

鲍辰的应用体系架构 笔记

目录

2021/9/13.....	5
应用程序的演化.....	7
有状态(stateful)和无状态(stateless)服务的区别.....	10
Spring Boot 中的作用域.....	10
Singonton 的代码实现.....	11
Prototype 的代码实现.....	12
Session 的代码实现.....	14
2021/9/16.....	15
消息机制.....	15
消息机制的特性.....	16
辩证地看待消息机制.....	17
消息的数据结构.....	18
消息机制的编程模型.....	19
2021/9/18.....	28
Kafka.....	29
Websocket.....	30
2021/9/23.....	36
事务管理.....	36
乐观锁和悲观锁.....	47
2021/9/27.....	49
Java 多线程.....	49
2021/9/30.....	57
可重入锁.....	57
Java Executors 框架.....	59
内存中的缓存.....	61
Memcached.....	62
Redis.....	65
2021/10/9.....	68
2021/10/11.....	74
SOAP 协议.....	74
Restful Service.....	81
2021/10/14.....	84
微服务.....	84
2021/10/18.....	93
Spring Security.....	94
CA (证书颁发机构)	97
配置服务器的私钥.....	101
2021/10/21.....	102
概述：怎么用好 MySQL.....	105
数据库中索引的作用.....	106
2021/10/25.....	107
多个索引和复合索引.....	107

值不值得在某列上建立索引？	108
UUID 和 GUID 作为主键.....	108
MySQL 中的复合索引(Composite Index).....	110
MySQL 倒序索引.....	112
数据库的设计.....	113
2021/10/28.....	113
Text 和 Blob 类型的优化.....	113
优化 MySQL 的表.....	114
table_open_cache.....	115
MySQL 中行和列的大小限制.....	116
InnoDB 的优化方法.....	117
Autocommit.....	118
优化 InnoDB Disk I/O.....	120
2021/11/01.....	121
InnoDB_buffer.....	121
数据库的备份和恢复.....	123
2021/11/04.....	125
2021/11/08.....	126
MySQL 的分区.....	126
MongoDB.....	131
2021/11/11.....	133
图数据库比起关系型数据库的优势.....	137
用图数据库来表示好友关系.....	138
2021/11/15.....	142
图计算概论.....	142
日志结构的数据库.....	143
行列混合负载.....	143
B 树.....	145
B+树.....	146
日志结构合并树（LSTM Tree）	146
LSTM 的读放大和写放大.....	147
LevelDB.....	149
RocksDB.....	150
混合存储.....	150
解决写阻塞问题.....	151
解决读放大问题.....	152
2021/11/18.....	152
时间序列数据库.....	152
influxDB.....	153
2021/11/22.....	156
云原生数据库.....	157
识别冷热数据的分层存储引擎 X-Engine.....	160
事务处理流水线技术.....	160
数据仓库.....	161

数据湖（Data Lake）	163
2021/11/25.....	165
Nginx.....	166
Redis.....	167
2021/11/29.....	170
Xen.....	170
容器.....	171
Java 虚拟机.....	173
Docker 的具体打包和运行.....	174
2021/12/02.....	177
Kubernetes.....	180
2021/12/06.....	183
GFS(Google File System).....	184
BigTable.....	184
MapReduce.....	185
Google Percolator.....	186
边缘计算.....	188
2021/12/09.....	188
Hadoop.....	191
Hadoop Mapper.....	195
Hadoop Reducer.....	195
2021/12/13.....	196
YARN.....	196
Spark.....	197
RDD Transformation.....	203
RDD Action.....	204
宽依赖和窄依赖.....	206
2021/12/16.....	208
Spark SQL.....	210
Strom.....	212
ZooKeeper.....	216
2021/12/20.....	217
HDFS.....	217
NameNode.....	219
DataNode.....	220
HDFS Replica.....	221
2021/12/23.....	226
HBase.....	226
HBase 的配置和使用.....	229
2021/12/27.....	240
Hive.....	240
schema-on-write.....	241
schema-on-read.....	241
2021/12/30.....	248

2021/9/13

不讲的东西 RMI: Java 的远程方法调用、包括 Java security

controller 在服务多个客户的时候创建几个? Service 呢? 开发的任务到底是有状态还是无状态的? Controller 和 service 是怎么交互的? 我们默认它们在一个 tomcat 进程中通信, 那么如果在两个 Tomcat 进程中呢? 赶上双十一的时候, 大量请求过来怎么样先把订单收下来之后再去处理呢? 默认: Tomcat 中支持 AMS, 当然也可以用消息中间件 kafka 先收下订单, 然后之后再去处理。我们只是先告诉客户收到了, 结果让客户之后再去看。(异步通信)

3. 现在都是 http 包括 ajax 和 html, 后端产生一个响应, 是页面 or 请求。但是后端不会主动给你, 后端要主动推送的情况下, 需要 websocket, 可以广播, 也可以点对点。

4. 谈谈多线程, 这个相对比较独立, 在学 OS 的时候里面也会有多线程, 但是都是 C++/C 在写程序。Java 中的多线程理论上肯定是符合 OS 中学到的多线程的机制, 比如生产者/消费者, 上锁/还锁, 这一套机制怎么拿 Java 来写。

5. 下成功了两个订单, 但是只有一本书/当电子书城支持支付的时候, 需要到支付的地方把它的钱扣掉, 再生成订单, 这两个操作必须同时完成, 以一个原子性的方式执行(要么都执行, 要么都不执行), 这就是事务管理, 在 Java 中事务的边界的划分是以 annotation 标注的方式来实现的, 不同的 annotation 标注在相同的代码上, 行为是不一样的。我们只需要学会怎么利用它来实现这个功能。Mysql 自带了缓存, 加载硬盘的东西的时候其实也放到了它的缓存里, 那么我们为什么还需要分布式缓存呢? 1.Mysql 的缓存在 Mysql 里, 程序在 Tomcat 里, 如果要去拿 Mysql 的缓存, 这是一个进程间通信, 开销比较大, 如果能把这个缓存放到自己这里来, 会更快。2.Mysql 的缓存不受我们的控制, 没有办法指定数据缓存。3.在对象里可能只有一部分数据来自 Mysql, 另一部分在 MongoDB 中, 需要在受控的内存中把它缓存住。缓存要解决的是不要频繁地读取硬盘, 问题就是在分布式的环境中如何高效的存储对象。

6. 全文搜索, 是标题中有 911 还是书的简介中有 911。这该怎么办呢? 直接遍历效率肯定不高, 我们可以把简介通过中文断词断开, 然后做一个索引。Eg: "中国", "美国", "911"。将来就可以直接通过索引来找到。

7. 和支付宝对接服务, 写完的东西都可以与我交互, 需要找到一个共同的语言进行交互。比如 Json, Xml。有了 web 服务后, 需要部署一整个程序吗, 可以按照一个独立的功能进行部署吗? 也就是系统要变成微服务的架构。它和服务访问网关、函数式编程是关联起来的。

8. 然后讲讲安全管理, 上学期的大作业中, 我们做了用户名和密码登录, 在传输过程中的网络设备上做了监听, 马上就能知道密码是什么。那就需要用到 SSL (安全套接字), 它是基于证书这个体系的, SSL 是怎么加到 HTTPS 上配置出来使用的。讲了这些东西以后就会发现, 前端给用户展示的东西没有发生变化, 但是可靠性安全性就会提高。

9. 第二块我们谈谈数据库的设计，在范式的基础上，从性能、可靠性的角度去考虑，光用 Mysql 的时候，它最多支持几个表，一个表里最多有几个字段？怎么设计主键来让性能更高？到底建单列的索引还是多列的复合索引？Mysql 的缓存开多大？怎么开？怎么控制参数？
10. 数据库的备份和恢复，比如从 `checkpoint` 拿出来再把操作做一遍。书店订单一下子就爆炸的情况，就不能放在一张表里，逻辑上是一张表，但是物理上要做分区，分到多台机器上去存储。怎么保证分到多台机器上又保证效率不太低？
11. Nosql 的问题，MongoDB，在它眼里的数据不像 Mysql 一样有严格的表结构，在 MongoDB 中是 `Collection`，要求表里的记录不一定要有相同的字段，但是大家可以组织到一张表里。它也不存在表间关联，所以分区容易实现。
12. 图数据库，最简单的就是社交网络中的数据，如果用关系型数据库去存，如果要查询好友的好友的好友，就要多次 `join` 拉低性能，但是在图数据库中就非常容易。我们也可以在社交网站中找到热点人物，比如用图神经网络。
13. 日志结构数据库，代表就是 RocksDB，就是在大家设计的电子书店中，十年前的订单和现在的订单放在一起，在搜索的时候没有根据时间有相应的优化，我们希望搜最近订单更快一点。这就像日志数据一样，最往前的日志数据越不容易被读取。这就适合存储随着时间推移访问频率越来越少的数据类型。
14. 数据库的第三块：云数据库，用啥呢，那就是华为的 OpenGauss 等。我们要谈一下云数据库的特点，基于云数据库开发云原生的系统应该是怎么样开发的。这三个内容就是我们的第二个数据库相关的单元。
15. 集群部署，怎么在两台机器上跑。用 nginx 进行集群部署。第二块就是云计算和边缘计算，云计算本质上就是有很多机器，屏蔽掉其管理的复杂性。我们不需要考虑在哪个物理机上运行，我们感受不到物理服务器如果崩了以后数据的自动迁移。边缘计算也就是物计算，也就是散布在你周围的边缘设备来解决问题。Eg：怎么通过高清摄像头找到在楼中走丢的小孩。如果每个摄像头都带有一定的处理能力，根据得到的图片来计算自己看到的场景中有没有小孩。
16. 第三个就是容器，也就是 Docker，开发微服务的时候，它不是一个完整的应用，不应该为了一个微服务就跑一个虚拟机。换句话说，多个微服务可以共用容器化的环境，把对系统的调用统一起来变成一份。
17. 在 docker 中运行的新问题，十个微服务都在 docker 中，完成下订单需要 a 服务调用 b 服务调用 c 服务，这些 docker 容器到底在哪个机器上。这该怎么管呢，可以用 k8s 来管理。
18. 再往下，是第四个单元，分布式并行处理的框架，数据大到一定程度的时候，处理起来就会比较困难。在集群中，我们让每个机器各做一部分事情。
19. 把所有操作在内存中去做 Spark (`scala` 语言，其包含面向对象编程和函数式编程)，`scala` 的程序是 `java` 程序的 $1/6$ 长。它不去频繁读写硬盘，而放到了内存里。
20. 数据源源不断地过来，流式数据过来怎么办呢？Twitter 就是 storm 的流处理方案。这些数据存在哪呢？1. 收数据全部分布式存储，需要有一个分布式的文件系统，文件全部切成小块存在不同的机器上。所以 Nosql 数据库一部分要放在这里，大部分都是 HBase。
21. Hive 数据仓库，对大数据进行分析，可以使用 Kettle 和 Kylin，我们最后谈一谈开发的东西应该以怎样的方式暴露出去。

考核

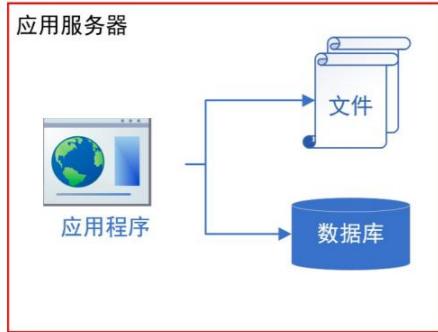
大概有 12 次作业，每次作业都 5 分。

期末考试 主观题 20 个，一个题 5 分，开卷（但没意义）。举个例子，RML 考过啥，RML

考了同一台机器上跑了两个 tomcat 一个下单 一个通知下单成功，需不需要通过 RML 协议实现。其实考察基本概念，在安全认证的时候有三种东西 code base, url, principle 访问，这三类有什么区别。只要作业是自己写的就没问题。不会让你写代码，不要写很多字。

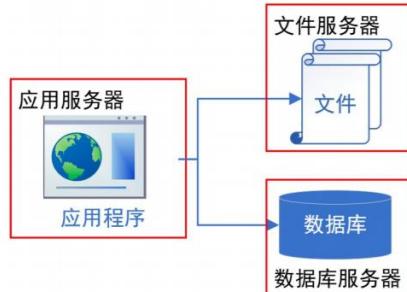
应用程序的演化

今天要谈应用程序是怎么样演化到现在这样的样子的。

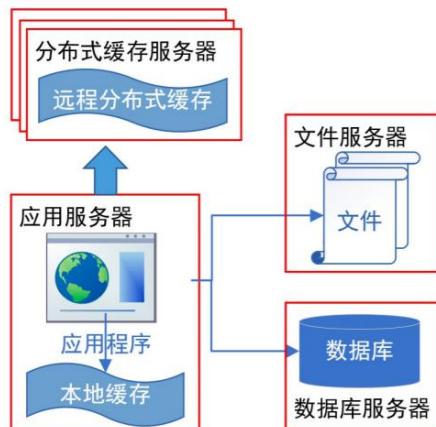


这是大二我们做的程序。

别人要访问到你，这台机器要有公网 ip，数据库也会在公网上暴露（比如防火墙忘记设置，并且默认密码没改）

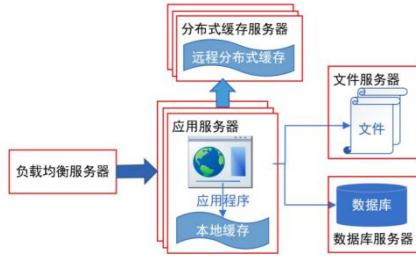


应该把数据库分配一个内网 ip，外网不能直接访问。不是说这三台机器构成了分布式系统，其实上面的单体也是分布式系统，只是分布在了不同的进程中。在上例中，三台主机可能使用不同的时间。时间很难一致，需要像手机一样统一授时。

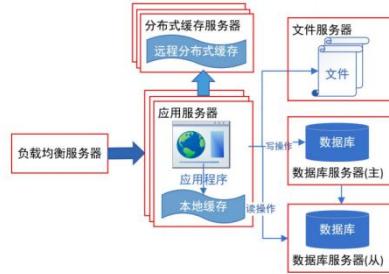


考虑到数据库比较大，缓存可能不止一个机器，构成一个分布式缓存，应用服务器大量地访

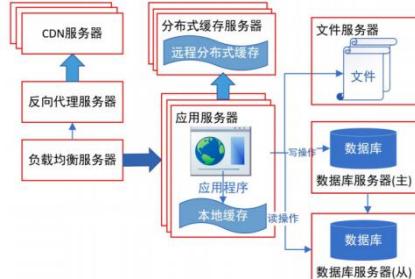
问这个缓存，性能就会高很多。在这个基础上，本地再搞一个本地缓存。这样就构成了一个分级缓存系统。本地缓存可以使用最近最少策略。



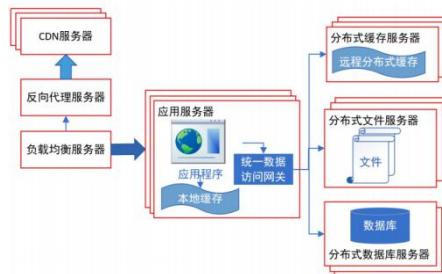
双十一到了，大量订单来，一台机子扛不住搞了三台机器，需要提供负载均衡器来分发请求到不同的应用服务器上。



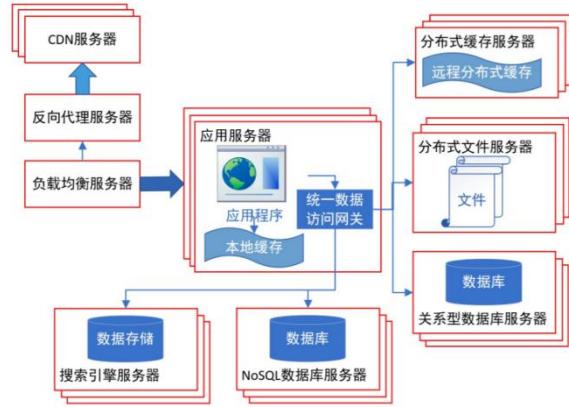
此时数据库成为瓶颈，一个数据库不行了，我们需要对数据库做主从备份，写主数据库，同步到从数据库，如果主数据库崩了就可以使用从数据库。在主从备份的情况下，从数据库还要承担一部分读数据库的操作。这样数据库也变成了集群。



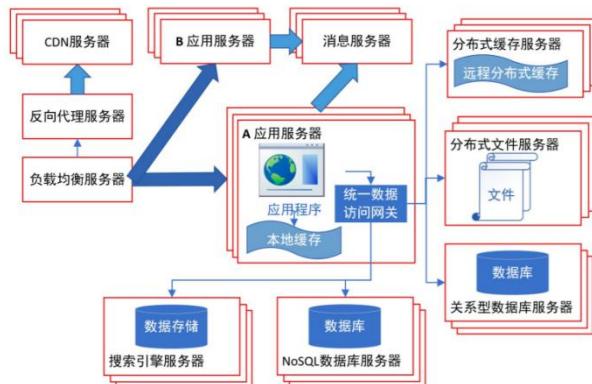
负载均衡器接收到请求分发，如果这个网站有大量的视频，光靠这些服务器是扛不住的，比如爱奇艺都是有一个 CDN（内容分发网络），它在全国骨干城市建了很多节点，用光纤连起来。如果用户在上海附近，那么 CDN 会调度到上海节点。事实上是我们有一个视频，也不是上海和北京的视频完全一样。比如视频数据原先存在纽约，中国不一定有很多人看。上海看的人逐渐多的情况下，才会在上海也放一份。反向代理是访问 CDN 网络中的某一个节点，我们也不知道具体是哪里。对我们看到的只是这一个地址。这样我们把大的视频和音频放在 CDN 中，而结构化的数据就可以放在我们自己的数据库服务器中。



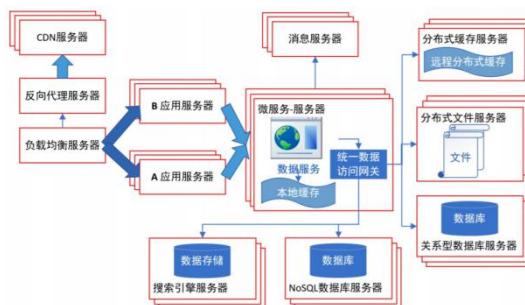
其实我们访问的数据，无论是在 CDN，分布式文件服务器，只是位置不一样，格式不一样。我们可以搞一个 DAO（统一数据访问网关）对应用程序来说只是 `getbook`, `getaudio`, `getimage`，至于内部的逻辑到底是先查哪里的数据，数据在哪里，这些细节就可以屏蔽掉，屏蔽数据的来源，我们眼里看到的都是数据。这样就好维护了。



数据库里未必都只用关系型数据库，也可以有 NoSQL 也可以有搜索引擎数据库，同样道理也会是集群。



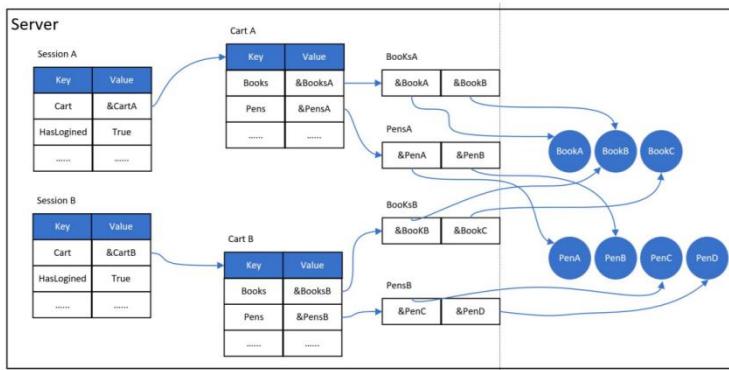
异步通信怎么办，就需要有消息中间件。前端的负载均衡器可以用来均衡 A,B,C 应用……。我们可以把 ABC 中共用的数据访问网关和其他公用的部分抽取出来



ABC 应用直接和我们暴露出来的微服务打交道。这样这些应用只需要实现自己特有的逻辑就可以了。大量的东西是和应用本身无关的，可以抽取出来作为框架。所以我们在课程中会学习怎么用第三方框架。

再反过来说，第一个单体系统外观上可能和最后的系统可能是一模一样的。但是数据访问效率、可靠性都提高了非常多。量变达到质变，量达到一定程度搞不定的时候，就必须要演化架构。

有状态(stateful)和无状态(stateless)服务的区别



HTTP 协议是一个有状态的协议，大二的时候我们谈到了 http session，是服务器端的一个对象。每个客户端拿到 id，在请求中 getsession 可以拿到 id。Server 里头的 session 其实就是放进去的 key-value。

Spring Boot 中的作用域

Spring Boot 中 Bean 的作用和对应注解：

对于 Spring 管理的对象，称为 Bean。用 SpringBoot 开发应用时，我们会用注解将对象交给 Spring 容器管理。这些注解包括：

`@Component, @Service, @Bean, @Controller, @Repository`

这些注解本质上都是 Spring 用来进行 Bean 的自动检测的标识。标注这些注解的类会被 Spring 容器管理。

- `@Componet` 一般的组件。
- `@Service` 是 Service 层组件。
- `@Bean` 定义在`@Configuration` 类中的 Bean，和早期 Spring 的 xml 配置对应。
- `@Controller` 是用在 SpringMVC 控制层。
- `@Repository` 是数据访问层。

Spring 这样设计，是因为，这些注解不光是要做自动检测。同时有不同的功能，比如 `@Repository` 注解，Spring 会增加增强处理，进行相关的异常处理。`@Controller` 的 bean 会处理网络请求相关逻辑。所以你给所有的 Bean 都标注同一个注解，确实都会注入 Spring 容器，但是功能可能就会失效。而且随着 Spring 版本升级，可能会增加更多差异化处理。所以我们应该按照规范来注解。

Q：在 Spring Boot 中，我们上学期写的书店里的 service 和 controller 在运行过程中到底有几个实例？它们运行时的作用域是什么？

A: Spring 作用域如下表

Scope	Description
Singleton(default)	在每个 Spring loc 容器中，一个 bean 定义对应只会有唯一的一个 bean

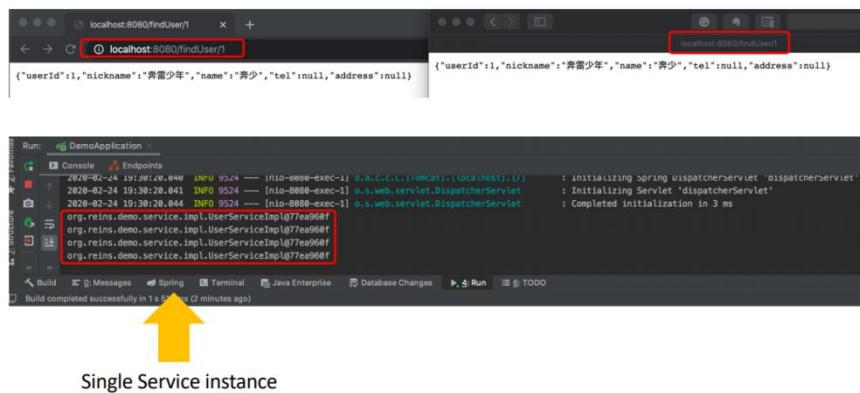
	实例。
prototype	一个 bean 定义可以有多个 bean 实例。
request	一个 bean 定义对于单个 HTTP 请求的生命周期，也就是说每个 HTTP 请求都有一个 bean 实例，且该实例仅在这个 HTTP 请求的生命周期里有效。该作用域只适用于 WebApplicationContext 环境。
session	一个 bean 定义对于单个 HTTP Session 的生命周期，也就是说每个 HTTP Session 都有一个 bean 实例，且该实例仅在这个 HTTP Session 的生命周期里有效。该作用域只适用于 WebApplicationContext 环境。
application	一个 bean 定义对应于单个 ServletContext 的生命周期。该作用域只适用于 WebApplicationContext 环境。
websocket	一个 bean 定义对应于单个 websocket 的生命周期。该作用域只适用于 WebApplicationContext 环境。

Singonton 的代码实现

```
UserController.java
@RestController
public class UserController {
    @Autowired
    private UserService userService;
    @GetMapping(value = "/findUser/{id}")
    public User findOne(@PathVariable("id") String id) {
        System.out.println(userService);
        return userService.findUserById(Integer.valueOf(id));
    }
}

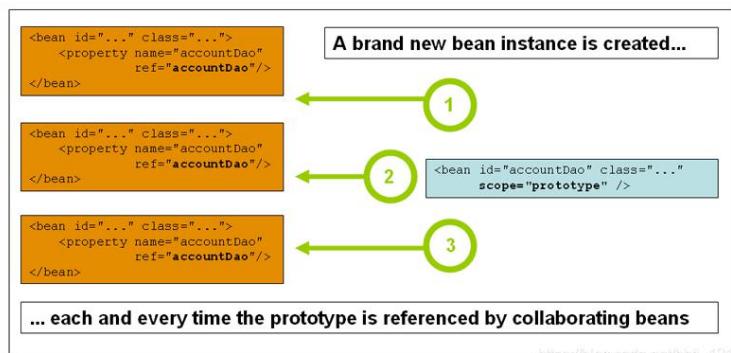
UserServiceImpl.java
@Service
public class UserServiceImpl implements UserService{
    @Autowired
    private UserDao userDao;
    @Override
    public User findUserById(Integer id) {
        return userDao.findOne(id);
    }
}
```

结果如下图所示，开两个浏览器各自发送两次请求，在后端日志中我们看到 service 对象的 id 是同一个。所以 controller 和 service 在默认的情况下都是一个实例。



Prototype 的代码实现

`prototype` 作用域表示的是一个 bean 定义可以创建多个 bean 实例，有点像一个类可以 `new` 多个实例一样。也就是说，当注入到其他的 bean 中或者对这个 bean 定义调用 `getBean()` 时，都会生成一个新的 bean 实例。



所以我们回到刚才的例子中，如果我们把 `UserServiceImpl` 加上了 `@Scope("prototype")`，而 `UserController` 还是维持默认的 `singleton` 的话，两个浏览器的访问结果还是访问了同一个对象，因为 `UserController` 只有一个，那么注入的 `UserServiceImpl` 自然是同一个。

只有两个都设置为 `@Scope("prototype")` 才可以：

```

UserController.java
@RestController
@Scope("prototype")
public class UserController {
    @Autowired
    private UserService userService;
    @GetMapping(value = "/findUser/{id}")
    public User findOne(@PathVariable("id") String id) {
        System.out.println(userService);
        return userService.findUserById(Integer.valueOf(id));
    }
}
UserService.java

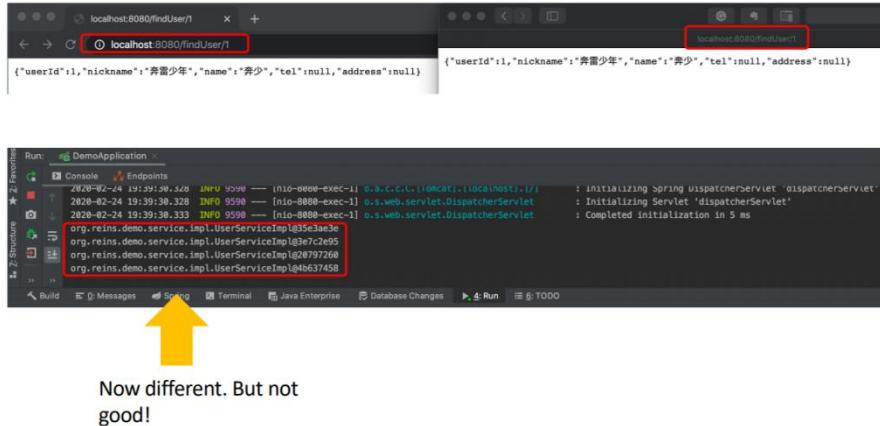
```

```

@Service
@Scope("prototype")
public class UserServiceImpl implements UserService{
    @Autowired
    private UserDao userDao;
    @Override
    public User findUserById(Integer id) {
        return userDao.findOne(id);
    }
}

```

结果如下图所示，在这种情况下，4次请求会创建4个controller，其各自注入了一个service。其实是不太好的，我们希望共用1~2个controller，然后对每个请求再安排一个service。



为了实现这个需求，我们把代码更改为如下：

```

UserController.java
@RestController
public class UserController {
    @Autowired
    WebApplicationContext applicationContext;

    @GetMapping(value = "/findUser/{id}")
    public User findOne(@PathVariable("id") String id) {
        UserService userService = applicationContext.getBean(UserService.class);
        return userService.findUserById(Integer.valueOf(id));
    }
}

UserServiceImpl.java
@Service
@Scope("prototype")
public class UserServiceImpl implements UserService{
    @Autowired
}

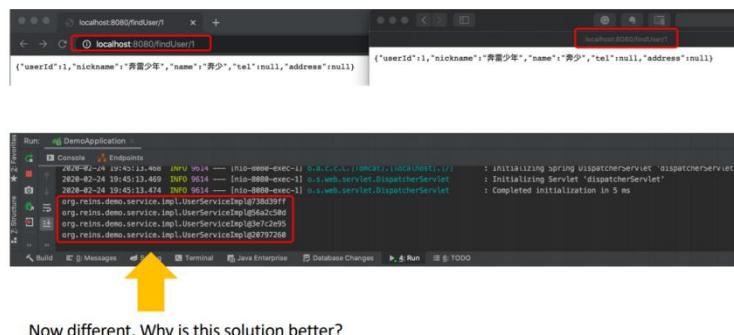
```

```

private UserDao userDao;
@Override
public User findUserById(Integer id) {
    return userDao.findOne(id);
}
}

```

此时，我们的 controller 是 singleton 模式，所以只有 1 个 controller，然后 service 不再是直接注入的方式，而是通过 `WebApplicationContext.getBean` 函数去获取，这就类似于 `new` 的作用，所以我们得到了 1 个 controller+4 个 service。



Now different. Why is this solution better?

Session 的代码实现

```

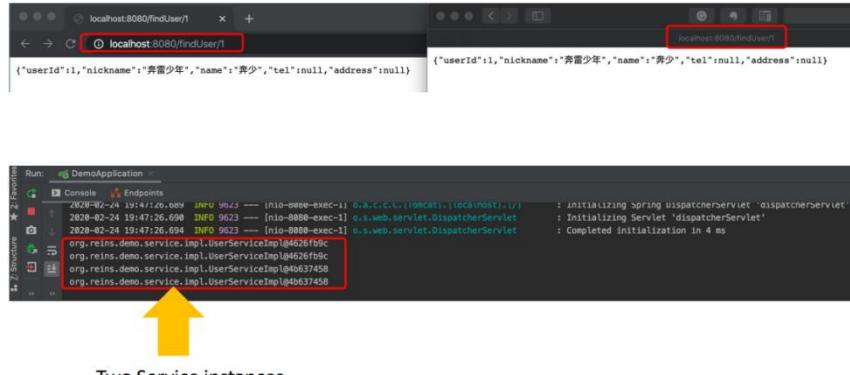
UserController.java
@RestController
public class UserController {
    @Autowired
    WebApplicationContext applicationContext;

    @GetMapping(value = "/findUser/{id}")
    public User findOne(@PathVariable("id") String id) {
        UserService userService = applicationContext.getBean(UserService.class);
        return userService.findUserById(Integer.valueOf(id));
    }
}

UserServiceImpl.java
@Service
@Scope("session")
public class UserServiceImpl implements UserService{
    @Autowired
    private UserDao userDao;
    @Override
    public User findUserById(Integer id) {
        return userDao.findOne(id);
    }
}

```

}



根据 session 来，就有 1 个 controller，2 个 service。

- 2 Browsers(2 Sessions), 4 Requests(2 times / session)

		Controller	
		Prototype	session
Service	prototype	4 service instances 4 controller instances	4 service instances 2 controller instances
	session	2 service instances 4 controller instances	2 service instances 2 controller instances

- How should we design the scopes of Beans?

总结一下，讲这么多有啥意义。如果有一个变量叫做用户 ID，我们希望把 uid 写入到 order 里，service 需要记住 uid 是谁，那应该是一个用户专用一个 service 对象，这个 uid 就叫做状态。

不同的用户不同的 uid，用不同的 service 把它维护起来。无状态的话，会用最新用户的 uid 来抹掉之前的 uid。而有状态就是三个 service 维护三个 uid。这两种区别其实都是内存的消耗问题。

在 tomcat 里，它实际上有一个实例池。如果 service 为一个用户创建一个对象，那么会创建几个呢。像购物车这种东西就不方便抹掉，如果实例池很小 (size=2)，A 用户访问 tomcat，新创建一个 service。B 也访问，创建一个 service。C 来了，没有 C 的对象，也没有空对象，也没有位置了。那就在最近最小的对象 A 的 service 把它的状态写到硬盘上去。空出实例池，放 C 的 service。A 继续来，A 的对象数据在硬盘上，淘汰掉对象池中的一个对象，读入硬盘中的 A service 状态。这些都是 tomcat 帮我们处理的。但是在这个过程中，有很多问题，需要频繁地从硬盘上进行读写，实例池开的小就是频繁读写；实例池开得大，平时的时候就浪费了内存。尽可能无状态的多一点。

2021/9/16

消息机制

怎么通过 java 来发送消息接收消息。

实际上消息机制对应的是异步通信机制，需要根据需求设计异步通信机制。上学期开发

的程序是怎么样的。在客户端是浏览器，发一个请求到 server，比如说我们的 Tomcat 应用。有分层 controller->service->repository 最终到数据库，上学期是 MySQL。产生结果以后返回一个客户端，前端发个请求过来以后页面就等在那里等待整套流程。我们用 ajax，比如前端看上了一本书下订单，点完这个页面不会变成白色，页面可以继续操作。当网络包返回响应以后，再把对应的地方刷掉。这就是异步，是它产生响应之后有一个回调函数去刷新。它什么时候能被刷掉我不知道，但是至少发完请求以后我们并没有阻塞在那里什么都干不了。

我们再看这个场景，下订单过来要处理去写数据库，要通过 jdbc 去访问，它的连接数是有上限的，tomcat 是用了连接池。比如我们的 mysql 的连接上限是 30 个，tomcat 有两种做法：1. 请求来的时候，如果没到上限就产生新的连接；如果到了上限，就分时复用，一个连接服务很多请求，挂起最近最少使用。2. 一次性创建好 30 个连接，客户端来了之后再这 30 个连接中找一个空闲的去执行 sql，执行完就释放出来给别的 client 去用。这种方法看起来不错，但是碰上了双十一就扛不住了。一个连接负责 30 万个客户，一个客户几毫秒也非常慢。这种情况下，因为在 session 上有超时，也就是会抛出异常了。这种情况下用户的体验就会非常差。

Eg：让同学把随堂卷子放到桌子上，老师有空了再去批。我们通过桌子这个中介，只要有卷子就拿过去批改。这样同学和老师就是 P2P 对等的两方。

1. 性能会更高，放完卷子就可以走了
2. 也不用管处理不处理得过来，只要放进来就行。

所谓的消息机制就是支持 P2P 的这种通信方式，为什么是 P2P 呢，同学和老师之间谈不上是服务器和客户端，角色是相同的都可以向消息机制中发消息和读消息。

异步是啥意思。Eg：同学交到桌子的上面，改完以后把卷子放到桌子的下面，同学要想知道卷子考了多少分，是事后不知道什么时候才知道的。

刚才全部性能的差异是数据库和后端的 30 个连接搞不定双十一流量。我们再仔细分析这个流程，controller 在响应客户端发来的请求时候原来是调用 service，调用 dao 到最底层回来，现在它也调用了 service。但是 service 现在把请求（下订单）放到一个队列中，controller 直接返回到。另一个 service 从队列中读过来请求数据，然后调用 dao，到数据库中做业务逻辑。这个时候就可以看到，service2 慢慢地处理请求，数据库可以不用同时处理很多请求。为什么要这么做，原因就是同步的不行。消息机制就是让我们实现一个 P2P 机制，实现我们松散的耦合，只要放在队列中就行。

处理请求可以由不同的对象来改，但是保证卷子都被改了，这就是解耦。

消息机制的特性

消息机制的两个特性：

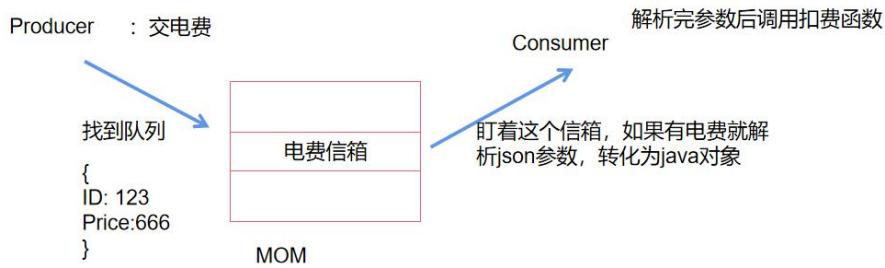
1. 异步，不需要 controller 等待数据库处理完，只需要扔到消息队列里就可以，至于什么时候真正处理完，到时候再想办法获取。
2. 可靠，如果我 controller 调用的对象 O 不存在/锁死/超时，这次请求就直接崩溃了，也就是交卷的时候老师不在就不知道怎么办了，而消息机制不需要要求老师当场收卷子。如果桌子有给老师的推送功能，老师可以没接收到，可以多尝试几次推送直到老师接收到。换句话说，试一次出错了没关系，不像原先直接就崩了。

一个例子：

在 javaee 企业版中的例子，有个百货公司查看商品目录发现缺货，给工厂发消息去造，

工厂有些配件是其他厂商造的，以此类推所有都收到了。相当于写封信放到了信箱里，人在不在无所谓，总会来看的。

消息的生产者和消费者，之间是消息中间件



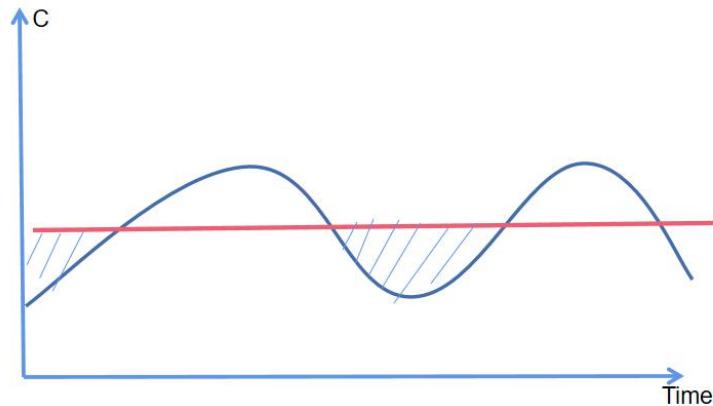
P 从来眼里就没有 C，眼中只有队列。C 只需要盯着这个队列，谁传递进来无所谓。C 程序升级了，觉得原先的程序不好，希望用新的逻辑去处理。接收到对象后想处理新的逻辑，所以 C 的代码重构不影响 P 的代码。

这种方式有什么坏处？它拿一个 json 对象过来组成一个 java 对象调用。如果 producer 的格式写错了，ID 写为了 IDX，此时 consumer 就没有办法解析。强类型的语言变成了弱类型，编译器不能爆出对应的错误异常。同时还需要把好好的一个对象通过一次纯文本的 json 表示，单个请求处理的速度是快了，但是响应的速度是慢了。有中介肯定就会带来性能损耗，所以不能把所有东西都改为消息机制传递。同时，需要有额外的一个推送机制告诉用户，如果抛出了异常的情况，不能马上告诉客户端，也需要客户端来拉。

辩证地看待消息机制

用不用消息机制的考量：

1. 如果卷子的题目比较少、人数比较少，那还是同步比较好，编程模型比较简单。
2. 当我们性能确实发现了问题，就需要使用了消息的方式。前提是老师会有不忙的时候，如果老师一直忙着，或者数据增加了，如下图所示：



换句话说，一定要存在阴影部分（消息累积速度小于处理速度的情况），消息机制才会起效。

JMS API，后面给了三个例子，用了卡夫卡，内嵌的 tomcat。

所谓消息驱动就是消息过来就驱动它执行。还有一种情况会碰到事务的机制，讲到事务

的时候再去讲。

消息的数据结构

消息就是一个对象，这个对象在发送的时候要求你包含消息头（就像写邮件一样，可以理解为邮件的信封，或者发短信的对象号码，必须要有对方的收件地址）、一些可选的消息属性、真正在信封里的信息（可选）。

Includes some pre-defined fields

- JMSDestination (S)
- JMSDeliveryMode (S)
- JMSMessageID (S)
- JMSTimestamp (S)
- JMSCorrelationID (C)
- JMSReplyTo (C)
- JMSRedelivered (P)
- JMSType (C)
- JMSExpiration (S)
- JMSPriority (S)

Clients can not extend the fields

1. 发送地址
2. 发送的模式，比如发送的时候人不在是保持一段时间内尝试转发还是转发失败直接扔掉（听起来不太合理，eg: 比如接收端是交易所打算推送行情，Producer 是用户，如果推送的行情因为网络失败转发失败了，此时如果在之后再转发，那用户就会收到过往的行情造成损失，要保证能收到的消息一定是最新的消息）。
3. ID, 时间戳
4. 关联 ID: 消息是否因为太大被切成了两个消息
5. Replyto: 发件人地址
6. Expiration 过期时间

C: 表示人为填进去的；S: 你在调用 S 的方法调用的时候自动填进去的；P: 中间件自动填进去的。尺寸是固定的。

Includes some pre-defined fields

- JMSXUserID (S)
- JMSXAppID (S)
- JMSXDeliveryCount (S)
- JMSXGroupID (C)
- JMSXGroupSeq (C)
- JMSXProducerTXID (S)
- JMSXConsumerTXID (S)
- JMSXRcvTimestamp (S)
- JMSXState (P)

Clients can extend the fields

- Property name: follows the rule of naming selectors
- Property value: boolean, byte, short, int, long, float, double, String

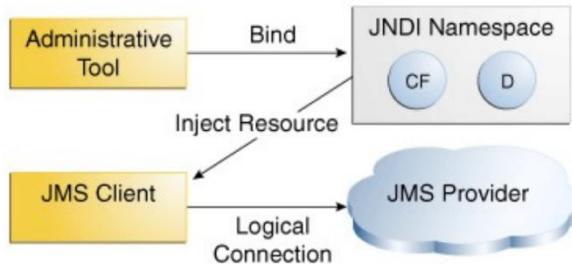
消息属性值中也预定义了这些 key-value 对，GroupID 主要处理群发和群收的情况。整个消息属性是可以扩展的，如果想告诉别人这条消息不是所有人都可以收，可以增加一些便于过滤的消息属性。

Message Type	Body Contains
TextMessage	A java.lang.String object (for example, the contents of an XML file).
MapMessage	A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

消息体就是消息中放的信息

1. 纯文本
2. 键值对的表
3. 放的是字节，和第一个纯文本有什么区别呢？如果我们想保证 P 和 C 都能读得懂，字节流就是考虑到了不同语言直接使用消息队列的情况。

这五种类型公共父类就是 `message` 类型。流类型就是中间放的是流对象，需要一定是可序列化的对象。

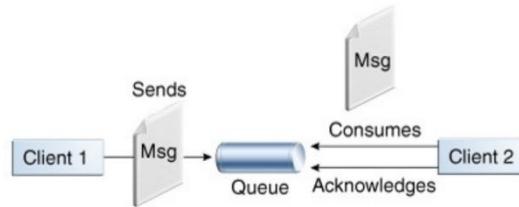


消息机制的编程模型

消息机制的编程模型，我们可以把消息中间件想象成 MySQL 一样的数据库，需要获取消息队列的连接，就像我们操作数据库中的表一样，需要创建一个发送/接收请求。所以先要到 JNDI 树（数据库对象绑定在之上）上，通过 `datasource` 获取到消息中间件的连接，看起来就像一个连接工厂，然后要找一下这个队列在哪里，这个队列也要绑定在 JNDI 树上，创建连接后创建发送者/接收者。

A **point-to-point** (PTP) product or application is built on the concept of message **queues**, **senders**, and **receivers**.

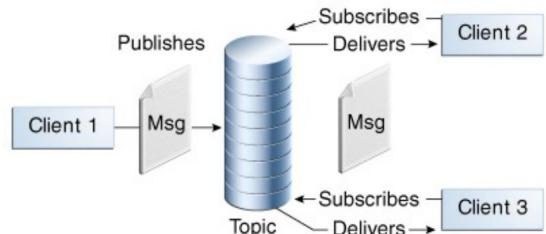
- Each message has only one consumer.
- The receiver can fetch the message whether or not it was running when the client sent the message.



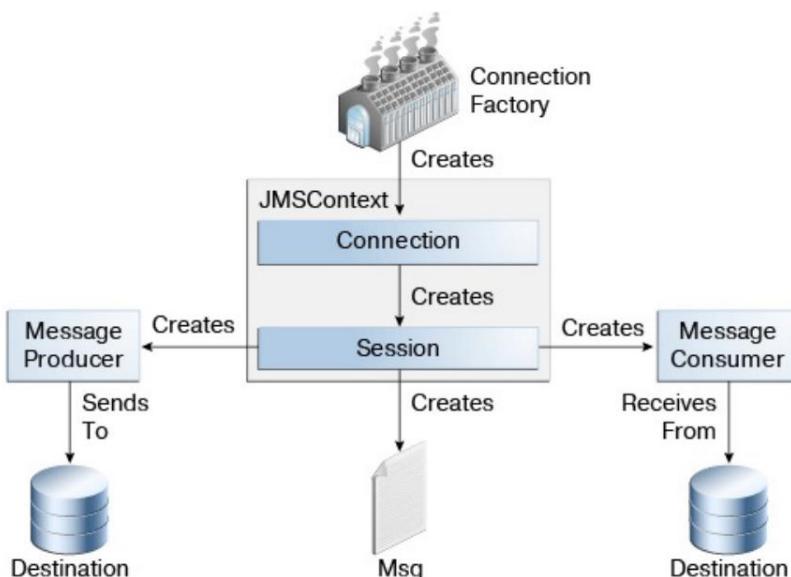
在标准 JMS 中，有两种发送和接收消息的模式，只有一点差异，所以在卡夫卡中不做区分。一种是点对点，一旦这个人拿到了这封信，其他人是不可能拿到的。每个消息只有一个接收者，共有一个信箱后通过名字进行区分。

In a **publish/subscribe** (pub/sub) product or application, clients address messages to a **topic**, which functions somewhat like a bulletin board. Publishers and subscribers can dynamically publish or subscribe to the topic.

- Each message can have multiple consumers.
- A client that subscribes to a topic can consume only messages sent *after* the client has created a subscription, and the consumer must continue to be active in order for it to consume messages.



发布预定模式，只要订阅了信息，就可以被多个客户端接收到。有多个消费者，这个目的地就叫 **topic**，每个信息都一些主题，按照订阅的主题去选。



编程的时候，创建工厂，创建连接，创建 MessageProducer/consumer，然后在 session

上创建消息。这个消息最终都是在消息队列里。

- Get a reference to `ConnectionFactory`
- A **connection factory** is the object a client uses to create a connection to a provider.

```
import javax.naming.*;
import javax.jms.*;
QueueConnectionFactory queueConnectionFactory;
Context messaging = new InitialContext();
queueConnectionFactory = (QueueConnectionFactory)
    messaging.lookup("QueueConnectionFactory");
```

- or

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
```

注入连接工厂，先获取这个 `QueueConnectionFactory`，先获取 JNDI 树的根，然后再去 `lookup`。JNDI 树其实就是 key-value。这个工厂拿到之后就可以在其之上创建连接。这就是我们获取消息中间件的途径，

- A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes.
 - In the PTP messaging style, destinations are called queues.
 - In the pub/sub messaging style, destinations are called topics.

```
Queue queue;
queue = (Queue)messaging.lookup("theQueue");
Topic topic;
topic = (Topic)messaging.lookup("theTopic");
```

- Or

```
@Resource(lookup = "jms/MyQueue")
private static Queue queue;
@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```

所有的东西都用 JMS。然后我们就在这个上下文上进行操作。其中包了 `connection` 和 `session` 对象，所以在它上面就可以创建一个消费者，在消费者上 `send` 方法对着某个目的地发消息过去。

先说同步调用，`consumer.receive()`阻塞，而 `receive(1000)` 就等一秒然后返回。

Create a Producer or a Consumer

```
try (JMSContext context = connectionFactory.createContext()) {  
    JMSProducer producer = context.createProducer();  
    ...  
    context.createProducer().send(dest, message);
```

Or

```
try (JMSContext context = connectionFactory.createContext()) {  
    JMSConsumer consumer = context.createConsumer(dest);  
    ...  
    Message m = consumer.receive();  
    Message m = consumer.receive(1000);
```

Create a message

```
TextMessage message = context.createTextMessage();  
message.setText(msg_text); // msg_text is a String  
context.createProducer().send(message);  
  
Message m = consumer.receive();  
if (m instanceof TextMessage) {  
    String message = m.getBody(String.class);  
    System.out.println("Reading message: " + message);  
} else {  
    // Handle error or process another message type  
}
```

消息从哪来呢，需要创建消息，在 context 上创建消息。我们可以创建一个纯文本消息。Sendtext 设置完就有消息了。还可以通过 setheader 或者 setattribute 去设置消息头和消息属性。这样我们就拿到了放的 string。

- JMS Message Listener
- A message listener is an object that acts as an asynchronous event handler for messages.

```
Listener myListener = new Listener();  
consumer.setMessageListener(myListener);
```

Listener:

```
void onMessage(Message inMessage)
```

那异步接收消息怎么办呢？我们要创建一个监听器类型，这个监听器里有一个 onMessage 接口要让我们去实现，想怎么处理就这么处理。注册完就结束了。一有消息就会触发 onMessage 消息，触发消费者上 listener 的 onMessage 方法。它比直接调用方法要复杂得多，更接近 JDBC 编程的方式，比较麻烦。这样也可以理解这个过程为什么是比较慢的。

- Selection of messages
- Producer:

```

String data;
TextMessage message;
message = session.createTextMessage();
message.setText(data);
message.setStringProperty("Selector", "Technology");

```

上图告诉我们怎么扩展消息的属性。

```

String selector;
selector = new String(" (Selector = 'Technology') ");

```

```
JMSConsumer consumer = context.createConsumer(dest, selector);
```

我们可以创建一个 `selector`, 作为第二个参数传进去, 对着这个 `destination`, 只接收符合标准的消息。只要这个消息中没有这个消息, 就一概不接收。

Durable subscription

```

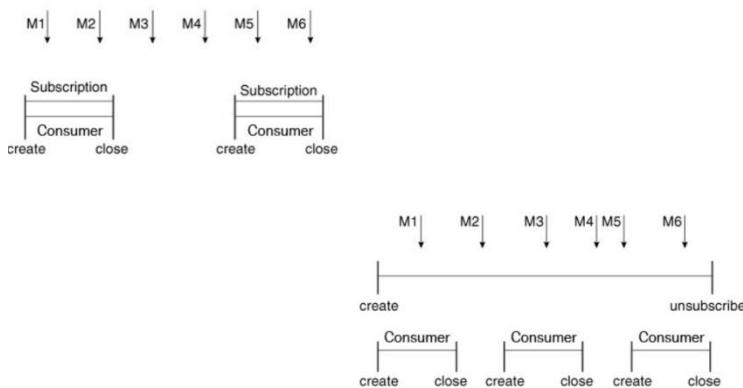
String subName = "MySub";
JMSConsumer consumer = context.createDurableConsumer(myTopic, subName);

consumer.close();
context.unsubscribe(subName);

```

```
JMSConsumer consumer =
context.createSharedDurableConsumer(topic, "MakeItLast");
```

交卷过来出去玩的时候, 此时数据也不能丢。这就是持久化预定。怎么做的呢? 创建的 consumer 需要 durableconsumer, 除了 destination 之外, 还需要增加一个 `subName` 作为标识。每次重启的时候, 就把之前堆积的消息一起给你。



只要在线就把离线之后堆积的消息都推送给消费者。

如果消息是给多个消费者消费的, 那么就可以用 `shareddurableconsumer`。

JMS Browser

- allows you to browse the messages in the queue and display the header values for each message.

```
QueueBrowser browser = context.createBrowser(queue);
```

Consumer 会真正消费掉消息，browser 只是看看队列中有什么，而不是真正消费掉什么。只要在这个过程中有异常，就是 JMSException。

- IntelliJ IDEA 创建消息驱动Bean - 接收JMS消息
 - <https://www.cnblogs.com/yangyquin/p/5346104.html>
- Wildfly Messaging Configuration
 - <https://docs.jboss.org/author/display/WFLY/Messaging+configuration>



JBOX 里头内嵌了消息中间件，刚才我们知道 Connection Factory 和消息队列都在 JNDI 树上。

1. 添加用户
2. 写在 mgmt-users.properties
3. 在 standalone.xml 中配置消息的目的地

在管理控制台中监督 server，比如我们可以看 JNDI 树。

```
public class JMSPub {  
    private static final String DEFAULT_USERNAME = "root";  
    private static final String DEFAULT_PASSWORD = "reins2011!";  
    private static final String INITIAL_CONTEXT_FACTORY = "org.wildfly.naming.client.WildFlyInitialContextFactory";  
    private static final String PROVIDER_URL = "http-remoting://localhost:8080";  
    private static final String DEFAULT_CONNECTION_FACTORY = "jms/RemoteConnectionFactory";  
    private static final String DEFAULT_DESTINATION = "/jms/topic/HelloWorldMDBTopic";  
    private static final Logger log = Logger.getLogger(JMSPub.class.getName());  
    public static void main(String[] args) throws Exception {  
        ConnectionFactory connectionFactory = null;  
        Connection connection = null;  
        Session session = null;  
        Topic topic = null;  
        Context context = null;  
        MessageProducer producer = null;  
        BufferedReader msgStream = null;  
        try {  
            log.info(msg: "设置JNDI访问环境信息也就是设置应用服务器的上下文信息!");  
        } catch (Exception e) {  
            log.error("Error occurred while setting JNDI environment: " + e.getMessage());  
        } finally {  
            if (msgStream != null) {  
                try {  
                    msgStream.close();  
                } catch (IOException e) {  
                    log.error("Error closing BufferedReader: " + e.getMessage());  
                }  
            }  
            if (producer != null) {  
                try {  
                    producer.close();  
                } catch (JMSException e) {  
                    log.error("Error closing MessageProducer: " + e.getMessage());  
                }  
            }  
            if (session != null) {  
                try {  
                    session.close();  
                } catch (JMSException e) {  
                    log.error("Error closing Session: " + e.getMessage());  
                }  
            }  
            if (connection != null) {  
                try {  
                    connection.close();  
                } catch (JMSException e) {  
                    log.error("Error closing Connection: " + e.getMessage());  
                }  
            }  
            if (connectionFactory != null) {  
                try {  
                    connectionFactory.close();  
                } catch (JMSException e) {  
                    log.error("Error closing ConnectionFactory: " + e.getMessage());  
                }  
            }  
        }  
    }  
}
```

```

final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
env.put(Context.PROVIDER_URL, PROVIDER_URL);
env.put(Context.SECURITY_PRINCIPAL, DEFAULT_USERNAME);
env.put(Context.SECURITY_CREDENTIALS, DEFAULT_PASSWORD);
context = new InitialContext(env);
log.info( msg: "初始化上下文, 'JNDI驱动类名', '服务提供者URL', '应用用户的账户', '密码'完毕。");
log.info( msg: "获取连接工厂!");
connectionFactory = (ConnectionFactory) context.lookup(DEFAULT_CONNECTION_FACTORY);
log.info( msg: "创建JMS连接、会话、主题!");
connection = connectionFactory.createConnection(DEFAULT_USERNAME, DEFAULT_PASSWORD);
session = connection.createSession( b: false, Session.AUTO_ACKNOWLEDGE);
topic = (Topic) context.lookup(DEFAULT_DESTINATION);
producer = session.createProducer(topic);
msgStream = new BufferedReader(new InputStreamReader(System.in));
String line = null;
boolean quitNow = false;
do {
    System.out.print("输入要发送的消息: (数字0退出)");
    line = msgStream.readLine();
    if (line != null && line.trim().length() != 0) {
        TextMessage textMessage = session.createTextMessage();
        textMessage.setText(line);
        textMessage.setStringProperty( s: "Selector", s1: "Funny");
        producer.send(textMessage);
        quitNow = line.equalsIgnoreCase( anotherString: "0");
    }
} while (!quitNow);

```

上述的代码中的一些细节在此处补充细节

Properties 类位于 `java.util.Properties`, 是 Java 语言的配置文件所使用的类, `Xxx.properties` 为 Java 语言常见的配置文件, 如数据库的配置 `jdbc.properties`, 系统参数配置 `system.properties`。**Properties** 类以 `key=value` 的键值对的形式进行存储值。

工厂方法抛弃设计模式的优点可以认为帮助我们 `new` 出了一个 `connection`。我们利用这个 `connection` 创建和消息队列通信的 `session`。

连接 (Connection)：连接是从客户端到 ORACLE 实例的一条物理路径。连接可以在网络上建立, 或者在本机通过 IPC 机制建立。通常会在客户端进程与一个专用服务器或一个调度器之间建立连接。

会话(Session) 是和连接(Connection)是同时建立的, 两者是对同一件事情不同层次的描述。简单讲, 连接(Connection)是物理上的客户端同服务器的通信链路, 会话(Session)是逻辑上的用户同服务器的通信交互。

此处的 `Session` 类是 JMS 中的, 要区分一下 `HttpSession` 类。

以下再复习一下 JDBC 和 JNDI

JDBC api 提供了在 java 中可编程的访问关系型数据的方法, JDBC 3.0 api 包含了 `java.sql` 和 `javax.sql`。

JDBC 需要有两个步骤:

1. 建立连接: JDBC API 定义了 `Connection` 接口来描述和底层的数据源的连接 (`DriverManager` 或者是 `DataSource`)
2. 执行 SQL 语句和操作结果

- **DatabaseMetadata**
- **Statement, PreparedStatement, and CallableStatement.**
- **ResultSet and RowSet**

为了获取一个连接，应用程序需要两者选一：

1. **DriverManager** 类和一或多个 **Driver** 类的实现
2. 实现 **DataSource** 类

```
DriverManager
- registerDriver and getConnection

package sample;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class AccessDB {

    public static void main(String[] args) throws
        ClassNotFoundException, SQLException
    {
        Class.forName("com.mysql.jdbc.Driver");

        String url = "jdbc:mysql://localhost:3306";
        String user = "root";
        String passwd = "12345678";

        Connection con = DriverManager.getConnection(url, user, passwd);
        System.out.println(con.getTransactionIsolation());
    }
}
```

上文中 **Class.forName** 函数将 mysql 的 driver 注册到 **DriverManager** 中。然后我们就可以从 **DriverManager** 中获取到和数据库的 **Connection**。

• **DataSource**

- A logical name is mapped to a **DataSource** object via a naming service that uses the Java Naming and Directory Interface™ (JNDI).

Property Name	Type	Description
databaseName	String	name of a particular database on a server
dataSourceName	String	a data source name
description	String	description of this data source
networkProtocol	String	network protocol used to communicate with the server
password	String	a database password
portNumber	int	port number where a server is listening for requests
roleName	String	the initial SQL rolename
serverName	String	database server name
user	String	user's account name

DataSource
- A logical name is mapped to a **DataSource** object via a naming service that uses the Java Naming and Directory Interface™ (JNDI).

```
package sample;
import java.sql.SQLException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.mysql.jdbc.Jdbc2.optional.MysqlDataSource;
public class Server {
    public static void main(String[] args) throws ClassNotFoundException,
        SQLException, NamingException
    {
        MysqlDataSource ds = new MysqlDataSource();
        ds.setServerName("localhost");
        ds.setPortNumber(3306);
        ds.setUser("root");
        ds.setPassword("12345678");

        Context namingContext = new InitialContext();
        namingContext.bind("rmi://localhost:1099/datasource", ds);
    }
}
```

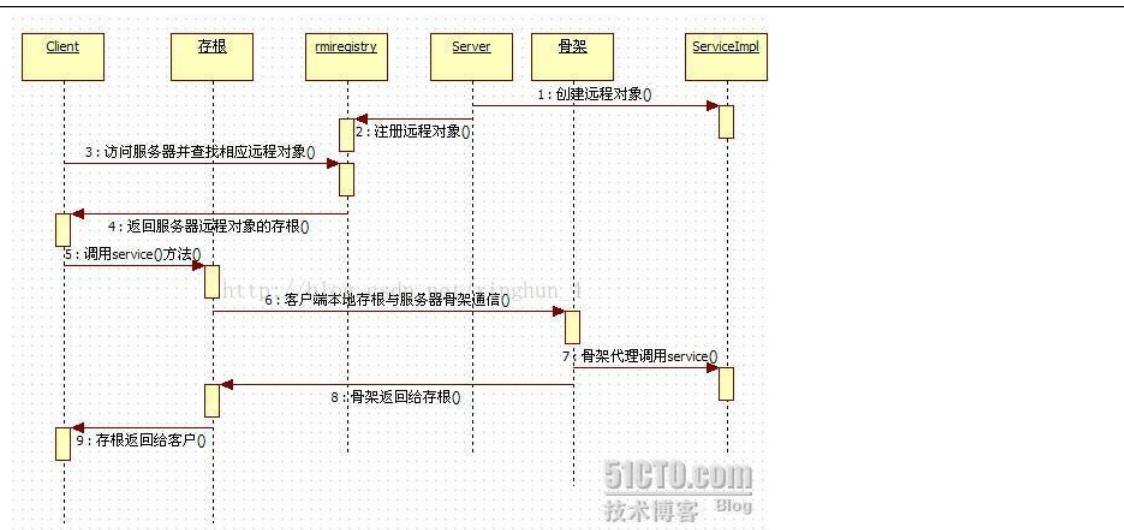
我们把一个本地的名字映射给 **DataSource** 对象，通过一个 Java 的命名服务 (JNDI,java naming and directory interface)

J2EE 规范要求所有 J2EE 容器都要提供 JNDI 规范的实现。JNDI 在 J2EE 中的角色就是“交换机”——J2EE 组件在运行时间接地查找其他组件、资源或服务的通用机制。

JNDI 说白了就是把资源取个名字，再根据名字来找资源。因为所有资源都可以通过这样来管理，就提供了统一的接口。

```
public class Client {
    public static void main(String[] args) throws
        NamingException, RemoteException, SQLException
    {
        Context namingContext = new InitialContext();
        String url = "rmi://localhost:1099/datasource";
        MysqlDataSource ds = (MysqlDataSource) namingContext.lookup(url);
        Connection con = ds.getConnection("root", "12345678");
        System.out.println(con.getTransactionIsolation());
    }
}
```

其中提到了 rmi 远程调用方法：RMI 是运行在一个 java 虚拟机上的对象调用运行在另一个 java 虚拟机上的对象的方法。RMI 的用途就是为 java 程序之间的远程通信提供服务。



RMI 的调用流程如上图所示，需要提供远程接口调用的 RMI server 需要先在 rmi service 中注册远程对象。RMIService 即 JDK 提供的一个可以独立运行的程序（bin 目录下的 rmiregistry），实际上 RMI service 也是一个 RMI 应用。

java RMI 原理详解 https://blog.csdn.net/xinghun_4/article/details/45787549

为了成功建立和 SQL 的 connection，首先我们需要配置数据库并在 rmiregistry 中注册，然后运行 server 和 client。

Statement: 定义了执行不含参数标记的 SQL 语句的方法。

PreparedStatement: 添加了设置输入参数的方法。

CallableStatement: 添加了从获取存储语句中输出参数的值的方法。

ResultSet

- Types

1. TYPE_FORWARD_ONLY
2. TYPE_SCROLL_INSENSITIVE
3. TYPE_SCROLL_SENSITIVE

- Concurrency

1. CONCUR_READ_ONLY
2. CONCUR_UPDATABLE

- Holdability

1. HOLD_CURSORS_OVER_COMMIT
2. CLOSE_CURSORS_AT_COMMIT

ResultSet 用法集锦 https://blog.csdn.net/leefang_cvic/article/details/90731507

用我们的属性去找 JMS factory，再去 lookup 一个 destination。也就是创建连接创建 connection，从 session 中创建 producer，循环 10 次，每次发一个消息。消费者前面都一样，获取连接、获取 destination，然后创建一个 consumer，不断在等，只要消息没收够，就不断地接收消息。每五秒判断一次有没有消息。

从代码中可以看到，无论是生产者还是消费者都意识不到对方存在，只知道要和消息中间件交互。底下的发布，前面都是一样的，在下面有 session.createTopic，在接收消息的时候会添加一个 selector，定义了一个匿名的 messagelistener 对象。我们就不用写一个 receive

去阻塞等待了。

实操：canvas 上的例子

第二个工程的例子是用 spring-boot 的 spring message 来写的。在收发的时候是一个 email 对象（一个简单的 java 对象和 getset），在 application 中需要 EnableJms，底下要注入一个 myFactory 对象，里面有一个 jackson。真正运行的时候，需要一个 JmsTemplate，这是 spring 创建的，会去拿上面定义的信息，连到消息中间件上。调用我们的 template 中的 convert and send

Receive 的时候怎么 receive 中，我们 @JmsListener，一旦有消息我们就调用 receiveMessage 方法。在收到消息之后，我们就输出得到的 Email 输出。

我们的作业是把下订单的转化为消息队列，所以又写了一个 controller，要求创建一个 Jmstemplate 中的 Convertandsend 发到前面去。我们再在写一个 receiver 即可。在前端页面发一个请求，controller 收到请求以后，把订单组成一个对象扔到消息队列里，外面有一个消息队列接收对象，然后放到数据库里。

卡夫卡是一个例子，它也是一个消息中间件。卡夫卡一定要一个 zookeeper 来支撑。卡夫卡实际上，在它眼里看到的点对点和发布话题是一样的，统一为了发送者和消费者。其他的没有什么差别。作业最简单的是参考第二个，用卡夫卡也可。

作业要求：蓝色是这节课新加的，消息中间件用哪个都可以。

2021/9/18

谈谈有状态和无状态

网页上有一个计数器标志着多少人访问过

如果这时候用户访问一个网站，服务器端肯定有一个 controller 来响应。计数器可以放在这个 controller 中，我们希望每个用户的请求都在同一个计数器上计数，这时候就应该设计成 sington，通常我们会把这个计数器写到数据库里。

我换了一个想法，如果有一个用户登陆上来，我们想告诉这个用户已经登录了多少时间了，eg: 40min，这种时间我们可以知道每个客户端都有一个，我们不能让客户端来做这件事情，因为很容易作假。这时候这个 timer（计时器）就应该和每个客户端匹配一个。这时候，我们就可以说只要这个 session 不过期，那么我们就保持维护这个 timer。又比如我们下订单开始计时，再下一个订单要创建新的计时器，这时候我们就发现这个 timer 比起之前的 timer 就不是 session 唯一的，而此时这个类的 scope 就应该是 prototype。所谓的状态就是，三个不同的客户端需要和其他人区别开的状态。然后我们再根据这个状态存活的周期，比如在一个 session 里或者一次请求里再去选择它的 scope。

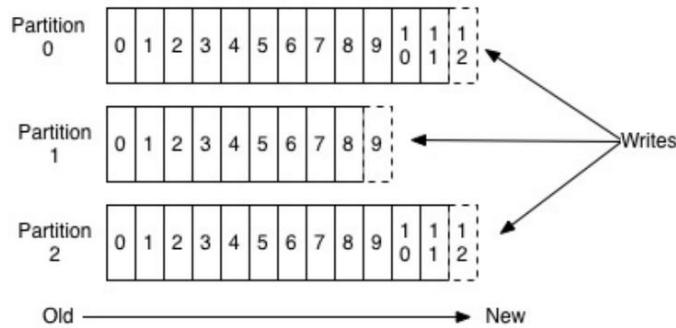
客户端发一个请求，controller 接收到以后，调用 service A，发到消息队列里。有另外一个 Service B 不断读这个消息队列中的消息。如果 serviceB 从 session 中获取消息，这样就是不对的，因为 service A 和 service B 应该是解耦的，看不见对方的。所以放进消息队列中以后就不应该再从 session 拿东西出来，因为放进消息队列以后，session 应该就没了。

Kafka

卡夫卡这个东西，其实点对点和发布预定模型没什么区别，所以卡夫卡就认为是一样的。在卡夫卡眼里，消息就是流一样的对象。Twitter 这种，肯定不是一台机器处理了所有请求，它支持集群化处理，在一个集群里去做。一旦做了分布式要就考虑到能不能扩展。加入新机器以后，新的 topic 能不能扩展。

如果要存到数据库里就可以使用 connector API，

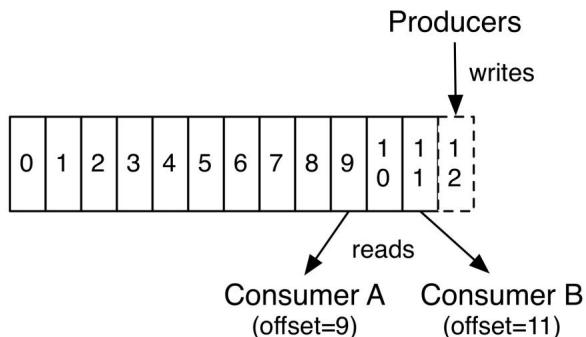
Anatomy of a Topic



卡夫卡是分区的（提高性能），比如大家对一条推文很感兴趣，对着一个话题发帖子。Topic 就是一个文件，或者是内存中的一块。大家排一个队肯定非常挤，分区以后就可以分流加快速度。并且如果我们对 topic 控制的好，比如体育新闻在 0 中，军事新闻 1 中，在读的时候，我们可以做到推送给感兴趣的人。虽然物理上是在三个机器上存了三个 partition，但是在逻辑上是一整份，我们在变成的时候意识不到在不同的机器上。

卡夫卡里面放进去的消息叫做记录，不断地往 topic 中追加，直到消息在 topic 中过期。消费者就通过在 topic 中根据偏移量读取消息（记录）。

The Kafka cluster durably persists all published records—**whether or not they have been consumed**—using a configurable retention period.



卡夫卡的实现是需要 zookeeper 来做支撑的。实际上 zookeeper 是一个集群的管理工具，在当前的场景之下，我们只需要知道要跑卡夫卡要先跑 zookeeper，因为卡夫卡强调这是一个集群。

配置卡夫卡（省略）

```

KafkaPro.java
public class KafkaPro {
    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.setProperty("transactional.id", "my-transactional-id");
        Producer<String, String> producer = null;
        try {
            producer = new KafkaProducer<String, String>(props, new StringSerializer(),
new StringSerializer());
            producer.initTransactions();
            producer.beginTransaction();
            for (int i = 0; i < 100; i++) {
                producer.send(new ProducerRecord<>("test", Integer.toString(i),
"Message" + Integer.toString(i)));
            }
            producer.commitTransaction();
        } catch (ProducerFencedException e) {
            producer.close();
        } catch (OutOfOrderSequenceException e) {
            producer.close();
        } catch (AuthorizationException e) {
            producer.close();
        } catch (KafkaException e) {
            producer.abortTransaction();
        }
        producer.close();
    }
}

```

先初始化一个事务出来，然后启动这个事务启动一百次。卡夫卡中 send 的都是 record 对象，要放 1.topic 名字 2.发的内容

一个消费者可以同时收多个 topic 中的东西。

实际上这时候，我们在 tomcat 的外面有一个第三方的消息中间件，仍然可以实现交互。思想就是异步通信，实现方式并没有差的很多。

Websocket

这节课讲讲 websocket，我们需要知道为什么需要 websocket。刚才我们的例子中，无论是异步通信还是同步的 service 同步操作数据库。我们不发请求是不能得到响应的，服务器比较被动，做不到没有请求的时候推送给。但是我们有这样的需求怎么办？

(**Websocket demo 展示**) 我们前端没有发请求，但是服务器一直在主动推送信息。实现的第二个方法 1.前端定时器写一秒一次，但是股票如果变化很慢一分钟变一次，那么 59

次就浪费了。所以我们很难控制好时间间隔，要么给后端带来负担，要么就会有刷新延迟。有大量客户端的时候这个压力就大了。

WebSocket 是 ws://，上节课讲的东西都是后台的 jms 怎么处理信息。现在的 ws 回到了前后台，但是不是 ajax 请求。RMI 是出现在卡夫卡进程和 tomcat 进程是怎么通信的。WebSocket 就是一个完全双工的（你能给我发，我也能给你发，双方完全对等）。用户体验就比较好。

WebSocket 在通信的时候先要握手，客户端先要和服务器握手握上，这样服务器就可以把消息推给客户端，每一个用户的浏览器都会刷新一下。WebSocket endpoint 就是一个服务器端的收发程序，在通讯的过程中，首先要建立一个通道。大家都可以通过这个连接来收发消息，可以在连接 open 的任意时间点发送消息。然后我们说握手过程稍微涉及一点加密。

- The client initiates the **handshake** by sending a request to a WebSocket endpoint using its URI.

- The handshake is compatible with existing HTTP-based infrastructure:
 - web servers interpret it as an HTTP connection upgrade request.

- An example handshake from a **client** looks like this:

```
GET /path/to/websocket/endpoint HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xqBt3ImNzJbYqRINxEF1kg==
Origin: http://localhost
Sec-WebSocket-Version: 13
```

在 header 头中添加 **Upgrade**，升级为 **websocket**，明文发送一个 **key** 过去。服务器端收到这个请求之后，尝试拿这个 **key**。比如客户端和服务器端都用证书对刚才的密钥做一下处理。用私钥去加密一下公钥，生成 **K'**，把 **K'** 加密，然后客户端用公钥解密一下，如果一致，就说明一样。***用私钥加密的东西只能用公钥解开，用公钥加密的东西只能用私钥解开。**或许 **hack** 有服务器的公钥，它能解开 **K** 得到原始的 **key**，但是 **hack** 没有服务器的私钥，所以不能通过客户端的校验。

客户端校验一致后就可以开始创建 **websocket** 连接。后面的代码就不再是 **http** 协议了，在前端用 **js** 脚本的 **endpoint** 的位置。

WebSocket endpoints are represented by URIs that have the following form:

- ws://host:port/path?query
- wss://host:port/path?query

- The **ws** scheme represents an unencrypted WebSocket connection, and
- the **wss** scheme represents an encrypted connection.
- The **port** component is optional:
 - the default port number is 80 for unencrypted connections and
 - 443 for encrypted connections.
- The **path** component indicates the location of an endpoint within a server.
- The **query** component is optional.

EchoEndpoint.java

```

public class EchoEndpoint extends Endpoint {
    @Override
    public void onOpen(final Session session, EndpointConfig config) {
        session.addMessageHandler(
            new MessageHandler.Whole<String>() {
                @Override
                public void onMessage(String msg) {
                    try {
                        session.getBasicRemote().sendText(msg);
                    } catch (IOException e) {}
                }
            });
    }
}

```

当有 ws 过来的时候，tomcat 会把这个连接包成一个 session 对象给你。在会话上可以注册一个消息处理器，这个处理器就有一个 onMessage 方法，就拿出来这个消息原封不动地发回去。

Programmatic Endpoints



- To deploy this programmatic endpoint, use the following code in your Java EE application:
`ServerEndpointConfig.Builder.create(EchoEndpoint.class, "/echo").build();`
- When you deploy your application, the endpoint is available at `ws://<host>:<port>/<application>/echo`;
 - for example, `ws://localhost:8080/echoapp/echo.`

需要部署到对应的路径是，在这个例子中，我们就可以通过 `ws://localhost:8080/echoapp/echo` 作为 endpoint 给到前端。

```

EchoEndpoint.java
@ServerEndpoint("/echo")
public class EchoEndpoint {
    @OnMessage
    public void onMessage(Session session, String msg) {
        try {
            session.getBasicRemote().sendText(msg);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

我们可以像这样用注释的方法去写，就比较直观了。

Annotation	Event	Example
OnOpen	Connection opened.	@OnOpen public void open(Session session, EndpointConfig conf) {}
OnMessage	Message received.	@OnMessage public void message (Session session, String msg) {}
OnError	Connection error.	@OnError public void error(Session session, Throwable error) {}
OnClose	Connection closed.	@OnClose public void close(Session session, CloseReason reason) {}

Send messages

```
@ServerEndpoint("/echoall")
public class EchoAllEndpoint {
    @OnMessage
    public void onMessage(Session session, String msg)
    {
        try {
            for (Session sess : session.getOpenSessions()) {
                if (sess.isOpen())
                    sess.getBasicRemote().sendText(msg);
            }
        } catch (IOException e) { ... }
    }
}
```

遍历这个 session，判断是不是还是一个 open 状态，如果 open 就把接收到的消息推送出去，就是一个广播机制。

```

Receive messages
@ServerEndpoint("/receive")
public class ReceiveEndpoint {
    @OnMessage
    public void textMessage(Session session, String msg)
    { System.out.println("Text message: " + msg); }

    @OnMessage
    public void binaryMessage(Session session, ByteBuffer msg)
    { System.out.println("Binary message: " + msg.toString()); }

    @OnMessage
    public void pongMessage(Session session, PongMessage msg)
    {
        System.out.println("Pong message: " +
                           msg.getApplicationData().toString());
    }
}

```

这里它允许 `onMessage` 的重载，允许根据不同的类型自动组装。可以是内置类型也可以是自定义类型。

```

@ServerEndpoint("/dukeetf")
public class ETFEndpoint {
    private static final Logger logger =
        Logger.getLogger("ETFEndpoint");
    static Queue<Session> queue = new ConcurrentLinkedQueue<>();

    public static void send(double price, int volume) {
        String msg = String.format("%.2f, %d", price, volume);
        try {
            for (Session session : queue) {
                session.getBasicRemote().sendText(msg);
                logger.log(Level.INFO, "Sent: {0}", msg);
            }
        } catch (IOException e) {
            logger.log(Level.INFO, e.toString());
        }
    }
}

```

17

考虑到这可能是一个多线程的程序，我们用了一个并发的链表形式实现的队列。底下 `send` 函数就是组成一个价格,数量的字符串，发送给 `queue` 里的每一个 `session`。因为这是一个向外发消息的 `server` 所以没有写 `onMessage`.

```

ETFListener.java
@WebListener
public class ETFListener implements ServletContextListener {
    private Timer timer = null;
    public void contextInitialized(ServletContextEvent event) {

```

```

        timer = new Timer(true);
        event.getServletContext().log("The Timer is started");
        timer.schedule(new ReportBean(evnet.getServletContext()), 0, 1000);
        event.getServletContext().log("The task is added");
    }
}

```

[ReportBean.java](#)

```

public class ReportBean extends TimerTask {
    private ServletContext context = null;
    private Random random = new Random();
    private double price = 100.0;
    private int volume = 300000;
    public ReportBean(ServletContext context) {
        this.context = context;
    }
    public void run() {
        context.log("Task started");
        price += 1.0 * (random.nextInt(100) - 50) / 100.0;
        volume += random.nextInt(5000) - 2500;
        ETFEndPoint.send(price, volume);
        context.log("Task ended");
    }
}

```

创建出 timer，并且每隔一秒创建一个新的 ReportBean，它是一个定时器任务，产生两个随机数，timer 调用 run 方法，每隔一秒就创建并发送。这里面潜在的问题就是 queue 里头多了以后，比如一百万个用户，这个循环执行的时间就特别长，有可能超过一秒还没执行完就会歇菜了。

这个复杂程序里用到了 encoder 和 decoder，收消息的时候可以解开组装。

Decoders

- Implement one of the following interfaces:
 - Decoder.Text<T> for text messages
 - Decoder.Binary<T> for binary messages

```

public class MessageTextDecoder implements Decoder.Text<Message> {
    @Override
    public void init(EndpointConfig ec) { }

    @Override
    public void destroy() { }

    @Override
    public String decode(String string) throws DecodeException {
        // Read message...
        if ( /* message is an A message */ ) return new MessageA(...);
        else if ( /* message is a B message */ ) return new MessageB(...);
    }

    @Override
    public boolean willDecode(String string) {
        // Determine if the message can be converted into either a
        // MessageA object or a MessageB object...
        return canDecode();
    }
}

```

Encoders

- Implement one of the following interfaces:
 - Encoder.Text<T> for text messages
 - Encoder.Binary<T> for binary messages

```

public class MessageATextEncoder implements Encoder.Text<MessageA> {
    @Override
    public void init(EndpointConfig ec) { }

    @Override
    public void destroy() { }

    @Override
    public String encode(MessageA msgA) throws EncodeException {
        // Access msgA's properties and convert to JSON text...
        return msgAJsonString;
    }
}

```

也就是给一个 java 对象转字符串，获得文本再转 java 对象。

这个聊天室代码该怎么写，先说前端（此处主要看 ppt）

2021/9/23

事务管理

这节课讲事务管理。什么是事务？然后是怎么用事务、管理事务，再谈谈涉及到多个数据库的情况。

我们需要知道为什么需要事务管理和怎么去设计事务。

整个在 JAVA 企业版规范中，有 java 事务 api，关键问题就是什么是事务。

我们举个例子

- Pseudocode:

```
begin transaction
    debit checking account
    credit savings account
    update history log
commit transaction
```

- A **transaction** is often defined as an indivisible unit of work
 - Either all or none of the three steps must complete.

- A transaction can end in two ways:

- with a commit or with a rollback.

比如我们要去做一次转账操作，先把第一个用户的钱扣掉，先扣 A 的 100 元，然后再把这 100 元加到 B 的账户上，再记录一个日志。

比如我们写这个转账函数

```
Transfer(){
    Withdraw();
    Deposit();
    Log();
}
```

我们要考虑两个问题

问题 1：先扣 A 的 100，再加给 B 100 元。如果扣完钱以后，因为数据连接断掉了或者 B 的数据锁死了。我们要回到 A 被减掉 100 元钱之前的状态，我们要恢复原先的情况。也就是在用户眼里看到 transfer 是一个原子性的操作，用户眼里看到的要么成功要么失败，如果执行到一半失败了需要恢复到原先的场景。

成功的情况下，我们就是 commit 这个操作；失败的情况下，我们需要 rollback。

我们现在很难保证转账这个操作是原子性的。

问题 2：如果现在 A 在给 B 转账 100，C 也要给 B 转 100，如果两个成功 B 应该拿到 200 元。如果 A 和 C 同时读 B 的 0 元余额，然后同时把 100 写回 B 的余额，这就会有问题。如果串行执行，那么就效率比较差。所以我们需要保证事务的隔离性。我们要把同时在操作的事务隔离开，大量的人对同一本书下单我们怎么样保证不能乱。

如果我们想实现它，其他它们都是在同时操作数据库，一个最简单的办法就是所有对数据库的操作全部放在数据库的 Cache 里，如果第二个操作没有成功，那么就把对应事务的 Cache 释放掉，那么硬盘上的数据库没有收到更高。如果事务能够 commit，那么内存中的数据就一次性写到硬盘上。

Isolation 也是一样的，在内存中加一个锁，在访问的时候先看一下内存中能不能写，如果不能写只能读的话，那么就不能修改内存中的值。

也就是我们把事务的中间状态用内存中的 cache 解决下来。这就是内存型事务管理器，还有前面两节课我们讲的 jms。比如说有十条消息要群收的，如果中间有一条消费失败了，应该把前面被收掉的再放回到 jms 中。

TM 包含 DBMS,JMS,Java Mail

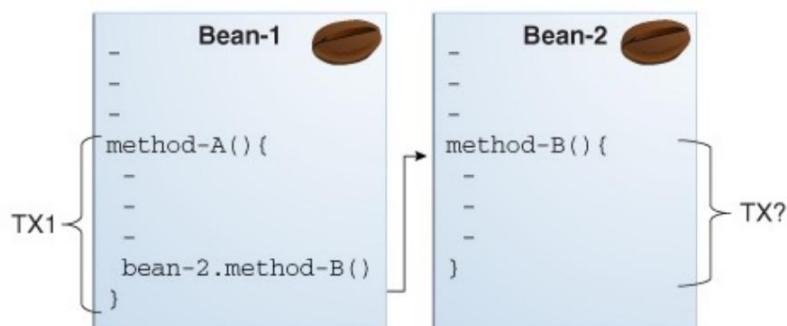
只要是事务性资源管理器，都有这些操作。我们需要知道事务的边界在哪里。事务从代码哪里开始执行到哪里，哪一系列的操作需要是原子的。Isolation 是不需要我们写代码去实现的。如果我们要去支持这件事，我们需要一个很大的缓存，在此之中保留事务的中间结果，事务成功就写到硬盘里，事务失败就抹去内存。可能会带来一个问题：如果操作很多，比如说转账的时候要求余额中大于 5 元，C 要同时给 10 个人转账。在缓存中执行的时候，并不是每次都是要写到硬盘中。在给 10 个人转账的时候，每个人转 100，在缓存中做成余额为 -100，此时就是不能写进去的。

所以事务是强调了一致性的，如果在事物上有约束，最后尝试写入硬盘的时候必须要检查约束，如果违反了约束，也不能写到硬盘中。

同样的，在缓存中 commit 的时候，如果写硬盘写失败了怎么办？这要求数据一旦提交，必须要求它 commit 的数据是持久化的。这四个属性就是事务的四个属性（A,C,I,D）

In an enterprise bean with container-managed transaction demarcation,

- the EJB container sets the boundaries of the transactions.



我们先解决原子性的问题，method-A 调用 Bean-2 上的 method-B。我们假设说程序都是多线程执行的，Bean-1 的 method-A 调用 Bean-2 的 method-B。这个线程已经开始了一个事务，调用到 method-B 以后，这会成为事务的一部分还是开启一个新的事务呢？这就是在告诉我们的 tomcat 容器，我们的事务边界在那里。怎么做呢？

NotSupportedException

- Invoking a method on an EJB with this transaction attribute suspends the transaction until the method is completed.



最常用的是用声明式的方式:

T 调用 W 和 D 方法

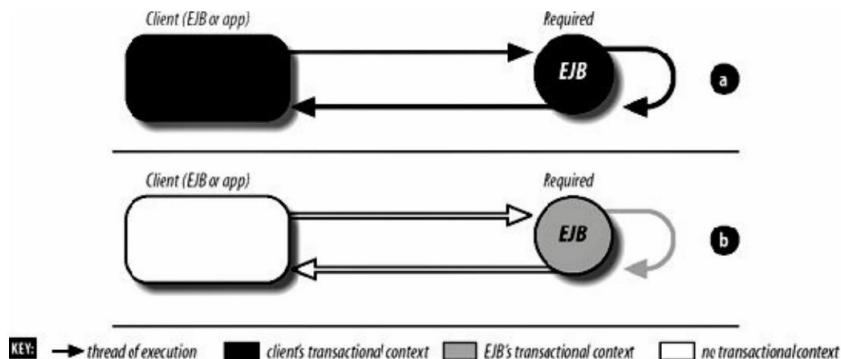
1.

默认属性的三个方法都是 required

也就是 T,W,D: required

Required

- This attribute means that the enterprise bean method must be invoked within the scope of a transaction.



@Transactional

```
public void book(String... persons) {  
    for (String person : persons) {  
        logger.info("Booking " + person + " in a seat...");  
        jdbcTemplate.update("insert into BOOKINGS(FIRST_NAME) values (?)", person);  
    }  
}
```

直接在 annotation 上添加@Transactional 即可。

这三个方法都是 required 的情况，客户端发一个请求过来，controller 截获之后开一个新的线程调用 T 函数，然后发现 T 是 required。它的逻辑就是如果线程上没有事务，就开一个新的事务 T1，添加 T 属于 T1。T 调用到 W 后，W 也是 required，相当于上面这种情况，我们线程上已经有事务 T1 了，那么我们把 W 也加入到 T1，当 W 结束之后，尝试提交事务 T1，但是发现 T1 事务不是 W 开启的，那么就什么都不做。然后把 D 加入到 T1，执行完尝试提交，也不成功。T 结束后，容器尝试提交 T1，发现事务 T1 就是 T 函数开启的，那么 T1 就被提交掉了。此时我们就发现 T, W, D 都在一个事务中执行。其中任何一个地方失败都会导致 T1

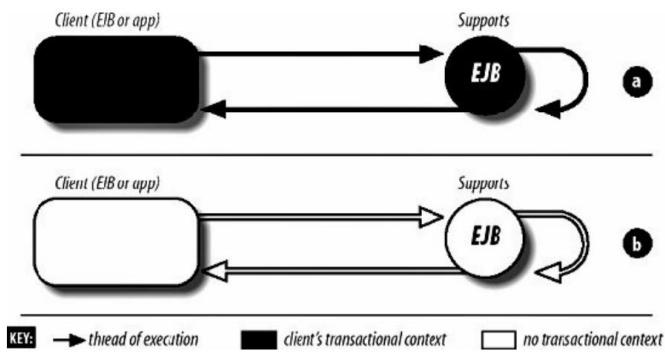
事务 rollback。

Required 的事务就是，如果线程上没有关联事务，就开一个新事务，如果线程上已经有了事务，就加入到这个事务

3. Support

Supports

- This attribute means that the enterprise bean method will be included in the transaction scope if it is invoked within a transaction.



如果 T 是 support, W 和 D 都是 required

Controller 调用 T 后，就在非事务状态下，W 就必须开一个新事务 t1，W 执行完尝试提交发现可以提交。执行 D 的时候会再开一个事务叫做 t2，执行完再提交。所以如果 D 失败了不会导致 W 回滚。

4. requiresNew

T:require, W: requiresNew

如果当前有事务的时候，调用 requiresNew 时，会挂起 T 开启的 T1 事务，开一个新的 T2 事务，执行完提交 T2，然后再恢复 T1，然后我们再把 D 加入到 T1 事务。

5. NotSupported

压根就不想加入到事务中。

T:required W:notsupported D:required

执行到 W 的时候会把 T1 挂起，在非事务下进行执行

6. Mandatory

必须在事务下执行，否则报错。

如果 T 改为 Mandatory，因为一开始调用的时候没有事务，就直接报错了。

7. Never

如果线程上有事务了，那么就直接抛出异常。

我们改了事务的属性，没有修改代码。它提供了各种各样的手段来作为事务的边界。

比如我们现在 TW D log 四个函数。

比如文件系统记录 log 中，失败的概率比较大，如果记录 log 导致整个事务失败，这就不太合理，我们就可以把 log 函数设置为 requiresNew 或者 NotSupported（记不成就算了，不要影响已经写成功的数据库的事务）

Transaction Attribute	Client's Transaction	Business Method's Transaction
Required	None	T2
Required	T1	T1
RequiresNew	None	T2
RequiresNew	T1	T2
Mandatory	None	Error
Mandatory	T1	T1
NotSupported	None	None
NotSupported	T1	None
Supports	None	None
Supports	T1	T1
Never	None	None
Never	T1	Error

A 方法的有没有事务

第三列:Bean-2 方法上有没有事务

Transaction attributes are specified by

- decorating the enterprise bean class or method with a `javax.ejb.TransactionAttribute` annotation
- and setting it to one of the `javax.ejb.TransactionAttributeType` constants.

Transaction Attribute	TransactionAttributeType Constant
Required	<code>TransactionAttributeType.REQUIRED</code>
RequiresNew	<code>TransactionAttributeType.REQUIRES_NEW</code>
Mandatory	<code>TransactionAttributeType.MANDATORY</code>
NotSupported	<code>TransactionAttributeType.NOT_SUPPORTED</code>
Supports	<code>TransactionAttributeType.SUPPORTS</code>
Never	<code>TransactionAttributeType.NEVER</code>

The following code snippet demonstrates how to use the `@TransactionAttribute` annotation:

```

@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
    ...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}
    public void thirdMethod() {...}
    public void fourthMethod() {...}
}

```

我们具体写代码的时候可以通过 annotation 来设置。这解决的就是事务的边界划分问题。为了说明这个问题，我们通过以下例子。

再强调一下，实现事务的回滚机制是靠操作的对象 JMS,DBMS 等自带事务处理的机制才能用。

如果我们在程序中定义一个 `count = 0`, 每次调用 `transfer` 的时候 `count ++`一次。如果某次失败导致这次事务回滚了, 这个 `count` 的值能回去吗? 回不去, 为什么数据库中能回去呢? 因为我们告诉那块 `cache` 的内存可以释放掉。而 `count` 是 Tomcat 中的数据, 是回不去的。如果我们要让 `count` 也回去, 我们要实现 `sessionsynchronization` 方法。

The `SessionSynchronization` interface, which is optional, allows stateful session bean instances to receive transaction synchronization notifications.

- The container invokes the `SessionSynchronization` methods (`afterBegin`, `beforeCompletion`, and `afterCompletion`) at each of the main stages of a transaction.

```
afterBegin:  
    oldvalue = value;  
  
afterCompletion(boolean b):  
    if (!b)  
        value = oldvalue;
```

可以在事务开始后和事务结束后调用, 如果成功了就`++`, 或者判断如果失败了就回滚。这个例子告诉我们, 自己写的变量不是 `transactional manager` 管理的范畴, 当然不能 `rollback`, 需要我们自己写代码 `rollback`。

Eg:`bookservice` 在操作数据库, 它有一个`@transactional` 的 `book` 方法, 如果不写属性的话, 默认就是 `required` 方法。在 `bookings` 插入这些人的名字。数据库只有一个 `id` 和一个 `firstname`。一开始传入三个长度小于 5 的名字是没有问题的。传入 6 个字符的名字的时候, 循环第二次插入的时候失败, 整个事务就失败了, 需要回滚, 会保证循环前面成功的 `insert` 也没有写进数据库。

隔离性的四个级别。多个事务操作数据库的时候, 有四种隔离机制。

`@Transactional isolation`

事务隔离级别	说明
<code>@Transactional(isolation = Isolation.READ_UNCOMMITTED)</code>	读取未提交数据(会出现脏读, 不可重复读), 基本不使用
<code>@Transactional(isolation = Isolation.READ_COMMITTED)</code> (SQLSERVER默认)	读取已提交数据(会出现不可重复读和幻读)
<code>@Transactional(isolation = Isolation.REPEATABLE_READ)</code>	可重复读(会出现幻读)
<code>@Transactional(isolation = Isolation.SERIALIZABLE)</code>	串行化

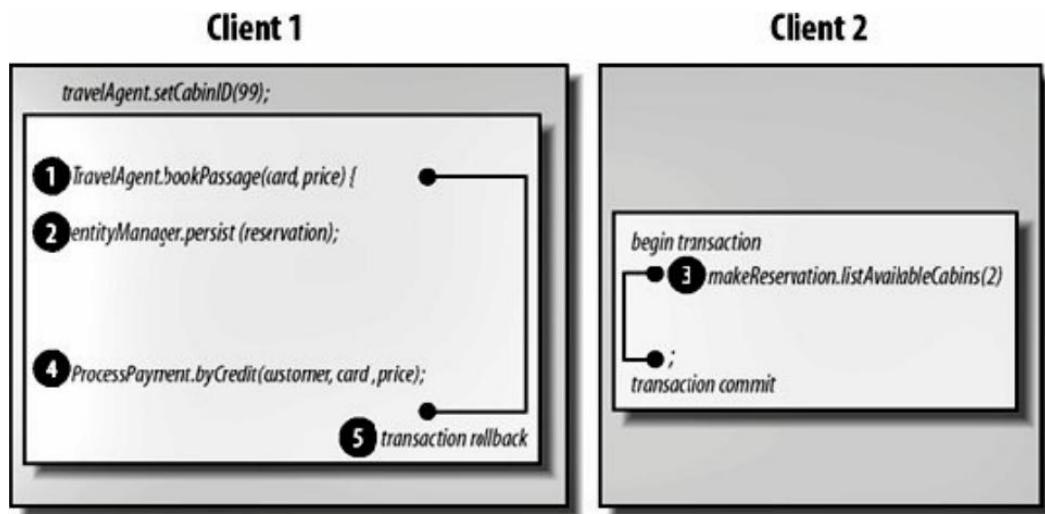
在性能和可能产生的问题之间需要找一个平衡点。每一种都会引起一些问题。假设刚才 `book` 这个例子:

Let's think about two separate client applications accessing the same data specifically.

```
public List listAvailableCabins(int bedCount) {  
    Query query = entityManager.createQuery(  
        "SELECT name FROM Cabin c  
        WHERE c.ship = :ship AND  
        c.bedCount = :beds AND NOT ANY (  
            SELECT cabin from Reservation res  
            WHERE res.cruise = :cruise);  
    query.setParameter("ship", cruise.getShip());  
    query.setParameter("beds", bedCount);  
    query.setParameter("cruise", cruise);  
    return query.getResultList();  
}
```

For this example, assume that both methods have a transaction attribute of **Required**.

比如列出当前可用的舱位，传进来的是希望买几张票。就可以列出可用买的票。假如两个客户端同时竞争去买票。



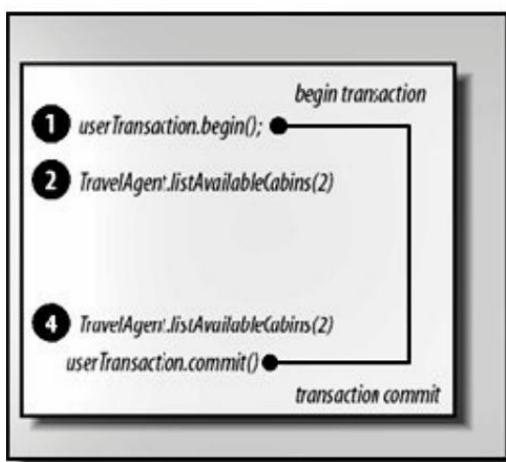
客户端 1：买两张票

客户端 2：我也想买两张票

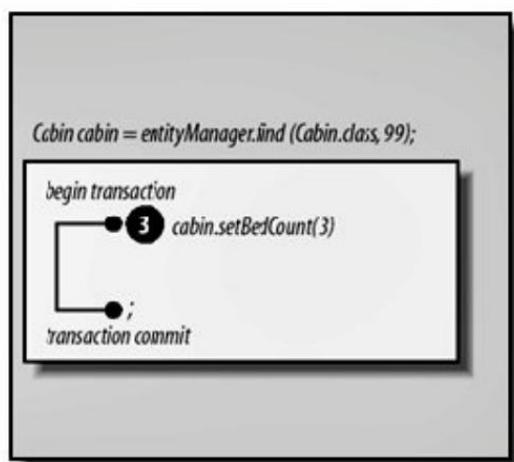
客户端 1 的事务执行的比较久，客户端 1 想要两张票，应该从数据库中扣掉两张票给 1，只剩了一张。客户端 2 就读不到票，这就是客户端 1 执行事务的中间状态被客户端 2 看到了，这就是脏读（读到的是一个脏数据）。这个的本质就是没有 `commit` 的数据被别人读到了。

第二个问题：

Client 1



Client 2



执行的慢的事务被速度快的事务抢到票了。在客户端 1 中读的数据被客户端 2 改掉了。后面在要用的时候，这个数据已经被客户端 2 改变了。这就是不可重复读问题。如果我们读的时候，锁死票量，只能读不能写，这就比较好。

第三种情况：在执行过程中反复读数据应该是一直的，在锁死写的情况下。但是我们可以改写数据库中的其他数据，这就是幻读问题。数据库中多出了一些符合我要求的东西。因为这读了一些我们需要写的东西，而没有锁其他的记录。这种情况下我们可以把整个表全部锁死来解决。

所以要实现事务的隔离就是为了加锁。

Database Locks



- **Read locks**

- Read locks prevent other transactions from changing data read during a transaction until the transaction ends, thus preventing **nonrepeatable** reads.
 - Other transactions can read the data but not write to it. The current transaction is **also** prohibited from making changes.

- **Write locks**

- Write locks are used for updates. A write lock prevents other transactions from changing the data until the current transaction is **complete** but allows **dirty reads** by other transactions and by the current transaction itself.
 - In other words, the transaction can read its own **uncommitted** changes.

读锁：防止其他事务来改写你正在锁死的数据，可以解决“不可重复读”的问题，可以防止其他事务在执行过程中修改读走的数据，这个锁的力度就比较强。

写锁：我要写数据的时候加个锁，防止其他人来写这个数据，但是其他人可以读走正在写的数据。可以读到加 100 的中间状态，如果回滚掉了，就会触发其他人读到了没有 **commit** 的数据，这就是脏读问题。

独占写锁：不能读到没有提交的数据，只能读到事务执行之前的数据或者提交后的数据。

Snapshot：所有事务操作都在快照上操作，操作完写会数据库中，可以解决所有问题但是性能比较差。

- **Read Uncommitted**

- The transaction can read uncommitted data (i.e., data changed by a different transaction that is still in progress).
 - Dirty reads, nonrepeatable reads, and phantom reads can occur.
 - Bean methods with this isolation level can read uncommitted changes.

- **Read Committed**

- The transaction cannot read uncommitted data; data that is being changed by a different transaction cannot be read.
 - Dirty reads are prevented; nonrepeatable reads and phantom reads can occur.
 - Bean methods with this isolation level cannot read uncommitted data.

- **Repeatable Read**
 - The transaction cannot change data that is being read by a different transaction.
 - Dirty reads and nonrepeatable reads are prevented; phantom reads can occur.
 - Bean methods with this isolation level have the same restrictions as those in the Read Committed level and can execute only repeatable reads.
- **Serializable**
 - The transaction has exclusive read and update privileges; different transactions can neither read nor write to the same data.
 - Dirty reads, nonrepeatable reads, and phantom reads are prevented.
 - This isolation level is the most restrictive.

Read uncommitted: 可以提交还没有 commit 的事务中的中间数据，可以认为事务之间就没有隔离性。

Read committed: 没有提交的中间状态是其他事务读不走的，给 B 加了 100 块钱，其他事务读的时候，只要这个转账还没有 commit，读到的 B 的余额不会是 100。真正提交以后，如果读走了 B 的余额还是 0 怎么办？C 也想给 B 转账的时候，这个事务我们可以比较 C 读走的 B 的余额和当前数据库的 B 的余额是否一样，如果不一样可以回滚 C 的这次转账事务，重新执行。

Repeated read: 可以重复读，在读数据库表的时候，把读走的数据锁死，别人不能对他进行任何修改，在执行事务的任何阶段读到都是相同的值。

串行化（serializable）： 别人改写了其他数据，满足了查询条件的幻读问题，序列化就意味着整个这张表锁死，任何人不能做写操作，解决了幻读问题。

这四个隔离级别：1.没有任何隔离 2.解决了脏读 3.解决了不可重复读 4.解决了幻读

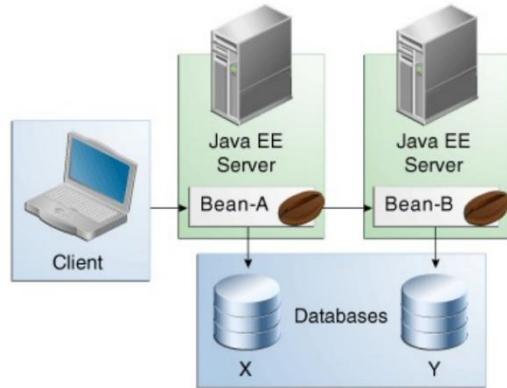
You to specify the transaction isolation level using the database's API.

For example:

```
DataSource source = (javax.sql.DataSource)
                    jndiCtxt.lookup("java:comp/env/jdbc/titanDB");
Connection con = source.getConnection();
con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

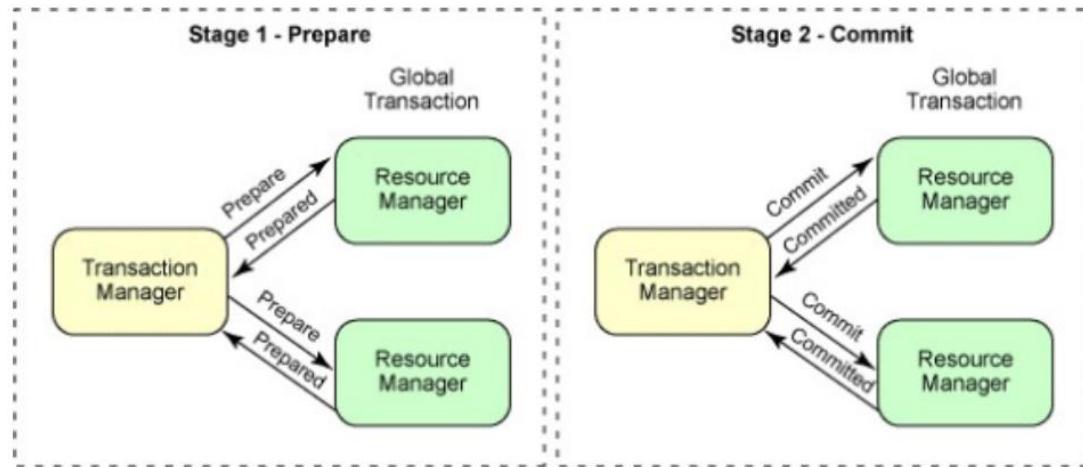
写代码的时候可以指定隔离级别，**serializable** 的时候性能就会非常差，但确实对数据的一致性和完整性的保护非常好，所以就需要做权衡。仔细考虑一下我们写的代码有没有可能脏读或者不可重复读的问题，重要的不是代码，而是我们的权衡和选择。

到目前为止，我们的操作都是在写数据库，JMS，如果转账操作是从建设银行转账到工商银行中，这就涉及到两个数据库。一个数据库的事务中间状态都在内存 cache 中，而两个数据库中，扣钱事务可以提交，另一个数据库存钱事务说不能提交，而它们数据库是不通的，这该怎么办呢？就需要 tomcat 这样的应用服务器来通知两个数据库做什么操作，返回结果是什么，tomcat 来决定是提交还是不提交。这就是分布式事务，也不一定要是数据库也有可能是 JMS 消息队列。



操作的时候只要涉及两个数据库，就是分布式事务。需要考虑一系列的数据约束。一个数据库能提交另一个数据库不一定能提交。

所以分布式数据库分为两阶段（两阶段一致性协议）：

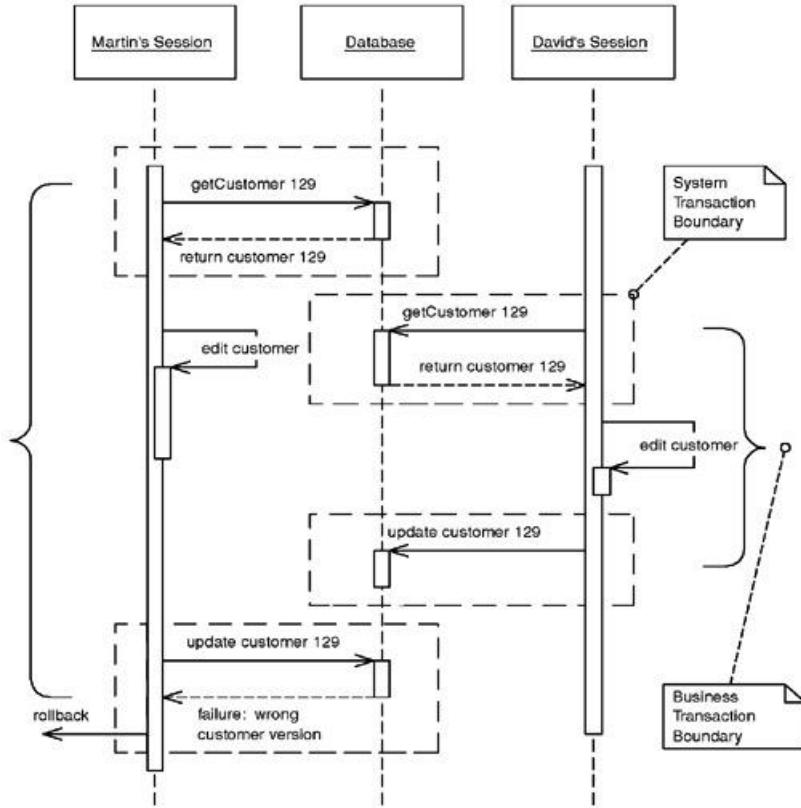


1. 询问两个数据库能否提交，通常采用一票否决策略。
2. Tomcat 告诉各个数据库去提交或者回滚。

尽管两个数据源不知道对方，但是可以做出一致性的操作。在没有收到 tomcat 的回复的时候(网络原因超时的情况下)，那就需要采用一个启发式算法（瞎猜一个）。

无论是几阶段，我们避免不了这种启发式动作猜一下的情况。这就是分布式里面我们讲的怎么让程序能跑。代码怎么实现呢？代码里面访问两个数据库的话，tomcat 就会自己我们去做这个分布式事务的事情。

在 A 和 C 并发操作 B 的账户做转账操作，该怎么设计数据库的表，应该怎么去锁。



中间数据库有一个 customer，中间 martin 编辑客户的状态到最后才提交。David 提交了事务。Martin 在写会数据库的就会抛异常，因为和读走的数据不一样。

乐观锁和悲观锁

乐观锁：认为这种冲突的概率比较小，为了这种事情升级隔离机制效率太低，干脆不锁，大家可以并发的读。我们要在每一次读走数据再写会的时候，要去检查读走的数据和当前数据库中的数据是不是被改写过。我们可以在每一条数据后加一个字段作为版本号，改写过就递增，加时间戳也可以。在这个场景下，它的逻辑就变成版本号和时间戳的值从数据库中读过来。A 读走版本号为 0 的 B 的余额，C 读走的 B 的余额版本号为 0.A 操作完了，B 的余额的版本号变成了 1.C 尝试写回的时候发现读走的版本号 0 和当前数据库中 B 的余额的版本号 1 不一致了，我们就知道 C 在读走 B 的数据之后，B 的余额又被更改过了，所以要回滚。

Pessimistic Offline Lock prevents conflicts by avoiding them altogether.

- It forces a business transaction to acquire a lock on a piece of data before it starts to use it, so that, most of the time, once you begin a business transaction you can be pretty sure you'll complete it without being bounced by concurrency control.

You implement Pessimistic Offline Lock in three phases:

- determining what type of locks you need,
- building a lock manager,
- and defining procedures for a business transaction to use locks.

Lock types:

- exclusive write lock
- exclusive read lock
- read/write lock
 - Read and write locks are mutually exclusive.
 - Concurrent read locks are acceptable.

In choosing the correct lock type think about maximizing system concurrency, meeting business needs, and minimizing code complexity.

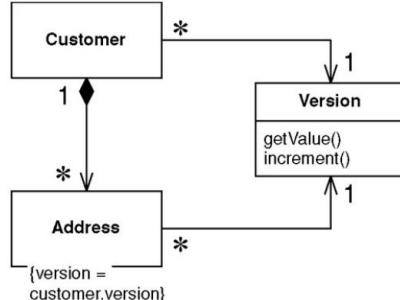
- Also keep in mind that the locking strategy must be understood by domain modelers and analysts.

悲观锁：不允许并发的读。对应的就是提升事务的隔离级别。

我们再谈一个 `customer`。一个人有多个地址，存在另一张表里。我们在访问 `customer` 的时候，我们不希望地址表中对应的条目被改掉。我们该怎么样实现这样一个粗粒度的锁呢？也分为乐观锁和悲观锁。

With **Optimistic Offline Lock**, having each item in a group share a version creates the single point of contention, which means sharing the **same** version, not an **equal** version.

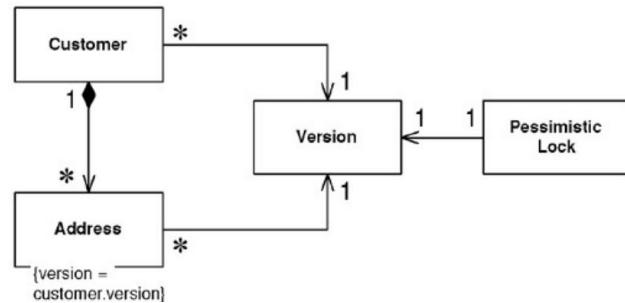
- Incrementing this version will lock the entire group with a shared lock.
- Set up your model to point every member of the group at the shared version and you have certainly minimized the path to the point of contention.



乐观锁：不太会出现两个人同时改写 `customer` 和对应 `address` 的情况下，这种情况下是不加锁的，但是我们要检查，在提交的时候判断出先后，让后来的人重新做这个事务。我们可以在数据库中加一个对象，让 `customer` 和 `address` 都引用到。再根据这个共用类的 `version` 对象来判断谁是先后。

A shared **Pessimistic Offline Lock** requires that each member of the group share some sort of lockable token, on which it must then be acquired.

- As Pessimistic Offline Lock is often used as a complement to Optimistic Offline Lock, a shared version object makes an excellent candidate for the lockable token role.



悲观锁：真的在数据库中有一个 `version` 表，然后在 `customer` 和 `address` 去引用这个版本号，当有一段操作 `address` 或者 `customer` 的时候，都会根据外键关联对 `version` 的一个记录加锁。这时候另外事务就获取不到这个对应 `customer` 和 `address` 的锁了。

整个 `transaction` 要实现的就是原子性（完整性和一致性）和隔离性（让程序的性能变得好，每一个事务都在操作数据库的时候，操作不能乱）。这些实际上都是我们的事务性的数据管理器在帮我们做这件事情。我们只是写代码来利用这种事务的能力。

2021/9/27

Java 多线程

在 java 按照自己的想法编写多线程程序是怎么样的，选定 `scope` 作为单例，如果计数的话，一个对象服务多个用户必然是多线程。这时候计数器会不会乱。

多线程会不会出现事务隔离级别不同而出现导致冲突呢？事务隔离级别怎么看呢，通过脚本录制的方式做压力测试。

多线程为什么在 java 中比较重要，因为线程对应的概念是进程，一个 java 虚拟机是一个进程，整个 tomcat 和部署进去的应用是一个进程。进程是有独立的存储空间的，一个进程要和另一个进程通信，是没有办法访问到对方的内存的，需要提供远程方法调用。线程是大家在共享同一个进程中的内存，如果没有做特殊的隔离的话。线程和进程很大的区别里，同一个进程里的线程互相之间通信是直接访问的，所以速度更快，但是也更容易出错。多线程的话就是分时调用的。

写线程的两种方法：

1. 实现 `Runnable` 的接口，只有一个方法是 `run`,

- *Provide a Runnable object.*

- The `Runnable` interface defines a single method, `run`, meant to contain the code executed in the thread. The `Runnable` object is passed to the `Thread` constructor, as in the `HelloRunnable` example:

```
public class HelloRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

它的意义就在于一个线程在跑 `run`, 一个线程在跑 `run` 这个东西。它确实是再起了一个线程去跑

2. 直接扩展 `Thread` 类

- *Subclass `Thread`.*

- The `Thread` class itself implements `Runnable`, though its `run` method does nothing. An application can subclass `Thread`, providing its own implementation of `run`, as in the `HelloThread` example:

```
public class HelloThread extends Thread {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

Java 叫单根继承，也就是它的父类只能有一个，如果我们实现的类比较复杂，那就不能从 `Thread` 继承下来。所以我们比较鼓励 `runnable object`，把单根继承的单根让出来允许我们去做更多的事情。

`Thread.sleep` causes the current thread to suspend execution for a specified period.

```
public class SleepMessages {  
    public static void main(String args[]) throws  
        InterruptedException  
{  
    String importantInfo[] = {  
        "Mares eat oats", "Does eat oats",  
        "Little lambs eat ivy", "A kid will eat ivy too"  
    };  
  
    for (int i = 0; i < importantInfo.length; i++) {  
        //Pause for 4 seconds  
        Thread.sleep(4000);  
        //Print a message  
        System.out.println(importantInfo[i]);  
    }  
}
```

这里面写了跑到里面去睡 4000 毫秒，睡 4 秒意味着阻塞在这里，每隔四秒输出这个东西。我要知道这个 4000 是不准的，它取决于不同的环境。*在严格的定时任务不能使用 `sleep` 方法。

What if a thread goes a long time without invoking a method that throws `InterruptedException`?

- Then it must periodically invoke `Thread.interrupted`, which returns true if an interrupt has been received. For example:

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

- In more complex applications, it might make more sense to throw an `InterruptedException`:

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

线程在执行过程中可以被强行中断，如果中断的话会抛出中断异常。中断和 `sleep` 不一样，中断意味着这个线程没有办法再执行了。我们应该去做判断 `Thread.interrupted()` 来判断这个线程有没有被中断掉。通常我们发现中断以后抛出异常或者什么都不做即可。

The `join` method allows one thread to wait for the completion of another.

- If `t` is a `Thread` object whose thread is currently executing,
`t.join();`
- causes the current thread to pause execution until `t`'s thread terminates.
- Overloads of `join` allow the programmer to specify a waiting period. However, as with `sleep`, `join` is dependent on the OS for timing, so you should not assume that `join` will wait exactly as long as you specify.
- Like `sleep`, `join` responds to an interrupt by exiting with an `InterruptedException`.

`t.join`, 主线程挂起来执行 `t`, 等待 `t` 再挂起的时候就再恢复当前进程。为什么要这么做呢？比如双方产生了资源的争用。`join` 就是让出自己占用的资源。

然后我们可以看一个稍微有一点用的 `thread`, 在这个 `SimpleThread` 中有一个 `main` 函数, 它会创建一个 `Runnable` 的线程去执行代码逻辑。

```
public class SimpleThreads {
    // Display a message, preceded by
    // the name of the current thread
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }

    private static class MessageLoop implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Mares eat oats", "Does eat oats", "Little lambs eat ivy",
                "A kid will eat ivy too"
            };
            try {
                for (int i = 0; i < importantInfo.length; i++) {
                    // Pause for 4 seconds
                    Thread.sleep(4000);
                    // Print a message
                    threadMessage(importantInfo[i]);
                }
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }

    public static void main(String args[]) throws InterruptedException {
        // Delay, in milliseconds before
        // we interrupt MessageLoop
        // thread (default one hour).
        long patience = 1000 * 60 * 60;

        // If command line argument
        // present, gives patience
        // in seconds.
        if (args.length > 0) {
            try {
                patience = Long.parseLong(args[0]) * 1000;
            } catch (NumberFormatException e) {
                System.err.println("Argument must be an integer.");
                System.exit(1);
            }
        }
    }
}
```

```

threadMessage("Starting MessageLoop thread");
long startTime = System.currentTimeMillis();
Thread t = new Thread(new MessageLoop());
t.start();

threadMessage("Waiting for MessageLoop thread to finish");
// loop until MessageLoop
// thread exits
while (t.isAlive()) {
    threadMessage("Still waiting...");
    // Wait maximum of 1 second
    // for MessageLoop thread
    // to finish.
    t.join(1000);
    if (((System.currentTimeMillis() - startTime) > patience)
        && t.isAlive()) {
        threadMessage("Tired of waiting!");
        t.interrupt();
        // Shouldn't be long now
        // -- wait indefinitely
        t.join();
    }
}
threadMessage("Finally!");
}

```

14

把 runnable 的 MessageLoop 传递给 thread 类的 t，并且跑起来 t。只要这个 message Loop 还活着，那么主线程就输出 still waiting。主线程就把自己的资源挂起，让 t 执行一秒。
(t.join(1000))

如果时间到了，我们就中断 t，然后再使用 t.join 把控制权再给到 t，让它抛出异常，也就是说出 “i wasn't done!”，最后就结束循环。这就是两个线程互相之间的协作。我们就可以根据这个例子看到线程之间是怎么样协作的。

我们要讲一个问题：多线程程序中最重要的问题就是它们互相之间在协作的时候如果出现了生产者和消费者之间的关系或者在争夺同一个问题的时候，可能导致互相之间产生干扰。内存之间的竞争还可以导致死锁、饥饿、活锁。

```

Consider a simple class called Counter
class Counter {
    private int c = 0;
    public void increment() { c++; }
    public void decrement() { c--; }
    public int value() { return c; }
}

```

如果在一个多线程程序中，如果有多个线程来访问它，比如线程 A 和 B 同时得到 `c=0`，A 加一写回 1 而 B 减一写回-1。我们该怎么去处理呢？在 A 先调用加一这个动作的时候，应该让 B 不能调用减一、加一这个动作。也就是在调用 `increment` 的时候，我们希望别的人不能调用 `Increment/decrement`，希望操作是串行的。现在就是内存的一致性产生了错误，我们要在操作之间建立起 Happens-before 的关系。它要确保一个动作在另一个动作执行之前已经完成了。

```

public class UnSafeMultipleThreads<Static, C> {
    static void threadMessage(String message) {
        String threadName =
            Thread.currentThread().getName();
        System.out.format("%s: %s%n",
            threadName,
            message);
    }

    Counter c = new Counter();

    private class CounterLoop implements Runnable {
        public void run() {
            try {
                for (int i = 0; i < 100; i++) {
                    // Pause for 1 seconds
                    Thread.sleep(1000);
                    // Print a message
                    threadMessage(String.valueOf(c.value()));
                    c.increment();
                }
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }
}

public static void main(String args[])
    throws InterruptedException {
    threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();

    UnSafeMultipleThreads s = new UnSafeMultipleThreads();
    UnSafeMultipleThreads.CounterLoop c1 = s.new CounterLoop();
    c1.setSleep(1000);
    Thread t1 = new Thread(c1);
    t1.start();
    UnSafeMultipleThreads.CounterLoop c2 = s.new CounterLoop();
    c2.setSleep(1000);
    Thread t2 = new Thread(c2);
    t2.start();

    threadMessage("Waiting for MessageLoop thread to finish");
}

```

它要输出线程的名字和消息，我们用上述的 `counter` 去创建对象 `c`，线程有 `sleep` 这个参数，`run` 的时候在循环一百次，每次 `sleep` 指定时间后调用在计数器上调用 `increment`。

```

Thread-0: 10
Thread-1: 10
Thread-0: 12
Thread-1: 13
Thread-0: 14
Thread-1: 15
Thread-0: 16
Thread-1: 17
Thread-0: 18
Thread-1: 19
Thread-0: 20

```

我们发现有间隔，这是因为两个线程执行`++`的时候还没有调用 `sysout` 就被又加了一遍。怎么解决这个问题呢？Java 提供了 `synchronized` 的关键字，意思是在这个类里面，如果有哪个带 `syn` 关键字的方法被调用了，那么其他线程是无法调用所有的 `syn` 的方法的。有了它，不可能存在两个 `syn` 的方法被同时执行。为什么能做到这件事情呢？它内部建立了一个 `happens-before` 的关系，也可以理解为必须获取到当前对象上的一个锁，当获取到了这个锁才能调用对应的 `syn` 方法。

每一个对象就是一个锁，这个锁叫做 `intrinsic lock` 或者 `monitor lock`。每一个对象都有这样一个关联的内部锁。只要想执行 `syn` 方法，当前线程必须获取这个对象上的内部锁。一旦获得，只要不释放其他任何线程都不能访问对应的 `syn` 方法。

有的时候我们看到 `syn` 关键字作用在了一个类中 `static` 方法，关于这种调用，我们就会去尝试获取这个类上的内部锁。有了这个东西之后，我们再来看一下 `counter` 方法。

`Syn` 确实解决了问题，但是粒度是方法级别的，还是非常大的，如果我们的方法中只有

一小部分是不能同时调用的，我们希望只控制几行代码。这就要去 `syn` 不能加到方法上，我们可以使用 `synchronized(this){}` 语句块，这就表示在执行这个 `block` 的时候必须获取当前对象的内部锁，这就提供了一个细粒度的内部锁。

```
public class MsLunch {  
    private long c1 = 0;  
    private long c2 = 0;  
  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void addName(String name) {  
        synchronized(this) {  
            lastName = name;  
            nameCount++;  
        }  
        nameList.add(name);  
    }  
  
    public void inc1() {  
        synchronized(lock1)  
        { c1++; }  
    }  
    public void inc2() {  
        synchronized(lock2)  
        { c2++; }  
    }  
}
```

我们还可以创建 `Object` 对象，获取其内部锁。`C1` 和 `C2` 可能被两个线程各自同时加。如果 `inc1` 中又有递归调用 `inc1`，这就会出现死锁问题。我们需要把它定义为可重入的锁，也就是防止已经获取了锁的代码间接再获取同一个锁，防止死锁的情况。

- **Reentrant Synchronization**
- Recall that a thread cannot acquire a lock owned by another thread.
 - But a thread *can* acquire a lock that it already owns.
- Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*.
 - This describes a situation where synchronized code, *directly or indirectly*, invokes a method that also contains synchronized code, and both sets of code use the same lock.
 - Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

`Synchronize` 实际上就是在帮助我们保证某些动作是原子性的。注意高级语言的`++`不能直接默认是原子性的，因为它拆解到汇编语言中也会有好几行。

我们想使用原子性操作，这样线程之间就不会对我们的代码进行干扰。在 `java` 中，它做了一种比较简单的方式让我们防范这件事情。我们在变量之前加一个关键字 `volatile`，就是告诉 `java` 保证可以帮我们去做串行，但是这并不是 100% 安全的，它只是降低了内存不一致的风险。

如果我们没抢到锁的时候，等待在原地可能出现死锁、活锁、饥饿的情况。死锁就是两个人鞠躬并且等待对方站起来的情况。解除死锁要求某些操作回滚，活锁是两个人还没有占用这个资源，有机会直接解除这个情况。

```

public synchronized void bow(Friend bower) {
    System.out.format("%s: %s"
        + " has bowed to me!%n",
        this.name, bower.getName());
    bower.bowBack(this);
}
public synchronized void bowBack(Friend bower) {
    System.out.format("%s: %s"
        + " has bowed back to me!%n",
        this.name, bower.getName());
}

```

两个线程都调用自己对象上的 `bow` 方法，所以线程 A 和 G 各自持有自己对象上的 lock，但是 `bow` 想要结束必须调用对方的 `bowBack` 方法，可是调用的时候发现锁已经被取走了。所以这个锁因为 `bow` 结束不了而不释放，这就是死锁问题。

Starvation: 饿死是单方面的，一方占用锁的时间非常长，导致另一个需要锁的线程超时了，这就是饿死了，线程就无法执行了。

总的来说，两个线程交互的时候在竞争资源，这就需要提前判断一下能不能鞠躬，我们写代码的时候要写成 `guarded lock`，比如在这里，上来不能直接输出，需要保证只有条件到了才行。但是 `while` 去判断 cpu 都在空转，非常浪费资源。

```

public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try { wait(); }
        catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}

```

When `wait` is invoked, the thread releases the lock and suspends execution.

- At some future time, another thread will acquire the same lock and invoke `Object.notifyAll`, informing all threads waiting on that lock that something important has happened:

```

public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}

```

- Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of `wait`.

`Wait` 会把自己持有的锁释放掉。我们释放锁的时候，可以发一个通知告诉所有需要锁的人，等待某一个线程执行完，通知所有等的锁。至于谁能抢到就看我们怎么写程序了。当我们实在不能结束的时候是不是能把锁释放掉，给对方，对方做完再把锁放回来。

这个实际上就是 `producer-consumer application`，两个线程通过共享对象来通讯。Drop 里可以认为是一个信箱，只能放一封信，可以放入或者取走。`Take` 和 `Put` 要做 `syn`，即便是

单线程在跑如果信箱为空，也 `take` 不到东西，所以需要等待，如果信箱为空，那么就 `wait` 这个 `take` 方法，一直到有人去 `notify` 它才去操作。如果不为空，它就把消息传递出来。Producer 尝试往信箱中放东西的时候，获取 `lock`，其他 producer 就会发现获取不到 `lock`，就会去 `wait`。如果 producer 获取锁以后发现信箱不空，那就释放锁。`Empty` 和 `lock` 就保证了没有线程会死掉。

```

public class Producer implements Runnable {
    private Drop drop;

    public Producer(Drop drop) { this.drop = drop; }

    public void run() {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        Random random = new Random();

        for (int i = 0;
             i < importantInfo.length;
             i++) {
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt( bound: 1000));
            } catch (InterruptedException e) {}
        }
        drop.put("DONE");
    }
}

public class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) { this.drop = drop; }

    public void run() {
        Random random = new Random();
        for (String message = drop.take();
             ! message.equals("DONE");
             message = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt( bound: 5000));
            } catch (InterruptedException e) {}
        }
    }
}

```

创建了一个 `drop` 同时丢给生产者和消费者，生产者就是去 `drop` 中 `put` 东西，在生产者线程中调用 `drop.put`，因为 `put` 是 `syn` 的，生产者必须获取到 `drop` 的锁。获取不到的时候只能 `wait` 掉，如果能获取锁，进来看如果信箱不为空，那就放弃这把锁 `wait`，让其他线程获取掉这把锁打破死锁。

在我的程序里，所有对象都是不可改动的，每次要改的时候创建新对象，这样也可以解决这个问题，但是这样代价似乎会很大。

SynchronizedRGB，建立一个对应 RGB 和名字。

```

public void set(int red, int green, int blue, String name) {
    check(red, green, blue);
    synchronized (this) {
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }
}

public synchronized int getRGB() {
    return ((red << 16) | (green << 8) | blue);
}
public synchronized String getName() { return name; }
public synchronized void invert() {
    red = 255 - red;
    green = 255 - green;
    blue = 255 - blue;
    name = "Inverse of " + name;
}

```

看起来好像没什么问题，我们来看以下这个例子。先读 `rgb` 分量，再去读名字。读完 `rgb` 分量以后锁就被释放了，如果在读 `name` 之前出现了一个线程调用了 `set`，那么我们读到的名字就不对了。

所以我们需要用不可变的对象 `immutable object`，也就是每个变量都是 `private final`，一旦赋值再不可变了。没有 `set` 方法了，所以 `RGB` 和 `name` 都是 `final` 和 `private`。当我们每次去使用的时候，每次都在创建新的对象。这是一种设计模式，变成不可变对象，线程绝对安全。

高级并发对象：

High Level Concurrency Objects



- We'll look at some of the high-level concurrency. Most of these features are implemented in the new `java.util.concurrent` packages.
 - `Lock objects` support locking idioms that simplify many concurrent applications.
 - `Executors` define a high-level API for launching and managing threads. Executor implementations provided by `java.util.concurrent` provide thread pool management suitable for large-scale applications.
 - `Concurrent collections` make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
 - `Atomic variables` have features that minimize synchronization and help avoid memory consistency errors.
 - `ThreadLocalRandom` provides efficient generation of pseudorandom numbers from multiple threads.

刚才我们都是 `new` 一个 `thread` 对象，执行完就会被销毁掉，其实可以使用线程池，当我们需要执行线程的时候，可以从线程池中拿一个对象池来，这样就可以不用频繁地创建对象，另外还有一个，在做 `websocket` 例子的时候，我们把所有的 `session` 放到了 `concurrent collections`，这就是一个并发的链表，支持并发的 `put` 和 `get`。然后就是原子性的变量，在 `counter` 的时候，我们可以保证`++`是原子性的操作。

2021/9/30

可重入锁

`Lock` 就是一个 `object`，任何类型对象都可以，访问的时候 `syn` 住就可以获取到对象锁，如果当前对象的 `syn` 不够，那么可以自己定义一个锁。我们会发现这个锁需要重入（递归程序会导致在持有锁的情况下再尝试获得锁），我们要解决上节课讲的鞠躬死锁的问题，这时候就需要可重入的锁。

```
public class Safelock {  
    static class Friend {  
        private final String name;  
        private final Lock lock = new ReentrantLock();  
        public Friend(String name) {this.name = name;}  
    }  
}
```

```

public String getName() {return this.name; }
public boolean impendingBow(Friend bower) {
    Boolean myLock = false;
    Boolean yourLock = false;
    try {
        myLock = lock.tryLock();
        yourLock = bower.lock.tryLock();
    } finally {
        if (! (myLock && yourLock)) {
            if (myLock) {
                lock.unlock();
            }
            if (yourLock) {
                bower.lock.unlock();
            }
        }
    }
    return myLock && yourLock;
}
public void bow(Friend bower) {
    if (impendingBow(bower)) {
        try {
            System.out.format("%s: %s has bowed to me !%n", name,
bower.getName());
            bower.bowBack(this);
        } finally {
            lock.unlock();
            bower.lock.unlock();
        }
    }
    else {
        System.out.format("%s: %s started to bow to me, but saw that I was
already bowing to him.%n", name, bower.getName());
    }
}
public void bowBack(Friend bower){
    System.out.format("%s: %s has bowed back to me!%n", name,
bower.getName());
}
}
}

```

`impendingBow`, 返回 true 就可以向对方鞠躬。我们需要判断一下是不是能拿到对方的锁和自己的锁。tryLock 要么成功要么失败, 如果获得了两个锁, 那就可以开始鞠躬, 不放锁。如果只获得了一个锁, 不能鞠躬, 需要把自己已经拿到的一个锁也释放掉。

如果 `Impending` 返回为 `true`, 那么就正常鞠躬, 让对方鞠躬。无论是正常结束还是异常结束, 都要把两个锁释放掉。

Java Executors 框架

`Executors`, 用这个可以执行我们的线程, 使用的是线程池的概念来做管理, 因为着一个池里有很多线程对象, 当我们有多核的时候, 每个核上可以调用一个线程去执行。我们不需要每次人工地创建一个 `Runnable` 对象去执行, 我们只需要用 `execute` 方法把 `Runnable` 给它就可以执行。举个例子就知道, 比如我们的 `controller` 是一个实例还是多个实例, 一个 `http` 请求过来的情况, 单实例下大量的用户过来每个请求就要开一个对象去处理, 如果真的是每次都开一个新的线程去服务的话, 攻击起来非常容易, 大量的请求过来攻击既可以。我们用线程池的情况下, 一开始为空, 我们创建一个新的线程去执行, 但是做完以后线程没有被消除, 新的用户上来可以直接用这个线程去跑。如果线程满了的情况, 我们可以通过最近最小使用去挂起最远的线程。我们使用池的概念让池的对象被复用, 不至于让系统在大量请求过来的时候崩掉。所以我们只需要把 `Runnable` 对象丢给 `Executor` 方法执行。然后我们在 `Future` 上获取结果就可以。

我们还可以呀 `Scheduled Executor`, 倒计时后去执行任务(定时器任务)。刚才我们说线程池中有固定数量的线程实例, 我们需要用到 `fork` 和 `join` 功能。

Fork/Join



- Suppose that you want to blur an image.
 - The original `source` image is represented by an array of integers, where each integer contains the color values for a single pixel.
 - Performing the blur is accomplished by working through the source array one pixel at a time.
 - Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array.
 - Since an image is a large array, this process can take a long time.
 - You can take advantage of concurrent processing on multiprocessor systems by implementing the algorithm using the `fork/join` framework.

例子, 我们尝试 `blur` 一个图像, 也就是把周围九宫格做平均值。这张图可能很大, 此时执行就比较费时间, 我们能不能把图切成两块丢给两个 CPU 去做, 还有可能设置更大的阈值去处理它。

```

public static BufferedImage blur(BufferedImage srcImage) {
    int w = srcImage.getWidth();
    int h = srcImage.getHeight();

    int[] src = srcImage.getRGB( startX: 0, startY: 0, w, h, [rgbArray: null, offset: 0, w]);
    int[] dst = new int[src.length];

    System.out.println("Array size is " + src.length);
    System.out.println("Threshold is " + sThreshold);

    int processors = Runtime.getRuntime().availableProcessors();
    System.out.println(Integer.toString(processors) + " processor"
        + (processors != 1 ? "s are " : " is ")
        + "available");

    ForkBlur fb = new ForkBlur(src, start: 0, src.length, dst);

    ForkJoinPool pool = new ForkJoinPool();

    long startTime = System.currentTimeMillis();
    pool.invoke(fb);
    long endTime = System.currentTimeMillis();

    System.out.println("Image blur took " + (endTime - startTime) +
        " milliseconds.");

    BufferedImage dstImage =
        new BufferedImage(w, h, BufferedImage.TYPE_INT_ARGB);
    dstImage.setRGB( startX: 0, startY: 0, w, h, dst, offset: 0, w);

    return dstImage;
}

```

Blur 就是先获取可用 CPU 核数，然后我们创建一个 ForkBlur 对象，这个类继承自 RecursiveAction，而 RecursiveAction 继承于 ForkJoinTask。Fb 的参数为整个图片，处理完的对象放到 dst 中。现在线程池中只有 fb，fb 在被 invoke 的时候，如果小于阈值，那么就直接处理。

```

protected void computeDirectly() {
    int sidePixels = (mBlurWidth - 1) / 2;
    for (int index = mStart; index < mStart + mLength; index++) {
        // Calculate average.
        float rt = 0, gt = 0, bt = 0;
        for (int mi = -sidePixels; mi <= sidePixels; mi++) {
            int mindex = Math.min(Math.max(mi + index, 0), mSource.length - 1);
            int pixel = mSource[mindex];
            rt += (float) ((pixel & 0x00ff0000) >> 16) / mBlurWidth;
            gt += (float) ((pixel & 0x0000ff00) >> 8) / mBlurWidth;
            bt += (float) ((pixel & 0x000000ff) >> 0) / mBlurWidth;
        }

        // Re-assemble destination pixel.
        int dpixel = (0xff000000)
            | (((int) rt) << 16)
            | (((int) gt) << 8)
            | (((int) bt) << 0);
        mDestination[index] = dpixel;
    }
}

```

也就是附近的九宫格中，计算平均值修改到 destination pixel 中。

```

protected void compute() {
    if (mLength < sThreshold) {
        computeDirectly();
        return;
    }

    int split = mLength / 2;

    invokeAll(new ForkBlur(mSource, mStart, split, mDestination),
              new ForkBlur(mSource, start: mStart + split, length: mLength - split,
                           mDestination));
}

```

如果图片比较大，我们就创建两个 `ForkBlur`，也就是可以被线程池调用的对象。`invokeAll` 就会把创建出来新的任务丢给线程池。线程池再去调用 `compute`，如果太大就继续切。只有所有线程执行完毕才会结束。至于线程池到底是怎么管理的，我们不用再仔细的写代码去维护它。

然后上节课我们最初的 `counter` 写的有问题，读到的 `0,2,4,6,8` 不是连续的，然后我们对 `inc` 和 `dec` 方法加上 `synchronize` 会解决这个问题，到底是哪些方法需要 `syn` 完全需要我们在代码中加上这个操作。我们可以用到 `atomic` 数据类型，比如 `java` 中的 `AtomicInteger`，它在帮我们保证原子性，不会出现上节课我们看到的错误，我们就不需要再去管哪些函数需要加上 `syn` 关键字。

```

import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger( initialValue: 0);

    public void increment() { c.incrementAndGet(); }

    public void decrement() { c.decrementAndGet(); }

    public int value() { return c.get(); }
}

```

多线程使用 `Random` 的时候，性能比较差。我们需要修改为 `ThreadLocalRandom.current()`，但是直接改的话，我们发现不同线程产生的随机数是一样的。我们每一次在线程中应该生成 `ThreadLocalRandom.current()` 对象，然后再去产生随机数。

多线程下 `ThreadLocalRandom` 用法 <https://www.cnblogs.com/little-fly/p/12431090.html>

内存中的缓存

我们这节课讲内存中的缓存。

Q: 为什么我要用缓存？

A: 我们已经讲到了在数据库 `DB` 中已经带了一个很大的缓存才能实现事务功能。比如说我们要做一个 `query`, `query` 的东西一定会在这个缓存中存活一段时间。但是这个缓存在 `MySQL` 中，不在我们的程序中，这个缓存不受我们控制。

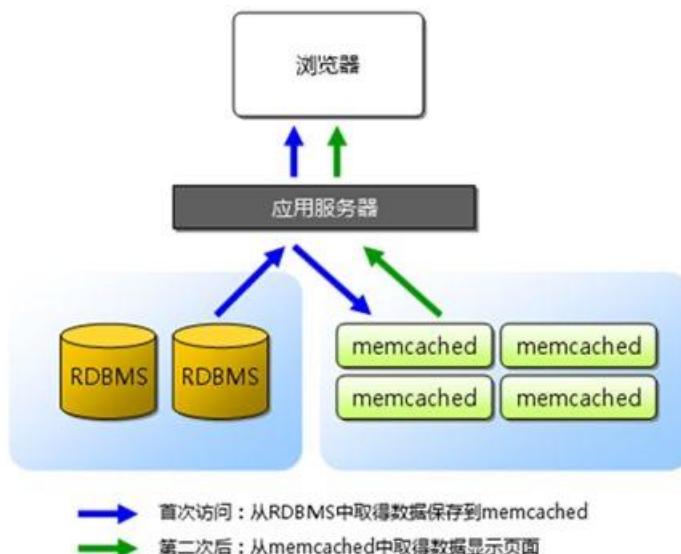
我们再看我们应用服务器 `tomcat`，我们再用 `spring jpa` 的时候，是把表格型的数据拿到 `tomcat` 内存中变成一个个对象，这不是已经缓存了吗？但是这个缓存要么是 `spring` 在控制，

要么是 hibernate 在控制，要么是 mybatis 在控制，到底替换掉什么我们也不能控制，而且我们的应用中不止有关系型数据，如果我们有一个文件系统把文件抽象成了一个对象，我们应该怎样缓存文件或者 MongoDB 中的数据呢？

我们的内存中有很多对象要么是我们管不到的，要么是别人不想管的对象。在这种情况下，我们需要一个 cache，也就是 redis 或者 memcached，我们把对象给它，这是进程间通讯，会不会耗时呢？无论从哪里去，都是对外设（硬盘）操作，而 redis 等都是在内存中操作，所以进程间通信的速度会比从硬盘上去读要快。

Memcached

Memcached，它在内存中做了一个 key-value 的存储，它是纯内存的，关键就是 key 怎么去做，跑起来比较简单，下载完去跑就行了。



我们把数据库从硬盘上读到内存之后，我们就可以塞到 memcached 中，它是可以做分布式的，可以存在一个集群中。所以我们写代码的时候，要先从 memcached 去找，找不到再到硬盘上去找，并且要同时塞到 memcached 中去。

LocalSSMConfiguration.java

```
@Configuration
public class LocalSSMConfiguration extends AbstractSSMConfiguration {
    @Bean
    @Override
    public CacheFactory defaultMemcachedClient() {
        final CacheConfiguration conf = new CacheConfiguration();
        conf.setConsistentHashing(true);
        final CacheFactory cf = new CacheFactory();
        cf.setCacheClientFactory(new com.google.code.ssm.providers.xmemcached.MemcacheClientFactoryImpl());
        cf.setAddressProvider(new DefaultAddressProvider("127.0.0.1:11211"));
        cf.setConfiguration(conf);
        return cf;
    }
}
```

以上为 memcached 的配置类，配置到端口就可以得到了像连接一样的东西。

```
PersonRepository.java
```

```
public interface PersonRepository extends JpaRepository<Person, Integer>{}
```

```
PersonDao.java
```

```
public interface PersonDao {  
    Person findOne(Integer id);  
}
```

```
PersonDaoImpl.java
```

```
@Repository  
public class PersonDaoImpl implements PersonDao {  
    @Autowired  
    private PersonRepository personRepository;  
    @Override  
    @ReadThroughSingleCache(namespace = "Alpha", expiration = 3600)  
    public Person findOne(@ParameterValueKeyProvider Integer id) {  
        return personRepository.getOne(id);  
    }  
}
```

我们在需要使用到的 person 的 dao 层上，我们需要加上@ReadThroughSingleCache，其中 3600 就是在 memorycached 中的存活时间是多少。Id 前多了注释@ParameterValueKeyProvider，也就是扔进去的时候，key 就是 person+id。

我们怎么维护一致性呢？那我们就要先修改 Redis 再修改硬盘，如果修改硬盘失败了怎么办？Redis/Memcached 和文件系统都不支持回滚呢？对于文件系统不能做回滚的情况，我们只能做补偿。

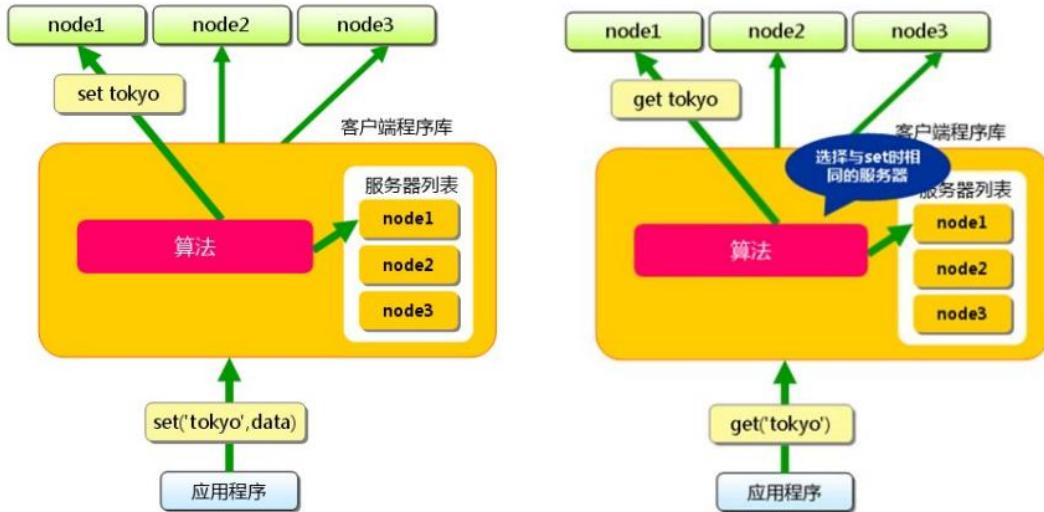
问题 1：memcached 可以做分布式，这是怎么分布的？是不是需要我们程序来指定数据在哪台机器上存取呢？

问题 2：小对象和大对象都是 key-value 存储（导致的内存碎片的大小不同），怎么该合理地使用内存呢？

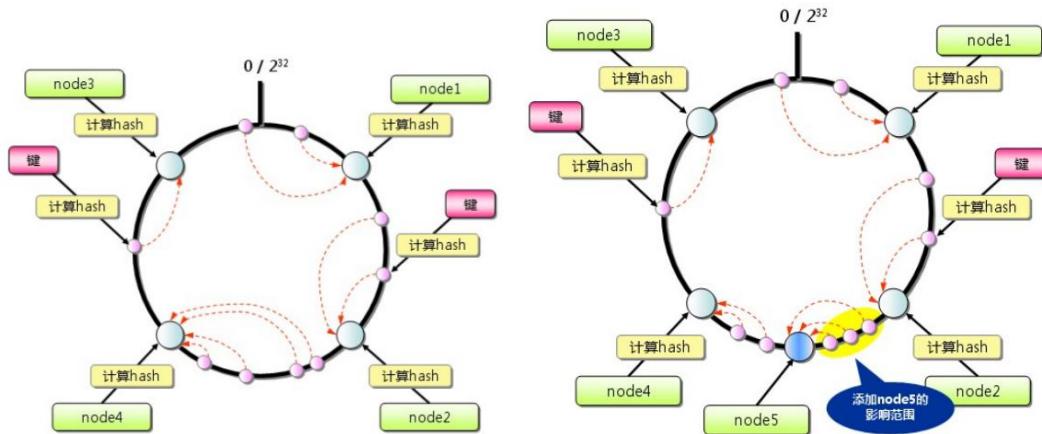
有了 memcached 之后，在应用服务器多了这么一层之后，我们的数据就不需要每次操作硬盘了。



A2：把内存分成很多部分，比如一个部分放的都是 88 字节的，另一个部分都是 112 字节的，每当我们要缓存一个对象的时候，我们找到浪费最少的一个区域放进去。



A1：它会根据一致性哈希算法，来知道这个 `tokyo` 在哪个节点上。这个是 memcached client 的一部分，不需要我们程序去做什么事情，只要这个算法是固定的，集群多大和我们的算法无关。



节点也赋一个 id，节点和数据的 id 不要重，本质上就是拿一堆整数在环上排列一下，排列好之后，数据就存在顺时针走所碰到的第一个节点上，这种方法存储数据之后，好处就是底下我们想多增加一个节点 `node5`，我们只发现只要把下面的三个数据从 `node4` 搬到 `node5` 上，本来要全部搬动一遍数据，现在只需要搬动个别几个。

Redis



- Redis is what is called a **key-value store**,
 - often referred to as a **NoSQL** database.
 - The essence of a key-value store is the ability to store some data, called a value, inside a key.
- Redis is an open source, BSD licensed, advanced key-value store.
 - It is often referred to as a data structure server since keys can contain **strings, hashes, lists, sets** and **sorted sets**.
- In order to achieve its outstanding performance, Redis works with an **in-memory dataset**.
 - Depending on your use case, you can persist it either by dumping the dataset to disk every once in a while, or by appending each command to a log.
- Redis also supports trivial-to-setup **master-slave replication**,
 - with very fast **non-blocking** first synchronization, auto-reconnection on net split and so forth.

Memcached 在内存中放不下了怎么办，用最近最小使用方式替换，而 redis 放不下的时候就往 redis 所在的硬盘上去做持久化，因为我们每次关机后再开启 memcached 一定是空的，而 redis 会有数据。

我们为什么又要把数据从 redis 存储到硬盘上呢？原先我们硬盘上的文件是 JSON 文件，读进来解析以后变成 JAVA 对象，现在我们 redis 向硬盘存储的不是 JSON 而是 java object，至少我们再也不需要做一次 JSON 解析的过程中了，省去了这个步骤。更重要的是，数据库为了安全起见可能只有一个内网 IP，而我们的 tomcat 和数据库在两台机器上通过内网连接，而 tomcat 还有一个对外暴露的公网 ip。而 tomcat 和 redis 在一台机器上，至少比从另外一台机器上读要快一点。我们可以在做持久化的时候往 SSD 上去写，这样就会快一点。我们甚至可以让 redis 直接操作到 SSD 上，把内存腾出来给别人。这样我们就理解了 redis 为什么又要把数据持久化到硬盘上，以及这和原先的硬盘的存储的区别。

1. 省掉网络开销
2. 省掉解析开销
3. 可以往 SSD 上做持久化

Redis 的具体操作略，详见文档。

```
application.properties
# Redis 数据库索引（默认为0）
spring.redis.database=0
# Redis 服务器地址
spring.redis.host=localhost
# Redis 服务器连接端口
spring.redis.port=6379
# Redis 服务器连接密码（默认为空）
spring.redis.password=
# 连接池最大连接数（使用负值表示没有限制）
```

```

spring.redis.jedis.pool.max-active=8
# 连接池最大阻塞等待时间（使用负值表示没有限制）
spring.redis.jedis.pool.max-wait=-1
# 连接池中的最大空闲连接
spring.redis.jedis.pool.max-idle=8
# 连接池中的最小空闲连接
spring.redis.jedis.pool.min-idle=0
# 连接超时时间（毫秒）
spring.redis.timeout=300

```

Redis 配置类。

```

RedisConfig.java
@Configuration
@EnableCaching
public class RedisConfig extends CachingConfigurerSupport {
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory factory) {
        RedisTemplate<String, Object> template = new RedisTemplate<String, Object>();
        template.setConnectionFactory(factory);
        Jackson2JsonRedisSerializer jacksonSeial = new
Jackson2JsonRedisSerializer(Object.class);
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jacksonSeial.setObjectMapper(om);
        template.setValueSerializer(jacksonSeial);
        template.setKeySerializer(new StringRedisSerializer());
        template.setHashKeySerializer(new StringRedisSerializer());
        template.setHashValueSerializer(jacksonSeial);
        template.afterPropertiesSet();
        return template;
    }
}

```

```

MessagingRedisApplication.java
@SpringBootApplication
public class MessagingRedisApplication {
    private static final Logger LOGGER =
LoggerFactory.getLogger(MessagingRedisApplication.class);
    @Bean
    RedisMessageListenerContainer container(RedisConnectionFactory connectionFactory,
MessageListenerAdapter listenerAdapter) {
        RedisMessageListenerContainer container = new
RedisMessageListenerContainer();

```

```

        container.setConnectionFactory(connectionFactory);
        container.addMessageListener(listenerAdapter, new PatternTopic("chat"));
        return container;
    }

    @Bean
    MessageListenerAdapter listenerAdapter(Receiver receiver) { return new
MessageListenerAdapter(receiver, "receiveMessage"); }

    @Bean
    Receiver receiver() { return new Receiver(); }

    @Bean
    StringRedisTemplate template(RedisConnectionFactory connectionFactory) {
        return new StringRedisTemplate(connectionFactory);
    }

    public static void main(String[] args) throws InterruptedException {
        ApplicationContext ctx = SpringApplication.run(MessagingRedisApplication.class,
args);
    }
}

@SpringBootApplication
public class MessagingRedisApplication {
    private static final Logger LOGGER =
LoggerFactory.getLogger(MessagingRedisApplication.class);

    @Bean
    RedisMessageListenerContainer container(RedisConnectionFactory connectionFactory,
MessageListenerAdapter listenerAdapter) {
        RedisMessageListenerContainer container = new
RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);
        container.addMessageListener(listenerAdapter, new PatternTopic("chat"));
        return container;
    }

    @Bean
    MessageListenerAdapter listenerAdapter(Receiver receiver) { return new
MessageListenerAdapter(receiver, "receiveMessage"); }

    @Bean
    Receiver receiver() { return new Receiver(); }

    @Bean
    StringRedisTemplate template(RedisConnectionFactory connectionFactory) {
        return new StringRedisTemplate(connectionFactory);
    }

    public static void main(String[] args) throws InterruptedException {
        ApplicationContext ctx = SpringApplication.run(MessagingRedisApplication.class,
args);
    }
}

```

```
}
```

Template 就是得到一个 Redis 的 template，要在启动时注入。一旦 receiver（消息进入到 redis）的时候，就会调用 receiveMessage 关联起来。

MsgController.java

```
@RestController
public class MsgController {
    @Autowired
    WebApplicationContext applicationContext;
    @GetMapping(value="/msg")
    public void findOne() {
        StringRedisTemplate template =
applicationContext.getBean(StringRedisTemplate.class);
        // Send a message with a POJO - the template reuse the message converter
        System.out.println("Sending an email message.");
        template.convertAndSend("chat", "Hello from Redis in HTTP!");
    }
}
```

Receiver.java

```
public class Receiver {
    private static final Logger LOGGER = LoggerFactory.getLogger(Receiver.class);
    private AtomicInteger counter = new AtomicInteger();
    public void receiveMessage(String message) {
        LOGGER.info("Received <" + message + ">");
        counter.incrementAndGet();
        LOGGER.info("counter<" + counter.get() + ">");
    }
    public int getCount() { return counter.get(); }
}
```

我们把对象变成纯文本后就往 redis 中发进去，在一个 chat 文本的 channel 中。Receiver 会说一旦有消息过来的时候，就会输出收到了多少消息。Receiver 一旦接收到消息就会调用 receiveMessage 函数。

这个例子中，redis 就不再是一个缓存，而是一个消息的目的地。

2021/10/9

<https://download.csdn.net/download/scgyus/10667172>

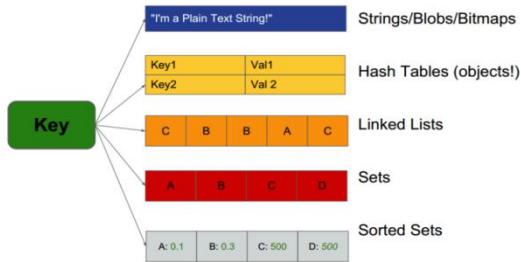
Key value 有五种不同的结构

2.Redis Data Structures

Key : 一个字符串。

Value : string、hash、list、set、sorted set等数据结构存储的数据。

Key => { Data Structures }



这样的话，id 搜索就可以有映射到里面。所以通过这种方式就构建了关系型数据库表的东西。

它里面还有一些不太常用的数据结构。在设计的时候，key 很重要，它不一定非要是 bookid，也可以是 book::author，可以通过冒号来分割成不同的层次。在这种情况下可以去找分属的用户是谁，key 是拿来做索引的，所以它本身体积不能太大。

3 Key-Value Design

3.1 Key Name

- ◆ 以业务名为前缀，作为namespace区分不同的应用，还可以防止keyname冲突。
比如,b2c_开头的key是b2c业务的数据。
- ◆ key的名字能够~~言简意赅~~体现数据内容，用“：“分割层次。
SET b2c:orders:id 456977122
- ◆ Key的长度在30个字符以内。Redis最大允许的字符串长度为512M。
- ◆ 不要包含空白字符（空格、换行符\\n、单双引号以及其他需要转义的字符）
- ◆ 设计原则
 - 1) 高可读性
 - 2) 简洁性

3 Key-Value Design

3.3 关系型数据转K-V

➤ 关系型数据表

id	first_name	last_name	login
1	Ryan	Briones	ryanbriones
...

3 Key-Value Design

3.2 value的设计

- ◆ 设计原则
- 1) 单个key的体积不要过大，大key拆分成多个小key：
 - ✓ String类型避免存储长度超过2k的字符串；
 - ✓ List、Hash、Set和Zset的元素个数不要超过5000个。
- 2) 尽可能使用简单的数据类型。
- 3) 选择合适的数据类型。

➤ 利用Redis存储关系型数据

```
INCR users:uids
"1"
SET users:1:first_name Ryan
OK
SET users:1:last_name Briones
OK
SET users:1:login ryanbriones
OK
```

Redis 在内存中放数据到底应该怎么做呢？比如我们的数据库中有一张表是 book，用户访问一次加载到 redis 中，但是还有一种极端做法是在 spring boot 启动的时候把 sql select all，全部取出来放到 redis 里。以后除非有写操作，否则操作会全部在内存里，这个做法也是可取的，只需要我们的内存够大。但是这个的前提就是明确什么样的东西应该放到 redis 里，

也就是基本上只用来读的数据，如果数据会被频繁地写，那么放到 redis 比较不太合适的。

对于 order 这个数据，我们写是只写一开始的一次的，如果我们经常要做订单的统计和数据，放到 redis 是比较有价值的。

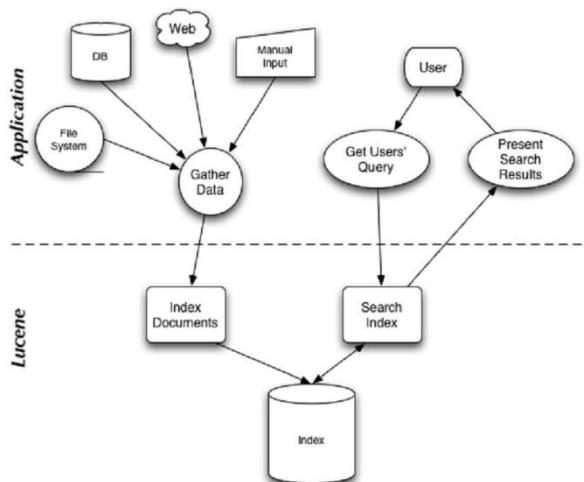
我们能不能操作全在 redis 里，每天同步到 sql 中？不行，因为断电 redis 中数据就没了，同时 mysql 可能被多个 app 访问，这会导致严重的数据不一致问题。

这节课讲全文搜索，c++ premier 中有一个例子，给一段文本，告诉你每个词出现过多少次，并且各自出现在第几行的第几个单词。就要把单词用空格断开，然后考虑动词的不同形式转化为同一个词，然后根据相同的 key 去合并在一起。这就是全文搜索的意思：在非结构化的文本（不像 excel 和 sql）中，去找到所有不同的单词的索引。搜索的时候比全文扫描一边要快。

全文搜索的工具：lucene，solr，elastic search。

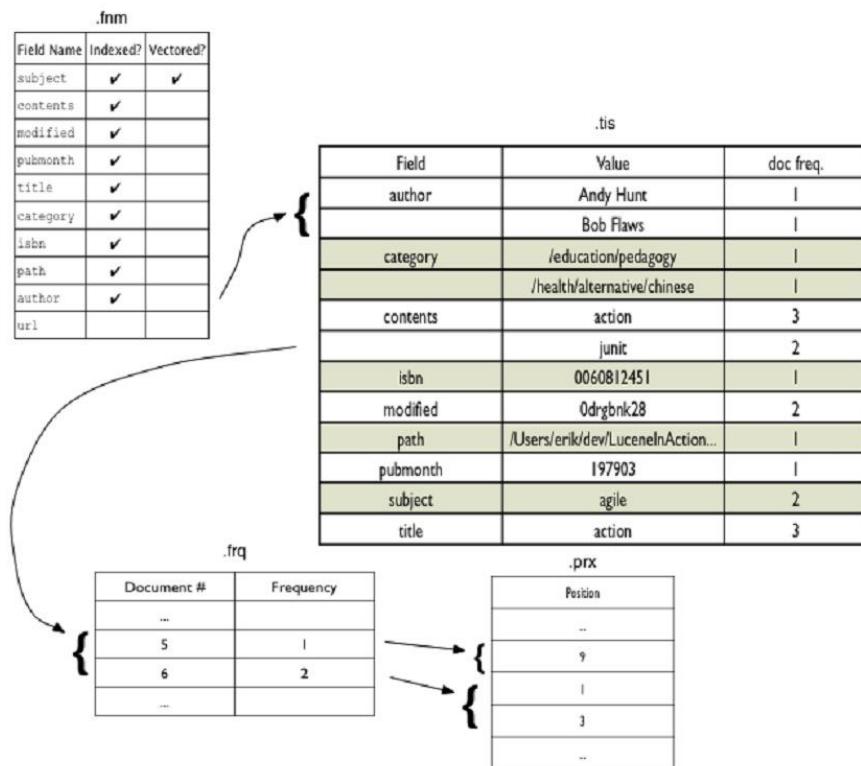
- Lucene is a high performance, scalable Information Retrieval (IR) library.
 - It lets you add indexing and searching capabilities to your applications.
 - Lucene is a mature, free, open-source project implemented in Java.
 - it's a member of the popular Apache Jakarta family of projects, licensed under the liberal Apache Software License.
- Lucene provides a simple yet powerful core API
 - that requires minimal understanding of full-text indexing and searching.

Lucene 最重要的就是建立全文的反向索引，也就是一个单词出现在文档的哪些位置上。
(正向索引是一个文档的什么位置有什么单词)



它支持多种文件格式（比如 html 就解析一下），进来以后建立反向索引，之后的 search 都是在其之上做的。

索引一般来说尺寸不应该太大，比如数据库中索引的树的高度决定了我们搜索的效率，如果索引文件太大，在硬盘和内存中不断地交换，也会影响我们的速度。



这些域分为四种，举一个例子，左上角的.fnm 表中 author 是要被索引起来的，它有两个不同取值 andy 和 bob，各自出现在一个文档中。Junit 出现在两个文档中，分别是 1 次和 2 次，1 次的出现在 5 号文档的第 9 个单词，而 2 次的是出现在 6 号文档的 1 号和 3 号单词。下次搜索的时候就可以直接查表找到位置，并且高亮出来其周边区域。

不管是哪种搜索，无非就是准确性（找到了 8 个，其中有几个是符合要求的）、召回率（比如 10 个满足条件的，找到了几个）。都得高才能说明搜索非常好，然后我们再去考虑速度怎么样，以及支持不支持高级的搜索（sql、通配符、支不支持排序、etc）。

Luceme 官网的例子：

A sample application



- Suppose you need to index and search files stored in a directory tree, not just in a single directory
- These example applications will familiarize you with Lucene's API, its ease of use, and its power.
- The code listings are complete, ready-to-use command-line programs.

```

// open an index and start file directory traversal
public static int index(File indexDir, File dataDir) throws IOException {
    if (!dataDir.exists() || !dataDir.isDirectory()) {
        throw new IOException(dataDir
            + " does not exist or is not a directory");
    }

    IndexWriter writer = new IndexWriter(indexDir,
        new StandardAnalyzer(), true);
    writer.setUseCompoundFile(false);           ← Create Lucene index

    indexDirectory(writer, dataDir);

    int numIndexed = writer.docCount();
    writer.optimize();
    writer.close();           ← Close index
    return numIndexed;
}

```

Index 这个函数才真正在 indexDir 目录（里面是很多很多文本）建立了索引，然后写到了 dataDir 中去。其中的 StandardAnalyzer 就是去断开单词、自动去掉标点符号，最终得到的是一个个去掉标点符号的单词。然后 indexDirectory 就是把建立的索引绑定到了 dataDir 中，只有 writer.optimize() 后才把多线程的结果合并成一个文件，真正写到了硬盘上。

```

// method to actually index a file using Lucene
private static void indexFile(IndexWriter writer, File f)
    throws IOException {

    if (f.isHidden() || !f.exists() || !f.canRead()) {
        return;
    }

    System.out.println("Indexing " + f.getCanonicalPath());

    Document doc = new Document();
    doc.add(Field.Text("contents", new FileReader(f)));           ← Index file content

    doc.add(Field.Keyword("filename", f.getCanonicalPath()));      ← Index file name
    writer.addDocument(doc);           ← Add document to Lucene index
}

```

这里才真正是 Lucene 它自带的东西，document 就是一个索引的 entry，要往中添加 Keyword 这么一个结构。然后调用 writer 写到 index 里去。

```

public static void search(File indexDir, String q)
    throws Exception {
    Directory fsDir = FSDirectory.getDirectory(indexDir, false);
    IndexSearcher is = new IndexSearcher(fsDir); ← Open Index

    Query query = QueryParser.parse(q, "contents", new StandardAnalyzer()); ← Parse query
    long start = new Date().getTime();
    Hits hits = is.search(query); ← Search Index
    long end = new Date().getTime();

    System.err.println("Found " + hits.length() + " document(s) (in " + (end - start) +
        " milliseconds) that matched query '" + q + "'"); ← Write search stats

    for (int i = 0; i < hits.length(); i++) { ← Retrieve matching document
        Document doc = hits.doc(i);
        System.out.println(doc.get("filename")); ← Display filename
    }
}

```

有了 query 之后，就在索引上搜索一下，遍历 hit 就是我们找到索引里面匹配搜索条件的记录。

比如我们想对我们 E-BOOK 中的所有 java 文件去做反向索引，首先文件名肯定是需要的，就要作为 keyword 的。File url 其实不需要参与到索引中（不用加入到树中），但是需要存在索引里，比如我们未来找到了 1.java，最好我们同时就能得到它的位置在哪里。并且文件的源码因为太大了，是不能放到索引的备注中的，所以它只能参与到索引中（不存、但是要索引），第四个东西就是注释可能经常要描述函数的功能是什么，既要被分析又要参与索引。

这四种就是

Keyword (不用 analyzer 断开，但是要索引)

Unindexed (不需要 analyzer 去处理，但是要存在索引里)

Unstored (不需要存在索引里，但是关心的关键词就要筛选出来做索引)

Text (既要有 analyzer 去断开，又要 lexeme 去分析)

Core indexing classes



- **IndexWriter**
 - This class creates a new index and adds documents to an existing index.
- **Directory**
 - The Directory class represents the location of a Lucene index.
- **Analyzer**
 - The Analyzer, specified in the IndexWriter constructor, is in charge of extracting tokens out of text to be indexed and eliminating the rest.
- **Document**
 - A Document represents a collection of fields.
- **Field**
 - Each field corresponds to a piece of data that is either queried against or retrieved from the index during search.

```

public abstract class BaseIndexingTestCase extends TestCase {
    protected String[] keywords = {"1", "2"};
    protected String[] unindexed = {"Netherlands", "Italy"};
    protected String[] unstored = {"Amsterdam has lots of bridges",
                                  "Venice has lots of canals"};
    protected String[] text = {"Amsterdam", "Venice"};
    protected Directory dir;

    protected void setUp() throws IOException {
        String indexDir =
            System.getProperty("java.io.tmpdir", "tmp") +
            System.getProperty("file.separator") + "index-dir";
        dir = FSDirectory.getDirectory(indexDir, true);
        addDocuments(dir);
    }

    protected void addDocuments(Directory dir) throws IOException {
        IndexWriter writer = new IndexWriter(dir, getAnalyzer(), true);
        writer.setUseCompoundFile(isCompound());
        for (int i = 0; i < keywords.length; i++) {
            Document doc = new Document();
            doc.add(Field.Keyword("id", keywords[i]));
            doc.add(Field.UnIndexed("country", unindexed[i]));
            doc.add(Field.UnStored("contents", unstored[i]));
            doc.add(Field.Text("city", text[i]));
            writer.addDocument(doc);
        }
        writer.optimize();
        writer.close();
    }

    protected Analyzer getAnalyzer() { return new SimpleAnalyzer(); }
    protected boolean isCompound() { return true; }
}

```

所以在建立索引的时候要搞清楚这四种不同的 **field**。在 **lucmem** 中，允许在 **document** 中每一个文件有相同的 **field**，但是这个约束没有那么严格，不同的行可以有不同的列。

索引已经建好了，新加了文件怎么办呢？我们可以通过 **appendable field** 去追加。可以增量式的去建立。

后面都是介绍 **api**，去找官方文档看吧。

2021/10/11

SOAP 协议

SOAP: 简单对象访问协议，要么就是用 **http** 协议，用 **get/put/delete/post**，也可以实现 **restful service**，关键的问题是知道什么样的业务应该包装成 **web service**，我们没有办法用远程方法调用去处理它，站在支付宝的角度应该怎么考虑这件事情，因为不同人拿不同语言的操作来调用，不可能为每个源去开发一个接口，所以要开发独立于具体编程语言的服务，所以这种东西就是基于纯文本。**Service** 就是独立于具体实现的意思，除了能够通过纯文本调用之外，还有一件事情就是保证协议“能够走这么远”，比如 **http** 就可以调用到地球对面的服务，而远程方法调用就做不到这件事情。

SOAP 协议只是用来封装消息用的。封装后的消息你可以通过各种已有的协议来传输，比如 **http**, **tcp/ip**, **smtp** 等等。

Web 协议常见的就是 **Http** 协议，但是它只是其中之一，比如我们发邮件的时候就会有 **IMAP**, **SMTP**, **POP3**，大家比较熟悉的还有 **FTP**, **TFTP**，它们都属于 **web** 的协议族。也就是互联网上跑这些协议都属于 **web service**。用户使用不同语言写的情况下，发送过来的请求，包含对象的情况下，我们就要转成 **JSON** 对象，**JSON** 本来就是一种纯文本格式。当我们想到要发送 **JSON** 的时候，其实就是在使用纯文本进行调用服务了。

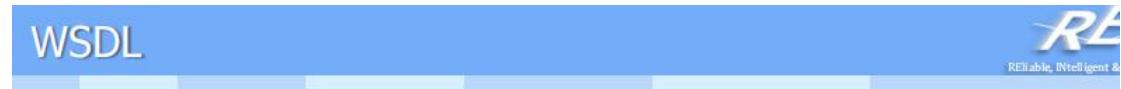
所以为什么要有 **webservice**，也就是我们需要以纯文本的格式进行调用。比如我们用 **java** 写了一个服务，叫做 **billing**（支付），这个服务是用 **java** 写的，能够进行调用。我们怎么样使用 **C#**的程序去调用 **java** 的这个 **billing** 服务呢？大家想到了要传统文本，比如 **JSON**，它的本质就是在传送输出；还有一种想法，**billing** 一定会有 **api**，比如里面有一个函数叫做 **pay**，**C#**说想要调用这个 **api**，然后返回对应的值，因为 **C#**不能直接调用 **pay**，需要把调用什么函数、参数是什么这件事情使用纯文本进行描述，这就是 **SOAP**（简单对象传输协议）。什么样的结构才能让它知道你想要干什么呢？我们需要定义这件事情，**SOAP** 也实际上在定义这

一件事情，在调用 pay 函数的时候也要按照固定的格式（xml 格式）。SOAP 相当于去规定在 xml 文件里存在哪些标签、每个标签的子标签能有多少个。

Imagine that you want to develop a web services component that implements the following interface:

```
public interface TravelAgent {  
    public String makeReservation(int cruiseID,  
        int cabinID, int customerId, double price);  
}
```

比如，我们有一个订舱位的函数，这是 java 端暴露出来的 travelAgent，作为 C# 的客户端想要调用它的时候，怎么描述呢？我们先要把这个接口描述成纯文本格式，这样 C# 拿到以后就知道了。



- A WSDL document that describes the `makeReservation()` method might look like this:

```
<?xml version="1.0"?>  
<definitions name="TravelAgent"  
    xmlns="http://schemas.xmlsoap.org/wsdl/"  
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
    xmlns:titan="http://www.titan.com/TravelAgent"  
    targetNamespace="http://www.titan.com/TravelAgent">  
    <!-- message elements describe the parameters and return values -->  
    <message name="RequestMessage">  
        <part name="cruiseId" type="xsd:int" />  
        <part name="cabinId" type="xsd:int" />  
        <part name="customerId" type="xsd:int" />  
        <part name="price" type="xsd:double" />  
    </message>  
    <message name="ResponseMessage">  
        <part name="reservationId" type="xsd:string" />  
    </message>  
  
    <!-- portType element describes the abstract interface of a web service -->  
    <portType name="TravelAgent">  
        <operation name="makeReservation">  
            <input message="titan:RequestMessage"/>  
            <output message="titan:ResponseMessage"/>  
        </operation>  
    </portType>  
    <!--binding element tells us which protocols and encoding styles are used -->  
    <binding name="TravelAgentBinding" type="titan:TravelAgent">  
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>  
        <operation name="makeReservation">  
            <soap:operation soapAction="" />  
            <input>  
                <soap:body use="literal" namespace="http://www.titan.com/TravelAgent"/>  
            </input>  
            <output>  
                <soap:body use="literal" namespace="http://www.titan.com/TravelAgent"/>  
            </output>  
        </operation>  
    </binding>
```

```

<!-- service element tells us the Internet address of a web service -->
<service name="TravelAgentService">
    <port name="TravelAgentPort" binding="titan:TravelAgentBinding">
        <soap:address location="http://www.titan.com/webservices/TravelAgent" />
    </port>
</service>
</definitions>

```

上面有 Port，也就是我们的 webservice 就像一个端口一样，要和我们的 java 接口名字对应，port 里就会有一系列的 operation，operation 里就有 Input 和 Output，告诉它输入输出的 message 是什么样的。在一开始我们定义了两个 message，也就是 request message 和 response message，这就已经把这个方法的输入参数讲清楚了。

同一个接口如果它能接受 http 协议，那就是一个服务，如果还能接受 ftp 协议，那就是第二个服务。所以在后端 java 其实暴露出来了多个服务。某个端口和 service 绑在一起就是 binding，我们可以看到它和 http 协议绑定在一起，在这个 binding 中传递的是 soap binding。

还剩下最后一件事情，就是到哪去调用这个服务，后面的就是定义的 bind 真正能被访问到的位置。自此一个 java 定义的服务，如何通过纯文本的形式去调用，就已经描述清楚了。

SOAP is defined by its own XML Schema and relies heavily on the use of XML Namespaces.

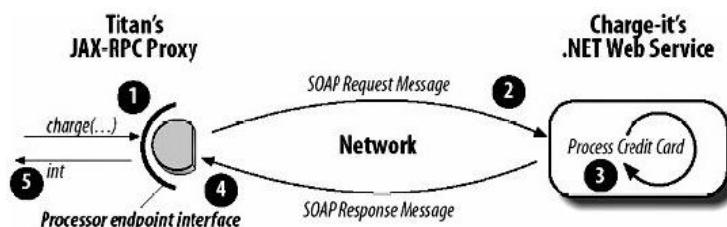
- Here's a SOAP request message that might be sent from a client to a server:

```

<?xml version='1.0' encoding='UTF-8' ?>
<env:Envelope
    xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
    <env:Header />
    <env:Body>
        <reservation xmlns="http://www.titan.com/Reservation">
            <customer>
                <!-- customer info goes here -->
            </customer>
        </reservation>
    </env:Body>
</env:Envelope>

```

Body 里就是我们刚才的东西，需要匹配我们刚才定义的 input 和 output。定义好这件事情以后，我们真的要给用户开始使用了。



JAVA EE

下载到了.WSDL文件，但是它是一个纯文本，我们可以用第三方工具(Axis)对着它进行解析，生成Java的接口类，它还会帮你生成实现了这个接口的一个类，很显然这个实现类里面会有charge()函数的实现，注意这里不是在实现业务，而是说将来有人调用了charge()之后，怎么把请求发送到C# .NET服务器去，应该怎么把java对charge的调用翻译成SOAP消息发送到C# .NET去，收到响应后该怎么转化成Java中的类。这都是第三方帮我们生成的，我们在java中只需要获取这个类，取决于我们使用的是什么工具，Object getEndPoint()后，就和本地调用一样了。

C# .NET

里有一个 C# class
存在一个函数叫做 int charge() {...}
描述一下就存在了一个 .WSDL文件
生成了这个文件以后就存在两种方法
1.支付宝很出名，有自己的网站，告诉你到这个位置来下载这个文件
2.小公司，把这个文件放到一个大家都认识的地方，比如说阿里云上，告诉你可以去那里下载
WCF也会工具这个生成一个接口去调用
C#的业务实现，其中存在proxy来翻译
soap的请求和响应。

如果客户端和服务器都是 Java，也可以使用这样的方法，来使用 http 协议来实现地域距离较远的服务调用。这就是用 Soap 的方式在实现交互。第三方工具第一个例子用的是 jaxb。因为我们需要第三方文件把 wsdl 执行完，生成了 class 文件才能编译通过，所以我们在编译前需要先解析 wsdl。

pom.xml

```
<!-- tag::xsd[] -->
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>jaxb2-maven-plugin</artifactId>
    <version>2.5.0</version>
    <executions>
        <execution>
            <id>xjc</id>
            <goals>
                <goal>xjc</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <sources>
            <source>${project.basedir}/src/main/resources/countries.xsd</source>
        </sources>
    </configuration>
</plugin>
<!-- end::xsd[] -->
```

countries.xsd

```
<xs:element name="getCountryRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="getCountryResponse">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="country" type="tns:country"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

```

import io.spring.guides.gs_producing_web_service.GetCountryRequest;
import io.spring.guides.gs_producing_web_service.GetCountryResponse;

@Endpoint
public class CountryEndpoint {
    private static final String NAMESPACE_URI = "http://spring.io/guides/gs-producing-web-service";
    private CountryRepository countryRepository;

    @Autowired
    public CountryEndpoint(CountryRepository countryRepository) {
        this.countryRepository = countryRepository;
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getCountryRequest")
    @ResponsePayload
    public GetCountryResponse getCountry(@RequestPayload GetCountryRequest request) {
        GetCountryResponse response = new GetCountryResponse();
        response.setCountry(countryRepository.findCountry(request.getName()));

        return response;
    }
}

```

这才是我们真正要写的暴露出去的 webservice，用`@Endpoint`注释的时候，才会生成.WSDL文件，它里面有一个`getCountry`的方法，这个方法接收国家的名字，返回国家的货币。在前面我们定义了`response`就是`country`，翻译成xml格式返回，格式就定义在`countries.xsd`。

```

@EnableWs
@Configuration
public class WebServiceConfig extends WsConfigurerAdapter {
    @Bean
    public ServletRegistrationBean messageDispatcherServlet(ApplicationContext applicationContext) {
        MessageDispatcherServlet servlet = new MessageDispatcherServlet();
        servlet.setApplicationContext(applicationContext);
        servlet.setTransformWsdlLocations(true);
        return new ServletRegistrationBean(servlet, "/ws/*");
    }
    @Bean(name = "countries")
    public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema countriesSchema) {
        DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition();
        wsdl11Definition.setPortTypeName("CountriesPort");
        wsdl11Definition.setLocationUri("/ws");
        wsdl11Definition.setTargetNamespace("http://spring.io/guides/gs-producing-web-service");
        wsdl11Definition.setSchema(countriesSchema);
        return wsdl11Definition;
    }
    @Bean
    public XsdSchema countriesSchema() {
        return new SimpleXsdSchema(new ClassPathResource("countries.xsd"));
    }
}

```

然后这是webservice的配置，主要就是在设置`port`叫什么名字、位置在那里，唯一要注意的时候，在跑服务器的时候，在maven里，我们需要`xjb2`中的`xjc`先编译一下，才能生成辅助文件，然后再编译整个项目才能通过。

我们要写个客户端访问怎么访问呢？一定要发一个`country request`发过来才可以。注意postman中的数据格式一定要设置为`text/xml`。

The screenshot shows the Postman interface for a POST request to `http://localhost:8080/ws`. The Headers tab is selected, showing a single header `Content-Type: text/xml`. The Body tab is selected, displaying the raw XML request:

```

1 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
2   <soapenv:Header/>
3   <soapenv:Body>
4     <gs:getCountryRequest>
5       <gs:name>Spain</gs:name>
6     </gs:getCountryRequest>
7   </soapenv:Body>
8 </soapenv:Envelope>

```

The response status is 200 OK with a size of 667 B.

我们还可以使用 JAX-WS web service 来实现，更加容易。`@WebService` 会把类中的除了 `main` 的方法映射为对应的 web service，带参数的只需要使用`@WebParam(name=...)`标注一下即可。

```

@WebService
public class Warehouse {
    public Warehouse() {
        prices = new HashMap<String, Double>();
        prices.put("Blackwell Toaster", 24.95);
        prices.put("ZapXpress Microwave Oven", 49.95);
    }

    public double getPrice(@WebParam(name="description") String description)
    {
        Double price = prices.get(description.trim());
        return price == null ? 0 : price;
    }

    private Map<String, Double> prices;

    public static void main(String[] argv) {
        Object implementor = new Warehouse ();
        String address = "http://localhost:9000/Warehouse";
        Endpoint.publish(address, implementor);
    }
}

```

它在跑起来以后，会在 `http://localhost:9000/Warehouse?wsdl` 把它生成的 wsdl 文件列出来，接下来我们怎么调用呢？发一个 Soap 消息过来即可。

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:gs="http://example/"
                   >
    <soapenv:Header/>
    <soapenv:Body>
        <gs:getPrice>
            <parameters>Blackwell Toaster</parameters>
            </gs:getPrice>
        </soapenv:Body>
    </soapenv:Envelope>

    <?xml version='1.0' encoding='UTF-8'?>
    <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
        <S:Body>
            <ns2:getPriceResponse xmlns:ns2="http://example/">
                <return>24.95</return>
            </ns2:getPriceResponse>
        </S:Body>
    </S:Envelope>

```

注意红框的名字，我们都没定义过，所以直接使用了默认值。

Generate classes with `wsimport`

`wsimport -s <generate your source file location> http://localhost:9000/Warehouse?wsdl`

使用命令行就可以生成对应的文件。

怎么看待基于 SOAP 的 web service

1. 指明 `input` 和 `output message` 的格式，一旦生成 `wsdl`，那么消息格式就耦合了，一旦我们想添加一个参数，所有客户端需要重新拉一份 WSDL 文件过来重新编译一下。
2. 需要解析 SOAP 格式的纯文本，为了实现异构的系统交互，从这一点去看，webservice 的性能肯定不如两边同构语言使用远程方法调用效率来的高。一定是 `controller` 的某一部分暴露成 webservice，理由就是要和异构的系统交互。

3. 在通讯的过程中要翻译成 SOAP 消息，它的信息比较冗余。后来就有人觉得压根就不需要用它，不要去调用对方的 api，还是大家一起传输数据，只要数据的格式不变，至于 api 到底名字叫什么不重要。所以之后我们使用 restful service。

Restful Service

- REpresentational State Transfer

- Representational:

- All data are resources. Representation for client.
 - Each resource can have different representations
 - Each resource has its own unique identity(URI)

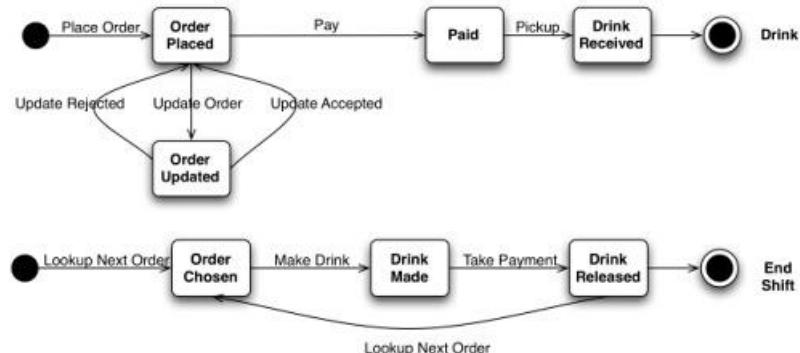
- State:

- It refers to state of client. Server is stateless.
 - The representation of resource is a state of client.

- Transfer:

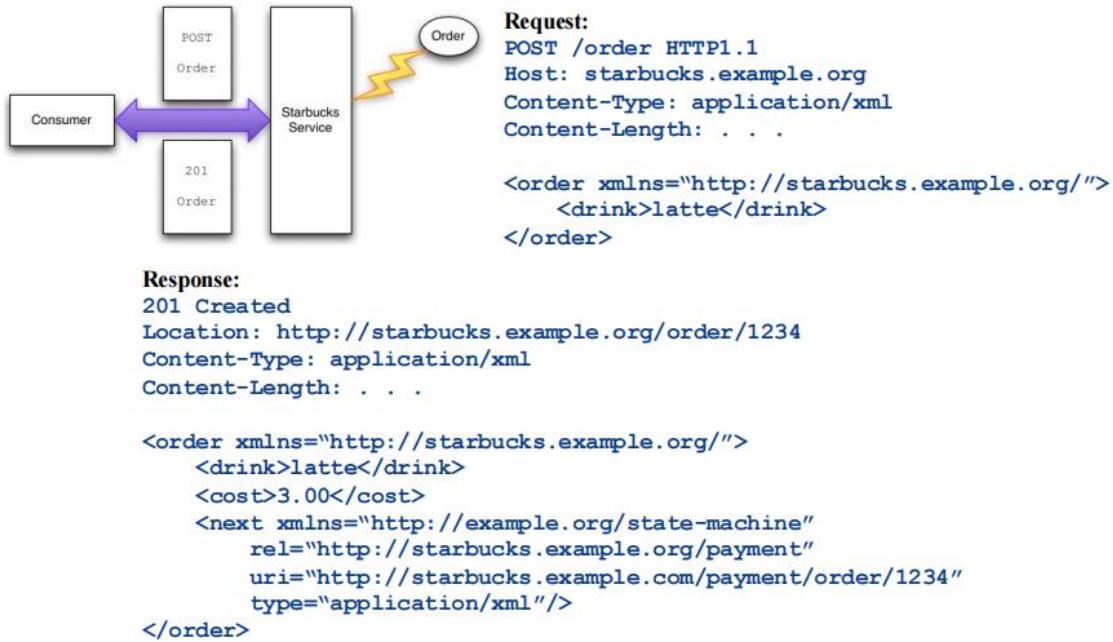
- Client's representation will be transferred when client access different resources by different URI.
 - It means the states of client are also transferred.
 - That is Representation State Transfer

所有数据都是资源，可以通过 URI 来描述。数据的描述应该推到客户端去。数据我们知道要在系统中支持增删改查四个动作，数据是在变化的，所以就认为客户端看到的数据的表示，数据构成了系统的状态，而状态是在不断地变化的。我们使用 4 个不同的 HTTP 方法(GET, PUT, POST, DELETE)，描述着客户端告诉服务器端想对数据做什么。



星巴克抽象的点咖啡操作和咖啡师的操作。一开始下了一个订单，在还没有被后端拿去做之前，还有机会修改它。存在两种情况，1：正在做，不能修改；2：还没做，可以修改。一旦做完了，付完钱就可以拿走。

如果是用 HTTP 的 restful 来描述的话：



一开始，使用 POST 方法发到 order 中，假设请求的 body 中的是用 xml 描述的。当我们创建成功之后，它会返回创建的订单的 URI（唯一标识）。我们需要就是在 order 上有个对应的 controller，监听下单请求并且更新到数据库中。

Body 中还会告诉你可以在/payment/order/1234 这个 uri 中去付款。这和大二的时候的区别是，大家全部都是基于 uri 在设计，所有的资源都是基于 uri 来的。

Request	Response
<pre> OPTIONS /order/1234 HTTP/1.1 Host: starbucks.example.org </pre>	<pre> 200 OK Allow: GET, PUT </pre>
<pre> PUT /order/1234 HTTP/1.1 Host: starbucks.example.com Expect: 100-Continue </pre>	<pre> 100 Continue </pre>

比如我们想修改这个订单，比如我们可以先 OPTIONS 知道能做 GET 和 PUT，PUT 对应的就是更新这个订单。100 表示可以继续做。

Update order

```
Request:  
PUT /order/1234 HTTP1.1  
Host: starbucks.example.com  
Content-Type: application/xml  
Content-Length: . . .  
  
<order xmlns="http://starbucks.example.org/">  
    <additions>shot</additions>  
</order>  
Response:  
200 OK  
Location: http://starbucks.example.org/order/1234  
Content-Type: application/xml  
Content-Length: . . .  
  
<order xmlns="http://starbucks.example.org/">  
    <drink>latte</drink>  
    <additions>shot</additions>  
    <cost>4.00</cost>  
    <next xmlns="http://example.org/state-machine"  
        rel="http://starbucks.example.org/payment"  
        uri="http://starbucks.example.com/payment/order/1234"  
        type="application/xml"/>  
</order>
```

返回 OK 就是修改成功了，还会告诉你数据就在这个位置，数据本身的 uri 是不会变的，但它现在的价格变成了 4 元。如果修改不成功，那就返回 409 Conflict。数据还是那个数据，不会发生变化。

同理，付钱的时候调用 /payment/order/1234 即可。我们写来写去都是在对 uri 进行操作，这是 restful service 的根本。

GreetingController.java

```
@RestController  
public class GreetingController {  
  
    private static final String template = "Hello, %s!";  
    private final AtomicLong counter = new AtomicLong();  
  
    @GetMapping("/greeting")  
    public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) {  
        return new Greeting(counter.incrementAndGet(), String.format(template, name));  
    }  
  
    @PostMapping("/greeting")  
    public Greeting greetingpost(@RequestParam(value = "name", defaultValue = "Spring") String name) {  
        return new Greeting(counter.incrementAndGet(), String.format(template, name));  
    }  
}
```

真正体现出 Restful 的地方是对于同一个 greeting 支持了 Get 和 Post，而 @RestController 只是说把返回值变成 Json 格式而已。

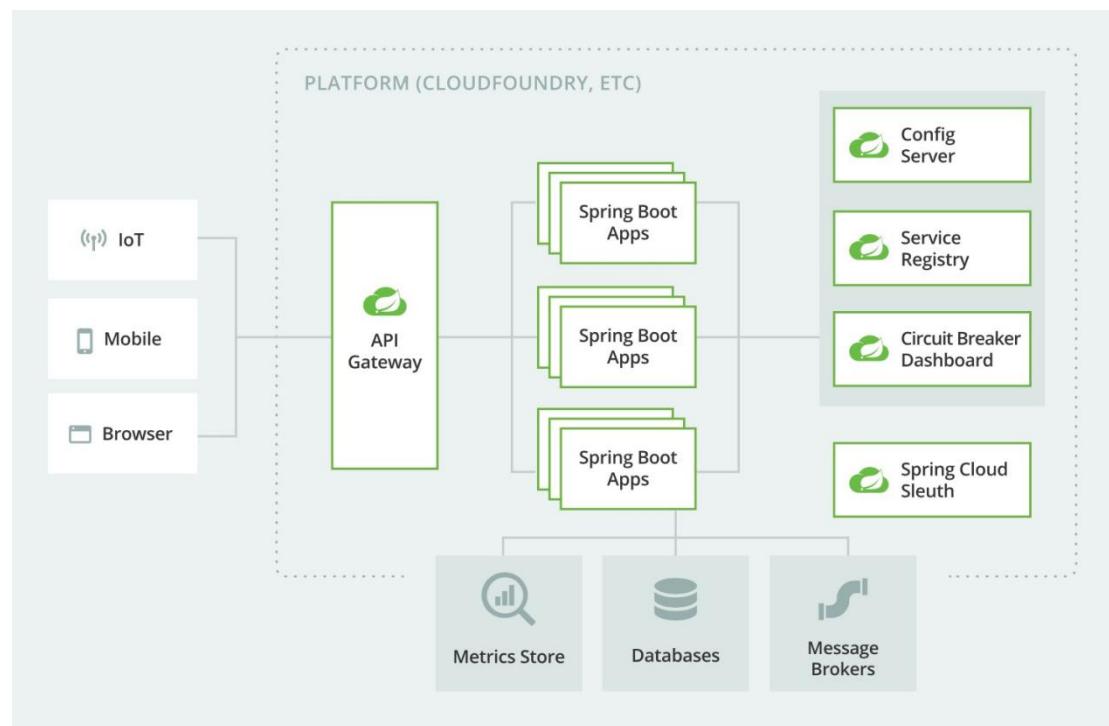
我们全部以数据为中心，因为 price、book 这种数据的名字是不容易改的，我们眼里只有数据而不是 API，数据的名字肯定比 API 的名字要不容易变，这就实现了前后端的完全解耦。Restful 更容易实现无状态的协议。

2021/10/14

微服务

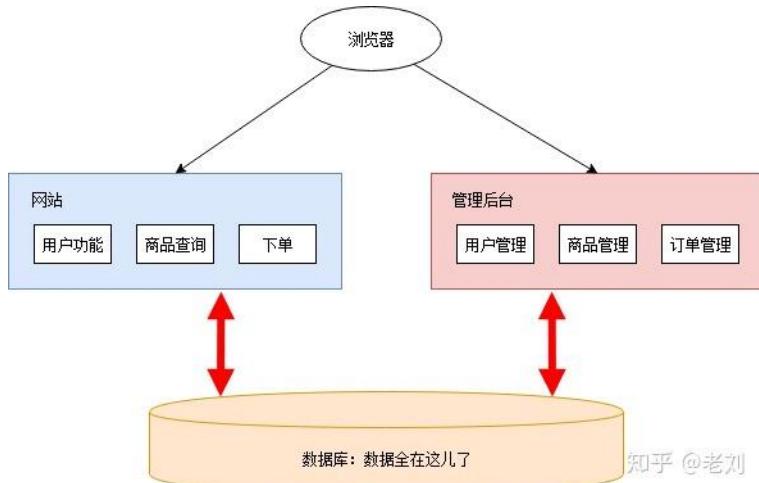
今天讲稍微复杂点的问题，也就是微服务。在我们讲到 SOAP 的时候，需要 WSDL 这个文件才能去跑，这个文件放在第三方的服务器上。也就是放在注册中心中，这就和图书馆中现在电脑中检索到书在哪里，然后再去找它。到哪去调用，就有一个路由的问题。

先说说微服务，微服务定义是就是一个很小的可以单独部署的运行代码，可以和其他模块隔离开。在 Spring 中，Spring Cloud 有一套东西来支持。换句话说，我们知道有这么一个 WSDL，我们就可以通过各种各样的维护信息来调用它，这就需要这个框架。在 Spring Cloud 中的应用有很多很多服务去构成，比如一个集群中部署了很多很多服务，Spring Cloud 会帮我们去记住，而且当服务崩的时候，spring cloud 就可以帮我自动去启动服务。

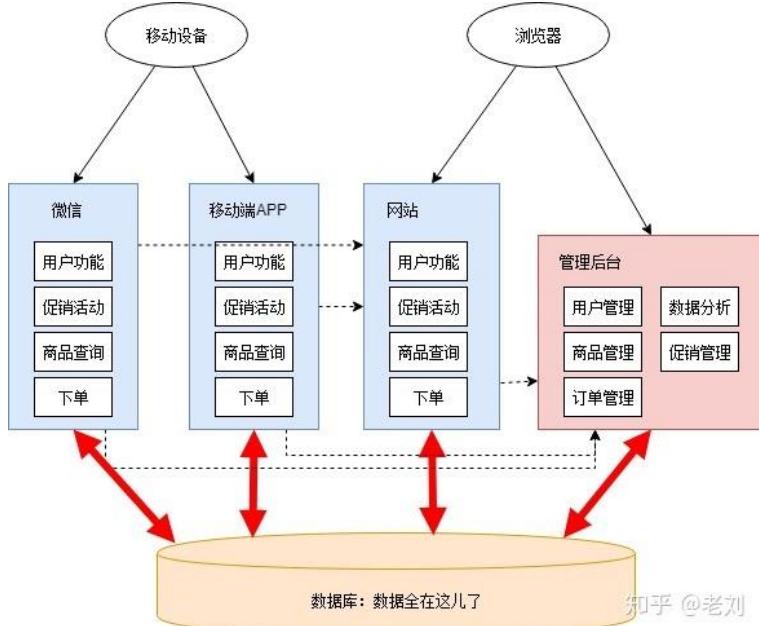


GateWay 就是一个统一入口，告诉你只要来到这里，后面的事情不需要你去管。门里面有配置服务器、注册中心、dashboard（可视化界面）、统一认证服务。我们不会去全部讲一遍，我们就讲怎么去注册服务。

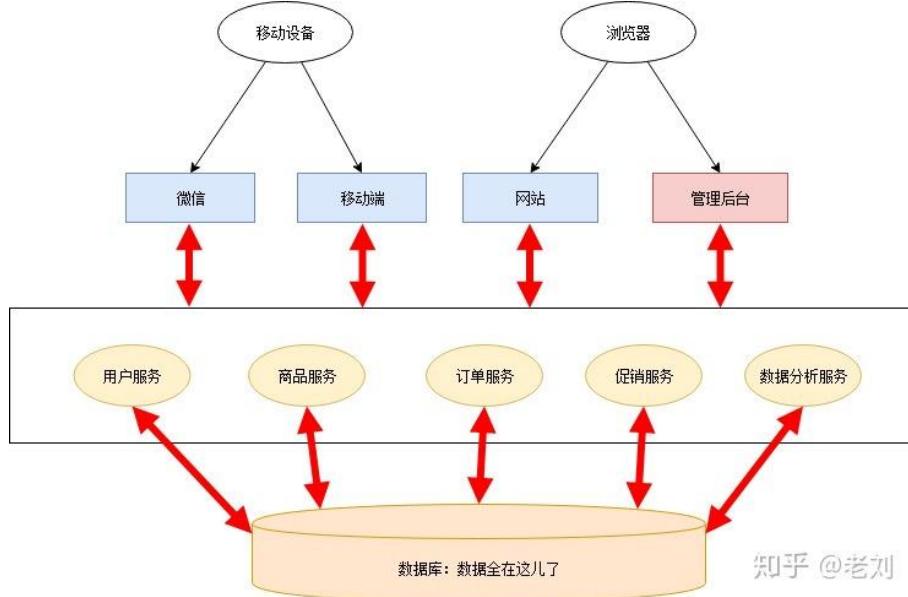
为了谈清楚为什么需要微服务，我们通过如下例子：



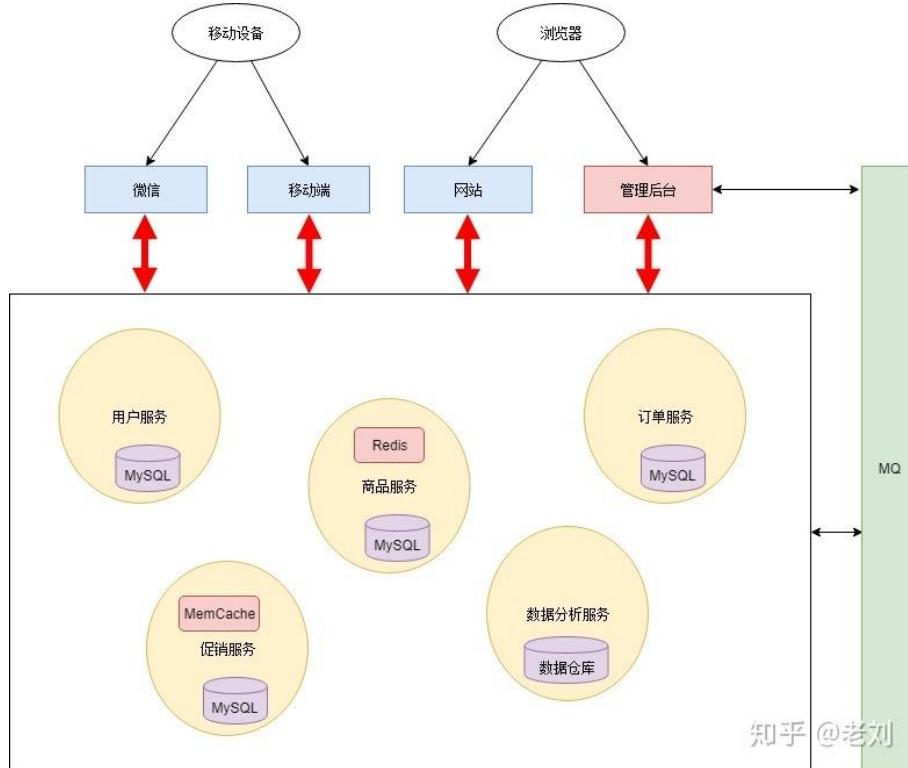
这个业务场景分成前端的下单网站和后端的管理网站。这个网站可以支持一定的用户量，它只支持浏览器页面的访问，如果我们需要支持移动端的访问，要写 app。



我们就需要重新写一遍微信的小程序、移动端的 APP。好处是这些三个前端都可以共用一个给用户的后端。因为所有的系统数据都存在一起，做数据分析的时候，可能就会很慢，同时分析的数据可能要加锁去实现，可能比较慢。无论给用户的功能还是后台要的功能，我们统一去实现，那么我们前端只是界面去做适配，功能都是统一的后台。

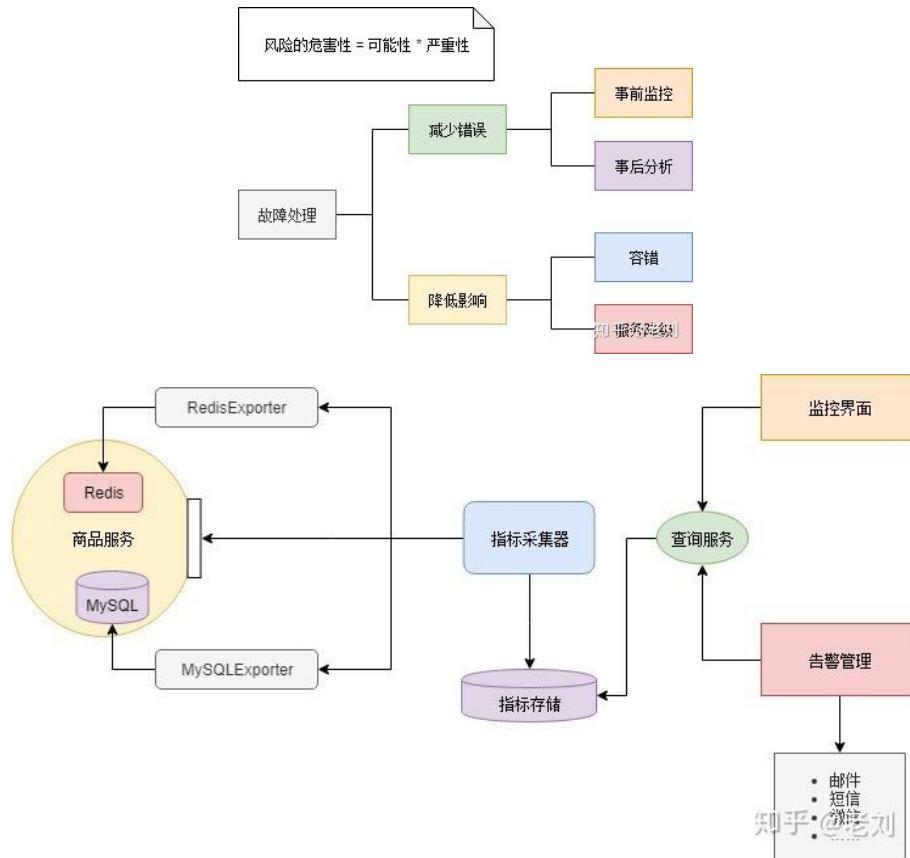


我们解决了功能的冗余，但是数据还是同一个数据库中。大家的模块之间耦合度比较高，导致数据库成为了瓶颈。

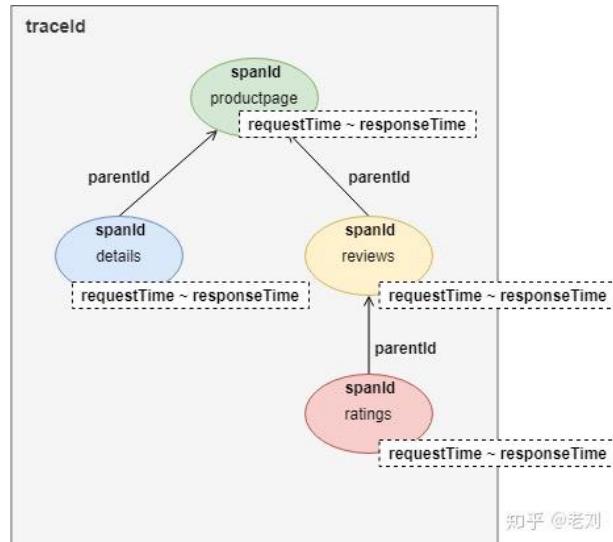


我们把模块之间的数据独立开，使用独立数据库和消息队列来做。每个模块之间的互相调用来实现业务逻辑。但是微服务之间是有依赖的，有一个服务崩掉了，可能别的服务都不能用了，我们要防止出现这种问题。出现故障分为：

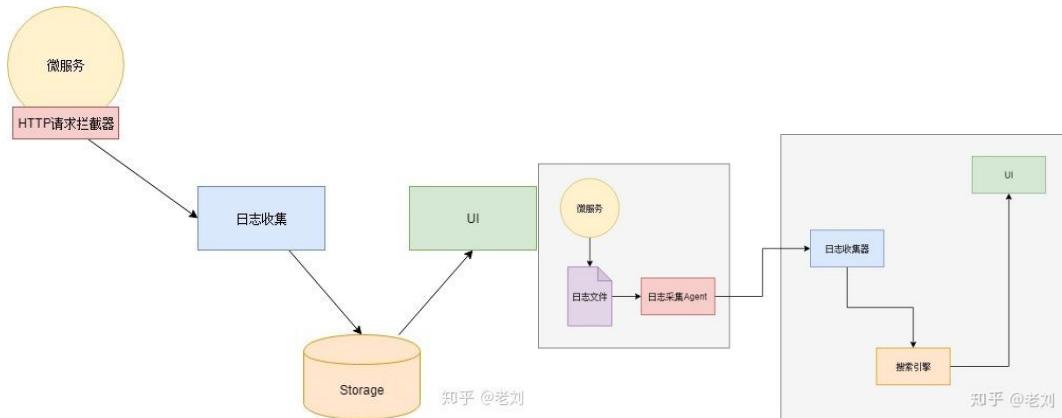
1. 尽量让微服务不要出故障，我们需要对数据做监控。
2. 在运行时刻确实产生了崩溃，我们能不能降低它的影响（使用容错副本马上接管，其他服务采用降级模式）



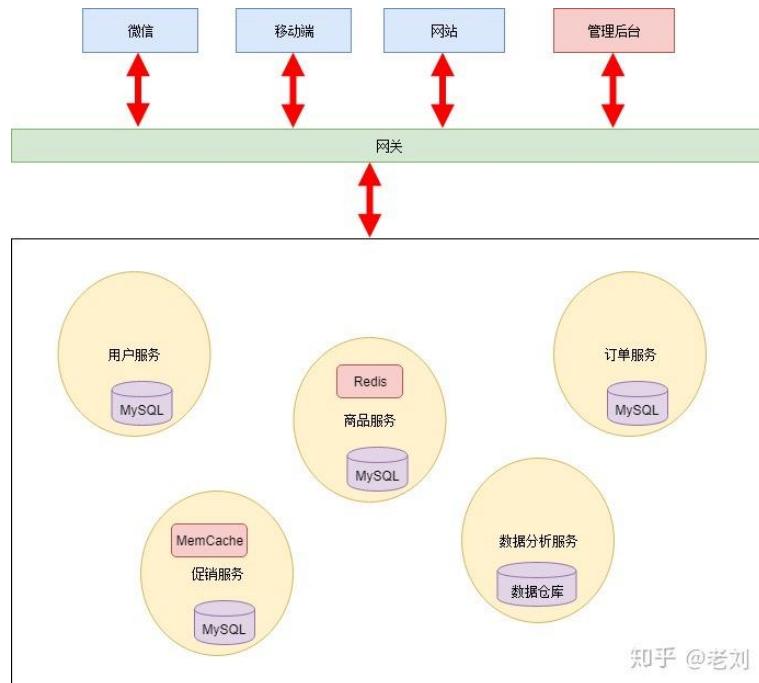
真正出现问题的时候，服务之间是在做互相调用，我们需要工具去追踪这个链路的问题。



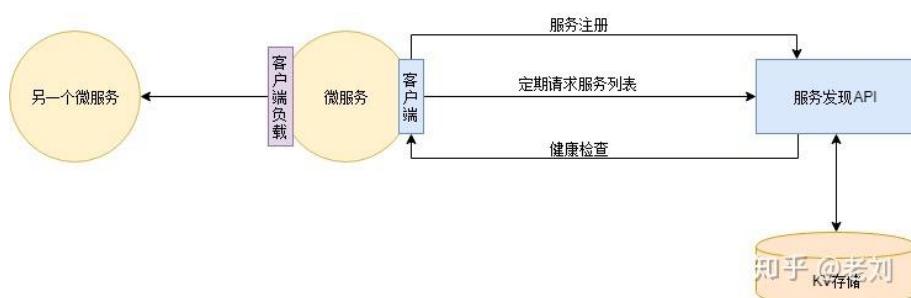
刚才说到日志的问题，比如微服务的所有 HTTP 请求，都可以在发过来的时候记录一下。



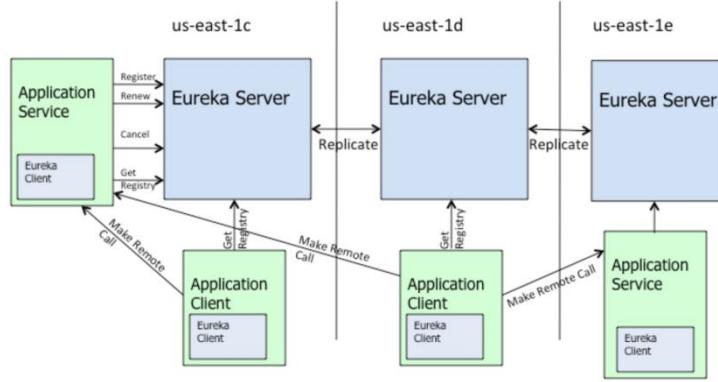
所以在整个微服务的环境中，有很多服务的工具帮我们做系统的管理。我们还需要解决前端的代码在访问的时候，我们需要知道这些服务在哪里。假设我们的三个服务分别在 8040,8080,8090 上，这就是一个网关帮我们做的事情，对上提供统一的访问地址，屏蔽掉底层的服务到底在哪个端口。



这些微服务需要把自己存到注册表中去，告诉注册表在哪个端口提供服务，但是注册表中的服务要是活着的服务，所以要求服务定期地 renew 去注册表。



注册表我们使用的是网飞的 Eureka，它就是一个服务注册中心，如果我们用的是 docker，可以用 k8s 来做管理。服务注册中心就是帮我们定位在当前环境中有多少个服务，它甚至可以去做负载均衡。



一台台机器分为了一个个 region，每个 region 中就会放一个 Eureka Server，其中就是在用 key-value 去存提供的服务。注意上面 Eureka Server 中是 replicate 的关系。

跑一个 Eureka 的 server 很简单：

Main java class

```
@EnableEurekaServer
@SpringBootApplication
public class ServiceRegistrationAndDiscoveryServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(
            ServiceRegistrationAndDiscoveryServiceApplication.class, args);
    }
}
```

resources/application.properties

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
logging.level.com.netflix.eureka=OFF
logging.level.com.netflix.discovery=OFF
```

```
@RestController
class ServiceInstanceRestController {

    @Autowired
    private DiscoveryClient discoveryClient;

    @RequestMapping("/service-instances/{applicationName}")
    public List<ServiceInstance> serviceInstancesByApplicationName(
        @PathVariable String applicationName) {
        return this.discoveryClient.getInstances(applicationName);
    }
}
```

这样就可以把自己注册到 Eureka 的注册应用中去。注意还要配置 spring.application.name=a-bootiful-client。这个应用的做的事情就访问自己的时候，找到对应应用名的实例然后返回。

然后我们访问网址 <http://localhost:8080/service-instances/a-bootiful-client> 的时候，就可以看到：

```
{
  "instanceId": "172.20.10.6:a-bootiful-client", "serviceId": "A-BOOTIFUL-CLIENT", "uri": "http://172.20.10.6:8080", "instanceInfo": {
    "instanceId": "172.20.10.6:a-bootiful-client", "app": "A-BOOTIFUL-CLIENT", "appGroupName": null, "ipAddr": "172.20.10.6", "sid": "na", "homePageUrl": "http://172.20.10.6:8080/", "statusPageUrl": "http://172.20.10.6:8080/actuator/info", "healthCheckUrl": "http://172.20.10.6:8080/actuator/health", "secureHealthCheckUrl": null, "vipAddress": "a-bootiful-client", "secureVipAddress": "a-bootiful-client", "countryId": 1, "dataCenterInfo": {
      "@class": "com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo", "name": "MyOwn"}, "hostName": "172.20.10.6", "status": "UP", "overriddenStatus": "UNKNOWN", "leaseInfo": {
        "renewalIntervalInSecs": 30, "durationInSecs": 90, "registrationTimestamp": 1621058604835, "lastRenewalTimestamp": 1621058604835, "evictionTimestamp": 0, "serviceUpTimestamp": 1621058604235}, "isCoordinatingDiscoveryServer": false, "metadata": {
        "management.port": "8080"}, "lastUpdatedTimestamp": 1621058604835, "lastDirtyTimestamp": 1621058604180, "actionType": "ADDED", "asgName": null}, "scheme": "http", "host": "172.20.10.6", "port": 8080, "metadata": {"management.port": "8080"}, "secure": false}
}
```

至少这个例子在告诉我们，要跑微服务先要跑 Eureka，并且把自己的服务注册到 Eureka 中。

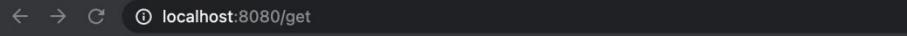
我们刚刚说的 gateway 的问题怎么解决。

```
Main Java Class
@SpringBootApplication
public class GatewayApplication {
    @Bean
    public RouteLocator myRoutes(RouteLocatorBuilder builder) {
        return builder.routes()
            // Add a simple re-route from: /get to: http://httpbin.org:80
            // Add a simple "Hello:World" HTTP Header
            .route(p -> p
                .path("/get") // intercept calls to the /get path
                .filters(f -> f.addRequestHeader("Hello", "World")) // add header
                .uri("http://httpbin.org:80")) // forward to httpbin
            .build();
    }

    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```

这个 gateway 实际上在做，如果访问的是/get 路径，就可以对 Header 做处理，比如做安全检查，gateway 做完处理以后再去做转发。

此时我们访问 localhost:8080/get 的时候，就可以接收到我们的 request 请求的结果：



```
{
  "args": {},
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
    "Accept-Encoding": "gzip, deflate, br",
    "Accept-Language": "zh-CN,zh;q=0.9,en;q=0.8",
    "Cache-Control": "max-age=0",
    "Content-Length": "0",
    "Forwarded": "proto=http;host=\"localhost:8080\";for=\"0:0:0:0:0:0:1:57682\"",
    "Hello": "World",
    "Host": "httpbin.org", "Host": "httpbin.org",
    "Sec-Ch-Ua": "\" Not A;Brand\";v=\"99\", \"Chromium\";v=\"90\", \"Google Chrome\";v=\"90\"",
    "Sec-Ch-Ua-Mobile": "?0",
    "Sec-Fetch-Dest": "document",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-Site": "none",
    "Sec-Fetch-User": "?1",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.212 Safari/537.36",
    "X-Amzn-Trace-Id": "Root=1-609f744e-42b239bf33e7115d666cf18a",
    "X-Forwarded-Host": "localhost:8080"
  },
  "origin": "0:0:0:0:0:1, 223.104.213.94",
  "url": "http://localhost:8080/get"
}
```

微服务的位置就可以被 gateway 管理起来，不需要指定具体服务的位置。
我们可以继续看网飞的 Zuul Gateway

- Main Java Class

```
@RestController  
@SpringBootApplication  
public class RoutingAndFilteringBookApplication {  
  
    @RequestMapping(value = "/available")  
    public String available() {  
        return "Spring in Action";  
    }  
  
    @RequestMapping(value = "/checked-out")  
    public String checkedOut() {  
        return "Spring Boot in Action";  
    }  
  
    public static void main(String[] args) {  
        SpringApplication.run(RoutingAndFilteringBookApplication.class, args);  
    }  
}
```

这是一个很简单的 web 工程，开在 8090 端口。如果我们直接到 8090，当然我们是可以访问到的。接下来我们来看 gateway 的实现：

```
public class SimpleFilter extends ZuulFilter {  
  
    private static Logger log = LoggerFactory.getLogger(SimpleFilter.class);  
    @Override  
    public String filterType() {  
        return "pre";  
    }  
  
    @Override  
    public int filterOrder() {  
        return 1;  
    }  
  
    @Override  
    public boolean shouldFilter() {  
        return true;  
    }  
  
    @Override  
    public Object run() {  
        RequestContext ctx = RequestContext.getCurrentContext();  
        HttpServletRequest request = ctx.getRequest();  
  
        log.info(String.format("%s request to %s", request.getMethod(), request.getRequestURL().toString()));  
        return null;  
    }  
}
```

34

zuul.routes.books.url=http://localhost:8090

ribbon.eureka.enabled=false

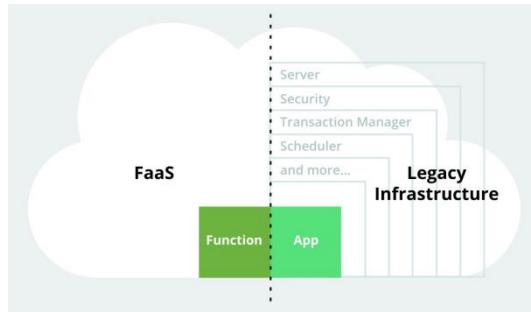
server.port=8080

然后我们发现，直接访问 8090 当然可以，我们也可以通过 8080/book 来转发到对应的服务器上：

- You can access the book application
 - directly at `localhost:8090/available` and
 - through the Gateway service at `localhost:8080/books/available`.
- Visit one of the Book service endpoints
 - (`localhost:8080/books/available` or `localhost:8080/books/checked-out`)
 - and you should see your request's method logged by the Gateway application before it is handed on to the Book application, as the following sample logging output shows:

```
2019-10-02 10:58:34.694 INFO 11608 --- [nio-8080-exec-4] c.e.r.filters.pre.SimpleFilter : GET
request to http://localhost:8080/books/available
```

如果我们的微服务部署在云里，云现在提供函数式编程。



云提供 **Serverless**，我们不需要去处理事务管理、安全管理，云的基础设施帮我们搞定，我们只需要关注业务开发。整个服务应该是无状态的。如果云提供的环境足够好的话，我们应用特有的逻辑就可以只写成函数。

- [Spring Cloud Function](#)
 - provides capabilities that lets Spring developers take advantage of serverless or FaaS platforms.
- Spring Cloud Function
 - provides adaptors so that you can run your functions on the most common FaaS services including :
 - [Amazon Lambda](#), [Apache OpenWhisk](#), [Microsoft Azure](#), and [Project Riff](#).
- Spring Cloud Function is a project with the following high-level goals:
 - Promote the implementation of business logic via functions.
 - Decouple the development lifecycle of business logic from any specific runtime target so that the same code can run as a web endpoint, a stream processor, or a task.
 - Support a uniform programming model across serverless providers, as well as the ability to run standalone (locally or in a PaaS).
 - Enable Spring Boot features (auto-configuration, dependency injection, metrics) on serverless providers.

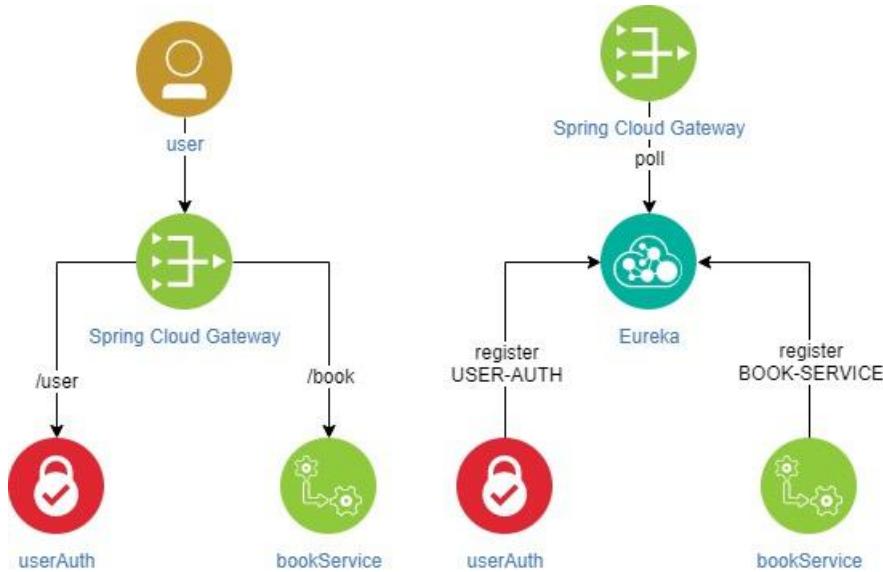
这时候我们就看到了它是一个 **serverless** 的应用。

```
@SpringBootApplication
public class FunctionsampleApplication {

    @Bean
    public Function<Flux<String>, Flux<String>> uppercase() {
        return flux -> flux.map(value -> value.toUpperCase());
    }

    public static void main(String[] args) {
        SpringApplication.run(FunctionsampleApplication.class, args);
    }
}
```

Function 应用，里面都是 **flux** 包装起来的东西。这个说的就是传入的是一个字符串，输出一个字符串。



这四个工程都要跑一下，先要跑起来 Eureka 和 Gateway。两个服务会注册到 Eureka 中，可以通过 gateway 去访问。Gateway 收到服务请求的时候可以去 Eureka 中去查一下是否还存活，如果存活那就转发请求。

2021/10/18

对资源的增删改查是适合 restful 的，而 SOAP 适合于不变的 API，比如我们实现了一个计算器，那么它就不适用使用 restful api，因为不是对资源做操作。我们实现 js+java 的前端传 json、使用 restful controller 的时候，不是已经解耦了吗？这已经是一个 restful webservice 了吗？如果我们已经实现了增删改查的 controller，这还不是 restful webservice，根据 rest 的定义来说这不行，因为 HTTP 属于一个纯粹的传输层的协议，而这样我们是在用 HTTP 作为应用层的协议，这不符合 restful 的设计理念。

上节课我们写了一个 book 的微服务，

其实认证的逻辑也是 controller 在验证，它在把自己注册到 eureka 上去的时候，注册了对 Auth 引用。

```

eureka:
  instance:
    prefer-ip-address: true
    ip-address: localhost
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:8040/eureka
  eureka-service-url-poll-interval-seconds: 10

```

而 gateway 说的是所有的连接过来以后，用 GET 和 POST 过来的方法全部要拦截掉。直接访问对应 Controller 的端口，如果重启了就可能会修改端口要求修改前端的代码，但是通过 gateway 拦截访问的时候就可以统一的位置来访问。

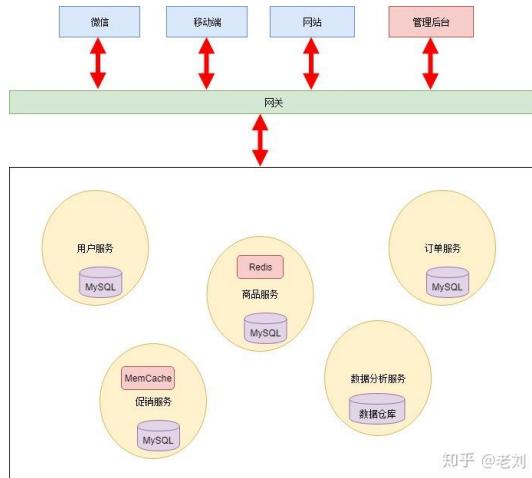
```

    @SpringBootApplication
    @EnableDiscoveryClient
    @EnableEurekaClient
    public class MainGateway {
        public static void main(String[] args) {
            SpringApplication.run(MainGateway.class, args);
        }
        @Autowired
        JwtCheckFilter jwtCheckFilter;
        @Bean
        public RouteLocator myRoutes(RouteLocatorBuilder builder) {
            return builder.routes()
                .route(r -> r.path("/book/**")
                    .filters(f -> f.rewritePath("/book", "").filter(jwtCheckFilter))
                    .uri("lb://BOOK-SERVICE"))
                .route(r->r.path("/user/**")
                    .filters(f->f.rewritePath("/user", ""))
                    .uri("lb://USER-AUTH"))
            ).build();
        }
    }

```

如果 localhost 的 url 带了 /book/**，找到 book service，把 book 这一段替换成空的，那就相当于 localhost:8080/book/buybook，它就会替换为 localhost:<bookservice's port>/buybook。如果我们不是在一台机器上，而是一个分布式的微服务，那么可能 localhost 的位置也会替换成服务部署的机器的 ip。我们发现无论 book service 部署在哪里，它会自动注册到 eureka 上，所以部署位置变化的时候，前端代码、gateway 的代码都不需要变化。Gateway 实现了对用户屏蔽掉后面服务的位置，gateway 如果再复杂一点，可以做到不同协议和数据格式（json、键值对、xml）的转换，这样我们甚至可以不需要管服务具体使用的协议是什么。做的更加复杂、商业化后，就把 gateway 做成了服务总线。

现在我们实现了服务的注册、查找和路由，那么什么叫微服务呢？它其实就是一个 controller，没什么特殊的东西。所以微服务实际上是一个概念上的东西，



我们完全可以不按照这样去定义微服务，微服务的划分粒度是我们自己定义的。到底粒度多大取决于我们和业务场景。微服务只是说它是一个功能上相对独立的一块，不是一个完整的应用。

Spring Security

这节课讲讲安全。

```

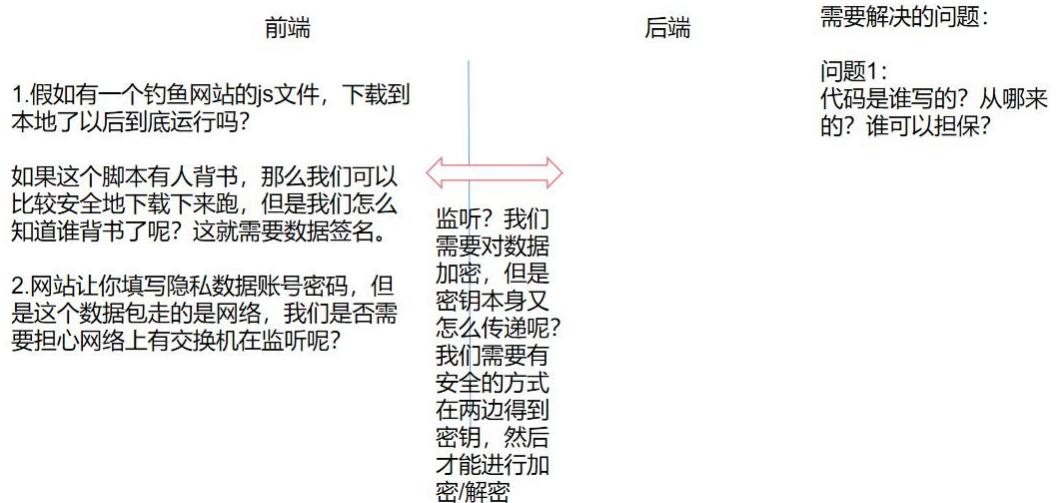
@Configuration
public class MvcConfig implements WebMvcConfigurer {

    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/home").setViewName("home");
        registry.addViewController("/").setViewName("home");
        registry.addViewController("/hello").setViewName("hello");
        registry.addViewController("/login").setViewName("login");
    }
}

@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/", "/home").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login")
            .permitAll()
            .and()
            .logout()
            .permitAll();
    }
}

```

这四个连接，点完就显示对应的内容，`login` 这里也是要做认证的，认证完才有全部的权限。`logout` 是没有权限验证的。Spring Security，就是在这个配置类写谁需要做什么认证。



首先，我们需要解决，会不会有人在网络上监听，拦截完代码然后添加了一段钓鱼脚本呢？我们该怎么解决这个问题呢？我们需要知道是否有人篡改过，我们需要生成一个消息摘要（message digest）。它 20 个字节，一共 160bit，它是用算法 SHA1(secure hash algorithm #1) 去构成的，算法的特点是如果我们修改了一个 bit，修改前后生成的消息摘要也会大相径庭。

Consider the following message by the billionaire father:

- "Upon my death, my property shall be divided equally among my children; however, my son **George** shall receive nothing."
- That message has an SHA1 fingerprint of
 - 2D 8B 35 F3 BF 49 CD B1 94 04 E0 66 21 2B 5E 57 70 49 E1 7E
- Now, suppose **George** wants to change the message so that **Bill** gets nothing. That changes the fingerprint to a completely different bit pattern:
 - 2A 33 0B 4B B3 FE CC 1C 9D 5C 01 A7 09 51 0B 49 AC 8F 98 92

```

MessageDigest alg = MessageDigest.getInstance("SHA-1");

InputStream in = ...
int ch;
while ((ch = in.read()) != -1)
    alg.update((byte) ch);

byte[ ] bytes = ...;
alg.update(bytes);

byte[ ] hash = alg.digest();

```

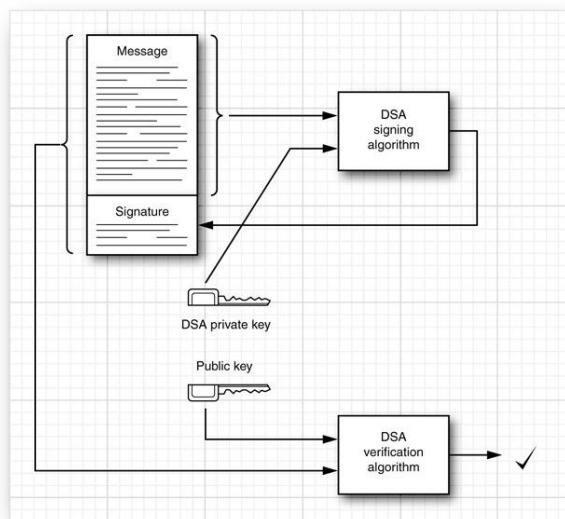
加密解密的包在 java 中都是工厂格式。

但是摘要本身不能被篡改，那么我们需要对它加密一下。如果我们拿个 `key` 对 `digest` 进行加密，如果加密解密用的是同一个 `key`，那么还是没用。所以我们需要使用公钥/密钥这种不对称的密钥对。

- The keys are quite long and complex. For example, here is a matching pair of public and private Digital Signature Algorithm (DSA) keys.
 - Public key:
 - Code View:
 - `p: fca682ce8e12cabab26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17 q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5g:678471b27a9cf44ee91a49c5147db1a9aaaf244f05a434d6486931d2d14271b9e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4 y: c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a8123ce5a8018b8161a760480fadd040b927281ddb22cb9bc4df596d7de4d1b977d50`
 - Private key:
 - Code View:
 - `p: fca682ce8e12cabab26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17 q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5g:678471b27a9cf44ee91a49c5147db1a9aaaf244f05a434d6486931d2d14271b9e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4 x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a`

公钥、私钥的特征：

如果我们使用私钥对信息加密，只能拿公钥来解密；同样的，如果使用公钥来加密信息，只能使用私钥去解密；公钥和私钥是不能互相靠对方计算出来的。



遗嘱生成了前面，拿私钥去加密了 digest，这就叫签名，把公钥给律师，当修改遗嘱重新生成 digest 后，因为它们没有私钥，就不能对 digest 加密，到时候拿公钥解密解不开，那就发现遗嘱被人篡改过了。这就可以解决把律师买通的问题了。所以就是需要把公钥分发给所有的相关人。

- To take advantage of public key cryptography, the public keys must be distributed.
 - One of the most common distribution formats is called X.509.
- The **keytool** program manages keystores, databases of certificates and private/public key pairs.
 - Each entry in the keystore has an alias.
 - Here is how Alice creates a keystore, alice.certs, and generates a key pair with alias alice.
 - `keytool -genkeypair -keystore alice.certs -alias alice`

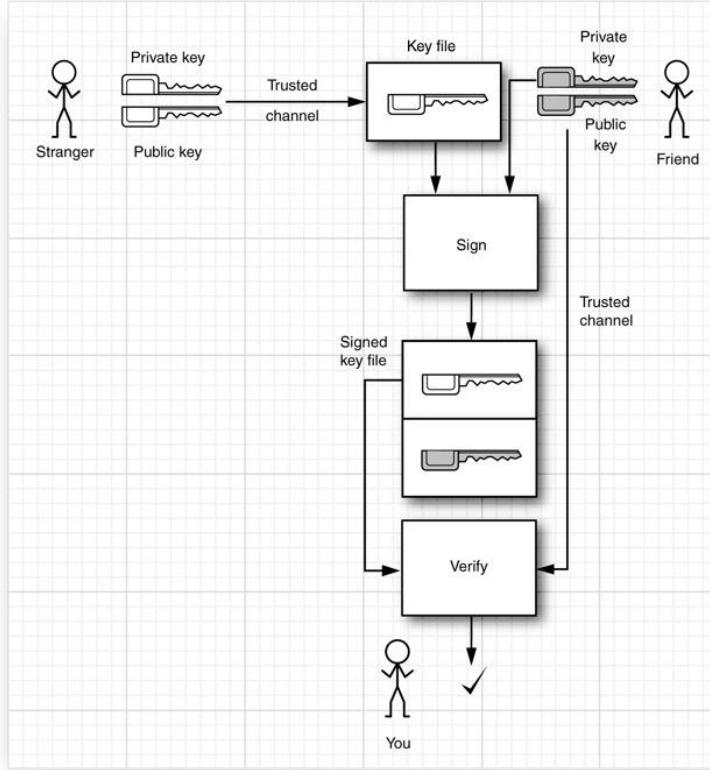
在 jdk 中有一个 **keytool** 工具，可以通过命令来生成密钥。它会问你一堆东西需要去回答。这些回答会参与到生成密钥的过程中。这样哪怕是同一个时间、同一台机器上跑的两个程序，也会因为问题回答的不一样而不一样。

- Once Bob trusts the certificate, he can import it into his keystore.
 - `keytool -importcert -keystore bob.certs -alias alice -file alice.cer`
- Now Alice can start sending signed documents to Bob.
 - `jar cvf document.jar document.txt`
 - `jarsigner -keystore alice.certs document.jar alice`
- When Bob receives the file, he uses the `-verify` option of the jarsigner program.
 - `jarsigner -verify -keystore bob.certs document.jar`
- If the JAR file is not corrupted and the signature matches, then the jarsigner program prints
 - `jar verified.`
 - Otherwise, the program displays an error message.

在证书确实可信的情况下，**bob** 可以导入这个 **Alice** 的公钥。**Alice** 就可以对它需要的文件进行签名，然后发送给 **Bob** 以后，**Bob** 就可以拿持有的公钥尝试解码。在这个过程中，**document** 本身是不加密的，我们目前的目的是让对方知道这个文件确实是我发给你的。如果是加密内容的话，**Alice** 应该把它需要发送的文档使用 **Bob** 的公钥进行加密，这样只有 **Bob** 才能使用自己的私钥进行解密。

CA（证书颁发机构）

怎么 assert 证书是正确的呢？打电话问显然很麻烦，但是如果有一个共同的朋友来分发这个公钥就可以相信了。



陌生人先把自己的公钥给到朋友，然后朋友拿到这个陌生人的公钥拿自己的私钥签了一个名，我们拿到以后就可以拿朋友的公钥解开朋友的前面，我们对比一个收到的陌生人的私钥和朋友签名完解密出来的私钥，如果两个文件的 digest 文件对比成功，那么就通过校验。这可以使用自动化的工具来完成整套流程。

那么谁愿意当这个共有朋友呢？比如 Verisigner 公司。证书分成了很多级别，浏览器有时候会提示不可靠的网站，因为证书级别很低，要提升证书级别就需要交钱。

- Suppose Alice wants to send her colleague Cindy a signed message
 - but Cindy doesn't want to bother with verifying lots of signature fingerprints.
 - Now suppose that there is an entity that Cindy trusts to verify signatures. In this example, Cindy trusts the Information Resources Department at ACME Software.
- That department operates a **certificate authority (CA)**.
 - Everyone at ACME has the CA's public key in their keystore, installed by a system administrator who carefully checked the key fingerprint.
 - The CA signs the keys of ACME employees.
 - When they install each other's keys, then the keystore will trust them implicitly because they are signed by a trusted key.

上例中 CA 就是这个共有的朋友。

可以模拟整个流程：

Here is how you can simulate this process.

- Create a keystore `acmesoft.certs`.
- Generate a key pair and export the public key:
 - `keytool -genkeypair -keystore acmesoft.certs -alias acmeroot`
 - `keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer`
- The public key is exported into a "self-signed" certificate.
- Then add it to every employee's keystore.
 - `keytool -importcert -keystore cindy.certs -alias acmeroot -file acmeroot.cer`
- An authorized staff member at ACME Software would verify Alice's identity and generate a signed certificate as follows:
 - `java CertificateSigner -keystore acmesoft.certs -alias acmeroot -infile alice.cer -outfile alice_signedby_acmeroot.cer`
- Now Cindy imports the signed certificate into her keystore:
 - `keytool -importcert -keystore cindy.certs -alias alice -file alice_signedby_acmeroot.cer`

倒数第二个命令是用库中的某个 CA 私钥对 Alice 的公钥签名。发送给 Cindy 以后，Cindy 可以使用最后一个命令把经过签名的文件导入到密钥库中。在这个过程中，就要进行校验，也就是看是否是信任的人的公钥能够解开的密钥。这就是实现了陌生人之间的信息的传递。

如果我们传递的是代码的话，也是一样的，比如 Google 对自己写的代码签个名，这就说明确实是 google 的。

我们要对数据加密是要用对象的公钥加密以后再传给对象。真正在开发 web 程序的时候，如果我们把用户的公钥都拿过来，可能就很麻烦。对称密钥的速度就很快，而非对称密钥体系速度就很慢。如果谁访问我，就要把对象的公钥就导入的话，很麻烦、性能也很差，这该怎么解决呢？我们就可以，先使用非对称密钥通信（公钥/私钥）传递一个对称密钥，之后双方都是用对称密钥对数据进行加密/解密。在对称密钥分发的过程中，是安全的，哪怕有人截获了他也解不开，不知道传递的是什么。

Symmetric Ciphers

- Cipher
 - Cipher cipher = Cipher.getInstance(algorithmName);
 - or
 - Cipher cipher = Cipher.getInstance(algorithmName, providerName);
 - The JDK comes with ciphers by the provider named "SunJCE".
 - The algorithm name is a string such as "AES" or "DES/CBC/PKCS5Padding".
 - int mode = ...; Key key = ...; cipher.init(mode, key);
 - The mode is one of
 - Cipher.ENCRYPT_MODE
 - Cipher.DECRYPT_MODE
 - Cipher.WRAP_MODE
 - Cipher.UNWRAP_MODE
- ```
int blockSize = cipher.getBlockSize();
byte[] inBytes = new byte[blockSize];
... // read inBytes
int outputSize= cipher.getOutputSize(inLength);
byte[] outBytes = new byte[outputSize];
int outLength = cipher.update(inBytes, 0, outputSize, outBytes);
... // write outBytes

outBytes = cipher.doFinal(inBytes, 0, inLength);
– Or
outBytes = cipher.doFinal();
– The call to doFinal is necessary to carry out padding of the final block.
L 01 if length(L) = 7
L 02 02 if length(L) = 6
L 03 03 03 if length(L) = 5
...
L 07 07 07 07 07 07 07 07 if length(L) = 1
08 08 08 08 08 08 08 08
```

通过它加密的文件长度都是 8 的整数倍，如果不够就需要补齐。

Follow these steps:

- Get a KeyGenerator for your algorithm.
  - Initialize the generator with a source for randomness. If the block length of the cipher is variable, also specify the desired block length.
  - Call the generateKey method.
- ```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecureRandom random = new SecureRandom();
keygen.init(random);
Key key = keygen.generateKey();
Or
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("AES");
byte[] keyData = ...; // 16 bytes for AES
SecretKeySpec keySpec = new SecretKeySpec(keyData, "AES");
Key key = keyFactory.generateSecret(keySpec);
```

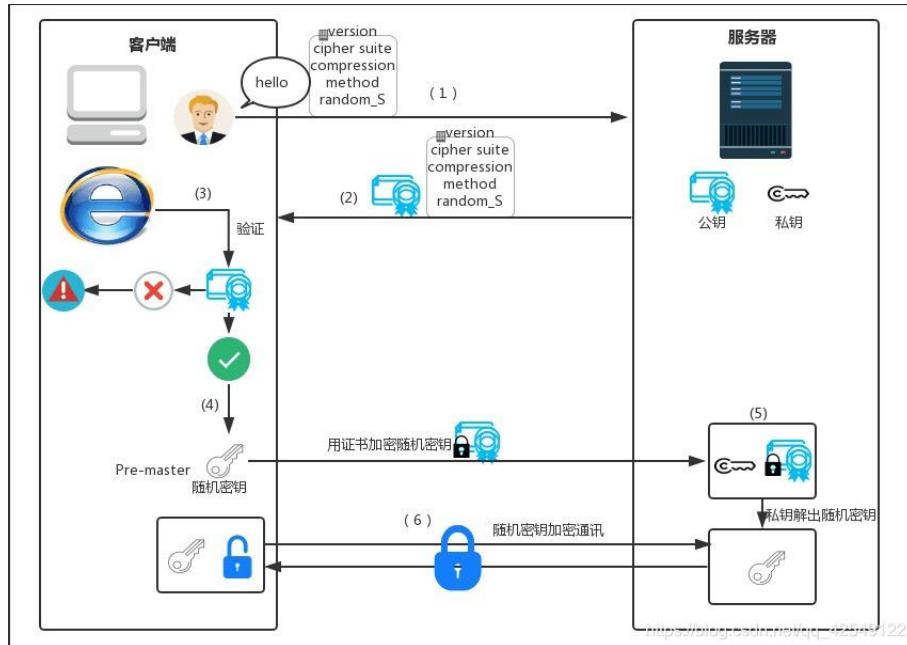
对称加密算法比如 AES，在生成实例的时候，一般要传递一个随机数。我们可以使用

SecureRandom 来生成更加安全的随机数。

之前所提到的第一次非对称加密、后面全部对称加密的方法如下：

This problem can easily be overcome by combining a public key cipher with a fast symmetric cipher, like this:

- Alice generates a random symmetric encryption key. She uses it to encrypt her plaintext.
- Alice encrypts the symmetric key with Bob's public key.
- Alice sends Bob both the encrypted symmetric key and the encrypted plaintext.
- Bob uses his private key to decrypt the symmetric key.
- Bob uses the decrypted symmetric key to decrypt the message.



我们在 url 中看到的 https 其实就是这个方法。https 其实就是在 http 上，通过加密机制搞一个安全的 socket 层，简称 SSL (secure sockets layer)。

它内部过程是，客户端告诉服务器要开始通讯了，客户端验证服务器端传过来的公钥。如果信任，那么客户端就用服务器的公钥加密随机对称密钥给服务器，服务器能解开说明没有人篡改过。公钥/私钥体系可以验证我是谁。

配置服务器的私钥

Configuration in Tomcat



- Create a keystore file to store the server's private key and self-signed certificate by executing the following command:
 - Windows:
“%JAVA_HOME%\bin\keytool” -genkey -alias tomcat -keyalg RSA -keystore “C:\Tomcat\conf\key\tomcat.keystore” -validity 365
 - Unix:
\$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA -keystore ./conf/key/tomcat.keystore -validity 365

- For Spring nested Tomcat -> Edit application.properties

```
server.port=8443  
server.ssl.key-store=/Users/chenhaopeng/apache-tomcat-9.0.31/conf/key/tomcat.keystore  
server.ssl.key-store-password=changeit  
server.ssl.keyAlias=tomcat
```

- Spring-boot Project

```
@SpringBootApplication  
public class DemoApplication {  
  
    @Bean  
    public Connector connector(){  
        Connector connector=new Connector("org.apache.coyote.http11.Http11NioProtocol");  
        connector.setScheme("http");  
        connector.setPort(8080);  
        connector.setSecure(false);  
        connector.setRedirectPort(8443);  
        return connector;  
    }  
  
    public static void main(String[] args) {  
        SpringApplication.run(DemoApplication.class, args);  
    }  
}
```

对 8080 端口的访问都会导向到 8443 端口（部署了 SSL 上）

Single Sign-On - wikipedia

REin
REliable, INtelligent & Scalable Systems

- **Single sign-on (SSO)** is a property of access control of multiple related, but independent software systems.
 - With this property a user logs in once and gains access to all systems without being prompted to log in again at each of them.
 - Conversely, **Single sign-off** is the property whereby a single action of signing out terminates access to multiple software systems.
- As different applications and resources support different authentication mechanisms,
 - single sign-on has to internally translate to and store different credentials compared to what is used for initial authentication.

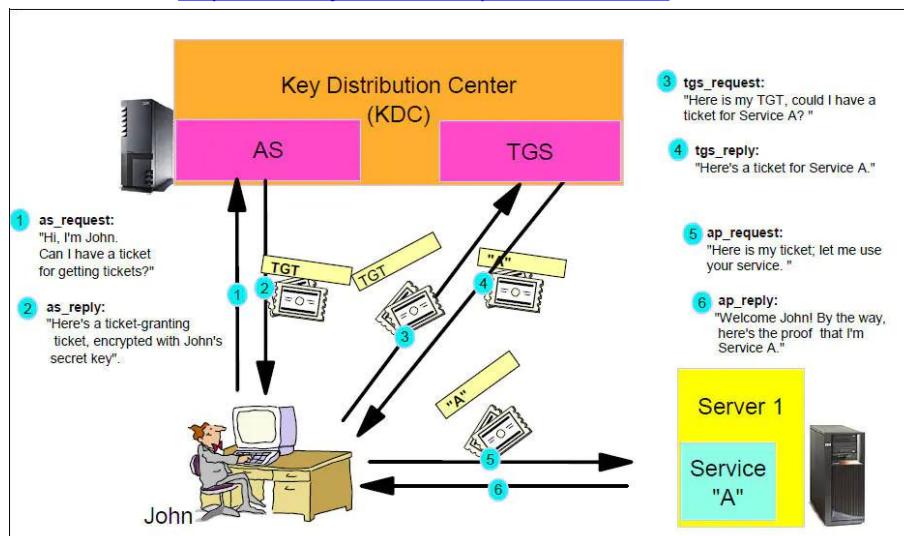
Common Single Sign-on

REin
REliable, INtelligent & Scalable Systems

- Kerberos based
 - MIT Kerberos protocol
- Smart card based
 - Initial sign-on prompts the user for the smart card.
 - Additional software applications also use the smart card, without prompting the user to re-enter credentials.
 - Smart card-based single sign-on can either use certificates or passwords stored on the smart card.
- OTP token
 - Also referred to as one-time password token.
- Security Assertion Markup Language
 - Security Assertion Markup Language (SAML) is an XML-based solution for exchanging user security information between an enterprise and a service provider.

2021/10/21

kerberos 入坑指南 <https://www.jianshu.com/p/fc2d2dbd510b>



kerberos 的 as tgs cs 认证基本原理

https://blog.csdn.net/qq_43536701/article/details/109615325

KDC (Kerberos Distribution Center) 密钥分发中心，维护所有账户的名称和 Master Key (key 的 hashcode)。提供：AS 认证服务、TGS 票据授予服务。

摘录自 <https://www.cnblogs.com/zhangwuji/p/9154153.html>

Kerberos 为什么要采用 3 步交互的形式来完成安全认证，那就要从 Kerberos 的使用场景说起。

相比 Kerberos，https 可能更为熟悉一点，通过证书和非对称加密的方式，让客户端可以安全的访问服务端，但这仅仅是客户端安全，通过校验，客户端可以保证服务端是安全可靠的，而服务端却无法得知客户端是不是安全可靠的。这也是互联网的一种特性。而 Kerberos 可以支持双向认证，就是说，可以保证客户端访问的服务端是安全可靠的，服务端回复的客户端也是安全可靠的。

想证明 client 和 server 都是可靠的，必然要引入第三方公证平台，这里就是 AS 和 TGS 两个服务。

Client 向 Kerberos 服务请求，希望获取访问 server 的权限。Kerberos 得到了这个消息，首先得判断 client 是否是可信赖的，也就是白名单黑名单的说法。这就是 AS 服务完成的工作，通过在 AD 中存储黑名单和白名单来区分 client。成功后，返回 AS 返回 TGT 给 client。

client 得到了 TGT 后，继续向 Kerberos 请求，希望获取访问 server 的权限。Kerberos 又得到了这个消息，这时候通过 client 消息中的 TGT，判断出了 client 拥有了这个权限，给了 client 访问 server 的权限 ticket。

client 得到 ticket 后，终于可以成功访问 server。这个 ticket 只是针对这个 server，其他 server 需要像 TGS 申请。

通过这 3 步，一次请求就完成了。当然这里会有个问题，这样也没比 https 快啊。解释一下

1. 整个过程 TGT 的获取只需要一次，其中有超时的概念，时间范围内 TGT 都是有效的，也就是说一般情况访问 server 只需要直接拿到 ticket 即可

2. 整个过程采用的是对称加密，相对于非对称加密会有性能上的优势

3. Kerberos 的用户管理很方便，只需要更新 AD 中的名单即可

当然整个过程的通信都是加密的，这里设计到两层加密，因为所有的认证都是通过 client，也就是说 Kerberos 没有和 server 直接交互，这样的原因是 Kerberos 并不知道 server 的状态，也无法保证同时和 server, client 之间通信的顺序，由 client 转发可以让 client 保证流程顺序。

第一层加密，Kerberos 对发给 server 数据的加密，防止 client 得到这些信息篡改。

第二层加密，Kerberos 对发给 client 数据的加密，防止其他网络监听者得到这些信息。client 和 server 的通信也是如此。

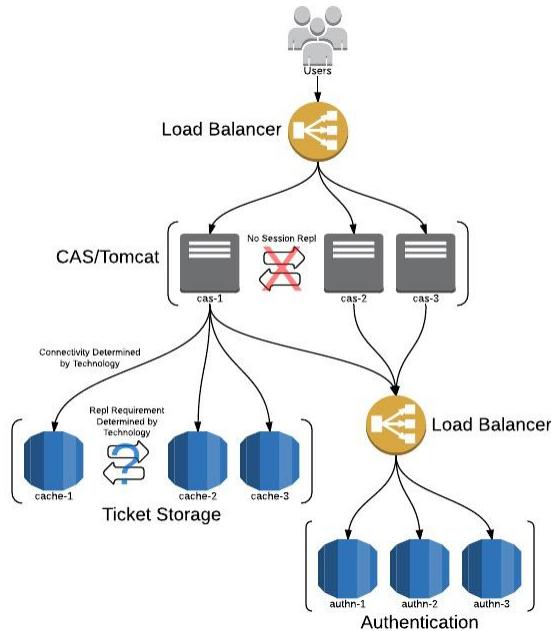
AS 就是看你的账号密码对不对，而 TGS 就是给你发一个和服务通信的票。因为服务解开了票所对应的信息，所以我们请求的服务也不是钓鱼网站。所以既认证了用户 (client)，也验证了服务 (service)。AS 和 TGS 是逻辑上独立的两个组件，但是通常部署在一个机器上。单点认证的意思就是只要 key 的 session 不过期，我们就再也不需要输入账号和密码。这就是 Kerberos 这个协议的通讯方式。

Kerberos 的问题就是如果 Key distribution center crush 了，那么整个系统全部都崩溃了。

Drawbacks and Limitations

- Single point of failure.
- Kerberos has strict time requirements, which means the clocks of the involved hosts must be synchronized within configured limits.
- The administration protocol is not standardized and differs between server implementations.
- Since all authentication is controlled by a centralized KDC, compromise of this authentication infrastructure will allow an attacker to impersonate any user.
- Each network service which requires a different host name will need its own set of Kerberos keys. This complicates virtual hosting and clusters.

CAS 就是基于 Kerberos 的实现，它为了解决单点故障的问题，就去做了集群，前面有多个 load balancer。



它里面核心的就是派票机制。

安全性总的分为：

1. 不可否认性：比如同学在开电子书店，我们一下子下了一万块的单，然后他去进货了。也就是你在网站上的所有动作不能否认。比如我们所有的数据都通过你的私钥加密，那么如果能用公钥解密，那么就一定是你做的。
2. 机密性：用对方的公钥加密只能用对方的私钥解开。
3. 完整性：数据在传输的时候需要是完整的，不能丢东西。那就是生成一个摘要（digest），如果数据的摘要不相等，那么我们很快就知道内容不一样了。
4. 我们需要知道服务器不是钓鱼网站。
5. 能审计，前提是东西可以审计，所以 log 就很重要。记录工具、可视化工具等。

抵御攻击里包括不可否认性、完整性、机密性、确认对方。所以数据存储时要加密。通信链路上也需要保证安，SSL 和 VPN。保持完整性有校验和和哈希。代码要有限的暴露，尽量代码中的所有都是 private。在 java 中就是接口和实现要分离。

概述：怎么用好 MySQL

我们讲讲怎么用好 MySQL，这个内容来自于 mysql 手册，虽然比较杂但是值得讲。我们目标是看完以后在做数据库设计的时候，有多少个表、表和表之间什么关系、怎么建立索引等做一些优化。

建立数据库的优化分成两个级别

1. 数据库本身的结构优化
2. 硬件层面，是不是要把关于 mysql 的缓存开大一点、允许同时打开表的数量弄大一点。

我们就说 book 是一张表合适，还是多张表合适。这要看我们 book 的结构。

title	author	price	introduction
-------	--------	-------	--------------

这张表在硬盘中是默认是一行行存的。我们读的时候，需要从硬盘中加载到内存中。是没有人经常靠 introduction 来做搜索的。如果没有 introduction，我们一次可以向内存中加载很多行，而有了 introduction 就会少很多。

比如我们的 introduction 是 varchar(20000)，还是 TEXT/ BLOB 存呢？显然后面两种好，因为后面两种类型存的是指针，指向了这个 introduction 的文件的存储的位置。我们把 introduction 存成指针的方式 load 进来效率比较高。

再比如我们给 author 类型定义好了，但是有些书没有 author，这个字段弄成可以为空好不好，还是不能为空呢？一定程度上 NULL 为额外占用一个位，会浪费空间，而且对可以为空的列去建立索引的情况下，索引的效果可能不好。

我们建立的索引能不能把我们的查询变得有效？

比如我们在 title、author、price 上都建立了一个索引（建立索引就是复制出来建立一棵树）。我们是建立四个索引合适呢，还是建立包含四个的复合索引合适呢？建立复合索引肯定省空间，比如我们建立了 title:author:price:introduction，但是在我们 select 的时候，如果直接查询 author = ...，或者 title = ... or author = ...。那么我们的索引就失效了，这就要求我们需要根据业务环境 query 的形式就去建立什么样的索引。

为什么复合索引更好一点呢？我们在数据库中 CRUD 的时候，索引也需要同步更新。更新一个索引（树）的效率肯定大于更新四个索引（树）的效率。

MySQL 中有不同的数据存储引擎，每种有不同的特点。比如内存临时表、innodb 等。不同的引擎需要考虑不同引擎的优势。

再往下，考虑到搜索、内存中 load 的效率，我们想把表拆开。但是 author 可能中国人两个字、外国人三十个字，难道我们要定义成 char(30)吗，所以这个行也可以压缩。行存储的时候，可以使用压缩模式来节省空间，但是有压缩模式，就需要解压，所以也存在额外的开销。

使用的锁合适吗？缓存的尺寸应该开多大呢？如果缓存满了会发生什么？

在硬件层面，如果数据库的应用最终在硬件上，硬件有没有可能对应用进行调优来降低开销？

我们需要考虑磁盘的寻找问题，数据在磁盘上怎么存，数据按行存看起来比较直观。双十一过去了，我们希望知道平台一共卖了多少钱。我们读购买数量和 price，连续读就很快。

这是在线分析处理（OLAP），这是我们在分析，而不是在写，这样按列存就比较合适。但是按列存对双十一当天很不友好，插入订单应该按行存更好。

数据库为了支持 OLAP 和 OLTP，

方法 1：按照两种方式存储，缺点是空间翻倍、同步还有问题

方法 2：行和列之间做转换，数据如果经常被统计，那么就按照列存。但是实现难度大。

这就是 HTAP（hybrid）

在 ROCKSDB 上，就在做这种混合负载的存储。所以我们需要知道磁盘寻道为什么比较复杂。磁盘的寻找发生在三维世界中。

SSD 就是 U 盘做大，是一维空间，已经很快了。但是如果我们将行存 OLAP，还是不是连续存的，读起来肯定很慢。

磁盘本身的读写也是问题，毕竟磁盘是外设，和内存的读写机制不一样，外设的总的速度慢很多。我们要尽量考虑内存中缓存更多的数据。这是我们在硬件上可以想的办法。

那么，我们在数据库设计的时候，需要平衡可移植性和性能，尤其是在使用某个数据库提供的具体功能。比如 MySQL 对 SQL 做了自定义的提高性能的扩展，在迁移的时候，我们就需要把这些 Oracle 中没有的扩展全部注释掉。

数据库中索引的作用

我们谈谈索引的作用，一般来说在作查找的时候，为了提高速度就去建立索引。在查找的时候，我们是怎么找的呢？如果没有索引会怎么样？比如我们搜索某列，我们就做一次全表的查找。如果内存还存不下所有的数据，那么还需要换进换出。数据结构的时候怎么办呢？我们把这一列的数据有不同的取值，我们建立一棵树。一开始我们学的都是二叉树，如果索引建立退化成链，那么就没什么用，所以至少我们需要二叉平衡树。我们搜索就 取决于树的高度，搜索次数为 $\log_2(N)$ ，看起来已经不错了，但是在数据库上百万条的时候，还是不能接受，所以我们需要 N 叉树，那就是我们的 B 树。B 树的查找效率不太稳定。所以就有了 B+ 树，所有的关键词都在叶子结点里头，叶子结点可以放多个 key，最后一个 key 有指向下一个 key 的指针。这个指针就是在我们做范围查询的时候，就非常方便了。我们就可以通过链表一次性访问。我们仔细去看索引是啥，其实就是把这一列的东西都搬了出来建了一棵树，所以空间大小其实就是一个 duplication。我们在支持的时候，是把索引 load 到内存里的，索引命中的时候，就对应的硬盘上的位置，我们就可以直接寻址过去。索引不能太大，因为内存有大小，不能让索引是换进换出的。我们最好是让索引在 4K 中可以放 4000 个，所以索引上的列绝对不能是特别长的取值。在几列上做的原因是，光看到书，直接拿书去比较是没什么意义的，因为同一个书名可能对应多种书，这时候就需要联合索引。整个目的就是加速搜索来解决全表扫描的问题。现在都是数值型、字符串型的索引，还有一种是离交大一公里内的饭馆，这就是基于地理位置的索引。我们可以记录经度和纬度，我们现在看到的地图都有高斯克隆投影、还有莫卡拓投影，还需要修正地理位置、高度等。在这种信息基础上建立索引，才能去找它。它有专门的索引的结构——R tree

2021/10/25

多个索引和复合索引

一旦我们改写数据库中的数据，那么需要对这棵树里的节点的位置做重新调整。如果我们对 ABC 列建立了三个索引，显然在三个索引树上调整效率比较低，但是在搜索的时候，三个列索引显然比对三列建立一个复合索引效率高。

搜索指的就是 where 子句中条件是什么。索引文件是要占空间的，事实上搜索的时候是把索引文件加载到内存中，在内存中构建出一棵树，然后再去做搜索，索引文件可能很大、不能一下子全部 load 进来，那么就要换出去或者加载新的一页进来。所以会有额外空间和时间的开销。所以我们不能对一个数据库无限次的建立索引，没意义的索引建立了纯粹浪费时间和空间。索引的代价就是写操作的时候，每一个索引都会更新。所以，我们要权衡一下。

- Indexes are used to find rows with specific column values quickly.
 - Without an index, MySQL must begin with the first row and then read through the [entire table](#) to find the relevant rows.
 - The larger the table, the more this costs.
 - If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data.
 - This is much faster than reading every row sequentially.
- Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in [B-trees](#). Exceptions:
 - Indexes on spatial data types use [R-trees](#);
 - MEMORY tables also support [hash indexes](#);
 - InnoDB uses [inverted lists](#) for FULLTEXT indexes.

这个表越大，建立索引以后效率的越高。每个表里要建立哪些索引呢？我们通过会根据检查查表的这一列去做索引。我们需要在主键上建立索引吗？id 是数据库自增的一个组件。真正的数据比如说是 id（数据库内部主键）：姓名：年龄：身份证号。

因为我们可能有另一张表，有一个外键在引用这个用户的 id。除了下订单，比如我们还有付账也在引用这个 id。这时候我们就可以看到在主键这里建立索引的好处了。比如我们在做别的表的查询的时候，外键有 id=290，我们就可以在这个主键索引中快速查询这个表中的数据。

关系型数据库必然有很多外键的关联，在我们做 join 操作的时候，我们必然会根据外键去快速地定位到数据在哪里。这就比我们按顺序扫描全表要快得多。

绝大多数的索引都是 B 树和 B+树，空间数据（包括维度、经度、海拔、投影方式等）用的是 R 树。搜索方式可能就是输入一个当前位置，搜索附近一公里中的美食。

还有一些是内存中的临时存储，就可以使用高效的哈希索引。但是哈希的时候就比较浪费空间，当我们全量数据非常大的时候，就不适合用哈希表了。

InnoDB 使用了 FULLTEXT 建立反向索引。

值不值得在某列上建立索引？

然后它在做这个操作的时候，为了加快我们读取的速度，我们在做索引的时候，要考虑值不值得在这个列上建立索引。

比如对于 `age` 列，假如我们有一千万用户，`age` 顶多从 3~120，我们如果在这一列上建立索引，那么一千万人大部分都是重合的，那么我们找到的结果会有一大堆，那么对 `age` 建立索引是没什么效果的。包括如果 `age` 是“保密”也就是为空，这种对索引的排序也是有害的。

刚才说的主键，如果我们做 `join` 操作的时候，我们在主键上建立索引就很有意义。索引的内容太长了怎么办？比如 `introduction` 列有 1000 个字符，这个索引就太大了、并且会导致搜索的时候效率也很低，必须不断地交换和 `load`，这时候我们就可以用前缀（比如前十个字符）来建立索引。

如果要用带前缀的方式来建立索引，那么效率就高。其实对于很小的表建立索引效率是不高的，因为 `logn` 和 `n` 曲线只有 `n` 比较大的时候，它们的差才比较大，如果 `n` 比较小，为了避免索引在修改后需要更新的操作，其实不建立索引也没问题。

eg: 比如今天过年了，所有人的年龄加一岁，这种操作的时候我们就不需要建立索引，因为我们这种操作就是要扫描全表的。

索引的目的就是在很大的一张表中做少量数据的查询。然后主键索引，我们无论怎么说都要在主键上建立索引。但是主键不一定是一个列，它有可能是两个列。

学生选课的时候，如果不做基于主键递增的话，那么学生 `id+课程 id` 这就可以作为主键。多列无非就是把多列数据拼起来做一个搜索。

Q: 主键为什么适合建立索引？

A: 因为它规定了不能为空、并且彼此不同，这棵树建立起来就很完美，没有两个指针指向两行。

在建立这个索引之后，在每一次搜索时，我们默认没有加任何索引，它会自动给主键加一个索引。并且它往硬盘上存储的时候，它也会按照主键排列的数据去存储。主键既要能在表中唯一标识这一行，并且在被选键中包含的列最小。我们会有基于整数递增的这一列是因为我们也不知道后面这几列哪些是唯一的。

Q: 数据库在做主键的时候到底要不要这个自动生成的键，默认是自动生成，其实我们也可以不要。我们怎么去选择呢？

A: 主键如果由太多的列构成，那么这个建立索引的效率就不太好，并且自动生成主键可以直接保证主键唯一，如果我们自定义几列是主键，我们需要手动地去检查是否重合。

UUID 和 GUID 作为主键

Q: 自动生成的主键是否应该使用递增的整数呢？

A: 还有 UUID, GUID 可以用。

Q: UUID(通用唯一识别码, Universally Unique Identifier)和 GUID(Microsoft's Globally Unique Identifiers)的作用是什么?

A: 它的用处是大量请求过来往数据库中插入的时候, 它们可以保证数据的 ID 不会重合。

案例: 双十一的订单处理

双十一的 `order` 可能用了三个服务器做一个集群, 当有用户请求过来的时候, 就负载均衡到一个服务器中插入记录, 如果三个机器中都有一个 `mysql` 并且按照整数递增, 那么可能用户 A 和用户 B 在服务器 A 和服务器 B 上插入了两个不同的表, 主键就重复了。如果我们使用全局唯一标识符, 那么我们就使用机器+时间戳+数据库的 `id+当前的对象的 id+随机数` 等, 它就会把这一系列的数据合起来凑成一个 32 位的 16 进制数。这里面重复的概率有多大, 在同一个机器的同一个进程的同一秒内通过同一个对象产生了两个相同的随机数才会重合, 这时候我们就发现这三台机器 `ip`、`对象`、`数据库 id` 都不一样, 这些都不一样的东西产生相同 `UUID` 的概率微乎其微。

但是用它有非常差的结果, 因为每个 `UUID` 有 16 个字节, 所以在其之上建立索引比较耗空间。并且基于整数递增的主键我们是知道谁先插入、谁后插入的, 因为它是按照时间递增的, 而 `uuid` 是不能判断时间先后的。

- MySQL permits creation of SPATIAL indexes on NOT NULL geometry-valued columns.
 - For comparisons to work properly, each column in a SPATIAL index must be SRID-restricted.
 - That is, the column definition must include an explicit SRID attribute, and all column values must have the same SRID.
- The optimizer considers SPATIAL indexes only for SRID-restricted columns:
 - Indexes on columns restricted to a Cartesian SRID enable Cartesian bounding box computations.
 - Indexes on columns restricted to a geographic SRID enable geographic bounding box computations.

要表达地理空间的概念, 可以基于一些数据来建立索引, 这种所以只在 R 树上来建。

外键的关联。如果我们数据库中的表有一些字段非常的长, 并且不一定经常被访问, 比如个人的介绍、个人的照片。如果我们每次加载这个人都加载 `introduction` 和 `img` 一起加载进来, 那么总的来说我们加载的东西会占掉我们的内存导致每次加载进内存的数据会非常少, 导致我们性能下降。这时候我们就可以拆成两个表, 后面我们可以看到甚至 `introduction` 和 `img` 存在 MongoDB 中, 我们只要维护住这种关联关系, 我们就可以进一步提高我们的性能。

我们每一张比较小的表, 都有主键快速查找数据, 我们使用 `join` 操作的时候, 都会使用主键的索引, 所以我们主键要尽可能使它简单。

因为我们在搜索数据的时候, 最终提高磁盘的 io 加载到内存中来处理, 分开来的好处就是我们不需要每次读取大量的冗余数据到磁盘中, 我们需要把经常访问的信息和不经常访问的数据拆成两个表, 这样我们就可以保证读取相同大小的数据的时候可以读到更多的记录数。

*`introduction` 的前缀索引, 我们就可以使用它的前十个字节来建立索引, 这个 `blob` 实际上存的是一个指针, 指向硬盘中的地址。当然前十位不能保证唯一, 可能就有一些行有相同的值, 这时候可能就是一个链表。数据库一行行存没错, 但是它会对数据做压缩, 就会涉及到行是以哪一种方式在做存储, 我们数据库的表到底是以哪一种方式在做存储可以在配置文件中修改 `REDUNDANT` 和 `COMPACT`。

Index Prefixes

- With `col_name(N)` syntax in an index specification for a string column, you can create an index that uses only the first *N* characters of the column.
 - Indexing only a prefix of column values in this way can make the index file much **smaller**. When you index a **BLOB** or **TEXT** column, you **must** specify a prefix length for the index. For example:
`CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));`
- Prefixes can be up to **767 bytes** long for InnoDB tables that use the **REDUNDANT** or **COMPACT** row format.
- The prefix length limit is **3072 bytes** for InnoDB tables that use the **DYNAMIC** or **COMPRESSED** row format.
- For MyISAM tables, the prefix length limit is **1000 bytes**.
- If a search term **exceeds the index prefix length**, the index is used to exclude non-matching rows, and the remaining rows are examined for possible matches.

如果搜索的长度超过了索引的前缀指令长度，这个索引会把前十个没有匹配的扔掉、把前十个匹配上的行数据拿出来，再去做剩余部分的匹配，这就相当于剪枝。

FULLTEXT Indexes

- FULLTEXT indexes are used for **full-text searches**.
 - Only the **InnoDB** and **MyISAM** storage engines support **FULLTEXT** indexes and only for **CHAR**, **VARCHAR**, and **TEXT** columns.
 - Indexing always takes place over the **entire column** and column prefix indexing is not supported.
- Optimizations are applied to certain kinds of FULLTEXT queries against single InnoDB tables. Queries with these characteristics are particularly efficient:
 - FULLTEXT queries that **only return the document ID, or the document ID and the search rank**.
 - FULLTEXT queries that **sort the matching rows in descending order of score and apply a LIMIT clause to take the top N matching rows**. For this optimization to apply, there must be no WHERE clauses and only a single ORDER BY clause in descending order.
 - FULLTEXT queries that **retrieve only the COUNT(*) value of rows matching a search term**, with no additional WHERE clauses. Code the WHERE clause as WHERE MATCH(*text*) AGAINST ('*other_text*'), without any > 0 comparison operator.

全文索引，之前已经讲过了，对于 **CHAR**, **VARCHAR**, **TEXT** 才能建立全文索引，原理就是把单词全部断开然后建立反向索引，将来搜索的时候就会说包含某个单词的记录有哪些，然后全部取出来。

- **Spatial Indexes**

- You can create indexes on spatial data types.
- MyISAM and InnoDB support **R-tree** indexes on spatial types.
- Other storage engines use **B-trees** for indexing spatial types (except for ARCHIVE, which does not support spatial type indexing).

- **Indexes in the MEMORY Storage Engine**

- The **MEMORY** storage engine uses HASH indexes by default, but also supports **BTREE** indexes.

空间数据就是 R 树，内存表可以使用哈希，也可以使用 B 树。

MySQL 中的复合索引(Composite Index)

我们拿多列建立一个索引的情况下，MySQL 中最多拿 16 个列来建立 **Composite index**，其中也可以包含取前缀的列。在组合索引上建立查询的时候，会按照建立索引的顺序来搜索，如果我们跳过第一列搜索后面几列，这个索引就废了。

Suppose that a table has the following specification:

```
CREATE TABLE test (
    id INT NOT NULL,
    last_name CHAR(30) NOT NULL,
    first_name CHAR(30) NOT NULL,
    PRIMARY KEY (id),
    INDEX name (last_name,first_name) );
```

Therefore, the **name** index is used for lookups in the following queries:

```
SELECT * FROM test
    WHERE last_name='Jones';
SELECT * FROM test
    WHERE last_name='Jones' AND first_name='John';
SELECT * FROM test
    WHERE last_name='Jones' AND (first_name='John' OR first_name='Jon');
SELECT * FROM test
    WHERE last_name='Jones' AND first_name >='M' AND first_name < 'N';
```

However, the **name** index is **not** used for lookups in the following queries:

```
SELECT * FROM test
    WHERE first_name='John';
SELECT * FROM test
    WHERE last_name='Jones' OR first_name='John';
```

复合索引的前提就是 **where** 语句中符合 **composite index** 中的最左前缀。

MySQL cannot use the index to perform lookups if the columns do **not form a leftmost prefix** of the index.

Suppose that you have the **SELECT** statements shown here:

```
SELECT * FROM tbl_name WHERE col1=val1; ✓
SELECT * FROM tbl_name WHERE col1=val1 AND col2=val2; ✓
```

```
SELECT * FROM tbl_name WHERE col2=val2; ✗
SELECT * FROM tbl_name WHERE col2=val2 AND col3=val3; ✗
```

If an index exists on (col1, col2, col3),

- only the first two queries use the index.
- The third and fourth queries do involve indexed columns, but do not use an index to perform lookups because (col2) and (col2, col3) are **not** leftmost prefixes of (col1, col2, col3).

怎么把数据翻译成 MySQL，怎么从 MySQL 中组装成对象，它的性能肯定比 jdbc 慢。并且我们对对象的操作会翻译成 MySQL 中的语句，我们怎么保证 JPA 翻译出来的 SQL 能用到这个索引中呢？所以，在这种情况下，这就是 JPA 中可能存在的问题：使用数据库的效率没有那么高，因为这个和语义相关，不是 JPA 决定的。

通过这个例子，我们更需要注意索引中列的排序。

- **B-Tree Index Characteristics**

- A B-tree index can be used for column comparisons in expressions that use the `=`, `>`, `>=`, `<`, `<=`, or `BETWEEN` operators.
 - The index also can be used for `LIKE` comparisons if the argument to `LIKE` is a constant string that does not start with a wildcard character.
-
- For example, the following **SELECT** statements use indexes:
`SELECT * FROM tbl_name WHERE key_col LIKE 'Patrick%';`
`SELECT * FROM tbl_name WHERE key_col LIKE 'Pat%_ck%';`
 - The following **SELECT** statements do **not** use indexes:
`SELECT * FROM tbl_name WHERE key_col LIKE '%Patrick%';`
`SELECT * FROM tbl_name WHERE key_col LIKE other_col;`

然后我们在 B 树索引里，支持做范围类型的操作，并且支持 `LIKE` 类型的操作，因为它在叶子结点有一个指针指向兄弟节点，这样范围查询就比较快。而 `Like` 的通配符不能是第一个，这样它就可以拿前缀去 B 树中找到一些符合要求的。如果以通配符开头，那就不能用到这个索引了。

在做索引的时候，下图的上方是一些 `where` 字句可以用到索引，而下图下方这些是没有办法用的。

B-Tree Index Characteristics

- The following WHERE clauses use indexes:

```
... WHERE index_part1=1 AND index_part2=2 AND other_column=3
/* index = 1 OR index = 2 */
... WHERE index=1 OR A=10 AND index=2
/* optimized like "index_part1='hello'" */
... WHERE index_part1='hello' AND index_part3=5
/* Can use index on index1 but not on index2 or index3 */
... WHERE index1=1 AND index2=2 OR index1=3 AND index3=3;
```

- These WHERE clauses do **not** use indexes:

```
/* index_part1 is not used */
... WHERE index_part2=1 AND index_part3=2
/* Index is not used in both parts of the WHERE clause */
... WHERE index=1 OR A=10
/* No index spans all rows */
... WHERE index_part1=1 OR index_part2=10
```

对于第二个优先级是 `AND` 高。第三行是可以对第一部分做索引然后剪枝。第三行第一种情况下可以索引两列，而计算后一半的时候只能索引一列。

然后刚才说的 B 树是在硬盘上存储的数据，而内存中存储可能用到散列的方式。它是不太支持做所谓的范围查找的，比如小于、大于。但是它单个值的搜索非常快，是 $O(1)$ 的复杂度，正是因为这些元素不支持范围，所以 `order by` 也是不行的。这就是哈希所带来的一些特性。它也不能知道在两个值之间有多少行，必须扫描全表。

MySQL 倒序索引

我们看到的一般来说建立 B 树索引的时候，都是从左到右升序的，当然也支持做倒序。倒序的意义就是做反向搜索的时候速度会比较快，然后它在做降序的时候，就可以在索引中指定是升序（`ASC`）还是降序（`DESC`）。

Consider the following table definition

```
CREATE TABLE t (
    c1 INT, c2 INT,
    INDEX idx1 (c1 ASC, c2 ASC),
    INDEX idx2 (c1 ASC, c2 DESC),
    INDEX idx3 (c1 DESC, c2 ASC),
    INDEX idx4 (c1 DESC, c2 DESC)
);
```

```
ORDER BY c1 ASC, c2 ASC -- optimizer can use idx1
ORDER BY c1 DESC, c2 DESC -- optimizer can use idx4
ORDER BY c1 ASC, c2 DESC -- optimizer can use idx2
ORDER BY c1 DESC, c2 ASC -- optimizer can use idx3
```

我们为什么要建立这 4 个索引呢？如果我们只建立第一个，那么后面三个操作我们查出

来以后还要额外排一个序。

数据库的设计

然后我们回到这两个问题：数据库结构应该怎么设计？表应该怎么设计？

数据库设计的 Motivation：提高性能（最小化 IO，关系型数据库还是要把有关的东西放在一起，但是从性能角度考虑，虽然一个人的简介和照片也是在描述同一个对象但是我们还是要拆成两个表）、可扩展性（数据量不断增多的时候，我们的表是不是还能适应）。

我们先从数据库的设计开始，

1. 数据的尺寸，我们在硬盘上存储的时候，这个空间能不能尽量少一些，那么我们就要对数据压缩。压缩的时候和数据是按行存还是按列存也有很大的关系。

比如 age 列中有：10,10,10,11,10,11,12,10，我们可以存 10:3,11,10,11,12,10。再比如说大家的学号都是 519021911XXX，我们可以存一个统一的前缀，然后每一行只存后面不同的几位。这就涉及到我们数据库的设计，比如是 varchar 还是 text（存成指针，一次性可以 load 进来更多的记录）。

这就是我们表的列是什么类型，应该怎么去设置。压缩方法？索引？表拆开做 join 还是在一张表中？是否符合范式（数据的冗余度越小可能导致性能比较差，代表互相之间的外键关联特别多，导致经常需要 join 操作，有数据冗余的话）？

比如气象站有一张表，记录经度：维度：气象站名字，有另外一张表记录气象站 id：年月日：检测指标，比如 1, 2021/10/25, 15:30, 10mm。比如我们有 30 个气象站存了一年有一亿条数据，如果我们想写北京气象站下午 15 点检测到 10mm 的降雨量，那么我们要做 join 操作。

我们也可以存到一张表中，

id	经度	维度	气象站	时间	降雨量
1			北京	15:00	10mm
2			北京	16:00	20mm

我们会发现第二种设计范式化程度低，有数据冗余，但它搜索起来不需要 join 操作，搜索效率高。

再说表的列，在设计的时候应该做到适合它的最小类型，我们可以使用 MEDIUMINT，如果够放就直接用。尽量不要让列是 NULL，因为索引不好建，并且还需要一个额外的 bit 表示是否为 NULL。

行也有简洁型、压缩型、动态型等。比如存储的时候，时间戳可以存偏移量，数据也可以只存偏移量。

2021/10/28

Text 和 Blob 类型的优化

碰到数值型的东西，直接用字符串存，这样就不需要读进来以后做类型转换。blob 和

`text` 存的是一个指向文件的指针，保证了一行不会超过数据库的要求的大小。

数据库中行尺寸的大小必须小于内存页中一半的大小。当我们存的东西小于 `8k` 的时候，我们可以使用 `varchar`，但如果大于 `8k` 的情况下，最好变成一个 `blob` 和 `text` 类型。

当我们发现字段太多的情况下，我们可以切分成常用的表和不常用的表，查询的时候做 `join`，保证表比较小，这样一次加载到内存的条目会比较多。主键可以用自增或者比较长的 `UUID`，但是 `UUID` 不能推断出数据插入的先后，我们可以在 `UUID` 的基础上再加一个升序的前缀。

`Blob` 是二进制对象，文本数据当 `blob` 去存的时候，可以在存之前压缩一下。如果我们想把 `blob` 中加到内存中，有关 `blob` 的表最好扔掉别的机器上去存。很长的文本信息比较的时候，因为一个个比较速度很慢，我们可以使用文件的哈希值作为额外的字段，直接比较这个哈希值（文件摘要）即可。

优化 MySQL 的表

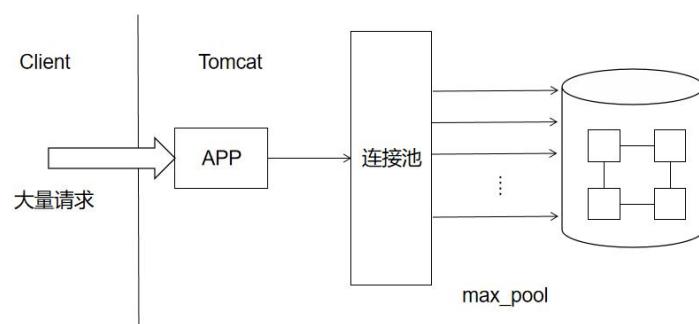
Q: 在 MySQL 中能打开多少个表建立多少个库？比如我们开一亿个表行不行？此时会出什么问题？

很显然，表多了，每一个表都需要有元数据。成千上万的表光是元数据就很大。MySQL 从这个角度告诉我们表不能无限增加下去。

- When you execute a `mysqladmin status` command, you should see something like this:
`Uptime: 426 Running threads: 1 Questions: 11082
Reloads: 1 Open tables: 12`
- The Open tables value of 12 can be somewhat puzzling if you have `fewer` than 12 tables.
- MySQL is `multithreaded`, so there may be many clients issuing queries for a given table simultaneously.
 - To minimize the problem with multiple client sessions having different states on the same table, `the table is opened independently by each concurrent session`.
 - This uses `additional memory` but normally `increases performance`.
 - With `MyISAM` tables, one extra file descriptor is required for the data file for each client that has the table open.

MySQL 的命令行工具中，如果执行 `mysqladmin status` 就可以看当前有多少线程、打开的表有多少。这个表的数量是后台的，不一定代表是我们自己打开的。因为在 MySQL 中做了一些处理。

比如我们的电子书店是部署在 Tomcat 里的，而 Tomcat 访问 MySQL 的时候会使用连接池中的连接，连接的数量是 Tomcat 的配置文件中设置的。每当有一个应用连接请求过来，Tomcat 会尝试在连接池中找到一个连接返回，如果没有连接就新建一个到 MySQL 的连接，直到连接数量到达了配置文件中的 `Max_pool` 数量，然后对连接去做分时复用。



比如我们设置 `Max_pool = 4`, 那么就有 4 个连接在同时访问 MySQL, MySQL 认为有 4 个并发会话。比如大家都在下订单, 那么都会去访问 `order` 这张表。

MySQL 为了保证大家看起来像在独占这张表, 会为每个并发线程单独打开一张表。所以 MySQL 在做并发统计的时候, 它会认为 `order` 打开了 4 个表。所以我们从 `mysqladmin status` 中看到的 `Open tables` 就会比我们认为的数量大一些。

打开表以后就是增删改查, 任何一个操作都是要读到内存中进行操作。所以, 在这种模式下, 一张表会占用很多倍的空间。所以我们要尽量把内存空间开大, 虽然浪费了空间, 但是大家都是互相隔离地打开表操作, 性能就会提高。

table_open_cache

在 MySQL 的配置文件中有两个参数。`table_open_cache` 就是允许线程打开表的数量。而 `max_connections` 就是允许连接的数量。

How MySQL Opens and Closes Tables

- The [table open cache](#) and [max connections](#) system variables affect the **maximum number** of files the server keeps open.
 - If you increase one or both of these values, you may run up against a limit imposed by your operating system on the per-process number of open file descriptors.
- [table open cache](#) is related to [max connections](#).
 - For example, for 200 concurrent running connections, specify a table cache size of **at least $200 * N$** , where N is the maximum number of tables per join in any of the queries which you execute. You must also reserve some extra file descriptors for temporary tables and files.
- Make sure that your operating system can handle the number of open file descriptors implied by the [table open cache](#) setting.
 - If [table open cache](#) is set **too high**, MySQL may run out of file descriptors and exhibit symptoms such as refusing connections or failing to perform queries.

那么 `table_open_cache` 怎么计算呢? 比如说我们 MySQL 允许 200 个并发线程连接, 而这 200 个线程请求中, 里面对 `join` 表操作的最大数量是 4 张表, 那么 `table_open_cache` 就是 $200 * 4 = 800$ 。也就是最坏情况下, 这 200 个操作都会使用 4 张表的 `join` 操作, 要让这些线程都能够正常访问。

如果 `table_open_cache` 太大了, 会消耗内存, 其次如果有一些连接失败了, 需要回滚的时候, 内部管理就会很麻烦。所以这个值也不能开的太高, 会提高内部管理成本。

Q: 什么时候关掉一张表呢 (移除 `table_open_cache` 中的表) ?

A: `table_open_cache` 如果不满, 通常没什么问题。但 `cache` 满了, 并且请求访问一张不在 `cache` 中的表, 这就会发生一次关闭, 其实就类似于内存的换进换出。当然 MySQL 也会周期性地检查是否有一些连接已经不再使用了, 把打开的不再使用的表关掉。

如果 `cache` 到达了上限, 它会使用最近最小使用尝试把表换出去, 如果 `LRU` 访问的表短到一定时间就说明大家都是常用表, 就临时扩大内存把我们新的表加载进去。但是这个临时打开的空间不允许持续, 使用完就必须立马关掉它。

刚才说的都是 `Innodb` 的实现。如果我们使用 `MyISAM` 引擎的话, 一个库里的文件存在一个目录里, 如果这个目录里的文件太多了就会导致文件的读取、查询都比较慢。如果我们在许多不同的表上去执行 `select` 操作。如果 `table_open_cache` 满了, 就会涉及到换进换出牵涉到额外的开销。所以我们物理内存上限决定了 `table_open_cache` 能开多大, 回推回来就是

如果文件数太多也会影响我们的性能。

Q: 那么我们有没有自己的办法控制里面有多少个表呢?

A: 我们自己的表当然是我们定义的,但是不要忘了 MySQL 在内存中还会生成有一些临时表,这依赖于 MySQL 引擎内部的实现机制,比如 UNION (把两张表合成一张表再写到硬盘上,过程中需要在内存中建立一个临时表)、VIEW 等。这都是 MySQL 引擎决定的,超出我们的管辖范围。

MySQL 中行和列的大小限制

MySQL 的文件本身没上限,但是会受操作系统的目录数量的限制。

- MySQL has **no limit** on the number of databases.
 - The **underlying file system** may have a limit on the number of directories.
- MySQL has **no limit** on the number of tables.
 - The **underlying file system** may have a limit on the number of files that represent tables.
 - Individual storage engines may impose engine-specific constraints. InnoDB permits up to **4 billion** tables.

我们可能弄了很多表,但是打开的时候报错告诉你当前打开的表已经达到上限了。

那么单个一张表的大小受操作系统文件大小的限制,所以肯定不能太大。当文件大到一定程度的时候,我们可以把它放到分布式文件系统上,或者从逻辑上把库切开(分区、或者从逻辑的设计上把它分开,比如不同首字母开头的用户存到不同的表里)。所以分区在逻辑上是一张表,而后者就是我们人为地切成了两张表。

If you encounter a **full-table error**, there are several reasons why it might have occurred:

- The disk might be **full**.
- You are using **InnoDB** tables and have run out of room in an InnoDB tablespace file.
 - The maximum tablespace size is also the maximum size for a table.
 - Generally, partitioning of tables into multiple tablespace files is recommended for tables **larger than 1TB in size**.
- You have hit an **operating system file size limit**.
 - For example, you are using **MyISAM** tables on an operating system that supports files **only up to 2GB in size** and you have hit this limit for the data file or index file.
- You are using a **MyISAM** table and the space required for the table exceeds what is permitted by the **internal pointer size**.
 - **MyISAM** permits **data and index files** to grow up to **256TB** by default, but this limit can be changed up to the maximum permissible size of **65,536TB** ($256^7 - 1$ bytes).

表是由行和列构成的,MySQL 列上限为 4096。就像我们上节课的复合索引不能超过 16 列。其实这个限制不光是列的上限,到底能放几列还需要看行的尺寸限制。

Row Size Limits

- The maximum row size for a given table is determined by several factors:
 - The internal representation of a MySQL table has a maximum row size limit of **65,535 bytes**, even if the storage engine is capable of supporting larger rows. **BLOB** and **TEXT** columns only contribute **9 to 12 bytes** toward the row size limit because **their contents are stored separately from the rest of the row**.
 - The maximum row size for an InnoDB table, which applies to data stored locally within a database page, is **slightly less than half a page** for 4KB, 8KB, 16KB, and 32KB **innodb page size** settings.
 - **Different storage formats** use different amounts of page header and trailer data, which affects the amount of storage available for rows.

为了减少行的大小，我们就需要把 Blob 和 Text 扔到外面去处理。

Row Size Limits Examples

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
   c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
   f VARCHAR(10000), g TEXT(6000)) ENGINE=MyISAM CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
   c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
   f VARCHAR(10000), g TEXT(6000)) ENGINE=InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)
```

用 TEXT 来存，这个字段只会占据 9~12 个字节，就存得下了。

Row Size Limits Examples

```
mysql> CREATE TABLE t1
   (c1 VARCHAR(32765) NOT NULL, c2 VARCHAR(32766) NOT NULL)
   ENGINE = InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t2
   (c1 VARCHAR(65535) NOT NULL)
   ENGINE = InnoDB CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not
counting BLOBs, is 65535. This includes storage overhead, check the manual. You have to change
some columns to TEXT or BLOBs

mysql> CREATE TABLE t2
   (c1 VARCHAR(65533) NOT NULL)
   ENGINE = InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.01 sec)
```

第二个报错的例子是因为要有额外的两个字节去记录里面存了多少。所以 varchar 这种类型需要 2 个字节来记录到底长为多少。

Row Size Limits Examples

```
mysql> CREATE TABLE t3
   (c1 VARCHAR(32765) NULL, c2 VARCHAR(32766) NULL)
   ENGINE = MyISAM CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting
BLOBs, is 65535. This includes storage overhead, check the manual. You have to change some columns
to TEXT or BLOBs
```

字段允许为空就要求字段再加一点空间来记录。

MySQL 的三种引擎在设计表、缓存、参数的时候应该怎么选择。

InnoDB 的优化方法

首先说 innodb，它一般来说当我们的表的尺寸达到比较稳定的，比如写操作集中在更新上，而表不会快速增长，就可以通过 SQL 中的 **optimize table** 去来重组表并且压缩表。

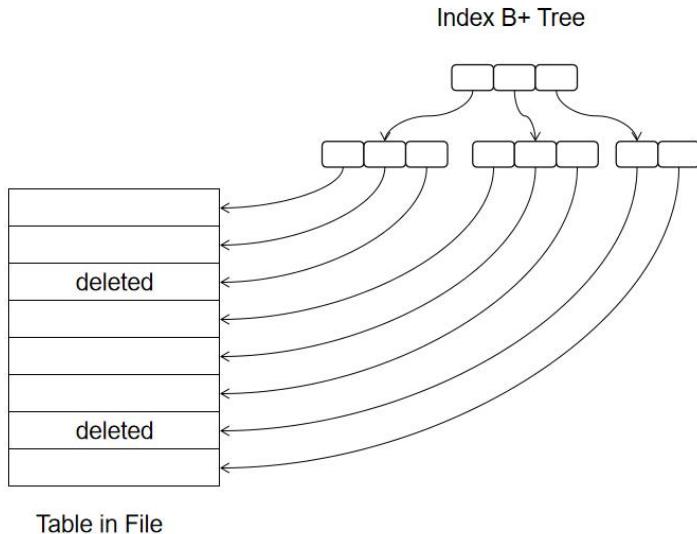


Table in File

我们原先的表，如果要删除一行的话，我们只能标志为 `deleted`，因为内存中的索引树还有一个个指针指过来，我们是不能改动位置的。

而 `optimize table` 就是帮我们在表的尺寸相对稳定了以后，把中间的已经删除的行都删掉，重新复制出一个紧凑表达的文件和索引树。

文件真正存的时候，存在磁盘不同的扇区上，通过指针连接。每隔一段时间它会整理一下，变成硬盘中连续存储的位置。主键一定要建立索引，所以主键不能太长。`Varchar` 可以压缩空间。

Autocommit

我们再来看一下事务，正常情况下客户端过来一个 SQL 语句，MySQL 做完以后会自动 `commit` 掉。但是这带来一个问题就是，`autocommit` 带来的结果就是从内存中要落到硬盘上。如果我们来一批数据过来都是 `update` 和 `insert`，性能就会很差，因为它会把内存中的一整个 page 都落到硬盘上。

我们可以把很多个涉及到写操作对应到一个事务里，一次性 `commit` 掉。

Optimizing InnoDB Transaction Management

- The default MySQL setting `AUTOCOMMIT=1` can impose performance limitations on a busy database server.
 - Where practical, wrap several related data change operations into a single transaction, by issuing `SET AUTOCOMMIT=0` or a `START TRANSACTION` statement, followed by a `COMMIT` statement after making all the changes.
- Alternatively, for transactions that consist only of a single `SELECT` statement, turning on `AUTOCOMMIT` helps InnoDB to recognize read-only transactions and optimize them.
- **Avoid** performing rollbacks after `inserting, updating, or deleting huge numbers of rows`.
 - If a big transaction is slowing down server performance, rolling it back can make the problem worse, potentially taking several times as long to perform as the original data change operations.
 - Killing the database process does not help, because the rollback starts again on server startup.

所以当性能明显受限的时候，我们可以禁用 `autocommit`，手动进行事务管理。还要注意我们的操作是在 `cache` 中操作，然后落到硬盘里。如果我们要做大量的写操作，`cache` 可能是不够的，它就会一部分一部分地往硬盘上落。一旦落到了硬盘上，如果我们要 `rollback` 的话就会比较麻烦了，不像内存中的 `rollback` 我们直接重新加载就行了。

真正在做事务管理的时候，数据都在 `cache` (`buffer pool`) 里，它会记录下我们所有对

数据中的写操作，要么写到硬盘上，要么从内存中删掉。我们要注意，一个事务里大量操作放进去，就会导致执行时间比较久，事务实际上是要加锁的，所以选择不同的隔离级别，会在隔离级别和性能之间做权衡。

我们应该拆分成若干个小的事务执行。如果我们在事务中发现大量的操作都是 `select`，MySQL 会对操作优化，开启只读事务，它就不会涉及到把内存中的数据落硬盘这个操作，它甚至都不需要给事务设置 `id`，因为只读事务将来也不涉及到回滚的事情。所以我们尽量让写的放在小事务里，对于读操作尽可能放到一个只读事务里。

如果我们 `cache` 中已经有一部分操作落到了硬盘上，那么 `rollback` 就要做 `redo` 操作，记录刚才已经落到硬盘上的 `log` 是哪些，它其实在硬盘上落下去了文件。所以我们要有一个 `log buffer`，`log buffer` 大了就代表我们可以记住够多的操作。

`redo` 这件事情涉及到 MySQL 内部的实现，所以我们只需要知道有这么一个 `log buffer` 即可。

在批量把数据库中往里面导入的时候，在 InnoDB 上，首先我们应该把 `autocommit` 关掉，否则我们 `load script` 的 `sql` 脚本里有大量的 `insert`，如果我们打开了就会降低 `load script` 的性能。把数据一个个插入的时候，键是否是 `unique` 的。如果每一次插入都要检查它的键是否是唯一的，这件事情似乎也比较麻烦。

Bulk Data Loading for InnoDB Tables

- When importing data into InnoDB, turn off `autocommit` mode, because it performs a `log flush to disk for every insert`.

```
SET autocommit=0;
... SQL import statements ...
COMMIT;
```
- If you have `UNIQUE` constraints on secondary keys, you can speed up table imports by `temporarily turning off the uniqueness checks` during the import session:

```
SET unique_checks=0;
... SQL import statements ...
SET unique_checks=1;
```

For big tables, this saves a lot of disk I/O because InnoDB can use its change buffer to write secondary index records in a batch. `Be certain that the data contains no duplicate keys.`

把 `autocommit` 关掉以后，所有 `unique` 的检查都在内存里检查，这样速度就会快一点。类似的还有外键，比如我们要判断引用的外键是否存在，我们也可以把这个检查给关掉。在插入的时候，我们尽量一个 `insert` 插入多个数据，在 `SQL` 中它被认为是一个操作。

Bulk Data Loading for InnoDB Tables

- If you have `FOREIGN KEY` constraints in your tables, you can speed up table imports by `turning off the foreign key checks` for the duration of the import session:

```
SET foreign_key_checks=0;
... SQL import statements ...
SET foreign_key_checks=1;
```

For big tables, this can save a lot of disk I/O.
- Use the `multiple-row INSERT` syntax to reduce communication overhead between the client and the server if you need to insert many rows:

```
INSERT INTO yourtable VALUES (1,2), (5,5), ...;
```

This tip is valid for inserts into any table, not just InnoDB tables.
- When doing bulk inserts into tables with auto-increment columns, set `innodb_autoinc_lock_mode` to 2 (interleaved) instead of 1 (consecutive).

对于 `query` 来说，主键索引不要指定太多的列，然后表上还需要根据访问频率建立很多辅助索引。我们提倡的是建立一个复合索引而不要在每一列上都建立索引。因为复合索引写

操作的时候，调整起来比较快。索引里如果包含 `null` 越多，那么查询的效率越差。因为 `null` 被认为是大量的相同的值（树中大量的值在同一个节点上）。并且 `null` 还是需要多占用一些空间的。

还有在定义数据库的时候，我们需要注意使用 `TRUNCATE TABLE` 一下子清空一张表，不要用 `delete` 一行行删，否则效率会低一点。

优化 InnoDB Disk I/O

Q：我们刚才提到所有的操作都发生在缓存（buffer pool）里，我们怎么内存是瓶颈呢？

A：资源管理器里，如果 CPU 没有跑到 70%，那么问题可能是内存和磁盘是瓶颈，我们再去查看内存占用即可。

Optimizing InnoDB Disk I/O

- If you follow best practices for database design and tuning techniques for SQL operations, but your database is still slow due to heavy disk I/O activity, consider these disk I/O optimizations.
 - If the Unix top tool or the Windows Task Manager shows that the CPU usage percentage with your workload **is less than 70%**, your workload is probably disk-bound.
-
- Increase buffer pool size
 - When table data is cached in the InnoDB buffer pool, it can be accessed repeatedly by queries without requiring any disk I/O. Specify the size of the buffer pool with the [innodb_buffer_pool_size](#) option.
 - Adjust the flush method
 - If database write performance is an issue, conduct benchmarks with the [innodb_flush_method](#) parameter set to `O_DSYNC`.
 - Configure an `fsync` threshold
 - You can use the [innodb_fsync_threshold](#) variable to define a threshold value, in bytes.
 - Specifying a threshold to force **smaller, periodic** flushes may be beneficial in cases where multiple MySQL instances use the same storage devices.

`flush` 本身也有不同的选项，比如我们可以先让数据写进去。我们还可以设置 `dirty page` 到达 `cache` 的什么比例之后调用 `flush` 来控制 `flush` 的频率。

Optimizing InnoDB Disk I/O

- Use a `noop` or `deadline` I/O scheduler with native AIO on Linux
 - InnoDB uses the asynchronous I/O subsystem (native AIO) on Linux to perform read-ahead and write requests for data file pages. This behavior is controlled by the [innodb_use_native_aio](#) configuration option, which is enabled by default.
- Use direct I/O on Solaris 10 for x86_64 architecture
 - To apply direct I/O only to InnoDB file operations rather than the whole file system, set [innodb_flush_method = O_DIRECT](#).
- Use raw storage for data and log files with Solaris 2.6 or later
 - Users of the Veritas file system VxFS should use the `convosync=direct` mount option.
- Use **additional** storage devices
 - Additional storage devices could be used to set up a RAID configuration.
 - Alternatively, InnoDB tablespace data files and log files can be placed on **different physical disks**.

OS 中还可以绕过内存直接对硬盘访问来直接把数据写上去。实在不行我们还可以换一个硬盘，硬盘慢是慢在磁盘转得慢，那么我们不用旋转型的存储介质，可以用 SSD。但是 SSD 也不能解决全部问题，要看应用场景，在随机读取中，SSD 的性能远好于 HDD；而在顺序读取的情况下，HDD 和 SSD 其实就差不多了。SSD 里的数据只能存一年左右。

Q: 500G 的 SSD 和 2T 的 HDD，我们的数据怎么样利用这样一个存储来达到访问效率最高？

A: 我们需要设计一些维度，查看数据的访问模式，比如数据经常是全盘扫描还是查找一条记录。我们需要根据不同模式来设计是在 SSD 上还是在 HDD 上，并且数据也会发生变化，所以可能要随着时间在两者之间做迁移。然后就是，我们要控制一下在后台运行的很多线程，比如 flush 线程，需要控制一下 IO 的线程质量。

Optimizing InnoDB Configuration Variables

- Controlling the types of data change operations for which InnoDB buffers the changed data, to **avoid frequent small disk writes**.
- Turning the **adaptive hash indexing feature** on and off using the [innodb adaptive hash index](#) option.
- Setting a limit on **the number of concurrent threads** that InnoDB processes, if context switching is a bottleneck.
- Controlling **the amount of prefetching** that InnoDB does with its read-ahead operations.
- Increasing **the number of background threads** for read or write operations, if you have a high-end I/O subsystem that is not fully utilized by the default values
- Controlling **how much I/O** InnoDB performs in the background.
- Controlling the algorithm that determines when InnoDB performs **certain types of background writes**.

我们也可以设置内存中的索引是用 **hash** 还是 **b tree**，我们可以人为地根据应用来调整索引格式。

2021/11/01

InnoDB_buffer

我们上节课提到了有两个缓存，一个是 **table_open_cache**，还有一个是 **InnoDB_buffer**，前者就是在内存中打开的表格，控制着能打开多少个表格，实际上放的是表的 **handler** 句柄。而 **InnoDB_buffer** 才是在放 **data** 和 **index**。如果 **table_open_cache** 中放了很多表，数据就要加载到 **buffer** 中，如果 **buffer** 放不下了，数据就要在硬盘和 **buffer** 之间做换进换出。

buffer 就是一个 **pool**，因为里面可以放多个实例，也就是说把一块内存切成了很多块相互独立的 **buffer** 实例。**buffer** 实例的数量 * 每个 **buffer** 实例的大小(默认 128M)就是最终 **buffer** 的大小。

```
shell> mysqld --innodb-buffer-pool-size=8G --innodb-buffer-pool-instances=16
mysql> SELECT @@innodb_buffer_pool_size/1024/1024/1024;
+-----+
|      @@innodb_buffer_pool_size/1024/1024/1024 |
+-----+
|          8.000000000000000 |
+-----+


shell> mysqld --innodb-buffer-pool-size=9G --innodb-buffer-pool-instances=16
mysql> SELECT @@innodb_buffer_pool_size/1024/1024/1024;
+-----+
|      @@innodb_buffer_pool_size/1024/1024/1024 |
+-----+
|          10.000000000000000 |
+-----+
```

在前一个例子中，我们尝试分配 8G 空间给 16 个 buffer 实例，每个实例为 512M，这是允许的。如果我们尝试给每个实例多分配一些内存，它会对每个实例自动向上取整到 128M 的整数倍。在后一个例子中，就是 $128M * 5 * 16 = 10G$ 。

Q: InnoDB_buffer 为什么要有多个实例呢？

A: 因为实例之间是互相隔离不会互相影响的。带来的好处就是，如果一个表被并发访问的可能性非常大，那么我们就在多个实例上各加载一次这个表，不同操作在不同实例上可以一起操作，这样就可以提升并发读的效率。

实例最多设置 64 个。

我们现在看到的是说当我们访问的时候，先到 `table_open_cache` 尝试获取表的 handler 的，如果获取到了，那么我们去 `buffer` 中看分了几个实例，我们要去往哪个实例里加载。缓存当然越大越好，这样操作都在内存中实现。内存大了以后就可以分实例进行调整。

数据被加载的时候放到了 `buffer` 中，如果 `buffer` 放不下了，那么其中实例的数据需要被清除出 `buffer`。我们使用 LRU（最近最小使用）找到需要换出去的数据，因为它可能是脏数据（被写过了），所以需要先同步到硬盘上。

但是现在有一个问题，比如我们现在做的是 `SUM` 或者 `COUNT` 数据，需要扫描全表。刚刚加载进 `buffer` 的数据是最热的，这意味着被加载进内存的数据需要等被换出去非常慢，为了解决这个问题，我们一进来就把数据放到 `3/8`（相对来说比较冷）的位置。如果这个数据真的是经常被访问的数据，那么它会慢慢地被移动到热数据的位置，如果确实是冷数据，很快就会被 LRU 选中换出去。

Multiple Key Caches

- For a busy server, you can use a strategy that involves three key caches:
 - A “hot” key cache that takes up **20%** of the space allocated for all key caches. Use this for tables that are heavily used for searches but that are not updated.
 - A “cold” key cache that takes up **20%** of the space allocated for all key caches. Use this cache for medium-sized, intensively modified tables, such as temporary tables.
 - A “warm” key cache that takes up **60%** of the key cache space. Employ this as the default key cache, to be used by default for all other tables.

- The following example assigns several tables each to `hot_cache` and `cold_cache`:

```
CACHE INDEX db1.t1, db1.t2, db2.t3 IN hot_cache
```

```
CACHE INDEX db1.t4, db2.t5, db2.t6 IN cold_cache
```

这么做就防止大量读取数据使得内存中全部重新被换成了不一定热的数据。这个 `buffer` 在读取数据的时候，如果这个数据在硬盘上比较稳定，我们就可以通过 `optimize table` 这个动作让它内部碎片少一点。在读取的时候，因为磁盘上连续存放，我们在读数据的时候，如果 `buffer` 还有空闲，我们可以通过 `prefetch` 机制让它多读一些后面的内容进来，读多少我们是可以设置的。

Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)

- A `read-ahead` request is an I/O request to prefetch multiple pages in the `buffer pool` **asynchronously**, in anticipation of **impending need** for these pages.
- The requests bring in all the pages in one `extent`. InnoDB uses two read-ahead algorithms to improve I/O performance:
 - **Linear** read-ahead is a technique that predicts what pages might be needed soon based on pages in the buffer pool being accessed sequentially.
 - **Random** read-ahead is a technique that predicts when pages might be needed soon based on pages already in the buffer pool, regardless of the order in which those pages were read.
- The `SHOW ENGINE INNODB STATUS` command displays statistics to help you evaluate the effectiveness of the read-ahead algorithm.
 - The information can be useful when fine-tuning the `innodb random read ahead` setting.

可以设置线性读还是随机读，总的来说就是一种预测后预先读的方法。预测地越准，效率越高。

然后我们要定义一下什么时候要把内存写回到硬盘上去。

Configuring Buffer Pool Flushing

- InnoDB performs certain tasks in the background, including flushing of dirty pages from the buffer pool.
 - **Dirty pages** are those that have been modified but are not yet written to the data files on disk.
- In MySQL 8.0, buffer pool flushing is performed by page cleaner threads.
 - The number of page cleaner threads is controlled by the [innodb_page_cleaners](#) variable, which has a default value of **4**.
 - However, if the number of page cleaner threads exceeds the number of buffer pool instances, [innodb_page_cleaners](#) is automatically set to the same value as [innodb_buffer_pool_instances](#).
- Buffer pool flushing is initiated when the percentage of dirty pages reaches the low water mark value defined by the [innodb_max_dirty_pages_pct_lwm](#) variable.
 - The default low water mark is **10%** of buffer pool pages. A [innodb_max_dirty_pages_pct_lwm](#) value of **0** disables this early flushing behaviour.

如果做了写操作，我们就标记为 **dirty page**。只要脏页的数量超过一个百分比，就要触发一次自动 **flush** 的操作。往硬盘上落的时候也可以是多线程的。

The MyISAM Key Cache

- To minimize disk I/O, MyISAM storage engine employs a cache mechanism to keep the most frequently accessed table blocks in memory:
 - For **index** blocks, a special structure called the **key cache** (or key buffer) is maintained. The structure contains a number of block buffers where the **most-used index blocks** are placed.
 - For **data** blocks, MySQL uses **no special cache**. Instead it relies on the **native operating system file system cache**.
- To control the size of the key cache, use the [key_buffer_size](#) system variable.
 - If this variable is set equal to **zero**, no key cache is used. The key cache also is **not** used if the [key_buffer_size](#) value is too small to allocate the minimal number of block buffers (8).
 - When the key cache is **not** operational, index files are accessed using only the native file system buffering provided by the operating system. (In other words, table index blocks are accessed using the same strategy as that employed for table data blocks.)

Shared Key Cache Access

- Threads can access key cache buffers **simultaneously**, subject to the following conditions:
 - A buffer that is **not** being updated can be accessed by multiple sessions.
 - A buffer that is being updated causes sessions that need to use it to **wait** until the update is complete.
 - Multiple sessions can initiate requests that result in cache block replacements, as long as they **do not interfere with each other** (that is, as long as they need different index blocks, and thus cause different cache blocks to be replaced).
- Shared access to the key cache enables the server to improve throughput significantly.

MyISAM 是比较轻量级的、实现的功能比较少，但是速度快，适合于小型事务数据库，而 InnoDB 就比较大型，实现的事务更加完备，适合比较大型的应用操作。

数据库的备份和恢复

接下来我们讲数据怎么做备份和恢复，虽然讲的是 mysql，但其他数据库差不多。

1. 系统崩溃，还是可以重启，但我们要注意崩溃的时候，buffer 中的脏页有没有机会刷新到硬盘上

2. 硬盘/CPU 坏了
3. 用户误删了数据

这是不同的问题，需要有办法：1.逻辑备份 2.物理备份，我们还需要注意是全量的备份还是增量式的备份（只记录全量备份之后发生了什么事情），即便是全量备份，是把文件全部 copy 出来吗？Linux 把文件复制出来的时候，是 **copy-on-write** 机制，即使做全量备份，也

不是文件整个 `copy` 一个出来，我们要考虑到底用什么样的方式备份、以及怎么恢复。备份的内容可能比较大，最好能压缩一下。备份出来的文件是不是能加密一下，总而言之就是要维护整个系统。

1. 物理备份，一个库对应一个目录，一个表对应几个文件，比如说索引和数据。好处就是如果数据库很大，恢复速度很快。

2. 逻辑备份我们可以弄出来一个脚本自动恢复，但比较大，物理备份如果我们想从 `mysql` 导入到 `Oracle` 中是做不到的，但是逻辑备份就可以直接把数据导入到 `Oracle` 中。

在物理备份中，最小粒度到文件级别，可以把数据库中的所有文件，比如 `log`，配置文件等，这是逻辑备份做不到的。但是它会带来一个问题，内存中的内容怎么办？它就需要把 `memory dump` 到硬盘上变成临时文件导出。在恢复的时候，要求是同一版本的 `mysql`。

逻辑备份就是要把数据库的内容转化成脚本，恢复的时候执行它速度就比较慢，但是可以在数据库执行的时候去做备份。逻辑备份的方式导出来是脚本，`insert` 内容看的一清二楚，数据都是裸露出来的，但是因为是一行行插入的，备份的粒度可以更细一点，可以备份一部分数据库或者一部分的表，并且它是可移植的，可以加载到 `Oracle` 中。

Online Versus Offline Backups

- `Online` backups take place while the MySQL server is `running` so that the database information can be obtained from the server.
- `Offline` backups take place while the server is `stopped`.
- This distinction can also be described as "`hot`" versus "`cold`" backups;
 - a "`warm`" backup is one where the server remains `running but locked against modifying data` while you access database files externally.
- `Online` backup methods have these characteristics:
 - The backup is `less intrusive to other clients`, which can connect to the MySQL server during the backup and may be able to access data depending on what operations they need to perform.
 - Care must be taken to `impose appropriate locking` so that data modifications do not take place that would compromise backup integrity. The MySQL Enterprise Backup product `does such locking automatically`.

`Online` 对用户的侵入性比较小，我们想去恢复的数据库需要锁住，所以需要合适的锁的机制。离线的话就是要关掉数据库，对用户可能有影响，但是只需要覆盖文件就可以了，实现比较简单。

真正在恢复的时候，还涉及到做的是 `local` 还是 `remote` 的备份。如果我们使用 `mysql dump` 得到 `sql` 的输出文件，这样的输出的结果是可以发送回 `client`，但是如果生成的是用给定的符号的文件，就不会返回给你。

也可以做快照备份，它利用的就是 `copy-on-write` 机制，因为直接把文件复制过来占空间也耗时间，当被 `copy` 的文件发生变化的时候，此时在对应写的位置发生 `copy`，这样我们节约了空间和时间。

全量备份包含了所有数据库需要的文件。而增量备份包含了给定一个时间区域中的所有文件。增量备份是通过 `binary log` 实现的。

18 点到 19 点每隔一小时写一个 `log` 文件。我们要恢复的时候把最近的一次全量的备份 `load` 进来，然后我们去找 `binary-log` 文件去恢复期间的操作。这是全量备份和增量备份的结合。比如我们每 6 小时做全量备份，每做完全量备份，之前时间的增量备份就可以删除了。这样全量备份就可以节约空间，因为增量备份很占空间。

在备份的时候，如果使用的是物理备份，那么需要把目录中的所有文件全部 `copy` 出来，必须保证文件是最新的，把表 `flush` 到硬盘上。在 `copy` 的时候，我们需要加锁，等于在写硬盘上这个表。

2021/11/04

我们可以使用 MySQL dump 工具，它可以使用多线程、文件压缩方式并发地把数据库导出到硬盘上。Dump 出来的语句可能是一堆 create 和一堆 insert，可以 dump 出来纯文本格式，也可以 dump 出来恢复的脚本。

Dump 文件可以用来备份、设置 replica、设置测试环境等。

Dumping Data in SQL Format with mysqldump

- By default, **mysqldump** writes information as SQL statements to the standard output. You can save the output in a file:
`shell> mysqldump [arguments] > file_name`
- To dump all databases, invoke **mysqldump** with the **--all-databases** option:
`shell> mysqldump --all-databases > dump.sql`
- To dump only specific databases, name them on the command line and use the **--databases** option:
`shell> mysqldump --databases db1 db2 db3 > dump.sql`
- To dump a single database, name it on the command line:
`shell> mysqldump --databases test > dump.sql`
- In the single-database case, it is permissible to omit the **--databases** option:
`shell> mysqldump test > dump.sql`
- To dump only specific tables from a database, name them on the command line following the database name:
`shell> mysqldump test t1 t3 t7 > dump.sql`

Mysqldump 可以指定数据库和数据库中的指定的表 dump 出来。Dump 出来之后，就可以重新读进去恢复了。

mysqldump 备份（全量+增量）方案操作记录

<https://blog.csdn.net/fuzhongfaya/article/details/80969073>

Reloading Delimited-Text Format Backups

- To reload a table, first change location into the output directory. Then process the **.sql** file with **mysql** to create an empty table and process the **.txt** file to load the data into the table:
`shell> mysql db1 < t1.sql`
`shell> mysqlimport db1 t1.txt`
- An alternative to using **mysqlimport** to load the data file is to use the **LOAD DATA** statement from within the **mysql** client:
`mysql> USE db1;`
`mysql> LOAD DATA INFILE 't1.txt' INTO TABLE t1;`
- If you used any data-formatting options with **mysqldump** when you initially dumped the table, you must use the same options with **mysqlimport** or **LOAD DATA** to ensure proper interpretation of the data file contents:
`shell> mysqlimport --fields-terminated-by=, --fields-enclosed-by=''' --lines-terminated-by=0x0d0a db1 t1.txt`
- Or:
`mysql> USE db1;`
`mysql> LOAD DATA INFILE 't1.txt' INTO TABLE t1 FIELDS TERMINATED BY ',' FIELDS ENCLOSED BY ''' LINES TERMINATED BY '\r\n';`

导出.sql 成脚本的缺点：不直观、比较耗费空间，好处就是如果导出的是标准 SQL 语句，可以在 Oracle 中复用脚本恢复数据库。

导出为.txt 纯文本文件：比较紧凑，是纯数据，但是依赖于 import 工具；并且这种文件是可以使用 excel 打开并可视化的。

所以用哪一个都可以实现我们的目的，完全取决于 dump 出来的文件我们想怎么用。

Mysqldump 工具的使用方式略。

Partition 就是把一张表拆分成多张表，可以存在多台机器上，也可以存在一台机器上的不同文件中。也就是防止出现特别大的问题，导致我们既存不下也不能做高效的检索。我们把数据存到不同的表里，可以通过水平的切分，也可以把不常用的字段放到一张表里（垂直切分），但是垂直切分需要我们自己做，MySQL 不能帮我们去做。

分区的时候最常见的就是对一列的取值范围进行划分，比如学生类型：本科生、硕士生、博士生，我们就可以划分成三张表。我们也可以按照（主键）哈希去做分区，所以在做分区的时候，最重要的是我们分区的逻辑是什么。

2021/11/08

MySQL 的分区

分区的种类：

- 按范围分区，这种会根据列的值是否落在一个给定的范围中进行分区。
- 按照 List 分区，分区基于列是否等于一些离散的值来进行分区。
- 按照哈希分区，分区是基于程序员定义的哈希表达式来进行划分的。哈希函数输入是每行的列的值，并且这个哈希函数必须对任何有效的 MySQL 表达式都有效。
- 按照 Key 来分区，和哈希分区类似，只是可以通过一列或多列进行计算，并且 MySQL 提供了其哈希函数。这些列可以包括任意类型的值，因为这个 MySQL 函数可以保证输出一个整数的结果。

一个非常常见的操作就是根据日期来进行分区

```
CREATE TABLE members (
    firstname VARCHAR(25) NOT NULL, lastname VARCHAR(25) NOT NULL,
    username VARCHAR(16) NOT NULL, email VARCHAR(35),
    joined DATE NOT NULL
)
PARTITION BY KEY(joined)
PARTITIONS 6;
```

其他划分类型要求一个分区表达式，产生的是整数或者 NULL

```
CREATE TABLE members (
    firstname VARCHAR(25) NOT NULL, lastname VARCHAR(25) NOT NULL,
    username VARCHAR(16) NOT NULL, email VARCHAR(35),
    joined DATE NOT NULL
)
PARTITION BY RANGE( YEAR(joined) ) (
    PARTITION p0 VALUES LESS THAN (1960), PARTITION p1 VALUES LESS THAN (1970),
    PARTITION p2 VALUES LESS THAN (1980), PARTITION p3 VALUES LESS THAN (1990),
    PARTITION p4 VALUES LESS THAN MAXVALUE
```

```
);
```

1. 按范围分区

如果是 `null` 的情况，我们可以插入到最小的分区里面。`Null` 如果没用的是 `list` 并且没有明确指示的情况下，插入 `null` 就会报错。

表中的日期超过了 2000 年以后要求还能用怎么办呢？

```
CREATE TABLE members (
    id INT,
    fname VARCHAR(25),
    lname VARCHAR(25),
    dob DATE
)
PARTITION BY RANGE( YEAR(dob) ) (
    PARTITION p0 VALUES LESS THAN (1980),
    PARTITION p1 VALUES LESS THAN (1990),
    PARTITION p2 VALUES LESS THAN (2000)
);
ALTER TABLE members ADD PARTITION (PARTITION p3 VALUES LESS THAN (2010));
```

1. 在后面添加一个分区，使用 `ALTER` 指令来添加一个新的分区，小于 2010 即可。

```
mysql> ALTER TABLE members
> ADD PARTITION (
> PARTITION n VALUES LESS THAN (1970));
ERROR 1463 (HY000): VALUES LESS THAN value must be strictly » increasing for each
partition
ALTER TABLE members
REORGANIZE PARTITION p0 INTO (
    PARTITION n0 VALUES LESS THAN (1970),
    PARTITION n1 VALUES LESS THAN (1980)
)
```

但是如果我们要在中间插入的话，按照上文这样做就会报错，我们需要使用 `REORGANIZE` 来调整，把原先的一个分区切分成两个分区。

```

CREATE TABLE tt (
    id INT,
    data INT
)
PARTITION BY LIST(data) (
    PARTITION p0 VALUES IN (5, 10, 15),
    PARTITION p1 VALUES IN (6, 12, 18)
);

ALTER TABLE tt ADD PARTITION (PARTITION p2 VALUES IN (7, 14, 21));

mysql> ALTER TABLE tt ADD PARTITION
    > (PARTITION np VALUES IN (4, 8, 12));
ERROR 1465 (HY000): Multiple definition of same constant » in list partitioning

```

我们要添加一个新的(7,14,21)，就追加就可以了，因为(7,14,21)和原先的分区的值是没有重复的。如果我们追加的一个重复的，那么就会报错。其实也很好理解，分区的值是不能重叠的，如果重叠了，那么它就不知道到底要放到哪个分区里了。

```

CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    hired DATE NOT NULL
)
PARTITION BY RANGE( YEAR(hired) ) (
    PARTITION p1 VALUES LESS THAN (1991),
    PARTITION p2 VALUES LESS THAN (1996),
    PARTITION p3 VALUES LESS THAN (2001),
    PARTITION p4 VALUES LESS THAN (2005)
);
ALTER TABLE employees ADD PARTITION (
    PARTITION p5 VALUES LESS THAN (2010),
    PARTITION p6 VALUES LESS THAN MAXVALUE
);

```

我们在追加的时候，我们可以一次性追加多个分区。

```

ALTER TABLE members REORGANIZE PARTITION n0 INTO (
    PARTITION s0 VALUES LESS THAN (1960),
    PARTITION s1 VALUES LESS THAN (1970)
);

ALTER TABLE members REORGANIZE PARTITION s0,s1 INTO (
    PARTITION p0 VALUES LESS THAN (1970)
);

ALTER TABLE members REORGANIZE PARTITION p0,p1,p2,p3 INTO (
    PARTITION m0 VALUES LESS THAN (1980),
    PARTITION m1 VALUES LESS THAN (2000)
);

ALTER TABLE tt ADD PARTITION (PARTITION np VALUES IN (4, 8));
ALTER TABLE tt REORGANIZE PARTITION p1,np INTO (
    PARTITION p1 VALUES IN (6, 18),
    PARTITION np VALUES IN (4, 8, 12)
);

```

重组的时候，可以把多个合成一个，也可以把原先的重新组织成多个。其实 REORGANIZE 其实就是把原先的值重新划分一下。

哈希就是用散列算法，如果要去掉一个分区就非常麻烦，所以就不能通过 ALTER 直接删掉，要使用 merge 算法。

```

CREATE TABLE clients (
    id INT,
    fname VARCHAR(30),
    lname VARCHAR(30),
    signed DATE
)
PARTITION BY HASH( MONTH(signed) )
PARTITIONS 12;
mysql> ALTER TABLE clients COALESCE PARTITION 4;
Query OK, 0 rows affected (0.02 sec)
ALTER TABLE clients ADD PARTITION PARTITIONS 6;

```

看起来像是删掉了 8 个分区，但是删除对于哈希是很难理解的，它未来不知道是按照 4 去散列还是 12 去散列，所以 MySQL 要求我们写上 COALESCE PARTITION 4。这样 MySQL 就会帮我们重新散列一下计算。

分区分完了有一个功能是比较特殊的，可以从一个表里把一个分区和另一个表的分区互换一下。

```

CREATE TABLE e (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30)
)
PARTITION BY RANGE (id) (
    PARTITION p0 VALUES LESS THAN (50),
    PARTITION p1 VALUES LESS THAN (100),
    PARTITION p2 VALUES LESS THAN (150),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
INSERT INTO e VALUES
(1669, "Jim", "Smith"),
(337, "Mary", "Jones"),
(16, "Frank", "White"),
(2005, "Linda", "Black");

mysql> CREATE TABLE e2 LIKE e;
Query OK, 0 rows affected (0.04 sec)
mysql> ALTER TABLE e2 REMOVE PARTITIONING;
Query OK, 0 rows affected (0.07 sec)
Records: 0 Duplicates: 0 Warnings: 0
mysql> SELECT PARTITION_NAME, TABLE_ROWS
FROM INFORMATION_SCHEMA.PARTITIONS
WHERE TABLE_NAME = 'e';

```

E2 和 e1 在表的结果是一模一样的，但是 e 分了 4 个区，而 e2 没有分区。其中 p0 里有 1 条，p3 里有 3 条。

```

+-----+-----+
| PARTITION_NAME | TABLE_ROWS |
+-----+-----+
| p0 | 1 |
| p1 | 0 |
| p2 | 0 |
| p3 | 3 |
+-----+-----+
2 rows in set (0.00 sec)

```

```

mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2;
Query OK, 0 rows affected (0.04 sec)
mysql> SELECT PARTITION_NAME, TABLE_ROWS
  FROM INFORMATION_SCHEMA.PARTITIONS
 WHERE TABLE_NAME = 'e';

```

```

+-----+-----+
| PARTITION_NAME | TABLE_ROWS |
+-----+-----+
| p0 | 0 |
| p1 | 0 |
| p2 | 0 |
| p3 | 3 |
+-----+-----+
4 rows in set (0.00 sec)

```

```
mysql> SELECT * FROM e2;
```

```

+---+-----+-----+
| id | fname | lname |
+---+-----+-----+
| 16 | Frank | White |
+---+-----+-----+
1 row in set (0.00 sec)

```

然后我们互换了这个信息，我们发现 e 里面的 p0 分区数据就没有了。而 e2 里就有 e 的 p0 中的数据了。

```

mysql> INSERT INTO e2 VALUES (51, "Ellen", "McDonald");
Query OK, 1 row affected (0.08 sec)
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2;
ERROR 1707 (HY000): Found row that does not match the partition
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2 WITHOUT
VALIDATION;
Query OK, 0 rows affected (0.02 sec)

```

我们如果插入了新的 e2，一个 51，我们发现一开始是因为 e 中的 p0 划分要求是小于 50，理论上是不能换过去的。但是我们可以添加上 WITHOUT VALIDATION 强制换过去，但是这会导致搜索的时候可能找不到 51，因为搜索的时候会按照分区做剪枝。

```
"cutting away" of unneeded partitions is known as pruning
CREATE TABLE t1 (
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    region_code TINYINT UNSIGNED NOT NULL,
    dob DATE NOT NULL
)
PARTITION BY RANGE( region_code ) (
    PARTITION p0 VALUES LESS THAN (64),
    PARTITION p1 VALUES LESS THAN (128),
    PARTITION p2 VALUES LESS THAN (192),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
SELECT fname, lname, region_code, dob
    FROM t1
   WHERE region_code > 125 AND region_code < 130;
```

按照分区分完了之后，搜索的时候就会根据

分区就是先对表建立分区的索引，如果搜索是基于分区的一列或者某几列，它就会在搜索的时候根据这几个索引去找，有助于提高性能。MySQL 在不同的区的数据可以在不同的设备和机器上。到此就是 MySQL 的这些优化。

MongoDB

我们要看一看关系型数据库之外的数据库。

一个例子：输入编号出现名字+年龄+图片。其中图片在 MongoDB 中。

可视化工具：MongoDB Robo 3T，其中就是一个 id+一个 base64 的字符串。关系型数据库中就是 id:age:name。

但是这个 id 是分别去关系型数据库中找的数据，去 MongoDB 找到的图片组合在一起发到前端。注意这是在 DAO 这一层组合的。

我们定义的 person 是这样定义的 @Table(name="persons")，会有 id、姓名等。底下多了一个 private PersonIcon icon；我们需要告诉 SpringJPA 这个东西不归你管，于是我们加了一个 @Transient 指明这个并不是关系型数据库表中的字段。

于是我们又有一个 PersonIcon 类。

```
@Document(collection = "personicon")
Public class PersonIcon {
    @Id
    private int id;
    private String imgBase64;
    public PersonIcon() ...
}

class PersonIconRepository extends MongoRepository<PersonIcon, Integer> {
```

```
}
```

在 DAO 这一层我们只要

```
@Autowired  
private PersonIconRepository personIconRepository;
```

在 `findOne` 中

我们分别得到不含 Icon 的 Person，和一个 Optional 类型的<PersonIcon>我们只需要手动组装在一起即可。

而在 Controller 和 Service 视角里，就只有完整的 Person 了，带有完整的 Icon 信息。

数据没有严格 schema 的情况下，关系型数据库就不太支持了。

硬盘的容量很容易做的，但读写速度提升相对比较慢。为了解决这个问题，我们想到的方法是从多个盘上读数据。

如果我们把数据错开读取，我们就可以并行地从 100 块硬盘上读取数据。

但是故障率是指数增加的，我们怎么恢复呢？比如多花五分之一的数据做异或的备份，也就是产生 replica 来解决故障的问题。

第二个问题是，我们的数据是在多个机器上，我们这该怎么处理呢？

如果我们将 RDBMS 的数据分发到多个物理机器中，会发生什么？

单张表的问题——当执行 SQL 语句时，表将被乐观/悲观地锁定，以确保数据的完整性。
- 共享锁允许其他线程（读取但不写入表-排除锁拒绝从其他线程）的任何访问，即其他语句将排队并等待锁释放。
- 对于后者，如果语句的数量较大，则性能相当差。单个表的索引树比较大，效率降低，不如一堆比较小的索引树。

垂直切分

不常访问的列拆成另一张表。

关系型数据库的集群版很难做，所以价格非常高。

- *Structured data*
 - is data that is organized into entities that have a defined format, such as XML documents or database tables that conform to a particular predefined schema.
 - This is the realm of the RDBMS.
- *Semi-structured data*
 - on the other hand, is looser, and though there may be a schema, it is often ignored, so it may be used only as a guide to the structure of the data
 - for example, a spreadsheet, in which the structure is the grid of cells, although the cells themselves may hold any form of data.
- *Unstructured data*
 - does not have any particular internal structure
 - for example, plain text or image data.

关系型数据库很难胜任非结构性的 schema

- **Bigtable:** A Distributed Storage System for Structured Data
 - ACM Transactions on Computer Systems, 2008, 26:1-26.
 - http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/zh-CN//archive/bigtable-osdi06.pdf
- **Dynamo:** amazon's highly available key-value store
 - Symposium on Operating Systems Principles, 2007:205-220.
 - <http://web.archive.org/web/20120129154946/http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf>
- **Cassandra**
 - <http://cassandra.apache.org/>
- **MemcacheDB**
 - <http://memcachedb.org/>
- **Apache CouchDB**
 - <http://couchdb.apache.org/>
- **MongoDB**
 - <http://www.mongodb.org/>

Document 就是记录，存放了键值对。很多文档就是 collection，collection 中的 document 互相之间格式可以不一样。每个文档一定有 “_id” 这个 key，其他的 key 都是我们写进去的，collection 为了

2021/11/11

假如我们现在想存储这样的数据

	Apple	Samsung
screen	S1	S2
Standby time	S3	S4
quality	S5	
price		S6

我们可以定义其中无值的地方为空。这样在对应的 attribute 列 (quality、price) 不能设置为 not null。

方法 2 是弄出来三张表，

表一：

id	Apple	Samsung
1	S1	S2
2	S3	S4

表二：

id	Quality
1	S5

表三：

id	price
2	S6

方法 3 是两张表

id	S	ST	Q
1	S1	S2	S5

id	S	ST	P
2	S3	S4	S6

但是来新手机的时候，我们还是要改很多东西。所以我们考虑把列转成行来存：

id	brand	name
1	APPLE	Iphone10
2	SAMSUNG	S10

Id	pid	name	value
1	1	Screen	S1
2	1	Stand time	S2
3	1	Quality	S5
4	2	Screen	S3

这样我们就不用每个手机有不同的参数。

像我们刚才说的

```
INSERT INTO 'mobiles' ('id', 'name', 'brand') VALUES
(1, 'iphone10', 'apple')
(2, 'S10', 'Samsung')

INSERT INTO 'mobile params' ('id', 'mobile_id', 'name', 'value') VALUES
...
```

所以我们要先做表的连接操作，但是如果第二张表很大的话，可能就比较大。

MongoDB 中的例子：

```
MongoDB
db.mobiles.ensureIndex({
  "params.name": 1,
  "params.value": 1
});
db.mobiles.insertOne({
  name: "iPhone Xs Max",
  brand: "Apple",
  params: [
    {name: "Standby time", value : 200},
    {name: "Screen", value : "OLED"},
    {name: "Quality", value : "SSS"}
  ]
});

MongoDB
db.mobiles.insertOne({
  name: "S10",
  brand: "Samsung",
  params: [
    {name: "Standby time", value : 300},
    {name: "Screen", value : "Curve"},
    {name: "Price", value : "Attractive"}
  ]
});
```

我们往里插入 `document` 的时候，有三个键值对，手机的名字、牌子、一个嵌套的键值对，里面就是三个参数，第三个就是多出来的和三星手机不一样的属性。我们发现这两个的 `schema` 是不完全一致的，有自己特殊的属性。

MongoDB

```
db.mobiles.find({  
    "params": {  
        $all: [  
            {$elemMatch: {"name": "Standby time", "value": {$gt: 100}}},  
            {$elemMatch: {"name": "Screen", "value": "OLED"}}  
        ]  
    }  
});  
  
db.mobiles.find({  
    "params": {  
        $all: [  
            {$elemMatch: {"name": "Standby time", "value": {$gt: 100}}},  
            {"$elemMatch: {"name": "Price", "value": "Attractive"}}  
        ]  
    }  
});
```

我们可以使用 MongoDB 中的查询语句。

Querying

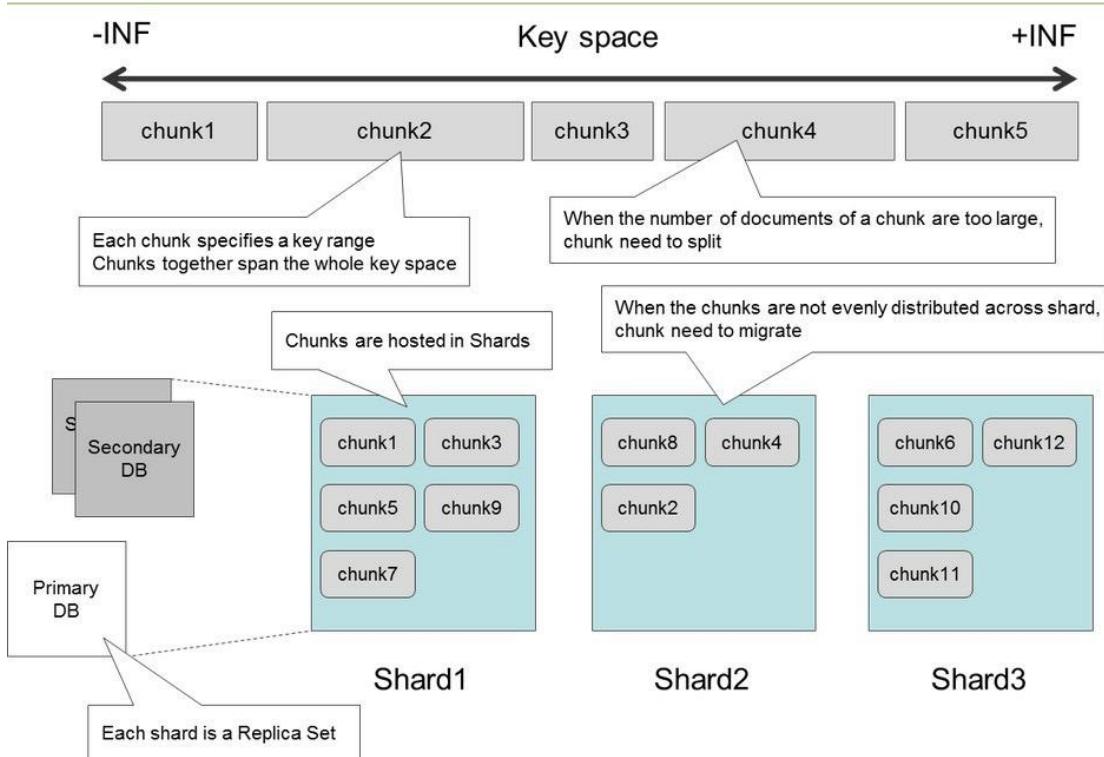


- Cursor Internals
 - There are two sides to a cursor: the **client-facing cursor** and the **database cursor** that the client-side one represents
- On the client side
 - > db.c.find().limit(3)
 - > db.c.find().skip(3)
 - > db.c.find().sort({username : 1, age : -1})
 - > db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})
 - > db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})

 - > // do not use: slow for large skips
 - > var page1 = db.foo.find(criteria).limit(100)
 - > var page2 = db.foo.find(criteria).skip(100).limit(100)
 - > var page3 = db.foo.find(criteria).skip(200).limit(100)

MongoDB 也可以指定升序还是降序，也可以建立地理信息的投影。

如果 MongoDB 存不下了，就会做 sharding，把很大的数据打成碎块，这就是 MongoDB 实现不断扩展的方式。我们拿很多服务器来存，每个服务器叫做 shard server。也就是我们把一张很大的表切成很多块存在不同的服务器上。我们就需要一个配置文件（存在 config server）来记录一下表里有多少个 shard，存在哪些 shard server 上。



当我们插入一个 `document` 到 `collection` 中，会自动赋值 `_id`，它的取值范围是 `-INF` 到 `INF`，这个 `key` 不是整数递增的。它太大了在一个机器上很难存的时候，它就会按照 `KEY` 的范围来切分成 `chunk`。一开始 `chunk` 分的比较少，当有一个 `document` 要插入到 `collection` 的时候，会自动生成 `_id`，我们就知道要插入到哪个 `chunk` 中。那么我们找到对应的 `shard server` 中。我们约定 `chunk` 的大小为 `64M`，如果超过了 `64M`，一个 `chunk` 就分裂成两个，重新切分 `key` 的范围。最后切着切着就出现了很多 `chunk` 存在不同的 `shard server` 上。并且 `MongoDB` 会自动保证最少的 `shard server` 和最多的 `shard server` 存储的 `chunk` 最多差 `2` 个，这样数据就存放得非常平衡了。

图数据库比起关系型数据库的优势

What is a Graph?

REin
REliable, Intelligent & Scalable Systems

- Formally, a graph is just a collection of *vertices* and *edges*
 - or, in less intimidating language, a set of *nodes* and the *relationships* that connect them.
 - Graphs represent entities as nodes and the ways in which those entities relate to the world as relationships.

The diagram shows three types of nodes: User, User, and Message. The first two User nodes have names: 'name: Ruth' and 'name: Harry'. The third node is a Message with attributes 'id: 101' and 'content: ...'. Relationships between nodes are labeled 'FOLLOWS'. Arrows point from Ruth to Harry and from Harry to Ruth. A vertical stack of three Message nodes below the first one is labeled 'CURRENT' at the top, 'PREVIOUS' in the middle, and 'PREVIOUS' again at the bottom. Arrows point from the top Message to the middle one, and from the middle one to the bottom one. The number '3' is centered at the bottom of the diagram.

Current 表示最先发的推文的什么。节点上有很多属性，发的新消息就可以用时序串起来。所以节点分为 message 和 user。关系有三种：follows、current、previous

图数据库就是专门用来存储图数据的，从存储引擎来说，它描述数据/存储数据的方式都完全不一样，并且它有自己的查询语句去做不同的查询。

如果使用关系型数据库来搜索谁买过草莓冰淇淋，这就很难，需要 join 四张表。

User					
UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...
99	Zach	99 South St.		zach@example.org	

Order	
OrderID	UserID
1234	1
5678	1
...	...
5588	99

LineItem		
OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...
5588	765	1

Product		
ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

因为关系型数据库面对 many-to-many 问题的时候需要拆成两个 one to many 问题，在关系型数据库中，它的技术特点要求我们这么建模，已经不代表真实数据的关系了，所以会反复地去做表和表之间的关联。

在关系型数据库中维护好友关系

Relational Databases Lack Relationships

Person		PersonFriend	
ID	Person	PersonID	FriendID
1	Alice	1	2
2	Bob	2	1
...	...	2	99
99	Zach
		99	1

Example 3. Alice's friends-of-friends

```
SELECT p1.Person AS PERSON,
       p2.Person AS FRIEND_OF_FRIEND
  FROM PersonFriend pf1 JOIN Person p1
    ON pf1.PersonID = p1.ID
   JOIN PersonFriend pf2
    ON pf2.PersonID = pf1.FriendID
   JOIN Person p2
    ON pf2.FriendID = p2.ID
   WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

Example 1. Bob's friends

```
SELECT p1.Person
  FROM Person p1 JOIN PersonFriend
    ON PersonFriend.FriendID = p1.ID
   JOIN Person p2
    ON PersonFriend.PersonID = p2.ID
   WHERE p2.Person = 'Bob'
```

Example 2-2. Who is friends with Bob?

```
SELECT p1.Person
  FROM Person p1 JOIN PersonFriend
    ON PersonFriend.PersonID = p1.ID
   JOIN Person p2
    ON PersonFriend.FriendID = p2.ID
   WHERE p2.Person = 'Bob'
```

这样存完以后，如果我们要找 Bob 的好友，就要 2 次 join 才能得到结果。而查询“谁是 Bob 的朋友”也要做两次 join。

如果我们做好友推荐，也就是查询 Alice 好友的好友是谁，很显然我们就要做三次 join 操作。

如果我们使用 NOSQL 存储，我们拿到 key 之后，我们还是要类似地在另一个的 collection 中去找。

用图数据库来表示好友关系

所以就有图数据库来表达好友关系：

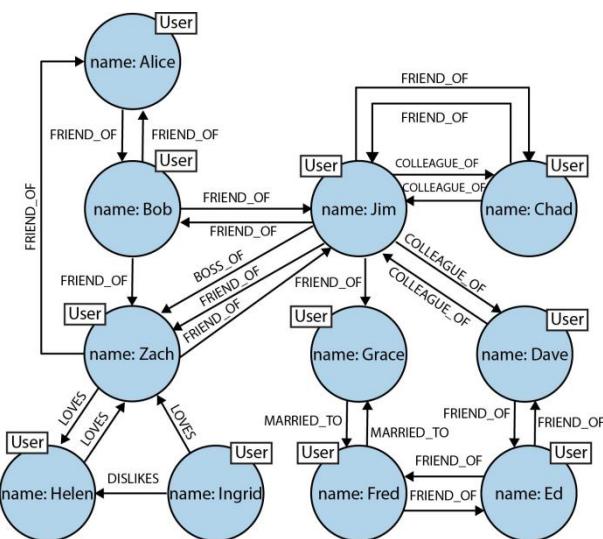
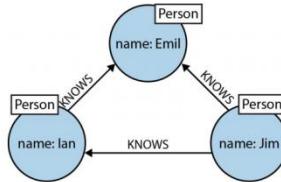


图 1 图数据库存储好友关系

我们就需要知道这个图是怎么存储的、怎么查询的。这就叫做一个带标签属性的图，每个图上的节点有一个标签标识着这个节点是什么。

- **Querying Graphs: Cypher**

- Cypher is an expressive (yet compact) graph database query language.



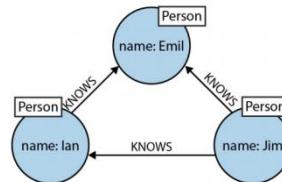
- Here's the equivalent ASCII art representation in Cypher:

```
(emil:Person {name:'Emil'}) <[:KNOWS]-(jim:Person {name:'Jim'}) -[:KNOWS]->(ian:Person {name:'Ian'}) -[:KNOWS]->(emil)
```

在 neo4j 中提供了一种描述图的语法 cypher，比如有一个叫 person 的对象，节点中带了一系列属性。描述完节点以后用箭头来表示出关系的方向。

Querying Graphs: Cypher

- Cypher is an expressive (yet compact) graph database query language.



Match:

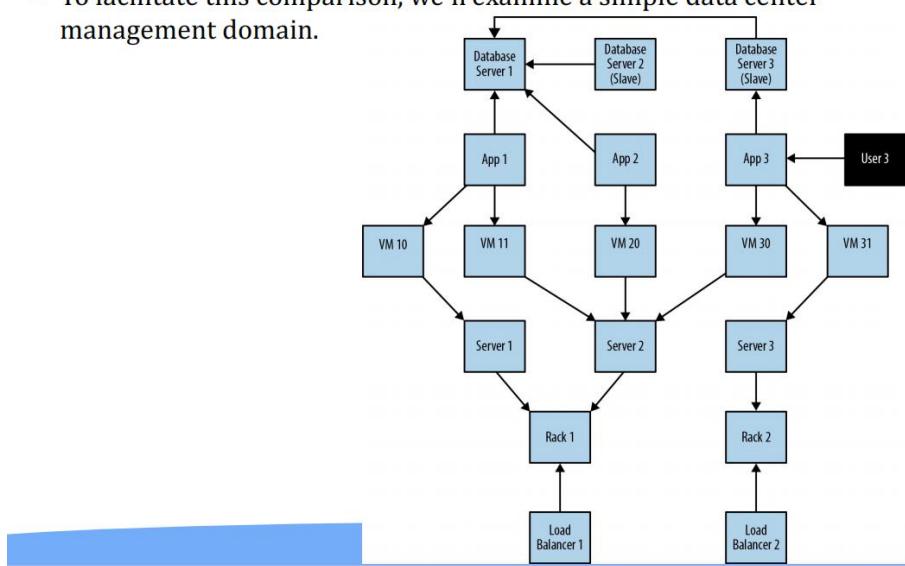
```

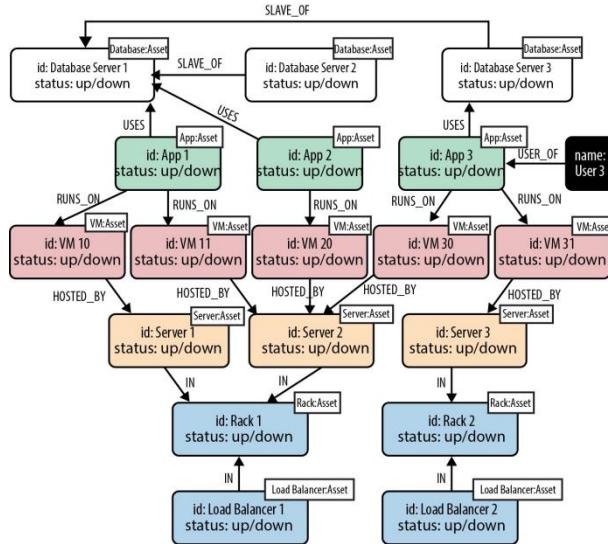
MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
WHERE a.name = 'Jim'
RETURN b, c
  
```

可以通过这个查询 Jim 认识的或一次关系间接认识的人。把 a 替换成 jim，去找即可。

A Comparison of Relational and Graph Modeling

- To facilitate this comparison, we'll examine a simple data center management domain.





一个 user 通过 1 次关系~5 次关系能访问到的所有的 asset,

Test Graph Model

- When a user reports a problem, we can limit the physical fault-finding to problematic network elements between the user and the application and its dependencies.
- In our graph we can find the faulty equipment with the following query:
 - `MATCH (user:User)-[*1..5]-(asset:Asset)`
 - `WHERE user.name = 'User 3' AND asset.status = 'down' RETURN DISTINCT asset`
- This allows us to match paths such as:

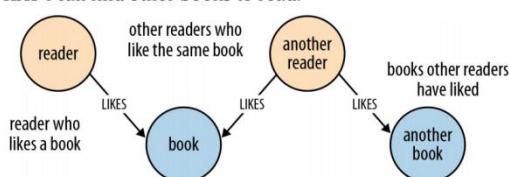

```
(user)-[:USER_OF]->(app)
(user)-[:USER_OF]->(app)-[:USES]->(database)
(user)-[:USER_OF]->(app)-[:USES]->(database)-[:SLAVE_OF]->(another-database)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)-[:IN]->(rack)
(user)-[:USER_OF]->(app)-[:RUNS_ON]->(vm)-[:HOSTED_BY]->(server)-[:IN]->(rack) <-[:IN]->(load-balancer)
```

我们拿到一个问题以后，我们怎么考虑这个图数据库该怎么建立呢？

Building a Graph Database Application



- Describe the Model in Terms of the Application's Needs**
- Here's an example of a user story for a book review web application:
 - AS A reader who likes a book, I WANT to know which books other readers who like the same book have liked, SO THAT I can find other books to read.



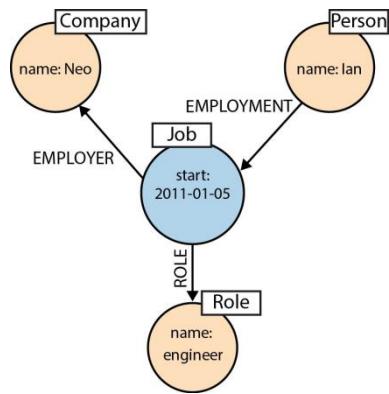
– since Alice likes Dune, find books that others who like Dune have enjoyed:

```
MATCH (:Reader {name:'Alice'})-[:LIKES]->(:Book {title:'Dune'})
    <-[:LIKES]->(:Reader)-[:LIKES]->(:Book)
RETURN books.title
```

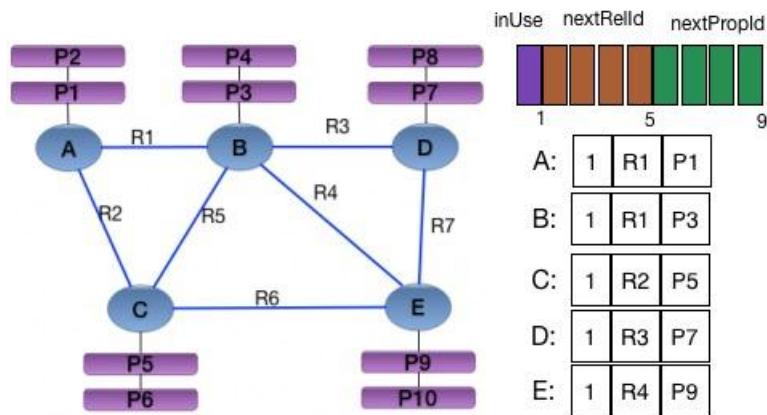
协同过滤：如果 Alice 喜欢沙丘这本书，去看看其他喜欢沙丘的书的人还喜欢什么书，推荐给 Alice。

在建模的时候，我们需要把我们识别出来的实体定义成节点，实体之间的链接定义成边。节点上有属性、边上也有属性。比如节点表示账户，而边上表示转账的金额。我们除了看到

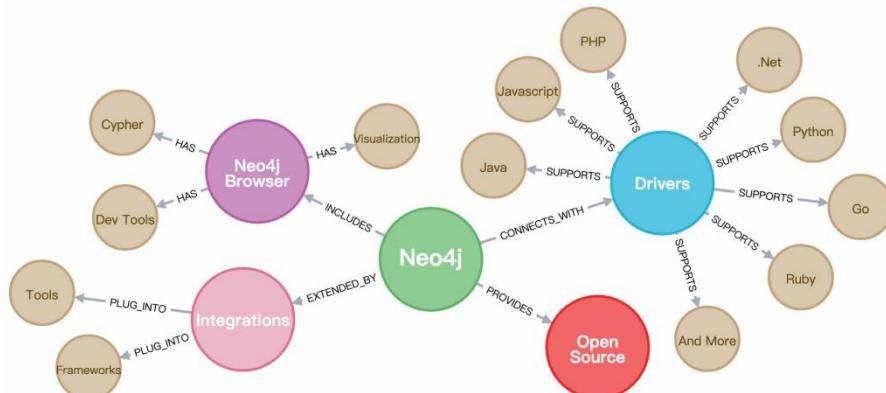
节点有属性，我们看到的一些事实也可以建立为节点。比如一个人被一个公司聘任为工程师，这是一个事实也可以抽象成为一个节点。



Neo4j 可以单独跑，也可以像 package 一样运行在应用服务器中。也可以去做分布式的，一个大图切成很多子图存储。



ABCDE 有 R1 到 R7 在做连接，每个节点上还有一些属性。它是怎么存储的呢？节点是下一个关系是谁，下一个属性是什么。



2021/11/15

图计算概论

我们再讲一下上节课没讲的部分，一张图上，它的顶点和边上都可以有属性。比如银行转账的操作的金额，就可以当做边的属性。现在的问题就是，我们能不能根据一个图回答一个人（账号）是否是一个社交活跃分子/银行账户中谁在洗钱/非法集资。因为洗钱会经过一个非常长的链条，可能过程中会分散出去，最终才会回来。到底链条多长才算洗钱呢？没有一个固定的值，所以我们想能不能用机器学习去把这个模式去学出来，机器学习能不能根据节点数据和关联数据训练一个洗钱与否的分类器。

图数据上的分类问题

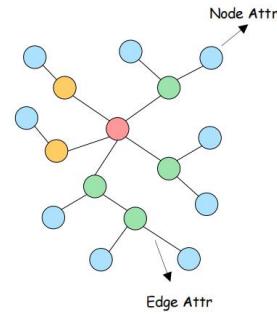
- 图粒度 vs. 节点粒度
- 节点特征 vs. 边特征

应用场景：

- 应用对象：社交网络、交易网络（边特征敏感）
- 应用任务：用户分类、异常行为检测…

图神经网络：

- 传统图神经网络未考虑边特征对于分类结果的影响
 - 谱图理论：Graph Convolutional Network (GCN)
 - 图注意力：Graph Attention Network (GAT)
- 仅有少数研究考虑了GNN在边特征上的拓展
 - 仅可处理标签化的边特征 (R-GCN)
 - 仅将边特征视为权重进行处理 (EGNN)

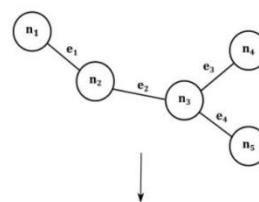


那么这个图我们该怎么放入卷积神经网络中卷积呢？图怎么样组织成输入呢？比如我们约定，对于一个节点来说，我们把选它 10 个邻居节点（多则随机选，少则补空节点）。这样我们对每个节点都这样处理，比如有 100 个节点，那么我们就有 $100 * 11$ 的矩阵，我们可以选择 $1 * 3$ 的卷积核。然后得到特征矩阵，我们继续去做卷积，继续提取特征，以此类推。最终这个神经网络就可以对每个节点分类成正常节点/洗钱节点等。

在这里比较难办的是：这个神经网络都是把节点特征拿去组合，可以解决边上是没有属性的图输入。但是在洗钱这个 case 中，边上的转账信息其实是更加重要的，所以我们需要把边的信息也 encoding 我们的 input 中，在提取和传递特征的时候，节点和边上的特征都要考虑进去。

➤ 边特征的集合形式 → 邻接形式

$$\vec{e}_p \rightarrow \vec{e}_{ij}$$



➤ 边映射矩阵 M_E

- 维度大小为 $N \times N \times M$
- 第三维度：one-hot 编码
- 可以预先构造

➤ 高维矩阵乘法

$$M_E \cdot E = E^*$$

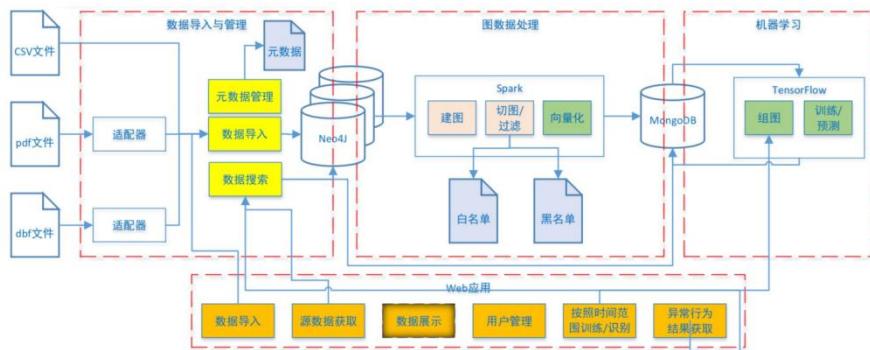
1	2	3	4	5
1		1		
2	1		2	
3		2		3 4
4			3	
5			4	

\vec{e}_1				
\vec{e}_2				
\vec{e}_3				
\vec{e}_4				

1		\vec{e}_1		
2	\vec{e}_1		\vec{e}_2	
3		\vec{e}_2		\vec{e}_3
4			\vec{e}_3	
5			\vec{e}_4	

图 3-3 图上的边特征映射变换举例

- 根据实体关联数据构建图 -> 根据样例学习图中的模式 -> 发现符合预期模式的数据，进行预警



日志结构的数据库

它里面数据结构是日志结构的合并树。我们讲讲数据库是怎么实现，这种数据库里面的数据库的表都是 SSTable。现在用的比较多的叫做 rocksDB 和 levelDB。这种数据库的好处就是冷数据都沉在底下，而热数据都在上面，每一层的尺寸不一样，它就比较支持那种日志数据。

日志的特点：

- 写入之后不会改写
- 数据有冷热之分，我们总是想读到刚刚写入的数据，对于刚刚写入的数据希望有更快的读取性能

这种数据库用到了 SSTable，它可以按照行存，也可以按列存。这种混合事务处理负载是怎么被支持的，也就是我们要支持事务性的处理和分析性的处理。因为我们之前和阿里有一个 rocksDB 从行存数据库改成行列存储都支持的混合处理的数据库。

行列混合负载

我们先来解释一些这种混合负载是什么意思。在关系型数据库中，MySQL 中的 Book 一张表，里面有各种各样的不同的字段。id:author:name:time。

比如我们现在双十一要往 order 表中插入数据。那么这种数据是一行行存比较好，也就是在硬盘上按照行连续存储。当我们要更新书的时候，我们就可以更新整条记录。我们要把记录从硬盘中完整地加载到内存中，按行存保证一次加载就可以完全加载到内存里，写回硬盘的时候也保证了连续的访存，写的速度比较快。对于我们操作一条条记录的 transaction 类的查询和修改，这种就没什么问题。

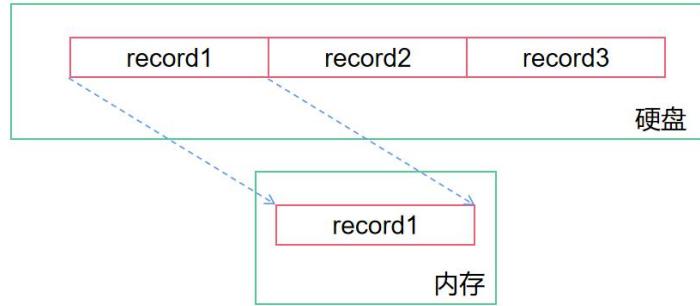


图 2 事务类操作按行存的情况

还有一种是在双十一结束了，我们希望统计销量。如果现在还是按行存，我们就需要在每个 record 中取出一部分拿到内存中。

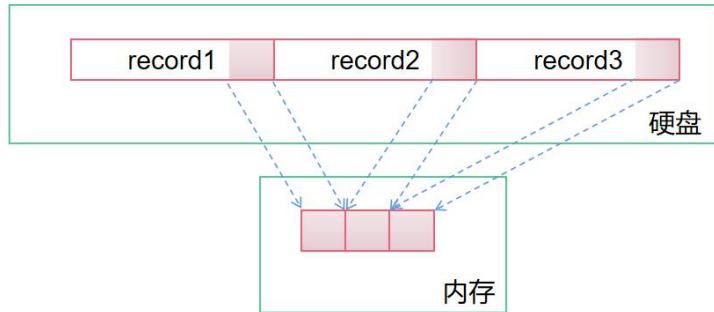


图 3 分析类操作按行存的情况

如果数据是按列存的，也就是在硬盘中，我们先存所有的书名，我们再存所有的价格，再存所有订单的金额。假如我们现在要拿订单的金额，它们是集中存储在一起的，我们就可以一起拿出来。

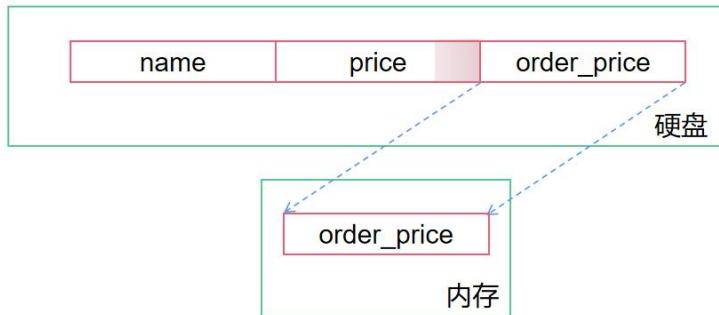


图 4 分析类操作按列存的情况

现在我们就是按列存储的，我们可以想象到一个问题，如果一个数据库有一百万行，肯定是放不下的。我们就可以每 1000 行设置为一个 group，让前面 1000 条的 name 存到一起。这就是分块来存储。

按列存储就支持分析型操作，分析型操作的特点就是有很多 scan（从头到尾扫描某一列），比如统计操作。即使加了时间范围，我们也需要通过 scan 操作。所以我们希望列是紧密存放的。这就是列存。

Q：现在的问题就来了，数据库既有 A(Analytical)又有 T(Transaction)的情况该怎么办呢？

A1: 一个比较牺牲性能的方法就是在数据库中同时存储这两个数据。带来的缺点就是空间翻倍了，我们修改数据的时候要同时修改两个地方，带来了一致性的问题，但好处就是无论是 A 操作还是 T 操作都能很好的支持。

A2: 还有一个办法就是数据库中数据的存储方式是可变的，当事务型操作多了以后就转化成按行存储，而当分析型操作多了以后就转化为按列存储。我们可以通过机器学习做一个预测，去统计这些数据被访问的模式，这样它就知道这些数据大概是被 A 处理得多还是被 T 处理得多。

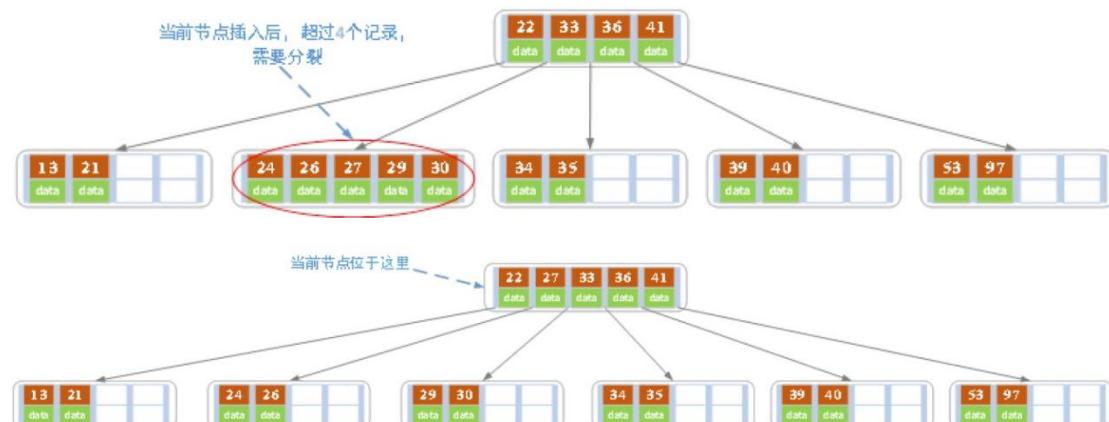
日志结构合并树结构的数据库和之前的 MongoDB 等，基本上都属于 key-value 存储的一类。存储的可以是嵌套的 json 等等。key-value 存储里面就是增删改查，提供 restful 的接口，还支持 scan 操作，可能就是查询表中的一个 key 的范围。一般来说，MySQL 里的存储用的都是 B 和 B+树。然后我们再看一下 LSTM 树。

B 树

B 树实际上就是 N 叉树，如果是 2 叉的，就是一个二叉搜索树。如果是多叉的，那么就用 m 表示它的阶数，也就是每一个节点最多放 $m-1$ 个元素，因为我们要考虑指针的问题。B 树和 B+树的差异就是，B 树里除了叶子节点之外，所有非叶子结点都在存储具体的元素。B 树在操作的时候，有一些约束，比如树不能太高了，约定根节点至少有一个 key，非根节点必须放满一半以上的 key，所以不能随便分类，并且所有的 key 都是排序好的。叶子节点里，每一个节点都包含一个值，它底下不会再分裂了。

在 B 树里做查找其实就是类似于二叉搜索树的方式，因为它是 m 阶的，所以它的深度为 $\log_m N$ ，在每个节点中又有有序的 $m-1$ 个 key，我们在节点内采用二分查找，所以总的查找复杂度为 $\log_2 m \cdot \log_m N$ 。

插入有可能导致 B 树的节点的上溢。



此时根节点要继续分裂（图略）。

B 树的删除也会导致节点之间合并的问题。

B+树

所有的 **key-value** 对只在叶子结点存储，而非叶子结点只存 **key** 不存 **value**。并且 B+树的叶子结点有一个指针指向它的兄弟节点。

Q：为什么不用 B 树而用 B+树？

A：索引本身也是一个文件，即使构成一棵树，它也要写到文件里。而索引文件也不是一次性加载进内存的，而是切成很多 4K 的小的 **page**，根据需要去加载的。B 树的缺点就是，每次加载进来的时候，**key-value** 对可能比较少，而 B+树比较多。

因为 B 树加载索引的时候，因为它要同时加载非叶子节点对应的值，所以如果我们要查询的数据在 B 树的相对靠近叶子结点的部分的话，它每次加载进内存的 **key** 的数量就比较少（需要留空间加载 **value**），所以索引效率就比较低，提升了访问内存次数。而 B+树可以一次性加载进来更多的 **key**，可以加速我们的搜索过程，只有最后一次确定 **key** 在什么位置的时候，才加载进 **value**。

所以 B+树相对于 B 树的好处就是搜索时候一次性加载更多的 **key**；拥有指向兄弟链的指针，所以我们范围查询会比较快。B+树的插入和删除也比较麻烦，涉及到节点的分裂和合并。

总的来说，B 树和 B+树都是存储我们的索引的数据结构。

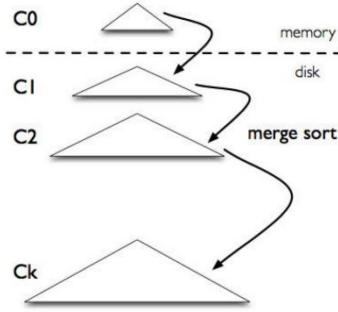
B+ 树

- 优点
 - 快速查找
 - B+：
 - 对 scan (range query, 范围扫描顺序访问) 更加友好
 - 索引节点内单个页存储元素个数可以更多，降低树的深度，尤其适合 value 较大的场景
 - B：查找更加快（不需要每次访问到子节点）
 - value 热更新开销小
- 缺点
 - 插入操作慢
 - 空间放大率高（空间利用率不高），数据结构变更容易产生文件空洞
 - 不同节点随机存储在磁盘上，会产生大量的随机 IO

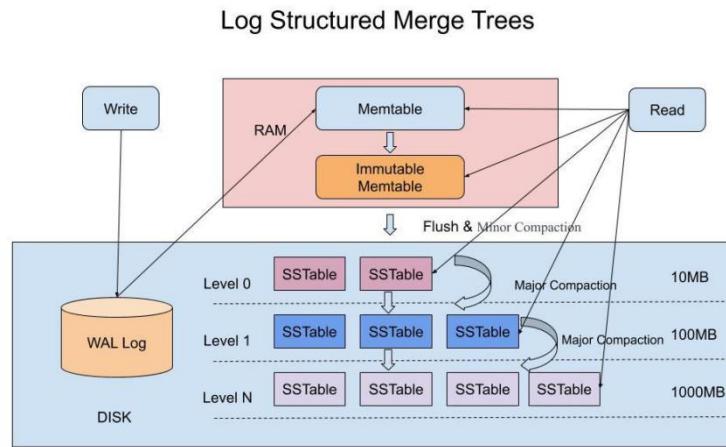
空间放大率是什么意思呢？也就是在 B+ 树中，因为每个节点只要存一半以上的 **key** 就满足要求了，所以存在一些空洞的位置占据空间，包括说因为 B+ 树的叶子数据维护了全量的 **key** 的数据，所以非叶子节点上存储的 **key** 等于额外存储的空间。所以存储时会浪费掉一点空间，这就是空间放大率。它占据的内存才是最致命的。

日志结构合并树（LSTM Tree）

所以我们就提出了日志结构合并树这种方式。它就是一个分层结构，每一层是上面层的若干倍（比如 10 倍）。它在内存里有一层，写入的数据都在内存里，内存满了以后往硬盘上落。硬盘上的第一层和底下的层会有一点差异。



所有的数据都是以追加的形式，没有删除和修改，好处就是它数据的写入能力非常强，但是它数据读取就比较麻烦。



上面是在内存里，客户端要写数据的时候，先写一个 **write-ahead log** 到硬盘上，只要写成功了，再把数据写入到内存里。一旦内存写满了，它上个锁就会往硬盘上落，打开另一块空内存。写到硬盘上时，要做一个 **minor compaction**（较小合并），表示直接把这一层落到硬盘上，写成一个 **SSTable**，**SSTable** 就是一个排序的字符串表。

在 **Level0** 这一层，就有 2 个 **SSTable**，它们的 **key** 的范围是允许重叠的，所以在这一层查找的时候就必须把两个 **SSTable** 都扫描一遍。我们可以设计一个布隆过滤器，我们可以很快地判断数据不在 **SSTable** 里。

minor compaction 没有做 **key** 的合并，所以 **key** 的范围会有重叠，不过硬盘的其余层都是 **major compaction**，所以都要做 **key** 的排序和切分，保证 **key** 的范围不重叠。不过正因为要对 **key** 做多路归并排序，所以整体上的速度比起 **minor** 的直接落硬盘肯定是要慢很多的。

此时如果有读操作的话，如果能在内存中命中，那么就可以快速返回，而内存中的数据就是我们最近写入的数据。当内存 **miss** 的时候，查询操作才会从硬盘中一层层查询。数据一层层往下落就代表着数据越来越老。所以在读旧数据的时候，效率就比较低，而读热数据的时候，效率就比较高。

所以日志结构合并树为什么叫合并树就是这个道理，底下的每一层的 **SSTable** 一直在做合并。它的性能就会有比较大的提升，但是它也有一些问题。

LSTM 的读放大和写放大

LSTM 的数据是加密排列的，所以空间放大率并不高。但是它有时间放大率，我们读一

个数据，它先读内存，再读一层层硬盘。只有把最后一层都扫描完以后才能返回“找不到数据”。

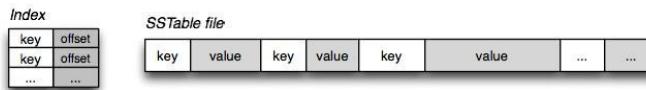
所以 LSTM 擅长访问热数据（新数据），数据的热度分布就和数据的深度有关系。它的写也有放大率，我们表在落硬盘的时候，假如 Level0 满了，那么我们要等 major compaction 到 L1。极端情况下下面每一层都是满的，那么我们就要做多次 major compaction。compaction 腾出空间这个事情全部发生在硬盘上，所以速度是相当慢的。

优点

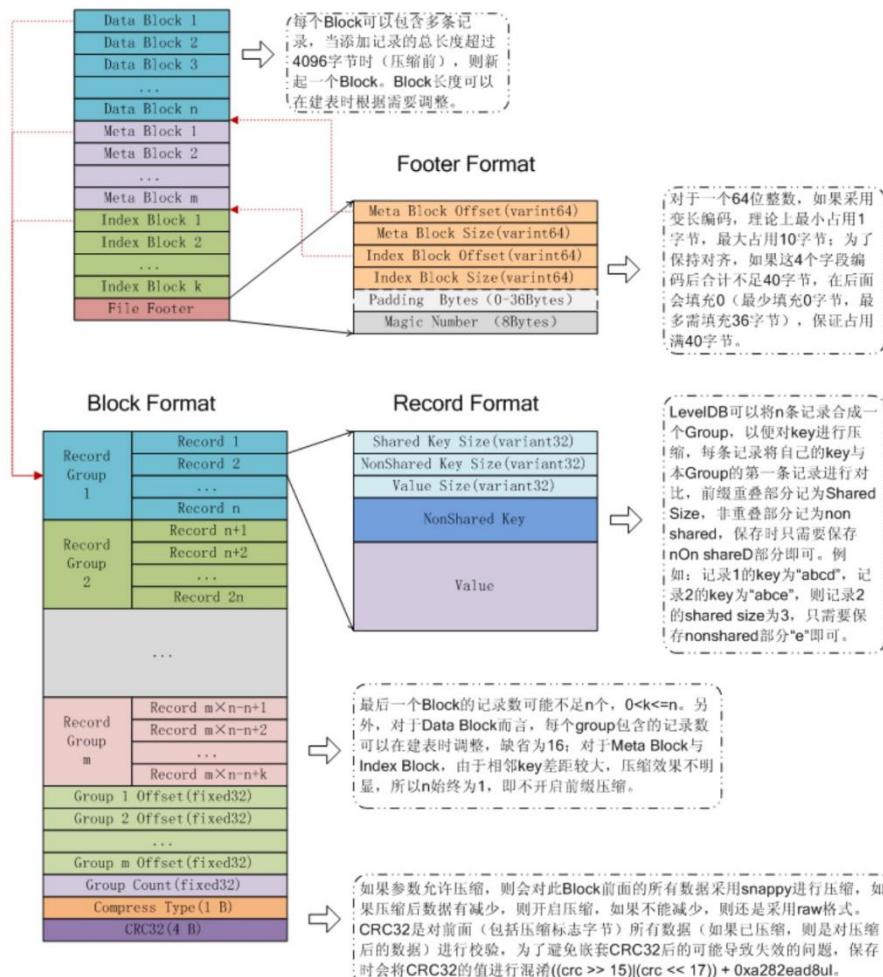
- 大幅度提高插入（修改、删除）性能
- 空间放大率降低
- 访问新数据更快，适合时序、实时存储
- 数据热度分布和level相关

缺点

- 牺牲了读性能（一次可能访问多个层）
- 读、写放大率提升



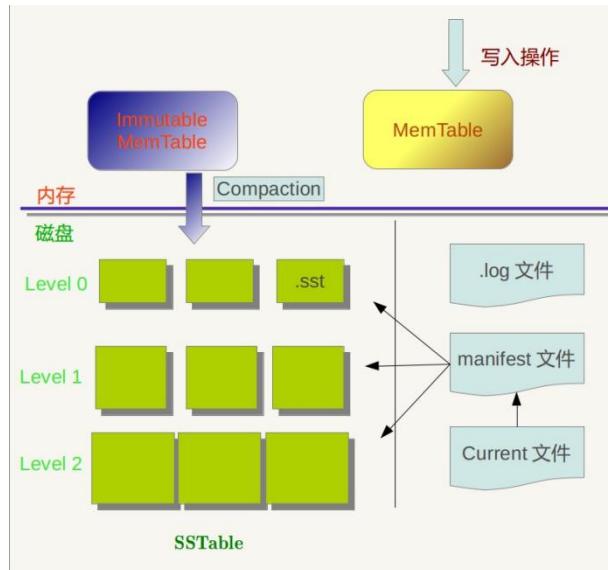
SSTable 的好处就是完全紧密排列，不需要设定字段多长，所以我们需要一个索引来记录偏移量，随机的读取效率非常高，我们可以通过偏移量直接寻址。缺点就是我们要修改 value 很麻烦，我们不能全部调整这张表，我们不能读出来以后再写回去。



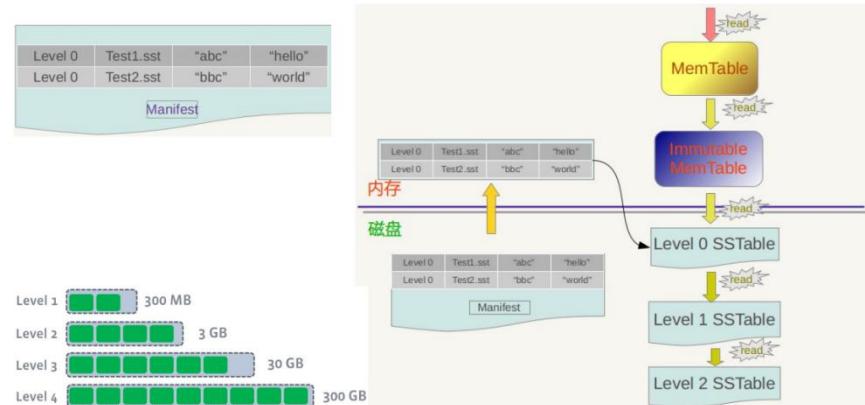
分为索引块、元数据块（有没有做压缩，压缩算法是什么）、数据块。一个数据块打开来看，它为了节约空间，把数据分了一下组，对数据找出公共前缀等。从形式上看，块是连续存储的，**SSTable** 就是以这种方式在存储。

LevelDB

我们应用之前学的这些，来开发数据库。



它其实就是内存写满了落到硬盘上。要注意它有一个 **manifest** 文件，是 **SSTable** 的索引，它记住了每一个 **SSTable** 中 key 的范围是多大。



存储分为数据索引块、元数据索引块、元数据块和数据块。每一块算完之后还会计算一

个循环校验码和压缩类型。

Record i	key共享长度	key非共享长度	value长度	key非共享内容	value内容
Record i+1	key共享长度	key非共享长度	value长度	key非共享内容	value内容

共享的 key 只存一次。SSTable 里的表在存的时候，索引的默认存储方法是跳表。跳表保证了上面的数据块的查询速度，跳表里存储的一定是 key 的非共享长度，是每个 group 里做一个跳表。而 group 本身是通过共享 key 的方式去存储。所以即使是 SSTable，里面的数据结构也很复杂。

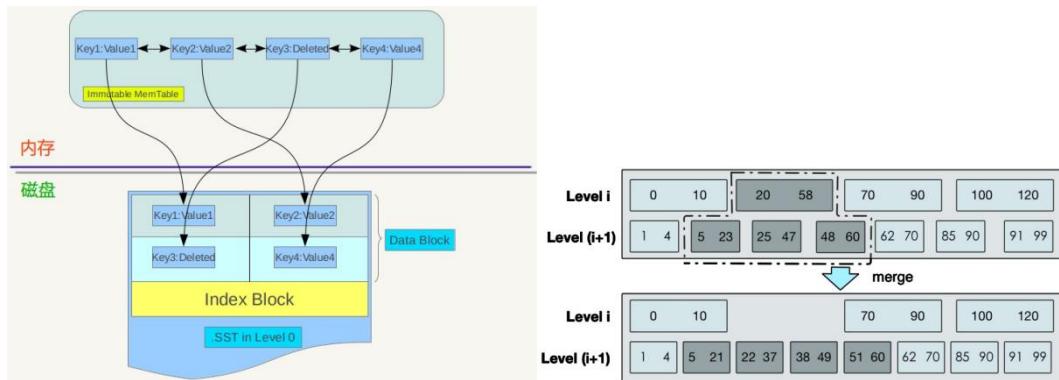


图 5 内存和硬盘中的 Compaction 的图例

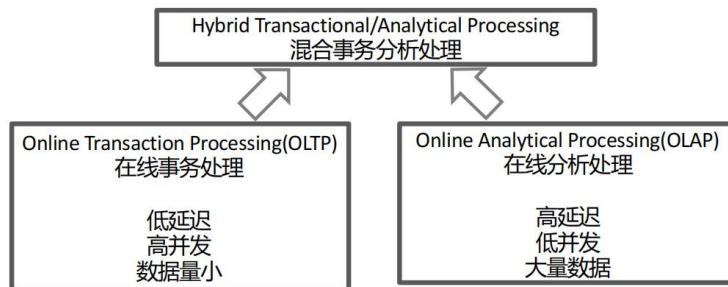
RocksDB

RocksDB 是基于 LevelDB 去开发的。

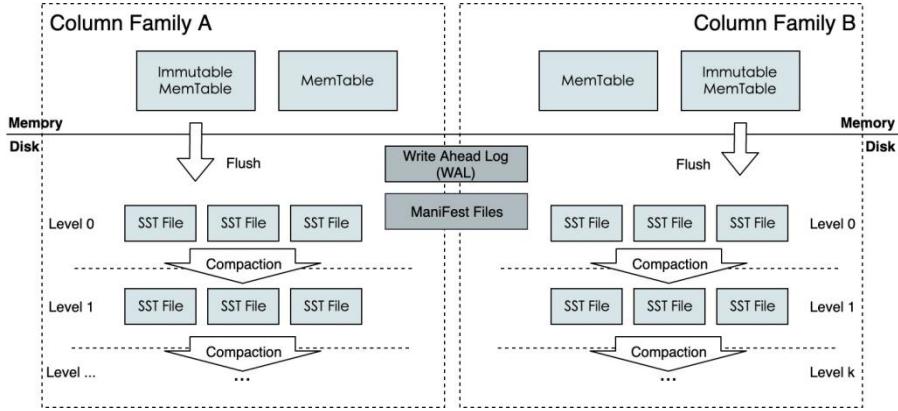
把 RocksDB 的按行存改成按列存的缺点：之前提到 LevelDB 中的每个 block 都会记录一个循环校验码来判断是否被破坏，如果是按行存，那么就是每若干行算一个 CRC。如果我们要按列存，CRC 到底怎么实现就成为一个问题。包括按列存要压缩空间，我们可以第一个值存储元数据，后面存列的偏移量。

RocksDB 对于热数据支持比较好。

混合存储

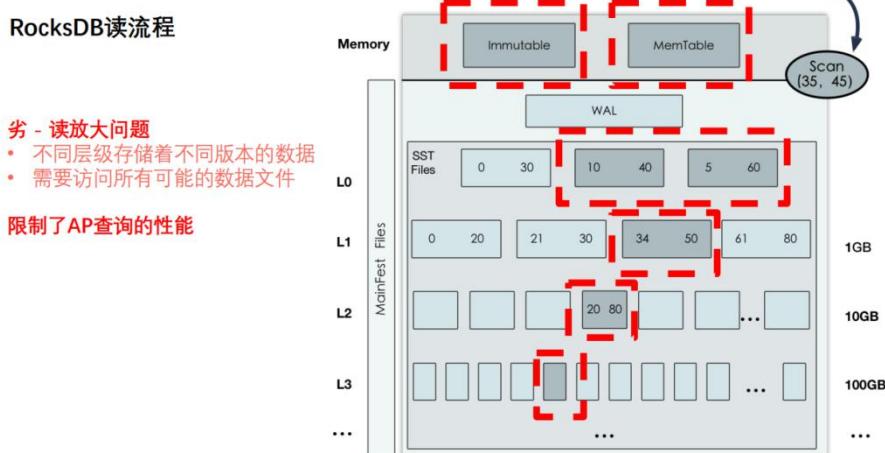
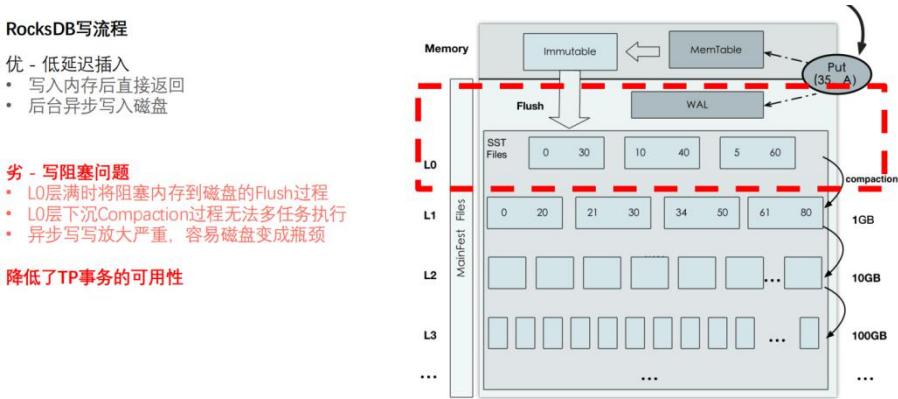


OLTP 和 OLAP 有不同的应用场景。统计几分钟是完全可以接受的，但是下单就要在几秒钟里支持。就要用到目录结构树这种结构。



RocksDB 的表里可以分成若干个列族，逻辑上还是一张表。分成列族了以后，可以分别操作。这种方式最大的问题就是，有可能会有写的阻塞问题。

当我们一张表在往下落的时候，如果有请求过来的话，可能读到错误的结果，所以要做加锁的动作，一直等到落完了再解锁。包括读的时候可能也要读到最底层。



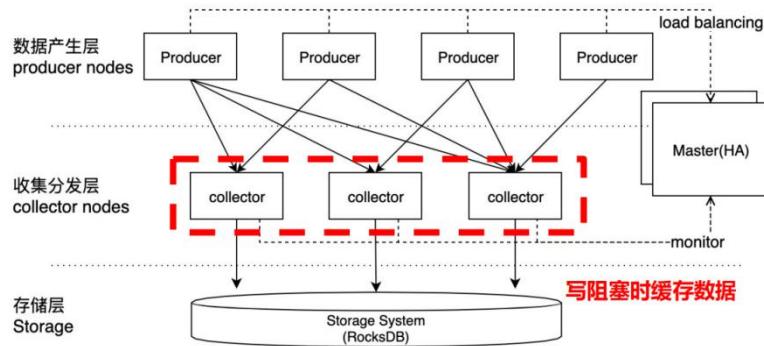
为了让它支持混合存储，我们就要做一个存储结构上的修改，比如我们要做行列混合存储，要么存两份，要么做行列差异存储（有些块是行存，有些块是列存，要基于查询来动态修改数据存储方式）。

解决写阻塞问题

具体来做的话，面对写阻塞问题，我们可以先把大量的请求收下来，我们可以使用 Flume

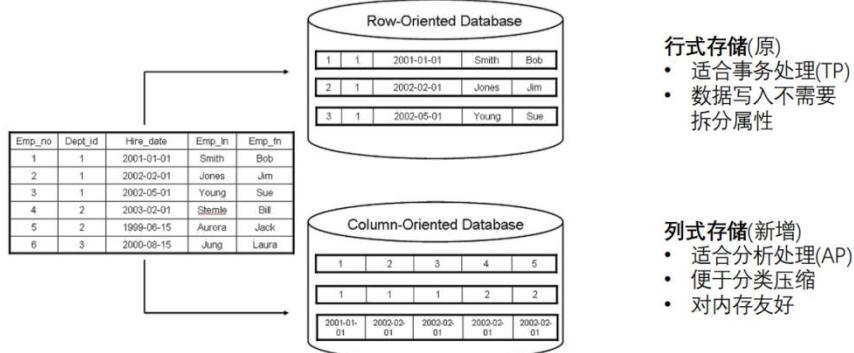
来接收大量的数据。

在RocksDB和数据源之间增加一个收集分发层，集群结构如下



在RocksDB中增加列式存储

列式存储在访问少量列时磁盘读取量更小，可以减少读放大的开销



上面应用于 TP 处理，下面应用于 AP 处理。列存就涉及到对行分组，然后一组以内的列存放在一起。

解决读放大问题

读放大问题就是说我们查询可能要沉到底下再返回查询结果。可能就需要增加列存储的布隆过滤器。这样就不用读取每块的完整内容。

2021/11/18

时间序列数据库

今天讲讲时间序列数据库。**InfluxDB** 带一个 web 控制台，它现在在监控笔记本上的 16 个 CPU。笔记本休眠的时候会自动插值。所以这里面一个是 InfluxDB 和采集数据的工具在跑。时间序列数据库里面也有和日志结构合并树类似的机制。

首先来说什么是时间序列数据库，也就是每个数据上带一个时间戳。比如北交所开业第四天，我们想抓取股票的交易信息、或者互联网上传感器抓到的数据，我们都要存下来。

Q: 如果拿关系型数据库来存储时间序列数据会怎么样呢？

A：如果我们使用关系数据库来存，那么在一张表里面，我们一定有 `id:time_stamp:stock_id:price:amount`。这里面有一个问题，如果我们要存从 `19700101` 到目前时间的表述，相对于真正有用的数据，时间戳可能占据了太多存储空间。并且因为时间戳可以相同，所以不能作为 `id`，需要复合索引或者使用自带的 `id` 列。

从存储的角度来说，拿关系型数据库来存，存储效率没那么高；从访问的角度来说，关系型数据库表中不断增加，数量就会非常大，无论我们是在股票的价格还是 `id` 上做索引，就会很慢。

如果像我们上节课讲的日志结构合并树(LSTM)的话，它的存储结构是逐层递增、日志的年龄也是逐层增加的。新写入的热数据访问速度就比较快，这就是时间序列数据库想要的。

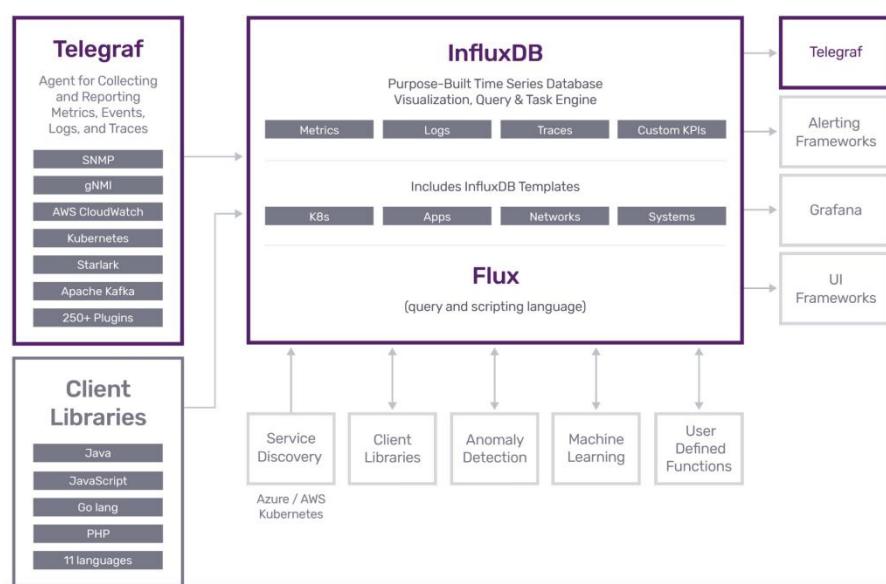
更重要的一点是，时序数据有一个 **retention policy**，也就是在库中的存活时间，比如说我们只保留小于三天的数据。在关系型数据库中没有这种概念。

总结一下：时序数据是有生命周期的，超过时间了要删除掉，在读数据的时候，希望新写入的数据读取速度较快。所以关系型数据库不太适合，就出现了时序数据库。

比如物联网的传感器不断地给你发数据，本身这些数据又比较简单，没什么复杂的关系，又比如监控电脑 CPU、内存，这些数据可能不是来自于多个数据源的。时序数据库用的比较多的就是 `influxDB` 和 `Prometheus`。还要一个就是 `OpenTSDB`。

influxDB

`influxDB` 就是一个时序数据库。它也提供一个云平台，我们可以把数据传上去。要应对数据量大和数据来源多的问题，要解决的一个就是系统监控。系统监控的工具可以有很多种，`influxDB` 提供的是 `Telegraf`，就是一个抓取系统参数的工具。所以我们在运行的时候，两个都要启动，然后告诉它要去监控一些什么样的指标。`Telegraf` 就会把数据写入到 `influxDB` 中，然后 `influxDB` 就可以通过一些查询语言来呈现结果。



`influxDB` 内部结构如上，它就没有关系型数据库中 `table` 的概念。在 `influxDB` 中，有很多的 `bucket`。

bucket: my_bucket

_time	_measurement	location	scientist	_field	_value
2019-08-18T00:00:00Z	census	klamath	anderson	bees	23
2019-08-18T00:00:00Z	census	portland	mullen	ants	30
2019-08-18T00:06:00Z	census	klamath	anderson	bees	28
2019-08-18T00:06:00Z	census	portland	mullen	ants	32

timestamps measurement Tag value Tag value Field key Filed value

我们要新建一个 bucket 并且设置 retention policy 来设置数据的删除时间。建立好了以后，我们就可以配置它的数据源，可以设置 telegraf 要监控电脑、docker 等。bucket 就相当于我们的库，要设置它的 organization。bucket 里第一列就是时间戳，可以精确到小数点后很多位，因为第二行开始存的都是时间的差值，占据空间就比较小。第二列是_measurement，我们可以认为它是一个表，不同的 measurement 是存在一起的。后面的四列分成 Tag (location、scientist) 和 field (分为域的 key 和域的 value)。完整的一行就是一个 data point。

在 influxDB 里，如果我们要去搜索 bees = 23，field 上是不建立索引的。而 Tag 上是建立索引的，查找速度就会变快。所以我们在设计 bucket 结构的时候，就要注意这个细节。

measurement 实际上就是把 bucket 中的数据分组以后描述一下这一组的数据。

Field set

- A field set is a collection of field key-value pairs associated with a timestamp. The sample data includes the following field sets:

```
census bees=23i,ants=30i 15660864000000000000  
census bees=28i,ants=32i 15660867600000000000
```

Field set

Field set 就是按照时间维度把数据组织起来，也就是一个时间点上采集到的所有 key-value 对的集合。

Q: 为什么 field 上不能建立索引呢？

A: 和它数据存储方式相关。field 使用的是列压缩方式（存差值），就很难建立索引。所以 field 如果经常被查询，那么我们最好把它转化为一个 tag。

在处理的时候，influxDB 提供了一个 flux 查询脚本语言，在这个库里我们需要用 flux 去做搜索。

```
from(bucket: "bucket-name")  
|> range(start: 2019-08-17T00:00:00Z, stop: 2019-08-19T00:00:00Z)  
|> filter(fn: (r) => r._field == "bees" and r._value == 23)
```

对于每个 record，搜索 field 中 bees = 23 的情况，这个查询就在 field 上做过滤，就需要遍历整张表。field 和 tag 相互转换有点像关系型数据库中按行存和按列存的转换。

Bucket 也有元数据，定义在 schema 里。

时序数据就是按照时间顺序来的一系列数据，我们想要唯一描述一个数据点，按照_time 和_measurement 是不够的，需要_measurement + tagset + _field 来唯一标识。

Series

- A **series key** is a collection of points that share a measurement, tag set, and field key. For example the [sample data](#) includes two unique series keys:

_measurement	tag set	_field
census	location=klamath,scientist=anderson	bees
census	location=portland,scientist=mullen	ants

- A **series** includes timestamps and field values for a given series key. From the sample data, here's a **series key** and the corresponding **series**:

```
# series key
census,location=klamath,scientist=anderson bees

# series
2019-08-18T00:00:00Z 23
2019-08-18T00:06:00Z 28
```

整个 influxDB 的设计来说，数据是按照时间升序存放的；这个数据基本上也是写入之后不改写的，要严格限制更新和删除数据的权限；一旦执行查询，所有其他操作（写操作）暂停，保证我们查到的数据和现在数据库里的数据是一样的。

Q: 这些数据在往里写的时候，如果一个人在两个数据点上发送了两次观测，得到了两个完全相同的值进来了，该怎么办呢？

A: influxDB 认为这种情况不应该发生，完全相同的数据不会存两次，如果同一个时间戳里数据不一样的话，它会使用覆盖的方法。

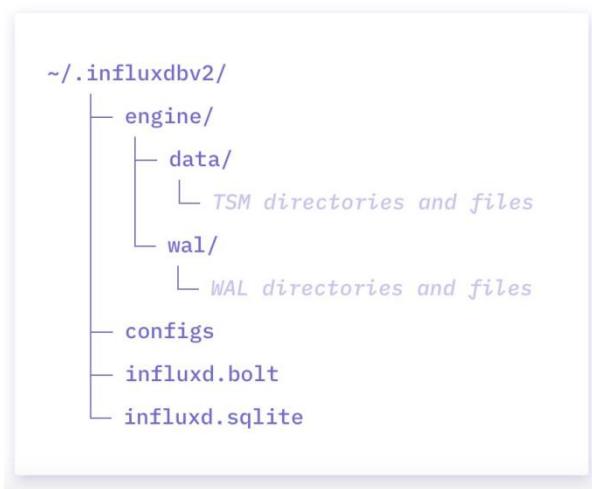
它的引擎里做了一些工作：

1. WAL, write-ahead log，操作先写到 WAL 里，然后再写 cache，cache 满了以后 flush 到硬盘上，这叫做时间结构合并树（TSM）。它会对时间序列做索引（mesurement+tag sets + field）。
2. 写数据的时候，使用的是 http 协议，通过 post 方式写数据。
3. 它的合并的时候有可能出现写放大，一直要落到最后一层。
4. 这个文件日积月累越来越多，数据从 level 0 往下落的时候应该压缩一下。

数据写的时候，一种方式是来一个数据处理一下写进去，但是这种效率比较低，最好把数据先组织在一起，按照 batch 的方式写入 influxDB。最好前面再放一个 flume 或者 telegraf，都实现了 batch 写入的操作。往硬盘中写数据使用的是 fsync() 系统函数，开销大一点，但是比较可靠。一旦写入到硬盘以后，内存中的内容就被清空了。

在缓存里的数据是不压缩的，要保证它的查询效率。压缩是发生在老数据做 compaction 的时候。

macOS file system overview



BoltDB 就是系统库，存放了和时间序列数据无关的东西，比如权限管理等。

时序数据一定会非常多，因为有源源不断的监控的数据过来。这个数据大到一定程度以后，在单机上就存不下了，所以它要分成 shard、切成小块。它可以把 bucket 拆开，把压缩过的时间序列数据切成很多不同的小块。而 `_measurement` 是数据语义上的，而 shard 是硬盘上的。shard 切开之后，我们可以对每个 shard 设置不同的存活时间。

有了 shard 以后，我们的数据就可以存在不同的 shard 里，默认情况下是不分 shard 的。

Bucket retention period	Default shard group duration
less than 2 days	1h
between 2 days and 6 months	1d
greater than 6 months	7d

如果我们这么分，系统中就会有 3 个 shard。duration 就是说 shard 的最大容量是几天的数据。我们也可以预先创建好 shard 的空间，并且不能对过去数据做 shard。我们还要对 shard 区分成新的（数据不压缩）和旧的（数据需要压缩）。

在 TSM 中，和 LSM 不太一样，因为它的数据都是按照时间升序排序的。

运行 influxDB，略。见 18-timeseries db P35~P4。

2021/11/22

在 canvas 上放了数据库课程方案包 V 1.1.0.zip 是华为课程中的 ppt 和实验指导书。GaussDB 是人家在线的数据库 OpenGauss 是可以在本地跑的。可以找到一个 docker 的 image。

<https://blog.opengauss.org/zh/>

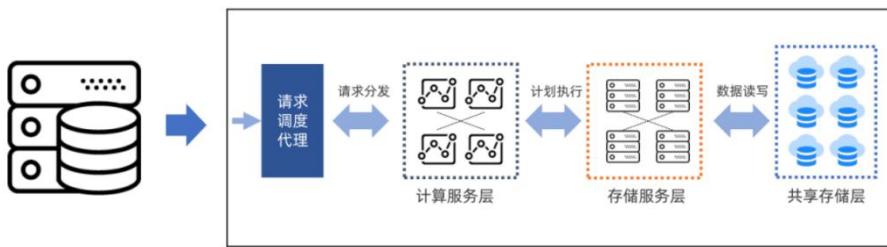
云原生数据库

云原生的数据库：本身就在云上，做了一些优化。自己的机房配置出一个集群和云原生有什么区别呢？阿里做了很多硬件的优化，所以实际上我们是没有办法自己搭一个 polarDB 的，因为有专用硬件的软硬件一体化。云原生的数据库未必是关系型数据库，GaussDB 包含各种各样的 MongoDB、Redis。

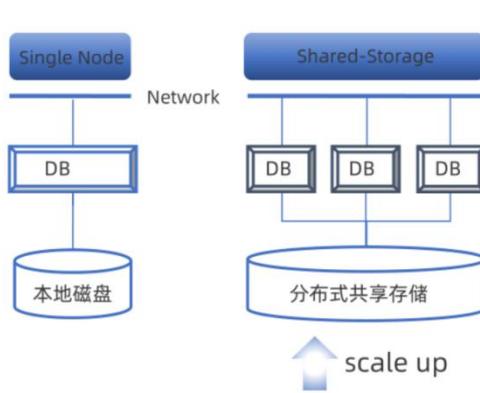
数据库在云上，所有的资源都是虚拟化的，如果拿大家的笔记本搭成一个云，大家的硬件是不同的，是异构的。但是对外暴露出的云的接口应该是统一的。所以虚拟化要做资源的池化，目的就是提高资源的利用率。在云上做存储就要考虑这些问题。

资源耦合：每台机器都只能访问自己的 CPU 和硬盘，在云上我们希望一台机器上的 CPU 访问另一台机器上的内存和存储。我们把 CPU、内存、硬盘弄成一个池，这样万一有一个坏了，我们可以在池中立马找到一个好的来使用，整个计算过程是不停止的。

弹性服务：数据存储和处理变得即插即用。



一百万个请求过来，按照某种逻辑调度请求，每一个节点要生成一个请求的执行计划。数据库在做存储的时候也有两种情况：把数据串起来，做成一个分布式文件系统，变成一个共享内存，大家全部往分布式文件系统去写；还有一种情况是，数据应该分开，因为大家通过共享存储来写的时候，硬盘的 IO 速度可能成为最终的瓶颈。



传统如：

- Oracle 及其 RAC 架构
- Sybase、SQL Server

优势：

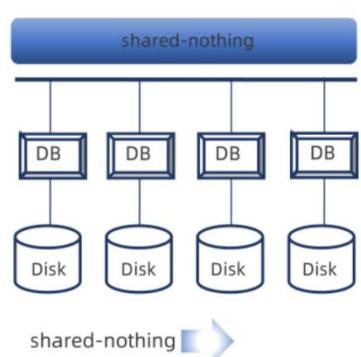
- 易于实现事务一致性
- 无需多层复杂管理

劣势：

- DB 节点扩展能力受限
- 存储扩展能力及 IO 性能依赖高端共享存储

服务器如果 crash 了，就可以去接管它，好处就是在做事务处理的时候比较容易设置隔离级别，但最终系统的瓶颈就卡在了共享存储的 IO 上。

所以现在的云数据库采用的方案是 share nothing，要交互就通过 API 进行。



代表如各类 数据分片、MPP分析数据库：

- Oracle 12c Sharding
- 阿里云 POLARDB分布式版
POLARDB-X: DRDS + POLARDB
- 阿里云 AnalyticDB (ADB)
- MyCat / TiDB / OB
- Teradata
- Sybase IQ

优势：

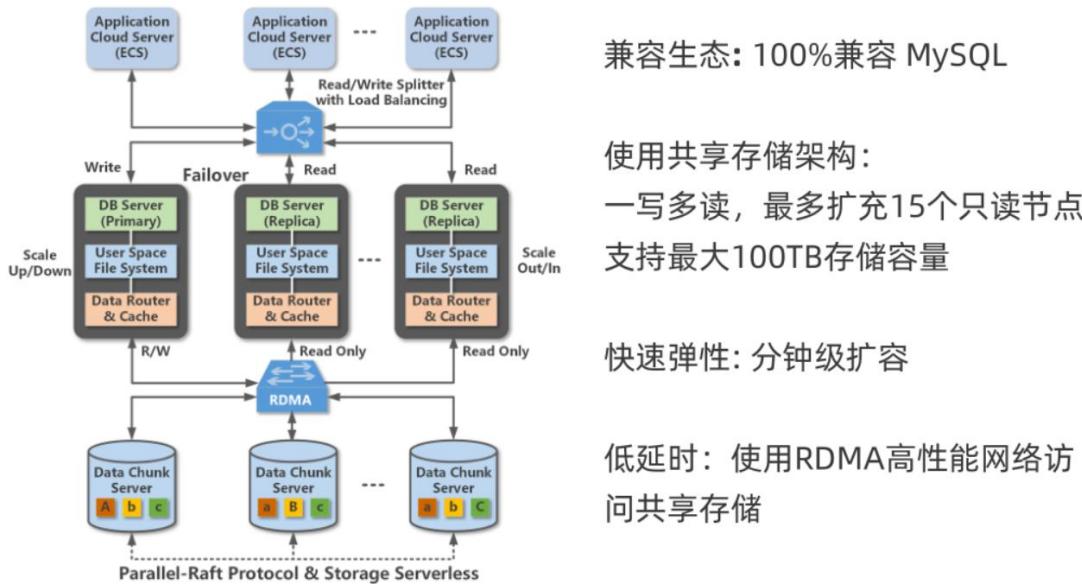
- 良好的可水平扩展能力
- 数据多副本存储，无需共享存储

劣势：

- 计算及存储能力需同时扩展弹性不足
- 分布式查询、分布式事务处理的开销

如果我们要做一个分布式查询，查询一整张表，此时如果分布在多台机器中，那么涉及到多个节点之间的交互的时候开销就比较大。

我们发现概念是相通的，我们之前讲到微服务的时候也是类似的设计思路。数据库的实例之间我们认为要隔离开。



兼容生态：100%兼容 MySQL

使用共享存储架构：

一写多读，最多扩充15个只读节点
支持最大100TB存储容量

快速弹性：分钟级扩容

低延时：使用RDMA高性能网络访问共享存储

应用就会来访问到云数据库，黑色框里的是数据库管理系统，而底下是数据库存储服务器。因为它要追求速度，在里面出现了 RDMA 的路由器。

现在我们应该 DB 的 server A 要和另一个 DB 的 server B 通信，原来如果它们是共享存储，那么很容易就访问到数据了。但是现在是分布式隔离存储，如果我们走最简单的 TCP/IP 网络协议，这种数据的组装过程非常麻烦，会消耗掉很多 CPU 的资源。慢的原因是，被请求的数据被 MySQL 读取之后，这块数据是在用户态下，而 TCP/IP 这两层东西是在内核态的。需要把数据交给内核态（这一步很慢）才能开始组装 TCP/IP 数据，处理完之后，才能发给网卡去处理。

如果我们可以把数据从用户空间直接交给网卡，这样我们就可以把数据从用户空间直接交给网卡，处理完再送出去。

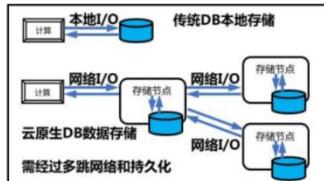
在 server B 这里的时候，server B 内存中的数据->server B 上支持 RDMA 的网卡->server A

上支持 RDMA 的网卡->server A 内存中的数据。

云原生架构下DB数据存取路径长，导致写入和查询延迟大，高速存储潜能难以发挥。

延迟优化是I/O性能优化中的难点问题

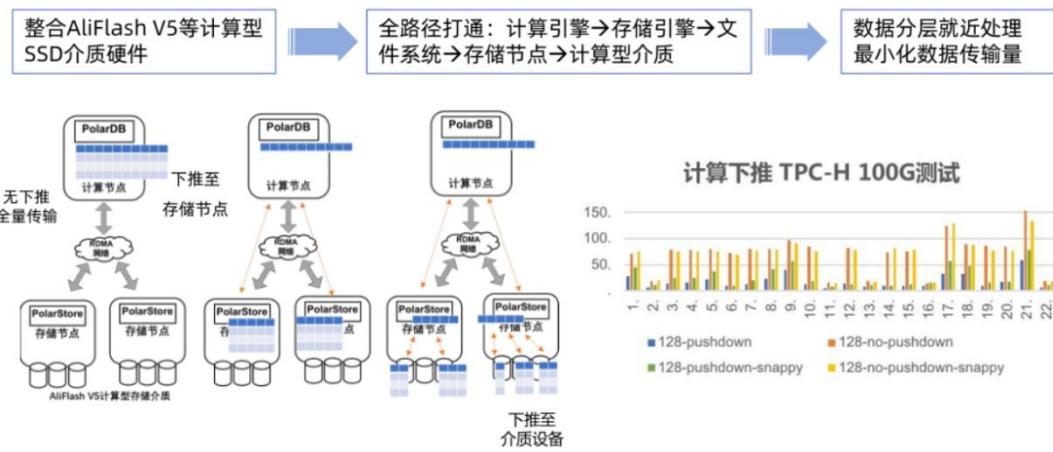
- 1) 路径长：存储侧需完成跨节点多副本一致性写入，I/O需跨多跳网络及多点持久化。
- 2) I/O栈开销重：传统Linux内核I/O栈已不适用于10微秒延迟量级的高速介质。



近年来网络、存储、计算方面新型软硬件发展迅速，DB性能优化需要充分发掘软硬件红利



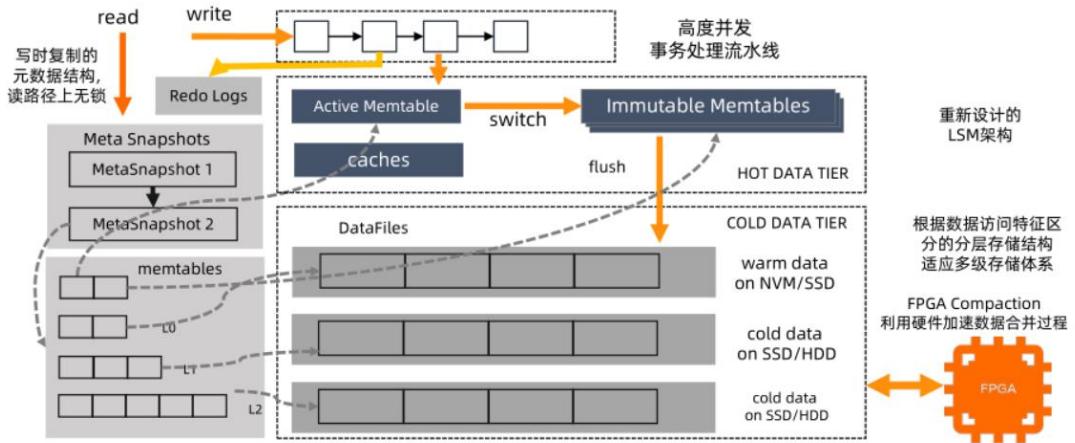
还有一个很重要的东西就是它把计算下推了。



云数据库分为计算节点和存储节点。计算节点：当有一个请求过来的时候，要去计算出来 join/select 的结果。如果我们的计算全部在计算节点上做，那么我们没有办法利用存储节点的部分计算能力，因为我们要在计算节点上去做计算，存储节点会把大量的数据发到计算结点，然后再让计算节点去做筛选，这样内部的带宽占用就比较多。我们可以把计算下推，利用起存储节点上的计算能力。把存储节点计算完的统计结果再向上传递。

这个没有标准化的实现，并且依赖于数据库本身的逻辑和 RDMA 这个网络。

识别冷热数据的分层存储引擎 X-Engine

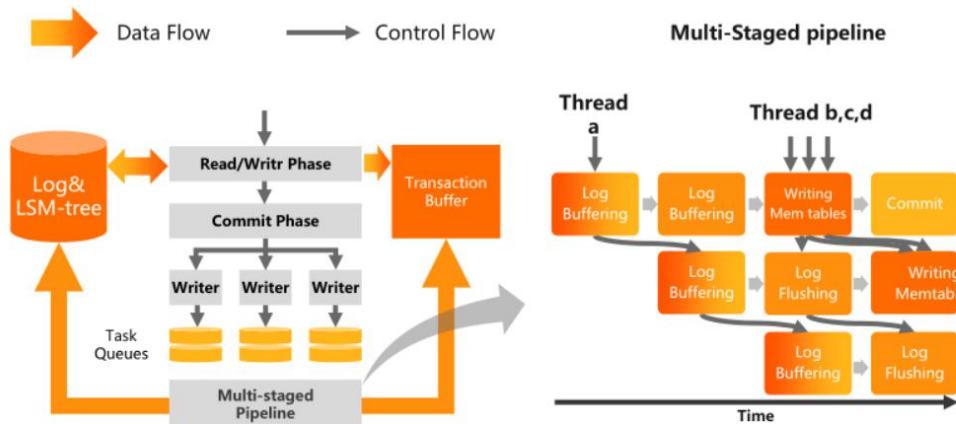


SIGMOD'2019: <<X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing>>

FAST'2020: << FPGA-Accelerated Compactions for LSM-based Key-Value Store>>

要用日志结构的存储，要通过 FPGA 来加速 compaction 操作。FPGA 在执行的时候，不需要 CPU 参与，并且支持并行操作。一谈到冷热数据，它必然是一个分层的结构。数据先写到内存里，然后往下落，写的时候先写日志。所有的数据库只要想支持冷热数据的区别对待，基本上都是使用了这个原理。

事务处理流水线技术



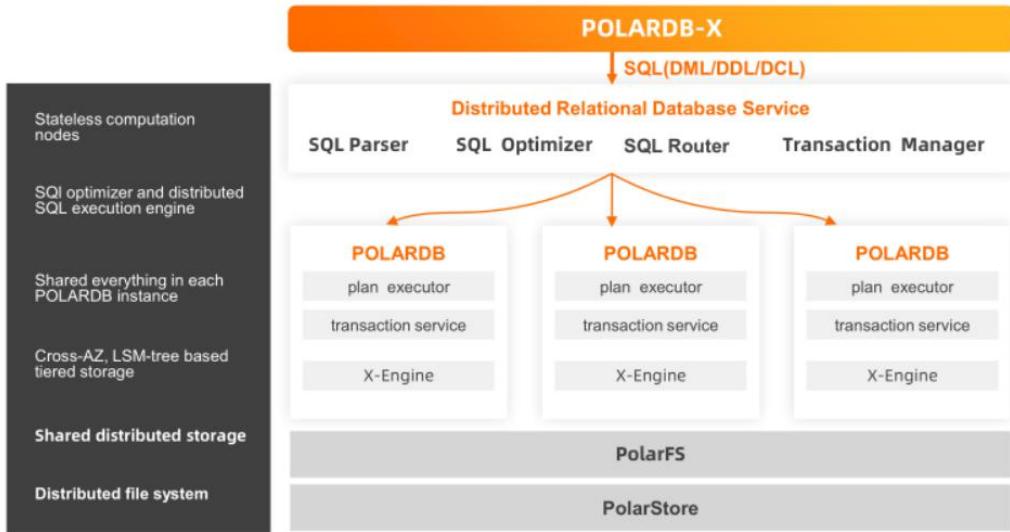
- Asynchronously buffering changes in transactions first
- Tuning thread-level parallelism for disk I/Os and memory writes

事务应该先写到 buffer 中，然后 flush 到硬盘上。右图是执行一条事务的流程，多个事务同时出现的时候，就变成了流水线去做。一个事务在完成某一个环节的时候，第二个事务

就可以继续在这个环节中完成自己的操作。在流水线操作的时候，还不能破坏一致性的逻辑。

在这一套机制上，无论是横向扩展（加节点）、纵向扩展（加 CPU 和内存）都会比较容易，它会使用高可用性的框架去监控它们。

- ✓ 同时支持线性扩展和存储计算分离弹性能力
- ✓ 通过代价计算判定请求类型，对不同类型查询使用物理隔离的资源进行分流
- 同时支持TP与AP的混合负载



华为 GaussDB

华为 Gauss 数据库里面分的很细，有 for MySQL, for MongoDB, for Redis 等。华为有自己的硬件，分为存储型、计算密集型的服务器。其中和 MySQL 兼容的 OpenGauss 开源出来了，可以跑在本地环境中。华为有分布式的数据库应用，对外就是以 OpenGauss 的行为。

数据库的系统用的也是和刚才类似的，每一个节点也是 share nothing 架构，和刚才的 polarDB 没什么区别。但是它有一个数据仓库的概念。

数据仓库

比如说我们的电子书店有点大了，我们开了一个公司，首先卖书的网站肯定是有，我们还需要客户关系管理系统，比如 HR, OA，采购都有一个系统。现在这些子系统都带自己的数据库，双十一过后我们想问一下哪本书最好卖。我们需要知道书是什么（书籍数据库）、订单数据（order 数据），然后去客户关系系统中抓取书的出版商的信息。真正在分析的时候，我们对于主题（包含了不同数据库不同表中的很多维度）感兴趣。

我们现在要做的事情是怎么把这 4 个数据库的信息拿过来做一个融合。在 book 系统里面，书的 id 是基于整数递增的，在客户管理系统中里面，可能只存了书籍的名字的 string，不同数据库对同一个实体可能以不同的形式存放，我们拿过来时候需要做一个转换。

同样地，我们在做统计的时候，我们只关心订单数据的统计。

我们想说，不同来源的数据拿过来了以后，我们可能要做很多类型转换、数据转换、语义的转换。

第二件事情是，书进到 eBook 系统以后，有一些书不再出版了。我们就需要做一些清洗，过滤掉不再出版的书。

这两件事情就是数据仓库的第一层，ETL（清洗转换加载）做的事情。

数据仓库的第二层就是我们要对于主题进行建模，这就是 OLAP（在线分析处理），我们要根据需求进行建模产生一些新的表，这些表看起来都是数据库的表。

数据仓库第三层，我们把数据拿出来，这才是前端的一些商业智能 BI 拿到的结果，然后做一些可视化，分析出来。

在数据仓库中，第二层主要是需求驱动的建模，而前面的各种各样的数据库其实是 OLTP（技术角度驱动）。

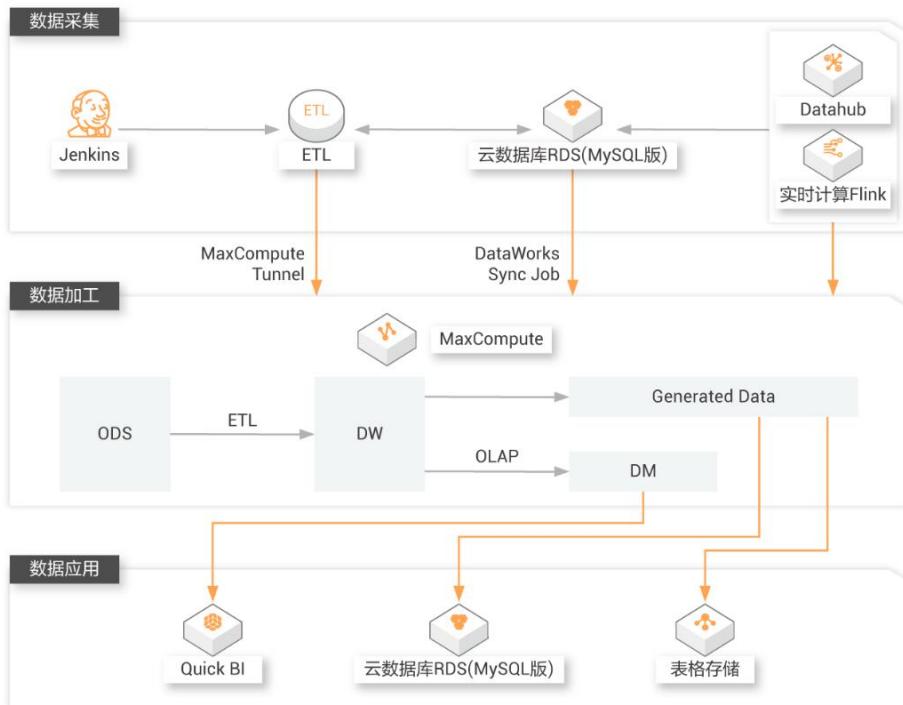
数据仓库有很多种。

Shard Server: 数据存在 3 个节点上。Shard server 是属于 share nothing 的情况，每个 shard server 有自己的存储和内存，要访问其中的数据必须通过 API 调用。GaussDB 利用了很多云的技术来做优化。

MaxCompute 架构



MaxCompute 原来是一个数据仓库，后来说支持数据湖，所以现在说的是湖仓一体，一定有统一的计算机，底下有很多的分布式存储和全局的元数据管理和一个智能调度系统。



数据从 ETL 中进入到数据仓库，也可以通过阿里云的 RDS 的数据系统进去，还可以用 Flink 这个框架。这些数据都可以进入到数据仓库中，然后我们就会看见 MaxCompute 会把原始的数据经过清洗转换进入数据仓库，然后可以通过 OLAP 操作进行数据挖掘，给前端使用。我们也可以产生一些原生的数据导入到其他仓库中。

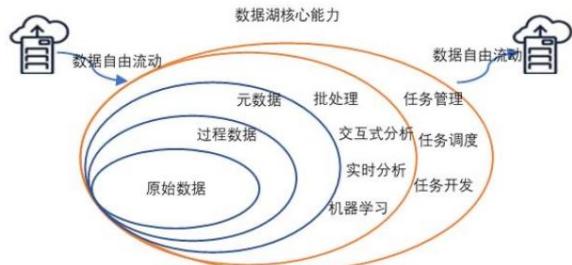
数据湖（Data Lake）

大家数据库用的都不是同一个系统，数据仓库还能搞定吗？数据湖：不管怎么存，数据文件最终总归是某种格式的文件，我们就把数据原来的格式直接存进来。

- Data lake是指使用大型二进制对象或文件这样的自然格式储存数据的系统，通常把所有的企业数据统一存储，既包括源系统中的原始副本，也包括转换后的数据
- Wikipedia

- 是一种不断演进中、可扩展的大数据存储、处理、分析的基础设施

- 以数据为导向，实现任意来源、速度、规模、类型数据的全生命周期管理



元数据管理还是需要做的，说清楚这个文件是从哪个系统来的、属于哪个数据库。当我们去做分析的时候，我们把想要的数据从中抽取出来导入到数据仓库中再去做分析。既然

访问到需要的时候再导入操作，为什么我们不一开始存进数据仓库呢？因为数据湖面向一个企业中大量的数据，但是我们认为不是所有数据进来之后就会被作为待分析的数据了，我们没有必要每个数据一导入进来就做什么 ETL 和主题表生成，所以我们用到了再去做。一旦我们全部用到了，那么我们的数据仓库中就包含了所有的数据，那么原先的数据库还有意义吗？实际上这样子是把前面的数据库的数据都复制了一遍，所以有人提出来数据应该再湖和仓库中来回迁移的，热数据都在数据仓库中而冷数据都在数据湖中。之前的数据在进入到数据湖中的时候，也会做一下分析，是不是数据直接进仓库（交易数据容易被经常分析）还是先进数据湖（被分析频率较小的数据）呢

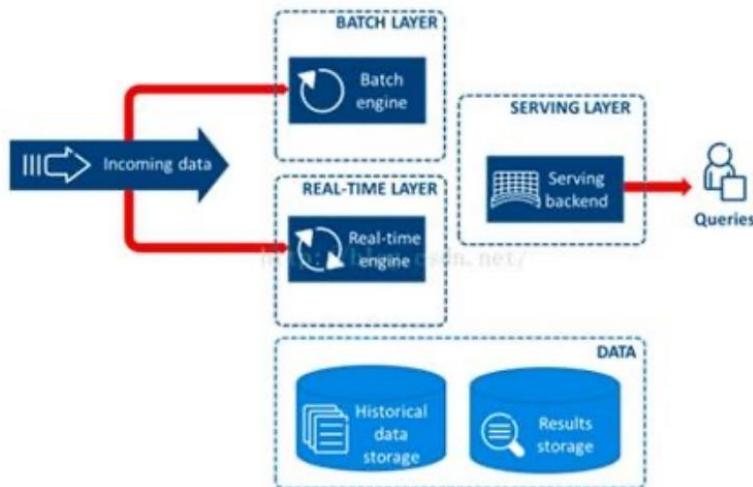
如果只是数据进来以后到湖了，用到了再加载到数据仓库中，那么时间久了数据仓库就等于是数据湖的一个备份了，而且还会导致一致性问题。有人认为数据仓库可以作为数据湖的一个缓存。

Data warehouse	Data lake
数据体系严格，提前建模	数据体系松散，事后建模
灵活性较低	灵活性较高
数据治理容易	数据治理困难
数据种类单一（结构化、半结构化）	数据种类丰富（结构化、半结构化、非结构化）
面向成熟数据的企业级分析与处理	面向异构数据的科学探查与价值挖掘
向特定引擎开放，易获得高度优化	向所有引擎开放，各引擎有限优化

数据仓库本质上还是建立出来的主题表。而数据湖就是数据拿过来原封不动放在这里。因为我们前面的数据建立好了，所以 schema 就比较稳定。数据的 governance 就比较好，数据难道要不断往里导入吗？数据是不是要定时删除？数据治理具体的手段是要自己定制的，包括 ETL 和安全性的审计。

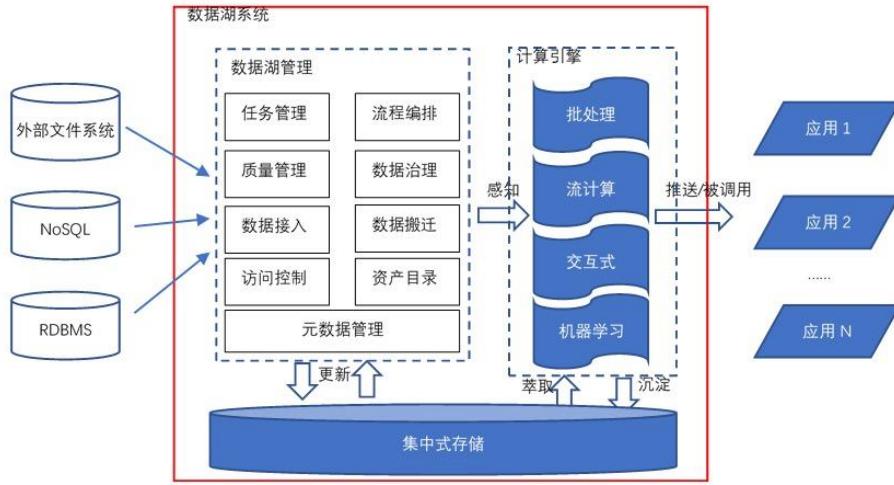
数据湖里底层就是分布式文件系统，而处理都是 map-reduce。

HDFS批处理无法满足实时性要求高的场景，因此产生了lambda架构“流批一体”



数据已经在这了，事后我们再处理。

后来我们调整时间窗口，就可以使用批处理来处理流数据，这时候我们就可以用统一的引擎来处理。数据湖里面，首先数据要做集中存储。



2021/11/25

这节课谈谈集群里的例子，举了两个例子去谈谈会话粘滞性的问题。我们可以通过 nginx 的哈希或者起一个 redis，把会话放到 redis 里共享，这样我们服务就算挂掉重启了也可以重新得到会话状态，还可以使用我们之前提到的 eureka+gateway 来解决。然后我们再谈谈 MySQL.

Q: 为什么要集群？

A: 提高应用的可靠性，一个崩了另一个可以接管；还可以做负载均衡。

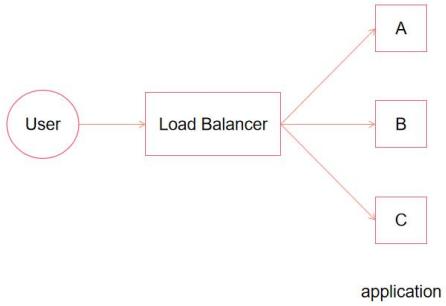
无论集群多大，我们希望对用户暴露出来的是一个节点。所以我们不能让集群中的所有节点暴露出来，路由对于用户应当是透明的。

接下来我们来看一下会话粘滞性是什么意思。用户请求来到负载均衡器了以后，有几种调度策略：

1. 轮训调度
2. 根据服务器上的连接数最少（认为压力最小），调度过去。当然连接数和压力不一定正相关，这个是比较粗糙的调度方法。
3. 云操作系统中会去讲用 K8S 监控所有机器的细节去调度它。
4. 过来的请求，一个用户一旦访问了一个集群中的节点，以后这个用户的所有请求都在这个服务器上。比如根据 IP 来哈希到机器上，这样就可以让用户会话状态绑定到这台机器上。

集群就是一堆机器来提高冗余，这就提高了可靠性。比如说，一台机器的故障率为 $f\%$ ，那么整个集群的可靠性就是 $1 - (1 - f\%)^n$ 。

所以会话粘滞性是集群中最难搞定的事情。我们来看如下的一个场景：



在我们现在的应用场景中，一个负载均衡器(LB, load balancer)后跑了 3 个相同的 application 实例。当用户请求过来的时候，它的 cookie 中会携带着 sessionID。然后我们的 application 会通过 `request.getSession()` 来得到维护的会话状态，如果为空就新开一个 session。业务逻辑处理完后，比如我们的购物车信息就放回到这个 session 会话中，并且把 sessionID 发回给 client。

那么问题在于，Load Balancer 如果使用轮询调度或者根据服务器负载来调度的话，第一次用户想把物品加入到购物车调度到了 A 上，第二次调度到了 B 上。因为 A 和 B 上的 session 会话不共享，所以用户就会发现之前加入到购物车的物品丢失了。这就是会话粘滞性问题。

解决方案：

1. 负载均衡器把同一个用户的请求调度到同一台机器上，不管 A 有多忙，我们也要强行把这个用户的请求调度到 A 上。也就是拿 IP 来做哈希。
2. 我们不要把 session 对象放到 A 的服务器上，而我们放到一个第三方存储中，比如 Redis, MemCached, MySQL 等。这样 A,B,C 都会尝试从这个存储上去存取 session 对象。

第一种方案违背了我们建立集群希望机器平摊负载的初衷，是有缺陷的。而方案二的问题就是我们使用的第三方存储，如 Redis，成为了一个单一故障节点，如果它 crash 了，那么多个 application 都会出问题。

Nginx

我们先看使用 Load Balancer 的情况。我们可以用 nginx，它有很多的网站都用它。它跑起来是很简单的，安装完直接启动就可以了。关键是我们要配置 nginx 怎么管理这些服务器。nginx 除了做负载均衡之外，还可以起到 gateway 的作用。

比如说我们现在有一个网站，它的 `index.html` 在 `/data/www` 路径下，而它的图片在 `/data/images` 下。我们在访问的时候肯定不希望使用 `localhost/data/www/index.html`，而我们希望直接就是 `localhost/index.html`。所以我们就可以使用如下配置：

```
server {
    location / {
        root /data/www;
    }
    location /images/ {
        root /data;
    }
}
```

所以它就会把所有访问到 `localhost/images` 的，都导航到 `localhost/images`，把剩下的访问请求都导航到 `localhost/data/www` 位置。所以它的作用和 `gateway` 是一致的。

然后它还可以做一些别的约定，比如我们可以约定以 `gif,png,jpg` 结尾的文件才做这么一个映射。

```
server {
    location / {
        proxy_pass http://localhost:8080/;
    }
    location ~ \.(gif|jpg|png)$ {
        root /data/images;
    }
}
```

我们还关心 `nginx` 做负载均衡的功能。它分为三种策略：

1. `round-robin`: 轮询
2. `least-connection`: 根据连接数的最小的机器来转发，所以它保证了多台机器上每台的连接数是一样多的。
3. `ip-hash`: 负载不均衡，但是可以保证会话粘滞性的问题。

```
http {
    upstream myapp1 {
        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
    }
    server {
        listen 80;
        location / {
            proxy_pass http://myapp1;
        }
    }
}
```

上面设置了 `myapp1` 对应了 3 个地址，也就是访问 80 端口的时候，就等于访问 `myapp1` 定义的三台机器，访问策略默认是轮询的。在这个配置之下，那么有请求到达 `localhost:80` 的时候，就会轮询地往这三个机器上发。我们可以在其中的所以加上 `least_conn;` 或者 `ip_hash;` 来设置对应的策略。还支持一些加权的轮询策略，因为三台机器的算力可能不一样。

Redis

第二种方案就是我们起一个第三方的存储来存储会话状态。我们可以跑 `Redis`，代码比较复杂。我们要先跑起 `Redis`，应用跑在 8002 端口接受请求。8001 的 `Gateway` 就是来做路由。

具体代码略，见 21-clustering, P25~P34。

在这个代码里头，`gateway` 这个应用里面，访问 `login` 的话就会导到 8002 这个端口里面，

前面加了个 session 的 url，其他都一样。

这是我们为了把会话导入 redis 所做的一个动作。

主类会访问 redis， 默认在 6379 端口上，你可以设一个 timeout（以毫秒计算）。这个会话状态如果写入 redis，那么内存里有一个 timeout，一旦 timeout 到了就会把会话状态删掉。

关键是要使用 redis，有两个问题要解决：

1. redis 是怎么配置的。
2. redis 的会话是怎么写入到 redis 里面去的。

刚才在里面 redis 已经配好了，那么要拿到 redis 的客户端就要带一些属性，比如机器、端口、redis 密码、timeout。都从前面的配置文件里拿到。拿到之后就用 jedis 获取连接池的对象。

这个例子跟我们本身写的没有太大关系，目的就是拿到 jedis 的 pool 就是拿到一个 jedis 的连接拿到一个 redis 的 template，通过它可以往 redis 里写东西。

由于前面的加了"/session"的过程，除了 session/login 之外，都需要被 redis session interceptor 这个拦截器给拦截掉（他就在检查你里面有没有 session）。他的目的就是对所有的请求过来都去看看你里面有没有 session。

底下就是这个 controller，本身没有什么特殊的，就是获取 session，然后 redis 的 template 把我这个设的键值对来说明这个用户曾经登陆过。而真正 redis 是怎么写入的是靠一个 annotation 实现的，就是在 redis config 里写了，有一条 enable redis http session。这个 annotation 在 spring 拿到之后就会把 session 写到 redis 里面去。就是如果除了 session 之外你还想写入别的东西，你也可以自己写。因为有指定使用 redis session，在拿 session 就是从 redis 里拿，而不是从 tomcat 里拿了。

跑起来之后大家可以看到，它本身是在 8002 端口，我们 getInfo，他会告诉我们没登陆过，他会出现 401，这是页面代码里写的。所有的请求都会被拦截然后进行验证，他就会说你这个 getInfo 是没有 session 的。

然后我们点一下 login，这个 login 他是不做检查不会被拦截的（被 exclude）。Login 不管过来之后，他就会 getSession，但是一开始 get 不到，他就会生成一个 session 出来，然后写到 redis 里面。在后面再访问比如说 getInfo，就能输出 session 的信息，证明确实有这个 session。

如果我们把这个应用停掉，那正常情况下，session 在 tomcat 里跑，停了内存就没了，重启之后就是空的。但是写到 redis 里，我们在这边可以直接拿到之前的 session。我们可以直接查看 redis 的客户端来验证是之前的 session。

让这个例子更复杂一点，我们把前面的 gateway 跑起来。gateway 会做一次路由。先 login 再 getInfo，可以看到 redis 里加入了一个新 session。

把 session 放在 redis 里的好处就是你在这三台机器上调度，用任何一台都可以读到某个 session。

缺陷就是要是把 redis 关掉，前台就无法获取 session。第二个问题就是如果你在用 IP hash 的话，每个服务器会存一部分 session，而现在存在 redis 里，服务器就需要取 redis 里找 session，他就是一个进程间的通信了，会更加耗时。

这两种方案都可以用，各有优缺点，看看你看重哪一边。你觉得小集群就用 redis 或者 ip hash 无所谓，大集群太不平衡了不好就不用 ip hash。

IP hash 是不保证负载均衡的，但保证会话在用户访问的那台机器上，调度的时候就比较简单。

Redis 相当于你要专门找一台机器去存。这就是单一故障节点，也许你应该再备份一个

redis 出来。也未必非要是 redis，你说用 mysql 行吗，也行但是比较奇怪。他是个内存对象，你非要给他写到数据库里，数据库是个持久化存储，写硬盘速度就会变慢。为了保证会话的粘滞性，会频繁读写，看起来也不好。一般用 redis 或者 memory cache。

现在说的都属于后端的前面在内存里面的瞬时性对象的缓存问题，那么后端的这个集群该怎么办。那么数据库该怎么建一个集群，数据库撑不住你 tomcat 开再多也没用。

那么数据库也要建集群，mysql 提供了建集群的方法而且非常简单。官方给的 cluster 怎么去建。Mysql 至少要 3 个才能开集群，两个不让。

他这个就在告诉你怎么建一个主从备份。要是不想在 docker 里跑，要注意的就是要给他加一些主从备份的配置。

这样一个集群功能，要注意的就是他必须有三个节点，然后集群实际上有两个目的：

1. 大家要复制一下，如果某一个崩了，数据还在，做备份
2. 前端你有再多的 tomcat，如果后端数据库扛不住也白搭，因此做备份的时候两个从节点也不是闲着的，可以在上面做一些读的任务（写只能在主节点上写）。那么主节点是 RW 就是可读写，从节点上 RO 就是 read-only。读的任务可以分给从节点。

对你来说想要的就是怎么建这个集群，就像我们之前看的 Nginx 一样，有一台机器比较强大，就给他多一点任务。现在的事情就是怎么去设。要是我们把数据库直接暴露在公网上，那么别人直接绕开 router 来访问不太好。因此通常情况下我们把服务器都布置在内网里，只暴露一个 Router 来访问。

你先找一台服务器跑了 mysql 之后，然后把它当成一个种子实例。如果你配的一主多从，那么这个种子服务器就可以当作 primary，如果要多个 primary，那它就是 primary 之一。

有了这个种子，后续添加 cluster 对象即可。

注意你在加实例进来的时候，他会有一个问题就是我们是希望这个机器崩了会有其他的来接管。为了保持数据一样他会怎么办，他会在从的上面重新执行一遍再开始接管。因此有一个设计上的策略，就是我们看到的机器是一个主一个从，在主上写然后重复到从，那么就会想，如果每次都要这么做，那么这个从是不是有点浪费。要是没有读的任务从就白写了，而且没法保证主崩掉的时候一定能接管。所以比较好的方式（策略）就是每隔一段时间（一小时），把主从角色互换一下。

默认策略是冷备份（Cold）到主再备份到从。隔一段时间交换就叫暖备份（Warm），热备份（Hot）是写的时候同时写到主从。

Cold：崩了的时候接管的时间花费比较长。

Warm：好一点

Hot：直接接管。

冷到热耗电量越来越高，可靠性也越来越高。

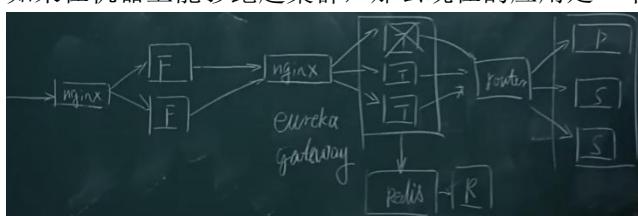
真正设计的时候在三种之间权衡。不局限于 Mysql，Tomcat 也有这样的操作。

做备份的方法：增量、克隆，和之前讲的数据库备份的方法是一样的。

还可以改变集群的拓扑结构，可以在多主节点和单主节点之间来回切换。

集群中的权重也可以设置，权重不一样，那么读写操作会按照权重来。如果出现需要接管的情况，接管的概率也会不同。

如果在机器上能够跑起集群，那么现在的应用是一个什么样子。



2021/11/29

Xen

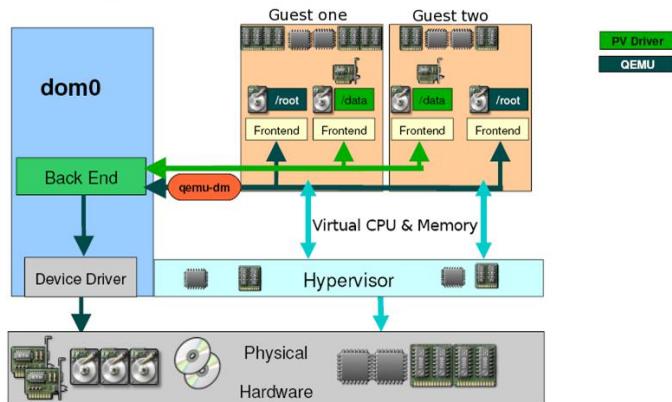
Xen 就是一个虚拟化的技术。Java 虚拟机（JVM）是什么概念？我们现在有 java 的字节码，它是在任何一个操作系统上都不能跑的，所以我们需要 JVM 来帮助解释这个字节中间码。虚拟机就是模拟当前不需要的系统，JVM 保证了同一个字节码在任何机器上都可以一致运行。像上节课讲的集群，A 跑在 8080 端口，B 跑在 8090 端口。这造成我们需要在 application.property 里去写清楚，这导致代码 A 和代码 B 的代码是不同的，但是我们希望一份代码可以无限扩展，这就需要我们的虚拟机去跑两个程序。

我们在讲分层结构，我们希望每一层对于上面一层来说就像是虚拟机一样。分层就是提高代码的独立性，通过 API 访问我们就实现了接口与实现分离。ORM 映射我们底层可能用了各种各样不同的数据库，但是上层代码希望把代码的实现细节屏蔽掉。虚拟机也是一样的。但是虚拟机带来的缺点就是它的效率要低很多，以 ORM 映射为例，它要把对于对象的操作翻译成 SQL 语句并且通过 JDBC 发送过去。我们插了一个很大的虚拟机来描述不存在的环境，所以性能显然受到影响。

虚拟机需要跑一个完整的操作系统。Xen 和 K8S 有一种想法就是我们把依赖的 Linux 内核改掉，改成对 Xen 的 hypervisor 的访问。我们对 Linux 内核的重新编译，修改为对 Xen 的 hypervisor 的调用，然后 hypervisor 再和硬件去交互。好处就是性能比较高，缺点就是要重新编译 Linux 内核。这就是半虚拟化和超虚拟化的方式。

Xen Full Virtualization Architecture

With the para-virtualized drivers



橙色的就是一个完整的操作系统，对上就可以跑各自的应用。这个原生的操作系统，内存和 CPU 的调用直接传给 Hypervisor 去调度。DOM0 可以认为是管理的虚拟机，IO 操作全部发送给 DOM0，然后转换为对真正物理机上的操作。还有一种用的是 qemu(Qemu-DM 帮助完全虚拟化客户机（Domain U HVM Guest）获取网络和磁盘的访问操作），它是软件去模拟的虚拟，后面演化为全部使用 PV driver。

The diagram illustrates the performance matrix for Xen, comparing different virtualization modes (HVM, PV, PVHVM) against various hardware and software features. The legend indicates:

- Poor Performance (Red)
- Scope for Improvement (Yellow)
- Optimal Performance (Green)

Legend definitions:

- P = Paravirtualized
- VS = Software Virtualized (QEMU)
- VH = Hardware Virtualized

Shortcut	Mode	With	Disk and Network	Interrupts & Timers	Emulated Motherboard, Legacy Boot	Privileged Instructions, Page Tables
HVM / Fully Virtualized	HVM		VS	VS	VS	VH
HVM + PV drivers	HVM	PV Drivers	P	VS	VS	VH
PVHVM	HVM	PVHVM Drivers	P	P	VS	VH

最终 Xen 的三种组合方式如上图所示。

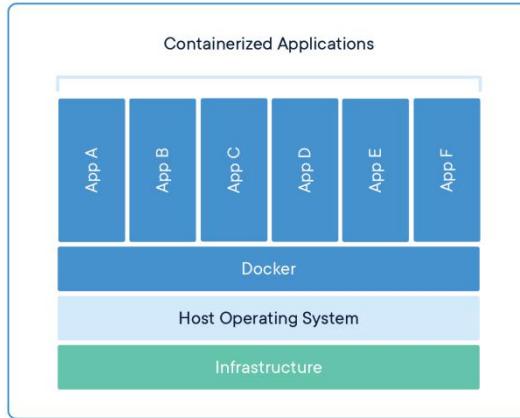
这里面我们要讲一个问题。我们这个虚拟机是怎么样跑起来的呢？我们一样需要一个 `image` 文件。这个 `image` 应该怎么去存放呢？它可能至少有 50 个 G。一种方法是我们要跑虚拟机的时候直接从硬盘上加载，但是这就太占空间了，因为我们可能跑多个不同的虚拟机。我们可以在集群中搞一个 `image` 的库，一旦集群中有机器要跑的时候，我们就从集群的 `image repository` 中把 `image` 拉过来，但是涉及到 50G 的传输似乎也很大。这时候可以从三备份的 `repo` 上同时拉同一个 `image` 的三块部分。

又比如说，在一台已经存放了 `win10 image` 的机器上我们想再跑一个 `win11` 的虚拟机，此时我们知道 `win10` 和 `win11` 的 `image` 有 30G 数据是相同的。那么此时我们就希望只拉去剩下的 20G 不同的 `win11` 的 `image` 的数据块，复用相同的 30G 的数据块。所以一个 `image` 也分成了一些可以复用的小的块。在 `Docker+K8S` 进行管理的时候，我们需要用 `OpenStack` 来管理 `KVM`。它可以配置虚拟机的可靠性要求是多少，如果虚拟机崩了，它就会自动再起一个。`OpenStack` 要监控 `KVM` 的集群，那么数据就会放在 `TSDM`（时间序列数据库）。我们可以让它全部写到 `influxDB` 中去写，这样我们看到整个虚拟机是容器部署，然后通过 `K8S` 和 `OpenStack` 去做监控，数据在时序数据库中。如果我们将要做分析就通过这个时序数据库来做。

容器

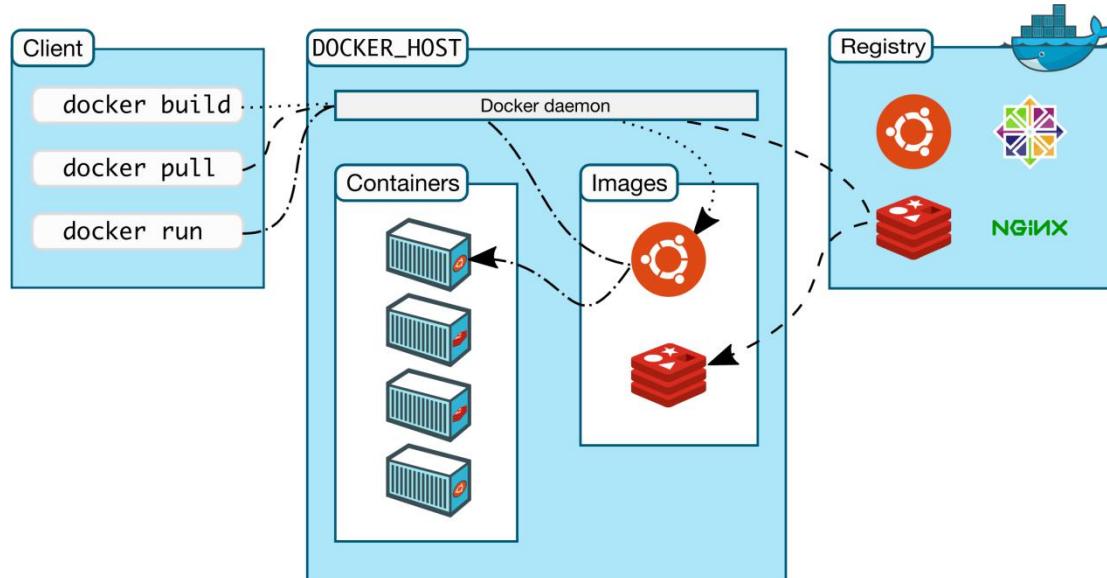
容器是什么呢？底下有一个 `hypervisor`，如果我们要跑一个完整的操作系统，就很浪费资源。我们是为了做隔离才让不同的程序跑在不同的虚拟机上（运行 A 的虚拟机崩了，程序 B 和 C 跑在其他的虚拟机上还是可以继续运行）。我们假设所有的程序都在 `Linux` 上开发，这样我们只需要把应用 A 调用到的第三方的库全部打包下来跑在 `docker` 里，这样我们就不需要跑一个完整的操作系统，这样容器就可以跑更多的应用。所以容器就是把当前的代码和依赖的所有东西都打包在一起。如果是对 `windows` 的调用，那么 `docker` 会帮我们做转换。

这些 `docker` 容器中，都不是一个完整的操作系统。在 `docker` 我们都当做一个单独的进程跑，所有对系统的调用都交给 `docker` 来做转换，所以同样的配置可以跑更多的 `docker image`。



Docker 里有一个 docker engine，里面最核心的东西是一个运行时。Docker 是一个容器，容器就不是一个完整的应用，是把我们写的代码和我们需要依赖的所有东西打包在一起。构成这样一个软件单元，这样将来在跑的时候只需要创建一个容器跑起来即可。注意，打包在一起也是分层打包的，比如 nodejs 从云端 pull 下来了很多 layer，当我们本地将来另一个软件需要复用相同的层做打包的时候，就可以不再从云端拉取。

容器的 image 在加载过来以后，我们在这里就可以起多个实例。也就是拿 docker image 跑多个容器过来。我们的系统调用直接全部发给 docker 引擎，让它自己去管理。



在运行的时候，有一个 image 的注册中心，里面已经打包好了各种各样的 image。用的时候非常方便，使用 `docker pull + docker run` 就可以直接跑起来了。我们将来总是在一个现有的 image 上去开发我们的应用。我们下载下来之后，`run` 的时候 docker 就在本地去找，如果找不到就去远端 pull，然后运行 image。比如我们一个 mysql 跑了 3 个实例，它们互相之间在做主从备份。每个容器有一个独立的 ip 和端口，当然我们可以端口映射过去。如果我们要通过 ip+端口去访问不同的容器，就很麻烦。

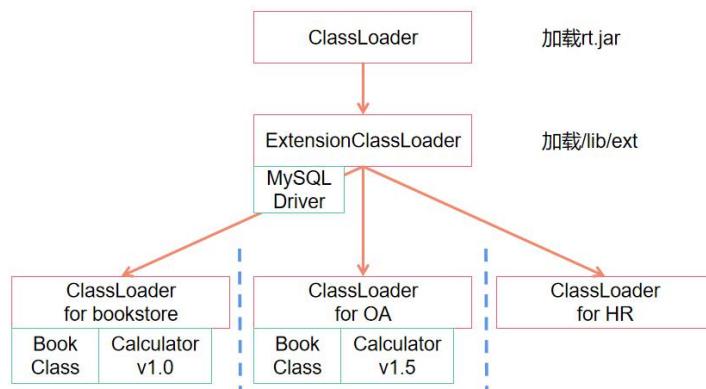
但是如果我们把当前机器的 1000 端口映射到 redis image 的机器的 6879 端口，这样我们就只要访问当前机器就可以了。这就和我们之前讲的 gateway 一样。

这里面有一点比较重要的是，比如一台机器上跑了 2 台虚拟机。虚拟机会不会互相访问到对方的硬盘呢？在 Linux 操作系统里面有一个分组的概念。或者说，我们 C++为了防止同名类，我们加了 namespace 来做隔离，同样的 OS 中也是使用了类似的机制做隔离。在 docker 里也是一样，必须要通过网络才可以访问别的 docker 中的数据。

为什么最好把所有东西隔离呢？比如我们现在开发的 `bookstore` 做大了以后，可能还添加了 `OA` 的网站和 `HR` 的网站，部署的时候可以部署到同一个 `tomcat` 的，但是这就带来了一个问题就是这个 `tomcat` 如果 `crash` 了，那么这三个应用都崩了。所以我们选择把这三个应用部署到三个 `container` 中。

Java 虚拟机

Java 虚拟机就是解释执行 `java` 代码的环境。因为 `java` 是面向对象的，编译出来都是 `.class` 文件，所以我们需要一个 `classLoader` 来加载这些类。`classLoader` 会先加载 `java` 最核心的包，也就是 `rt.jar`。然后它加载一些扩展对象：`extensionClassLoader`，也就是加载 `/lib/ext` 里的东西。然后 `tomcat` 每碰到一个新的应用，比如我们的 `bookstores`，`tomcat` 就会新创建一个 `classloader` 来加载 `bookstore` 中的类（使用到了再被加载）。所以，我们的 `bookstore`，`OA`，`HR` 应用之间是隔离的。



每当我们需要加载一个类的时候，比如 `bookstore` 这个应用想加载 `Book` 类，它会往上问有没有父 `ClassLoader` 加载过了，如果没有的话就自己来加载。所以我们可以知道 `OA` 是不能访问 `bookstore` 中的 `Book Class` 的，要自己去加载，这就保证了 Application 之间的隔离。但是当我们想加载 MySQL 驱动的时候，不应该把 `mysql` 的驱动程序打包到应用里，而应该放到当前运行环境的 `/lib/ext` 里，这样就可以避免重复加载浪费内存了。比如说，比如 `bookstore` 和 `OA` 都依赖于一个第三方计算器插件 `calculator`，但是依赖的版本不一样且不太兼容，这样我们就可以分别打包到各自的 `war/jar` 包中，来实现隔离。

所以在 `tomcat` 中，`classLoader` 就是类似于一种 `namespace` 的机制把大家隔离开。

Docker 的具体打包和运行

The screenshot shows a code editor interface with an 'EXPLORER' sidebar on the left containing files: 'spec', 'src', 'package.json' (which is selected), and 'yarn.lock'. The main area displays the contents of 'package.json':

```
1  "name": "101-app",
2  "version": "1.0.0",
3  "main": "index.js",
4  "license": "MIT",
5  "scripts": {
6    "prettyify": "prettier -l --write",
7    "test": "jest",
8    "dev": "nodemon src/index.js"
9  },
10 }
11 "dependencies": {
12   "body-parser": "^1.19.0"
}
```

现在有一个例程，文件夹内容如上所示。我们需要在包含 `package.json` 的相同目录下创建 `Dockerfile` 文件。

```
FROM node:12-alpine
RUN sed -i 's/dl-cdn.alpinelinux.org/mirrors.aliyun.com/g' /etc/apk/repositories
RUN apk add --no-cache python g++ make
WORKDIR /app
COPY .
RUN yarn install --production
CMD ["node", "src/index.js"]
```

在/app 包里面的东西一起打包进去，然后 build 一下。所以 docker 现在有所有运行时的环境。

我们使用命令 “`docker build -t getting-started .`” -t 就是取别名。

“`docker run -dp 3000:3000 getting-started`” -d: 后台运行容器，并返回容器 ID; -p: 指定端口映射，格式为：主机(宿主)端口:容器端口



Remove a container using the CLI

- Get the ID of the container by using the docker ps command.
 - `$ docker ps`
- Use the docker stop command to stop the container.
 - `# Swap out <the-container-id> with the ID from docker ps`
 - `$ docker stop <the-container-id>`
- Once the container has stopped, you can remove it by using the docker rm command.
 - `$ docker rm <the-container-id>`

Start the updated app container

- Now, start your updated app.
 - `$ docker run -dp 3000:3000 getting-started`

如上是 docker 的一些功能。在之前的例子中，容器如果被删掉再重启内容就没了，因为我们的内容没有做持久化，那么能不能让数据在容器被删了之后还在呢？我们就要考虑把内存持久，也就是真正写到数据库里。每一个 contrainer 进去以后看到的是自己的文件系统。我们在 contrainer1 里修改数据，如果我们添加 container2 里，我们是看不到修改的，因为这两者之间是隔离的。

- The container's filesystem
 - When a container runs, it uses the various layers from an image for its filesystem.
 - Each container also gets its own “scratch space” to create/update/remove files.
 - Any changes **won't** be seen in another container, *even if* they are using the same image.
- To see this in action, we're going to start two containers and create a file in each.
 - Start an ubuntu container that will create a file named `/data.txt` with a random number between 1 and 10000.
 - `$ docker run -d ubuntu bash -c "shuf -i 1-10000 -n 1 -o /data.txt && tail -f /dev/null"`
 - Start another ubuntu container.
 - `$ docker run -it ubuntu`

每次都要启动的时候手动加到`/data.txt`比较麻烦，能不能每次启动的时候自动把文件加进去呢？

Volume 就是一个卷的意思，可以理解为就是一些数据，起程序的时候，我们要把外部的 volume 挂载到容器中的文件系统中。可以认为是一个目录挂在到 docker 里。

假设我们要把一个数据存到硬盘上，我们就要先创建一个叫做 `todo-db` 的 volume。

1. Create a volume by using the docker volume create command.
 - \$ docker volume create todo-db
 - Stop and remove the todo app container once again in the Dashboard (or with docker rm -f <id>), as it is still running without using the persistent volume.
2. Start the todo app container, but add the **-v** flag to specify a volume mount.
 - We will use the named volume and mount it to **/etc/todos**, which will capture all files created at the path.
 - \$ docker run -dp 3000:3000 -v todo-db:/etc/todos getting-started
3. Once the container starts up, open the app and add a few items to your todo list.



然后在 docker 启动的时候挂载到容器中的 /etc/todos 里。我们可以往 /etc/todos 里挂载不同的卷得到不同的效果。注意现在是把数据写到文件上。

```
(base) MacBook-Pro-7:~ chenhaopeng$ docker volume inspect todo-db
[{"Name": "todo-db", "Driver": "local", "Scope": "local", "Mountpoint": "/var/lib/docker/volumes/todo-db/_data", "Options": {}, "Labels": {}}, {"Name": "nodejs", "Driver": "local", "Scope": "local", "Mountpoint": "/var/lib/docker/volumes/nodejs/_data", "Options": {}, "Labels": {}}]
```

mountpoint 就是在当前本地机器的 **volume** 文件的存放地址。这样就告诉我们，容器重启以后，只要挂载进相同的 **volume** 就恢复了数据。

问题：挂载远程服务器上的 **volume** 行不行？

A: **named volume**: docker 处理存在哪里； **bind amount**: 可以挂载一个远程的卷。

- Start a dev-mode container
 - To run our container to support a development workflow, we will do the following:
 - Mount our source code into the container
 - Install all dependencies, including the “dev” dependencies
 - Start nodemon to watch for filesystem changes

1. Make sure you don't have any previous getting-started containers running.

2. Run the following command in the directory of **getting-started-master/app**

```
$ docker run -dp 3000:3000 \
  -w /app -v "$(pwd):/app" \
  node:12-alpine \
  sh -c "yarn install && yarn run dev"
```

- **-dp 3000:3000** - same as before. Run in detached (background) mode and create a port mapping
- **-w /app** - sets the “working directory” or the current directory that the command will run from
- **-v "\$(pwd):/app"** - bind mount the current directory from the host in the container into the /app directory
- **node:12-alpine** - the image to use. Note that this is the base image for our app from the Dockerfile
- **sh -c "yarn install && yarn run dev"** - the command. We're starting a shell using sh (alpine doesn't have bash) and running yarn install to install **all** dependencies and then running yarn run dev. If we look in the package.json, we'll see that the dev script is starting nodemon.

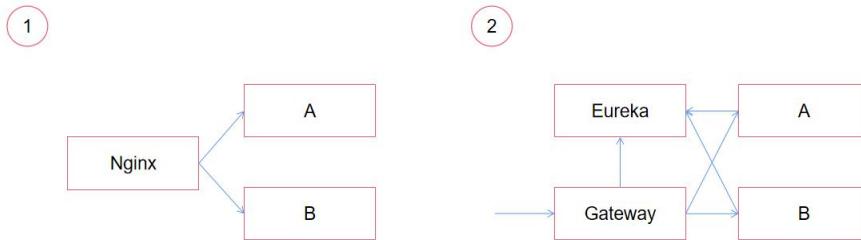
使用 **-v "\$(pwd):/app"**，把本地的 **app** 目录挂载到了 docker 里的 **app** 目录里去，那么带

来的好处就是我们修改本地的 app 目录，里面就直接变了。不需要再把 docker image 重新编译一下才能看到里面的效果。

Q: 我们希望把数据存在数据库怎么办？

A: MySQL 一个容器，应用一个容器。数据通过网络让容器之间通信。

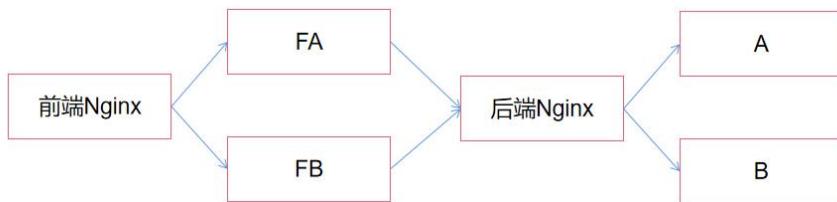
2021/12/02



对于图 2 来说，目前我们有两个实例，所以就根据注册的名字来做负载均衡策略。前端需要全部指向 gateway。

对于作业 10，这两个方案都可以用。通常情况下情况 2 更好一点，因为机器如果 crash 了，因为重启以后会重新到 eureka 注册一遍，所以不会有任何问题。但是在方案 1 中，如果 server 的 ip 和 port 变了，那么可能需要修改一些配置文件。

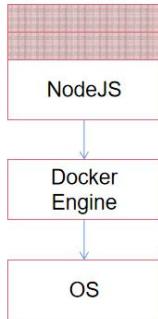
所以这两种方法都可以实现负载均衡策略。微服务只是说可以单独去部署，所以是不是微服务和上述两种方案不矛盾。



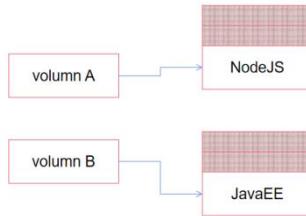
如果我们加上前端，那么就是前端也有一个 nginx，请求全部发到前端的 nginx，有 FA 和 FB，然后再发送到后端的 nginx 即可。我们要先弄出集群，然后再说能不能在 docker 中跑。因为在 docker 中跑可能遇到各种各样的问题，比如说我们要 pull 一个 image 的时候，我们可能遇到一些障碍，比如来自外网的一些 image。

容器实际上是在我们跑的时候，会给我们打包一些依赖的库。但是它又觉得大多数的库都很像，所以它比如说打包了一个 nodejs 出来。那么我们就可以拉下这个 nodejs 下来。然后我们再往上打包进入我们的应用和代码，于是这个应用就可以在 docker 容器中跑起来。

Docker 的引擎就是把我们所有的调用转换为对底下操作系统的调用。于是 docker 引擎做了很多其他的处理，比如文件系统的管理、任务的调度、内存的管理，它通过调用底层的操作系统来实现的。我们要注意 docker linux 是相当复杂的。



卷可以在不同的容器中被复用。这就实现了我们的数据和要跑的容器的解耦。如果我们的文件系统在其他地方呢？



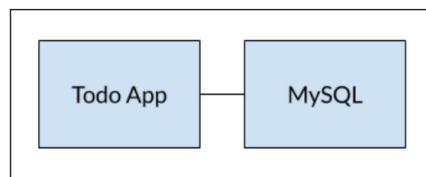
于是 docker 就说，它允许我们通过 `bind mount` 方式来指定任意一个目录放到容器里的一个指定的位置上。所以我们可以把我们写的 `/app` 目录 `mount` 到 docker 容器的里面去。所以，我们在我们的机器上做了开发，在容器里看到的就是修改之后的状态，这时候容器的 `image` 并没有被停掉重新 `build`。所以用 `named volume` 和 `bind amount` 允许我们在容器之外做开发。所以一个 `image` 可以挂载不同的 `volume` 来表现出差异。

其实 `volume` 这一块也未必是 docker 指定位置的东西，它甚至可以是一个远程的存储，比如在阿里云中有一个远程的存储放进来也是可以的，但是性能会受到影响。

下一个问题是，我们的数据不再是那么简单的一个文件系统，如果我们的数据现在在数据库里。

In general, **each container should do one thing and do it well.** A few reasons:

- There's a good chance you'd have to scale APIs and front-ends differently than databases
- Separate containers let you version and update versions in isolation
- While you may use a container for the database locally, you may want to use a managed service for the database in production. You don't want to ship your database engine with your app then.
- Running multiple processes will require a process manager (the container only starts one process), which adds complexity to container startup/shutdown



我们现在的存储存在一个 MySQL 容器里面。那么我们为什么不做一个包含 NodeJS 和 MySQL 的容器呢？因为我们希望 NodeJS 崩了，MySQL 还能跑。因为我们未来扩容的时候可能 NodeJS 是一个实例，而 MySQL 是两个实例（主从备份）。它们此时就不再是一一对应的关系了，打包在一起不合适。另一个角度是，一个 `image` 也不是一个大的文件，`image` 是分层的，比如 NodeJS 和 MySQL 各自可能就分出了若干层。即使是不同的 `image`，可能里面部

分层也是一样的，所以我们在 pull 一个 image 的时候，它会独自拉取很多的层。在 docker 管理里面，要 pull 一个 nodejs 就会有一个 layers 的拉取清单。然后把本地原先就有的 layer 和新拉取的层合在一起即可，所以我们也没有必要把 NodeJS 和 MySQL 变成一个 image。

所以我们放在两个容器里的时候，它们之间的通信就要走网络了。

Start MySQL

- Create the network.

```
docker network create todo-app
```

- Start a MySQL container and attach it to the network. We're also going to define a few environment variables that the database will use to initialize the database.

```
docker run -d \
--network todo-app --network-alias mysql \
-v todo-mysql-data:/var/lib/mysql \
-e MYSQL_ROOT_PASSWORD=secret \
-e MYSQL_DATABASE=todos \
mysql:5.7
```

为了要通，我们要创建一个网络，它就像一个标签一样，物理上肯定是通的，但我们要让它逻辑上也是通的。这个 MySQL 的数据库，我们把本地的 volume 的形式挂载到我们 MySQL 的容器里。所以这个容器的数据就会都存在本地的 /var/lib/mysql 这个地方。

比如我们起了两个 MySQL 的容器，实际上我们用的是相同的 image，但是只要我们挂载的是不同的文件夹，那么它们的数据之间也是不影响的。

其实我们在 todo-mysql-data 里，就存了一个 todos 这个表。

Start MySQL

- To confirm we have the database up and running, connect to the database and verify it connects.
`docker exec -it <mysql-container-id> mysql -u root -p`

- When the password prompt comes up, type in **secret**. In the MySQL shell, list the databases and verify you see the todos database.
`mysql> SHOW DATABASES;`

- You should see output that looks like this:

```
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| sys            |
| todos          |
+-----+
5 rows in set (0.00 sec)
```

我们就可以用这条命令打开 MySQL 的控制台。进去之后我们就像在操作本地的一个 MySQL 数据库一样。所以底下的 todos 就是我们通过卷挂载进去的。

然后我们要想让它访问到呢，我们还需要起后端的应用。

Connect to MySQL

- We're going to make use of the [nicolaka/netshoot](#) container,
 - which ships with a *lot* of tools that are useful for troubleshooting or debugging networking issues.

- Start a new container using the [nicolaka/netshoot](#) image. Make sure to connect it to the same network.

```
docker run -it --network todo-app nicolaka/netshoot
```

- Inside the container, we're going to use the dig command, which is a useful DNS tool.

- We're going to look up the IP address for the hostname mysql.

```
dig mysql
```

我们可以先其一个 [nicolaka/netshoot](#)，它提供了很多工具。其中的 `dig` 工具就可以在网络里探索我们的 `mysql` 在网络里的什么位置上。这个工具是不一定要跑的，只是更方便让我们 debug。

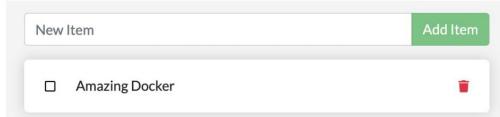
实际上我们的应用是 Todo APP，希望访问 MySQL。

Run your app with MySQL

- We'll specify each of the environment variables above, as well as connect the container to our app network.

```
docker run -dp 3000:3000 -w /app -v "$(pwd):/app" --network todo-app -e MYSQL_HOST=mysql -e MYSQL_USER=root -e MYSQL_PASSWORD=secret -e MYSQL_DB=todos node:12-alpine sh -c "yarn install && yarn run dev"
```

- Open the app in your browser and add a few items to your todo list.



所以我们也要对 Todo APP 的 docker 设置网络，并且挂载当前工作目录的卷。

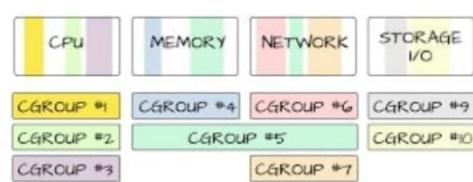
在 docker 容器中提供了一个 compose 的操作，它允许我们把多个容器合到一起让 docker 一次性启动起来。这就回避掉了我们刚才要连续起两个 docker 的情况。如果先启动了 docker 的应用再启动 docker 数据库的情况，这是不能正常运行的（address mysql is not defined）。所以启动顺序的不同可能会导致出现一些意想不到的问题。为了解决这个问题，最好是我们通过 docker compose 人为地指定启动的顺序。

docker-compose up -d 即可启动。docker-compose down 即可关闭。如果我们要在 IDEA 中开发 docker，可以参考

<https://www.jetbrains.com/help/idea/deploying-a-web-app-into-an-app-server-container.html>

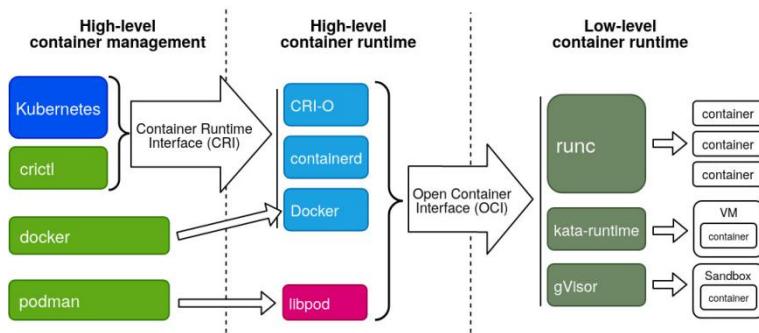
Kubernetes

- cgroup 可以将进程组织成在限制和监控下使用各种类型资源的分层组
- 一个 cgroup 包含多个 task (线程)，可以组成的树状结构层级控制
- cgroup 可以对资源做限制 (最大值、分配比例)、优先级分配以及审计



docker 是存在隐患的。Volume 看起来是为了灵活期间可以 mount 到不同的容器。但是如果脱离的容器直接把 mount 的文件夹改掉。那安全性就会下降。

我们能不能找到一个轻量级的 runtime 放到 docker 里让它跑起来，目的是实现一个网络的软件定义。

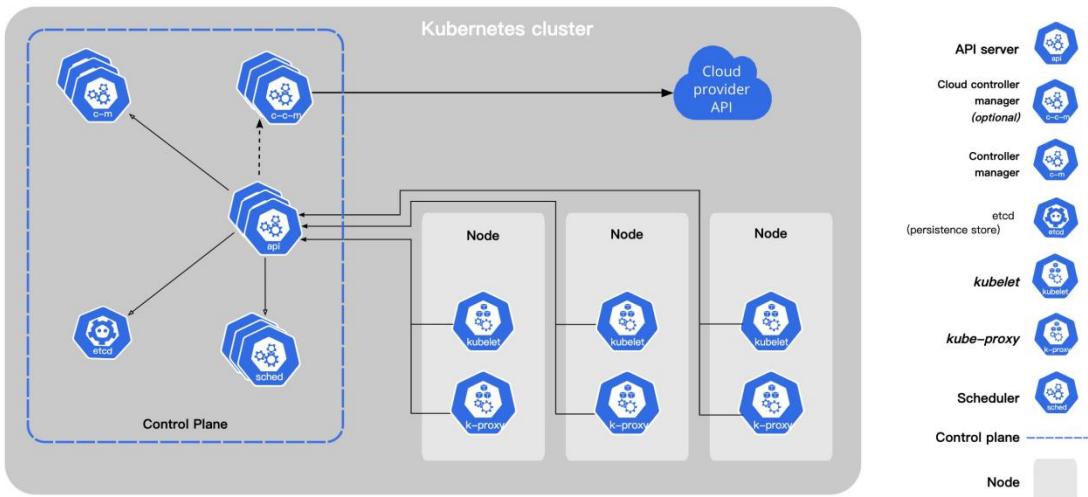


K8S 是拿 Go 语言去写的。刚才我们起 docker 跑的时候起了一个 nodejs。我们现在发现

一个 nodejs 扛不住，希望起多个。如果我们有一个集群有三台机器，每台机器都可以跑 nodejs，但是刚才的操作每台机器都得手动启动，如果容器崩了那么我们得手动发现手动重启。有没有工具能够自动帮我实现这一套东西。我们希望工具来判断把新的容器实例可以放到哪里去，包括容器的重启和回滚。针对虚拟机 KVM 或者 Xen 的，我们就可以使用 OpenStack 就可以使用这个来管理。

所以 K8S 就是用来管理容器包裹起来部署的 workload，workload 可能是一个程序、云盘或者其他东西。它真正跑起来的典型的 workload 就是应用。K8S 就是负责容器的全生命周期的管理，创建、管理、销毁。K8S 可以实现如下的功能：

1. 服务发现和负载均衡，一样的道理，所有的容器都要在 K8S 这里注册一下，用的就是注册中心和 gateway 的方式。
2. 存储编排，把哪些卷怎么样挂接到 container 里去。除了卷，K8S 里还允许卷来自于第三方，比如百度云。
3. 因为程序逻辑的问题，导致程序状态不对了。K8S 可以把程序回滚，在另一个地方启动一个一模一样的容器把状态回滚到过去的某一个点上。
4. 背包问题，容器就是物品，物理机就是背包，怎么让每个包的负载都一致，没有一个特别满的包。
5. 自愈问题，软件导致容器不能正常运行了。管理的状态发生了变化。比如程序陷入了死循环或者抛出了异常，此时就会释放掉资源再起一个新的实例出来。
6. 安全和配置管理的问题，比如挂载的卷的信息的加密、权限管理等。

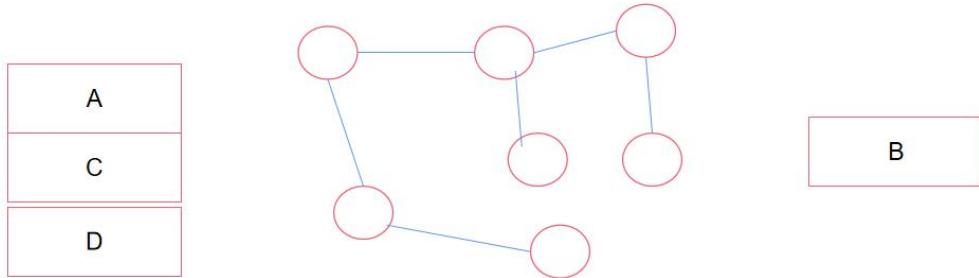


它的结构里就包含了 K8S 里所有的构建。这些 Node 就表示集群中可以跑容器的物理机。这里面虚线框出来的就是 K8S 的控制面，就是它要完成的所有功能。它包含了这 5 种东西，可以每部分在不同的节点上，当然推荐的是控制面所有的构建都跑在一个机器上。

1. API Server，它就类似于我们刚才说的 Gateway，整个访问的入口节点全部在 API 这里，它就有可能是系统中的一个瓶颈，所以 API Server 可以做集群化部署。

2. Sched，这是一个调度器，当有应用要求部署进来的时候，它就去集群里面去判断把这些应用启动在哪些节点上是最好的。所以它要去监控节点状态的数据，存在 etcd 这个实时的存储库中。这里面就出现了一个问题：schedule 的默认逻辑是按照 CPU/内存不能高于多少，希望让大家所有的机器的资源利用率都差不多。比如我们有三台机器 40%, 60%, 80%，那么它可能放到 40% 的机器上。但是我们现在在研究一个网络中，网络节点有一定的图拓扑连接关系。假设 A 想发一个数据给 B，物理的拓扑结构已经在这了，这些圆的节点都是带有交换功能的网络设备。比如 A 是一个前端侦查的无人机拍照，要把一个 1G 数据传输

给 B，而 B 可能是后台的一个指挥员，决定是否要打击目标。我们数据发送的时候就有很多选择，比如到达某个节点以后是否可以压缩一下、做一下加密、到最后是不是一个弄一个防护墙。如果是这样的话，圆的节点都是一个个程序（VNF，虚拟网络单元）。因为 A 要给 B 发一个 1G 数据，在物理链路上可能最终以某条路径来走。



我们说如果有多个请求（A,C,D）过来要发给 B，每个带有 1G 数据，我们网络上能不能在不同的节点去做压缩，当请求少了以后可能这些额外的压缩节点就不需要了。所以我们跑的 VNF 物理结构是没有发生变化，但是从逻辑上这些节点的功能到底呈现出什么功能是我们来定义的。

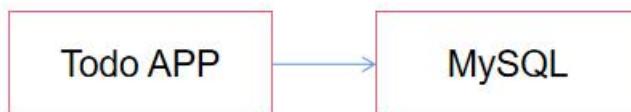
在这种应用情况下，我们不再是根据资源利用率来判断，而是根据用户的需求来启动 VNF（虚拟网络单元），这样我们调度策略就要根据应用的需求来。这样我们就需要去改 K8S 中的调度算法。这就是我们的软件定义网络（SDN）。虽然物理拓扑结构不变，但是每个节点的功能是任务驱动的。

这就是 **scheduler** 要做的事情，在一个非常大的集群里的话，也是要跑多个实例的。这就是在一个集群中的所有 node 一个监控的状态。Etcd 的存放不能再是一个集群，必须存在一个地方。里面存了监控状态、用户名、密码、证书等东西。我们怎么监控多个容器的状态呢？这时候我们就要一个容器的管理器去收集容器的状态。如果我们在公有云上还组了一些虚拟机节点，那么 K8S 还提供了扩展云相关的管理器。比如可以通过阿里云的 API，把阿里云上的虚拟机的状态拿过来做监控。

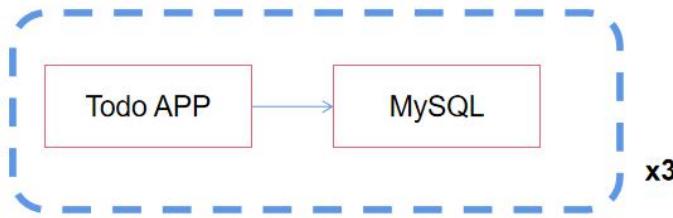
每个节点上都要装两个东西，一个是控制端的小程序（kubelet），发回数据到控制面去做进一步管理；还有一个是 k-proxy，就在管理网络的东西，让集群之间部署的东西可以互相打通，建立一些逻辑上的带标签的网络。

所以总结一下，K8S 实际上就像 ZooKeeper 一样。它眼里监控的就是物理机的节点和容器，而 ZooKeeper 也是在监控一堆机器。

在 K8S 眼里，要部署的就是一个 workload，可以理解为就是一堆需要一起工作的东西，以 pod 的形式出现。Pod（最小可部署单元）就是来描述负载里是由多少种容器构成的。



如上就是一个要一起部署的一个 pod。



如上的 3 个就是构成了一个 **replica set**。所以 **pod** 就和 **image** 一样，只是现在一个 **image** 包含了多个应用。一个 **pod** 就是 **scheduler** 调度的最小资源。

在做容器的迁移的时候，用户是感知不到的。这些复杂性全部由 **K8S** 来管理。

2021/12/06

我们这节课来讲讲云计算。

网格计算：在 20 年以前，把全世界接入互联网的计算机像电力网一样连接起来，当我们有一个计算任务要执行的时候，我们就把任务丢到互联网上，去自动地做分布式处理和任务分发。除非是 P2P 的时候，大家不太希望贡献出自己的计算机。

最早 IBM 看到了机会，把自己的计算资源租出去给别人用。这就是云计算。云的含义是：我们看不见，是在一个远程的拥有大量的资源的机房；我们也看不见我们租借的机器的配置，这些都是屏蔽掉的。它就是一个大量的分布式的高扩展性的计算资源。

硬件也要以服务的形式发布出来，称为虚拟机。如果共享的是软件，那么就是一个软件服务。在云计算里，把所有东西通过 **web** 服务暴露出来，分为三类：

1. 基础设施：把硬件资源暴露出来
2. **PaaS** (**platform as a service**)：如阿里云上的 MySQL 数据库，那么不仅仅有硬件，还配置好了给你。
3. **SaaS** (**software as a service**)：百度提供的 API 服务。

无论是哪种，我们都是通过互联网来访问服务，复杂性全部被屏蔽掉了。它是按需租用的、它可以弹性扩展，这是我们最需要的，这比我们花钱构建一个固定数量机器的机房要灵活地多。

所以它是一个虚拟化的服务。它为什么要虚拟化呢？因为不同人的需求不一样，需要在服务器上装上一种操作系统，然后在上面跑虚拟机，这样就可以满足用户的不断扩展资源的需求。

对于云服务提供商来说，可以使用特殊的机架、特殊的机房，甚至用海水降温来降低成本。而小企业如果要盖自己的机房，投入非常大，云计算就提供租的机会，降低了初期成本，可以快速上线。

云计算提供了很多灵活的定价策略，分为长期 **reserved** 的、短期的和拍卖的策略。第二就是弹性的可扩展，机器拍卖过来以后，关键在于要快速加入到集群中去，如果用注册中心 +**gateway**，那么就是要快速注册进去。快速供给的意思就是，我们确实需要加资源，必须要在 1 分钟内给到添加的资源，我们要预先打包好一个容器或者虚拟机，要快速拿到服务器上启动起来。这就是为什么容器中是分层的，这样一些基础层在机器上本身就有。

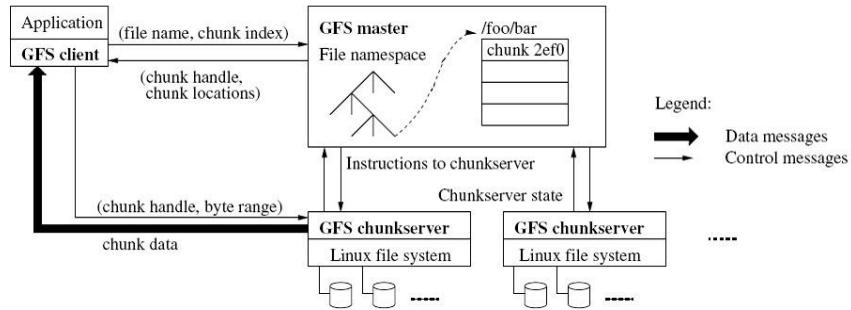
在云里要做负载的优化。然后就是服务的集成管理很重要，我们需要知道服务快不行了，如果服务不行了我们要快速转移到其他地方去。

Q：一个操作系统要解决什么问题？

A: I/O, 内存管理, 作业调度和文件系统。

GFS(Google File System)

操作系统中的文件系统。集群中的文件系统肯定不是一台机器上的, 比如分布式的文件系统 hadoop 中的 hdfs, 它就是 chunk server 来存一个个大块的文件。

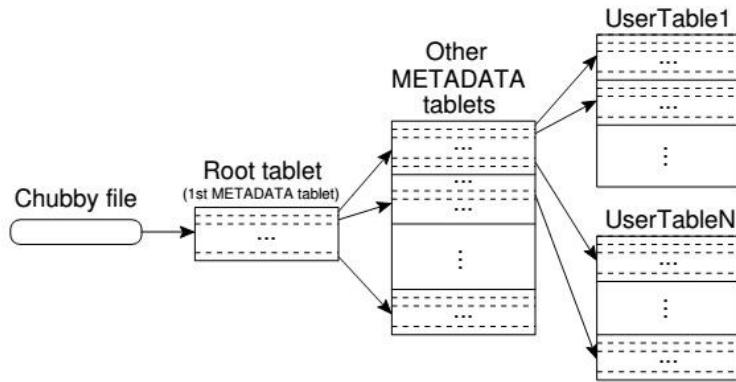


GFS master 节点上就要存储每个文件的 chunk 对应的位置在哪里。这和我们本地文件系统中, 一个 inode 要维护一个 block list 是一致的。

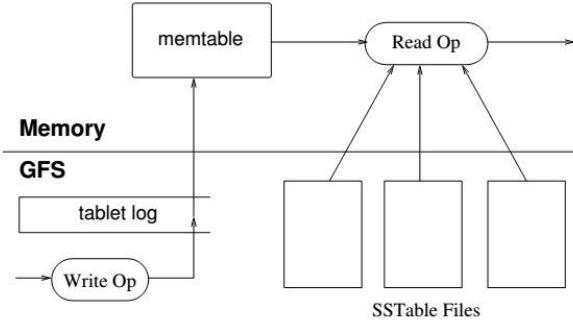
BigTable

在云上的存储需要符合一定的格式才方便查找, 所以就提出了 BigTable 的概念。这个表的规模可以很大很大, 底下是分布式文件系统, 可以切分成很多个 chunk, 理论上没有上限。因为是一张表, 所以就不存在 MongoDB 中多张表带来的问题。

BigTable 做了二级划分, 先要定义列族, 列族里面放一些列。并且列族和列可以动态地增加, 这样比关系型数据库严格的 schema 定义要宽松一些, 表达能力更强。并且它可以存若干个时间戳的值, 它会存状态的变化量。



如果我们系统中有很多的 table, 分出来的 tablet 会很大, 并且导致 metadata tablet 一个机器也存不下, 需要用一个 Root tablet 来存, 不过 root tablet 我们就认为是可以在一台机器上存的下的。

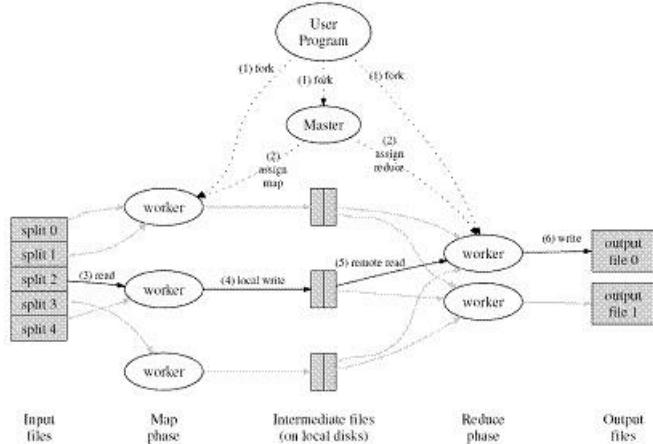


在 BigTable 中，数据写满了也是以 SSTable 的形式落到硬盘上，实现机制就是日志结构合并树。

云里面不存在一个唯一的内存，所以不需要额外的内存。我们可以用 hadoop 跑一下我们学到的知识。hadoop 中提供了 MapReduce, yarn, hbase 等。

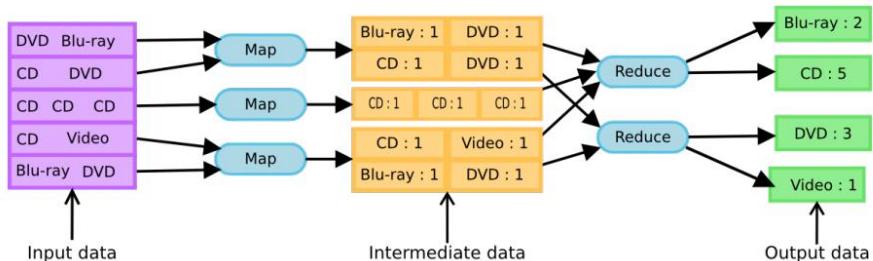
MapReduce

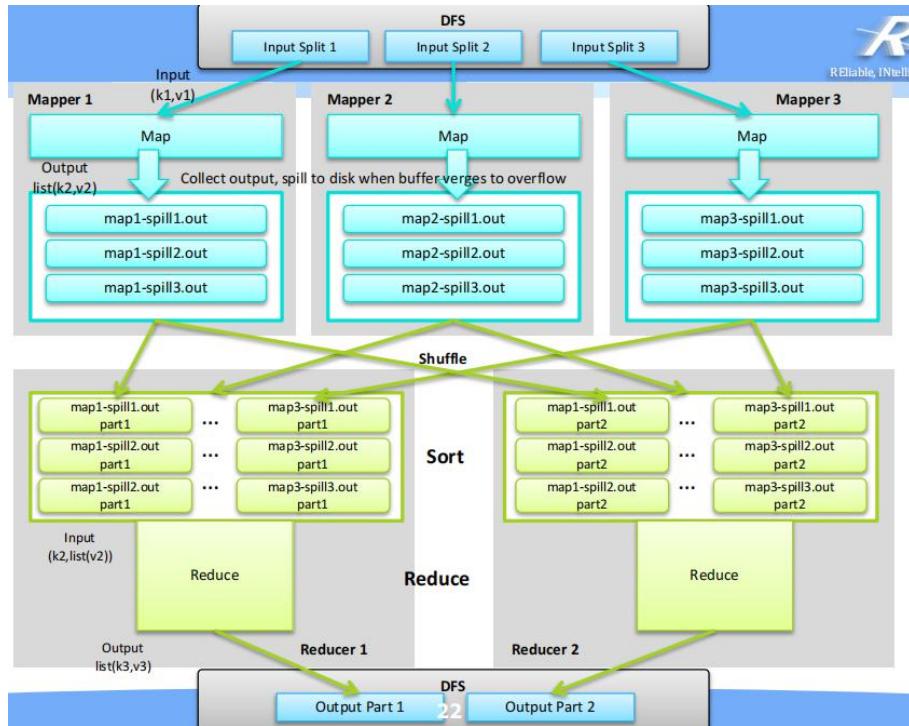
在云操作系统中，MapReduce 对应的就是作业调度，它是一种计算的方法。大的前提是数据一定要在那里了，也就是批数据。而推特、微博上的数据是流式数据，是每秒出现的。



因为 reducer = 2，所以 mapper 的中间结果分成 2 块发给后面的 reducer。一直要等红楼梦的 108 回处理完，mapper 才能结束。而 reducer 就是整合 mapper 的计算结果。

Map 就是把输入映射成中间结果。reduce 就是在把大家分开做的事情合并到一起。最常见的就是 word count，如下例所示：





因为每一块 mapper 分到的任务是 100G，而内存 16G，不能一次性处理完，所以我们先每 16G 写出一个文件。

Google Percolator

大表对事务的支持比较差，上锁以后效率比较低。我们可以使用 percolator 做处理，允许在大表上做很多小的更新。在执行的过程中，允许我们访问不同的时间戳，然后它去控制这些时间戳。

<i>key</i>	<i>bal:data</i>	<i>bal:lock</i>	<i>bal:write</i>
Bob	6: 5: \$10	6: 5:	6: data @ 5 5:
Joe	6: 5: \$2	6: 5:	6: data @ 5 5:

1. Initial state: Joe's account contains \$2 dollars, Bob's \$10.

balance 列族中存了 3 个列。*lock* 这一列是未提交的状态，用户读到的是提交后的状态 *write*。

Bob	7:\$3 6: 5: \$10	7: I am primary 6: 5:	7: 6: data @ 5 5:
Joe	6: 5: \$2	6: 5:	6: data @ 5 5:

2. The transfer transaction begins by locking Bob's account balance by writing the lock column. This lock is the primary for the transaction. The transaction also writes data at its start timestamp, 7.

Bob	7: \$3 6: 5: \$10	7: I am primary 6: 5:	7: 6: data @ 5 5:
Joe	7: \$9 6: 5: \$2	7: primary @ Bob.bal 6: 5:	7: 6: data @ 5 5:

3. The transaction now locks Joe's account and writes Joe's new balance (again, at the start timestamp). The lock is a secondary for the transaction and contains a reference to the primary lock (stored in row "Bob," column "bal"); in case this lock is stranded due to a crash, a transaction that wishes to clean up the lock needs the location of the primary to synchronize the cleanup.

Bob	8: 7: \$3 6: 5: \$10	8: 7: 6: 5:	8: data @ 7 7: 6: data @ 5 5:
Joe	7: \$9 6: 5: \$2	7: primary @ Bob.bal 6: 5:	7: 6: data @ 5 5:

4. The transaction has now reached the commit point: it erases the primary lock and replaces it with a write record at a new timestamp (called the commit timestamp): 8. The write record contains a pointer to the timestamp where the data is stored. Future readers of the column "bal" in row "Bob" will now see the value \$3.

	8: 7: \$3 6: 5: \$10	8: 7: 6: 5:	8: data @ 7 7: 6: data @ 5 5:
Joe	8: 7: \$9 6: 5:\$2	8: 7: 6: 5:	8: data @ 7 7: 6: data @ 5 5:

5. The transaction completes by adding write records and deleting locks at the secondary cells. In this case, there is only one secondary: Joe.

边缘计算

边缘计算也叫物计算。云是在远端看不见的机房的机器，而物就是在我们的身边。软件学院在 2018 年承担了一个项目，软件学院布置了 128 个摄像头和 51 个微蜂窝基站。摄像头里有计算能力，可以做人脸匹配。这样我们就把计算任务推到了边缘的终端，计算不在云里。连接这些摄像头的就是边缘服务器，再把计算结果汇集到云里。边缘计算考虑的问题就是从终端到云在任何一个节点上都可以去做一些处理，带来的好处就是：

1. 数据在靠近其生成的地方去做处理效率是最高的，省掉了通信开销。
2. 网络是一个移动网络，不保证网络连接是一直可靠的。在数据源头直接处理，就会比我们在不可靠的连接后传回机房处理要可靠。

边缘计算要解决的技术任务就是到底什么样的任务在边缘、边缘服务器、云上执行呢？云边端融合计算，这就是大家想看到的实现。

2021/12/09

我们说到边缘计算，主要解决两个问题

1. 边缘要往云里传，对带宽会造成很大压力，要解决的问题是数据的处理要靠数据的源头。和数据库的问题一样，如果在数据库里写存储过程，它的执行效率比读出来放到程序里更高（靠近数据源更快）。数据应该就近存储，就近处理。如果是很大的数据，他就是分布式处理。
2. 有很多移动的设备，它的连接就不是很可靠。我们就不一定要连到云里去，如果基站能直接帮我们处理掉就可以了。

边缘计算的挑战就是云、边缘服务器（基站）、设备三者联合来解决问题。

例如一百多个摄像头找一个小孩，所有的信息全都传给云来处理，那么它的计算量会非常大。而如果能够让每个摄像头带一点计算资源，每个摄像头只返回找到或者没找到，这样就不需要一个很大的服务器，而且对带宽的占用很少。这就是典型的靠边缘解决的场景。

智能家居也是这样解决的，考虑到它的安全性，比如家用摄像头，所有隐私都上传云来解决是不合理的，考虑只把需要报警的信息上传。

边缘计算的兴起是各种终端都带有计算能力，闲着就浪费了。

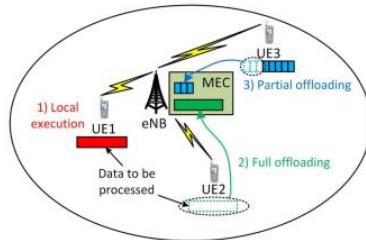
使用边缘计算的注意点：

边缘计算资源是怎么管理的，到底哪些云资源可以用，哪些边缘资源可以用。如果把边缘计算看成一棵树的结构，实际上每个层级都得到了一定的资源。资源怎么管理，服务怎么管理，**workload** 怎么优化就是要考虑的问题。

讲点具体的：

在一个所谓云里面，计算任务需要被迁移掉，就是 **offloading**（卸载或迁移），转移到别人那里去执行。

Offloading 的场景：有多个手机，他们都可以连接到基站。基站既可以理解为一个服务器。这个时候手机拍了张照片想 ps，但是这台手机性能又不够，这个时候如何解决。



一种方法：我们把照片发到基站，让基站来处理返回。这个时候功耗只是传递照片。这就是一种完全的卸载。

另一种：我们手机有足够的电，本地就做掉了。

还有一种：有一定的资源，但不是特别多，因此把照片切一部分自己做，另一部分基站做，这就是部分的卸载。

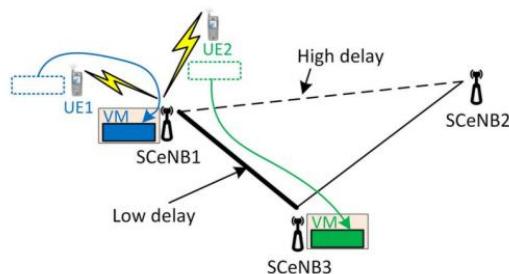
需要根据实际情况来选择。

有可能这个基站也不能搞定，他会接着往上走，连到一个更大的宏基站，这个宏基站还做不了，就可以连到云里，这就是一个树形的结构。于是你就看到其实每一层上都有一个决策机制，能做就做，做不了就往上。

在这种情形下，资源的利用率就会比较高，不会浪费。

对用户来说速度也比较快，不至于把一条带宽占满。

逻辑简单，但是执行起来还有很多因素考虑，迁移会产生问题：



1. 基站和用户都搞不定，于是希望基站卸载过去，但是卸载的时候基站说我也没那么大能力，我只能处理一个用户。比如上面跑了一个执行他的任务的虚拟机，他跑了一个之后，继续说这个手机只能跟我连接，然后你把任务推给我处理不了，就把任务再卸载到其他基站。这时候就看到真正处理的资源在另外一个基站上，然后用户是通过基站在访问它。这个时候基站之间的流量是透明流量基本上是直接传，除了有网络流量之外，虚拟机的

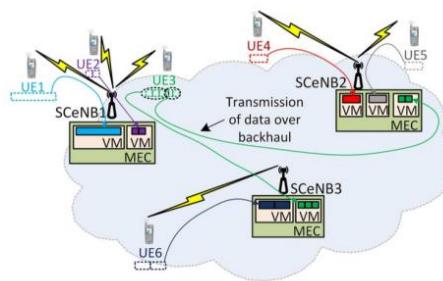
资源在上面这个基站上是不需要分配的。

所以就看到现在有两个用户像连接基站资源，但是基站只够一个虚拟机的时候他就会去跟其他的协商，产生一个选择。因为时延不一样，他选择一个低时延的。

他讲了一种场景，就是在做边缘计算时，你不能把边缘服务器想象成有无穷的资源，小基站之间的迁移以及到宏基站的迁移，复杂性就出来了，你怎么管理这些资源。

2. 现在看到的这有一个特别复杂的场景：

这个绿颜色的用户把他需要的资源切成五个单位，他说我需要虚拟机帮我处理，但是这个虚拟机没有资源，于是要协商，分别找了三个和两个。合起来凑足了五个单位使用。他需要在一定范围内去找时延满足要求的基站协商。因为他们都不能单独满足需求。

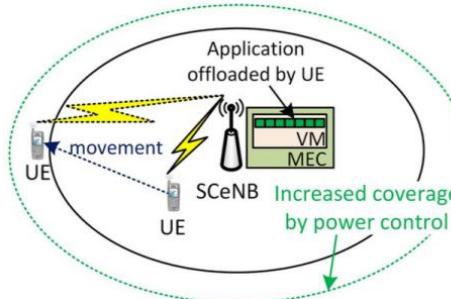


3. 手机无线通信，无线通信有一个最大麻烦，是他们的带宽不是固定的，于是会产生问题：

所以要考虑到用户和基站之间的距离。

这个距离会对两方面产生影响：

一个是带宽，考虑到数据要传输，这些数据传输的时候不能以固定的带宽传输，可能有变化。比如出了这个黑颜色的圈，它的能耗就越高，但是带宽就越低。在这种时候就要考虑速度和能耗。那对于手机来说能耗是一个重要指标，这个时候要注意功率控制。在这种情况下是不是应该切换到另一个基站。

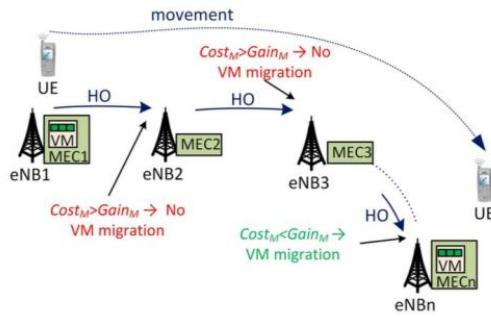


另一个麻烦就是他可能在移动，从一个位置到另一个位置。在第一个基站上连上了，用了虚拟机执行任务。但是可能移动了一段时间发现已经不能和这个基站连接了。但是从运营商的角度来说，要持续为用户统一用户提议。那么在这个虚拟机无法连接的时候，就要把他迁移到他能连接的基站上去，这就是一个虚拟机的迁移问题。

城市里的移动会有一些限制，比如说标了一个目的的，我可以预判你的路径，从而提前安排虚拟机。还有车联网里听广播或者加载导

航，同样来预判路径，提前把数据预取。

因此有人研究路径预判、用户分类、数据预取

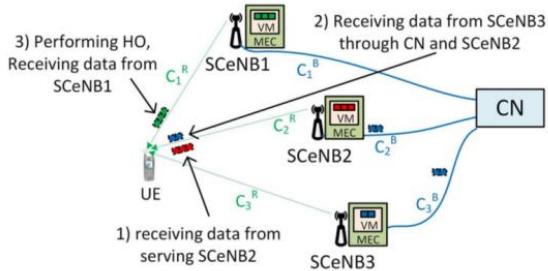


4. 然后就是路径的选择：

看到这个人在移动，那么他到底会选择哪个路径，我怎么去预判给数据？

在城市里判断一条路径是可行的，他是按照道路来的。在野外是不行的。

如果说道路已知的情况下，他需要判断时延和能量功耗。



总的来说，边缘计算是物理计算，他的目的是在周边，利用周围的东西来解决问题。

设想一个场景，比如说有一个彩色打印机，平时没用，想要挣钱，我就把它挂到网上，就像淘宝卖家一样，有人在附近想要打印。简单来说能否做一个计算机软件服务的淘宝。需要精准的定位。边缘计算就是要利用身边的资源，而不是远在天边的机房。

Hadoop

本质上来说 **hadoop** 就是一个可扩展的分布式计算场景，它里面核心就是要跑一个分布式文件系统。然后它会做 **MapReduce**，后台有一个 **YARN** 来调度我们的 **MapReduce**。然后我们跑它需要有一个基础，我们需要编辑一下 **hadoop** 的文件。意思就是说，我们要起一个分布式文件系统，将来要访问它的时候端口在哪里，分布式文件系统写入的路径在哪里；包括说因为分布式文件系统的节点比较容易崩溃，那就是要创建几个副本。

这样配好以后就能跑了，要保证能访问到我们可以使用 **SSH** 去试一下。我们可以通过 **sbin/start-dfs.sh** 开启守护进程，并且在 <http://localhost:9870> 端口去看。



Overview 'localhost:9000' (active)

Started:	Sat Dec 04 14:00:39 +0800 2021
Version:	3.2.2.7a3b09000502578ace278d74264906f07a932
Compiled:	Sun Jan 03 17:26:00 +0800 2021 by hexiaoqiao from branch-3.2.2
Cluster ID:	CID-3046bbb-8004-485e-aa0f-9001ef6d6087f
Block Pool ID:	BP-1664417573-127.0.0.1-638597623149

Summary

Security is off.
Safemode is off.
1 files and directories, 0 blocks (0 replicated blocks, 0 erasure coded block groups) = 1 total filesystem object(s).
Heap Memory used 105.33 MB of 161 MB Heap Memory. Max Heap Memory is 4 GB.
Non Heap Memory used 47.82 MB of 52 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.
Configured Capacity: 931.55 GB
Configured Remote Capacity: 0 B
DFS Used: 4 KB (0%)

当然我们今天跑这个例子不需要把 hadoop 跑起来，它是把 hadoop 当做一个包嵌入。

Pseudo-code

- Word Count Example

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

我们今天的例子主要管的就是 workcount 问题。丢给 reduce 前做了一次 shuffle 排序，然后就是对每个单词出现次数加起来。

真正在写代码的时候，要求 key 和 value 要使用 hadoop 提供的接口，而 reduce 会去拿到 key 和列表之后做一个合并。例子如下：

例子：word count

其实我们就是要写一个 mapper 类和 reducer 类，因为这个例子足够简单，所以我们实现为内部类。

```
public class WordCount {
    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{
        private final static IntWritable one = new IntWritable(1); //Hadoop 指定的数字的类型
        private Text word = new Text(); //Hadoop 指定的 word 的类型
        public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
            //key 就是文件名，value 就是要做 WordCount 任务的文本
            StringTokenizer itr = new StringTokenizer(value.toString());
```

```

//先把 Hadoop 的 Text 类型转换成 Java 的 StringTokenizer 类型，也就是使用空格断开
//这里考虑的比较简单，我们认为文本中没有标点符号
while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
    //把处理结果写到 context 里，之后可以传入后续代码去做处理
}
}

public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable>{
    private IntWritable result = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable> values, Context context
) throws IOException, InterruptedException {
        //此时的 key 就是单词了，所以 Hadoop 这个框架在中间做了些事情
        //此时的 value 是一个集合类（可迭代的对象）
        int sum = 0;
        for (IntWritable val : values) {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum); //要把 int 类设置进 IntWritable 中
            context.write(key, result);
        }
    }
}

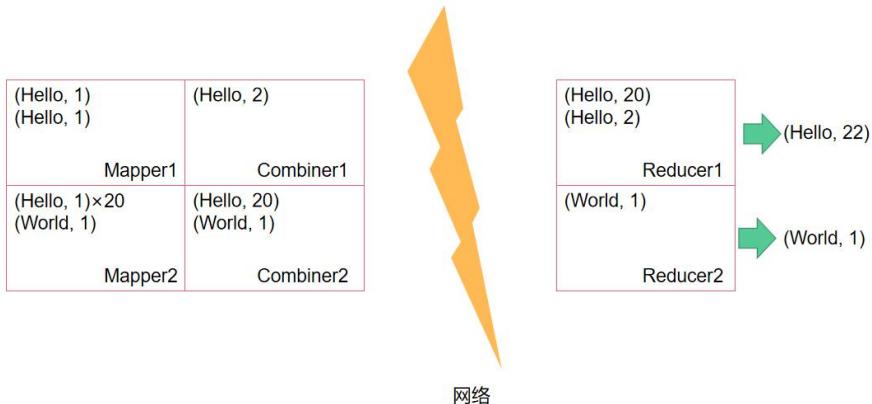
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    conf.set("dfs.defaultFS", "hdfs://hadoop:9000");
    //告诉 Hadoop 它依赖的分布式文件系统在哪
    //Hadoop 就像是云上的一个操作系统，所以真正做 WordCount 叫做一个作业(job)
    Job job = Job.getInstance(conf, "word count");
    //Job 就是一个工厂类，通过传入配置获取
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    //Combiner 在做什么呢？
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

注意到 TokenizerMapper 和 IntSumReducer 我们都用了 static class 修饰。这样的话，我们就不需要创建 WordCount 类，就可以创建 TokenizerMapper 和 IntSumReducer。



Combiner 在做什么呢？它负责处理一个 Mapper 机器上的输出。以下图为例



如果我们直接把 Mapper 的输出传给 Reducer，那么数据显然是冗余的，我们需要通过网络传递 22 个(Hello, 1)，所以 Combiner 做的事情就是把本机的 Mapper 的输出结果再合并一次。然后再传给 Reducer，在上例代码中，因为 Combiner 和 Reducer 的逻辑完全一样，所以我们设置的 Combiner 类和 Reducer 类都是 IntSumReducer.class。

所以 Combiner 可以减少在网络上传输的信息数量，但是我们也需要知道 Combiner 不是在所有场合都适用，比如说我们希望求每个单词在全文单词中占的百分比，此时 Combiner 就不应该在单机上求一次百分比再把结果传递给 Reducer，因为这样的话求出来的数值就没有意义了。

这样我们就可以执行这个代码了，执行完会生成一个 `output` 目录里面有几个文件，其中有一个文件就是执行的结果。

- Lucene1.txt
Hello World Bye World
 - Lucene2.txt
Hello Hadoop Bye Hadoop
 - part-r-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
-
-
- For the given sample input the first map emits:
`<Hello, 1>`
`<World, 1>`
`<Bye, 1>`
`<World, 1>`
 - The second map emits:
`<Hello, 1>`
`<Hadoop, 1>`
`<Goodbye, 1>`
`<Hadoop, 1>`

因为我们目前就 2 个文档，会使用 2 个 mapper 来处理，但是也并不是严格和文档的数量一一对应，如果我们的文档很大的话，它就会把大文档切分成很多小块来处理。

我们把上图右侧产生出来的 mapper 的结果丢进 Combiner 中去处理，得到下图：

- The output of the first map after combiner's processing :


```
< Bye, 1>
< Hello, 1>
< World, 2>
```
- The output of the second map after combiner's processing :


```
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>
```

Combiner 处理完以后就要丢给 Reducer 去处理，并且 Reducer 的数量是可以控制的。如果不设置的话，它就会根据当前计算节点的数量和 CPU 核的数量去决定。

- Thus the output of the reducer is:

```
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>
```

在上例中的 Reducer 是 1 个。

Hadoop Mapper

对于 Mapper 来说，它需要把输入的键值对映射成中间体的键值对表示，每个都是一个独立的工作。所以在 Hadoop 框架中，每个输入的 InputSplit 都会创建一个 map task。

并且 Mapper 的输出是有序的，并且会按照 Reducer 的数量做划分。所以总的划分的数量就等于 Reduce task 的数量。

Mapper 的数量通常会根据总的输入文件的大小而变化，一个正常的水平是单个节点 10-100 个 map 任务，都让对于一些不太需要 CPU 的工作，可以上升到 300 个节点。如果我们尝试设置 blocksize 为 128M，输入 10T 数据，那么我们最终将得到 82000 个节点。

Hadoop Reducer

Reducer 分为三个主要的阶段 shuffle, sort 和 reduce。Shuffle 的前提是排序，只有我们排好序了我们才能说 1950 年之前的数据放到一台 reducer 上；1950 年之后的数据放到另一个 reducer 上。所以 Sort 和 Shuffle 是同时在做的。做完之后才是 reduce 得到我们的结果。我们也可以设计一个自定义的比较器去排序。

- The right number of reduces seems to
 - be 0.95 or 1.75 multiplied by (*<no. of nodes> * <no. of maximum reducers per node>*).
 - With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish.
 - With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.
- Increasing the number of reduces
 - increases the framework overhead,
 - but increases load balancing and lowers the cost of failures.
- The scaling factors above are slightly less than
 - whole numbers to reserve a few reduce slots in the framework for speculative-tasks and failed tasks.

0.95 就是比较保守的策略，也就是我们剩一些资源，一旦有 reducer 挂了我们就可以视同这些新的资源重启 reducer。而 1.75 就是说我们希望一下子启动出来很多 reducer，做完自己的任务以后就去休息。

[InputSplit](#) represents the data to be processed by an individual Mapper.

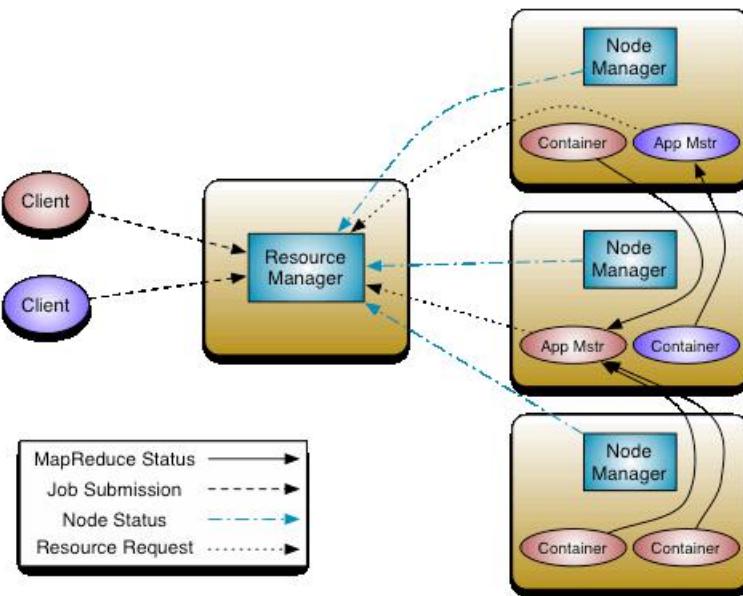
- Typically [InputSplit](#) presents a byte-oriented view of the input, and it is the responsibility of [RecordReader](#) to process and present a record-oriented view.

[FileSplit](#) is the default [InputSplit](#).

- It sets `mapreduce.map.input.file` to the path of the input file for the logical split.

2021/12/13

YARN



每一个应用的 master 自己来管理自己的任务，把代码发到容器里，让它们去跑，100个任务和3个 work 的情况下，不断在其之间分发，并且做心跳检测。把应用的管理和硬件资源的管理切分开。前面还是 MapReduce，后面执行的时候进化到了 YARN 的框架。

这个很像之前的 K8S，根据负载来开新的容器，每个 master 只管之间的作业，把职责区分开。但是不影响我们写 MapReduce 的代码，只是后台的调度机制发生了变化。

Spark

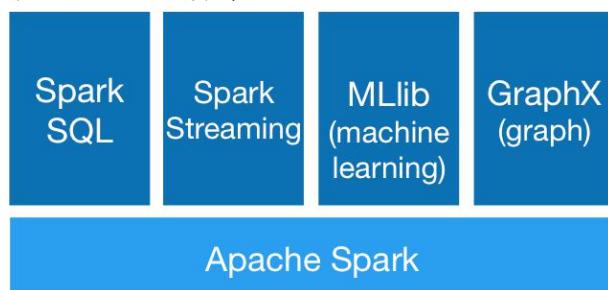
这节课讲讲 SPARK。当我们有一个 Spark 作业想做进行处理的时候，它有一个弹性分布式数据集，对其处理是多台机器上并行处理的。所以它的体系特别复杂，我们只讨论其核心的。

Spark 简单来说就是一个内存计算的产物，在大数据面前内存不够怎么办呢？它就认为这个系统一定是一个分布式的系统，所有操作都要在 n 台机器上同时执行。

Spark 主要是从 Hadoop 来的，底下用到了一些开源的东西，所以最初处理的时候仍然是一个批处理的数据。微观上，它还是批处理，也就是说每 5 秒对新到来的数据以批处理的方式处理一下，宏观地来看就是不断地处理数据。

然后也支持以 SQL 为基础的查找数据的方法，底层不一定要求是结构化的数据。我们可以用类似 SQL 的方式访问到分布式的数据集。如果系统中有各种各样的数据库和非关系型数据库，Spark SQL 提供了一种类似于数据湖的统一管理的方式。只是说底层我们要配置各种各样的 driver。

再往下就是支持一个可伸缩的数据处理操作。数据进来之后是在多台机器上并行操作的，master 会自动地在 worker 上去分布。



图数据并不是 neo4j 只存储图数据，它支持图的切分等操作。Spark 是拿 Scala 来写的，所有操作都在内存里，不需要频繁地读写硬盘。

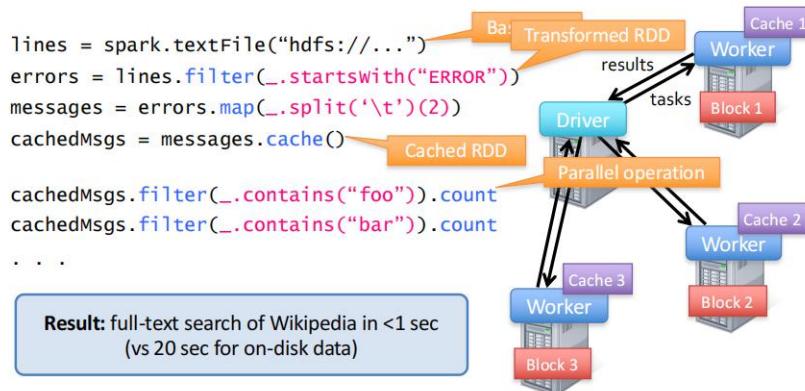
Scala 是函数式编程，基本想法是真正的函数式语言是不存在给变量赋完之后第二次赋值的。Python 中对 `dataframe` 的所有操作，不影响原来的 `dataframe`。

Scala 的效率比较高，所以速度比 Hadoop 快（Hadoop 还涉及了很多磁盘 IO 操作）。他也是一个分布式系统，所以底下可以继承分布式文件系统，比如每一台机器都提供自己所持有的文件。

Example: Log Mining

REIN
Reliable, INtelligent & Scalable Systems

- Load error messages from a log into memory, then interactively search for various patterns



这个 log 文件是在分布式系统中（如 3 台机器），这三台机器构成了一个集群，这个 log 就落在 3 台机器上。Driver 就相当于 MapReduce 中的 master。

`lines = spark.textFile("hdfs://")` #从分布式文件系统中加载文件，一旦加载进来就可以视为一个内存中的弹性数据集。弹性分布数据集可以理解为一个数组或者链表。这些数据物理上虽然不在一起，合起来在逻辑上是一体的。这个 RDD 是在内存里的，RDD 一旦产生，读进来后是不能改变的，所以它是有可能被复用的。

第二步就是在弹性数据集上做操作，`filter/map` 操作进来是一个数据集，出去还是一个数据集。这一类就叫做 `transformation`，变换完的结果是一个新的 RDD。`Lines.filter` 就是对每一行找一找是不是以 `error` 开头的，如果是才保留下。`lines` 是不变的，我们必须把结果赋值为新的 RDD: `errors`。

每一行的格式为 `ERROR reasons`，所以我们拿到 `error` 以后去 `map` 一下，对每一条记录是用 `tab` 键隔开。接下来我们才拿到了真正有效的信息。这样看起来比较浪费，所以内存里只要满了，就把最近最小使用的 RDD 换到硬盘上去。`Message.cache()` 的意思就是把内存锁死，要求不参与垃圾回收等动作，要求必须放进内存里。

真正执行到第四步的时候，程序才真正开始做。如果前三步后面没有 `cache` 这样的动作，提早做完占据内存是没有意义的，所以 Spark 说除非一定要做了，否则会延迟执行。这就是书在加载订单项的时候可以只加载 `order_master`，而 `order_item` 是 lazy 加载的。因为不加载 `order_item` 可以显著节约内存。

所以只有到第四步的时候才会执行前三步。第五步和第六步就是得到了包含特定错误的 Error 信息。我们做所有操作的时候，就是一堆的 `transformation` 和 `action` 执行，而 `count` 是并行执行的，三台机器汇总给 `driver` 后才汇总得到一个最终的结果。

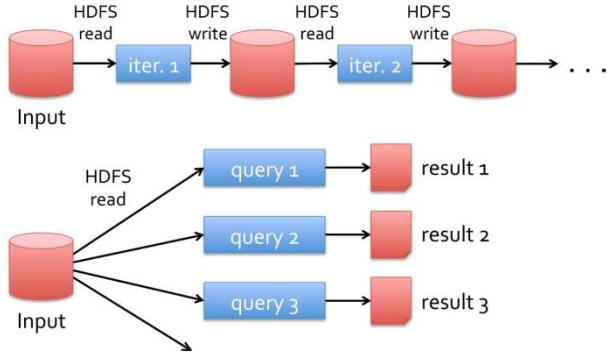


图 6 Hadoop 工作流

Driver 分发任务包含两个动作，分发 jar 包让机器执行作业，以及告诉大家什么时候开始执行动作。它们会把结果回报给 driver。Cache 就会告诉三台机器上把对应的数据锁死在内存中，接下来的 count 操作我们就只需要在内存中对 RDD 做操作就行了。所以文件只被读了一次，后面的操作全部在内存中，文件在内存中可以保留比较长的时间，这是由我们自己保证的。

并且惰性执行机制就是看到了.cache 以后才开始做。Cache 住是真的 cache 吗？也不一定，我们回想到 c++ 中，最快的类型是寄存器类型，它就在 CPU 里。但是 c++ 又会告诉你，这只是程序员的期望，不代表 C++ 一定会保证放在 register 中。这就和这里的 cache 是一样的，spark 会尽力满足这个要求，但是如果发现要求不合理的时候，还是会使用额外的压缩方法来节省空间，或者换到硬盘上。

从这个执行逻辑可以看到速度是很快的，如果使用 Hadoop 等方法往硬盘上写数据需要 20 秒的一个任务，在 spark 上不到 1 秒就可以解决。

在 spark 中，所有操作全部发生在内存里，

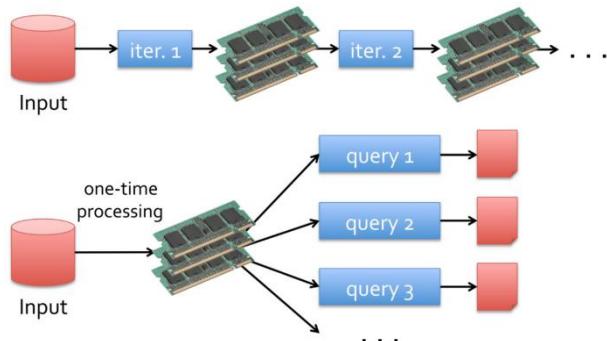
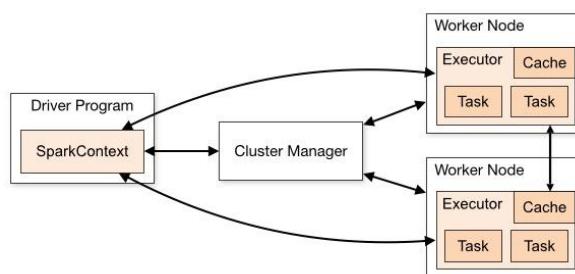


图 7 Spark 工作流

内存和硬盘相比主要是贵和不容易做大，所以我们在做大数据处理的时候，需要大集群插了很多内存条才能支持大任务。



在 Spark 里执行逻辑和 Hadoop 非常像，真正执行操作的是 worker 节点，里面有 executor 节点来负责任务，内存里有一个很大的 cache 来放 RDD。这个集群中有一个 manager 和

`WorkerNode` 去沟通，将来有任务过来应该分配哪个节点。Driver program 相当于 master，我们写的代码都是通过 `spark` 上下文来进行操作的。

我们程序里所有的操作对象都是对于 `SparkContext` 的。

Specifically,

- to run on a cluster, the `SparkContext` can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, Mesos, YARN or Kubernetes), which allocate resources across applications.
- Once connected, Spark acquires *executors* on nodes in the cluster, which are processes that run computations and store data for your application.
- Next, it sends your application code (defined by JAR or Python files passed to `SparkContext`) to the executors.
- Finally, `SparkContext` sends *tasks* to the executors to run.

这个 *cluster manager* 是负责集群上的管理，可以是 YARN，K8S，还可以用 Mesos，或者 Spark 单体的 *cluster manager*。我们还可以使用 zookeeper 或者 OpenStack 等方法。Spark 想集群管理器请求，集群管理器就会返回哪个 `workerNode` 可以做对应的任务。

每个 *application* 都有自己的 *executor* 进程，专属于它执行自己的任务。*executor* 可以多线程地执行待完成的任务。`workNode` 和 `driver` 程序应该是可以通过网络交互的，数据集是分布在多个机器上，做操作的时候是需要同时做的，所以这些 `workernode` 应该物理上靠的比较近比较好。

提交给 `spark` 后，每个节点上都有一个 `ui` 可以去跑。在单机中，我们要先跑一个 `master`，再跑一个 `node`。

Starting a Cluster Manually

- You can start a standalone master server by executing:
- `$./sbin/start-master.sh`
- Once started, the master will print out a `spark://HOST:PORT` URL for itself, which you can use to connect workers to it, or pass as the “`master`” argument to `SparkContext`.
- You can also find this URL on the master’s web UI, which is <http://localhost:8080> by default.

在每一个节点上都要放一个 `spark` 代码，上面一般是给 `master` 跑，而下面这个是给 `worker` 跑。

- Similarly, you can start one or more workers and connect them to the master via:
- `$./sbin/start-worker.sh <master-spark-URL>`
- Once you have started a worker, look at the master’s web UI (<http://localhost:8080> by default). You should see the new node listed there, along with its number of CPUs and memory (minus one gigabyte left for the OS).

Cluster Launch Scripts

- To launch a Spark standalone cluster with the launch scripts, you should create a file called `conf/workers` in your Spark directory, which must contain the **hostnames of all the machines where you intend to start Spark workers, one per line.**
 - If `conf/workers` does **not exist**, the launch scripts defaults to a single machine (`localhost`), which is useful for testing.
 - Once you've set up this file, you can launch or stop your cluster with the following shell scripts, based on Hadoop's deploy scripts, and available in `SPARK_HOME/sbin`:
 - `sbin/start-master.sh` - Starts a master instance on the machine the script is executed on.
 - `sbin/start-workers.sh` - Starts a worker instance on each machine specified in the `conf/workers` file.
 - `sbin/start-worker.sh` - Starts a worker instance on the machine the script is executed on.
 - `sbin/start-all.sh` - Starts both a master and a number of workers as described above.
 - `sbin/stop-master.sh` - Stops the master that was started via the `sbin/start-master.sh` script.
 - `sbin/stop-worker.sh` - Stops all worker instances on the machine the script is executed on.
 - `sbin/stop-workers.sh` - Stops all worker instances on the machines specified in the `conf/workers` file.
 - `sbin/stop-all.sh` - Stops both the master and the workers as described above.

使用方法

1. 使用 shell 访问
 2. 使用 scala/Python 访问
 3. 使用 sparkSQL 访问。
 - Start it by running the following in the Spark directory:
 - **Scala:** \$./bin/spark-shell
 - **Python:** \$./bin/pyspark

Spark's primary abstraction is a distributed collection of items called a **Dataset**.

- Datasets can be created from Hadoop **InputFormats** (such as HDFS files) or by transforming **other Datasets**.
 - Let's make a new DataFrame from the text of the README file in the Spark source directory:

```
- scala> val textFile = spark.read.textFile("README.md")
- textFile: org.apache.spark.sql.Dataset[String] = [value: string]
-
- scala> textFile.count()
- res0: Long = 109
-
- scala> textFile.first()
- res1: String = # Apache Spark
-
- scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
- linesWithSpark: org.apache.spark.sql.Dataset[String] = [value: string]
-
- scala> textFile.filter(line => line.contains("Spark")).count() // How many lines contain
"Spark"?
- res2: Long = 19
-
- scala> textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
- res3: Int = 16
```

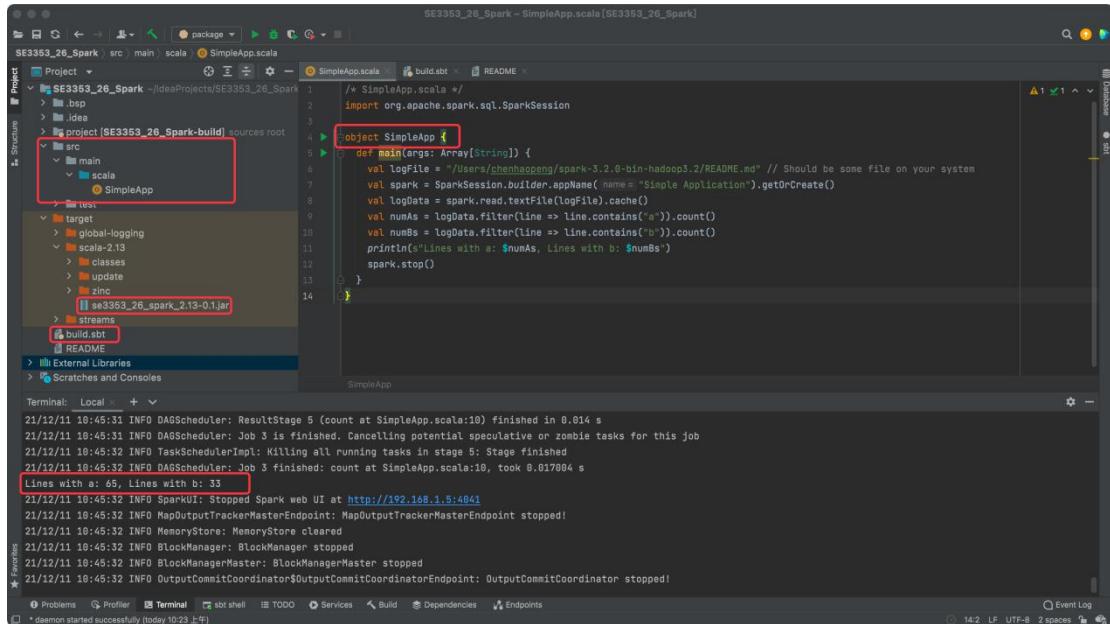
这是一个 scala 控制台，我们需要用 scala 语言来写。注意 scala 语言也是动态类型推断的。这种动态类型语言，在编译的时候是不知道类型的，从内存使用上是不需要预分配的，比较节约空间。

Caching

- Spark also supports pulling data sets into a **cluster-wide in-memory cache**.
- This is very useful when data is accessed repeatedly, such as when querying a small “hot” dataset or when running an iterative algorithm like PageRank.
- As a simple example, let’s mark our `linesWithSpark` dataset to be cached:
- `scala> linesWithSpark.cache()`
- `res6: linesWithSpark.type = [value: string]`
- `scala> linesWithSpark.count()`
- `res7: Long = 19`
- `scala> linesWithSpark.count()`
- `res8: Long = 19`

注意这个`.cache` 是跨集群的 **cache**，三个机器各自锁定自己的一部分。

Scala 工程分了好几种，我们需要创建 **sbt** 工程。



我们就是加载路径下的 `readme` 文件。然后通过这个对象创建哪一个应用。

Our application depends on the Spark API, so we’ll also include an sbt configuration file, **build.sbt**,

- which explains that Spark is a **dependency**.
- This file also adds a repository that Spark depends on:

```
name := "SE3353_26_Spark"  
version := "0.1"  
scalaVersion := "2.13.7"  
  
libraryDependencies += "org.apache.spark" %% "spark-sql" % "3.2.0"
```

并且我们还需要如上的配置文件。

```

SE3353_26_SparkSample - SimpleApp.py
SimpleApp.py  README.md
1  """SimpleApp.py"""
2  from pyspark.sql import SparkSession
3
4  logFile = "/Users/chenhaopeng/spark-3.2.0-bin-hadoop3.2/README.md" # Should be some file on your system
5  spark = SparkSession.builder.appName("SimpleApp").getOrCreate()
6  logData = spark.read.text(logFile).cache()
7
8  numAs = logData.filter(logData.value.contains('a')).count()
9  numBs = logData.filter(logData.value.contains('b')).count()
10
11  print("Lines with a: %i, lines with b: %i" % (numAs, numBs))
12
13  spark.stop()

```

Run: SimpleApp

```

WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
21/12/11 23:49:27 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Lines with a: 65, lines with b: 33

```

Process finished with exit code 0

图 8 python 版本

```
$ /Users/chenhaopeng/spark-3.2.0-bin-hadoop3.2/bin/spark-submit \
--class "SimpleApp" \
--master local \
SimpleApp.py
```

执行命令：

一定要记住的是不能在 ide 中直接跑这个程序，需要通过 `spark-submit` 方法。

RDD 在处理的时候，会并行地处理任务。RDD 要么是从硬盘上加载数据，要么是从线程的 RDD 里通过 `filter/map` 产生新的 RDD。如果我们在内存中定义了一个集合类，我们需要转为 RDD，我们就需要 `sc.parallelize()` 方法。

在 RDD 上能够执行的操作分为 `transformation`（生成一个新的 RDD 数据集）和 `action`（只返回一个值）。本质上它想说的是 `transformation` 是可以在本地做的，比如我们的 RDD 放在了是台机器上，我们只需要把 RDD 的十块在本地处理完即可，并且新生成的 RDD 也是被分成了十块的，每一块是依赖于前面这一块的。而 `action` 是要返回一个值给最终的 `driver program`，涉及到把十块的信息都要拿到合在一起产生一个值返回给 `driver program`，一旦涉及到多个 `partition`，这件事情的计算开销就比较大。

举个例子，`map` 就是最典型的 `transformation`，而 `reduce` 就是一个 `action`。所有的 `transformation` 都是 `lazy` 的，没有必要立即把内存全部占满，因为 `transformation` 操作都是在本地做，没有必要和其他机器做交互，我们希望尽量节约内存一直到必须要和其他机器交互的时候才去做。

所有的 RDD 都可以做持久化，所谓的持久化不是写到硬盘上，而是一直把 RDD 放在内存里。这样我们每次在访问的时候，我们可以把中间的计算结果留下，以免我们还需要重新计算一下。

RDD Transformation

第一个是 `map`，它会去执行 `lambda` 表达式，这里指的就是在 RDD 上，对里面每一个值映射成其值的平方。

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
```

`flatMap` 稍微复杂一点，我们有两个元素组成一个数组。我们用 `lambda` 表达式用空格断开，这样就可以得到单个单词的元素。也就是原先的每个元素可以映射到 0 个、1 个或多个元素中。

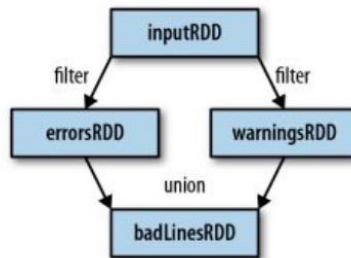
```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
```

`filter` 就是得到符合条件的（包含 `error` 或 `warning`）的 `RDD`:

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
```

`union` 就是对两个 `RDD` 做合并，它的结果要赋值给一个新的 `RDD`:

```
badLinesRDD = errorsRDD.union(warningsRDD)
```



For `RDD {1, 2, 3, 3}`

Transformations	Meaning	Example	Result
<code>map(func)</code>	Return a new distributed dataset formed by passing each element of the source through a function <code>func</code> .	<code>rdd.map(x => x + 1)</code>	{2, 3, 4, 4}
<code>flatMap(func)</code>	Similar to <code>map</code> , but each input item can be mapped to 0 or more output items (so <code>func</code> should return a <code>Seq</code> rather than a single item).	<code>rdd.flatMap(x => x.to(3))</code>	{1, 2, 3, 2, 3, 3, 3}
<code>filter(func)</code>	Return a new dataset formed by selecting those elements of the source on which <code>func</code> returns true.	<code>rdd.filter(x => x != 1)</code>	{2, 3, 3}
<code>distinct()</code>	Return a new dataset that contains the distinct elements of the source dataset.	<code>rdd.distinct()</code>	{1, 2, 3}
<code>union(otherDataset)</code>	Return a new dataset that contains the union of the elements in the source dataset and the argument.	<code>rdd.union(other)</code>	{1, 2, 3, 3, 4, 5}
<code>intersection(otherDataset)</code>	Return a new <code>RDD</code> that contains the intersection of elements in the source dataset and the argument.	<code>rdd.intersection(other)</code>	{3}
<code>cartesian(otherDataset)</code>	When called on datasets of types <code>T</code> and <code>U</code> , returns a dataset of <code>(T, U)</code> pairs (all pairs of elements).	<code>rdd.cartesian(other)</code>	{(1, 3), (1, 4), ..., (3, 5)}

这些都是 `transformation`，结果都是 `RDD`，没有改变原先 `RDD` 的值，并且会生成新的 `RDD`。因为 `RDD` 是分布式存储的，所以 `transformation` 会在本地做，拿自己持有的那一块去本地做。

RDD Action

第一个 `action` 就是 `reduce`，这里的逻辑比较绕，说的是如果集合中有 2 个元素，那么我们就拿出来用它们的和来替换它们，直到最后只有一个元素位置，所以下例的结果就是一个和。

```
rdd = sc.parallelize([1, 2, 3, 4])
sum = rdd.reduce(lambda x, y: x + y)
```

而 count 就是数 RDD 集合的个数。

```
print "Input had" + badLinesRDD.count() + "concerning lines"
```

但是这里就出现了一个问题，无论是求和还是计数，RDD 可能分布在多个机器上，我们最终是要访问多个机器上的多个块，最后合并成一个值，一定有网络开销，要汇集给 driver，driver 最后再统计一下。一旦碰到了一个 action，那么这个 action 之前的 transformation 就要全部做一遍。

For RDD {1, 2, 3, 3}

Actions	Meaning	Example	Result
<code>collect()</code>	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.	<code>rdd.collect()</code>	{1, 2, 3, 3}
<code>count()</code>	Return the number of elements in the dataset.	<code>rdd.count()</code>	4
<code>reduce(func)</code>	Aggregate the elements of the dataset using a function <code>func</code> (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.	<code>rdd.reduce((x,y) => x+y)</code>	9

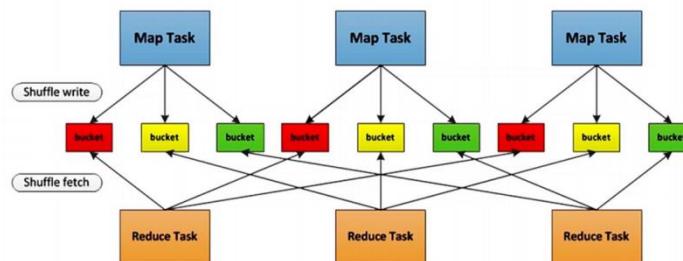
所以 Spark 不脱离于 MapReduce 的框架，就是多个逻辑堆叠到一起。

Spark 的基本流程

- 从外部数据创建一些作为输入的 RDD
- 使用类似 filter 之类的变换(Transformations)来定义出新的 RDD
- 要求 Spark 对需要重用的任何中间 RDD 进行 persist
- 启用类似 count 之类动作(Actions)进行并行计算

Certain operations within Spark trigger an event known as the **shuffle**.

- The shuffle is Spark's mechanism for **re-distributing data** so that it's grouped differently across partitions. This typically involves copying data **across** executors and machines, making the shuffle a complex and costly operation.
- The Shuffle is an **expensive** operation since it involves disk I/O, data serialization, and network I/O.

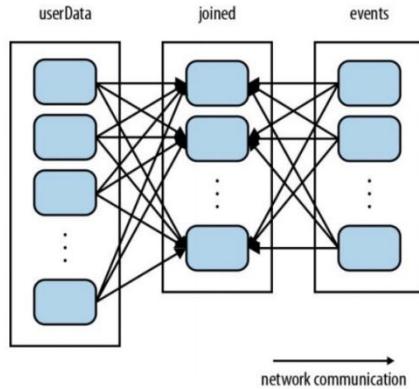


它在执行的过程中就是既有 transformation 和 action 的操作，在这中间，一旦涉及到了 reduce 这种 action，之所以耗时就是因为它涉及到了多台机器。每个 reduce 都要从多台机器上去拿数据。期间也涉及到 shuffle 的操作，对数据进行重分布，所以它是耗时的。

我们讲一个简单例子：

假设程序在内存中持有一个非常大的用户信息表(UserID, UserInfo)，其中 UserInfo 包含用户订阅的主题列表；还有一个很小的表，记录过去 5 分钟里在网页上点击过链接的事件，键值对为(UserID, LinkInfo)。程序周期性地将这两个表合并、查询。做完 join 操作以后就知道哪一些类型的用户喜欢看怎么样的文章。我们这里考察怎么把这两个表做 join 操作再做查

询。



左边是用户数据，右边是点击事件，中间是 join 出来的结果。如果说这个数据我们不去做人为的分区，比如我们原先对 `userData` 和 `events` 都按照 `userId` 做排序然后再分区，这是我们人在控制分区，效果会比较好。

假设我们没做人为地控制分区，带来的问题是，中间一个 `join` 块，我们拿到一个 `event` 数据，我们也不知道对应的用户在 `userData` 的哪里，我们就要每一个 `userData` 块都问一问有没有这个数据。这些黑线全部都是网络开销。

但是如果我们人为控制一下分区，拿用户的 ID 做哈希得到 100 个分区，这样中间的 `join` 拿到一个 `event` 数据中的 `userId` 后，可以用相同的哈希逻辑找到对应的 `userData` 块。这时候，我们再做 `join`，对于每个 `user` 数据，我们都知道在哪里，我们就省去了很多网络开销。所以我们需要有分区。

partitionBy - 指定分区

- 指定分区
- Spark 知道其为哈希分区，在执行 `join` 时会利用这一信息

```
val sc = new SparkContext(...)  
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")  
.partitionBy(new HashPartitioner(100)) // Create 100 partitions  
.persist()
```

宽依赖和窄依赖

`transformation` 不是马上执行的，它到底应该什么时候执行？我们就要知道窄依赖和宽依赖的区别。

Narrow Dependencies

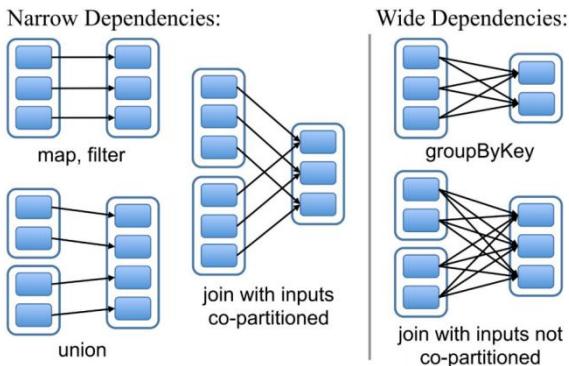
- 窄依赖
- 父RDD的每个分区只被子RDD的一个分区所使用
- map, union, ...

Wide Dependencies

- 宽依赖
- 父RDD的每个分区都可能被多个子RDD分区所使用
- Shuffle
- Join with inputs not co-partitioned, ...

我们现在有一个 RDD 要生成一个新的 RDD。如果生成的 RDD 只和它的对应分区的父 RDD 相关，这就是窄依赖，比如 Map、Union 这种 transformation 都是窄依赖。

而宽依赖的代表就是 reduce，要产生一个 reduce 我们必须拿父 RDD 的所有 partition 过来。宽依赖有可能带来很多的网络开销。



宽依赖意味着，如果后续节点崩了，我们需要前面的父 RDD 所有分区才能恢复它，而窄依赖只需要单个分区就可以恢复了。

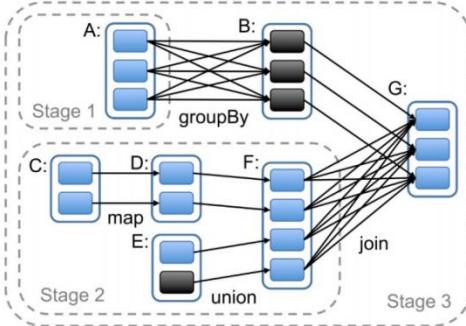
Narrow Dependencies VS. Wide Dependencies

- 宽依赖往往意味着Shuffle操作，可能涉及多个节点的数据传输
- 当RDD分区丢失时，Spark会对数据进行重算
- 窄依赖只需计算丢失RDD的父分区，不同节点间可以并行计算，能更有效地进行节点的恢复
- 宽依赖中，重算的子RDD分区往往来自多个父RDD分区，其中只有一部分数据用于恢复，造成了不必要的冗余，甚至需要整体重新计算

所以在真正计算的时候，如下图所示：

Stage

- 每个阶段stage内部尽可能多地包含一组具有窄依赖关系的transformations操作，以便将它们流水线并行化（pipeline）
- 边界有两种情况：一是宽依赖上的Shuffle操作；二是已缓存分区



注意每个 stage 内部都是窄依赖，stage 的意思就是整个可以先不执行，到 stage 结束了必须要做 action 的时候，再回推回来把 stage 内部的操作都执行掉。也就是当代码执行到 B 和 F 要 join 的时候，才开始把 stage2 中的 CDEF 给做了。划分成 stage 的好处就是，每一个 stage 我们都执行一遍，如果 G 崩了，前面的 stage 我们都执行过了，可以把结果 cache 住，也就是把 B 和 F cache 住。目的就是执行地快且节省内存，所以我们在写代码的时候，尽可能让 action 少一点。

2021/12/16

Dependencies

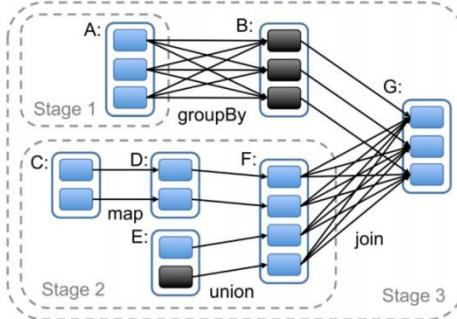
- Narrow Dependencies
 - 窄依赖
 - 父RDD的每个分区只被子RDD的一个分区所使用
 - map, union, ...
- Wide Dependencies
 - 宽依赖
 - 父RDD的每个分区都可能被多个子RDD分区所使用
 - Shuffle
 - Join with inputs not co-partitioned, ...

上节课讲到宽依赖和窄依赖。也就是做计算的时候后面的分区会依赖于前面的多个分区。本质上就是这里面存在一个类似 shuffle 的动作。宽依赖就因为着在一个分布式的环境中，这部分的依赖会比较大。

基于这样的考虑，我们把一个 RDD 通过 action 产生一个新的 RDD，然后通过 transformation 产生一个新的 RDD，这就生成了一个有向无环图。

Stage

- 每个阶段stage内部尽可能多地包含一组具有窄依赖关系的transformations操作，以便将它们流水线并行化（pipeline）
- 边界有两种情况：一是宽依赖上的Shuffle操作；二是已缓存分区



因为 transformation 是 lazy 执行的，所以在计算一个 RDD 的时候，会根据产生 RDD 的有向无环图来推算需要哪些 RDD 来参与计算，然后按照宽依赖和窄依赖的方式去生成 stage，在一个 stage 中是以流水线的方式一次性做完的。

在这个图里面，黑色指的就是 RDD 已经被 cache 了，也就是不能因为内存不足了而销毁掉，而是要求一直存在。一旦 RDD 被持久化了，它就已经在那了，所以 stage1 就没有必要再去计算了，因为 stage1 的目的是生成 B，而 B 已经被持久化了。在这种情况下，被调用后就先执行 stage2，再执行 stage3，这就是之前提到的 transformation 的特性。

为什么不需要占内存呢？因为 transformation 都是窄依赖，所以只有当用到的时候才去一次性算出来。

把 RDD cache 住是什么意思呢？我们可以通过`.cache()`或者`.persist()`方法。我们知道它底层的实现是 java，也就是在做 GC（垃圾回收）的时候，我们需要去看一下 RDD 是否已经过时了。如果 cache 了那就是告诉 GC，对应的 RDD 我们希望一直放在内存里。

- In addition, each persisted RDD can be stored using a different *storage level*.
- The full set of storage levels is:

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer , but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory . This requires off-heap memory to be enabled.

OFF_HEAP 无论是 java 写的还是 scala 写的代码，最终是要跑在 JVM 中的。里面基本上分为保留区、代码区、全局变量区。从上往下是栈，从下往上是堆。每次调用的方法我们产生一个 frame 放在栈上，而堆就是我们使用 new 对象产生的东西。JVM 把堆又分成了几代（generation）。因为它在堆里创建了对象之后，如果不用了，我们希望尽快删掉。在所有对象这里它维护了一个引用计数器，如果为 0 了就回收掉。所以每次会把老的一代的引用数为 0 的删除，放到新一代中。这个垃圾回收的效率必须要高，也就是这个堆要搞得小一点，

但是堆小了 RDD 还怎么放呢？

如果我们在堆里放 RDD，是 JVM 自己管理的，而 OS 不能干预。所以 JAVA 虚拟机就分成了 **on-heap**（JVM 自己管理的比较小的堆）和 **off-heap**（OS 可以直接帮我们管理，甚至可以让其他进程通过这个 OS 管理来共享访问）。Off-heap 可以搞得比较大一点。

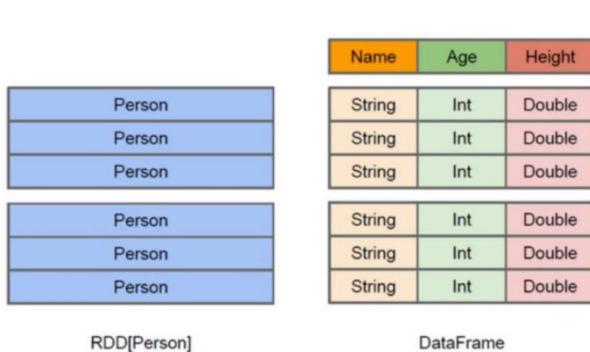
可以看到 storage 的选择已经比较多了。如果 RDD 要持久化到硬盘上，除了我们要把它序列化之外，还有什么别的方法可以让它比较小一点呢？它也开放了数据压缩的接口。

Removing Data

- Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (**LRU**) fashion.
 - If you would like to manually remove an RDD instead of waiting for it to fall out of the cache, use the **RDD.unpersist()** method.
 - Note that this method **does not block** by default.
 - To block until resources are freed, specify **blocking=true** when calling this method.

Spark SQL

它是专门用来处理结构化数据的，我们可以理解为就是一个内存数据库。我们在操作的时候可以通过 SQL 的接口，也可以通过 dataset 的接口。类似于 JPA/JDBC 直接访问数据库。这个数据集是这样的，是一个结构化的表，但是它在内存中存储的时候默认是按列存而不是按行存。



因为列存的好处就是 `age` 可能不是存原始的数据，比如第一个值存原始值，后面全部存差值，这样会压缩内存占用。

```
val df = spark.read.json("examples/src/main/resources/people.json")
// Displays the content of the DataFrame to stdout
df.show()
// +---+-----+
// | age|    name|
// +---+-----+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +---+-----+
```

在 spark 里面提供了很多个适配器，它可以读很多个不同格式的文件，比如说 json, csv。一旦通过这条语句读出来，读进来就是一个 `dataframe` 对象了。

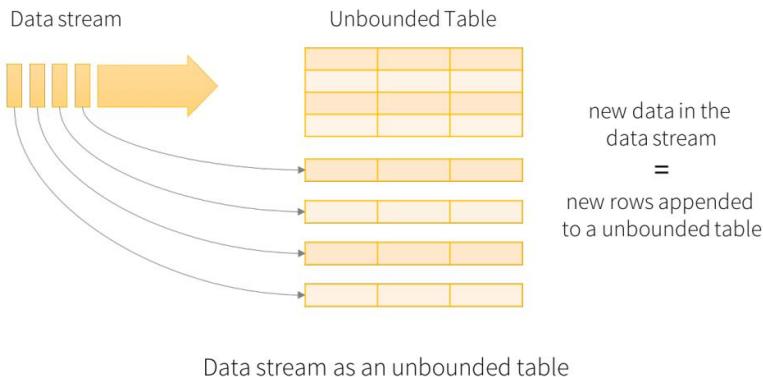
注意我们的操作都是作用在多台机器上去做一个操作，所以还是比较复杂的。我们也可

以抽取我们想用的东西创建一个临时视图。如果我们要创建跨 session 的 view 的话，需要使用 `df.createGlobalTempView("people")`。

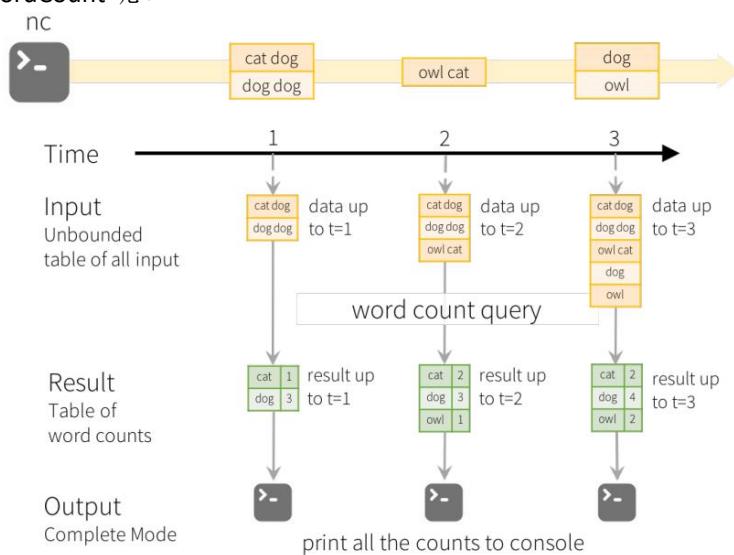
除了最重要的这两个东西以外，还有结构化的流式数据的处理。

- Structured Streaming

– a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.



也就是数据不断地过来，看起来好像是不断地在这个无限的表中做 `append`。此时我们改怎么样做 `WordCount` 呢？



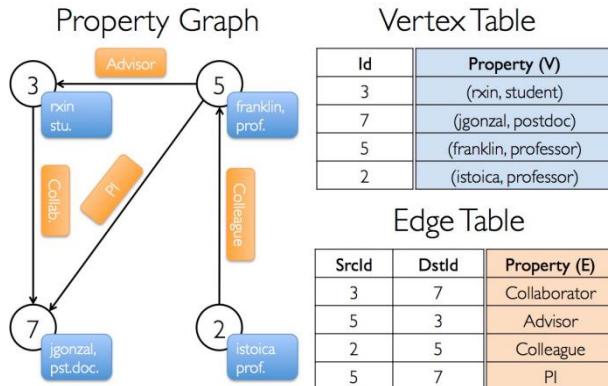
Model of the Quick Example

上一个时间点到下一个时间点之间前端都会收集到一些 word。它就把这段时间内收集到的数据做了 `WordCount`。在时间点 2 的时候，因为我们的数据不断在追加到表中，就可以再做一次 `WordCount`。



前端不断监听消息队列，然后以流式数据推送到 Spark 中，然后对数据进行批处理。

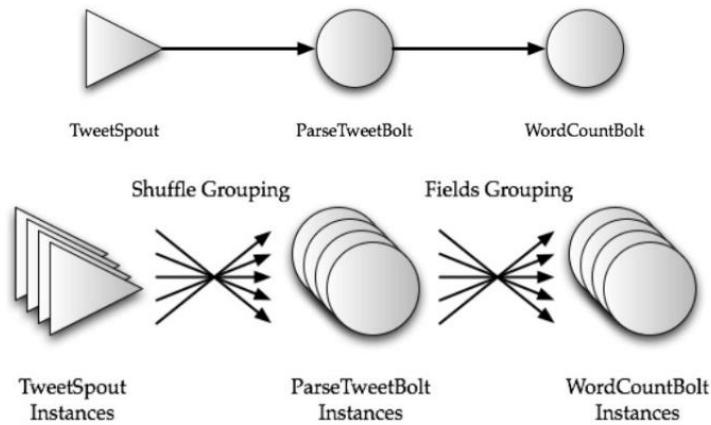
我们这学期讲了很多数据库，比如 neo4j，在 spark 中也专门有一个 graphX，图可能也非常大，所以也借助了 RDD 的方式来分布式地存储这个图。分为节点表和边表，并且节点和边上都有属性。



目标就是要保持存储图的语义是一致的，所以存储可能存在一些冗余。graphX 提供了对图的存储和访问，但是真正要做什么，还需要算法做支持，所以我们还需要 Machine Learning Library(MLib)，也被包含在了 Spark 中。

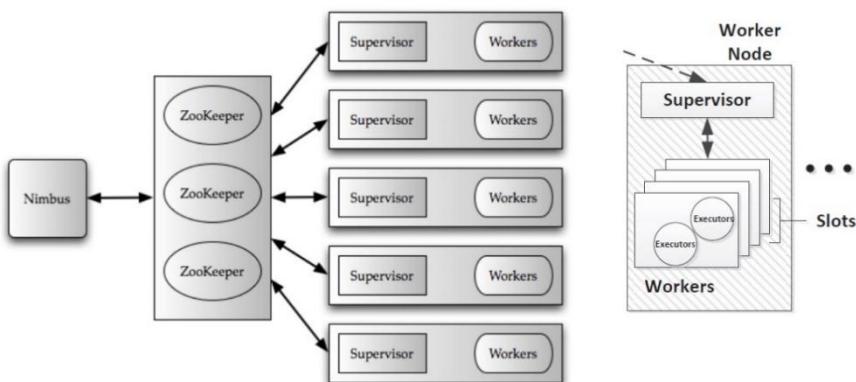
Strom

我们讲讲 storm，它是专门用来做流式处理的。我们来拿 storm 来讲一讲。它把流式数据想象成水龙头。产生数据的程序就是水龙头。后台就有一些处理数据的程序叫做 bolt。这些 bolt 全部可以复用。所以我们从一个 spout 产生一个结果，通过一系列的 bolt（大小写转换、敏感词处理、图片尺寸压缩），这样一些流程最终合在一起就成为了一个拓扑结构，构成了一个有向无环图。所以和 hadoop 的 MapReduce 的区别可能就是这个程序一直在运行。



每一个节点每一秒都可以处理很多元组，这些事情一个机器是做不了的，我们需要一个 **zookeeper** 来管理这些。

数据可以从队列和数据库中来，所以可以做集成。真正在运行时，我们会在集群中设置不同的 Spout 和 Bolt 的实例数量。比如说第一个 Spout 是根据地区划分的，那么我们可能就要根据语言做 Shuffle，分配到不同的语言 ParseTweetBolt 中。



它里面跑的这些东西都叫做元组。

来看一个具体的例子，元组是啥样子的，我们对一个输入的数据要产生一个输出，这个输出从哪里来，在不同的 spout 和 Bolt 之间通信，我们 **topologies** 的那条边就是他们的通信，在这条边上流的就是 **Collector**。

当我们数据过来的时候，storm 会调这个 bolt 的 **prepare** 数据，把数据传给你，拿到之后就可以执行。

如果我有这个输入给的 tuple，我在 **collector** 这里给他放进去东西，是个 **emit**，键就是这个输入，值是它的平方和立方构成的元组。这样键值对就放到了 **collector** 里面。

激发之后就会连到下一个 Bolt，通过 **collector** 数据就传过去了。

为了规范数据格式，所以这里有一个 **declare**，包含两个域

```
@Override public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("double", "triple"));
}
```

创建好之后，一旦有数据进来，流出的就是这样的数据。

- Let's look at the ExclamationTopology definition from storm-starter:

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("words", new TestWordSpout(), 10);
builder.setBolt("exclaim1", new ExclamationBolt(), 3)
    .shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2)
    .shuffleGrouping("exclaim1");
```

- This topology contains a spout and two bolts.
 - The spout emits words, and each bolt appends the string "!!!" to its input.
 - The nodes are arranged in a line: the spout emits to the first bolt which then emits to the second bolt.
 - If the spout emits the tuples ["bob"] and ["john"], then the second bolt will emit the words ["bob!!!!!!"] and ["john!!!!!!"].

这个 topology 有一个 spout，是 testwordspout 实例化出来的，数量是 10 个
把 10 个发到 3 个里面，就一定会有 shuffle 的动作。

下面又有 Bolt，接的就是前面的 exclamationbolt，他是一个串行结构获取。
进去一个单词，出来的时候后面加了六个感叹号
至于具体的细节如 shuffle 怎么实现等，都是 storm 这个框架帮你做的。

```
builder.setBolt("exclaim2", new ExclamationBolt(), 5)
    .shuffleGrouping("words")
    .shuffleGrouping("exclaim1");
```

也可以像这样让 exclaim2 直接接入来改写拓扑。

- Spouts are responsible for emitting new messages into the topology.
 - `TestWordSpout` in this topology emits a random word from the list ["nathan", "mike", "jackson", "golda", "bertels"] as a 1-tuple every 100ms.
 - The implementation of `nextTuple()` in `TestWordSpout` looks like this:

```
public void nextTuple() {
    Utils.sleep(100);
    final String[] words =
        new String[] {"nathan", "mike", "jackson", "golda", "bertels"};
    final Random rand = new Random();
    final String word = words[rand.nextInt(words.length)];
    _collector.emit(new Values(word));
}
```

那么 `TestWordSpout` 是怎样来产生数据的，它里面有一个 `tuple` 方法来产生输出。

他有一个单词的数组，每次随机选中一个。

他对 `collector emit` 出去一个值，注意不能直接传 `word`，要包装在 `values` 里。

这个 spout 就是每睡 0.1s，发送一个 word

所以 spout 也没有特别之处，关键就是让他的 `next` 方法能 `emit` 出来东西。

- **ExclamationBolt** appends the string "!!!!" to its input.

```
public static class ExclamationBolt implements IRichBolt {
    OutputCollector _collector;

    @Override public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        _collector = collector;
    }

    @Override public void execute(Tuple tuple) {
        _collector.emit(tuple, new Values(tuple.getString(0) + "!!!!"));
        _collector.ack(tuple);
    }

    @Override public void cleanup() {}

    @Override public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
    @Override public Map<String, Object> getComponentConfiguration() {
        return null;
    }
}
```

这个 Bolt 里主要有几个方法，一个是在创建时 `prepare` 把 `collector` 丢出来。
他的 `execute` 会从发过来的 `tuple` 拿内容然后处理并发出。

`Ack` 就是发一个应答，表示确实接收到了输入。

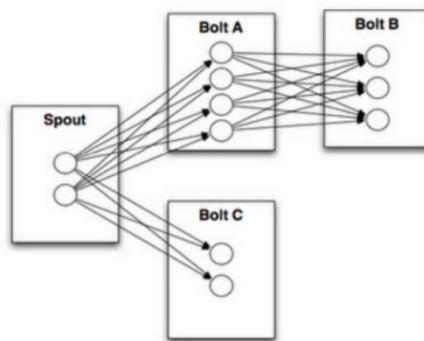
`Cleanup` 就是析构函数。

`Declare` 申明数据的结构。

没写东西的方法也可以直接省略掉。

- Storm has two modes of operation:
 - local mode and distributed mode.
 - In local mode, Storm executes completely in a process by simulating worker nodes with threads.
Local mode is useful for testing and development of topologies.
- Common configurations for local mode
 - **Config.TOPOLOGY_MAX_TASK_PARALLELISM**: This config puts a ceiling on the number of threads spawned for a single component.
 - **Config.TOPOLOGY_DEBUG**: When this is set to true, Storm will log a message every time a tuple is emitted from any spout or bolt.
 - To launch your topology in local mode you could run
– `$ storm local topology.jar <MY_MAIN_CLASS> -c topology.debug=true`

然后要跑起来。打成一个 jar 包，发给 storm 让他去跑。

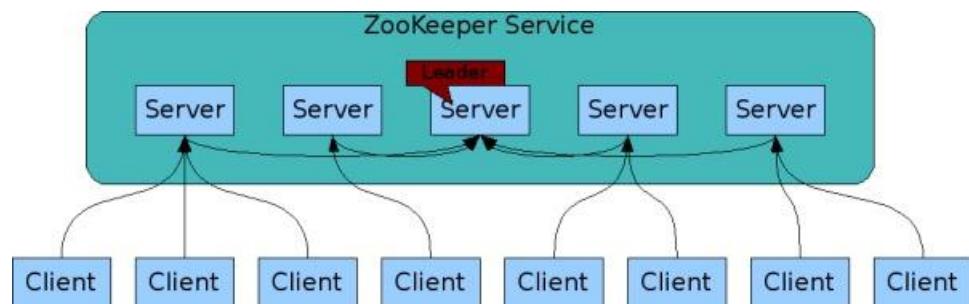
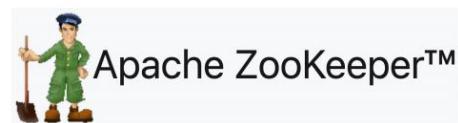


ZooKeeper

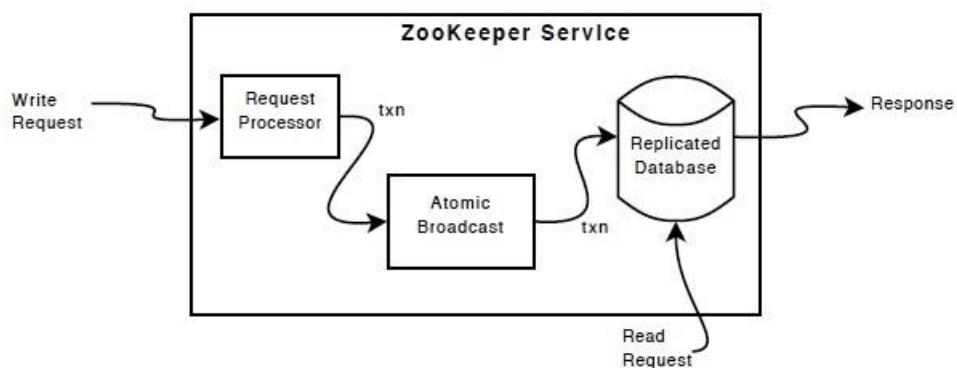
Apache ZooKeeper

REin
Reliable, Intelligent & Scalable Systems

- What is ZooKeeper?
 - ZooKeeper is a high-performance coordination service for **distributed applications**.
 - It exposes common services - such as naming, configuration management, synchronization, and group services - in a **simple interface** so you don't have to write them from scratch.
 - You can use it off-the-shelf to implement consensus, group management, leader election, and presence protocols. And you can build on it for your own, specific needs.
- <https://zookeeper.apache.org/>



如果一个应用会跑 3 个实例，它一定会分布在不同的机器上去跑。



如果操作是一个读请求，那么就直接去读；如果是一个写请求，要广播给其他节点让其他节点不要再写了。

Zookeeper 下载下来有一个配置文件 conf/zoo.cfg，需要配置一下三个值。

To start ZooKeeper you need a configuration file. Here is a sample, create it in [conf/zoo.cfg](#):

```
tickTime=2000  
dataDir=/var/zookeeper  
clientPort=2181
```

Here are the meanings for each of the fields:

- **tickTime**: the basic time unit in milliseconds used by ZooKeeper. It is used to do heartbeats and the minimum session timeout will be twice the tickTime.
- **dataDir**: the location to store the in-memory database snapshots and, unless specified otherwise, the transaction log of updates to the database.
- **clientPort**: the port to listen for client connections

Now that you created the configuration file, you can start ZooKeeper:

```
bin/zkServer.sh start
```

设置集群中的心跳间隔，如果在 2000ms 中没有收到心跳，那么需要把集群对应节点中的数据做迁移。

所以在跑 storm 的时候要跑 3 个命令行的窗口。



- The Storm release contains a file at [conf/storm.yaml](#)

```
storm.zookeeper.servers:  
  - "127.0.0.1"  
nimbus.seeds: [" 127.0.0.1 "]  
storm.local.dir: $STORM_HOME/storm-local  
supervisor.slots.ports:  
  - 6700  
  - 6701  
  - 6702  
  - 6703
```
- Here's how to run the Storm daemons:
 - **Nimbus**: Run the command [bin/storm nimbus](#) under supervision on the master machine.
 - **Supervisor**: Run the command [bin/storm supervisor](#) under supervision on each worker machine.

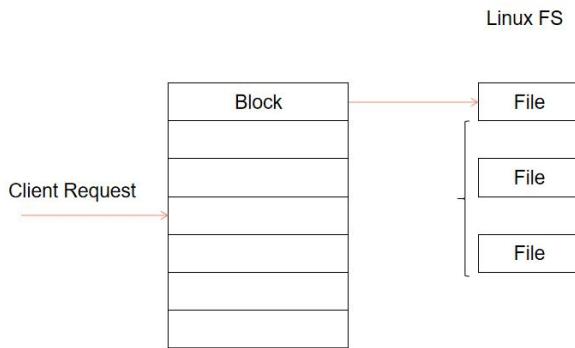
2021/12/20

我们这节课讲讲 HDFS，它用起来很简单，在 CANVAS 上有一个工程，拿 java 对 HDFS 做操作，包括创建目录往里读写文件。我们来讲讲 HDFS 里的实现，讲讲里面最重要的 name node 和 data node。

HDFS

hdfs 就是用来存储大文件的，实际上在装 hadoop 的时候就有了，我们来讲讲它的原理。我们上课的时候提到过，实际上 hdfs 根本不是一个从底层开始的文件系统，它是利用 Linux 文件系统的一个文件系统。数百 G 的大文件切成很多小块，存到底层的 Linux 文件系统。底下是一堆的机器，它在存的时候，落到机器上的时候，用的就是 Linux 的文件系统，从这个

节点的角度来看，就是存了一个个的标准文件。



现在唯一要处理的就是 **block array**，用户请求过来的时候，是先到 **block** 列表中查找文件存在哪些机器上，然后到对应机器上读这个文件。所以首先来说，它适合的是放尺寸很大的文件，放很小的文件的时候就不适用了，性能会下降地非常厉害。每个 **block** 都是 128M，存几 K 的小文件就会有浪费。

而且高效是体现在读取大文件的时候可以从多个机器上并行地去读，可以用数倍的带宽。所以我们的读取就快，如果我们的文件都很小，最多存在一个机器上，那么我们去读它带宽没有增大，但是又要增加 **hdfs** 这一层中间层找到这个机器，再像访问一个普通 **Linux** 文件一样去做。

所以 **hdfs** 一定要存大文件。第二个就是说，它要以写入一次、多次读取的方式去访问，因为我们把一个大文件切开放在这里，然后根据这个表找到在哪些机器上去读。如果说要在中间插入一些数据，此时我们已经切分好了 **block**，就意味着中间插入了数据以后，后面的 **block** 要重新划分重新存储，所以不允许多次写，只有在最后追加的时候，像我们之前将的时序数据，就比较适合这种场景。

所以 **hdfs** 适合流式的数据的追加访问。再往下，它假设集群中的机器都是廉价的机器，机器可能随时随地崩溃，数据有可能丢。为了解决这个问题，提高可靠性，它就要加冗余。所以系统存储数据的时候就会带 **replica**，多个 **replica** 在存的时候，存在同一个机器上没什么用，所以它要有策略来保证机器崩了几台以后，数据还可以获取。

它不适合把本地的小文件快速读出来，因为它加了一层中间层，它只适合大文件。它也不适合并发写、多次修改。

整个 **hdfs** 不复杂，就是在 **Linux** 文件系统之上加了一层，没什么大的改变。

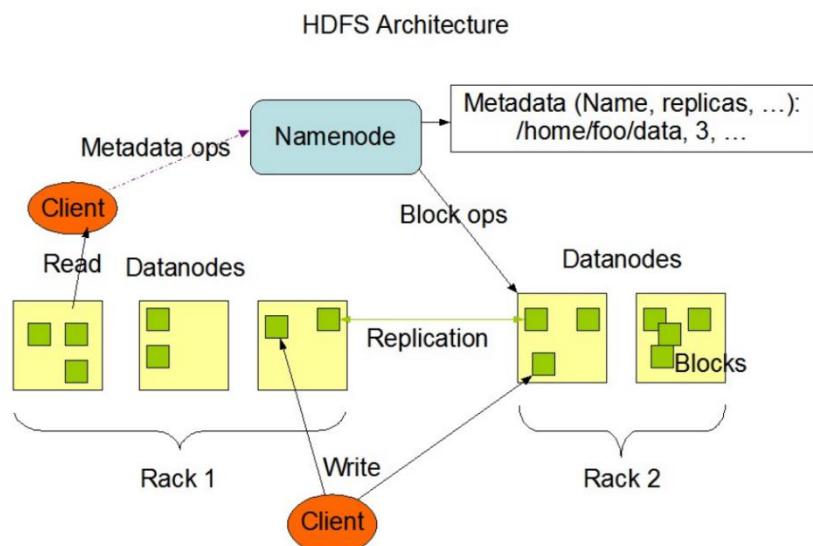


图 1 HDFS 的 master/slave 架构

hdfs 的 master 就是 Namenode，这些 slave 就是 datanode（存数据的）。而 namenode 就是在存文件和 block 之间的映射关系，以及到底哪些 block 在哪些机器上的关系。所以它就在存元数据，并且它还记住了哪些文件有哪些副本，每个副本的 id 是什么，并且存在了哪些机器的哪里。

NameNode

然后这个 NameNode 就会把用户请求导到存了 data blocks 的 datanode 上，它有一个就近原则。在集群中，上面有分布式文件系统，每一个集群节点上还跑了 tomcat、nginx 等应用，就会产生对分布式文件系统的访问。这样的话，我们的一些静态文件存在 hdfs 中，有一个机器要读这个文件，NameNode 会去查表，知道这个文件在哪些节点上有，如果 NameNode 发现了发送请求的机器上就有这个文件，就会就近让机器去读文件。哪怕有 replica 关系，但是在 NameNode 没有主从数据的关系，它只会按照集群中的就近关系去调度文件的请求。

就近原则还可以扩展到机架之间的物理位置关系。所以 replica 和我们 MySQL 中的一主多从不太一样，MySQL 中真的是我们要去写主，而靠主去同步从。而在 hdfs 中，主从关系弱化了，slave 可以做 workload 的均衡。

Q: NameNode 把用户请求调度到有文件的机器上，读数据的时候还需要和 NameNode 交互吗？

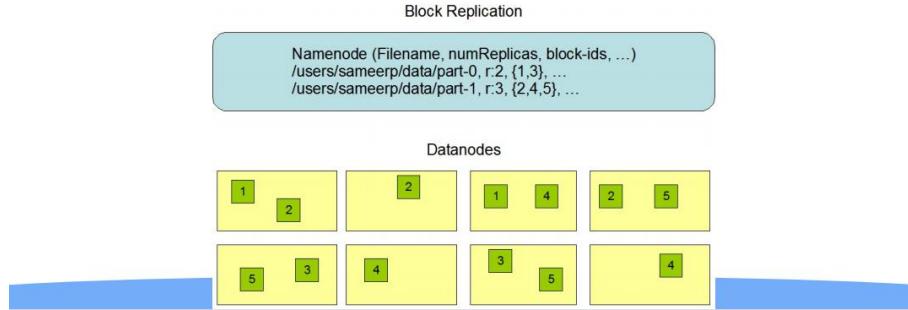
A: 不需要了，不需要靠 NameNode 把文件拿过来给它，这就是因为 NameNode 的责任不能太重了，它只告诉你在哪里。这个道理就和负载均衡一样，调度过去以后就直接和 Tomcat 交互了，不再通过 nginx。

总的来说，在分布式文件系统里，文件是要分块的，要落在 DataNode 上变成一个个标准的 Linux 文件，所谓的 master 就是 NameNode，我们要先和 NameNode 交互知道文件在哪里，而 DataNode 就是在真的存储这个数据。

hdfs 里通过 NameNode 来暴露 hdfs 的名字空间，他就在存储用户的大数据，文件一般来说都会切成多个块，每个块默认的是 128M。NameNode 就要来管理在文件系统中执行的所有操作，比如打开、关闭、重命名，维护映射关系和元数据。NameNode 专门就是一台机器放，不要同时又跑一个 DataNode。

The NameNode

- makes all decisions regarding replication of blocks.
- It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster.
- Receipt of a Heartbeat implies that the DataNode is functioning properly.
- A Blockreport contains a list of all blocks on a DataNode.



NameNode 在存储的时候，replica 的数量可以根据重要性来自定义，不一定每个文件的 replica 是一样的。

现在有 8 个 DataNode，无论是哪个文件，它的副本不会在同一台机器上。在处理的时候，DataNode 不断给 NameNode 发心跳包，如果没有心跳了，认为 DataNode 崩了。先把这台机器标记为不能转发请求，并且文件的副本数（replica factor）不够了，就立马启动一个副本拷贝操作拷贝到另一台机器上。

Block Report 就是在告诉 Master 说自己存了哪些 block。

Q: 心跳和 Block Report 不能一起发吗？

A: 心跳占带宽很小，而 Block Report 很大。并且心跳包的频率要更高。

Q: 能不能在一个 DataNode 中设置一个定时器线程，如每秒向 NameNode 发送一次心跳？

A: 计时器线程活着不一定代表这个 DataNode 可以正常读写。应该是我们后台有一个守护进程在 DataNode 上运行，比如一个死循环的程序上去发心跳。一旦在循环里发心跳，那么时间就不准了。所以时间不会特别准，NameNode 不能严格地卡死时间，只能是在一段时间内心跳不来才认为 DataNode 死了。

DataNode

DataNode 就是真正在处理用户过来的请求。站在 DataNode 的角度，它压根不知道有大文件的存在，它知道自己存的那个是文件，至于大文件切了多少块，每个 DataNode 都不知道。

它比较特殊的一点是，当我们有大量的 block 存到一个 DataNode 上。对于整个 hdfs 来说，这个文件是有一个目录的。比如 /users/1.txt , /users/my/2.txt 注意这个目录是在 hdfs 下的，然后它会把这个文件切成很多块存到每个 DataNode 里去。

从 DataNodes 的角度，只知道有人给了我一块数据让我存，比如存到了本地的 /hdfs/1_1.txt 和 /hdfs/2_1.txt 和 /hdfs/2_2.txt。这个信息就在 NameNode 的表里存着。所以在 DataNode 里不一定会保证路径也一样，是通过 NameNode 的查表来维护这个映射的。

如果 NameNode 让 DataNode 的文件都放到 /hdfs 目录底下，那么这个目录中的文件可能非常多，那么将来要找文件的时候，就会很慢。所以 DataNode 上的 hdfs 会按照不同的日期等逻辑去建立很多子目录来把文件存下来的时候，去减少文件将来查找时候的开销。总的来说，DataNode 上的存储方式和 hdfs 文件系统中记录的文件路径是没有任何关系的。

所以怎么从 `hdfs` 看到的文件名对应到机器上的各个 `block` 去拿出来，我们就是通过 `NameNode` 来做映射。一旦确定了数据在哪些 `DataNode` 上，用户读写数据都是直接和 `DataNode` 交互的，不会再通过 `NameNode`。

本身 `hdfs` 不是一个新的文件系统，它就是在 `Linux FS` 上去跑，所以我们可以看到，我们可以直接在自己的笔记本上跑 `hdfs` 文件系统的，它就是在利用现有的 `Linux FS` 之上再加了一层。

整个 `hdfs` 是用 `java` 写的，所以它的跨平台比较好。一般来说，`NameNode` 是用一台专门的机器去跑，尽量让 `NameNode` 机器的负载轻一点，而 `DataNode` 一般是一台机器一个。

`hdfs` 也支持我们做层次化管理，也就是文件放进 `hdfs` 存的时候支持创建目录，注意这个目录并不代表真正在 `DataNode` 上的目录，我们只认为是管理数据，因为在 `NameNode` 上建立目录搜索起来就快。

`hdfs` 支持用户配额（用户最多使用多少量的数据）和访问控制，它不支持链接和软链接。

HDFS Replica

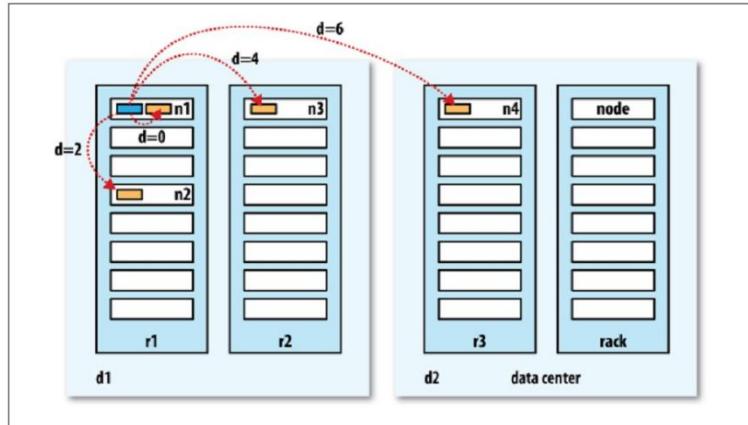
副本放置策略：根据机架来判断 `replica` 怎么放。

Replica Placement: The First Baby Steps



- The placement of replicas is critical to HDFS reliability and performance.
 - The purpose of a **rack-aware replica placement policy** is to improve data reliability, availability, and network bandwidth utilization.
- Large HDFS instances run on a cluster of computers that commonly spread across many racks.
 - Communication between two nodes in different racks has to go through switches.
 - In most cases, network bandwidth between machines in the same rack is greater than network bandwidth between machines in different racks.
- The NameNode determines the rack id each DataNode belongs to via the process outlined in [Hadoop Rack Awareness](#).
 - A simple but non-optimal policy is to place replicas on unique racks.
 - This policy **evenly distributes replicas** in the cluster which makes it easy to balance load on component failure.
 - However, this policy **increases the cost of writes** because a write needs to transfer blocks to multiple racks.

副本最好要跨机架、跨数据中心去放。



它会尽可能保证数据的副本放得远一点。在 **Append** 的时候，我们是以 **pipeline** 的形式去让每个持有副本的机器去写。写完的 **DataNode** 传递给下一个 **DataNode** 去写。数据不会经过 **NameNode**，如果我们放在不同机架上，网络开销就会越大。所以它会在中间做平衡。所以副本从可靠性角度来说，提供的是不同层次的可靠性。

replica 一般来说就是 3 个。

- For the common case, when the replication factor is **three**,
 - HDFS's placement policy is to put **one replica on the local machine if the writer is on a datanode**, otherwise on a random datanode in the same rack as that of the writer,
 - **another replica** on a node in a different (remote) rack, and **the last** on a different node in the same remote rack.
- If the replication factor is **greater than 3**,
 - the placement of the **4th and following replicas** are determined **randomly** while keeping the number of replicas per rack below **the upper limit** (which is basically $(\text{replicas} - 1) / \text{racks} + 2$).

每个机架有一个副本上限，保证副本在每个机架上的数量比较平均。

HDFS 启动的时候可以先进入安全模式。它不做任何副本的操作。先去获取每个 **DataNode** 的心跳包和 **Block Report**。如果满足副本要求，那么就可以开始读写了。否则，它要还去做复制，超过一定比例以后才允许用户读写。

On startup, the **NameNode** enters a special state called **Safemode**.

- Replication of data blocks does **not** occur when the NameNode is in the Safemode state.
- The NameNode receives **Heartbeat** and **Blockreport** messages from the DataNodes.
- A Blockreport contains the list of data blocks that a DataNode is hosting.
- Each block has a specified **minimum number of replicas**.
- A block is considered safely replicated when the minimum number of replicas of that data block has checked in with the NameNode.
- After a **configurable percentage of safely replicated data blocks** checks in with the NameNode (plus an additional 30 seconds), the NameNode exits the Safemode state.
- It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas.
- The NameNode then replicates these blocks to other DataNodes.

要注意之前的 MySQL 和 RocksDB 都有一个 **WAL (write-ahead log)**，一旦写数据我们要记录下来，否则我们不能去做这个动作。它的目的就是系统崩溃以后就可以做回滚，恢复到安全状态。

The Persistence of File System Metadata



- The HDFS namespace is stored by the NameNode.
 - The NameNode uses a transaction log called the **EditLog** to persistently record every change that occurs to file system metadata.
 - For example, **creating a new file in HDFS** causes the NameNode to **insert a record** into the EditLog indicating this.
 - Similarly, **changing the replication factor of a file** causes a **new record** to be inserted into the EditLog.
 - The NameNode uses a file in its **local host OS file system** to store the EditLog.
 - The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in a file called the **FsImage**.
 - The FsImage is stored as a file in the NameNode's **local file system** too.

目的是恢复 HDFS，所以不能存储在 `hdfs` 中，所以 `EditLog` 要存在本地操作系统里。

The Persistence of File System Metadata



- The NameNode keeps an image of the entire file system namespace and file Blockmap in memory.
 - When the NameNode **starts up**, or a **checkpoint is triggered** by a configurable threshold, it reads the **FsImage** and **EditLog from disk**, applies all the transactions from the EditLog to the **in-memory representation of the FsImage**, and flushes out this new version into a **new FsImage on disk**.
 - It can then **truncate the old EditLog** because its transactions have been applied to the persistent FsImage.
 - This process is called a **checkpoint**.
 - The purpose of a checkpoint is to make sure that HDFS has a consistent view of the file system metadata by taking a **snapshot** of the file system metadata and saving it to FsImage.
 - A checkpoint can be triggered at a **given time interval** (`dfs.namenode.checkpoint.period`) expressed in **seconds**, or after a **given number of filesystem transactions** have accumulated (`dfs.namenode.checkpoint.txns`).
 - If both of these properties are set, the first threshold to be reached triggers a checkpoint.

每十条记录，我们就往所有操作都写到 `FsImage` 中。一旦写成功，就把 `EditLog` 清空。将来如果发生什么事情，我们就可以回滚到这个点上。

The Persistence of File System Metadata



- The DataNode stores HDFS data in files in its local file system.
 - The DataNode has **no** knowledge about HDFS files.
 - It stores each block of HDFS data in a **separate file in its local file system**.
 - The DataNode does **not** create all files in the same directory.
 - Instead, it uses a **heuristic** to determine the **optimal number of files per directory** and **creates subdirectories appropriately**.
 - It is **not optimal** to create all local files in **the same directory** because the local file system might not be able to efficiently support a huge number of files in a single directory.
 - When a DataNode starts up, it scans through its local file system, generates a list of all HDFS data blocks that correspond to each of these local files, and sends this report to the NameNode.
 - The report is called the **Blockreport**.

DataNode 会用启发式算法来决定每个目录里的文件数量。

- All HDFS communication protocols are layered on top of the **TCP/IP** protocol.
 - A **client** establishes a connection to a **configurable TCP port** on the NameNode machine.
 - It talks the **Client Protocol** with the NameNode.
 - The DataNodes talk to the NameNode using the **DataNode Protocol**.
 - A Remote Procedure Call (**RPC**) abstraction wraps both the Client Protocol and the DataNode Protocol.
 - By design, the NameNode **never** initiates any RPCs. Instead, it **only responds** to RPC requests issued by DataNodes or clients.

通信是通过 **TCP/IP** 协议，整个流程是通过远程过程调用。NameNode 是一个被动节点，不会主动联系 client 和 DataNode。

Robustness



- The primary objective of HDFS is to store data reliably even in the presence of failures.
 - The three common types of failures are **NameNode** failures, **DataNode** failures and **network partitions**.
- **Data Disk Failure, Heartbeats and Re-Replication**
 - Each DataNode sends a **Heartbeat** message to the NameNode **periodically**.
 - A **network partition** can cause a subset of DataNodes to lose connectivity with the NameNode.
 - The NameNode detects this condition by the absence of a Heartbeat message.
 - The NameNode marks DataNodes **without recent Heartbeats as dead** and does **not** forward any new IO requests to them. Any data that was registered to a **dead DataNode** is **not** available to HDFS any more.

Cluster Rebalancing

The HDFS architecture is **compatible** with **data rebalancing schemes**.

- A scheme might automatically move data from one DataNode to another if the free space on a DataNode falls below a certain threshold.
- In the event of a **sudden high demand** for a particular file, a scheme might **dynamically create additional replicas** and rebalance other data in the cluster.
- These types of data rebalancing schemes are not yet implemented.

负载突然变高的时候，可以动态修改 **replica**。

Data Integrity

It is possible that a block of data fetched from a DataNode arrives **corrupted**.

- This corruption can occur because of faults in **a storage device, network faults, or buggy software**.
- The HDFS client software implements **checksum checking** on the contents of HDFS files.
- When a client creates an HDFS file, it computes a **checksum of each block of the file** and **stores these checksums in a separate hidden file** in the same HDFS namespace.
- When a client retrieves file contents it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file.
- If **not**, then the client can opt to retrieve that block from **another DataNode** that has a replica of that block.

数据完整性是通过校验和来做的。

Metadata Disk Failure

- The **FsImage** and the **EditLog** are central data structures of HDFS.
 - A corruption of these files can cause the HDFS instance to be non-functional.
 - For this reason, the NameNode can be configured to support **maintaining multiple copies of the FsImage and EditLog**.
 - Any update to either the FsImage or EditLog causes each of the FsImages and EditLogs to get updated **synchronously**.
 - When a NameNode restarts, it selects the latest consistent FsImage and EditLog to use.
- **Snapshots**
 - **Snapshots** support storing a copy of data at a particular instant of time. One usage of the snapshot feature may be to roll back a corrupted HDFS instance to a previously known good point in time.

如果存元数据的 EditLog 和 FSimage 坏了，hdfs 就坏了，所以这个也需要做多 replica 和同步更新。

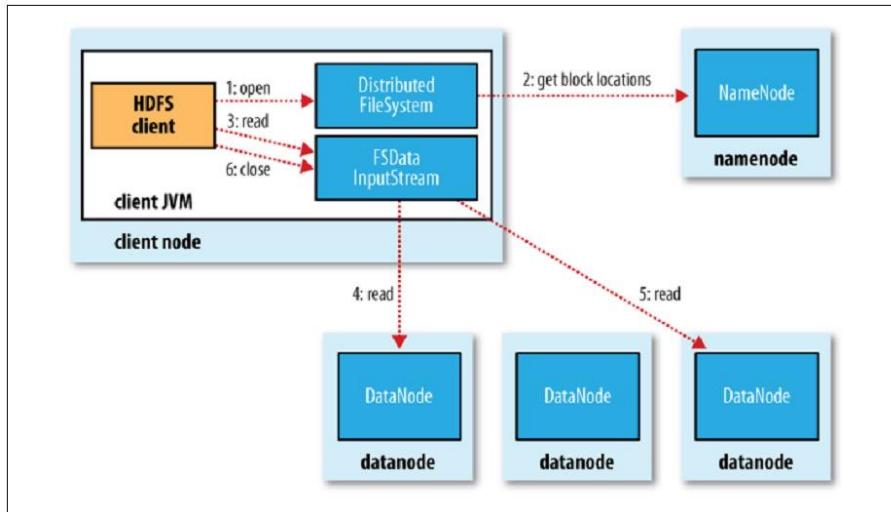


图 2 HDFS 读数据

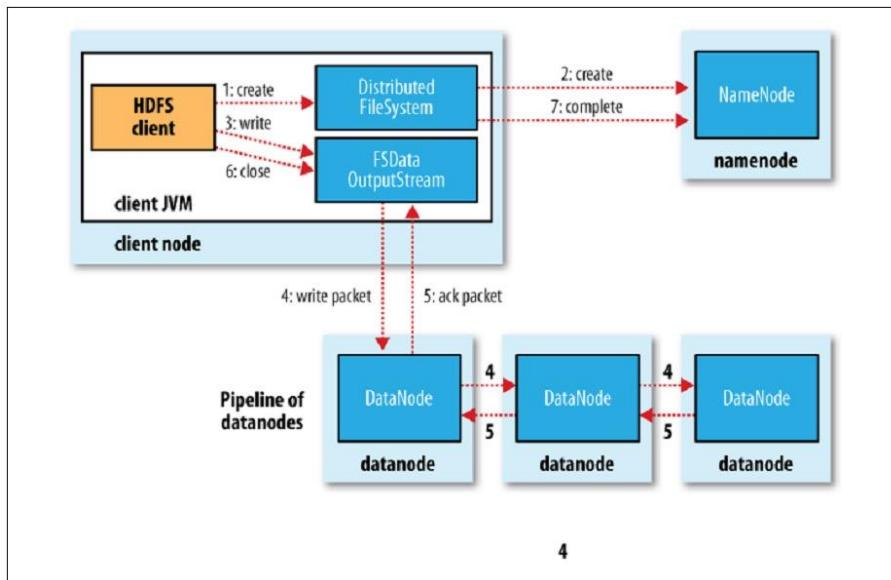


图 3 HDFS 写数据

注意到写数据的时候就是以流水线的方法去写，速度可以快一点。

HDFS can be accessed from applications in many different ways.

- Natively, HDFS provides a [FileSystem Java API](#) for applications to use.
- A [C language wrapper for this Java API](#) and [REST API](#) is also available.
- In addition, an HTTP browser and can also be used to browse the files of an HDFS instance.
- By using [NFS gateway](#), HDFS can be mounted as part of the client's local file system.

Browser Interface

- A typical HDFS install configures a [web server](#) to expose the HDFS namespace through a configurable TCP port.
- This allows a user to navigate the HDFS namespace and view the contents of its files using a web browser.

和 HDFS 的交互方式如上图所示。

- **File Deletes and Undeletes**

- If trash configuration is enabled, files removed by [FS Shell](#) is **not** immediately removed from HDFS.
- Instead, HDFS moves it to a **trash directory** (each user has its own trash directory under /user/<username>/Trash).
- The file can be restored quickly as long as it remains in trash.
- Most recent deleted files are moved to the current trash directory (/user/<username>/Trash/Current), and in a **configurable interval**, HDFS creates checkpoints (under /user/<username>/Trash/<date>) for files in current trash directory and deletes old checkpoints when they are expired.
- After the **expiry** of its life in trash, the NameNode deletes the file from the HDFS namespace.
- The deletion of a file causes **the blocks associated with the file to be freed**.
- Note that there could be an appreciable time delay between the time a file is deleted by a user and the time of the corresponding increase in free space in HDFS.

等垃圾箱中的过期了以后，所有空间才会被释放。

使用方法和代码略。

2021/12/23

HBase

我们今天来讲讲 HBase，我们在讲 hadoop 的时候讲有一个 big table。而 HBase 就是一个开源版的 Big Table 来存储对结构要求不高的数据，它和 MongoDB 还是有点差异的。

- [Apache HBase™](#) is the [Hadoop](#) database, a **distributed, scalable, big data store**.
 - Use Apache HBase™ when you need **random, realtime read/write access** to your Big Data.
 - This project's goal is the hosting of very large tables -- **billions of rows X millions of columns** -- atop clusters of commodity hardware.
 - Apache HBase is an [open-source, distributed, versioned, non-relational database](#) modeled after Google's [Bigtable: A Distributed Storage System for Structured Data](#) by Chang et al.
 - Just as Bigtable leverages the distributed data storage provided by the Google File System, Apache HBase provides **Bigtable-like capabilities** on top of [Hadoop](#) and [HDFS](#).
- <https://hbase.apache.org/>

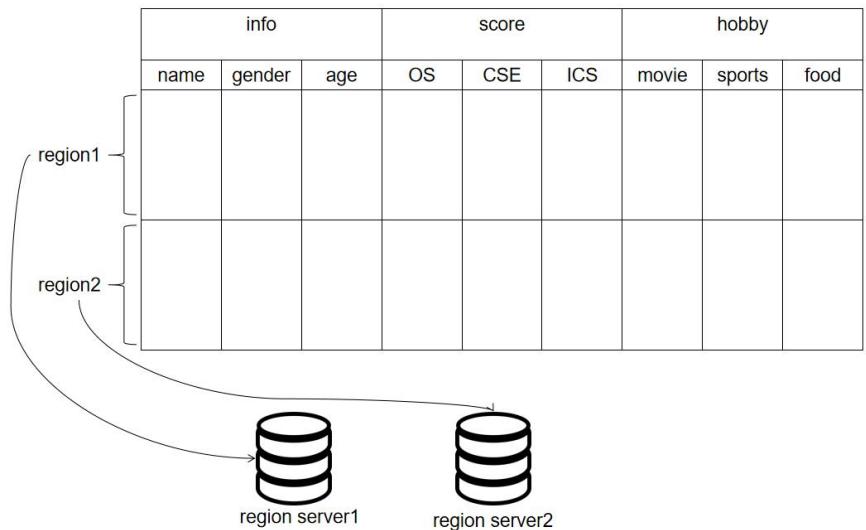
这个系统有几个关键词，数据表中的行和列都会非常多，面对的就是这种大数据的存储，所以它要支持分布式存储和版本。在行列交叉的地方，它有一个 **cell**，可以存若干个版本。它实际上在运行的时候，它会在集群里跑一个分布式，所以它底层要依赖于 [hadoop](#) 和分布式的文件系统。

HBase 是按照列来存储的，为了按列存储，这些列还分成列族。

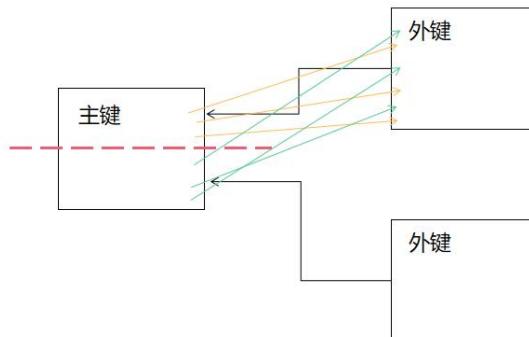
HBase 的一张表的例子如下：

info			score			hobby		
name	gender	age	OS	CSE	ICS	movie	sports	food

而在关系型数据库中，我们可能就要拆分成三张表，然后通过人的 id 做外键关联。在 HBase 中，我们就认为 info、score 和 hobby 是列族的概念，应该拼在一起变成一张表。



当这张 HBase 的表内容变多以后，我们就可以按行拆分出一个个 region 然后分布存到不同的 region server 上。



主键Region切分并没有保证从表中也是可分的，还是需要额外搜索和排序

而在关系型数据库中，我们对于主表按行切分完以后，注意到从表中的数据并不保证也是按行可分的，我们需要从主表开始做搜索，手动地排序好。这就不如 HBase 这种机制方便。

第二个例子是，比如说我们现在有一个张三同学，它的操作系统考了三次，成绩分别是 50、55 和 60。这三个值在逻辑上都是张三的 OS 成绩，但是属于不同的版本，为了区分它们，每一个版本上我们都要加一个时间戳。取值的时候，如不加特别声明，读到的就是最新的值。当然也可以指定返回历史所有的成绩，HBase 对于一个 cell 中可以保存几个版本是没有上限的。

The diagram shows a 9x3 HBase table with columns: info (name, gender, age), score (OS, CSE, ICS), and hobby (movie, sports, food). The 'name' column for the first row contains '张三'. The 'OS' column for this row contains three cells with values 21, 50, and 60, each with a timestamp. Dashed lines connect these three cells to a single '张三' entry in the 'info' row.

图 9 张三历次 OS 成绩的存储方式

Q: 然后它为什么是面向列的呢?

A: 一个是因为它有列族的概念。更重要的是，我们把每个 `region` 存到硬盘上的时候也不是占据一个连续的空间，而是每一个列族可能存放在不同的地方，列族内部是连续存放的。

Q: 为什么要这么存呢?

A: 因为这种大数据它认为是和时间相关的数据，而它认为列的取值不是很大，所以它在这里存的时候就不一定存的是原始数据了，可能存的是第一个 60 分和之后的偏移量。所以它认为按照列存的时候可以省空间，并且在编码机制上还可以继续压缩。列存的话，编码机制和压缩机制都可以保证存储效率更低。

在 HBase 中，每一行都有一个 `id`，这个 `id` 应该怎么设计让它排完序以后按照我们的要求就显得比较重要。

场景：我们在爬一些网页，这时候我们发现有关软件学院的页面的内容在排完序之后应当就近存放。那我们怎么做呢？比如说软件学院的 URL，就是 `se.sjtu.edu.cn`，而不同的实验室就是在最前面还有前缀。我们爬虫以后按照网址倒序排序，那么就是 `cn.edu.sjtu.se.....` 这样就可以把软件学院的网页排序排在一起。所以我们就要把排序之后的结果变为 `region`。按照这种方式的话，相关的数据就放在了一个 `region` 中，到时候我们就可以很方便地拿出来了。

它不像是关系型数据库中一个表放在这里，它这里每一个行列交叉的地方还可以存版本。所以这个比较适合做分布式地存储，我们可以约定每个 `region` 是 100G。这就和 HDFS 一样，每次产生数据的时候产生新的 `block`，所以理论上来说它的扩展性更好，并且元数据可以映射有多少个表，每个表切成了多少块放在了哪里。元数据又有元数据的元数据，对索引分块。所以它的扩展效率基本上是线性的，无论我们加了多少台机器，和一台的性能差不多。

因为它是关系型数据库，所以它是不支持 SQL 的。可能大学生的爱好和中学生的爱好是不同的，可能大学生根本没有一些选项。所以它的 `schema` 是不严格的。

info			score			hobby		
name	gender	age	OS	CSE	ICS	movie	sports	food
张三	男	21	50	75	89	战狼2	跑步	
退休人	男	60	55					米饭

图 10 HBase 表的例子（灰色是指没有数据）

最后我们会看到这张表里有很多地方是空，比如我们存一个退休的人，就根本没有学习成绩。最终导致了这个表里有很多空洞，最终可能画出来就是一个稀疏矩阵。所以它的优化是，列表中没有值的地方就不存。

所以逻辑上它就是一个稀疏的表格，所以它不会支持结构化查询，因为大家列放在一起个数都不一样。

比如我们爬网页，我们可以存在表里，有一些属性基本相同，但是“是否包含第三方链

接”、“是否包含图片”这些属性不同的网页都不一样。

所以 bigtable 给了大家一个启示，Hbase 就是另一拨人实现的。HBase 官网上宣传的特性：

1. 线性可扩展的，一个表可以无限增长上去，并且它存在一些优化。如果这个表已经有 100 个 region 了，我们可能就把前面 98 个 region 整体压缩一下存到硬盘上，只留 2 个在内存中。
2. 严格一致的读写，基本上就是要做事务控制。
3. 表可以做 sharding，和 MongoDB 长的一样，这里压出来的碎片就叫 shard，放到 region server，可以靠 HDFS 来做冗余。
4. 底下就是一些实现的细节，里面在执行的所有操作是可以支持 MapReduce 的。
5. 它做了一些缓存机制，加了 bloom 过滤器来提升实时查询。但是布隆过滤器可以很快确定数据不在 region server 上。
6. 把查询下推到 server-side filter。
7. 使用 thrift gateway 来做认证。
8. Ganglia 类似于 influxDB 来监控集群中的数据。

一张张表的行一个个单元格（cell）是可以做多个版本的存储的，而且它没有做任何限制，理论上一个同学如果 OS 没有及格，那可以一直加数据进去。

每一行的 keys 是字节数组。每一行是通过 key 的字节序去排列的。列组成了列族，当我们在引用一个列的时候，我们在讲的是列族：列名。我们在设计表的时候就要改变关系型数据库中的设计思路，合并成一张表。如果语义上不是那么紧耦合的，我们就拆开来形成一个列族。

列族相当于列的前缀。我们看到的这个表的行是排序的，列是可以动态增长的，而行列交叉处就可以增加版本。我们可以想一下关系型数据库为什么不容易让列动态增长的，这还是和存储方法相关，因为关系型数据库默认是按照行存的，这样就要把所有 entry 往后挪，代价就很大。

HBase 的配置和使用

数据插入到一定程度，数据量就很大了，就会动态地切开它，这是一个自动的水平分割，产生另一个 region，这一点放到 HDFS 中就是一模一样（写第一个 block 写满之后，写第二个 block，不同的 block 可以存在不同地方。）

Region 就是按照行排序的一系列数据，需要记录 row keys 是从哪到哪。HBase 就可以根据我们里面存的那个内容来记录这个范围。

在它的表上是可以加锁进行处理的。

- Set the `JAVA_HOME` environment in `conf/hbase-env.sh` file.

```
# Set environment variables here.
# The java implementation to use.
export JAVA_HOME=$(/usr/libexec/java_home)
```

- Edit `conf/hbase-site.xml`

```
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>file:///Users/user/hbase</value>
  </property>
</configuration>
```

需要设置 HBase 中的 `JAVA_HOME` 在哪里。并且要告诉它，将来我们的数据要存到哪里去。

- The `bin/start-hbase.sh` script is provided as a convenient way to start HBase.
- You can use the `jps` command to verify that you have one running process called **HMaster**.

```
hbase-2.4.8 % jps
71047 HMaster
59738
71149 Jps
59199
```

- In standalone mode HBase runs all daemons within this single JVM, i.e.
 - the **HMaster**, a single **HRegionServer**, and the **ZooKeeper** daemon.

这个脚本跑的时候，本地版本用到一个 Master 和一个 RegionServer 还有一个 ZooKeeper。

Go to <http://localhost:16010> to view the HBase Web UI.

The screenshot shows the Apache HBase Web UI interface. At the top, there's a navigation bar with links for Home, Table Details, Process Metrics, Local Logs, Log Level, Debug Dump, Metrics Dump, Profiler, and HBase Configuration. Below the navigation bar, the title is "Backup Master MacBook-Pro-7". It shows the current active master is "MacBook-Pro-7". Under the "Tasks" section, there's a table with columns for Start Time, Description, State, and Status. One entry is listed: "Tue Dec 21 12:50:02 CST 2021" for "Master startup", state "RUNNING (since 29sec ago)", and status "Initializing Master file system (since 29sec ago)". The "Software Attributes" section lists various system properties with their values and descriptions. For example, "HBase Version" is "2.2.4, revision=6779d1a325a4f78a46bf0339e73bf7588bac" and its description is "HBase version and revision". Other attributes include HBase Compiled (2020年 03月 11日 星期三 12:57:39 CST, ha), HBase Source Checksum (19adabab3544a5a6cc0caac0d5f299ca), Hadoop Version (2.8.5, revision=0b4464075227fe02de7f2410377bd3d3d5f8), Hadoop Compiled (2018-09-10T11:00Z, jdk), and Hadoop Source Checksum (9942c05c745417c14d1883f420723).

- Create a table.

- Use the `create` command to create a new table. You must specify the **table name** and the **ColumnFamily name**.

```
hbase(main):001:0> create 'test', 'cf'
0 row(s) in 0.4170 seconds
=> Hbase::Table - test
```

- List Information About your Table

- Use the `list` command to confirm your table exists

```
hbase(main):002:0> list 'test'
TABLE
test
1 row(s) in 0.0180 seconds
=> ["test"]
```

在关系型数据库中，库的概念在 HBase 中就是 **namespace**，创建表的时候如果不指定，那么就是默认的 **namespace** 中。列族的数量最好不要多于三个。因为我们必须记录 **region** 中每个列族都在什么位置，列族太多就会影响检索的效率。所以我们可以多放一些列，因为列是挨着存的，存的多一点不要紧。

Now use the `describe` command to see details, including configuration defaults

```
hbase(main):003:0> describe 'test'
Table test is ENABLED
test
COLUMN FAMILIES DESCRIPTION
{NAME => 'cf', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION_BEHAVIOR =>
'false', KEEP_DELETED_CELLS => 'FALSE', CACHE_DATA_ON_WRITE => 'false',
DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REPLICATION_SCOPE
=> '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_MEMORY => 'false',
CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BLOCKS_ON_OPEN => 'false', COMPRESSION =>
'NONE', BLOCKCACHE => 'true' BLOCKSIZE => '65536'}
1 row(s)
Took 0.9998 seconds
64K: 存的最小的单位
```

关键是我们要往里读写。

Put data into your table.

- To put data into your table, use the `put` command.

```
hbase(main):003:0> put 'test', 'row1', 'cf:a', 'value1'
0 row(s) in 0.0850 seconds
hbase(main):004:0> put 'test', 'row2', 'cf:b', 'value2'
0 row(s) in 0.0110 seconds
hbase(main):005:0> put 'test', 'row3', 'cf:c', 'value3'
0 row(s) in 0.0100 seconds
```

- Here, we insert three values, one at a time.
- The first insert is at **row1**, column **cf:a**, with a value of **value1**.
- Columns in HBase are comprised of a column family prefix, **cf** in this example, followed by a colon and then a column qualifier suffix, **a** in this case.

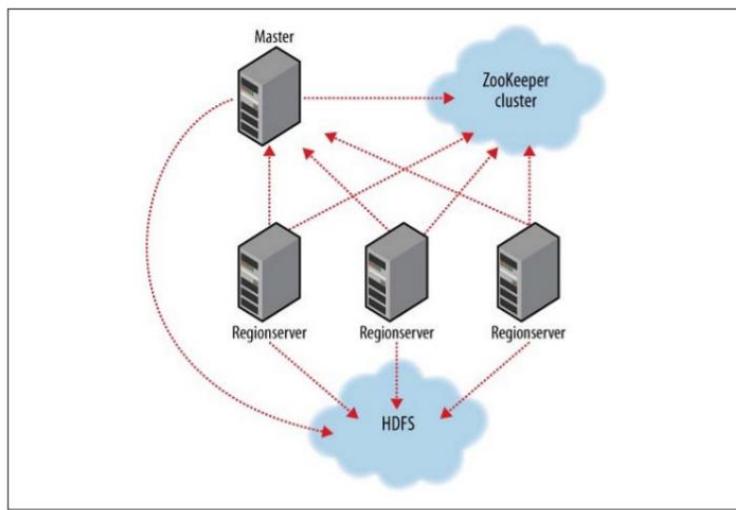
在 java 中，如果要写东西，我们就要创建一个 `put` 对象。这里我们就可以发现列是自动添加的，我们在创建 **cf (column family)** 的时候并没有指定 **a**、**b**、**c** 列。并且创建的表确实是一个稀疏的表，因为它的结构都不一样。从关系型数据库的角度就会认为，这三个数据就不应该放在一张表里。但是从 **HBase** 应用的角度，可能这个就是一门课三个不同的作业形式。所以 **HBase** 允许我们让一些语义上紧耦合但是结构上不一样的数据存在一起。

Scan the table for all data at once.

- One of the ways to get data from HBase is to scan.
- Use the `scan` command to scan the table for data.
- You can limit your scan, but for now, all data is fetched.

```
hbase(main):006:0> scan 'test'  
ROW      COLUMN+CELL  
row1    column=cf:a, timestamp=1421762485768, value=value1  
row2    column=cf:b, timestamp=1421762491785, value=value2  
row3    column=cf:c, timestamp=1421762496210, value=value3  
3 row(s) in 0.0230 seconds
```

我要删除一张表的话，必须要先 `disable` 掉。



底层它依赖于 HDFS，上层依赖于 ZooKeeper。

我们可以把它想象为一个多维的表。每一个列族包含列，每一列包含多个行。其中还包含了多个版本信息。

列族在物理上是放在一起的，但是列族和列族之间可以不放在一起。不同的版本就是使用时间戳来表示的。

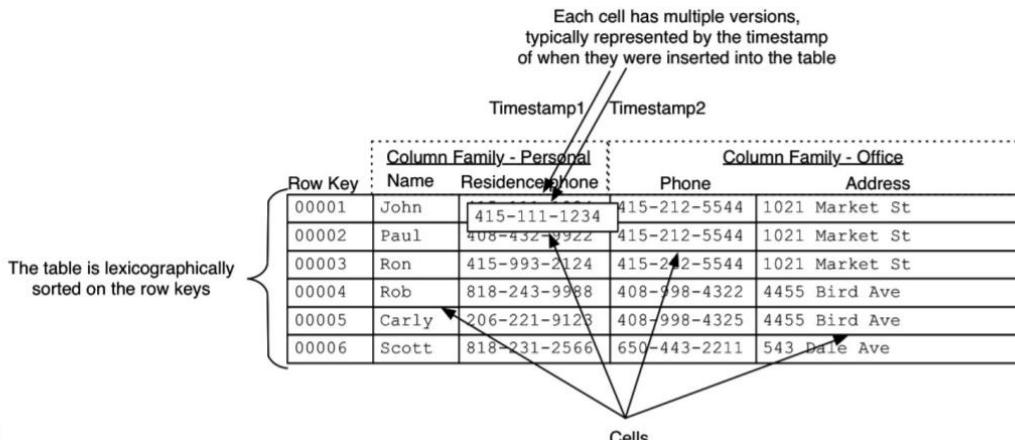
如下是一个对网页爬虫的例子：

- Cells in this table that appear to be **empty** do not take space, or in fact exist, in HBase.
- This is what makes HBase "**sparse**."

Row Key	Time Stamp	ColumnFamily contents	ColumnFamily anchor	ColumnFamily people
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"	
"com.cnn.www"	t8		anchor:my.look.ca = "CNN.com"	
"com.cnn.www"	t6	contents:html = "<html>..."		
"com.cnn.www"	t5	contents:html = "<html>..."		
"com.cnn.www"	t3	contents:html = "<html>..."		
"com.example.www"	t5	contents:html = "<html>..."		people:author = "John Doe"

有三个列族，内容、外联和人相关的信息。对于 `anchor` 在不同的时间戳看到的是不一样的。所以在物理上是这样存的：

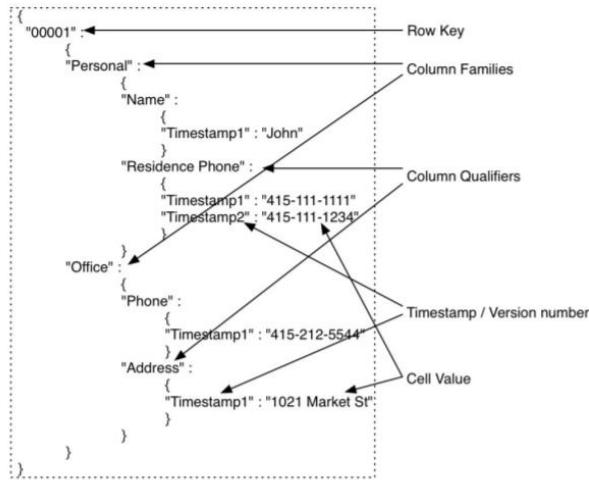
Cells in this table that appear to be **empty** do not take space, or in fact exist, in HBase. This is what makes HBase "sparse."



当它有多个版本的时候，在存储的时候是按照时间戳的顺序倒序排序的。当我们做 get 操作的时候，如果没有指定版本，我们就会拿各个列上最新的数据。

The following represents the same information as a **multi-dimensional map**.

```
{
  "com.cnn.www": {
    "contents": {
      "t6": "contents:html: <html>..." ,
      "t5": "contents:html: <html>..." ,
      "t3": "contents:html: <html>..." ,
    }
    "anchor": {
      "t9": "anchor:cnnssi.com = \"CNN\"",
      "t8": "anchor:my.look.ca = \"CNN.com\""
    }
    "people": {}
  }
  "com.example.www": {
    "contents": {
      "t5": "contents:html: <html>..." ,
    }
    "anchor": {}
    "people": {
      "t5": "people:author: \"John Doe\""
    }
  }
}
```



这里面所有内容都属于 `cnn` 这一行。Contents 有 3 个时间戳，存的内容是不一样的。而 anchor 是有两列，分别是在 t8 和 t9 插入。我们在 t10 就可以读到三列最新的值。

- A namespace is a logical grouping of tables analogous to a **database** in relation database systems. This abstraction lays the groundwork for upcoming multi-tenancy related features:
 - Quota Management ([HBASE-8410](#)) - Restrict the amount of resources (i.e. regions, tables) a namespace can consume.
 - Namespace Security Administration ([HBASE-9206](#)) - Provide another level of security administration for tenants.
 - Region server groups ([HBASE-6721](#)) - A namespace/table can be pinned onto a subset of RegionServers thus guaranteeing a coarse level of isolation.

我们可以认为地控制 namespace 的数据存到哪几台机器上。我们要引用的时候就需要

在表的前面再去加一个冒号。

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class HBaseTest {
    //获取配置信息
    public static Configuration conf;
    static{
        conf = HBaseConfiguration.create();
    }
    //1.判断一张表是否存在
    public static boolean isExist(String tableName){
        //对表操作需要使用 HbaseAdmin
        try {
            Connection connection =ConnectionFactory.createConnection(conf);
            //管理表
            HBaseAdmin admin = (HBaseAdmin) connection.getAdmin();

            return admin.tableExists(TableName.valueOf(tableName));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }
    //2.在 hbase 创建表
    public static void createTable(String tableName, String... columnfamily){
        try {
            //对表操作需要使用 HbaseAdmin
            Connection connection =ConnectionFactory.createConnection(conf);
            //管理表
            HBaseAdmin admin = (HBaseAdmin) connection.getAdmin();
            //1.表如果存在, 请输入其他表名
            if(isExist(tableName)){
                System.out.println("表存在, 请输入其他表名");
            }else{
                //2.注意:创建表的话, 需要创建一个描述器
                HTableDescriptor htd = new HTableDescriptor(TableName.valueOf(tabl
```

```

eName));
    //3. 创建列族
    for(String cf:columnfamily){
        htd.addFamily(new HColumnDescriptor(cf));
    }
    //4. 创建表
    admin.createTable(htd);
    System.out.println("表已经创建成功！");
}
} catch (IOException e) {
    e.printStackTrace();
}
}

//3.删除 hbase 的表
public static void deleteTable(String tableName) {
    try {
        //对表操作需要使用 HbaseAdmin
        Connection connection =ConnectionFactory.createConnection(conf);
        //管理表
        HBaseAdmin admin = (HBaseAdmin) connection.getAdmin();
        //1.表如果存在, 请输入其他表名
        if (!isExist(tableName)) {
            System.out.println("表不存在");
        } else {
            //2.如果表存在, 删除
            admin.disableTable(TableName.valueOf(tableName));
            admin.deleteTable(TableName.valueOf(tableName));
            System.out.println("表删除了");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

//4.添加数据 put 'user','rowkey','info:name','tony'
public static void addRow(String tableName,String rowkey,String cf,String column,String value){
    try {
        //对表操作需要使用 HbaseAdmin
        Connection connection =ConnectionFactory.createConnection(conf);
        Table t = connection.getTable(TableName.valueOf(tableName));
        //1.表如果存在, 请输入其他表名
        if (!isExist(tableName)) {
            System.out.println("表不存在");
        } else {

```

```

//2.用 put 方式加入数据
Put p = new Put(Bytes.toBytes(rowkey));
//3.加入数据
p.addColumn(Bytes.toBytes(cf),Bytes.toBytes(column),Bytes.toBytes(value));
t.put(p);
}
} catch (IOException e) {
e.printStackTrace();
}
}

//5.删除表中一行数据
public static void deleteRow(String tableName,String rowkey,String cf ){
try {
//对表操作需要使用 HbaseAdmin
Connection connection = ConnectionFactory.createConnection(conf);
Table t = connection.getTable(TableName.valueOf(tableName));
//1.表如果存在, 请输入其他表名
if (!exist(tableName)) {
System.out.println("表不存在");
} else {
//1.根据 rowkey 删除数据
Delete delete = new Delete(Bytes.toBytes(rowkey));
//2.删除
t.delete(delete);
System.out.println("删除成功");
}
} catch (IOException e) {
e.printStackTrace();
}
}

//6.删除多行数据
public static void deleteAll(String tableName,String... rowkeys){
try {
//对表操作需要使用 HbaseAdmin
Connection connection = ConnectionFactory.createConnection(conf);
Table t = connection.getTable(TableName.valueOf(tableName));
//1.表如果存在, 请输入其他表名
if (!exist(tableName)) {
System.out.println("表不存在");
} else {
//1.把 delete 封装到集合
List<Delete> list = new ArrayList<Delete>();
//2.遍历
}
}
}

```

```

        for (String row:rowkeys){
            Delete d = new Delete(Bytes.toBytes(row));
            list.add(d);
        }
        t.delete(list);
        System.out.println("删除成功");
    }
} catch (IOException e) {
    e.printStackTrace();
}
}

//7.扫描表数据 scan 全表扫描
public static void scanAll(String tableName){
    try {
        //对表操作需要使用 HbaseAdmin
        Connection connection =ConnectionFactory.createConnection(conf);
        Table t = connection.getTable(TableNamespace.valueOf(tableName));
        //1.实例 scan
        Scan s = new Scan();
        //2.拿到Scanner 对象
        ResultScanner rs = t.getScanner(s);
        //3.遍历
        for (Result r:rs){
            Cell[] cells = r.rawCells();
            //遍历具体数据
            for (Cell c : cells){
                System.out.print("行键为: "+Bytes.toString(CellUtil.cloneRow(c))+"
");
                System.out.print("列族为: "+Bytes.toString(CellUtil.cloneFamily(c))
+");
                System.out.print("列名为: "+Bytes.toString(CellUtil.cloneQualifier
(c))+"
");
                System.out.println("值为: "+Bytes.toString(CellUtil.cloneValue(c)));
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void getRow(String tableName,String rowkey) throws IOException {
    Connection connection =ConnectionFactory.createConnection(conf);
    //拿到表对象
    Table t = connection.getTable(TableNamespace.valueOf(tableName));
    //1.扫描指定数据需要实例对象 Get
}

```

```

Get get = new Get(Bytes.toBytes(rowkey));
//2. 可加过滤条件
get.addFamily(Bytes.toBytes("info"));
Result rs = t.get(get);
//3. 遍历
Cell[] cells = rs.rawCells();
for (Cell c : cells){
    System.out.print("行键为: "+Bytes.toString(CellUtil.cloneRow(c))+ " ");
    System.out.print("列族为: "+Bytes.toString(CellUtil.cloneFamily(c))+ " ");
    System.out.print("列名: "+Bytes.toString(CellUtil.cloneQualifier(c))+ " ");
    System.out.println("值为: "+Bytes.toString(CellUtil.cloneValue(c))+ " ");
}
}

public static void main(String[] args) throws IOException {
    System.out.println(isExist("test"));
    createTable("test","info");
    // deleteTable("test");
    addRow("test","101","info","age","20");
    // deleteRow("test","101","info");
    // deleteAll("test","1001","1002");
    scanAll("test");
    getRow("test","101");
}
}

```

- pom.xml

```

?xml version="1.0" encoding="UTF-8"?
<project>
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>SE3353_29_HBaseSample</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.apache.hbase/hbase-client -->
        <dependency>
            <groupId>org.apache.hbase</groupId>
            <artifactId>hbase-client</artifactId>
            <version>2.4.8</version>
        </dependency>
    </dependencies>
</project>

```

我们也可以在访问数据的时候指定一下我们要访问哪个版本，理论上版本是没有上限的。我们也可以使用'alter 't1', Name => 'f1', VERSIONS =>5'设置版本上限为 5.

- By default, i.e. if you specify no explicit version, when doing a get, the cell whose version has the largest value is returned (which may or may not be the latest one written, see later).
- The default behavior can be modified in the following ways:
 - to return more than one version, see [Get.setMaxVersions\(\)](#)
 - to return versions other than the latest, see [Get.getTimeRange\(\)](#)

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...
Get get = new Get(Bytes.toBytes("row1"));
get.setMaxVersions(3); // will return last 3 versions of row
Result r = table.get(get);
byte[] b = r.getvalue(CF, ATTR); // returns current version of value
List<Cell> cells = r.getcolumncells(CF, ATTR); // returns all versions of this column
```

我们也可以设置一下 Get 类一下子返回 3 个版本。

Versions

- **Implicit Version Example**
 - The following Put will be implicitly versioned by HBase with the current time.

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...
Put put = new Put(Bytes.toBytes(row));
put.add(CF, ATTR, Bytes.toBytes(data));
table.put(put);
```
- **Explicit Version Example**
 - The following Put has the version timestamp explicitly set.

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();
...
Put put = new Put(Bytes.toBytes(row));
long explicitTimeInMs = 555; // just an example
put.add(CF, ATTR, explicitTimeInMs, Bytes.toBytes(data));
table.put(put);
```

这样就可以省空间。

HBase and Schema Design



- **Schema Creation**
 - HBase schemas can be created or updated using the [The Apache HBase Shell](#) or by using [Admin](#) in the Java API.
 - Tables must be disabled when making ColumnFamily modifications, for example:

```
Configuration config = HBaseConfiguration.create();
Admin admin = new Admin(conf);
TableName table = TableName.valueOf("myTable");
admin.disableTable(table);
HColumnDescriptor cf1 = ...;
admin.addColumn(table, cf1); // adding new columnFamily
HColumnDescriptor cf2 = ...;
admin.modifyColumn(table, cf2); // modifying existing columnFamily
admin.enableTable(table);
```

我们把表全部修改完之后，再去 enable 一下。

真正在设计的时候要注意几个地方：

1. region 的尺寸有一个大小的范围，10~50GB。
2. 一个表的 region 数量也不能太多，应该控制在 50~100 个。
3. 不鼓励太长的 column family，因为名字要参与索引可能影响效率。

对于时序的 log 数据，我们必须保证在拿到时序数据之后严格按照时间排序。在做 HBase 的设计的时候，我们可以把 row key 设计为 device ID 加上时间。

当我们在写一个列族的时候，我们一定不要使用 getRow，而应该去直接拿一个列族出来。因为列族是存储在不同的地方的。

2021/12/27

jps 是 jdk 提供的一个查看当前 java 进程的小工具。HBase 也有一个 web 控制台，这个控制台它是这个样子的。

Hive

The screenshot shows the Apache HBase web interface. At the top, there is a navigation bar with links: Home, Table Details, Local Logs, Log Level, Debug Dump, Metrics Dump, and HBase Configuration. Below the navigation bar, it says "Master master". Under "Region Servers", there is a table with columns: ServerName, Start time, Requests Per Second, and Num. Regi. The table has two rows: one for "master,16020,1519988455016" with start time "Fri Mar 02 06:00:55 EST 2018" and requests per second "0", and another row for "Total:1" with requests per second "0" and num regions "3". Below the table, there is a section titled "Backup Masters" with a table having columns: ServerName, Port, and Start Time. The table has one row labeled "Total:0".

ServerName	Start time	Requests Per Second	Num. Regi
master,16020,1519988455016	Fri Mar 02 06:00:55 EST 2018	0	3
Total:1		0	3

Backup Masters

ServerName	Port	Start Time
Total:0		

Hive 是数据仓库，它把数据导入之后也就是一张张的表。它也有它的 DDL（数据定义语言）和 DML（数据操作语言）。

这节课要解决的问题：为什么我们要用 Hive？Hive 这个数据仓库和数据湖有什么关系？

我们现在大家改写完电子书店网站以后，出现的是什么情况呢？我们有一个 Tomcat 跑我们的应用，然后我们让大家用了很多数据库：MySQL、Neo4j、MongoDB、 RocksDB、InfluxDB、HBase 等。当我们数据在这个系统里的这么多地方都有的时候，我们就会遇到一些问题：

Q：我们为什么要这么多数据？

A：从技术的角度来看，我们确实需要这么多数据，比如把好友信息存到 Neo4j 中。但是从业务的角度来看，我们可能有需求要把所有的数据库中的数据放到一起去做分析。在这种情况下，我们应该怎么办呢？我们就需要一个数据仓库，注意到数据仓库也不是万能的，也是要把数据导入到其中变成一张一张的表的。它可能支持纯文本的格式导入（要符合一定的规则），

比如以 key、value 的格式，在 Hive 中是要 `ctrl+a` 隔开的）以及其他的一些约定的文件格式，可以通过这些格式导入后形成一张张的表。

现在的问题就在于这些数据库中的表是怎么映射成这些 `txt` 文件呢？那就要开发一个个的适配器（Adapter）转换成适配于 Hive 的导入格式，然后导入到数据仓库中。导入以后，大家都是一样的数据，看不出原先来源于哪个数据库，并且可以放在一起做处理了。

注意数据在从原先数据库进数据仓库的时候，要做 ETL 三个操作（清洗、转换、加载），比如过滤掉质量较差的数据、统一不同数据库中的数据单位等。

从数据仓库的角度来看，现在所有数据都在数据仓库里，我们支持在这些数据上做一些操作，使用类 SQL 语言。

那么如果我们只有一个 MySQL 数据库的话，也需要数据仓库吗？

A：也是需要的，因为我们在设计 MySQL 的时候，从技术角度拆开成表和表之间的关联。但在系统做整体分析的时候，可能还从支付宝 API 等拉过来了一条数据，这些数据放在一起的时候，也需要经过过滤转换加载。更重要的是，数据仓库中数据可能是以立体的方式建立的，所谓的搜索就是对立体做切面。那这是数据仓库要解决的问题。

schema-on-write

我们会在实践中会看到把数据清洗、转换、校验加载到了数据仓库中，那么我们什么时候去做 ETL 这个事情呢？是在业务运行过程中去做吗？

我们可以先在数据仓库中定义好数据表的结构是怎么样的。因为我们的数据库是有了 schema 再往里写数据的，所以这是 schema on write。如果表里插入的数据不符合原先的定义，那么它就进不去。所以当数据往里写的时候，就立马做校验，如果不符校验就写不进去。

站在数据仓库的角度，我们在加载进 MySQL 的表的时候，我们就要定义好表的格式是怎么样的，然后我们做清洗、转换、校验。中间如果出错了就会报错。但是数据仓库认为，如果表在加载的时候就要做这件事情，因为库可能很大，要很久才能通知用户加载完了。万一我们加载进来的表一直都没用怎么办？

schema-on-read

它就认为这样的 schema-on-write 不值，我们回忆 spark 中的 RDD 计算是 lazy 的方式，在数据仓库中的“加载文件”，它也只是记住了文件在什么地方，并没有做校验。只有有查询语句来的时候，它采取做校验，这就是 schema-on-read。这和 spark 中碰到一个宽依赖的时候倒推应该怎么做是类似的。

这就是数据仓库中的 schema-on-read 的数据加载方式，这样看起来性能好像不错了。但是数据仓库还有一个问题，在 `load` 数据进来之后，原始数据还在原来的库里，这两个数据还是处于分离的状态。

那天讲的数据湖是什么概念？数据仓库的问题就在于清洗转换后的数据之后，原数据就没了。数据湖就是说把原始的文件放在里面，它就不去做格式的转换加载，数据湖就在存原始的文件。

Q: 数据仓库和数据湖是什么关系呢?

A:

有三种关系

1. 所有的数据不要进库，全部扔到数据湖里，当有人要访问数据湖里的数据时，把它导入到数据仓库的表里进行操作，所以在这种关系中，数据湖是数据仓库的数据源。数据湖可以是一个分布式的文件系统，在此基础之上做一些文件的管理。
2. 在不同格式的数据库上，每一个都配一个 SQL，转换成它原生操作的查询语言，也可以提供这样的功能。用户的 SQL->SQL_converter->原生查询语言。这样我们底层就是异构的数据库，对上层暴露的就是相同的接口，这样用户就好像站在数据湖上，他看到的所有数据都可以用 SQL 来访问，甚至可以跨数据库来访问。把数据湖的功能来做强，这样我们就把数据湖来当做数据仓库，就不再需要数据仓库了。但是这种带来的问题就是它的性能肯定要比数据仓库的查询要差一点。
3. 你的操作原先确实是通过 SQL converter 转换成原先的查询语句来做，等到查询频率超过一定的时候，就转换到数据仓库中，这就等于热数据在数据仓库中，而冷数据在数据湖中。

现在讲湖仓一体（lake house）的存储，它就没有标准的协议只有参考的实现。

Hive 原先只是 hadoop 的组件，它的目标就是做大数据的分析。它是把数据 load 进来之后变成表，实际上还是在它的表的结构上操作。它必须依赖于 hadoop 的 hdfs 上。它提供的查询是类似 SQL，所以本质上 hive 想把异构的数据进行清洗、转换、加载放到数据仓库里，让那些熟悉 SQL 的人用类 SQL 语句来对数据做 操作。

Hive 的几个特点非常明显：基本上所有的 SQL 的操作都是用 map reduce 的操作来执行的，也就是在多个节点上同时执行作业。

我们需要记录每张表里有些什么，数据可能要做分区存储在不同的 hdfs 节点上。关于表的结构信息就存在 metastore 里。

我先要把 hadoop 的 hdfs 跑起来，必须用 jps 来检查一下 name node 和 data node 是否都跑起来了。

我们之前提到，hdfs 它有一个 name node 来记录一些元数据（某一个文件分了几块，存在哪），它存了几个 datanode，客户端在真正访问数据的时候不经过 name node。

Hive 要用 hdfs 存数据，所以要创建一个存储的地方。

- Hive uses Hadoop, so:
 - you must have Hadoop in your path OR
 - `$ export HADOOP_HOME=<hadoop-install-dir>`
- Start NameNode daemon and DataNode daemon:
 - `$ sbin/start-dfs.sh`
- In addition,
 - you must use below HDFS commands to create `/tmp` and `/user/hive/warehouse`
 - (aka `hive.metastore.warehouse.dir`) and set them `chmod g+w` before you can create a table in Hive.
 - `$ HADOOP_HOME/bin/hadoop fs -mkdir /tmp`
 - `$ HADOOP_HOME/bin/hadoop fs -mkdir /user/hive/warehouse`
 - `$ HADOOP_HOME/bin/hadoop fs -chmod g+w /tmp`
 - `$ HADOOP_HOME/bin/hadoop fs -chmod g+w /user/hive/warehouse`

我们需要 `tmp` 目录做一些中间的处理，去做清洗转换，最后再加载到 `warehouse` 中。
注意到两个目录是要建立在 `hdfs` 中，而不是在本地的文件系统中。

准备工作做完以后就可以跑 `hive`。

- To use the Hive [command line interface](#) (CLI) from the shell:
 - `$ HIVE_HOME/bin/hive`
- **Running HiveServer2 and Beeline**
 - Starting from Hive 2.1, we need to run the `schematool` command below as an initialization step. For example, we can use "derby" as db type.
 - `$ HIVE_HOME/bin/schematool -dbType <db type> -initSchema`
- [HiveServer2](#) (introduced in Hive 0.11) has its own CLI called [Beeline](#).
 - HiveCLI is now deprecated in favor of Beeline, as it lacks the multi-user, security, and other capabilities of HiveServer2. To run HiveServer2 and Beeline from shell:
 - `$ HIVE_HOME/bin/hiveserver2`
 - `$ HIVE_HOME/bin/beeline -u jdbc:hive2://$HS2_HOST:$HS2_PORT`

可以通过简单的 `hive` 命令启动，也可以通过 `HiveServer2+Beeline` 的方式来启动。

我们都要用 `HIVE_HOME/bin/schematool -dbType <db type> -initSchema`。设置把 metadata 放在哪个数据库里。

`Hive` 对于大多数请求会用 `mapreduce`，对于小文件的查询可能会退化为 `local` 的请求。

它使用的方法是和 `SQL` 类似的：

The Hive DDL operations are documented in [Hive Data Definition Language](#).

Creating Hive Tables

- `hive> CREATE TABLE pokes (foo INT, bar STRING);`
- creates a table called `pokes` with two columns, the first being an integer and the other a string.
- `hive> CREATE TABLE invites (foo INT, bar STRING) PARTITIONED BY (ds STRING);`
- creates a table called `invites` with two columns and a `partition column` called `ds`.
- The partition column is a `virtual` column. It is `not` part of the data itself but is derived from the partition that a particular dataset is loaded into.
- By default, tables are assumed to be of text input format and the `delimiters` are assumed to be `^A(ctrl-a)`.

它是在分布式存储上来存储的，在 MySQL 中要做分布式存储的时候要用 `partition`。在 hive 里也可以按照某列来做 `partition`。比如上例第二行中，就是按照 `ds` 这个虚拟的列来做 `partition`。在 `table` 里加载数据的时候，要用 `ctrl-a` 隔开。还有另外一种流行的格式，可以让数据库全部转成那个格式再导入。

Altering and Dropping Tables

- Table names can be [changed](#) and columns can be [added or replaced](#):
- `hive> ALTER TABLE events RENAME TO 3koobecaf;`
- `hive> ALTER TABLE pokes ADD COLUMNS (new_col INT);`
- `hive> ALTER TABLE invites ADD COLUMNS (new_col2 INT COMMENT 'a comment');`
- `hive> ALTER TABLE invites REPLACE COLUMNS (foo INT, bar STRING, baz INT COMMENT 'baz replaces new_col2');`
- Note that `REPLACE COLUMNS` replaces all existing columns and `only` changes the table's schema, `not` the data.
- The table must use a native SerDe. `REPLACE COLUMNS` can also be used to drop columns from the table's schema:
- `hive> ALTER TABLE invites REPLACE COLUMNS (foo INT COMMENT 'only keep the first column');`
- Dropping tables:
- `hive> DROP TABLE pokes;`

Metadata Store

- Metadata is in an [embedded Derby database](#) whose disk storage location is determined by the Hive configuration variable named `javax.jdo.option.ConnectionURL`.
 - By default this location is `./metastore_db` (see `conf/hive-default.xml`).
- Right now, in the default configuration, this metadata can only be seen by one user at a time.
- Metastore can be stored in any database that is supported by [JPOX](#).
- The location and the type of the RDBMS can be controlled by the two variables `javax.jdo.option.ConnectionURL` and `javax.jdo.option.ConnectionDriverName`.
- In the future, the metastore itself can be a standalone server.
- If you want to run the metastore as a network server so it can be accessed from multiple nodes, see [Hive Using Derby in Server Mode](#).

Metadata store 存了有哪些表，表的结构是怎么样的。

The Hive DML operations are documented in [Hive Data Manipulation Language](#).

- Loading data from flat files into Hive:
- `hive> LOAD DATA LOCAL INPATH './examples/files/kv1.txt' OVERWRITE INTO TABLE pokes;`
- Loads a file that contains two columns separated by `ctrl-a` into `pokes` table.
- '`LOCAL`' signifies that the input file is on the local file system. If '`LOCAL`' is omitted then it looks for the file in HDFS.
- The keyword '`OVERWRITE`' signifies that existing data in the table is deleted. If the '`OVERWRITE`' keyword is omitted, data files are appended to existing data sets.

NOTES:

- `NO` verification of data against the schema is performed by the load command.
- If the file is in hdfs, it is moved into the Hive-controlled file system namespace.
- The root of the Hive directory is specified by the option `hive.metastore.warehouse.dir` in `hive-default.xml`. We advise users to create this directory before trying to create tables via Hive.

我们要从本地文件系统加载数据，也支持这个数据在 hdfs 里加载数据。

The screenshot shows the Hadoop Web UI interface. A modal window titled "File information - kv1.txt" is open over a "Browse Directory" page. The modal contains sections for "Block information" (Block 0) and "File contents". The "Block information" section displays details such as Block ID (1073741825), Block Pool ID (BP-591783681-127.0.0.1-1640354044142), Generation Stamp (1001), Size (5812), and Availability (localhost). The "File contents" section shows the file's raw data, which consists of several lines of binary values.

这就代表数据导入成功了，数据导入成功后就可以对其进行操作。

Data Load

- `hive> LOAD DATA LOCAL INPATH './examples/files/kv2.txt' OVERWRITE INTO TABLE invites PARTITION (ds='2008-08-15');`
- `hive> LOAD DATA LOCAL INPATH './examples/files/kv3.txt' OVERWRITE INTO TABLE invites PARTITION (ds='2008-08-08');`
- The two LOAD statements above load data into **two different partitions** of the table **invites**.
- Table **invites** must be created as partitioned by the key **ds** for this to succeed.

也可以把数据导入到不同的分区里，到时候我们可以在不同的分区里单独做搜索。

The screenshot shows the Hadoop Web UI interface. A "Browse Directory" page is displayed with the path "/user/hive/warehouse/invites". The directory listing table has columns: Name, Owner, Group, Size, Last Modified, Replication, and Block Size. Two entries are shown: "ds=2008-08-08" and "ds=2008-08-15". Both entries have the same details: Owner: chenhaopeng, Group: supergroup, Size: 0 B, Last Modified: Dec 24 22:09, Replication: 0, and Block Size: 0 B. The "Name" column for both entries is highlighted with a red box.

Note that in all the examples that follow, **INSERT** (into a Hive table, local directory or HDFS directory) is optional.

- `hive> INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out' SELECT a.* FROM invites a WHERE a.ds='2008-08-15';`
- selects all rows from partition ds=2008-08-15 of the invites table **into an HDFS directory**.
- The result data is **in files (depending on the number of mappers)** in that directory.
NOTE: partition columns if any are selected by the use of *. They can also be specified in the projection clauses.
- Partitioned tables must always have a partition selected in the **WHERE** clause of the statement.

把执行结果放到目录下，这个目录里可能有多个文件。因为在搜索的时候是使用 MapReduce，取决于我们用了几个 reducer。

The screenshot shows the Hadoop Web UI interface. On the left, there's a 'Browse Directory' panel with a red box highlighting the path '/tmp/hdfs_out'. It lists one entry: '000000_0'. Below it, there's a message 'Showing 1 to 1 of 1 entries' and the date 'Hadoop, 2021.'. On the right, a modal window titled 'File information - 000000_0' is open. This window contains several tabs: 'Download', 'Head the file (first 32K)', and 'Tail the file (last 32K)'. The 'Block information - Block 0' tab is selected, showing details like 'Block ID: 1073741832', 'Block Pool ID: BP-591783681-127.0.0.1-1640354044142', 'Generation Stamp: 1008', 'Size: 11291', and 'Availability: localhost'. The 'File contents' tab shows a list of 11 rows, each containing a timestamp and a value. The 'File contents' table has columns 'Time' and 'Value'. The last row is '135val_1362008-08-15'. At the bottom of the modal is a 'Close' button.

此时文件比较小，我们就只使用了 1 个 reducer，所以只产生了一个 000000_0 文件。

SELECTS and FILTERS

- `hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/local_out' SELECT a.* FROM pokes a;`
 - selects all rows from pokes table into a local directory.
-
- `hive> INSERT OVERWRITE TABLE events SELECT a.* FROM profiles a;`
 - `hive> INSERT OVERWRITE TABLE events SELECT a.* FROM profiles a WHERE a.key < 100;`
 - `hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/reg_3' SELECT a.* FROM events a;`
 - `hive> INSERT OVERWRITE DIRECTORY '/tmp/reg_4' select a.invites, a.pokes FROM profiles a;`
 - `hive> INSERT OVERWRITE DIRECTORY '/tmp/reg_5' SELECT COUNT(*) FROM invites a WHERE a.ds='2008-08-15';`
 - `hive> INSERT OVERWRITE DIRECTORY '/tmp/reg_5' SELECT a.foo, a.bar FROM invites a;`
 - `hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/sum' SELECT SUM(a.pc) FROM pcl a;`
 - selects the **sum** of a column. The **avg**, **min**, or **max** can also be used.

因为原始数据在各个数据库里，所以我们对 hive 里的数据做修改是没有任何意义的。所以通常只有 Load 操作。

这是官网上的一个例子，用户对电影的评分记录：用户 id: 电影 id: 打分: 时间戳。

MovieLens User Ratings

- First, create a table with tab-delimited text file format:
- `CREATE TABLE u_data (`
- `userid INT,`
- `movieid INT,`
- `rating INT,`
- `unixtime STRING)`
- `ROW FORMAT DELIMITED`
- `FIELDS TERMINATED BY '\t'`
- `STORED AS TEXTFILE;`

文本文件中每一列是用 tab 隔开的。

The screenshot shows the Ambari interface for a Hive database. A modal window titled "File information - u.data" is open over a "Browse Directory" view. The "File contents" section of the modal is highlighted with a red box, showing the following data:

userid	movieid	rating	unixtime
196	242	3	891250949
186	202	3	891717743
22	377	1	878687116
244	51	2	880606023
166	346	1	886397596
298	474	4	884182806
115	265	2	881171488
253	465	5	891628467

Now we can do some complex data analysis on the table `u_data`:

Create `weekday_mapper.py`:

```
import sys
import datetime
for line in sys.stdin:
    line = line.strip()
    userid, movieid, rating, unixtime = line.split('\t')
    weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()
    print ('\t'.join([userid, movieid, rating, str(weekday)]))
```

它支持用 python 写 map reduce 的代码丢到 hive 里去执行。

```

Use the mapper script:
CREATE TABLE u_data_new (
  userid INT,
  movieid INT,
  rating INT,
  weekday INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t';

add FILE weekday_mapper.py;

INSERT OVERWRITE TABLE u_data_new
SELECT
  TRANSFORM (userid, movieid, rating, unixtime)
  USING 'python weekday_mapper.py'
  AS (userid, movieid, rating, weekday)
FROM u_data;

SELECT weekday, COUNT(*)
FROM u_data_new
GROUP BY weekday;

```

我们可以看到其中的'python weekday_mapper.py'就是用这个 python 文件来对 new data 中的每一行做转换。如果这个文件很大，那么真的把这个 python 的逻辑当做多个 mapper 的逻辑去进行操作。

2021/12/30

最后我们要拿一个例子来讲一下在实际的场景之下是怎么样的，我们用去年和中远海运合作的报告做的东西。

中远海运的例子可能涉及到版权问题，略。

复习

我们不能搞到最省的 singleton。大二的时候应用在访问 Mysql 的时候都是同步访问的方式。我们在对象上调用 `o.set(field, value);` 它带来的问题就是当这句话执行完之前，代码就不能执行下去，我们的这个线程就必须阻塞在这里。但是当有大量的用户同时访问的时候，连到 MySQL 的连接只有 30 个，就会有大量的线程阻塞和超时，所以我们需要有异步的通讯方式，也就是我们要把执行的操作建立下来，然后事后我们去处理，前提是事后有算力去处理。但是如果请求越堆越多那就会有问题。所以消息队列能解决问题的基本问题就是平峰的时候有闲暇的时候，异步的方式带来的是响应的速度快了，用它来缓解我们的压力。回答“我收到了，你可以去做自己的事情了。”所以我们有一部分操作是同步的，有一部分操作是异步的，所以这就是用消息机制去做订单的概念。

事务可以用描述性的方式在那里写了很多 annotation。误解：`request_new` 是一个嵌套型的事务，如果 A 的方法里调用了 C,D，其中 A 是 `not Supported`，C 是 `requireNew`。嵌套事务的前提是 A 是 `mandatory`、`required`、`requireNew`，也就是强制 A 这里有新的事务的时候，

C 这里才开了一个新的事务去执行。

其实 C 和 A 根本不是一回事。C 是把原先的事务挂起，执行完再提交掉。嵌套型事务的意思就是，我们允许里面的事务尝试十次再继续往下执行。我们要控制事务的边界，本质就是在靠数据库中的大量的缓存来满足的我们的要求。为了提高性能，我们把程序变成多线程的，它要解决的问题就是多个写操作会导致状态不一致，那么我们就要去加锁。

锁有很多种，一种是显式定义的，还有一个是每个对象都是一个锁。要用 `wait` 和 `notify` 机制去释放掉已经有的锁，让别人去等。总的来说，系统中没有写操作就简单了。这也是 spark 中的 RDD 只能写一次，可以读多次的原因。

再往下，我们就说要用 `cache`。用 `redis`, `memorycache` 来做缓存，本质上就是不要让数据每一次从硬盘上读操作，以后访问的时候就可以直接从缓存中读出来。还要注意的就是说，在分布式的环境中，我们需要使用一致性哈希去做。因为我们存和取的时候用的是同一个哈希算法，只要 `key` 是我们想要的，那么就一定可以 `hash` 到拥有数据的机器上去。包括 `redis` 可以缓存 `http session`，我们需要有一个单一的地方去存 `http session`，这样用户就不用担心负载均衡后 `session` 没有了。

再往下我们讲了一下 `webservice`，有 `soap`（事件驱动）和 `restful`（数据驱动）。要解决异构的问题，也就是开发的 C# 要被 java 调用，为什么要用 `web` 协议，因为存在跨域的问题。

在 `webservice` 中为什么要使用微服务？微服务是一种概念，每个服务带自己的存储和缓存，单独就是一个应用系统，可以分成几个独立的系统去分布，一个 `crash` 了，另一个还能用，还可以有针对性地去加实例，可以在大型集群中让资源被更好的利用。在微服务里，在预先编好的代码里不知道在什么位置，所以微服务就是注册到注册中心里，其实就是存了一些 `key: value` 对。所以用户会发请求到 `gateway`，然后到注册中心中找到 `value` 转发到对应端口。

这样服务器端的内容就已经结束了，已经比较强大了。

后面我们过渡到了数据库上，从 `Mysql` 去深挖了四节课，比如数据库中的缓存是怎么去管理的，比如它要记住所有打开的表有哪些。还有就是真正 `load` 进来的索引和数据，也要有一块缓存，设的太大有什么问题？设的太小有什么问题？我们必须设置 `8G->10G`，没有 `9G` 的概念。

我们最先讲的是表的设计，一张表应该设计什么字段，太大应该用 `text` 和 `blob`，每个字段多少字节，一个库做多有几个表。然后我们讲数据库的备份，有逻辑的备份从操作备份，还有一种是把文件系统中的表全部 `copy` 出来，比较省空间，但是要复制出所有文件出来，可能不能跨版本兼容。

然后我们讲了 `partition`，上节课说了按照某一列做某一种形式的 `partition` 到底合不合适，本质上我们就要去判断 `partition` 对读写性能的提高有没有帮助。

讲完 `mysql` 以后就扩展出来了很多 `nosql` 数据库。

1. `MongoDB` 里存的是 `json`，面向文档的数据库，表叫做 `collection`，三星和苹果手机的例子就告诉我们，对 `schema` 的要求不严格，大家的字段可以不一样。`MongoDB` 也可以支持 `sharding`，把数据切成一个个 `block`，最后落到了一个个的 `sharding server`，并且保证 `block` 的数量只差 2.

2. neo4j，存储图数据。在真实生活中大量数据呈现出来的是图数据。如果要做好好友的好友的查询的时候，就比较简单，速度比较快。

3. 日志结构合并树，这种数据针对的是数据是按照时序并且有冷热差异的数据库，后来又降了 influxDB，我们都谈到了 WAL(write-ahead log)，要想执行写操作，先把日志记录下来。在 LSM 中，是没有空间放大的，把数据都放到对应层。有写放大和读放大（查找不到的要查找到做好）。InfluxDB 为了压缩数据，设计了特殊的格式，有 tag 和 field。

4.

这些东西都有了之后，就会发现系统中的数据库的数量会很多。我们就谈到了数据湖的应用。这样我们的数据可以存到很多地方了。接下来我们谈到了应用服务器是怎么建立集群的，一个最大的问题是会话状态怎么维护。

我们也可以不用 nginx 的方式，用 gateway 调度的方式。集群的目的就是在分布式环境中，让不同的机器承担这个负载。Mysql 的主从备份。

我们只有一台服务器怎么跑呢？我们就跑 docker，看起来就好像跑了很多节点一样。一旦容器化部署以后，在集群内要有一个管理的功能，我们简单聊了一个 K8S。我们再介绍了 cloud，又介绍了一些边缘计算的。

讲完了以后，我们又回过头来，这些数据在集群中做数据分析的时候，我们讲了一些框架 hadoop，它里面最经典的就是 MapReduce。它虽然很经典，但是它效率不高，它在每一个阶段是怎么去读写硬盘的，后来就出现了内存计算框架 spark。但是内存是一维的，管理比较复杂，就出现了 RDD 这种分布式的弹性数据集。在这个时候我们讲的还是批数据的处理，如果要走流式数据，就需要 storm。

讲完之后，我们只讲了 hadoop 中资源的调度方式。我们后来又讲了 hdfs（把大文件切成很多小文件落到本地文件系统中），在 hdfs 之上出现了 hbase 和 hive。一个是数据库，一个是数据仓库，就是要来做大数据的数据处理。Hive 强调的是，它在加载数据的时候是不做校验的，用的是 read-on-schema。有了这些东西之后，所有东西都可以构建在 hdfs 之上。

到这里就是我们上课的所有内容。