目录

A. Java basic knowledge 9
1. Immutable Object 9
2. Java 4 types of references 9
3. Composition and inheritance 13
4. three characters of object-oriented language 15
3.1 encapsulation 15
3.2 Inheritance 15
3.3 Polymorphism 15
5. private,protected and public keywords. 16
6. clean code principle 16
7. code review 16
8. IOC and DI 17
9. hashmap hashset 17
10. linkedhashmap 19
11. interface is better than abstract class because of scalability. 19
12. Unit Test vs Integration Test 19
13. JVM this definition need to be changed 19
14. generic eraser 20
15. JVM-class load process 20
16. Talk about the JVM runtime data area 21
17. static keywords 21
18. Can static method be inherited and override? 21
19. transient keyword 22
20. java clone 22
21. checkedexception and uncheckedexception 23
22. Why is char[] preferred over String for passwords? 24
23. String questions 24
B. design pattern 25
1. What are SOLID principles 25
1.1 Single responsibility principle 25
1.2 Open close Principle: 25
1.3 Dependency Inversion Principle 26
1.4 Interface Segregation Principle: 29
1.5 Liskov Substitution Principle: 30
1.6 Low of demeter principle: 31
2. Describe some typical design pattern? 32
1 Decorator: 32
2 Singleton: 32
3 Factory: 33
4 Builder: 34
5 Strategy: 34
6 delegate design 37
7 Observer Pattern 37

8 Proxy Pattern 37
9 Adapter Pattern 37
10 composite design.(it is not composite which is compartment of inheritance)38
C. Java version 8 new features 38
D. JVM 40
1. Difference between Thread dumps and Heap dumps 40
2. Out of memory and memory leak 40
3. How does jvm judge a object can be collected 41
4. If a Java process has a full gc every 10 minutes, what can I do 41
5. GC algorithms 41
6. TAKE AN example of how you did fix a jvm issue ? 41
E. Data Structure and sql 42
1. B and B+ Differences between B and B+ trees 42
2. Red black tree 45
3. AVL tree 46
4. binary heap 46
5. graph 47
6. minimum spanning tree 47
7. Quick sort(fasted sorting algorithm) 47
8. merge sort 47
9. Time complexity of quick sort and merge sort , and why JDK chose quick sort . 48
10. BloomFilter 48
Strengths: 49
11. sweep line explanation 52
12. differences among sorts alg 52
13. What is Trie 52
14. Rolling Hash 53
15. KMP 53
F. Distributed System(microservices) 54
1. Distributed system 54
2. What is Consistency in CAP theory 55
3. 服务发现 api gateway 链路追踪，监控，分布式事务 55
3.1 Service discovery: 55
3.2 api gateway: 57
3.3 Monitor: 57
3.4 whats is Distributed transaction: 有什么框架?,how to implement? 57
4. Rpc framework dubbo 57
5. Rpc and http 58
6. Montior responsibility 59
7. TCP and UDP 60
G. Redis 60
1. redis datastructure 60
2. redis usage scenarios 60
3. Consistency issue or Cache Invalidation Strategies 61

https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/64224412a763c048c9763975 61
4. redis and mongoDB 61
5. how to solve REDIS DB inconsistency 61
6. all questions about REDIS 64
7. why does redis so rapid 64
8. what is skip list 65
9. simple dynamic string 65
10. Why does redis use hash index 66
11. redis 数据持久化,可用性 67
12. Cache Replacement Policies 67
13. redis 常见故障 68
H. Kafka or MQ 70
1. How to ensure that messages are not lost 70
2. How to deal with duplicated Messages 72
3. How to ensure the order of the message 72
4. How do you handle message piling 74
5. 什么是死信队列 Dead letter QUEUE 74
6. Messaging patterns 77
7. Kafka structure and interview questions 78
8. RabbitMQ vs. Kafka vs. ActiveMQ: A Battle of Messaging Brokers 78
9. kafka 原理 79
10. Kafka 架构设计 81
11. Kafka 为什么性能这么好 84
12. Kafka 怎样处理消息丢失 86
I. Devops and CI/CD 86
1. CI /cd pipeline 86
1.1 how to do performance test 86
1.2 how to do integraton test 87
2. Auto test 87
3. unit test code coverage rate 88
4. What's the difference between a blue/green deployment and a rolling deployment? 88
5. What are the differences between continuous integration, continuous delivery, and continuous deployment? 88
6. What is the difference between a Docker image and a container? 89
7. What is Docker 89
8. Docker 90
8.1 What is docker: 90
8.2 What is k8s 91
8.3 what is pod 92
8.4 what is service 92
8.5 container and pod 92
J. AWS 93

1. ec2 auto scaling type;ELB 93
2. Lambda serveless 93
3. Docker 93
4. S3 type 94
5. API Gateway 94
6. S3 and EFS 95
三类存储对比表 95
7. available zone 97
8. what will happen , if ec2 crashed 97
9. availability zone , VPC, region 97
10. what is ELB 97
K. MultiThread 97
1. Differences of Class Lock and Object Lock and Method Lock 98
1.1 Method Lock is the same as Object Lock 98
1.2 Class Lock 98
2. atomic visibility reorder 98
3. join 102
4. Priority 102
5. yield 102
6. Thread States 102
7. what happens if a thread throw an exception 103
8. What is dead lock 103
9. How to avoid dead lock 104
10. What is blocking queue 105
11. Countdownlatch/CyclicBarrier/BlockingQueue 105
12. readwriteLock 107
13. Difference between sychronized and lock 108
14. when to use and not use concurrenthashmap 108
15. What does re-entrant mean(sychronized) 110
16. CAS 110
17. Future(callable) 111
18. Threadpool 111
19. #####thread safe Class##### 112
20. what is threadlocal 112
21. Volatile 和 final 112
22. methods of creating and Start Thread 112
20.1 extends Thread 112
20.2 implements Runnable 112
20.3 inmplements callable : which can get a result by return sth 112
23. 多线程操作中为什么使用 while 而不是 if 来做判断状态是否就绪 114
24. 对象发布逸出 115
25. 如何用一个互斥锁实现，读写锁？ 115
26. A typical thread safe code 117
27. junit 测试多线程安全性 118

L. Spring and spring boot 119
1. log level 119
2. AOP 119
3. ACTUATOR 120
4. SCHEDULER @EnableScheduling 121
5. spring boot advantages or different from spring 121
6. What are starter dependencies? 121
7. Spring boot common annotations? 122
8. Spring Security 122
9. Swagger 123
10. how to share a common module and how to setup in parent pom 123
11. generate object according to table name 124
12. Spring 事件解耦 124
13. spring transaction 事务失效 124
1 Private 必须 public 修饰 125
2 final or static 125
3 method called within a class 125
4.未被 spring 管理(Not Managed By Spring) 127
5 多线程调用 127
6 表不支持, DB doesnot support transaction 127
14. Spring 事务不回滚 128
1 设置了错误的事务传导机制 128
4 自定义回滚异常，但是实际上抛出了其他异常 129
15. Spring 事务隔离级别 132
16. 事务的嵌套 132
17. Spring DI 的方式有哪些 132

变量注入（Field Injection） 132

构造器注入（Constructor Injection） 133

setter 方法注入 （Setter Injection） 133

18. Spring bean 如何创建 134
19. Spring bean 生命周期 135
instantiation 136
20. Spring bean scope 136
21. IOC and DI 137
22. bean 的懒加载和提前加载 137
23. Spring 怎样避免循环依赖 137
24. spring MVC 138
25. 拦截器 过滤器 138
M. WEB 138
1. Get post difference 138
2. SOAP Web service 138

3. how does two processes communicate 139
4. Http Code 139
500 Internal Server Error 139
5. n-tier architecture 139
6. RPC Vs REST :Understanding RPC Vs REST For HTTP APIs 140
7. Restful 优势 141
8. Security vulnerabilities 141
8.1 What is cross-site scripting (XSS) attacks 142
8.2 Use encryption (e.g., HTTPS) to secure communication between the client and server, protecting sensitive information and preventing eavesdropping. . 142
9. websocket and http(long polling) 142
10. Cookie session and token 144
11. difference between http and https 144
12. tcp and udp 145
13. Tcp three times shake hands and why is tcp secure 145
N. DATABASE 145
1. Sql T Y P E S 145
2. Sql tuning,improvement 146
3. 3NF in table design 146
4. Join 150
5. DB triggers 151
6. syntax 152
7. ACID 152
7.1 Isolation 152
7.2 Atomicity and consistency 154
mysql 怎样实现隔离性 持久性 原子性 155
8. Mysql primary/secondary replication 156
8.1 when is primary/secondary replication necessary 157
8.2 how to code if primary/secondary replication is underway 158
9. index 索引问题 158
Single and Multiple Column Indexes 158
Five Types of Indexes 159
10. 聚簇索引和非聚簇索引 160
11. B＋树的完美解释，聚簇索引和非聚簇索引 161
12. Differences between sql and nosql 166
13. describe scenarios in which indexes are not effective 169
14. mysql 语句执行流程 169
O. Bug position or performance improvement tool 172
1. how do you know where the performance issue occurs 173
2. How to monitor API usage, load, like TPS 173
3. How to position a bug by a tool 173
P. Behavior questions 173
4. What will you do when you have a different opinion/ conflict with your colleagues

in your work? 173
5. work alone or work in a team 174
6. A challenging project 174
http-proxy-middleware 174
7. what makes you angry? 174
8. your ideal manager 175
9. a time persuading others. 175
10. Different opinion in the meeting 175
11. another conflict story 176
12. What would you be a good fit for this position? How would you affect our culture? 176
Q. Do you have any questions for me? 176
1. Why is this position open — is this a new role? If not, why did the person who held this role leave it? 176
2. If I were hired for this role, what would you want me to achieve in my first months in the position? 176
3. What's one thing that makes you happy to come into work today? 177
4. Can you talk about the company culture a bit? 177
5. Career path 177
R. Why our company 177
1. This position in your company is a good match for my plan. At the same time, I am very optimistic about the industry of your company. I think your company attaches great importance to talents, and this job is very suitable for me. 177
2. Must be familiar with those industries 177
S. Tools 177
1. idea 好用的插件 177
2. Ahportal Dynatrace 177
3. performance review through a tool 178
4. monitor and alerts tool 178
5. how to position bug through a tool 178
6. how to do unit test and integration test through a tool 178
T. difficulties fixed in one project 178
U. Fundamental of System design 179
1. Key characteristics of distributed system 179
2. API gateway 179
3. DB partition(分区) and sharding(分库) 179
1.0 Critical English Version: 179
1.1 Why we need to use sharding 179
1.2 分库(sharding)分区(partition) 180
1.3 Types of partition: 180
1.4 partition 后潜在问题的解决方式 180
4. redis and mongoDB(怎样设计点赞系统) 181
5. TPS QPS 182

6. Latency and Throughput 182
2 Latency: 183
7. LOAD BALANCER 183
8. distributed consensus 183
8.1 what is consensus problem: 184
8.2 Types of consistency 184
8.3 General solution to consistency issue--distributed transaction 185
8.4 What is a two-phase commit? 185
8.5 Saga pattern(communication or emitting message by MQ) 186
Advantages of the Saga pattern 188
8.6 which framework for distributed transaction? 188
8.7 Zookeeper usage 188
8.8 how to achieve consensus 189
8.9 How does Mysql achieve consistency 192
8.10 Qurom to make consistency 192
9. what is map reduce 192
10. what is rate limiter 193
The Oracle and the Timer Wheel 195
11. Polling and web socket 197
12. SQL and NOSQL, scenarios to choose 198
13. what is consistent hashing, usage 199
14. Long-Polling vs WebSockets vs Server-Sent Events 199
15. Distributed message Queue 199
16. caching 200
17. batch process and stream process 200
18. network protocols 200
19. Leader elections 200
V. System Design Examples 201
1. Design a two Player Online Chess Game 201
2. Design payment system 201
3. How to design a huge file upload system including some query functions 201
1 先用 spring batch 导入数据库 201
2 上传到 s3, ... 201
AWS 建议超过 100M 的就分段上传 201
4. How to design a stock exchange system 202
5. how to design a real time stock trading system 202
6. how to design a park garage 202
7. how to design a IAM system 202
W. Big Data Processing 203
1. SpringBoot 实现 Mysql 千万级数据量导出并避免 OOM 的解决方案 203
1 pagination 203
2 Streaming(best) 204
3 Cursor query 205
2. all questions 206

3. 怎样进行上千万数据的插入和更新 206
4. differences between batch processing and stream processing 206
4 Find duplicates in large file like 1T 207
X. linux command ..... 错误！未定义书签。

A. Java basic knowledge

1. Immutable Object

Immutable Objects mean that once an object is created, **its state (the object's data, i.e., the object's property values) cannot be changed.** Any changes to it should result in a new object.so no setter methods could be provided.

1.1 Don't provide "setter" methods — methods that modify fields or objects referred to by fields.

1.2 Make all fields final and private, if the field is reference one , it should be immutable as well

1.3 Don't allow sub classes to override methods

1.4 Special attention when having mutable instance variables

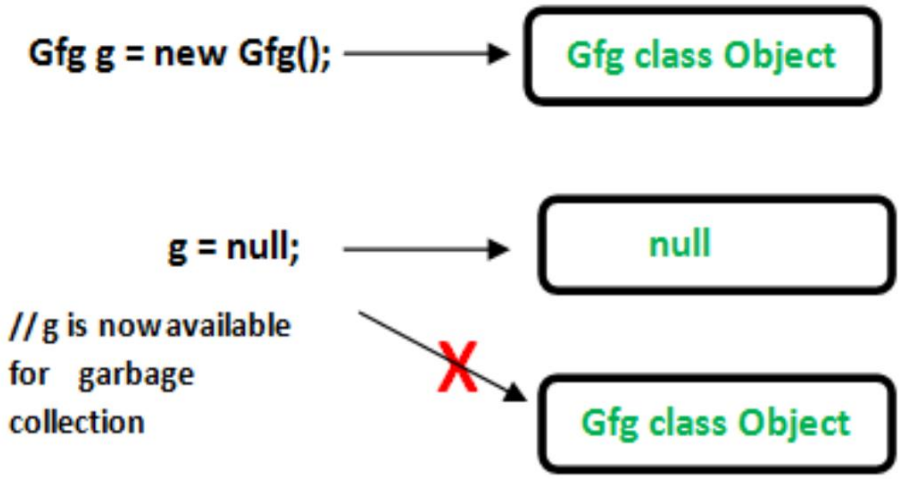
2. Java 4 types of references

https://blog.csdn.net/l540675759/article/details/73733763

English
https://www.geeksforgeeks.org/types-references-java/

Strong References:

This is the default type/class of Reference Object. Any object which has an active strong reference are not eligible for garbage collection. The object is garbage collected only when the variable which was strongly referenced points to null.



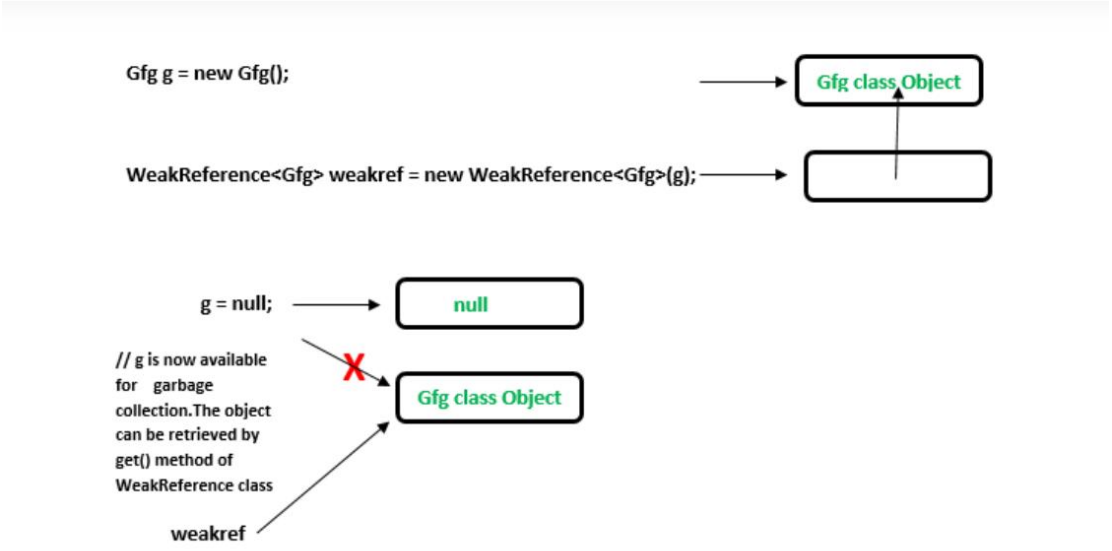
Soft References:

In Soft reference, even if the object is free for garbage collection then also its not garbage collected, until JVM is in need of memory badly.T**he objects gets cleared from the memory when JVM runs out of memory.(difference from weak references)T**o create such references java.lang.ref.SoftReference class is used.

Weak References:

Weak Reference Objects are not the default type/class of Reference Object and they should be explicitly specified while using them.

- This type of reference is used in WeakHashMap to reference the entry objects .
- If JVM detects an object with only weak references (i.e. no strong or soft references linked to any object object), this object will be marked for garbage collection. **当 JVM 进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。**
- To create such references java.lang.ref.WeakReference class is used.
- These references are used in real time applications while establishing a DBConnection which might be cleaned up by Garbage Collector when the application using the database gets closed.




```java
public static void main(String[] args)
{
    // Strong Reference
    Gfg g = new Gfg();
    g.x();

    // Creating Weak Reference to Gfg-type object to which 'g'
    // is also pointing.
    WeakReference<Gfg> weakref = new WeakReference<Gfg>(g);

    //Now, Gfg-type object to which 'g' was pointing earlier
    //is available for garbage collection.
    //But, it will be garbage collected only when JVM needs memory.
    g = null;

    // You can retrieve back the object which
    // has been weakly referenced.
    // It successfully calls the method.
    g = weakref.get();

    g.x();
}
```

Typical usage for avoiding memory leak:

内存泄漏的一个常见来源是缓存，一旦你把对象引用放入到缓存中，就很容易遗忘。比如:之前项目在一次上线的时候，应用启动奇慢直到夯死，就是因为代码中会加载一个表中的数据到缓存(内存)中，测试环境只有几百条数据，但是生产环境有几百万的数据。|

对于这个问题，可以使用 WeakHashMap（弱引用）代表缓存，此种 Map 的特点是，当除了自身有对 key 的引用外，此 key 没有其他引用那么此 map 会自动丢弃此值。

So it is a good solution to use weakhashmap to be a cache, in this way, every time gc runs, the keys in weakhashmap without outside references will be removed

```
public class MapTest {
    static Map wMap = new WeakHashMap();
    static Map map = new HashMap();

    public static void main(String[] args) {
        init();
        testWeakHashMap();
        testHashMap();
    }

    public static void init() {
        String ref1 = new String("obejct1");
        String ref2 = new String("obejct2");
        String ref3 = new String("obejct3");
        String ref4 = new String("obejct4");
        wMap.put(ref1, "cacheObject1");
        wMap.put(ref2, "cacheObject2");
        map.put(ref3, "cacheObject3");
        map.put(ref4, "cacheObject4");
        System.out.println("String 引用 ref1，ref2，ref3，ref4 消失");

    }

    public static void testWeakHashMap() {

        System.out.println("WeakHashMap GC 之前");
        for (Object o : wMap.entrySet()) {
            System.out.println(o);
        }
        try {
            System.gc();
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("WeakHashMap GC 之后");
        for (Object o : wMap.entrySet()) {
            System.out.println(o);
        }
        System.out.println("wMap 中无数据");
```

```
    }

    public static void testHashMap() {
        System.out.println("HashMap GC 之前");
        for (Object o : map.entrySet()) {
            System.out.println(o);
        }
        try {
            System.gc();
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("HashMap GC 之后");
        for (Object o : map.entrySet()) {
            System.out.println(o);
        }
        System.out.println("Map 中有数据");
    }

}
```

Phantom reference

3. Composition and inheritance

composition is better.

Unlike method calls, inheritance breaks encapsulation **.In other words, a subclass relies on the implementation details of its super class to ensure its correct functionality.**The implementation of the super class may change from release to release, and if so, **the subclass may be broken even if its code has not changed at all.**

As per effective Java (by Joshua Bloch), composition is favorable over inheritance. Composition has several advantages over inheritance. One of the intuitions for this is as follows: Consider a sub-class which inherits from a base class. So any change in the

base class will make the sub-class fragile as the sub-class depends on the base class. By using inheritance, we are making a binding on the sub-class to depend on the base class, which makes our code fragile. However by using composition, we can remove this limitation. Composition is done by establishing a 'has-a relationship' between classes instead of 'is-a' relationship as in inheritance. 'Delegate pattern' and 'Decorator pattern' both are examples of how composition can be achieved. You might want to read the chapter on 'composition vs inheritance' in the effective java book as it is quite informative.

Another example:
https://www.zhihu.com/question/21862257

定义的接口。

```
public interface IBrakeBehavior {
    public void brake();
}

public class AStyleBrake implements IBrakeBehavior {
    public void brake() {
        aStyleBrake(); // A 风格的刹车行为
    }
}

public class BStyleBrake implements IBrakeBehavior {
    public void brake() {
        bStyleBrake(); // B 风格的刹车行为
    }
}
```

```
}

//通过给下面的类赋予 AStyleBrake 或 BStyleBrake 可以完成不同 Model 的刹车行为的切换

// 同理， 汽车其他的行为（如启动 launch） 也可以用类似的方法实现
// 不同型号的汽车实现， 可以通过赋予不同风格的行为实例来“组装”出来的， 也就不需要为 Car 定义
public class Car{
    protected IBrakeBehavior brakeBehavior;

    public void brake() {
        brakeBehavior.brake();
    }

    public void setBrakeBehavior(final IBrakeBehavior brakeType) {
        this.brakeBehavior = brakeType;
    }
}
```

4. three characters of object-oriented language

3.1 encapsulation

- Encapsulation is the process of binding object state(**fields**) and behaviors(**methods**) together in a single entity called “Class”.
- Since it wraps both fields and methods in a class, it will be **secured** from the outside **access**.
- We can **restrict the access to the members of a class** using access modifiers such as private,protected and public keywords.
- When we create a class in Java, it means we are doing encapsulation.
- Encapsulation helps us to achieve the **re-usability** of code without compromising the **security**.

3.2 Inheritance

- One class inherits or acquires the **properties** of another class.
- Inheritance **provides the idea of re-usability** of code and **each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent/super class.**

3.3 Polymorphism

- It is the concept **where an object behaves differently in different situations.**
- Since the **object takes multiple forms, it is called Polymorphism.**
- In java, we can achieve it using method overloading and method overriding.
- There are 2 types of Polymorphism available in Java,

Method overloading(static Polymorphism) compilation

In this case, which method to call will be decided at the compile time itself based on number or type of the parameters. Static/Compile Time polymorphism is an example for

method overloading.
Method overriding(dynamic Polymorphism) runtime
In this case, which method to call will be decided at the runtime time based on what object is actually pointed by the reference variable.

5. private,protected and public keywords.

Modifier	Class	Package	Subclass	World
public	✔	✔	✔	✔
protected	✔	✔	✔	✘
*no modifier**	✔	✔	✘	✘
private	✔	✘	✘	✘

6. clean code principle

Answer to this question is similar to code review , in order to ensure high quality of your code.

7. code review

5.1 **meaningful name** must clearly
Naming is very important for **readability**. I personally prefer to have a long function/class name, b**ut it must clearly indicate what the function/class does.**
5.2 **Single Responsibility Principle**
A class does only one kind of related thing
A method, focuses on one thing
5.3 behavior must be unified
Whether the error handling is unified, use spring boot globally handle the exception.
5.4 duplicate code
Ide can check whether the code is duplicate , if it is, i should extract the duplicate part.
5.5 no hard code

I should write this piece of code into the config files.
5.6 robustness
Verify parameters
Whether to consider the integrity of the business, logic is robust
Are there any potential bugs?
5.7 error handling
Write proper try catch exception
5.8 performance
Take time complexity into consideration.
5.9 write comment
6.0 check whether your sql can get right results. And whether it is necessary to add index by using database management tools like navicat
6.1 can not use try catch to catch Runtimeexception or UncheckedException(can not be confirmed when compile, only can be confirmed during runtime) but can use try catch to catch checkedException or(compileException)

Solarlint plugin can be used for checking code

8. IOC and DI

DI(dependency injection) is the process of providing the dependencies ; and IoC(Inversion of control) is the end result of DI. ...**the responsibility of creating objects is shifted from our application code to the Spring container; this phenomenon is called IoC.** Dependency Injection can be done **by setter injection or constructor injection or field injection**(@Autowired).

9. hashmap hashset

8.1 hash collisions and its solutions linkedlist method:
In this simple implementation, we use **an array of linked lists and a hash code function**. To insert a key (which might be a string or essentially any other data type) and value, we do the following:

1. First, **compute the key's hash code**, which will usually be an int or long. Note that two different keys could have the same hash code, as there may be an infinite number of keys and a finite number of ints.
2. Then, **map the hash code to an index in the array**. This could be done with something like hash (key) % array_length. Two different hash codes could, of course, map to the same index.
3. At this index, there is a linked list of keys and values. **Store the key and value in this index. We must use a linked list because of collisions:** you could have two different keys with the same hash code, or two different hash codes that map to the same index.
To retrieve the value pair by its key, you repeat this process. Compute the hash code from the key, and then compute the index from the hash code. **Then, search through the linked list f**or the value with this key.

Method 2 :
Hashing with linear probing
Another approach to implementing hashing is to
store N key-value pairs in a hash table of size M > N, relying on empty entries in the table to help with collision resolution. Such methods are called open-addressing hashing methods.

The simplest open-addressing method is called linear probing: when there is a colli

sion (when we hash to a table index that is already occupied with a key different from the search key), then we just check the next entry in the table (by incrementing the index). Linear probing is characterized by identifying three possible outcomes:
n
Key equal to search key: search hit
n
Empty position (null key at indexed position): search miss
n
Key not equal to search key: try next entry
We hash the key to a table index, check whether the search key matches the key there, and continue (incrementing the index, wrapping back to the beginning of the table if we reach the end) until finding either the search key or an empty table entry. It is customary to refer to the operation of determining whether or not a given table entry

holds an item whose key is equal to the search key as a probe. We use the term inter

changeably with the term compare that we have been using, even though some probes are tests for null.

The essential idea behind hashing with open addressing is this: rather than using mem

ory space for references in linked lists, we use it for the empty entries in the hash table, which mark the ends of probe sequences. As you can see from LinearProbingHashST (Algorithm 3.6), applying this idea to implement the symbol-table API is quite straightforward. We implement the table with parallel arrays, one for the keys and one

for the values, and use the hash function as an index to access the data as just discussed.

10.linkedhashmap

Difference from hashmap
Ensure the sequence

11.interface is better than abstract class because of scalability.

Sub class can implements multiple interfaces, **so single interface can only have unique responsibility**. It conforms to **Single Responsibility Principl**e.and **Interface Segregation Principle**(we can use multiple Special-Interfaces instead of a single Interface including many responsibilities),

12.Unit Test vs Integration Test

Unit Testing t**ests only the functionality of the units themselves a**nd **may not catch integration errors, or other system-wide issues**
Integrating testing may detect errors when modules are integrated to build the overall system

See ci/cd for further information

https://www.youtube.com/watch?v=99ar2jEwoEg

13.JVM this definition need to be changed

Java Virtual machine (JVM) is the **virtual machine that runs the Java bytecodes.** You **get this bytecode by compiling the .java files into .class files. .class files contain the bytecodes understood by the JVM.**
JRE JVM + core class library
JDK JRE + javac.exe + jar.exe

14.generic eraser

https://www.cnblogs.com/drizzlewithwind/p/6101081.html

Java generics are valid for the compiler and are deleted at run time, which means that all generic parameter types are cleared after compilation

```
public class Foo {
    public void listMethod(List<String> stringList){
    }
    public void listMethod(List<Integer> intList) {
    }
}
```

Error occurs during compilation
Reason:

Avoid JVM **overhauls.If the JVM keeps generic types into the run time, the JVM will need to do a lot of refactoring at run time, so erasure can increase the run time efficiency.**

Another important term in java generics is "**type erasure**". It essentially means that all the extra information added using generics into **source code** will be **removed from bytecode** generated from it. Inside bytecode, it will be old java syntax which you will get if you don't use generics at all.

15.JVM-class load process

Class loading is divided into the following five steps:

- Loading: **find the corresponding class f**ile according to the search path and import it;
- Verification: Check the correctness of the loaded class file;
- Preparedness: Allocates memory space to static variables in the class;
- Resolution: The process by which a virtual machine replaces a symbolic reference in a constant pool with a direct reference.A symbolic reference is understood as an identifier, whereas a direct reference points directly to an address in memory.

  举个例子来说，现在调用方法 hello()，这个方法的地址是 1234567，那么 hello 就是符号引用，1234567 就是直接引用。
- Initialization: Performs initialization on static variables and blocks of static code.

16.Talk about the JVM runtime data area

- **Program Counter Register**

- **Java Virtual Machine Stacks**

  It is used to store local variable tables, operand stacks, dynamic links, method exits and other information.
- **Native Method Stack**
- **Java Heap**

  The largest chunk of memory in the Java virtual machine,it is shared by all threads, where almost all object instances allocate memory.
- **Method Area**

  It is used to store class information, constants, static variables,real-time compiled code and other data that has been loaded by the virtual machine.

17.static keywords

The most important thing about static variables is that they belong to class level. What it means is that there can be only one copy of variable in runtime.

When you define a static variable in class definition, each instance/object of class will have access to that single copy.

18.Can static method be inherited and override?

Inherited? YES!
Override ? No!
No!
https://blog.csdn.net/caonima0001112/article/details/50482222
https://blog.csdn.net/zhouhong1026/article/details/19114589

19.transient keyword

The modifier transient in java can be applied to **field members** of a class to **turn off serialization** on these **field members**.

20.java clone

- Shallow clone is “default implementation” in Java. **In overridden clone method, if you are not cloning all the object types (not primitives),** then you are making a shallow copy.
- Deep clone is the desired behavior in most of the cases. In the deep copy, we create a clone which is **independent of original object** and **making changes in the cloned object should not affect original object.**

- **Typical IO Stream**

```
import java.util.Scanner;

public class ReadAndWriteTest {
    public static void main(String[] args) throws IOException {
        File file = new File( pathname: "D:\\工作常用工具\\java_web_test\\stephen\\src\\main\\java\\basic\\knowledge\\stephen\\io\\routes.txt");
        FileInputStream fis = new FileInputStream(file);
        Scanner scanner = new Scanner(new BufferedInputStream(fis),  charsetName: "UTF-8");


        File outFile = new File( pathname: "D:\\工作常用工具\\java_web_test\\stephen\\src\\main\\java\\basic\\knowledge\\stephen\\io\\Out.txt");
        OutputStream outputStream = new BufferedOutputStream(new FileOutputStream(outFile));
        while(scanner.hasNextLine()){
            String str = scanner.nextLine();
            outputStream.write(str.getBytes());
            outputStream.write( b: '\n');
        }
        outputStream.close();
        fis.close();


    }
```

21.checkedexception and uncheckedexception

- CheckedException (means checked during compile-time)

Java verifies checked exceptions at compile-time.
Some common checked exceptions in Java are **IOException, SQLException and ParseException,classnotfoundexception.**

The Exception class is the superclass of checked exceptions, so we can create a custom checked exception by extending Exception.

- UncheckedExcpetion (RuntimeException)

If a program throws an unchecked exception, it reflects some error inside the program logic.
For example, if we divide a number by 0, Java will throw ArithmeticException:

Java does not verify unchecked exceptions at compile-time. Furthermore, we don't have to declare unchecked exceptions in a method with the throws keyword. And although the above code does not have any errors during compile-time, it will throw **ArithmeticException** at runtime.

Some common unchecked exceptions in Java are **NullPointerException, ArrayIndexOutOfBoundsException and IllegalArgumentException.**

The RuntimeException class is the superclass of all unchecked exceptions, so we can create a custom unchecked exception by extending RuntimeException:

- **What is callback Function**

下文讲得非常好
https://blog.csdn.net/liu16659/article/details/82959540?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-1-82959540-blog-105261755.pc_relevant_3mothn_strategy_recovery&spm=1001.2101.3001.4242.2&utm_relevant_index=4

https://github.com/LittleLawson/TestCallBack

本机电脑 也有代码

上述代码中: server 就相当于是支付平台 system design 的第三方服务

上述代码的调用过程如下：
weChatUser.pay() => server.payToTelecomByWeChat() => server.notifyToWeChat() => weChatUser.notifyToUser()
可以看到最开始由WeChat.pay()出发，最后回到weChatUser.notifyToUser()方法【注意他们都是WeChat这个类中的方法】，故称回调。

5.总结

https://www.51cto.com/article/645183.html

22.Why is char[] preferred over String for passwords?

https://stackoverflow.com/questions/8881291/why-is-char-preferred-over-string-for-passwords

Strings are immutable. That means once you've created the String, if another process can dump memory, there's no way (aside from reflection) you can get rid of the data before garbage collection kicks in.
With an array, you can explicitly wipe the data after you're done with it. You can overwrite the array with anything you like, and the password won't be present anywhere in the system, even before garbage collection.
So yes, this is a security concern - but even using char[] only reduces the window of opportunity for an attacker, and it's only for this specific type of attack.
As noted in the comments, it's possible that arrays being moved by the garbage collector will leave stray copies of the data in memory. I believe this is implementation-specific - the garbage collector may clear all memory as it goes, to avoid this sort of thing. Even if it does, there's still the time during which the char[] contains the actual characters as an attack window.

https://blog.csdn.net/youanyyou/article/details/107754450

Best answer :
https://www.zhihu.com/question/36734157

23.String questions

https://blog.csdn.net/u011541946/article/details/79865160

B. design pattern

1. What are SOLID principles

https://zhuanlan.zhihu.com/p/24614363

1.1 Single responsibility principle

Using interface instead of abstract class

1.2 Open close Principle:

Open close Principle: key
https://blog.csdn.net/Peng_Hong_fu/article/details/52571664

Does Not conform to **Open close Principle: Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification**

Bad practise: pay methods is open to modification

```
public class PayHandler {

        public Result<T> pay(Param param) {
                if(param.getType() == "ALIPAY") {
                        // 支付宝付款调用
                        ...
                } else if(param.getType() == "WeChatPay") {
                    // 微信支付付款调用
                    ...
                }
        }
}
```

Better ,Conform to: **Open close Principle**

Wont change pay methods , it means like alipay wechat pay won't be changed at all

```
public class PayHandler {

        private Map<String, PayProcessor> processors;

        public Result<T> pay(Param param) {
                PayProcessor payProcessor = processors.get(param.getType());
                // 异常处理略
                return payProcessor.handle(param);
        }
}

interface PayProcessor {
        Result<T> handle(Param param);
}

public class AlipayProcessor implements PayProcessor {
        ...
}

public class WeChatPayProcessor implements PayProcessor {
        ...
}
```

1.3 Dependency Inversion Principle

The Dependency Inversion Principle (DIP) states that high level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.

Put it in another way, we should lower the decoupling of two Objects

https://blog.csdn.net/ljw124213/article/details/82011162?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_title~default-0-82011162-blog-73826719.pc_relevant_multi_platform_whitelistv1&spm=1001.2101.3001.4242.1&utm_relevant_index=3

定义：高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

场景：类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作；假如修改类A，会给程序带来不必要的风险。

解决方案：遵循依赖倒置原则，将类A修改为依赖接口I，类B和类C各自实现接口I，类A通过接口I间接与类B或者类C发生联系，则会大大降低修改类A的几率。

上面用到很多泛泛的词语，为了帮助理解，对其进行大白话化：抽象就是指接口或抽象类，两者都是不能直接被实例化的；细节就是实现类，实现接口或继承抽象类而产生的类就是细节，其特点就是，可以直接被实例化。高层模块就是调用端，低层模块就是具体实现类。依赖倒置原则在 Java 语言中的表现就是：**模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的。**这又是一个将理论抽象化的实例，其实一句话就可以概括：面向接口编程，或者说是面向抽象编程，这里的抽象指的是接口或者抽象类。

理论抽象化的实例，其实一句话就可以概括：面向接口编程，或者说是面向抽象编程，这里的抽象指的是接口或者抽象类。

Program to an interface, not an implementation

Bad practice:

篮球类：

```java
/**
 * Created by LJW on 2018/8/24.
 * 篮球类
 */
public class Basketball {

    public String playBall(){
        return "一群孩子正在开心的打篮球.....";
    }
}
```

孩子类：

```java
/**
 * Created by LJW on 2018/8/24.
 * 孩子类
 */
public class Children{

    public void play(Basketball basketball){ //通过参数依赖了篮球类
        System.out.println(basketball.playBall());
    }
}
```

Good practice

```java
/**
 * Created by LJW on 2018/8/24.
 * 孩子类
 */
public class Children{

    public void play(IPlayBall playBall){
        System.out.println(playBall.playBall());
    }
}
```

最后修改孩子类，不再让其依赖任何具体的球类，而是依赖球类接口，这样就遵循了依赖倒置原则：

```java
/**
 * Created by LJW on 2018/8/24.
 * 孩子类
 */
public class Children{

    public void play(IPlayBall playBall){
        System.out.println(playBall.playBall());
    }
}
```

https://blog.csdn.net/yabay2208/article/details/73826719

> **宝马有了，却不能让张三开起来，这也太不合理了，我们的设计出了问题：司机类和奔驰类之间是紧耦合的关系，其导致的结果就是系统的可维护性大降低，可读性降低。//若需要改为开宝马，必须得更改司机类的public void drive(Benz benz)（低层模块，太不应该）现根据DIP原则，重新编写如下**

In this example , driver is coupled with BMW, which means Zhangsan cannot drive benz if he has a new car with Benz brand. The improvement should let Driver be coupled with Icar , instead a concrete one.

司机：

```java
//司机接口
interface IDriver{
    public void driver(ICar car);
}
//司机源代码
class Driver implements IDriver{
    private String name;
    public Driver(String name) {this.name = name;}
    //是司机就应该会驾驶汽车
    public void driver(ICar car){
        System.out.println(this.name + "老司机开车了，" + car.run());
    }

}
```

http://blog.csdn.net/yabay2208

场景：

```java
}
```

场景：

```java
//场景类源代码
public class Client{
    @Test
    public void test(){
        Driver zhangsan = new Driver("张三");
        Benz benz = new Benz();
        BMW bmw = new BMW();
        //张三开奔驰车
        zhangsan.driver(benz);
        //张三开宝马
        zhangsan.driver(bmw);
    }
}
```

http://blog.csdn.net/yabay2208

运行结果：

```
D:\NEWBEGIN\jdk1.8\bin\java ...
```

1.4 Interface Segregation Principle:

https://blog.csdn.net/yabay2208/article/details/73739514#comments_19020461

The difference between single responsibility and interface isolation:

The principle of single responsibility focuses on responsibility; The interface isolation principle focuses on the isolation of interface dependencies.

The single responsibility principle is mainly to constrain the class, followed by the interface and method, it is aimed at the implementation and details in the program; The interface isolation principle mainly restricts the interface, mainly for abstraction, for the construction of the overall framework of the program.

1.5 Liskov Substitution Principle:

A subclass can inherit the functions of the super class, but it cannot change the original function of the super class

https://blog.csdn.net/yabay2208/article/details/73804831

```java
    public int sub(int a, int b){
        return a - b;
    }
}
class Add extends Subtraction{
    public int sub(int a, int b){
        return a + b;
    }

    public int add(int a, int b){
        return sub(a,b) + 100;
    }
}
public class Client2{
    @Test
    public void test(){
        Add add = new Add();
        System.out.println("100-50=" + add.sub(100, 50));
        System.out.println("100-80=" + add.sub(100, 80));
        System.out.println("100+20+100=" + add.add(100, 20));
    }
}
```

http://blog.csdn.net/yabay2208

我们发现原来运行正常的相减功能发生了错误。原因就是类Add在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类Add的重写后的方法，造成原本运行正常的功能出现了错误。
在本例中，使用Subtraction类完成的功能，换成子类Add之后，发生了异常。在实际编程中，我们常会通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的几率非常大。如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通谷的基类，原来的继承关系去掉，采用依赖、聚合、组合等关系替代。

How to avoid this: use composite instead of inheritance

https://www.kbasm.com/blog/cn/oop-prefer-composition-over-inheritance

继承:

```
class ClassA {
    functions of ModuleA;
    functions of ModuleB;
};

class ClassB extends ClassA {
    functions of ModuleC;
};
```

组合:

组合:

```
class ModuleA {};
class ModuleB {};
class ModuleC {};

class ClassA {
    ModuleA ma;
    ModuleB mb;
};

class ClassB {
    ModuleA ma;
    ModuleB mb;
    ModuleC mc;
};

//或者:

class ClassB {
    ClassA ca;
    ModuleC mc;
};
```

1.6 Low of demeter principle:

The least known principle, a class can only communicate with immediate classes, can not bypass.
Must use transitivity. A--> B--> C

https://blog.csdn.net/zhengzhb/article/details/7296930

现在这个设计的主要问题出在CompanyManager中，根据迪米特法则，只与直接的朋友发生通信，而SubEmployee类并不是CompanyManager类的直接朋友（以局部变量出现的耦合不属于直接朋友），从逻辑上讲总公司只与他的分公司耦合就行了，与分公司的员工并没有任何联系，这样设计显

2. Describe some typical design pattern?

https://blog.csdn.net/tanzuai/article/details/48677037

1 Decorator:

Suppose we have a Christmas tree object and we want to decorate it. **The decoration does not change the object itself**; it's just that **in addition to** the Christmas tree, we're adding some decoration items like garland, bubble lights, etc.: then use IO Stream in java Fileinputstream BufferedInputStream.

2 Singleton:

https://blog.csdn.net/qq_40885085/article/details/103363774

- Eager Initialization

```
public class HungrySingleton {
    // 类加载，初始化
    private static final HungrySingleton instance = new HungrySingleton();

    private HungrySingleton() {}

    public static HungrySingleton getInstance() {
        return instance;
    }
}
```

- Lazy Initialization

See mooc website

```
public class LazySingleton {
    // 被 volatile 修饰的变量可以确保多个线程能正常处理
    private volatile static LazySingleton instance = null;

    private LazySingleton() {}

    public static LazySingleton getInstance() {
```

```
        // 第一层判断，如果实例已经创建，跳过
        if(instance == null) {
            synchronized(LazySingleton.class) {
                // 第二层判断，如果实例创建，跳过
                if(instance == null) {
                    instance = new LazySingleton();
                }
            }
        }
        return instance;
    }
}
```

The singleton pattern is a design pattern that **restricts the instantiation of a class to one object.**

```
 * 支持多线程调用,线程安全
 */
public class Singleton {

    private static volatile Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

3 Factory:

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface(Shape).

steps

- Create an interface.
  - Shape.java
- Create concrete classes implementing the same interface.

    - Rectangle.java
    - Square.java
    - Circle.java
- Create a Factory to **generate object of concrete class based on given information.**
    - ShapeFactory.java
- Use the Factory to **get object of concrete class** by passing an information such as type.
- Verify the output.

4 Builder:

Builder pattern builds a complex object using simple objects and using a step by step approach, and these steps were done in sequence. Users do not know the building details.

5 Strategy:

https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

> 3、JAVA AWT 中的 LayoutManager。
>
> **优点：** 1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性良好。
>
> **缺点：** 1、策略类会增多。 2、所有策略类都需要对外暴露。

Define a family of algorithms, encapsulate each one, and make them interchangeable.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object(context object has strategy interface as a field). The strategy object changes the executing algorithm of the context object.

Step 1

Create an interface.

Strategy.java

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

Step 2

Create concrete classes implementing the same interface.

OperationAdd.java

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

OperationSubstract.java

```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

OperationMultiply.java

```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

Step 3

Create *Context* Class.

Context.java

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

Step 4

Use the *Context* to see change in behaviour when it changes its *Strategy*.

StrategyPatternDemo.java

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

6 delegate design

In software engineering, the delegation pattern is an object-oriented design pattern that allows object composition to achieve the same code reuse as inheritance. In delegation, an object handles a request by delegating to a second object (the delegate). The delegate is a helper object, but with the original context.

7 Observer Pattern

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

8 Proxy Pattern

Proxy is a structural design patte**rn that lets you provide a substitute or placeholder f**or another object. A proxy controls access to the original object, allowing you to p**erform something either before or after the request** gets through to the original object.

9 Adapter Pattern

https://www.geeksforgeeks.org/adapter-pattern/?ref=lbp

This pattern is easy to understand as the real world is full of adapters. For example consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other. we use an adapter that converts one to other. **This example is pretty analogous to Object Oriented Adapters. In design, adapters are used when we have a class (Client) expecting some type of object and we have an object (Adaptee) offering the same features but exposing a different interface.**

To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter **translates that request** on the adaptee using the adaptee interface.
3. Client receive the results of the call and is unaware of adapter's presence.

10 composite design.(it is not composite which is compartment of inheritance)

Composite pattern is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object. T**he intent of a composite is to** “compose” objects into **tree structures** to represent **part-whole** hierarchies. It allows you to have a tree structure and ask each node in the tree structure to perform a task.

C. Java version 8 new features

https://geek-docs.com/java/java8/java-8%e6%b5%81.html
https://www.cnblogs.com/binghe001/p/12995007.html

- Creating stream

stream()
parallelStream()
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

- forEach

Random random = new Random();
random.ints().limit(10).forEach(System.out::println);

- map

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5); //get list of unique squares
List<Integer> squaresList = numbers.stream().map(i -> i*i).distinct().collect(Collectors.toList());

- filter

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
//get count of empty string
 int count = strings.stream().filter(string -> string.isEmpty()).count();//out put :2

- limit

Random random = new Random();
random.ints().limit(10).forEach(System.out::println);

- sorted

Random random = new Random();

```
random.ints().limit(10).sorted().forEach(System.out::println);
```

- parallel

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
//get count of empty string
int count = strings.parallelStream().filter(string -> string.isEmpty()).count();
```

- Collectors

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList()); System.out.println("Filtered List: " + filtered);
String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(", "));
System.out.println("Merged String: " + mergedString);
```

- Statistics

```
List numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
IntSummaryStatistics stats = integers.stream().mapToInt((x) -> x).summaryStatistics();
System.out.println("Highest number in List : " + stats.getMax());
System.out.println("Lowest number in List : " + stats.getMin());
System.out.println("Sum of all numbers : " + stats.getSum());
System.out.println("Average of all numbers : " + stats.getAverage());
```

- HashMap

getOrDefault

putIfAbsent

computeIfAbsent

differences

https://segmentfault.com/a/1190000039756899

https://github.com/bin392328206/six-finger

D. JVM

1. Difference between Thread dumps and Heap dumps

- Thread dumps

The Thread Dump is a **snapshot** (all thread stacks) of an application's **currently active threads.** Simply put, the Thread Dump log contains the details of all active threads in the application **at a given time.**

- Heap dumps

Just as the Thread Dump file **records the running of threads in the JVM at that time, the Heap Dump records the running of Heap memory in the JVM**

How to export heap dumps :
https://www.cnblogs.com/feibazhf/p/14457834.html

How to analyze a dump file:

https://blog.csdn.net/xiaowanzi_zj/article/details/122335429?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1.pc_relevant_aa&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1.pc_relevant_aa&utm_relevant_index=2

2. Out of memory and memory leak

https://blog.csdn.net/weixin_43899792/article/details/124304136

3. How does jvm judge a object can be collected

See GC roots here
https://www.callibrity.com/blog/a-basic-overview-of-jvm-garbage-collection

4. If a Java process has a full gc every 10 minutes, what can I do

https://blog.csdn.net/qq_34679704/article/details/123511327
Good example

5. GC algorithms

ß
https://www.51cto.com/article/704422.html

6. TAKE AN example of how you did fix a jvm issue ?

https://zhuanlan.zhihu.com/p/515784028

I can take the actual example in Suncorp: I update premium total amount .

E. Data Structure and sql

1. B and B+ Differences between B and B+ trees

Chinese:
https://segmentfault.com/a/1190000038444596
https://www.cnblogs.com/chanshuyi/p/tree-data-structure-03-b-tree.html

English: B and B+Tree
https://www.baeldung.com/cs/b-trees-vs-btrees

B tree:

Definition:

B-trees are a type of self-balancing tree structure designed for storing huge amounts of data for fast query and retrieval. They can be often confused with their close relation – the Binary Search Tree(2-way). Although they're both a type of m-way search tree, the Binary Search Tree is considered to be a special type of B-tree.

B-trees can have multiple key/value pairs in a node, sorted ascendingly by key order(withing the node). We call this key/value pair a payload (difference from B+ tree). Occasionally we may hear the value part of the payload being referred to as "satellite data" or just "data".

In the context of databases, a key may be the primary key or indexed column of a row, and the value could be the actual row record itself or a reference to it.

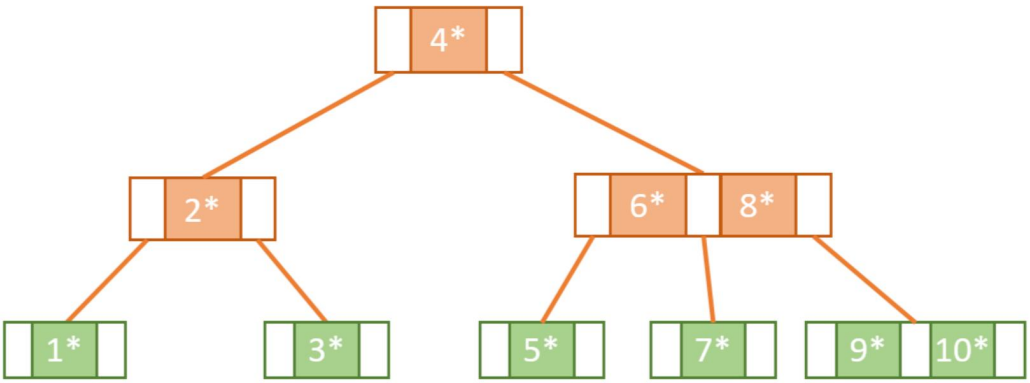
Application: MongoDB

Properties:
For a tree to be classified as a B-tree, it must fulfill the following conditions:

- the nodes in a B-tree of order can have a maximum of children(in this example , m = 3)
- each internal node (**non-leaf and non-root**) can have at least (/2) children (rounded up)
- the root should have at least two children – unless it's a leaf
- a non-leaf node with children should have keys

  Node with 6,8, two keys has 3 children nodes .
- all leaves(leaf nodes) must appear on the same level

The image below shows us an example of a B-tree. The maximum number of children in this tree is 3, so we can conclude that this is a tree of order 3. All the leaves are on the same level, and the root and internal nodes have a minimum of two children fulfilling all the conditions of a B-tree:

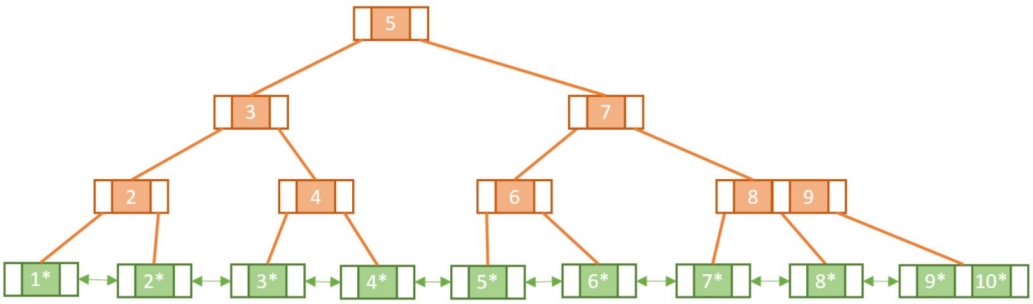


B+ tree

The most well-known version of the B-tree is the B+tree. What distinguishes the B+tree from the B-tree are two main aspects:

- all leaf nodes are linked together in a doubly-linked list
- satellite data is stored on the leaf nodes only. Internal nodes only hold keys and act as routers to the correct leaf node

Let's see how our B-tree would look like if we were to represent its content as a B+tree:



why use B +

Here, are reasons for using B+ Tree:

- Key are primarily utilized to aid the search by directing to the proper Leaf.
- B+ Tree uses a “fill factor” to manage the increase and decrease in a tree.
- In B+ trees, numerous keys can easily be placed on the page of memory because they do not have the data associated with the interior nodes. Therefore, it will quickly access tree data that is on the leaf node.
- A comprehensive full scan of all the elements is a tree that needs just one linear pass because all the leaf nodes of a B+ tree are linked with each other.

Main differences:

与 B 树相比，B+ 树有着如下的好处：

目录

1. B+ 树的磁盘读写代价更低

B+ 树的内部结点并没有指向关键字具体信息的指针，所以其内部结点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，所以一次性读入内存中的需要查找的关键字也就越多。相对来说 IO 读写次数也就降低了，查找速度就更快了。

2. B+ 树查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以 B+ 树中任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。而对于 B 树来说，因为其每个节点都存具体的数据，因此其查询速度可能更快，但是却并不稳定。

3. B+ 树便于范围查询（最重要的原因，范围查找是数据库的常态）

B 树在提高了 IO 性能的同时，并没有解决元素遍历效率低下的问题。为了解决这个问题，B+ 树应用而生。B+ 树只需要去遍历叶子节点就可以实现整棵树的遍历。在数据库中基于范围的查询是非常频繁的，因此 MySQL 的 Innodb 引擎就使用了 B+ 树作为其索引的数据结构。

leaf nodes of a B+ tree are linked with each other.

B+ Tree vs. B Tree

Here, are the main differences between B+ Tree vs. B Tree

B+ Tree	B Tree
Search keys can be repeated.	Search keys cannot be redundant.
Data is only saved on the leaf nodes.	Both leaf nodes and internal nodes can store data
Data stored on the leaf node makes the search more accurate and faster.	Searching is slow due to data stored on Leaf and internal nodes.
Deletion is not difficult as an element is only removed from a leaf node.	Deletion of elements is a complicated and time-consuming process.
Linked leaf nodes make the search efficient and quick.	You cannot link leaf nodes.

B+树应用: MYSQL INNODB 里的聚簇索引

2. Red black tree

https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html

Time complexity:o(logn)

Data Structures and Algorithms

8.2 Red-Black Trees

A *red-black tree* is a binary search tree with one extra attribute for each node: the *colour*, which is either red or black. We also need to keep track of the parent of each node, so that a red-black tree's node structure would be:

```
struct t_red_black_node {
    enum { red, black } colour;
    void *item;
    struct t_red_black_node *left,
                            *right,
                            *parent;
}
```

For the purpose of this discussion, the NULL nodes which terminate the tree are considered to be the leaves and are coloured black.

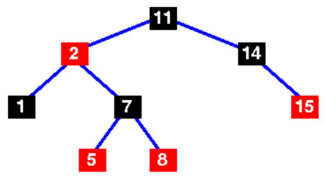
Definition of a red-black tree

A red-black tree is a binary search tree which has the following *red-black properties*:

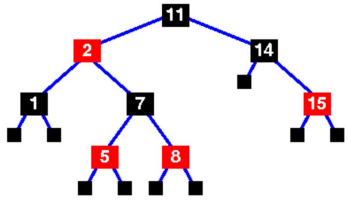
1. Every node is either red or black.
2. Every leaf (NULL) is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

3. implies that on any path from the root to a leaf, red nodes must not be adjacent.
However, any number of black nodes may appear in a sequence.

Structure



A basic red-black tree



Basic red-black tree with the **sentinel** nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node.

They are the NULL black nodes of property 2.

3. AVL tree

https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor Adelson, Velski & Landis, AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

Time complexity:o(logn)
An AVL tree is a binary search tree which has the following properties:

- The sub-trees of every node differ in height by at most one.
- Every sub-tree is also an AVL tree.

4. binary heap

Time complexity:o(log2n)
A **binary heap** is a heap data structure that takes the form of a **binary tree**. **Binary heaps are a common way of implementing priority queues**. ... Heap property: t**he key stored in each node is either greater than or equal to (≥) or less than or equal to (≤) the keys in the node's children,** according to some total order.

5. graph

A graph is a set of **vertices** ....and a collection of **edges** that each edge connects a pair of vertices.

6. minimum spanning tree

A minimum spanning tree is a minimally connected sub-graph of a graph that **contains all the vertices of the original graph** and **the sum of the weights of all the edges** is as small as possible.

7. Quick sort(fasted sorting algorithm)

Quick sort is a **divide and conquer method** for sorting. It works by partition an array into two subarrays. If we **find a correct partition point(critical)** , and elements in left subarray are less than or equal to this point and elements in right subarray are more than or equal to this point
Then sorting these two subarrays independently. **When the two subarrays are sorted, the whole array is sorted automatically.**

https://blog.csdn.net/qq_36770641/article/details/82669788
https://cloud.tencent.com/developer/article/1350860

快速排序快的主要原因是大大减少了比较和交换的次数，因为按基准数切分的两半数组，在一个数组里面的数据是绝对不会和第二个数组里面的数字产生比较的机会的，所以大幅度降低了做无用功的机会。 快速排序需要比较次数： 10000 * log2 10000 =14*10000=14万次。 2018年9月29日

腾讯云

8. merge sort

In merge sort , we break the array into **two subarrays(by equally dividing the array**),each of them to be sorted and then combine the sorted subarrays to make the whole array sorted .

9. Time complexity of quick sort and merge sort , and why JDK chose quick sort

https://www.zhihu.com/question/24727766

Primitive Object , use quick sort
Reference Object, use merge sort

Reason:
https://bbs.huaweicloud.com/blogs/127449

那么排序算法的「稳定性」在什么情况下才会变得有意义呢？

> 举个例子，一个班的学生已经按照学号大小排好序了，我现在要求按照年龄从小到大再排个序，如果年龄相同的，必须按照学号从小到大的顺序排列。那么问题来了，你选择的年龄排序方法如果是不稳定的，是不是排序完了后年龄相同的一组学生学号就乱了，你就得把这组年龄相同的学生再按照学号拍一遍。如果是稳定的排序算法，我就只需要按照年龄排一遍就好了。
>
> 从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。要排序的内容是一个复杂对象的多个数字属性，且其原本的初始顺序存在意义，那么我们需要在二次排序的基础上保持原有排序的意义，才需要使用到稳定性的算法。

> 有很多算法你现在看着没啥，但是当放在大数据云计算的条件下它的稳定性非常重要。举个例子来说，对淘宝网的商品进行排序，按照销量，价格等条件进行排序，它的数据服务器中的数据非常多，因此，当时用一个稳定性效果不好的排序算法，如堆排序、shell排序，当遇到最坏情形，会使得排序的效果非常差，严重影响服务器的性能，影响到用户的体验。

10.BloomFilter

https://www.interviewcake.com/concept/java/bloom-filter

A **bloom filter** is a space-efficient data structure that lets you quickly check whether or not an item is in a set.

When you look up an item in a bloom filter, the possible answers are:

- **It's definitely not in the set. This is a true negative.**

- **It might be in the set. This could be a <u>false positive</u>, or it could be a <u>true positive.</u>**

Strengths:

- Space-efficient. Bloom filters take up O(1) space, regardless of the number of items inserted. (But, their accuracy goes down as more elements are added.)
- Fast. Insert and lookup operations are both O(1) time.

Weaknesses

- Probabilistic. Bloom filters can only definitively identify true negatives. They cannot identify true positives. **If a bloom filter says an item is present, that item might actually be present (a true positive) or it might not (a false positive).**
- Limited Interface. **Bloom filters only support the insert and lookup operations. You can't iterate through the items in the set or delete items.**

Implementation

Under the hood, a bloom filter is just a bitmap. Initially, all the bits are set to 0.

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

> Fixed size is crucial here. Remember, bloom filters require $O(1)$ space.

Two ways to avoid high false true:

1:
The bigger the m(BitMap Size) is , the more accurate the result is!

Bitmap Size

As the number of items inserted into a bloom filter increases, so does the likelihood of a false positive. That's because over time, we'll end up setting most of the bits to 1, instead of 0.

The larger the bitmap, the less likely false positives are. When using a bloom filter, it's helpful to know ahead of time roughly how many elements will be in your set. That way you can make the bitmap large enough to avoid a high false positive rate.

2: in lookup if we check carrot and every hash functions has the same result , we have higher probabilistic that carrot is in the set, it means it lowers the false positive.

Multiple Hash Functions

Another way to avoid a high false positive rate is to use multiple hash functions.

In the examples above we only used one hash function, checking one bit in the bitmap for each lookup.

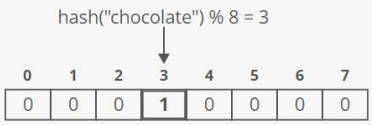
When using multiple hash functions:

- To add a new item: generate a bitmap index with *each* hash function, and set *all* of those bits to 1.
- To check if an item is present: just check if *any* of the bits checked are still 0. If so, the item is definitely not present.

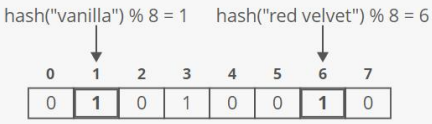
Inserts

Let's add some cake flavors to our bloom filter, starting with "chocolate." To do this:

1. Hash the string ("chocolate")
2. Mod the result by the length of our bitmap
3. Set that bit to 1



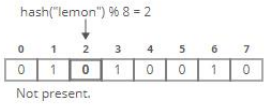
Same thing for adding "vanilla" and "red velvet":



Lookups

To check if a word is in our bloom filter we:

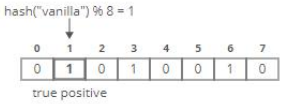
1. Hash it
2. Mod the result
3. Check if the corresponding bit is 0 or 1

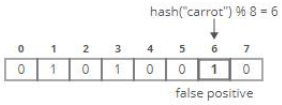


If the bit is 0, then that word **definitely isn't** in our set. If it were, we would have set that bit to 1.

But if the bit is 1, then that word **might be** in our set. But it could be a **false positive**.

In our example, checking for "vanilla" gives us a true positive—we *did* put that in our bloom filter earlier. But checking for "carrot" gives us a false positive—we set that bit to 1 for "red velvet".





This false positive isn't a bug; it's a tradeoff we've made to keep checks fast. Bloom filters are only appropriate when it's feasible to handle false positives.

11.sweep line explanation

https://zhuanlan.zhihu.com/p/103616664

4. 第二条边为入边，区间为[15,25.5]，则[15,25.5]+1(此时区间[10,15]=1,[15,20]=2,[20,25.5]=1)

12.differences among sorts alg

https://blog.csdn.net/ccblogger/article/details/82384480

13.What is Trie

https://zhuanlan.zhihu.com/p/28891541

14.Rolling Hash

https://www.geeksforgeeks.org/string-hashing-using-polynomial-rolling-hash-function/

15.KMP

Leetcode 2430 by KMP

```
class Solution {
    public int deleteString(String s) {
        int n = s.length();
        int[] dp = new int[n];
        Arrays.fill(dp, 1);
        for(int idx = n - 2; idx >= 0; idx--) {
            int[] pat = kmp(s.substring(idx, n));
            for(int i = 1; i < pat.length; i++) {
                if(i + 1 == pat[i] * 2) {
                    dp[idx] = Math.max(dp[idx], dp[idx + pat[i]] + 1);
                }
            }
        }
        return dp[0];
    }
```

```java
private int[] kmp(String s) {
    int[] pattern = new int[s.length()];
    for(int i = 1, cnt = 0; i < s.length();) {
        if(s.charAt(i) == s.charAt(cnt)) {
            pattern[i] = ++cnt;
            i++;
        } else if(cnt == 0) {
            i++;
        } else {
            cnt = pattern[cnt - 1];
        }
    }
    // System.out.println(s + "," + Arrays.toString(pattern));
    return pattern;
}
```

Leetcode 2430 by Hash
https://leetcode.com/problems/maximum-deletions-on-a-string/discuss/2648931/Rolling-Hash-or-DP-or-C%2B%2B-or-Easy-to-Understand

KMP 解释:
https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/KMP.java.html

Or see 左程云那本书

F. Distributed System(microservices)

1. Distributed system

Advantages:
1 Enhance concurrency , 2 Enhance availability ;3 promote fault tolerance; 4 low latency
Disadvantages:
Dependent on network, if network can't work. System can't work.
Maintenance cost is higher

https://www.ruanyifeng.com/blog/2019/11/fault-tolerance.html

Because of CAP theory: Consistency, availability, and partition fault tolerance cannot be met at the same time.
Consistency : all nodes see the same data at the same time
Availability :reads and writes always succeed
partition fault tolerance:the system continues to operate despite arbitrary message loss or failure of part of the system

Best explanation
https://www.zhihu.com/question/54105974

2. What is Consistency in CAP theory

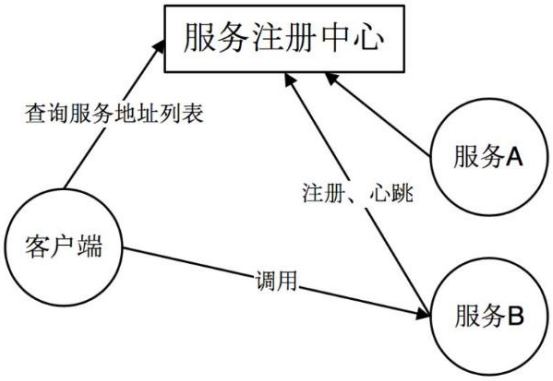
In cluster, if one node has run the write operation, this updated has to be made in all nodes , this is data consistency in cluster.

3. 服务发现 api gateway 链路追踪，监控，分布式事务

3.1 Service discovery:

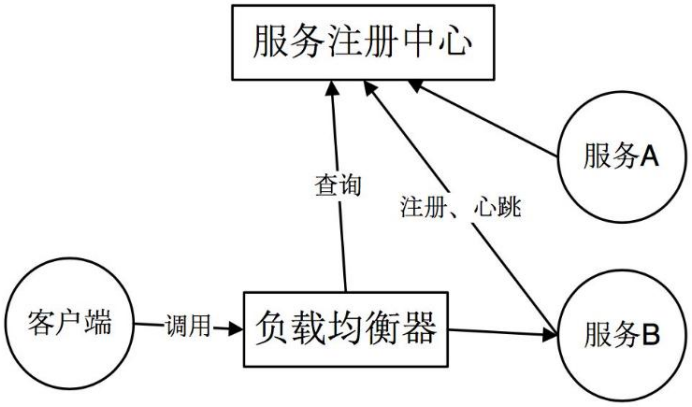
https://zhuanlan.zhihu.com/p/32027014
客户端

客户端服务发现



在使用客户端发现方式时，客户端通过查询服务注册中心，获取可用的服务的实际网络地址（IP和端口）。然后通过负载均衡算法来选择一个可用的服务实例，并将请求发送至该服务。优点：架构简单，扩展灵活，方便实现负载均衡功能，缺点：强耦合，有一定开发成本。

服务端服务发现



客户端向load balancer 发送请求。load balancer 查询服务注册中心找到可用的服务，然后转发请求到该服务上。和客户端发现一样，服务都要到注册中心进行服务注册和注销。优点：服务的发现逻辑对客户端是透明的。缺点：需要额外部署和维护高可用的负载均衡器。

3.2 api gateway:

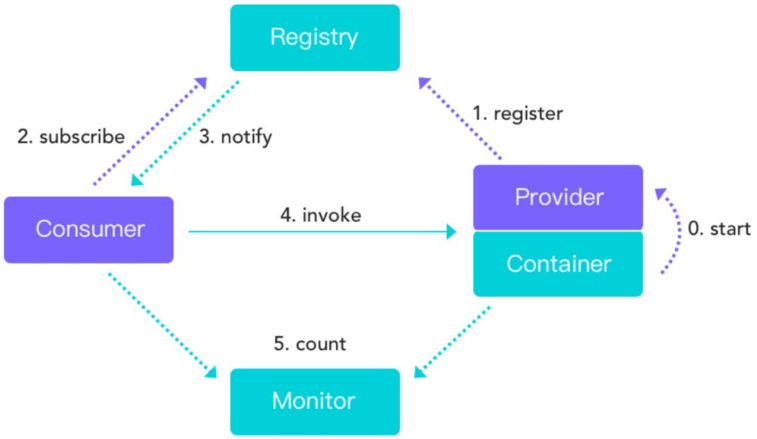
3.3 Monitor:

https://www.51oneday.com/2019/11/java-xiangmuxingnengjiankonghediaoyougongju-javamelody/

3.4 whats is Distributed transaction: 有什么框架?,how to implement?

https://xiaomi-info.github.io/2020/01/02/distributed-transaction/
seata
用于保证 consistency

4. Rpc framework dubbo



0: The server container is responsible for starting, loading, and running the service provider
1: The service provider can expose the service to the registry after startup by storing id , ip ,port onto the registry.
2: Service consumers can subscribe to the registry for the services they want after startup
3: The registry returns a list of service calls to the service consumer
4: The service consumer invokes the service provider's service based on the soft load balancing algorithm. This service provider may be a list of service providers, and that service provider is called according to the load balancing
5: The service provider and the service consumer will **regularly push the number and time of service invocation** stored in memory to the monitoring center

Node role description:
Provider: The service Provider that exposes the service.
Consumer: A service Consumer that invokes a remote service.
Registry: A Registry for service registration and discovery. **zookeeper**
Monitor: A monitoring center that counts the invocation times and invocation times of a service.
Container: The service runs the Container

5. Rpc and http

https://cloud.tencent.com/developer/article/1753834

通信协议不同：HTTP 使用文本协议，RPC 使用二进制协议。 调用方式不同：**HTTP 接口通过 URL 进行调用，RPC 接口通过函数调用进行调用。 参数传递方式不同：HTTP 接口使用 URL 参数或者请求体进行参数传递，RPC 接口使用函数参数进行传递。**

以下是一个 RPC 接口的示例代码，使用 Protocol Buffers 和 gRPC 框架实现：

```
syntax = "proto3";

package helloworld;

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

在该示例中，定义了一个 RPC 接口 `Greeter` ，包含一个方法 `SayHello` ，输入参数为 `HelloRequest` ，输出参数为 `HelloReply` 。

http(based on tcp) or tcp packet has Head-of-line blocking
HoL blocking arises when a packet that is blocked due to link congestion precludes another packet traveling in the same channel (but which requires a different outbound link) from making forward progress.

How to solve it ? rpc comes using multiplexing or asynchronous way to handle many requests

HTTP 接口和 RPC 接口的应用场景

HTTP 接口适用于 Web 应用程序和浏览器之间的通信。它通常用于传输 HTML、CSS、JavaScript 和其他 Web 资源，以及 RESTful 风格的 API 服务。

RPC 接口适用于分布式系统之间的通信。它可以在多种编程语言之间进行通信，支持多种协议和数据格式。RPC 接口通常用于处理高并发、高吞吐量的场景，例如大型的分布式计算、大数据处理等。

6. Montior responsibility

Heartbeat detection deals with the abnormal downtime of a service (such as a power outage), when the registry **does not know that** the service is offline and can cause problems if it is called**.In order to avoid this situation, the registry will add a heartbeat detection function, which will detect the heartbeat of the service Provider.** For example, it will send a heartbeat every 30 seconds. If the three heartbeat results do not return a value, it will consider that the service has been offline, and update the

Consumer's service list immediately.Tell the Consumer to call another machine.

7. TCP and UDP

G. Redis

1. redis datastructure

string
list
hash
set
zset
Bitmap
Bitmap is a bit to support the storage of information, can be used to implement Bloom filter（BloomFilter）;

2. redis usage scenarios

- Caching of hot data

Like login user information.

- control concurrency

We use incr to avoid two users start the machine in the meanwhile

- Leaderboards

Use zset

- filter

Bitmap datastruture

- Reduce latency
- Improved system performance
- Reduced network load

3. Consistency issue or Cache Invalidation Strategies

https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/64224412a763c048c9763975

https://betterprogramming.pub/how-to-ensure-the-consistency-between-redis-and-database-62f09de0bdde

4. redis and mongoDB

https://www.cnblogs.com/111testing/p/11190823.html

Redis: used as cache;much higher TPS; high concurrency;requests of high frequency;less data scenarios
mongoDB: used as DB;higher TPS; has mapreduce(analyse data);hugedata scenarios

现在的分布式项目基本都会用到redis和mongodb，可是redis和mongdb到底有什么不同呢，今天我就基于我们公司的项目来具体介绍一下redis和mongodb的各自的应用场景。
首先我们这个项目中有两种应用场景：
场景一：要求TPS（不知道的右转百度）特别高的，比如我们项目有一个点赞的功能，这个点赞的功能促发频率特别高，而且并发量也会特别大，但是它的数据量不会特别大。基于这种情况下，我们采用redis来实现点赞功能。
场景二：项目中涉及评论的内容，而且这个评论表的数据后期会非常大（海量的数据），最后在数据量非常大的情况下还要求比较复杂的查询。基于上述这些情况，我们采用mongodb作为评论表存储数据库。

1. java
18)
2. andr
法(706
3. pyth
y文件中
4. Excel
色(4456
5. java

5. how to solve REDIS DB inconsistency

Critical :
https://betterprogramming.pub/how-to-ensure-the-consistency-between-redis-and-database-62f09de0bdde

including the following:

- Set a slightly shorter expiration time: within this expiration time, it will be inconsistent. The disadvantage is that a shorter expiration time means a higher load on the database
- Double delete: delete the cache once, delay for a few hundred milliseconds and then delete it again. This approach only further reduces the probability of inconsistency, but it is not forbidden

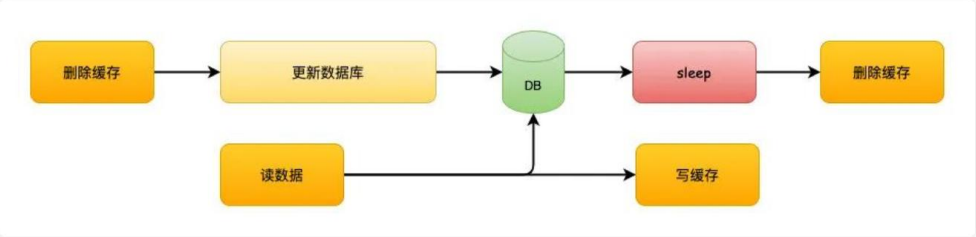
https://blog.csdn.net/qinlingcici/article/details/119519250

Best explanation:

sleep的时间要对业务读写缓存的时间做出评估，sleep时间大于读写缓存的时间即可。

流程如下：

1. 线程1删除缓存，然后去更新数据库
2. 线程2来读缓存，发现缓存已经被删除，所以直接从数据库中读取，这时候由于线程1还没有更新完成，所以读到的是旧值，然后把旧值写入缓存
3. 线程1，根据估算的时间，sleep，由于sleep的时间大于线程2读数据+写缓存的时间，所以缓存被再次删除
4. 如果还有其他线程来读取缓存的话，就会再次从数据库中读取到最新值



在线程A更新完数据库值以后，我们可以让它先sleep一小段时间，再进行一次缓存删除操作。

之所以要加上sleep的这段时间，就是为了让线程B能够先从数据库读取数据，再把缺失的数据写入缓存，然后，线程A再进行删除。所以，线程A sleep的时间，就需要大于线程B读取数据再写入缓存的时间。这个时间怎么确定呢？建议你在业务程序运行的时候，统计下线程读数据和写缓存的操作时间，以此为基础来进行估算。

这样一来，其它线程读取数据时，会发现缓存缺失，所以会从数据库中读取最新值。因为这个方案会在第一次删除缓存值后，延迟一段时间再次进行删除，所以我们也把它叫做“延迟双删”。

下面的这段伪代码就是“延迟双删”方案的示例，你可以看下。

```
redis.delKey(X)
db.update(X)
Thread.sleep(N)
redis.delKey(X)
```

为什么要延时呢？因为 mysql 和 redis 主从节点数据不是实时同步的，同步数据需要时间。

数据工作的大致流程：

1. 服务节点删除 redis 主库数据。
2. 服务节点修改 mysql 主库数据。
3. 服务节点使得当前业务处理 `等待一段时间` ，等 redis 和 mysql 主从节点数据同步成功。
4. 服务节点从 redis 主库删除数据。
5. 当前或其它服务节点读取 redis 从库数据，发现 redis 从库没有数据，从 mysql 从库读取数据，并写入 redis 主库。

6. all questions about REDIS

https://github.com/AobingJava/JavaFamily

- 读懂Redis源码，我总结了这7点心得
- Redis常见线上故障及其解决方案
  - 缓存击穿、雪崩、穿透
  - 布隆过滤器实战【防止缓存击穿】
  - 分布式锁、并发竞争、双写一致性
- Redis常见面试题
  - Redis面试题
  - Redis常见面试题
  - Redis为什么变慢了？一文讲透如何排查Redis性能问题
  - Redis不是一直号称单线程效率也很高吗，为什么又采用多线程了？
  - 缓存一致性问题怎么解决？
  - 内存耗尽后Redis会发生什么？
  - 妈妈再也不担心我面试被Redis问得脸都绿了

7. why does redis so rapid

.	
	内存存储： Redis 通常将数据存储在内存中，这使得数据的读写速度非常快，因为内存访问速度远高于磁盘访问速度。
. .	
	单线程模型： Redis 采用单线程的事件驱动模型，避免了多线程的锁竞争和上下文切换带来的开销。这在一定程度上简化了并发控制，对于处理大量的快速操作非常有效。
. .	
	非阻塞 I/O： Redis 使用非阻塞 I/O，通过事件轮询机制来处理多个客户端连接，从而在同一个线程内高效处理多个客户端请求。
. .	
	高效数据结构： Redis 使用了多种高效的数据结构，如哈希表、跳表等，来支持不同的数据操作。其中，动态字符串和跳表是两个关键的数据结构。

8. what is skip list

https://juejin.cn/post/6893072817206591496

目的:
主要解决链表查询 o(n)的问题
局限:
只能用于有序链表
定义:
跳表（skip list）对表的是平衡树（AVL Tree）和 二分查找，是一种 插入/删除/搜索 都是 O(logn) 的数据结构。

9. simple dynamic string

https://gentryhuang.com/posts/aa1d8127/index.html

Redis 使用最多的数据结构

2. sds的结构

```c
struct sdshdr{
  // 记录buf数组中已使用字节的数量等价于sds所保存字符串的长度
  int len;
  // 记录buf数组中未使用字节的数量
  int free;
  // 字节数组，用于保存字符串
  char buf[];
}
```

c语言使用的简单字符串表示方式，并不能满足redis对字符串在安全性、效率以及功能方面的要求。

3.1 常数复杂度获取字符串长度

> 因为c字符串并不记录自身的长度信息，所以要获取长度就必须遍历整个字符串，故获取字符串长度的复杂度为O(N)。而redis的sds结构中通过len属性记录了sds的长度，故获取字符串长度的复杂度为O(1)。注意，设置和更新sds长度的工作是由sds的API在执行时自动完成的，使用sds无须进行任何手动修改长度的操作。

3.2 避免缓冲区溢出

> c字符串不记录自身长度，很容易造成缓冲区溢出。与c字符串不同，sds的空间分配策略完全杜绝了发生缓冲区溢出的可能性，当sds api需要对sds进行修改时，api会先检查sds的空间是否满足需要，如果不满足的话，api会自动将sds的空间扩展至执行所需的大小，然后才执行实际的修改操作，因此使用sds既不需要手动修改sds的空间大小，也不会出现c字符串中可能出现的缓冲区溢出问题。

10.Why does redis use hash index

Redis 使用的是哈希索引（hash index）来支持高效的数据存储和查找，这种索引结构在某些场景下能够提供非常快速的数据访问速度。以下是一些原因解释为什么 Redis 选择使用哈希索引：

1. **快速查找和存储**：
   哈希索引允许将每个数据元素与一个唯一的键相关联，使得在读取和写入数据时可以以O(1)的平均时间复杂度进行操作。这使得 Redis 在缓存、键值存储等场景下能够以非常高的速度进行数据访问。

2. **内存效率**：
Redis 在内存中存储数据，而哈希索引允许将键映射到相应的内存地址。相对于传统的数据库系统中使用的 B 树或 B+树索引结构，哈希索引通常更紧凑，因此可以更有效地利用内存空间。

3. **简单的数据结构**：
哈希索引是一种相对简单的数据结构，具有较低的复杂性。这使得 Redis 的实现更加轻量级，并且有助于提高性能，特别是在数据量较小且访问频繁的情况下。

4. **高并发读写**：
哈希索引适用于高并发读写场景。由于其 O(1)的查找和存储时间复杂度，它能够快速响应来自多个客户端的读写请求。

尽管哈希索引在很多情况下表现良好，但也有一些限制。例如，哈希索引通常不适用于范围查询（例如查找在某个范围内的键值对）。此外，哈希索引的缺点之一是，它不支持对键的字典排序。

总的来说，Redis 选择使用哈希索引是为了在大多数常见的键值存储和缓存场景中提供高效、快速的数据存储和访问方式。

11.redis 数据持久化,可用性

https://www.jianshu.com/p/c26e6dc94546

但是持久化就一定能保证数据不丢失吗?

首先我们要了解Redis为了保证高可用性，一般都是从一下几个方面入手的：

- 数据持久化
- 主从复制
- 自动故障恢复
- Redis集群

常见的Redis持久化有两种：（1）RDB快照持久化（2）AOF持久化，区别在于RDB持久化保存完整的数据，AOF缓存的是redis指令（当然我们该可以简化AOF缓存的指令）。缺点也很明显，RDB是某一时刻的数据快照，因此它的数据并不全，生成RDB文件的代价是比较大的，它会消耗大量的CPU和内存资源。RDB持久化适用于主从全量同步数据、数据库备份。AOF适合做增量数据的持久化。Redis启动会优先加载AOF文件进行数据恢复。

12.Cache Replacement Policies

LRU :the least recently used item
LFU : the least frequently used
FIFO:

13. redis 常见故障

- **雪崩**

何为雪崩

举个简单的例子：如果所有首页的Key失效时间都是12小时，中午12点刷新的，我零点有个秒杀活动大量用户涌入，假设当时每秒 6000 个请求，本来缓存在可以扛住每秒 5000 个请求，但是缓存当时所有的Key都失效了。此时 1 秒 6000 个请求全部落数据库，数据库必然扛不住，它会报一下警，真实情况可能DBA都没反应过来就直接挂了。此时，如果没用什么特别的方案来处理这个故障，DBA 很着急，重启数据库，但是数据库立马又被新的流量给打死了。这就是我理解的缓存雪崩。

解决办法

处理缓存雪崩简单，在批量往**Redis**存数据的时候，把每个Key的失效时间都加个随机值就好了，这样可以保证数据不会在同一时间大面积失效，我相信，Redis这点流量还是顶得住的。

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

或者设置热点数据永远不过期，有更新操作就更新缓存就好了（比如运维更新了首页商品，那你刷下缓存就完事了，不要设置过期时间），电商首页的数据也可以用这个操作，保险。

- **穿透**

何为穿透

嗯，了解，我先说一下缓存穿透吧，缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，我们数据库的 id 都是1开始自增上去的，如发起为id值为 -1 的数据或 id 为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大，严重会击垮数据库。

解决办法:

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

或者用 redis 里的 boomfilter 做校验

击穿:

至于**缓存击穿**嘛，这个跟**缓存雪崩**有点像，但是又有一点不一样，缓存雪崩是因为大面积的缓存失效，打崩了DB，而缓存击穿不同的是**缓存击穿**是指一个Key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个Key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个完好无损的桶上凿开了一个洞。

解决办法:

缓存击穿的话，设置热点数据永远不过期。或者加上互斥锁就能搞定了

总结

一般避免以上情况发生我们从三个时间段去分析下：

- 事前：**Redis** 高可用，主从+哨兵，**Redis cluster**，避免全盘崩溃。
- 事中：本地 **ehcache** 缓存 + **Hystrix** 限流+降级，避免**MySQL**被打死。
- 事后：**Redis** 持久化 **RDB+AOF**，一旦重启，自动从磁盘上加载数据，快速恢复缓存数据。

上面的几点我会在吊打系列Redis篇全部讲一下这个月应该可以吧Redis更完，限流组件，可以设置每秒的请求，有多少能通过组件，剩余的未通过的请求，怎么办？**走降级**！可以返回一些默认的值，或者友情提示，或者空白的值。

好处：

数据库绝对不会死，限流组件确保了每秒只有多少个请求能通过。只要数据库不死，就是说，对用户来说，3/5 的请求都是可以被处理的。只要有 3/5 的请求可以被处理，就意味着你的系统没死，对用户来说，可能就是点击几次刷不出来页面，但是多点几次，就可以刷出来一次。

H. Kafka or MQ

Scenarios of using MQ:

- asynchronous processing
- decoupling services.
- Flow control

Type:
Queue and
Pub/sub

1. How to ensure that messages are not lost

Producer message Stage:

- The producer sends a message to the Broker and needs to process the Broker's response. Whether the message is sent synchronously or asynchronously, both synchronous and asynchronous callbacks need to do a good try-catch to properly process the response. If the Broker returns an error message such as a failed write, the sending needs to be **retried**. **Alarm, logging and so on are required when**

sending fails for several times.

生产消息

生产者发送消息至 `Broker` ，需要处理 `Broker` 的响应，不论是同步还是异步发送消息，同步和异步回调都需要做好 `try-catch` ，妥善的处理响应，如果 `Broker` 返回写入失败等错误消息，需要重试发送。当多次发送失败需要作报警，日志记录等。

这样就能保证在生产消息阶段消息不会丢失。

Store Message Stage:

- **Broker should be deployed in a cluster,**

存储消息

存储消息阶段需要在**消息刷盘之后**再给生产者响应，假设消息写入缓存中就返回响应，那么机器突然断电这消息就没了，而生产者以为已经发送成功了。

如果 `Broker` 是集群部署，有多副本机制，即消息不仅仅要写入当前 `Broker` ,还需要写入副本机中。那配置成至少写入两台机子后再给生产者响应。这样基本上就能保证存储的可靠了。一台挂了还有一台还在呢（假如怕两台都挂了..那就再多些）。

那假如来个地震机房机子都挂了呢？emmmmmm...大公司基本上都有异地多活。

那要是这几个地都地震了呢？emmmmmm...这时候还是先关心关心人吧。

Consume Message Stage:

- First of all , close auto-ack.So we should ,after the consumer has completely executed the business logic, **then send to the Broker signal that consumption succeeds.**

这里经常会有同学犯错，有些同学当消费者拿到消息之后直接存入内存队列中就直接返回给 `Broker` 消费成功，这是不对的。

你需要考虑拿到消息放在内存之后消费者就宕机了怎么办。所以我们应该在**消费者真正执行完业务逻辑之后，再发送给 `Broker` 消费成功**，这才是真正的消费了。

https://www.51cto.com/article/648847.html

kafka
https://www.lixueduan.com/posts/kafka/09-avoid-msg-lost/

2. How to deal with duplicated Messages

idempotent:

Add a field called version, every time before consuming a message , we should judge whether version equals 1. if not ,then consume, otherwise do nothing. When consuming add 1 to version.

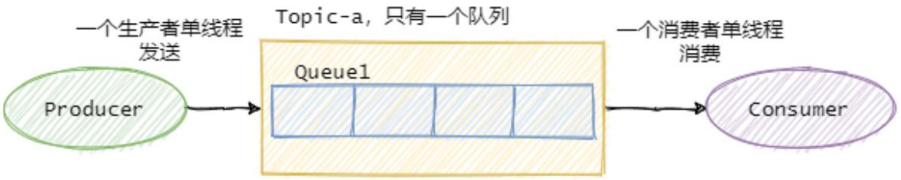
3. How to ensure the order of the message

We can **divide the Topic into the number of queues** we need, send messages to **fixed queues** using a specific policy, **and then each queue corresponds to a single threaded consumer.This meets part of the order requirement and improves message processing efficiency through concurrency of the number of queues.**

全局有序

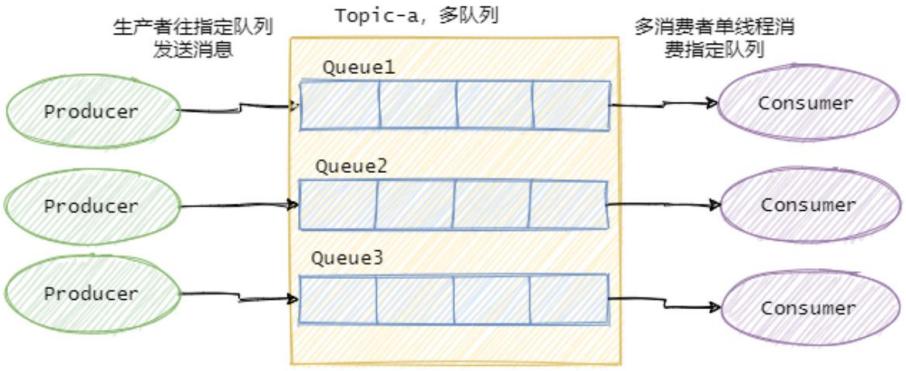
如果要保证消息的全局有序，首先只能由一个生产者往 `Topic` 发送消息，并且一个 `Topic` 内部只能有一个队列（分区）。消费者也必须是单线程消费这个队列。这样的消息就是全局有序的！

不过一般情况下我们都不需要全局有序，即使是同步 `MySQL Binlog` 也只需要保证单表消息有序即可。



部分有序

因此绝大部分的有序需求是部分有序，部分有序我们就可以将 `Topic` 内部划分成我们需要的队列数，把消息通过特定的策略发往固定的队列中，然后每个队列对应一个单线程处理的消费者。这样即完成了部分有序的需求，又可以通过队列数量的并发来提高消息处理效率。



4. How do you handle message piling

Messages tend to pile up because speed of producing does not match the speed of consuming.

At first check whether there is a bug. Or whether the code has lower performance.

If not then :

Horizontal expansion: increase **the number of queues and consumers of Topic.** Note that the numb**er of queues must be increased, otherwise the newly add**ed consumers will have nothing to consume.Within a Topic, a queue is assigned to only one consumer.

如果处理消息堆积

消息的堆积往往是因为**生产者的生产速度与消费者的消费速度不匹配**。有可能是因为消息消费失败反复重试造成的，也有可能就是消费者消费能力弱，渐渐地消息就积压了。

因此我们需要**先定位消费慢的原因**，如果是 `bug` 则处理 `bug` ，如果是因为本身消费能力较弱，我们可以优化下消费逻辑，比如之前是一条一条消息消费处理的，这次我们批量处理，比如数据库的插入，一条一条插和批量插效率是不一样的。

假如逻辑我们已经都优化了，但还是慢，那就得考虑水平扩容了，增加 `Topic` 的队列数和消费者数量，**注意队列数一定要增加**，不然新增加的消费者是没东西消费的。**一个Topic中，一个队列只会分配给一个消费者**。

当然你消费者内部是单线程还是多线程消费那看具体场景。不过要注意上面提高的消息丢失的问题，如果你是将接受到的消息写入**内存队列**之后，然后就返回响应给 `Broker` ，然后多线程向内存队列消费消息，假设此时消费者宕机了，内存队列里面还未消费的消息也就丢了。

5. 什么是死信队列 Dead letter QUEUE

https://mfrank2016.github.io/breeze-blog/2020/05/04/rabbitmq/rabbitmq-how-to-use-dead-letter-queue/

死信队列（DLQ）是一种特殊类型的消息队列，用于临时存储软件系统由于错误而无法处理的消息。 消息队列是支持分布式系统中的异步通信的软件组件。 借助它们，您可以在软件服务之间发送任何数量的消息，且不需要消息接收器始终处于可用。 死信队列特别用于存储没有目的地或无法由预期接收器处理的错误的消息。

点亮勋章

死信队列

死信队列：没有被及时消费的消息存放的队列。

> 一般来说，producer将消息投递到queue中，consumer从queue取出消息进行消费，但某些时候由于特定的原因导致queue中的某些消息无法被消费，这样的消息如果没有后续的处理，就变成了死信(Dead Letter)，所有的死信都会放到死信队列中。“死信”消息会被RabbitMQ进行特殊处理，如果配置了死信队列信息，那么该消息将会被丢进死信队列中，如果没有配置，则该消息将会被丢弃。

死信消息具有以下特性：

不会再被消费者正常消费。
有效期与正常消息相同，均为3天，3天后会被自动删除。因此，请在死信消息产生后的3天内及时处理。

死信队列具有以下特性：

一个死信队列对应一个Group ID， 而不是对应单个消费者实例。
如果一个Group ID未产生死信消息，消息队列RocketMQ版不会为其创建相应的死信队列。
一个死信队列包含了对应Group ID产生的所有死信消息，不论该消息属于哪个Topic。

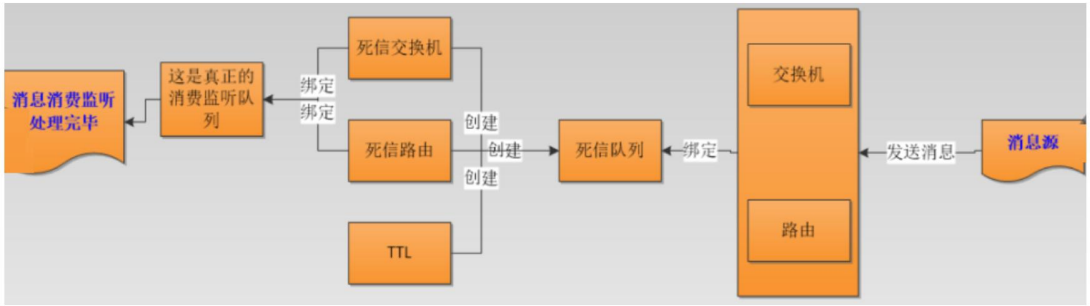
点亮勋章

造成原因

消息被拒绝(basic.reject / basic.nack)，并且requeue = false

消息TTL过期

队列达到最大长度（队列满了，无法再添加数据到mq中）



点亮勋章

应用场景

一般用在较为重要的业务队列中，确保未被正确消费的消息不被丢弃，一般发生消费异常可能原因主要有由于消息信息本身存在错误导致处理异常，处理过程中参数校验异常，或者因网络波动导致的查询异常等等，当发生异常时，当然不能每次通过日志来获取原消息，然后让运维帮忙重新投递消息。通过配置死信队列，可以让未正确处理的消息暂存到另一个队列中，待后续排查清楚问题后，编写相应的处理代码来处理死信消息，这样比手工恢复数据要好太多了 。

查询方式

查询方式	查询条件	查询类别	说明
按Group ID查询	Group ID + 时间段	范围查询	根据Group ID和时间范围，批量获取符合条件的所有消息；查询量大，不易匹配。
按Message ID查询	Group ID + Message ID	精确查询	根据Group ID和Message ID可以精确定位任意一条消息。

4、死信队列的应用

- 保证消息不会丢失，保证数据的完整性；
- 可以借助延时消费的特性完成特定的功能(比如订单生成但是未支付，超过30分钟自动取消的业务场景)

1、什么是死信

死信产生主要来自于两个过程角色

- 来自于消费者端
- 来自于queue

产生死信的三种情况：

- 如果queue中的消息被消费者接收， 但是消费者拒绝消费（消费者执行了reject 或nack 并将 requee 参数设置为 false ）的时候，这个消息就会变成死信。
- 消息本身设置了过期时间(TTL), 并且消息过期时间已经生效， 还未被消费的消息就会变成死信【特点是每个消息的过期时间都不同】
- 可以设置队列中所有消息的过期时间，如果消息过期时间已经生效，消息还未被消费
- 队列设置了最大长度限制， 当队列已满， 之后从交换机路由到该队列的消息会自动变成死信。

English:

The dead letter queue pattern is used to handle erroneous or invalid messages that cannot be processed by consumers. These messages are sent to a dedicated queue called the dead letter queue, where they can be monitored, analyzed, or reprocessed. **This pattern helps to ensure that problematic messages do not block the processing of other messages in the system.** Key characteristics of Dead Letter Queue messaging include:

- Error handling: Unprocessable messages are redirected to the DLQ, preventing

them from blocking the primary message queue.

- Monitoring and debugging: DLQs can be monitored to detect issues with message processing and help identify the root cause of problems.
- Retention: Messages in the DLQ can be retained for a specific period, allowing for the possibility of retries or manual intervention.
- Fault isolation: DLQs help isolate problematic messages, reducing the impact of errors on the rest of the system.

Example: An email delivery system where messages that fail to be delivered (due to invalid addresses, full mailboxes, etc.) are sent to a dead letter queue. Administrators can then inspect the messages, fix any issues, and attempt to redeliver them.

6. Messaging patterns

Point-to-Point (Direct Messaging):

Example: An order processing system where an order is placed by a customer, and a single consumer processes the order. The order messages are stored in a queue, and each consumer picks up one order at a time for processing.

Publish-Subscribe (Pub/Sub):

In the publish-subscribe pattern, messages are sent from a producer (publisher) to multiple consumers (subscribers). The messages are sent to a topic, and all subscribers that have expressed interest in that topic receive the messages. This pattern is ideal for applications requiring the dissemination of information to multiple recipients or when the sender doesn't need to know who the recipients are. Key characteristics of publish-subscribe messaging include:

Example: A stock market ticker application where stock price updates are sent to a topic. Multiple subscribers, such as trading applications, financial news services, or individual investors, receive these updates in real time.

Request-Reply (Request-Response):

In the request-reply pattern, a producer (requester) sends a message to a consumer (responder) and waits for a response. This pattern is often used for synchronous communication between two parties, where the sender requires a response before continuing its operation.

Example: An e-commerce application where a user submits a payment request. The payment gateway processes the request and sends back a response indicating whether the payment was successful or not. The e-commerce application waits for the response before proceeding with the order confirmation. Key characteristics of request-reply messaging include:

- Synchronous communication: This pattern is often used for synchronous communication between components, where the consumer waits for the reply before proceeding.
- Coupling: The pattern can introduce tighter coupling between producers and consumers since the consumer must wait for a reply.
- Latency: Request-reply messaging can introduce latency as the producer must wait for the reply to arrive.

7. Kafka structure and interview questions

https://www.lixueduan.com/categories/kafka/

8. RabbitMQ vs. Kafka vs. ActiveMQ: A Battle of Messaging Brokers

Here are the top differences between RabbitMQ, Kafka, and ActiveMQ:

- **Performance and Scalability**: Kafka is designed for high throughput and horizontal scalability, making it well-suited for handling large volumes of data. RabbitMQ and ActiveMQ both offer high performance, but Kafka generally outperforms them in terms of throughput, particularly in scenarios with high data volume.

- Message Ordering: RabbitMQ and ActiveMQ guarantee message ordering within a single queue or topic, respectively. Kafka ensures message ordering within a partition but not across partitions within a topic.

- Message Priority: RabbitMQ and ActiveMQ support message prioritization, allowing messages with higher priority to be processed before those with lower priority. Kafka does not have built-in message priority support.

- Message Model: RabbitMQ uses a queue-based message model following the Advanced Message Queuing Protocol (AMQP), while Kafka utilizes a distributed log-based model. ActiveMQ is built on the Java Message Service (JMS) standard and also uses a queue-based message model.

- Durability: All three message brokers support durable messaging, ensuring that messages are not lost in case of failures. However, the mechanisms for achieving durability differ among the three, with RabbitMQ and ActiveMQ offering configurable durability options and Kafka providing built-in durability through log replication.

- Message Routing: RabbitMQ provides advanced message routing capabilities through exchanges and bindings, while ActiveMQ uses selectors and topics for more advanced routing. Kafka's message routing is relatively basic and relies on topic-based partitioning.

- Replication: RabbitMQ supports replication through Mirrored Queues, while Kafka features built-in partition replication. ActiveMQ uses a Master-Slave replication mechanism.

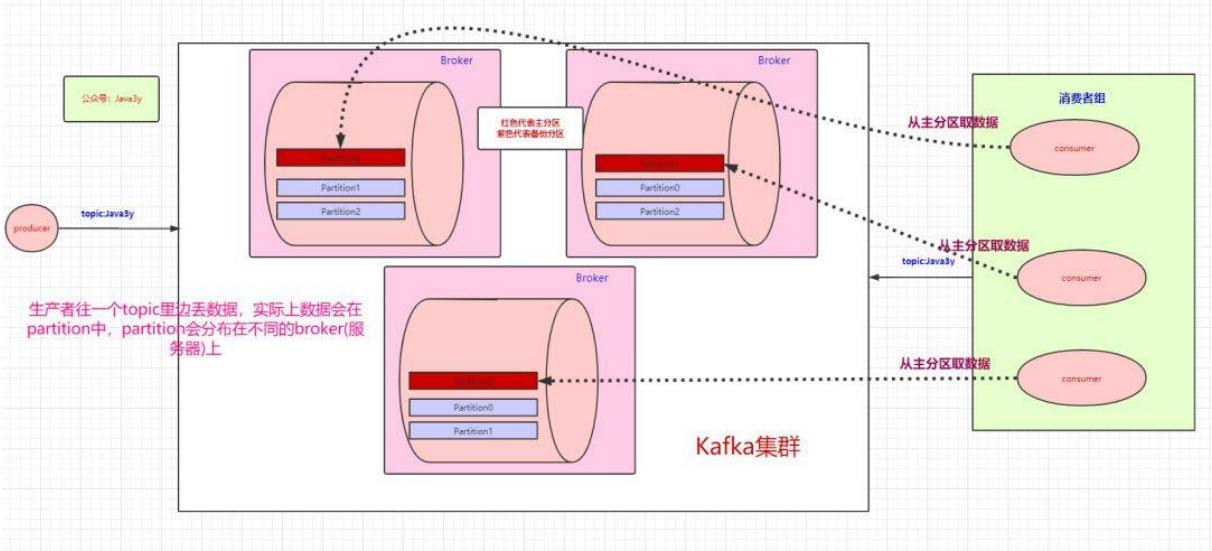
- Stream Processing: Kafka provides native stream processing capabilities through Kafka Streams, similarly RabbitMQ offers stream processing too, while ActiveMQ relies on third-party libraries for stream processing.

- Latency: RabbitMQ is designed for low-latency messaging, making it suitable for use cases requiring near-real-time processing.

- License: RabbitMQ is licensed under the Mozilla Public License, while both Kafka and ActiveMQ are licensed under the Apache 2.0 License.

9. kafka 原理

https://github.com/AobingJava/JavaFamily/blob/master/docs/all/mq/kafka.md



前面讲解到了生产者往topic里丢数据是存在partition上的，而partition持久化到磁盘是IO顺序访问的，并且是先写缓存，隔一段时间或者数据量足够大的时候才批量写入磁盘的。

Offset 机制：

吗？那是怎么做的呀？

- 比如上面也提到：如果一个消费者组中的某个消费者挂了，那挂掉的消费者所消费的分区可能就由存活的消费者消费。那**存活的消费者是需要知道挂掉的消费者消费到哪了**，不然怎么玩。

这里要引出 `offset` 了，Kafka就是用 `offset` 来表示消费者的消费进度到哪了，每个消费者会都有自己的 `offset` 。说白了 `offset` 就是表示消费者的**消费进度**。

在以前版本的Kafka，这个 `offset` 是由Zookeeper来管理的，后来Kafka开发者认为Zookeeper不合适大量的删改操作，于是把 `offset` 在broker以内部topic(`__consumer_offsets`)的方式来保存起来。

每次消费者消费的时候，都会提交这个 `offset` ，Kafka可以让你选择是自动提交还是手动提交。

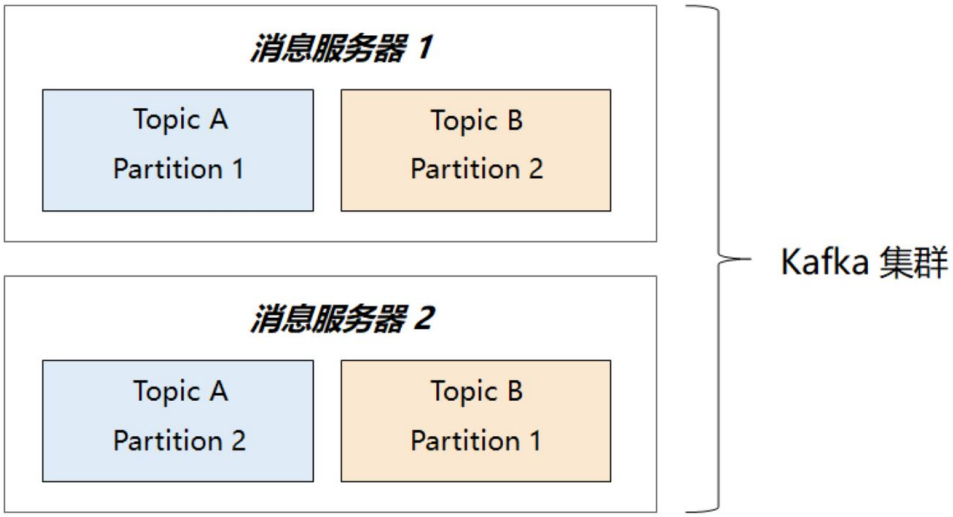
既然提到了Zookeeper，那就多说一句。Zookeeper虽然在新版的Kafka中没有用作于保存客户端的 `offset` ，但是Zookeeper是Kafka一个重要的依赖。

10.Kafka 架构设计

https://mp.weixin.qq.com/s/8wfZEsNDpeLr-_uu2CawFw

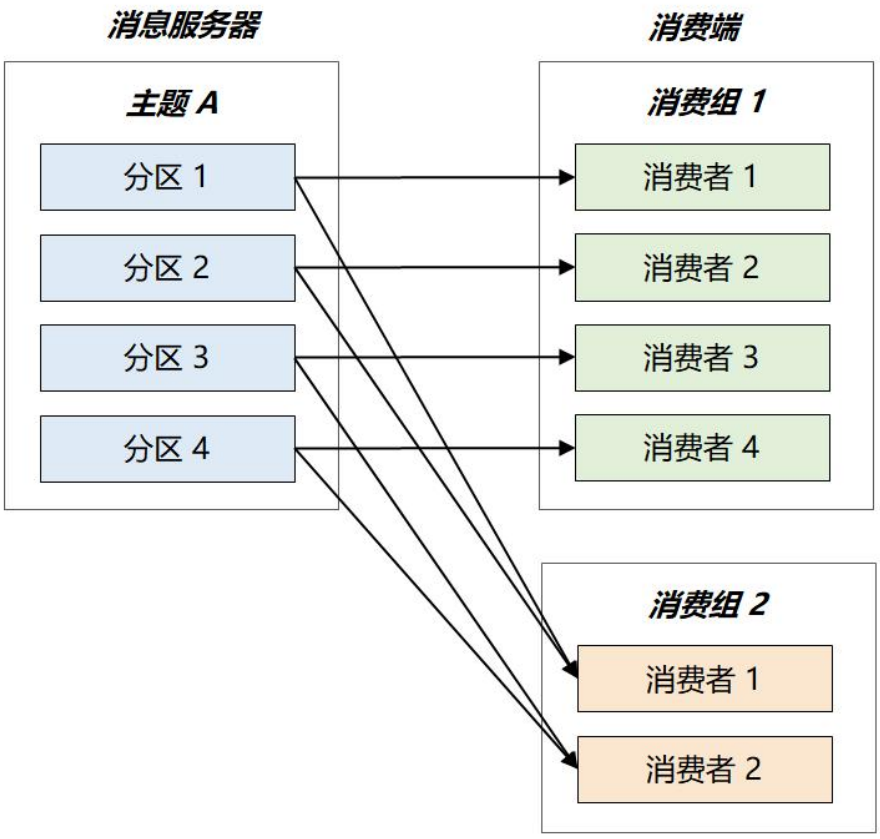
高性能:

1 partition 从存储着手



可以看到：同一个 Topic 的两个 Partition 分布在不同的消息服务器上，能做到消息的分布式存储了。但是对于 Kafka 这个高并发系统来说，仅存储可扩展还不够，消息的拉取也必须并行才行，否则会遇到极大的性能瓶颈。

2 从消费者着手



如果要加快消息的处理速度，该如何做呢？也很简单，向消费组 2 中增加新的消费者即可，Kafka 将以 Partition 为单位重新做负载均衡。当增加到 4 个消费者时，每个消费者仅需处理 1 个 Partition，处理速度将提升两倍。

高可用:

在 Kafka 集群中，每台机器都存储了一些 Partition，一旦某台机器宕机，上面的数据不就丢失了吗？

此时，你一定会想到对消息进行持久化存储，但是持久化只能解决一部分问题，它只能确保机器重启后，历史数据不丢失。但在机器恢复之前，这部分数据将一直无法访问。这对于高并发系统来说，是无法忍受的。

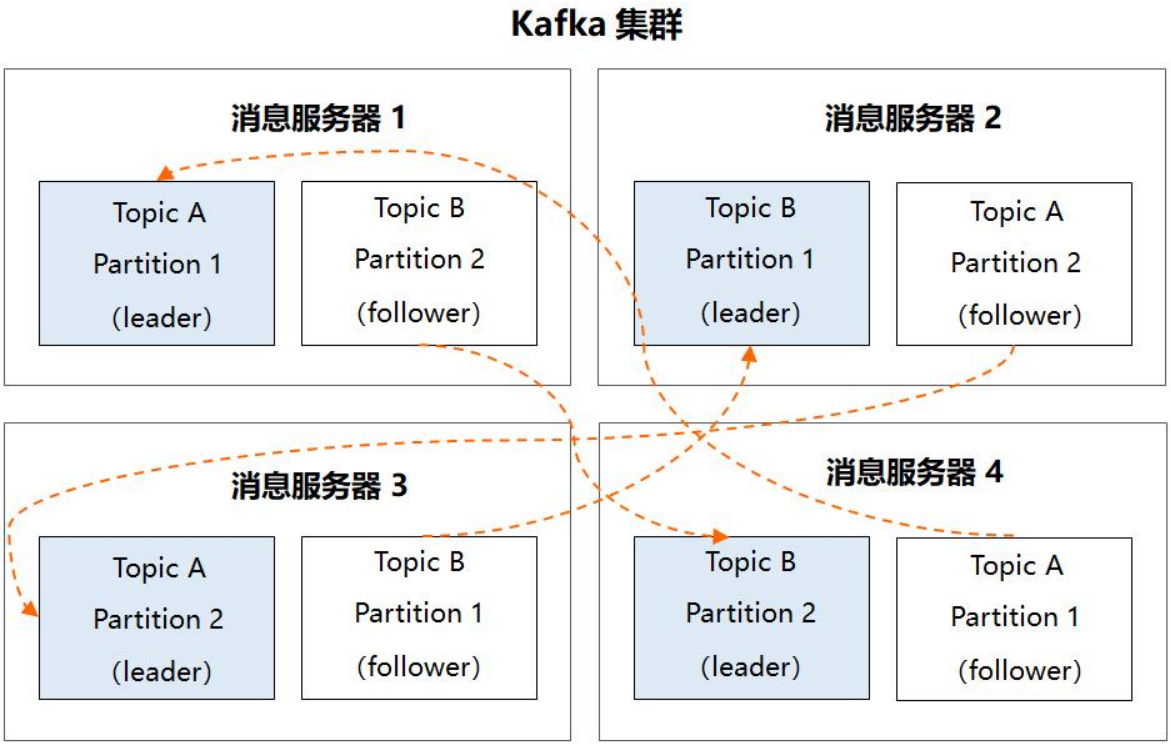
所以 Kafka 必须具备故障转移能力才行，当某台机器宕机后仍然能保证服务可用。

如果大家去分析任何一个高可靠的分布式系统，比如 ElasticSearch、Redis Cluster，其实它们都有一套多副本的冗余机制。

没错，Kafka 正是通过 Partition 的多副本机制解决了高可用问题。在 Kafka 集群中，每个 Partition 都有多个副本，同一分区的不同副本中保存的是相同的消息。

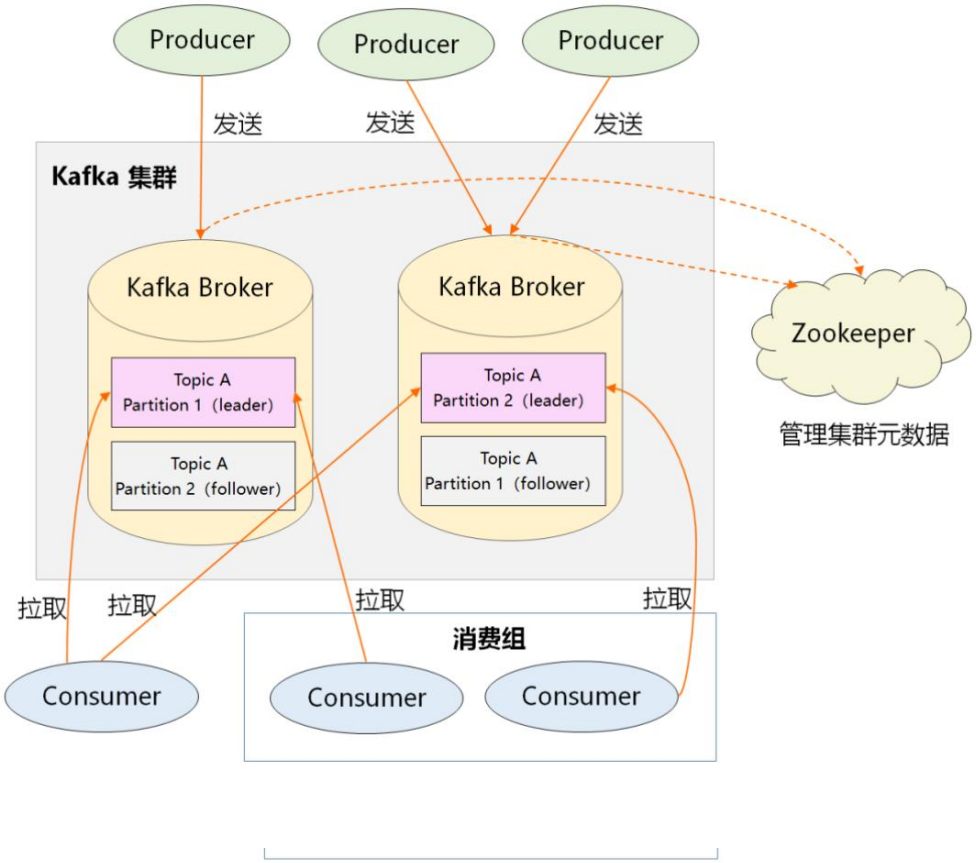
副本之间是“一主多从”的关系，其中 leader 副本负责读写请求，follower 副本只负责和 leader 副本同步消息，当 leader 副本发生故障时，它才有机会被选举成新的 leader 副本并对外提供服务，否则一直是待命状态。

现在，我假设 Kafka 集群中有 4 台服务器，主题 A 和主题 B 都有两个 Partition，且每个 Partition 各有两个副本，那最终的多副本架构将如下图所示：



整体架构

理解了上面这些内容，最后我们再反过来看下 Kafka 的整体架构：



1、Producer：生产者，负责创建消息，然后投递到 Kafka 集群中，投递时需要指定消息所属的 Topic，同时确定好发往哪个 Partition。

2、Consumer：消费者，会根据它所订阅的 Topic 以及所属的消费组，决定从哪些 Partition 中拉取消息。

3、Broker：消息服务器，可水平扩展，负责分区管理、消息的持久化、故障自动转移等。

4、Zookeeper：负责集群的元数据管理等功能，比如集群中有哪些 broker 节点以及 Topic，每个 Topic 又有哪些 Partition 等。

11. Kafka 为什么性能这么好

https://mp.weixin.qq.com/s/kMIhPW2uLdy-mgS9sF6agw
https://mp.weixin.qq.com/s/kImrkVLE4dtpVnb-Yp479Q
https://mp.weixin.qq.com/s/YJFltTP4J5si1Z5SbuMUJw

采用的典型方式:

- **从 IO 的角度(多用复路 IO)**
- **从消息本身处理的角度(compressor)**
- **数据处理角度(batch or stream 和内存池复用):**

前面说过 Producer 发送消息是批量的，因此消息都会先写入 Producer 的内存中进行缓冲，直到多条消息组成了一个 Batch，才会通过网络把 Batch 发给 Broker。

当这个 Batch 发送完毕后，显然这部分数据还会在 Producer 端的 JVM 内存中，由于不存在引用了，它是可以被 JVM 回收掉的。

但是大家都知道，JVM GC 时一定会存在 Stop The World 的过程，即使采用最先进的垃圾回收器，也势必会导致工作线程的短暂停顿，这对于 Kafka 这种高并发场景肯定会带来性能上的影响。

有了这个背景，便引出了 Kafka 非常优秀的内存池机制，它和连接池、线程池的本质一样，都是为了提高复用，减少频繁的创建和释放。

具体是如何实现的呢？其实很简单：Producer 一上来就会占用一个固定大小的内存块，比如 64MB，然后将 64 MB 划分成 M 个小内存块（比如一个小内存块大小是 16KB）。

当需要创建一个新的 Batch 时，直接从内存池中取出一个 16 KB 的内存块即可，然后往里面不断写入消息，但最大写入量就是 16 KB，接着将 Batch 发送给 Broker ，此时该内存块就可以还回到缓冲池中继续复用了，根本不涉及垃圾回收。最终整个流程如下图所示：

- **存储角度 partition**
- **消费端批量拉取**

4、批量拉取

和生产者批量发送消息类似，消息者也是批量拉取消息的，每次拉取一个消息集合，从而大大减少了网络传输的 overhead。

https://www.lixueduan.com/categories/kafka/

12.Kafka 怎样处理消息丢失

https://www.51cto.com/article/646916.html

I. Devops and CI/CD

1. CI /cd pipeline

Github;; jenkins ;ansible

1.1 how to do performance test

Jenkins+ jmeter, how to setup
https://www.cnblogs.com/a00ium/p/10847823.html

https://www.youtube.com/watch?v=99ar2jEwoEg

What is the jmx file

Basically , it is the test plan, including the jdbcdata source ,result tree , parameter,

samplers, assertion
https://www.cnblogs.com/iloverain/p/9165829.html

1.2 how to do integraton test

Jenkins + postman
https://www.cnblogs.com/leiziv5/p/12950320.html

2. Auto test

Jenkins + Jmeter: how to setup performance test

https://blog.csdn.net/weixin_41948075/article/details/97894080

Dynatrace: monitor
https://www.dynatrace.com/support/help/get-started/what-is-dynatrace

Dynatrace enables monitoring of your entire infrastructure including your hosts, processes, and network. You can perform log monitoring and view information such as the total traffic of your network, the CPU usage of your hosts, the response time of your processes, and more.

3. unit test　code coverage rate

We could use jacoco to test this rate
How to use:
https://blog.csdn.net/lvyuan1234/article/details/82836052

4. What's the difference between a blue/green deployment and a rolling deployment?

In Blue Green Deployment, you have TWO complete environments. One is Blue environment which is running and the Green environment to which you want to upgrade. Once you swap the environment from blue to green, the traffic is directed to your new green environment. You can delete or save your old blue environment for backup until the green environment is stable.

蓝绿部署是不停老版本，部署新版本然后进行测试，确认 OK，将流量切到新版本，然后老版本同时也升级到新版本。

In Rolling Deployment, you have only ONE complete environment. The code is deployed in the subset of instances of the same environment and moves to another subset after completion.

5. What are the differences between continuous integration, continuous delivery, and continuous deployment?

- Developers practicing continuous integration merge their changes back to the main branch as often as possible. By doing so, you avoid the integration hell that usually happens when people wait for release day to merge their changes into the release

branch.

- Continuous delivery is an extension of continuous integration to make sure that you can release new changes to your customers quickly in a sustainable way. This means that on top of having automated your testing, you also have automated your release process and you can deploy your application at any point of time by clicking on a button.
- Continuous deployment goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released to your customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production.

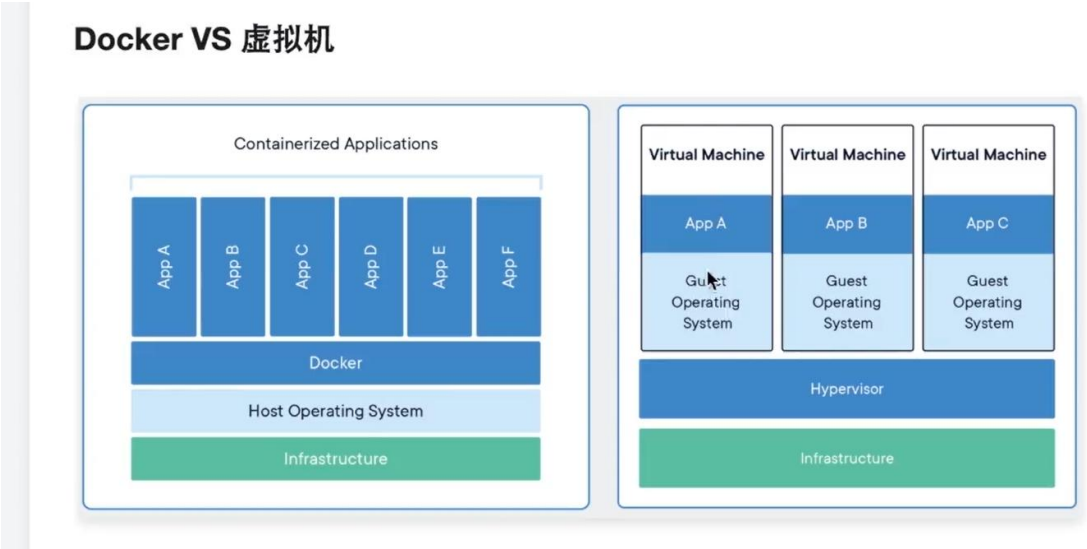
6. What is the difference between a Docker image and a container?

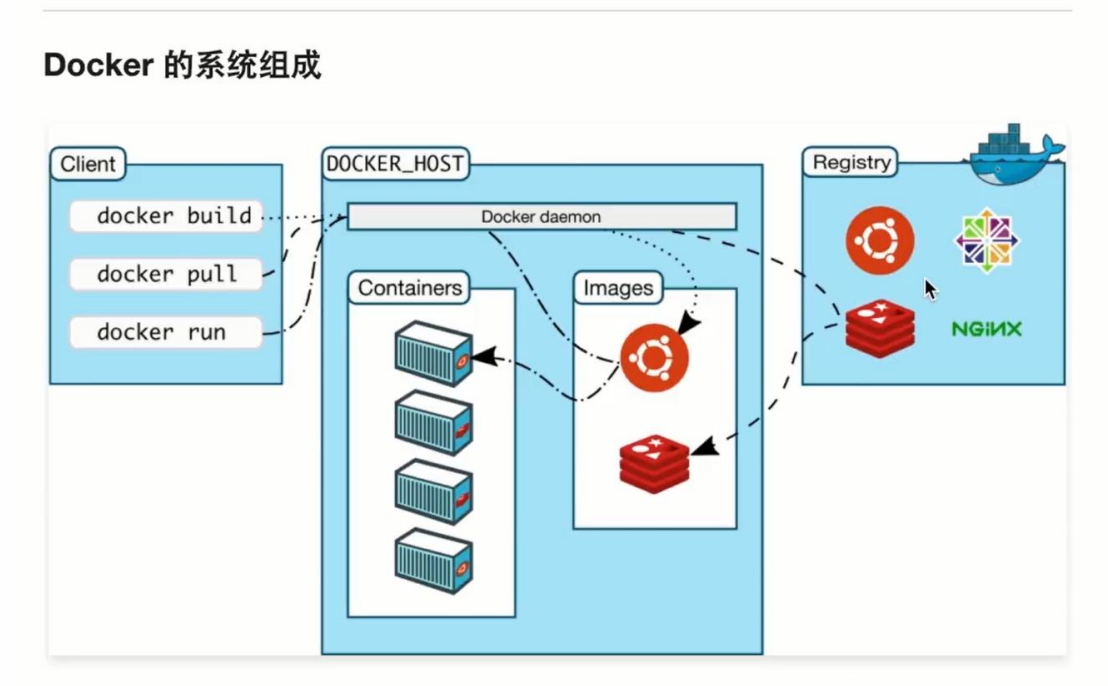
- An instance of an image is called a container. You have an image, which is a set of layers. If you start this image, you have a running container of this image. You can have many running containers of the same image.
- You can see all your images with docker images whereas you can see your running containers with docker ps (and you can see all containers with docker ps -a).
- **So a running instance of an image is a container.**

- **Image is immutable**

7. What is Docker

Package Software into standard units for Development, Shipment(Delivery 交付) and ,Deployment， which avoid the requirement for the operation system





8. Docker

8.1 What is docker:

It is a technology or platform to form:

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

8.2 What is k8s

Kubernetes is an open-source container orchestration platform. It eliminates the manual processes associated with deploying and scaling containerized applications. **Furthermore, this system allows you to manage and coordinate containerized applications on a cluster of machines through small units called pods.**

In Kubernetes, each container is isolated from other processes and runs on a computer, physical server, or virtual machine. Furthermore, containers are pretty lightweight, fast, and portable because, unlike a virtual machine, **containers do not need to include an operating system in each instance** and can instead

leverage the functionalities and resources of the host machine's operating system.

8.3 what is pod

The pod is one of the fundamentals of Kubernetes because it's the primary component that developers may manipulate. **It represents a set of processes running within a cluster node. A pod is, therefore, a basic execution unit of a Kubernetes application. It is the smallest and simplest object model. a pod within a node has:**

8.4 what is service

A Kubernetes Service is an abstraction layer which defines a logical set of Pods and enables external traffic exposure, load balancing and service discovery for those Pods.

8.5 container and pod

https://www.baeldung.com/ops/kubernetes-pod-vs-container

J. AWS

1. ec2 auto scaling type;ELB

auto scaling type:

- **Always keep the current instance level:** For example, always keep an ASG with a constant number of 3 healthy instances
- **Manual scaling**: Manually change the parameter's maximum capacity, minimum capacity, or required capacity to control the number of instances in ASG
- **Expand on schedule**: Expand the number of instances flexibly according to the specific time defined
- **Scaling according to demand**: **Use CloudWatch to scale based on parameters** (for example, automatically scale up when CPU utilization is above 70% for 10 minutes, i.e. increase the number of EC2 instances;When CPU utilization is below 30% for 10 minutes, it will automatically scale down, reducing the number of EC2 instances

2. Lambda serveless

Lambda is a compute service that lets you run code without provisioning or managing servers. Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, **including server and operating system maintenance,** capacity provisioning and automatic scaling, code monitoring and logging. With Lambda, you can run code for virtually any type of application or backend service.
When using Lambda, you are responsible only for your code. Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources to run your code.

3. Docker

- **Docker is a containerization platform which packages your application and all its dependencies together in the form of containers so as to ensure that your application works seamlessly in any environment be it development or test or production.**
- **Docker containers, wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries etc. anything that can be installed on a server.**
- **This guarantees that the software will always run the same, regardless of its environment.**

4. S3 type

- **Standard** - Default storage class: S3 allocates this type of storage if the object is not specified when uploaded
- **Reduced Redundancy** - Low Redundancy Storage (RRS) : Can be used to store renewable data because this type of file has the lowest durability.RRS prices are currently higher than Standard in most areas, so RRS will no longer be used in most cases, and this type of RRS will be phased out in the near future.
- **Standard -- IA** (inFrequently Accessed) : Used to hold data that is Infrequently Accessed, but can be Accessed quickly when it is needed.Storage is cheaper than standard S3, but reading is more expensive than standard S3, which is why infrequently accessed data is placed on this type of S3.And the data is across multiple AWS geographies.
- **Onezone -- IA**: Ditto, but only to one AWS available zone
- **Glacier**: Very cheap, only for archiving.Reading data from the Glacier takes three to five hours.

5. API Gateway

The Amazon API Gateway enables developers to create, publish, maintain, monitor, and secure APIs of any size.You can create APIs that can access AWS, other Web services, and data stored in the AWS cloud.

Usage:

- The API Gateway **can cache content so that some commonly used content can be delivered to users faster**

- The API Gateway is a low-cost, **serverless solution that scales automatically and flexibly (like ELB, NAT Gateway).**
- **The API Gateway can be throttled to prevent malicious attacks**
- **The API Gateway logs can be placed in CloudWatch**

6. S3 and EFS

https://www.bioaws.com/blogs/2020-01-26-s3-ebs-efs-compare/

三类存储对比表

特点	S3	EBS	EFS
存储成本（宁夏区域为例）	[标准型] 第一个 50TB/月部分: 0.1755 元/GB； 50TB-450TB 部分: 0.1719 元/GB 500TB 以上/月部分：0.1629 元/GB	gp2 通用型 SSD: 0.664 元/GB-月 io1 预配 IOPS SSD：存储 0.764 元/GB-月，0.399 元/预配置 IOPS-月 st1 吞吐优化 HDD：0.299 元/GB-月 sc1 cold HDD: 0.166 元/GB-月 EBS 磁性介质：0.364 元/GB-月，0.364/100 万 I/O 请求	标准存储：2.145 元/GB-月 不频繁访问存储：0.200 元/GB-月 不频繁访问请求：0.071 元（每 GB 传输数据） 预置吞吐量（MB/秒-月）：42.891 元(MB/S-月)
存储大小	存储对象数量无限制	最大为 16TB	无限制，自动弹性扩展
文件大小	单个 S3 对象最小 0 byte，最大 5TB	无限制	单个文件最大 47.9TiB
数据吞吐和 I/O	支持分段上传，建议大于 100MB 的文件都采用该上传方式 调用 PUT API 进行上传的单个文件限制为 5GB	基于 SSD 和 HDD 两类 如果对 I/O 有要求，推荐采用 SDD+预留 IOPS	所有连接的 client：默认 3GB/s，吞吐量和 IOPS 随着文件系统的扩大而扩展，可以在短时间内快速达到更高的吞吐量级别
性能	高扩展性，支持 3500 PUT/LIST/DELETE 请求/秒 支持 5500 GET 请求/秒	可手动扩展卷大小而不用停止实例 基准性能：3 IOPS/GB gp2 更高性能要求请选择 io1	高度可扩展的托管服务 支持最高 7000 次/秒的文件系统操作
数据	数据存储在区域内 在一个区域内的多个可用区备份 可以通过跨区域复制（CRR）达到跨区域级的容灾	数据存储在同一可用区 可以在多个可用区创建副本提高可用性	EFS 数据存储在区域内 在区域内的多可用区备份
数据访问	访问策略控制权限 可通过 Internet 访问	只能被一台 EC2 实例访问	能被 1-1000s 台 EC2 跨 AZ 的实例同时访问
文件系统限制	可以作为文件系统被 mount（s3fs 服务），但不推荐。	支持各类文件系统，包括 ext3 和 ext4	EC2 的文件存储服务 EFS 还能基于 AWS Direct Connect 为本地服务器提供 NFS 服务

特点	S3	EBS	EFS
	默认情况下桶的权限不会自动传递到桶内的文件夹。		
支持的加密机制	服务器端加密（SSE-Amazon S3, SSE-KMS, SSE-C） 客户端加密（CMK+客户端 master key）	AWS KMS 托管的 CMK 和 AES 256-bit 加密标准	AWS KMS 托管的 CMK 和 AES 256-bit 加密标准
访问控制	设置桶策略和用户策略 通过 ACLs（access control list）来管理访问 基于 IAM 角色的预签名 URL 访问	安全组 NACL 基于用户的授权（如 IAM）	安全组 基于用户的授权（如 IAM）
可用性	一年内 99.99%的可用性	99.99%可用性	高可用性
可用区容灾	能承受 2 个可用区挂掉	不能承受所在可用区挂掉（如果没有及时创建 EBS 镜像的话）	分布式存储 多个可用区 能承受 1 个可用区挂掉
持久性	99.999999999%	是普通硬盘的 20 倍	高持久性
最终一致性问题	S3 在所有区域提供 PUTS 和 DELETES 的最终一致性	无此问题	无此问题

7. available zone

Availability Zones are distinct locations within an AWS Region that are engineered to be isolated from failures in other Availability Zones. They provide inexpensive, low-latency network connectivity to other Availability Zones in the same AWS Region. Important. Each region is completely independent.

8. what will happen , if ec2 crashed

If one of the EC2 instances crashes, ELB stops to route requests to the crashed instances. The auto-scaling group replaces the crashed EC2 instance within minutes, and ELB begins to route requests to the new instance.

9. availability zone , VPC, region

10. what is ELB

Elastic Load Balancing (ELB) is a load-balancing service for Amazon Web Services (AWS) deployments. ELB automatically distributes incoming application traffic and scales resources to meet traffic demands. ELB helps an IT team adjust capacity according to incoming application and network traffic.

11 . cloud front

K. MultiThread

join priority thread pool ; object and class;transient;callable advantage;deadlock;When

there are multiple threads running, and one thread throws StackOverflowException, then what happens? Other still working

1. Differences of Class Lock and Object Lock and Method Lock

https://blog.csdn.net/zcg_741454897/article/details/108540717

1.1 Method Lock is the same as Object Lock

When a thread accesses a synchronized method, all other synchronized methods cannot be accessed by other threads because of the object lock

https://blog.csdn.net/C_AJing/article/details/106275720

1.2 Class Lock

Once a static method is declared synchronized, when the method is called. all instantiated objects of that class share the same lock, called class lock, .

2. atomic visibility reorder

原子性如何解决 atomic 读写安全

syschronized (monitorenter / monitorexit)

可见性如何解决 visibility 读安全 写不安全

volatile、、final

Final 修饰的引用数据类型，里面的属性可以被修改，但是不能直接对引用类型的变量进行修改，
https://www.cnblogs.com/mianlaoshu/articles/3648403.html

引用变量被 final 修饰,则读安全,写不安全

> final类型的成员变量的值，包括那些用final引用指向的collections的对象，是读线程安全而无需使用synchronization的
>
> 注意，如果你有一个指向collection，数组或其他可变对象的final引用，如果存在其他线程访问，仍然需要使用同步机制来访问该对象（或使用ConcurrentHashMap）。

基本类型 or immutable class 被 final 修饰，则读安全，写安全

因此，不可变对象（指所有的成员都是final并且成员要么是基本类型，要么指向另一个不可变对象）可以并发访问而无需使用同步机制。**通过final引用**读取“实际不可变”对象（指成员虽然实际并不是final，然而却从不会改变）也是安全的。然而，从程序设计的角度来看，在此种情况下强化不可变性是明智的（如用Collections.unmodifiableList()封装一个collection）。That way, you'll spot bugs introduced when one of your colleagues naughtily attempts to modify a collection that you didn't intend to be modified!

对象逃逸

```java
public class Demo {
    // final修饰的int类型的变量value
    private final int value;

    // 构造函数
    public Demo() {
        // Demo类的构造函数中使用了 this 引用并将其传递给了 OtherClass.doSomething() 方法。这个方法可能会在对象完全构
        // 由于 Java 中对象的构造过程是非常复杂的，对象的状态可能在构造函数中的不同阶段发生变化，因此在对象完全构造之前访问
        OtherClass.doSomething(this);
        value = 1;
    }
    public int getValue() {
        return value;
    }
}

class OtherClass {
    public static void doSomething(Demo demo) {
        // 使用 Demo 中的 value
        int value = demo.getValue();
        // ...
    }
}
```

常理上，用 final 修饰的变量具有内存可见性的特点，而且是不会变化的，可以保证他是不可变的。但是，仅仅是在是在一个不可变的对象在正确的创建的出来，才是线程安全的。

那什么时候，是不正确的创建方式呢？答案就是：在创建的过程中，有没有发生 this 的引用逃逸的情况。

3. threadsafe and non thread safe. examples

Or see thread-safe class 此例源自 Java 并发编程实战，非常重要， 怎样写出即安全又有效率的代码
https://www.cnblogs.com/blueSkyline/p/9096725.htm

4. Volatile

有序性如何解决 instruction reorder 读安全 写不安全

https://www.cnblogs.com/hollischuang/p/11386988.html

本文从两方面论述了 volatile 的重要性以及不可替代性：

一方面是因为 synchronized 是一种锁机制，存在阻塞问题和性能问题，而 volatile 并不是锁，所以不存在阻塞和性能问题。

另外一方面，因为 volatile 借助了内存屏障来帮助其解决可见性和有序性问题，而内存屏障的使用还为其带来了一个禁止指令重排的附件功能，所以在有些场景中是可以避免发生指令重排的问题的。

Singleton 必须被 volatile 修饰. new 的过程还是会被重排

https://www.51cto.com/article/709935.html

```java
package basic.knowledge.henry.design_Pattern._00DesignPattern_Sing

/**
 * 有双重检查的多线程使用,这是单利的标准写法
 * 支持多线程调用,线程安全
 */
public class Singleton {

    private static volatile Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

下面的可以不加，因为即使重排也不影响结果,但今后最好都加了

```java
package basic.knowledge.henry.ThreadRelevant._24InstanceLockAndClassLock.instanceLock;



//https://blog.csdn.net/zcg_741454897/article/details/108540717
public class ThreadSecurtiy implements Runnable{
    int ticket = 100;
    @Override
    public void run() { sellTicket();//设置线程任务是调用addSalary接口 }

    public synchronized void sellTicket(){
        while (ticket > 0){
            try {
                Thread.sleep( millis: 100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if(ticket > 0) ticket--;//当票数大于0的时候，就可以卖票
            System.out.println(Thread.currentThread().getName() + "正在卖" + ticket + "
        }
    }

    public static void main(String[] args) {
        ThreadSecurtiy ts = new ThreadSecurtiy();//创建一个ts对象
        for(int i = 0;i < 10;i++){//通过ts创建了10个线程，这10个线程对同一个对象ts的salary进行
            new Thread(ts).start();
        }
    }
}
```

5. join

https://zhuanlan.zhihu.com/p/99374488
Block other threads and let itself finish execution at first.

6. Priority

Just lower/higher the possibility of a thread to be executed

7. yield

Thread.yield() suspends the currently executing Thread object and executes another Thread

三者的区别:
sleep 抱着锁转为阻塞态 **blocked**，同时其他所有线程争抢 cpu 资源

yield 抱着锁转为就绪态 runnable，只有和调用 yield 的线程 priority 相同或更高的线程才有资格争抢 cpu 资源（调用 yield 的线程也加入争抢，所以会出现调用了 yield 的线程依旧继续执行的情况）

注意上面两个的区别

被其他线程调用 join 而被插队的线程必须等到 join 的线程执行完毕之后才会从阻塞态 blocked 转为就绪态 runnable（被插了队就不敢吭声了！）

wait 会放弃锁和 cpu，转为阻塞态 blocked，需由 notify 或 notifyAll 唤醒。

8. Thread States

- **New**: create a new Thread.
- **Runnable**: call the start()
- **Running**:Runnable thread get the CPU and execution starts
- **Blocked** :
    1. waiting blocked:a running thread calls wait() method

2. Synchronized blocked: When a running thread **try to acquire** the synchronization lock of an object, **if the synchronization lock is held by another thread.**
3. Other blocked: a running thread calls sleep() or join() methods

- **Dead**: The thread has finished **executing** or exited run() due to an exception.

9. what happens if a thread throw an exception

Other threads continue to run normally

10.What is dead lock

A deadlock is a situation where a thread is waiting for **an object lock** that **another thread** hold,a**nd this second thread is waiting for an object lock that the first thread holds. They both remain waiting for each other to release lock .**

进程 A 占有对象 1 的锁，进程 B 占有对象 2 的锁，进程 A 需要对象 2 的锁才能继续执行，所以进程 A 会等待进程 B 释放对象 2 的锁，而进程 B 需要对象 1 的锁才能继续执行，同样会等待进程 A 释放对象 1 的锁，由于这两个进程都不释放已占有的锁，所以导致他们进入无限等待中

```java
public class DeadLock {
    public static final String LOCK_1 = "lock1";
    public static final String LOCK_2 = "lock2";

    public static void main(String[] args) {
        Thread threadA = new Thread(() -> {
            try {
                while (true) {
                    synchronized (DeadLock.LOCK_1) {
                        System.out.println(Thread.currentThread().getName() + " 锁住 lock
                        Thread.sleep(1000);
                        synchronized (DeadLock.LOCK_2) {
                            System.out.println(Thread.currentThread().getName() + " 锁住
                        }
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        });

        Thread threadB = new Thread(() -> {
            try {
                while (true) {
                    synchronized (DeadLock.LOCK_2) {
                        System.out.println(Thread.currentThread().getName() + " 锁住 lock
                        Thread.sleep(1000);
                        synchronized (DeadLock.LOCK_1) {
                            System.out.println(Thread.currentThread().getName() + " 锁住
                        }
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        });

        threadA.start();
        threadB.start();
    }
}
```

ThreadA holds the object 1 lock, ThreadB holds the object 2 lock, ThreadA needs lock object 2 to continue, so the ThreadA will wait for the ThreadB to release the lock object 2 , and the ThreadB needs object 1 lock to continue, it has also to wait for the ThreadA to release the lock object 1, because of these two Threads are not release have possession of locks, Which leads them to wait indefinitely

So this concept has two threads and two locks.

11.How to avoid dead lock

https://blog.51cto.com/lovebetterworld/2859077

程互相等待对方先释放锁，所以就出现了死锁。

死锁是不应该在程序中出现的，在编写程序时应该尽量避免出现死锁。下面有几种常见的方式用来解决死锁问题：

1. 避免多次锁定。尽量避免同一个线程对多个 Lock 进行锁定。例如上面的死锁程序，主线程要对 A、B 两个对象的 Lock 进行锁定，副线程也要对 A、B 两个对象的 Lock 进行锁定，这就埋下了导致死锁的隐患。
2. 具有相同的加锁顺序。如果多个线程需要对多个 Lock 进行锁定，则应该保证它们以相同的顺序请求加锁。比如上面的死锁程序，主线程先对 A 对象的 Lock 加锁，再对 B 对象的 Lock 加锁；而副线程则先对 B 对象的 Lock 加锁，再对 A 对象的 Lock 加锁。这种加锁顺序很容易形成嵌套锁定，进而导致死锁。如果让主线程、副线程按照相同的顺序加锁，就可以避免这个问题。
3. 使用定时锁。程序在调用 acquire() 方法加锁时可指定 timeout 参数，该参数指定超过 timeout 秒后会自动释放对 Lock 的锁定，这样就可以解开死锁了。
4. 死锁检测。死锁检测是一种依靠算法机制来实现的死锁预防机制，它主要是针对那些不可能实现按序加锁，也不能使用定时锁的场景的。

12.What is blocking queue

If the queue is full, “put” **blocks until space becomes available**; if the queue is empty, **“take” blocks until an element is available.**
Blocking queue can simplify producer-consumer pattern.

13.Countdownlatch/CyclicBarrier/BlockingQueue

Countdownlatch:
It can coordinate multiple threads, appoints each thread do its **sub-task**, till every thread finish its own sub-task, then **main thread** combine subtasks into a final result.

```
2 usages
private volatile List<Long> resList = new CopyOnWriteArrayList<>();
1 usage  heng
private long calculate() {
    try {
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool( nThreads: 5);
        CountDownLatch countDownLatch = new CountDownLatch(5);
        fixedThreadPool.execute(new MyThread( start: 1,  end: 200000000, countDownLatch));
        fixedThreadPool.execute(new MyThread( start: 200000001,  end: 400000000, countDownLatch));
        fixedThreadPool.execute(new MyThread( start: 400000001,  end: 600000000, countDownLatch));
        fixedThreadPool.execute(new MyThread( start: 600000001,  end: 800000000, countDownLatch));
        fixedThreadPool.execute(new MyThread( start: 800000001,  end: 1000000000, countDownLatch));
        countDownLatch.await();

        long sum = 0;
        for (long res : resList) {
            sum = sum + res;
        }
        return sum ;
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return -1;
}
```

```java
private class MyThread implements Runnable {
    2 usages
    private int start;
    2 usages
    private int end;
    2 usages
    private CountDownLatch countDownLatch;
    3 usages
    private final Lock lock = new ReentrantLock();
    no usages
    private final Condition condition = lock.newCondition();

    5 usages  heng
    public MyThread(int start, int end, CountDownLatch countDownLatch) {
        this.start = start;
        this.end = end;
        this.countDownLatch = countDownLatch;
    }

    heng
    @Override
    public void run() {
        try {
            lock.lock();
                System.out.println(Thread.currentThread().getName() + ",  begins");
                resList.add(getPartSum(start, end));
                System.out.println(Thread.currentThread().getName() + ",  ends");
            lock.unlock();
        } finally {
            countDownLatch.countDown();
        }
    }

    1 usage  heng
    private long getPartSum(int start, int end) {
        long sum = 0;
        for (int i = start; i <= end; i++) {
            sum = sum + i;
        }
```

CyclicBarrier:

CyclicBarrier 和 CountDownLatch 相比，更为灵活，可以被重复使用(difference from CountdownLatch)。前者可以用来分段任务，假如有个任务需要分三个阶段来完成，每个阶段可以多线程并发执行，但是进入下一个阶段的时候，必须所有线程都完成了第一阶段的执行。那么通过 CyclicBarrier，在每个线程的每个阶段开始前都设置屏障点，可以很轻松地实现。

BlockingQueue:

14.readwriteLock

Operations on the **read lock use the shared** acquire and release methods; operations

on the **write lock use the exclusive** acquire and release methods.
Write lock degrade:
If a thread holds write lock, it can hold read lock without releasing that write lock;
Read lock upgrade:
If a thread holds read lock, it must release it before holding write lock.

15.Difference between sychronized and lock

Same: **both can keep code atomic and visible.**
Differences :
Or Extended capabilities include:

- The ability to have more than one condition variable per monitor. Monitors that use the synchronized keyword can only have one. This means reentrant locks support more than one wait()/notify() queue.
- **The ability to make the lock fair. Synchronized blocks are unfair.**
  "[fair] locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order."

- **The ability to check if the lock is being held.**
- The ability to get the list of threads waiting on the lock.

The disadvantages of reentrant locks are:

- **Need to add import statement.**
- **Need to wrap/release lock acquisitions in a try/finally block. This makes it more ugly than the synchronized keyword.**
- The synchronized keyword can be put in method definitions which avoids the need for a block which reduces nesting.

16.when to use and not use concurrenthashmap

ConcurrentHashMap:

The `ConcurrentHashMap` allows concurrent access to its entries without the need for explicit synchronization. Multiple threads can safely perform read and write operations without causing data corruption.

Atomic Operations:

Operations like `get`, `put`, and `remove` are atomic in `ConcurrentHashMap`. This means that when one thread performs a `put` operation, it will not interfere with another thread performing a `get` operation on a different key.

Efficiency:

`ConcurrentHashMap` is optimized for high concurrency, allowing multiple threads to read and write data simultaneously. It uses a lock-free, non-blocking algorithm for read operations, and a fine-grained locking mechanism for write operations, minimizing lock contention and improving performance.

When to Use ConcurrentHashMap:

Simple, Independent Operations: When your operations are simple and independent (like individual inserts, updates, or removals), and each operation needs to be thread-safe.

High Concurrency: When your application requires a high level of concurrency with minimal blocking. `ConcurrentHashMap` is designed for situations where many threads are expected to access the map concurrently.

Avoiding Complexity: When you want to avoid the complexity and overhead of managing explicit locks (`synchronized` or `ReentrantLock`), which might be unnecessary for basic operations.

Example of Why Not to Use ConcurrentHashMap

If you had more complex operations, such as needing to update multiple related entries in the map as a single atomic operation, or if you needed to perform compound actions that involved checking a condition and then modifying the map, you might still need explicit locking. For example:

java
复制代码
```
public void putIfAbsentAndLog(String key, String value) {
    synchronized(this) {
        if (!cache.containsKey(key)) {
            cache.put(key, value);
            System.out.println("Added key: " + key);
        }
    }
}
```

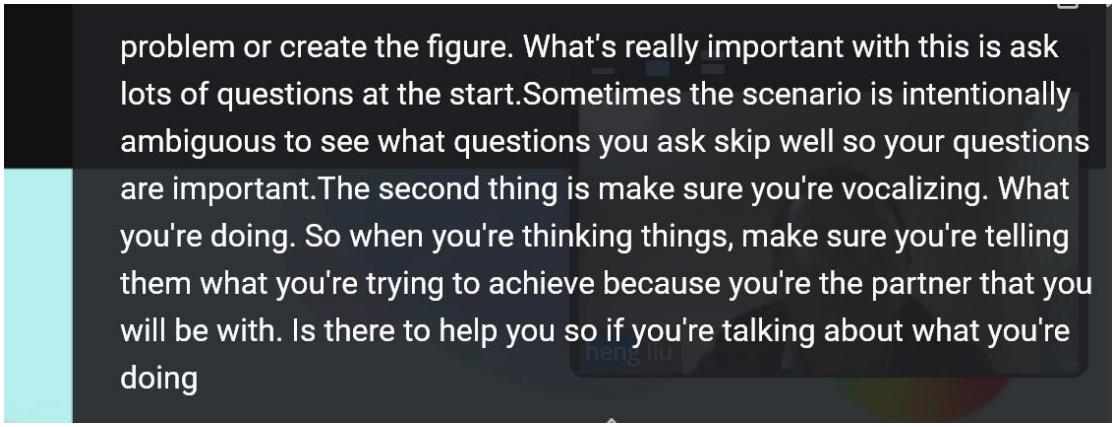
In this case, `ConcurrentHashMap` alone might not be sufficient because the `containsKey` check and `put` operation together need to be atomic. For such compound actions, you might need additional synchronization, making a `ConcurrentHashMap` less advantageous.

Conclusion:

Use `ConcurrentHashMap` when you need a simple, efficient way to handle concurrent access to a map where each operation is independent and atomic. It simplifies your code by eliminating the need for explicit synchronization while providing excellent performance in multi-threaded environments.

17.What does re-entrant mean(sychronized)

Java synchronized keyword is **re-entrant** in nature， it means if a synchronized method calls another synchronized method **which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.**



18.CAS

There are 3 parameters for a CAS operation:

- A memory location V where value has to be replaced
- Old value A which was read by thread last time
- New value B which should be written over V

CAS says "I think V should have the value A; if it does, put B there, otherwise don't change it but tell me I was wrong." CAS **is an optimistic technique—**it proceeds with the update in the hope of success, and can detect failure if another thread has updated

the variable since it was last examined.

19.Future(callable)

效果跟 countdownlatch 一样

```java
public static void main(String[] args) {
    FutureTaskForMultiCompute inst=new FutureTaskForMultiCompute();
    // 创建任务集合
    List<FutureTask<Integer>> taskList = new ArrayList<~>();
    // 创建线程池
    ExecutorService exec = Executors.newFixedThreadPool( nThreads: 8);
    for (int i = 0; i < 10; i++) {
        // 传入Callable对象创建FutureTask对象,生产十个子线程去计算0+99,Integer为结果类型
        FutureTask<Integer> ft = new FutureTask<Integer>(inst.new ComputeTask(i,  taskName: ""+i));
        taskList.add(ft);
        // 提交给线程池执行任务，也可以通过exec.invokeAll(taskList)一次性提交所有任务;
        //虽然5个线程run,但call只执行一次
        exec.submit(ft);
    }
    //异步执行其他业务逻辑
    System.out.println("所有计算任务提交完毕，主线程接着干其他事情！");

    // 开始统计各计算线程计算结果
    Integer totalResult = 0;
    for (FutureTask<Integer> ft : taskList) {
        try {
            //FutureTask的get方法会自动阻塞,直到获取计算结果为止
            totalResult = totalResult + ft.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }

    // 关闭线程池
    exec.shutdown();
    System.out.println("多任务计算后的总结果是:" + totalResult);
```

```java
    }

    private class ComputeTask implements Callable<Integer> {

        private Integer result = 0;
        private String taskName = "";

        public ComputeTask(Integer iniResult, String taskName){
            result = iniResult;
            this.taskName = taskName;
            System.out.println("生成子线程计算任务: "+taskName);
        }

        public String getTaskName() { return this.taskName; }

        @Override
        public Integer call() throws Exception {
            // TODO Auto-generated method stub

            for (int i = 0; i < 100; i++) {
                result =+ i;
            }
            // 休眠5秒钟，观察主线程行为，预期的结果是主线程会继续执行，到要取得FutureTask的结果是等待直至完成。
            Thread.sleep( millis: 5000);
            System.out.println("子线程计算任务: "+taskName+" 执行完成!,result为:"+result);
            return result;
        }
    }
}
```

20.Threadpool

https://www.zhihu.com/question/41134816

21.#####thread safe Class#####

https://www.cnblogs.com/blueSkyline/p/9096725.html

22.what is threadlocal

https://www.imooc.com/read/49/article/953

23.Volatile 和 final

Final:

> 也必须正确的赋值了。
>
> **final的对象引用**
>
> 对象的final成员成员的值在当退出构造函数时，他们也是最新的。这意味着：
>
> > final类型的成员变量的值，包括那些用final引用指向的collections的对象，是读线程安全而无需使用synchronization的
>
> 注意，如果你有一个指向collection，数组或其他可变对象的final引用，如果存在其他线程访问，仍然需要使用同步机制来访问该对象（或使用ConcurrentHashMap）。
>
> 因此，不可变对象（指所有的成员都是final并且成员要么是基本类型，要么指向另一个不可变对象）可以并发访问而无需使用同步机制。**通过final引用**读取“实际不可变”对象（指成员虽然实际并不是final，然而却从不会改变）也是安全的。然而，从程序设计的角度来看，在此种情况下强化不可变性是明智的（如用Collections.unmodifiableList()封装一个collection）。That way, you'll spot bugs introduced when one of your colleagues naughtily attempts to modify a collection that you didn't intend to be modified!
>
> **使用final的限制条件和局限性**

24.methods of creating and Start Thread

20.1 extends Thread

20.2 implements Runnable

20.3 inmplements callable : which can get a result by return sth

```java
import java.util.concurrent.Callable;

public class CallableDemo implements Callable<Integer> {

    private int sum;
    @Override
    public Integer call() throws Exception {
        System.out.println("Callable子线程开始计算啦！");
        Thread.sleep( millis: 2000);

        for(int i=0 ;i<5000;i++){
            sum=sum+i;
        }
        System.out.println("Callable子线程计算结束！");
        return sum;
    }

```

```java

public class CallableTest {
    public static void main(String[] args) {
        //创建线程池
        ExecutorService es = Executors.newSingleThreadExecutor();
        //创建Callable对象任务
        CallableDemo calTask=new CallableDemo();
        //创建FutureTask
        FutureTask<Integer> futureTask=new FutureTask<>(calTask);
        //执行任务
        es.submit(futureTask);
        //关闭线程池
        es.shutdown();
        try {
            Thread.sleep( millis: 2000);
            System.out.println("主线程在执行其他任务");

            if(futureTask.get()!=null){
                //输出获取到的结果
                System.out.println("futureTask.get()-->"+futureTask.get());
            }else{
                //输出获取到的结果
                System.out.println("futureTask.get()未获取到结果");
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("主线程在执行完成");
    }

}

```

```java
public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(Runtime.getRuntime().ava
    List<FutureTask<Integer>> futureTaskList = new ArrayList<~>();
    for(int i = 0;i < 10 ;i ++){
        final int index = i;
        FutureTask<Integer> ft = new FutureTask<~>(new Callable<Integer>() {
            private  Integer result = 0;
            @Override
            public Integer call() throws Exception {
                for(int j = 0;j < 100;j++){
                    result += j;
                }
                Thread.sleep( millis: 5000);
                System.out.println("子线程: " + index + ",执行完成！");
                return result;
            }
        });
        futureTaskList.add(ft);
        executorService.submit(ft);
    }
```

```java
        futureTaskList.add(ft);
        executorService.submit(ft);
    }

    System.out.println("子线程提交完毕，主线程继续执行！");

    try {
        Thread.sleep( millis: 1000 * 10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("主线程执行完毕！");

    Integer totalResult = 0;
    for(FutureTask<Integer> ft : futureTaskList){
        try {
            totalResult += ft.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
    System.out.println("子线程计算的结果为: " + totalResult);
}
```

25.多线程操作中为什么使用 while 而不是 if 来做判断状态是否就绪

https://www.cnblogs.com/limingluzhu/p/4325784.html

原因：

在线程中notify或者notifyAll会唤醒一个或多个线程，当线程被唤醒后，被唤醒的线程继续执行阻塞后的操作。

这里分析一下get操纵： 当某个线程得到锁时storage为空，此时它应该wait，下次被唤醒时（任意线程调用notify），storage可能还是空的。因为有可能其他线程清空了storage。如果此时用的是if它将不再判断storage是否为空，直接继续，这样就引起了错误。但如果用while则每次被唤醒时都会先检查storage是否为空再继续，这样才是正确的操作；生产也是同一个道理。

26.对象发布逸出

https://blog.csdn.net/a19881029/article/details/8165188

27. 如何用一个互斥锁实现，读写锁?

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class SimpleReadWriteLock {
    private int readers = 0;   // Number of active readers
    private boolean writing = false;   // Indicates if there's an active writer

    private final Lock lock = new ReentrantLock();
    private final Condition canRead = lock.newCondition();
    private final Condition canWrite = lock.newCondition();

    // Acquire the read lock
    public void lockRead() throws InterruptedException {
        lock.lock();
        try {
            // Wait while a writer is writing
            while (writing) {
                canRead.await();
            }
            readers++;
        } finally {
            lock.unlock();
        }
    }

    // Release the read lock
```

```
    public void unlockRead() {
        lock.lock();
        try {
            readers--;
            // If no more readers, signal writers
            if (readers == 0) {
                canWrite.signal();
            }
        } finally {
            lock.unlock();
        }
    }

    // Acquire the write lock
    public void lockWrite() throws InterruptedException {
        lock.lock();
        try {
            // Wait while there are active readers or a writer is writing
            while (readers > 0 || writing) {
                canWrite.await();
            }
            writing = true;
        } finally {
            lock.unlock();
        }
    }

    // Release the write lock
    public void unlockWrite() {
        lock.lock();
        try {
            writing = false;
            // Signal all readers and writers
            canRead.signalAll();
            canWrite.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

28.A typical thread safe code

https://www.cnblogs.com/zwh0910/p/14440248.html
线程安全的容器，特别是 Map，很多情况下一个业务中 涉及容器的操作有多个（读 get 写 put,remove），即复合操作，而在并发执行时，线程安全的容器只能保证自身的数据不被破 坏，和数据在多个线程间是可见的，但无法保证业务的行为是否正确，即 ConcurrentHashMap 多线程操作不能保证数据同步。

下面的例子结果不是 6000,count 无法被多个线程同时做写操作

```
public class ConcurrentHashMapDemo2 {
    public static void main(String[] args) {
        final Map<String, Integer> count = new ConcurrentHashMap<>();
//        final Map<String, Integer> count = new HashMap<>();
//        final Map<String, Integer> count = new Hashtable<>();
        count.put("count",0);
        Runnable task = new Runnable(){
            @Override
            public void run() {
                int value;
                for (int i = 0; i < 2000; i++) {
                    value = count.get("count");
                    count.put("count",value + 1);
                }
            }
        };
        new Thread(task).start();
        new Thread(task).start();
        new Thread(task).start();
        try {
            Thread.sleep(1000l);
            System.out.println(count);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

29. junit 测试多线程安全性

```java
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;

2 usages  new *
public class ThreadSafeCounter {
    3 usages
    private final ConcurrentHashMap<String, AtomicInteger> counters = new ConcurrentHashMap<>();

    // Increment the counter for a specific key
    1 usage  new *
    public void increment(String key) {
        counters.computeIfAbsent(key, k -> new AtomicInteger( initialValue: 0)).incrementAndGet();

    }

    // Decrement the counter for a specific key
    1 usage  new *
    public void decrement(String key) {
        counters.computeIfAbsent(key, k -> new AtomicInteger( initialValue: 0)).decrementAndGet();

    }

    // Get the current value of the counter for a specific key
    2 usages  new *
    public int getCount(String key) {
        return counters.getOrDefault(key, new AtomicInteger( initialValue: 0)).get();
    }
}
```

```java
    no usages   new *
    @Test
    public void testThreadSafety() throws InterruptedException {
        int numThreads = 100;
        int numIncrementsPerThread = 1000;
        ExecutorService executor = Executors.newFixedThreadPool(numThreads);
        CountDownLatch latch = new CountDownLatch(numThreads);

        Runnable task = () -> {
            try {
                for (int i = 0; i < numIncrementsPerThread; i++) {
                    counter.increment( key: "task1");
                    counter.decrement( key: "task2");
                }
            } finally {
                latch.countDown();
            }
        };

        // Execute tasks concurrently
        for (int i = 0; i < numThreads; i++) {
            executor.execute(task);
        }

        // Wait for all threads to complete
        latch.await();

        // Validate final counts
        assertEquals( expected: numThreads * numIncrementsPerThread, counter.getCount( key: "task1"));
        assertEquals( expected: -numThreads * numIncrementsPerThread, counter.getCount( key: "task2"));

        // Shutdown the executor
        executor.shutdown();
    }
}
```

L.Spring and spring boot

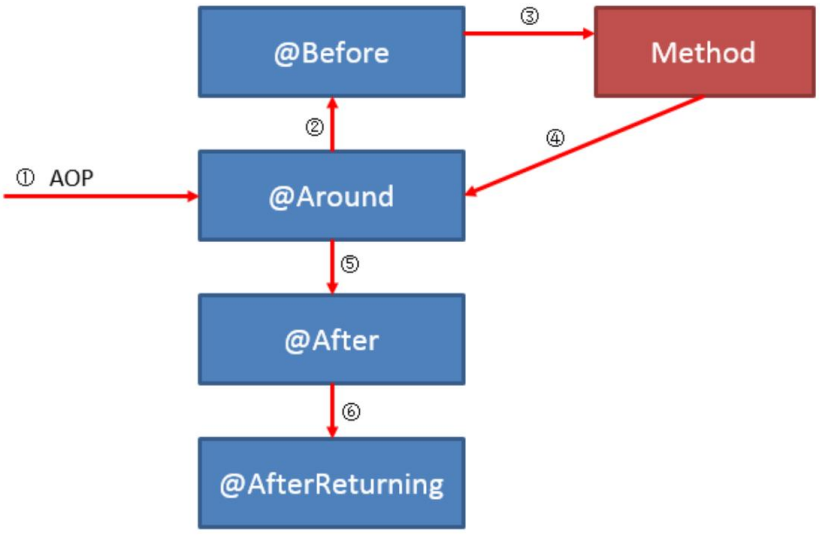
1. log level

Seriousness : Debug < info< warn < error< fatal

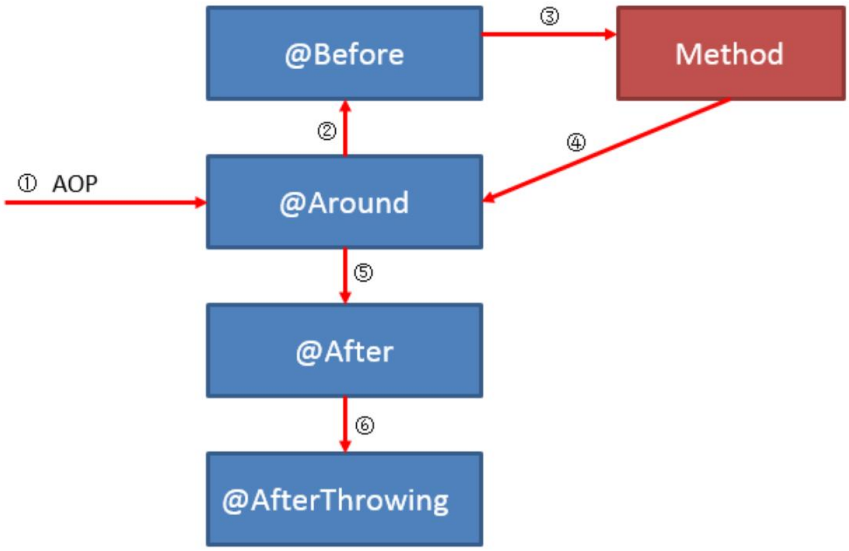
2. AOP

怎样实现:
https://www.cnblogs.com/tuyang1129/p/12878549.html

Two key words: pointCuts and types of advice
Execution sequence
Normal:



Exception throw:



3. ACTUATOR

Spring Boot Actuator provides secured endpoints for **monitoring and managing your Spring Boot application**. By default, all actuators endpoints are secured.

4. SCHEDULER @EnableScheduling

Step 1:Add @EnableScheduling annotation to your spring boot application class.
Step2:

```
@Scheduled(cron = "0 10 10 10 * ?")
public void run() {
    logger.info("Current time is :: " + Calendar.getInstance().getTime());
}
```

5. spring boot advantages or different from spring

- **Simplify coding**

For example, if you want to create a Web project, you will need to add **multiple dependencies** to the POM file when you use Spring. **Spring Boot will help you to quickly start a Web container**. In Spring Boot, you need to create a Web project.**All we need to do is add the following Starter-Web** dependency to the POM file.

- **Simplify the configuration** @Configuration @Bean

Spring, while making Java EE a lightweight framework, was once known as "configuration hell" because of its cumbersome configuration.The variety o**f XML and Annotation** configurations can be confusing, and with so many configurations, it's hard to figure out why something goes wrong.Spring Boot takes more of a Java Config approach.

- **Simplify deployment**

With Spring, the projec**t deployment requires us to deploy Tomcat** on the server and then drop the project into Tomcat **as a WAR package**.
With Spring Boot, we don't need to deploy Tomcat on the serve, **Because Spring Boot has Tomcat embedded**,We jus**t need to package project to JAR** and use a command java -jar XXX. Jar to launch the project .

- **Simplify the monitoring**

We can introduce Spring-Boot-Start-Actuation dependency and directly <u>use REST to obtain the runtime performance parameters of the process,</u> so as to achieve the purpose of monitoring. It is more convenient.

6. What are starter dependencies?

Spring Boot starters are maven templates **that contain a collection of all the relevant transitive dependencies that are needed to start a particular functionality.**

For example, If we want to create a Spring WebMVC application then in a traditional setup, we would have included all required dependencies ourselves. It leaves the chances of version conflict which ultimately result in more runtime exceptions.

With Spring boot, to create web MVC application, all we need to import is **spring-boot-starter-web** dependency. Transitively, it brings in all other required dependencies to build a web application e.g. spring-webmvc, spring-web, hibernate-validator, tomcat-embed-core, tomcat-embed-el, tomcat-embed-websocket, jackson-databind, jackson-datatype-jdk8, jackson-datatype-jsr310 and jackson-module-parameter-names.

7. Spring boot common annotations?

The most commonly used and important spring boot annotations are as below:

- @EnableAutoConfiguration – It enable auto-configuration mechanism.
- @ComponentScan – enable component scanning in application classpath.
- @SpringBootApplication – enable all 3 above three things in one step i.e. enable auto-configuration mechanism, enable component scanning and register extra beans in the context.
- @ImportAutoConfiguration comapred to @EnableAutoConfiguration
- – imports and apply only the specified auto-configuration classes. We should use this when we don't want to enable the default auto-configuration.

As you probably know, @EnableAutoConfiguration attemps to configure beans that are located on your classpath eg tomcat-embedded.jar.
Whereas @ImportAutoConfiguration only runs the configuration classes that you provided in the annotation.

- @AutoConfigureBefore, @AutoConfigureAfter, @AutoConfigureOrder – shall be used if the configuration needs to be applied in a specific order (before of after).
- @Conditional – annotations such as @ConditionalOnBean, @ConditionalOnWebApplication or @ConditionalOnClass allow to register a bean only when the condition meets.

8. Spring Security

How to check username password

9. Swagger

10.how to share a common module and how to setup in parent pom

```
<build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
          <classifier>exec</classifier>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
-                   <mainClass>PremiumbreakdownServiceApplication
-                   </mainClass>
-                   <mainClass>com.suncorp.premiumbreakdown.PremiumBreakdownWebApplication
+                   <mainClass>com.suncorp.premiumbreakdown.PremiumbreakdownServiceApplication
                    </mainClass>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
```

Show more

```
                <configuration>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                    <encoding>${project.build.sourceEncoding}</encoding>
                </configuration>
            </plugin>
        </plugins>
```

11.generate object according to table name

https://www.jb51.net/article/214588.htm

12.Spring 事件解耦

https://blog.csdn.net/qq_34874784/article/details/125540407

13. spring transaction 事务失效

When will roll back, which exception will trigger a roll back
https://z.itpub.net/article/detail/18A4D9564A61EC7AF8EAA66FCA251444

1 Private 必须 public 修饰

2 final or static

3 method called within a class

有时候我们需要在某个 Service 类的某个方法中，调用另外一个事务方法，比如：

```
@Service
public class UserService {

    @Autowired
    private UserMapper userMapper;

    @Transactional
    public void add(UserModel userModel) {
        userMapper.insertUser(userModel);
        updateStatus(userModel);
    }

    @Transactional
    public void updateStatus(UserModel userModel) {
        doSameThing();
    }
}
```

我们看到在事务方法 add 中，直接调用事务方法 updateStatus。从前面介绍的内容可以知道，updateStatus 方法拥有事务的能力是因为 spring aop 生成代理了对象，但是这种方法直接调用了 this 对象的方法，所以 updateStatus 方法不会生成事务。

由此可见，在同一个类中的方法直接内部调用，会导致事务失效。

Solutions:
1 add a new serviceB
这个方法非常简单，只需要新加一个 Service 方法，把@Transactional 注解加到新 Service 方法上，把需要事务执行的代码移到新方法中。具体代码如下：

```java
@Servcie
public class ServiceA {
   @Autowired
   prvate ServiceB serviceB;

   public void save(User user) {
         queryData1();
         queryData2();
         serviceB.doSave(user);
   }
}

@Servcie
public class ServiceB {

   @Transactional(rollbackFor=Exception.class)
   public void doSave(User user) {
      addData1();
      updateData2();
   }
}
```

2 injection itself

如果不想再新加一个 Service 类，在该 Service 类中注入自己也是一种选择。具体代码如下：

```java
@Servcie
public class ServiceA {
   @Autowired
   prvate ServiceA serviceA;

   public void save(User user) {
         queryData1();
         queryData2();
         serviceA.doSave(user);
   }

   @Transactional(rollbackFor=Exception.class)
   public void doSave(User user) {
      addData1();
      updateData2();
   }
}
```

```
@Servcie
public class ServiceA {
   @Autowired
   prvate ServiceA serviceA;

   public void save(User user) {
         queryData1();
         queryData2();
         serviceA.doSave(user);
   }

   @Transactional(rollbackFor=Exception.class)
   public void doSave(User user) {
       addData1();
       updateData2();
    }
 }
```

4.未被 spring 管理(Not Managed By Spring)

5 多线程调用

这样会导致两个方法不在同一个线程中，获取到的数据库连接不一样，从而是两个不同的事务。如果想doOtherThing方法中抛了异常，add方法也回滚是不可能的。

如果看过spring事务源码的朋友，可能会知道spring的事务是通过数据库连接来实现的。当前线程中保存了一个map，key是数据源，value是数据库连接。

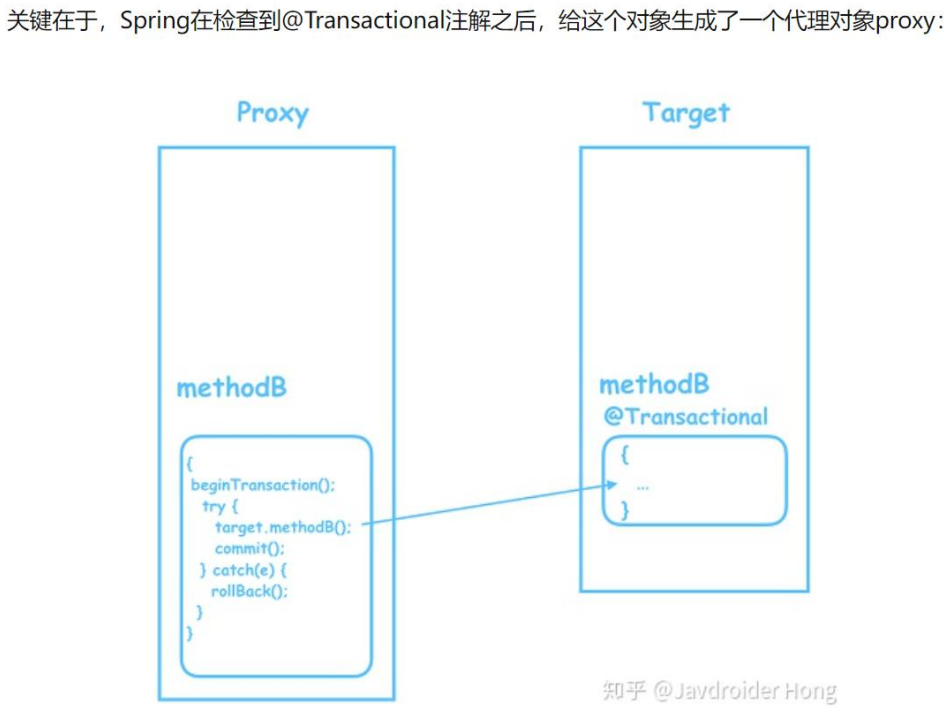
```
private static final ThreadLocal<Map<Object, Object>> resources =

  new NamedThreadLocal<>("Transactional resources");
```

我们说的同一个事务，其实是指同一个数据库连接，只有拥有同一个数据库连接才能同时提交和回滚。如果在不同的线程，拿到的数据库连接肯定是不一样的，所以是不同的事务。

6 表不支持, DB doesnot support transaction

14.Spring 事务不回滚

Spring 事务底层也是用的 spring AOP



1 设置了错误的事务传导机制

https://www.baeldung.com/spring-transactional-propagation-isolation

REQUIRED is the default propagation. Spring checks if there is an active transaction, and if nothing exists, it creates a new one. Otherwise, the business logic appends to the currently active transaction:

For ***SUPPORTS***, Spring first checks if an active transaction exists. If a transaction exists, then the existing transaction will be used. If there isn't a transaction, it is executed non-transactional:

When the propagation is ***MANDATORY***, if there is an active transaction, then it will be used. If there isn't an active transaction, then Spring throws an exception:

For transactional logic with NEVER propagation, Spring throws an exception if there's an active transaction:

NOT_SUPPORT: If a current transaction exists, first Spring suspends it, and then the business logic is executed without a transaction:

When the propagation is ***REQUIRES_NEW***, Spring suspends the current transaction if it exists, and then creates a new one:

2 try catch 了 自己吞了异常 ,没有 throw

```
// @Service
public class OrderServiceImpl implements OrderService {

    @Transactional
    public void updateOrder(Order order) {
        try {
            // update order
        } catch {

        }
    }

}
```

注意 不 try catch，会抛出 runtimexception 默认注解可以回滚

3 try catch 了自己的异常然后又 throw 了 eXCEPTION which is not runtime exception

```
@Slf4j
@Service
public class UserService {

    @Transactional
    public void add(UserModel userModel) throws Exception {
        try {
             saveData(userModel);
             updateData(userModel);
        } catch (Exception e) {
            log.error(e.getMessage(), e);
            throw new Exception(e);
        }
    }
}
```

上面的这种情况，开发人员自己捕获了异常，又手动抛出了异常：Exception，事务同样不会回滚。

因为spring事务，默认情况下只会回滚 RuntimeException （运行时异常）和 Error （错误），对于普通的 Exception（非运行时异常），它不会回滚。

如果想抛 exception 异常,必须加 add 属性 attribute rollBackFor = Exception.class

4 自定义回滚异常，但是实际上抛出了其他异常
Self-defined on exception , however another exception was thrown

但如果这个参数的值设置错了，就会引出一些奇奇怪怪的问题，例如：

```
@Slf4j
@Service
public class UserService {

    @Transactional(rollbackFor = BusinessException.class)
    public void add(UserModel userModel) throws Exception {
        saveData(userModel);
        updateData(userModel);
    }
}
```

如果在执行上面这段代码，保存和更新数据时，程序报错了，抛了SqlException、DuplicateKeyException等异常。而BusinessException是我们自定义的异常，报错的异常不属于BusinessException，所以事务也不会回滚。

> 即使rollbackFor有默认值，但阿里巴巴开发者规范中，还是要求开发者重新指定该参数。
>
> 这是为什么呢？
>
> 因为如果使用默认值，一旦程序抛出了Exception，事务不会回滚，这会出现很大的bug。所以，建议一般情况下，将该参数设置成：Exception或Throwable。

`@Transactional` 的rollbackFor用于指定能够触发事务回滚的异常类型，可以指定多个，用逗号分隔。
`rollbackFor` 默认值为UncheckedException，包括了RuntimeException和Error.
当我们直接使用 `@Transactional` 不指定 `rollbackFor` 时，Exception及其子类都不会触发回滚。

所以一般要指定 rollbackOn = Exception.class 扩大回滚异常的范围

```
// 回滚
@Transactional(rollbackOn = Exception.class)
public void test21() throws Exception {
    User user = new User(1, "15000000000", "d243ewa", "Comma");
    saveUser(user);
    throw new BusinessException("test21 error");
}
```

@Transactional注解有两个，一个是spring提供的 `@org.springframework.transaction.annotation.Transactional` 另一个是jdk提供 `@javax.transaction.Transactional` 。在回滚的设置上，spring提供的是 `rollbackFor` ，jdk提供的是 `rollbackOn` ，在使用方法上是一致的。
这两个注解在我有限的测试里表现是一样的。

回滚测试:
https://www.cnblogs.com/use-D/p/12571839.html

> A类的service A方法调用B类的service
>
> ```
> ServiceA {
>  void methodA() {
>  ServiceB.methodB();
>  }
> }
> ServiceB {
>  void methodB() {
>  }
> }
> ```

场景2

A使用默认事务注解：@Transactional(rollbackFor = Exception.class)
B使用新事务注解：@Transactional(propagation = Propagation.REQUIRES_NEW,rollbackFor = Exception.class)
测试条件　结果
B发生异常，A未catch　A不回滚，B回滚
B发生异常，Acatch了异常，记录日志，未抛出异常　A不回滚，B回滚
B发生异常，Acatch了异常，记录日志，抛出异常　A回滚，B回滚
B执行成功后，A发生了异常　A回滚，B不回滚

结论：A发生异常，只有A回滚，B不回滚；B发生异常，只有B回滚，A不回滚

即：AB属于两个独立的事务，互不影响。

15.Spring 事务隔离级别

测试结果: mysql
https://www.cnblogs.com/dennyzhangdd/p/9602670.html

https://blog.csdn.net/weixin_44259720/article/details/112826960

16.事务的嵌套

https://blog.51cto.com/u_3631118/3119745

嵌套和传播机制
https://www.cnblogs.com/use-D/p/12571839.html

17.Spring DI 的方式有哪些

https://blog.csdn.net/Weixiaohuai/article/details/120981207
三种注入方式:

变量注入（Field Injection）

```java
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    private UserMapper userMapper;

    //...

}
```

构造器注入（Constructor Injection）

```java
@Service
public class UserServiceImpl implements UserService {

    private final UserMapper userMapper;

    @Autowired
    public UserServiceImpl(UserMapper userMapper) {
        this.userMapper = userMapper;
    }

}
```

setter 方法注入 （Setter Injection）

```java
@Service
public class UserServiceImpl implements UserService {

    private UserMapper userMapper;

    @Autowired
    public void setUserMapper(UserMapper userMapper) {
        this.userMapper = userMapper;
    }

}
```

18.Spring bean 如何创建

https://www.cnblogs.com/lgjlife/p/11060570.html

方式1：

使用Spring XML方式配置，该方式用于在纯Spring 应用中，适用于简单的小应用，当应用变得复杂，将会导致XMI配置文件膨胀，不利于对象管理。

```
<bean id="xxxx"  class="xxxx.xxxx"/>
```

方式2：

使用@Component,@Service,@Controler,@Repository注解

这几个注解都是同样的功能，被注解的类将会被Spring 容器创建单例对象。

@Component : 侧重于通用的Bean类

@Service：标识该类用于业务逻辑

@Controler：标识该类为Spring MVC的控制器类

@Repository: 标识该类是一个实体类，只有属性和Setter,Getter

```
@Component
public class User{
}
```

当用于Spring Boot应用时，被注解的类必须在启动类的根路径或者子路径下，否则不会生效。

如果不在，可以使用@ComponentScan标注扫描的路径。

spring xml 也有相关的标签<component-scan />

```
@ComponentScan(value={"com.microblog.blog","com.microblog.common"})
public class MicroblogBlogApplication {
    public static void main(String args[]){
        SpringApplication.run(MicroblogBlogApplication.class,args);
    }
}
```

方式3：

使用@Bean注解,这种方式用在Spring Boot 应用中。

@Configuration 标识这是一个Spring Boot 配置类，其将会扫描该类中是否存在@Bean 注解的方法，比如如下代码，将会创建User对象并放入容器中。

@ConditionalOnBean 用于判断存在某个Bean时才会创建User Bean.

这里创建的Bean名称默认为方法的名称user。也可以@Bean("xxxx")定义。

```
@Configuration
public class UserConfiguration{

      @Bean
      @ConditionalOnBean(Location.class)
      public User user(){
           return new User();
      }

}
```

19.Spring bean 生命周期

https://segmentfault.com/a/1190000040365130

https://blog.csdn.net/liangjiabao5555/article/details/116420821?csdn_share_tail=%7B%22type%22%3A%22blog%22%2C%22rType%22%3A%22article%22%2C%22rId%22%3A%22116420821%22%2C%22source%22%3A%22unlogin%22%7D

5、bean的生命周期?

bean的生命周期，大致可以分为以下 5 步，当然最核心的就是前 3 步。

1）bean的实例化

2）bean的属性赋值

3）bean的初始化

4）bean的使用

5）bean的销毁

instantiation
Population
initialization
Usage of bean
Destroy of bean

20.Spring bean scope

https://www.cnblogs.com/javastack/p/13271944.html

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of an HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.
global session	Scopes a single bean definition to the lifecycle of a global HTTP `Session`. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring `ApplicationContext`.

> **Singleton**

这也是Spring默认的scope，表示Spring容器只创建一个bean的实例，Spring在创建第一次后会缓存起来，之后不再创建，就是设计模式中的单例模式。

> **Prototype**

代表线程每次调用这个bean都新创建一个实例。

> **Request**

表示每个request作用域内的请求只创建一个实例。

> **Session**

表示每个session作用域内的请求只创建一个实例。

> **GlobalSession**

这个只在porlet的web应用程序中才有意义，它映射到porlet的global范围的session，如果普通的web应用
器会把它作为普通的session作用域的scope创建。

0
推荐

21.IOC and DI

DI is the process of providing the dependencies ; and IoC is the end result of DI. ...the responsibility of creating objects is shifted from our application code to the Spring container; this phenomenon is called IoC. Dependency Injection can be done by setter injection or constructor injection or field injection(@Autowired).

22.bean 的懒加载和提前加载

https://www.cnblogs.com/yanze/p/10243348.html

23.Spring 怎样避免循环依赖

三级缓存
https://mp.weixin.qq.com/s?__biz=MzkwNjMwMTgzMQ==&mid=2247490271&idx=1&sn=e4476b631c48882392bd4cd06d579ae9&source=41#wechat_redirect

https://zhuanlan.zhihu.com/p/84267654

24.spring MVC

https://zhuanlan.zhihu.com/p/39607030

25.拦截器 过滤器

https://segmentfault.com/a/1190000037755221

M. WEB

1. Get post difference

Get is retrieving data, post is creating data, put is editing data
Get :Parameters are kept in the browser history. Not safe; Post: Parameters are not kept in the browser history. Safe
Get:length of URL limited; POST: unlimited

2. SOAP Web service

SOAP stands for **Simple Object Access Protocol.** It is a **XML-based protocol** for accessing web services.

SOAP is XML based protocol. **It is platform independent and language independent**. **By using SOAP, you will be able to interact with other programming language applications.**

用 XML 交流 你给我 XML 我给你 XML

3. how does two processes communicate

- pipeline

Process 1 copies the data from user space to the kernel buffer, and process 2 reads the data from the kernel buffer. This mechanism provided by the kernel is called interprocess communication.
Advantage:
Easy
Disadvantage:
Pipeline communication is inefficient and not suitable for frequent data exchange between processes.

- MQ

One is the communication is not timely, the other is the attachment also has the size limit

4. Http Code

500 Internal Server Error
400 Bad Request, parameter issue
401 Unauthorized
503 Service Unavailable
504 Gateway Timeout
200 OK
403 Forbidden
204 No Content

5. n-tier architecture

presentation tier
Service tier
Data access tier

Client tier(frontend)represents Web browser, a Java or other application, Applet, WAP phone etc. The client tier makes requests to the Web server who will be serving the request by either returning static content if it is present in the Web server or forwards the request to either Servlet or JSP in the application server for either static or dynamic content.
Presentation tier (mvc) encapsulates the presentation logic required to serve clients. A Servlet or JSP in the presentation tier intercepts client requests, manages logons, sessions, accesses the business services, and finally constructs a response, which gets delivered to client.
Business tier (service)provides the business services. This tier contains the business logic and the business data. All the business logic is centralized into this tier as opposed to 2-tier systems where the business logic is scattered between the front end and the backend. The benefit of having a centralized business tier is that same business logic can support different types of clients like browser, WAP (Wireless Application Protocol) client, other standalone applications written in Java, C++, C# etc.
Integration tier (**dao)**is responsible for communicating with external resources such as databases, legacy systems, ERP systems, messaging systems like MQSeries etc. The components in this tier use JDBC, JMS, J2EE Connector Architecture (JCA) and some proprietary middleware to access the resource tier.
Resource tier (**database**)is the external resource such as a database, ERP system, Mainframe system etc responsible for storing the data. This tier is also known as Data Tier or EIS (Enterprise Information System) Tier.

6. RPC Vs REST :Understanding RPC Vs REST For HTTP APIs

https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/

RPC-based HTTP and REST-based HTTP
https://www.geeksforgeeks.org/difference-between-rest-api-and-rpc-api/

REST 是一种软件架构风格。这种风格的典型应用，就是 HTTP。其因为简单、扩展性强的特点而广受开发者的青睐。

而 RPC 呢，是 Remote Procedure Call Protocol 的简写，中文描述是远程过程调用，它可以实现客户端像调用本地服务(方法)一样调用服务器的服务(方法)。

1	REST is resource-oriented.	RPC is action-oriented.
2	Supports HTTP methods GET, POST, PUT, PATCH, and DELETE.	RPC only supports GET and POST requests.
3	It provides flexibility for hardware architecture	no flexibility in RPC for hardware architecture
4	It supports hypermedia and hyperlinks	It does not supports hypermedia and hyperlinks

RPC 和REST 两者的定位不同，REST 面向资源，更注重接口的规范，因为要保证通用性更强，所以对外最好通过REST。 而RPC 面向方法，主要用于函数方法的调用，可以适合更复杂通信需求的场景。 RESTful API客户端与服务端之间采用的是同步机制，当发送HTTP请求时，客户端需要等待服务端的响应。 2019年12月10日

HTTP 接口和 RPC 接口的应用场景

HTTP 接口适用于 Web 应用程序和浏览器之间的通信。它通常用于传输 HTML、CSS、JavaScript 和其他 Web 资源，以及 RESTful 风格的 API 服务。

RPC 接口适用于分布式系统之间的通信。它可以在多种编程语言之间进行通信，支持多种协议和数据格式。RPC 接口通常用于处理高并发、高吞吐量的场景，例如大型的分布式计算、大数据处理等。

7. Restful 优势

Restful api cantains no resource status. In restful api we just use noun instead of a verb. It is very convenient

8. Security vulnerabilities

https://blog.csdn.net/weixin_44185213/article/details/119823714?spm=1001.2101.3001.6661.1&utm_medium=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-119823714-blog-94592341.235%5Ev38%5Epc_relevant_sort_base3&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-119823714-blog-94592341.235%5Ev38%5Epc_relevant_sort_base3&utm_relevant_index=1

8.1 What is cross-site scripting (XSS) attacks

8.2 Use encryption (e.g., HTTPS) to secure communication between the client and server, protecting sensitive information and preventing eavesdropping.

9. websocket and http(long polling)

https://www.jianshu.com/p/db3ca83bf7ce

WebSocket is an event-driven protocol(proactively pushing data to clients , if there is an event), which means you can actually use it for truly realtime communication. Unlike HTTP, where you have to constantly request updates, with websockets, updates are sent immediately when they are available.

https://www.geeksforgeeks.org/what-is-web-socket-and-how-it-is-different-from-the-http/

Differences between HTTP and WebSocket Connection:

WebSocket Connection	HTTP Connection

WebSocket is a bidirectional communication protocol that can send the data from the client to the server or from the server to the client by reusing the established connection channel. The connection is kept alive until terminated by either the client or the server.	The HTTP protocol is a unidirectional protocol that works on top of TCP protocol which is a connection-oriented transport layer protocol, we can create the connection by using HTTP request methods after getting the response HTTP connection get closed.
Almost all the real-time applications like (trading, monitoring, notification) services use WebSocket to receive the data on a single communication channel.	Simple RESTful application uses HTTP protocol which is stateless.
All the frequently updated applications used WebSocket because it is faster than HTTP Connection.	When we do not want to retain a connection for a particular amount of time or reuse the connection for transmitting data; An HTTP connection is slower than WebSockets.

https://ably.com/blog/websockets-vs-long-polling

Blog: Home Company Engineering Developers Serverless WebSockets

Long polling disadvantages

- Long polling is more resource intensive on the server than a WebSocket connection.
- Long polling can come with a latency overhead because it requires several hops between servers and devices. Gateways often have different ideas of how long a typical connection is allowed to stay open, so sometimes close while processing is still underway.
- Reliable message ordering can be an issue with long polling because it is possible for multiple HTTP requests from the same client to be in flight simultaneously. For example, if a client has two browser ta open consuming the same server resource, and the client-side application is persisting data to a local store such as `localStorage` or `IndexedDb`, there is no in-built guarantee that duplicate data won't be written more than once.

WebSocket advantages

- WebSockets keeps a unique connection open while eliminating the latency problems that arise with long polling.
- Full-duplex asynchronous messaging is supported so that both the client and the server can stream messages to each other independently.
- WebSockets generally do not use `XMLHttpRequest`, and as such, headers are not sent on each server requ This, in turn, reduces the size of data payloads.
- WebSockets pass through most firewalls without any reconfiguration and have an origin-based security model.

Hi, do

transfer.

When not to use WebSocket: WebSocket can be used if we want any real-time updated or continuous streams of data that are being transmitted over the network. If we want to fetch old data, or want to get the data only once to process it with an application we should go with **HTTP protocol**, old data which is not required very frequently or fetched only once can be queried by the simple HTTP request, so in this scenario, it's better not use WebSocket.

Note: RESTful web services are sufficient to get the data from the server if we are loading the data only once.

10.Cookie session and token

https://cloud.tencent.com/developer/article/1683290

11.difference between http and https

HTTP (Hypertext Transfer Protocol):

- **Unsecured:** HTTP stands for "Hypertext Transfer Protocol," and it is used to transmit data between a web browser (client) and a web server. However, the data transmitted using HTTP is not encrypted or secured in any way.
- **Data Integrity:** Since the data is not encrypted, it can be intercepted by attackers, potentially leading to data breaches, unauthorized access, and other security risks.
- **Port:** HTTP typically uses port 80 for communication.

HTTPS (Hypertext Transfer Protocol Secure):

- **Secured:** HTTPS stands for "Hypertext Transfer Protocol Secure." It operates similarly to HTTP but adds an extra layer of security through encryption.
- **Encryption:** With HTTPS, data exchanged between the browser and the server is encrypted using technologies like SSL (Secure Sockets Layer) or its successor, TLS (Transport Layer Security). This encryption makes it much more difficult for attackers to intercept and read the data being transmitted.
- **Data Integrity:** The encryption ensures data integrity, meaning that the data sent and received remains intact and unchanged during transit.
- **Authentication:** HTTPS also involves a level of authentication, as the server provides a digital certificate that confirms its identity. This helps prevent "man-in-the-middle" attacks, where attackers pose as legitimate servers.
- **Port:** HTTPS typically uses port 443 for communication.
- **Green Padlock:** Browsers often indicate a secure connection with a green padlock icon and "https://" in the address bar.

12. tcp and udp

https://www.freecodecamp.org/chinese/news/tcp-vs-udp-which-is-faster/
https://blog.51cto.com/iteyer/3239821

TCP (Transmission Control Protocol):

- **Connection-Oriented:** TCP is a connection-oriented protocol, which means that it establishes a reliable and ordered communication channel between two devices before data transfer begins.
- **Reliable Delivery:** TCP ensures that data packets are delivered reliably and in the correct order. If a packet is lost or arrives out of order, TCP handles retransmission and reordering to ensure data integrity.

UDP (User Datagram Protocol):

- **Connectionless:** UDP is a connectionless protocol, meaning it does not establish a dedicated connection before data transfer. Each UDP packet is treated independently.
- **Unreliable Delivery:** Unlike TCP, UDP does not guarantee reliable delivery of packets. **There is no mechanism for acknowledgment, retransmission, or order maintenance.**

13.Tcp three times shake hands and why is tcp secure

https://www.geeksforgeeks.org/tcp-3-way-handshake-process/

数据完整性： TCP 协议通过校验和等机制保证数据的完整性。数据在传输过程中，如果在传输过程中发生数据的丢失、损坏或篡改，TCP 会通过重传、错误检测和校正来保证数据的完整性。

顺序性： TCP 保证数据按照发送顺序进行传输。这有助于确保数据在接收端能够正确还原发送端的数据流。

流量控制和拥塞控制： TCP 的流量控制和拥塞控制机制可以减轻网络拥塞，防止过多的数据注入网络，从而提高网络的稳定性和可靠性。

N. DATABASE

1. Sql T Y P E S

DDL

DDL（data definition language）数据库定义语言：其实就是我们在创建表的时候用到的一些sql，比如说：CREATE、ALTER、DROP等。DDL主要是用在定义或改变表的结构，数据类型，表之间的链接和约束等初始化工作上。

```
CREATE TABLE 表名称
(
列名称1 数据类型,
列名称2 数据类型,
列名称3 数据类型,
```

DQL

DQL（data query language）数据查询语言：就是我们最经常用到的 SELECT。 主要用来对数据库的数据进行一些查询操作。

DML

DML（data manipulation language）数据操纵语言：就是我们最经常用到的UPDATE、 INSERT、DELETE。 主要用来对数据库的数据进行一些增，删，改操作。

2. Sql tuning,improvement

https://www.zhihu.com/column/p/27879034

1 avoid scaaning the whole table ,instead,create indexes
2 avoid judging null values for a field, in the where clause which might make index ineffective. instead, set the field a default value ,like 0.
3 avoid use != < > , not equal, less tha, more than , symnolds in the where clause, other wise the engine will bypass the index and scan the whole table.
4 avoid using or as the concat the condition in where clause, instead use union,

select id from t where num=10 or num=20 可以这样查询： select id from t where num=10 union all select id from t where num=20

5 % percent symbol shouldn't be ahead of the field when using like to search by a text field.
7 avoid manipulate a field in where clause, wher num/2 = 100, will make index inefffective or scan the whole table
8 field sequance must be the same as index sequance, when using a compound indexes like (name,age) name is ahead of age. you jave to use name equals and age equals or just name equals in the where clause.
9 too many indexs will affect the performance for insert or update operarions DML
10

3. 3NF in table design

http://www.gitta.info/LogicModelin/en/html/DataConsiten_Norm1NF.html

1: A relation is in first **normal form** if every attribute in every row can **contain only one single (atomic) value.**

Students

FirstName	LastName	Knowledge
Thomas	Mueller	Java, C++, PHP
Ursula	Meier	PHP, Java
Igor	Mueller	C++, Java

Startsituation

Result after Normalisation

Students

FirstName	LastName	Knowledge
Thomas	Mueller	C++
Thomas	Mueller	PHP
Thomas	Mueller	Java
Ursula	Meier	Java
Ursula	Meier	PHP
Igor	Mueller	Java
Igor	Mueller	C++

2:A relation is in second normal form if it is in 1NF and **every non key attribute is <u>fully</u> functionally dependent on the primary key. 注意这个 fully**

This means that each non-key attribute must be dependent on the entire primary key, not just a part of it.

一是表必须有一个主键；二是没有包含在主键中的列必须完全依赖于主键，而不能只依赖于主键的一部分。

The following functional dependencies exist:

1. The attribute ProfessorName is functionally dependent on attribute IDProf (IDProf --> ProfessorName)

2. The attribute StudentName is functionally dependent on IDSt (IDSt --> StudentName)

3. The attribute Grade is fully functional dependent on IDSt and IDProf (IDSt, IDProf --> Grade)

Students

IDSt	LastName	IDProf	Prof	Grade
1	Mueller	3	Schmid	5
2	Meier	2	Borner	4
3	Tobler	1	Bernasconi	6

Startsituation

Result after normalisation

Students

ID	LastName
1	Mueller
2	Meier
3	Tobler

Professors

IDProf	Professor
1	Bernasconi
2	Borner
3	Schmid

Grades

IDStIDProf	Grade	
1	3	5
2	2	4
3	1	6

3: A table is in 3NF if it is in 2NF and all the attributes are functionally dependent only on the primary key. **There should be no transitive dependency, which means that non-key attributes should not depend on other non-key attributes.**

Third normal form (3NF)

Third normal form:
A relation is in third normal form if it is in 2NF and no non key attribute is transitively dependent on the primary key.
A bank uses the following relation:

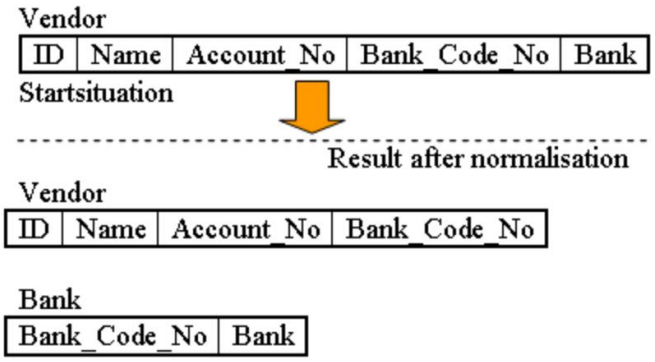
Vendor(ID, Name, Account_No, Bank_Code_No, Bank)

The attribute ID is the identification key. All attributes are single valued (1NF). The table is also in 2NF.

The following dependencies exist:

1. Name, Account_No, Bank_Code_No are functionally dependent on ID (ID --> Name, Account_No, Bank_Code_No)

2. Bank is functionally dependent on Bank_Code_No (Bank_Code_No --> Bank)



Example Third normal form

The table in this example is in 1NF and in 2NF. But there is a transitive dependency between Bank_Code_No and Bank, because Bank_Code_No is not the primary key of this relation. To get to the third normal form (3NF), we have to put the bank name in a separate table together with the clearing number to identify it.

第三范式（3NF）

在1NF基础上，任何非主属性不依赖于其它非主属性[在2NF基础上消除传递依赖]。

第三范式（3NF）是第二范式（2NF）的一个子集，即满足第三范式（3NF）必须满足第二范式（2NF）。

首先是 2NF，另外非主键列必须**直接依赖**于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。 考虑一个订单表【Order】（OrderID，OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity）主键是（OrderID）。

其中 OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity 等非主键列都完全依赖于主键（OrderID），所以符合 2NF。不过问题是 CustomerName，CustomerAddr，CustomerCity 直接依赖的是 CustomerID（非主键列），而不是直接依赖于主键，它是通过传递才依赖于主键，所以不符合 3NF。

通过拆分【Order】为【Order】（OrderID，OrderDate，CustomerID）和【Customer】（CustomerID，CustomerName，CustomerAddr，CustomerCity）从而达到 3NF。

differences between 2NF and 3NF

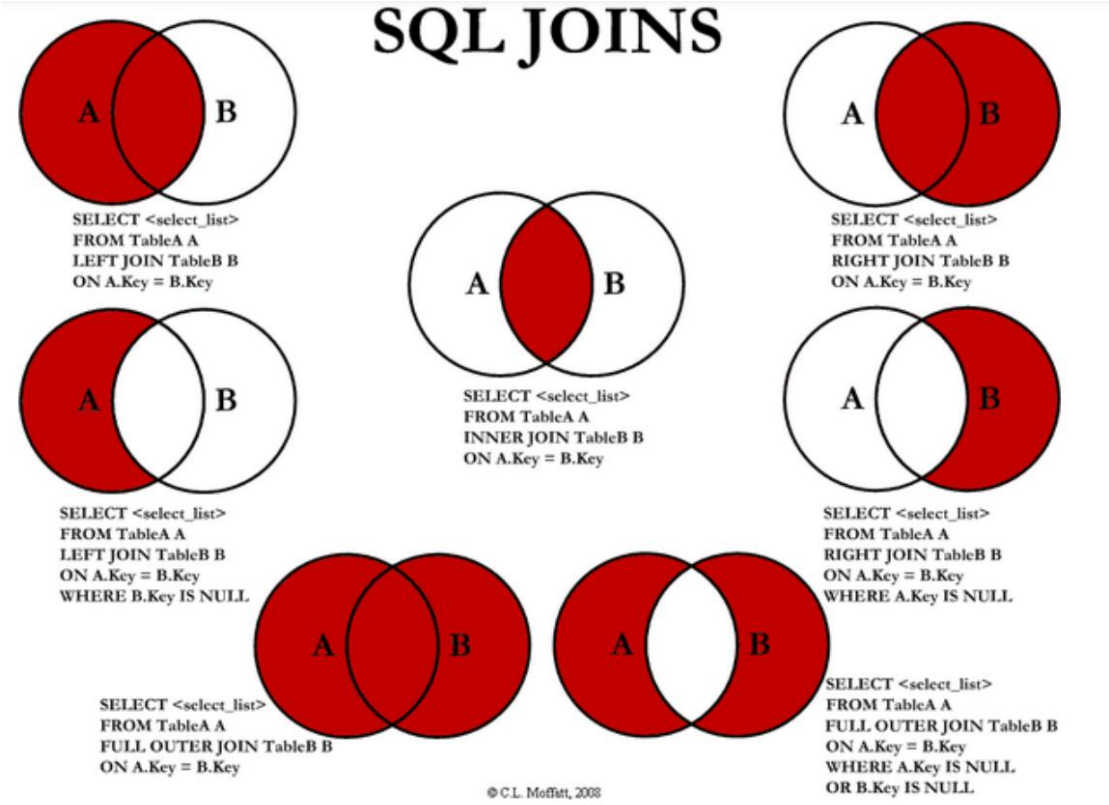
- **2NF** focuses on eliminating partial dependencies (dependencies on part of a composite key).

· **3NF** focuses on eliminating transitive dependencies (dependencies of non-key attributes on other non-key attributes).

All these NFS can be disobeyed for correlated subquery and convenient for adding indexes

4. Join

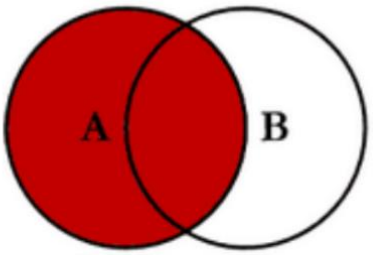
https://stackoverflow.com/questions/5706437/whats-the-difference-between-inner-join-left-join-right-join-and-full-join



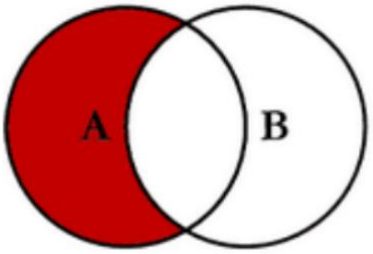
There are different types of joins available in SQL:

INNER JOIN: **returns rows when there is a match in both tables.**

LEFT JOIN: **returns all rows from the left table, even if there are no matches in the right table.**




```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

RIGHT JOIN: **returns all rows from the right table, even if there are no matches in the left table.**

FULL JOIN: **It combines the results of both left and right outer joins.**

The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

SELF JOIN: is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

CARTESIAN JOIN: returns the Cartesian product of the sets of records from the two or more joined tables.

5. DB triggers

https://docs.oracle.com/database/121/TDDDG/tdddg_triggers.htm#TDDDG52100

6. syntax

https://zhuanlan.zhihu.com/p/32390852
Case when:

```
CASE WHEN <求值表达式> THEN <表达式>
     WHEN <求值表达式> THEN <表达式>
     WHEN <求值表达式> THEN <表达式>
     .....
     ELSE <表达式>
 END
```

7. ACID

7.1 Isolation

https://www.51cto.com/article/611908.html

differnet levels of atomicity can prevent the following issues

为了帮助及加深理解，我们采用熟悉的"葫芦娃"作为例子。最先藤上有七个葫芦，每当有一个葫芦娃诞生时，老爷爷就会将他的信息录入数据库表中，如下：

ID	名称	能力
1	大娃	力大无穷
2	二娃	顺风耳千里眼
3	三娃	铜墙铁壁

- dirty reads

(insert)Dirty read: Read uncommitted(from another transaction) data in your own transaction

某一天，四娃诞生了。事务A(爷爷)访问了数据库，目的是往数据库里添加新诞生葫芦娃的信息，但是还没有提交事务。

```
insert into T values (4, '四娃','喷水');
```

这时，来了另一个事务B(蛇精)，她进行了查询操作，想查询所有已诞生的葫芦娃信息

```
select * from T;  //结果中出现了四娃的信息
```

这时，如果事务之间没有有效隔离，那么 蛇精 查询数据库返回的结果中就会出现 四娃 的信息，这就是脏读。

(1) 脏读：指在自己的事务中读到了别人未提交的数据

- unrepeatable reads

(update)Unrepeatable read: the system reads data twice in its own transaction , but query data is different

第二天，事务A(蛇精)一大早就查询了数据库中关于四娃的信息

```
select * from T  where ID = 4; //名称：四娃 能力：喷水
```

这时，事务B(爷爷)来了，因为爷爷发现四娃其实是会喷火而不是喷水，所以更新一下将改了四娃的能力为喷火，然后提交了事务。

```
update T set Ability= '喷火' where ID = 4;
```

接着，蛇精(事务A)还想再查看一次用于确认四娃的信息，于是又执行了

```
select * from T  where ID = 4;//名称：四娃 能力：喷火
```

这时候她惊讶地发现两次读出来四娃的信息，能力竟然不相同!这就是不可重复读。

(2) 不可重复读：指在自己的事务中读取两次，前后的数据不一样

- phantom reads

(inset and query a list)It is about query a list , the reason is similar to that of unrepeatable reads

第三天

事务A(蛇精)访问了数据库，她想要看看到底已经诞生的葫芦娃有哪些，于是她执行了

```
select * from T;   //查出一共有四条葫芦娃信息
```

复制

这时候因为五娃诞生了，所以事务B(爷爷)打开了数据库并将五娃的信息录入

```
insert into T values (5, '五','喷水');
```

复制

这时候，事务A(蛇精)想要再查询一次所有已诞生的葫芦娃信息进行确认，于是又执行了查询

```
select * from T;   //查出一共有五条葫芦娃信息
```

复制

这个时候事务A(蛇精)可能就会蒙了，以为自己产生了幻觉。这种情况就叫“幻读。

InnoDB offers all four transaction isolation levels described by the SQL:1992 standard:

◆ READ UNCOMMITTED, all three above issues will happen

Read uncommitted means that uncommitted content can be read. Because queries at this isolation level are not locked, "dirty reads", "unrepeatable reads", and "phantom reads" can occur. This isolation level is rarely used in real development without special circumstances.

◆ READ COMMITTED, avoiding dirty read

you can only read the content that has already been submitted. This is the most commonly used isolation level and **the default level for Oracle and SQLServer** to effectively avoid dirty reads.

◆ REPEATABLE READ, avoiding unrepeatable reads

Repeatable reads. This level effectively avoids "unrepeatable reads" and is the **default level for** MySQL database InnoDB.

● SERIALIZABLE. avoid dirty reads, unrepeatable reads, and phantom reads

Serialization is the highest isolation level of a database. At this level, transactions are serialized one by one in order to avoid dirty reads, unrepeatable reads, and phantom reads

. High security results in low execution efficiency and maximum performance overhead, so it is seldom used in actual development.

7.2 Atomicity and consistency

https://www.zhihu.com/question/30272728

原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability)！

mysql 怎样实现隔离性 持久性 原子性

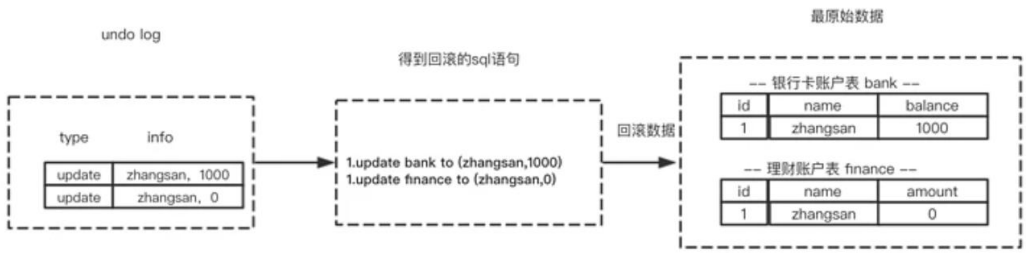
https://segmentfault.com/a/1190000039139809

Undo log　for Atomicity

根据上面流程可以得出如下结论：

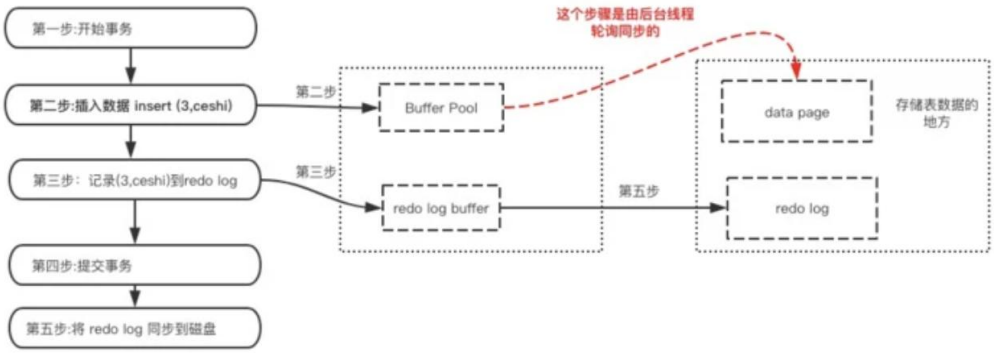
1. 每条数据变更(insert/update/delete)操作都伴随一条undo log的生成,并且回滚日志必须先于数据持久化到磁盘上
2. 所谓的回滚就是根据回滚日志做逆向操作，比如delete的逆向操作为insert，insert的逆向操作为delete，update的逆向为update等。

回滚过程如图



Redo log for　Durability

于是 redo log就派上用场了。下面有下redo log是什么时候产生的



既然redo log也需要存储，也涉及磁盘IO为啥还用它?

（1）刷脏是随机IO，因为每次修改的数据位置随机，但写redo log是追加操作，属于顺序IO。

（2）刷脏是以数据页（Page）为单位的，MySQL默认页大小是16KB，一个Page上一个小修改都要整页写入；而redo log中只包含真正需要写入的部分，无效IO大大减少。

8. Mysql primary/secondary replication

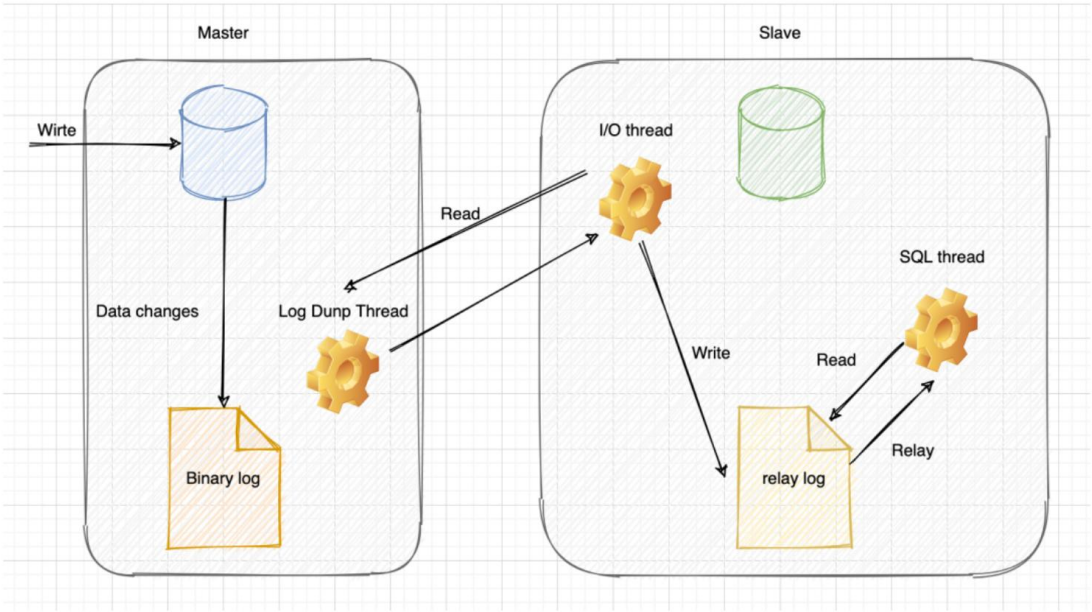
https://ost.51cto.com/posts/11721
https://www.cnblogs.com/rickiyang/p/13856388.html

过程

工作过程

MySQL 的主从复制工作过程大致如下：

1. 从库生成两个线程，一个 I/O 线程，一个 SQL 线程;
2. I/O 线程去请求主库的 binlog，并将得到的 binlog 日志写到 relay log(中继日志) 文件中;
3. 主库会生成一个 log dump 线程，用来给从库 I/O 线程传 binlog;
4. SQL 线程会读取 relay log 文件中的日志，并解析成具体操作，来实现主从的操作一致，而最终数据一致;



工作过程

请求流程

MySQL 建立请求的主从的详细流程如下：

1. 当从服务器连接主服务器时，主服务器会创建一个 log dump 线程，用于发送 binlog 的内容。在读取 binlog 的内容的操作中，会对象主节点上的 binlog 加锁，当读取完成并发送给从服务器后解锁。
2. 当从节点上执行 start slave 命令之后，从节点会创建一个 IO 线程用来连接主节点，请求主库中更新 binlog。IO 线程接收主节点 binlog dump 进程发来的更新之后，保存到 relay-log 中。

作用

主从复制带来了很多好处，当我们的主服务器出现问题，可以切换到从服务器;可以进行数据库层面的读写分离;可以在从数据库上进行日常备份。还可以保证：

1. 数据更安全：做了数据冗余，不会因为单台服务器的宕机而丢失数据;
2. 性能大大提升：一主多从，不同用户从不同数据库读取，性能提升;
3. 扩展性更优：流量增大时，可以方便的增加从服务器，不影响系统使用;
4. 负载均衡：一主多从相当于分担了主机任务，做了负载均衡。

应用场景MySQL 主从复制集群功能使得 MySQL 数据库支持大规模高并发读写成为可能，同时有效地保护了物理服务器宕机场景的数据备份。

横向扩展

将工作负载分发到各 Slave 节点上，从而提高系统性能。

在这个场景下，所有的写(write)和更新(update)操作都在 Master 节点上完成;所有的读(read)操作都在 Slave 节点上完成。通过增加更多的 Slave 节点，便能提高系统的读取速度。

8.1 when is primary/secondary replication necessary

当我们线上应用用户量小的时候，所有的读与写操作都在一台服务器上，这时候还不会遇到什么问题。 当用户量逐渐增加，访问数据库的请求也越来越多，这时候就给 MySQL 服务

器增加了负担，容易导致服务崩溃等问题。 因此，主从复制模式可以缓解单服务器的压力，将写操作给主服务器，读操作给从服务器，从服务器可以部署多台，分摊压力。

As the number of users gradually increases, more and more requests to access the database will increase the burden/playload on the MySQL server, which will easily lead to server breakdown or other problems. Therefore, the primary/secondary replication mode relieves the pressure on a single server. Write operations are assigned to the primary server and read operations are assigned to the secondary servers. Multiple secondary servers can be deployed to share the pressure.

8.2 how to code if primary/secondary replication is underway

https://github.com/15029274098/springboot-
###critical https://blog.csdn.net/bj_chengrong/article/details/103814720
https://blog.51cto.com/u_15294985/2991323

9. index 索引问题

Chinese:
https://blog.csdn.net/qq_32828253/article/details/109630546
English:
https://vanseodesign.com/web-design/the-types-of-indexes-you-can-add-to-mysql-tables/

Single and Multiple Column Indexes

The first thing to know (and the first way we can distinguish index types) is that you can add an index to a single column or to multiple columns. In fact, in MySQL 8 an index can contain up to 16 columns though you probably won't ever need that many. Whether to use a single or multiple column index depends on the type of data for which you commonly search.

For example, you might have a customers table that holds columns for first_name and last_name. If you only ever search the last_name values then a single column index on the last_name would be appropriate. If you often search for both first and last names in

the same query, then you might consider a multiple column index on first_name and last_name.

Five Types of Indexes

When you create an index or add one to an existing table, you'll create it as one of several types of indexes.

Unique:
A unique index i**s one in which all column values must be unique(can be null but unique).** In a single column unique index there can be no duplication of values in the column being indexed. **In a multi-column unique index the values can be duplicated in a single column, but the combination of column values in each row must be unique**. You use a unique index to prevent duplicate values and you often define the index after a table has been created.

Unique and not null:
A primary key is **a unique index in which no value can be NULL. Every row must have a value for the column or combination of columns. You would usually define a primary key on the smallest number of columns possible because of thi**s, and most of the time a primary key will be set on a single column. Also, once set the column values in the primary key can't be changed.

Not unique and can be null:
A simple, regular, or normal index is an index where the values don't need to be unique and they can be NULL. This is the index I've mostly been talking about to this point. They're added simply to help the database find things faster.

A fulltext index, as the name implies, are used for full text searches. **Sometimes you want to find the blob of text that contains a certain word or group of words or maybe you want to find a certain substring within the larger block of text.**

Instead of the entire value being indexed as a whole, a fulltext index **will index the individual words inside each text block**. This makes it faster to find the specific words and phrases inside the whole text.

Index Data Structures

10.聚簇索引和非聚簇索引

https://cloud.tencent.com/developer/article/1541265
https://www.geeksforgeeks.org/difference-between-clustered-and-non-clustered-index/

MySQL has options in the type of data structure to use when creating index tables. **The indexes can be either clustered or non-clustered**. ***In a clustered index, the rows of data are stored on disk in the same order as the index.*** **Because of this you can only ever have a single clustered index on any table.**

A non-clustered index uses pointers to the data. In other words the structure of the index is separate from the structure of the data rows in the table. Because of the separate structures, you can use as many as you want and you can rearrange the index and still have everything pointing to the right place.

Clustered indexes are generally faster to read since you can get all the information from the index and not have to also consult the original table. A non-clustered index requires you find the specific index and use it to look up the data in the table. A clustered index can take longer to write to, especially if the new data requires the existing data to be reorganized for the index.

InnoDB tables in MySQL use the primary key as the clustered index. If no primary key exists, it will look for the first unique index without any NULL values. If it can't find either, it will create a clustered index based on a synthetic column containing row ID values. All other indexes are non-clustered.

1. InnoDB使用的是聚簇索引，将**主键组织到一棵B+树**中，而**行数据就储存在叶子节点**上，若使用"where id = 14"这样的条件查找主键，则**按照B+树的检索算法即可查找到对应的叶节点，之后获得行数据。**

2. 若**对Name列进行条件搜索，则需要两个步骤：第一步在辅助索引B+树中检索Name，到达其叶子节点获取对应的主键。**第二步**使用主键在主索引B+树种再执行一次B+树检索操作，最终到达叶子节点即可获取整行数据。（重点在于通过其他键需要建立辅助索引**）

MyISM使用的是非聚簇索引，**非聚簇索引的两棵B+树看上去没什么不同**，节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于**索引树是独立的，通过辅助键检索无需访问主键的索引树。**

为什么主键通常建议使用自增id

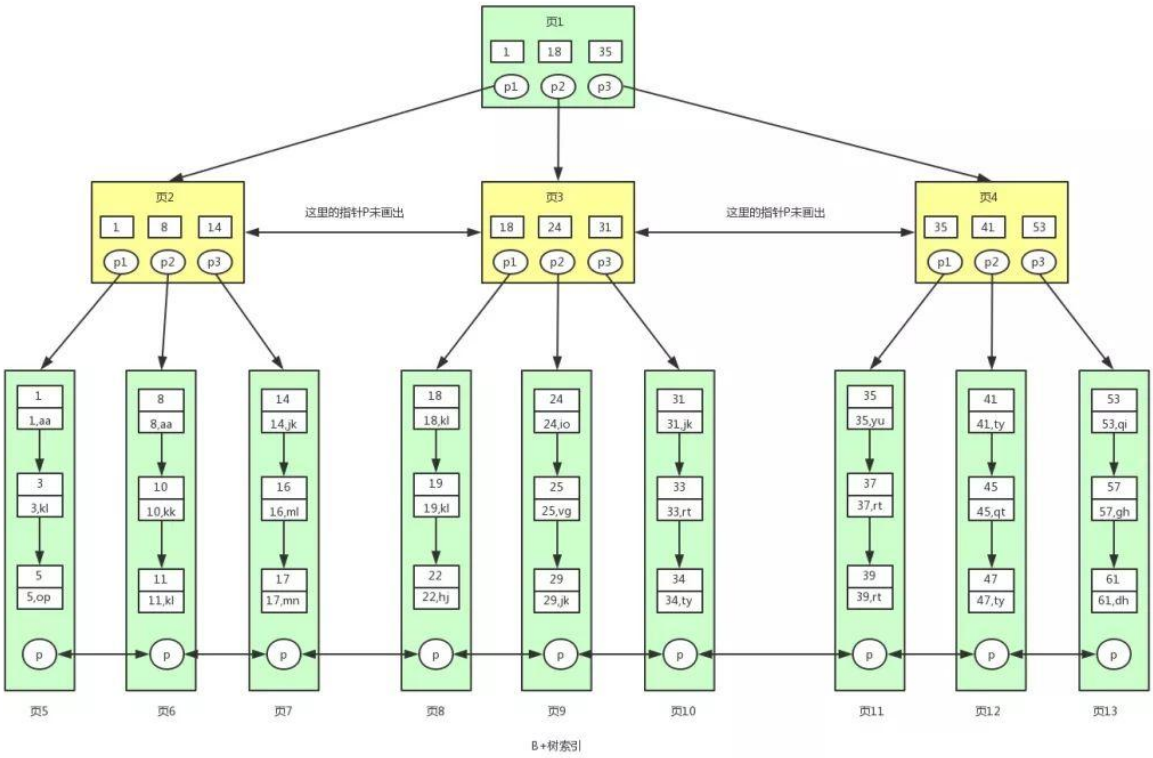
聚簇索引的数据的物理存放顺序与索引顺序是一致的，即：**只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上的。**如果主键不是自增id，那么可以想 象，它会干些什么，不断地调整数据的物理地址、分页，当然也有其他一些措施来减少这些操作，但却无法彻底避免。但，如果是自增的，那就简单了，它只需要一 页一页地写，索引结构相对紧凑，磁盘碎片少，效率也高。

11.B＋树的完美解释，聚簇索引和非聚簇索引

https://cloud.tencent.com/developer/article/1543335

B+树

B+树是对 B 树的进一步优化。让我们先来看下 B+树的结构图：



1. B+树非叶子节点上是不存储数据的，仅存储键值，而 B 树节点中不仅存储键值，也会存储数据。之所以这么做是因为在数据库中页的大小是固定的，innodb 中页的默认大小是 16KB。如果不存储数据，那么就会存储更多的键值，相应的树的阶数（节点的子节点树）就会更大，树就会更矮更胖，如此一来我们查找数据进行磁盘的 IO 次数有会再次减少，数据查询的效率也会更快。

另外，B+树的阶数是等于键值的数量的，如果我们的 B+树一个节点可以存储 1000 个键值，那么 3 层 B+树可以存储 1000×1000×1000=10 亿个数据。一般根节点是常驻内存的，所以一般我们查找 10 亿数据，只需要 2 次磁盘 IO。

2. 因为 B+树索引的所有数据均存储在叶子节点，而且数据是按照顺序排列的。那么 B+树使得范围查找，排序查找，分组查找以及去重查找变得异常简单。而 B 树因为数据分散在各个节点，要实现这一点是很不容易的。

有心的读者可能还发现上图 B+树中各个页之间是通过双向链表连接的，叶子节点中的数据是通过单向链表连接的。

其实上面的 B 树我们也可以对各个节点加上链表。其实这些不是它们之前的区别，是因为在 mysql 的 innodb 存储引擎中，索引就是这样存储的。也就是**说上图中的 B+树索引就是 innodb 中 B+树索引真正的实现方式，准确的说应该是聚集索引（聚集索引和非聚集索引下面会讲到）。**

通过上图可以看到，在 innodb 中，我们通过数据页之间通过双向链表连接以及叶子节点中数据之间通过单向链表连接的方式可以找到表中所有的数据。

MyISAM 中的 B+树索引实现与 innodb 中的略有不同。在 MyISAM 中，B+树索引的叶子节点并不存储数据，而是存储数据的文件地址。

1. **The non-leaf nodes of the B+ tree store no data but only key values, while the B tree node stores not only key values but also data.** This is done because the size of a page in a database is fixed, and the default size of a page in innodb is 16KB. I**f the data is not stored, then more key values will be stored, the corresponding order of the tree (the node's child node tree) will be larger, the tree will be shorter and fatter, so that we find the data to disk I/O times will be reduced again, and the efficiency of the data query will be faster.**

In addition, the order of the B+ tree is equal to the number of key values, if our B+ tree a node can store 1000 key values, then the 3-layer B+ tree can store 1000×1000×1000=1 billion data. In general, the root node is resident in memory, so in general, we find 1 billion data and only need 2 disk IO.

2. **Because all the data of the B+ tree index is stored in the leaf node, and the data is arranged in order.** So B+ trees make range lookup, sort lookup, group lookup, and de-lookup extremely easy. This is not easy for B-trees because the data is scattered across nodes.

Interested readers may also notice that the pages in the B+ tree above are connected by a bidirectional linked list, and **the data in the leaf nodes are connected by a unidirectional linked list.**

In fact, in the B-tree above we can also add a linked list to each node. In fact, these are not the previous differences, because this is how indexes are stored in mysql's innodb storage engine. That is to say, the B+ tree index in the figure above is the real implementation of the B+ tree index in innodb, which should be accurately said to be clustered index (clustered index and non-clustered index will be discussed below).

As can be seen from the figure above, in innodb, we can find all the data in the table by connecting the data pages through a bidirectional linked list and connecting the data in the leaf node through a unidirectional linked list.

The B+ tree indexing implementation in MyISAM is slightly different from that in innodb. In MyISAM, the leaf node of the B+

tree index does not store data, but rather the file address of the data.

12.Differences between sql and nosql

https://zhuanlan.zhihu.com/p/63371253

> 找到约 24,500,000 条结果 （用时 0.44 秒）
>
> SQL databases are vertically scalable, while NoSQL databases are horizontally scalable. SQL databases are table-based, while NoSQL databases are document, key-value, graph, or wide-column stores. SQL databases are better for multi-row transactions, while NoSQL is better for unstructured data like documents or JSON.

Sql
advantages:

- Transaction
- Support complex query like join
- commonly used sql script make db operation very easily

Disadvantages:

- Low R/W performance
- Difficult to modify structure of tables
- Bottleneck of IO makes the concurrency performance be lower
- If DB includes Big data, performance of query will be low

Nosql

非关系型数据库包括：

临时性键值存储：memcached、Redis=

永久性键值存储：ROMA、Redis

面向文档的数据库：MongoDB、CouchDB

面向列的数据库：Cassandra、HBase

主要代表：MongoDB，Redis，CouchDB

- **面向高性能并发读写的 key-value 数据库**：key-value数据库的主要特点即使具有极高的并发读写性能，Redis，Tokyo Cabinet，Flare 就是这类的代表
- **面向海量数据访问的面向文档数据库**：这类数据库的特点是，可以在海量的数据中快速的查询数据，典型代表为 MongoDB 以及 CouchDB
- **面向可扩展性的分布式数据库**：这类数据库想解决的问题就是传统数据库存在可扩展性上的缺陷，这类数据库可以适应数据量的增加以及数据结构的变化

面向扩展性的分布式数据库 cassandra

阶，这类数据库可以适应数据量的增加以及数据结构的变化

2.3 非关系型数据库优点

非关系型数据库的出现，多是源于关系型数据库的性能不足，故非关系型数据库的优点也很明显：

- **读写性能**：无需经过 SQL 层的解析，读写性能很高。主要例子有Redis，由于其逻辑简单，而且纯内存操作，使得其性能非常出色，单节点每秒可以处理超过10万次读写操作；
- **简单的扩展**：基于键值对，数据没有耦合性，容易扩展。典型例子是 Cassandra，由于其架构是类似于经典的 P2P，所以能通过轻松地添加新的节点来扩展这个集群；
- **存储格式多**：支持key-value形式、文档形式、图片形式，而关系型数据库则只支持基础类型；
- **低廉的成本**：这是大多数分布式数据库共有的特点，因为主要都是开源软件，没有昂贵的License成本；

2.4 非关系型数据库缺点

- **不提供对 SQL 的支持**：如果不支持 SQL 这样的工业标准，将会对用户产生一定的学习和应用迁移成本
- **支持的特性不够丰富**：现有产品所提供的功能都比较有限，大多数 NoSQL 数据库都不支持事务，也不像 MS SQL Server 和 Oracle 那样能提供各种附加功能，比如 BI 和报表等
- **现有产品的不够成熟**：大多数产品都还处于初创期，和关系型数据库几十年的完善不可同日而语

3. 关系型数据库与 NoSQL

3.1 NoSQL 使用场景

并不是任何场景，NoSQL 都要优于关系型数据库。NoSQL 使用场景常见如下：

数据库表 schema 经常变化
比如在线商城，维护产品的属性经常要增加字段，这就意味着 ORMapping 层的代码和配置要改，如果该表的数据量过百万，新增字段会带来额外开销（重建索引等）。

数据库表字段是复杂数据类型
对于复杂数据类型，比如 SQL Sever 提供了可扩展性的支持，像 xml 类型的字段。DB 层对 xml 字段很难建高效索引，应用层又要做从字符流到 dom 的解析转换。NoSQL 以 json 方式存储，提供了原生态的支持，在效率方便远远高于传统关系型数据库。

高并发数据库请求
此类应用常见于 web2.0 的网站，很多应用对于数据一致性要求很低，而关系型数据库的事务以及大表 JOIN 反而成了”性能杀手”。

海量数据的分布式存储
海量数据的存储如果选用大型商用数据，如 Oracle，那么整个解决方案的成本是非常高的，要花很多钱在软硬件上。NoSQL 分布式存储，可以部署在廉价的硬件上，是一个性价比非常高的解决方案。

方案。

3.2 NoSQL 和关系数据库结合

一般把 NoSQL 和关系数据库进行结合使用，各取所长，需要使用关系特性的时候我们使用关系数据库，需要使用 NoSQL 特性的时候我们使用 NoSQL 数据库，各得其所。NoSQL 数据库是关系数据库在某些方面（性能，扩展）的一个弥补。

举个简单的例子吧，比如用户评论的存储，评论大概有主键 id、评论的对象 aid、评论内容 content、用户 uid 等字段。我们能确定的是评论内容 content 肯定不会在数据库中用 where content=’’ 查询，评论内容也是一个大文本字段。那么我们可以把主键 id、评论对象 aid、用户 id 存储在数据库，评论内容存储在 NoSQL，这样数据库就节省了存储 content 占用的磁盘空间，从而节省大量 IO，对 content 也更容易做 Cache。

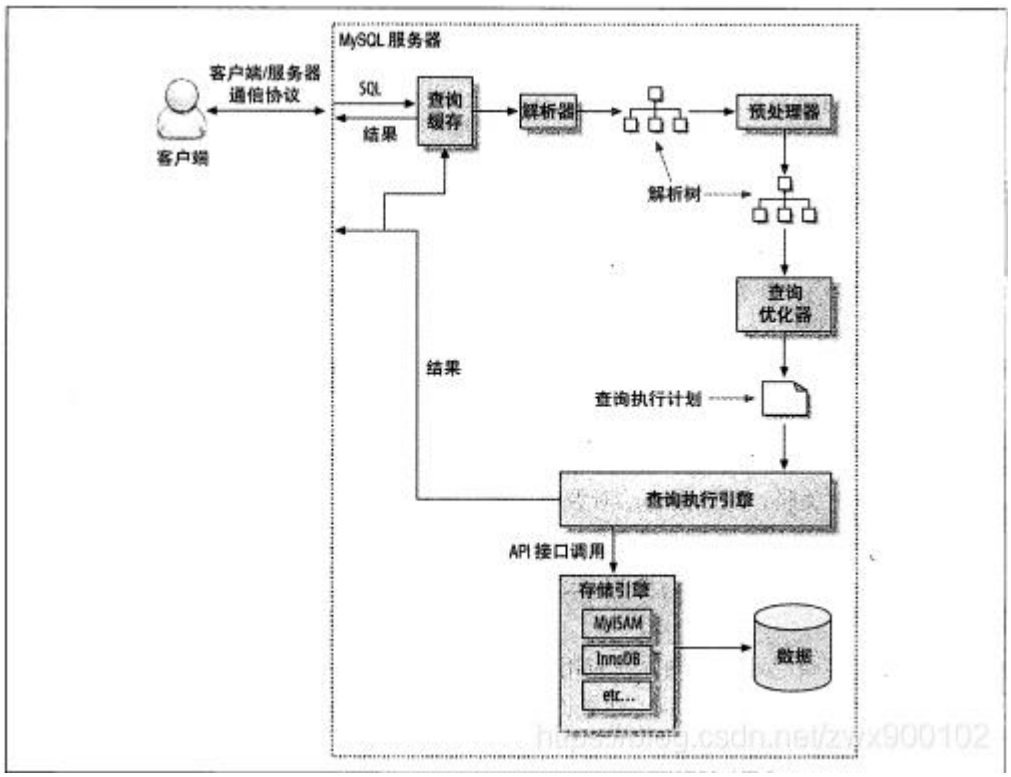
另外，**可使用 NoSQL 作为缓存服务器。**MySQL + Memcached 的架构中，Memcached 这类内存缓存服务器缓存的数据大小受限于内存大小，如果用 NoSQL 来代替 Memcached 来缓存数据库的话，就可以不再受限于内存大小。虽然可能有少量的磁盘IO读写，可能比 Memcached 慢一点，但是完全可以用来缓存数据库的查询操作。

13.describe scenarios in which indexes are not effective

https://www.51cto.com/article/702691.html

14.mysql 语句执行流程

https://blog.csdn.net/zwx900102/article/details/106741454



根据流程图，一条select查询大致经过以下六个步骤：

1、客户端发起一个请求时，首先会建立一个连接

2、服务端会检查缓存，如果命中则直接返回，否则继续之后后面步骤

3、服务器端根据收到的sql语句进行解析，然后对其进行词法分析，语法分析以及预处理

4、由优化器生成执行计划

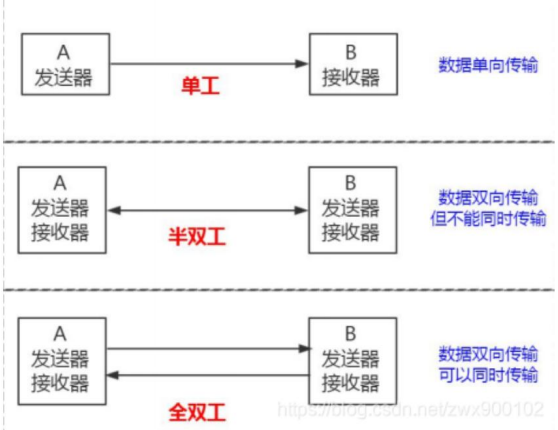
5、调用存储引擎层API来执行查询

6、返回查询到的结果

建立连接:

第一步建立连接，这个很容易理解，需要特别指出的是MySQL服务端和客户端的通信方式采用的是**半双工协议**。

通信方式主要可以分为三种：单工，半双工，全双工，如下图：



- 单工：通信的时候，数据只能单向传输。比如说遥控器，我们只能用遥控器来控制电视机，而不能用电视机来控制遥控器。
- 半双工：通信的时候，数据可以双向传输，但是同一时间只能有一台服务器在发送数据，当A给B发送数据的时候，那么B就不能给A发送数据，必须等到A发送结束之后，B才能给A发送数据。比如说对讲机。
- 全双工：通信的时候，数据可以双向传输，并且可以同时传输。比如说我们打电话或者用通信软件进行语音和视频通话等。

半双工协议让MySQL通信简单快速，但是也在一定程度上限制了MySQL的性能，因为一旦从一端开始发送数据，另一端必须要接收完全部数据才能做出响应。所以说我们批量插入的时候尽量拆分成多次插入而不要一次插入太大数据，同样的查询语句最好也带上limit限制条数，避免一次返回过多数据。

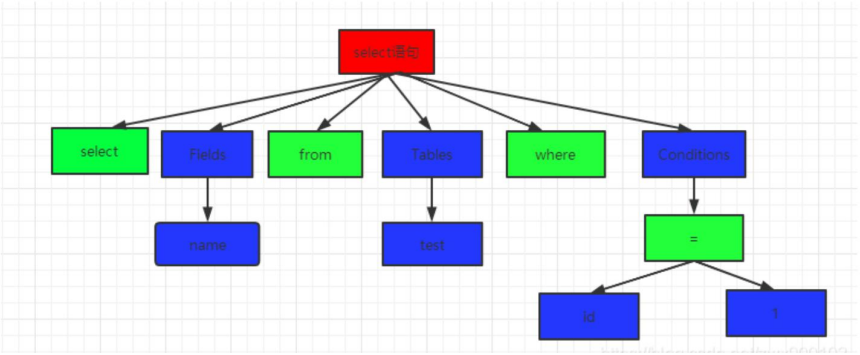
MySQL单次传输数据包的大小可以通过参数max_allowed_packet控制,默认大小为4MB

查询缓存(略)

解析器(检查语法结构)和预处理器(检查表名,列名)

词法解析和语法解析(Parser)

这一步主要的工作就是检查sql语句的语法对不对，在这里，首先会把我们整个SQL语句打碎，比如：select name from test where id=1，就会被打散成select，name，from，test，where，id，=，1 这8个字符，并且能识别出关键字和非关键字，然后根据sql语句生成一个数据结构，也叫做解析树(select_lex)，如下图：



预处理器(Preprocessor)

经过了前面的词法和语法解析，那么至少我们一条sql语句的语法格式是满足要求了，接下来我们还需要做什么呢？自然是检查表名，列名以及其他一些信息等是不是真实存在的，**预处理就是做一个表名和字段名等相关信息合法性的检测。**

查询优化器(Query Optimizer),very critical

优化器可以做哪些优化

优化器可以替我们做很多优化，下面列举一些常用的优化：

- 重新定义关联的顺序。优化器并不一定按照我们写的查询关联语句中的关联顺序，而是会按照优化后的顺序进行查询。
- 将外连接转为为内连接。
- 使用等价转换原则。比如a<b and a=5会被转换为a=5 and b>5
- 优化COUNT(),MIN()和MAX()
- 预估并转化为常数表达式
- 覆盖索引扫描。想要详细了解覆盖索引的可以点击这里。
- 子查询优化。
- 提前终止查询。比如我们使用了一个不成立的条件，则会立刻返回空。
- 等值传播。
- 优化IN()语句。在其他很多数据库中in等同于or语句，但是MySQL中会讲in中的值先进行排序，然后按照二分查找的方法来确定是否满足条件。

实际当中优化器能做的优化远远比上面列举的更多，所以有时候我们不要觉得比优化器更聪明，所以大部分情况下我们都可以让优化器做出优化就可以了，如果有些我们确定优化器没有选择最优的查询方案，我们也可以在查询中通过添加hint提示告知到优化器，比如通过force index强制使用索引或者straight_join语句强制优化器按我们想要的表顺序进行关联。

但切记优化器不是万能的,合理的 sql 优化还是很必要

优化器并不是万能的

MySQL优化器也并不是万能的，并不是总能把我们写的糟糕的sql语句优化成一个高效的查询语句，而且也有很多种原因会导致优化器做出错误的选择：

存储引擎查询

当Server层得到了一条sql语句的执行计划后，这时候就会去调用存储引擎层对应的API，执行查询了。

O. Bug position or performance improvement tool

1. how do you know where the performance issue occurs

Including slow sql

2. How to monitor API usage, load, like TPS

https://www.51oneday.com/2019/11/java-xiangmuxingnengjiankonghediaoyougongju-javamelody/

dynatrace

3. How to position a bug by a tool

Purepath in dynatrace

P. Behavior questions

4. What will you do when you have a different opinion/ conflict with your colleagues in your work?

In hsbc, there was ever a task about getting pdf stream from i2i side. i needed to query two dif**ferent tables. Which table should be chosen was decided by parameter sent from frontend. my colleague also a candidate to be a leader said that my code was wrong and at the beginning she insisted my code logic is not correct.**
At first , i assumed my code was not correct. because she is more experienced and there is bigger possibility that i was wrong. And i have to attach more importance to

her opinion. So i checked email about the business logic and **scrutinized** my code and tested again and again . and finally i am sure my code was correct. After that i discussed with her and told her what i thought and did in this requirement.we discussed the whole execution flow by **sharing screen** in **zoom**, and together found that she ignored the annotation which is called @defaultvalue on the service project, instead she focused on connector project. And she accepted my opinion.
From this lesson, when somebody, especially more experienced one has different opinion with mine, at first i have to check whether this issue actually occurs because of my own fault. The second point is that, we should just focus on the task itself instead of person/my colleague, every one may make mistakes, i should have neutral attitude towards any problems , and try to solve it together without any emotion.all in all, I should keep in mind that all teammates face the same goal, that is we are supposed to finish task on time. We all hope every thing goes well.

5. work alone or work in a team

Both, when i begin coding , i like a quiet atmosphere, but before coding we need to discuss with teammate to clarify the requirement. And within a team meeting regularly is necessary, Exchanging opinions, reporting progress.

6. A challenging project

http-proxy-middleware

和 axios 之争

In leyaoyao company .which is a IOT Company. We need to deploy code to product environment every week, so working there is stressful .I was assigned to count every time of usage information of every machine.this is a high concurrency scenario so i need to import mongoDB, which is **totally new to me**. And i just have two days for coding before deployment to dev environment. At first i designed the coding logic : temporarily store usage information in mongodb , and every 30 minutes i copy data from mongodb to postgreSql. I goolged on how to use mongodb , worked till midnight and just slept for 5 hours those two days. but finally i successfully completed this task.
I think sometimes maybe i will come across those urgent tasks, it’s quite common in a company , but once i was assigned a task, i must try everything to make it done on time.

7. what makes you angry?

If anyone or anything is going **off track,** I want to know about it early. If, after that kind

of open communication and follow up, someone isn't getting the job done, I'll want to know why. If there's no good reason, then I'll get impatient and angry…and take appropriate steps from there. But if you hire good people, motivate them to strive for excellence and then follow up constantly, it almost never gets to that state.

8. your ideal manager

Track project progress of his subordinates , by organizing meeting regularly.
Good communication skills and willing to communicate with his teammate. And be patient and open-minded to suggestions from his subordinates.
Sometimes Assign subordinates a challenging task, which can make subordinates progress and give them feel of achievement.

9. a time persuading others.

There is a task i need to split a text into 20 sections and every section has 4000 characters. I was a new beginner in HSBC at that time, My leader told me use Object in which there are like section1 section2 ,section3 . **I didn't think a lot at first and proceed according to this design**. But i found I have to use java reflection yo position the field. And I designed a new structure of the parameter by using a list in another list , and i tried locally passed the unit test and make sure it is better. Then i walked to my leader's seat sitting next to her and explained what i thought and she accept ed my advice.
I think before giving advice to somebody especially a leader , i should at least make everything clear.

10.Different opinion in the meeting

The goal of meeting is to solve a problem, it is everyone's responsibility to express his views.it is the process of exchanging ideas .
I have to make some preparations before the meeting, be more familiar with the content of the meeting.
When i express my idea, i should express it **logically , reasonably, and professionally .**
Besides, I should keep a harmonious atmosphere, when it's my turn to speak, i should use soft tone; **when somebody else is speaking , I should not interrupt him and listen carefully to what he is saying** .otherwise it may upset him. **When others give good advice, i should contemplate his idea and if it is better ,i should be open-minded to accept it.**
If the final group opinion is **confirmed**, even if it is different from mine, I should commit it absolutely.

11. another conflict story

our product page response time is very slow, another colleague/classmate and I are responsible for improvement.My idea is that the design of our database Table is not reasonable, which leads to too many requests and slows down the speed, so we should redesign it.My colleague's idea was that changing the database would cause a lot of code to need to be changed, which would be potentially risky, so caching should be introduced.We each felt we had a point and couldn't convince each other.The time was tight and the task was heavy, so I proposed to the other party that we all wanted to make the product better, and it was difficult to analyze only in this way. How about we cooperate and spend half a day together to write a simplified version of the two methods, and then test them, and finally choose a better solution?The other party agreed with my proposal, and finally, after comprehensive analysis, he thought my plan was good. Then my colleague/classmate also understood my idea, and of course I also understood his idea. On this basis, the efficiency of our cooperation became higher, and the cooperation between us became more smooth in the future.

12.What would you be a good fit for this position? How would you affect our culture?

Q. Do you have any questions for me?

1. **Why is this position open — is this a new role? If not, why did the person who held this role leave it?**
2. **If I were hired for this role, what would you want me to achieve in my first months in the position?**

3. **What's one thing that makes you happy to come into work today?**
4. **Can you talk about the company culture a bit?**
5. **Career path**

R. Why our company

1. **This position in your company is a good match for my plan. At the same time, I am very optimistic about the industry of your company. I think your company attaches great importance to talents, and this job is very suitable for me.**
2. **Must be familiar with those industries**

S. Tools

1. idea 好用的插件

性能工具:Jprofile arthas,
Bug position or performance improvement: dynatrace,jconsole.visualVM
Code generate: Easycode
Unit test coverage rate: jacoco
Code perfect: alibaba java code guideline; solarLint

2. Ahportal Dynatrace

Setup liveness and readiness in ahportal and send request periodically

3. performance review through a tool

Jmeter 设置在 jenkinsjob,

4. monitor and alerts tool

For spring boot 项目
https://www.51oneday.com/2019/11/java-xiangmuxingnengjiankonghediaoyougongju-javamelody/

Suncorp 使用的是 splunk,dynatrace

5. how to position bug through a tool

Purepath in Dynatrace can help you position the bug as soon as possible

6. how to do unit test and integration test through a tool

Jmeter configured in jenkinsjob

T.difficulties fixed in one project

1. example

In Suncorp: cors issue when deployed project on openshift

Background: transform old project by splitting frontend and backend
Keywords:
At first , think about all possible solutions, then make the best use of resourcese.

Tow lines to solve it: frontend engineer keep trying to use http-middleware-proxy and based on this , trying to find solution

I deploy a new project based on another branch , and trying to solve it from backend, and cha

U. Fundamental of System design

1. Key characteristics of distributed system

https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/6426c14d1e9abd7d40f2c3fd

2. API gateway

https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/641ed36e12e1f4e2a46ca14d

An API gateway is focused on **routing(based on url, method or parameters)** requests to the appropriate microservice, while a load balancer is focused on distributing requests evenly across a group of backend servers.

3. DB partition(分区) and sharding(分库)

1.0 Critical English Version:

https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/64224b41a763c048c9763d8f

1.1 Why we need to use sharding

The reasons why we need to use partition or sharding:
1 Data size is too large,query will be slow()
2 if concurrency pressure is too large ,all requests go to the same endpoint, I/O of a disk will be affected a lot.
3 and after sql optimization methods can not improve performance(so 一般先尝试 sql 优化，再考虑 partition)

简单地说，**数据量大，就分表；并发高，就分库。**

1.2 分库(sharding)分区(partition)

https://www.singlestore.com/blog/database-sharding-vs-partitioning-whats-the-difference/

Sharding and partitioning are both about breaking up a large data set into smaller subsets. The difference is that sharding implies the data is spread across multiple computers while partitioning does not. Partitioning is about grouping subsets of data within a single database instance.

1.3 Types of partition:

2.2、分区类型
MySQL 5.1中可用的分区类型包括：
- RANGE分区(portioning)：根据列值所属的范围区间，将元组分配到各个分区。
- LIST分区：类似于按RANGE分区，区别在于LIST分区是基于列值匹配一个离散值集合中的某个值来进行选择。
- HASH分区：根据用户定义的函数的返回值来进行选择的分区，该表达式使用将要插入到表中的这些行的列值进行计算。这个函数可以包含MySQL 中有效的、产生非负整数值的任何表达式。
- KEY分区：类似于按HASH分区，区别在于KEY分区只支持计算一列或多列，且MySQL 服务器提供其自身的哈希函数。

https://www.cnblogs.com/hustcat/archive/2009/12/24/1631674.html
https://news.sangniao.com/p/2684880278

1.4 partition 后潜在问题的解决方式

https://zq99299.github.io/note-book/back-end-storage/03/01.html

比如我们把 **订单** ID 作为 Sharding Key 来拆分订单表，那拆分之后，如果我们按照订单 ID 来查订单，就需要先根据订单 ID 和分片算法计算出，我要查的这个订单它在哪个分片上，也就是哪个库哪张表中，然后再去那个分片执行查询就可以了。

但是，当我打开**「我的订单」**这个页面的时候，它的 **查询条件是用户** ID，这里没有订单 ID，那就没法知道我们要查的订单在哪个分片上，就没法查了。当然你要强行查的话，那就只能把所有分片都查一遍，再合并查询结果，这个就很麻烦，而且性能很差，还不能分页。

那要是把用户 ID 作为 Sharding Key 呢？也会面临同样的问题，使用订单 ID 作为查询条件来查订单的时候，就没办法找到订单在哪个分片了。这个问题的解决办法是，在生成订单 ID 的时候，把用户 ID 的后几位作为订单 ID 的一部分，比如说，可以规定，18 位订单号中，第 10-14 位是用户 ID 的后四位，这样按订单 ID 查询的时候，就可以根据订单 ID 中的用户 ID 找到分片。

那我们系统对订单的查询方式，肯定不只是按订单 ID 或者按用户 ID 这两种啊。比如说，商家希望看到的是自己店铺的订单，还有各种和订单相关的报表。对于这些查询需求，**我们一旦对订单做了分库分表，就没法解决了**。那怎么办呢？

一般的做法是，把订单数据同步到其他的存储系统中去，在其他的存储系统里面解决问题。比如说，我们可以再构建一个以店铺 ID 作为 Sharding Key 的只读订单库，专门供商家来使用。或者，把订单数据同步到 HDFS 中，然后用一些大数据技术来生成订单相关的报表。

DFS 中，然后用一些大数据技术来生成订单相关的报表。

所以你看，**一旦做了分库分表，就会极大地限制数据库的查询能力**，之前很简单的查询，分库分表之后，可能就没法实现了。所以我们在之前的课程中，先讲了各种各样的方法，来缓解数据多、并发高的问题，而一直没讲分库分表。**分库分表一定是，数据量和并发大到所有招数都不好使了，我们才拿出来的最后一招。**

这么看 还不如 nosql
Solution:
https://blog.51cto.com/u_16213433/7299970?articleABtest=1

4. redis and mongoDB(怎样设计点赞系统)

https://www.cnblogs.com/111testing/p/11190823.html

Redis: used as cache;much higher TPS; less data scenarios
mongoDB: used as DB;higher TPS; has mapreduce(analyse data);huge data scenarios

现在的分布式项目基本都会用到redis和mongodb，可是redis和mongdb到底有什么不同呢，今天我就基于我们公司的项目来具体介绍一下redis和mongodb的各自的应用场景。
首先我们这个项目中有两种应用场景：
场景一：要求TPS（不知道的右转百度）特别高的，比如我们项目有一个点赞的功能，这个点赞的功能促发频率特别高，而且并发量也会特别大，但是它的数据量不会特别大。基于这种情况下，我们采用redis来实现点赞功能。
场景二：项目中涉及评论的内容，而且这个评论表的数据后期会非常大（海量的数据），最后在数据量非常大的情况下还要求比较复杂的查询。基于上述这些情况，我们采用mongodb作为评论表存储数据库。

1. java
18)
2. andr
法(706
3. pyth
y文件中
4. Excel
色(4456
5. java

5. TPS QPS

个人理解如下：

1、Tps即每秒处理事务数，包括了

- 用户请求服务器
- 服务器自己的内部处理
- 服务器返回给用户

这三个过程，每秒能够完成N个这三个过程，Tps也就是N；

2、Qps基本类似于Tps，但是不同的是，对于一个页面的一次访问，形成一个Tps；但一次页面请求，可能产生多次对服务器的请求，服务器对这些请求，就可计入“Qps”之中。

例子：

> 例如：访问一个页面会请求服务器3次，一次放，产生一个“T”，产生3个“Q”
>
> 例如：一个大胃王一秒能吃10个包子，一个女孩子0.1秒能吃1个包子，那么他们是不是一样的呢？答案是否定的，因为这个女孩子不可能在一秒钟吃下10个包子，她可能要吃很久。这个时候这个大胃王就相当于TPS，而这个女孩子则是QPS。虽然很相似，但其实是不同的。

6. Latency and Throughput

1 background : take washer(30) and dryer(60) as example
overlapping the processing , called pipeline system,each of the processing steps is called a stage of pipeline, the rate at which inputs move through the pipeline is determined by the slowest pipeline stage.

In other words, the effective processing rate of our overlapped laundry system is one

load every 60 minutes .so once the processing is underway N loads of laundry takes N *60 minutes

Total = max(washer,dryer)

2 Latency:
The delay from when an input is established until output associated with that input becomes valid.

Harvard laundry : 90 minutes **lower and better**
6.004 laundry : 120 minutes

3 throughput
The rate at which the inputs and outputs are processed
Harvard laundry : 1/90 outputs/minutes or one load of laundry every 90 minutes
6.004 laundry : 1/60 outputs/minutes or one load of laundry every 60 minutes **higher and better**

This example Illustrates the common trade-off between latency and throughput

In short, the throughput is a function of how many stages are in parallel while latency is a function of how many are in series when there are multiple stages in the processing. The stage with the lowest throughput determines the overall throughput.

从这个比喻中我们可以看出，延迟测量的是每个客户（每个应用程序）感受到的时间长短，而吞吐量测量的是整个银行（整个操作系统）的处理效率，是两个完全不同的概念。用作者的原话说是：

7. LOAD BALANCER

https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/641da6bad2bb4146954f1130

8. distributed consensus

Data consistency and trade-offs is distributed systems
https://www.bmc.com/blogs/cap-theorem/

8.1 what is consensus problem:

This concepts involves two scenarios:

- In cluster, if one node has run the write operation, this updated has to be made in all nodes , this is data consistency in cluster which **can be achieved by zookeeper**

- Or among different sub systems in a distributed system, data should be consistent , and consistency **can be achieved by distributed transaction**

但实际上上面两个场景的数据一致性解决原理是一样的

8.2 Types of consistency

https://developer.aliyun.com/article/140480

Strong consistency
When the update operation is complete, any subsequent accesses by multiple processes or threads will return the latest updated value. This is the most user-friendly, the user is guaranteed to read what they wrote at the last time. According to CAP theory, this implementation requires a sacrifice of availability.

Weak consistency
There is no guarantee that process or thread access will return the latest updated value.

Eventual Consistency

弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。
A particular form of weak consistency. The system guarantees that the value of the last update operation will be returned if there is no subsequent update. On the premise that no failure occurs, the time of inconsistent window is mainly affected by communication delay, system load and the number of duplicate copies.

8.3 General solution to consistency issue--distributed transaction

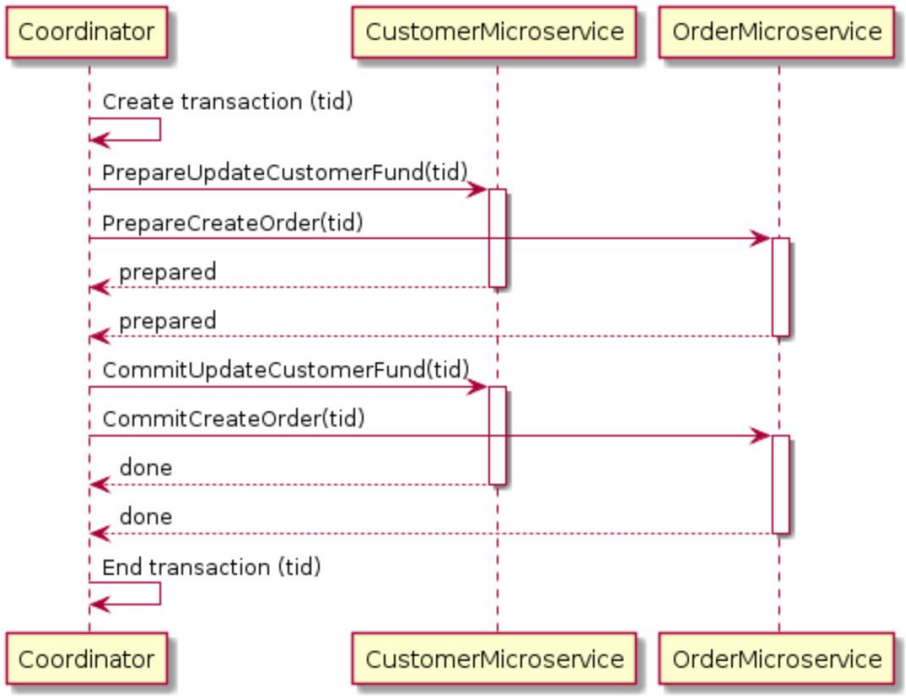
Chinese:
https://mp.weixin.qq.com/s/ujRRtdLOeKEHsHrtDRNXGA
https://mp.weixin.qq.com/s/XknegP66mnYboiBx556Kzw

English:
https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture#possible_solutions

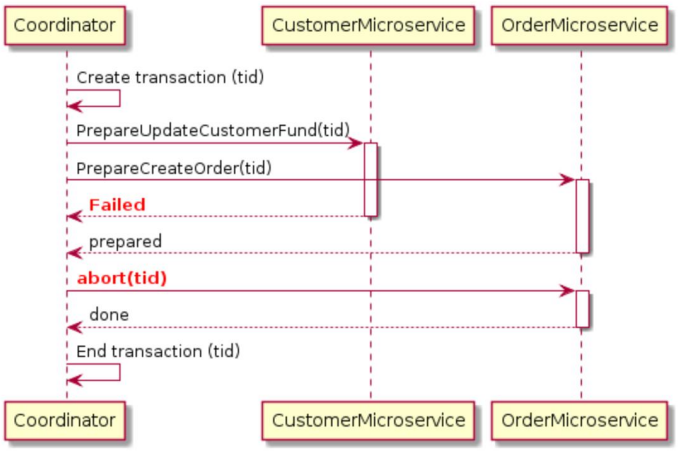
8.4 What is a two-phase commit?

As its name hints, 2pc has two phases: **A prepare phase and a commit phase.** In the prepare phase, all microservices will be asked to prepare for some data change that could be done atomically. Once all microservices are prepared, the commit phase will ask all the microservices to make the actual changes.
Normally, there needs to be a global coordinator to maintain the lifecycle of the transaction, and the coordinator will need to call the microservices in the prepare and commit phases.

Here is a 2pc implementation for the customer order example:



If at any point a single microservice fails to prepare, the `Coordinator` will abort the transaction and begin the rollback process. Here is a diagram of a 2pc rollback for the customer order example:



In the above example, the `CustomerMicroservice` failed to prepare for some reason, but the `OrderMicroservice` has replied that it is prepared to create the order. The `Coordinator` will request an abort on the `OrderMicroservice` with the transaction and the `OrderMicroservice` will then roll back any changes made and unlock the database objects.

Peter Dotchev • a year ago
Nice overview for a common problem with microservices.
What happens in 2PC if a commit fails (e.g. due to network issues)?
2 | • Reply • Share ›

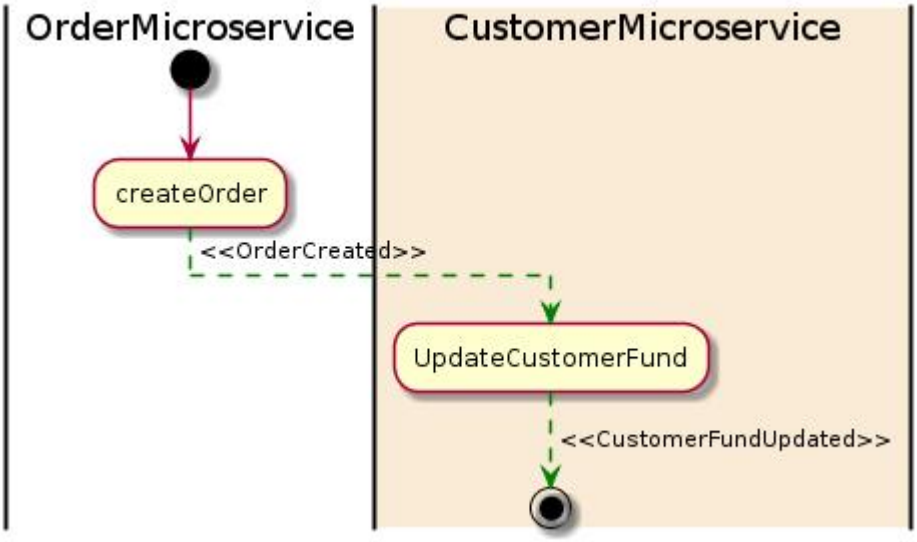
Josh West ↗ Peter Dotchev • a year ago
When a 2PC commit fails for any reason between the prepare and commit phase the Transaction is in a "Heuristic State" (undetermined state). The transaction manager is responsible for recovering these Heuristic state transactions, and typically keeps a log of transaction Ids (XIDs) on the local filesystem. This is another reason why 2PC transactions are not preferred in the microservices world, given the XID log makes the microservice process stateful. There are configurations, such as storing the XID log in database (Narayana TX Mgr does this), or 1.5 phase commits, that make distributed transactions still workable with the microservice design traits.
1 | • Reply • Share ›

Shashank Dhyani ↗ Peter Dotchev • a year ago

8.5 Saga pattern(communication or emitting message by MQ)

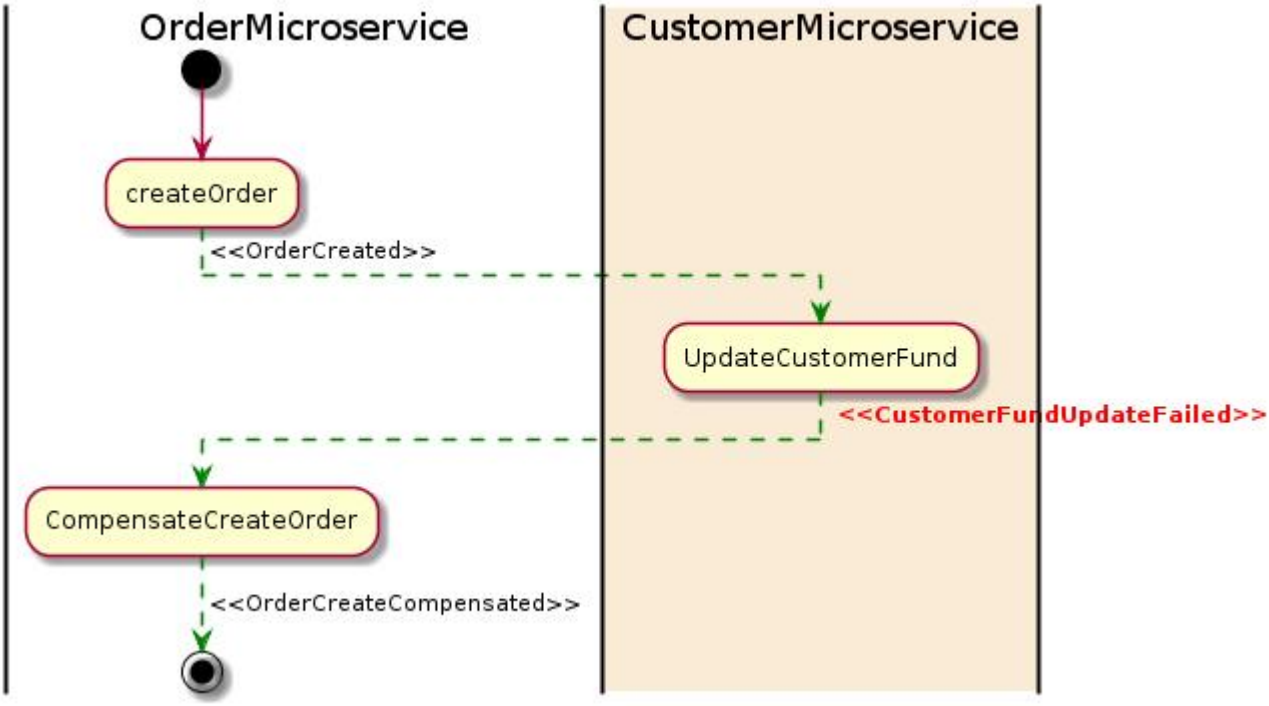
The Saga pattern is another widely used pattern for distributed transactions. It is different from 2pc, which is synchronous. The Saga pattern is asynchronous and reactive. In a Saga pattern, the distributed transaction is fulfilled by asynchronous local transactions on all related microservices. **The microservices communicate with each other through an event bus.**

Here is a diagram of the Saga pattern for the customer order example:



In the example above, the **OrderMicroservice** receives a request to place an order. It first starts a local transaction to create an order and then emits an **OrderCreated** event. The **CustomerMicroservice** listens for this event and updates a customer fund once the event is received. If a deduction is successfully made from a fund, a **CustomerFundUpdated** event will then be emitted, which in this example means the end of the transaction.

If any microservice fails to complete its local transaction, the other microservices will run compensation transactions to roll back the changes. Here is a diagram of the Saga pattern for a compensation transaction:



In the above example, the **UpdateCustomerFund** failed for some reason and it then emitted a **CustomerFundUpdateFailed** event. The **OrderMicroservice** listens for the event and start its compensation transaction to revert the order that was created.

Advantages of the Saga pattern
One big advantage of the Saga pattern is its support for long-lived transactions. Because each microservice focuses only on its own local atomic transaction, other microservices are not blocked if a microservice is running for a long time. This also allows transactions to continue waiting for user input. Also, because all local transactions are happening in parallel, there is no lock on any object.

Disadvantages of the Saga pattern
The Saga pattern is difficult to debug, especially when many microservices are involved. Also, the event messages could become difficult to maintain if the system gets complex. Another disadvantage of the Saga pattern is it does not have read isolation. For example, the customer could see the order being created, but in the next second, the order is removed due to a compensation transaction.

Adding a process manager
To address the complexity issue of the Saga pattern, it is quite normal to add a process manager as an **orchestrator**. The process manager is responsible for listening to events and triggering endpoints.

Conclusion
The Saga pattern is a preferable way of solving distributed transaction problems for a microservice-based architecture. However, it also introduces a new set of problems, such as how to atomically update the database and emit an event.

Adoption of the Saga pattern requires a change in mindset for both development and testing. It could be a challenge for a team unfamiliar with this pattern. There are many variants that simplify its implementation. Therefore, it is important to choose the proper way to implement it for a project.

8.6 which framework for distributed transaction?

Seata 里面支持 sage pattern

8.7 Zookeeper usage

https://www.cnblogs.com/YuyuanNo1/p/12877611.html

https://zhuanlan.zhihu.com/p/25594630

Cluster management and election of master

Load balancer

Configuration center

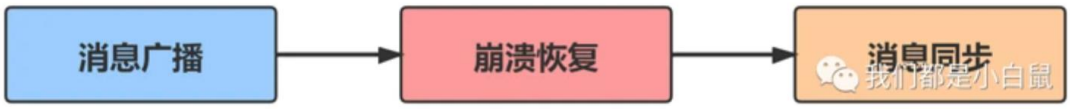
Distributed lock

Naming service

8.8 how to achieve consensus

How to achieve it by zookeeper
https://segmentfault.com/a/1190000022683594
https://mp.weixin.qq.com/s/k0zAExTvMsrugCaNJxs7hQ

ZooKeeper ensures data consistency through ZAB protocol(ZooKeeper Atomic Broadcast) which includes three processes: message broadcast, crash recovery, and data synchronization.

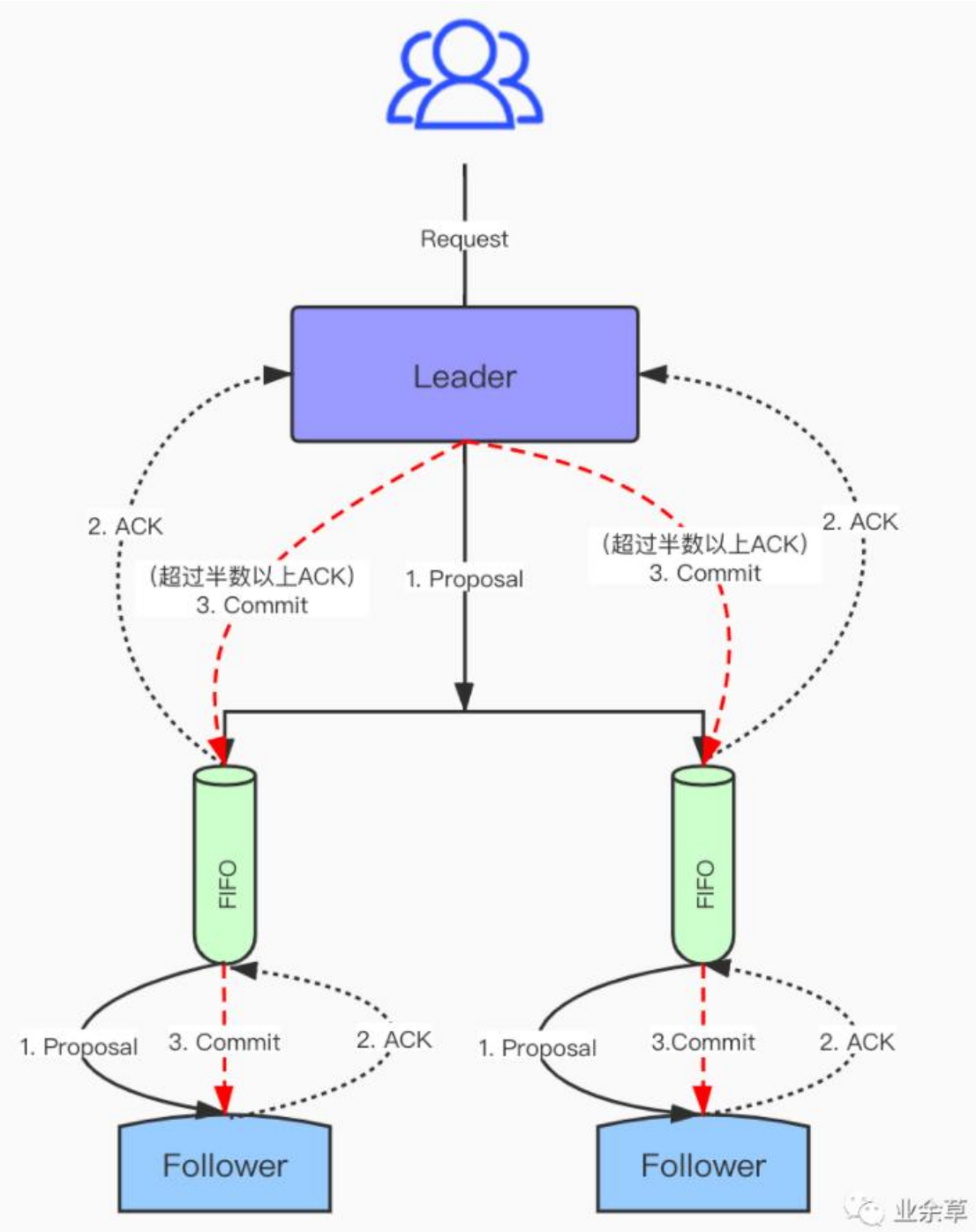


1. **message broadcast**

- After a transaction request (Write) comes in, the Leader wraps the Write reque st into a Proposal transaction and adds a globally unique and incrementing 64 -bit transaction ID, known as the Zxid (messages are ordered by comparing zxids).
- The Leader node broadcasts the Proposal transaction to other nodes in the cl uster. The Leader node and the followers node are decoupled. The communic

ation will pass through a FIFO message queue, and the Leader will allocate a separate FIFO queue to each Follower node. The Proposal is then sent to the queue;

- After receiving the Proposal, the Follower node persists it to the disk and sends an ACK to the Leader.
- When the Leader node receives more than half of the Follower nodes' ACKS (Quorum mechanism), it will commit the transaction on the local machine and broadcast the COMMIT(command) at the same time. After the Follower nodes receive the COMMIT, they will complete their own transaction commitment



2. crash recovery

ZAB protocol introduces crash recovery mode to solve the problem of Leader failure and data inconsistency **caused by Leader failure.**

- In ZAB protocol, in order to ensure the correct operation of the program, a new Leader server needs to be elected after the whole recovery process.
- The Leader election algorithm not only has to make the Leader itself know that it has been elected as the Leader, but also has to make all other machines in the cluster quickly perceive the new Leader server generated by the election.

Crash recovery mode must address the following issues:

- When Sever1 recovers and joins the cluster again, it must ensure that a Proposal1 is discarded, that the discarded message does not reappear. **The newly elected Leader must not include uncommitted proposals.**
- The new Leader elected must have the Proposal with the largest Zxid of all the machines in the cluster, that is, ensure that the messages that have been processed cannot be lost.

3. data synchronization

All normally running servers become either a Leader or a Follower which keeps in sync with the Leader.

- After the Leader election (the new Leader with the highest ZXID) is completed, and before the official start of work (i mean receiving client requests). the Leader server will first confirm whether all proposals in the transaction log have been **submitted(committed)** by more than half of the machines in the cluster, that is, whether **data synchronization has been completed,**

- The Leader server needs to ensure that all Follower servers can receive each transaction Proposal and correctly apply all submitted transaction proposals to the in-memory data. After the Follower server synchronizes all its unsynchronized transaction proposals from the Leader server and successfully persists them to the local database, the Leader server adds the Follower server to the real list of available followers.

4. 总结

ZAB 协议是 CAP 理论中 CP 的典型实现，其崩溃恢复阶段涉及到的 Leader 节点选举过程和数据同步选举完成后的数据同步过程都是对外不提供服务的，就是为了保证数据的强一致性牺牲了可用性。

8.9 How does Mysql achieve consistency

https://blog.csdn.net/xmh594603296/article/details/82461222
https://segmentfault.com/a/1190000039139809

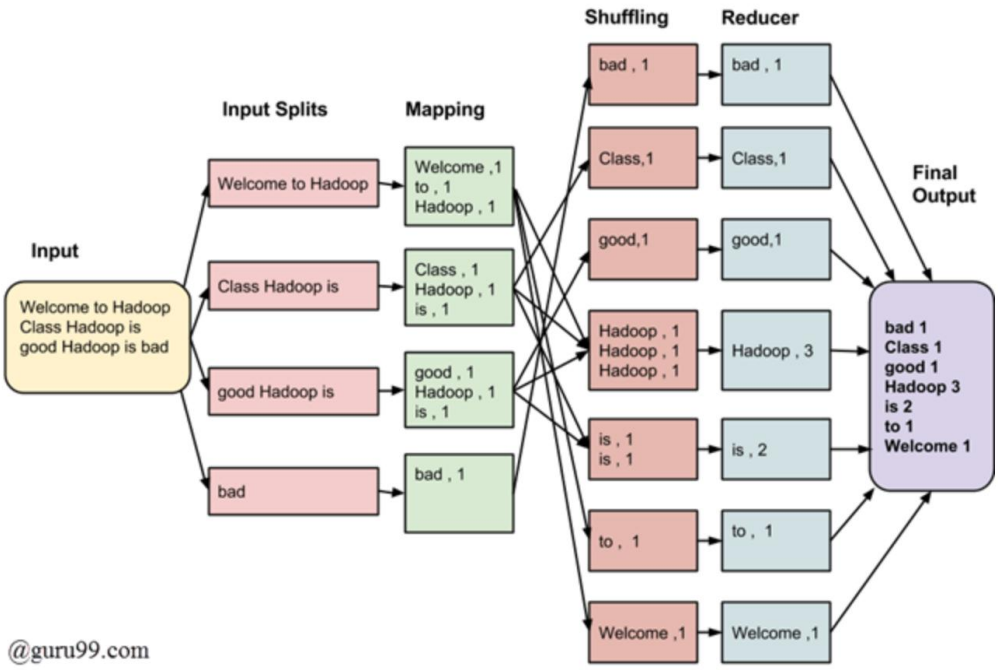
Or see ACID in DATABASE

8.10 Qurom to make consistency

9. what is map reduce

https://www.guru99.com/introduction-to-mapreduce.html

MapReduce program work in two phases, namely, Map and Reduce. Map tasks deal with splitting and mapping of data while Reduce tasks shuffle and reduce the data. These processes are handled by several nodes



10.what is rate limiter

https://medium.com/geekculture/system-design-basics-rate-limiter-351c09a57d14

Before rate limiting this perfect solution, How can we handle so many requests:

1 vertical scale, increase the capacity of the server

2 use rpc protocol instead of http ,
rpc can handle requests in asynchronous way

3 message compression and decompression

4 number of client connection; like chat system don't break the connection. Keep connection alive for a long time

5 Pull/Push Hybrid: Celebrity Post Things Like Instagram.Clients positively pull something from Celebrities. Do not use push because celebrity might has millions of followers

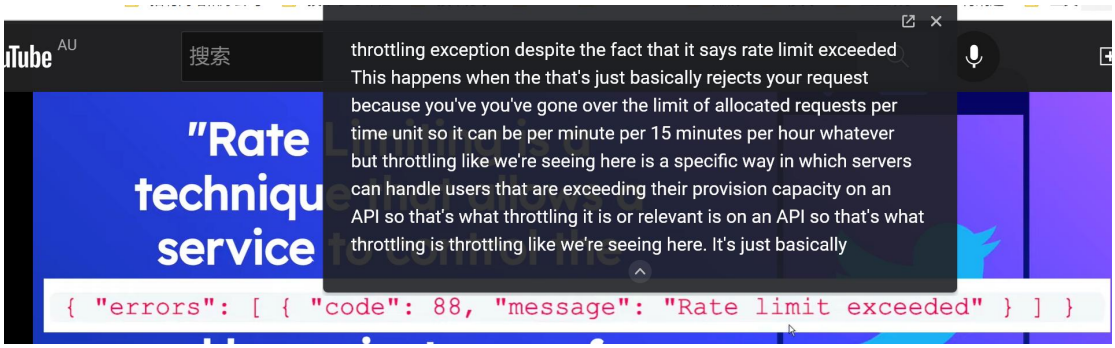
6 Graceful degration:

What is Rate Limiter?

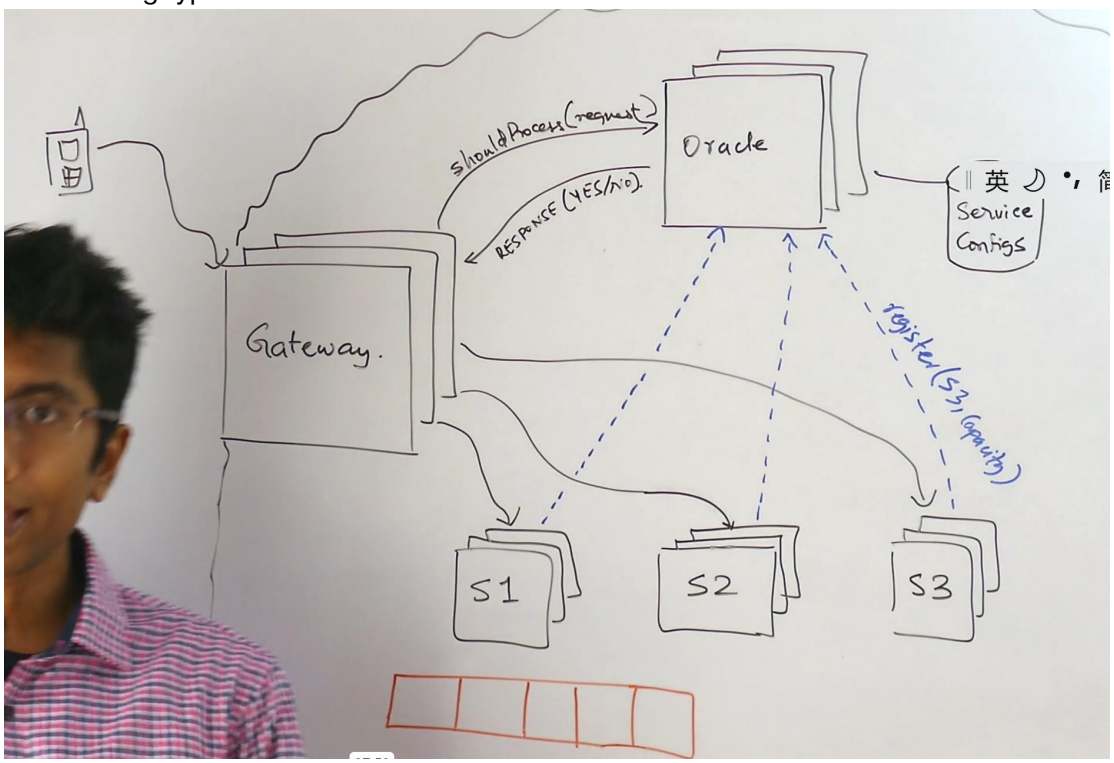
Rate limiting refers to preventing the frequency of an operation from exceeding a defined limit. In large-scale systems, rate limiting is commonly used to protect underlying services and resources. Rate limiting is generally used as a defensive mechanism in distributed systems, so that shared resources can maintain availability.

Rate limiting protects your APIs from unintended or malicious overuse by limiting the number of requests that can reach your API in a given period of time. Without rate limiting any user can bombard your server with requests leading to spikes that starves other users.

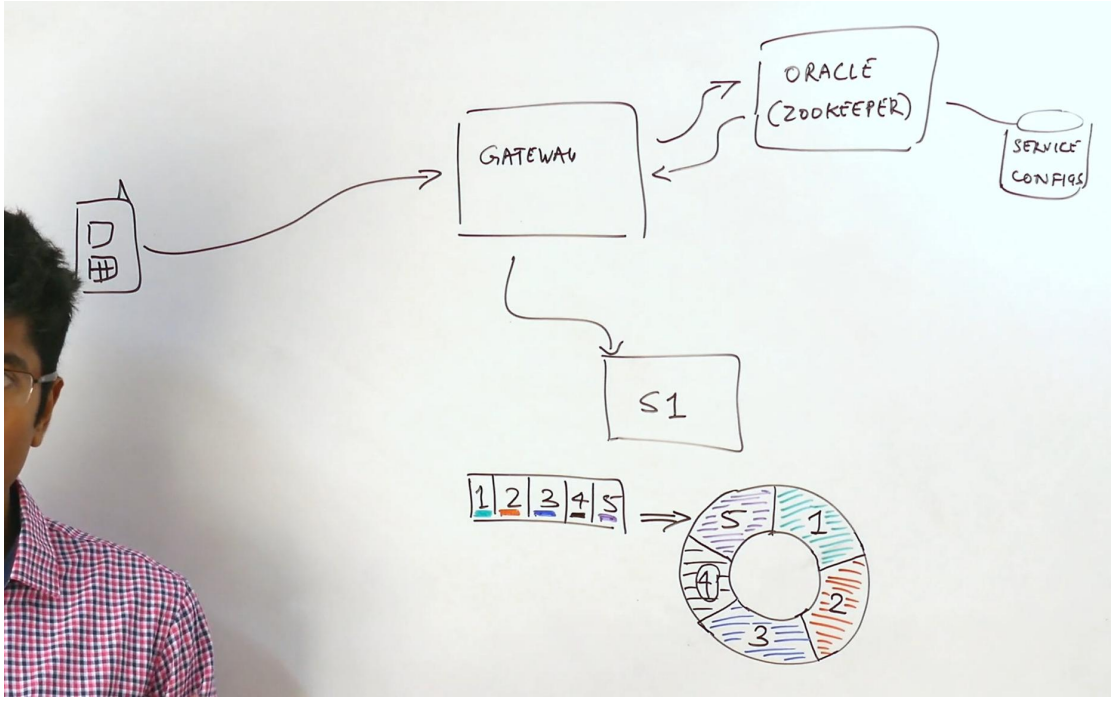
This is a typical throttling from face book



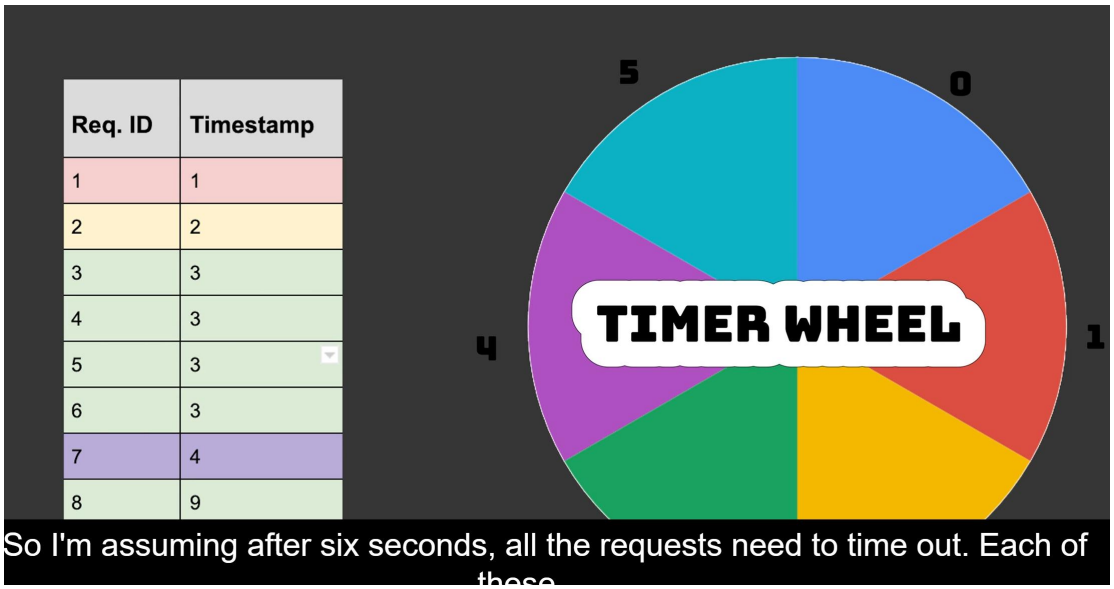
Rate limiting typical structure:



Service1,2,3 register to oracle, similar to heartbeat or services register LB.In fact, oracle could be load balancer and heartbeat monitor(zookeeper??)



Use time wheel ,size of the wheel is the time out of the request



The ways in which rate limiting can be implemented in a distributed environment.

The Oracle and the Timer Wheel

If our services are overloaded with requests then the time required to send a response will increase resulting in a bad user experience.

If the number of requests keeps on increasing then our servers might crash due to Out of Memory exceptions.

We want to avoid cascading failure.

A cascading failure is a failure that grows over time due to positive feedback. It can occur when one service of a distributed system fails, increasing the probability that other services fail. Let's understand this with an example.

We have two servers A and B. Both can handle at most 500 requests.

Currently, A is serving 400 requests and B is serving 300 requests.

Now suddenly B is getting 600 requests. It cannot handle that many requests and crashes. Now we need to route all the requests to the other server i.e., A. But A cannot handle 400 + 600 requests so it also crashes. This is known as cascading failure.

Timer Wheel

In a timer wheel,

Size of the wheel (Number of buckets) = Timeout of the incoming request

Each bucket is numbered from 0 to Timeout -1

Each bucket can store a limited number of requests

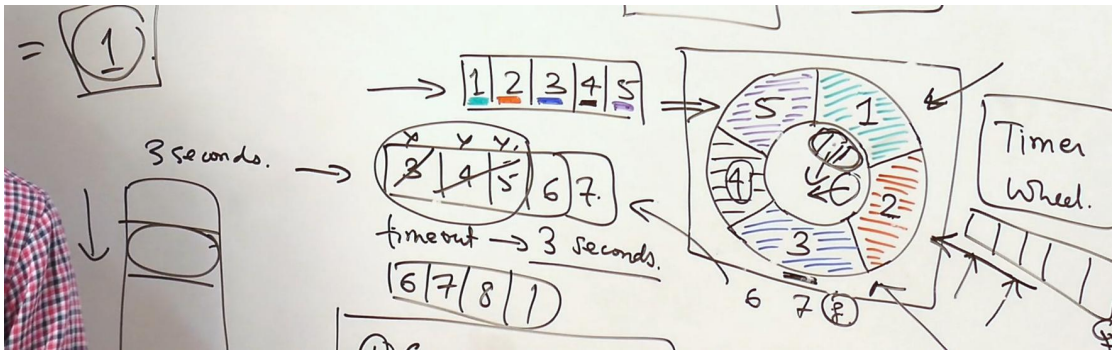
Every time we add a request to the bucket number Time % Number of buckets

Our system can pull requests from this queue one by one.

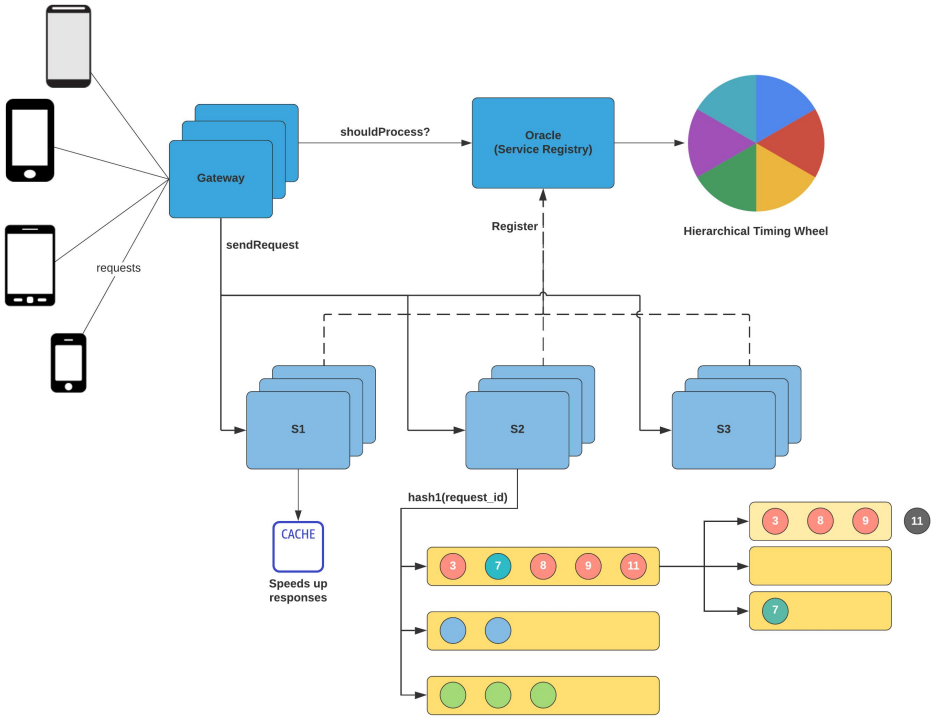
Before inserting a new request into a bucket it deletes all the existing requests (These requests have not been processed by the system).

Note: If there are a lot of slots then the distance between any two requests can be very large. In such cases, we will be wasting a lot of system resources and time. To tackle this issue we can use Hierarchal Timer Wheel.

Well now we have rate-limited external requests but our internal services are also talking to each other. One service sending a lot of requests to another service might cause the other service to fail. So let's discuss how we are going to solve this issue in the course video!!



Architecture:



11.Polling and web socket

(http polling)

Polling means server waits for a request from client and then gives back a response.(used when data updated not very frequently)

(web socket)
Streaming means server proactively push data to the clients . so low latency for streaming. So clients can continuously get new message,without having to repeatedly request them from server(used when data updated very frequently)

12.SQL and NOSQL, scenarios to choose

> ## 如何选择
>
> ### 适合使用SQL开发的项目：
>
> - 可以预先定义逻辑相关的离散数据的需求
> - 数据一致性与完整性是必要的
> - 具有良好的开发者经验和技术支持的标准的成熟技术
>
> ### 适合使用NoSQL开发的项目：
>
> - 不相关，不确定和逐步发展的数据需求
> - 更简单或者更宽松的能够快速开始编程的项目
> - 速度和可扩展性至关重要的，需要对大数据库有性能要求
> - 需要使用 CouchDB但因为数据改变太频繁而占满内存
>
> SQL是精确的，它最适合于具有精确标准的定义明确的项目，典型的使用场景是

Sql sharding is more difficult
Nosql is automatically sharded

Sql 强一致性
Nosql 最终一致性

When to choose:
Why it is more faster?
https://zhuanlan.zhihu.com/p/63371253

13.what is consistent hashing, usage

14.Long-Polling vs WebSockets vs Server-Sent Events

https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/6426df6a209d28c9a2d0d330

15.Distributed message Queue

https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/642d84748c4aa2edbd1287c7

16. caching

https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/64223e739f49016739e02a1b

17.batch process and stream process

https://zhuanlan.zhihu.com/p/551677449

Examples and explainations:
https://www.cnblogs.com/xueqiuqiu/articles/13054745.html

18.network protocols

19.Leader elections

选举机制
https://blog.csdn.net/reggergdsg/article/details/96329552

Leader 和 follower 的分工
Redis
https://blog.51cto.com/u_13259/6676424
zookeeper

> 2 . 节点读写服务分工
>
> 1.ZooKeeper 集群的所有机器通过一个 Leader 选举过程来选定一台被称为『Leader』的机器，Leader服务器为客户端提供读和写服务。
>
> 2.Follower 和 Observer 都能提供读服务，不能提供写服务。两者唯一的区别在于，Observer机器不参与 Leader 选举过程，也不参与写操作的『过半写成功』策略，因此 Observer 可以在不影响写性能的情况下提升集群的读性能。

V. System Design Examples

1. Design a two Player Online Chess Game

High-overview:
https://www.linkedin.com/pulse/system-design-interview-two-player-online-chess-game-shubham-pathak/

Low overview:
https://www.geeksforgeeks.org/design-a-chess-game/

2. Design payment system

https://newsletter.pragmaticengineer.com/p/designing-a-payment-system

3. How to design a huge file upload system including some query functions

1 先用 spring batch 导入数据库
https://stackoverflow.com/questions/47933599/spring-batch-for-large-files1billion-to-5billion-flat-file-data
2 上传到 s3,
AWS 建议超过 100M 的就分段上传
https://blog.csdn.net/qq_42026590/article/details/124152736

Or
https://codeantenna.com/a/k28l2VAmwN

3 数据库用 nosql，也可以用 sql 不过 sql 存储数据量太大，要考虑分区．显得麻烦,更何况查询并不复杂,所以用 Nosql

https://zhuanlan.zhihu.com/p/63371253

https://leetcode.com/discuss/interview-question/system-design/942087/System-Design%3A-Design-a-system-to-process-data-in-different-formats-from-different-sources

4. How to design a stock exchange system

https://www.youtube.com/watch?v=XuKs2kWH0mQ

Low-overview:
https://leetcode.com/discuss/interview-question/object-oriented-design/613466/Design-a-stock-trading-system/554578

5. how to design a real time stock trading system

https://www.youtube.com/watch?v=FE9A9xiG5JQ

6. how to design a park garage

https://www.youtube.com/watch?v=NtMvNh0WFVM&t=660s

7. how to design a IAM system

https://www.geeksforgeeks.org/architecture-of-identity-access-management-in-cloud-computing/

W. Big Data Processing

1. SpringBoot 实现 Mysql 千万级数据量导出并避免 OOM 的解决方案

https://zhuanlan.zhihu.com/p/421952313

https://www.yisu.com/zixun/535142.html
https://ifeve.com/mysql%E4%B8%AD%E4%BD%BF%E7%94%A8jdbc%E6%B5%81%E5%BC%8F%E6%9F%A5%E8%AF%A2%E9%81%BF%E5%85%8D%E6%95%B0%E6%8D%AE%E9%87%8F%E8%BF%87%E5%A4%A7%E5%AF%BC%E8%87%B4oom/

1 pagination

```java
@SneakyThrows
@Override
public void pageQuery() {
        @Cleanup Connection conn = dataSource.getConnection();
        @Cleanup Statement stmt = conn.createStatement();
        long start = System.currentTimeMillis();
        long offset = 0;
        int size = 100;
        while (true) {
                String sql = String.format("SELECT COLUMN_A, COLUMN_B, COLUMN_C FROM YOU_TABLE LIMIT %s, %s", offset, size);
                @Cleanup ResultSet rs = stmt.executeQuery(sql);
                long count = loopResultSet(rs);
                if (count == 0) break;
                offset += size;
        }
        log.info("分页查询耗时 :: {} ", System.currentTimeMillis() - start);
}
```

DISADVANTAGES:

上述方式比较简单，但是在不考虑 LIMIT 深分页优化情况下，线上数据库服务器就凉了，亦或者你能等个几天时间检索数据

2 Streaming(best)

Code 云服务 猿如意 statement.setfetchsize 搜索 会员中心 足迹 动态 消息 创作中心 发布

Statement setFetchSize(Integer.MIN_VALUE)

原创 genghaihua 于 2019-04-13 19:26:31 发布 2914 收藏 1 版权

分类专栏： 数据库

数据库 专栏收录该内容 0 订阅 14 篇文章 订阅专栏

当statement设置以下属性时，采用的是流数据接收方式，每次只从服务器接收部份数据，直到所有数据处理完毕，不会发生JVM OOM。

```java
setResultSetType(ResultSet.TYPE_FORWARD_ONLY);
setFetchSize(Integer.MIN_VALUE);
```

流式查询有一点需要注意：必须先读取（或关闭）结果集中的所有行，然后才能对连接发出任何其他查询，否则将引发异常

使用流式查询，则要保持对产生结果集的语句所引用的表的并发访问，因为其 查询会独占连接，所以必须尽快处理

```
@SneakyThrows
public void streamQuery() {
        @Cleanup Connection conn = dataSource.getConnection();
        @Cleanup Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONL
Y);
        stmt.setFetchSize(Integer.MIN_VALUE);
        long start = System.currentTimeMillis();
        @Cleanup ResultSet rs = stmt.executeQuery("SELECT COLUMN_A, COLUMN_B, COLUMN_C FROM YOU_TABLE");
        loopResultSet(rs);
        log.info("      流式查询耗时 :: {} ", (System.currentTimeMillis() - start) / 1000);
}
```

流式查询库表数据量 500w 单次调用时间消耗：≈ 6s

https://www.bmabk.com/index.php/post/5317.html
https://blog.csdn.net/10km/article/details/50404694

切换到 Markdown 编辑器下 ResultSet 会使用 RowDataStatic 实例，在生成 RowDataStatic 对象时就会把 ResultSet 中所有记录读到内存里，之后通过 next() 再一条条从内存中读

当采用流式处理时，ResultSet 使用的是 RowDataDynamic 对象，而这个对象 next() 每次调用都会发起 IO 读取单行数据

总结来说就是，默认的 RowDataStatic 读取全部数据到客户端内存中，也就是我们的 JVM；RowDataCursor 一次读取 fetchSize 行，消费完成再发起请求调用；RowDataDynamic 每次 IO 调用读取一条数据

3 Cursor query

需要在数据库连接信息里拼接 &useCursorFetch=true。其次设置 Statement 每次读取数据数量，比如一次读取 1000

```
@SneakyThrows
public void cursorQuery() {
        @Cleanup Connection conn = dataSource.getConnection();
        @Cleanup Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);
        stmt.setFetchSize(1000);
        long start = System.currentTimeMillis();
        @Cleanup ResultSet rs = stmt.executeQuery("SELECT COLUMN_A, COLUMN_B, COLUMN_C FROM YOU_TABLE");
        loopResultSet(rs);
        log.info("游标查询耗时 :: {} ", (System.currentTimeMillis() - start) / 1000);
}
```

游标查询库表数据量 500w 单次调用时间消耗：≈ 18s

2. all questions

https://zq99299.github.io/note-book/back-end-storage/03/

3. 怎样进行上千万数据的插入和更新

https://www.51cto.com/article/580903.html

4. differences between batch processing and stream processing

https://www.geeksforgeeks.org/difference-between-batch-processing-and-stream-processing/

4 Find duplicates in large file like 1T

假设有一个非常大的文件，包含了一亿个数字，无法全部加载到内存中进行排序。我们想要对这些数字进行去重和排序。

1. **分割阶段**：
   - 将大文件分割为多个小文件，每个小文件只包含一部分数据。
   - 例如，将一亿个数字分成 100 个小文件，每个文件包含 100 万个数字。
2. **内部排序阶段**：
   - 对每个小文件进行内部排序。由于每个小文件可以被加载到内存中，我们可以使用常见的排序算法（如归并排序、快速排序）进行排序。
   - 对每个小文件排序后，得到了排序好的多个小文件。
3. **合并阶段**：
   - 使用归并排序的合并过程来将排序好的小文件合并成一个大的有序文件。这个过程也要保证去除重复的数字。
   - 对于合并过程，可以使用优先队列（最小堆）来维护每个小文件的当前最小元素，然后依次选取最小的元素进行合并。
4. **结果输出**：
   - 合并完成后，就得到了一个包含所有数据的有序文件，其中重复的数字已经被去除。

X. work style

1. TDD(Test-Driven-Design)

TDD is a software development process where you write tests before writing the code that implements the functionality. The cycle is often summarized as **Red-Green-Refactor**.

> **Red:** Write a test for a new feature or function. Since you haven't written the code yet, the test will fail. This is the "red" phase.
>
> **Green:** Write the minimum amount of code necessary to make the test pass. This is the "green" phase.

Refactor: Improve the code without changing its behavior. You clean up the code while ensuring all tests still pass.

This cycle ensures that your code is always tested and that you only write the code that's necessary.

2. DDD(Domain-Driven-Design)

DDD is an approach to **software development t**hat emphasizes collaboration between technical and domain experts to create software that accurately models complex business domains. The key idea is to align your software's structure and terminology with the domain it represents.
Imagine you're working on an e-commerce application. The domain involves concepts like Order, Product, Customer, Inventory, etc. In DDD, you would work closely with business stakeholders to understand these concepts and their relationships.

Summary

- **TDD** helps you **write reliable, well-tested code by starting w**ith tests and iterating through development.
- **DDD helps you design software that accurately represents the complex domain by focusing on the business logic and its alignment with real-world concepts.**