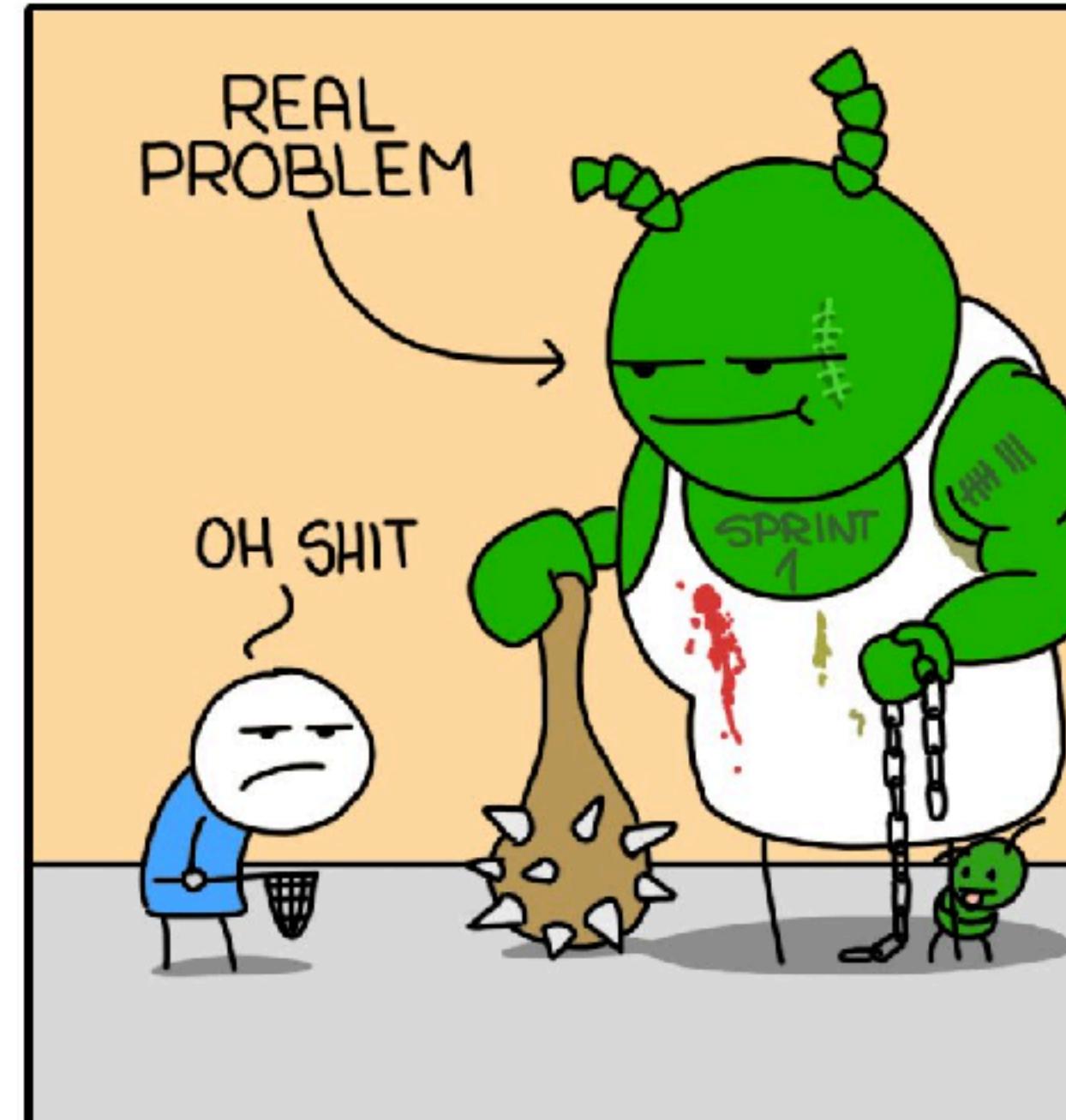
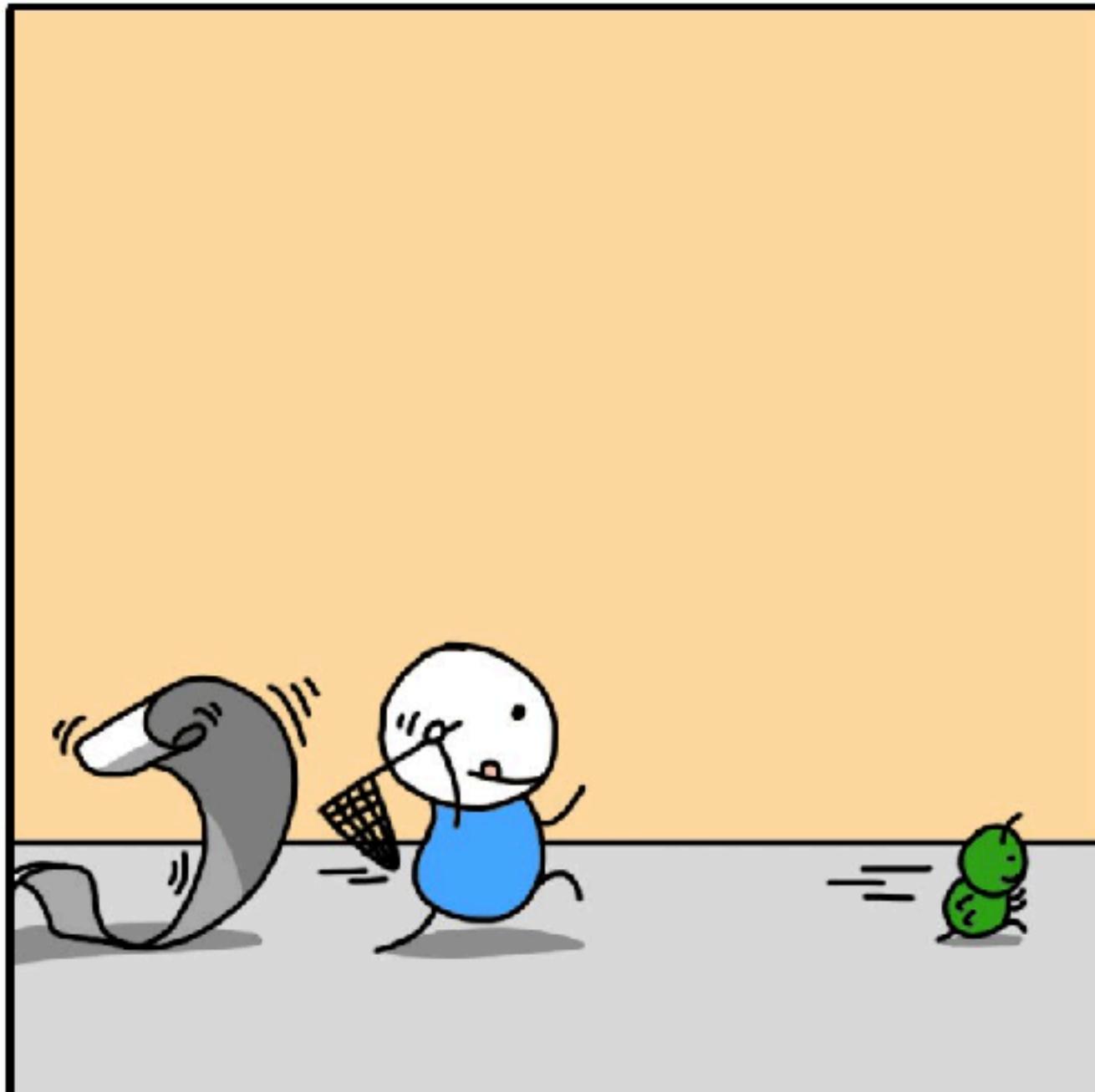




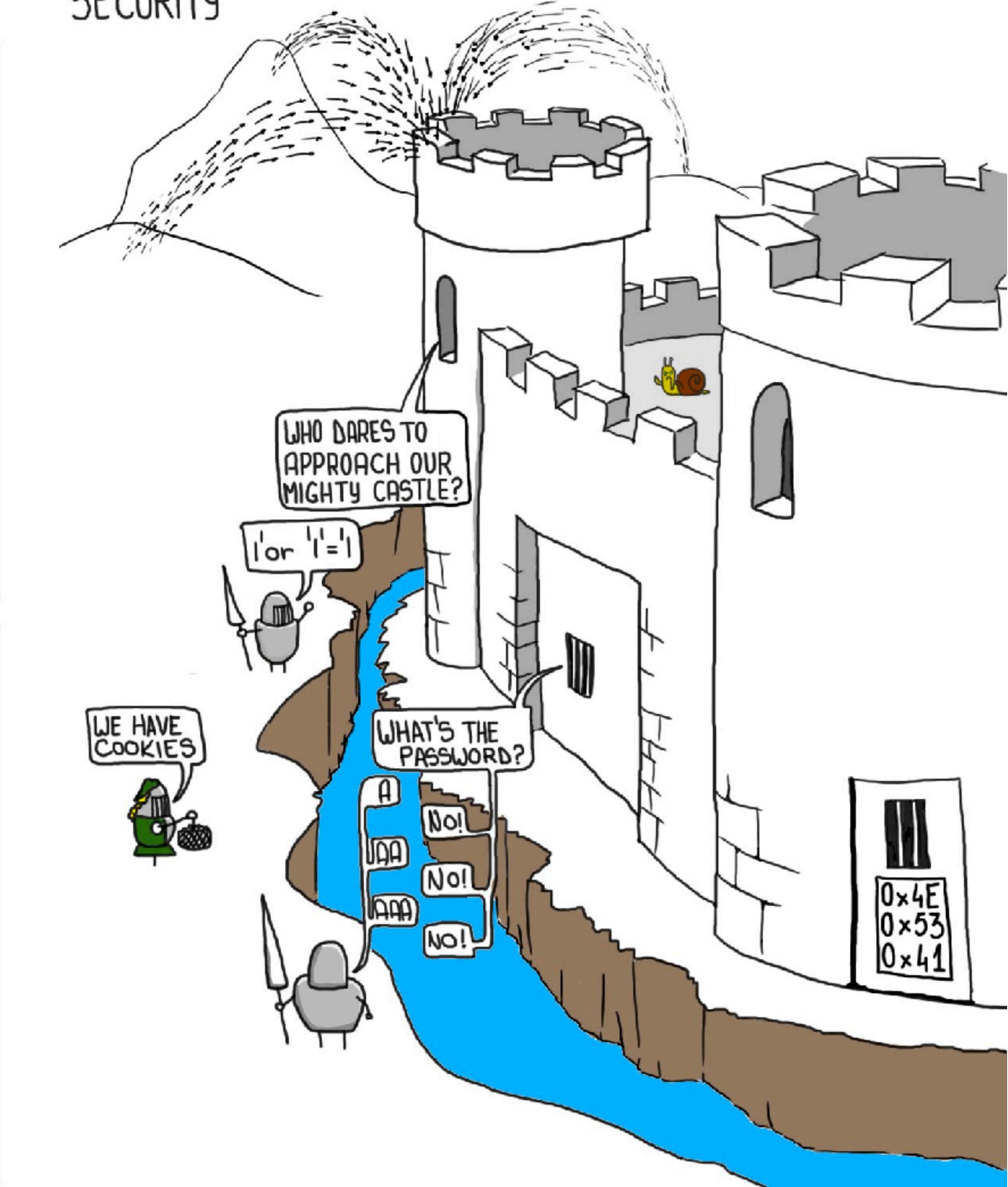
JAVA 达摩班

NoSQL

## ROOT CAUSE



## SECURITY





# Redis安装 与常用命令

1

# Redis概念

**Redis = REmote DIctionary Server**

- Redis由Salvatore Sanfilippo开发于2009年，Pieter Noordhuis和Matt Stancliff也做出了贡献
- 开源（BSD），灵活，内存数据结构存储
- 可用作数据库，缓存和消息经纪人
- 支持丰富的数据结构：strings, hashes, sets, lists, sorted sets, bitmaps, hyperloglogs 和 geospatial indexes
- 内置备份，Lua脚本，LRU驱出算法，事务和不同类型的on-disk持久化
- 结合Redis集群，通过哨兵和自动分区功能实现高可用性
- 多系统支持，Linux, OSX, OpenBSD, NetBSD, FreeBSD；支持大端和小端架构，32位和64位系统



# Redis概念

特性：

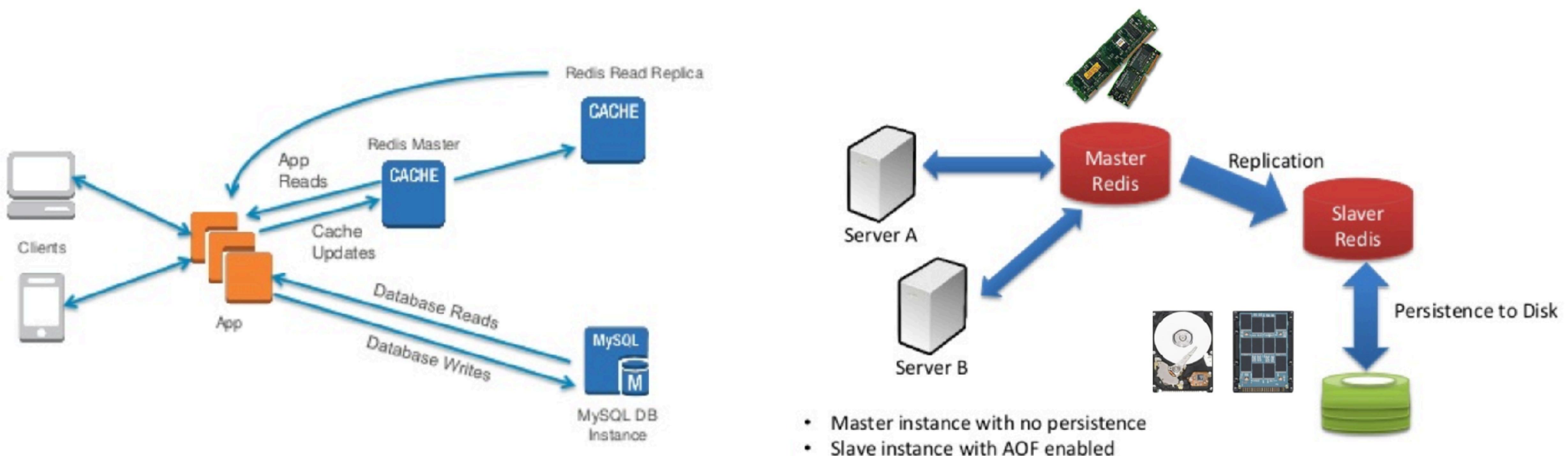
- **速度**：整个数据集加载在内存中，操作速度高达110,000 SETs/每秒和81,000 GETs/每秒，Redis 支持命令管道，一行命令实现多个get 和set操作计算，从而提高client通信速度。
- **持久化**：所有数据在内存中，数据改变是异步的保存在硬盘上，Redis提供了基于运行时间和更新次数的保存策略。Redis支持追加型（append-only）持久化模型。
- **数据结构**：支持多种数据结构：strings, hashes, sets, lists, sorted sets with range queries, bitmaps, hyperloglogs and geospatial索引用于半径查询.
- **原子操作**：安全
- **多语言支持**：ActionScript, C, C++, C#, Clojure, Common Lisp, D, Dart, Erlang, Go, Haskell, Haxe, Io, Java, JavaScript (Node.js), Julia, Lua, Objective-C, Perl, PHP, Pure Data, Python, R, Racket, Ruby, Rust, Scala, Smalltalk and Tcl.
- **主/从（Master/Slave）备份**：Redis支持简单的Master/Slave备份。配置简单，同步速度快。
- **分片**：跨多个Redis实例的分布式存储实现简单，就像普通的key-value操作。
- **便携式**：Redis基于ANSI C实现，可以运行在大多数POSIX系统，包括Linux, BSD, Mac OS X, Solaris等。Redis通过Cygwin可以运行在WIN32系统下，但非官方支持。

Redis和其它key-value型数据库区别？

- Redis提供多种数据类型数据，并且提供数据的原子操作。
- Redis是内存型，但存储在硬盘的数据库，所以它具有较高的读写速度，但数据集体积受到内存所限。同时，相比硬盘，在内存操作复杂数据结构操作更简单，所以它能实现更多内部操作。

# Redis架构

Redis是一种高级的key-value存储实现，可以用于NoSQL数据库，也可以用作内存缓存（memory-cache），这样可以提高存储在系统内存中的数据性能。



# Redis架构

**最大内存：**64位系统默认没有内存限制，32位系统默认提供3GB内存限制。大内存明显会增加命中率，但是一定程度的内存命中率也会保持同样的级别。

**淘汰算法：**当缓存到达内存限制（或人工配置上限），旧数据需要被新数据替换，redis有如下替换策略：

**Last Recently Used (LRU)** 检测key的最后一次使用时间。可以某个key长久没有时间，但最近被使用过，它同样不会被淘汰。

**Least Frequently Used (LFU)** (redis 4.0开始支持) 统计key的使用次数。最常用的key会留下来，使用次数少的会被替换。它存在刚刚被缓存的，但只用过一次的key会被替换的风险，redis团队通过降低长期未被调用的key的使用次数解决上面问题。

**持久化：**系统启动后缓存为空，持久化对于恢复常用数据是有用的。redis支持三种类型之久化：

**RDB**：在一定时间间隔后，或者写次数后，它会在某一时刻备份快照（point-in-time snapshots）。该算法我们需要在快照间隔时间和淘汰过期数据之间寻找平衡。

**AOF**：为每一个写操作创建持久化日志。为取得最好性能，需要考虑appendfsync配置参数下的fsync策略。同时使用RDB和AOF。

**谨记：**任何额外配置或操作如fsync会消耗CPU，所以非必要情况下，不需要打开所有持久化选项。

# 安装并启动

安装最新版redis, 4.0.\*

*brew install redis*

启动redis服务器

守护进程: *brew services start redis*

正常启动: *redis-server /usr/local/etc/redis.conf*

启动redis客户端

*redis-cli*

*redis 127.0.0.1:6379> ping*

*PONG*

# 配置

Redis支持热配置，修改配置不需要重启redis服务

打开配置文件

*nano /usr/local/etc/redis.conf*

查看/修改配置

*config get save*

*config get \**

*config set save "900 1 300 10 60 100000"*

```
127.0.0.1:6379> config get save
1) "save"
2) "900 1 300 10 60 100000"
127.0.0.1:6379> config set save "900 1 300 10 60 100000"
OK
127.0.0.1:6379> config get save
1) "save"
2) "900 1 300 10 60 100000"
127.0.0.1:6379> █
```

完整的redis命令查看：<https://redis.io/commands>

# 类型

Redis是key-value存储类型，传统缓存数据库中key和value都只能是字符串，redis支持更多类型

- Binary-safe strings.
- Lists
- Sets
- Sorted sets
- Hashes
- Bit arrays (or simply bitmaps)
- HyperLogLogs

key是二进制安全的，支持任何二进制序列，最大512M  
空字符串，普通字符串，JPEG文件

# 类型

**Value: String类型**

**最常用的数据类型， 提供如下命令**

set name "dharma"  
get name  
del name  
incr name  
incrby name 5  
expire name 10

```
127.0.0.1:6379> set name "dharma"
OK
127.0.0.1:6379> get name
"dharma"
127.0.0.1:6379> del name
(integer) 1
127.0.0.1:6379> get name
(nil)
127.0.0.1:6379> incr name
(integer) 1
127.0.0.1:6379> incrby name 5
(integer) 6
127.0.0.1:6379> get name
"6"
127.0.0.1:6379> expire name 10
(integer) 1
127.0.0.1:6379> get name
"6"
127.0.0.1:6379> get name
"6"
127.0.0.1:6379> get name
(nil)
```

# 类型

**Value: List类型**

代表一组有序值， 基于linked list实现

lpush table b

rpush table f

lrange table 0 1

lpop table

rpop table

lpush table a

lrem table 1 a

lpush table a

lpush table b

lpush table c

ltrim table 1 2

```
127.0.0.1:6379> lpush table d
(integer) 1
127.0.0.1:6379> lpush table c
(integer) 2
127.0.0.1:6379> lpush table b
(integer) 3
127.0.0.1:6379> rpush table e
(integer) 4
127.0.0.1:6379> rpush table f
(integer) 5
127.0.0.1:6379> rpush table a
(integer) 6
127.0.0.1:6379> lpush table --
(integer) 7
127.0.0.1:6379> rpush table --
(integer) 8
127.0.0.1:6379> lrange table 0 7
1) "--"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
7) "a"
8) "--"
```

```
127.0.0.1:6379> lpop table
"--"
127.0.0.1:6379> rpop table
"--"
127.0.0.1:6379> lrange table 0 5
1) "b"
2) "c"
3) "d"
4) "e"
5) "f"
6) "a"
127.0.0.1:6379> lrem table 1 a
(integer) 1
127.0.0.1:6379> lpush table a
(integer) 6
127.0.0.1:6379> lrange table 0 5
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
127.0.0.1:6379> ltrim table 4 5
OK
127.0.0.1:6379> lrange table 0 5
1) "e"
2) "f"
```

# 类型

**Value: Set类型**

代表一组无序非重复值

sadd com a

sadd com b

sadd com a

smembers com

sinter com

sadd net b

sadd net c

sinter com net

sismember com m

srandmember net

srandmember net

```
127.0.0.1:6379> sadd com a
(integer) 1
127.0.0.1:6379> sadd com b
(integer) 1
127.0.0.1:6379> sadd com a
(integer) 0
127.0.0.1:6379> smembers com
1) "b"
2) "a"
127.0.0.1:6379> sinter com
1) "b"
2) "a"
127.0.0.1:6379> sadd net b
(integer) 1
127.0.0.1:6379> sadd net c
(integer) 1
127.0.0.1:6379> sinter com net
1) "b"
127.0.0.1:6379> sismember com m
(integer) 0
127.0.0.1:6379> srandmember net
"c"
127.0.0.1:6379> srandmember net
"b"
```

# 类型

**Value: Hash类型**  
代表有多个字段的对象

hset person linlin 18  
hmset persons a 1 b 2 c 3 d 4  
hmget persons c  
hget persons c  
hmget persons c a  
hgetall persons  
hgetall person  
hvals persons

```
127.0.0.1:6379> hset person linlin 18
(integer) 1
127.0.0.1:6379> hmset persons a 1 b 2 c 3 d 4
OK
127.0.0.1:6379> hget persons
(error) ERR wrong number of arguments for 'hget' command
127.0.0.1:6379> hmget persons
(error) ERR wrong number of arguments for 'hmget' command
127.0.0.1:6379> hmget persons c
1) "3"
127.0.0.1:6379> hget persons c
"3"
127.0.0.1:6379> hmget persons c a
1) "3"
2) "1"
127.0.0.1:6379> hgetall persons
1) "a"
2) "1"
3) "b"
4) "2"
5) "c"
6) "3"
7) "d"
8) "4"
127.0.0.1:6379> hgetall person
1) "linlin"
2) "18"
```

# 类型

Value: 有序Set类型

set和hash组合类型，每个元素都有一个分数

zadd box 0 a

zadd box 0 b

zadd box 0 c

zadd box 1 a

zadd box 1 d

zadd box 10 f

zadd box 1 m

zrange box 0 1

zrange box 1 4

zrange box 0 10

Zrevrange box 0 10

zrangebyscore box 1 9

zrem box d a

zrangebyscore box 1 2

```
127.0.0.1:6379> zadd box 0 a
(integer) 1
127.0.0.1:6379> zadd box 0 b
(integer) 1
127.0.0.1:6379> zadd box 0 c
(integer) 1
127.0.0.1:6379> zadd box 1 a
(integer) 0
127.0.0.1:6379> zadd box 1 d
(integer) 1
127.0.0.1:6379> zadd box 10 f
(integer) 1
127.0.0.1:6379> zadd 1 m
(error) ERR wrong number of arguments for 'zadd' command
127.0.0.1:6379> zadd box 1 m
(integer) 1
127.0.0.1:6379> zrange box
(error) ERR wrong number of arguments for 'zrange' command
127.0.0.1:6379> zrange box 0 1
1) "b"
2) "c"
127.0.0.1:6379> zrange box 1 4
1) "c"
2) "a"
3) "d"
4) "m"
```

```
127.0.0.1:6379> zrange box 0 10
1) "b"
2) "c"
3) "a"
4) "d"
5) "m"
6) "f"
127.0.0.1:6379> ZREVRANGE box 0 10
1) "f"
2) "m"
3) "d"
4) "a"
5) "c"
6) "b"
127.0.0.1:6379> zrangebyscore box 1 9
1) "a"
2) "d"
3) "m"
127.0.0.1:6379> zrem box d a
(integer) 2
127.0.0.1:6379> zrangebyscore box 1 2
1) "m"
```

# PUB/SUB

## Redis的发布者/订阅者模式

subscribe: 订阅

psubscribe: 按模式订阅

pubsub channels: 查看pub/sub状态

publish: 发布

unsubscribe 退订

punsubscribe 按模式退订

```
└─[130] <> redis-cli
127.0.0.1:6379> subscribe solo
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "solo"
3) (integer) 1
1) "message"
2) "solo"
3) "solo:hi"
1) "message"
2) "solo"
3) "solo: hello"
```

```
└─[1] <> redis-cli
127.0.0.1:6379> psubscribe com[012]
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "com[012]"
3) (integer) 1
1) "pmESSAGE"
2) "com[012]"
3) "com1"
4) "com:hi"
```

```
└─[1] <> redis-cli
127.0.0.1:6379> publish solo "solo:hi"
(integer) 1
127.0.0.1:6379> publish com1 "com:hi"
(integer) 1
127.0.0.1:6379> pubsub channels
1) "solo"
127.0.0.1:6379> unsubscribe solo
1) "unsubscribe"
2) "solo"
3) (integer) 0
127.0.0.1:6379> publish solo "solo: hello"
(integer) 1
127.0.0.1:6379> pubsub channels
1) "solo"
```

# 更多命令

**multi + exec**命令用于同时执行多条命令

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> incr name
QUEUED
127.0.0.1:6379> incr name
QUEUED
127.0.0.1:6379> incr name007
QUEUED
127.0.0.1:6379> exec
1) (integer) 1
2) (integer) 2
3) (integer) 1
127.0.0.1:6379> get name
"2"
127.0.0.1:6379> get name007
"1"
```

**ping** - 检测redis服务连接正常

**keys \*** - 列出所有redis keys

**flushall** - 删除所有redis存储

```
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> keys *
1) "7"
127.0.0.1:6379> flushall
OK
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> █
```

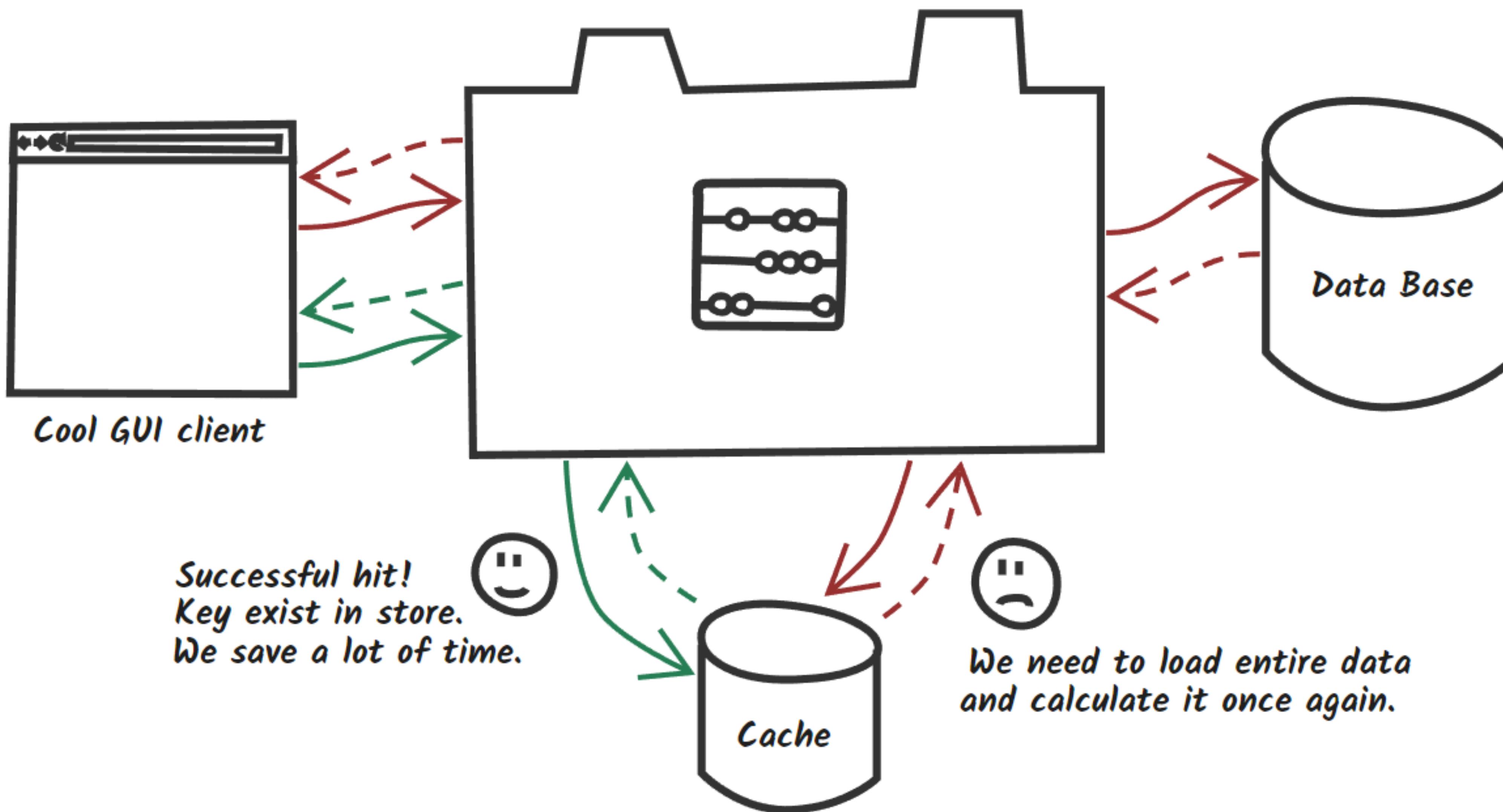


# Spring + Redis

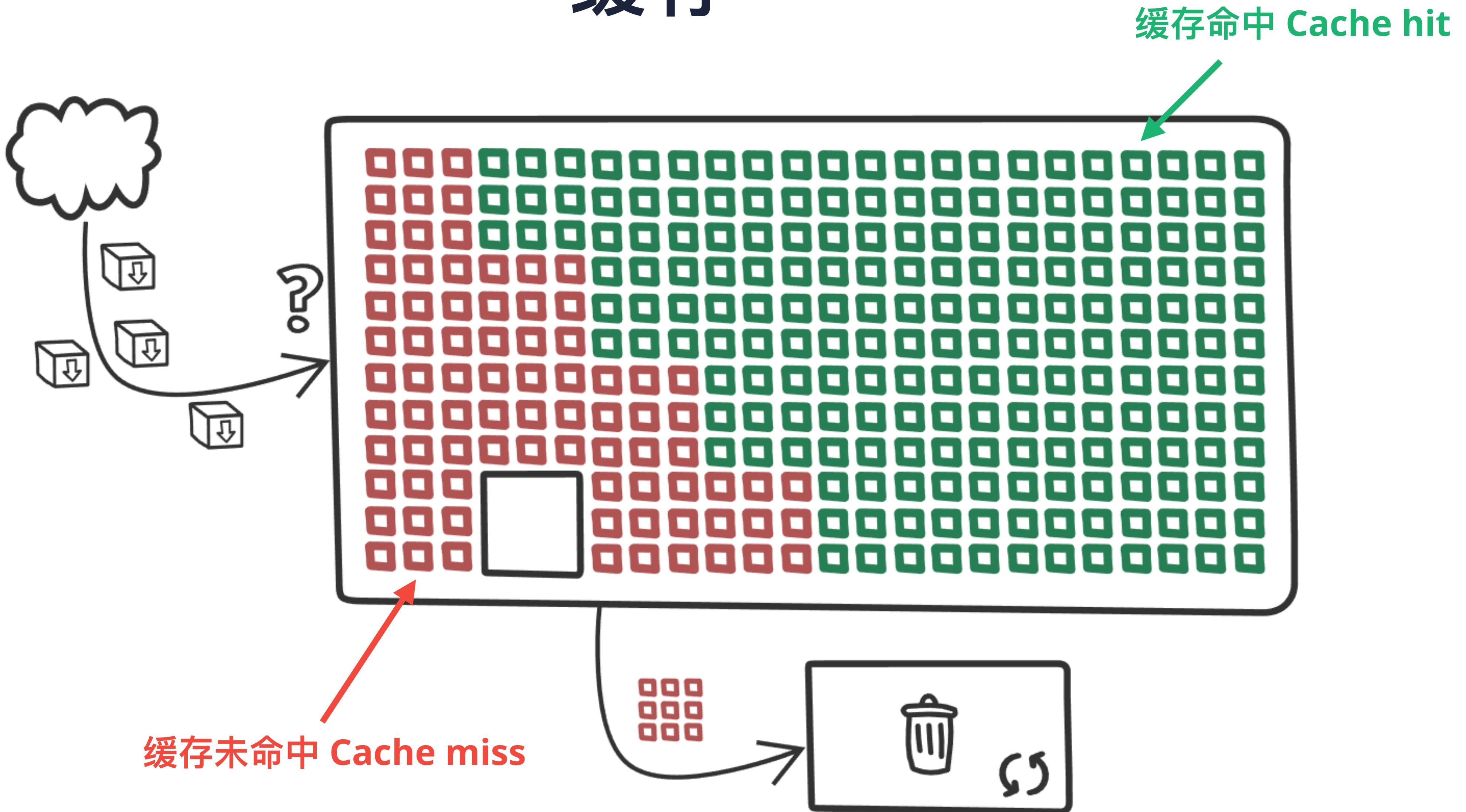
## 实现缓存

2

# 缓存



# 缓存



# Spring缓存注释

**Spring Framework**提供了一组通用的，实现缓存的注释，它可以用于多种缓存实现，包括Redis, EhCache, Hazelcast, Infinispan等

**@Cacheable**: 第一次方法调用后结果会被缓存，如果方法再次执行同样的参数，结果直接来自缓存。支持条件式缓存，即只缓存符合条件（命中率高）的执行结果。

```
@Cacheable(value = "post-single", key = "#id", unless = "#result.shares < 500")
@GetMapping("/{id}")
public Post getPostByID(@PathVariable String id) throws PostNotFoundException {
    log.info("get post with id {}", id);
    return postService.getPostByID(id);
}
@Cacheable(value = "post-top")
@GetMapping("/top")
public List<Post> getTopPosts() {
    return postService.getTopPosts();
}
```

# Spring缓存注释

**@CachePut:** 用于更新缓存结果

```
@CachePut(value = "post-single", key = "#post.id")
@PutMapping("/update")
public Post updatePostByID(@RequestBody Post post) throws PostNotFoundException {
    log.info("update post with id {}", post.getId());
    postService.updatePost(post);
    return post;
}
```

**@CacheEvict:** 删除缓存，可以删除一条，也可以删除所有缓存。

```
@CacheEvict(value = "post-single", key = "#id")
@DeleteMapping("/delete/{id}")
public void deletePostByID(@PathVariable String id) throws PostNotFoundException {
    log.info("delete post with id {}", id);
    postService.deletePost(id);
}
```

# Spring缓存注释

**@CacheEvict(value = "post-top")**

```
@GetMapping("/top/evict")
public void evictTopPosts() {
    log.info("Evict post-top");
}
```

**@EnableCaching**: 用于启动缓存，后置处理器会检查所有的bean是否带有缓存注释，如果是会创建代理去拦截方法调用。用于**SpringBootApplication**上。

**@Caching**: 聚合同类型的多个注释。

**@CacheConfig**: 类级别缓存配置，用于指定全局的缓存name，key生成器。

# Spring boot + redis

Spring框架提供了用于增加，更新和删除缓存的注释，并且spring boot提供了简化连接redis的配置  
注释可以应用到多种缓存实现，包括Redis, EhCache, Hazelcast, Infinispan等

## 1. pom.xml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

## 2. BootStrapApplication打开cache

```
@EnableCaching
```

## 3. application.properties

```
spring.cache.type=redis
spring.redis.host=localhost
spring.redis.port=6379
```

# Spring boot + redis

## 4. controller增加注释

```
@Cacheable(value = "product", key = "#id", unless = "#result.price > 5000")
@RequestMapping(value = "/get/id/{id}", method = RequestMethod.GET)
public Product get(@PathVariable Integer id) {
    logger.info("get product by id = " + id);
    return productService.getProductById(id);
}
```

```
@CacheEvict(value = "product", allEntries = true)
@RequestMapping(value = "/delete/id/{id}", method = RequestMethod.DELETE)
public int delete(@PathVariable Integer id) {
    logger.info("delete product by id = " + id);
    return productDaoImpl.deleteById(id);
}
```

```
@CachePut(value = "product", key = "#id")
@RequestMapping(value = "/update/{id}/{name}/{price}", method = RequestMethod.PUT)
public EProduct update(@PathVariable Integer id, @PathVariable String name, @PathVariable Double price) {
    logger.info("update product by id =" + id);
    EProduct product = new EProduct(id, name, price);
    return productDao.save(product);
}
```

# 非注释实现

Spring data提供了redis的封装模版和操作，开发者可以自己实现对String, List, Hash, Set等数据类型的缓存

## 1. 定义实现缓存的操作组件（这里以string类型为例）

```
@Component
public class RedisComponent {
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    public void set(String key, String value) {
        ValueOperations<String, String> ops = this.stringRedisTemplate.opsForValue();
        if (!this.stringRedisTemplate.hasKey(key)) {
            ops.set(key, value);
        }
    }

    public String get(String key) {
        return this.stringRedisTemplate.opsForValue().get(key);
    }

    public void del(String key) {
        this.stringRedisTemplate.delete(key);
    }
}
```

# 非注释实现

## 2. Controller判断并实现缓存

```
@RequestMapping(value = "/get/{id}/price", method = RequestMethod.GET)
public String getPrice(@PathVariable Integer id) {
    String cachedPrice = redisCache.get(id.toString());
    if(null == cachedPrice) {
        EProduct product = productDao.findById(id).orElse(new EProduct("null", -1.0));
        String price = product.getPrice().toString();
        redisCache.set(id.toString(), price);
    }
    return cachedPrice;
}
```



# 从RDBMS 到NoSQL

3

# 分布式系统

**分布式系统 (distributed system)** 由多台计算机和通信的软件组件通过计算机网络连接（本地网络或广域网）组成

分布式系统是建立在网络之上的软件系统。正是因为软件的特性，所以分布式系统具有高度的内聚性和透明性。因此，网络和分布式系统之间的区别更多的在于高层软件（特别是操作系统），而不是硬件。

分布式系统可以应用在不同的平台上如：Pc、工作站、局域网和广域网上等。

**分布式计算的优点**

**可靠性（容错）**：分布式计算系统中的一个重要的优点是可靠性。一台服务器的系统崩溃并不影响到其余的服务器。

**可扩展性**：在分布式计算系统可以根据需要增加更多的机器。

**资源共享**：共享数据是必不可少的应用，如银行，预订系统。

**灵活性**：由于该系统是非常灵活的，它很容易安装，实施和调试新的服务。

**更快的速度**：分布式计算系统可以有多台计算机的计算能力，使得它比其他系统有更快的处理速度。

**开放系统**：由于它是开放的系统，本地或者远程都可以访问到该服务。

**更高的性能**：相较于集中式计算机网络集群可以提供更高的性能（及更好的性价比）。

**分布式计算的缺点**

**故障排除**：故障排除和诊断问题。

**软件**：更少的软件支持是分布式计算系统的主要缺点。

**网络**：网络基础设施的问题，包括：传输问题，高负载，信息丢失等。

**安全性**：开放系统的特性让分布式计算系统存在着数据的安全性和共享的风险等问题。

# 关系数据库ACID原则

事务有如下四个特性：

## 1、A (Atomicity) 原子性

原子性很容易理解，也就是说事务里的所有操作要么全部做完，要么都不做，事务成功的条件是事务里的所有操作都成功，只要有一个操作失败，整个事务就失败，需要回滚。

比如银行转账，从A账户转100元至B账户，分为两个步骤：1) 从A账户取100元；2) 存入100元至B账户。这两步要么一起完成，要么一起不完成，如果只完成第一步，第二步失败，钱会莫名其妙少了100元。

## 2、C (Consistency) 一致性

一致性也比较容易理解，也就是说数据库要一直处于一致的状态，事务的运行不会改变数据库原本的一致性约束。

例如现有完整性约束 $a+b=10$ ，如果一个事务改变了a，那么必须得改变b，使得事务结束后依然满足 $a+b=10$ ，否则事务失败。

## 3、I (Isolation) 独立性

所谓的独立性是指并发的事务之间不会互相影响，如果一个事务要访问的数据正在被另外一个事务修改，只要另外一个事务未提交，它所访问的数据就不受未提交事务的影响。

比如现在有个交易是从A账户转100元至B账户，在这个交易还未完成的情况下，如果此时B查询自己的账户，是看不到新增加的100元的。

## 4、D (Durability) 持久性

持久性是指一旦事务提交后，它所做的修改将会永久的保存在数据库上，即使出现宕机也不会丢失。

# CAP定理 (CAP theorem)

在计算机科学中, CAP定理 (CAP theorem) , 又被称作 布鲁尔定理 (Brewer's theorem) , 它指出对于一个分布式计算系统来说, 不可能同时满足以下三点:

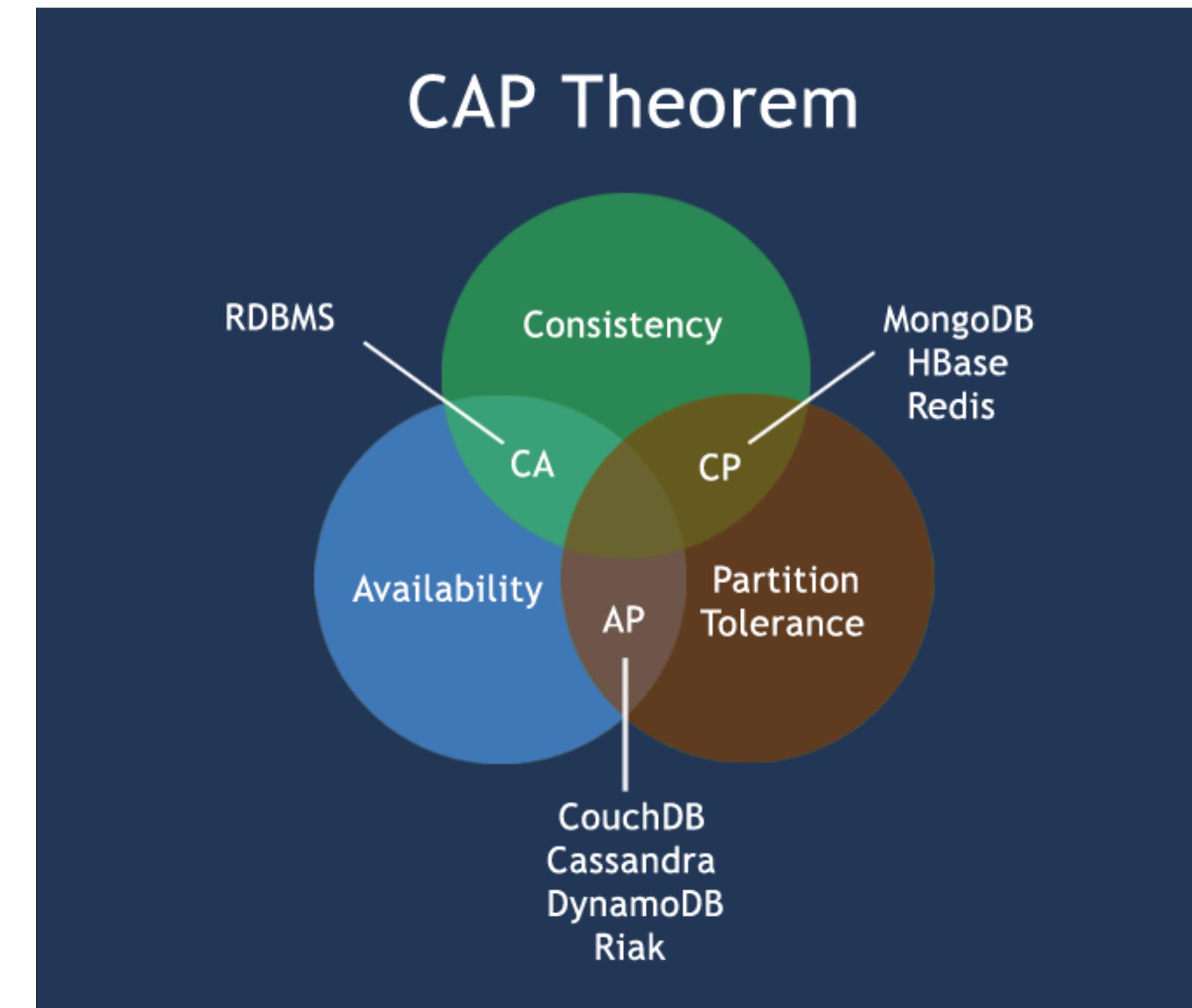
1. 一致性(Consistency) (所有节点在同一时间具有相同的数据)
2. 可用性(Availability) (保证每个请求不管成功或者失败都有响应)
3. 分隔容忍(Partition tolerance) (系统中任意信息的丢失或失败不会影响系统的继续运作)

CAP理论的核心是: 一个分布式系统不可能同时很好的满足一致性, 可用性和分区容错性这三个需求, 最多只能同时较好的满足两个。因此, 根据 CAP 原理将 NoSQL 数据库分成了满足 CA 原则、满足 CP 原则和满足 AP 原则三 大类:

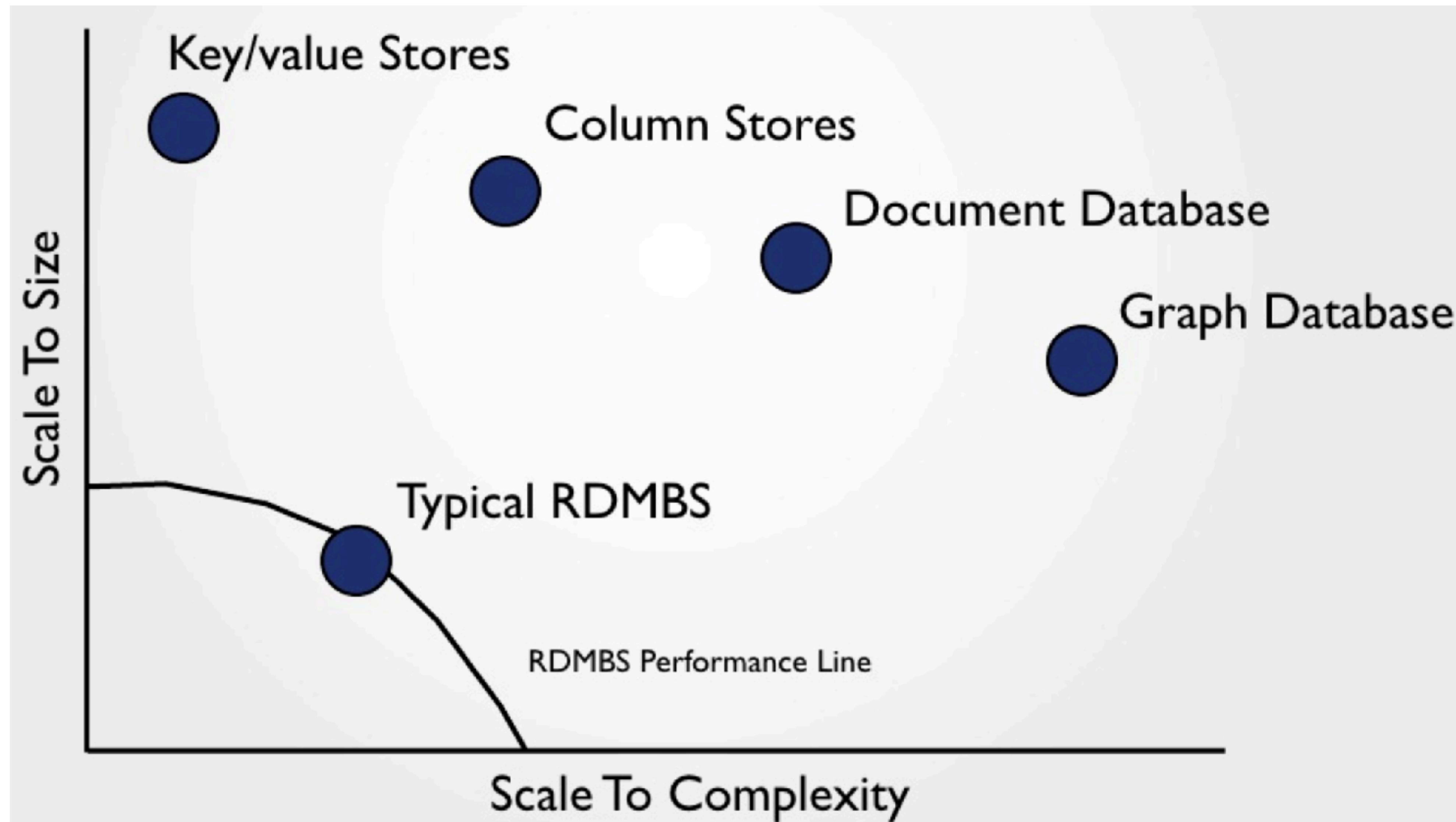
**CA - 单点集群, 满足一致性, 可用性的系统, 通常在可扩展性上不太强大。**

**CP - 满足一致性, 分区容忍性的系统, 通常性能不是特别高。**

**AP - 满足可用性, 分区容忍性的系统, 通常可能对一致性要求低一些。**



# 为什么是NoSQL

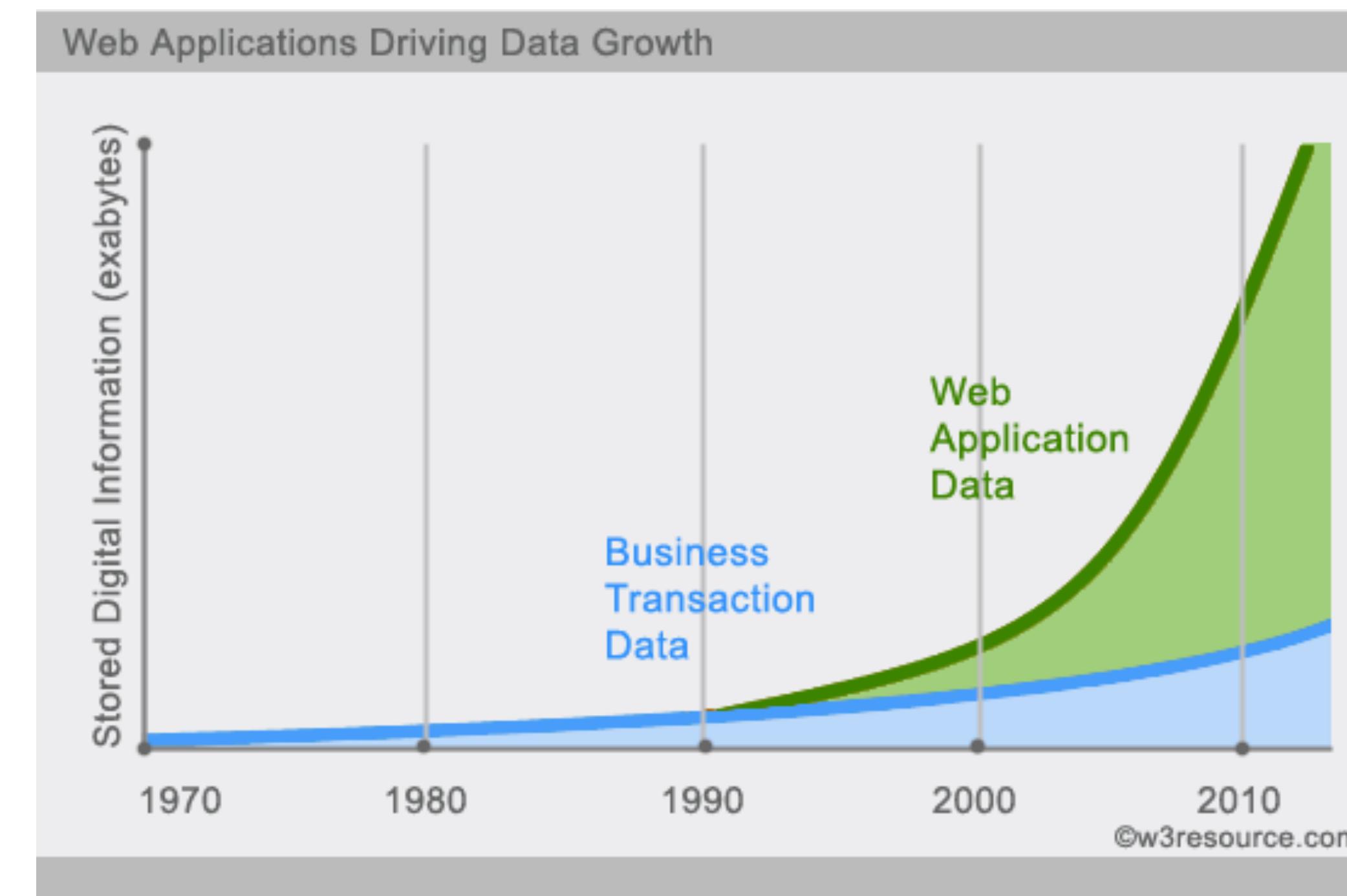


# 为什么是NoSQL

NoSQL，指的是非关系型的数据库。NoSQL有时也称作Not Only SQL的缩写，是对不同于传统的关系型数据库的数据管理系统的统称。

NoSQL用于超大规模数据的存储。（例如谷歌或Facebook每天为他们的用户收集万亿比特的数据）。这些类型的数据存储不需要固定的模式，无需多余操作就可以横向扩展。

今天我们可以通过第三方平台（如：Google,Facebook等）可以很容易的访问和抓取数据。用户的个人信息，社交网络，地理位置，用户生成的数据和用户操作日志已经成倍的增加。我们如果要对这些用户数据进行挖掘，那SQL数据库已经不适合这些应用了，NoSQL数据库的发展也却能很好的处理这些大的数据。



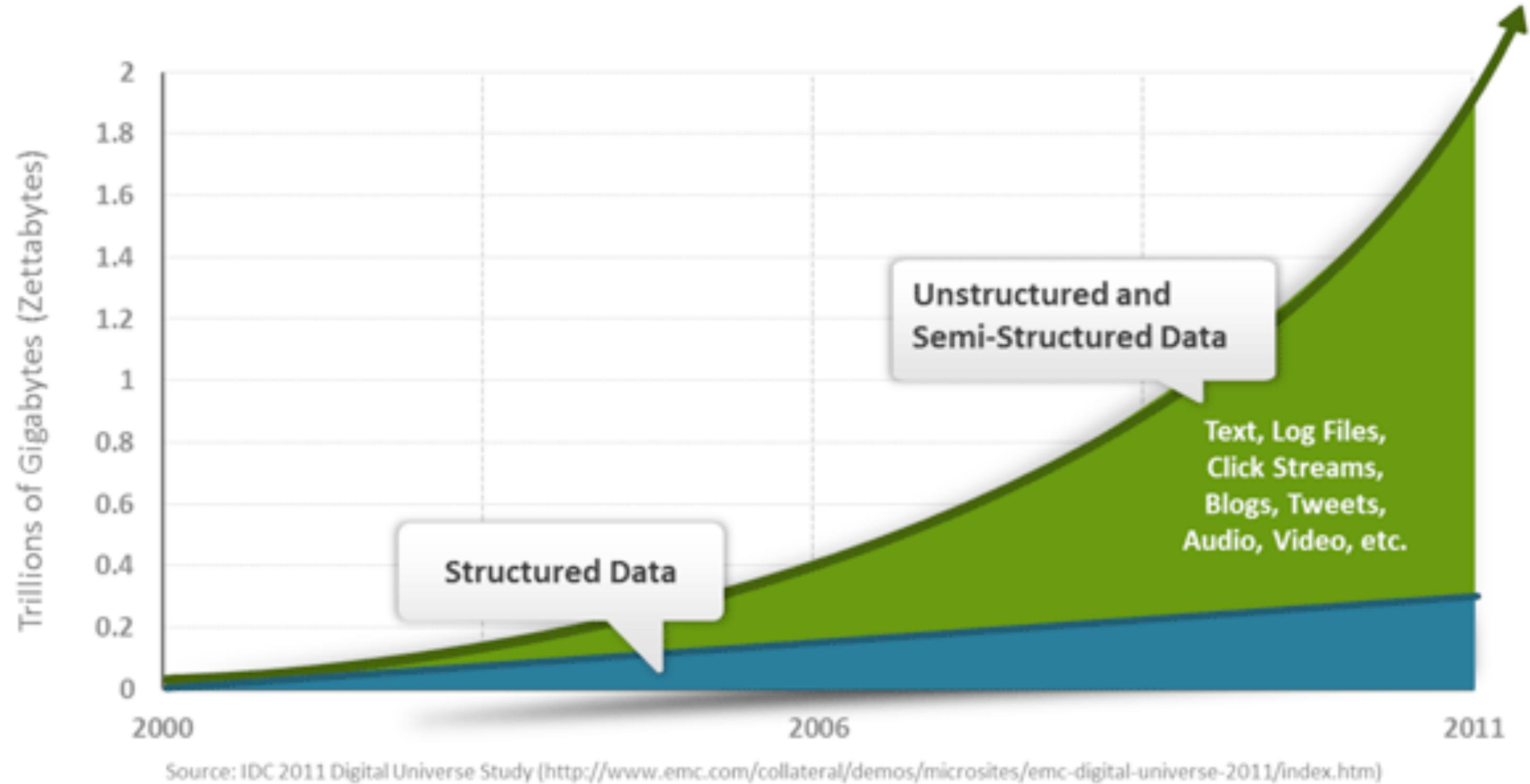
# RDBMS vs NoSQL

## RDBMS

- 高度组织化结构化数据
- 结构化查询语言 (SQL) (SQL)
- 数据和关系都存储在单独的表中。
- 数据操纵语言, 数据定义语言
- 严格的一致性
- 基础事务

## NoSQL

- 代表着不仅仅是SQL
- 没有声明性查询语言
- 没有预定义的模式
- 键 - 值对存储, 列存储, 文档存储, 图形数据库
- 最终一致性, 而非ACID属性
- 非结构化和不可预知的数据
- CAP定理
- 高性能, 高可用性和可伸缩性



# NoSQL 简史

NoSQL一词最早出现于1998年，是Carlo Strozzi开发的一个轻量、开源、不提供SQL功能的关系数据库。

2009年，Last.fm的Johan Oskarsson发起了一次关于分布式开源数据库的讨论，来自Rackspace的Eric Evans再次提出了**NoSQL**的概念，这时的NoSQL主要指非关系型、分布式、不提供ACID的数据设计模式。

2009年在亚特兰大举行的"no:sql(east)"讨论会是一个里程碑，其口号是"select fun, profit from real\_world where relational=false;". 因此，对**NoSQL**最普遍的解释是"非关联型的"，强调**Key-Value Stores**和文档数据库的优点，而不是单纯的反对RDBMS。



# NoSQL 优缺点

## 优点:

- 高可扩展性
- 分布式计算
- 低成本
- 架构的灵活性，半结构化数据
- 没有复杂的关系

## 缺点:

- 没有标准化
- 有限的查询功能（到目前为止）
- 最终一致是不直观的程序

## BASE

**BASE: Basically Available, Soft-state, Eventually Consistent,** 由 Eric Brewer 定义。

BASE是NoSQL数据库通常对可用性及一致性的弱要求原则:

**Basically Available** – 基本可用

**Soft-state** – 软状态/柔性事务。 "Soft state" 可以理解为"无连接"的, 而 "Hard state" 是"面向连接"的

**Eventual Consistency** – 最终一致性, 也是 ACID 的最终目的。

# NoSQL分类

类型	部分代表	特点
列存储	Hbase Cassandra Hypertable	顾名思义，是按列存储数据的。最大的特点是方便存储结构化和半结构化数据，方便做数据压缩，对针对某一列或者某几列的查询有非常大的IO优势。
文档存储	MongoDB CouchDB	文档存储一般用类似json的格式存储，存储的内容是文档型的。这样也就有机会对某些字段建立索引，实现关系数据库的某些功能。
key-value 存储	Tokyo Cabinet / Tyrant Berkeley DB MemcacheDB Redis	可以通过key快速查询到其value。一般来说，存储不管value的格式，照单全收。（Redis包含了其他功能）
图存储	Neo4J FlockDB	图形关系的最佳存储。使用传统关系数据库来解决的话性能低下，而且设计使用不方便。
对象存储	db4o Versant	通过类似面向对象语言的语法操作数据库，通过对对象的方式存取数据。
xml数据库	Berkeley DB XML BaseX	高效的存储XML数据，并支持XML的内部查询语法，比如XQuery,Xpath。



# MongoDB安装 与常用操作

4

# MongoDB安装

## 1. 官网安装

### 2. Homebrew

brew install mongodb

启动：

正常： mongod --config /usr/local/etc/mongod.conf

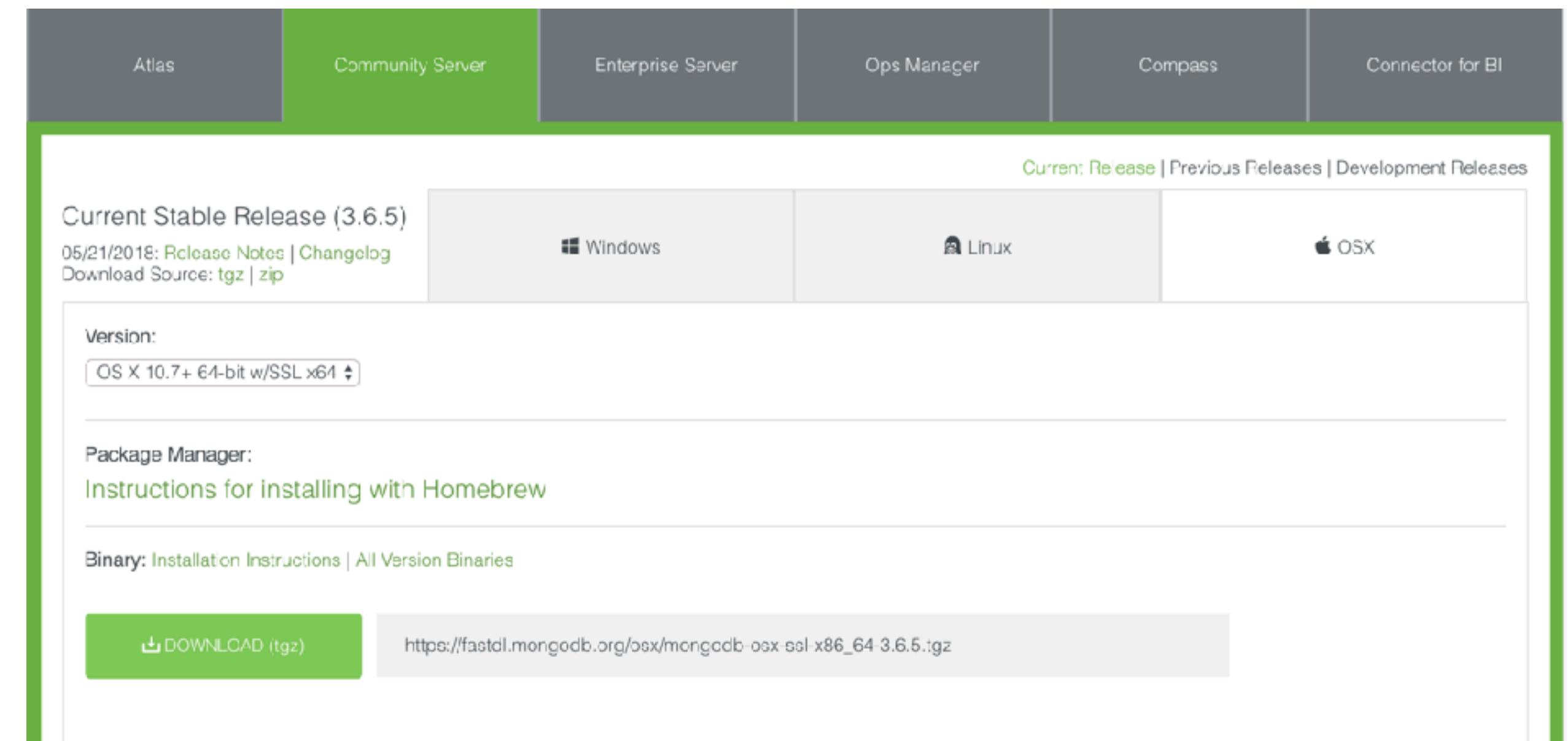
后台进程： brew services start mongodb

默认数据库路径为 /data/db，否则通过--dbpath 指定

启动客户端：

/usr/local/Cellar/mongodb/3.6.5/bin/mongo

默认端口27017，可以通过--host <host> --port <port\_number>连接远程mongodb服务器特定端口



```
←[0] <> /usr/local/Cellar/mongodb/3.6.5/bin/mongo
MongoDB shell version v3.6.5
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.0.7
WARNING: shell and server versions do not match
```

# SQL vs MongoDB

SQL术语/概念	MongoDB术语/概念	解释/说明
database	database	数据库
table	collection	数据库表/集合
row	document	数据记录行/文档
column	field	数据字段/域
index	index	索引
table joins		表连接,MongoDB不支持
primary key	primary key	主键,MongoDB自动将_id字段设置为主键

	<b>id</b>	<b>user_name</b>	<b>email</b>	<b>age</b>	<b>city</b>
	1	Mark Hanks	mark@abc.com	25	Los Angeles
	2	Richard Peter	richard@abc.com	31	Dallas



```
{
  "_id": ObjectId("5146bb52d8524270060001f3"),
  "age": 25,
  "city": "Los Angeles",
  "email": "mark@abc.com",
  "user_name": "Mark Hanks"
}
{
  "_id": ObjectId("5146bb52d8524270060001f2"),
  "age": 31,
  "city": "Dallas",
  "email": "richard@abc.com",
  "user_name": "Richard Peter"
}
```

# Shell 命令

help – 帮助

help admin – 管理员帮助

help connect – 连接DB帮助

help keys – 快捷键

help misc – 更多

help mr – map-reduce操作

show dbs – 显示数据库名称列表

show collections – 显示当前数据库合集

show users – 显示当前数据库用户

show profile – 显示最近system.profile信息

show logs – 显示所有可访问日志名称

show log [name] – 打印内存日志的最后一段

use <db\_name> – 设置当前数据库

it – 最后一行执行结果，用于迭代

exit – 退出

```
> show dbs
local 0.078GB
> show collections
> show users
> show profile
db.system.profile is empty
Use db.setProfilingLevel(2) will enable profiling
Use db.system.profile.find() to show raw profile entries
> show logs
global
> exit
bye
```

# Shell 命令

## 1. 命令行执行

*use local*

*show collections*

*exit*

## 2. 脚本执行

2.1 创建脚本，假设名为 mongo-script.js：

```
db = db.getSiblingDB('local')
```

```
print(db.getCollectionNames())
```

2.2 执行脚本

*/usr/local/Cellar/mongodb/3.6.5/bin/mongo*

*mongo-script.js*

```
[0] > /usr/local/Cellar/mongodb/3.6.5/bin/mongo
MongoDB shell version v3.6.5
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.0.7
WARNING: shell and server versions do not match
> use local
switched to db local
> show collections
startup_log
system.indexes
> exit
bye
```

```
[0] > /usr/local/Cellar/mongodb/3.6.5/bin/mongo mongo-script.js
MongoDB shell version v3.6.5
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.0.7
WARNING: shell and server versions do not match
startup_log,system.indexes
```

# 数据库

**Database = Collection:(document, document...) + Collection:(document, document...) + ...**

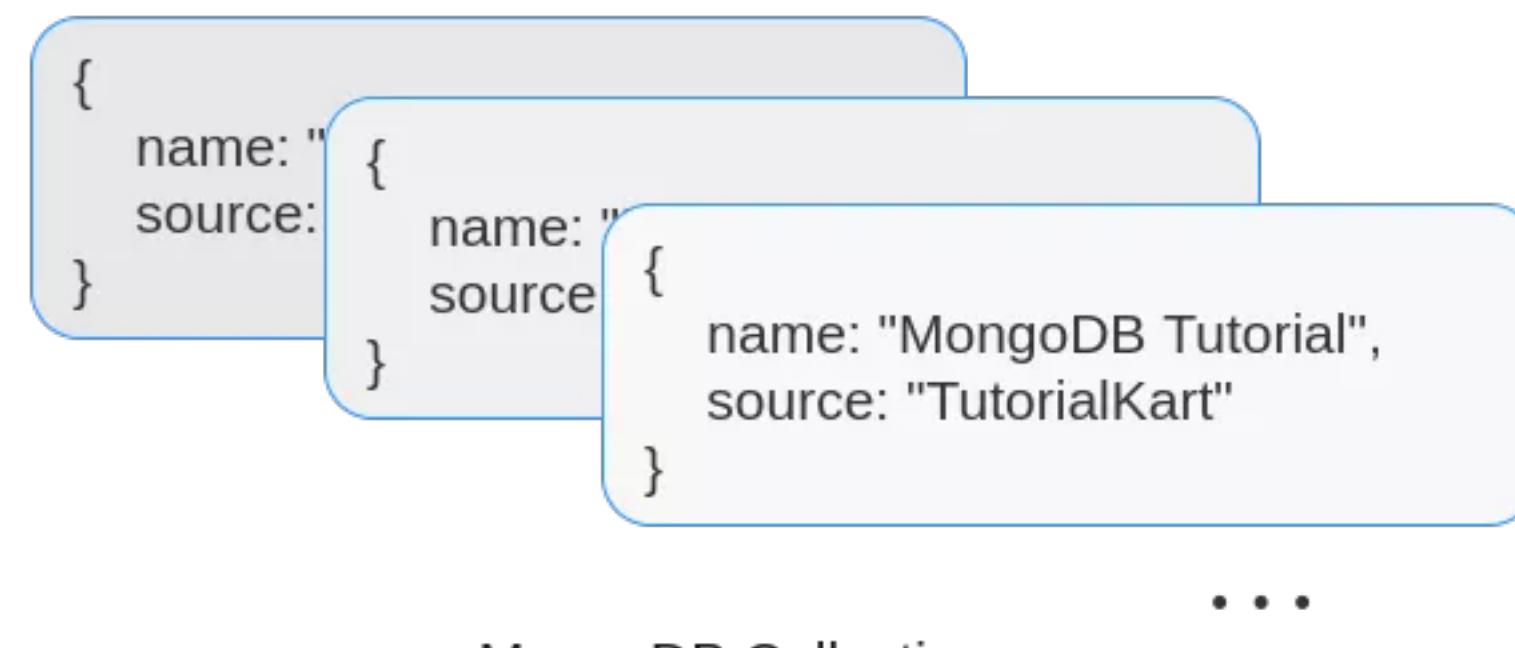
use命令可以用于选择（使用）指定数据库，也可以创建同名db，但它只有在第一次插入数据才会创建db

```
show dbs  
use test  
show dbs  
db.product.insertOne(  
  { name: "Nokia x61", price: 1699 }  
)  
show dbs  
db.dropDatabase()  
show dbs
```

```
> show dbs  
local 0.078GB  
> use test  
switched to db test  
> show dbs  
local 0.078GB  
> db.product.insertOne( { name: "Nokia x61", price: 1699 } )  
{  
  "acknowledged" : true,  
  "insertedId" : ObjectId("5b2401e7c4b41a4779c635e2")  
}  
> show dbs  
local 0.078GB  
test 0.078GB  
> db.dropDatabase()  
{ "dropped" : "test", "ok" : 1 }  
> show dbs  
local 0.078GB
```

# 集合

集合（Collection）用存储文档（document）的地方，类似关系数据库的表，是记录行的地方  
集合没有数据结构（schema），代码领域模型的改变，不需要改变数据库



## 隐式创建集合

*use test*

*show collections*

*db.product.insert(*

*{id:1, name:"mi8", price:"2999"}*

*)*

*show collections*

```
> use test
switched to db test
> db.product.insert({id:1, name:"mi8", price:"2999"})
WriteResult({ "nInserted" : 1 })
> show collections
product
system.indexes
```

# 集合

结合选项：capped是否有大小限制，size表达集合最大尺寸（单位字节），max表示最大集合数量  
所有选项都非必须的，当插入数据超出设置的大小或数量，旧数据会被淘汰出去

## 显式创建集合

```
use shoppingList
show collections
db.createCollection("wishList", {
  capped : true, size : 280000, max : 1000
})
show collections
db.wishList.drop()
show collections
db.nonExist.drop()
```

```
> use shoppingList
switched to db shoppingList
> show collections
> db.createCollection("wishList", { capped : true, size : 280000, max : 1000 })
{ "ok" : 1 }
> show collections
system.indexes
wishList
> db.wishList.drop()
true
> show collections
system.indexes
> db.nonExist.drop()
false
```

# 文档

**文档 (document)** 是包含零个或多个**key-value**对的实体，类似关系数据库的包含多个字段的一行数据  
文档遵守**BSON**规范，它是一个类似JSON的二进制编码规范，方便进行数据转换。文档可以包含多个字段，每个字段的value支持多种数据类型。

```
{  
    field1:value1;  
    field2:value2;  
    .  
    .  
    fieldN:valueN;  
}
```

文档内可以嵌套文档，文档组成集合

文档包含如何操作：插入 (Insert) ，查询 (Query) ，更新 (Query) 和删除 (Delete)

# 数据类型

数据类型	描述
String	字符串。存储数据常用的数据类型。在 MongoDB 中，UTF-8 编码的字符串才是合法的。
Integer	整型数值。用于存储数值。根据你所采用的服务器，可分为 32 位或 64 位。
Boolean	布尔值。用于存储布尔值（真/假）。
Double	双精度浮点值。用于存储浮点值。
Min/Max keys	将一个值与 BSON（二进制的 JSON）元素的最低值和最高值相对比。
Array	用于将数组或列表或多个值存储为一个键。
Timestamp	时间戳。记录文档修改或添加的具体时间。
Object	用于内嵌文档。
Null	用于创建空值。
Symbol	符号。该数据类型基本上等同于字符串类型，但不同的是，它一般用于采用特殊符号类型的语言。
Date	日期时间。用 UNIX 时间格式来存储当前日期或时间。你可以指定自己的日期时间：创建 Date 对象，传入年月日信息。
Object ID	对象 ID。用于创建文档的 ID。
Binary Data	二进制数据。用于存储二进制数据。
Code	代码类型。用于在文档中存储 JavaScript 代码。
Regular expression	正则表达式类型。用于存储正则表达式。

# 文档操作

## 插入单条数据

```
show collections  
db.product.insertOne({id: 2, name: "nokia X61", price: 1699})  
db.product.insertOne({id: 3, name: "iPhoneX", price: 9699})
```

## 插入多条数据

```
db.product.insertMany(  
[  
  {id: 4, name: "R15", price: 2999},  
  {id: 5, name: "X21", price: 3299},  
  {id: 6, name: "R1", price: 8848},  
])  
)
```

acknowledged为true表示插入成功

ObjectId表示文档ID

```
> show collections  
product  
system.indexes  
> db.product.insertOne({id: 2, name: "nokia X61", price: 1699})  
{  
  "acknowledged" : true,  
  "insertedId" : ObjectId("5b247fb4c4b41a4779c635e4")  
}  
> db.product.insertOne({id: 3, name: "iPhoneX", price: 9699})  
{  
  "acknowledged" : true,  
  "insertedId" : ObjectId("5b247fcbc4b41a4779c635e5")  
}
```

```
> db.product.insertMany(  
... [  
...   {id: 4, name: "R15", price: 2999},  
...   {id: 5, name: "X21", price: 3299},  
...   {id: 6, name: "R1", price: 8848},  
... ]  
... )  
{  
  "acknowledged" : true,  
  "insertedIds" : [  
    ObjectId("5b24806dc4b41a4779c635e6"),  
    ObjectId("5b24806dc4b41a4779c635e7"),  
    ObjectId("5b24806dc4b41a4779c635e8")  
]
```

# 文档操作

## 通过查询条件过滤数据

```
db.product.find({})
```

```
criteria = {name:"R1"}
```

```
db.product.find(criteria)
```

```
criteria = {name: "r1"}
```

```
db.product.find(criteria)
```

## 限制过滤字段

**1:包含； 0:不包含**

```
projection_doc={
```

```
    name: 1, price: 1, _id: 0
```

```
}
```

```
db.product.find({}, projection_doc)
```

```
> db.product.find({})
{ "_id" : ObjectId("5b2469fec4b41a4779c635e3"), "id" : 1, "name" : "mi8", "price" : 2999 }
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699 }
{ "_id" : ObjectId("5b247fcbe4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699 }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999 }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299 }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "R1", "price" : 8848 }
> criteria = {name:"R1"}
{ "name" : "R1" }
> db.product.find(criteria)
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "R1", "price" : 8848 }
> criteria = {name: "r1"}
{ "name" : "r1" }
> db.product.find(criteria)
> projection_doc={name: 1, price: 1, _id: 0}
{ "name" : 1, "price" : 1, "_id" : 0 }
> db.product.find({}, projection_doc)
{ "name" : "mi8", "price" : 2999 }
{ "name" : "nokia X61", "price" : 1699 }
{ "name" : "iPhoneX", "price" : 9699 }
{ "name" : "R15", "price" : 2999 }
{ "name" : "X21", "price" : 3299 }
{ "name" : "R1", "price" : 8848 }
```

# 文档操作

文档更新语法为：*db.collection\_name.update(criteria, update, options)*

criteria为过滤条件，即要更新的数据条件；update为新的数据，默认需要包含所有字段，如果字段不完全已有数据会被覆盖，如果只想更新某些字段，是否\$set指定；options是可选的更新选项

选项	默认值/作用	描述
upsert	默认为false，表示如果没找到数据不插入新数据	如果设置为true，它会插入不存在的数据
multi	默认值为false，表示只更新一条数据	如果设置为true，它更新所有满足条件的数据
collation	文档类型，默认使用二进制比较方式比较字符串	指定特定语言规则，参考 <a href="#">Collation</a>
WriteConcern	文档类型，参考 <a href="#">Write Concern</a> .	描述写关注的确认级别

# 文档操作

## 更新单条数据

```
criteria = {name:"R1"}  
db.product.find(criteria)  
update = {name: "Smartisan R1"}  
db.product.update(criteria, update)  
db.product.find(criteria)
```

```
criteria = {name: "Smartisan R1"}  
db.product.find(criteria)  
update = {id: 6, name: "Smartisan R1", "price": 8848}  
db.product.update(criteria, update)  
db.product.find(criteria)
```

## 更新多条指定数据

```
db.product.find({}).count()  
update={$set:{category: "mobile"} }  
options={multi:true}  
db.product.update({}, update, options)  
db.product.find({})
```

```
> criteria = {name:"R1"}  
{ "name" : "R1" }  
> db.product.find(criteria)  
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "R1", "price" : 8848 }  
> update = {name: "Smartisan R1"}  
{ "name" : "Smartisan R1" }  
> db.product.update(criteria, update)  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
> db.product.find(criteria)  
> criteria = {name: "Smartisan R1"}  
{ "name" : "Smartisan R1" }  
> db.product.find(criteria)  
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "name" : "Smartisan R1" }  
> update = {id: 6, name: "Smartisan R1", "price": 8848}  
{ "id" : 6, "name" : "Smartisan R1", "price" : 8848 }  
> db.product.update(criteria, update)  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
> db.product.find(criteria)  
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848 }
```

```
> db.product.find({}).count()  
6  
> update={$set:{category: "mobile"} }  
{ "$set" : { "category" : "mobile" } }  
> options={multi:true}  
{ "multi" : true }  
> db.product.update({}, update, options)  
WriteResult({ "nMatched" : 6, "nUpserted" : 0, "nModified" : 6 })  
> db.product.find({})  
{ "_id" : ObjectId("5b2469fec4b41a4779c635e3"), "id" : 1, "name" : "mi8", "price" : 2999, "category" : "mobile" }  
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699, "category" : "mobile" }  
{ "_id" : ObjectId("5b247fcfc4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699, "category" : "mobile" }  
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }  
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299, "category" : "mobile" }  
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848, "category" : "mobile" }
```

# 文档操作

## 删除文档

```
criteria = { category: "mobile" }
db.product.find(criteria)
db.product.remove(criteria, true)
db.product.find(criteria)
```

remove方法第二个参数just\_one， 表示是否最多只删除一条

## 删除集合内所有文档

```
db.product.remove({})
```

## 限制查询结果条数

```
db.product.find({})
db.product.find{}.limit()
db.product.find{}.limit(3)
db.product.find{}.count()
db.product.find{}.limit(3).count()
db.product.find{}.limit().count()
```

```
> use test
switched to db test
> db.product.find({})
{ "_id" : ObjectId("5b2469fec4b41a4779c635e3"), "id" : 1, "name" : "mi8", "price" : 2999, "category" : "mobile" }
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699, "category" : "mobile" }
{ "_id" : ObjectId("5b247fcbe4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848, "category" : "mobile" }
> criteria = { category: "mobile" }
{ "category" : "mobile" }
> db.product.remove(criteria, true)
WriteResult({ "nRemoved" : 1 })
> db.product.find(criteria)
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699, "category" : "mobile" }
{ "_id" : ObjectId("5b247fcbe4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848, "category" : "mobile" }
```

```
> db.product.find({})
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699, "category" : "mobile" }
{ "_id" : ObjectId("5b247fcbe4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848, "category" : "mobile" }
> db.product.find{}.limit(3)
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699, "category" : "mobile" }
{ "_id" : ObjectId("5b247fcbe4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }
> db.product.find{}.count()
5
> db.product.find{}.limit(3).count()
5
> db.product.find{}.limit().count()
5
> db.product.find{}.limit()
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699, "category" : "mobile" }
{ "_id" : ObjectId("5b247fcbe4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848, "category" : "mobile" }
```

# 文档操作

## 跳过文档

```
db.product.find({})
```

```
db.product.find({}).skip(3)
```

```
db.product.find({}).skip(2).limit(1)
```

```
db.product.find({}).skip()
```

## 查询结果排序

```
db.product.find().sort({})
```

```
db.product.find().sort({price: -1})
```

```
> db.product.find({})
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699, "category" : "mobile" }
{ "_id" : ObjectId("5b247fcbe4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848, "category" : "mobile" }
> db.product.find({}).skip(3)
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848, "category" : "mobile" }
> db.product.find({}).skip(2).limit(1)
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }
> db.product.find({}).skip()
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699, "category" : "mobile" }
{ "_id" : ObjectId("5b247fcbe4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848, "category" : "mobile" }
```

```
> db.product.find().sort({})
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699, "category" : "mobile" }
{ "_id" : ObjectId("5b247fcbe4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848, "category" : "mobile" }
> db.product.find().sort({price: -1})
{ "_id" : ObjectId("5b247fcbe4b41a4779c635e5"), "id" : 3, "name" : "iPhoneX", "price" : 9699, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e8"), "id" : 6, "name" : "Smartisan R1", "price" : 8848, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e7"), "id" : 5, "name" : "X21", "price" : 3299, "category" : "mobile" }
{ "_id" : ObjectId("5b24806dc4b41a4779c635e6"), "id" : 4, "name" : "R15", "price" : 2999, "category" : "mobile" }
{ "_id" : ObjectId("5b247fb4c4b41a4779c635e4"), "id" : 2, "name" : "nokia X61", "price" : 1699, "category" : "mobile" }
```

# MongoDB集群(单机)

## 1. 建立两个数据库存放路径

```
mkdir -p /usr/local/var/mongodb
```

```
mkdir -p /usr/local/var/mongodb1
```

## 2. 建立两个配置

```
touch /usr/local/etc/mongod.conf
```

```
touch /usr/local/etc/mongod1.conf
```

## 3. 修改配置内容，为将要启动的两个实例配置dbpath和port

```
nano /usr/local/etc/mongod1.conf
```

```
systemLog:
```

```
  destination: file
```

```
  path: /usr/local/var/log/mongodb/mongo.log
```

```
  logAppend: true
```

```
storage:
```

```
  dbPath: /usr/local/var/mongodb1
```

```
net:
```

```
  bindIp: 127.0.0.1
```

```
  port: 27018
```

```
nano /usr/local/etc/mongod.conf
```

```
systemLog:
```

```
  destination: file
```

```
  path: /usr/local/var/log/mongodb/mongo.log
```

```
  logAppend: true
```

```
storage:
```

```
  dbPath: /usr/local/var/mongodb
```

```
net:
```

```
  bindIp: 127.0.0.1
```

```
  port: 27017
```

# MongoDB集群(单机)

## 4. 启动两个mongodb实例， 集群名为rs0

```
mongod --config /usr/local/etc/mongod.conf --replSet rs0
```

```
mongod --config /usr/local/etc/mongod1.conf --replSet rs0
```

## 5. 启动mongo shell

```
/usr/local/Cellar/mongodb/3.6.5/bin/mongo --port 27017
```

初始化集群， shell提示会变成rs0:PRIMARY>

```
rs.initiate()
```

查看集群状态， 目前只有一个成员

```
rs.status()
```

添加备份

```
rs.add("localhost:27018")
```

查看集群状态

```
rs.status()
```

## 6. 改变数据， 查看集群变化

在27017实例上， 执行命令增加一条数据

```
use test
```

```
db.product.insertOne({name: "replic", price: 10000, category: "database"})
```

```
db.product.find({name: "replic"})
```

```
rs0:PRIMARY> rs.add("127.0.0.1:27018")
{ "ok" : 1 }
2018-06-17T18:33:54.054+0800 I NETWORK [thread1] trying reconnect to 127.0.0.1:27017 (127.0.0.1) failed
2018-06-17T18:33:54.055+0800 I NETWORK [thread1] reconnect 127.0.0.1:27017 (127.0.0.1) ok
rs0:SECONDARY> rs.status()
{
  "set" : "rs0",
  "date" : ISODate("2018-06-17T10:36:54.529Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "127.0.0.1:27017",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 1576,
      "optime" : Timestamp(1529231634, 1),
      "optimeDate" : ISODate("2018-06-17T10:33:54Z"),
      "electionTime" : Timestamp(1529231636, 1),
      "electionDate" : ISODate("2018-06-17T10:33:56Z"),
      "configVersion" : 47091,
      "self" : true
    },
    {
      "_id" : 1,
      "name" : "127.0.0.1:27018",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 178,
      "optime" : Timestamp(1529231634, 1),
      "optimeDate" : ISODate("2018-06-17T10:33:54Z"),
      "lastHeartbeat" : ISODate("2018-06-17T10:36:54.347Z"),
      "lastHeartbeatRecv" : ISODate("2018-06-17T10:36:54.347Z"),
      "pingMs" : 0,
      "configVersion" : 47091
    }
  ],
  "ok" : 1
}
rs0:PRIMARY> █
```

# MongoDB集群(单机)

进入27018实例，它是slave机，shell提示为rs0:SECONDARY>，然后查看数据是否同步

```
/usr/local/Cellar/mongodb/3.6.5/bin/mongo --port 27018
```

```
use test
```

```
db.product.find({name: "replic"})
```

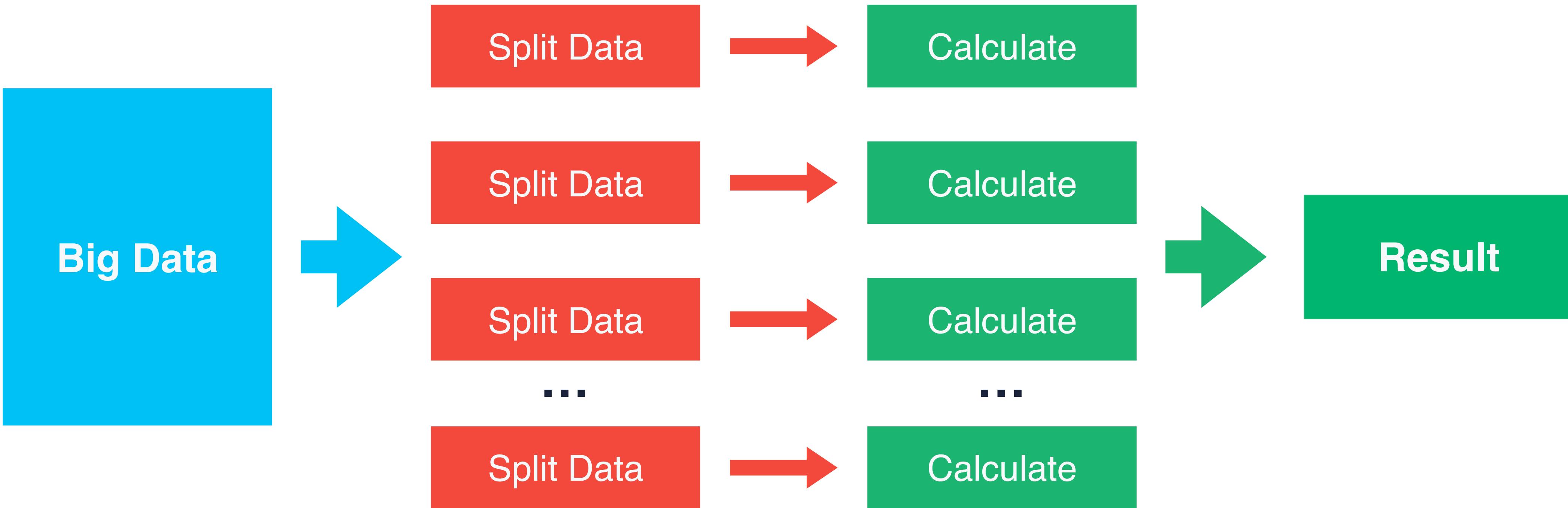
```
rs.slaveOk()
```

```
db.product.find({name: "replic"})
```

```
rs0:PRIMARY> db.product.insertOne({name: "replic", price: 10000, category: "database"})
{
  "acknowledged" : true,
  "insertedId" : ObjectId("5b263b6f55e9d0e9016b0442")
}
rs0:PRIMARY> db.product.find({name: "replic"})
{ "_id" : ObjectId("5b263b6f55e9d0e9016b0442"), "name" : "replic", "price" : 10000, "category" : "database" }
```

```
rs0:SECONDARY> db.product.find({name: "replic"})
Error: error: { "$err" : "not master and slaveOk=false", "code" : 13435 }
rs0:SECONDARY> rs.slaveOk()
rs0:SECONDARY> db.product.find({name: "replic"})
{ "_id" : ObjectId("5b263b6f55e9d0e9016b0442"), "name" : "replic", "price" : 10000, "category" : "database" }
```

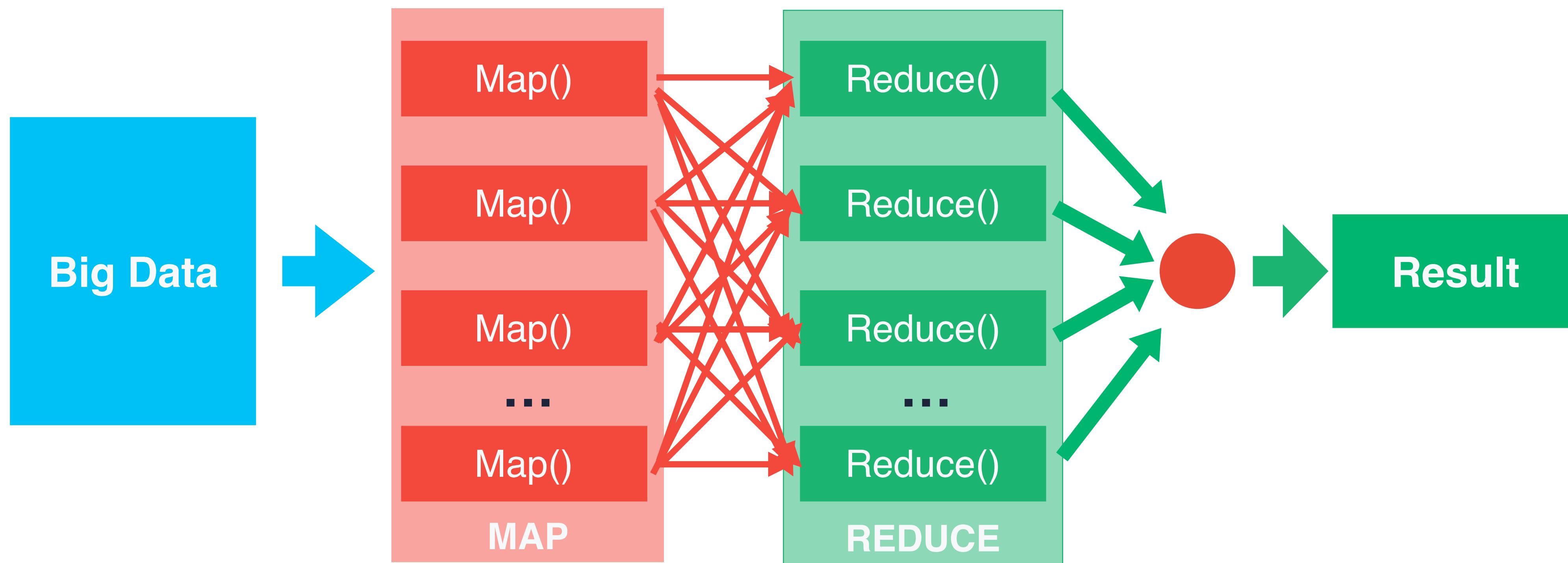
# 传统大数据分析



1. 关键路径问题：一台机器的延迟会导致其他已经完成机器的等待
2. 可靠性问题：机器失败管理
3. 大数据拆分问题
4. 容错性：一个机器失败导致无法得到最终结果
5. 聚合结果

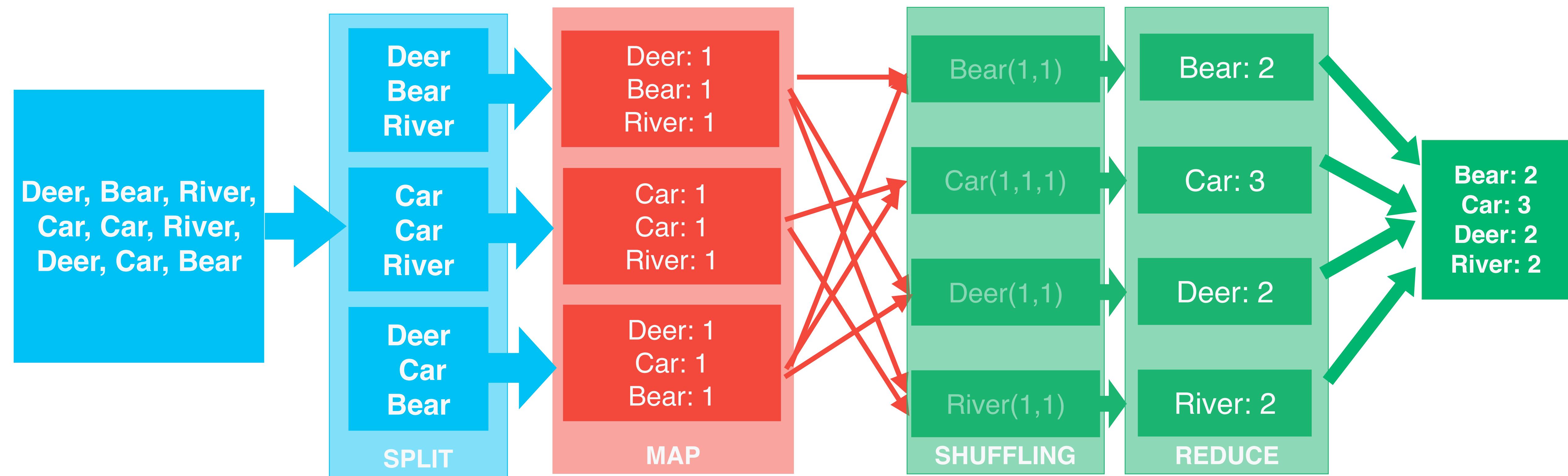
# MapReduce

- 2004年谷歌在一片论文中发布MapReduce技术，它是一个编程模型，用于在大数据集中执行并行和分布式的数据处理
- MapReduce = Map() + Reduce()
- Map：读取并处理分块数据，生成key-value结构的中间数据
- Reduce：读取多个map结果，归纳为更小的结果集
- Map()的结果是Reduce()的输入



# MapReduce

## 案例：字数统计



# MongoDB MapReduce

首先按照category为key, name+price组合作为value映射为中间值, 然后根据category聚合该类别下所有产品

**\*\*更常见的方式是写成Mongo脚本执行MapReduce操作**

```
var map = function() {emit(this.category, 1);};  
var reduce = function(name,products) {return Array.sum(products);};  
db.product.mapReduce(map, reduce, {out: "category"});  
db.category.find()
```

```
> var map = function() {emit(this.category, 1);};  
> var reduce = function(name,products) {return Array.sum(products);};  
> db.product.mapReduce(map, reduce, {out: "category"});  
{  
    "result" : "category",  
    "timeMillis" : 2,  
    "counts" : {  
        "input" : 11,  
        "emit" : 11,  
        "reduce" : 1,  
        "output" : 4  
    },  
    "ok" : 1  
}  
> db.category.find()  
{ "_id" : "database", "value" : 1 }  
{ "_id" : "fruit", "value" : 1 }  
{ "_id" : "laptop", "value" : 1 }  
{ "_id" : "mobile", "value" : 8 }  
> █
```



# Spring Boot + MongoDB 实践

5

# MongoDB CRUD

## 1. 引入spring mongo支持

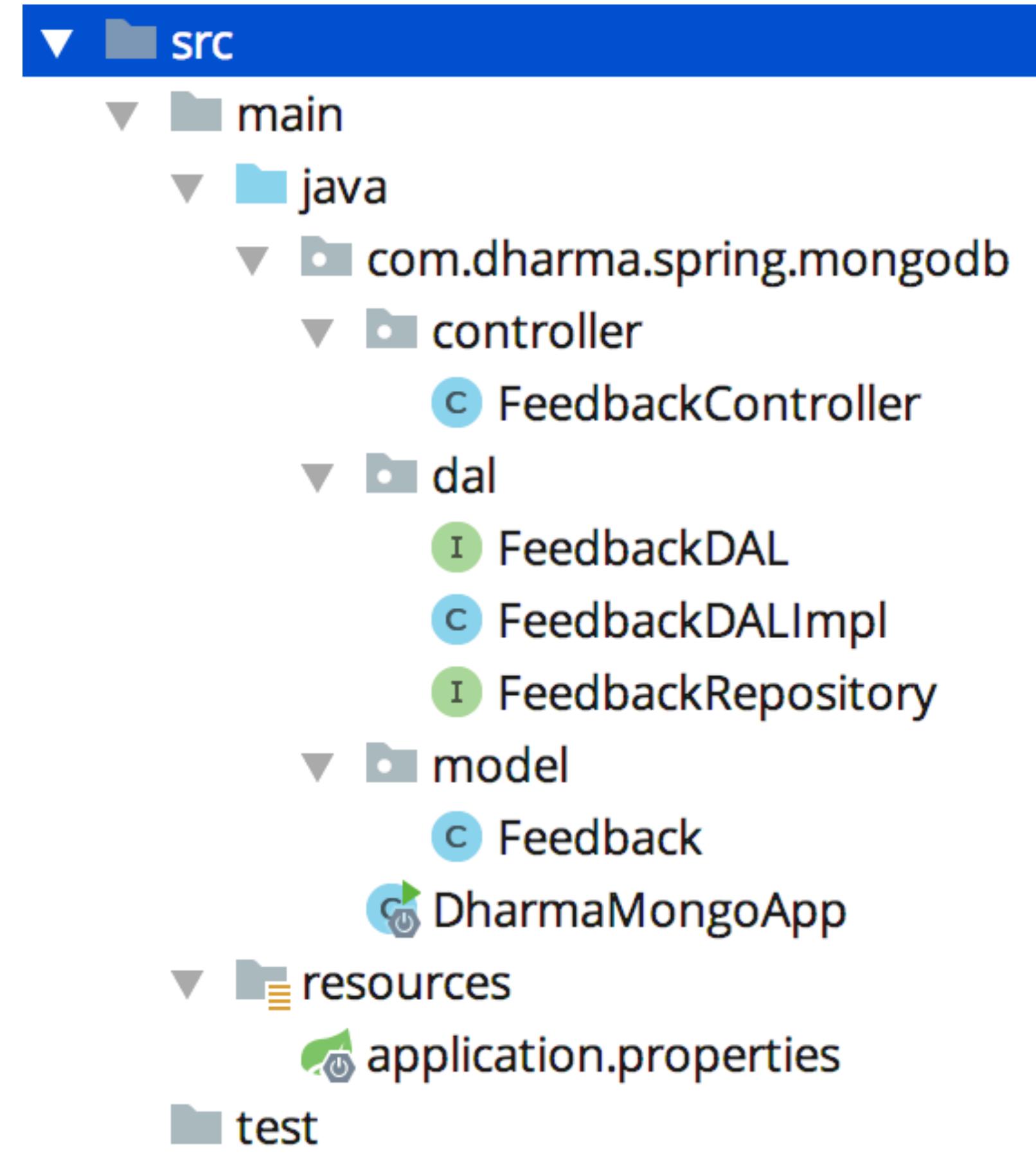
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

## 2. application.properties增加数据库连接配置

```
spring.data.mongodb.database=dharma
spring.data.mongodb.port=27017
spring.data.mongodb.host=localhost
```

## 3. 定义Model领域对象

```
@Document
public class Feedback {
  @Id
  private String id;
  .....
```



# MongoDB CRUD

## 4. 实现CRUD操作

### 4.1 Repository

@Repository

```
public interface FeedbackRepository extends MongoRepository<Feedback, String> {  
}
```

### 4.2 MongoTemplate

定义模版，推荐使用模版方式连接mongodb

@Repository

```
public class FeedbackDALImpl implements FeedbackDAL {
```

@Autowired

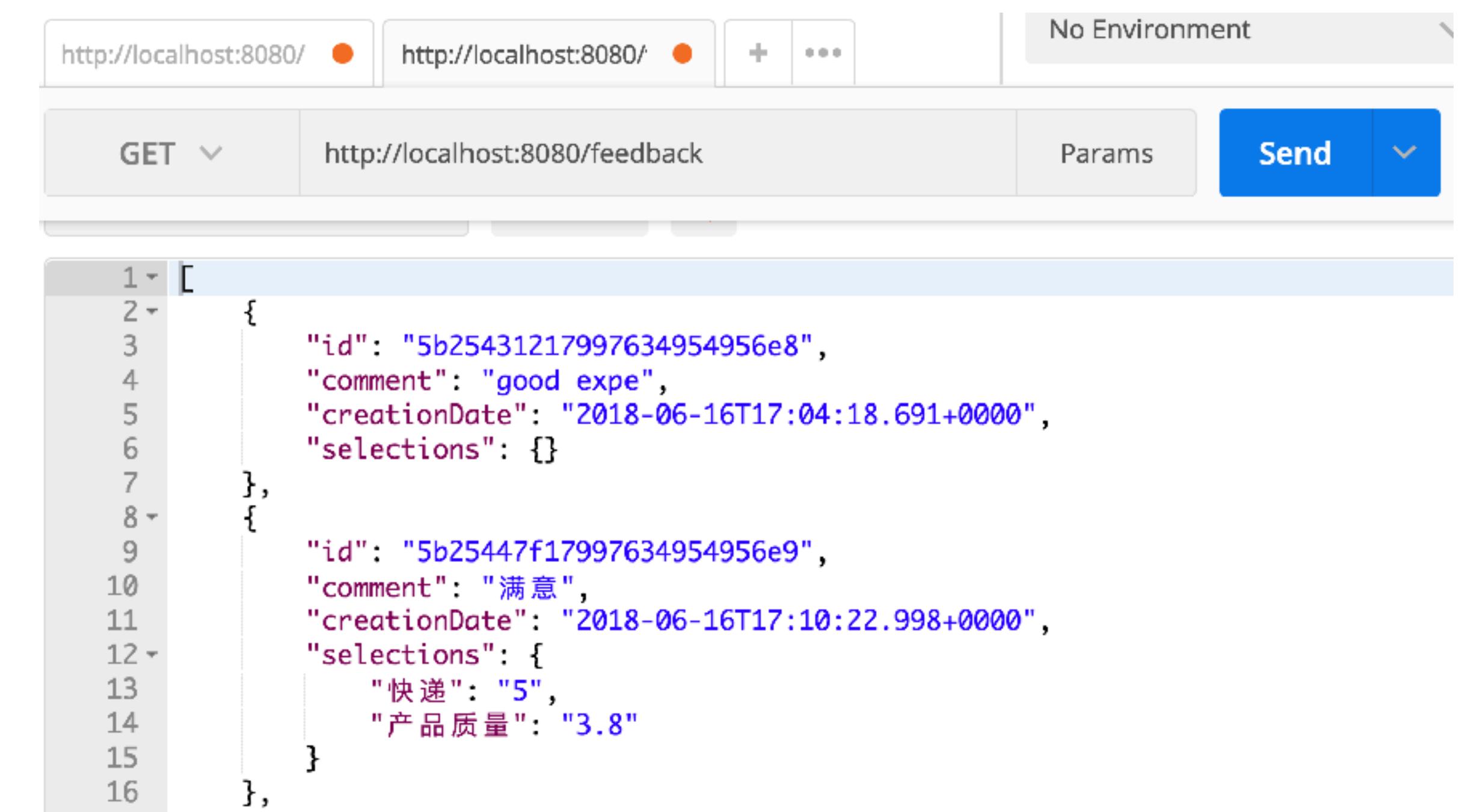
```
private MongoTemplate mongoTemplate;
```

.....

定义查询操作

@Override

```
public List<Feedback> getAllFeedbacks() {  
    return mongoTemplate.findAll(Feedback.class);  
}
```



```
@Override  
public Feedback getFeedbackById(String feedbackId) {  
    Query query = new Query();  
    query.addCriteria(Criteria.where("id").is(feedbackId));  
    return mongoTemplate.findOne(query, Feedback.class);  
}
```

## 定义更新操作

```
@Override  
public Feedback addNewFeedback(Feedback feedback) {  
    mongoTemplate.save(feedback);  
    return feedback;  
}
```

```
@Override  
public String addFeedbackSelections(String feedbackId, String key, String value) {  
    Query query = new Query();  
    query.addCriteria(Criteria.where("id").is(feedbackId));  
    Feedback feedback = mongoTemplate.findOne(query, Feedback.class);  
    if (feedback != null) {  
        feedback.getSelections().put(key, value);  
        mongoTemplate.save(feedback);  
        return "Key added.";  
    } else {  
        return "Feedback not found."  
    }  
}
```

# MongoDB CRUD

## 5. 实现Controller

通过构造函数注入**Repository**实现和**MongoTemplate**实现

```
private final FeedbackRepository feedbackRepository;  
private final FeedbackDAL feedbackDAL;  
public FeedbackController(FeedbackRepository feedbackRepository, FeedbackDAL feedbackDAL) {  
    this.feedbackRepository = feedbackRepository;  
    this.feedbackDAL = feedbackDAL;  
}
```

实现**POST**请求映射，下面分别用**Repository**和**MongoTemplate**分别实现

```
@RequestMapping(value = "/create", method = RequestMethod.POST)  
public Feedback createNewFeedback(@RequestBody Feedback feedback) {  
    return feedbackRepository.save(feedback);  
}  
  
@RequestMapping(value = "/add", method = RequestMethod.POST)  
public Feedback addNewFeedback(@RequestBody Feedback feedback) {  
    return feedbackDAL.addNewFeedback(feedback);  
}
```

# MongoDB CRUD

实现GET请求映射，下面分别用Repository和MongoTemplate分别实现

```
@RequestMapping(value = "/{feedbackId}", method = RequestMethod.GET)
public Feedback getFeedback(@PathVariable String feedbackId) {
    return feedbackRepository.findById(feedbackId).orElse(null);
}
```

```
@RequestMapping(value = "/get/{feedbackId}", method = RequestMethod.GET)
public Feedback getFeedbackById(@PathVariable String feedbackId) {
    return feedbackDAL.getFeedbackById(feedbackId);
}
```

# MongoDB MapReduce

MongoTemplate提供了Mapreduce方法，可以实现和mapreduce命令一样的操作  
map和reduce方法本质是javascript函数， map用于生成key-value对， reduce用于归纳key-values

```
function() {  
    ...  
    emit(key, value);  
}  
  
function(key, values) {  
    ...  
    return result;  
}
```

Spring中有两种实现方式：

## 1. 内联

```
final String map = "function() {emit(this.star, 1);}";  
final String reduce = "function(name, count) {return Array.sum(count);}";  
MapReduceResults<KeyValuePair> results = mongoTemplate.mapReduce(COLLECTION_NAME, map, reduce,  
KeyValuePair.class);
```

## 2. 外部引用

```
MapReduceOptions options = MapReduceOptions.options();  
options.outputCollection("countStar");  
options.outputTypeReduce();  
MapReduceResults<KeyValuePair> results = mongoTemplate.mapReduce(COLLECTION_NAME, "classpath:map.js",  
"classpath:reduce.js", options, KeyValuePair.class);  
}
```



# Thanks!

Any questions?