF4J？SLF4J的作用到底是什么？

### 3.2、了解SLF4J

**Java简易日志门面**（Simple Logging Facade for Java，缩写SLF4J），是一套包装Logging 框架的界面程式，以外观模式实现。可以在软件部署的时候决定要使用的 Logging 框架，目前主要支援的有Java Logging API、Log4j及logback等框架。以MIT 授权方式发布。

SLF4J 的作者就是 Log4j和Logback 的作者 Ceki Gülcü，他宣称 SLF4J 比 Log4j 更有效率，而且比 Apache Commons Logging (JCL) 简单、稳定。

其实，SLF4J其实只是一个门面服务而已，他并不是真正的日志框架，真正的日志的输出相关的实现还是要依赖 Log4j、logback等日志框架的。

官网：<https://www.slf4j.org/>

### 3.3、SLF4J的使用

#### 3.3.1、创建工程



导入slf4j依赖：

```
1 <dependencies>
2   <dependency>
3     <groupId>org.slf4j</groupId>
4     <artifactId>slf4j-api</artifactId>
5     <version>1.7.26</version>
6   </dependency>
7 </dependencies>
```

### 3.3.2、编写日志代码

```
1 package cn.itcast.log.Demo;
2
3 import org.slf4j.Logger;
4 import org.slf4j.LoggerFactory;
5
6 public class Demo {
7
8     private static final Logger LOGGER = LoggerFactory.getLogger(Demo.class);
9
10    public static void main(String[] args) {
11
12        LOGGER.info("这是info日志信息。代码之前进行打印信息。");
13    }
```

```
13
14     System.out.println("测试代码");
15
16     LOGGER.error("这是error日志信息。代码之后进行打印信息。");
17 }
18 }
19
```

### 3.3.3、与JDK14整合

导入依赖：

```
1 <dependency>
2     <groupId>org.slf4j</groupId>
3     <artifactId>slf4j-jdk14</artifactId>
4     <version>1.7.26</version>
5 </dependency>
```

运行上面的代码进行测试：

```
Demo x
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
测试代码
三月 04, 2019 6:17:56 下午 cn.itcast.log.Demo.Demo main
信息：这是info日志信息。代码之前进行打印信息。
三月 04, 2019 6:17:56 下午 cn.itcast.log.Demo.Demo main
严重：这是error日志信息。代码之后进行打印信息。
```

### 3.3.4、与Simple整合

导入依赖：（将slf4j-jdk14依赖注释掉）

```
1 <dependency>
2     <groupId>org.slf4j</groupId>
3     <artifactId>slf4j-simple</artifactId>
4     <version>1.7.26</version>
5 </dependency>
```

运行上面的代码进行测试：

```
Demo x
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
[main] INFO cn.itcast.log.Demo.Demo - 这是info日志信息。代码之前进行打印信息。
测试代码
[main] ERROR cn.itcast.log.Demo.Demo - 这是error日志信息。代码之后进行打印信息。
|
Process finished with exit code 0
```

### 3.3.5、与Log4j整合

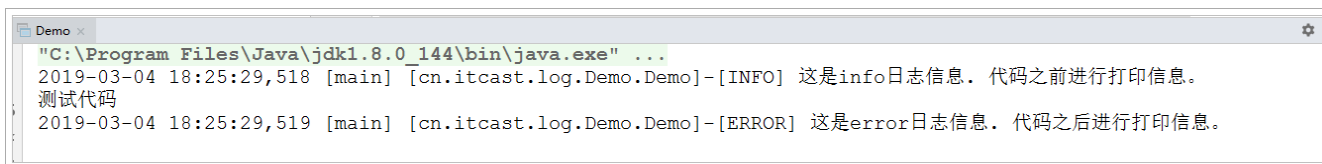


```
1 <dependency>
2   <groupId>org.slf4j</groupId>
3   <artifactId>slf4j-log4j12</artifactId>
4   <version>1.7.26</version>
5 </dependency>
```

log4j还需要配置log4j.properties文件：

```
1 log4j.rootLogger=DEBUG,A1
2
3 log4j.appender.A1=org.apache.log4j.ConsoleAppender
4 log4j.appender.A1.layout=org.apache.log4j.PatternLayout
5 log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p]
  %m%n
```

运行上面的代码进行测试：

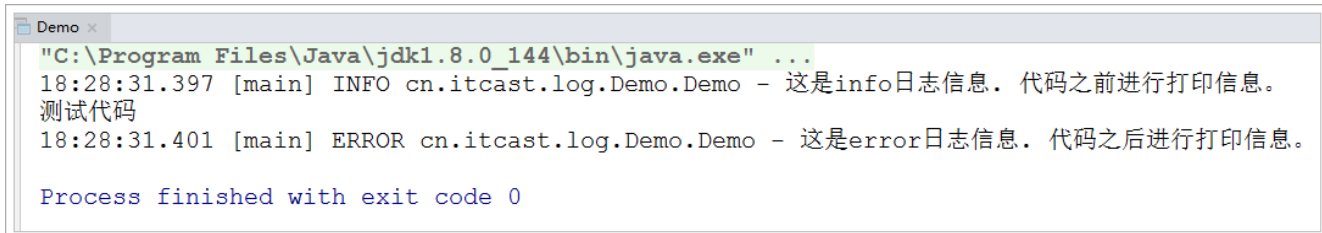


### 3.3.6、与Logback整合

导入依赖：

```
1 <dependency>
2   <groupId>ch.qos.logback</groupId>
3   <artifactId>logback-core</artifactId>
4   <version>1.1.9</version>
5 </dependency>
6 <dependency>
7   <groupId>ch.qos.logback</groupId>
8   <artifactId>logback-classic</artifactId>
9   <version>1.1.9</version>
10 </dependency>
```

运行效果如下：



### 3.3.7、小结

可以看到，无论和什么日志框架整合，我们编写的代码都不需要修改，只需要变更依赖即可，这就是slf4j框架的意义所在。

## 3.4、手册中的其它规范



2. 【强制】日志文件推荐至少保存 15 天，因为有些异常具备以“周”为频次发生的特点。

3. 【强制】应用中的扩展日志（如打点、临时监控、访问日志等）命名方式：

appName\_logType\_logName.log。logType:日志类型，推荐分类有

stats/desc/monitor/visit 等；logName:日志描述。这种命名的好处：通过文件名就可知道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找。

——禁止用于商业用途，违者必究——

19 / 35

#### 阿里巴巴 Java 开发手册

**正例：**mppserver 应用中单独监控时区转换异常，如：

mppserver\_monitor\_timeZoneConvert.log

**说明：**推荐对日志进行分类，如将错误日志和业务日志分开存放，便于开发人员查看，也便于通过日志对系统进行及时监控。

4. 【强制】对 trace/debug/info 级别的日志输出，必须使用条件输出形式或者使用占位符的方式。

**说明：**logger.debug("Processing trade with id: " + id + " and symbol: " + symbol);  
如果日志级别是 warn, 上述日志不会打印, 但是会执行字符串拼接操作, 如果 symbol 是对象, 会执行 toString() 方法, 浪费了系统资源, 执行了上述操作, 最终日志却没有打印。

**正例：**（条件）

```
if (logger.isDebugEnabled()) {  
    logger.debug("Processing trade with id: " + id + " and symbol: " + symbol);  
}
```

**正例：**（占位符）

```
logger.debug("Processing trade with id: {} and symbol : {}", id, symbol);
```



5. 【强制】避免重复打印日志，浪费磁盘空间，务必在 `log4j.xml` 中设置 `additivity=false`。

正例：`<logger name="com.taobao.dubbo.config" additivity="false">`

6. 【强制】异常信息应该包括两类信息：案发现场信息和异常堆栈信息。如果不处理，那么通过关键字 `throws` 往上抛出。

正例：`logger.error(各类参数或者对象 toString + "_" + e.getMessage(), e);`

7. 【推荐】谨慎地记录日志。生产环境禁止输出 `debug` 日志；有选择地输出 `info` 日志；如果使用 `warn` 来记录刚上线时的业务行为信息，一定要注意日志输出量的问题，避免把服务器磁盘撑爆，并记得及时删除这些观察日志。

说明：大量地输出无效日志，不利于系统性能提升，也不利于快速定位错误点。记录日志时请思考：这些日志真的有人看吗？看到这条日志你能做什么？能不能给问题排查带来好处？

8. 【参考】可以使用 `warn` 日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适从。注意日志输出的级别，`error` 级别只记录系统逻辑出错、异常等重要的错误信息。如非必要，请不要在此场景打出 `error` 级别。

## 4、异常规范

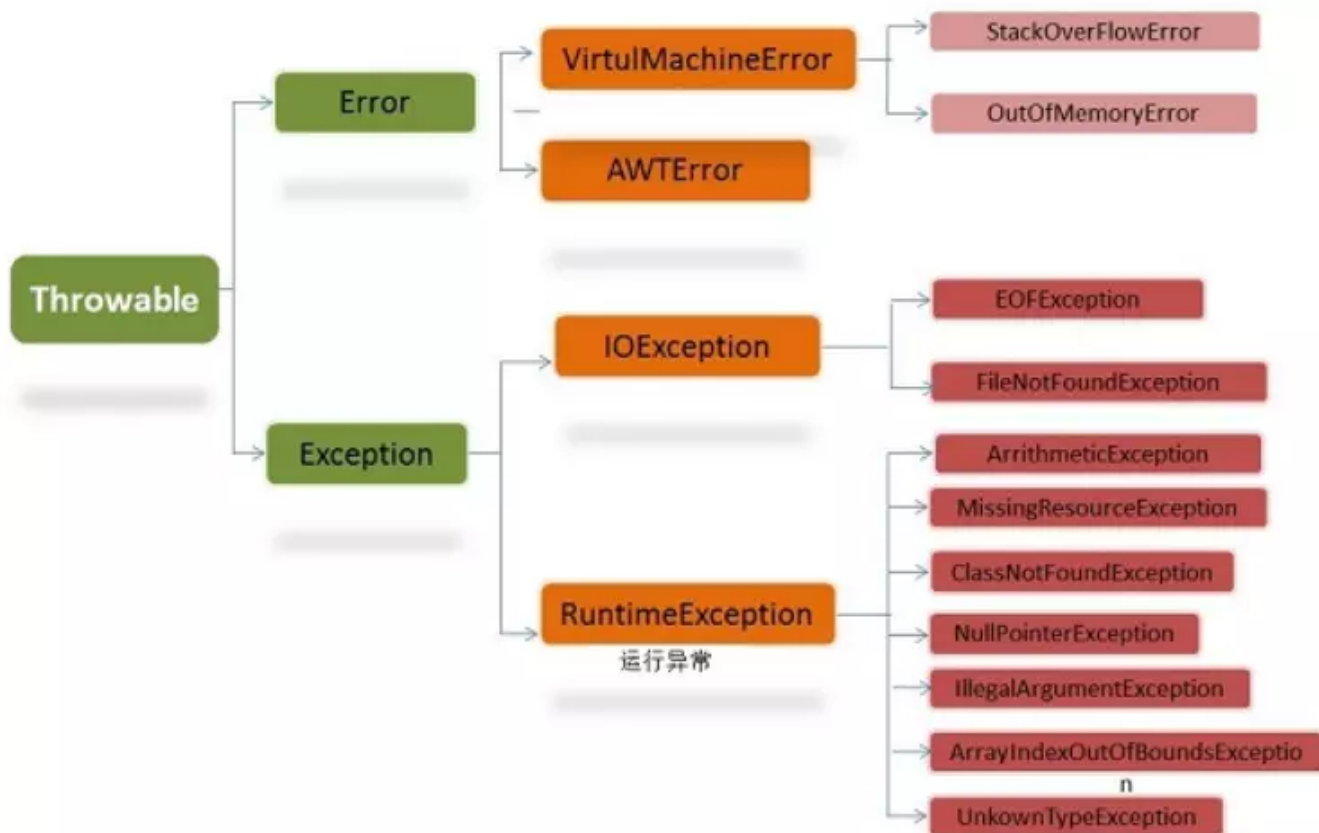
### 4.1、异常概念

异常是发生在程序执行过程中阻碍程序正常执行的错误事件，当一个程序出现错误时，可能的情况有如下3种：

- 语法错误 代码的格式错了，某个字母输错了
- 运行时错误 空指针异常，数组越界，除数为零等
- 逻辑错误 运行结果与预想的结果不一样，这是一种很难调试的错误

Java中的异常处理机制主要处理运行时错误。

异常分类：



在 Java 应用程序中，异常处理机制为：抛出异常，捕捉异常。

- 抛出异常

- 当一个方法出现错误引发异常时，方法创建异常对象并交付运行时系统，异常对象中包含了异常类型和异常出现时的程序状态等异常信息。运行时系统负责寻找处置异常的代码并执行。

- 捕获异常

- 在方法抛出异常之后，运行时系统将转为寻找合适的异常处理器（exception handler），进行处理。

## 4.2、异常规范



1. 【强制】Java 类库中定义的一类 `RuntimeException` 可以通过预先检查进行规避，而不应该通过 `catch` 来处理，比如：`IndexOutOfBoundsException`，`NullPointerException` 等等。

说明：无法通过预检查的异常除外，如在解析一个外部传来的字符串形式数字时，通过 `catch NumberFormatException` 来实现。

正例：`if (obj != null) {...}`

反例：`try { obj.method() } catch (NullPointerException e) {...}`

2. 【强制】异常不要用来做流程控制，条件控制，因为异常的处理效率比条件分支低。
3. 【强制】对大段代码进行 `try-catch`，这是不负责任的表现。`catch` 时请分清稳定代码和非稳定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的 `catch` 尽可能进行区分异常类型，再做对应的异常处理。
4. 【强制】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请将异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的内容。

5. 【强制】有 `try` 块放到了事务代码中，`catch` 异常后，如果需要回滚事务，一定要注意手动回滚事务。

6. 【强制】`finally` 块必须对资源对象、流对象进行关闭，有异常也要做 `try-catch`。

说明：如果 JDK7 及以上，可以使用 `try-with-resources` 方式。

7. 【强制】不能在 `finally` 块中使用 `return`，`finally` 块中的 `return` 返回后方法结束执行，不会再执行 `try` 块中的 `return` 语句。

8. 【强制】捕获异常与抛异常，必须是完全匹配，或者捕获异常是抛异常的父类。

说明：如果预期对方抛的是绣球，实际接到的是铅球，就会产生意外情况。

9. 【推荐】方法的返回值可以为 `null`，不强制返回空集合，或者空对象等，必须添加注释充分说明什么情况下会返回 `null` 值。调用方需要进行 `null` 判断防止 NPE 问题。

说明：本手册明确防止 NPE 是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败、序列化失败、运行时异常等场景返回 `null` 的情况。



10. 【推荐】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景：

1) 返回类型为基本数据类型，return 包装数据类型的对象时，自动拆箱有可能产生 NPE。

反例：public int f() { return Integer 对象}， 如果为 null，自动解箱抛 NPE。

——禁止用于商业用途，违者必究——

18 / 35

#### 阿里巴巴 Java 开发手册

2) 数据库的查询结果可能为 null。

3) 集合里的元素即使 isEmpty，取出的数据元素也可能为 null。

4) 远程调用返回对象时，一律要求进行空指针判断，防止 NPE。

5) 对于 Session 中获取的数据，建议 NPE 检查，避免空指针。

6) 级联调用 obj.getA().getB().getC()；一连串调用，易产生 NPE。

正例：使用 JDK8 的 Optional 类来防止 NPE 问题。

11. 【推荐】定义时区分 unchecked / checked 异常，避免直接抛出 new RuntimeException()，更不允许抛出 Exception 或者 Throwable，应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：DAOException / ServiceException 等。

12. 【参考】在代码中使用“抛异常”还是“返回错误码”，对于公司外的 http/api 开放接口必须使用“错误码”；而应用内部推荐异常抛出；跨应用间 RPC 调用优先考虑使用 Result 方式，封装 isSuccess()方法、“错误码”、“错误简短信息”。

说明：关于 RPC 方法返回方式使用 Result 方式的理由：

1) 使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。

2) 如果不加栈信息，只是 new 自定义异常，加入自己的理解的 error message，对于调用端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输的性能损耗也是问题。

13. 【参考】避免出现重复的代码 (Don't Repeat Yourself)，即 DRY 原则。

说明：随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是组件化。

正例：一个类中有多个 public 方法，都需要进行数行相同的参数校验操作，这个时候请抽取：

```
private boolean checkParam(DTO dto) {...}
```

## 5、其它规范

阿里巴巴写的Java开发代码规范质量还是很高的，建议学员们进行查看学习。

推荐：

- 编程规范s

- MySQL数据库规范
- 单元测试