

自我介绍

我是3y，很高兴你能关注到我，并读到这篇自我介绍。

我目前在一线互联网公司搬砖，平时酷爱写博客。原创已经**400+**了，GitHub的star数也已经将近**20k**，关注**Java3y**公众号的同学也近**100K**

随着工龄的增加，自己在看文章的时，发现在网上搜到很多博客都写得不怎么样，只有少部分是不错的。很现实的是，解决大部分疑惑的问题都需要靠那「少部分」写得不错的文章才能得以解决。

当时就觉得：有一份「**好的学习资料**」是真的很重要，是可以节省很多很多的时间。

后来自己习惯于写博客，这几年也写了挺多，技术博客应该将近400篇了。大多数人对我的博客评价都是：**通俗易懂**

说实话，我在这个过程中还是蛮开心的，毕竟自己写的东西得到了他人的认可。

「面试」是所有程序员都得迈过去的一个「坎」。毕竟从学生时期需要经历「校招」，进职场之后是肯定需要考虑「跳槽」的，一个程序员的职业生涯都会经历数次的面试。

同样地，我也需要面试，即便是我写完的知识点，也会忘记，我也需要复习。

与其看别人的面试资料，还不如看我自己写的面试资料？

他们写的面试，有我写得好？多半数没有（自信脸）

于是《对线面试官》系列在这个背景之下就应运而生了。

假设我写这个系列，能达到下面的效果，我就非常非常开心了

当你需要准备去面试，你要是能在脑海里想起：“我记得我看有个叫做《对线面试官》的系列，写得不错的，我去翻来看看”。

愿景：《对线面试官》能成为Java程序员口口相传的面试系列

想请我喝咖啡☕ 可扫描下方二维码 ↴(金额多少不是事，最主要想证明下我的电子书是有价值的)

支付宝扫一扫，向我付款

无需加好友，扫二维码向我付钱



@稀土掘金技术社区

想添加我的个人微信吹牛逼围观我日常的朋友圈，拉进交流群的可以扫描下方的二维码 ↴



扫一扫上面的二维码图案，加我微信
@稀土掘金技术社区

00-面试前准备

01、简历

当我们有面试的打算的时候，就需要提前去更新自己的简历。比如说，我是打算9月面试的，那么我就需要在7月份就对自己的简历有所准备。

编写简历的过程实际上就是回顾自己所掌握知识的过程

对于开发程序员而言，简历最主要由三部分组成

- 个人简介
- 项目系统
- 专业技能



01、项目系统

项目系统需要回顾自己以前做了什么项目，挑自己熟悉的放在简历的前面

1、梳理系统的项目背景以及整个系统架构设计与运转流程

这个过程主要是大体回顾自己的项目。无论是哪场面试，只要问到项目，面试官总会提问：“要不你先来简单介绍下你所负责的这个项目吧”

这个答案实际上就是聊「梳理系统的项目背景以及整个系统架构设计与运转流程」

这点很重要，需要好好准备下，是面试官能否通过短时间了解你项目的关键。

2、梳理项目技术或业务上的亮点

这个过程实际上就是寻找项目的亮点，能够写在简历上的事项。面试官有很多时候看到一个项目，即便是听完你的描述，可能还是无从问起（很多时候，他本身就没跟你做过一样的东西，没有感同身受）

所以，我们需要挖掘自己的项目亮点，写在简历上，让面试官有问题可问（大多面试官还是比较乐意去了解这些“亮点”）

技术上的亮点：

- “我在处理数据的过程中，实现了数据的一致性和可靠性，做到了数据零丢失”
- “项目引入了规则引擎，其中解决了xxx的业务问题，使得工作效率极大提升”
- ...

业务上的亮点：

- “实现了业务隔离，不同类型之间的业务互不干扰，从原来的xxx提升到了xxx”
- “参考自某平台的xx功能，从零开始实现了业务，使得平台收益增加了xxx”
- ...

3、梳理项目还可提升的地方

在聊项目的时候，前两点是必问的，这时候体现自己牛逼就在能表述自己有思考。

经过项目的探讨之后，面试官可能会在里面掺杂各种的技术细节以及问你业务上的理解。完了之后，可能面试官还想问：“目前这项目还在运行中嘛，那你觉得还有什么可以优化的地方吗”

一个项目总不可能是十全十美的，总会有地方可以进行优化。无论是提高性能，还是提高工作效率，总会有的。

这时候如果能吹下自己对比过某某公司的同类型系统，借鉴了某某某优点，基于目前自身的业务觉得还有哪里可以继续优化

项目背景和
运转流程

技术或业务
上的亮点

可提升的地
方

到这里，项目就差不多了。总的来说，在准备编写简历的时候，就需要猜面试官可能会问什么问题，这样在回顾或者复习的时候就有所准备。等真正面到了，心里就不慌了

02、专业技能

这块内容我想吐槽很久的了

我会经常在群里或者网上的论坛上看到各种关于面试的问题：“Java实习生需要懂微服务吗”、“Java工作一年需要懂Netty吗”、“面试的过程中会问Netty吗？”.....

其实吧，真正面试的时候，你简历有什么内容，一般面试官就问什么。

如果问到了我简历上没写的，我要是知道简单的概念，我就告诉他对该知识点的理解，并告诉他这块内容我不太了解。如果是没看过，那就直接说不懂这块。这样处理，我认为一点问题都没有。

之前我有幸也担任过面试官，当时我的leader就直接跟我说：“最好还是问候候选者简历上写过的内容，可以稍微有一两个问题超出简历外，但不能多”

我们程序员可专注的领域有很多，每个人的特长都不一样（术业有专攻）。懂大数据的人，可能没你懂Spring。懂Spring的人，可能没你懂Netty

一定是针对简历上的专业内容进行复习

于是乎，在专业技能这块上，写上的东西最好是自己有理解的，了解其设计原理以及思想的（最好还看过部分核心功能的源码）

不要求你写很多专业技能上去，只求被问到了这些技术栈，你都能有自己的看法以及理解。

举个例子，我之前工作就没用过SpringCloud/Dubbo/Netty/Docker等等，这些我都不写。问我到我，我就说这块不会。

但我之前看得比较多以及工作中用得比较多的，比如说：Redis、Flink、Kafka、HBase、SSM等等，这些我都写上去。

如果遇到一个“大聪明”面试官，一直问你简历上没写过的技术栈，那这种公司不去也罢（面试官格局不太行）。

每一个主流的技术栈的技术点都大有可挖，写上几个过后，面试官还找不到问题可问，那也挺说不过去的。

专业技能写自己熟悉的，面试前就针对简历上所写的技术栈重点复习

03、简历常见QA

1、简历应该有多少页？

我们的简历不应该超过**2页**，禁止简历有3页或者3页以上的情况。简历这东西不是写得越多就越好的，要把重点给突出来。

我在最开始投简历的时候，把简历写了3页，后来被热心的同学给提醒一下，最好简历控制在两页之内。我当时曾经收到过的

2、要不要放照片？

这个无所谓的，我个人是不放照片的。像我用 markdown 写的，也不方便放简历。如果你用的是别的模板，你放照片也是OK的。

3、项目经历没有怎么办？

很多同学可能没写过项目，就开始写简历了。我个人认为，像Java开发这种，在简历上是一定要出现项目的。将心比心，如果你是面试官，应聘者没有项目，那作为面试官的你该问什么问题呢？

所以，一定要有自己的项目（如果真没自己的，也得去网上找一个出来，死扣这个项目）

4、简历格式是Word还是PDF？

显然，肯定是 PDF 格式的。word格式在不同的环境中排版很可能不一致，强烈建议将自己的简历转成 PDF 。

5、其余要注意的？

1. 文件命名最好根据对方所要求来命名，比如说：姓名-学校-学历-岗位-性别。如果对方没有要求，我建议文件的命名应该要有以下信息：姓名-学校-求职岗位-联系方式（邮件地址/手机号）
2. 简历上不要写多余的东西，校园生活这些我们不关心的哈。
3. 专业术语最好规范名称，该大写的大写，该小写的小写。
4. 简历格式正常统一，用其他模板的同学该对齐就对齐。

04、面试前总结

1、多花点时间在自己的简历上，一份好的简历是面试的敲门砖，简历代表着你

2、面向自己简历复习，不要盲目



微博 @吃花椒的喵酱

有的同学看完可能还没什么感觉，别说了，**两份简历模板**在公众号下回复「简历」即可直接获取。

第一时间获取**BATJTMD一线互联网大厂**最新的面试资料以及内推机会关注公众号「**对线面试官**」



01-Java基础

01、注解

面试官：来讲讲什么是注解吧

候选者：注解在我的理解下：就是代码中的**特殊标记**。这些标记可以在编译、类加载、运行时被读取，并执行相对应的处理。

面试官：你这讲得有点抽象，你先说说你在开发中有没有用到注解吧。

候选者：注解其实在开发中是非常常见的，比如我们在使用各种框架时（像我们Java程序员接触最多的还是Spring框架一套），就会用到非常多的注解，`@Controller / @Param / @Select` 等等。一些项目也用到**lombok**的注解，`@Slf4j / @Data` 等等

候选者：除了框架实现的注解，Java原生也有`@Overried`、`@Deprecated`、`@FunctionalInterface`等基本注解。**Java原生的基本注解大多数用于「标记」和「检查」**

候选者：原生Java除了这些提供基本注解之外，还有一种叫做**元Annotation**（元注解），所谓的元Annotation就是用来修饰注解的。常用的元Annotation有`@Retention` 和`@Target`。`@Retention`注解可以简单理解为设置注解的生命周期，而`@Target`表示这个注解可以修饰哪些地方（比如方法、还是成员变量、还是包等等）

面试官：嗯，听得出来你在工作中是真的用到注解的，这些可能是Spring、Mybatis等框架原生提供的，可能是Java原生的。**那你自己写过注解吗？**

候选者：嗯，写过的。背景是这样的：我司有个监控告警系统，对外提供了客户端供我们自己使用。监控一般的指标就是QPS、RT和错误嘛。

候选者：原生的客户端需要在代码里指定上报，这会导致这种监控的代码会跟业务代码混合，比较恶心。

```
public void send(String userName) {  
    try {  
        // qps 上报  
        qps(params);  
    } catch (Exception e) {  
        log.error("上报失败, error: " + e.getMessage());  
    }  
}
```

```
long startTime = System.currentTimeMillis();

// 构建上下文(模拟业务代码)
ProcessContext processContext = new ProcessContext();
UserModel userModel = new UserModel();
userModel.setAge("22");
userModel.setName(userName);
//...

// rt 上报
long endTime = System.currentTimeMillis();
rt(endTime - startTime);
} catch (Exception e) {

// 出错上报
error(params);
}

}
```

候选者: 其实这种基础的监控信息，显然都可以通过**AOP切面**的方式去处理掉（可以看到都是方法级的）。而再用注解这个载体配置相关的信息，配合AOP解析就会比较优雅

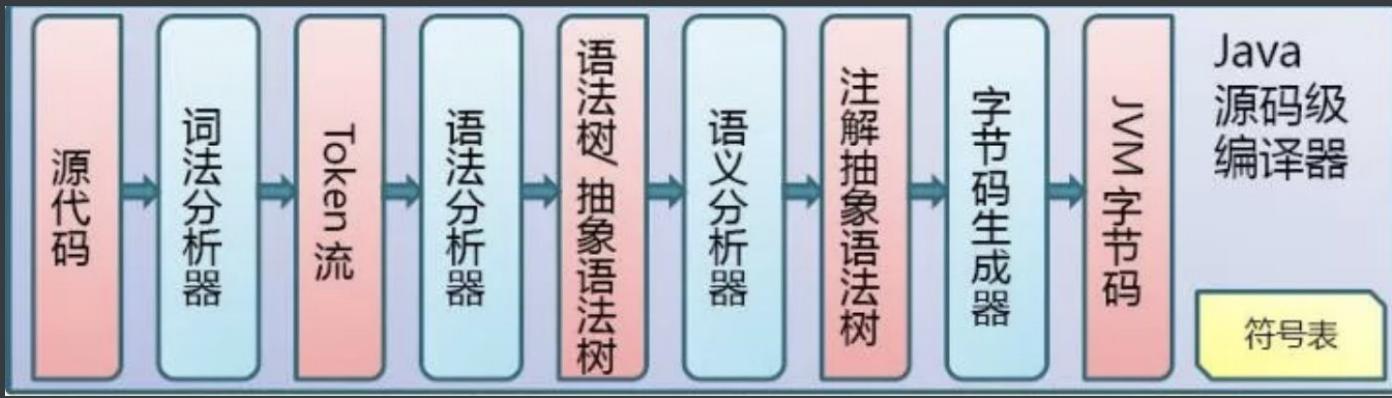
候选者: 接下来讲讲我是怎么做的吧。

候选者: 要写自定义的注解，首先考虑**我们是在什么时候解析这个注解**。这就需要用到前面所说的@Retention注解，这个注解会修饰我们自定义注解生命周期。

候选者: @Retention注解传入的是RetentionPolicy枚举，该枚举有三个常量，分别是 SOURCE、CLASS和RUNTIME

候选者: SOURCE代表着注解仅保留在源级别中，并由编译器忽略。CLASS代表着注解在编译时由编译器保留，但Java虚拟机（JVM）会忽略。RUNTIME代表着标记的注解会由JVM保留，因此运行时环境可以使用它。

候选者: 理解这块就得了解从.java文件到.class文件再到class被jvm加载的过程了。下面的图描述着从.java文件到编译为.class文件的过程



候选者：从上面的图可以发现有个「注解抽象语法树」，这里其实就会去解析注解，然后做处理的逻辑。

候选者：所以重点来了，**如果你想要在编译期间处理注解相关的逻辑，你需要继承 AbstractProcessor 并实现process方法**。比如可以看到lombok就用AnnotationProcessor继承了AbstractProcessor。

候选者：一般来说，只要自定义的注解中@Retention注解设置为SOURCE和CLASS这两个级别，那么就需要继承并实现（因为SOURCE和CLASS这两个级别等加载到jvm的时候，注解就被抹除了）

候选者：从这里又引申出：lombok的实现原理就是在这（为什么使用了个@Data这样的注解就能有set/get等方法了，就是在这里加上去的）

面试官：嗯，你是还有点东西哦

候选者：一般来说，我们自己定义的注解都是**RUNTIME级别的**，因为大多数情况我们是根据运行时环境去做一些处理。

候选者：我们现实在开发的过程中写自定义注解需要配合反射来使用（因为反射是Java获取运行时的信息的重要手段）。

候选者：所以，我当时就用了自定义注解，在Spring AOP的逻辑处理中，判断是否带有自定义注解，如果有则将监控的逻辑写在方法的前后。这样，只要在方法上加上我的注解，那就可对方法监控的效果（RT、QPS、ERROR）

```

@Around("@annotation(com.sanwai.service.openapi.monitor.Monitor)")
public Object antispan(ProceedingJoinPoint pjp) throws Throwable {
  ...
}
  
```

```
String functionName = pjp.getSignature().getName();
Map<String, String> tags = new HashMap<>();

logger.info(functionName);

tags.put("functionName", functionName);
tags.put("flag", "done");

monitor.sum(functionName, "start", 1);

//方法执行开始时间
long startTime = System.currentTimeMillis();

Object o = null;
try {
    o = pjp.proceed();
} catch (Exception e) {
    //方法执行结束时间
    long endTime = System.currentTimeMillis();

    tags.put("flag", "fail");
    monitor.avg("rt", tags, endTime - startTime);

    monitor.sum(functionName, "fail", 1);
    throw e;
}

//方法执行结束时间
long endTime = System.currentTimeMillis();

monitor.avg("rt", tags, endTime - startTime);

if (null != o) {
    monitor.sum(functionName, "done", 1);
}
```

```
    return o;  
}
```

面试官：嗯，总体来看，你对注解这块基础还是扎实的



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zhongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

02、泛型

面试官：这次咱们就来聊聊泛型呗？**你对泛型有多少了解？**

候选者：在Java中的泛型简单来说就是：在创建对象或调用方法的时候才明确下具体的类型

候选者：使用泛型的好处就是代码更加简洁（不再需要强制转换），程序更加健壮（在编译期间没有警告，在运行期就不会出现ClassCastException异常）

面试官：平时在工作中用得多吗？

候选者: 在操作集合的时候，还是很多的，毕竟方便啊。

```
List<String> lists = new ArrayList<>();
lists.add("对线面试官");
```

候选者: 如果是其他场景的话，那就是在写「基础组件」的时候了。

面试官: 来，说说背景吧，你是怎么写的。

候选者: 再明确一下泛型就是「在创建对象或调用方法的时候才明确下具体的类型」，而组件为了做到足够的通用性，是不知道「用户」传入什么类型参数进来的，所以在这种情况下用泛型就是很好的实践。

候选者: 这块可以参考 SpringData JPA 的 JpaRepository 写法。

```
public interface JpaRepository<T, ID> extends
PagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> {

    List<T> findAll();

    List<T> findAll(Sort sort);

    List<T> findAllById(Iterable<ID> ids);

    <S extends T> List<S> saveAll(Iterable<S> entities);

    void flush();

    <S extends T> S saveAndFlush(S entity);

    void deleteInBatch(Iterable<T> entities);

    void deleteAllInBatch();

    T getOne(ID id);
```

```
@Override  
<S extends T> List<S> findAll(Example<S> example);  
  
@Override  
<S extends T> List<S> findAll(Example<S> example, Sort sort);  
}
```

候选者: 要写组件, 还是离不开Java反射机制 (能够从运行时获取信息), 所以一般组件是泛型+反射来实现的。

候选者: 回到我所讲的组件吧, 背景是这样的: 我这边有个需求, 需要根据某些字段进行聚合。

候选者: 换到 SQL 其实就是 `select sum(column1),sum(column2) from table group by field1,field2`

候选者: 需要 `sum` 和 `group by` 的列肯定是由业务方自己传入, 而SQL的表 其实就是我们的POJO (传入的字段也肯定是 POJO 的属性)

候选者: 单个业务实际可以在参数上写死POJO, 但为了做得更加通用, 我把入参设置为泛型

候选者: 拿到参数后, 通过反射获取其字段具体的值, 做累加就好了。

```
// 传入 需要group by 和 sum 的字段名  
public void cacheMap(List<String> groupByKeys, List<String> sumValues) {  
    this.groupByKeys = groupByKeys;  
    this.sumValues = sumValues;  
}  
  
private void excute(T e) {  
  
    // 从pojo 取出需要group by 的字段 list  
    List<Object> key = buildPrimaryKey(e);  
  
    // primaryMap 是存储结果的Map
```

```
T value = primaryMap.get(key);

// 如果从存储结果找到有相应记录
if (value != null) {
    for (String elem : sumValues) {
        // 反射获取对应的字段，做累加处理
        Field field = getDeclaredField(elem, e);
        if (field.get(e) instanceof Integer) {
            field.set(value, (Integer) field.get(e) + (Integer)
field.get(value));
        } else if (field.get(e) instanceof Long) {
            field.set(value, (Long) field.get(e) + (Long) field.get(value));
        } else {
            throw new RuntimeException("类型异常，请处理异常");
        }
    }
}

// 处理时间记录
Field field = getDeclaredField("updated", value);
if (null != field) {
    field.set(value, DateTimeUtils.getCurrentTime());
}
} else {
    // group by 字段 第一次进来
    try {
        primaryMap.put(key, Tclone(e));
        createdMap.put(key, DateTimeUtils.getCurrentTime());
    } catch (Exception ex) {
        log.info("first put value error {}" , e);
    }
}
}
```

候选者：理解了泛型的作用之后，再去审视自己代码时，就可以判断是否需要用到泛型了。

面试官：嗯，可以的，总体来看有自己的见解。



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



03、JavaNIO

面试官：这次咱们就来聊聊Java 的NIO呗？你对NIO有多少了解？

候选者：嗯，我对Java NIO还是有一定的了解的，NIO是JDK 1.4 开始有的，其目的是为了提高速度。NIO翻译成 no-blocking io 或者 new io 都无所谓啦，反正都说得通

面试官：你先来讲讲NIO和传统IO有什么区别吧

候选者：传统IO是一次一个字节地处理数据，NIO是以块（缓冲区）的形式处理数据。最主要的是，NIO可以实现非阻塞，而传统IO只能是阻塞的。

候选者：IO的实际场景是文件IO和网络IO，NIO在网络IO场景下提升就尤其明显了。

候选者：在Java NIO有三个核心部分组成。分别是Buffer（缓冲区）、Channel（管道）以及Selector（选择器）

候选者：可以简单的理解为：Buffer是存储数据的地方，Channel是运输数据的载体，而Selector用于检查多个Channel的状态变更情况，

候选者：我曾经写过一个NIO Demo，面试官可以看看。

```
public class NoBlockServer {
```

```
public static void main(String[] args) throws IOException {

    // 1. 获取通道
    ServerSocketChannel server = ServerSocketChannel.open();

    // 2. 切换成非阻塞模式
    server.configureBlocking(false);

    // 3. 绑定连接
    server.bind(new InetSocketAddress(6666));

    // 4. 获取选择器
    Selector selector = Selector.open();

    // 4.1 将通道注册到选择器上，指定接收“监听通道”事件
    server.register(selector, SelectionKey.OP_ACCEPT);

    // 5. 轮训地获取选择器上已“就绪”的事件--->只要select()>0，说明已就绪
    while (selector.select() > 0) {
        // 6. 获取当前选择器所有注册的“选择键”（已就绪的监听事件）
        Iterator<SelectionKey> iterator =
        selector.selectedKeys().iterator();

        // 7. 获取已“就绪”的事件，（不同的事件做不同的事）
        while (iterator.hasNext()) {

            SelectionKey selectionKey = iterator.next();

            // 接收事件就绪
            if (selectionKey.isAcceptable()) {

                // 8. 获取客户端的链接
                SocketChannel client = server.accept();

                // 8.1 切换成非阻塞状态
                client.configureBlocking(false);
            }
        }
    }
}
```

```
// 8.2 注册到选择器上-->拿到客户端的连接为了读取通道的数据  
(监听读就绪事件)  
client.register(selector, SelectionKey.OP_READ);  
  
} else if (selectionKey.isReadable()) { // 读事件就绪  
  
    // 9. 获取当前选择器读就绪状态的通道  
    SocketChannel client = (SocketChannel)  
selectionKey.channel();  
  
    // 9.1读取数据  
    ByteBuffer buffer = ByteBuffer.allocate(1024);  
  
    // 9.2得到文件通道，将客户端传递过来的图片写到本地项目下(写模  
式、没有则创建)  
    FileChannel outChannel =  
FileChannel.open(Paths.get("2.png"), StandardOpenOption.WRITE,  
StandardOpenOption.CREATE);  
  
    while (client.read(buffer) > 0) {  
        // 在读之前都要切换成读模式  
        buffer.flip();  
  
        outChannel.write(buffer);  
  
        // 读完切换成写模式，能让管道继续读取文件的数据  
        buffer.clear();  
    }  
}  
// 10. 取消选择键(已经处理过的事件，就应该取消掉了)  
iterator.remove();  
}  
}  
}
```

```
}
```

```
public class NoBlockClient {

    public static void main(String[] args) throws IOException {

        // 1. 获取通道
        SocketChannel socketChannel = SocketChannel.open(new
InetSocketAddress("127.0.0.1", 6666));

        // 1.1切换成非阻塞模式
        socketChannel.configureBlocking(false);

        // 1.2获取选择器
        Selector selector = Selector.open();

        // 1.3将通道注册到选择器中， 获取服务端返回的数据
        socketChannel.register(selector, SelectionKey.OP_READ);

        // 2. 发送一张图片给服务端吧
        FileChannel fileChannel =
FileChannel.open(Paths.get("X:\\\\Users\\\\ozc\\\\Desktop\\\\面试造火箭\\\\1.png"),
StandardOpenOption.READ);

        // 3.要使用NIO，有了Channel，就必然要有Buffer， Buffer是与数据打交道的呢
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        // 4.读取本地文件(图片)，发送到服务器
        while (fileChannel.read(buffer) != -1) {

            // 在读之前都要切换成读模式
            buffer.flip();
```

```
socketChannel.write(buffer);

// 读完切换成写模式，能让管道继续读取文件的数据
buffer.clear();
}

// 5. 轮训地获取选择器上已“就绪”的事件--->只要select()>0, 说明已就绪
while (selector.select() > 0) {
    // 6. 获取当前选择器所有注册的“选择键”(已就绪的监听事件)
    Iterator<SelectionKey> iterator =
    selector.selectedKeys().iterator();

    // 7. 获取已“就绪”的事件, (不同的事件做不同的事)
    while (iterator.hasNext()) {

        SelectionKey selectionKey = iterator.next();

        // 8. 读事件就绪
        if (selectionKey.isReadable()) {

            // 8.1得到对应的通道
            SocketChannel channel = (SocketChannel)
selectionKey.channel();

            ByteBuffer responseBuffer =
            ByteBuffer.allocate(1024);

            // 9. 知道服务端要返回响应的数据给客户端, 客户端在这里接收
            int readBytes = channel.read(responseBuffer);

            if (readBytes > 0) {
                // 切换读模式
                responseBuffer.flip();
                System.out.println(new
String(responseBuffer.array(), 0, readBytes));
            }
        }
    }
}
```

```
        }
    }

    // 10. 取消选择键(已经处理过的事件，就应该取消掉了)
    iterator.remove();
}

}

}
```

面试官：这都是些API相关的知识，能看得出来你有一定的基础

面试官：你知道IO模型有几种吗

候选者：在Unix下IO模型分别有：阻塞IO、非阻塞IO、IO复用、信号驱动以及异步I/O。在开发中碰得最多的就是阻塞IO、非阻塞IO以及IO复用。

面试官：嗯，来重点讲讲IO复用模型吧

候选者：我就以Linux系统为例好了，我们都知道Linux对文件的操作实际上就是通过文件描述符(fd)

候选者：IO复用模型指的就是：通过一个进程监听多个文件描述符，一旦某个文件描述符准备就绪，就去通知程序做相对应的处理

候选者：这种以通知的方式，优势并不是对于单个连接能处理得更快，而是在于它能处理更多的连接。

候选者：在Linux下IO复用模型用的函数有select/poll和epoll

面试官：那你来讲讲这select和epoll函数的区别呗？

候选者：嗯，先说select吧。

候选者：select函数它支持最大的连接数是1024或2048，因为在select函数下要传入fd_set参数，这个fd_set的大小要么1024或2048（其实就看操作系统的位数）

候选者: fd_set就是bitmap的数据结构，可以简单理解为只要位为0，那说明还没数据到缓冲区，只要位为1，那说明数据已经到缓冲区。

候选者: 而select函数做的就是每次将fd_set遍历，判断标志位有没有发现变化，如果有变化则通知程序做中断处理。

候选者: epoll 是在Linux2.6内核正式提出，完善了select的一些缺点。

候选者: 它定义了epoll_event结构体来处理，不存在最大连接数的限制。

候选者: 并且它不像select函数每次把所有的文件描述符(fd)都遍历，简单理解就是epoll把就绪的文件描述符(fd)专门维护了一块空间，每次从就绪列表里边拿就好了，不再进行对所有文件描述符(fd)进行遍历。



面试官: �恩。你知道什么叫做零拷贝吗？

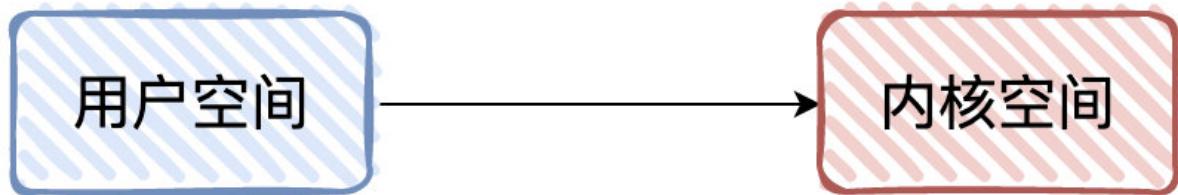
候选者: 知道的。我们以读操作为例，假设用户程序发起一次读请求。

候选者: 其实会调用read相关的「系统函数」，然后会从用户态切换到内核态，随后CPU会告诉DMA去磁盘把数据拷贝到内核空间。

候选者: 等到「内核缓冲区」真正有数据之后，CPU会把「内核缓存区」数据拷贝到「用户缓冲区」，最终用户程序才能获取到。

候选者: 稍微解释一下：为了保证内核的安全，操心系统将虚拟空间划分为「用户空间」和「内核空间」，所以在读系统数据的时候会有状态切换

为了 安全



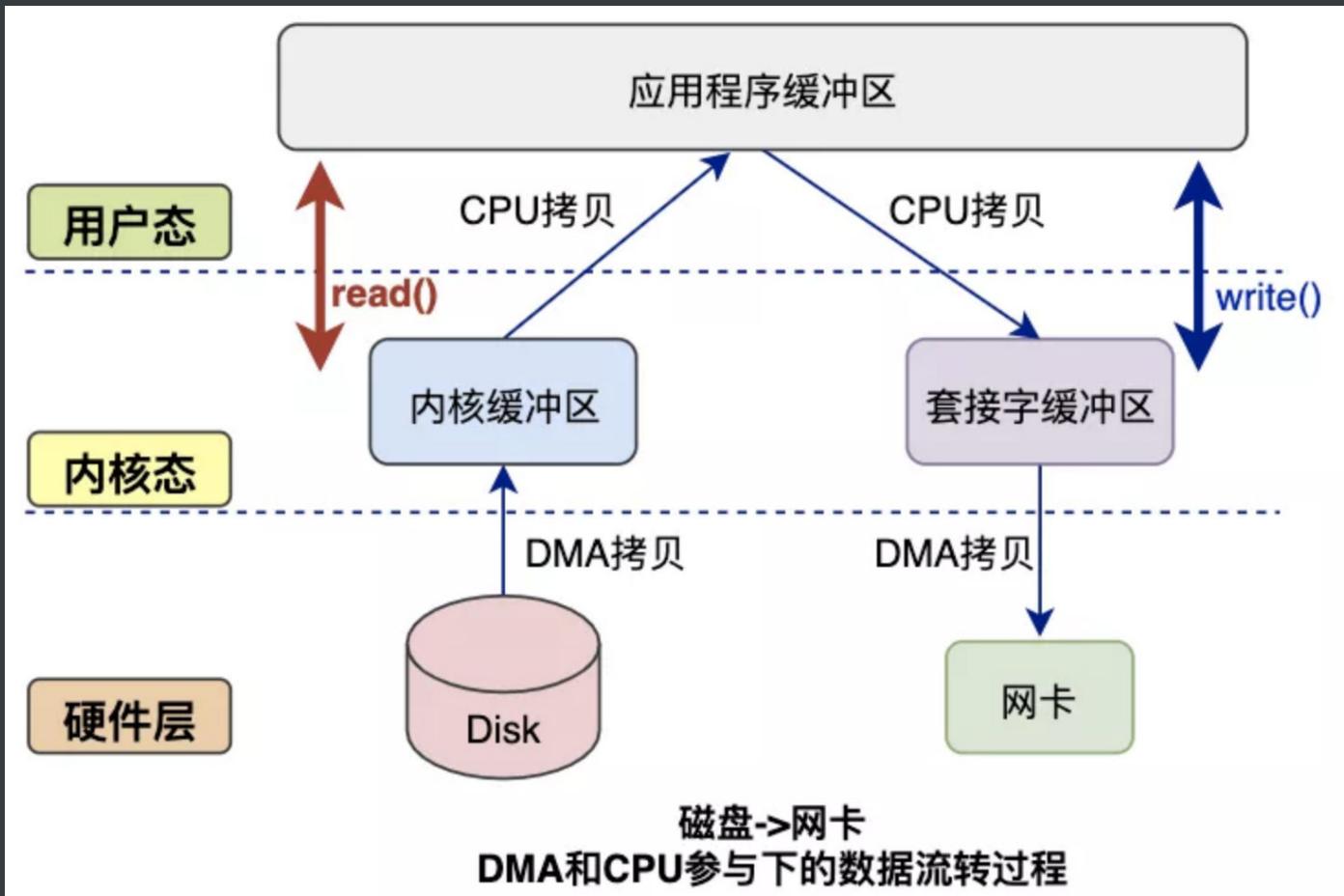
候选者：因为应用程序不能直接去读取硬盘的数据，从上面描述可知需要依赖「内核缓冲区」

候选者：一次读操作会让DMA将磁盘数据拷贝到内核缓冲区，CPU将内核缓冲区数据拷贝到用户缓冲区。

候选者：所谓的零拷贝就是将「CPU将内核缓冲区数据拷贝到用户缓冲区」这次CPU拷贝给省去，来提高效率和性能

候选者：常见的零拷贝技术有mmap（内核缓冲区与用户缓冲区的共享）、sendfile（系统底层函数支持）。

候选者：零拷贝可以提高数据传输的性能，这块在Kafka等框架也有相关的实践。



面试官：嗯，了解了



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

04、反射和动态代理

面试官：今天要不来聊聊Java反射？你对Java反射了解多少？

候选者：嗯，Java反射在JavaSE基础中还是很重要的。

候选者：简单来说，反射就是Java可以给我们在运行时获取类的信息

候选者：在初学的时候可能看不懂、又或是学不太会反射，因为初学的时候往往给的例子都是用反射创建对象，用反射去获取对象上的方法/属性什么的，感觉没多大用

候选者: 但毕竟我已经不是以前的我了，跟以前的看法就不一样了。

候选者: 理解反射重点就在于理解什么是「运行时」，为什么我们要在「运行时」获取类的信息

候选者: 在当时学注解的时候，我们可以发现注解的生命周期有三个枚举值（当时我还告诉过面试官你呢~）

候选者: 分别是SOURCE、CLASS和RUNTIME，其实一样的，RUNTIME就是对标着运行时

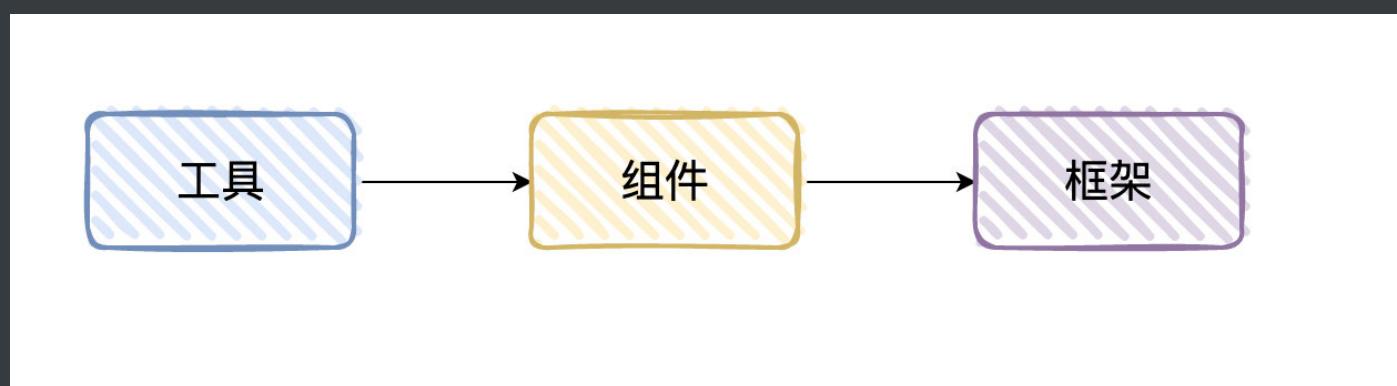
候选者: 我们都知道：我们在编译器写的代码是 .java 文件，经过javac 编译会变成 .class 文件， class 文件会被JVM装载运行（这里就是真正运行着我们所写的代码（虽然是被编译过的），也就所谓的运行时。）

面试官: �恩， 你说了很多，就讲述了什么是运行时，还是快点进入重点吧

候选者: 在运行时获取类的信息，其实就是为了让我们所写的代码更具有「通用性」和「灵活性」

候选者: 要理解反射，需要抛开我们日常写的业务代码。以更高的维度或者说是抽象的思维去看待我们所写的“工具”

候选者: 所谓的“工具”：在单个系统使用叫做“Utils”、被多个系统使用打成jar包叫做“组件”、组件继续发展壮大就叫做“框架”



候选者: 一个好用的“工具”是需要兼容各种情况的。

候选者: 你肯定是不知道用该“工具”的用户传入的是什么对象，但你需要帮他们得到需要的结果。

候选者: 例如SpringMVC 你在方法上写上对象，传入的参数就会帮你封装到对象上

候选者: Mybatis可以让我们只写接口，不写实现类，就可以执行SQL

候选者: 你在类上加上@Component注解，Spring就帮你创建对象

候选者: 这些统统都有反射的身影：约定大于配置，配置大于硬编码。

候选者: 通过“约定”使用姿势，使用反射在运行时获取相应的信息（毕竟作为一个“工具”是真的不知道你是怎么用的），实现代码功能的「通用性」和「灵活性」

面试官: 嗯，明白了

面试官: 结合之前说的泛型，想问下：**你应该知道泛型是会擦除的，那为什么反射能获取到泛型的信息呢？**

面试官: 我再补充一下：泛型的信息只存在编译阶段，在class字节码就看不到泛型的信息了。那为什么下面这段代码能获取得到泛型的信息呢？

```
// 抽象类，定义泛型<T>
public abstract class BaseDao<T> {
    public BaseDao(){
        Class clazz = this.getClass();
        ParameterizedType pt = (ParameterizedType)
        clazz.getGenericSuperclass();
        clazz = (Class) pt.getActualTypeArguments()[0];
        System.out.println(clazz);
    }
}

// 实现类
public class UserDao extends BaseDao<User> {
    public static void main(String[] args) {
        BaseDao<User> userDao = new UserDao();
    }
}
```

```
}

// 执行结果输出
class com.entity.User
```

候选者：嗯，这个问题我在学习的时候也想过

候选者：其实是这样的，可以理解为泛型擦除是有范围的，定义在类上的泛型信息是不会被擦除的。

候选者：Java 编译器仍在 class 文件以 Signature 属性的方式保留了泛型信息

候选者：Type作为顶级接口，Type下还有几种类型，比如TypeVariable、ParameterizedType、WildCardType、GenericArrayType、以及Class。通过这些接口我们就可以在运行时获取泛型相关的信息。

面试官：好，了解

面试官：你了解动态代理吗？

候选者：嗯，了解的。动态代理其实就是代理模式的一种，代理模式是设计模式之一。

候选者：代理模型有静态代理和动态代理。静态代理需要自己写代理类，实现对应的接口，比较麻烦。

候选者：在Java中，动态代理常见的又有两种实现方式：JDK动态代理和CGLIB代理

候选者：JDK动态代理其实就是运用了反射的机制，而CGLIB代理则用的是利用ASM框架，通过修改其字节码生成子类来处理。

候选者：JDK动态代理会帮我们实现接口的方法，通过invokeHandler对所需要的方法进行增强。

JDK动态代理

CGLIB代理

候选者： 动态代理这一技术在实际或者框架原理中是非常常见的

候选者： 像上面所讲的Mybatis不用写实现类，只写接口就可以执行SQL，又或是SpringAOP等等好用的技术，实际上用的就是动态代理。

面试官： �恩，了解了。



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



02-Java并发

01、多线程基础

面试官：首先你来讲讲进程和线程的区别吧？

候选者：进程是系统进行资源分配和调度的独立单位，每一个进程都有它自己的内存空间和系统资源

候选者：进程实现多处理机环境下的进程调度，分派，切换时，都需要花费较大的时间和空间开销

候选者：为了提高系统的执行效率，减少处理机的空转时间和调度切换的时间，以及便于系统管理，所以有了线程，线程取代了进程了调度的基本功能

候选者：简单来说，进程作为资源分配的基本单位，线程作为资源调度的基本单位

进程

线程

资源分配基本单位

资源调度基本单位

面试官：那我们为什么要用多线程呢？你平时工作中用得多吗？

候选者：使用多线程最主要的原因是提高系统的资源利用率。

候选者：现在CPU基本都是多核的，如果你只用单线程，那就是只用到了一个核心，其他的核心就相当于空闲在那里了。

候选者：在平时工作中多线程是随时都可见的。

候选者：比如说，我们系统Web服务器用的是Tomcat，Tomcat处理每一个请求都会从线程连接池里边用一个线程去处理。

候选者：又比如说，我们用连接数据库会用对应的连接池 Druid/C3P0/DBCP...

候选者：...等等这些都用了多线程的。

候选者：上面这些框架已经帮我们屏蔽掉「手写」多线程的问题

多线程

提高资源利用效率

面试官：嗯，了解，那你实际开发中有用过吗？

候选者：当然有了，在我所负责的系统也会用到多线程的。

候选者：比如说，现在要跑一个定时任务，该任务的链路执行时间和过程都非常长，我这边就用一个线程池将该定时任务的请求进行处理。

候选者：这样做的好处就是可以及时返回结果给调用方，能够提高系统的吞吐量。

```
// 请求直接交给线程池来处理
public void push(PushParam pushParam) {
    try {
        pushServiceThreadExecutor.submit(() -> {
            handler(pushParam);
        });
    } catch (Exception e) {
        logger.error("pushServiceThreadExecutor error, exception{}:", e);
    }
}
```

候选者：还有就是我的系统中用了很多生产者与消费者模式，会用多个线程去消费队列的消息，来提高并发度

面试官：你如果在项目中用到了多线程，那肯定得考虑线程安全的问题的吧

面试官：要不你来讲讲什么是线程安全？

候选者：在我的理解下，在Java世界里边，所谓线程安全就是多个线程去执行某类，这个类始终能表现出正确的行为，那么这个类就是线程安全的。

候选者：比如我有一个count变量，在service方法不断的累加这个count变量。

```
public class UnsafeCountingServlet extends GenericServlet implements
Servlet {
    private long count = 0;

    public long getCount() {
        return count;
    }

    public void service(ServletRequest servletRequest, ServletResponse
servletResponse) throws ServletException, IOException {
        ++count;
        // To something else...
    }
}
```

候选者：假设相同的条件下，count变量每次执行的结果都是相同，那我们就可以说是线程安全的。

候选者：显然上面的代码肯定不是线程安全的。

线程安全

多次运行，结果相同

面试官：那你平时是怎么解决，或者怎么思考线程安全问题的呢？

候选者：其实大部分时间我们在代码里边都没有显式去处理线程安全问题，因为这大部分都由框架所做了。

候选者：正如上面提到的Tomcat、Druid、SpringMVC等等。

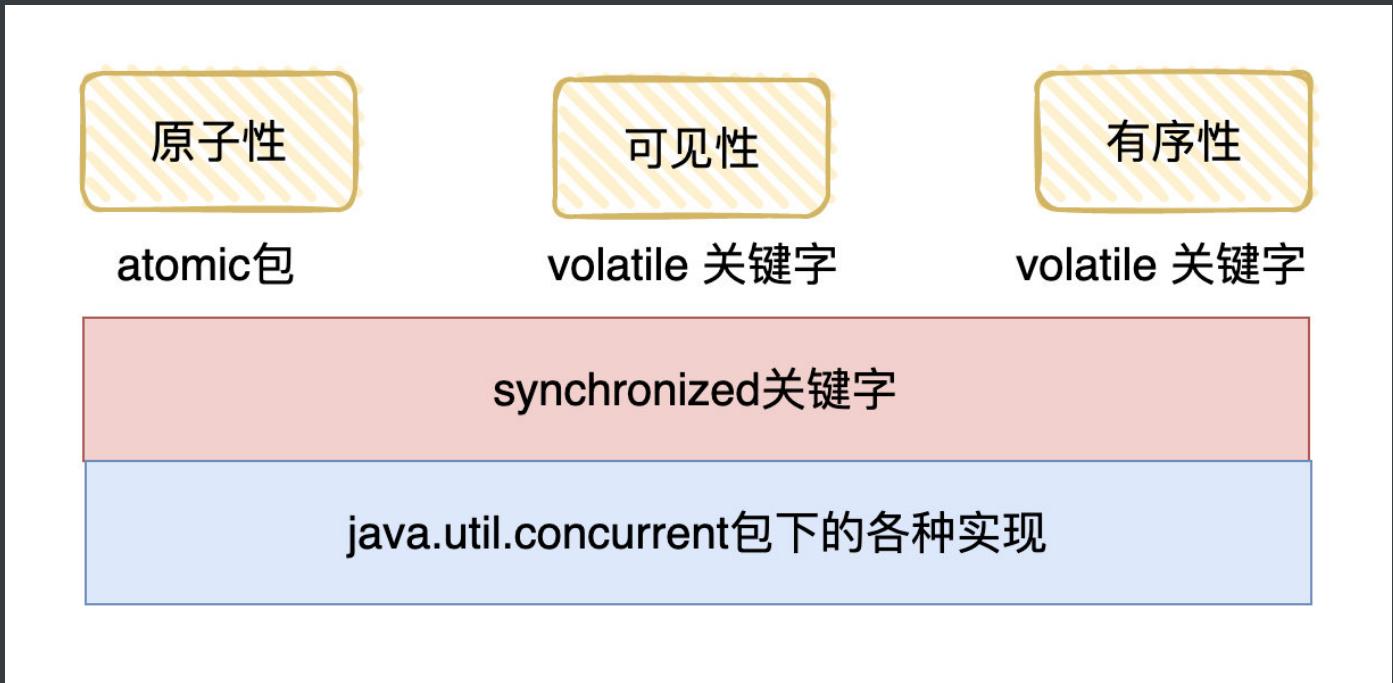
候选者：很多时候，我们判断是否要处理线程安全问题，就看有没有多个线程同时访问一个共享变量。

候选者：像SpringMVC这种，我们日常开发时，不涉及到操作同一个成员变量，那我们就很少需要考虑线程安全问题。

候选者：我个人解决线程安全问题的思路有以下

- 能不能保证操作的原子性，考虑atomic包下的类够不够我们使用。
- 能不能保证操作的可见性，考虑volatile关键字够不够我们使用
- 如果涉及到对线程的控制（比如一次能使用多少个线程，当前线程触发的条件是否依赖其他线程的结果），考虑CountDownLatch/Semaphore等等。
- 如果是集合，考虑java.util.concurrent包下的集合类。

- 如果synchronized无法满足，考虑lock包下的类
-



候选者：总的来说，就是先判断有没有线程安全问题，如果存在则根据具体的情况去判断使用什么方式去处理线程安全的问题。

候选者：虽然synchronized很牛逼，但无脑使用synchronized会影响我们程序的性能的。

面试官：死锁你了解吗？什么情况会造成死锁？要是你能给我讲清楚死锁，我就录取你了

候选者：要是你录取我，我就给你讲清楚死锁

面试官：那我还是继续面面吧

候选者：开始你的表演吧

候选者：造成死锁的原因可以简单概括为：当前线程拥有其他线程需要的资源，当前线程等待其他线程已拥有的资源，都不放弃自己拥有的资源。

候选者：避免死锁的方式一般有以下方案：

1. 固定加锁的顺序，比如我们可以使用Hash值的大小来确定加锁的先后
2. 尽可能缩减加锁的范围，等到操作共享变量的时候才加锁。
3. 使用可释放的定时锁（一段时间申请不到锁的权限了，直接释放掉）

固定加锁顺序

减小锁的范围

定时释放锁

解决死锁的思路

面试官：了解了



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

02、CAS

面试官：今天我们来聊聊CAS吧？你对CAS了解多少？

候选者：好，CAS的全称为compare and swap，比较并交换

候选者：虽然翻译过来是「比较并交换」，但它是一个原子性的操作，对应到CPU指令为cmpxchg

面试官：好家伙，CPU指令你都知道？

候选者：这没什么，都是背的。

面试官：....

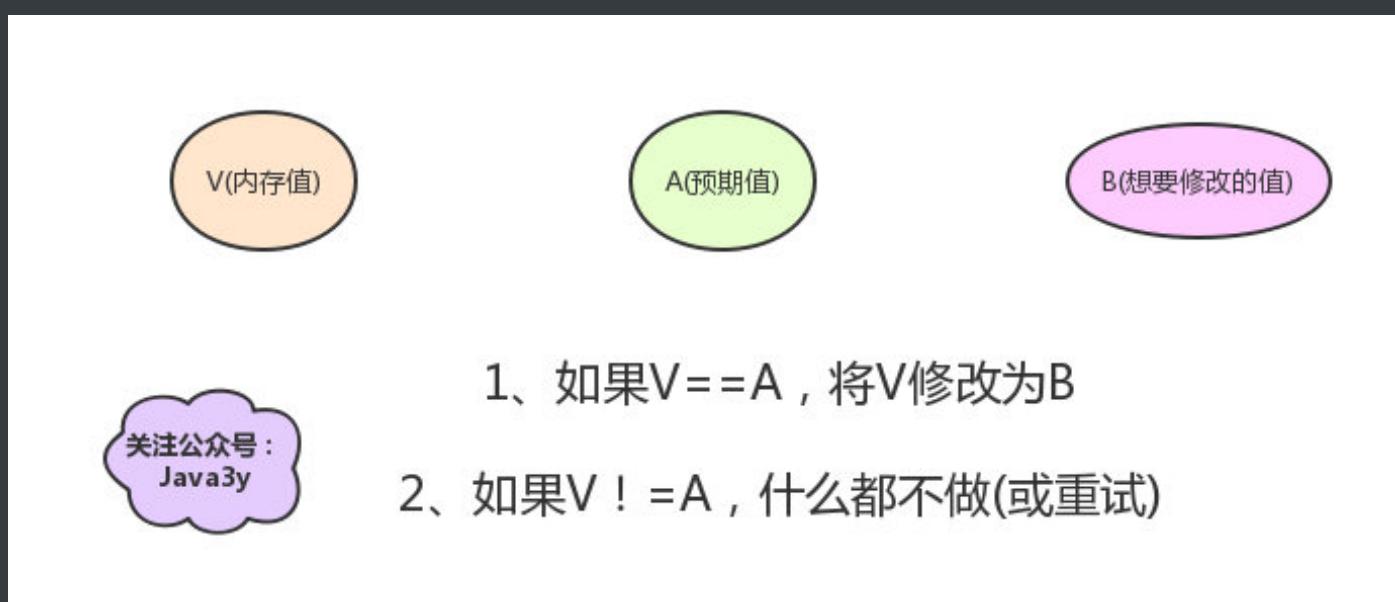
候选者：回到CAS上吧，CAS的操作其实非常简单。

候选者：CAS 有三个操作数：当前值A、内存值V、要修改的新值B

候选者：假设 当前值A 跟 内存值V 相等，那就将 内存值V 改成B

候选者：假设 当前值A 跟 内存值V 不相等，要么就重试，要么就放弃更新

候选者：将当前值与内存值进行对比，判断是否有被修改过，这就是CAS的核心



面试官：确实， 那为什么要用**CAS**呢？

候选者：嗯， 要讲到**CAS**就不得不说**synchronized**锁了， 它是Java锁...然后...

面试官：稍微打断一下， **synchronized**锁你稍微讲下就好了， 后面会专门问的，在这不用细讲。

候选者：ok， 其实就是**synchronized**锁每次只会让一个线程去操作共享资源

候选者：而**CAS**相当于没有加锁， 多个线程都可以直接操作共享资源，在实际去修改的时候才去判断能否修改成功

候选者：在很多的情况下会**synchronized**锁要高效很多

候选者：比如， 对一个值进行累加， 就没必要使用**synchronized**锁， 使用juc包下的Atomic类就足以。

面试官：了解， 那你知道**CAS**会有什么缺点吗？

候选者：CAS有个缺点就是会带来ABA的问题

候选者：从**CAS**更新的时候， 我们可以发现它只比对当前值和内存值是否相等， 这会带来个问题， 下面我举例说明下：

候选者：假设线程A读到当前值是10， 可能线程B把值修改为100， 然后线程C又把值修改为10。

候选者：等到线程A拿到执行权时， 因为当前值和内存值是一致的， 线程A是可以修改的！

候选者：站在线程A的角度来说， 这个值是从未被修改的（：

候选者：这是不合理的， 因为我们从上帝的角度来看， 这个变量已经被线程B和线程C修改过了。

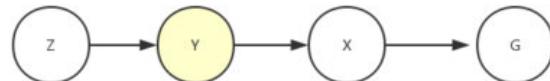
候选者：这就是所谓的ABA问题

线程A

现在有三个线程，一个链表。我现在要删除节点Y

公众号：Java3y

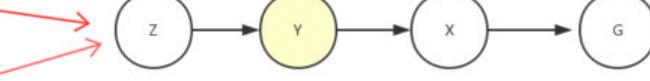
线程C



线程A和线程C同时读取到链表

线程A

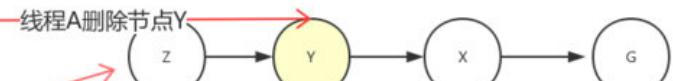
线程C



线程A得到CPU执行权，将节点Y删除，此时线程C因某原因暂时无法得到执行权

线程A

线程C

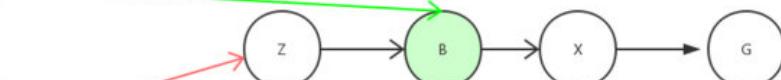


线程A执行完已结束，节点Y已经被GC。此时线程B得到执行权，创建出一个新节点B，位置正好插入到之前被删除Y的位置。

线程B

线程C

创建出的B节点，整位置正好插入到之前被删除Y的位置上



此时线程C得到执行权，同样把线程B创建的新节点删除掉

线程C

因为线程B新创建出来的B节点和之前线程C读到的Y节点的地址是相同的，所以可以执行CAS操作(CAS会比较地址值，这里的地址值是相同的)

但是，我们知道在逻辑上，节点Y和节点B是不同的，所以这是有风险的！

候选者：要解决ABA的问题，Java也提供了AtomicStampedReference类供我们用，说白了就是加了个版本，比对的就是内存值+版本是否一致

面试官：嗯，了解。

面试官：阿里巴巴开发手册提及到 推荐使用 LongAdder 对象，比 AtomicLong 性能更好（减少乐观锁的重试次数），你能帮我解读一下吗？

候选者: AtomicLong做累加的时候实际上就是多个线程操作同一个目标资源

候选者: 在高并发时，只有一个线程是执行成功的，其他的线程都会失败，不断自旋（重试），自旋会成为瓶颈

候选者: 而LongAdder的思想就是把要操作的目标资源「分散」到数组Cell中

候选者: 每个线程对自己的 Cell 变量的 value 进行原子操作，大大降低了失败的次数

候选者: 这就是为什么在高并发场景下，推荐使用LongAdder 的原因（：

面试官: OK



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



03、synchronized

面试官：今天我们来聊聊synchronized吧？

候选者：嗯嗯嗯，没问题

候选者：synchronized是一种互斥锁，一次只能允许一个线程进入被锁住的代码块

候选者：synchronized是Java的一个关键字，它能够将代码块/方法锁起来

候选者：如果synchronized修饰的是实例方法，对应的锁则是对象实例

候选者：如果synchronized修饰的是静态方法，对应的锁则是当前类的Class实例

候选者：如果synchronized修饰的是代码块，对应的锁则是传入synchronized的对象实例



面试官：嗯，要不你来讲讲synchronized的原理呗？

候选者: 通过反编译可以发现

候选者: 当修饰方法时，编译器会生成 ACC_SYNCHRONIZED 关键字用来标识

候选者: 当修饰代码块时，会依赖monitorenter和monitorexit指令

候选者: 但前面已经说了，无论synchronized修饰的是方法还是代码块，对应的锁都是一个实例（对象）

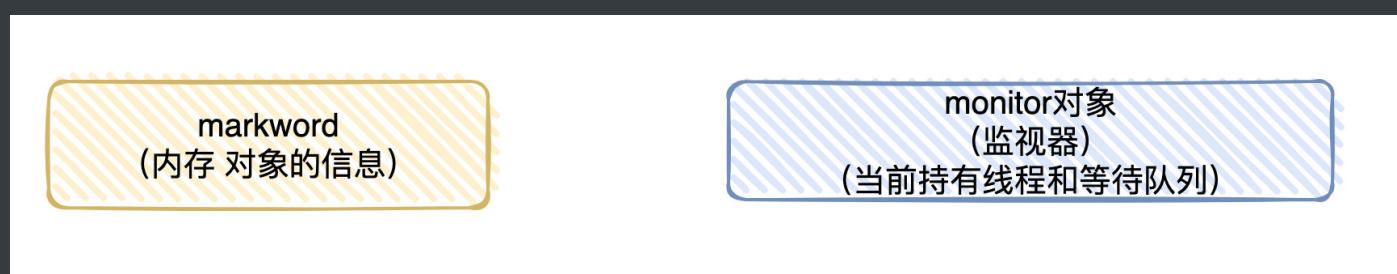
候选者: 在内存中，对象一般由三部分组成，分别是对象头、对象实际数据和对齐填充

候选者: 重点在于对象头，对象头又由几部分组成，但我们重点关注对象头Mark Word的信息就好了

候选者: Mark Word会记录对象关于锁的信息

候选者: 又因为每个对象都会有一个与之对应的monitor对象，monitor对象中存储着当前持有锁的线程以及等待锁的线程队列

候选者: 了解Mark Word和monitor对象是理解 synchronized 原理的前提



面试官: 嗯，听说synchronized锁在 JDK 1.6 之后做了很多的优化，这块你了解多少呢？

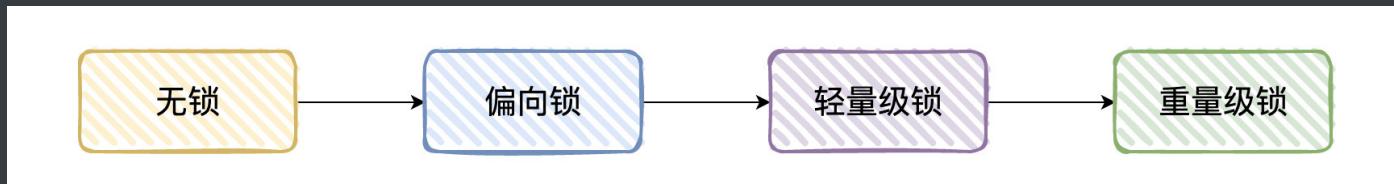
候选者: 其实是这样的，在JDK 1.6之前是重量级锁，线程进入同步代码块/方法 时

候选者: monitor对象就会把当前进入线程的Id进行存储，设置Mark Word的monitor对象地址，并把阻塞的线程存储到monitor的等待线程队列中

候选者: 它加锁是依赖底层操作系统的 mutex 相关指令实现，所以会有用户态和内核态之间的切换，性能损耗十分明显

候选者：而JDK1.6以后引入偏向锁和轻量级锁在JVM层面实现加锁的逻辑，不依赖底层操作系统，就没有切换的消耗

候选者：所以，Mark Word对锁的状态记录一共有4种：无锁、偏向锁、轻量级锁和重量级锁



面试官：简单来说说偏向锁、轻量级锁和重量级锁吧

候选者：嗯，没问题

候选者：偏向锁指的就是JVM会认为只有某个线程才会执行同步代码（没有竞争的环境）

候选者：所以在Mark Word会直接记录线程ID，只要线程来执行代码了，会比对线程ID是否相等，相等则当前线程能直接获取得到锁，执行同步代码

候选者：如果不相等，则用CAS来尝试修改当前的线程ID，如果CAS修改成功，那还是能获取得到锁，执行同步代码

候选者：如果CAS失败了，说明有竞争环境，此时会对偏向锁撤销，升级为轻量级锁。

候选者：在轻量级锁状态下，当前线程会在栈帧下创建Lock Record，LockRecord会把Mark Word的信息拷贝进去，且有个Owner指针指向加锁的对象

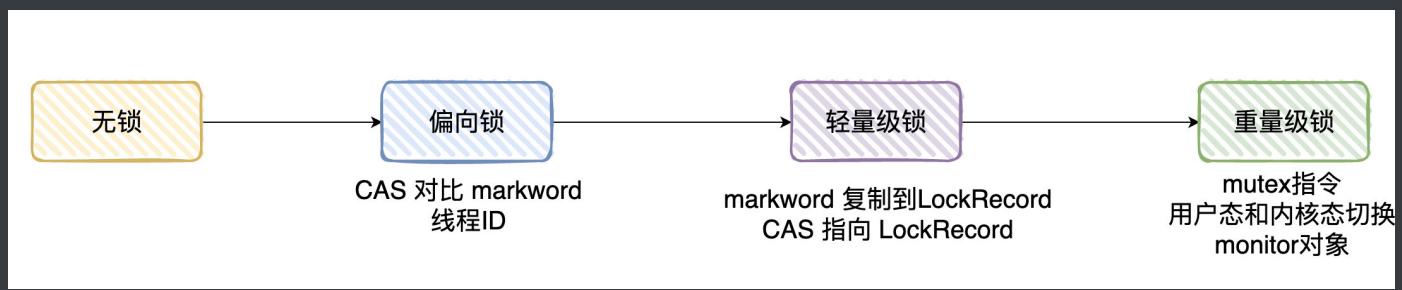
候选者：线程执行到同步代码时，则用CAS试图将Mark Word的指向到线程栈帧的Lock Record，假设CAS修改成功，则获取得到轻量级锁

候选者：假设修改失败，则自旋（重试），自旋一定次数后，则升级为重量级锁

候选者：简单总结一下

候选者：synchronized锁原来只有重量级锁，依赖操作系统的mutex指令，需要用户态和内核态切换，性能损耗十分明显

候选者: 重量级锁用到monitor对象，而偏向锁则在Mark Word记录线程ID进行比对，轻量级锁则是拷贝Mark Word到Lock Record，用CAS+自旋的方式获取。



候选者: 引入了偏向锁和轻量级锁，就是为了在不同的使用场景使用不同的锁，进而提高效率。锁只有升级，没有降级

候选者: 1) 只有一个线程进入临界区，偏向锁

候选者: 2) 多个线程交替进入临界区，轻量级锁

候选者: 3) 多线程同时进入临界区，重量级锁

面试官: OK, 明白了。



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zhongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

04、AQS和ReentrantLock

面试官：今天我们来聊聊lock锁吧？

候选者：嗯嗯嗯，没问题

面试官：先问点简单的吧，刚睡醒，还是有点困的。

候选者：刚睡醒来面我干嘛？你就这态度？

面试官：哈？你刚说了什么？

候选者：没事，我没说话…

面试官：你知道什么叫做公平和非公平锁吗

候选者：公平锁指的就是：在竞争环境下，先到临界区的线程比后到的线程一定更快地获取得到锁

候选者：那非公平就很好理解了：先到临界区的线程未必比后到的线程更快地获取得到锁

面试官：如果让你实现的话，你怎么实现公平和非公平锁？

候选者：公平锁可以把竞争的线程放在一个先进先出的队列上

候选者：只要持有锁的线程执行完了，唤醒队列的下一个线程去获取锁就好了

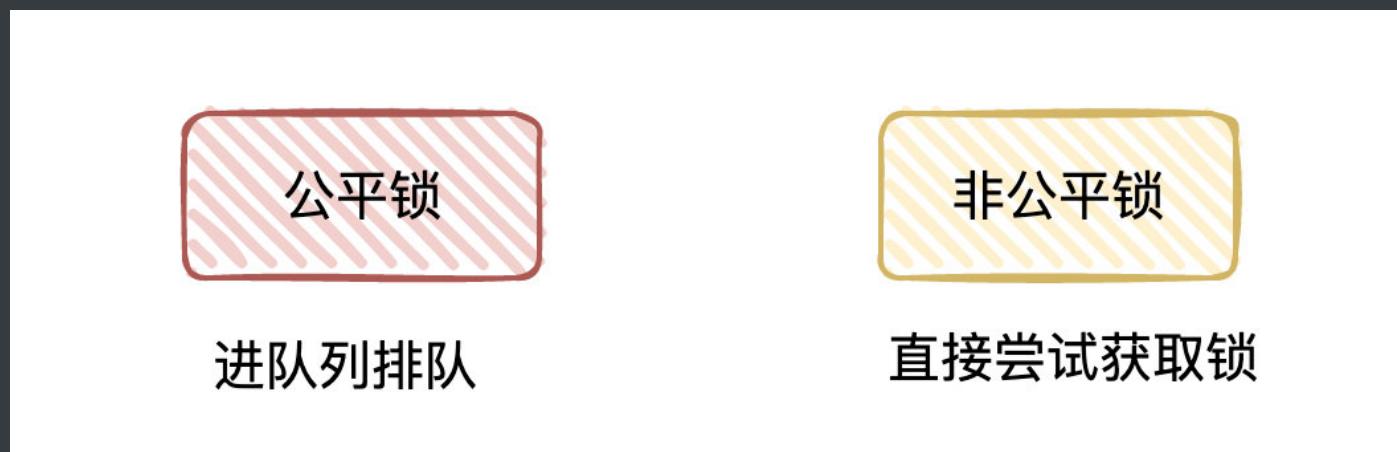
候选者：非公平锁的概念上面已经提到了：后到的线程可能比前到临界区的线程获取得到锁

候选者: 那实现也很简单，线程先尝试能不能获取得到锁，如果获取得到锁了就执行同步代码了

候选者: 如果获取不到锁，那就再把这个线程放到队列呗

候选者: 所以公平和非公平的区别就是：线程执行同步代码块时，是否会去尝试获取锁。

候选者: 如果会尝试获取锁，那就是非公平的。如果不会尝试获取锁，直接进队列，再等待唤醒，那就是公平的。



面试官: 为什么要进队列呢？线程一直尝试获取锁不就行了么？

候选者: 一直尝试获取锁，专业点就叫做自旋，需要耗费资源的。

候选者: 多个线程一直在自旋，而且大多数都是竞争失败的，哪有人会这样实现的

候选者: 不会吧，不会吧，你不会就是这样实现的吧

面试官: 我就问问...

面试官: 那上次面试所问的synchronized锁是公平的还是非公平的？

候选者: 非公平的。

候选者: 偏向锁很好理解，如果当前线程ID与markword存储的不相等，则CAS尝试更换线程ID，CAS成功就获取得到锁了

候选者: CAS失败则升级为轻量级锁

候选者: 轻量级锁实际上也是通过CAS来抢占锁资源（只不过多了拷贝Mark Word到Lock Record的过程）

候选者: 抢占成功到锁就归属给该线程了，但自旋失败一定次数后升级重量级锁

候选者: 重量级锁通过monitor对象中的队列存储线程，但线程进入队列前，还是会先尝试获取得到锁，如果能获取不到才进入线程等待队列中

候选者: 综上所述，synchronized无论处理哪种锁，都是先尝试获取，获取不到才升级||放到队列上的，所以是非公平的



面试官: 嗯，讲得挺仔细的。**AQS**你了解吗？

候选者: 嗯嗯，AQS全称叫做AbstractQueuedSynchronizer

候选者: 是可以给我们实现锁的一个「框架」，内部实现的关键就是维护了一个先进先出的队列以及state状态变量

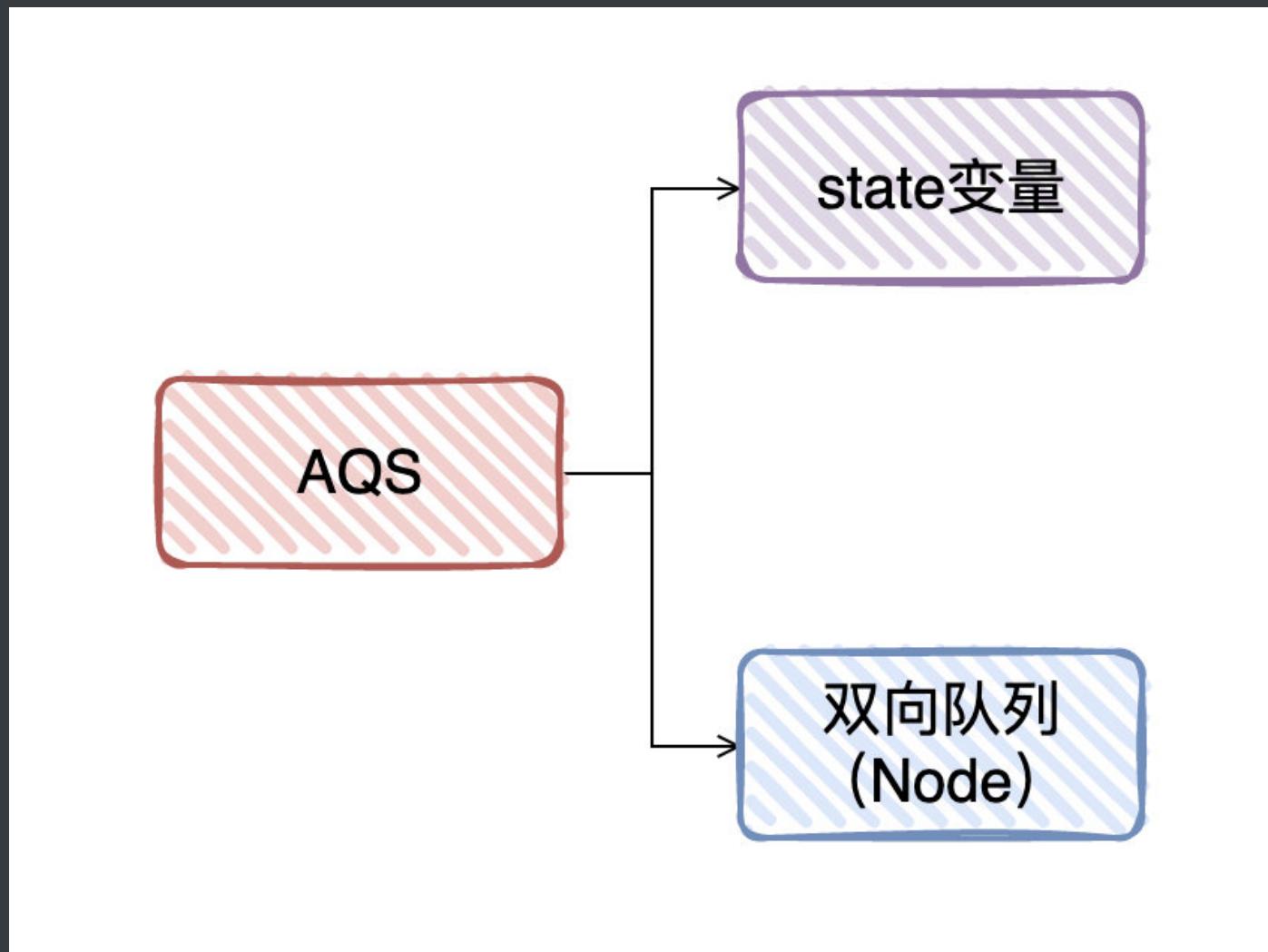
候选者: 先进先出队列存储的载体叫做Node节点，该节点标识着当前的状态值、是独占还是共享模式以及它的前驱和后继节点等等信息

候选者: 简单理解就是：AQS定义了模板，具体实现由各个子类完成。

候选者: 总体的流程可以总结为：会把需要等待的线程以Node的形式放到这个先进先出的队列上，state变量则表示为当前锁的状态。

候选者: 像ReentrantLock、ReentrantReadWriteLock、CountDownLatch、Semaphore这些常用的实现类都是基于AQS实现的

候选者: AQS支持两种模式：独占（锁只会被一个线程独占）和共享（多个线程可同时执行）



面试官: 你以**ReentrantLock**来讲讲加锁和解锁的过程呗

候选者: 以非公平锁为例，我们在外界调用lock方法的时候，源码是这样实现的

候选者: 1):CAS尝试获取锁，获取成功则可以执行同步代码

候选者: 2):CAS获取失败，则调用acquire方法，acquire方法实际上就是AQS的模板方法

候选者: 3):acquire首先会调用子类的tryAcquire方法（又回到了ReentrantLock中）

候选者: 4):tryAcquire方法实际上会判断当前的state是否等于0, 等于0说明没有线程持有锁, 则又尝试CAS直接获取锁

候选者: 5):如果CAS获取成功, 则可以执行同步代码

候选者: 6):如果CAS获取失败, 那判断当前线程是否就持有锁, 如果是持有的锁, 那更新state的值, 获取得到锁（这里其实就是处理可重入的逻辑）

候选者: 7):CAS失败&&非重入的情况, 则回到tryAcquire方法执行「入队列」的操作

候选者: 8):将节点入队列之后, 会判断「前驱节点」是不是头节点, 如果是头结点又会用CAS尝试获取锁

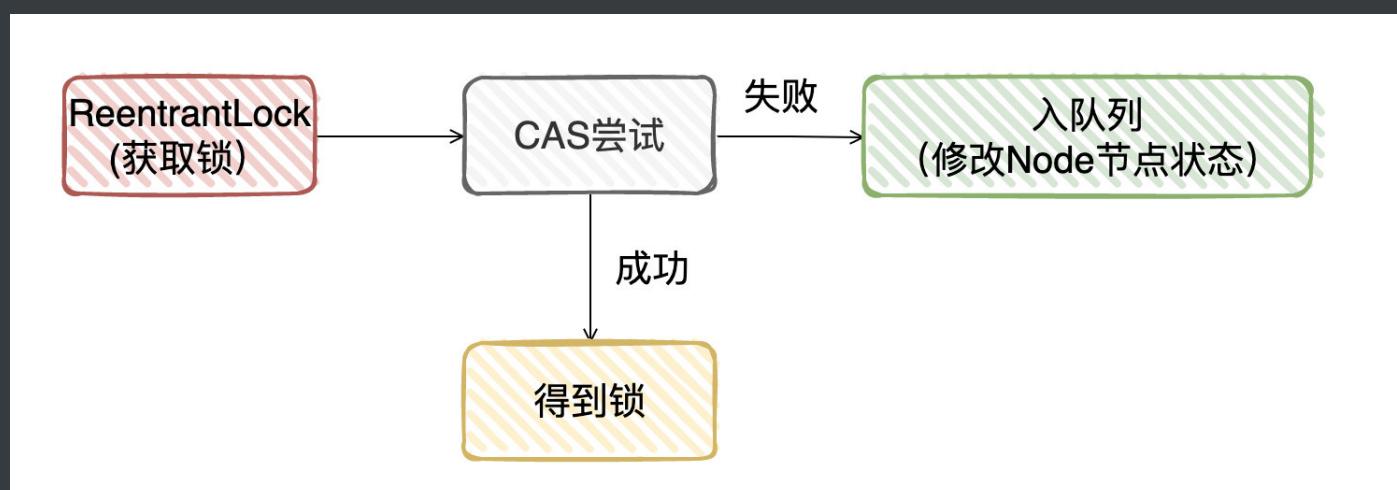
候选者: 9):如果是「前驱节点」是头节点并获取得到锁, 则把当前节点设置为头结点, 并且将前驱节点置空（实际上就是原有的头节点已经释放锁了）

候选者: 10):没获取得到锁, 则判断前驱节点的状态是否为SIGNAL, 如果不是, 则找到合法的前驱节点, 并使用CAS将状态设置为SIGNAL

候选者: 11):最后调用park将当前线程挂起

面试官: 你说了一大堆, 麻烦使用压缩算法压缩下加锁的过程。

候选者: 压缩后: 当线程CAS获取锁失败, 将当前线程入队列, 把前驱节点状态设置为SIGNAL状态, 并将自己挂起。



面试官：为什么要设置前驱节点为**SIGNAL**状态，有啥用？

候选者：其实就是表示后继节点需要被唤醒

候选者：我先把解锁的过程说下吧

候选者：1):外界调用unlock方法时，实际上会调用AQS的release方法，而release方法会调用子类tryRelease方法（又回到了ReentrantLock中）

候选者：2):tryRelease会把state一直减（锁重入可使state>1），直至到0，当前线程说明已经把锁释放了

候选者：3):随后从队尾往前找节点状态需要<0，并离头节点最近的节点进行唤醒

候选者：唤醒之后，被唤醒的线程则尝试使用CAS获取锁，假设获取锁得到则把头节点给干掉，把自己设置为头节点

候选者：解锁的逻辑非常简单哈，把state置0，唤醒头结点下一个合法的节点，被唤醒的节点线程自然就会去获取锁

面试官：嗯，了解了。

候选者：回到上一个问题，为什么要设置前驱节点为**SIGNAL**状态

候选者：其实归根结底就是为了判断节点的状态，去做些处理。

候选者：Node中节点的状态有4种，分别是：CANCELLED(1)、SIGNAL(-1)、CONDITION(-2)、PROPAGATE(-3)和0

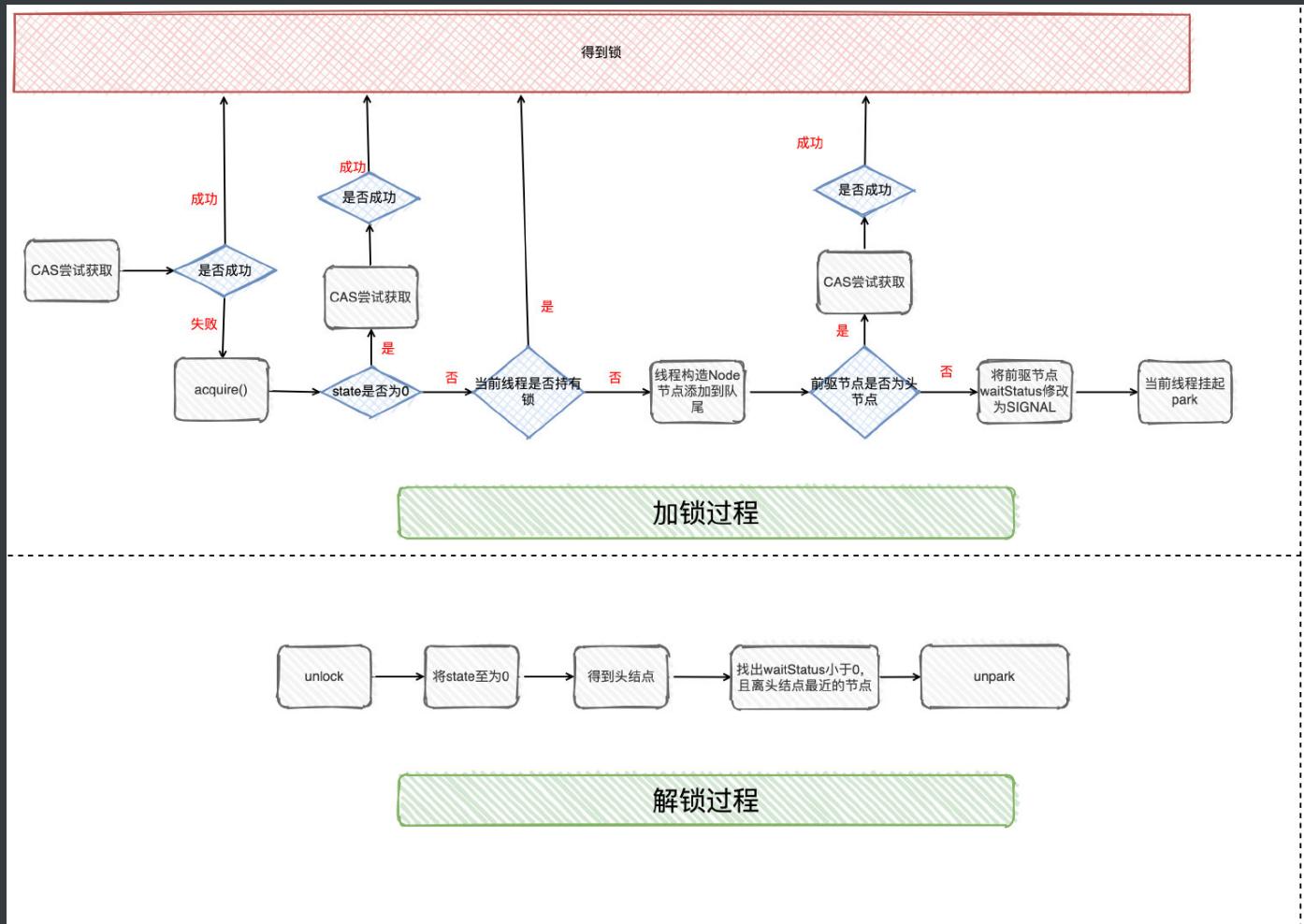
候选者：在ReentrantLock解锁的时候，会判断节点的状态是否小于0，小于等于0才说明需要被唤醒

候选者：另外一提的是：公平锁的实现与非公平锁是很像的，只不过在获取锁时不会直接尝试使用CAS来获取锁。

候选者：只有当队列没节点并且state为0时才会去获取锁，不然都会把当前线程放到队列中

面试官：最后画个流程图吧，你画好了，他们会给你点赞和转发的

候选者：真的假的？





第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



05、线程池

面试官：今天来聊聊线程池呗，你对Java线程池了解多少？或者换个问法：为什么需要线程池？

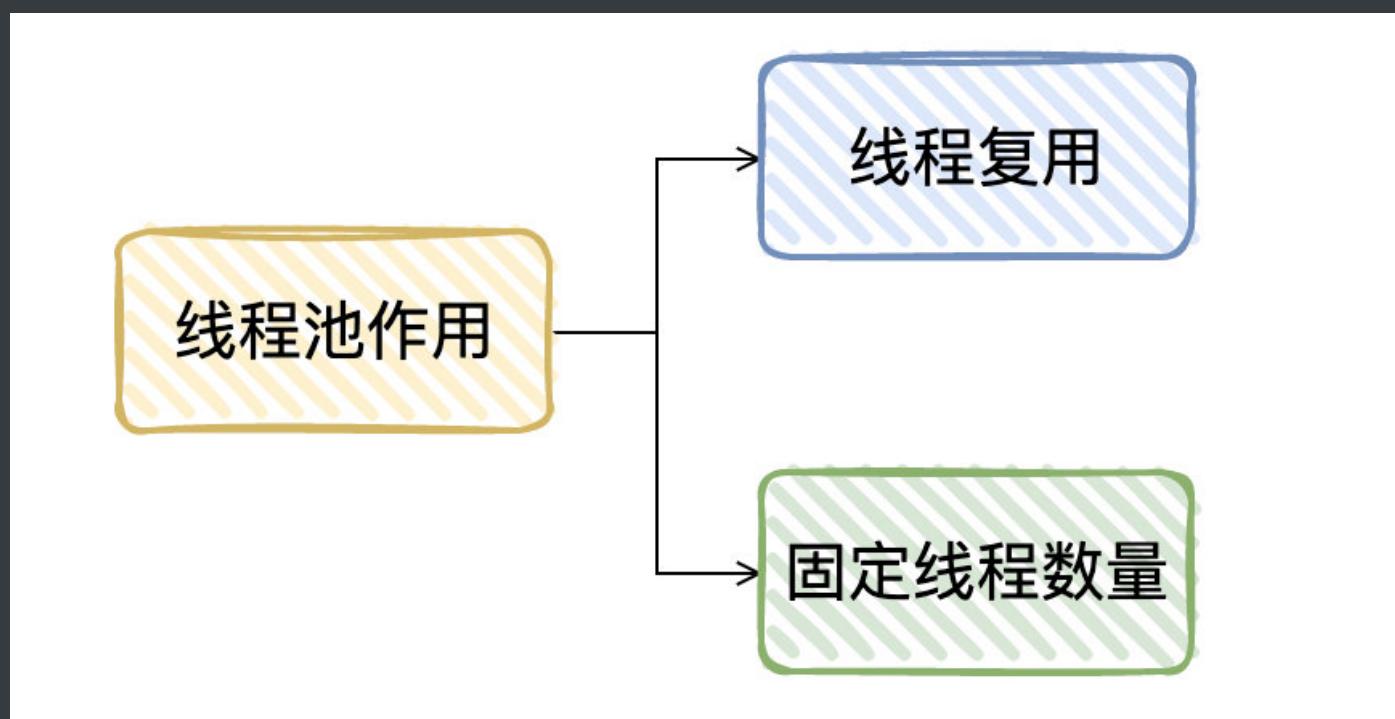
候选者：其实是这样子的

候选者：JVM在HotSpot的线程模型下，Java线程会一对一映射为内核线程

候选者：这意味着，在Java中每次创建以及回收线程都会去内核创建以及回收

候选者：这就有可能导致：创建和销毁线程所花费的时间和资源可能比处理的任务花费的时间和资源要更多

候选者：线程池的出现是为了提高线程的复用性以及固定线程的数量！！！



面试官：你在项目中用到了线程池吗？

候选者：嗯，用到的。我先说下背景吧

候选者：我所负责的项目是消息管理平台，提供其中一个功能就是：运营会圈定人群，然后群发消息

候选者：主要流程大致就是：创建模板->定时->群发消息->用户收到消息

候选者：运营圈定的人群实际上在模板上只是一个ID，我这边要通过ID去获取到HDFS文件

候选者：对HDFS文件进行遍历，然后继续往下发

候选者：「接收到定时任务，再对HDFS进行遍历」这里的处理，我用的就是线程池处理

面试官：为什么选择用线程池呢？

候选者：HDFS遍历其实就是IO的操作，我把这个过程给异步化，为了提高系统的吞吐量，于是我这里用的线程池。

候选者：即便遍历HDFS出现问题，我这边都有完备的监控和告警可以及时发现。

面试官：那你是怎么用线程池的呢？用 `Executors` 去创建的吗？

候选者：不是的，我这边用的`ThreadPoolExecutor`去创建线程池

候选者：其实看阿里巴巴开发手册就有提到，不要使用`Executors`去创建线程。

候选者：最主要的目的就是：使用`ThreadPoolExecutor`创建的线程你是更能了解线程池运行的规则，避免资源耗尽的风险

候选者：`ThreadPoolExecutor`在构造的时候有几个重要的参数，分别是：

候选者：`corePoolSize`（核心线程数量）、`maximumPoolSize`（最大线程数量）、`keepAliveTime`（线程空余时间）、`workQueue`（阻塞队列）、`handler`（任务拒绝策略）

候选者：这几个参数应该很好理解哈，我就说下任务提交的流程，分别对应着几个参数的作用吧。

候选者：1):首先会判断运行线程数是否小于`corePoolSize`，如果小于，则直接创建新的线程执行任务

候选者：2):如果大于`corePoolSize`，判断`workQueue`阻塞队列是否已满，如果还没满，则将任务放到阻塞队列中

候选者: 3):如果workQueue阻塞队列已经满了，则判断当前线程数是否大于maximumPoolSize，如果没大于则创建新的线程执行任务

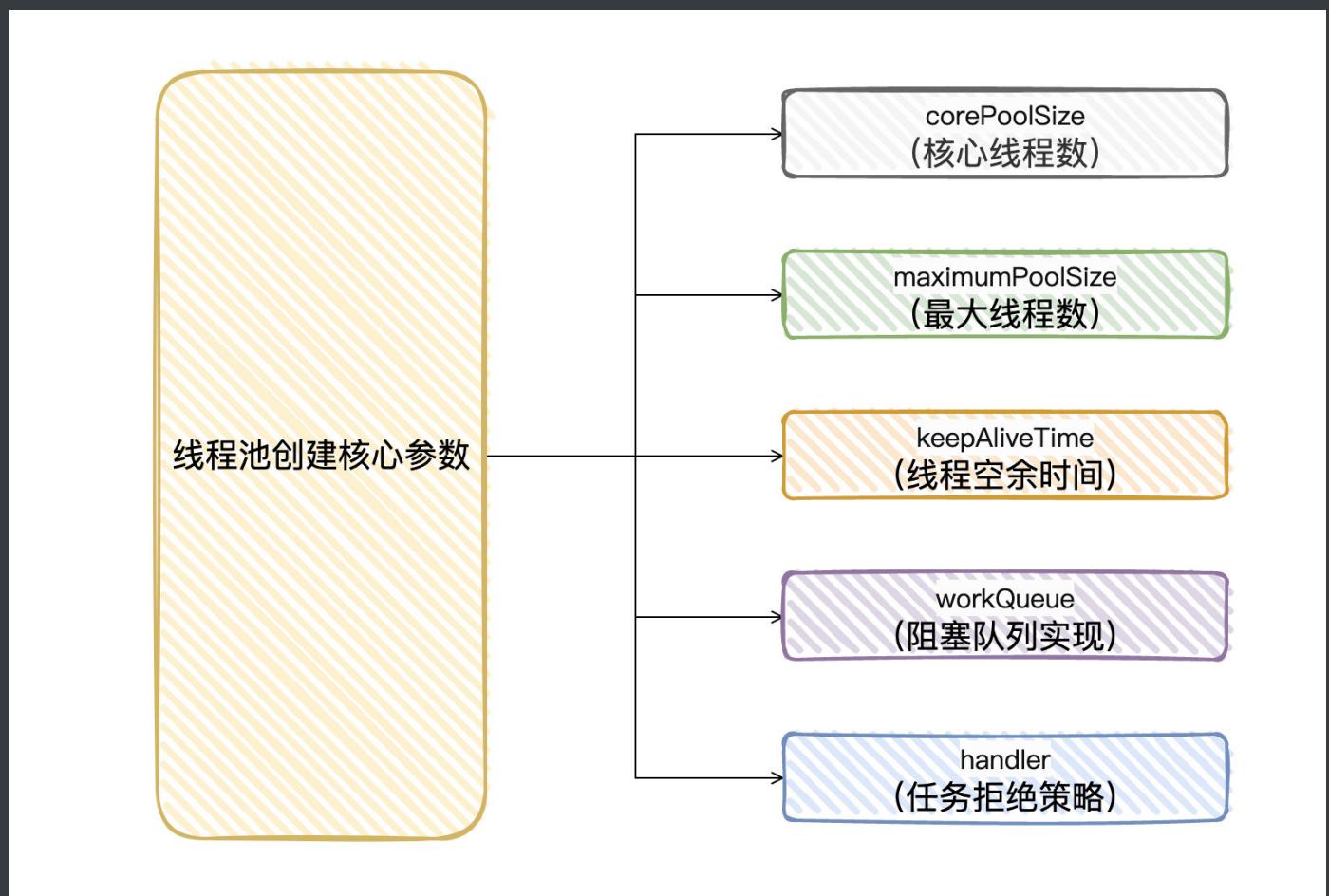
候选者: 4):如果大于maximumPoolSize，则执行任务拒绝策略（具体就是你自己实现的handler）

候选者: 这里有个点需要注意下，就是workQueue阻塞队列满了，但当前线程数小于maximumPoolSize，这时候会创建新的线程执行任务

候选者: 源码就是这样实现的

候选者: 不过一般我们都是将corePoolSize和maximumPoolSize设置相同数量

候选者: keepAliveTime指的就是，当前运行的线程数大于核心线程数了，只要空闲时间达到了，就会对线程进行回收



面试官: 嗯，了解了。

面试官：那我再问一个问题，你创建线程池肯定会指定线程数的嘛，你这块是怎么考量的。

候选者：线程池指定线程数这块，首先要考量自己的业务是什么样的

候选者：是cpu密集型的还是io密集型的，假设运行应用的机器CPU核心数是N

候选者：那cpu密集型的可以先给到 $N+1$ ，io密集型的可以给到 $2N$ 去试试

候选者：上面这个只是一个常见的经验做法，具体究竟开多少线程，需要压测才能比较准确地定下来

候选者：线程不是说越大越好，在之前的面试我也提到过，多线程是为了充分利用CPU的资源

候选者：如果设置的线程过多，线程大量有上下文切换，这一部分也会带来系统的开销，这就得不偿失了

面试官：ThreadPoolExecutor你看过源码吗？

候选者：看过的，其实上面说的ThreadPoolExecutor几个参数，在源码的顶部注释都有

候选者：在执行的时候，重点就在于它维护了一个ctl参数，这个ctl参数的高3位表示线程池的状态，低29位来表示线程的数量

候选者：里边用到了大量的位运算符操作，具体细节我就忘了，但是流程还是上面所讲的

面试官：好吧



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

06、ThreadLocal

面试官：今天要不来聊聊ThreadLocal吧？

候选者：我个人对ThreadLocal理解就是

候选者：它提供了线程的局部变量，每个线程都可以通过set/get来对这个局部变量进行操作

候选者：不会和其他线程的局部变量进行冲突，实现了线程的数据隔离



面试官：你在工作中有用到过**ThreadLocal**吗？

候选者：这块是真不多，不过还是有一处的。就是我们项目有个的DateUtils工具类

候选者：这个工具类主要是对时间进行格式化

候选者：格式化/转化的实现是用的SimpleDateFormat

候选者：但众所周知SimpleDateFormat不是线程安全的，所以我们就用ThreadLocal来让每个线程装载着自己的SimpleDateFormat对象

候选者：以达到在格式化时间时，线程安全的目的

候选者：在方法上创建SimpleDateFormat对象也没问题，但每调用一次就创建一次有点不优雅

候选者：在工作中ThreadLocal的应用场景确实不多，但要不我给你讲讲Spring是怎么用的？

面试官：好吧，你讲讲呗

候选者：Spring提供了事务相关的操作，而我们知道事务是得保证一组操作同时成功或失败的

候选者：这意味着我们一次事务的所有操作需要在同一个数据库连接上

候选者: 但是在我日常写代码的时候是不需要关注这点的

候选者: Spring就是用的ThreadLocal来实现， ThreadLocal存储的类型是一个Map

候选者: Map中的key 是DataSource, value 是Connection (为了应对多数据源的情况, 所以是一个Map)

候选者: 用了ThreadLocal保证了同一个线程获取一个Connection对象, 从而保证一次事务的所有操作需要在同一个数据库连接上

```
public abstract class TransactionSynchronizationManager {  
  
    private static final Log logger = LogFactory.getLog(TransactionSynchronizationManager.class);  
  
    private static final ThreadLocal<Map<Object, Object>> resources =  
        new NamedThreadLocal<Map<Object, Object>>("Transactional resources");  
  
    private static final ThreadLocal<Set<TransactionSynchronization>> synchronizations =  
        new NamedThreadLocal<~>("Transaction synchronizations");
```

面试官: 了解

面试官: 你知道ThreadLocal内存泄露这个知识点吗?

候选者: 怎么都喜欢问这个...

候选者: 了解的, 要不我先来讲讲ThreadLocal的原理?

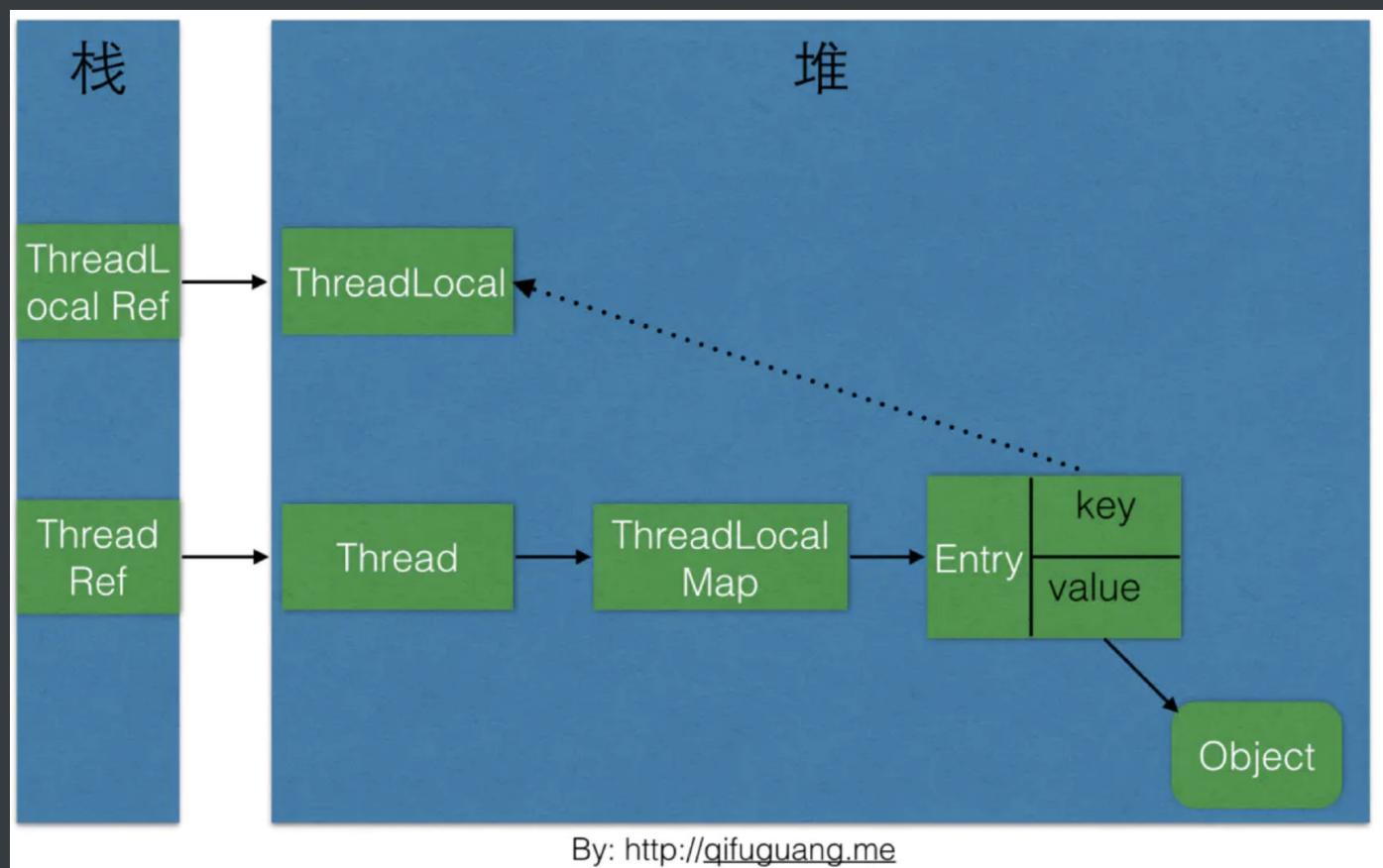
面试官: 请开始你的表演吧

候选者: ThreadLocal是一个壳子, 真正的存储结构是ThreadLocal里有ThreadLocalMap这么个内部类

候选者: 而有趣的是, ThreadLocalMap的引用是在Thread上定义的

候选者: ThreadLocal本身并不存储值, 它只是作为key来让线程从ThreadLocalMap获取value

候选者: 所以，得出的结论就是ThreadLocalMap该结构本身就在Thread下定义，而ThreadLocal只是作为key，存储set到ThreadLocalMap的变量当然是线程私有的咯



面试官: 那我想问下，我可以在ThreadLocal下定义Map，key是Thread，value是set进去的值吗？

面试官: 就是说，为啥我要把ThreadLocal做为key，而不是Thread做为key？这样不是更清晰吗？

候选者: 嗯，我明白你的意思。

候选者: 理论上是可以，但没那么优雅。

候选者: 你提出的做法实际上就是所有的线程都访问ThreadLocal的Map，而key是当前线程

候选者: 但这有点小问题，一个线程是可以拥有多个私有变量的嘛，那key如果是当前线程的话，意味着还点做点「手脚」来唯一标识set进去的value

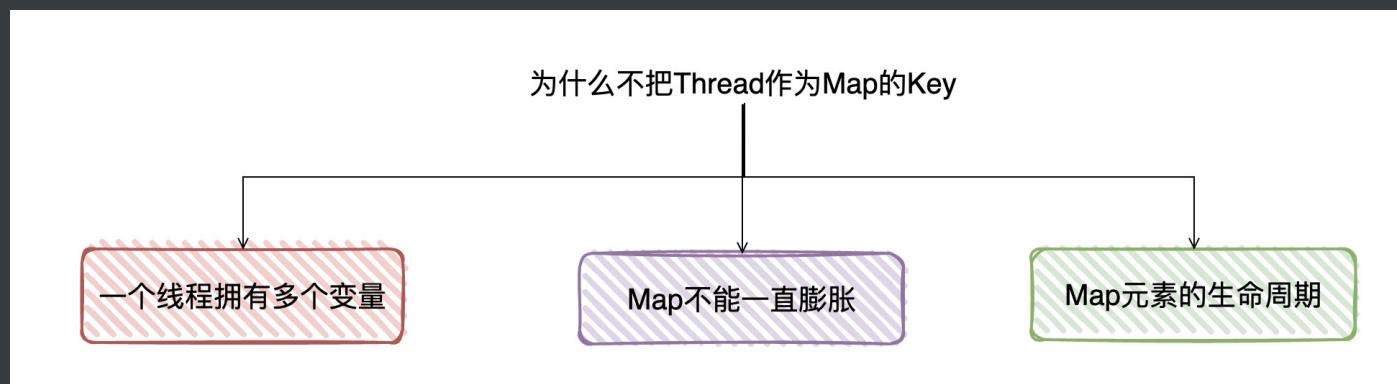
候选者: 假设上一步解决了，还有个问题就是：并发量足够大时，意味着所有的线程都去操作同一个Map，Map体积有可能会膨胀，导致访问性能的下降

候选者：这个Map维护着所有的线程的私有变量，意味着你不知道什么时候可以「销毁」

候选者：现在JDK实现的结构就不一样了。

候选者：线程需要多个私有变量，那有多个ThreadLocal对象足以，对应的Map体积不会太大

候选者：只要线程销毁了，ThreadLocalMap也会被销毁



面试官：嗯，了解。

面试官：回到**ThreadLocal**内存泄露上吧，谈谈你对这个的理解呗

候选者：ThreadLocal内存泄露其实发生的概率非常非常低，我也不知道为什么这么喜欢问。

候选者：回到原理上，我们知道Thread在创建的时候，会有栈引用指向Thread对象，Thread对象内部维护了ThreadLocalMap引用

候选者：而ThreadLocalMap的Key是ThreadLocal，value是传入的Object

候选者：ThreadLocal对象会被对应的栈引用关联，ThreadLocalMap的key也指向着ThreadLocal

候选者：ThreadLocalRef && ThreadLocalMap Entry key ->ThreadLocal

候选者：ThreadRef->Thread->ThreadLoalMap-> Entry value-> Object

候选者：网上大多分析的是ThreadLocalMap的key是弱引用指向ThreadLocal

面试官：嗯...要不顺便讲讲Java的4种引用吧

候选者：强引用是最常见的，只要把一个对象赋给一个引用变量，这个引用变量就是一个强引用

候选者：强引用：只要对象没有被置null，在GC时就不会被回收

候选者：软引用相对弱化了一些，需要继承 SoftReference实现

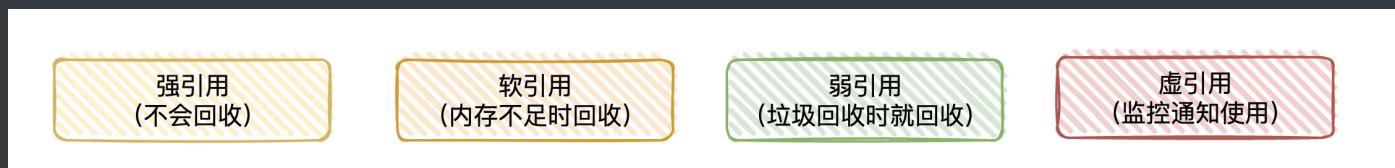
候选者：软引用：如果内存充足，只有软引用指向的对象不会被回收。如果内存不足了，只有软引用指向的对象就会被回收

候选者：弱引用又更弱了一些，需要继承WeakReference实现

候选者：弱引用：只要发生GC，只有弱引用指向的对象就会被回收

候选者：最后就是虚引用，需要继承PhantomReference实现

候选者：虚引用的主要作用是：跟踪对象垃圾回收的状态，当回收时通过引用队列做些「通知类」的工作



候选者：了解了这几种引用之后，再回过头来看ThreadLocal

面试官：嗯..

候选者：ThreadLocal内存泄露指的是：ThreadLocal被回收了，ThreadLocalMap Entry的key没有了指向

候选者：但Entry仍然有ThreadRef->Thread->ThreadLocalMap-> Entry value-> Object 这条引用一直存在导致内存泄露

面试官： 嗯..

候选者： 为什么我说导致内存泄露的概率非常低呢，我觉得是这样的

候选者： 首先ThreadLocal被两种引用指向

候选者： 1):ThreadLocalRef->ThreadLocal (强引用)

候选者： 2):ThreadLocalMap Entry key ->ThreadLocal (弱引用)

候选者： 只要ThreadLocal没被回收（使用时强引用不置null），那ThreadLocalMap Entry key的指向就不会在GC时断开被回收，也没有内存泄露一说法

候选者： 通过ThreadLocal了解实现后，又知道ThreadLocalMap是依附在Thread上的，只要Thread销毁，那ThreadLocalMap也会销毁

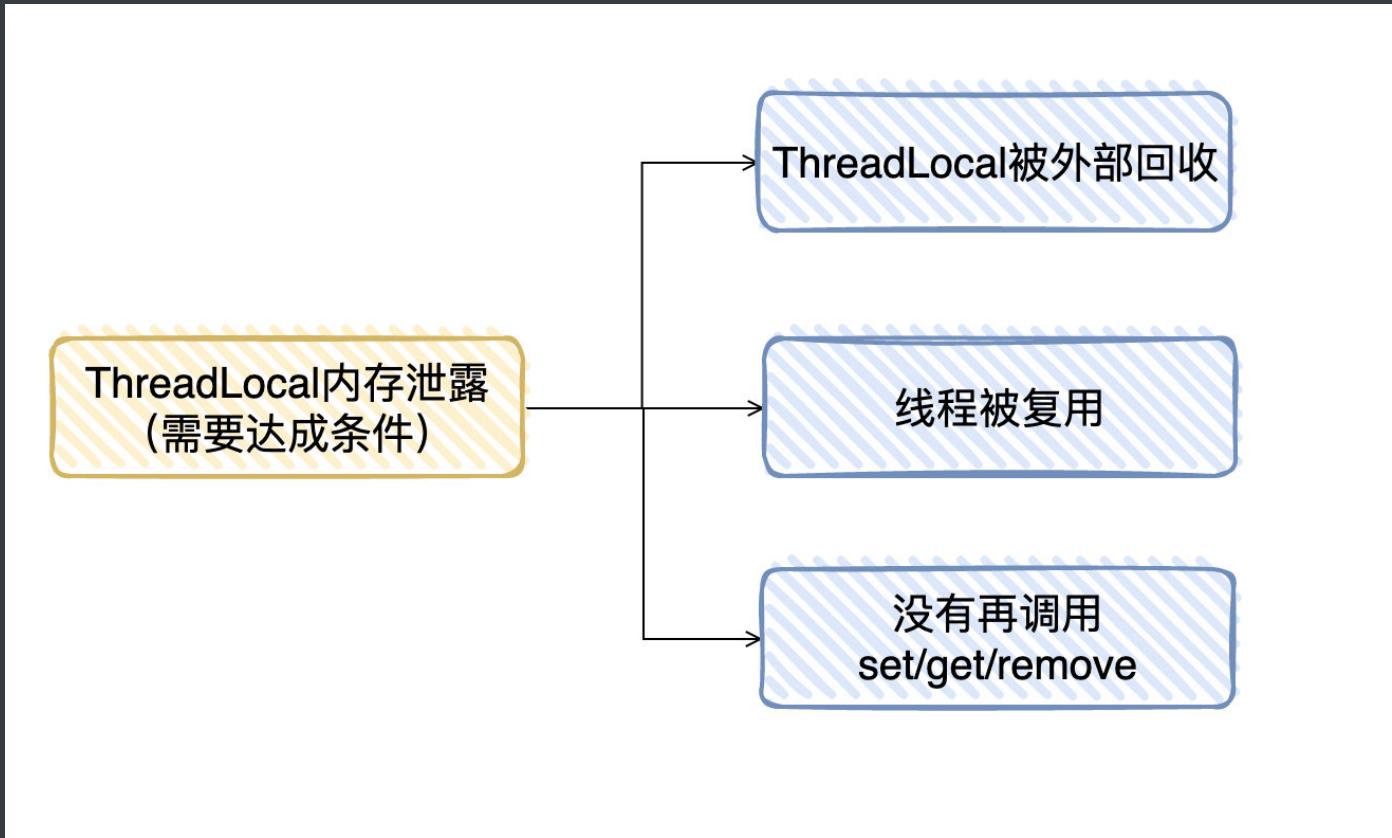
候选者： 那非线程池环境下，也不会有长期性的内存泄露问题

候选者： 而ThreadLocal实现下还做了些“保护”措施，如果在操作ThreadLocal时，发现key为null，会将其清除掉

候选者： 所以，如果在线程池（线程复用）环境下，如果还会调用ThreadLocal的set/get/remove方法

候选者： 发现key为null会进行清除，不会有长期性的内存泄露问题

候选者： 那存在长期性内存泄露需要满足条件：ThreadLocal被回收&&线程被复用&&线程复用后不再调用ThreadLocal的set/get/remove方法



候选者： 使用ThreadLocal的最佳实践就是：用完了，手动remove掉。就像使用Lock加锁后，要记得解锁

面试官： 那我想问下，为什么要将ThreadLocalMap的key设置为弱引用呢？强引用不香吗？

候选者： 外界是通过ThreadLocal来对ThreadLocalMap进行操作的，假设外界使用ThreadLocal的对象被置null了，那ThreadLocalMap的强引用指向ThreadLocal也毫无意义啊。

候选者： 弱引用反而可以预防大多数内存泄漏的情况

候选者： 毕竟被回收后，下一次调用set/get/remove时ThreadLocal内部会清除掉

面试官： 我看网上有很多人说建议把ThreadLocal修饰为static，为什么？

候选者： ThreadLocal能实现了线程的数据隔离，不在于它自己本身，而在于Thread的ThreadLocalMap

候选者： 所以，ThreadLocal可以只初始化一次，只分配一块存储空间就足以了，没必要作为成员变量多次被初始化。

面试官：最后想问个问题：什么叫做内存泄露？

候选者：.....

候选者：意思就是：你申请完内存后，你用完了但没有释放掉，你自己没法用，系统又没法回收。

面试官：清楚了

本文总结：

- **什么是ThreadLocal**：它提供了线程的局部变量，每个线程都可以通过set/get来对这个局部变量进行操作，不会和其他线程的局部变量进行冲突，实现了线程的数据隔离。
- **ThreadLocal实际用处（举例）**：Spring事务，ThreadLocal里存储Map，Key是DataSource，Value是Connection
- **ThreadLocal设计**：Thread有ThreadLocalMap引用，ThreadLocal作为ThreadLocalMap的Key，set和get进去的Value则是ThreadLocalMap的value
- **ThreadLocal内存泄露**：ThreadLocal被回收&&线程被复用&&线程复用后不再调用ThreadLocal的set/get/remove方法 才可能发生内存泄露（条件还是相对苛刻）
- **ThreadLocal最佳实践**：用完就要remove掉



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



07、CountDownLatch和CyclicBarrier

面试官：今天来聊聊并发相关的场景问题吧？

候选者：嗯？ 你说

面试官：现在我有50个任务，这50个任务在完成之后，才能执行下一个「函数」，要是你，你怎么设计？

候选者：嗯，我想想哈。

候选者：可以用JDK给我们提供的线程工具类，CountDownLatch和CyclicBarrier都可以完成这个需求。

候选者：这两个类都可以等到线程完成之后，才去执行某些操作

CountDownLatch

CyclicBarrier

线程同步类工具

面试官：那既然都能实现的话？那CountDownLatch和CyclicBarrier有什么区别呢？

候选者：主要的区别就是CountDownLatch用完了，就结束了，没法复用。而CyclicBarrier不一样，它可以复用。

面试官：那如果是这样的话，那我多次用CountDownLatch不也可以解决问题吗？

候选者：....

面试官：要不今天面试就到这里就结束了？你有什么想问我的吗？

候选者：....

候选者：念在我发了这么多次红包的份上，要不来讲讲为什么这次把我挂了？

面试官：是这样的，我提出了个场景，它确实很像可以用CountDownLatch和CyclicBarrier解决

面试官：但是，作为面试者的你可以尝试向我获取更多的信息

面试官：我可没说一个任务就用一个线程处理哦

面试官：放一步讲，即便我是想考察CountDownLatch和CyclicBarrier的知识

面试官：但是过程也是很重要的，我会看你思考的以及沟通的过程

候选者：...

面试官：你提到了CountDownLatch和CyclicBarrier这些关键词，不能就直接就抛给我

面试官：我是希望你能讲下什么是CountDownLatch和CyclicBarrier分别是什么意思

面试官：比如说：CountDownLatch和CyclicBarrier都是线程同步的工具类

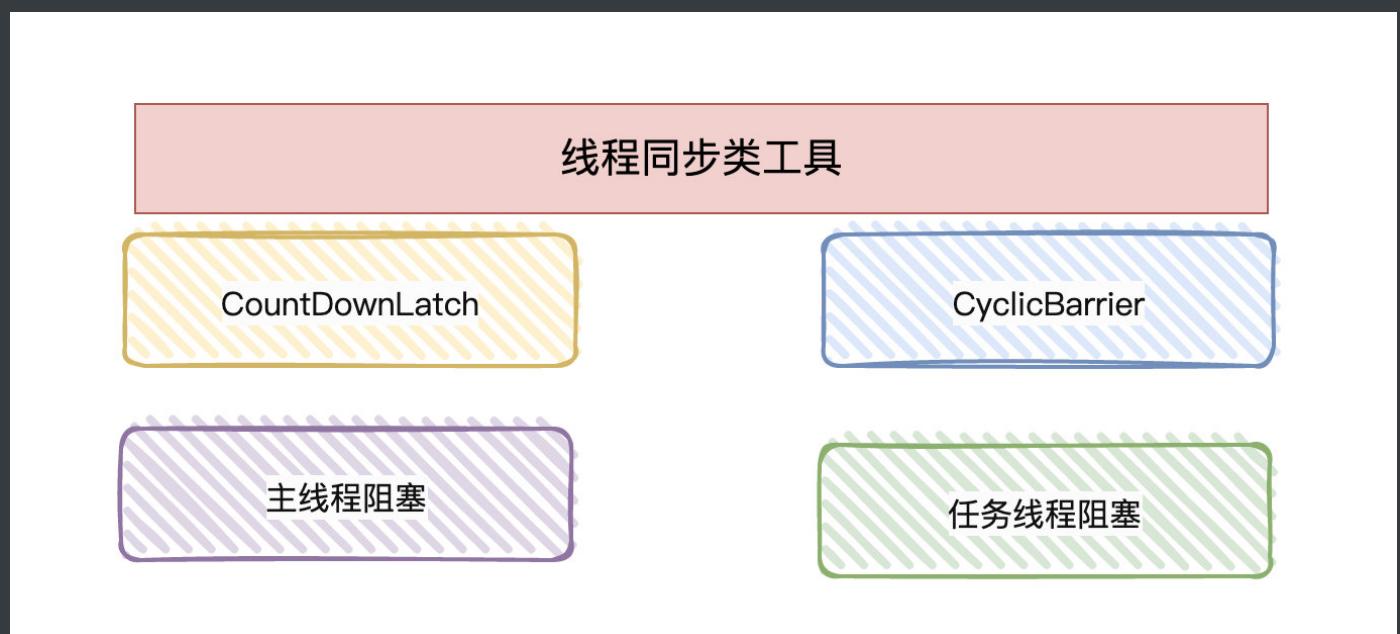
面试官：CountDownLatch允许一个或多个线程一直等待，直到这些线程完成它们的操作

面试官：而CyclicBarrier不一样，它往往是当线程到达某状态后，暂停下来等待其他线程，等到所有线程均到达以后，才继续执行

面试官：可以发现这两者的等待主体是不一样的。

面试官：CountDownLatch调用await()通常是主线程/调用线程，而CyclicBarrier调用await()是在任务线程调用的

面试官：所以，CyclicBarrier中的阻塞的是任务的线程，而主线程是不受影响的。



候选者：....

面试官：简单叙述完这些基本概念后，可以特意抛出这两个类都是基于AQS实现的

面试官：反正你在前几次面试的过程中都说过AQS了，我知道你是懂的，你可以抛出来

面试官：至于问不问，我可能会问，也可能会不问嘛，但问的概率还是挺大的。

候选者：草，学到了

面试官：其实我在问CountDownLatch和CyclicBarrier有什么什么区别的时候，你就围绕源码可以给我讲讲

面试官：而不是随便说CountDownLatch是一次性的，而CyclicBarrier可在完成时重置进而重复使用就来敷衍我

面试官：比如说CountDownLatch你就可以回答：前面提到了CountDownLatch也是基于AQS实现的，它的实现机制很简单

面试官：当我们在构建CountDownLatch对象时，传入的值其实就会赋值给 AQS 的关键变量 state

面试官：执行countDown方法时，其实就是利用CAS 将state 减一

面试官：执行await方法时，其实就是判断state是否为0，不为0则加入到队列中，将该线程阻塞掉（除了头结点）

面试官：因为头节点会一直自旋等待state为0，当state为0时，头节点把剩余的在队列中阻塞的节点也一并唤醒。

面试官：是不是经过解释后，就会让人觉得清晰很多？

候选者：你说得对

面试官：回到CyclicBarrier上，代码也不难，重点就是await方法

面试官：从源码不难发现的是，它没有像CountDownLatch和ReentrantLock使用AQS的state 变量，而CyclicBarrier是直接借助ReentrantLock加上Condition 等待唤醒的功能 进而实现的

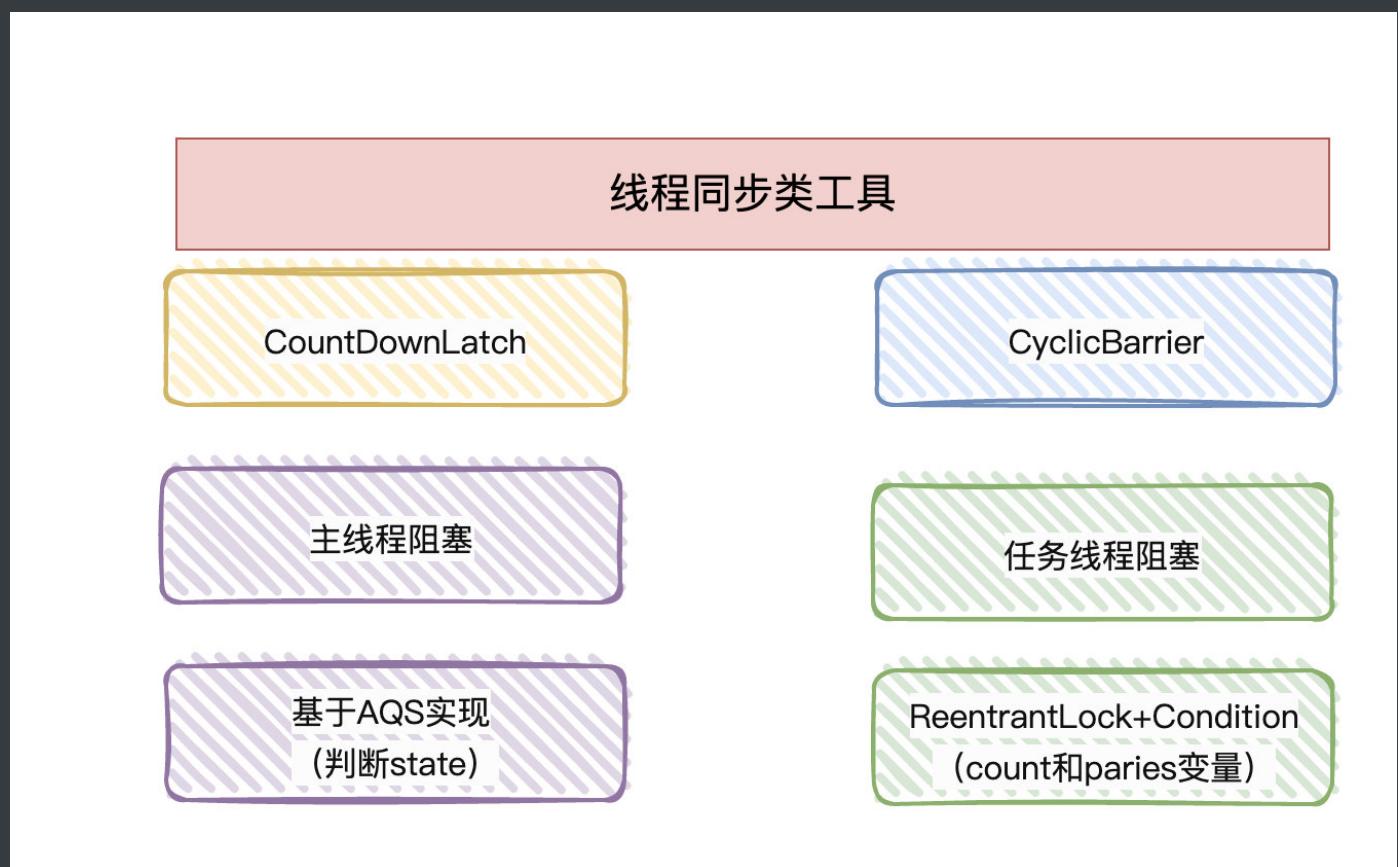
面试官：在构建CyclicBarrier时，传入的值会赋值给CyclicBarrier内部维护count变量，也会赋值给parties变量（这是可以复用的关键）

面试官：每次调用await时，会将count -1，操作count值是直接使用ReentrantLock来保证线程安全性

面试官：如果count不为0，则添加到condition队列中

面试官：如果count等于0时，则把节点从condition队列添加至AQS的队列中进行全部唤醒，并且将parties的值重新赋值为count的值（实现复用）

面试官：是不是不难？



面试官：再简单总结下：CountDownLatch基于AQS实现，会将构造CountDownLatch的入参传递至state，countDown()就是在利用CAS将state减1，await()实际就是让头节点一直在等待state为0时，释放所有等待的线程

面试官：而CyclicBarrier则利用ReentrantLock和Condition，自身维护了count和parties变量。每次调用await将count-1，并将线程加入到condition队列上。等到count为0时，则将condition队列的节点移交至AQS队列，并全部释放。

面试官：等你讲完这一套东西时，时间已经过了好几分钟了

面试官：一般我也不会用一个问题探讨太久，觉得还可以就问下一个问题了。

候选者：学到了



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zhongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

08、为什么需要Java内存模型

面试官：今天想跟你聊聊Java内存模型，这块你了解过吗？

候选者：嗯，我简单说下我的理解吧。那我就从为什么要有Java内存模型开始讲起吧

面试官：开始你的表演吧。

候选者：那我先说下背景吧

候选者: 1. 现有计算机往往是多核的，每个核心下会有高速缓存。高速缓存的诞生是由于「CPU与内存(主存)的速度存在差异」，L1和L2缓存一般是「每个核心独占」一份的。

候选者: 2. 为了让CPU提高运算效率，处理器可能会对输入的代码进行「乱序执行」，也就是所谓的「指令重排序」

候选者: 3. 一次对数值的修改操作往往是非原子性的（比如`i++`实际上在计算机执行时就会分成多个指令）

候选者: 在永远单线程下，上面所讲的均不会存在什么问题，因为单线程意味着无并发。并且在单线程下，编译器/runtime/处理器都必须遵守as-if-serial语义，遵守as-if-serial意味着它们不会对「数据依赖关系的操作」做重排序。

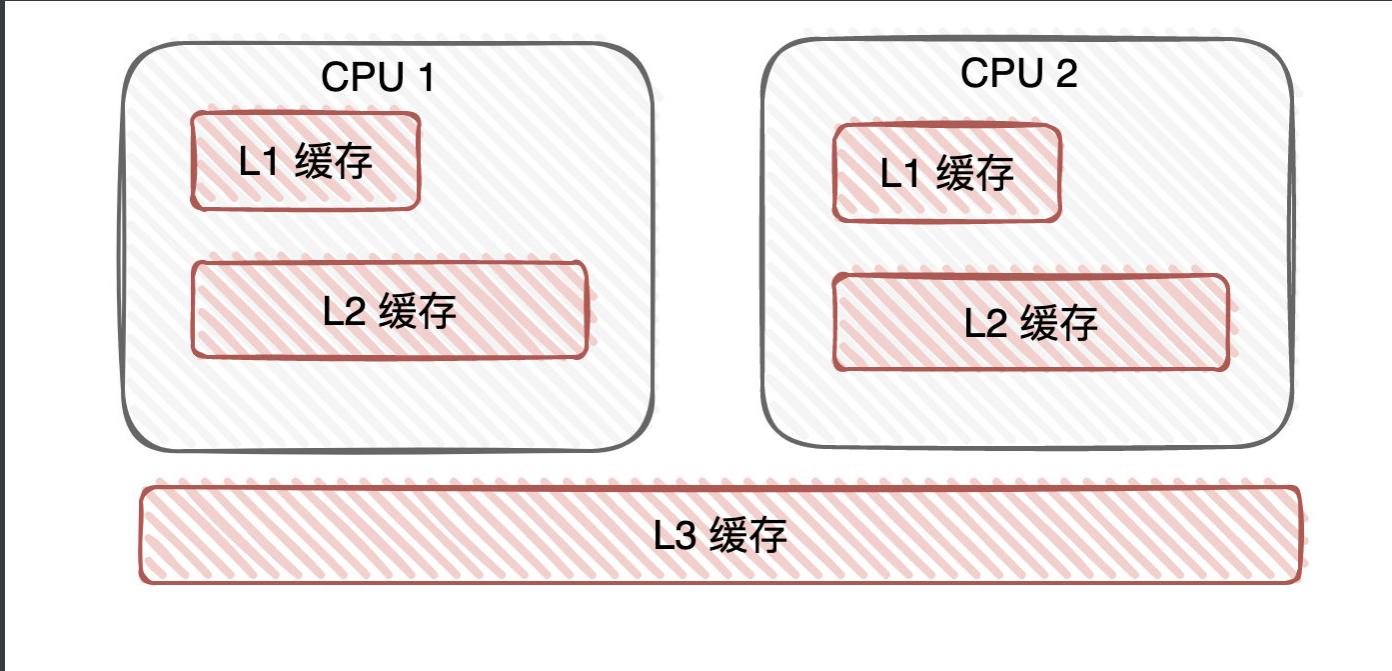


候选者: CPU为了效率，有了高速缓存、有了指令重排序等等，整块架构都变得复杂了。我们写的程序肯定也想要「充分」利用CPU的资源啊！于是乎，我们使用起了多线程

候选者: 多线程在意味着并发，并发就意味着我们需要考虑线程安全问题

候选者: 1. 缓存数据不一致：多个线程同时修改「共享变量」，CPU核心下的高速缓存是「不共享」的，那多个cache与内存之间的数据同步该怎么做？

候选者: 2. CPU指令重排序在多线程下会导致代码在非预期下执行，最终会导致结果存在错误的情况。



候选者: 针对于「缓存不一致」问题，CPU也有其解决办法，常被大家所认识的有两种：

候选者: 1. 使用「总线锁」：某个核心在修改数据的过程中，其他核心均无法修改内存中的数据。（类似于独占内存的概念，只要有CPU在修改，那别的CPU就得等待当前CPU释放）

候选者: 2. 缓存一致性协议（MESI协议，其实协议有很多，只是举个大家都可能见过的）。MESI拆开英文是（Modified（修改状态）、Exclusive（独占状态）、Share（共享状态）、Invalid（无效状态））

候选者: 缓存一致性协议我认为可以理解为「缓存锁」，它针对的是「缓存行」(Cache line)进行“加锁”，所谓「缓存行」其实就是高速缓存存储的最小单位。



粒度不一样

面试官：嗯...

候选者：MESI协议的原理大概就是：当每个CPU读取共享变量之前，会先识别数据的「对象状态」(是修改、还是共享、还是独占、还是无效)。

候选者：如果是独占，说明当前CPU将要得到的变量数据是最新的，没有被其他CPU所同时读取

候选者：如果是共享，说明当前CPU将要得到的变量数据还是最新的，有其他的CPU在同时读取，但还没被修改

候选者：如果是修改，说明当前CPU正在修改该变量的值，同时会向其他CPU发送该数据状态为invalid(无效)的通知，得到其他CPU响应后（其他CPU将数据状态从共享(share)变成invalid(无效)），会当前CPU将高速缓存的数据写到主存，并把自己的状态从modify(修改)变成exclusive(独占)

候选者：如果是无效，说明当前数据是被改过了，需要从主存重新读取最新的数据。

识别"对象"状态
(MESI) Modify-Exclusive-Share-Invalid
不同的状态对应不同的处理

候选者：其实MESI协议做的就是判断「对象状态」，根据「对象状态」做不同的策略。关键就在于某个CPU在对数据进行修改时，需要「同步」通知其他CPU，表示这个数据被我修改了，你们不能用了。

候选者：比较于「总线锁」，MESI协议的"锁粒度"更小了，性能那肯定会更高咯

面试官：但据我了解，CPU还有优化，你还知道吗？

候选者：嗯，还是了解那么一点点的。

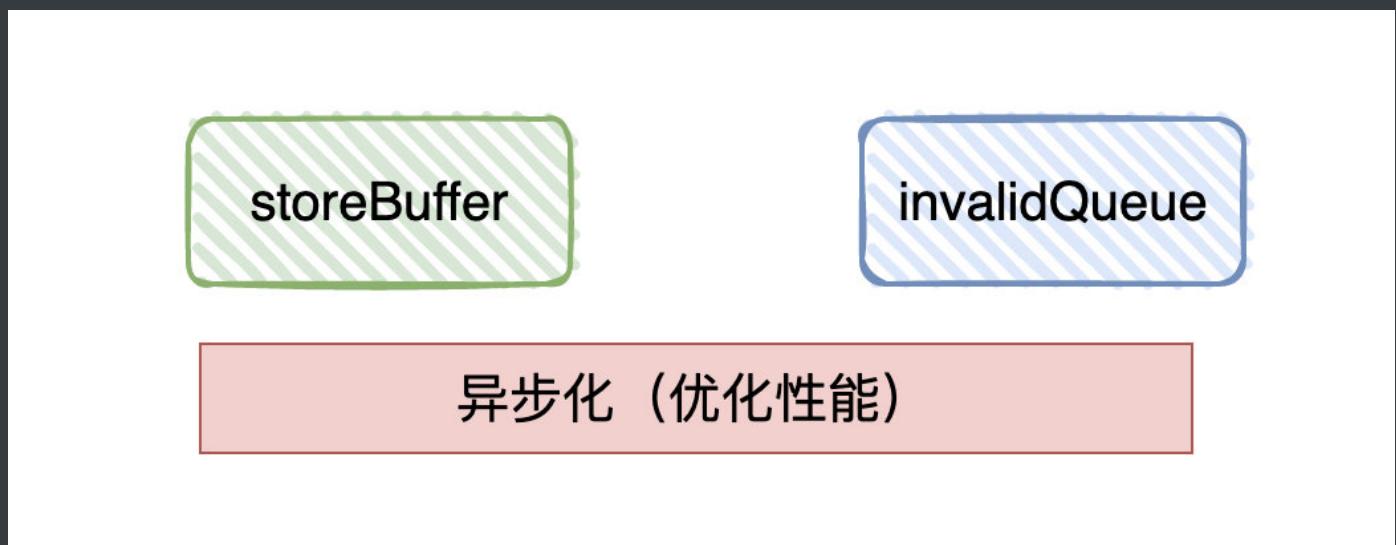
候选者：从前面讲到的，可以发现的是：当CPU修改数据时，需要「同步」告诉其他的CPU，等待其他CPU响应接收到invalid(无效)后，它才能将高速缓存数据写到主存。

候选者：同步，意味着等待，等待意味着什么都干不了。CPU肯定不乐意啊，所以又优化了一把。

候选者：优化思路就是从「同步」变成「异步」。

候选者：在修改时会「同步」告诉其他CPU，而现在则把最新修改的值写到「store buffer」中，并通知其他CPU记得要改状态，随后CPU就直接返回干其他事了。等到收到其它CPU发过来的响应消息，再将数据更新到高速缓存中。

候选者：其他CPU接收到invalid(无效)通知时，也会把接收到的消息放入「invalid queue」中，只要写到「invalid queue」就会直接返回告诉修改数据的CPU已经将状态置为「invalid」



候选者：而异步又会带来新问题：那我现在CPU修改完A值，写到「store buffer」了，CPU就可以干其他事了。那如果该CPU又接收指令需要修改A值，但上一次修改的值还在「store buffer」中呢，没修改至高速缓存呢。

候选者: 所以CPU在读取的时候，需要去「store buffer」看看存不存在，存在则直接取，不存在才读主存的数据。【Store Forwarding】

候选者: 好了，解决掉第一个异步带来的问题了。（相同的核心对数据进行读写，由于异步，很可能会导致第二次读取的还是旧值，所以首先读「store buffer」）。

面试官: 还有其他？

候选者: 那当然啊，那「异步化」会导致相同核心读写共享变量有问题，那当然也会导致「不同」核心读写共享变量有问题啊

候选者: CPU1修改了A值，已把修改后值写到「store buffer」并通知CPU2对该值进行invalid(无效)操作，而CPU2可能还没收到invalid(无效)通知，就去做了其他的操作，导致CPU2读到的还是旧值。

候选者: 即便CPU2收到了invalid(无效)通知，但CPU1的值还没写到主存，那CPU2再次向主存读取的时候，还是旧值...

候选者: 变量之间很多时候是具有「相关性」($a=1;b=0;b=a$)，这对于CPU又是无感知的...

候选者: 总体而言，由于CPU对「缓存一致性协议」进行的异步优化「store buffer」「invalid queue」，很可能导致后面的指令很可能查不到前面指令的执行结果（各个指令的执行顺序非代码执行顺序），这种现象很多时候被称作「CPU乱序执行」

候选者: 为了解决乱序问题（也可以理解为可见性问题，修改完没有及时同步到其他的CPU），又引出了「内存屏障」的概念。

异步化（优化性能）

storeBuffer

invalidQueue

异步化又导致缓存一致性问题

面试官：嗯...

候选者：「内存屏障」其实就是为了解决「异步优化」导致「CPU乱序执行」 / 「缓存不及时可见」的问题，那怎么解决的呢？嗯，就是把「异步优化」给“禁用”掉（：

候选者：内存屏障可以分为三种类型：写屏障，读屏障以及全能屏障（包含了读写屏障），屏障可以简单理解为：在操作数据的时候，往数据插入一条“特殊的指令”。只要遇到这条指令，那前面的操作都得「完成」。

候选者：那写屏障就可以这样理解：CPU当发现写屏障的指令时，会把该指令「之前」存在于「store Buffer」所有写指令刷入高速缓存。

候选者：通过这种方式就可以让CPU修改的数据可以马上暴露给其他CPU，达到「写操作」可见性的效果。

候选者：那读屏障也是类似的：CPU当发现读屏障的指令时，会把该指令「之前」存在于「invalid queue」所有的指令都处理掉

候选者：通过这种方式就可以确保当前CPU的缓存状态是准确的，达到「读操作」一定是读取最新的效果。

内存屏障

特殊的指令
(刷storeBuffer、刷invalidQueue)
"禁用"异步优化

候选者：由于不同CPU架构的缓存体系不一样、缓存一致性协议不一样、重排序的策略不一样、所提供的内存屏障指令也有差异，为了简化Java开发人员的工作。Java封装了一套规范，这套规范就是「Java内存模型」

候选者：再详细地说，「Java内存模型」希望 屏蔽各种硬件和操作系统的访问差异，保证了Java程序在各种平台下对内存的访问都能得到一致效果。目的是解决多线程存在的原子性、可见性（缓存一致性）以及有序性问题。

Java内存模型

Java为了屏蔽硬件和操作系统访问内存的各种差异，对内存的访问都能得到一致效果

面试官：那要不简单聊聊Java内存模型的规范和内容吧？

候选者：不了，怕一聊就是一个下午，下次吧？

本文总结：

- 并发问题产生的三大根源是「可见性」 「有序性」 「原子性」
- 可见性：CPU架构下存在高速缓存，每个核心下的L1/L2高速缓存不共享（不可见）
- 有序性：主要有三方面可能导致打破
 - 编译器优化导致重排序（编译器可以在不改变单线程程序语义的情况下，可以对代码语句顺序进行调整重新排序）
 - 指令集并行重排序（CPU原生就有可能将指令进行重排）
 - 内存系统重排序（CPU架构下很可能有store buffer /invalid queue 缓冲区，这种「异步」很可能会导致指令重排）
- 原子性：Java的一条语句往往需要多条CPU指令完成(i++），由于操作系统的线程切换很可能导致 i++ 操作未完成，其他线程“中途”操作了共享变量 i，导致最终结果并非我们所期待的。
- 在CPU层级下，为了解决「缓存一致性」问题，有相关的“锁”来保证，比如“总线锁”和“缓存锁”。
 - 总线锁是锁总线，对共享变量的修改在相同的时刻只允许一个CPU操作。
 - 缓存锁是锁缓存行(cache line)，其中比较出名的是MESI协议，对缓存行标记状态，通过“同步通知”的方式，来实现(缓存行)数据的可见性和有序性
 - 但“同步通知”会影响性能，所以会有内存缓冲区(store buffer/invalid queue)来实现「异步」进而提高CPU的工作效率
 - 引入了内存缓冲区后，又会存在「可见性」和「有序性」的问题，平日大多数情况下是可以享受「异步」带来的好处的，但少数情况下，需要强「可见性」和「有序性」，只能“禁用”缓存的优化。
 - “禁用”缓存优化在CPU层面下有「内存屏障」，读屏障/写屏障/全能屏障，本质上是插入一条“屏障指令”，使得缓冲区(store buffer/invalid queue)在屏障指令之前的操作均已被处理，进而达到 读写 在CPU层面上是可见和有序的。
- 不同的CPU实现的架构和优化均不一样，Java为了屏蔽硬件和操作系统访问内存的各种差异，提出了「Java内存模型」的规范，保证了Java程序在各种平台下对内存的访问都能得到一致效果



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「**对线面试官**」



09、深入浅出Java内存模型

面试官：我记得上一次已经问过了为什么要有Java内存模型

面试官：我记得你的最终答案是：Java为了屏蔽硬件和操作系统访问内存的各种差异，提出了「Java内存模型」的规范，保证了Java程序在各种平台下对内存的访问都能得到一致效果

候选者：嗯，对的

面试官：要不，你今天再来讲讲Java内存模型这里边的内容呗？

候选者：嗯，在讲之前还是得强调下：Java内存模型它是一种「规范」，Java虚拟机会实现这个规范。

候选者：Java内存模型主要的内容，我个人觉得有以下几块吧

候选者：1. Java内存模型的抽象结构

候选者：2. happen-before规则

候选者：3. 对volatile内存语义的探讨（这块我后面再好好解释）

Java内存模型的抽象结构

happen-before规则

volatile语义

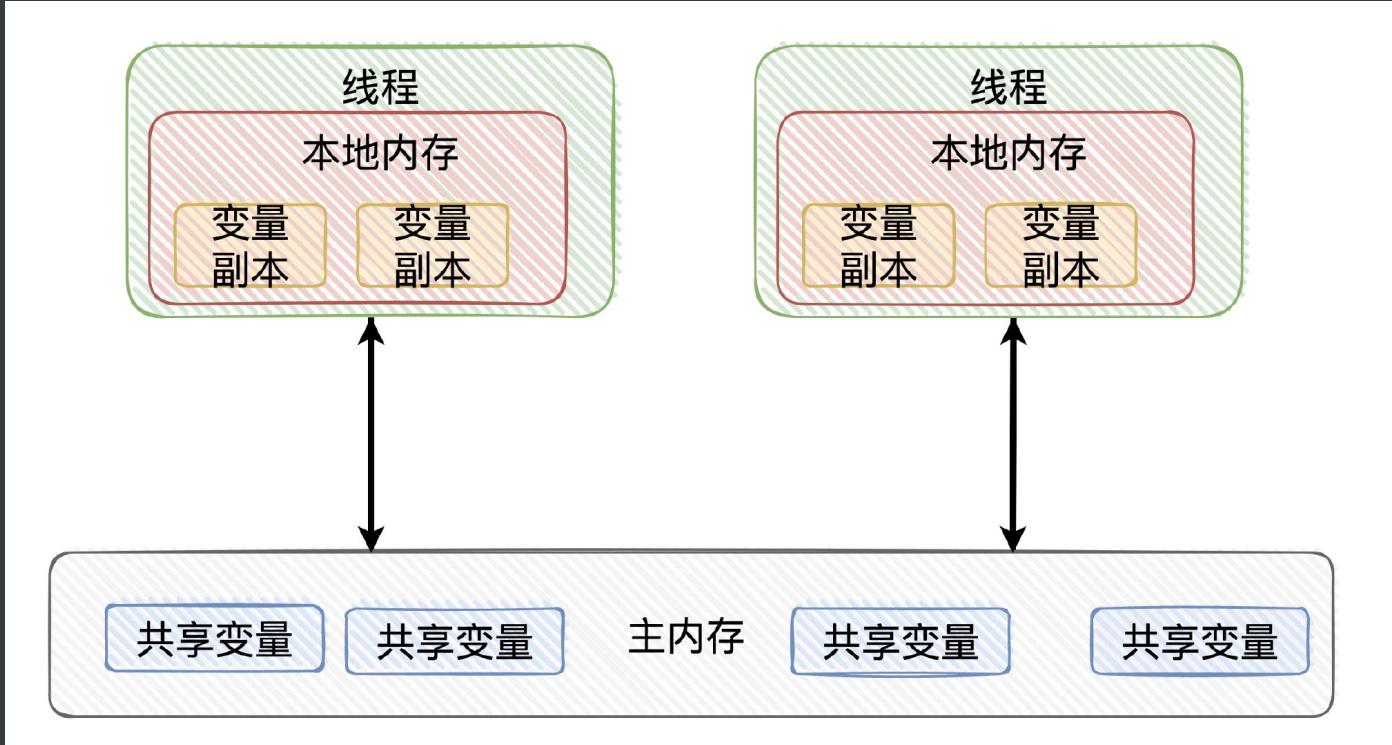
面试官：那要不你就从第一点开始呗？先聊下Java内存模型的抽象结构？

候选者：嗯。Java内存模型定义了：Java线程对内存数据进行交互的规范。

候选者：线程之间的「共享变量」存储在「主内存」中，每个线程都有自己私有的「本地内存」，「本地内存」存储了该线程以读/写共享变量的副本。

候选者：本地内存是Java内存模型的抽象概念，并不是真实存在的。

候选者：顺便画个图吧，看完图就懂了。

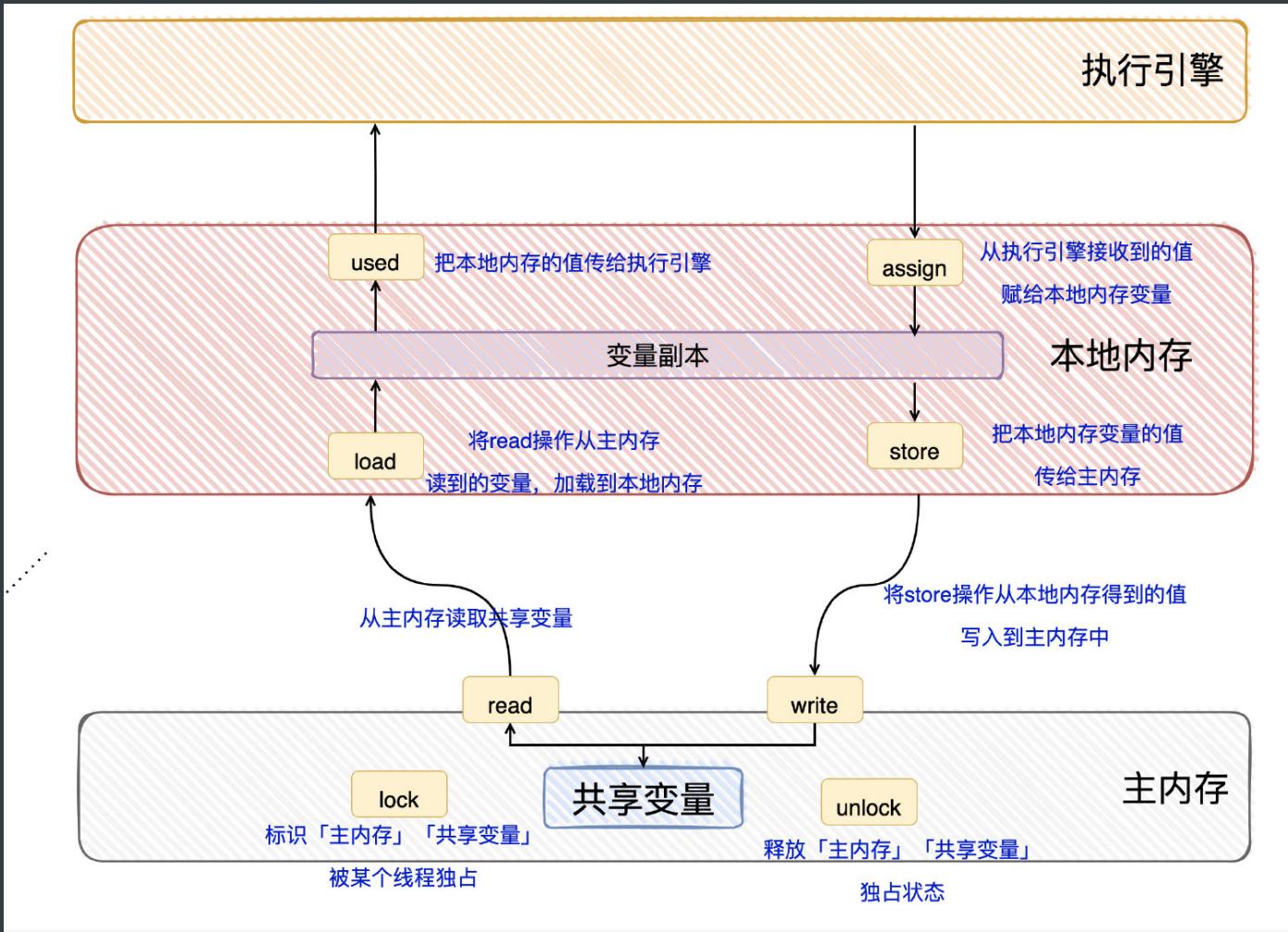


候选者：Java内存模型规定了：线程对变量的所有操作都必须在「本地内存」进行，「不能直接读写主内存」的变量

候选者：Java内存模型定义了8种操作来完成「变量如何从主内存到本地内存，以及变量如何从本地内存到主内存」

候选者：分别是read/load/use/assign/store/write/lock/unlock操作

候选者：看着8个操作很多，对变量的一次读写就涵盖了这些操作了，我再画个图给你讲讲



候选者：懂了吧？无非就是读写用到了各个操作（：

面试官：懂了，很简单，接下来说什么是happen-before吧？

候选者：嗯，好的（：

候选者：按我的理解下，happen-before实际上也是一套「规则」。Java内存模型定义了这套规则，目的是为了阐述「操作之间」的内存「可见性」

候选者：从上次讲述「指令重排」就提到了，在CPU和编译器层面上都有指令重排的问题。

候选者：指令重排虽然是能提高运行的效率，但在并发编程中，我们在兼顾「效率」的前提下，还希望「程序结果」能由我们掌控的。

候选者：说白了就是：在某些重要的场景下，这一组操作都不能进行重排序，「前面一个操作的结果对后续操作必须是可见的」。

happen-before规则

前一个操作的结果对后续的操作必须是可见的

传递性

监视器锁

volatile

....

面试官：嗯...

候选者：于是，Java内存模型就提出了happen-before这套规则，规则总共有8条

候选者：比如传递性、volatile变量规则、程序顺序规则、监视器锁的规则...（具体看规则的含义就好了，这块不难）

候选者：只要记住，有了happen-before这些规则。我们写的代码只要在这些规则下，前一个操作的结果对后续操作是可见的，是不会发生重排序的。

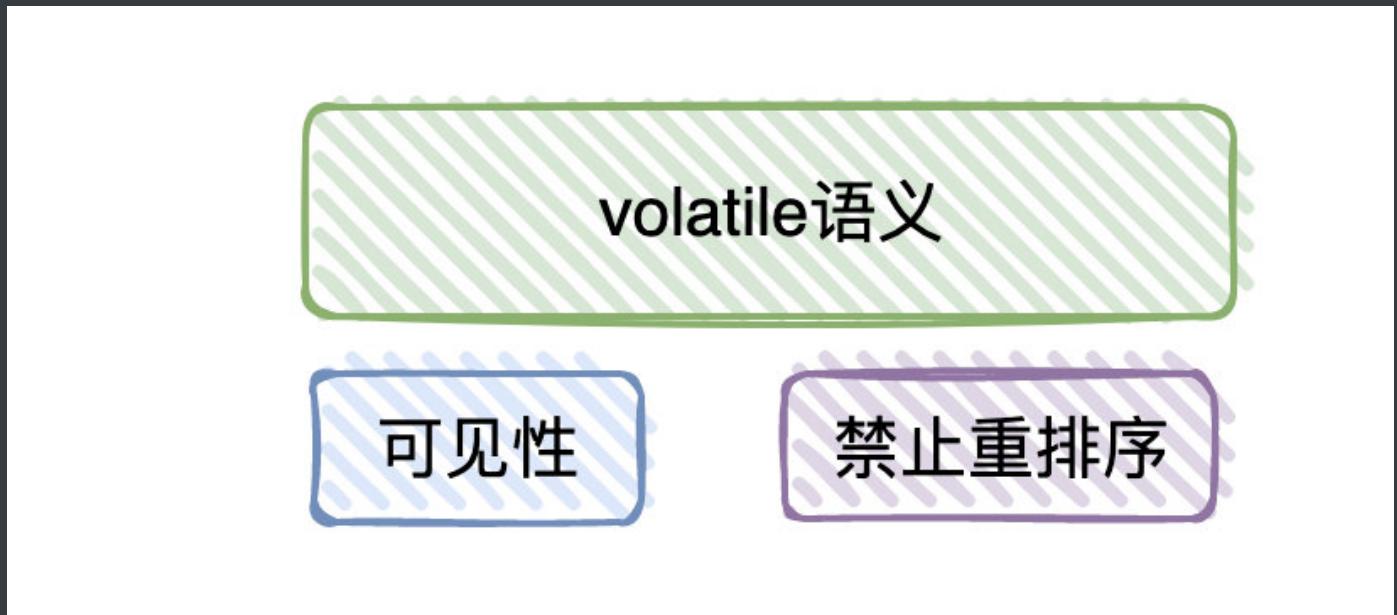
面试官：我明白你的意思了

面试官：那最后说下volatile？

候选者：嗯，volatile是Java的一个关键字

候选者：为什么讲Java内存模型往往就会讲到volatile这个关键字呢，我觉得主要是它的特性：可见性和有序性(禁止重排序)

候选者: Java内存模型这个规范，很大程度下就是为了解决可见性和有序性的问题。



面试官: 那你来讲讲它的原理吧，**volatile**这个关键字是怎么做到可见性和有序性的

候选者: Java内存模型为了实现volatile有序性和可见性，定义了4种内存屏障的「规范」，分别是LoadLoad/LoadStore/StoreLoad/StoreStore

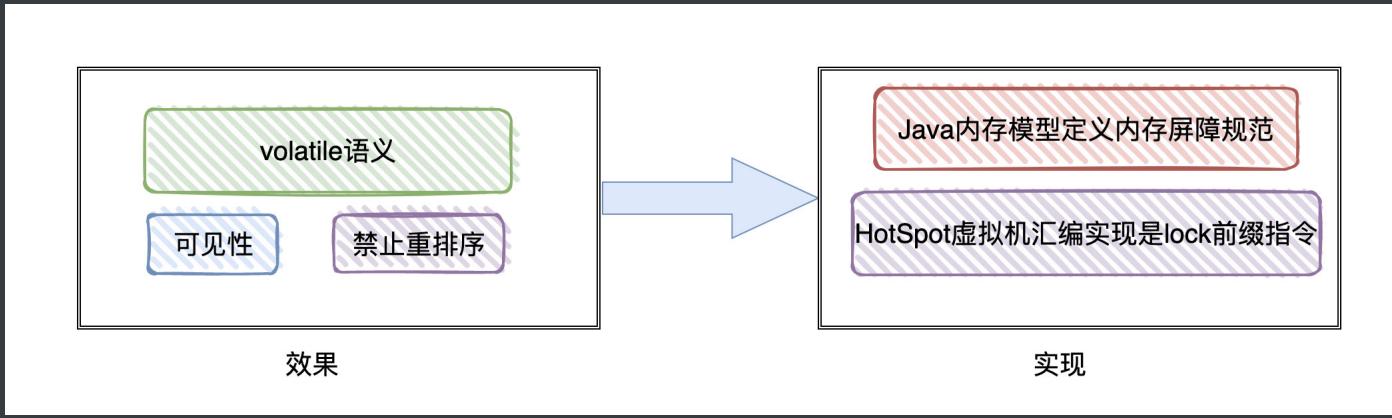
候选者: 回到volatile上，说白了，就是在volatile「前后」加上「内存屏障」，使得编译器和CPU无法进行重排序，致使有序，并且写volatile变量对其他线程可见。

候选者: Java内存模型定义了规范，那Java虚拟机就得实现啊，是不是？

面试官: 嗯...

候选者: 之前看过Hotspot虚拟机的实现，在「汇编」层面上实际是通过Lock前缀指令来实现的，而不是各种fence指令（主要原因就是简便。因为大部分平台都支持lock指令，而fence指令是x86平台的）。

候选者: lock指令能保证：禁止CPU和编译器的重排序（保证了有序性）、保证CPU写核心的指令可以立即生效且其他核心的缓存数据失效（保证了可见性）。



面试官：那你提到这了，我想问问volatile和MESI协议是啥关系？

候选者：它们没有直接的关联。

候选者：Java内存模型关注的是编程语言层面上，它是高维度的抽象。MESI是CPU缓存一致性协议，不同的CPU架构都不一样，可能有的CPU压根就没用MESI协议...

候选者：只不过MESI名声大，大家就都拿他来举例子了。而MESI可能只是在「特定的场景下」为实现volatile的可见性/有序性而使用到的一部分罢了（：

面试官：嗯...

候选者：为了让Java程序员屏蔽上面这些底层知识，快速地入门使用volatile变量

候选者：Java内存模型的happen-before规则中就有对volatile变量规则的定义

候选者：这条规则的内容其实就是：对一个 volatile 变量的写操作相对于后续对这个 volatile 变量的读操作可见

候选者：它通过happen-before规则来规定：只要变量声明了volatile关键字，写后再读，读必须可见写的值。（可见性、有序性）

面试官：嗯...了解了

本文总结：

- **为什么存在Java内存模型**：Java为了屏蔽硬件和操作系统访问内存的各种差异，提出了「Java内存模型」的规范，保证了Java程序在各种平台下对内存的访问都能得到一致效果
- **Java内存模型抽象结构**：线程之间的「共享变量」存储在「主内存」中，每个线程都有自己私有的「本地内存」，「本地内存」存储了该线程以读/写共享变量的副本。线程对变量的所有操作都必须在「本地内存」进行，而「不能直接读写主内存」的变量
- **happen-before规则**：Java内存模型规定在某些场景下（一共8条），前面一个操作的结果对后续操作必须是可见的。这8条规则成为happen-before规则
- **volatile**：volatile是Java的关键字，修饰的变量是可见性且有序的（不会被重排序）。可见性由happen-before规则完成，有序性由Java内存模型定义的「内存屏障」完成，实际HotSpot虚拟机实现Java内存模型规范，汇编底层通过Lock指令来实现。



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

03-Java容器

01、List集合

面试官：要不今天来讲讲Java的List吧，你对List了解多少？

候选者：List在Java里边是一个接口，常见的实现类有ArrayList和LinkedList，在开发中用得最多的是ArrayList

候选者：ArrayList的底层数据结构是数组，LinkedList底层数据结构是链表。



面试官：那Java本身就有数组了，为什么要用ArrayList呢？

候选者：原生的数组会有一个特点：你在使用的时候必须要为它创建大小，而ArrayList不用。

候选者：在日常开发的时候，往往我们是不知道数组的大小的

候选者：如果数组的大小指定多了，内存浪费；如果数组大小指定少了，装不下。

候选者：假设我们给定数组的大小是10，要往这个数组里边填充元素，我们只能添加10个元素。

候选者：而ArrayList不一样，ArrayList我们在使用的时候可以往里边添加20个，30个，甚至更多的元素

候选者：因为ArrayList是实现了动态扩容的

ArrayList 可以动态扩容

数组在初始化的时候必须指明大小

候选者：大概的意思就是：

候选者：当我们new ArrayList()的时候，默认会有一个空的Object数组，大小为0。

候选者：当我们第一次add添加数据的时候，会给这个数组初始化一个大小，这个大小默认值为10

候选者：使用ArrayList在每一次add的时候，它都会先去计算这个数组够不够空间

候选者：如果空间是够的，那直接追加上去就好了。如果不夠，那就得扩容

面试官：那怎么扩容？一次扩多少？

候选者：在源码里边，有个grow方法，每一次扩原来的1.5倍。比如说，初始化的值是10嘛。

候选者：现在我第11个元素要进来了，发现这个数组的空间不够了，所以会扩到15

候选者：空间扩完容之后，会调用arraycopy来对数组进行拷贝

扩容本质：数组拷贝

面试官：哦，可以的。

面试官：那为什么你在前面提到，在日常开发中用得最多的是ArrayList呢？

候选者：是由底层的数据结构来决定的，在日常开发中，遍历的需求比增删要多，即便是增删也是往往在List的尾部添加就OK了。

候选者：像在尾部添加元素，ArrayList的时间复杂度也就O(1)

候选者：另外的是，ArrayList的增删底层调用的copyOf()被优化过

候选者：现代CPU对内存可以块操作，ArrayList的增删一点儿也不会比LinkedList慢

ArrayList 很快 (操作系统内存块操作)

面试官：Vector你了解吗？

候选者：嗯，Vector是底层结构是数组，一般现在我们已经很少用了。

候选者：相对于ArrayList，它是线程安全的，在扩容的时候它是直接扩容两倍的

候选者：比如现在有10个元素，要扩容的时候，就会将数组的大小增长到20

Vector 线程安全（基本不用） 扩容是原来的两倍

面试官：嗯，那如果我们不用Vector，线程安全的List还有什么？

候选者：首先，我们也可以用Collections来将ArrayList来包装一下，变成线程安全。

候选者：但这肯定不是你想听的，对吧。在java.util.concurrent包下还有一个类，叫做CopyOnWriteArrayList

候选者：要讲CopyOnWriteArrayList之前，我还是想说说copy-on-write这个意思，下面我会简称为cow。

候选者：比如说在Linux中，我们知道所有的进程都是init进程fork出来的

候选者：除了进程号之外，fork出来的进程，默认跟父进程一模一样的。

候选者：当使用了cow机制；子进程在被fork之后exec之前，两个进程用的是相同的内存空间的

候选者：这意味着子进程的代码段、数据段、堆栈都是指向父进程的物理空间

候选者：当父子进程中有更改的行为发生时，再为子进程分配相应物理空间。

候选者：这样做好处就是，等到真正发生修改的时候，才去分配资源，可以减少分配或者复制大量资源时带来的瞬间延时。

候选者: 简单来说，就可以理解为我们的懒加载，或者说单例模式的懒汉式。等真正用到的时候再分配

CopyOnWrite机制 (写时复制)

面试官: 嗯

候选者: 在文件系统中，其实也有cow的机制。

候选者: 文件系统的cow就是在修改数据的时候，不会直接在原来的数据位置上进行操作，而是重新找个位置修改。

候选者: 比如说：要修改数据块A的内容，先把A读出来，写到B块里面去。

候选者: 如果这时候断电了，原来A的内容还在。这样做的好处就是可以保证数据的完整性，瞬间挂掉了容易恢复。

候选者: 再回头来看CopyOnWriteArrayList吧，CopyOnWriteArrayList是一个线程安全的List，底层是通过复制数组的方式来实现的。

候选者: 我来说说它的add()方法的实现吧

面试官: 好

候选者: 在add()方法其实他会加lock锁，然后会复制出一个新的数组，往新的数组里边add真正的元素，最后把array的指向改变为新的数组

候选者: get()方法又或是size()方法只是获取array所指向的数组的元素或者大小。读不加锁，写加锁

候选者: 可以发现的是，CopyOnWriteArrayList跟文件系统的COW机制是很像的

CopyOnWriteArrayList原理

写时 加lock锁复制数组，替换变量

读不加锁

面试官: 那你能说说CopyOnWriteArrayList有什么缺点吗？

候选者: 很显然，CopyOnWriteArrayList是很耗费内存的，每次set()/add()都会复制一个数组出来

候选者: 另外就是CopyOnWriteArrayList只能保证数据的最终一致性，不能保证数据的实时一致性。

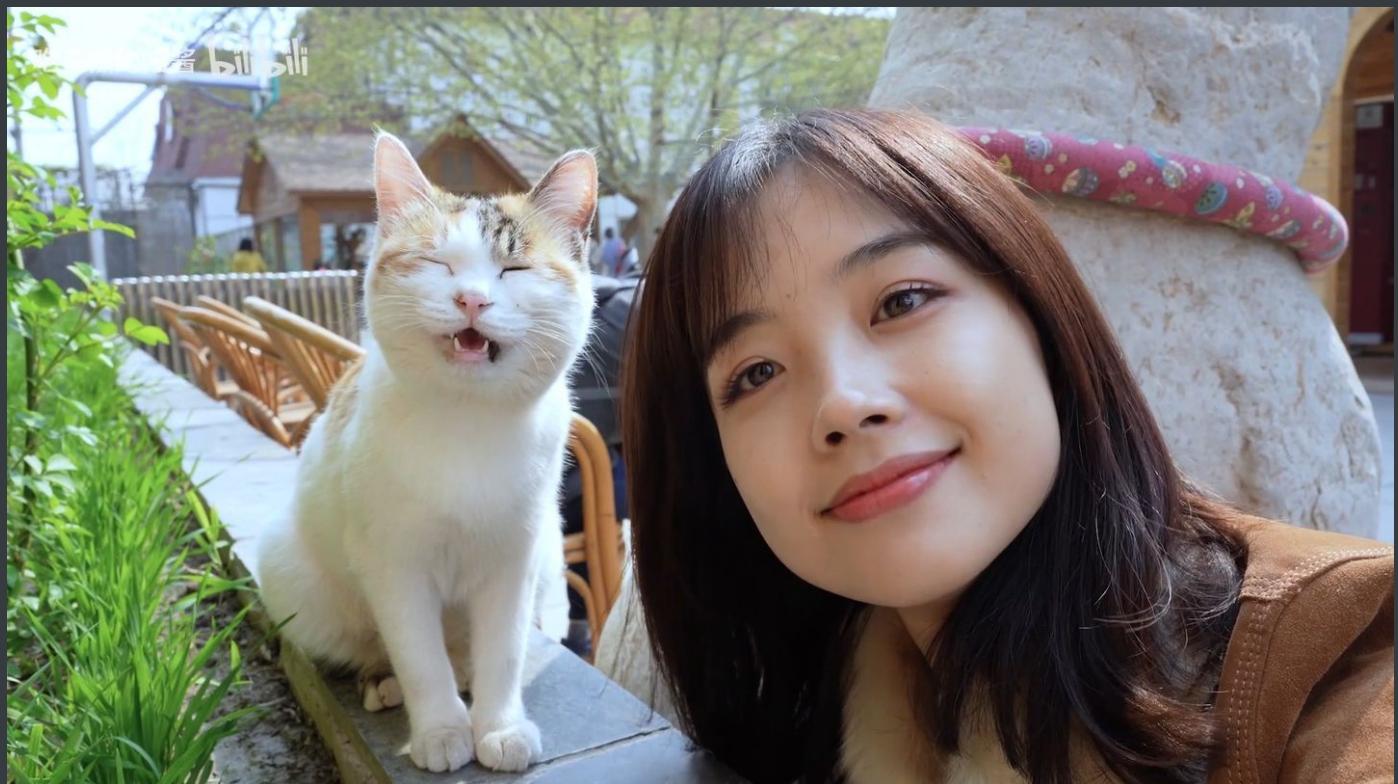
候选者: 假设两个线程，线程A去读取CopyOnWriteArrayList的数据，还没读完

候选者: 现在线程B把这个List给清空了，线程A此时还是可以把剩余的数据给读出来。

只能保证最终一致性

复制数组 耗内存

面试官：嗯，还可以。



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



02、Map集合

面试官：今天来讲讲**Map**吧，你对**Map**了解多少？就讲**JDK 1.8**就好咯

候选者：Map在Java里边是一个接口，常见的实现类有HashMap、LinkedHashMap、TreeMap和ConcurrentHashMap

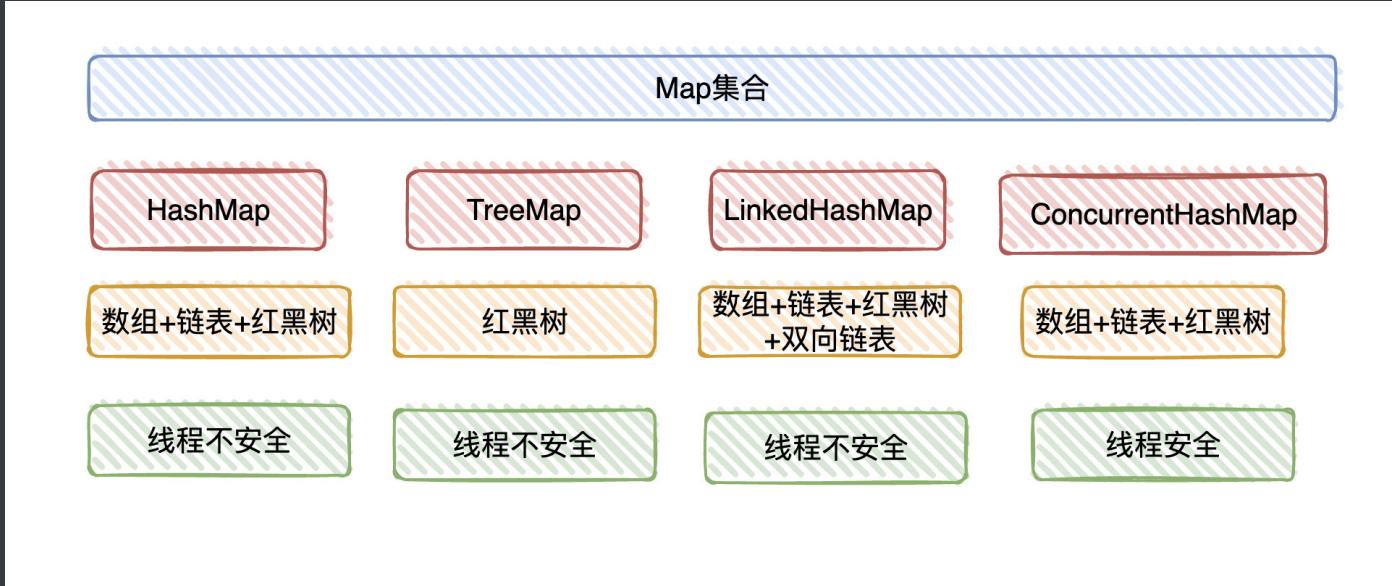
候选者：在Java里边，哈希表的结构是数组+链表的方式。

候选者：HashMap底层数据结构是数组+链表/红黑树

候选者：LinkedHashMap底层数据结构是数组+链表/红黑树+双向链表

候选者：TreeMap底层数据结构是红黑树

候选者：而ConcurrentHashMap底层数据结构也是数组+链表/红黑树



面试官：我们先以HashMap开始吧，你能讲讲当你new一个HashMap的时候，会发生什么吗？

候选者：HashMap有几个构造方法，但最主要的就是指定初始值大小和负载因子的大小。

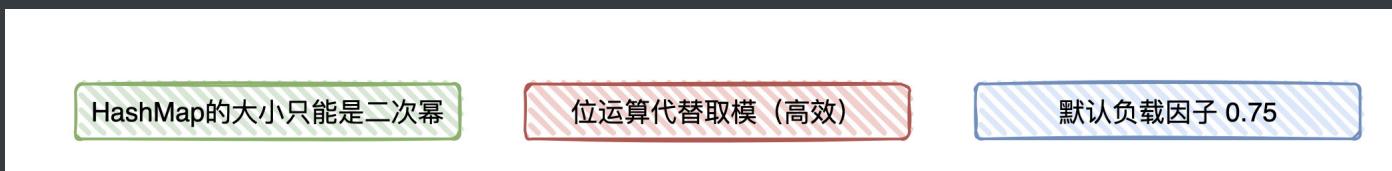
候选者：如果我们不指定，默认HashMap的大小为16，负载因子的大小为0.75

候选者：HashMap的大小只能是2次幂的，假设你传一个10进去，实际上最终HashMap的大小是16，你传一个7进去，HashMap最终的大小是8，具体的实现在tableSizeFor可以看到。

候选者：我们把元素放进HashMap的时候，需要算出这个元素所在的位置（hash）。

候选者：在HashMap里用的是位运算来代替取模，能够更加高效地算出该元素所在的位置。

候选者：为什么HashMap的大小只能是2次幂，因为只有大小为2次幂时，才能合理用位运算替代取模。



候选者: 而负载因子的大小决定着哈希表的扩容和哈希冲突。

候选者: 比如现在我默认的HashMap大小为16, 负载因子为0.75, 这意味着数组最多只能放12个元素, 一旦超过12个元素, 则哈希表需要扩容。

候选者: 怎么算出是12呢? 很简单, 就是 16×0.75 。每次put元素进去的时候, 都会检查HashMap的大小有没有超过这个阈值, 如果有, 则需要扩容。

HashMap动态扩容 (两倍)

候选者: 鉴于上面的说法 (HashMap的大小只能是2次幂), 所以扩容的时候时候默认是扩原来的2倍

候选者: 扩容这个操作肯定是耗时的, 那能不能把负载因子调高一点, 比如我要调至为1, 那我的HashMap就等到16个元素的时候才扩容呢。

候选者: 是可以的, 但是不推荐。负载因子调高了, 这意味着哈希冲突的概率会增高, 哈希冲突概率增高, 同样会耗时 (因为查找的速度变慢了)

面试官: 那我想问下, 在put元素的时候, 传递的Key是怎么算哈希值的?

候选者: 实现就在hash方法上, 可以发现的是, 它是先算出正常的哈希值, 然后与高16位做异或运算, 产生最终的哈希值。

候选者: 这样做的好处可以增加了随机性, 减少了碰撞冲突的可能性。

```
static final int hash(Object key) {  
    int h;  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}
```

计算Key的hash

高16位做异或运算（增加随机性）
减小哈希冲突概率

面试官：了解，你简单再说下put和get方法的实现吧

候选者：在put的时候，首先对key做hash运算，计算出该key所在的index。

候选者：如果没碰撞，直接放到数组中，如果碰撞了，需要判断目前数据结构是链表还是红黑树，根据不同的情况进行插入。

候选者：假设key是相同的，则替换到原来的值。最后判断哈希表是否满了(当前哈希表大小*负载因子)，如果满了，则扩容

候选者：在get的时候，还是对key做hash运算，计算出该key所在的index，然后判断是否有hash冲突

候选者：假设没有冲突直接返回，假设有冲突则判断当前数据结构是链表还是红黑树，分别从不同的数据结构中取出。

HashMap在put元素时

计算Hash

是否发生碰撞
(无碰撞直接进数组)
(有碰撞判断此时的数据结构)

key完全相同时，需要替换

是否需要扩容

面试官：那在HashMap中是怎么判断一个元素是否相同的呢？

候选者：首先会比较hash值，随后会用==运算符和equals()来判断该元素是否相同。

候选者：说白了就是：如果只有hash值相同，那说明该元素哈希冲突了，如果hash值和equals() || == 都相同，那说明该元素是同一个。

HashMap判断Key是否相同

对比Hash

对比equals

面试官：你说HashMap的数据结构是数组+链表/红黑树，那什么情况下才会用到红黑树呢？

候选者：当数组的大小大于64且链表的大小大于8的时候才会将链表改为红黑树，当红黑树大小为6时，会退化为链表。

候选者：这里转红黑树退化为链表的操作主要出于查询和插入时对性能的考量。

候选者：链表查询时间复杂度 $O(N)$ ，插入时间复杂度 $O(1)$ ，红黑树查询和插入时间复杂度 $O(\log N)$

链表转化红黑树

红黑树退化链表

数组大小 大于64

链表大小 大于8

红黑树大小 小于6

查询和插入性能综合考量

面试官：你在日常开发中LinkedHashMap用的多吗？

候选者：其实在日常开发中LinkedHashMap用得不多。

候选者：在前面也提到了，LinkedHashMap底层结构是数组+链表+双向链表，实际上它继承了HashMap，在HashMap的基础上维护了一个双向链表。

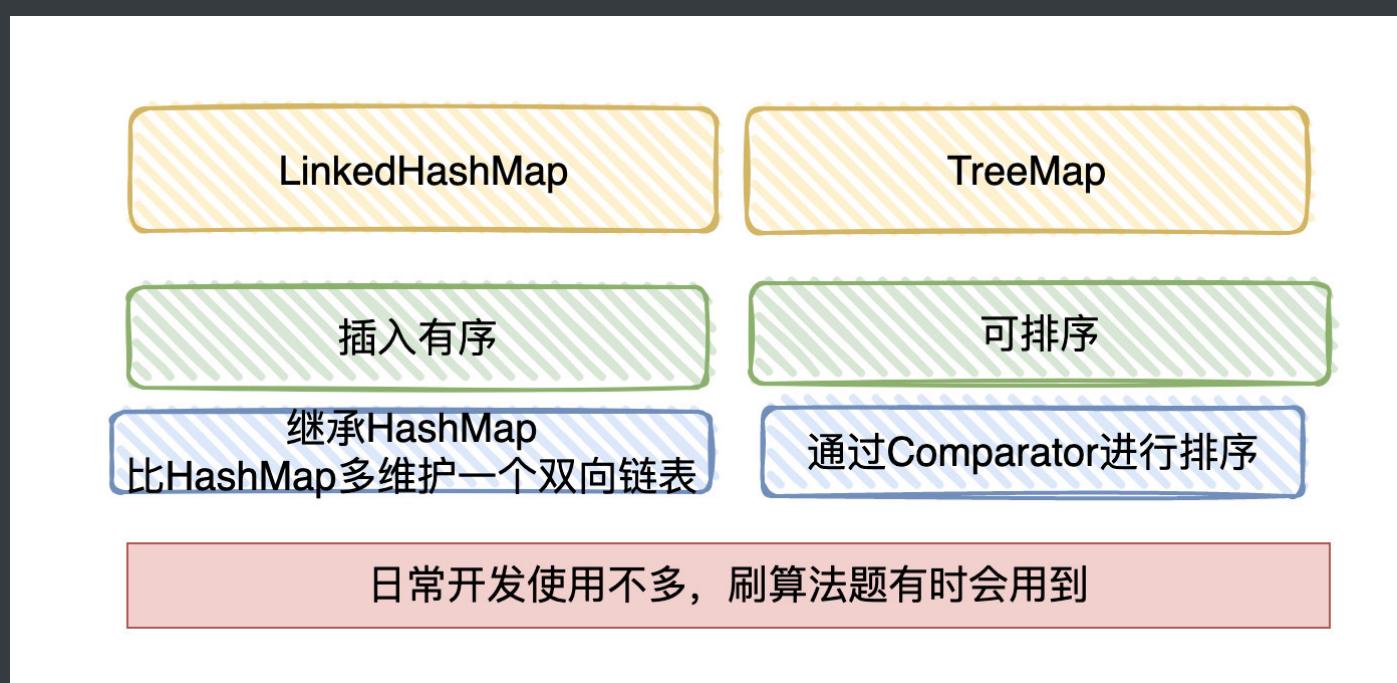
候选者：有了这个双向链表，我们的插入可以是有序的，这里的有序不是指大小有序，而是插入有序。

候选者：LinkedHashMap在遍历的时候实际用的是双向链表来遍历的，所以LinkedHashMap的大小不会影响到遍历的性能

面试官：那TreeMap呢？

候选者：TreeMap在现实开发中用得也不多，TreeMap的底层数据结构是红黑树

候选者：TreeMap的key不能为null（如果为null，那还怎么排序呢），TreeMap有序是通过Comparator来进行比较的，如果comparator为null，那么就使用自然顺序



面试官：再来讲讲线程安全的Map吧？HashMap是线程安全的吗？

候选者：HashMap不是线程安全的，在多线程环境下，HashMap有可能会有数据丢失和获取不了最新数据的问题，比如说：线程Aput进去了，线程Bget不出来。

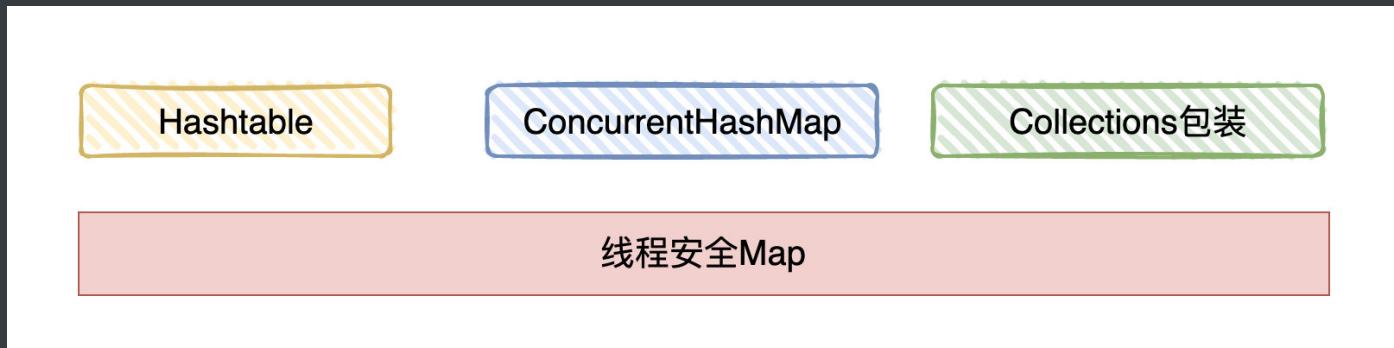
候选者：我们想要线程安全，可以使用ConcurrentHashMap

候选者：ConcurrentHashMap是线程安全的Map实现类，它在juc包下的。

候选者: 线程安全的Map实现类除了ConcurrentHashMap还有一个叫做Hashtable。

候选者: 当然了，也可以使用Collections来包装出一个线程安全的Map。

候选者: 但无论是Hashtable还是Collections包装出来的都比较低效（因为是直接在外层套synchronize），所以我们一般有线程安全问题考量的，都使用ConcurrentHashMap



候选者: ConcurrentHashMap的底层数据结构是数组+链表/红黑树，它能支持高并发的访问和更新，是线程安全的。

候选者: ConcurrentHashMap通过在部分加锁和利用CAS算法来实现同步，在get的时候没有加锁，Node都用了volatile给修饰。

候选者: 在扩容时，会给每个线程分配对应的区间，并且为了防止putVal导致数据不一致，会给线程的所负责的区间加锁

面试官: 你能给我讲讲JDK 7 和JDK8中HashMap和ConcurrentHashMap的区别吗？

候选者: 不能，我不会

候选者: 我在学习的时候也看过JDK7的HashMap和ConcurrentHashMap，其实还是有很多不一样的地方

候选者: 比如JDK 7 的HashMap在扩容时是头插法，在JDK8就变成了尾插法，在JDK7 的HashMap还没有引入红黑树....

候选者： ConcurrentHashMap 在JDK7 还是使用分段锁的方式来实现，而JDK 8 就又不一样了。但JDK 7细节我大多数都忘了。

候选者： 我就没用过JDK 7的API，我想着现在最低应该也是用JDK8了吧？所以我就没去仔细看了。

面试官： 你这很危险！



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



04-Java虚拟机

01、Java编译到执行的过程

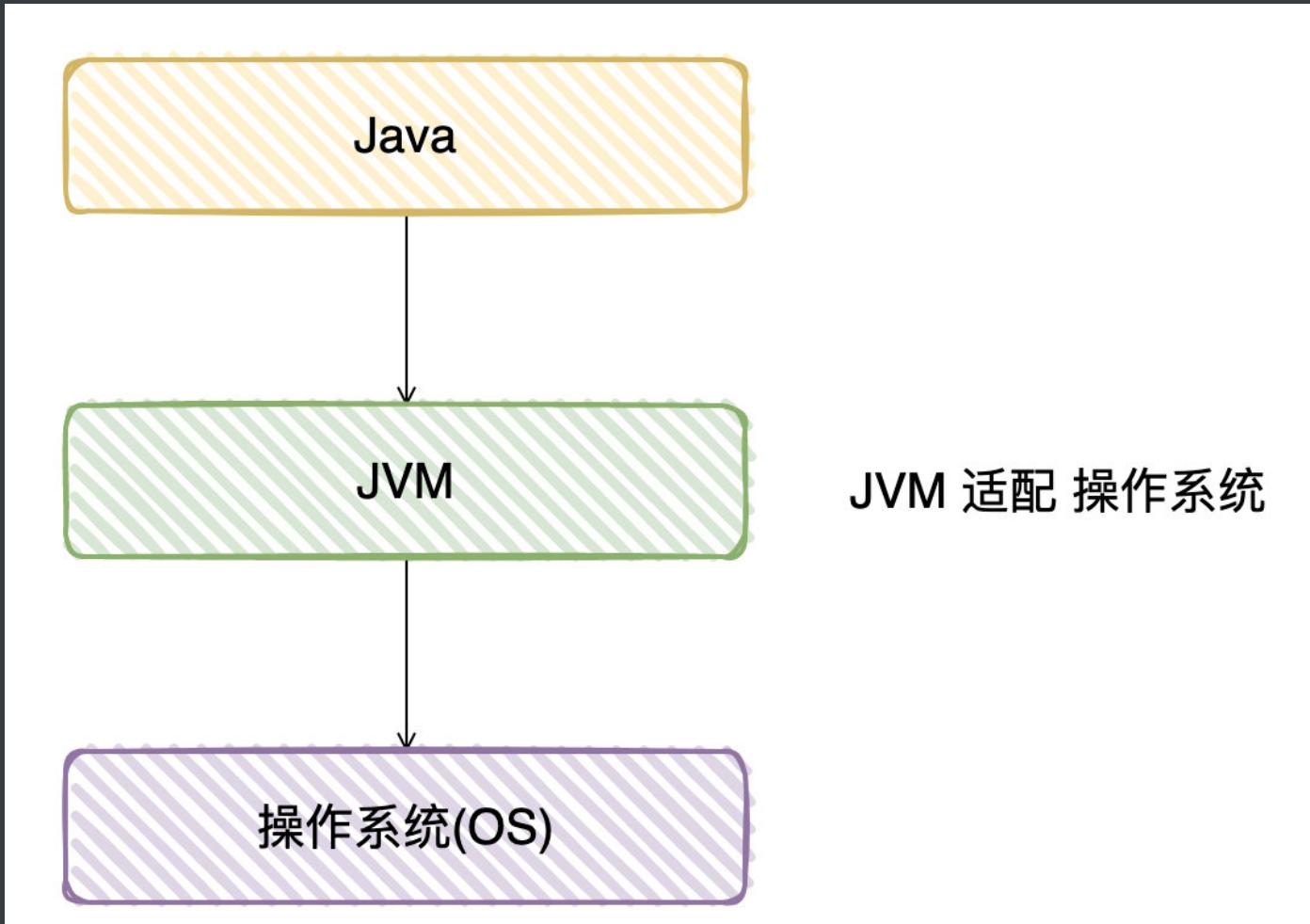
面试官：今天从基础先问起吧，你是怎么理解Java是一门「跨平台」的语言，也就是「一次编译，到处运行的」？

候选者：很好理解啊，因为我们有JVM。

候选者：Java源代码会被编译为class文件，class文件是运行在JVM之上的。

候选者：当我们日常开发安装JDK的时候，可以发现JDK是分「不同的操作系统」，JDK里是包含JVM的，所以Java依赖着JVM实现了『跨平台』

候选者：JVM是面向操作系统的，它负责把Class字节码解释成系统所能识别的指令并执行，同时也负责程序运行时内存的管理。



面试官：那要不你来聊聊从源码文件(.java)到代码执行的过程呗？

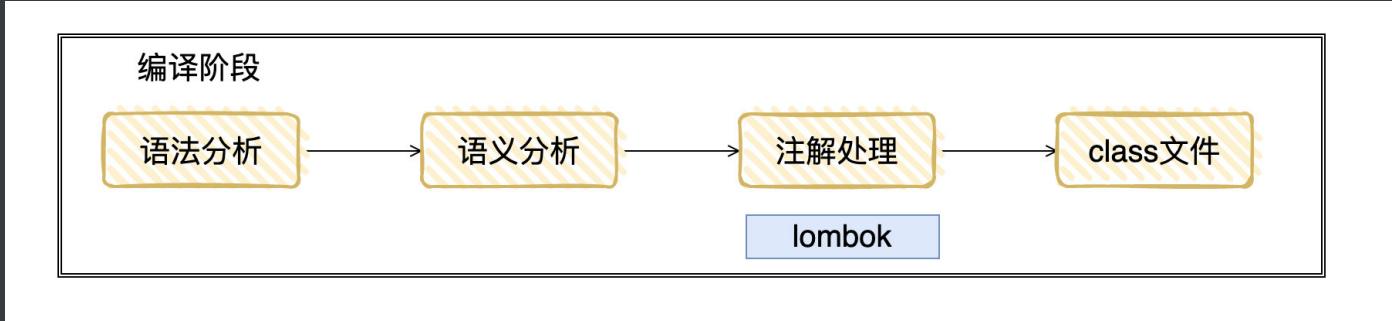
候选者：嗯，没问题的

候选者：简单总结的话，我认为就4个步骤：编译->加载->解释->执行

候选者：编译：将源码文件编译成JVM可以解释的class文件。

候选者：编译过程会对源代码程序做「语法分析」「语义分析」「注解处理」等等处理，最后才生成字节码文件。

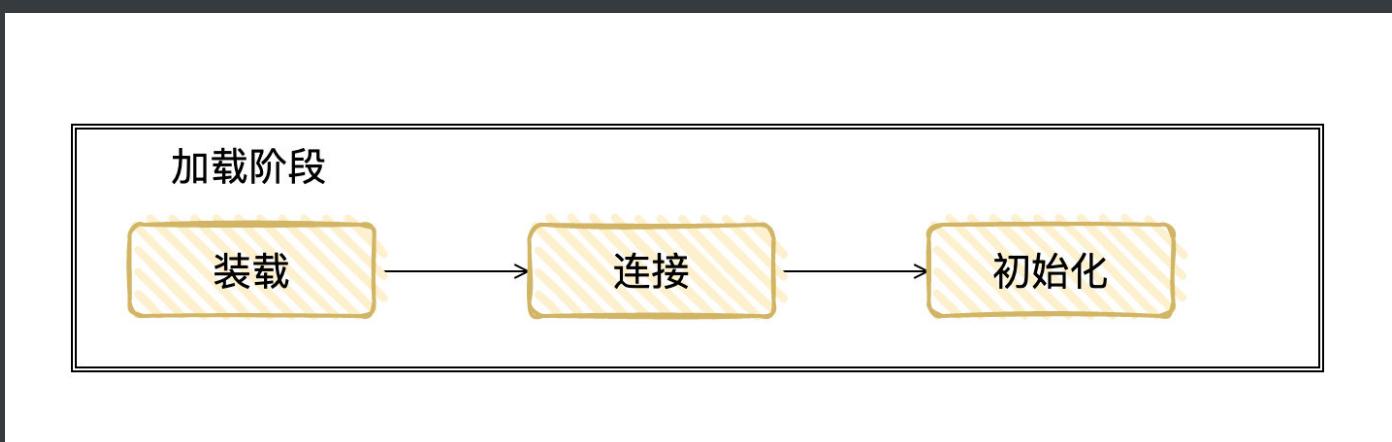
候选者：比如对泛型的擦除和我们经常用的Lombok就是在编译阶段干的。



候选者：加载：将编译后的class文件加载到JVM中。

候选者：在加载阶段又可以细化几个步骤：装载->连接->初始化

候选者：下面我对这些步骤又细说下哈。



候选者：【装载时机】为了节省内存的开销，并不会一次性把所有的类都装载至JVM，而是等到「有需要」的时候才进行装载（比如new和反射等等）

候选者：【装载发生】class文件是通过「类加载器」装载到jvm中的，为了防止内存中出现多份同样的字节码，使用了双亲委派机制（它不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上）

候选者：【装载规则】JDK 中的本地方法类一般由根加载器（Bootstrap loader）装载，JDK 中内部实现的扩展类一般由扩展加载器（ExtClassLoader）实现装载，而程序中的类文件则由系统加载器（AppClassLoader）实现装载。

候选者：装载这个阶段它做的事情可以总结为：查找并加载类的二进制数据，在JVM「堆」中创建一个java.lang.Class类的对象，并将类相关的信息存储在JVM「方法区」中

面试官：嗯...

候选者：通过「装载」这个步骤后，现在已经把class文件装载到JVM中了，并创建出对应的Class对象以及类信息存储至方法区了。

候选者：「连接」这个阶段它做的事情可以总结为：对class的信息进行验证、为「类变量」分配内存空间并对其赋默认值。

候选者：连接又可以细化为几个步骤：验证->准备->解析

候选者：1. 验证：验证类是否符合 Java 规范和 JVM 规范

候选者：2. 准备：为类的静态变量分配内存，初始化为系统的初始值

候选者：3. 解析：将符号引用转为直接引用的过程

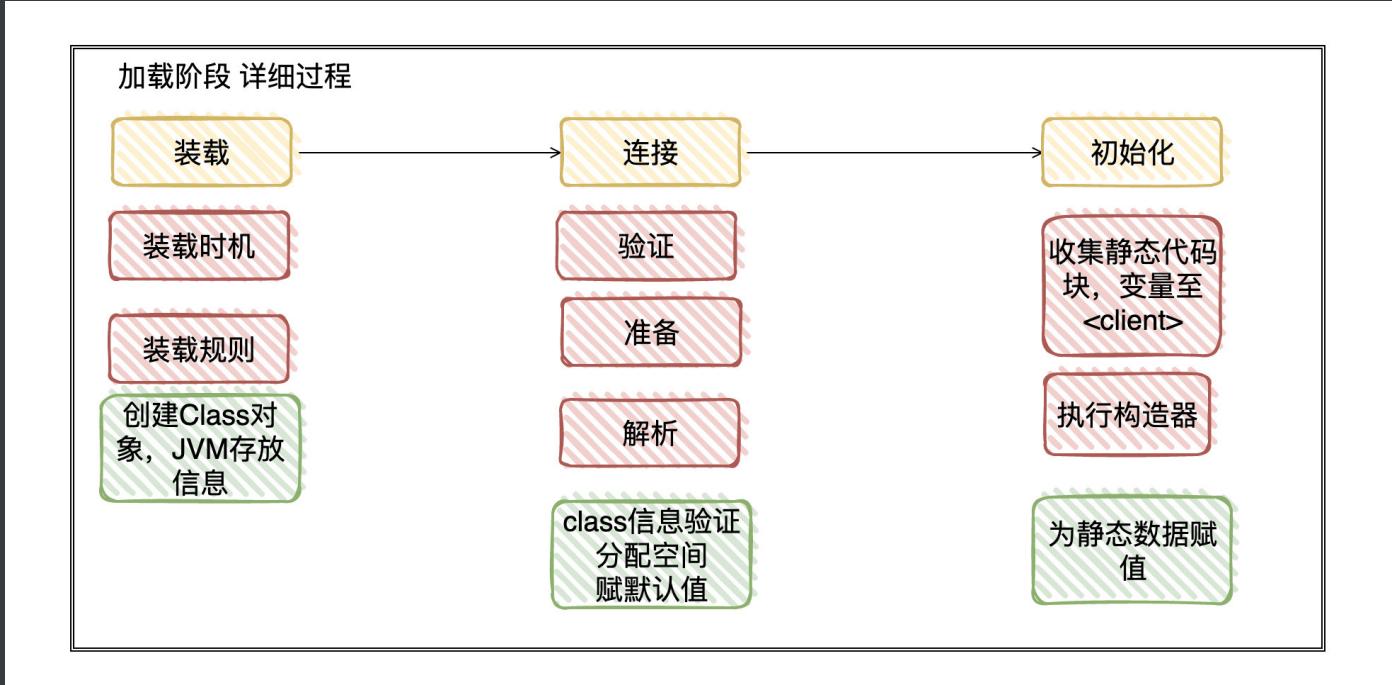
面试官：嗯...

候选者：通过「连接」这个步骤后，现在已经对class信息做校验并分配了内存空间和默认值了。

候选者：接下来就是「初始化」阶段了，这个阶段可以总结为：为类的静态变量赋予正确的初始值。

候选者：过程大概就是收集class的静态变量、静态代码块、静态方法至()方法，随后从上往下开始执行。

候选者：如果「实例化对象」则会调用方法对实例变量进行初始化，并执行对应的构造方法内的代码。



候选者：扯了这么多，现在其实才完成至(编译->加载->解释->执行)中的加载阶段，下面就来说下【解释阶段】做了什么

候选者： 初始化完成之后，当我们尝试执行一个类的方法时，会找到对应方法的字节码的信息，然后解释器会把字节码信息解释成系统能识别的指令码。

候选者：「解释」这个阶段它做的事情可以总结为：把字节码转换为操作系统识别的指令

候选者：在解释阶段会有两种方式把字节码信息解释成机器指令码，一个是字节码解释器、一个是即时编译器(JIT)。



候选者：JVM会对「热点代码」做编译，非热点代码直接进行解释。当JVM发现某个方法或代码块的运行特别频繁的时候，就有可能把这部分代码认定为「热点代码」

候选者：使用「热点探测」来检测是否为热点代码。「热点探测」一般有两种方式，计数器和抽样。HotSpot使用的是「计数器」的方式进行探测，为每个方法准备了两类计数器：方法调用计数器和回边计数器

候选者：这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发JIT编译。

候选者：即时编译器把热点方法的指令码保存起来，下次执行的时候就无需重复的进行解释，直接执行缓存的机器语言

面试官：嗯...

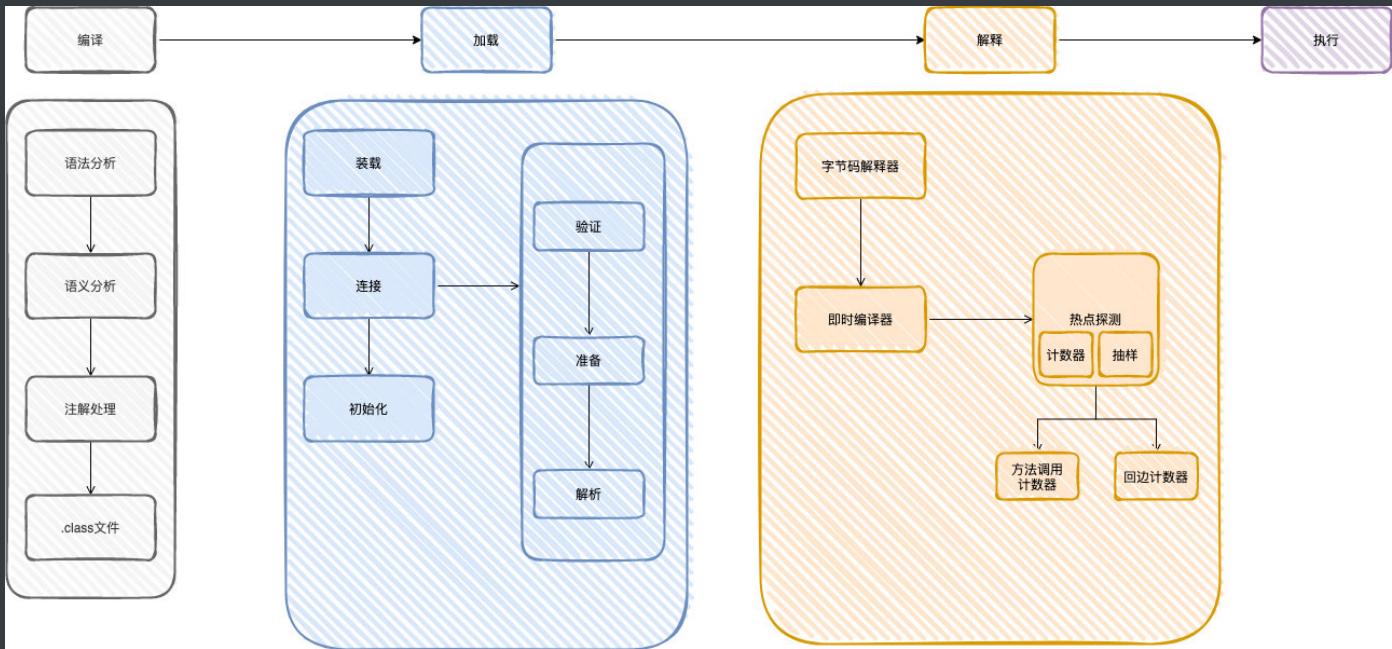
候选者：解释阶段结束后，最后就到了执行阶段。

候选者：「执行」这个阶段它做的事情可以总结为：操作系统把解释器解析出来的指令码，调用系统的硬件执行最终的程序指令。

候选者：上面就是我对从源码文件(.java)到代码执行的过程的理解了。

面试官：嗯...我还想问下你刚才提到的双亲委派模型...

候选者：下次一定！



本文总结：

- Java跨平台因为有JVM屏蔽了底层操作系统
- Java源码到执行的过程，从JVM的角度看可以总结为四个步骤：编译->加载->解释->执行
 - 「编译」 经过 语法分析、语义分析、注解处理 最后才生成会class文件
 - 「加载」 又可以细分步骤为： 装载->连接->初始化。 装载则把class文件装载至JVM， 连接则校验class信息、分配内存空间及赋默认值， 初始化则为变量赋值为正确的初始值。 连接里又可以细化为： 验证、准备、解析
 - 「解释」 则是把字节码转换成操作系统可识别的执行指令，在JVM中会有字节码解释器和即时编译器。 在解释时会对代码进行分析，查看是否为「热点代码」， 如果为「热点代码」则触发JIT编译， 下次执行时就无需重复进行解释， 提高解释速度
 - 「执行」 调用系统的硬件执行最终的程序指令



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

02、双亲委派机制

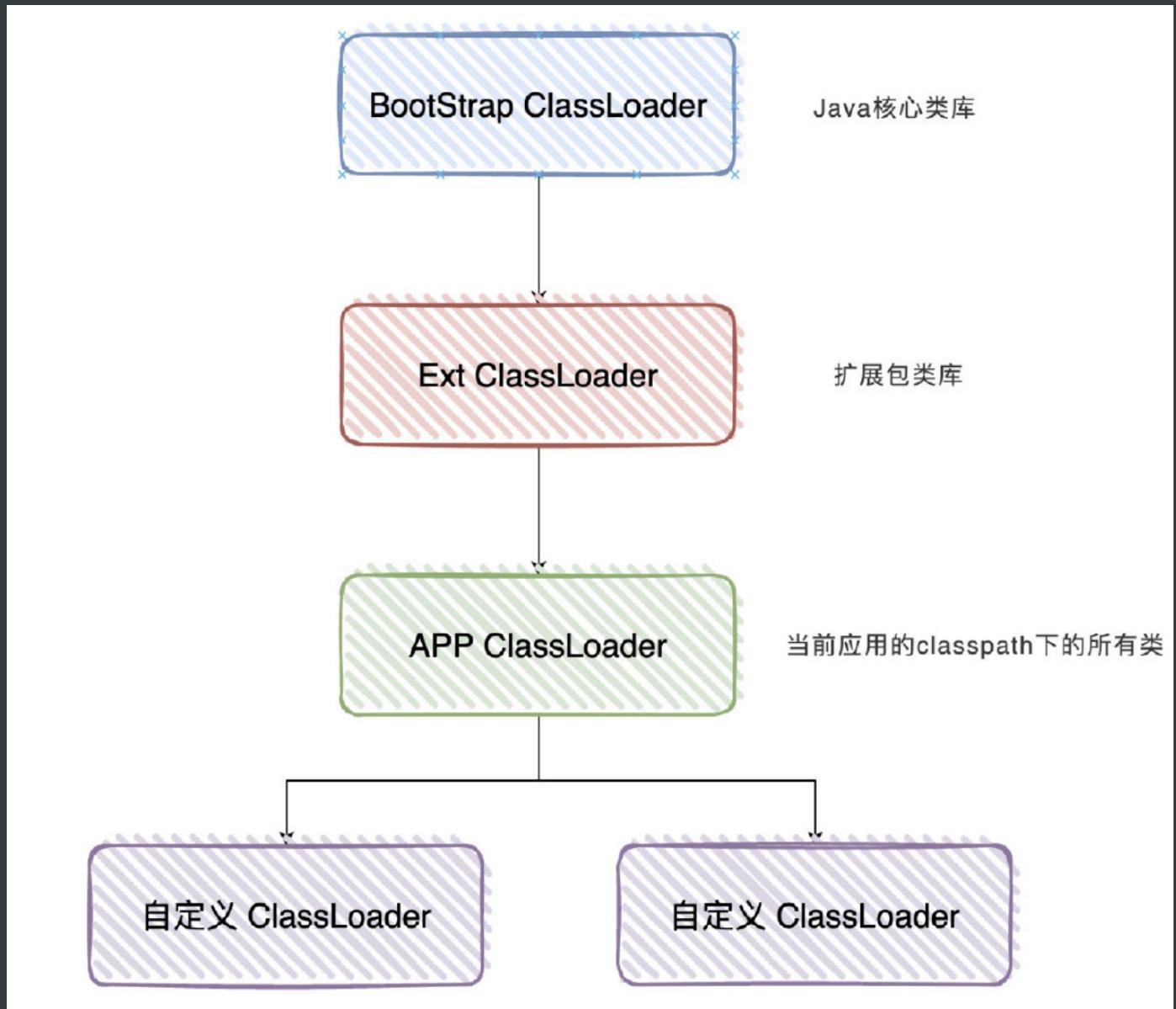
面试官：要不你今天来详细讲讲双亲委派机制？

候选者：嗯，好的。

候选者：上次提到了：class文件是通过「类加载器」装载至JVM中的

候选者：为了防止内存中存在多份同样的字节码，使用了双亲委派机制（它不会自己去尝试加载类，而是把请求委托给父加载器去完成，依次向上）

候选者: JDK 中的本地方法类一般由根加载器 (Bootstrap loader) 装载, JDK 中内部实现的扩展类一般由扩展加载器 (ExtClassLoader) 实现装载, 而程序中的类文件则由系统加载器 (AppClassLoader) 实现装载。



候选者: 这应该很好理解吧?

面试官: 雀食(确实)!

面试官: 顺着话题, 我想问问, 打破双亲委派机制是什么意思?

候选者: 很好理解啊, 意思就是: 只要我加载类的时候, 不是从APPClassLoader->ExtClassLoader->BootStrap ClassLoader 这个顺序找, 那就算是打破了啊

候选者: 因为加载class核心的方法在LoaderClass类的loadClass方法上（双亲委派机制的核心实现）

候选者: 那只要我自定义个ClassLoader，重写loadClass方法（不依照往上开始寻找类加载器），那就算是打破双亲委派机制了。

面试官: 这么简单？

候选者: 嗯，就是这么简单

面试官: 那你知道有哪个场景破坏了双亲委派机制吗？

候选者: 最明显的就Tomcat啊

面试官: 详细说说？

候选者: 在初学时部署项目，我们是把war包放到tomcat的webapp下，这意味着一个tomcat可以运行多个Web应用程序（：

候选者: 是吧？

面试官: 嗯..

候选者: 那假设我现在有两个Web应用程序，它们都有一个类，叫做User，并且它们的类全限定名都一样，比如都是com.yyy.User。但是他们的具体实现是不一样的

候选者: 那么Tomcat是如何保证它们是不会冲突的呢？

候选者: 答案就是，Tomcat给每个Web应用创建一个类加载器实例（WebAppClassLoader），该加载器重写了loadClass方法，优先加载当前应用目录下的类，如果当前找不到了，才一层一层往上找（：

候选者: 那这样就做到了Web应用层级的隔离

WebAppClassLoader

打破双亲委派机制

Web应用之间 依赖隔离

面试官：嗯， 那你还知道Tomcat还有别的类加载器吗？

候选者： 嗯， 知道的

候选者： 并不是Web应用程序下的所有依赖都需要隔离的， 比如Redis就可以Web应用程序之间共享（如果有需要的话）， 因为如果版本相同， 没必要每个Web应用程序都独自加载一份啊。

候选者： 做法也很简单， Tomcat就在WebAppClassLoader上加了个父类加载器（SharedClassLoader）， 如果WebAppClassLoader自身没有加载到某个类， 那就委托SharedClassLoader去加载。

候选者： （无非就是把需要应用程序之间需要共享的类放到一个共享目录下嘛）

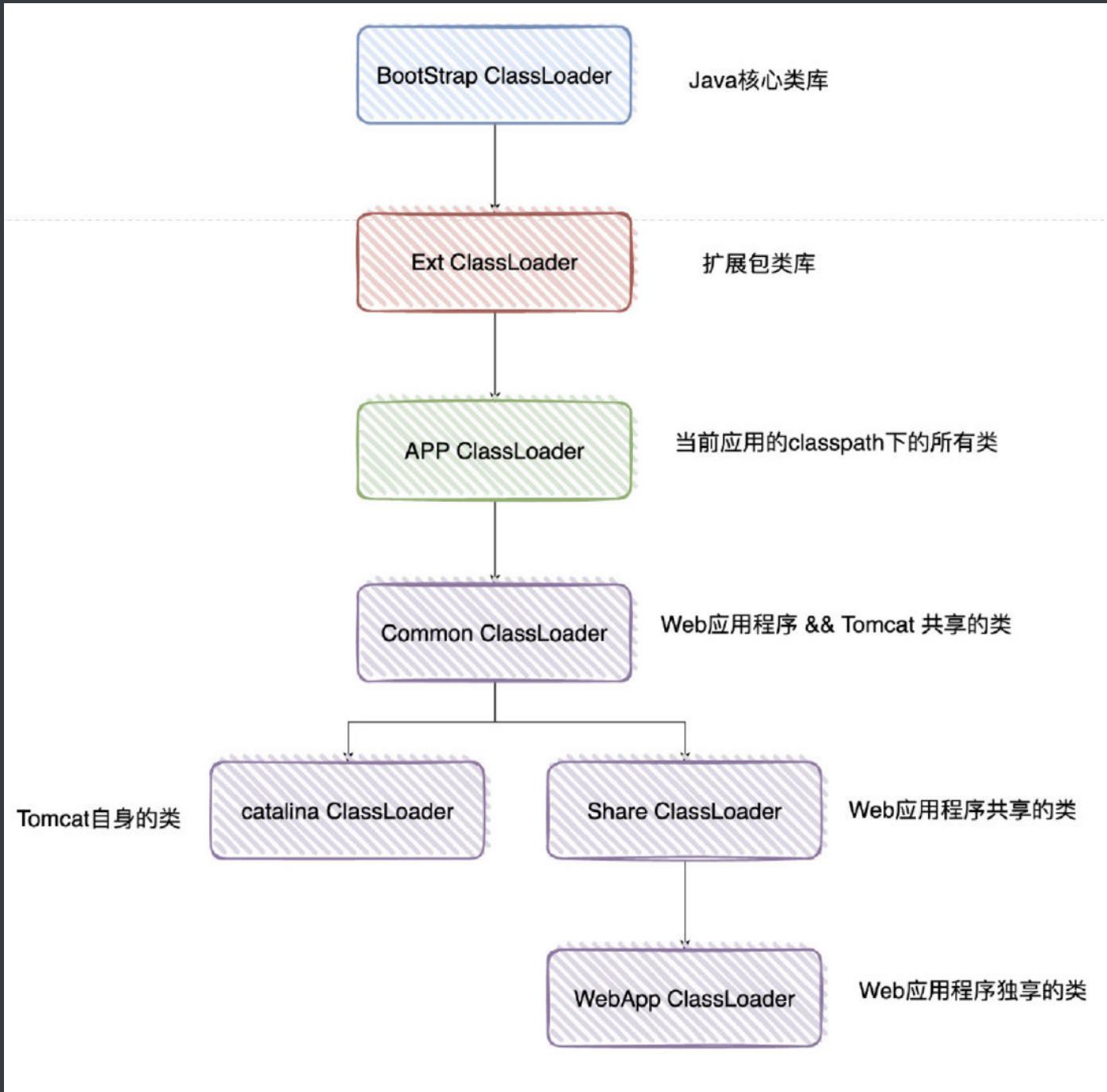
面试官： �恩..

候选者: 为了隔绝Web应用程序与Tomcat本身的类，又有类加载器(CatalinaClassLoader)来装载Tomcat本身的依赖

候选者: 如果Tomcat本身的依赖和Web应用还需要共享，那么还有类加载器(CommonClassLoader)来装载进而达到共享

候选者: 各个类加载器的加载目录可以到tomcat的catalina.properties配置文件上查看

候选者: 我稍微画下Tomcat的类加载结构图吧，不然有点抽象



面试官：嗯，还可以，我听懂了，有点意思。

面试官：顺便，我想问下，JDBC你不是知道吗，听说它也是破坏了双亲委派模型的，你怎么理解的。

候选者：Eumm，这个有没有破坏，见仁见智吧。

候选者：JDBC定义了接口，具体实现类由各个厂商进行实现嘛(比如MySQL)

候选者: 类加载有个规则：如果一个类由类加载器A加载，那么这个类的依赖类也是由「相同的类加载器」加载。

候选者: 我们用JDBC的时候，是使用DriverManager进而获取Connection，DriverManager在java.sql包下，显然是由BootStrap类加载器进行装载

候选者: 当我们使用DriverManager.getConnection()时，得到的一定是厂商实现的类。

候选者: 但BootStrap ClassLoader会能加载到各个厂商实现的类吗？

候选者: 显然不可以啊，这些实现类又没在java包中，怎么可能加载得到呢

面试官: 嗯..

候选者: DriverManager的解决方案就是，在DriverManager初始化的时候，得到「线程上下文加载器」

候选者: 去获取Connection的时候，是使用「线程上下文加载器」去加载Connection的，而这里的线程上下文加载器实际上还是App ClassLoader

候选者: 所以在获取Connection的时候，还是先找Ext ClassLoader和BootStrap ClassLoader，只不过这俩加载器肯定是加载不到的，最终会由App ClassLoader进行加载

线程上下文加载器

直接指定对应的类加载器

面试官: 嗯..

候选者: 那这种情况，有的人觉得破坏了双亲委派机制，因为本来明明应该是由BootStrap ClassLoader进行加载的，结果你来了一手「线程上下文加载器」，改掉了「类加载器」

候选者: 有的人觉得没破坏双亲委派机制，只是改成由「线程上下文加载器」进行类加载，但还是遵守着：「依次往上找父类加载器进行加载，都找不到时才由自身加载」。认为"原则"上是没变的。

面试官: 那我了解了

本文总结:

- **前置知识**: JDK中默认类加载器有三个：AppClassLoader、Ext ClassLoader、BootStrap ClassLoader。AppClassLoader的父加载器为Ext ClassLoader、Ext ClassLoader的父加载器为BootStrap ClassLoader。这里的父子关系并不是通过继承实现的，而是组合。
- **什么是双亲委派机制**: 加载器在加载过程中，先把类交由父类加载器进行加载，父类加载器没找到才由自身加载。
- **双亲委派机制目的**: 为了防止内存中存在多份同样的字节码（安全）
- **类加载规则**: 如果一个类由类加载器A加载，那么这个类的依赖类也是由「相同的类加载器」加载。
- **如何打破双亲委派机制**: 自定义ClassLoader，重写loadClass方法（只要不依次往上交给父加载器进行加载，就算是打破双亲委派机制）
- **打破双亲委派机制案例**: Tomcat
 - 为了Web应用程序类之间隔离，为每个应用程序创建WebAppClassLoader类加载器
 - 为了Web应用程序类之间共享，把ShareClassLoader作为WebAppClassLoader的父类加载器，如果WebAppClassLoader加载器找不到，则尝试用ShareClassLoader进行加载
 - 为了Tomcat本身与Web应用程序类隔离，用CatalinaClassLoader类加载器进行隔离，CatalinaClassLoader加载Tomcat本身的类
 - 为了Tomcat与Web应用程序类共享，用CommonClassLoader作为CatalinaClassLoader和ShareClassLoader的父类加载器
 - ShareClassLoader、CatalinaClassLoader、CommonClassLoader的目录可以在

Tomcat的catalina.properties进行配置

- **线程上下文加载器**: 由于类加载的规则, 很可能导致父加载器加载时依赖子加载器的类, 导致无法加载成功 (BootStrap ClassLoader无法加载第三方库的类), 所以存在「线程上下文加载器」来进行加载。



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



03、Java内存结构

面试官：今天来聊聊JVM的内存结构吧？

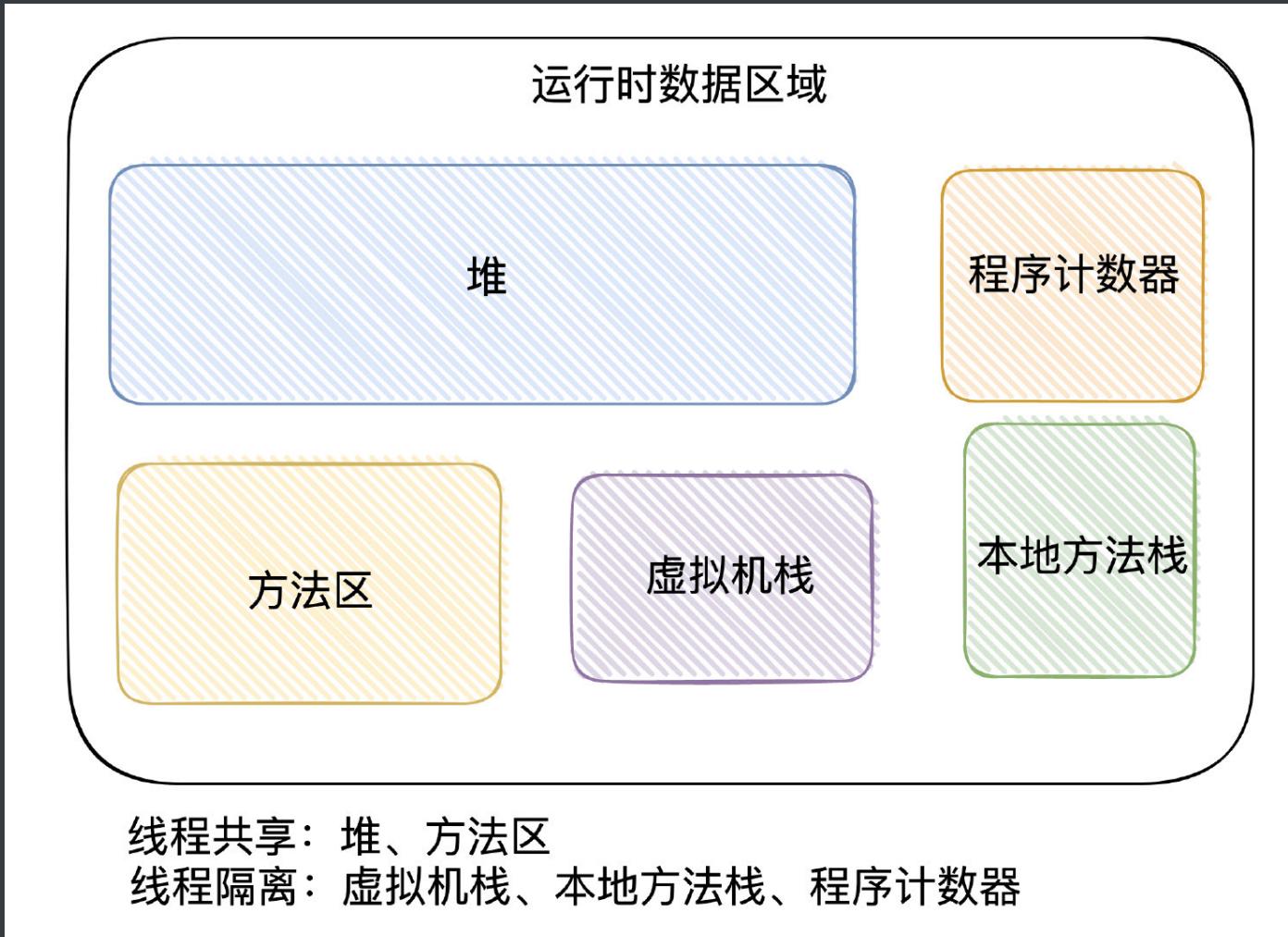
候选者：嗯，好的

候选者：前几次面试的时候也提到了：class文件会被类加载器装载至JVM中，并且JVM会负责程序「运行时」的「内存管理」

候选者：而JVM的内存结构，往往指的就是JVM定义的「运行时数据区域」

候选者：简单来说就分为了5大块：方法区、堆、程序计数器、虚拟机栈、本地方法栈

候选者：要值得注意的是：这是JVM「规范」的分区概念，到具体的实现落地，不同的厂商实现可能是有所区别的。



面试官：嗯，顺便讲下你这图上每个区域的内容吧。

候选者：好的，那我就先从「程序计数器」开始讲起吧。

候选者：Java是多线程的语言，我们知道假设线程数大于CPU数，就很有可能有「线程切换」现象，切换意味着「中断」和「恢复」，那自然就需要有一块区域来保存「当前线程的执行信息」

候选者：所以，程序计数器就是用于记录各个线程执行的字节码的地址（分支、循环、跳转、异常、线程恢复等都依赖于计数器）

面试官：好的，理解了。

候选者：那接下来我就说下「虚拟机栈」吧

候选者：每个线程在创建的时候都会创建一个「虚拟机栈」，每次方法调用都会创建一个「栈帧」。每个「栈帧」会包含几块内容：局部变量表、操作数栈、动态连接和返回地址

线程

虚拟机栈

栈帧

操作数栈 局部变量表

方法返回值 动态链接

栈帧

操作数栈 局部变量表

方法返回值 动态链接

栈帧

操作数栈 局部变量表

方法返回值 动态链接

每次方法执行都会生成一个栈帧

候选者：了解了「虚拟机栈」的组成后，也不难猜出它的作用了：它保存方法了局部变量、部分变量的计算并参与了方法的调用和返回。

面试官：ok，了解了

候选者：下面就说下「本地方法栈」吧

候选者：本地方法栈跟虚拟机栈的功能类似，虚拟机栈用于管理 Java 函数的调用，而本地方方法栈则用于管理本地方法的调用。这里的「本地方法」指的是「非Java方法」，一般本地方方法是使用C语言实现的。

面试官：嗯...

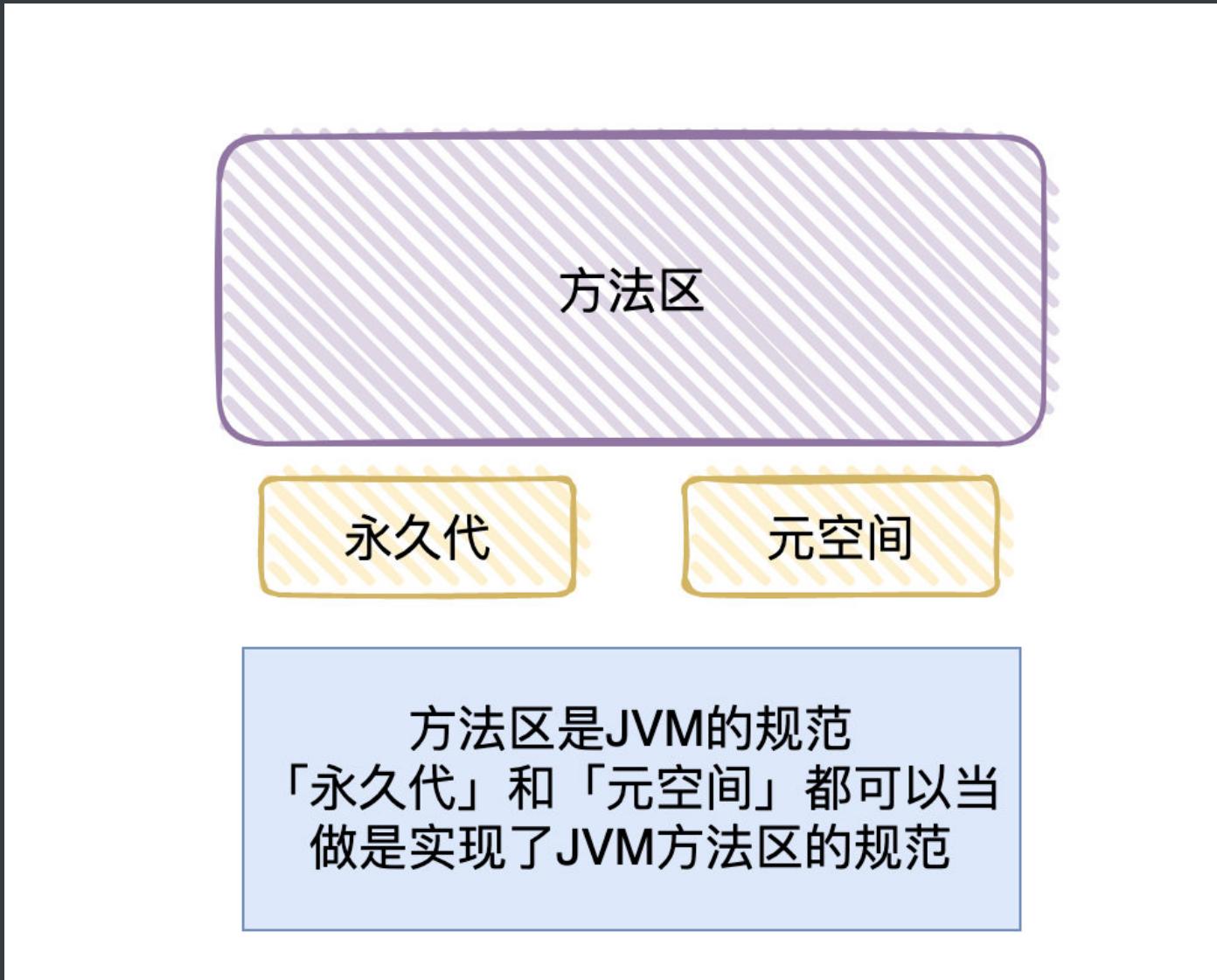
候选者：嗯，说完了「本地方法栈」、「虚拟机栈」和「程序计数器」，哦，下面还有「方法区」和「堆」

候选者：那我先说「方法区」吧

候选者：前面提到了运行时数据区这个「分区」是JVM的「规范」，具体的落地实现，不同的虚拟机厂商可能是不一样的

候选者：所以「方法区」也只是 JVM 中规范的一部分而已。

候选者：在HotSpot虚拟机，就会常常提到「永久代」这个词。HotSpot虚拟机在「JDK8 前」用「永久代」实现了「方法区」，而很多其他厂商的虚拟机其实是没有「永久代」的概念的。



候选者：我们下面的内容就都用HotSpot虚拟机来说明好了。

候选者：在JDK8中，已经用「元空间」来替代了「永久代」作为「方法区」的实现了

面试官：嗯...

候选者：方法区主要是用来存放已被虚拟机加载的「类相关信息」：包括类信息、常量池

候选者：类信息又包括了类的版本、字段、方法、接口和父类等信息。

候选者：常量池又可以分「静态常量池」和「运行时常量池」

候选者：静态常量池主要存储的是「字面量」以及「符号引用」等信息，静态常量池也包括了我们说的「字符串常量池」。

候选者：「运行时常量池」存储的是「类加载」时生成的「直接引用」等信息。

方法区主要是用来存放已被虚拟机加载的「类相关信息」

面试官：嗯...

候选者：又值得注意的是：从「逻辑分区」的角度而言「常量池」是属于「方法区」的

候选者：但自从在「JDK7」以后，就已经把「运行时常量池」和「静态常量池」转移到了「堆」内存中进行存储（对于「物理分区」来说「运行时常量池」和「静态常量池」就属于堆）

面试官：嗯，这信息量有点多

面试官：我想问下，你说从「JDK8」已经把「方法区」的实现从「永久代」变成「元空间」，有什么区别？

候选者：最主要的区别就是：「元空间」存储不在虚拟机中，而是使用本地内存，JVM 不会再出现方法区的内存溢出，以往「永久代」经常因为内存不够用导致跑出OOM异常。

候选者：按JDK8版本，总结起来其实就相当于：「类信息」是存储在「元空间」的（也有人把「类信息」这块叫做「类信息常量池」，主要是叫法不同，意思到位就好）

候选者：而「常量池」用JDK7开始，从「物理存储」角度上就在「堆中」，这是没有变化的。

方法区

类信息

元空间（本地内存）

常量池

JVM 堆

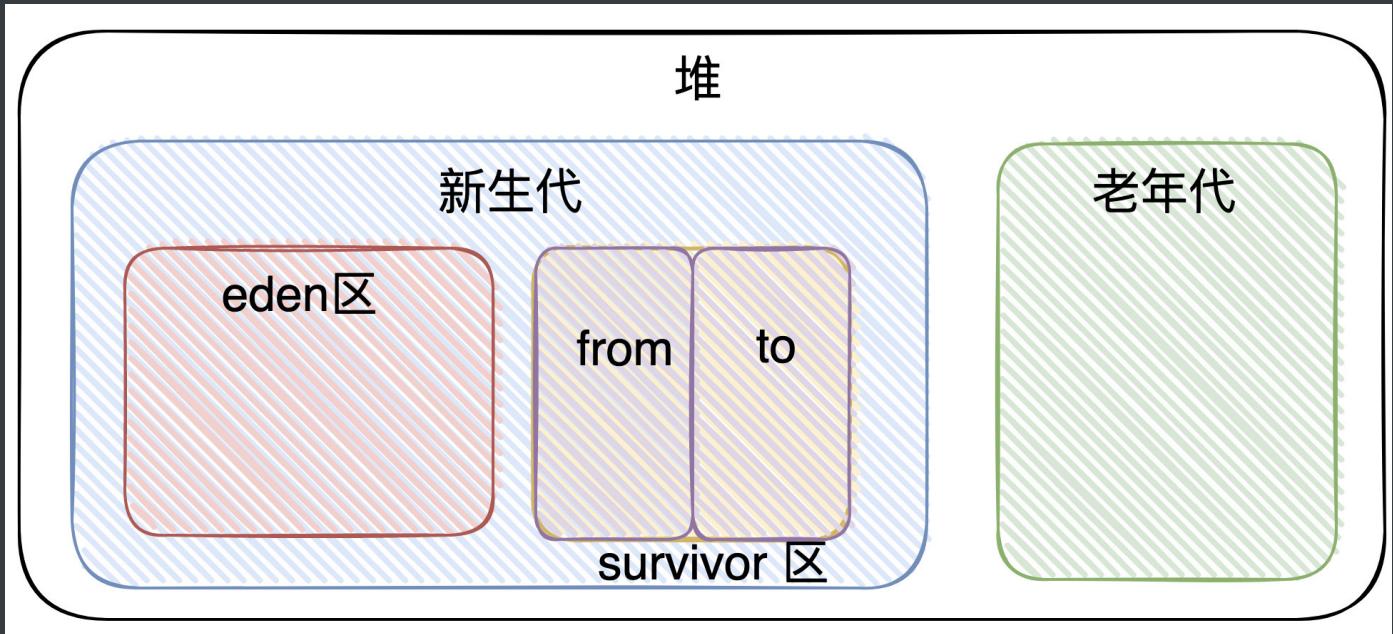
面试官：嗯，我听懂了

面试官：最后来讲讲「堆」这块区域吧

候选者：嗯，「堆」是线程共享的区域，几乎类的实例和数组分配的内存都来自于它

候选者：「堆」被划分为「新生代」和「老年代」，「新生代」又被进一步划分为 Eden 和 Survivor 区，最后 Survivor 由 From Survivor 和 To Survivor 组成

候选者：不多BB，我也画图吧



候选者：将「堆内存」分开了几块区域，主要跟「内存回收」有关（垃圾回收机制）

面试官：那垃圾回收这块等下次吧，这个延伸下去又很多东西了

面试官：你要不先讲讲JVM内存结构和Java内存模型有啥区别吧？

候选者：他们俩没有啥直接关联，其实两次面试过后，应该你就有感觉了

候选者：Java内存模型是跟「并发」相关的，它是为了屏蔽底层细节而提出的规范，希望在上层(Java层面上)在操作内存时在不同的平台上也有相同的效果

候选者：Java内存结构（又称为运行时数据区域），它描述着当我们的class文件加载至虚拟机后，各个分区的「逻辑结构」是如何的，每个分区承担着什么作用。

面试官：了解了

今日总结：JVM内存结构组成（JVM内存结构又称为「运行时数据区域」）。主要有五部分组成：虚拟机栈、本地方法栈、程序计数器、方法区和堆。其中方法区和堆是线程共享的。虚拟机栈、本地方法栈以及程序计数器是线程隔离的）



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

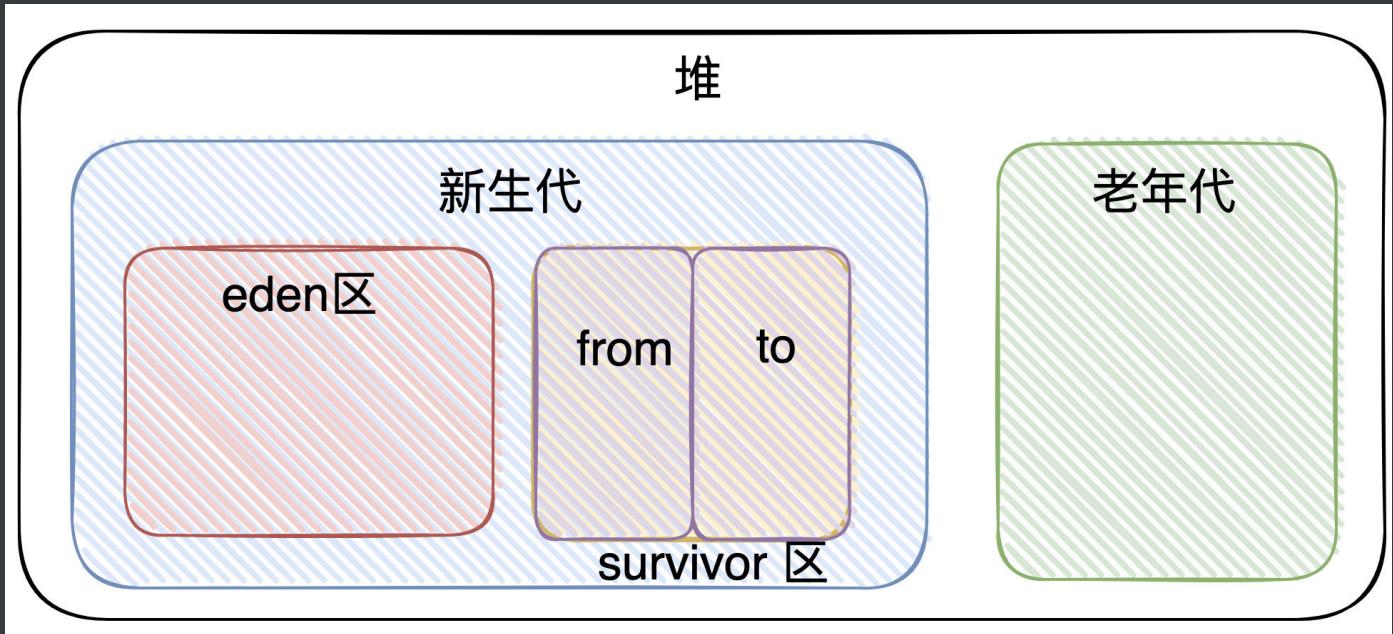
04、垃圾回收机制

面试官：我还记得上次你讲到JVM内存结构（运行时数据区域）提到了「堆」，然后你说是分了几块区域嘛

面试官：当时感觉再讲下去那我可能就得加班了

面试官：今天有点空了，继续聊聊「堆」那块吧

候选者：嗯，前面提到了堆分了「新生代」和「老年代」，「新生代」又分为「Eden」和「Survivor」区，「survivor」区又分为「From Survivor」和「To Survivor」区



候选者：说到这里，我就想聊聊Java的垃圾回收机制了

面试官：那你开始你的表演吧

候选者：我们使用Java的时候，会创建很多对象，但我们未曾「手动」将这些对象进行清除

候选者：而如果用C/C++语言的时候，用完是需要自己free(释放)掉的

候选者：那为什么在写Java的时候不用我们自己手动释放"垃圾"呢？原因很简单，JVM帮我们做了（自动回收垃圾）

面试官：嗯...

候选者：我个人对垃圾的定义：只要对象不再被使用了，那我们就认为该对象就是垃圾，对象所占用的空间就可以被回收

对象不再使用，就是垃圾

面试官：那是怎么判断对象不再被使用的呢？

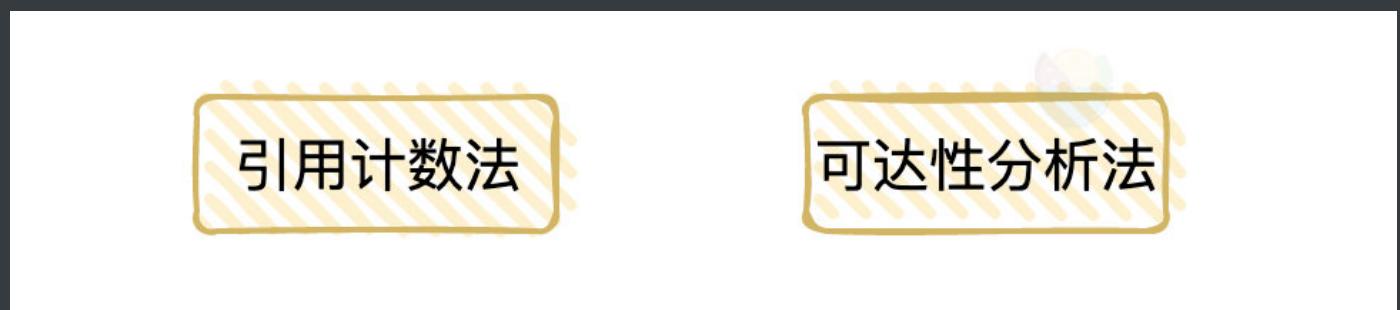
候选者: 常用的算法有两个「引用计数法」和「可达性分析法」

候选者: 引用计数法思路很简单：当对象被引用则+1，但对象引用失败则-1。当计数器为0时，说明对象不再被引用，可以被回收

候选者: 引用计数法最明显的缺点就是：如果对象存在循环依赖，那就无法定位该对象是否应该被回收（A依赖B，B依赖A）

面试官: 嗯...

候选者: 另一种就是可达性分析法：它从「GC Roots」开始向下搜索，当对象到「GC Roots」都没有任何引用相连时，说明对象是不可用的，可以被回收



候选者: 「GC Roots」是一组必须「活跃」的引用。从「GC Root」出发，程序通过直接引用或者间接引用，能够找到可能正在被使用的对象

面试官: 还是不太懂，那「GC Roots」一般是什么？你说它是一组活跃的引用，能不能举个例子，太抽象了。

候选者: 比如我们上次不是聊到JVM内存结构中的虚拟机栈吗，虚拟机栈里不是有栈帧吗，栈帧不是有局部变量吗？局部变量不就存储着引用嘛。

候选者: 那如果栈帧位于虚拟机栈的栈顶，是不是就可以说明这个栈帧是活跃的（换言之，是线程正在被调用的）

候选者: 既然是线程正在调用的，那栈帧里的指向「堆」的对象引用，是不是一定是「活跃」的引用？

候选者: 所以，当前活跃的栈帧指向堆里的对象引用就可以是「GC Roots」

面试官：嗯...

候选者：当然了，能作为「GC Roots」也不单单只有上面那一小块

候选者：比如类的静态变量引用是「GC Roots」，被「Java本地方法」所引用的对象也是「GC Roots」等等...

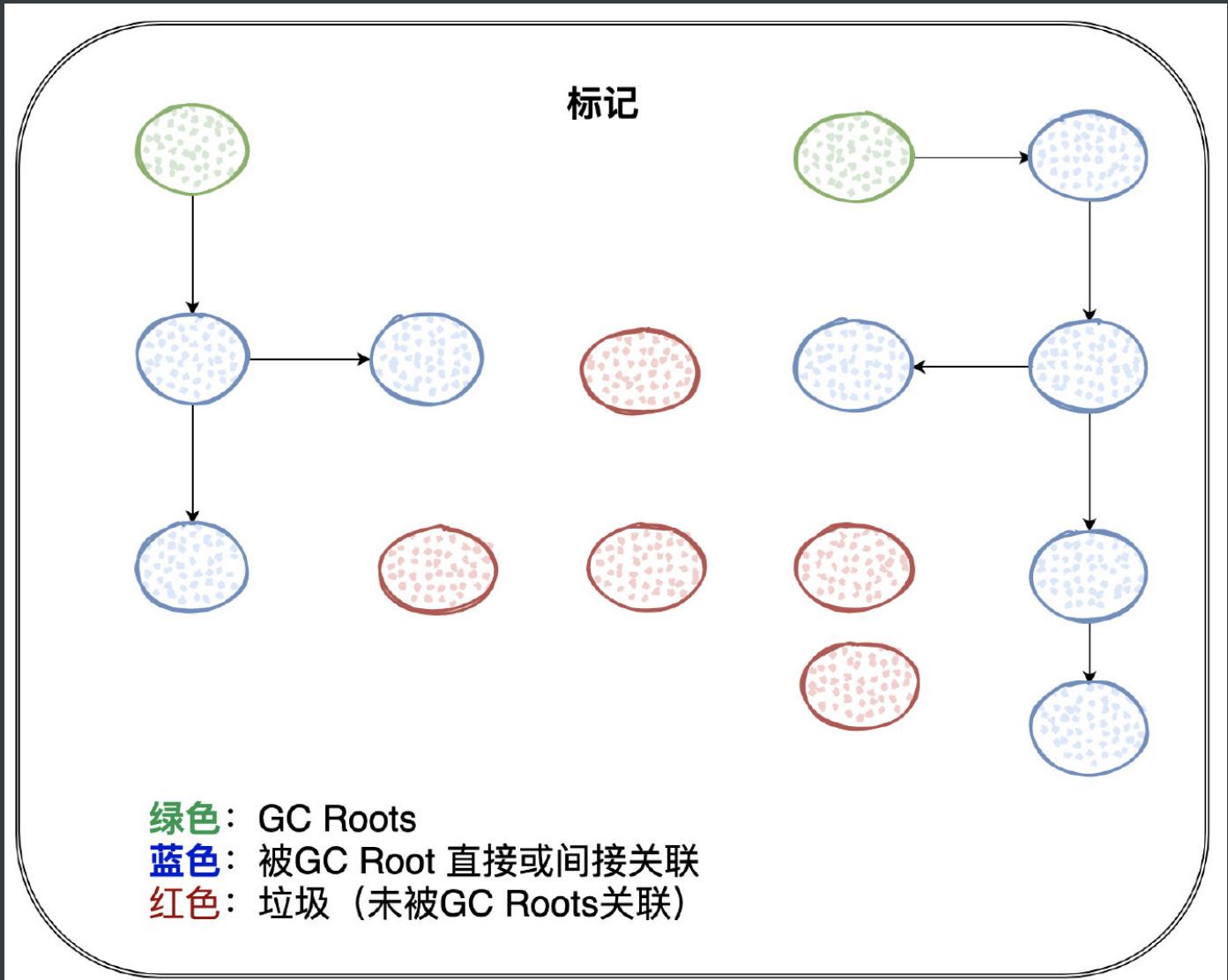
GC Roots是一组一定活跃的引用

候选者：回到理解的重点：「GC Roots」是一组必须「活跃」的「引用」，只要跟「GC Roots」没有直接或者间接引用相连，那就是垃圾

候选者：JVM用的就是「可达性分析算法」来判断对象是否垃圾

面试官：懂了

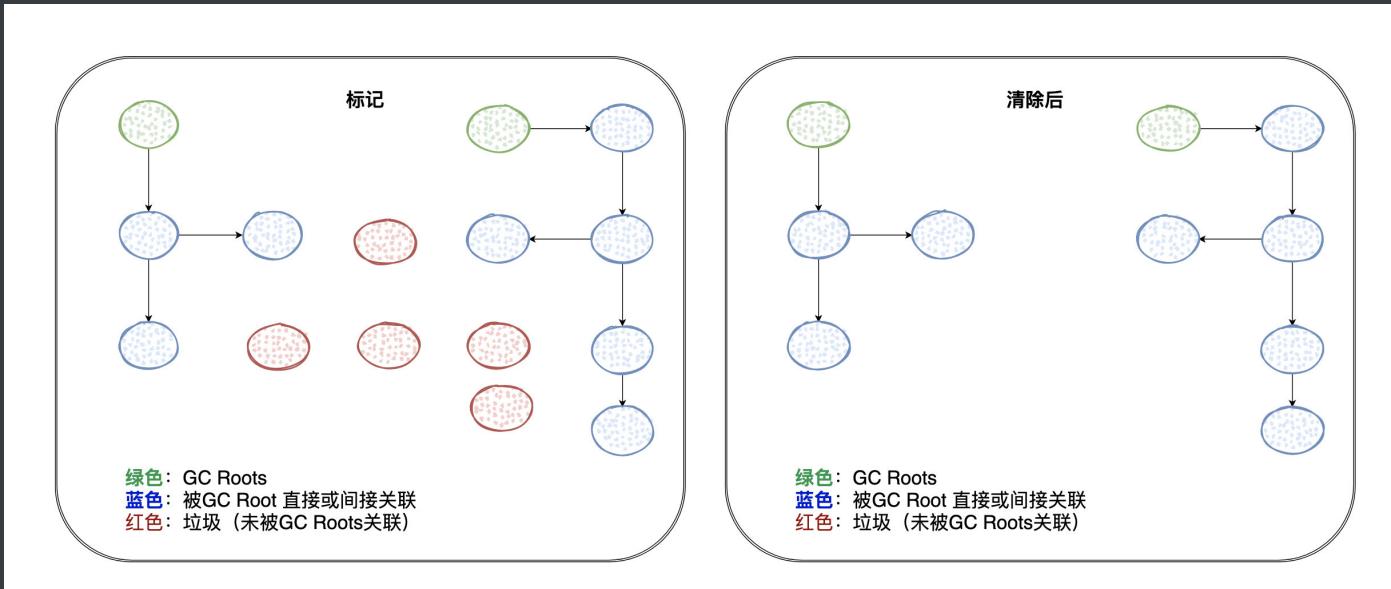
候选者：垃圾回收的第一步就是「标记」，标记哪些没有被「GC Roots」引用的对象



候选者: 标记完之后，我们就可以选择直接「清除」，只要不被「GC Roots」关联的，都可以干掉

候选者: 过程非常简单粗暴，但也存在很明显的问题

候选者: 直接清除会有「内存碎片」的问题：可能我有10M的空余内存，但程序申请9M内存空间却申请不下来（10M的内存空间是垃圾清除后的，不连续的）



候选者: 那解决「内存碎片」的问题也比较简单粗暴，「标记」完，不直接「清除」。

候选者: 我把「标记」存活的对象「复制」到另一块空间，复制完了之后，直接把原有的整块空间给干掉！这样就没有内存碎片的问题了

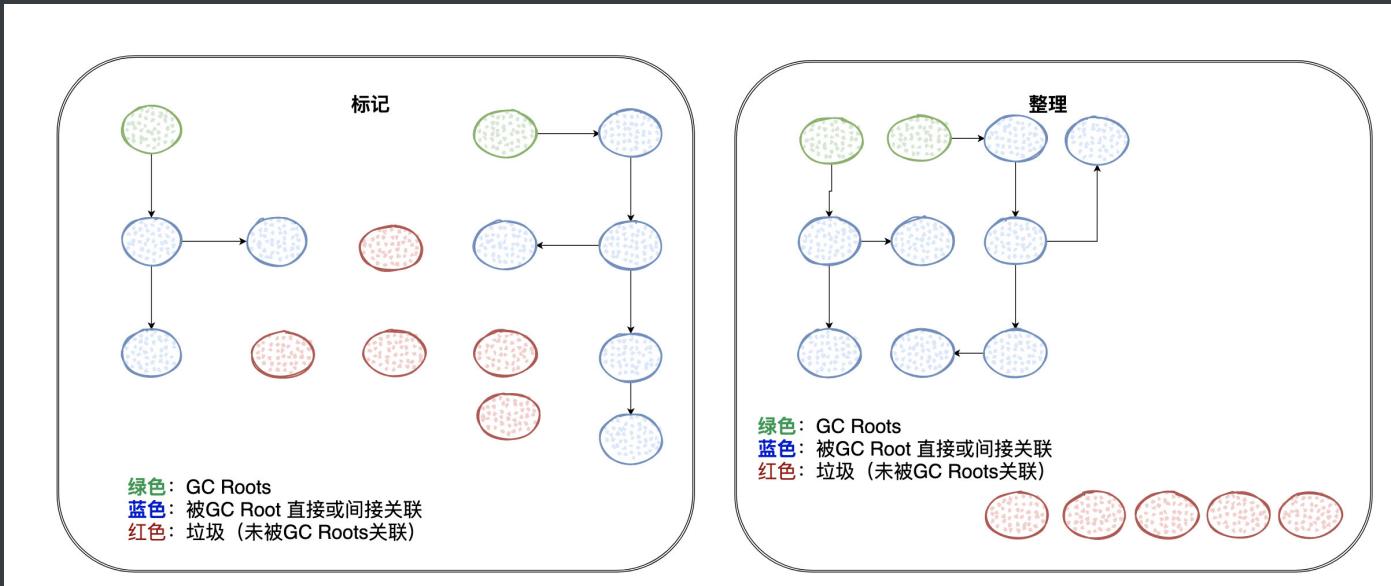
候选者: 这种做法缺点又很明显：内存利用率低，得有一块新的区域给我复制(移动)过去

面试官: 嗯...

候选者: 还有一种「折中」的办法，我未必要有一块「大的完整空间」才能解决内存碎片的问题，我只要能在「当前区域」内进行移动

候选者: 把存活的对象移到一边，把垃圾移到一边，那再将垃圾一起删除掉，不就没有内存碎片了嘛

候选者: 这种专业的术语就叫做「整理」



候选者: 扯了这么久，我们把思维再次回到「堆」中吧

候选者: 经过研究表明：大部分对象的生命周期都很短，而只有少部分对象可能会存活很长时间

候选者: 又由于「垃圾回收」是会导致「stop the world」（应用停止访问）

候选者: 理解「stop the world」应该很简单吧：回收垃圾的时候，程序是有短暂的时间不能正常继续运作啊。不然JVM在回收的时候，用户线程还继续分配修改引用，JVM怎么搞（：

候选者: 为了使「stop the world」持续的时间尽可能短以及提高并发式GC所能应付的内存分配速率

候选者: 在很多的垃圾收集器上都会在「物理」或者「逻辑」上，把这两类对象进行区分，死得快的对象所占的区域叫做「年轻代」，活得久的对象所占的区域叫做「老年代」

大部分对象生命周期都很短

为了使「stop the world」持续的时间尽可能短以及提高并发式
GC所能应付的内存分配速率

分代（年轻代、老年代）

候选者：但也不是所有的「垃圾收集器」都会有，只不过我们现在线上用的可能都是JDK8，
JDK8及以下所使用到的垃圾收集器都是有「分代」概念的。

候选者：所以，你可以看到我的「堆」是画了「年轻代」和「老年代」

候选者：要值得注意的是，高版本所使用的垃圾收集器的ZGC是没有分代的概念的（：

候选者：只不过我为了好说明现状，ZGC的话有空我们再聊

面试官：嗯...好吧

候选者：在前面更前面提到了垃圾回收的过程，其实就对应着几种「垃圾回收算法」，分别是：

候选者：标记清除算法、标记复制算法和标记整理算法【「标记」「清除」「复制」「整理」】

候选者： 经过上面的铺垫之后，这几种算法应该还是比较好理解的

标记

清除

复制

整理

候选者： 「分代」和「垃圾回收算法」都搞明白了之后，我们就可以看下在JDK8生产环境及以下常见的垃圾回收器了

候选者： 「年轻代」的垃圾收集器有：Serial、Parallel Scavenge、ParNew

候选者： 「老年代」的垃圾收集器有：Serial Old、Parallel Old、CMS

候选者： 看着垃圾收集器有很多，其实还是非常好理解的。Serial是单线程的，Parallel是多线程

候选者： 这些垃圾收集器实际上就是「实现了」垃圾回收算法（标记清除、标记复制以及标记清除算法）

候选者： CMS是「JDK8之前」是比较新的垃圾收集器，它的特点是能够尽可能减少「stop the world」时间。在垃圾回收时让用户线程和GC线程能够并发执行！

垃圾收集器

实现了

垃圾回收算法

标记清除

标记复制

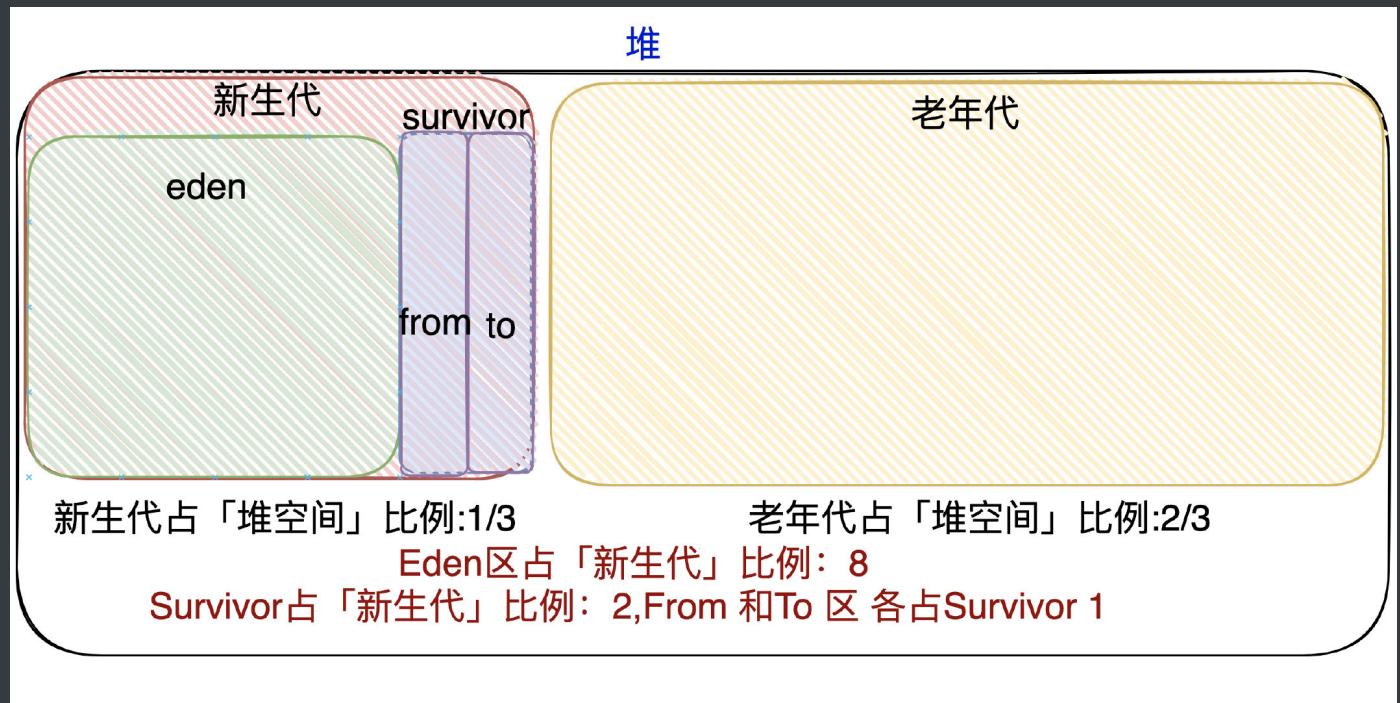
标记整理

候选者: 又可以发现的是，「年轻代」的垃圾收集器使用的都是「标记复制算法」

候选者: 所以在「堆内存」划分中，将年轻代划分出Survivor区（Survivor From 和 Survivor To），目的就是为了有一块完整的内存空间供垃圾回收器进行拷贝(移动)

候选者: 而新的对象则放入Eden区

候选者: 我下面重新画下「堆内存」的图，因为它们的大小是有默认的比例的



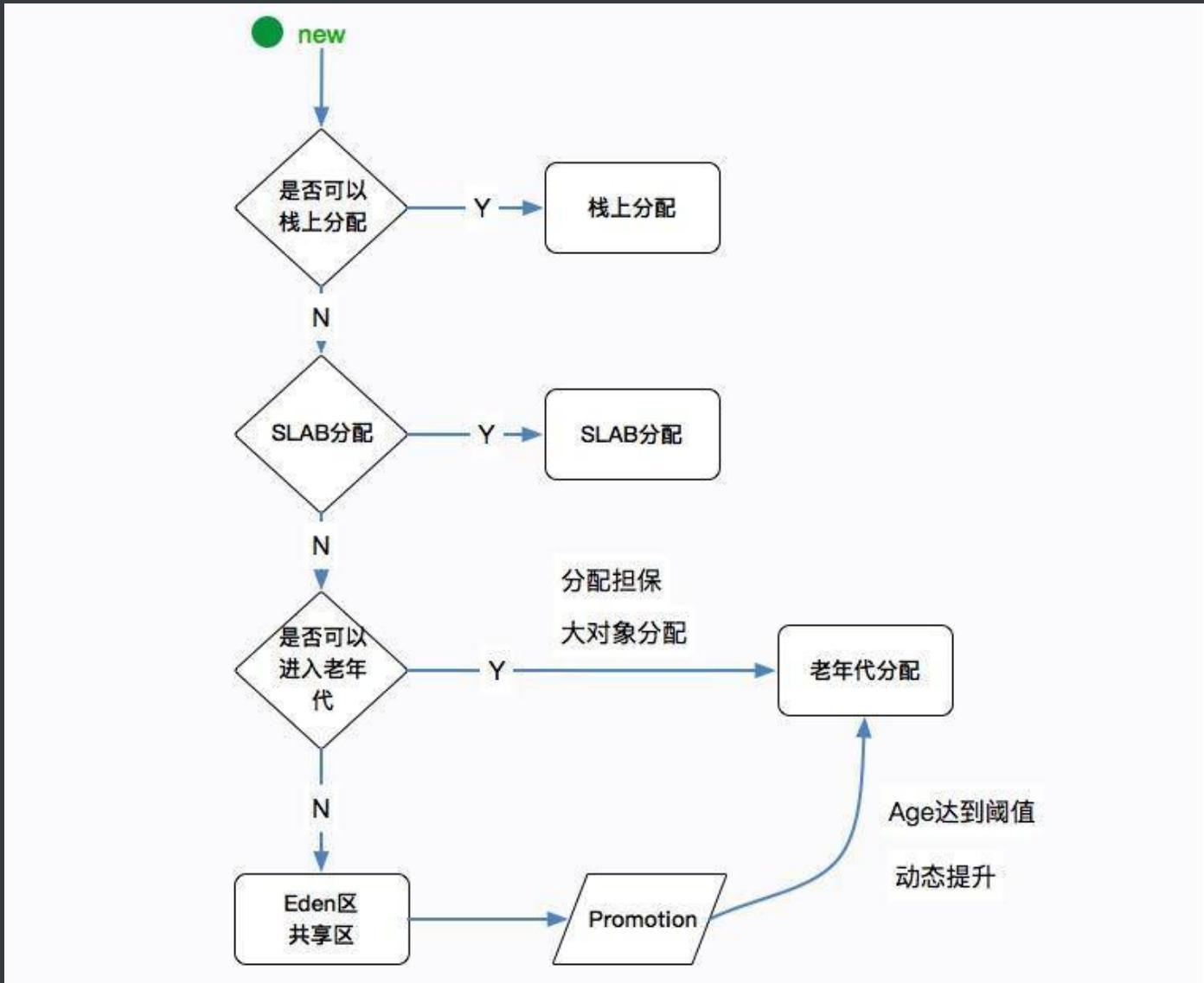
候选者: 图我已经画好了，应该就不用我再说明了

面试官: 我还想问问，就是，新创建的对象一般是在「新生代」嘛，那在什么时候会到「老年代」中呢？

候选者: 嗯，我认为简单可以分为两种情况：

候选者: 1. 如果对象太大了，就会直接进入老年代（对象创建时就很大 || Survivor区没办法存下该对象）

候选者: 2. 如果对象太老了，那就会晋升至老年代（每发生一次Minor GC，存活的对象年龄 +1，达到默认值15则晋升老年代 || 动态对象年龄判定 可以进入老年代）



面试官：既然你又提到了Minor GC，那Minor GC什么时候会触发呢？

候选者：当Eden区空间不足时，就会触发Minor GC

面试官：Minor GC 在我的理解就是「年轻代」的GC，你前面又提到了「GC Roots」嘛

面试官：那在「年轻代」GC的时候，从GC Roots出发，那不也会扫描到「老年代」的对象吗？那那那..不就相当于全堆扫描吗？

候选者：这JVM里也有解决办法的。

候选者：HotSpot 虚拟机「老的GC」（G1以下）是要求整个GC堆在连续的地址空间上。

候选者: 所以会有一条分界线（一侧是老年代，另一侧是年轻代），所以可以通过「地址」就可以判断对象在哪个分代上

候选者: 当做Minor GC的时候，从GC Roots出发，如果发现「老年代」的对象，那就不往下走了（Minor GC对老年代的区域毫无兴趣）

物理分代可以通过物理地址直接区分

面试官: 但又有个问题，那如果「年轻代」的对象被「老年代」引用了呢？（老年代对象持有年轻代对象的引用），那时候肯定是不能回收掉「年轻代」的对象的。

候选者: HotSpot虚拟机下有「card table」（卡表）来避免全局扫描「老年代」对象

候选者: 「堆内存」的每一小块区域形成「卡页」，卡表实际上就是卡页的集合。当判断一个卡页中有存在对象的跨代引用时，将这个页标记为「脏页」

候选者: 那知道了「卡表」之后，就很好办了。每次Minor GC 的时候只需要去「卡表」找到「脏页」，找到后加入至GC Root，而不用去遍历整个「老年代」的对象了。

card table 避免全局扫描老年代

跨代引用(老年代指向年轻代)，则标记为脏页

面试官: 嗯嗯嗯，还可以的啊，要不继续聊聊CMS?

候选者: 这面试快一个小时了吧，我图也画了这么多了。下次？下次吧？有点儿累了

本文总结：

- **什么是垃圾**: 只要对象不再被使用，那即是垃圾
- **如何判断为垃圾**: 可达性分析算法和引用计算算法，JVM使用的是可达性分析算法
- **什么是GC Roots**: GC Roots是一组必须活跃的引用，跟GC Roots无关联的引用即是垃圾，可被回收
- **常见的垃圾回收算法**: 标记清除、标记复制、标记整理
- **为什么需要分代**: 大部分对象都死得早，只有少部分对象会存活很长时间。在堆内存上都会在物理或逻辑上进行分代，为了使「stop the world」持续的时间尽可能短以及提高并发式GC所能应付的内存分配速率。
- **Minor GC**: 当Eden区满了则触发，从GC Roots往下遍历，年轻代GC不关心老年代对象
- **什么是card table【卡表】**: 空间换时间（类似bitmap），能够避免扫描老年代的所有对应进而顺利进行Minor GC（案例：老年代对象持有年轻代对象引用）
- **堆内存占比**: 年轻代占堆内存1/3，老年代占堆内存2/3。Eden区占年轻代8/10，Survivor区占年轻代2/10（其中From 和To 各站1/10）



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



05、CMS垃圾收集器

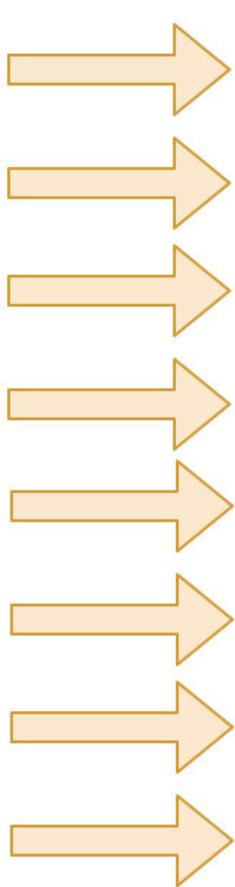
面试官：今天还是来聊聊**CMS**垃圾收集器呗？

候选者：嗯啊...

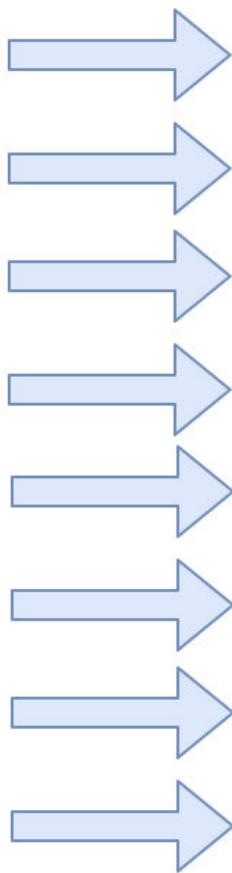
候选者：如果用**Seria**和**Parallel**系列的垃圾收集器：在垃圾回收的时，用户线程都会完全停止，直至垃圾回收结束！

Serial && Serial Old
Parallel Scanvenging && Parallel Old
ParNew 垃圾收集器

用户线程



GC线程



这些收集器 GC 线程工作时
用户线程均停止 (Stop the Word)

候选者：CMS的全称：Concurrent Mark Sweep，翻译过来是「并发标记清除」

候选者：用CMS对比上面的垃圾收集器(Serial和Parallel和parNew)：它最大的不同点就是「并发」：在GC线程工作的时候，用户线程「不会完全停止」，用户线程在「部分场景下」与GC线程一起并发执行。

候选者：但是，要理解的是，无论是什么垃圾收集器，Stop The World是一定无法避免的！

候选者：CMS只是在「部分」的GC场景下可以让GC线程与用户线程并发执行

候选者：CMS的设计目标是为了避免「老年代 GC」出现「长时间」的卡顿（Stop The World）

CMS: 用户线程与回收线程 并行

面试官：那你清楚CMS的工作流程吗？

候选者：只了解一点点，不能多了。

候选者：CMS可以简单分为5个步骤：初始标记、并发标记、并发预清理、重新标记以及并发清除

候选者：从步骤就不难看出，CMS主要是实现了「标记清除」垃圾回收算法

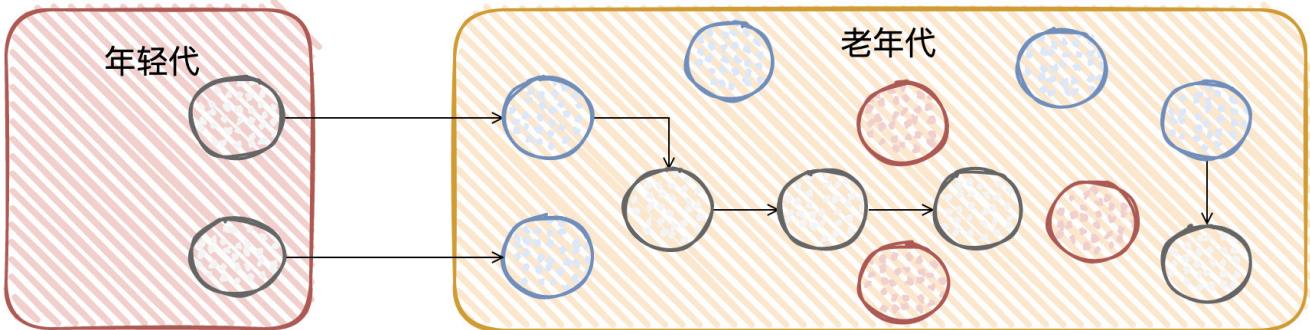
面试官：嗯...是的

候选者：我就从「初始标记」来开始吧

候选者：「初始标记」会标记GCRoots「直接关联」的对象以及「年轻代」指向「老年代」的对象

候选者：「初始标记」这个过程是会发生Stop The World的。但这个阶段的速度算是很快的，因为没有「向下追溯」（只标记一层）

初始标记（一层标记）



蓝色: GC Root

灰白色: GC Root 关联的对象 (非垃圾)

红色: GC Root 未关联的对象 (垃圾)

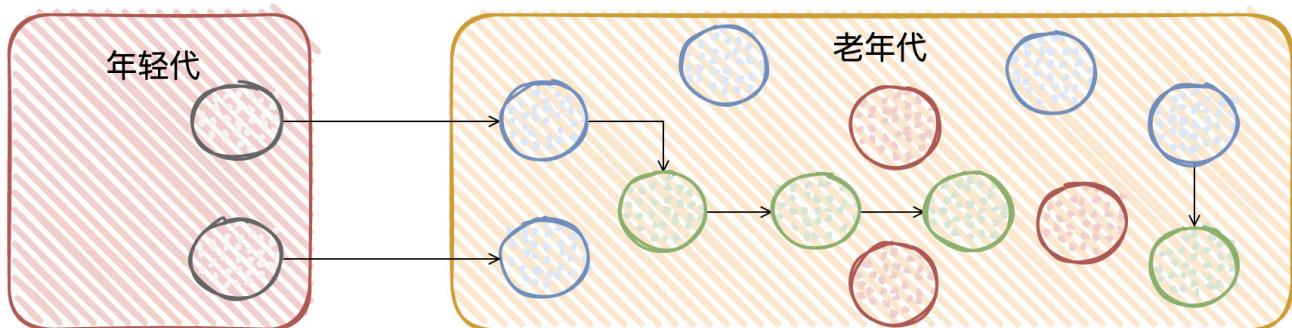
1. GC Roots 直接关联对象
2. 年轻代指向老年代的对象

候选者：在「初始标记」完了之后，就进入了「并发标记」阶段啦

候选者：「并发标记」这个过程是不会停止用户线程的（不会发生 Stop The World）。这一阶段主要是从GC Roots向下「追溯」，标记所有可达的对象。

候选者：「并发标记」在GC的角度而言，是比较耗费时间的（需要追溯）

并发标记（追溯 GC Roots下的对象）



蓝色: GC Root

绿色: 本阶段追溯GC Root 关联的对象 (非垃圾)

红色: GC Root 未关联的对象 (垃圾)

1. 追溯GC Roots关联的对象
(从灰白色变成绿色)

候选者：「并发标记」这个阶段完成之后，就到了「并发预处理」阶段啦

候选者：「并发预处理」这个阶段主要想干的事情：希望能减少下一个阶段「重新标记」所消耗的时间

候选者：因为下一个阶段「重新标记」是需要Stop The World的

面试官：嗯...

候选者：「并发标记」这个阶段由于用户线程是没有被挂起的，所以对象是有可能发生变化的

候选者：可能有些对象，从新生代晋升到了老年年代。可能有些对象，直接分配到了老年年代（大对象）。可能老年年代或者新生代的对象引用发生了变化...

面试官：那这个问题，怎么解决呢？

候选者：针对老年年代的对象，其实还是可以借助类card table的存储（将老年年代对象发生变化所对应的卡页标记为dirty）

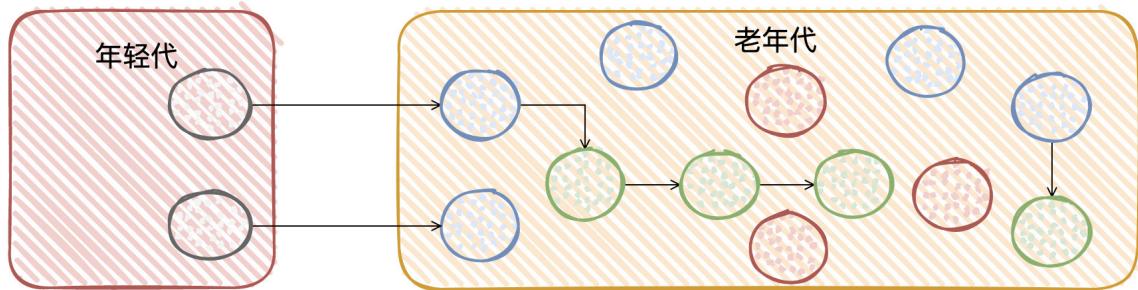
候选者：所以「并发预处理」这个阶段会扫描可能由于「并发标记」时导致老年年代发生变化的对象，会再扫描一遍标记为dirty的卡页

面试官：嗯...

候选者：对于新生代的对象，我们还是得遍历新生代来看看在「并发标记」过程中有没有对象引用了老年年代...

候选者：不过JVM里给我们提供了很多「参数」，有可能在这个过程中会触发一次 minor GC（触发了minor GC 是意味着就可以更少地遍历新生代的对象）

并发预处理（减少下一个阶段「重新标记」的处理时间）



蓝色: GC Root

绿色: 本阶段追溯GC Root 关联的对象 (非垃圾)

红色: GC Root 未关联的对象 (垃圾)

1. 并发标记阶段没有停止用户线程

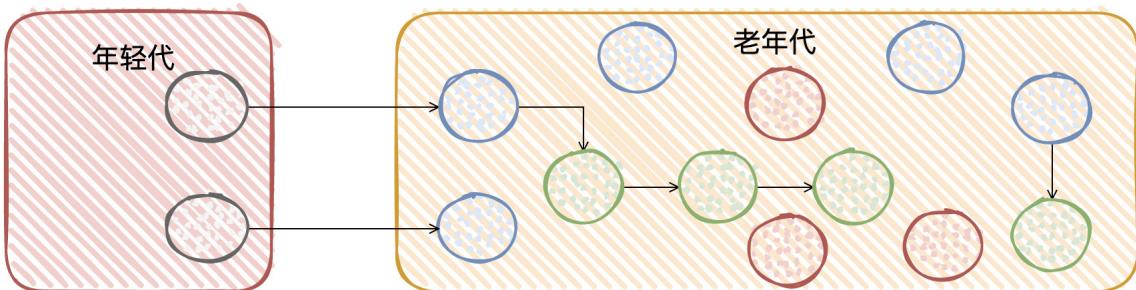
2. 老年代引用发生变化 (标记dirty), 重新标记处理

3. 新生代有可能有新的引用指向老年代, 重新标记处理 (这个过程中有可能发生Minor GC来减少扫描时间)

候选者：「并发预处理」这个阶段阶段结束后，就到了「重新标记」阶段

候选者：「重新标记」阶段会Stop The World，这个过程的停顿时间其实很大程度上取决于上面「并发预处理」阶段（可以发现，这是一个追赶的过程：一边在标记存活对象，一边用户线程在执行产生垃圾）

重新标记 (Stop The Word)



蓝色: GC Root

绿色: 本阶段追溯GC Root 关联的对象 (非垃圾)

红色: GC Root 未关联的对象 (垃圾)

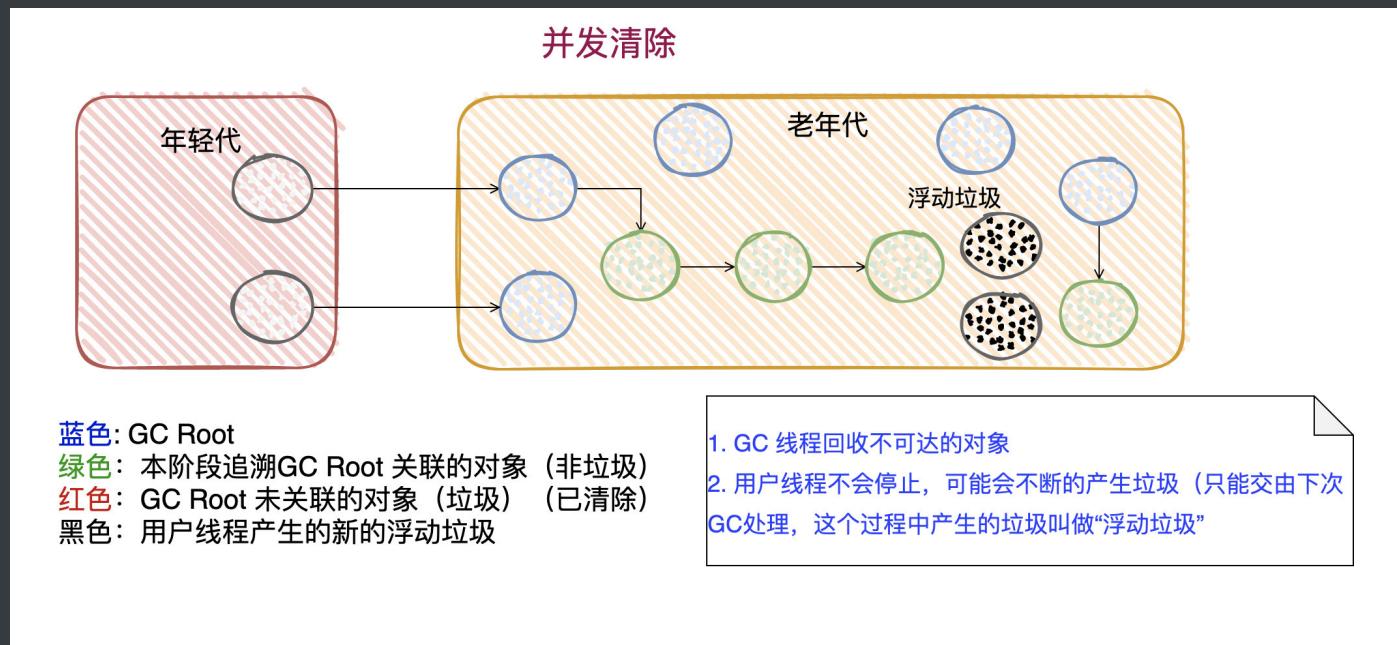
1. 停止用户线程, 扫描老年代 (dirty card) 和年轻代
找出存活的老年代对象

候选者：最后就是「并发清除」阶段，不会Stop The World

候选者：一边用户线程在执行，一边GC线程在回收不可达的对象

候选者: 这个过程，还是有可能用户线程在不断产生垃圾，但只能留到下一次GC进行处理了，产生的这些垃圾被叫做“浮动垃圾”

候选者: 完了以后会重置 CMS 算法相关的内部数据，为下一次 GC 循环做准备



面试官: 哎，CMS的回收过程，我了解了

面试官: 听下来，其实就把垃圾回收的过程给“细分”了，然后在某些阶段可以不停止用户线程，一边回收垃圾，一边处理请求，来减少每次垃圾回收时 Stop The World的时间

面试官: 当然啦，中间也做了很多的优化 (dirty card标记、可能中途触发minor gc等等，在我理解下，这些应该都提供了CMS的相关参数配置)

面试官: 不过，我看现在很多企业都在用G1了，那你觉得CMS有什么缺点呢？

候选者: 1.空间需要预留：CMS垃圾收集器可以一边回收垃圾，一边处理用户线程，那需要在这个过程中保证有充足的内存空间供用户使用。

候选者: 如果CMS运行过程中预留的空间不够用了，会报错 (Concurrent Mode Failure)，这时会启动 Serial Old垃圾收集器进行老年代的垃圾回收，会导致停顿的时间很长。

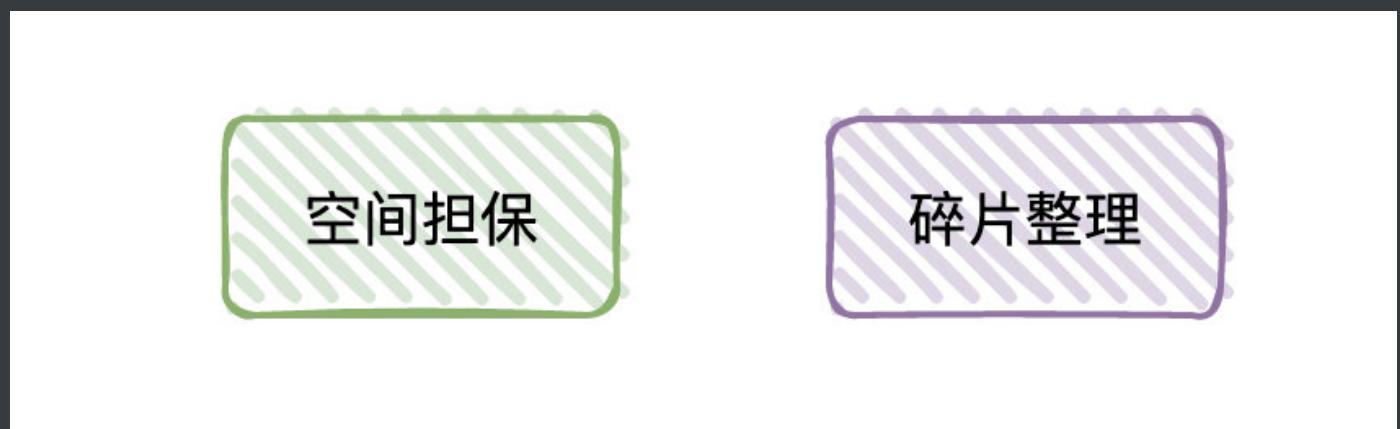
候选者: 显然啦，空间预留多少，肯定是有参数配置的

候选者: 2. 内存碎片问题: CMS本质上是实现了「标记清除算法」的收集器（从过程就可以看得出），这会意味着会产生内存碎片

候选者: 由于碎片太多，又可能会导致内存空间不足所触发full GC, CMS一般会在触发full GC这个过程对碎片进行整理

候选者: 整理涉及到「移动」 / 「标记」，那这个过程肯定会Stop The World的，如果内存足够大（意味着可能装载的对象足够多），那这个过程卡顿也是需要一定的时间的。

面试官: 嗯...



候选者: 使用CMS的弊端好像就是一个死循环:

候选者: 1. 内存碎片过多，导致空间利用率减低。

候选者: 2. 空间本身就需要预留给用户线程使用，现在碎片内存又加剧了空间的问题，导致有可能垃圾收集器降级为Serial Old，卡顿时间更长。

候选者: 3. 要处理内存碎片的问题（整理），同样会卡顿

候选者: 不过，技术实现就是一种trade-off（权衡），不可能你把所有的事情都做得很完美

候选者: 了解这个过程，是非常有趣的

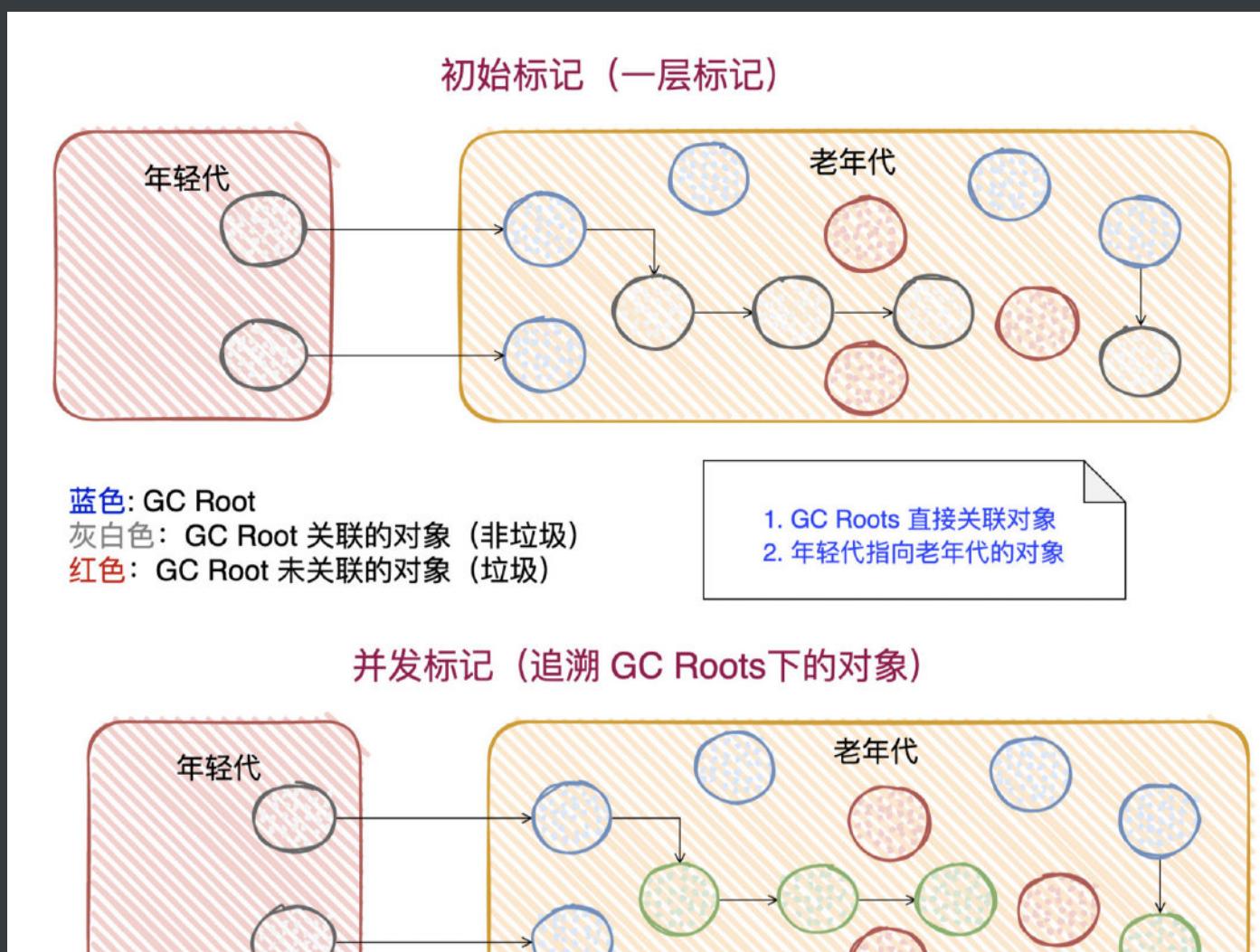
面试官：那G1垃圾收集器你了解吗

候选者：只了解一点点，不能多了

候选者：不过，留到下次吧，先让你消化下，不然怕你顶不住了。

本文总结：

- **CMS垃圾回收器设计目的**：为了避免「老年代 GC」出现「长时间」的卡顿（Stop The World）
- **CMS垃圾回收器回收过程**：初始标记、并发标记、并发预处理、重新标记和并发清除。初始标记以及重新标记这两个阶段会Stop The World
- **CMS垃圾回收器的弊端**：会产生内存碎片&&需要空间预留：停顿时间是不可预知的



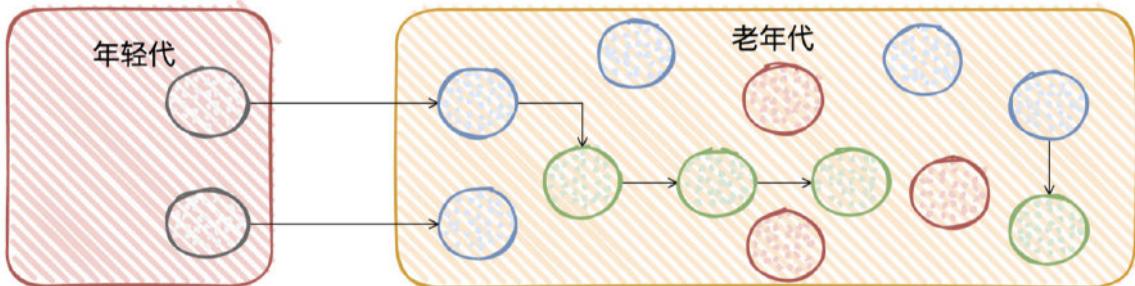
蓝色: GC Root

绿色: 本阶段追溯GC Root 关联的对象 (非垃圾)

红色: GC Root 未关联的对象 (垃圾)

1. 追溯GC Roots关联的对象
(从灰白色变成绿色)

并发预处理 (减少下一个阶段「重新标记」的处理时间)



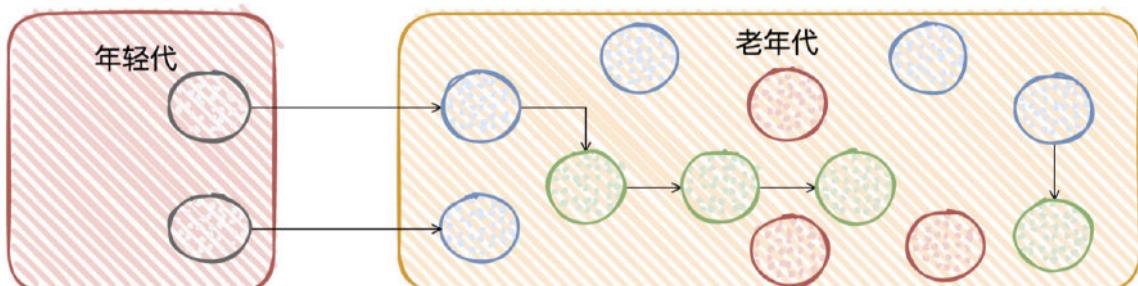
蓝色: GC Root

绿色: 本阶段追溯GC Root 关联的对象 (非垃圾)

红色: GC Root 未关联的对象 (垃圾)

1. 并发标记阶段没有停止用户线程
2. 老年代引用发生变化 (标记dirty), 重新标记处理
3. 新生代有可能有新的引用指向老年代, 重新标记处理 (这个过程中有可能发生Minor GC来减少扫描时间)

重新标记 (Stop The Word)



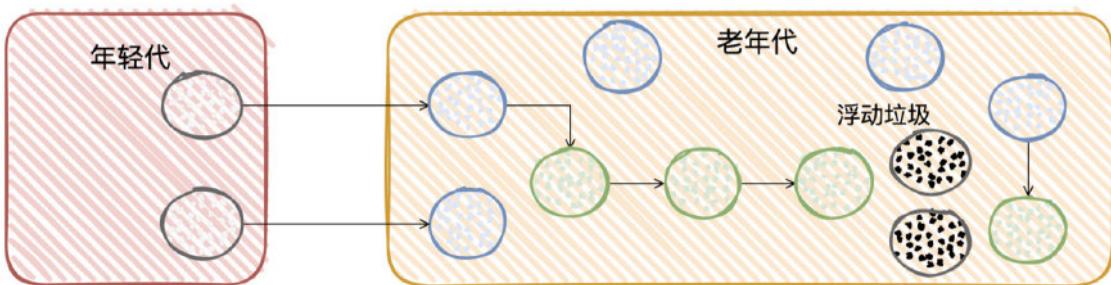
蓝色: GC Root

绿色: 本阶段追溯GC Root 关联的对象 (非垃圾)

红色: GC Root 未关联的对象 (垃圾)

1. 停止用户线程, 扫描老年代 (dirty card) 和年轻代
找出存活的老年代对象

并发清除



蓝色: GC Root

1. GC 线程回收不可达的对象

绿色：本阶段追溯GC Root 关联的对象（非垃圾）
红色：GC Root 未关联的对象（垃圾）（已清除）
黑色：用户线程产生的新的浮动垃圾

2. 用户线程不会停止，可能会不断的产生垃圾（只能交由下次GC处理，这个过程中产生的垃圾叫做“浮动垃圾”）



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zhongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

06、G1垃圾收集器

面试官：要不这次来聊聊G1垃圾收集器？

候选者：嗯嗯，好的呀

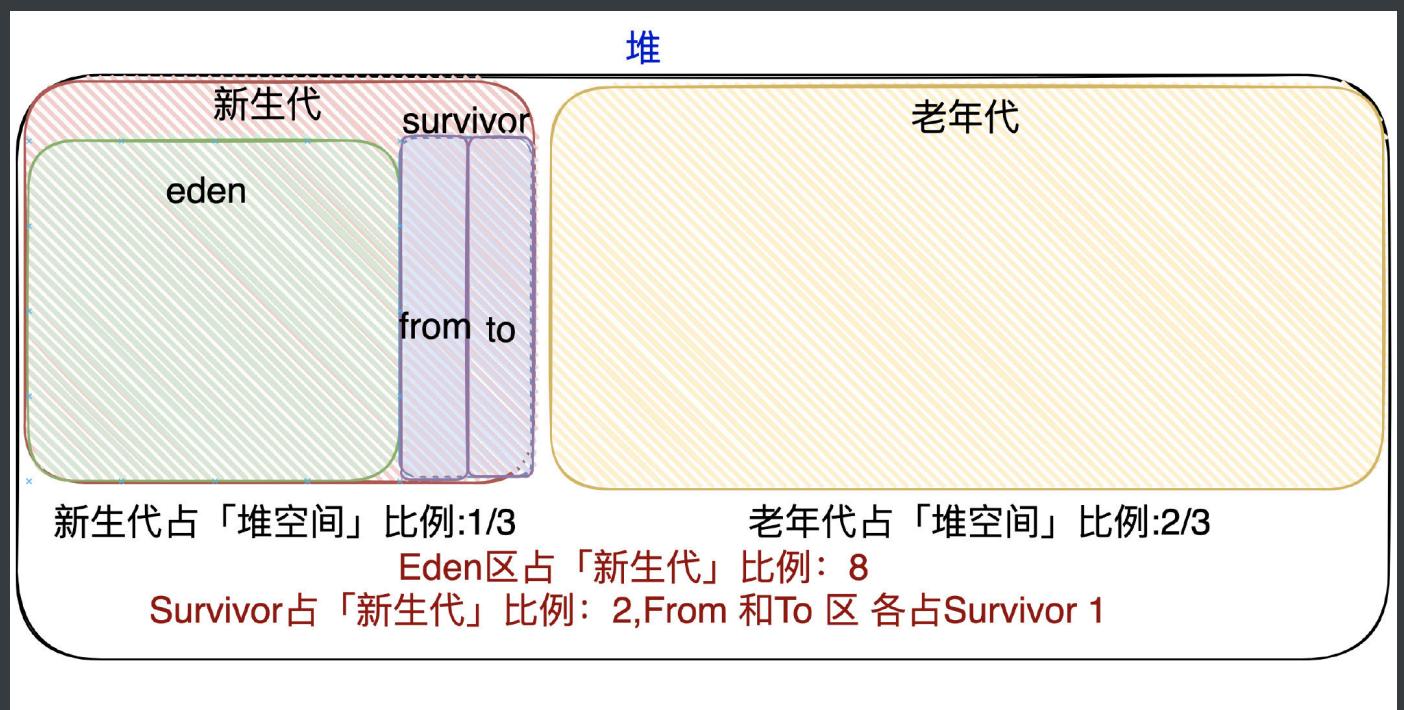
候选者：上次我记得说过，CMS垃圾收集器的弊端：会产生内存碎片&&空间需要预留

候选者: 这两个问题在处理的时候，很有可能会导致停顿时间过长，说白了就是CMS的停顿时间是「不可预知的」

候选者: 而G1又可以理解为在CMS垃圾收集器上进行"升级"

候选者: G1 垃圾收集器可以给你设定一个你希望Stop The Word 停顿时间，G1垃圾收集器会根据这个时间尽量满足你

候选者: 在前面我在介绍JVM堆的时候，是画了一张图的。堆的内存分布是以「物理」空间进行隔离

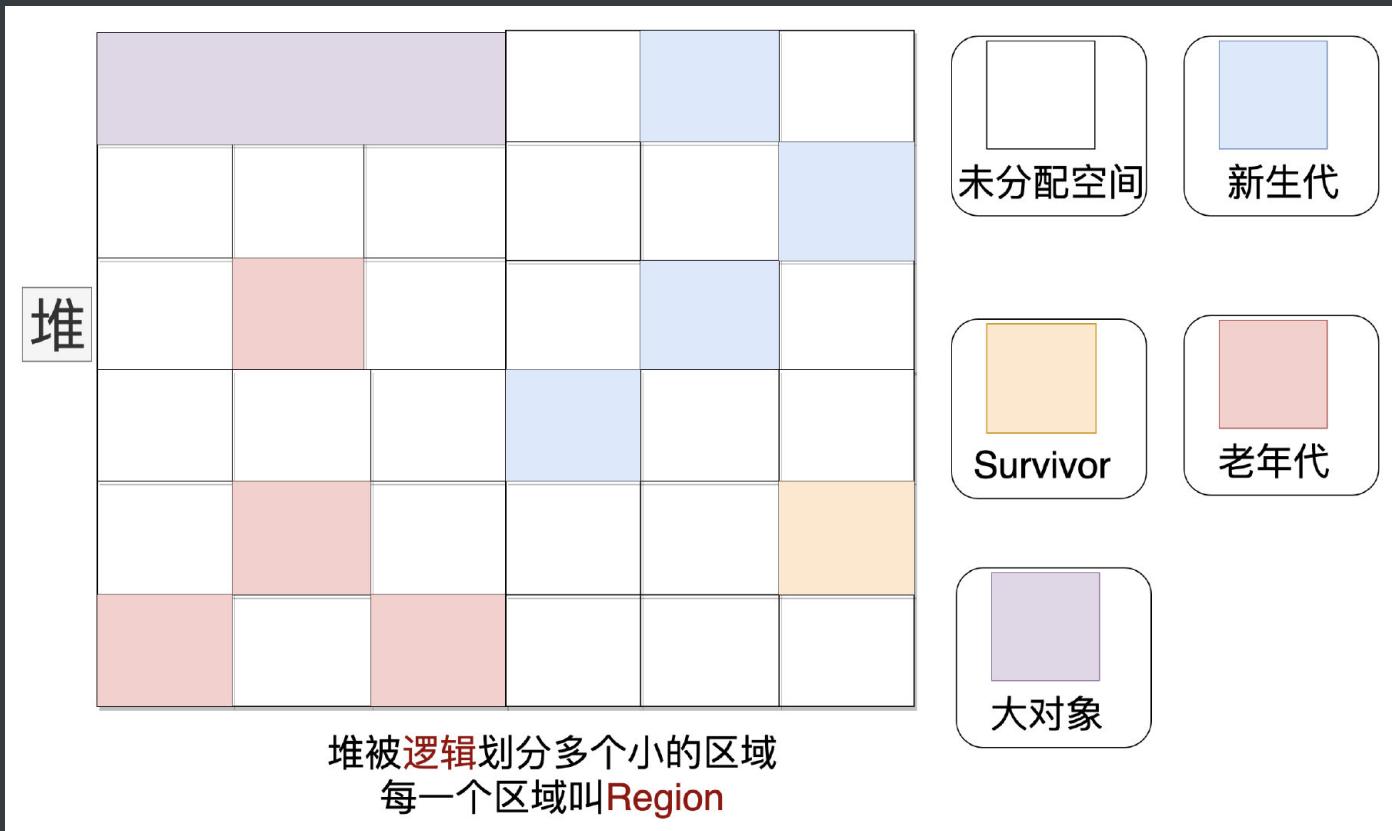


候选者: 在G1垃圾收集器的世界上，堆的划分不再是「物理」形式，而是以「逻辑」的形式进行划分

候选者: 不过，像之前说过的「分代」概念在G1垃圾收集器的世界还是一样奏效的

候选者: 比如说：新对象一般会分配到Eden区、经过默认15次的Minor GC新生代的对象如果还存活，会移交到老年代等等...

候选者: 我来画下G1垃圾收集器世界的「堆」空间分布吧



候选者：从图上就可以发现，堆被划分了多个同等份的区域，在G1里每个区域叫做Region

候选者：老年代、新生代、Survivor这些应该就不用我多说了吧？规则是跟CMS一样的

候选者：G1中，还有一种叫 Humongous（大对象）区域，其实就是用来存储特别大的对象（大于Region内存的一半）

候选者：一旦发现没有引用指向大对象，就可直接在年轻代的Minor GC中被回收掉

面试官：嗯...

候选者：其实稍微想一下，也能理解为什么要将「堆空间」进行「细分」多个小的区域

候选者：像以前的垃圾收集器都是对堆进行「物理」划分

候选者：如果堆空间（内存）大的时候，每次进行「垃圾回收」都需要对一整块大的区域进行回收，那收集的时间是不好控制的

候选者：而划分多个小区域之后，那对这些「小区域」回收就容易控制它的「收集时间」了

逻辑划分多个小区域，垃圾回收容易控制StopTheWorld 时间

面试官：嗯...

面试官：那我大概了解了。那要不你讲讲它的GC过程呗？

候选者：嗯，在G1收集器中，可以主要分为有Minor GC(Young GC)和Mixed GC，也有些特殊场景可能会发生Full GC

候选者：那我就直接说Minor GC先咯？

面试官：嗯，开始吧

候选者：G1的Minor GC其实触发时机跟前面提到过的垃圾收集器都是一样的

候选者：等到Eden区满了之后，会触发Minor GC。Minor GC同样也是会发生Stop The World 的

候选者：要补充说明的是：在G1的世界里，新生代和老年代所占堆的空间是没那么固定的（会动态根据「最大停顿时间」进行调整）

候选者：这块要知道会给我们提供参数进行配置就好了

候选者：所以，动态地改变年轻代Region的个数可以「控制」Minor GC的开销

在G1中的年轻代比例会动态调整，同样也是Eden区满了触发Minor GC

面试官：嗯，那Minor GC它的回收过程呢？可以稍微详细补充一下吗

候选者：Minor GC我认为可以简单分为三个步骤：根扫描、更新&&处理 RSet、复制对象

候选者：第一步应该很好理解，因为这跟之前CMS是类似的，可以理解为初始标记的过程

候选者：第二步涉及到「Rset」的概念

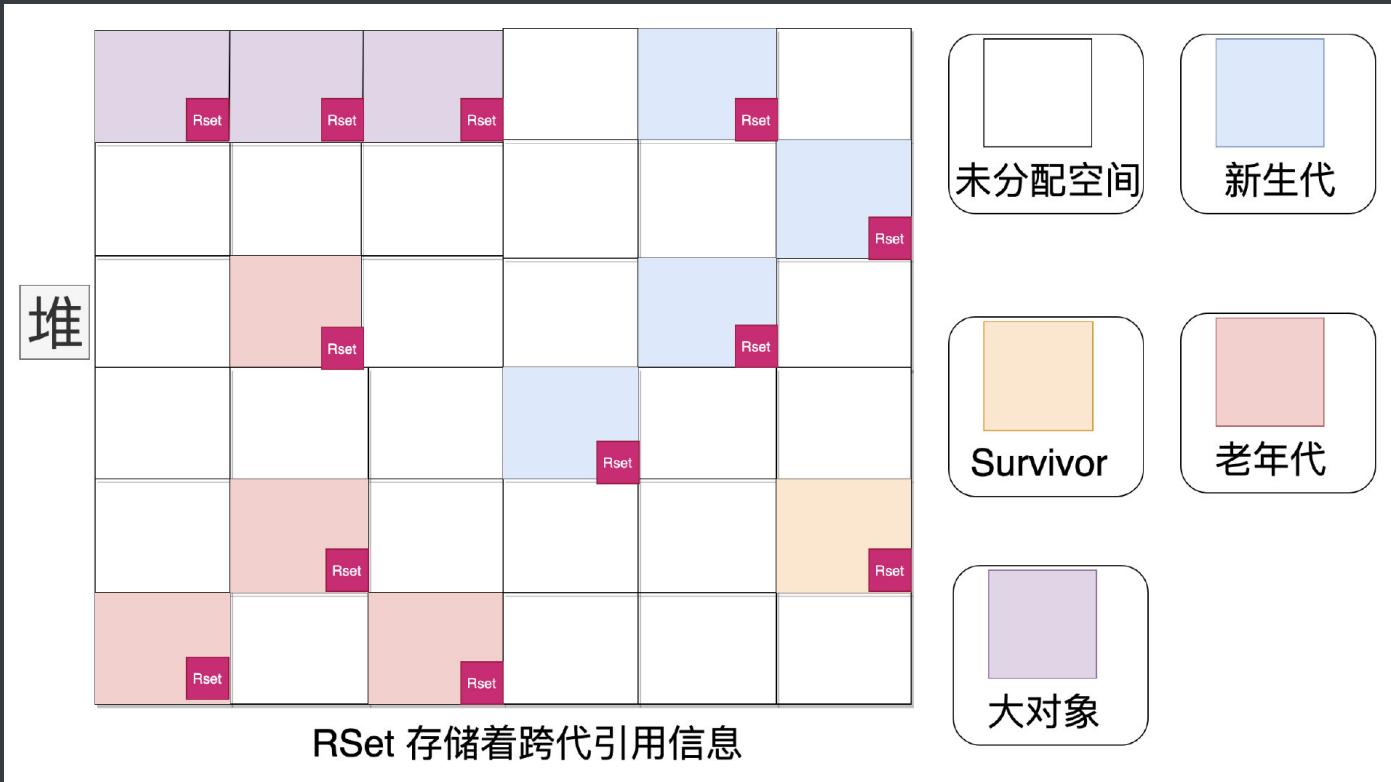
面试官：嗯...

候选者：从上一次我们聊CMS回收过程的时候，同样讲到了Minor GC，它是通过「卡表」(cart table)来避免全表扫描老年代的对象

候选者：因为Minor GC 是回收年轻代的对象，但如果老年代有对象引用着年轻代，那这些被老年代引用的对象也不能回收掉

候选者：同样的，在G1也有这种问题（毕竟是Minor GC）。CMS是卡表，而G1解决「跨代引用」的问题的存储一般叫做RSet

候选者：只要记住，RSet这种存储在每个Region都会有，它记录着「其他Region引用了当前Region的对象关系」



候选者: 对于年轻代的Region, 它的RSet 只保存了来自老年代的引用 (因为年轻代的没必要存储啊, 自己都要做Minor GC了)

候选者: 而对于老年代的 Region 来说, 它的 RSet 也只会保存老年代对它的引用 (在G1垃圾收集器, 老年代回收之前, 都会先对年轻代进行回收, 所以没必要保存年轻代的引用)

面试官: 嗯...

候选者: 那第二步看完RSet的概念, 应该也好理解了吧?

候选者: 无非就是处理RSet的信息并且扫描, 将老年代对象持有年轻代对象的相关引用都加入到GC Roots下, 避免被回收掉

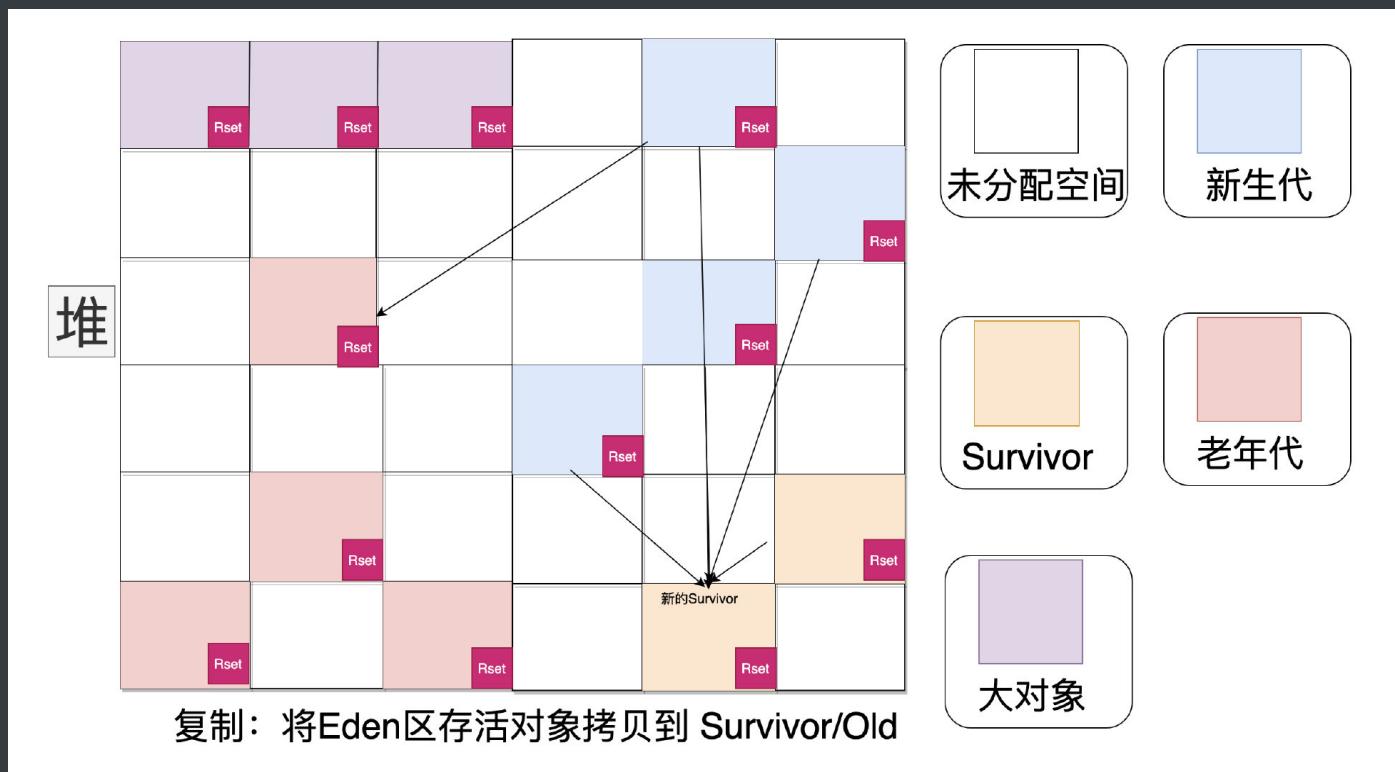
候选者: 到了第三步也挺好理解的: 把扫描之后存活的对象往「空的Survivor区」或者「老年代」存放, 其他的Eden区进行清除



候选者: 这里要提下的是，在G1还有另一个名词，叫做CSet。

候选者: 它的全称是 Collection Set，保存了一次GC中「将执行垃圾回收」的Region。CSet中的所有存活对象都会被转移到别的可用Region上

候选者: 在Minor GC 的最后，会处理下软引用、弱引用、JNI Weak等引用，结束收集



面试官: 嗯，了解了，不难

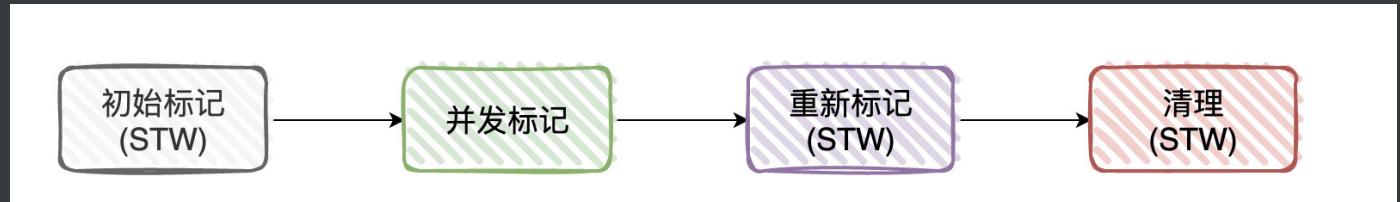
面试官: 我记得你前面提到了Mixed GC，要不来聊下这个过程呗？

候选者: 好，没问题的。

候选者: 当堆空间的占用率达到一定阈值后会触发Mixed GC (默认45%，由参数决定)

候选者: Mixed GC 依赖「全局并发标记」统计后的Region数据

候选者: 「全局并发标记」它的过程跟CMS非常类似，步骤大概是：初始标记（STW）、并发标记、重新标记（STW）以及清理（STW）



面试官: 确实很像啊，你继续来聊聊具体的过程呗？

候选者: 嗯嗯，还是想说明下：Mixed GC它一定会回收年轻代，并会采集部分老年代的Region进行回收的，所以它是一个“混合”GC。

候选者: 首先是「初始标记」，这个过程是「共用」了Minor GC的 Stop The World (Mixed GC 一定会发生 Minor GC)，复用了「扫描GC Roots」的操作。

候选者: 在这个过程中，老年代和新生代都会扫

候选者: 总的来说，「初始标记」这个过程还是比较快的，毕竟没有追溯遍历嘛

面试官: ...

候选者: 接下来就到了「并发标记」，这个阶段不会Stop The World

候选者: GC线程与用户线程一起执行，GC线程负责收集各个 Region 的存活对象信息

候选者: 从GC Roots往下追溯，查找整个堆存活的对象，比较耗时

面试官：嗯...

候选者：接下来就到「重新标记」阶段，跟CMS又一样，标记那些在「并发标记」阶段发生变化的对象

候选者：是不是很简单？

面试官：且慢

面试官：CMS在「重新标记」阶段，应该会重新扫描所有的线程栈和整个年轻代作为root

面试官：据我了解，G1好像不是这样的，这块你了解吗？

候选者：嗯，G1确实不是这样的，在G1中解决「并发标记」阶段导致引用变更的问题，使用的是SATB算法

候选者：可以简单理解为：在GC开始的时候，它为存活的对象做了一次「快照」

候选者：在「并发阶段」时，把每一次发生引用关系变化时旧的引用值给记下来

候选者：然后在「重新标记」阶段只扫描着块「发生过变化」的引用，看有没有对象还是存活的，加入到「GC Roots」上

STAB算法

快照：只记录被更改过的引用

候选者: 不过SATB算法有个小的问题，就是：如果在开始时，G1就认为它是活的，那就在此次GC中不会对它回收，即便可能在「并发阶段」上对象已经变为了垃圾。

候选者: 所以，G1也有可能会存在「浮动垃圾」的问题

候选者: 但是总的来说，对于G1而言，问题不大（毕竟它不是追求一次把所有的垃圾都清除掉，而是注重 Stop The World时间）

面试官: 嗯...

候选者: 最后一个阶段就是「清理」，这个阶段也是会Stop The World的，主要清点和重置标记状态

候选者: 会根据「停顿预测模型」（其实就是设定的停顿时间），来决定本次GC回收多少Region

候选者: 一般来说，Mixed GC会选定所有的年轻代Region，部分「回收价值高」的老年代Region（回收价值高其实就是垃圾多）进行采集

候选者: 最后Mixed GC 进行清除还是通过「拷贝」的方式去干的

候选者: 所以，一次回收未必是将所有的垃圾进行回收的，G1会依据停顿时间做出选择Region数量（：

一般会回收年轻代所有区域以及老年代部分区域

面试官: 嗯，过程我大致是了解了

面试官: 那G1会什么时候发生full GC?

候选者: 如果在Mixed GC中无法跟上用户线程分配内存的速度，导致老年代填满无法继续进行Mixed GC，就又会降级到serial old GC来收集整个GC heap

候选者: 不过这个场景相较于CMS还是很少的，毕竟G1没有CMS内存碎片这种问题 (:

本文总结(**G1垃圾收集器特点**):

- 从原来的「物理」分代，变成现在的「逻辑」分代，将堆内存「逻辑」划分为多个 Region
- 使用CSet来存储可回收Region的集合
- 使用RSet来处理跨代引用的问题（注意：RSet不保留年轻代相关的引用关系）
- G1可简单分为：Minor GC 和Mixed GC以及Full GC
- 【Eden区满则触发】Minor GC 回收过程可简单分为：(STW) 扫描 GC Roots、更新&&处理Rset、复制清除
- 【整堆空间占一定比例则触发】Mixed GC 依赖「全局并发标记」，得到CSet(可回收 Region)，就进行「复制清除」
- R大描述G1原理的时候，从宏观的角度看G1其实就是「全局并发标记」和「拷贝存活对象」
- 使用SATB算法来处理「并发标记」阶段对象引用可能会修改的问题
- 提供可停顿时间参数供用户设置 (**G1会尽量满足该停顿时间来调整 GC时回收Region的数量**)



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「**对线面试官**」



07、JVM调优

面试官：今天要不来聊聊JVM调优相关的吧？

面试官：你曾经在生产环境下有过调优JVM的经历吗？

候选者：没有

面试官：...

候选者：嗯...是这样的，我们一般优化系统的思路是这样的

候选者：1. 一般来说关系型数据库是先到瓶颈，首先排查是否为数据库的问题

候选者：（这个过程中就需要评估自己建的索引是否合理、是否需要引入分布式缓存、是否需要分库分表等等）

候选者：2. 然后，我们会考虑是否需要扩容（横向和纵向都会考虑）

候选者：（这个过程中我们会怀疑是系统的压力过大或者是系统的硬件能力不足导致系统频繁出现问题）

候选者：3. 接着，应用代码层面上排查并优化

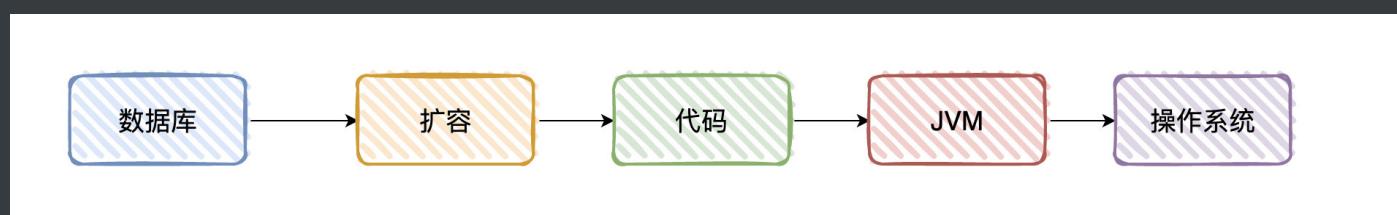
候选者：（扩容是不能无止境的，里头里外都是钱阿。这个过程中我们会审视自己写的代码是否存在资源浪费的问题，又或者是在逻辑上可存在优化的地方，比如说通过并行的方式处理某些请求）

候选者：4. 再接着，JVM层面上排查并优化

候选者：（审视完代码之后，这个过程我们观察JVM是否存在多次GC问题等等）

候选者：5. 最后，网络和操作系统层面排查

候选者：（这个过程查看内存/CPU/网络/硬盘读写指标是否正常等等）



候选者：绝大多数情况下，到第三步就结束了，一般经过「运维团队」给我们设置的JVM和机器上的参数，已经满足绝大多数的需求了。

候选者：之前有过其他团队在「大促」发现接口处理超时的问题，那时候查各种监控怀疑是 FULL GC导致的

候选者：第一想法不是说去调节各种JVM参数来进行优化，而是直接加机器

候选者：（用最粗暴的方法，解决问题是最简单的，扩容YYDS）

面试官：确实

候选者：不过，我是学过JVM相关的调优命令和思路的。

候选者：在我的理解下，调优JVM其实就是在「理解」JVM内存结构以及各种垃圾收集器前提下，结合自己的现有的业务来「调整参数」，使自己的应用能够正常稳定运行。

候选者：一般调优JVM我们认为会有几种指标可以参考：『吞吐量』、『停顿时间』和『垃圾回收频率』

候选者：基于这些指标，我们就有可能需要调整：

候选者：1. 内存区域大小以及相关策略（比如整块堆内存占多少、新生代占多少、老年代占多少、Survivor占多少、晋升老年代的条件等等）

候选者：比如（-Xmx：设置堆的最大值、-Xms：设置堆的初始值、-Xmn：表示年轻代的大小、-XX:SurvivorRatio：伊甸区和幸存区的比例等等）

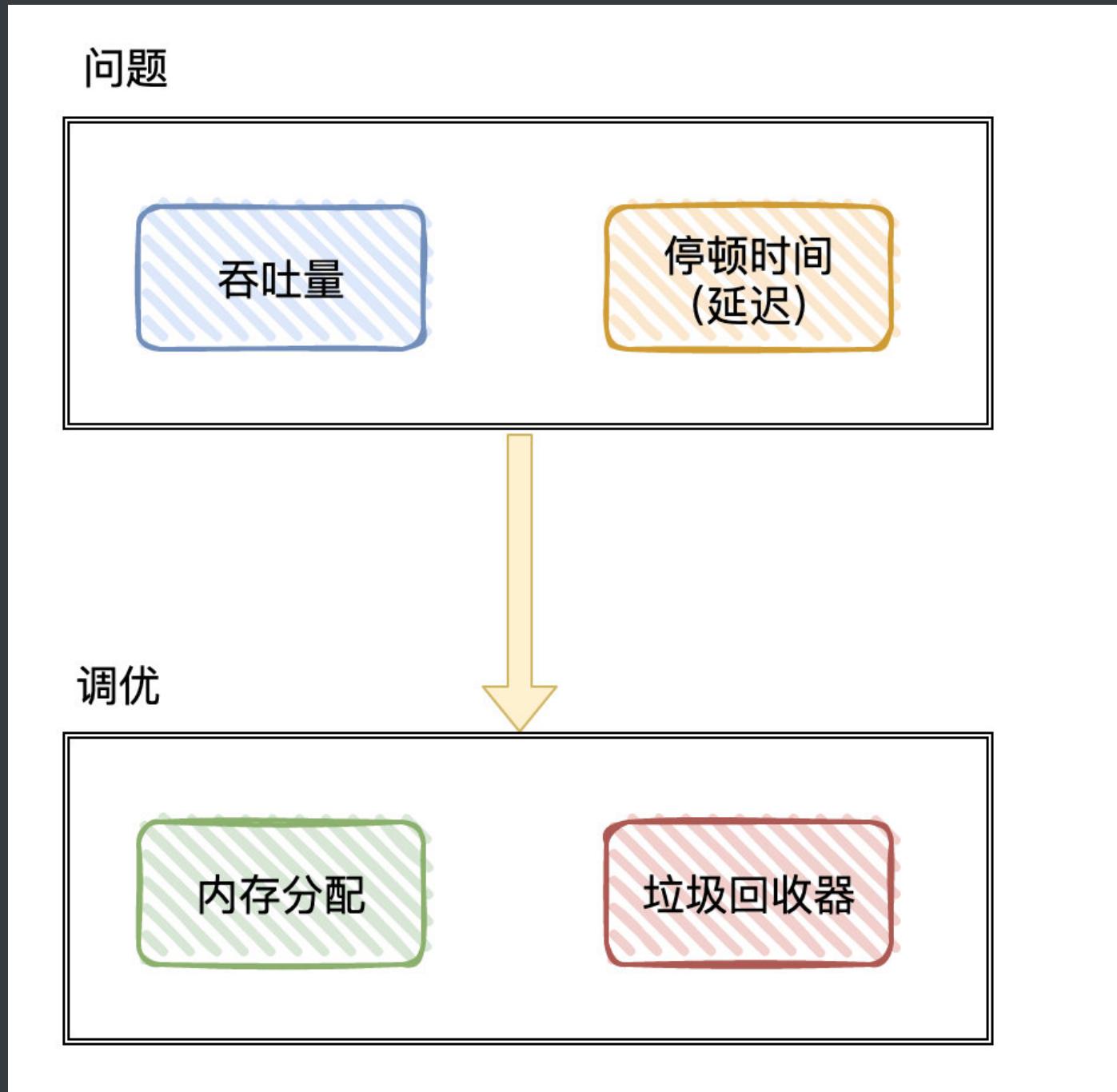
候选者：（按经验来说：IO密集型的可以稍微把「年轻代」空间加大些，因为大多数对象都是在年轻代就会灭亡。内存计算密集型的可以稍微把「老年代」空间加大些，对象存活时间会更长些）

候选者：2. 垃圾回收器（选择合适的垃圾回收器，以及各个垃圾回收器的各种调优参数）

候选者：比如（-XX:+UseG1GC：指定JVM使用的垃圾回收器为G1、-XX:MaxGCPauseMillis：设置目标停顿时间、-XX:InitiatingHeapOccupancyPercent：当整个堆内存使用达到一定比例，全局并发标记阶段就会被启动等等）

候选者：没错，这些都是因地制宜，具体问题具体分析（前提是得懂JVM的各种基础知识，基础知识都不懂，谈何调优）

候选者：在大多数场景下，JVM 已经能够达到「开箱即用」



面试官：确实

候选者：一般我们是「遇到问题」之后才进行调优的，而遇到问题后需要利用各种的「工具」进行排查

候选者：1. 通过jps命令查看Java进程「基础」信息（进程号、主类）。这个命令很常用的就
是用来看当前服务器有多少Java进程在运行，它们的进程号和加载主类是啥

候选者：2. 通过jstat命令查看Java进程「统计类」相关的信息（类加载、编译相关信息统
计，各个内存区域GC概况和统计）。这个命令很常用于看GC的情况

候选者：3. 通过jinfo命令来查看和调整Java进程的「运行参数」。

候选者：4. 通过jmap命令来查看Java进程的「内存信息」。这个命令很常用于把JVM内存信
息dump到文件，然后再用MAT(Memory Analyzer tool 内存解析工具)把文件进行分析

候选者：5. 通过jstack命令来查看JVM「线程信息」。这个命令常用于排查死锁相关的问题

候选者：6. 还有近期比较热门的Arthas（阿里开源的诊断工具），涵盖了上面很多命令的功
能且自带图形化界面。这也是我这边常用的排查和分析工具



面试官：嗯...好吧。之前聊JVM的时候，你也提到过在「解释」阶段，会有两种方式把字节码
信息解释成机器指令码，一个是字节码解释器、一个是即时编译器(JIT)

面试官：我想问问，你了解JVM的JIT优化技术嘛？

候选者：JIT优化技术比较出名的有两种：方法内联和逃逸分析

候选者: 所谓方法内联就是把「目标方法」的代码复制到「调用的方法」中，避免发生真实的方法调用

候选者: 因为每次方法调用都会生成栈帧（压栈出栈记录方法调用位置等等）会带来一定的性能损耗，所以「方法内联」的优化可以提高一定的性能

候选者: 在JVM中也有相关的参数给予我们指定 (-XX:MaxFreqInlineSize、 -XX:MaxInlineSize)

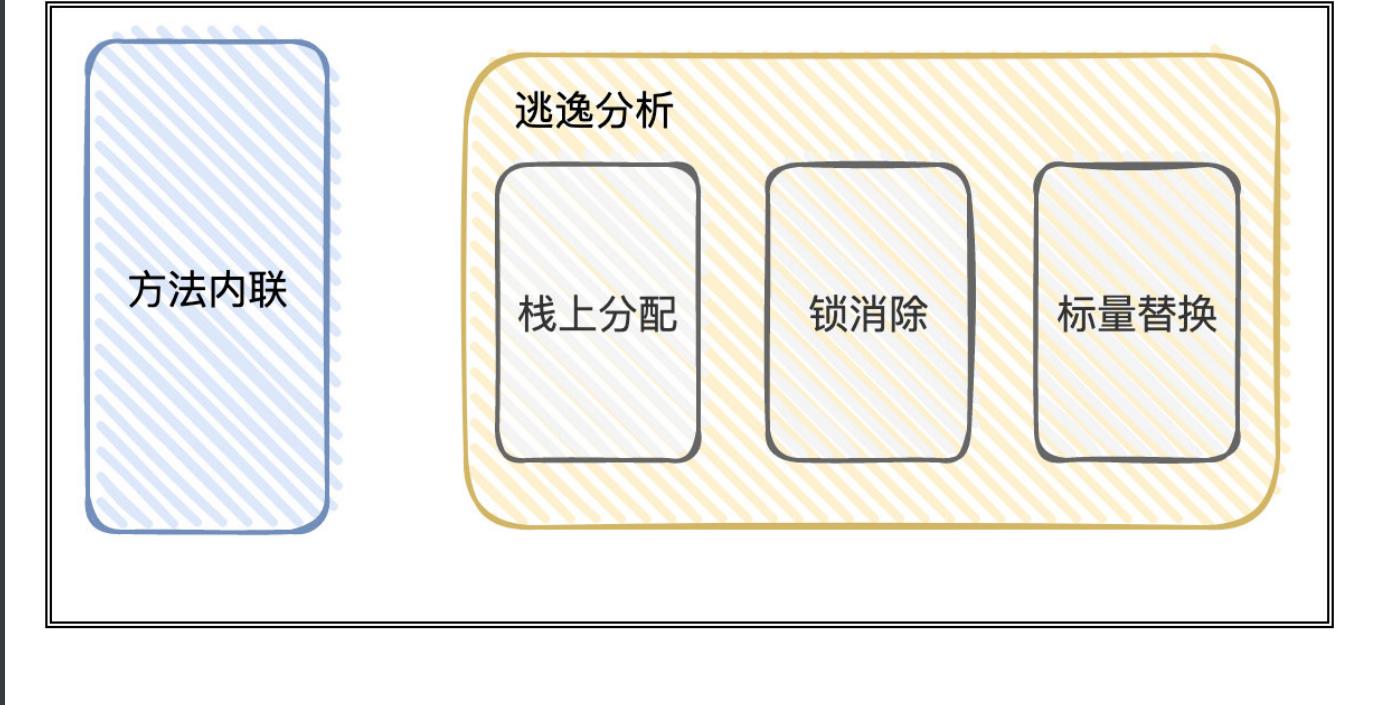
候选者: 而「逃逸分析」则是判断一个对象是否被外部方法引用或外部线程访问的分析技术，如果「没有被引用」，就可以对其进行优化，比如说：

候选者: 1. 锁消除（同步忽略）：该对象只在方法内部被访问，不会被别的地方引用，那么就一定是线程安全的，可以把锁相关的代码给忽略掉

候选者: 2. 栈上分配：该对象只会在方法内部被访问，直接将对象分配在「栈」中（Java默认是将对象分配在「堆」中，是需要通过JVM垃圾回收期进行回收，需要损耗一定的性能，而栈内分配则快很多）

候选者: 3. 标量替换/分离对象：当程序真正执行的时候可以不创建这个对象，而直接创建它的成员变量来代替。将对象拆分后，可以分配对象的成员变量在栈或寄存器上，原本的对象就无需分配内存空间了

JIT常见优化



候选者：不过扯了这么多，不同的JVM版本对JIT的优化都不太相同（：这里也只能算是一个参考

面试官：懂了。

建议阅读资料：[【美团技术博客】Java中9种常见的CMS GC问题分析与解决](#)



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zhongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

05-Spring

01、SpringMVC

面试官：今天要不来聊聊**SpringMVC**吧？

候选者：我先简单说下我对**SpringMVC**的理解哈

候选者：**SpringMVC**我觉得它是对**Servlet**的封装，屏蔽掉**Servlet**很多的细节

候选者：举几个例子

候选者：可能我们刚学**Servlet**的时候，要获取参数需要不断的getParameter

候选者：现在只要在**SpringMVC**方法定义对应的**JavaBean**，只要属性名与参数名一致，**SpringMVC**就可以帮我们实现「将参数封装到**JavaBean**」上了

候选者：又比如，以前使用**Servlet**「上传文件」，需要处理各种细节，写一大堆处理的逻辑（还得导入对应的jar）

候选者：现在一个在**SpringMVC**的方法上定义出**MultipartFile**接口，又可以屏蔽掉上传文件的细节了。

候选者：例子还有很多，我就不一一赘述了。

对**Servlet**进行封装，屏蔽很多好用的细节

面试官：既然你说**SpringMVC**是对**Servlet**的封装，你了解**SpringMVC**请求处理的流程吗？

候选者：嗯，当然了，我看过了源码。总体流程大概是这样的

候选者：1)首先有个统一处理请求的入口

候选者: 2): 随后根据请求路径找到对应的映射器

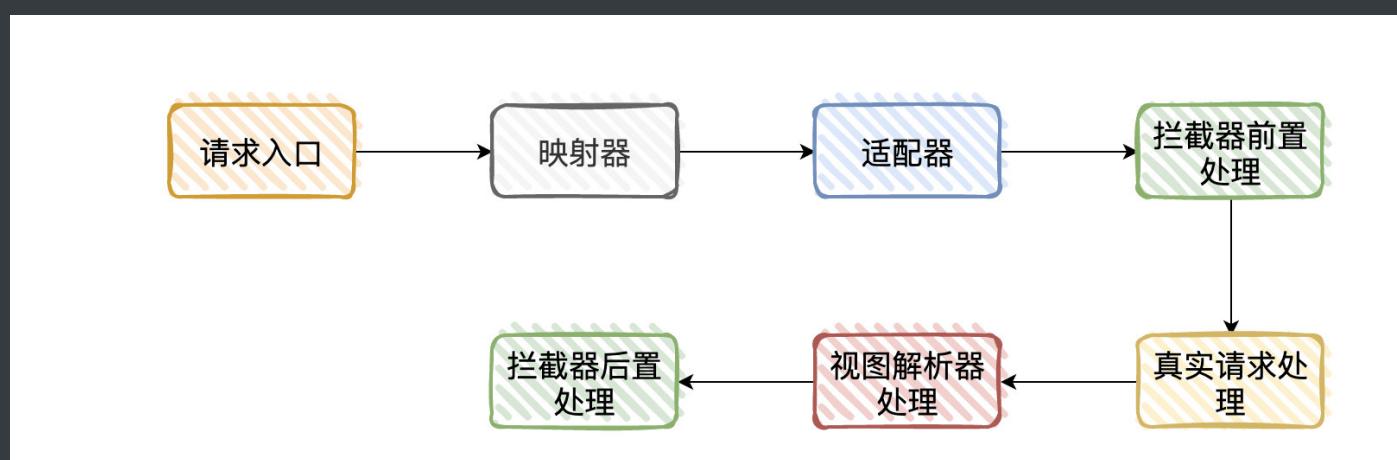
候选者: 3): 找到处理请求的适配器

候选者: 4): 拦截器前置处理

候选者: 5): 真实处理请求 (也就是调用真正的代码)

候选者: 6): 视图解析器处理

候选者: 7): 拦截器后置处理



面试官: 嗯，了解，可以再稍微深入点吗？

面试官: 毕竟这随便在网上找张 SpringMVC 流程图，就可以答出来了，看不出来你看过源码啊

候选者: 哦？那我就简单补充下细节吧。

候选者: 统一的处理入口，对应 SpringMVC 下的源码是在 DispatcherServlet 下实现的

候选者: 该对象在初始化就会把映射器、适配器、视图解析器、异常处理器、文件处理器等等给初始化掉

候选者: 至于会初始化哪些具体实例，看下 `DispatcherServlet.properties` 就知道了，都配置在那了

候选者: 所有的请求其实都会被 `doService` 方法处理，里边最主要就是调用 `doDispatch` 方法

候选者：通过doDispatch方法我们就可以看到整个SpringMVC处理的流程

DispatcherServlet

初始化很多对象（适配器、映射器、文件处理器等等）
初始哪些对象主要是看配置文件

候选者：查找映射器的时候实际就是找到「最佳匹配」的路径，具体方法实现我记得好像是在lookupHandlerMethod方法上

候选者：从源码可以看到「查找映射器」实际返回的是HandlerExecutionChain，里边有映射器Handler+拦截器List

候选者：前面提到的拦截器前置处理和后置处理就是用的HandlerExecutionChain中的拦截器List

映射器处理：会返回 映射器Handler+拦截器的List

候选者：获取得到HandlerExecutionChain后，就会去获取适配器，一般我们获取得到的就是RequestMappingHandlerAdapter

候选者：在代码里边可以看到的是，经常用到的@ResponseBody和@RequestBody的解析器

候选者：就会在初始化的时候加到参数解析器List中

候选者：得到适配器之后，就会执行拦截器前置处理

面试官：嗯

候选者: 拦截器前置处理执行完后，就会调用适配器对象实例的handle方法执行真正的代码逻辑处理

候选者: 核心的处理逻辑在invokeAndHandle方法中，会获取到请求的参数并调用，处理返回值

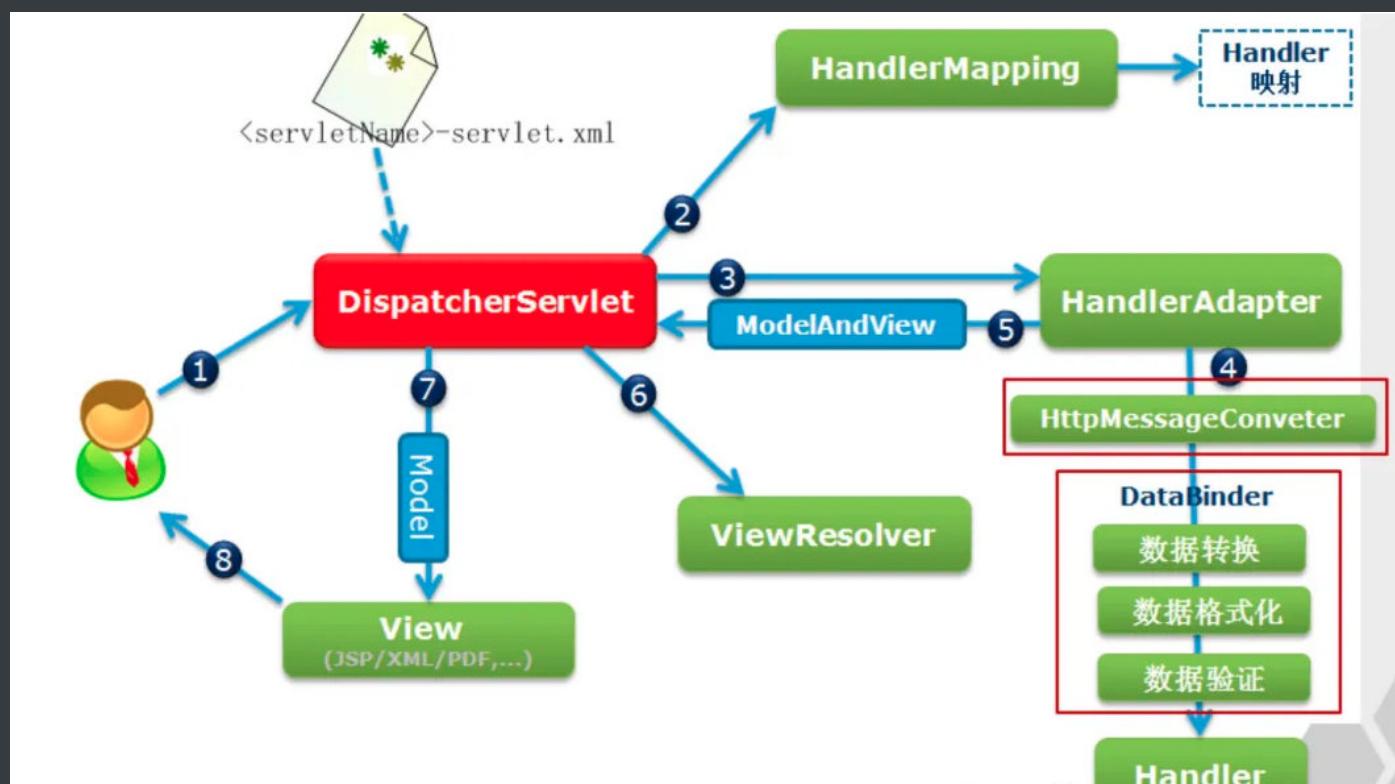
候选者: 参数的封装以及处理会被适配器的参数解析器进行处理，具体的处理逻辑取决于HttpMessageConverter的实例对象

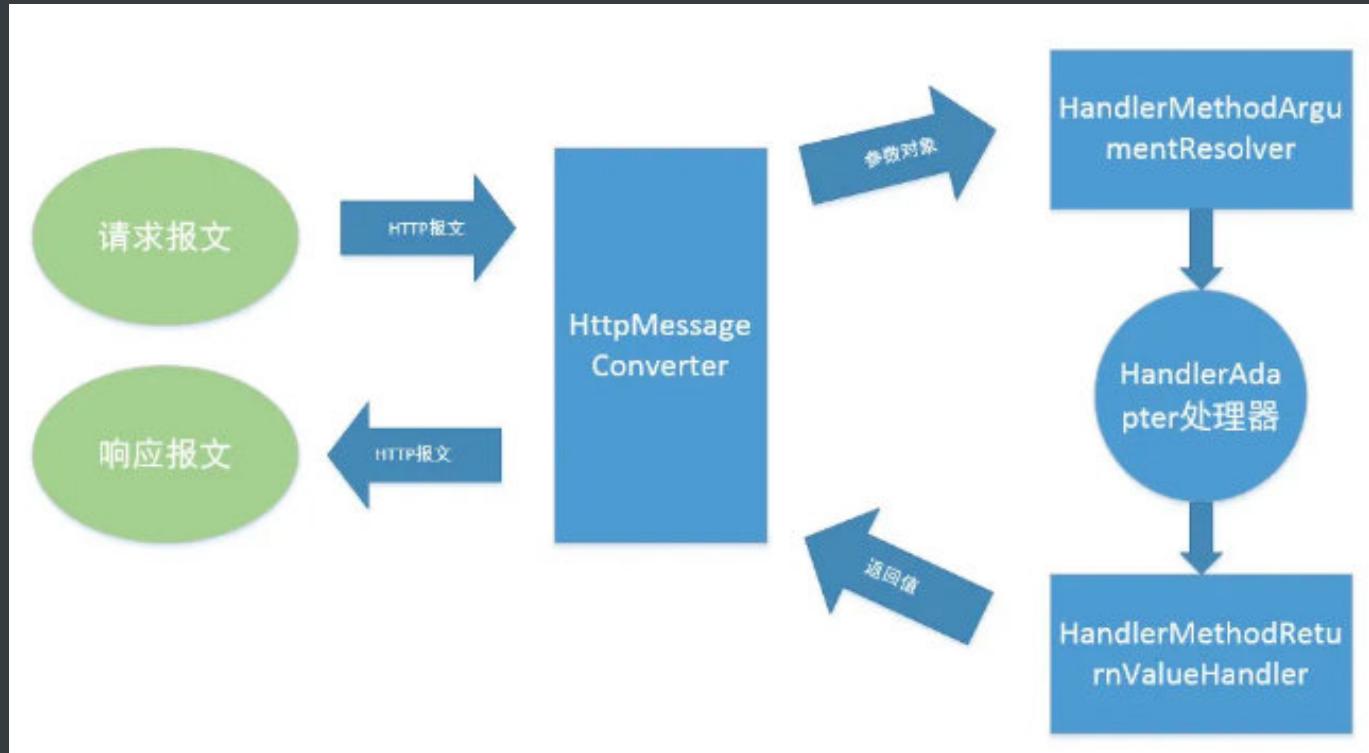
面试官: 嗯，了解了。要不你再压缩下关键的信息

候选者: DispatcherServlet (入口) ->DispatcherServlet.properties (会初始化的对象) ->HandlerMapping (映射器) ->HandlerExecutionChain(映射器+拦截器List) ->HttpRequestHandlerAdapter(适配器)->HttpMessageConverter(数据转换)

面试官: 最后来画张流程图吧？

候选者: 没问题 (上网找的比我画的还要好)





第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



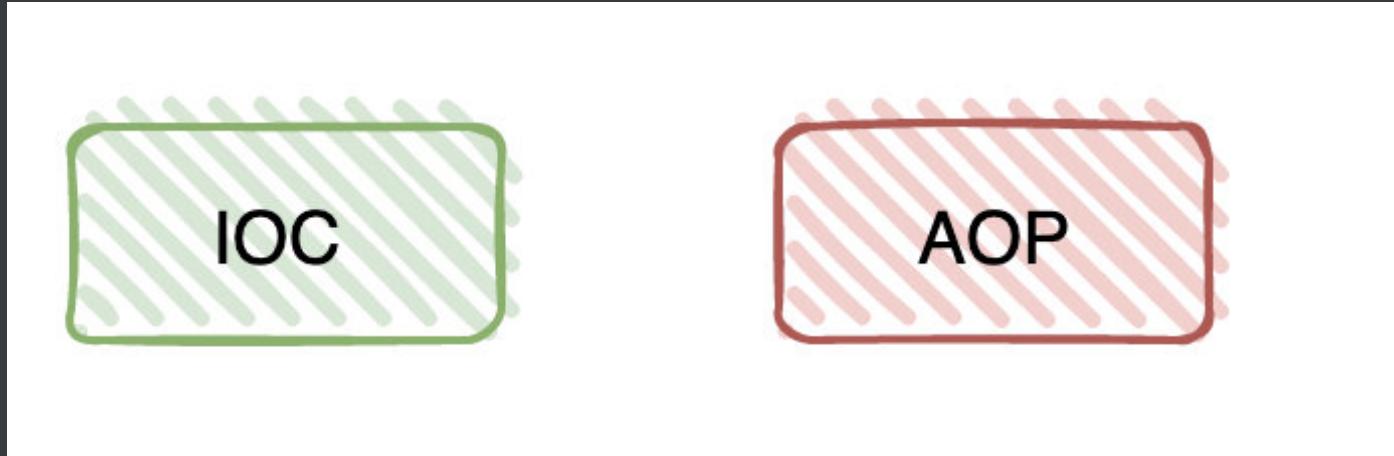
02、Spring基础

面试官：我看到你的简历写着熟悉Spring

面试官：要不你来讲讲Spring 的IOC和AOP你是怎么理解的呗？

候选者：嗯嗯， IOC和AOP是Spring非常核心的知识点

候选者：我就先来讲讲Spring IOC？



面试官：嗯

候选者：我个人理解下：Spring IOC 解决的是对象管理和对象依赖的问题。

候选者：本来是我们自己手动new出来的对象，现在则把对象交给Spring的IOC容器管理

候选者：IOC容器可以理解为一个对象工厂，我们都把该对象交给工厂，工厂管理这些对象的创建以及依赖关系

候选者：等我们需要用对象的时候，从工厂里边获取就好了

面试官：嗯，说起IOC，就可以在网上或书籍经常看到的两个概念

候选者：哦，你说的就是「控制反转」和「注入依赖」吧？

面试官：你怎么还抢答的咯...**那你顺便说说你对这两个概念的理解呗？**

候选者：我认为「控制反转」指的就是：把原有自己掌控的事交给别人去处理

候选者：它更多的是一种思想或者可以理解为设计模式

候选者：比如：本来由我们自己new出来的对象，现在交由IOC容器，把对象的控制权交给它方了

候选者：而「依赖注入」在我的理解下，它其实是「控制反转」的实现方式

候选者：对象无需自行创建或者管理它的依赖关系，依赖关系将被「自动注入」到需要它们的对象当中去

控制反转是一种思想，依赖注入是实现方式

面试官：嗯，那我想问问，**用Spring IOC有什么好处吗？**

面试官：或者换个问法：本来我可以new出来的对象，为什么我要交由**Spring IOC**容器管理呢？

候选者：主要的好处在于「将对象集中统一管理」并且「降低耦合度」

候选者：如果面试官理解了「工厂模式」，那就知道为什么我们不直接new对象

面试官：好家伙，不行，这答案我观众不满意！

候选者：要说理由的话，可以举很多例子，比如说：

候选者：我用Spring IOC 可以方便 单元测试、对象创建复杂、对象依赖复杂、单例等等的，什么都可以交给Spring IOC

候选者：理论上自己new出来的都可以解决上面的问题，Spring在各种场景组合下有可能不是最优解

候选者：但new出来的你要自己管理，可能你得自己写工厂，得实现一大套的东西才能满足需求

候选者：写着写着有可能还是Spring的那一套

候选者：但现在Spring现在已经帮你实现了啊！

候选者：如果项目里的对象都是就new下就完工了，没有多个实现类，那没事，不用Spring也没啥问题

候选者：并且Spring核心不仅仅IOC啊，除了把对象创建出来，还有一整套的Bean生命周期管理

候选者：比如说你要实现对象增强，AOP不就有了吗？不然你还得自己创建代理..

享受单例

单元测试

屏蔽对象创建

一套对Bean生命周期的扩展

面试官：好好好

面试官：但我看这届观众好像还是不太满意？

候选者：不，他们已经满意了。

面试官：那你继续来聊下Spring AOP呗？

候选者：Spring AOP 解决的是 非业务代码抽取的问题

候选者：AOP 底层的技术是动态代理，在Spring内实现依赖的是BeanPostProcessor

候选者：比如我们需要在方法上注入些「重复性」的非业务代码，就可以利用Spring AOP

候选者：所谓的「面向切面编程」在我理解下其实就是在方法前后增加非业务代码

AOP 底层是动态代理技术

面试官：那你在工作中实际用到过AOP去优化你的代码吗？

候选者：有的。当时我用AOP来对我们公司现有的监控客户端进行封装

候选者：一个系统离不开监控，监控基本的指标有QPS、RT、ERROR等等

候选者：对外暴露的监控客户端只能在代码里写对应的上报信息（灵活，但会与业务代码掺杂在一起）

候选者：于是我利用注解+AOP的方式封装了一把，只要方法/类上带有我自定义的注解

候选者：方法被调用时，就会上报AQS、RT等信息

候选者：实现了非业务代码与业务代码分离的效果（：

封装 监控客户端，不在代码里写硬代码

面试官：你们项目一般是怎么把对象交给IOC容器管理的？

面试官：换个问法：一般是怎么定义Bean的？

候选者：Spring提供了4种方式，分别是：

候选者：1):注解 2):XML 3):JavaConfig 4):基于Groovy DSL配置

候选者：一般项目我们用注解或XML比较多，少部分用JavaConfig

候选者：日常写业务代码一般用注解来定义各种对象，责任链这种一般配置在XML，「注解」解决不了的就用JavaConfig

候选者：总体而言，还是得看项目的代码风格吧（：

候选者：反正就是定义元数据，能给到Spring解析就好了



面试官：嗯，了解。

面试官：要不来聊聊你使用Spring的感受？

候选者：嗯嗯..

候选者: 当我还是初学Spring的时候，我觉得Spring很麻烦，需要有一大堆的配置信息才能跑起来

候选者: 光是搭建环境就需要耗费我好长的时间

候选者: 毕竟版本冲突，依赖冲突什么的就可能一个下午就过去了

候选者: 但毕竟一个系统环境只搭一次嘛，所以还好

候选者: (后来用上了SpringBoot这又更方便了)

配置地狱

面试官: �恩...

候选者: 话说回来，IOC和AOP在工作用的时候还是很爽的

候选者: 毕竟搞个注解什么的，配置下就可以把对象交给Spring管理了

候选者: 配合Spring的生态，@Transactional注解什么的，都好用得飞起

候选者: 不过，Spring给我们封装得太好了

候选者: 经常就会有奇奇怪怪的”bug“出现，也踩过很多的坑了

候选者: Bean经常没办法创建成功，导致项目启动失败

候选者: 对象的循环依赖问题...

候选者: 同一个接口，多个实现，识别不出我要创建哪个对象...

候选者: 为什么catch了异常，Spring事务为什么还会自动回滚

候选者: 等等等.....

在使用的时候也需要理解各种细节

候选者：总的来说，Spring给我们封装了一个很好的环境，实现对我们屏蔽了

候选者：但是如果理解不深的话，很有可能就会触发各种bug

面试官：了解



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zhongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

03、SpringBean生命周期

面试官：今天要不来聊聊Spring对Bean的生命周期管理？

候选者：嗯，没问题的。

候选者：很早之前我就看过源码，但Spring源码的实现类都太长了

候选者：我也记不得很清楚某些实现类的名字，要不我大概来说下流程？

面试官：没事，你开始吧

候选者：首先要知道的是

候选者：普通Java对象和Spring所管理的Bean实例化的过程是有些区别的

候选者：在普通Java环境下创建对象简要的步骤可以分为：

候选者：1):java源码被编译为class文件

候选者：2):等到类需要被初始化时（比如说new、反射等）

候选者：3):class文件被虚拟机通过类加载器加载到JVM

候选者：4):初始化对象供我们使用

候选者：简单来说，可以理解为它是用Class对象作为「模板」进而创建出具体的实例



Class作为类的模板

候选者：而Spring所管理的Bean不同的是，除了Class对象之外，还会使用BeanDefinition的实例来描述对象的信息

候选者：比如说，我们可以在Spring所管理的Bean有一系列的描述：@Scope、@Lazy、@DependsOn等等

候选者：可以理解为：Class只描述了类的信息，而BeanDefinition描述了对象的信息

面试官：嗯，这我大致了解你的意思了。

面试官：你就是想告诉我，**Spring有BeanDefinition来存储着我们日常给Spring Bean定义的元数据（@Scope、@Lazy、@DependsOn等等）**，对吧？

候选者：不愧是你

面试官：赶紧的，继续吧

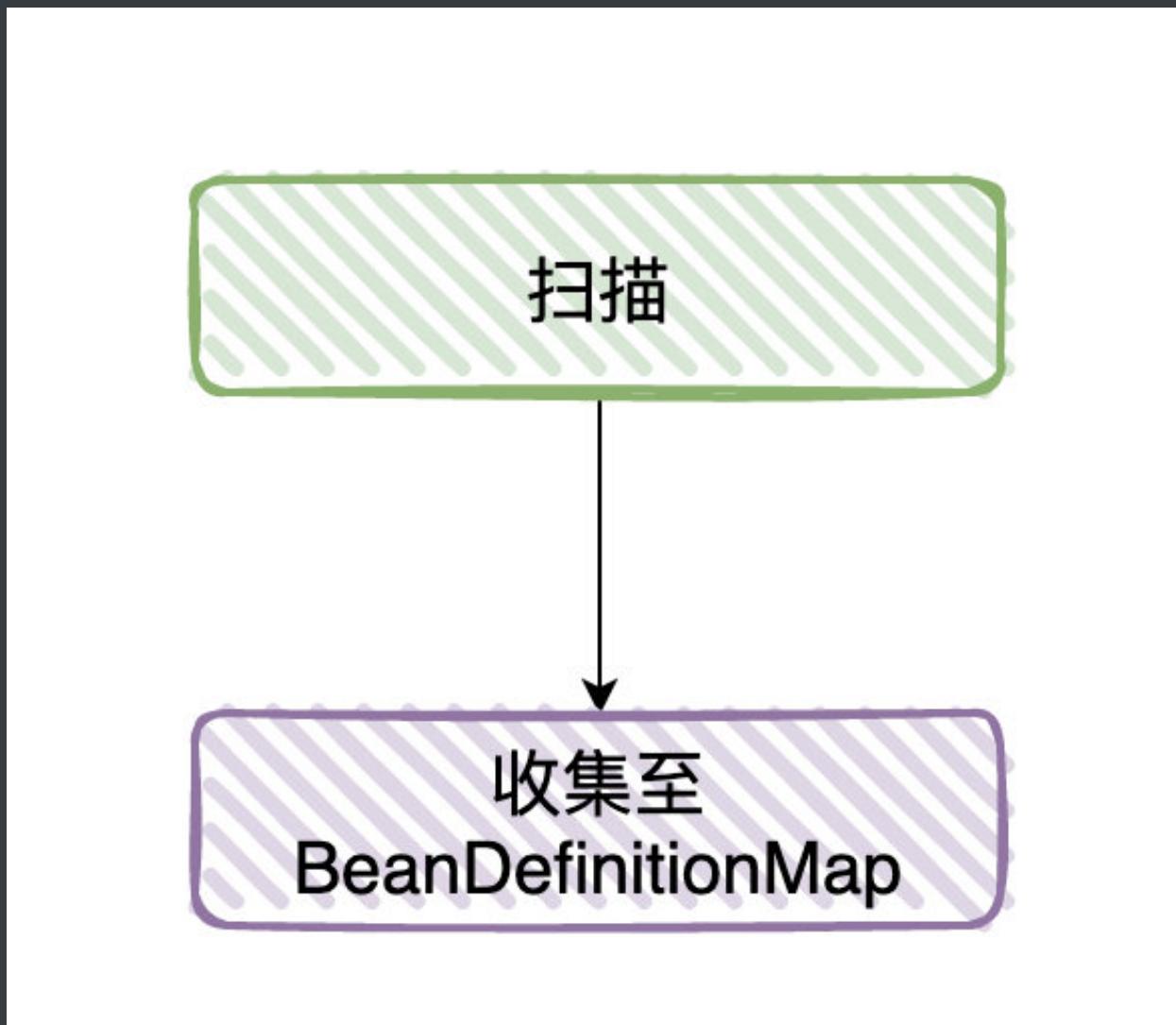
BeanDefinition 定义
SpringBean的类信息

候选者: Spring在启动的时候需要「扫描」在XML/注解/JavaConfig 中需要被Spring管理的Bean信息

候选者: 随后，会将这些信息封装成BeanDefinition，最后会把这些信息放到一个beanDefinitionMap中

候选者: 我记得这个Map的key应该是beanName，value则是BeanDefinition对象

候选者: 到这里其实就是把定义的元数据加载起来，目前真实对象还没实例化



候选者: 接着会遍历这个beanDefinitionMap，执行BeanFactoryPostProcessor这个Bean工厂后置处理器的逻辑

候选者: 比如说，我们平时定义的占位符信息，就是通过BeanFactoryPostProcessor的子类PropertyPlaceholderConfigurer进行注入进去

候选者: 当然了，这里我们也可以自定义BeanFactoryPostProcessor来对我们定义好的Bean元数据进行获取或者修改

候选者: 只是一般我们不会这样干，实际上也很少的使用场景

BeanFactoryPostProcessor 可对Bean元信息进行修改

面试官: 噢....

候选者: BeanFactoryPostProcessor后置处理器执行完了以后，就到了实例化对象啦

候选者: 在Spring里边是通过反射来实现的，一般情况下会通过反射选择合适的构造器来把对象实例化

候选者: 但这里把对象实例化，只是把对象给创建出来，而对象具体的属性是还没注入的。

候选者: 比如我的对象是UserService，而UserService对象依赖着SendService对象，这时候的SendService还是null的

候选者: 所以，下一步就是把对象的相关属性给注入 (:

```
graph LR; A[实例化] --> B[依赖注入]
```

实例化

依赖注入

候选者: 相关属性注入完之后，往下接着就是初始化的工作了

候选者: 首先判断该Bean是否实现了Aware相关的接口，如果存在则填充相关的资源

候选者: 比如我这边在项目用到的: 我希望通过代码程序的方式去获取指定的Spring Bean

候选者: 我们这边会抽取成一个工具类, 去实现ApplicationContextAware接口, 来获取ApplicationContext对象进而获取Spring Bean

是否实现了Aware接口 (用于对SpringBean的扩展)

候选者: Aware相关的接口处理完之后, 就会到BeanPostProcessor后置处理器啦

候选者: BeanPostProcessor后置处理器有两个方法, 一个是before, 一个是after (那肯定是before先执行、after后执行)

BeanPostProcessor AOP的关键 before after方法

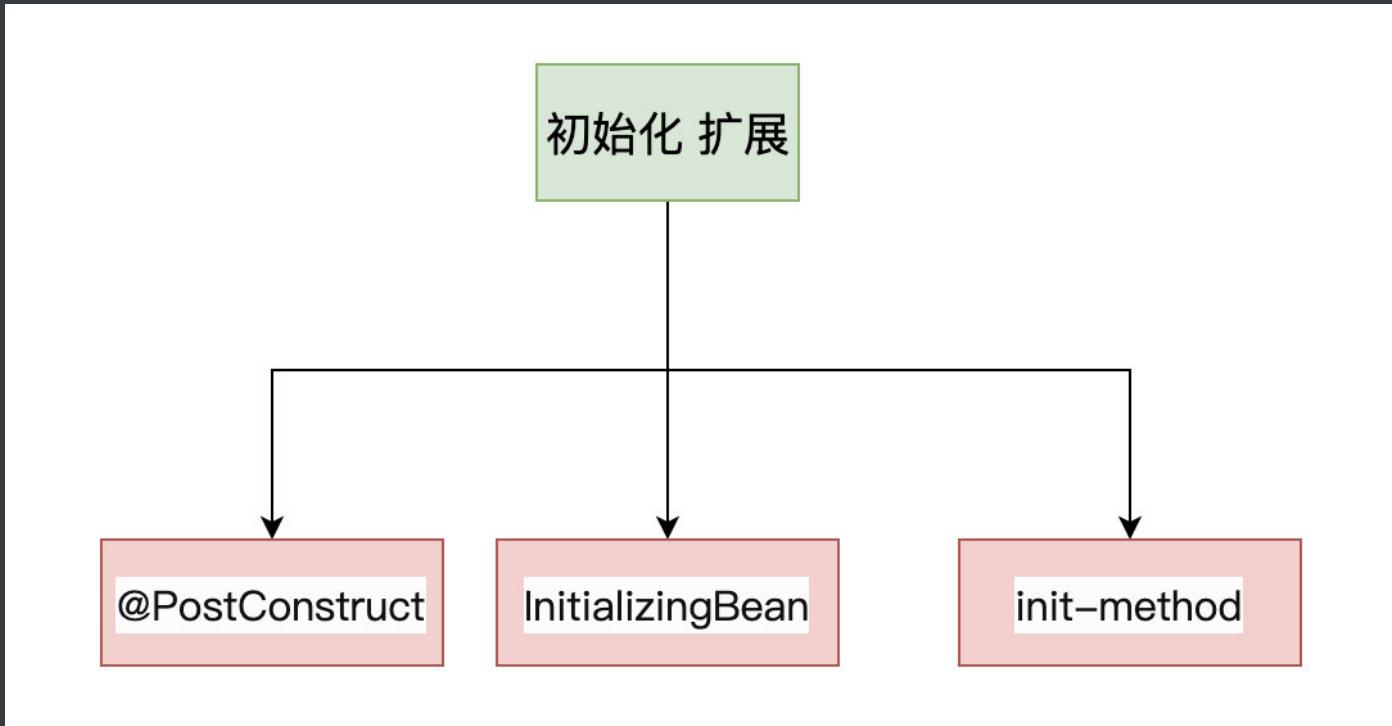
候选者: 这个BeanPostProcessor后置处理器是AOP实现的关键 (关键子类AnnotationAwareAspectJAutoProxyCreator)

候选者: 所以, 执行完Aware相关的接口就会执行BeanPostProcessor相关子类的before方法

候选者: BeanPostProcessor相关子类的before方法执行完, 则执行init相关的方法, 比如说@PostConstruct、实现了InitializingBean接口、定义的init-method方法

候选者: 当时我还去官网去看他们的被调用「执行顺序」分别是: @PostConstruct、实现了InitializingBean接口以及init-method方法

候选者: 这些都是Spring给我们的「扩展」, 像@PostConstruct我就经常用到



候选者: 比如说：对象实例化后，我要做些初始化的相关工作或者就启个线程去Kafka拉取数据

候选者: 等到init方法执行完之后，就会执行BeanPostProcessor的after方法

候选者: 基本重要的流程已经走完了，我们就可以获取到对象去使用了

候选者: 销毁的时候就看有没有配置相关的destroy方法，执行就完事了

面试官: �恩，了解，但我的观众好像不太满意，总感觉少了些什么。

面试官: 你看过Spring是怎么解决循环依赖的吗？

面试官: 如果现在有个A对象，它的属性是B对象，而B对象的属性也是A对象

面试官: 说白了就是A依赖B，而B又依赖A，Spring是怎么做的？

候选者: 嗯，这块我也是看过的，其实也是在Spring的生命周期里面嘛

候选者: 从上面我们可以知道，对象属性的注入在对象实例化之后的嘛。

候选者: 它的大致过程是这样的：

候选者: 首先A对象实例化，然后对属性进行注入，发现依赖B对象

候选者: B对象此时还没创建出来，所以转头去实例化B对象

候选者: B对象实例化之后，发现需要依赖A对象，那A对象已经实例化了嘛，所以B对象最终能完成创建

候选者: B对象返回到A对象的属性注入的方法上，A对象最终完成创建

候选者: 上面就是大致的过程；

面试官: 听起来你还会原理哦？

候选者: Absolutely

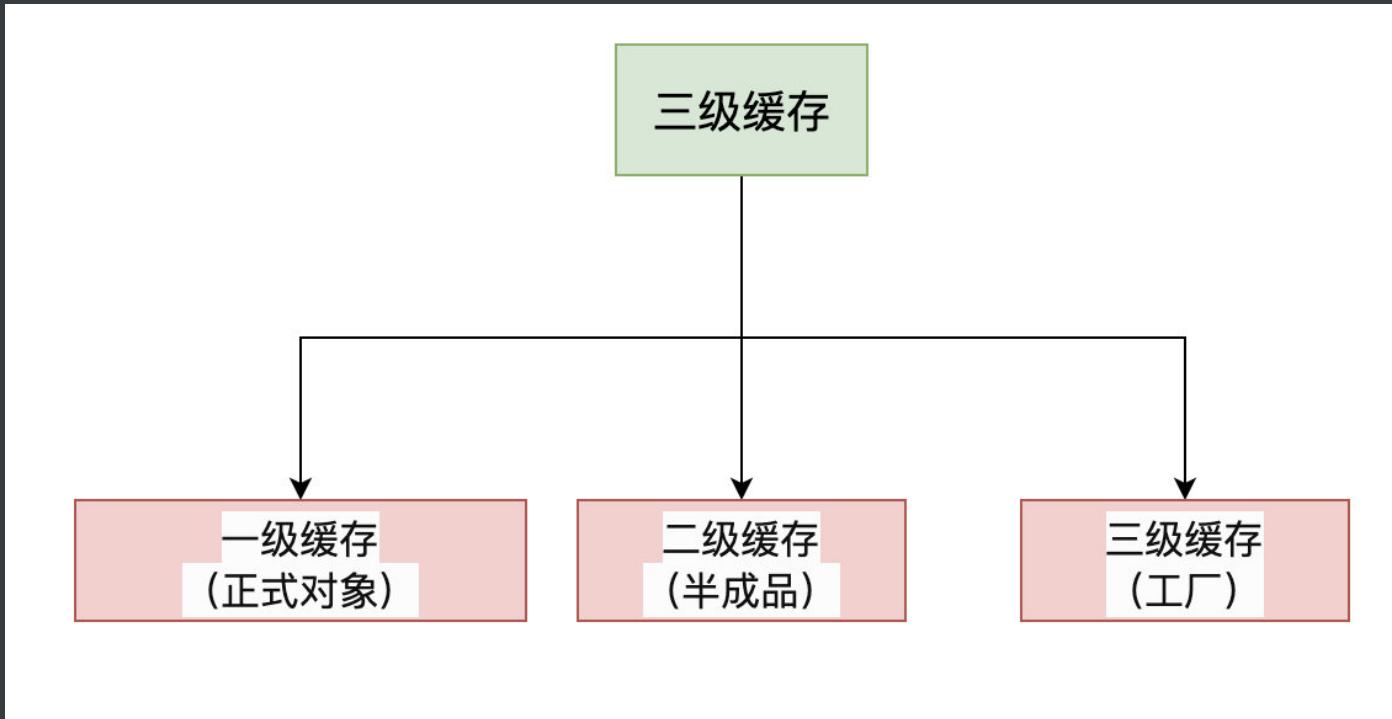
候选者: 至于原理，其实就是用到了三级的缓存

候选者: 所谓的三级缓存其实就是一个Map...首先明确一定，我对这里的三级缓存定义是这样的：

候选者: singletonObjects (一级，日常实际获取Bean的地方)；

候选者: earlySingletonObjects (二级，还没进行属性注入，由三级缓存放进来)；

候选者: singletonFactories (三级，Value是一个对象工厂)；



候选者：再回到刚才讲述的过程中，A对象实例化之后，属性注入之前，其实会把A对象放入三级缓存中

候选者：key是BeanName，Value是ObjectFactory

候选者：等到A对象属性注入时，发现依赖B，又去实例化B时

候选者：B属性注入需要去获取A对象，这里就是从三级缓存里拿出ObjectFactory，从ObjectFactory得到对应的Bean（就是对象A）

候选者：把三级缓存的A记录给干掉，然后放到二级缓存中

候选者：显然，二级缓存存储的key是BeanName，value就是Bean（这里的Bean还没做完属性注入相关的工作）

候选者：等到完全初始化之后，就会把二级缓存给remove掉，塞到一级缓存中

候选者：我们自己去getBean的时候，实际上拿到的是一级缓存的

候选者：大致的过程就是这样

面试官：那我想问一下，为什么是三级缓存？

候选者: 首先从第三级缓存说起 (就是key是BeanName, Value为ObjectFactory)

候选者: 我们的对象是单例的, 有可能A对象依赖的B对象是有AOP的 (B对象需要代理)

候选者: 假设没有第三级缓存, 只有第二级缓存 (Value存对象, 而不是工厂对象)

候选者: 那如果有AOP的情况下, 岂不是在存入第二级缓存之前都需要先去做AOP代理? 这不合适嘛

候选者: 这里肯定是需要考虑代理的情况的, 比如A对象是一个被AOP增强的对象, B依赖A时, 得到的A肯定是代理对象的

候选者: 所以, 三级缓存的Value是ObjectFactory, 可以从里边拿到代理对象

候选者: 而二级缓存存在的必要就是为了性能, 从三级缓存的工厂里创建出对象, 再扔到二级缓存 (这样就不用每次都要从工厂里拿)

候选者: 应该很好懂吧?

第三级缓存 考虑 代理
第二级缓存 考虑 性能

面试官: 确实 (:

候选者: 我稍微总结一下今天的内容吧

候选者: 怕你的观众说不满意, 那我就没有赞了, 没有赞我就很难受

候选者: 首先是Spring Bean的生命周期过程, Spring使用BeanDefinition来装载着我们给Bean定义的元数据

候选者: 实例化Bean的时候实际上就是遍历BeanDefinitionMap

候选者: Spring的Bean实例化和属性赋值是分开两步来做的

候选者: 在Spring Bean的生命周期, Spring预留了很多的hook给我们去扩展

候选者: 1) : Bean实例化之前有BeanFactoryPostProcessor

候选者: 2) : Bean实例化之后, 初始化时, 有相关的Aware接口供我们去拿到Context相关信息

候选者: 3) : 环绕着初始化阶段, 有BeanPostProcessor (AOP的关键)

候选者: 4) : 在初始化阶段, 有各种的init方法供我们去自定义

候选者: 而循环依赖的解决主要通过三级的缓存

候选者: 在实例化后, 会把自己扔到三级缓存 (此时的key是BeanName, Value是ObjectFactory)

候选者: 在注入属性时, 发现需要依赖B, 也会走B的实例化过程, B属性注入依赖A, 从三级缓存找到A

候选者: 删掉三级缓存, 放到二级缓存

面试官: 嗯, 你要不后面放点关键的源码吧

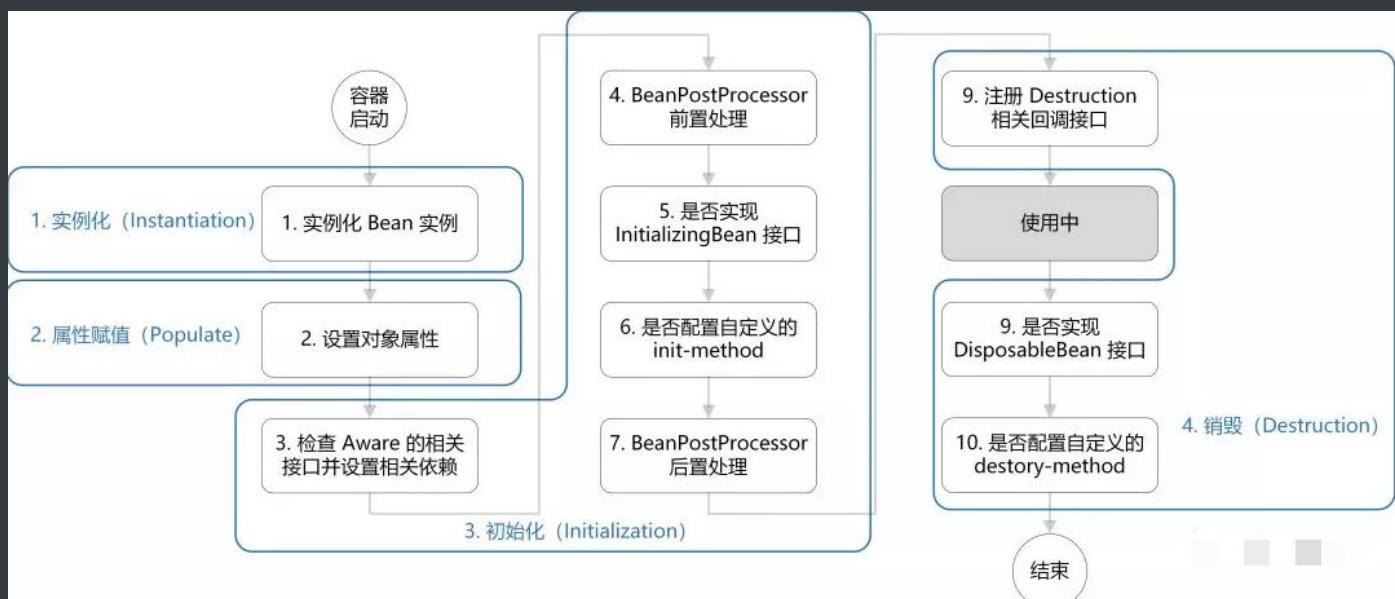
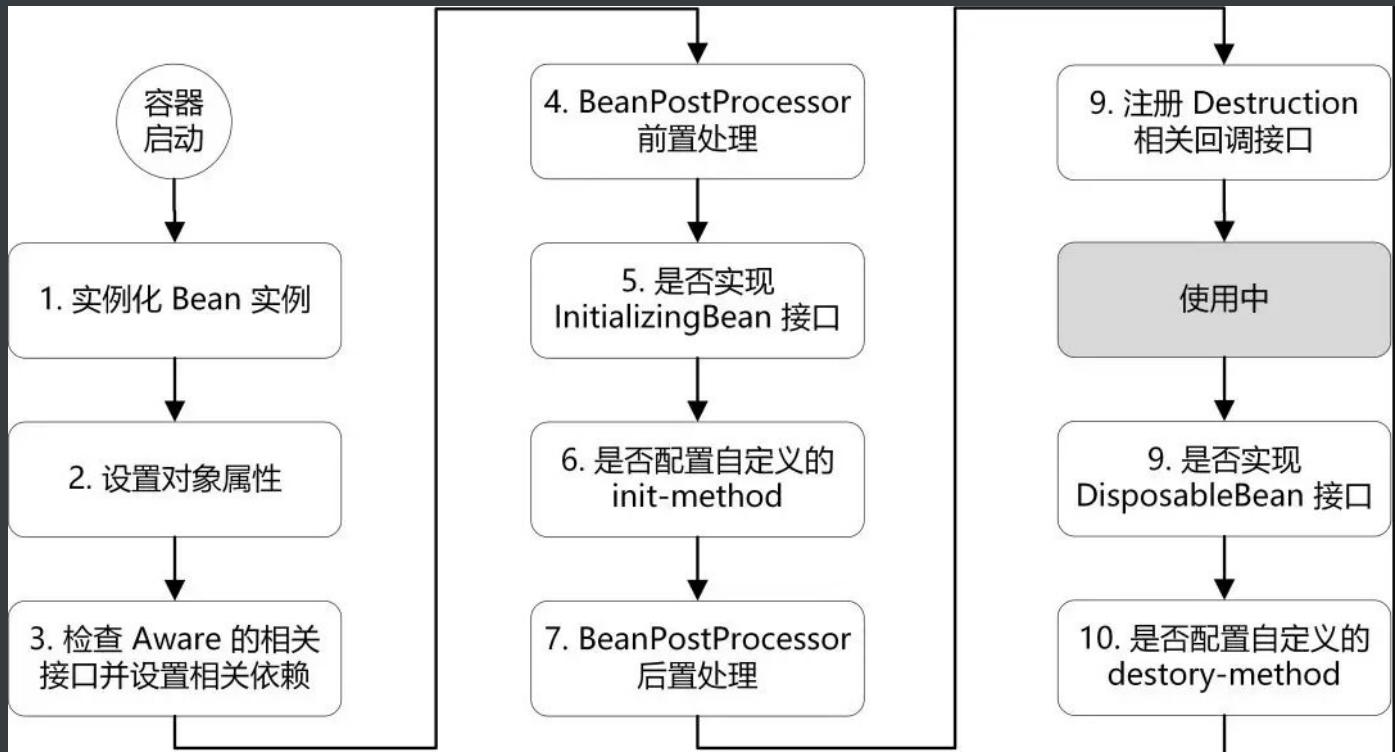
候选者: 这你倒是提醒我了, 确实有必要

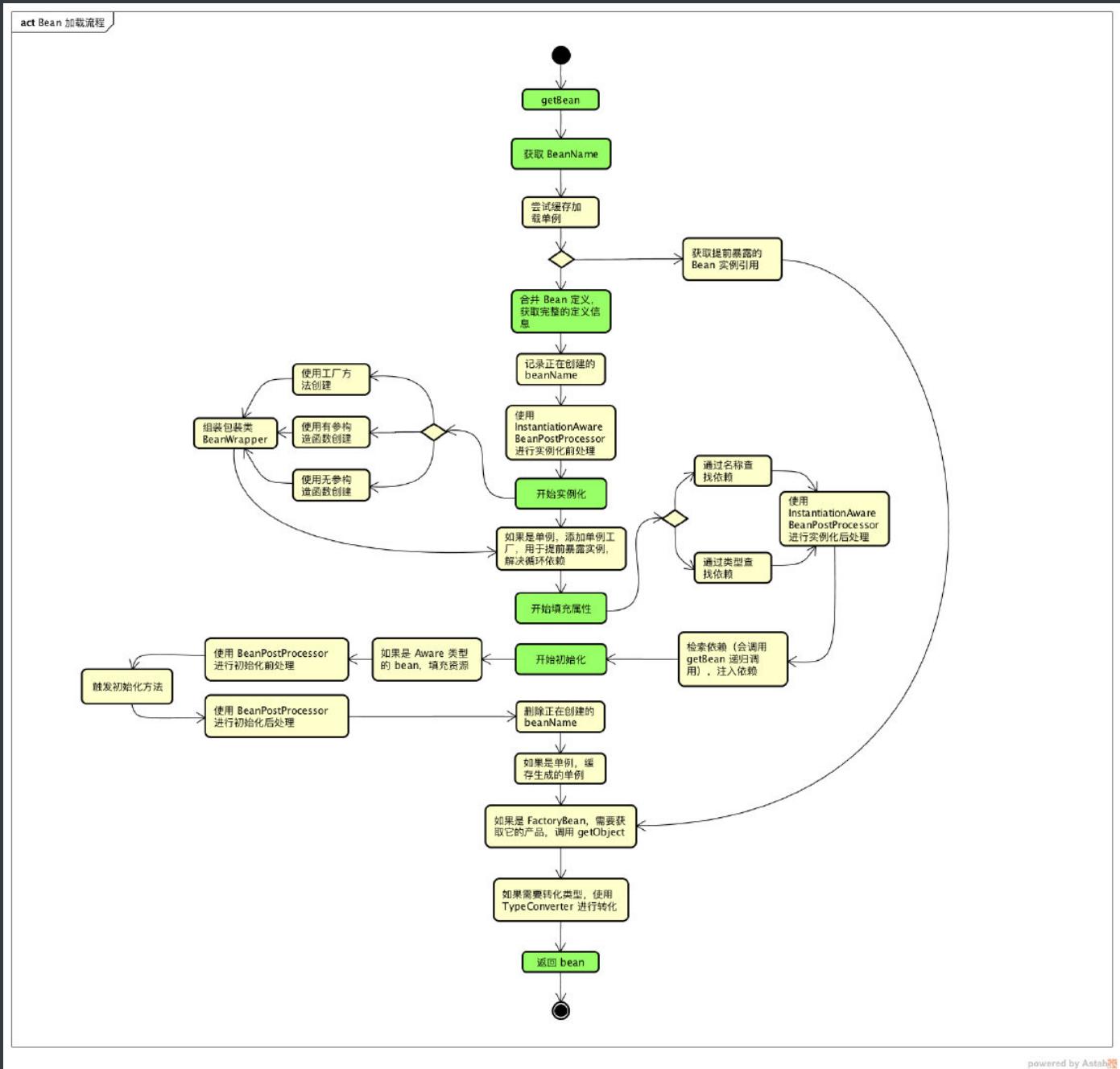
面试官: 这要是能听懂, 是真的看过源码才行 (: 还好我看过

关键源码方法 (强烈建议自己去撸一遍)

- `org.springframework.context.support.AbstractApplicationContext#refresh` (入口)
- `org.springframework.context.support.AbstractApplicationContext#finishBeanFactoryInitialization` (初始化单例对象入口)
- `org.springframework.beans.factory.config.ConfigurableListableBeanFactory#preInstantiateSingletons` (初始化单例对象入口)

- `org.springframework.beans.factory.support.AbstractBeanFactory#getBean(java.lang.String)` (万恶之源, 获取并创建Bean的入口)
- `org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean` (实际的获取并创建Bean的实现)
- `org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton(java.lang.String)` (从缓存中尝试获取)
- `org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBean(java.lang.String, org.springframework.beans.factory.support.RootBeanDefinition, java.lang.Object[])` (实例化Bean)
- `org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#doCreateBean` (实例化Bean具体实现)
- `org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBeanInstance` (具体实例化过程)
- `org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#addSingletonFactory` (将实例化后的Bean添加到三级缓存)
- `org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#populateBean` (实例化后属性注入)
- `org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#initializeBean(java.lang.String, java.lang.Object, org.springframework.beans.factory.support.RootBeanDefinition)` (初始化入口)





powered by Astah



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



06-Redis

01、Redis基础

面试官：今天要不来聊聊Redis吧？

候选者：好

候选者：我个人是这样理解的：无论Redis也好、MySQL也好、HDFS也好、HBase也好，他们都是存储数据的地方

候选者：因为它们的设计理念的不同，我们会根据不同的应用场景使用不同的存储

候选者：像Redis一般我们会把它用作于缓存

候选者：当然啦，日常有的应用场景比较简单，用个HashMap也能解决很多的问题了，没必要上Redis

候选者: 这就好比，有的单机限流可能应对某些场景就够用了，也没必要说一定要上分布式限流把系统搞得复杂

不同的业务场景，使用不同的存储

面试官: 你在项目里有用到Redis吗？怎么用的？

候选者: Redis肯定是用到的，我负责的项目几乎都会有Redis的踪影

候选者: 我举几个我这边项目用的案例呗？

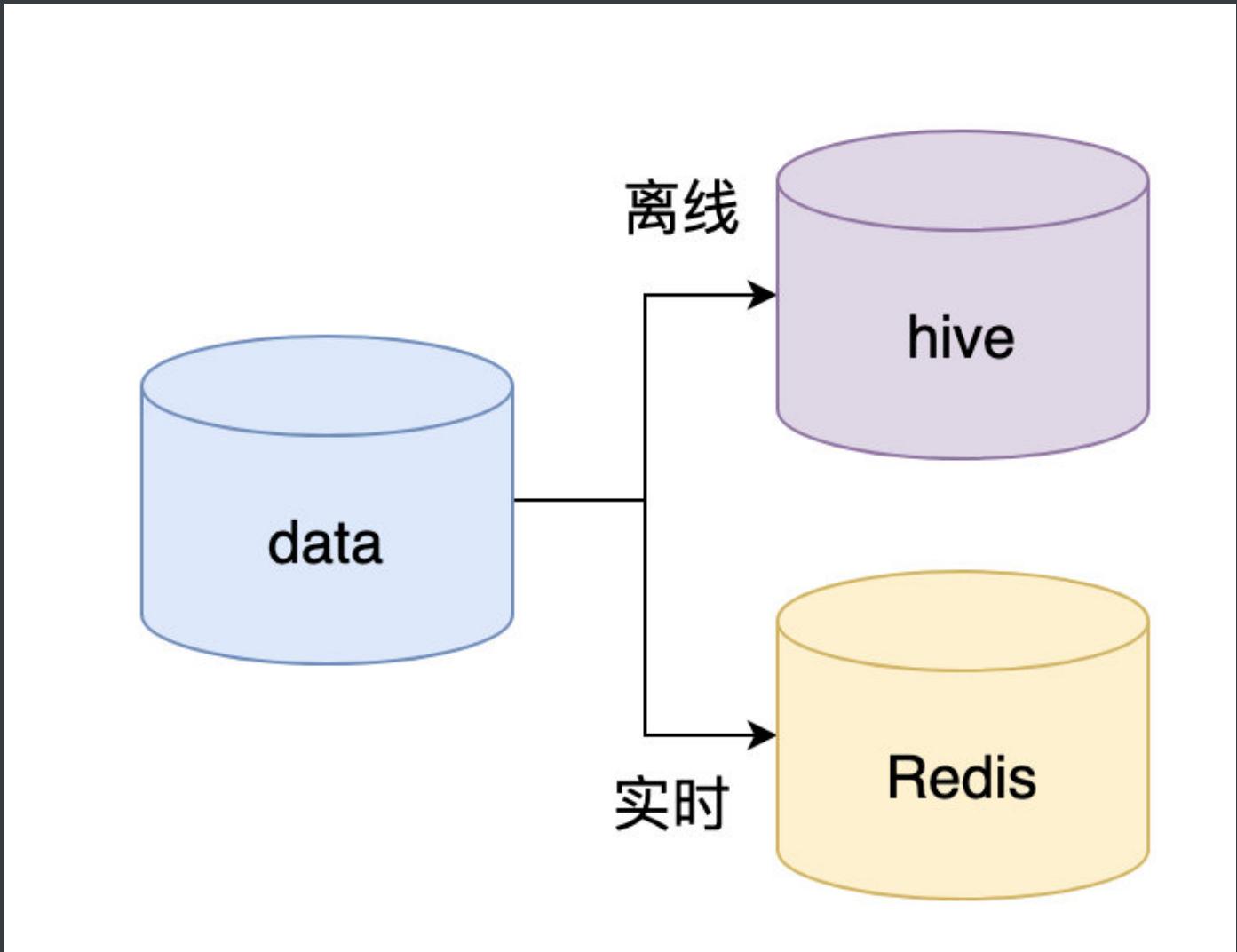
面试官: 嗯

候选者: 我这边负责消息管理平台，简单来说就是发消息的

候选者: 那发完消息肯定我们是得知道消息有没有下发成功的，是吧？

候选者: 于是我们系统有一套完整的链路追踪体系

候选者: 其中实时的数据我们就用Redis来进行存储，有实时肯定就会有离线的嘛（离线的数据我们是存储到Hive的）



候选者：对消息进行实时链路追踪，我这边就用了Redis好几种的数据结构

候选者：分别有Set、List和Hash

面试官：嗯....

候选者：我再稍微铺垫下链路追踪的背景吧~

候选者：要在消息管理平台发消息，首先得在后台新建一个「模板」，有模板自然会有一个模板ID

候选者：对模板ID进行扩展，比如说加上日期和固定的业务参数，形成的ID可以唯一标识某个模板的下发链路

候选者：在系统上，我这边叫它为UMPID

候选者：在发送入口处会对所有需要下发的消息打上UMPID，然后在关键链路上打上对应的点位

业务标识ID进行链路追踪

面试官：嗯，你继续吧

候选者：接下来的工作就是清洗出统一的模型，然后根据不同维度进行处理啦。比如说：

候选者：我要看某一天下发的所有模板有哪些，那只要我把清洗出来后数据的，将对应UMPID扔到了Set就好了

候选者：我要看某一个模板的消息下发的整体链路情况，那我以UMPID为Key，Value是Hash结构，Key是state，Value则是人数

候选者：这里的state我们在下发的过程中打的关键点位，比如接收到消息打个51，消息被去重了打个61，消息成功下发了打个81...

自定义一套打点体系，关键链路打日志

候选者：以UMPID为Key，Hash结构的Key（State）进行不断的累加，就可以实现某一个模板的消息下发的整体链路情况

候选者：我要看某个用户当天下发的消息有哪些，以及这些消息的整体链路是如何。

候选者：这边我用的是List结构，Key是userId，Value则是UMPID+state(关键点位)+processTime（处理时间）

面试官：嗯....

候选者：简单来说，就是通过Redis丰富的数据结构来实现对下发消息多个维度的统计

候选者：不同的应用场景选择不同的数据结构，再等到透出做处理的时候，就变得十分简单了

候选者：消息下发过程中去重或者一般正常的场景就直接Key-Value就能符合需求了

候选者：像bitmap、hyperloglogs、sortset、steam等等这些数据结构在我所负责的项目用得是真不多

候选者：要是我有机会去到贵公司，贵公司有相关的应用场景，我相信我也很快就能掌握

依赖Redis丰富数据结构，进行多维度分析

候选者：这些数据结构底层都由对应的object来支撑着，object记录对应的「编码」

候选者：其实就是会根据key-value存储的数量或者长度来使用选择不同的底层数据结构实现

候选者：比如说：ziplist压缩列表这个底层数据结构有可能上层的实现是list、hash和sortset

候选者：Hash结构的底层数据结构可能是hash和ziplist

候选者：在节省内存和性能的考量之中切换

候选者：Redis还是有点屌的啊。

每种数据结构下底层可能有几套实现 在性能与内存之间traff-off

面试官：就你上面那个实时链路场景，可以用其他的存储替代吗？

候选者：嗯，理论上是可以的（或许可以尝试用HBase），但总体来说没这么好吧

候选者：因为Redis拥有丰富的数据结构，在透出的时候，处理会非常的方便。

候选者：如果不用Redis的话，还得做很多解析的工作

候选者：并且，我那场景的并发还是相当大的（就一条消息发送，可能就产生10条记录）

候选者：监控峰值命令处理数会去到20k+QPS，当然了，这场景我肯定用了Pipeline的（不然处理会慢很多）

候选者：综合上面并发量和实时性以及数据结构，用Redis是一个比较好的选择

丰富数据结构

缓存快

拥有过期时间

...

面试官：嗯....你觉得为什么Redis可以这么快？

候选者：首先，它是纯内存操作，内存本身就很快

候选者：其次，它是单线程的，Redis服务器核心是基于非阻塞的IO多路复用机制，单线程避免了多线程的频繁上下文切换问题

候选者：至于这个单线程，其实官网也有过说明（：表示使用Redis往往的瓶颈在于内与网络，而不在于CPU

面试官：了解。



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



02、Redis持久化

面试官：今天要不来聊聊Redis的持久化机制吧？

候选者：嗯，没问题的

候选者：在上一次面试已经说过了Redis是基于内存的

候选者：假设我们不做任何操作，只要Redis服务器重启（或者中途故障挂掉了），那内存的数据就会没掉

候选者：我们作为使用方，肯定是不想Redis里头的数据会丢掉

候选者：所以Redis提供了持久化机制给我们用，分别是RDB和AOF

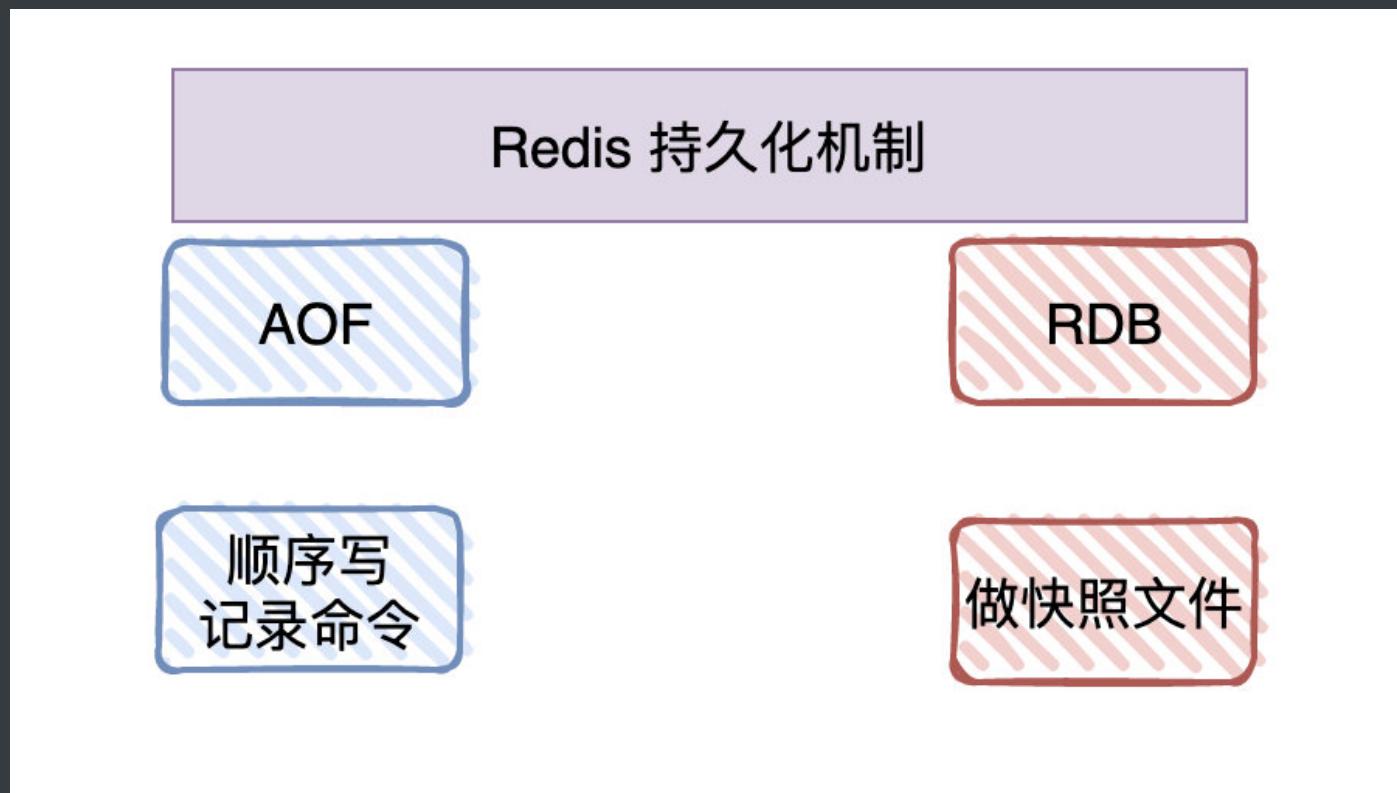


候选者：RDB指的就是：根据我们自己配置的时间或者手动去执行BGSAVE或SAVE命令，Redis就会去生成RDB文件

候选者：这个RDB文件实际上就是一个经过压缩的二进制文件，Redis可以通过这个文件在启动的时候来还原我们的数据

候选者：而AOF则是把Redis服务器接收到的所有写命令都记录到日志中

候选者：Redis重跑一遍这个记录下的日志文件，就相当于还原了数据



面试官：那我就想问了，你上次不是说Redis是单线程吗

面试官：那比如你说的RDB，它会执行SAVE或BESAVE命令，生成文件

面试官：那不是非常耗时的吗，那如果只有一个线程处理，那其他的请求不就得等了？

候选者：嗯，没错，Redis是单线程的。

候选者：以RDB持久化的过程为例，假设我们在配置上是定时去执行RDB存储

候选者：Redis有自己的一套事件处理机制，主要处理文件事件（命令请求和应答等等）和时间事件（RDB定时持久化、清理过期的Key等的）

候选者：所以，定时的RDB实际上就是一个时间事件

候选者：线程不停地轮询就绪的事件，发现RDB的事件可执行时，则调用BGSAVE命令

候选者：而BGSAVE命令实际上会fork出一个子进程来进行完成持久化（生成RDB文件）



候选者：在fork的过程中，父进程(主线程)肯定是阻塞的。

候选者：但fork完之后，是fork出来的子进程去完成持久化。处理请求的进程该干嘛的就干嘛

候选者：所以说啊，Redis是单线程，理解是没错的，但没说人家不能fork进程来处理事情呀，对不对

候选者：还有就是，其实Redis在较新的版本中，有些地方都使用了多线程来进行处理

候选者：比如说，一些删除的操作（UNLINK、FLUSHALL ASYNC等等）还有Redis 6.x之后对网络数据的解析都用了多线程处理了。

候选者：只不过，核心的处理命令请求和响应还是单线程。

核心命令处理和响应还是单线程

面试官：那AOF呢？AOF不是也要写文件吗？难道也是fork了个子进程去做的？

候选者：emm，不是的。AOF是在命令执行完之后，把命令写在buffer缓冲区的（直接追加写）

候选者: 那想要持久化，肯定得存盘嘛。Redis提供了几种策略供我们选择什么时候把缓冲区的数据写到磁盘

候选者: 我记得有：每秒一次/每条命令都执行/从不存盘；一般我们会选每秒一次

候选者: Redis会启一个线程去刷盘，也不是用主线程去干的

面试官: 那如果把执行过的命令都存起来

面试官: 等启动的时候是可以再把这些写命令再执行一遍，达到恢复数据的效果

面试官: 这样会有什么样的问题吗？

候选者: 嗯，问题就是，如果这些写入磁盘的「命令集合」不做任何处理，那该「命令集合」就会一直膨胀

候选者: 其实就是该文件会变得非常大

候选者: Redis当然也考虑了这一点，它会fork个子进程会对「原始」命令集合进行重写

候选者: 说白了就是会压缩，压缩完了之后只要替换原始文件就好了

AOF刷盘是后台线程异步的，文件重写是fork子进程

面试官: 那我又想问了，既然它是fork一个进程来对AOF进行重写的

面试官: 前面你也提到了再fork时，主进程是阻塞的，但fork后，主进程会继续接收命令

面试官: 你是说重写完（压缩）会进行文件覆盖

面试官: 那这样不会丢数据吗？毕竟主进程在fork之后是一直会接收命令的

候选者: 哦，我明白你的意思了。

候选者: 其实做法很简单啊，在fork子进程之后，把新接收到命令再写到另一个缓冲区不就好了吗

面试官: 可以

面试官: 那AOF和RDB用哪一个呢？

候选者: 主要是看业务场景吧，我们这边是基于Redis自研了一套key-value存储

面试官: 自研的？你们的Redis架构是什么？

候选者: 别别别，当我说。就是开源的，开源的。我们回到RDB和AOF上吧。

候选者: 在新增namespace（实例）的时候也会让你选择对应的使用场景

候选者: 就是会让你通过不同的应用场景进行配置选择

候选者: 比如说，业务上是允许重启时部分数据丢失的，那RDB就够用了（：

候选者: RDB在启动的时候恢复数据会比AOF快很多

候选者: 在Redis4.0以后也支持了AOF和RDB混合

AOF 和RDB混合

候选者: 在官网是不建议仅仅只使用AOF的，如果对数据丢失容忍度是有要求的，建议是开启AOF+RDB一起用

候选者: 总的来说，不同的场景使用不同的持久化策略吧

面试官：了解

面试官：顺便我想问下，假如Redis的内存满了，但业务还在写数据，会怎么样？

候选者：嗯，这个问题我也遇到过

候选者：一般来说，我们会淘汰那些「不活跃」的数据，然后把新的数据写进去

候选者：更多情况下，还是做好对应的监控和容量的考量吧

候选者：等容量达到阈值的时候，及时发现和扩容

面试官：你这懂得有点多啊

本文总结：

- **Redis持久化机制**: RDB和AOF
- **RDB持久化**: 定时任务，BGSAVE命令 fork一个子进程生成RDB文件（二进制）
- **AOF持久化**: 根据配置将写命令存储至日志文件中，顺序写&&异步刷盘(子线程)，重写AOF文件也是需要 fork 子进程。Redis4.0之后支持混合持久化，用什么持久化机制看业务场景



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zhongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

03、Redis主从架构

面试官：要不你来讲讲你最近在看的点呗？可以拉出来一起讨论下(今天我也不知道要问什么)

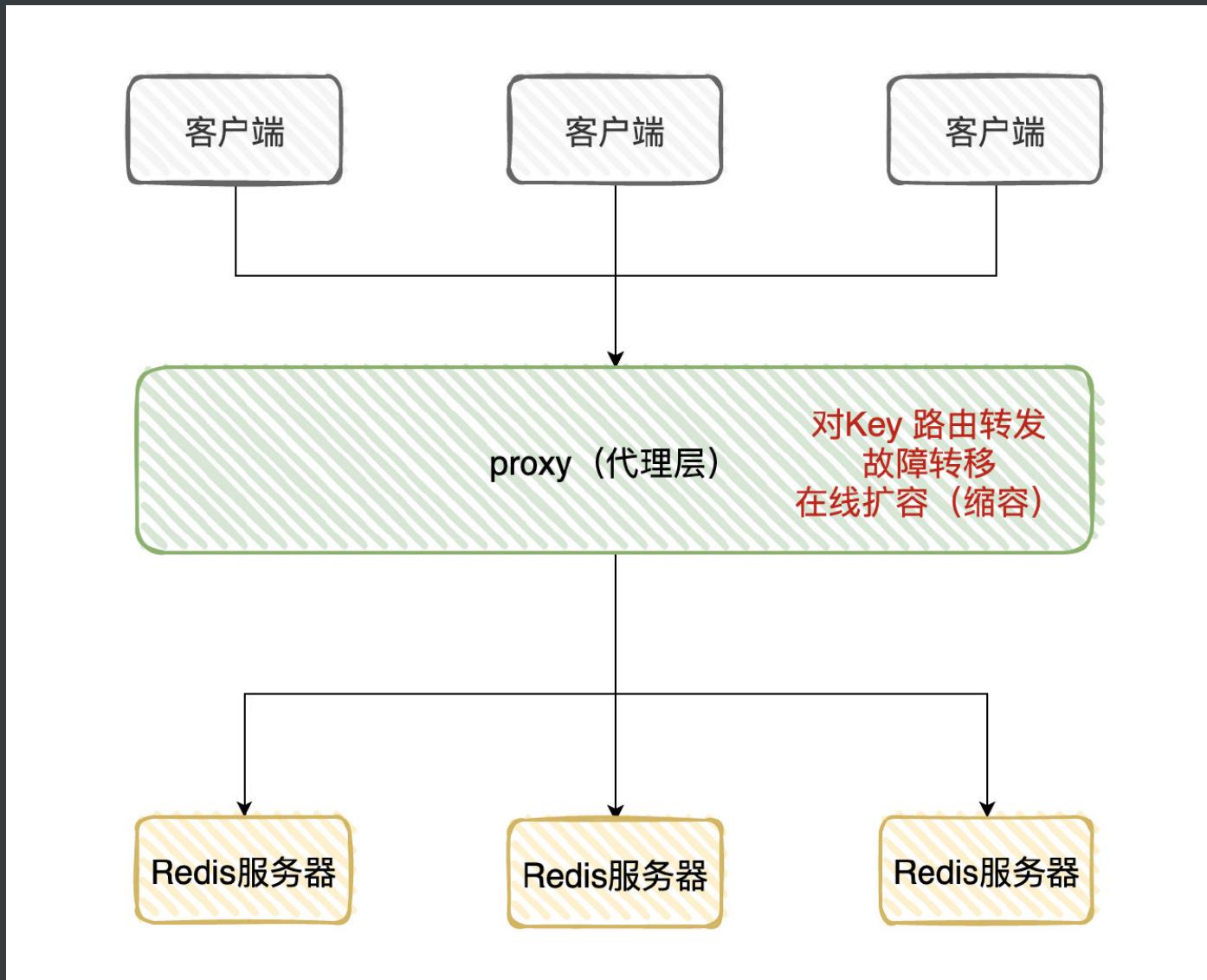
候选者：最近在看「Redis」相关的内容

面试官：嗯，我记得已经问过Redis的基础和持久化了

面试官：要不你来讲讲你公司的Redis是什么架构的咯？

候选者：我前公司的Redis架构是「分片集群」，使用的是「Proxy」层来对Key进行分流到不同的Redis服务器上

候选者：支持动态扩容、故障恢复等等...



面试官：那你来聊下Proxy层的架构和基本实现原理？

候选者：抱歉，这块由中间件团队负责，具体我也没仔细看过

候选者：...

面试官：....

候选者：不过，我可以给你讲讲现有常见开源的Redis架构（：

面试官：那只能这样了，好吧，你开始吧

候选者：那我从基础讲起吧？

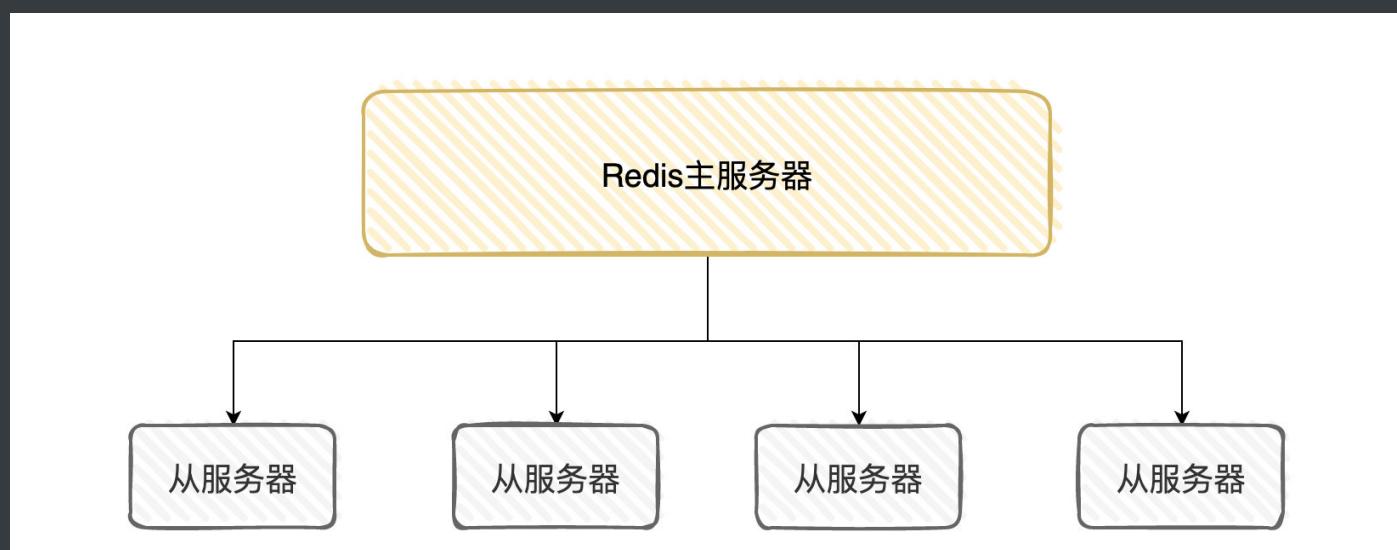
候选者: 在之前提到了Redis有持久化机制，即便Redis重启了，可以依靠RDB或者AOF文件对数据进行重新加载

候选者: 但在这时，只有一台Redis服务器存储着所有的数据，此时如果Redis服务器「暂时」没办法修复了，那依赖Redis的服务就没了

候选者: 所以，为了Redis「高可用」，现在基本都会给Redis做「备份」：多启一台Redis服务器，形成「主从架构」

候选者: 「从服务器」的数据由「主服务器」复制过去，主从服务器的数据是一致的

候选者: 如果主服务器挂了，那可以「手动」把「从服务器」升级为「主服务器」，缩短不可用时间



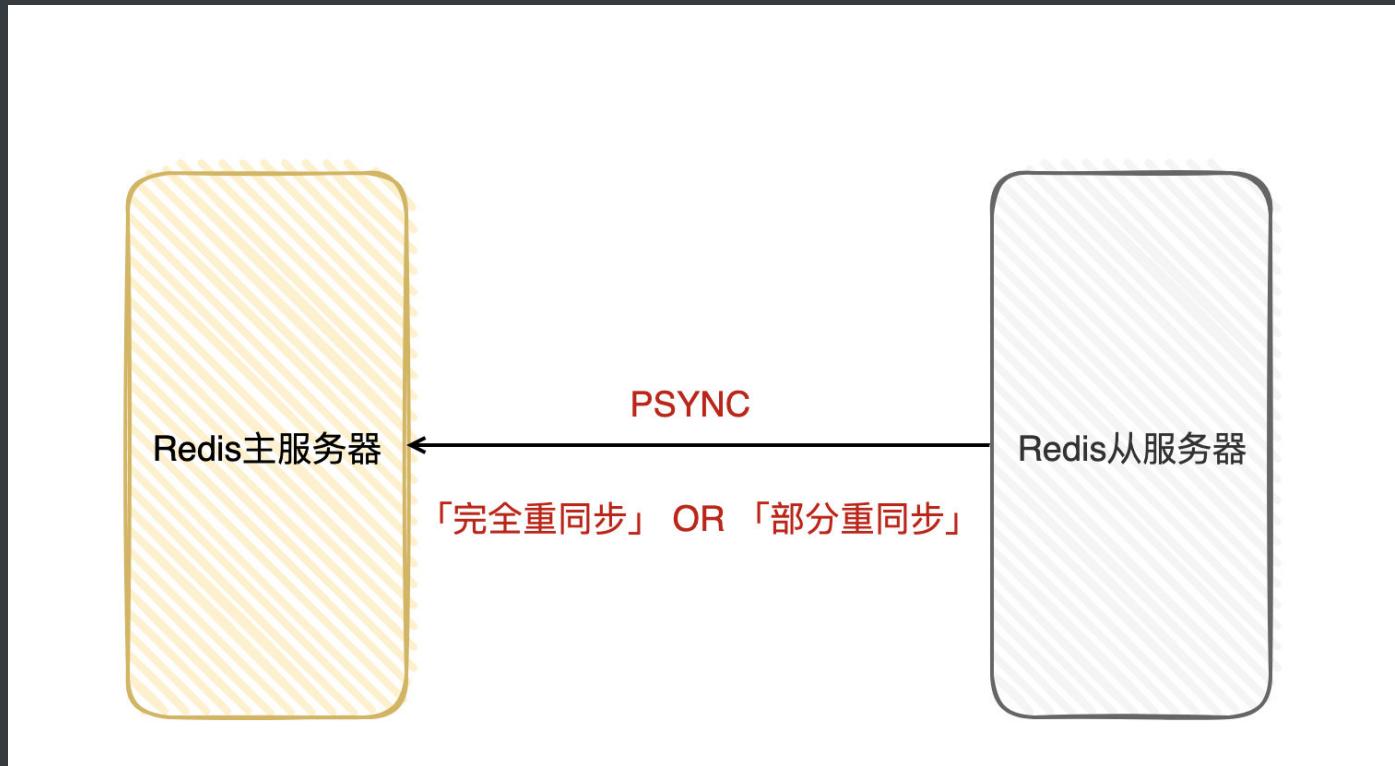
面试官: 那「主服务器」是如何把自身的数据「复制」给「从服务器」的呢？

候选者: 「复制」也叫「同步」，在Redis使用的是「PSYNC」命令进行同步，该命令有两种模型：完全重同步和部分重同步

候选者: 可以简单理解为：如果是第一次「同步」，从服务器没有复制过任何的主服务器，或者从服务器要复制的主服务器跟上次复制的主服务器不一样，那就会采用「完全重同步」模式进行复制

候选者：如果只是由于网络中断，只是「短时间」断连，那就会采用「部分重同步」模式进行复制

候选者：（假如主从服务器的数据差距实在是过大了，还是会采用「完全重同步」模式进行复制）



面试官：那「同步」的原理过程可以稍微讲下嘛？

候选者：嗯，没问题的

候选者：主服务器要复制数据到从服务器，首先是建立Socket「连接」，这个过程会干一些信息校验啊、身份校验啊等事情

候选者：然后从服务器就会发「PSYNC」命令给主服务器，要求同步（这时会带「服务器ID」RUNID和「复制进度」offset参数，如果从服务器是新的，那就没有）

候选者：主服务器发现这是一个新的从服务器（因为参数没带上来），就会采用「完全重同步」模式，并把「服务器ID」(runId)和「复制进度」(offset)发给从服务器，从服务器就会记下这些信息。

面试官：嗯...

候选者：随后，主服务器会在后台生成RDB文件，通过前面建立好的连接发给从服务器

候选者：从服务器收到RDB文件后，首先把自己的数据清空，然后对RDB文件进行加载恢复

候选者：这个过程中，主服务器也没闲着（继续接收着客户端的请求）

面试官：嗯...

候选者：主服务器把生成RDB文件「之后修改的命令」会用「buffer」记录下来，等到从服务器加载完RDB之后，主服务器会把「buffer」记录下的命令都发给从服务器

候选者：这样一来，主从服务器就达到了数据一致性了（复制过程是异步的，所以数据是『最终一致性』）

面试官：嗯...

面试官：那「部分重同步」的过程呢？

候选者：嗯，其实就是靠「offset」来进行部分重同步。每次主服务器传播命令的时候，都会把「offset」给到从服务器

候选者：主服务器和从服务器都会将「offset」保存起来（如果两边的offset存在差异，那么说明主从服务器数据未完全同步）

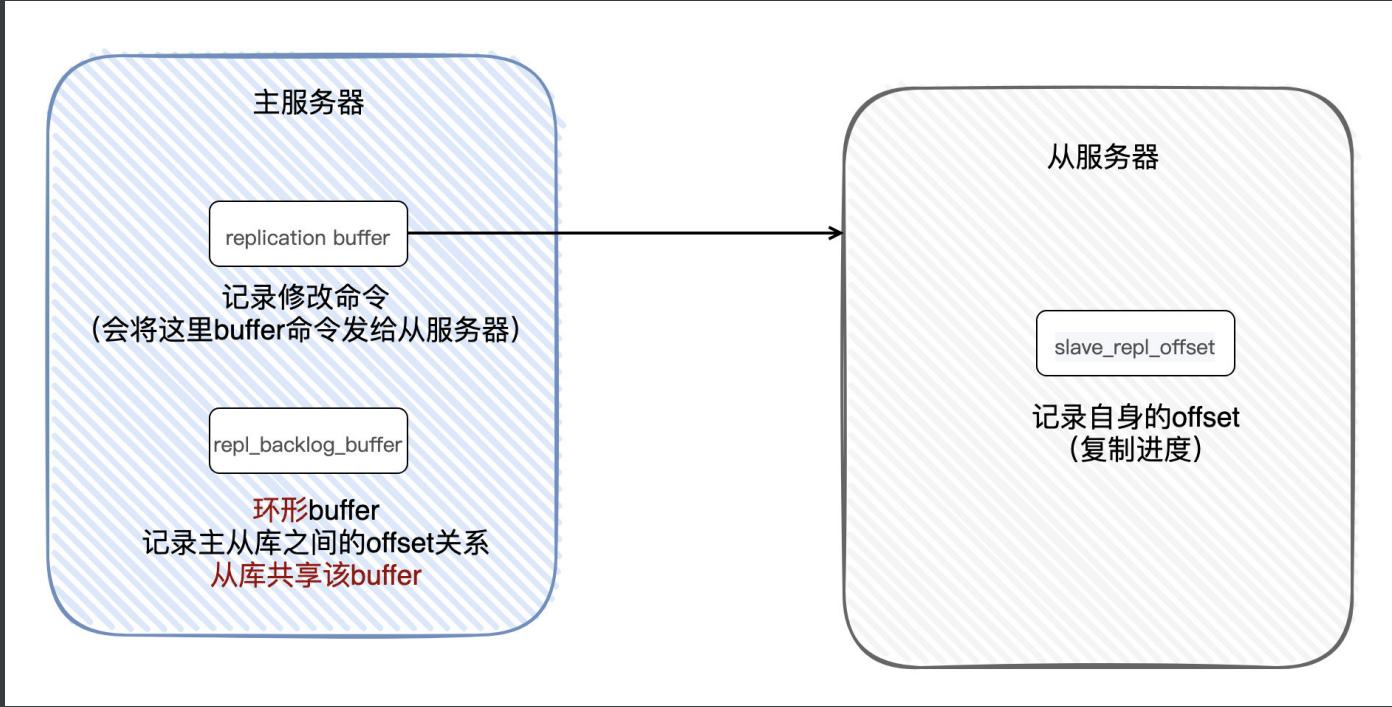
候选者：从服务器断连之后，就会发「PSYNC」命令给主服务器，同样也会带着RUNID和offset（重连之后，这些信息还是存在的）

面试官：嗯...

候选者：主服务器收到命令之后，看RUNID是否能对得上，对得上，说明这可能以前就复制过一部分了

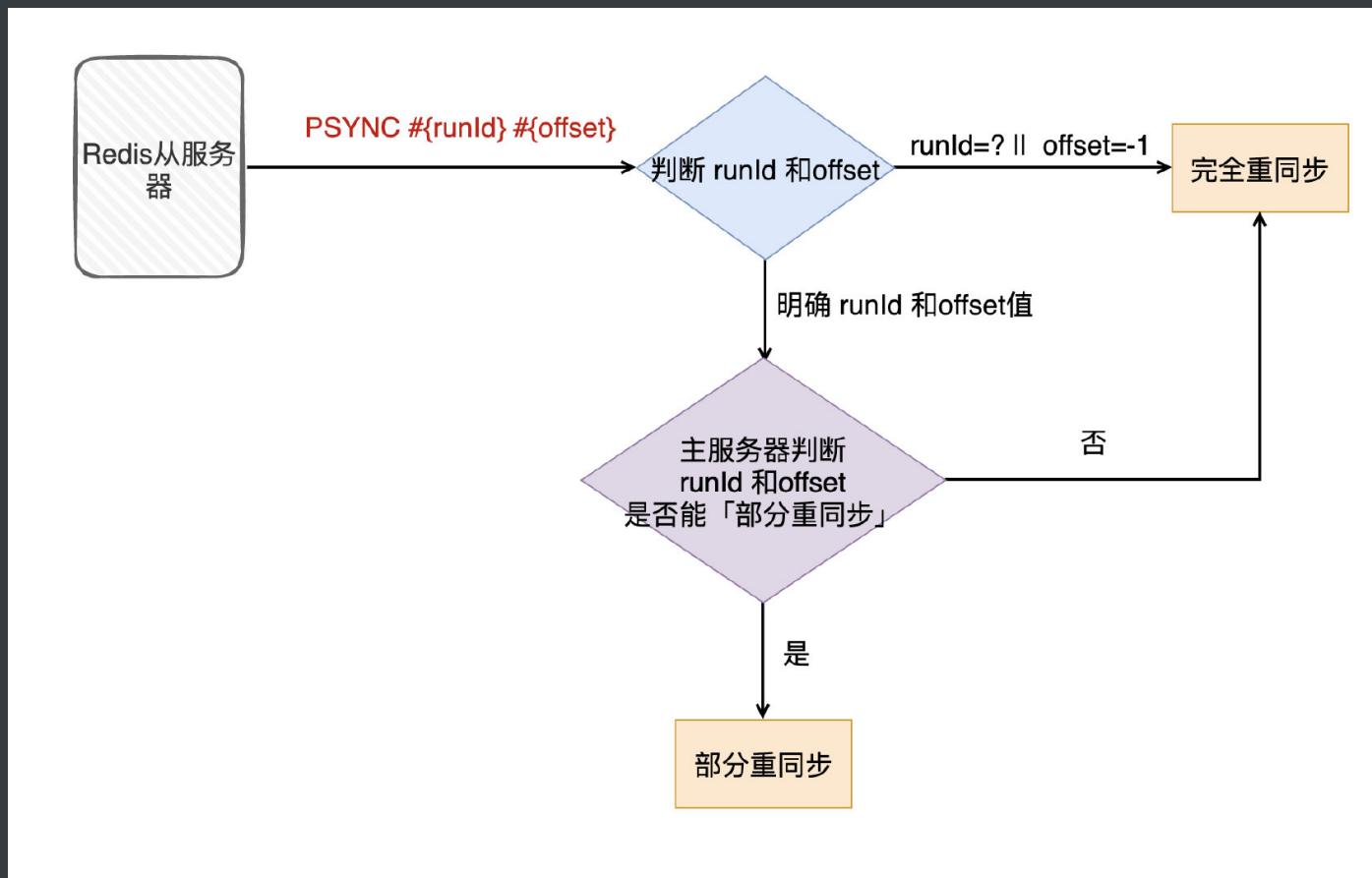
候选者：接着检查该「offset」是否在主服务器记录的offset还存在

候选者：（这里解释下，因为主服务器记录offset使用的是一个环形buffer，如果该buffer满了，会覆盖以前的记录）



候选者: 如果找到了，那就把从缺失的一部分offer开始，把对应的修改命令发给从服务器

候选者: 如果从环形buffer没找到，那只能使用「完全重同步」模式再次进行主从复制了



面试官：主从复制这块我了解了，那你说到现在，Redis主库如果挂了，你还是得「手动」将从库升级为主库啊

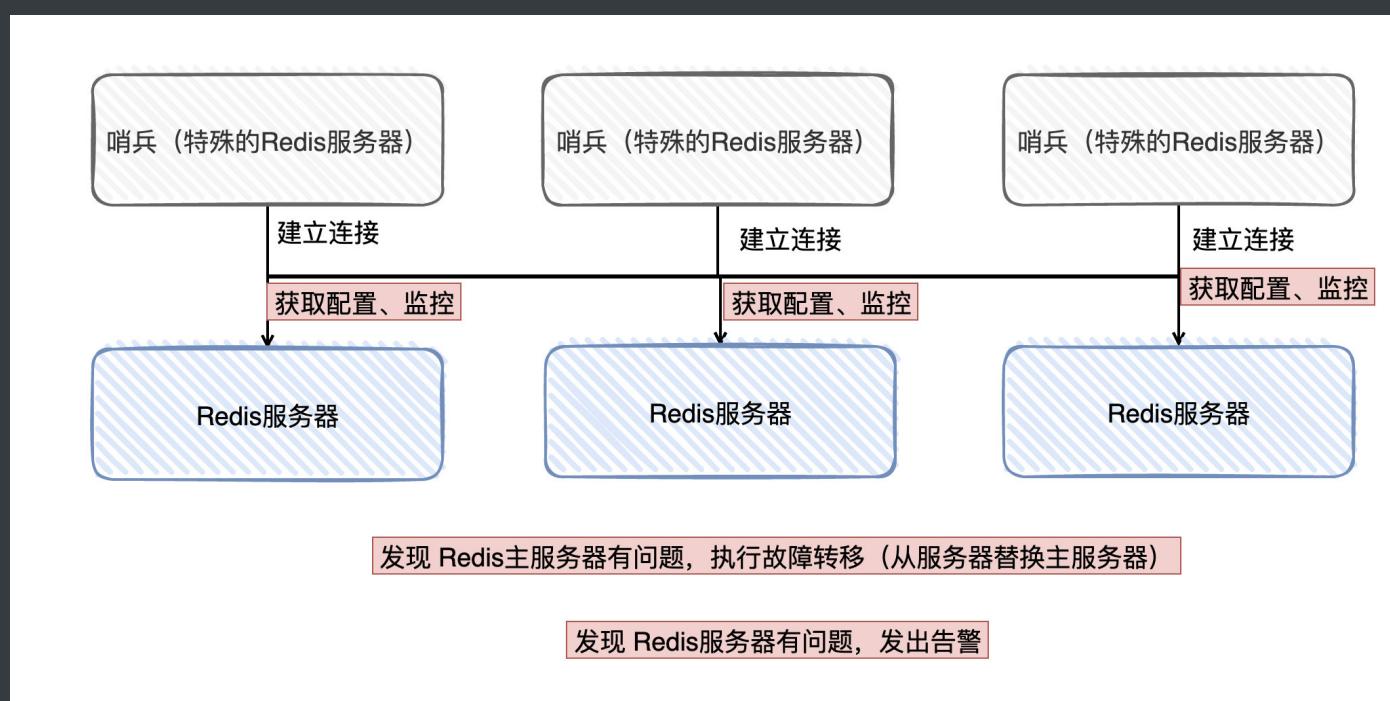
面试官：你知道有什么办法能做到「自动」进行故障恢复吗？

候选者：必须的啊，接下来就到了「哨兵」登场了

面试官：开始你的表演吧。

候选者：「哨兵」干的事情主要就是：监控（监控主服务器的状态）、选主（主服务器挂了，在从服务器选出一个作为主服务器）、通知（故障发送消息给管理员）和配置（作为配置中心，提供当前主服务器的信息）

候选者：可以把「哨兵」当做是运行在「特殊」模式下的Redis服务器，为了「高可用」，哨兵也是集群架构的。



候选者：首先它需要跟Redis主从服务器创建对应的连接（获取它们的信息）

候选者：每个哨兵不断地用ping命令看主服务器有没有下线，如果主服务器在「配置时间」内没有正常响应，那当前哨兵就「主观」认为该主服务器下线了

候选者：其他「哨兵」同样也会ping该主服务器，如果「足够多」（还是看配置）的哨兵认为该主服务器已经下线，那就认为「客观下线」，这时就要对主服务器执行故障转移操作。

面试官：嗯...

候选者：「哨兵」之间会选出一个「领头」，选出领头的规则也比较多，总的来说就是先到先得(哪个快，就选哪个)

候选者：由「领头哨兵」对已下线的主服务器进行故障转移

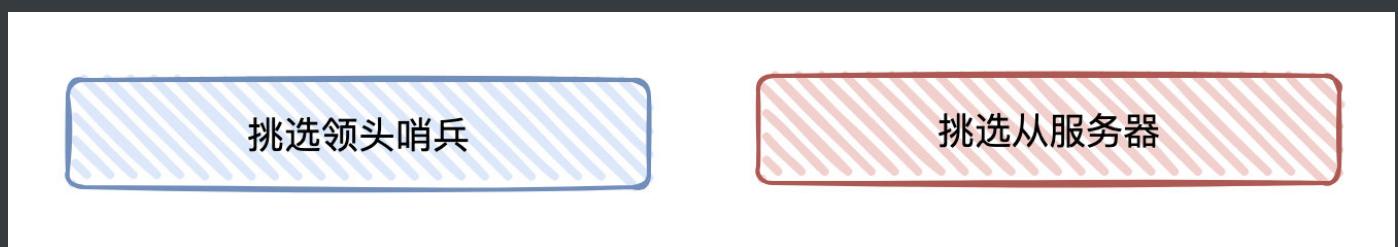
面试官：嗯...

候选者：首先要在「从服务器」上挑选出一个，来作为主服务器

候选者：(这里也挑选讲究，比如：从库的配置优先级、要判断哪个从服务器的复制offset最大、RunID大小、跟master断开连接的时长...)

候选者：然后，以前的从服务器都需要跟新的主服务器进行「主从复制」

候选者：已经下线的主服务器，再次重连的时候，需要让他成为新的主服务器的从服务器



面试官：嗯...我想问问，Redis在主从复制的和故障转移的过程中会导致数据丢失吗

候选者：显然是会的，从上面的「主从复制」流程来看，这个过程是异步的（在复制的过程中：主服务器会一直接收请求，然后把修改命令发给从服务器）

候选者：假如主服务器的命令还没发完给从服务器，自己就挂掉了。这时候想要让从服务器顶上主服务器，但从服务器的数据是不全的（：

候选者：还有另一种情况就是：有可能哨兵认为主服务器挂了，但真实是主服务器并没有挂(网络抖动)，而哨兵已经选举了一台从服务器当做是主服务器了，此时「客户端」还没反应过来，还继续写向旧主服务器写数据

候选者：等到旧主服务器重连的时候，已经被纳入到新主服务器的从服务器了...所以，那段时间里，客户端写进旧主服务器的数据就丢了

异步 导致数据丢失

脑裂 导致数据丢失

候选者：上面这两种情况（主从复制延迟&&脑裂），都可以通过配置来「尽可能」避免数据的丢失

候选者：（达到一定的阈值，直接禁止主服务器接收写请求，企图减少数据丢失的风险）

面试官：要不再来聊聊Redis分片集群？

候选者：嗯...分片集群说白了就是往每个Redis服务器存储一部分数据，所有的Redis服务器数据加起来，才组成完整的数据（分布式）

候选者：要想组成分片集群，那就需要对不同的key进行「路由」（分片）

候选者：现在一般的路由方案有两种：「客户端路由」(SDK)和「服务端路由」(Proxy)

候选者：客户端路由的代表（Redis Cluster），服务端路由的代表（Codis）

面试官：要不来详细讲讲它们的区别呗？

候选者：今天有点儿困了，要不下次呗？

本文总结：

- **Redis实现高可用：**

- AOF/RDB持久化机制
- 主从架构（主服务器挂了，手动由从服务器顶上）
- 引入哨兵机制自动故障转义

- **主从复制原理：**

- PSYNC命令两种模式：完全重同步、部分重同步

- 完全重同步：主从服务器建立连接、主服务器生成RDB文件发给从服务器、主服务器不阻塞（相关修改命令记录至buffer）、将修改命令发给从服务器
- 部分重同步：从服务器断线重连，发送RunId和offset给主服务器，主服务器判断offset和runId，将还未同步给从服务器的offset相关指令进行发送
- **哨兵机制：**
 - 哨兵可以理解为特殊的Redis服务器，一般会组成哨兵集群
 - 哨兵主要工作是监控、告警、配置以及选主
 - 当主服务器发生故障时，会「选出」一台从服务器来顶上「客观下线」的服务器，由「领头哨兵」进行切换
- **数据丢失：**
 - Redis的主从复制和故障转移阶段都有可能发生数据丢失问题（通过配置尽可能避免）



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



04、Redis分片集群

面试官：聊下Redis的分片集群，先聊 Redis Cluster好咯？

面试官：Redis Cluser是Redis 3.x才有的官方集群方案，这块你了解多少？

候选者：嗯，要不还是从基础讲起呗？

候选者：在前面聊Redis的时候，提到的Redis都是「单实例」存储所有的数据。

候选者：1. 主从模式下实现读写分离的架构，可以让多个从服务器承载「读流量」，但面对「写流量」时，始终是只有主服务器在抗。

候选者：2. 「纵向扩展」升级Redis服务器硬件能力，但升级至一定程度下，就不划算了。

候选者：纵向扩展意味着「大内存」，Redis持久化时的"成本"会加大（Redis做RDB持久化，是全量的，fork子进程时有可能由于使用内存过大，导致主线程阻塞时间过长）

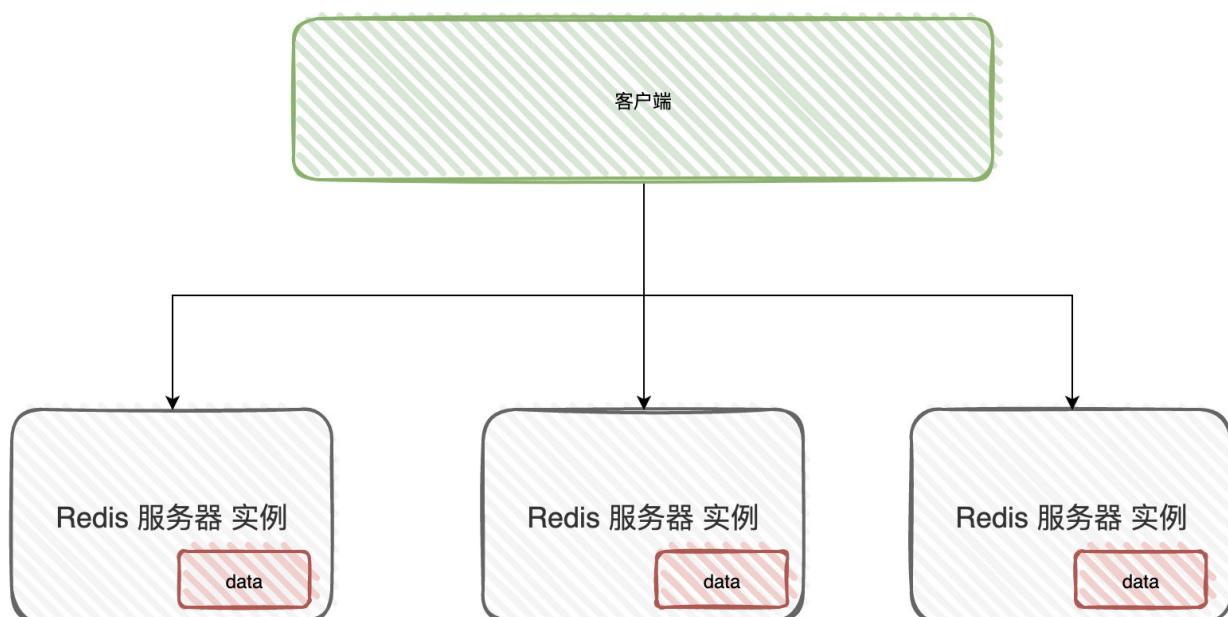
候选者：所以，「单实例」是有瓶颈的

单实例 存在瓶颈 (纵向扩容的成本 + 持久化的成本)

候选者：「纵向扩展」不行，就「横向扩展」呗。

候选者：用多个Redis实例来组成一个集群，按照一定的规则把数据「分发」到不同的Redis实例上。当集群所有的Redis实例的数据加起来，那这份数据就是全的

候选者：其实就是「分布式」的概念（：只不过，在Redis里，好像叫「分片集群」的人比较多？



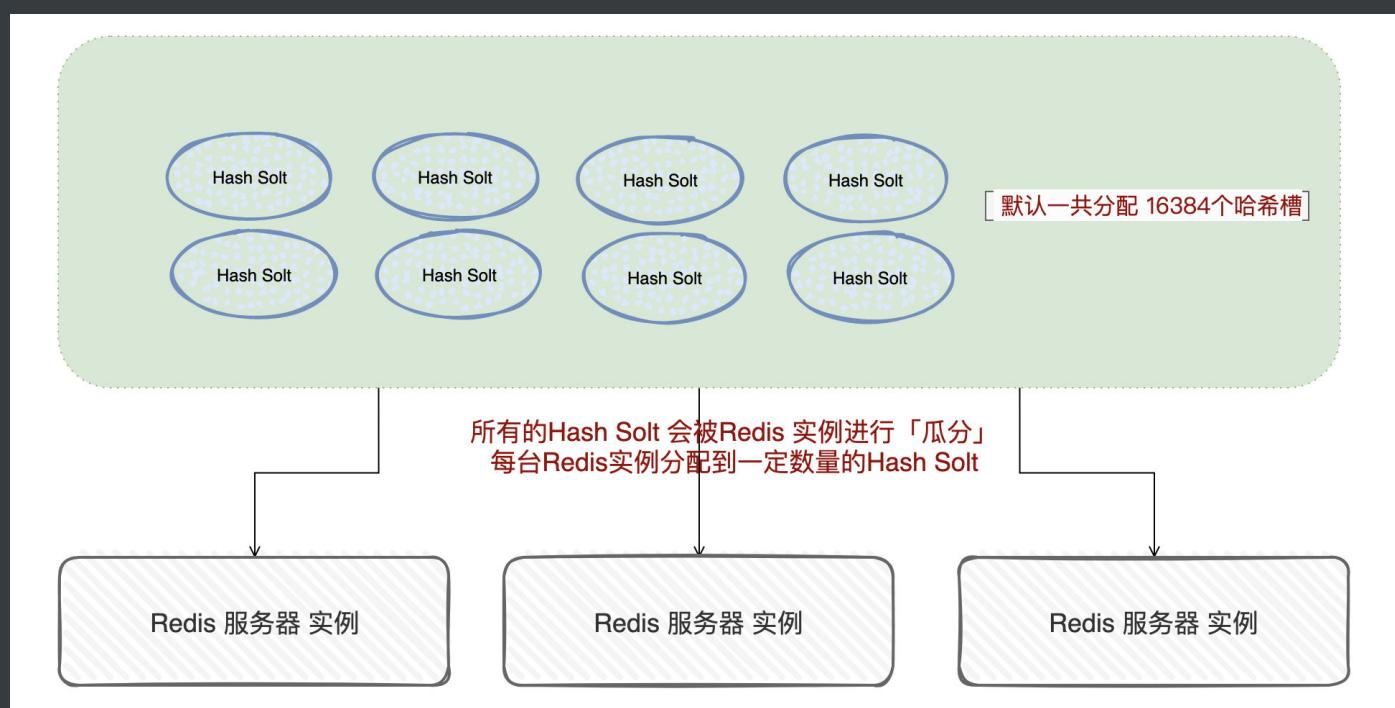
每个Redis实例 存储着 一部分数据，把所有的Redis实例数据组合起来，即是全量数据

候选者：从前面就得知了，要「分布式存储」，就肯定避免不了对数据进行「分发」(也是路由的意思)

候选者：从Redis Cluster讲起吧，它的「路由」是做在客户端的（SDK已经集成了路由转发的功能）

候选者：Redis Cluster对数据的分发的逻辑中，涉及到「哈希槽」(Hash Slot)的概念

候选者：Redis Cluster默认一个集群有16384个哈希槽，这些哈希槽会分配到不同的Redis实例中



候选者：至于怎么「瓜分」，可以直接均分，也可以「手动」设置每个Redis实例的哈希槽，全由我们来决定

候选者：重要的是，我们要把这16384个都得瓜分完，不能有剩余！

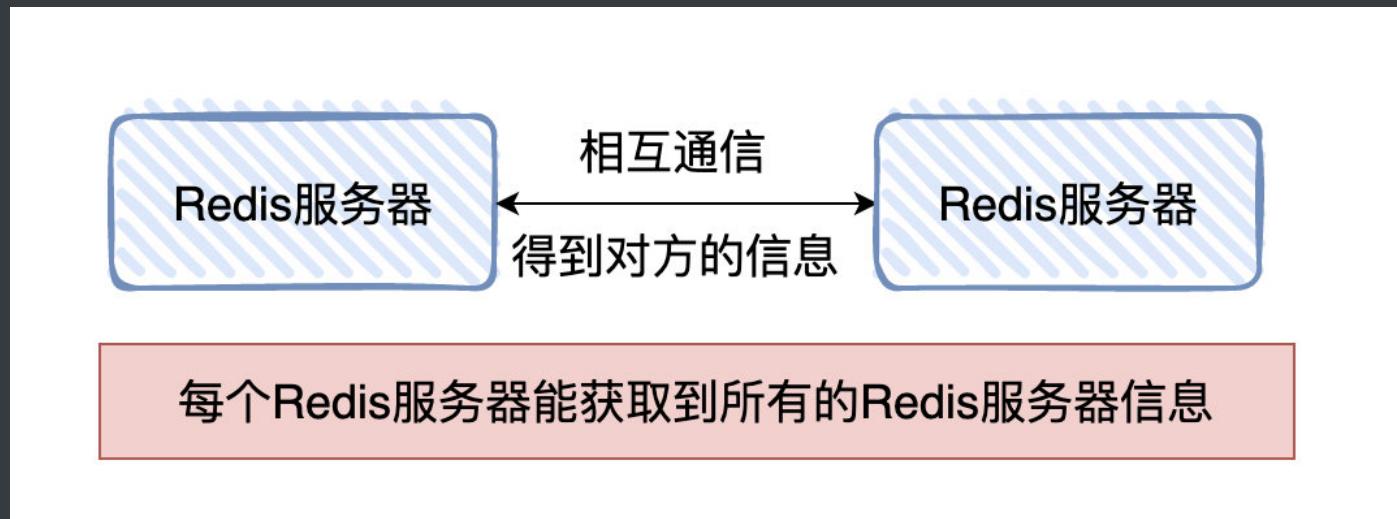
候选者：当客户端有数据进行写入的时候，首先会对key按照CRC16算法计算出16bit的值（可以理解为就是做hash），然后得到的值对16384进行取模

候选者：取模之后，自然就得到其中一个哈希槽，然后就可以将数据插入到分配至该哈希槽的Redis实例中

面试官：那问题就来了，现在客户端通过hash算法算出了哈希槽的位置，那客户端怎么知道这个哈希槽在哪台Redis实例上呢？

候选者: 是这样的，在集群的中每个Redis实例都会向其他实例「传播」自己所负责的哈希槽有哪些。这样一来，每台Redis实例就可以记录着「所有哈希槽与实例」的关系了（）：

候选者: 有了这个映射关系以后，客户端也会「缓存」一份到自己的本地上，那自然客户端就知道去哪个Redis实例上操作了



面试官: 那我又有问题了，在集群里也可以新增或者删除Redis实例啊，这个怎么整？

候选者: 当集群删除或者新增Redis实例时，那总会有某Redis实例所负责的哈希槽关系会发生变化

候选者: 发生变化的信息会通过消息发送至整个集群中，所有的Redis实例都会知道该变化，然后更新自己所保存的映射关系

候选者: 但这时候，客户端其实是不感知的（）

候选者: 所以，当客户端请求时某Key时，还是会请求到「原来」的Redis实例上。而原来的Redis实例会返回「moved」命令，告诉客户端应该要去新的Redis实例上去请求啦

候选者: 客户端接收到「moved」命令之后，就知道去新的Redis实例请求了，并且更新「缓存哈希槽与实例之间的映射关系」

候选者: 总结起来就是：数据迁移完毕后被响应，客户端会收到「moved」命令，并且会更新本地缓存

moved 命令 告诉客户端 需要重定向至 Redis服务器查询

面试官：那数据还没完全迁移完呢？

候选者：如果数据还没完全迁移完，那这时候会返回客户端「ask」命令。也是让客户端去请求新的Redis实例，但客户端这时候不会更新本地缓存

面试官：了解了

面试官：说白了就是，如果集群Redis实例存在变动，由于Redis实例之间会「通讯」

面试官：所以等到客户端请求时，Redis实例总会知道客户端所要请求的数据在哪个Redis实例上

面试官：如果已经迁移完毕了，那就返回「move」命令告诉客户端应该去找哪个Redis实例要数据，并且客户端应该更新自己的缓存(映射关系)

面试官：如果正在迁移中，那就返回「ack」命令告诉客户端应该去找哪个Redis实例要数据

候选者：不愧是你...

面试官：那你知道为什么哈希槽是16384个吗？

候选者：嗯，这个。是这样的，Redis实例之间「通讯」会相互交换「槽信息」，那如果槽过多（意味着网络包会变大），网络包变大，那是不是就意味着会「过度占用」网络的带宽

候选者：另外一块是，Redis作者认为集群在一般情况下是不会超过1000个实例

候选者：那就取了16384个，即可以将数据合理打散至Redis集群中的不同实例，又不会在交换数据时导致带宽占用过多

作者认为：

1. Redis 集群 实例 不超过 1000个，16384个可以均衡分配了
2. 实例过多，会导致Redis服务器通信的网络包过大（网络开销）

面试官：了解了

面试官：那你知道为什么对数据进行分区在Redis中用的是「哈希槽」这种方式吗？而不是一致性哈希算法

候选者：在我理解下，一致性哈希算法就是有个「哈希环」，当客户端请求时，会对Key进行hash，确定在哈希环上的位置，然后顺时针往后找，找到的第一个真实节点

候选者：一致性哈希算法比「传统固定取模」的好处就是：如果集群中需要新增或删除某实例，只会影响一小部分的数据

候选者：但如果在集群中新增或者删除实例，在一致性哈希算法下，就得知道是「哪一部分数据」受到影响了，需要进行对受影响的数据进行迁移

面试官：嗯...

候选者：而哈希槽的方式，我们通过上面已经可以发现：在集群中的每个实例都能拿到槽位相关的信息

候选者：当客户端对key进行hash运算之后，如果发现请求的实例没有相关的数据，实例会返回「重定向」命令告诉客户端应该去哪儿请求

候选者：集群的扩容、缩容都是以「哈希槽」作为基本单位进行操作，总的来说就是「实现」会更加简单（简洁，高效，有弹性）。过程大概就是把部分槽进行重新分配，然后迁移槽中的数据即可，不会影响到集群中某个实例的所有数据。

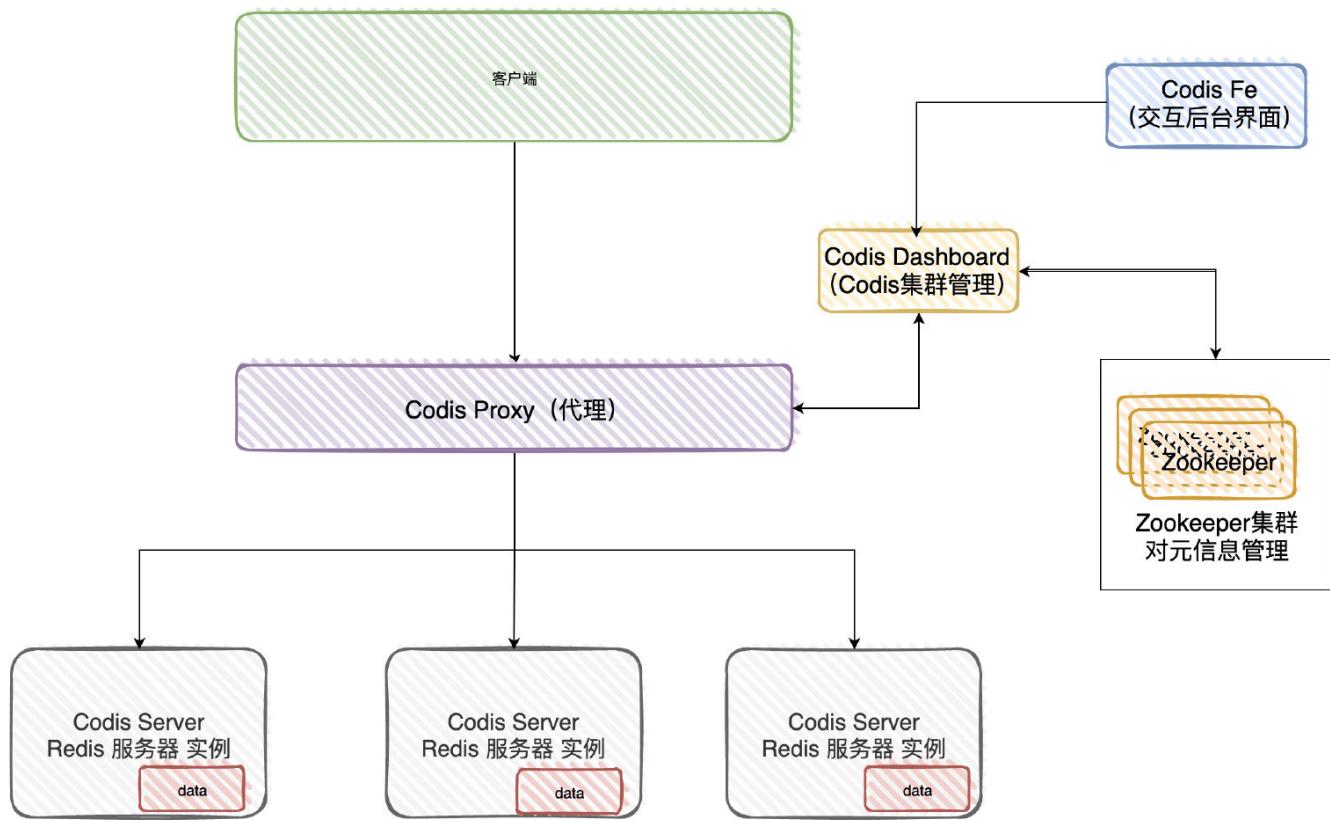
以Solt为载体，实现简单，高效

面试官：那你了解「服务端 路由」的大致原理吗？

候选者：嗯，服务端路由一般指的就是，有个代理层专门对接客户端的请求，然后再转发到Redis集群进行处理

候选者：上次最后面试的时候，也提到了，现在比较流行的是Codis

候选者：它与Redis Cluster最大的区别就是，Redis Cluster是直连Redis实例的，而Codis则客户端直连Proxy，再由Proxy进行分发到不同的Redis实例进行处理



候选者：在Codis对Key路由的方案跟Redis Cluster很类似，Codis初始化出1024个哈希槽，然后分配到不同的Redis服务器中

候选者：哈希槽与Redis实例的映射关系由Zookeeper进行存储和管理，Proxy会通过Codis DashBoard得到最新的映射关系，并缓存在本地上

面试官：那如果我要扩容Codis Redis实例的流程是怎么样的？

候选者：简单来说就是：把新的Redis实例加入到集群中，然后把部分数据迁移到新的实例上

候选者：大概的过程就是：1.「原实例」某一个Solt的部分数据发送给「目标实例」。2.「目标实例」收到数据后，给「原实例」返回ack。3.「原实例」收到ack之后，在本地删除掉刚刚给「目标实例」的数据。4.不断循环1、2、3步骤，直至整个solt迁移完毕

候选者：Codis也是支持「异步迁移」的，针对上面的步骤2，「原实例」发送数据后，不等待「目标实例」返回ack，就继续接收客户端的请求。

候选者：未迁移完的数据标记为「只读」，不会影响到数据的一致性。如果对迁移中的数据存在「写操作」，那会让客户端进行「重试」，最后会写到「目标实例」上

以Solt为载体，发送Solt数据中每个Key的数据，直至完成

同步则需要ack

异步则「只读」和重试

候选者：还有就是，针对 bigkey，异步迁移采用了「拆分指令」的方式进行迁移，比如有个set元素有10000个，那「原实例」可能就发送10000条命令给「目标实例」，而不是一整个bigkey一次性迁移（因为大对象容易造成阻塞）

面试官：了解了。

对比维度	Codis	Redis Cluster
数据路由信息	中心化保存在Zookeeper, proxy在本地缓存	每个实例都保存一份
集群扩容	增加codis server和codis proxy	增加Redis实例
数据迁移	支持同步迁移、异步迁移	支持同步迁移
客户端兼容性	兼容单实例客户端	需要开发支持Cluster功能的客户端
可靠性	codis server: 主从集群机制保证可靠性 codis proxy: 无状态设计, 故障后重启就行 Zookeeper: 可靠性高, 只要有半数以上节点存在就能继续服务	实例使用主从集群保证可靠性

本文总结:

- **分片集群诞生理由**: 写性能在高并发下会遇到瓶颈&&无法无限地纵向扩展 (不划算)
- **分片集群**: 需要解决「数据路由」和「数据迁移」的问题
- **Redis Cluster数据路由**:
 - Redis Cluster默认一个集群有16384个哈希槽, 哈希槽会被分配到Redis集群中的实例中
 - Redis集群的实例会相互「通讯」, 交互自己所负责哈希槽信息 (最终每个实例都有完整的映射关系)
 - 当客户端请求时, 使用CRC16算法算出Hash值并模以16384, 自然就能得到哈希槽进而得到所对应的Redis实例位置
- **为什么16384个哈希槽**: 16384个既能让Redis实例分配到的数据相对均匀, 又不会影响Redis实例之间交互槽信息产生严重的网络性能开销问题
- **Redis Cluster 为什么使用哈希槽, 而非一致性哈希算法**: 哈希槽实现相对简单高效, 每次扩缩容只需要动对应Slot (槽) 的数据, 一般不会动整个Redis实例
- **Codis数据路由**: 默认分配1024个哈希槽, 映射相关信息会被保存至Zookeeper集群。Proxy会缓存一份至本地, Redis集群实例发生变化时, DashBoard更新Zookeeper和Proxy的映射信息
- **Redis Cluster和Codis数据迁移**: Redis Cluster支持同步迁移, Codis支持同步迁移&&异步迁移
 - 把新的Redis实例加入到集群中, 然后把部分数据迁移到新的实例上 (在线)



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zhongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

07-消息队列

01、Kafka基础

面试官：今天要不来聊聊消息队列吧？我看你项目不少地方都写到Kafka了

候选者：嗯嗯

面试官：那你简单说明下你使用Kafka的场景吧

候选者：使用消息队列的目的总的来说可以有三种情况：解耦、异步和削峰

候选者：比如举我项目的例子吧，我现在维护一个消息管理平台系统，对外提供接口给各个业务方调用

候选者：他们调用接口之后，实际上『不是同步』下发了消息。

候选者：在接口处理层只是把该条消息放到了消息队列上，随后就直接返回结果给接口调用者了。

候选者：这样的好处就是：

候选者：1. 接口的吞吐量会大幅度提高（因为未做真正实际调用，接口RT会非常低）【异步】

候选者：2. 即便有大批量的消息调用接口都不会让系统受到影响（流量由消息队列承载）【削峰】



候选者：又比如说，我这边还有个项目是广告订单归因工程，主要做的事情就是得到订单数据，给各个业务广告计算对应的佣金。

候选者: 订单的数据是从消息队列里取出的

候选者: 这样设计的好处就是:

候选者: 1. 交易团队的同学只要把订单消息写到消息队列，该订单数据的Topic由各个业务方自行消费使用 【解耦】 【异步】

候选者: 2. 即便下单QPS猛增，对下游业务无太大的感知（因为下游业务只消费消息队列的数据，不会直接影响到机器性能） 【削峰】

面试官: 嗯，那我想问下，**你觉得为什么消息队列能到削峰？**

面试官: 或者换个问法，**为什么Kafka能承载这么大的QPS？**

候选者: 消息队列「最核心」的功能就是把生产的数据存储起来，然后给各个业务把数据再读取出来。

候选者: 跟我们处理请求时不一样：在业务处理时可能会调别人的接口，可能会需要去查数据库...等等等一系列的操作才行

候选者: 像Kafka在「存储」和「读取」这个过程中又做了很多的优化

候选者: 举几个例子，比如说：

候选者: 我们往一个Topic发送消息或者读取消息时，实际内部是多个Partition在处理 【并行】

候选者: 在存储消息时，Kafka内部是顺序写磁盘的，并且利用了操作系统的缓冲区来提高性能 【append+cache】

候选者: 在读写数据中也减少CPU拷贝文件的次数 【零拷贝】



面试官：嗯，你既然提到减少CPU拷贝文件的次数，可以给我说下这项技术吗？

候选者：嗯，可以的，其实就是零拷贝技术。

候选者：比如我们正常调用read函数时，会发生以下的步骤：

候选者：1. DMA把磁盘的拷贝到读内核缓存区

候选者：2. CPU把读内核缓冲区的数据拷贝到用户空间

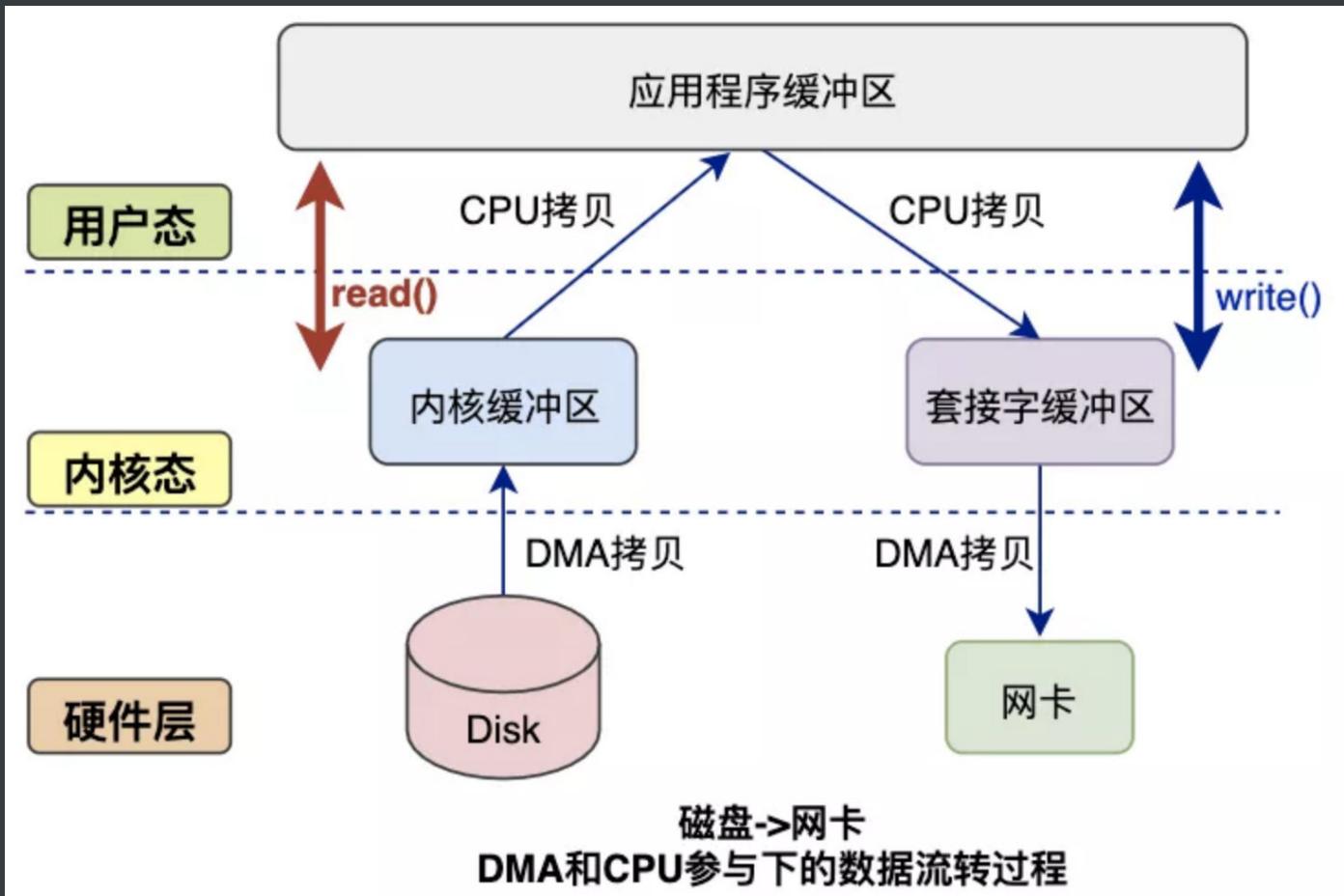
候选者：正常调用write函数时，会发生以下的步骤：

候选者：1. CPU把用户空间的数据拷贝到Socket内核缓存区

候选者：2. DMA把Socket内核缓冲区的数据拷贝到网卡

候选者：可以发现完成「一次读写」需要2次DMA拷贝，2次CPU拷贝。而DMA拷贝是省不了的，所谓的零拷贝技术就是把CPU的拷贝给省掉

候选者：并且为了避免用户进程直接操作内核，保证内核安全，应用程序在调用系统函数时，会发生上下文切换（上述的过程一共会发生4次）



候选者：目前零拷贝技术主要有：mmap和sendfile，这两种技术会一定程度下减少上下文切换和CPU的拷贝

候选者：比如说：mmap是将读缓冲区的地址和用户空间的地址进行映射，实现读内核缓冲区和应用缓冲区共享

候选者：从而减少了从读缓冲区到用户缓冲区的一次CPU拷贝

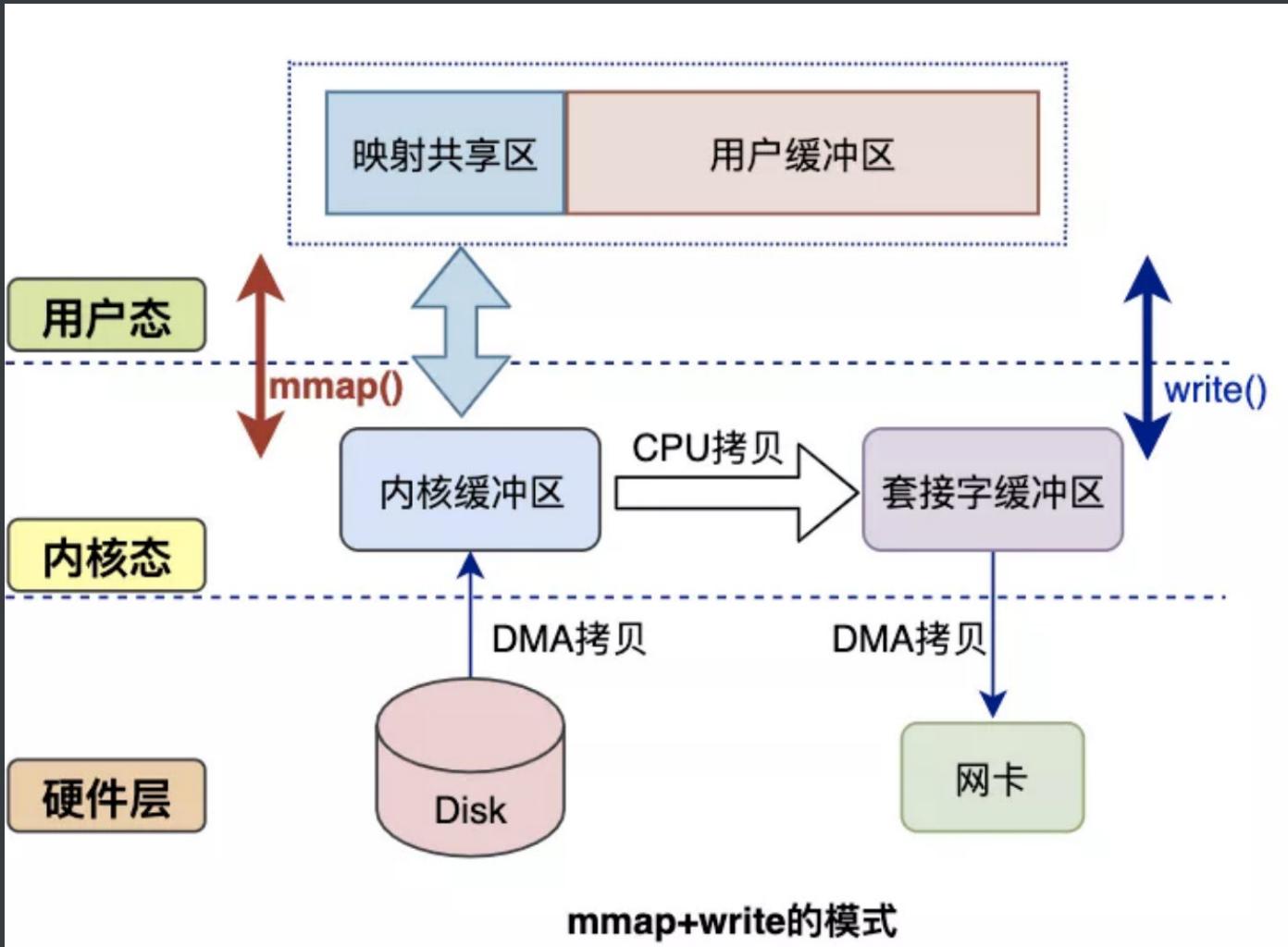
候选者：使用mmap的后一次读写就可以简化为：

候选者：一、DMA把硬盘数据拷贝到读内核缓冲区。

候选者：二、CPU把读内核缓存区拷贝至Socket内核缓冲区。

候选者：三、DMA把Socket内核缓冲区拷贝至网卡

候选者：由于读内核缓冲区与用户空间做了映射，所以会省了一次CPU拷贝



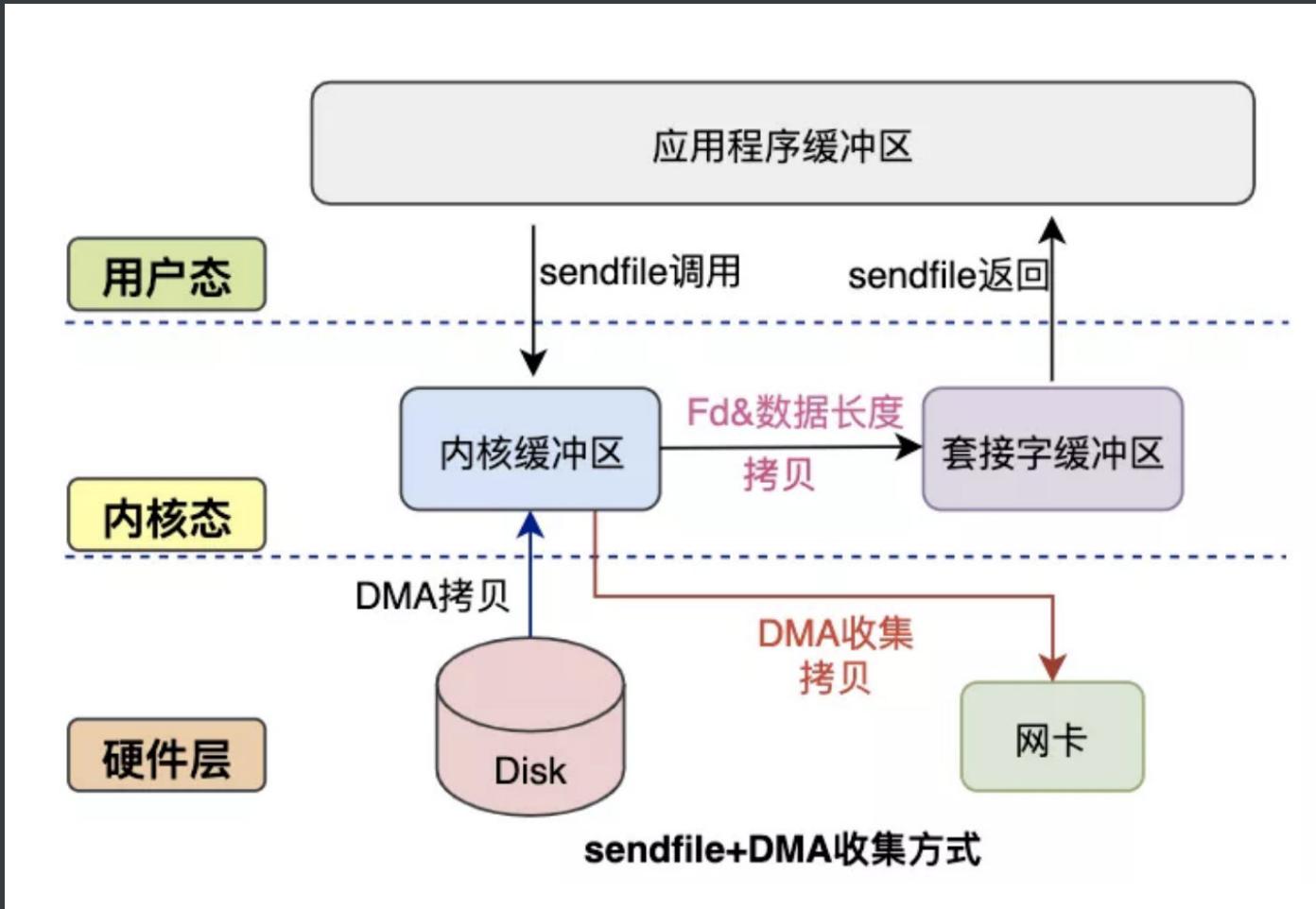
候选者：而sendfile+DMA Scatter/Gather则是把读内核缓存区的文件描述符/长度信息发到Socket内核缓冲区，实现CPU零拷贝

候选者：使用sendfile+DMA Scatter/Gather一次读写就可以简化为：

候选者：一、DMA把硬盘数据拷贝至读内核缓冲区。

候选者：二、CPU把读缓冲区的文件描述符和长度信息发到Socket缓冲区。

候选者：三、DMA根据文件描述符和数据长度从读内核缓冲区把数据拷贝至网卡



候选者：回到Kafka上

候选者：从Producer->Broker, Kafka是把网卡的数据持久化硬盘，用的是mmap（从2次CPU拷贝减至1次）

候选者：从Broker->Consumer, Kafka是从硬盘的数据发送至网卡，用的是sendFile（实现CPU零拷贝）

面试官：我稍微打断下，我还有点事忙，我总结下你说的话吧

面试官：你用Kafka的原因是为了异步、削峰、解耦

面试官：Kafka能这么快的原因就是实现了并行、充分利用操作系统cache、顺序写和零拷贝

面试官：没错吧？

候选者：嗯

面试官：ok，下次继续面吧，我这边有点忙



第一时间获取**BATJTMD一线互联网大厂**最新的面试资料以及内推机会关注公众号「对线面试官」



02、使用Kafka会考虑什么问题

面试官：今天我想问下，你觉得**Kafka**会丢数据吗？

候选者：嗯，使用Kafka时，有可能会有以下场景会丢消息

候选者：比如说，我们用Producer发消息至Broker的时候，就有可能会丢消息

候选者：如果你不想丢消息，那在发送消息的时候，需要选择带有 callBack的api进行发送

候选者：其实就意味着，如果你发送成功了，会回调告诉你已经发送成功了。如果失败了，那收到回调之后自己在业务上做重试就好了。

候选者：等到把消息发送到Broker以后，也有可能丢消息

候选者：一般我们的线上环境都是集群环境下嘛，但可能你发送的消息后broker就挂了，这时挂掉的broker还没来得及把数据同步给别的broker，数据就自然就丢了

候选者：发送到Broker之后，也不能保证数据就一定不丢了，毕竟Broker会把数据存储到磁盘之前，走的是操作系统缓存

候选者：也就是异步刷盘这个过程还有可能导致数据会丢



候选者：嗯，到这里其实我已经说了三个场景了，分别是：producer -> broker， broker->broker之间同步，以及broker->磁盘

候选者：要解决上面所讲的问题也比较简单，这块也没什么好说的...

候选者: 不想丢数据，那就使用带有callback的api，设置 acks、retries、factor等等些参数来保证Producer发送的消息不会丢就好啦。

面试官: �恩...

候选者: 一般来说，还是client 消费 broker 丢消息的场景比较多

面试官: 那你们在消费数据的时候是怎么保证数据的可靠性的呢？

候选者: 首先，要想client端消费数据不能丢，肯定是不能使用autoCommit的，所以必须是手动提交的。

消费者 不希望数据丢失，需要 手动提交 offset

候选者: 我们这边是这样实现的：

候选者: 一、从Kafka拉取消息（一次批量拉取500条，这里主要看配置）时

候选者: 二、为每条拉取的消息分配一个msgId（递增）

候选者: 三、将msgId存入内存队列（sortSet）中

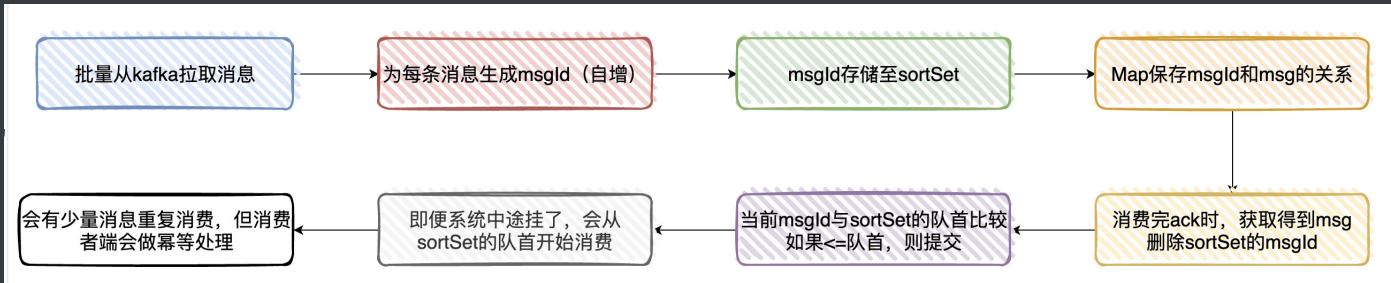
候选者: 四、使用Map存储msgId与msg(有offset相关的信息) 的映射关系

候选者: 五、当业务处理完消息后，ack时，获取当前处理的消息msgId，然后从sortSet删除该msgId（此时代表已经处理过了）

候选者: 六、接着与sortSet队列的首部第一个Id比较（其实就是最小的msgId），如果当前 msgId<=sort Set第一个ID，则提交当前offset

候选者: 七、系统即便挂了，在下次重启时就会从sortSet队首的消息开始拉取，实现至少处理一次语义

候选者: 八、会有少量的消息重复，但只要下游做好幂等就OK了。



面试官：嗯，你也提到了幂等，你们这业务怎么实现幂等性的呢？

候选者：嗯，还是以处理订单消息为例好了。

候选者：幂等Key我们由订单编号+订单状态所组成（一笔订单的状态只会处理一次）

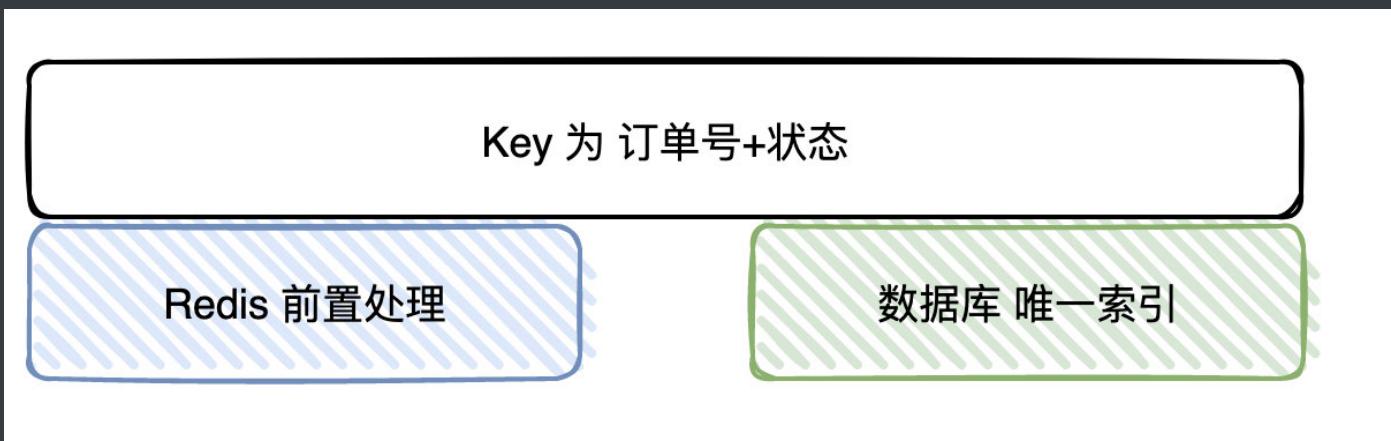
候选者：在处理之前，我们首先会去查Redis是否存在该Key，如果存在，则说明我们已经处理过了，直接丢掉

候选者：如果Redis没处理过，则继续往下处理，最终的逻辑是将处理过的数据插入到业务DB上，再到最后把幂等Key插入到Redis上

候选者：在一般情况下，通过Redis是无法保证幂等的（：

候选者：所以，Redis其实只是一个「前置」处理，最终的幂等性是依赖数据库的唯一Key来保证的（唯一Key实际上也是订单编号+状态）

候选者：总的来说，就是通过Redis做前置处理，DB唯一索引做最终保证来实现幂等性的



面试官：你们那边遇到过顺序消费的问题吗？

候选者: 嗯，也是有的，我举个例子

候选者: 订单的状态比如有 支付、确认收货、完成等等，而订单下还有计费、退款的消息报

候选者: 理论上来说，支付的消息报肯定要比退款消息报先到嘛，但程序处理的过程中可不一定的嘛

候选者: 所以在这边也是有消费顺序的问题

候选者: 但在广告场景下不是「强顺序」的，只要保证最终一致性就好了。

候选者: 所以我们这边处理「乱序」消息的实现是这样的：

候选者: 一、宽表：将每一个订单状态，单独分出一个或多个独立的字段。消息来时只更新对应的字段就好，消息只会存在短暂的状态不一致问题，但是状态最终是一致的

候选者: 二、消息补偿机制：另一个进行消费相同topic的数据，消息落盘，延迟处理。将消息与DB进行对比，如果发现数据不一致，再重新发送消息至主进程处理

候选者: 还有部分场景，可能我们只需要把相同userId/orderId发送到相同的partition（因为一个partition由一个Consumer消费），又能解决大部分消费顺序的问题了呢。



面试官: 啊...懂了



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

08-MySQL

01、MySQL索引

面试官：我看你简历上写了MySQL，对MySQL InnoDB引擎的索引了解吗？

候选者：嗯啊，使用索引可以加快查询速度，其实上就是将无序的数据变成有序（有序就能加快检索速度）

候选者：在InnoDB引擎中，索引的底层数据结构是B+树

面试官：那为什么不使用红黑树或者B树呢？

候选者：MySQL的数据是存储在硬盘的，在查询时一般是不能「一次性」把全部数据加载到内存中

候选者：红黑树是「二叉查找树」的变种，一个Node节点只能存储一个Key和一个Value

候选者：B和B+树跟红黑树不一样，它们算是「多路搜索树」，相较于「二叉搜索树」而言，一个Node节点可以存储的信息会更多，「多路搜索树」的高度会比「二叉搜索树」更低。

候选者：了解了区别之后，其实就很容易发现，在数据不能一次加载至内存的场景下，数据需要被检索出来，选择B或B+树的理由就很充分了（一个Node节点存储信息更多（相较于二叉搜索树），树的高度更低，树的高度影响检索的速度）

候选者：B+树相对于B树而言，它又有两种特性。

候选者：一、B+树非叶子节点不存储数据，在相同的数据量下，B+树更加矮壮。（这个应该不用多解释了，数据都存储在叶子节点上，非叶子节点的存储能存储更多的索引，所以整棵树就更加矮壮）

候选者：二、B+树叶子节点之间组成一个链表，方便于遍历查询（遍历操作在MySQL中比较常见）

B+树是多路搜索树
树的层级更低（检索更快）

B+树只有叶子节点存储数据
B+树比B树更加矮

B+树叶子节点有双向链表
便于遍历数据（遍历场景多）

候选者：我稍微解释一下吧，你可以脑补下画面

候选者：我们在MySQL InnoDB引擎下，每创建一个索引，相当于生成了一颗B+树。

候选者：如果该索引是「聚集(聚簇)索引」，那当前B+树的叶子节点存储着「主键和当前行的数据」

候选者：如果该索引是「非聚簇索引」，那当前B+树的叶子节点存储着「主键和当前索引列值」

候选者：比如写了一句sql：select * from user where id >=10，那只要定位到id为10的记录，然后在叶子节点之间通过遍历链表(叶子节点组成的链表)，即可找到往后的记录了。

候选者：由于B树是会在非叶子节点也存储数据，要遍历的时候可能就得跨层检索，相对麻烦些。

候选者：基于树的层级以及业务使用场景的特性，所以MySQL选择了B+树作为索引的底层数据结构。

候选者：对于哈希结构，其实InnoDB引擎是「自适应」哈希索引的（hash索引的创建由InnoDB存储引擎自动优化创建，我们是干预不了）

面试官：嗯...那我了解了，顺便想问下，你知道什么叫做回表吗？

候选者：所谓的回表其实就是，当我们使用索引查询数据时，检索出来的数据可能包含其他列，但走的索引树叶子节点只能查到当前列值以及主键ID，所以需要根据主键ID再去查一遍数据，得到SQL所需的列

候选者：举个例子，我这边建了给订单号ID建了个索引，但我的SQL是：select orderId,orderName from orderdetail where orderId = 123

候选者：SQL都订单ID索引，但在订单ID的索引树的叶子节点只有orderId和Id，而我们还想检索出orderName，所以MySQL会拿到ID再去查出orderName给我们返回，这种操作就叫回表

回表：当前索引无法检索出完整的内容，需要通过主键二次查询

候选者：想要避免回表，也可以使用覆盖索引（能使用就使用，因为避免了回表操作）。

候选者：所谓的覆盖索引，实际上就是你想要查出的列刚好在叶子节点上都存在，比如我建了orderId和orderName联合索引，刚好我需要查询也是orderId和orderName，这些数据都存在索引树的叶子节点上，就不需要回表操作了。

面试官：既然你也提到了联合索引，我想问下你了解最左匹配原则吗？

候选者：嗯，说明这个概念，还是举例子比较容易说明

候选者：如有索引 (a,b,c,d)，查询条件 $a=1 \text{ and } b=2 \text{ and } c>3 \text{ and } d=4$ ，则会在每个节点依次命中a、b、c，无法命中d

候选者：先匹配最左边的，索引只能用于查找key是否存在（相等），遇到范围查询(>、<、between、like左匹配)等就不能进一步匹配了，后续退化为线性查找

候选者：这就是最左匹配原则

覆盖索引，最左匹配原则是优化查询的常见思路

面试官：嗯嗯，我还想问下你们主键是怎么生成的？

候选者：主键就自增的

面试官：那假设我不用MySQL自增的主键，你觉得会有什么问题呢？

候选者：首先主键得保证它的唯一性和空间尽可能短吧，这两块是需要考虑的。

候选者：另外，由于索引的特性（有序），如果生成像uuid类似的主键，那插入的性能是比自增的要差的

候选者：因为生成的uuid，在插入时有可能需要移动磁盘块（比如，块内的空间在当前时刻已经存储满了，但新生成的uuid需要插入已满的块内，就需要移动块的数据）

面试官：OK...

非主键自增：需要考虑长度、唯一性以及块移动的问题

本文总结：

- **为什么B+树？** 数据无法一次load到内存，B+树是多路搜索树，只有叶子节点才存储数据，叶子节点之间链表进行关联。（树矮，易遍历）
- **什么是回表？** 非聚簇索引在叶子节点只存储列值以及主键ID，有条件下尽可能用覆盖索引避免回表操作，提高查询速度
- **什么是最左匹配原则？** 从最左边为起点开始连续匹配，遇到范围查询终止
- **主键非自增会有什么问题？** 插入效率下降，存在移动块的数据问题



第一时间获取**BATJTMD一线互联网大厂**最新的面试资料以及内推机会关注公众号「对线面试官」



02、MySQL事务和锁机制和MVCC

面试官：你是怎么理解InnoDB引擎中的事务的？

候选者：在我的理解下，事务可以使「一组操作」要么全部成功，要么全部失败

候选者：事务其目的是为了「保证数据最终的一致性」。

候选者：举个例子，我给你发支付宝转了888块红包。那自然我的支付宝余额会扣减888块，你的支付宝余额会增加888块。

候选者：而事务就是保证我的余额扣减跟你的余额增添是同时成功或者同时失败的，这样这次转账就正常了

保证事务的最终一致性

面试官：嗯，那你了解事务的几大特性吗？

候选者: 嗯，就是ACID嘛，分别是原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

候选者: 原子性指的是：当前事务的操作要么同时成功，要么同时失败。原子性由undo log日志来保证，因为undo log记载着数据修改前的信息。

候选者: 比如我们要 insert 一条数据了，那undo log 会记录的一条对应的 delete 日志。我们要 update 一条记录时，那undo log会记录之前的「旧值」的update记录。

候选者: 如果执行事务过程中出现异常的情况，那执行「回滚」。InnoDB引擎就是利用undo log记录下的数据，来将数据「恢复」到事务开始之前

**原子性：要么同时成功，要么同时失败
底层依赖 undo log 来实现**

候选者: 一致性我稍稍往后讲，我先来说下隔离性

面试官: 嗯...

候选者: 隔离性指的是：在事务「并发」执行时，他们内部的操作不能互相干扰。如果多个事务可以同时操作一个数据，那么就会产生脏读、重复读、幻读的问题。

候选者: 于是，事务与事务之间需要存在「一定」的隔离。在InnoDB引擎中，定义了四种隔离级别供我们使用：

候选者: 分别是：read uncommit(读未提交)、read commit (读已提交)、repeatable read (可重复读)、serializable (串行)

候选者: 不同的隔离级别对事务之间的隔离性是不一样的（级别越高事务隔离性越好，但性能就越低），而隔离性是由MySQL的各种锁来实现的，只是它屏蔽了加锁的细节。

隔离性：事务之间需要有隔离，互不影响
MySQL提供四种隔离级别给我们使用，隔离级别底层实现是锁

候选者：持久性指的就是：一旦提交了事务，它对数据库的改变就应该是永久性的。说白了就是，会将数据持久化在硬盘上。

候选者：而持久性由redo log 日志来保证，当我们要修改数据时，MySQL是先把这条记录所在的「页」找到，然后把该页加载到内存中，将对应记录进行修改。

候选者：为了防止内存修改完了，MySQL就挂掉了（如果内存改完，直接挂掉，那这次的修改相当于就丢失了）。

候选者：MySQL引入了redo log，内存写完了，然后会写一份redo log，这份redo log记载着这次在某个页上做了什么修改。

候选者：即便MySQL在中途挂了，我们还可以根据redo log来对数据进行恢复。

候选者：redo log 是顺序写的，写入速度很快。并且它记录的是物理修改（xxxx页做了xxx修改），文件的体积很小，恢复速度也很快。

持久性：一旦提交事务，数据是永久记录的
MySQL底层使用的是redo log 来持久化数据

候选者：回头再来讲一致性，「一致性」可以理解为我们使用事务的「目的」，而「隔离性」「原子性」「持久性」均是为了保障「一致性」的手段，保证一致性需要由应用程序代码来保证

候选者：比如，如果事务在发生的过程中，出现了异常情况，此时你就得回滚事务，而不是强行提交事务来导致数据不一致。

一致性：一致性是事务的目的
需要通过应用程序来保证一致性

面试官：嗯，挺好的，讲了蛮多的

面试官：刚才你也提到了隔离性嘛，然后你说在MySQL中有四种隔离级别，能分别来介绍下吗？

候选者：嗯，为了讲清楚隔离级别，我顺带来说下MySQL锁相关的知识吧。

候选者：在InnoDB引擎下，按锁的粒度分类，可以简单分为行锁和表锁。

候选者：行锁实际上是作用在索引之上的（索引上次已经说过了，这里就不赘述了）。当我们的SQL命中了索引，那锁住的就是命中条件内的索引节点（这种就是行锁），如果没有命中索引，那我们锁的就是整个索引树（表锁）。

候选者：简单来说就是：锁住的是整棵树还是某几个节点，完全取决于SQL条件是否有命中到对应的索引节点。

候选者：而行锁又可以简单分为读锁（共享锁、S锁）和写锁（排它锁、X锁）。

候选者：读锁是共享的，多个事务可以同时读取同一个资源，但不允许其他事务修改。写锁是排他的，写锁会阻塞其他的写锁和读锁。

行锁

表锁

读锁

写锁

InnoDB引擎下，命中了索引是行锁，不命中则是表锁

候选者: 我现在就再回到隔离级别上吧，就直接以例子来说明啦。

面试官: 嗯...

候选者: 首先来说下read uncommitt(读未提交)。比如说：A向B转账，A执行了转账语句，但A还没有提交事务，B读取数据，发现自己账户钱变多了！B跟A说，我已经收到钱了。A回滚事务【rollback】，等B再查看账户的钱时，发现钱并没有多。

候选者: 简单的定义就是：事务B读取到了事务A还没提交的数据，这种用专业术语来说叫做「脏读」。

候选者: 对于锁的维度而言，其实就是在read uncommitt隔离级别下，读不会加任何锁，而写会加排他锁。读什么锁都不加，这就让排他锁无法排它了。

read uncommitted 原理

读不加锁

修改加写锁

候选者: 而我们又知道，对于更新操作而言，InnoDB是肯定会加写锁的（数据库是不可能允许在同一时间，更新同一条记录的）。而读操作，如果不加任何锁，那就会造成上面的脏读。

候选者: 脏读在生产环境下肯定是无法接受的，那如果读加锁的话，那意味着：当更新数据的时，就没办法读取了，这会极大地降低数据库性能。

候选者: 在MySQL InnoDB引擎层面，又有新的解决方案（解决加锁后读写性能问题），叫做MVCC(Multi-Version Concurrency Control)多版本并发控制

MVCC：提高读写性能，多版本并发控制

候选者：在MVCC下，就可以做到读写不阻塞，且避免了类似脏读这样的问题。那MVCC是怎么做的呢？

候选者：MVCC通过生成数据快照（Snapshot），并用这个快照来提供一定级别（语句级或事务级）的一致性读取

候选者：回到事务隔离级别下，针对于 read commit (读已提交) 隔离级别，它生成的就是语句级快照，而针对于repeatable read (可重复读)，它生成的就是事务级的快照。

MVCC需要在 read committed 和 repeatable read 隔离级别下奏效

read committed 语句级快照

repeatable read 事务级快照

候选者：前面提到过read uncommit隔离级别下会产生脏读，而read commit (读已提交) 隔离级别解决了脏读。思想其实很简单：在读取的时候生成一个“版本号”，等到其他事务commit了之后，才会读取最新已commit的“版本号”数据。

候选者：比如说：事务A读取了记录(生成版本号)，事务B修改了记录(此时加了写锁)，事务A再读取的时候，是依据最新的版本号来读取的(当事务B执行commit了之后，会生成一个新的版本号)，如果事务B还没有commit，那事务A读取的还是之前版本号的数据。

候选者：通过「版本」的概念，这样就解决了脏读的问题，而「版本」其实就是对应快照的数据。

候选者：read commit (读已提交) 解决了脏读，但也会有其他并发的问题。「不可重复读」：一个事务读取到另外一个事务已经提交的数据，也就是说一个事务可以看到其他事务所做的修改。

候选者: 不可重复读的例子: A查询数据库得到数据, B去修改数据库的数据, 导致A多次查询数据库的结果都不一样 【危害: A每次查询的结果都是受B的影响的】

候选者: 了解MVCC基础之后, 就很容易想到repeatable read (可重复复读)隔离级别是怎么避免不可重复读的问题了 (前面也提到了)。

候选者: repeatable read (可重复复读)隔离级别是「事务级别」的快照! 每次读取的都是「当前事务的版本」, 即使当前数据被其他事务修改了(commit), 也只会读取当前事务版本的数据。

mvcc本质上就是对比 版本

候选者: 而repeatable read (可重复复读)隔离级别会存在幻读的问题, 「幻读」指的是指在一个事务内读取到了别的事务插入的数据, 导致前后读取不一致。

候选者: 在InnoDB引擎下的repeatable read (可重复复读)隔离级别下, 快照读MVCC影响下, 已经解决了幻读的问题 (因为它是读历史版本的数据)

候选者: 而如果是当前读 (指的是 select * from table for update) , 则需要配合间隙锁来解决幻读的问题。

候选者: 剩下的就是serializable (串行)隔离级别了, 它的最高的隔离级别, 相当于不允许事务的并发, 事务与事务之间执行是串行的, 它的效率最低, 但同时也是最安全的。

面试官: 嗯, 可以的。我看你提到了MVCC了, 不妨来说下他的原理?

候选者: MVCC的主要通过read view和undo log来实现的

MVCC 原理

read view (版本信息)

undo log

候选者: undo log前面也提到了，它会记录修改数据之前的信息，事务中的原子性就是通过undo log来实现的。所以，有undo log可以帮我们找到「版本」的数据

候选者: 而read view 实际上就是在查询时，InnoDB会生成一个read view，read view 有几个重要的字段，分别是：trx_ids（尚未提交commit的事务版本号集合），up_limit_id（下一次要生成的事务ID值），low_limit_id（尚未提交版本号的事务ID最小值）以及creator_trx_id（当前的事务版本号）

候选者: 在每行数据有两列隐藏的字段，分别是DB_TRX_ID（记录着当前ID）以及DB_ROLL_PTR（指向上一个版本数据在undo log 里的位置指针）

候选者: 铺垫到这了，很容易就发现，MVCC其实就是靠「比对版本」来实现读写不阻塞，而版本的数据存在于undo log中。

候选者: 而针对于不同的隔离级别（read commit和repeatable read），无非就是read commit隔离级别下，每次都获取一个新的read view，repeatable read隔离级别则每次事务只获取一个read view

面试官: 嗯，OK的。细节就不考究了，今天就到这里吧。

本文总结：

- 事务为了保证数据的最终一致性
- 事务有四大特性，分别是原子性、一致性、隔离性、持久性
 - 原子性由undo log保证
 - 持久性由redo log 保证
 - 隔离性由数据库隔离级别供我们选择，分别有read uncommit,read commit,repeatable read,serializable
 - 一致性是事务的目的，一致性由应用程序来保证
- 事务并发会存在各种问题，分别有脏读、重复读、幻读问题。上面的不同隔离级别可以解决掉由于并发事务所造成的问题，而隔离级别实际上就是由MySQL锁来实现的
- 频繁加锁会导致数据库性能低下，引入了MVCC多版本控制来实现读写不阻塞，提高数据库性能
- MVCC原理即通过read view 以及undo log来实现



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

03、MySQL调优

面试官：要不你来讲讲你们对MySQL是怎么调优的？

候选者：哇，这命题很大阿...我认为，对于开发者而言，对MySQL的调优重点一般是在「开发规范」、「数据库索引」又或者说解决线上慢查询上。

候选者：而对于MySQL内部的参数调优，由专业的DBA来搞。

面试官：扯了这么多，你就是想表达你不会MySQL参数调优，对吧

候选者：草，被发现了。

面试官：那你来聊聊你们平时开发的规范和索引这块，平时是怎么样的吧。

候选者：嗯，首先，我们在生产环境下，创建数据库表，都是在工单系统下完成的（那就自然需要DBA审批）。如果在创建表时检测到没有创建索引，那就会直接提示warning（：

规范上：只要有查询需求，都应该建索引

候选者：理论上来说，如果表有一定的数据量，那就应该要创建对应的索引。从数据库查询数据需要注意的地方还是蛮多的，其中很多都是平时积累来的。比如说：

候选者：1. 是否能使用「覆盖索引」，减少「回表」所消耗的时间。意味着，我们在select的时候，一定要指明对应的列，而不是select *

候选者：2. 考虑是否组建「联合索引」，如果组建「联合索引」，尽量将区分度最高的放在最左边，并且需要考虑「最左匹配原则」

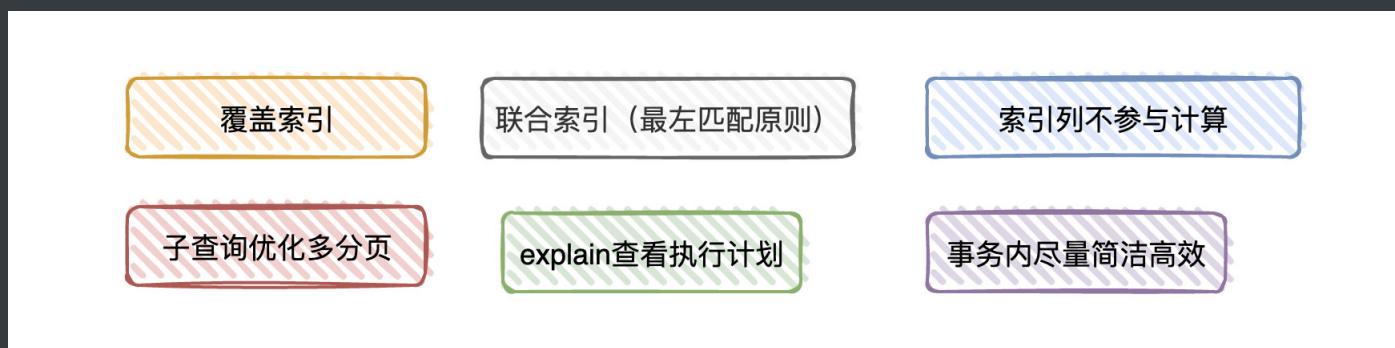
候选者：3. 对索引进行函数操作或者表达式计算会导致索引失效

候选者：4. 利用子查询优化超多分页场景。比如 limit offset , n 在MySQL是获取 offset + n 的记录，再返回n条。而利用子查询则是查出n条，通过ID检索对应的记录出来，提高查询效率。

面试官：嗯...

候选者：5. 通过explain命令来查看SQL的执行计划，看看自己写的SQL是否走了索引，走了什么索引。通过show profile 来查看SQL对系统资源的损耗情况（不过一般还是比较少用到的）

候选者: 6. 在开启事务后，在事务内尽可能只操作数据库，并有意识地减少锁的持有时间（比如在事务内需要插入&&修改数据，那可以先插入后修改。因为修改是更新操作，会加行锁。如果先更新，那并发下可能会导致多个事务的请求等待行锁释放）



面试官: 嗯，你提到了事务，之前也讲过了事务的隔离级别嘛，**那你线上用的是什么隔离级别？**

候选者: 嗯，我们这边用的是Read Commit（读已提交），MySQL默认用的是Repeatable read（可重复读）。选用什么隔离级别，主要看应用场景嘛，因为隔离级别越低，事务并发性能越高。

候选者: (一般互联网公司都选择Read Commit作为主要的隔离级别)

候选者: 像Repeatable read（可重复读）隔离级别，就有可能因为「间隙锁」导致的死锁问题。

候选者: 但可能你已经知道，MySQL默认的隔离级别为Repeatable read。很大一部分原因是在最开始的时候，MySQL的binlog没有row模式，在read commit隔离级别下会存在「主从数据不一致」的问题

候选者: binlog记录了数据库表结构和表数据「变更」，比如update/delete/insert/truncate/create。在MySQL中，主从同步实际上就是应用了binlog来实现的（：

候选者: 有了该历史原因，所以MySQL就将默认的隔离级别设置为Repeatable read

互联网公司一般的隔离级别是 read committed
MySQL默认隔离级别 是由于历史原因导致 (binlog 主从一致性)

面试官：嗯，那我顺便想问下，你们遇到过类似的问题吗：即便走对了索引，线上查询还是慢。

候选者：嗯嗯，当然遇到过了

面试官：那你们是怎么做的？

候选者：如果走对了索引，但查询还是慢，那一般来说就是表的数据量实在是太大了。

候选者：首先，考虑能不能把「旧的数据」给“删掉”，对于我们公司而言，我们都会把数据同步到Hive，说明已经离线存储了一份了。

候选者：那如果「旧的数据」已经没有查询的业务了，那最简单的办法肯定是“删掉”部分数据咯。数据量降低了，那自然，检索速度就快了…

面试官：嗯，但一般不会删的

候选者：没错，只有极少部分业务可以删掉数据（：

候选者：随后，就考虑另一种情况，能不能在查询之前，直接走一层缓存（Redis）。

候选者：而走缓存的话，又要看业务能不能忍受读取的「非真正实时」的数据（毕竟Redis和MySQL的数据一致性需要保证），如果查询条件相对复杂且多变的话（涉及各种group by 和 sum），那走缓存也不是一种好的办法，维护起来就不方便了…

候选者：再看看是不是有「字符串」检索的场景导致查询低效，如果是的话，可以考虑把表的数据导入至Elasticsearch类的搜索引擎，后续的线上查询就直接走Elasticsearch了。

候选者：MySQL->Elasticsearch需要有对应的同步程序(一般就是监听MySQL的binlog，解析binlog后导入到Elasticsearch)

候选者：如果还不是的话，那考虑要不要根据查询条件的维度，做相对应的聚合表，线上的请求就查询聚合表的数据，不走原表。

候选者：比如，用户下单后，有一份订单明细，而订单明细表的量级太大。但在产品侧(前台)透出的查询功能是以「天」维度来展示的，那就可以将每个用户的每天数据聚合起来，在聚合表就是一个用户一天只有一条汇总后的数据。

候选者：查询走聚合后的表，那速度肯定杠杠的（聚合后的表数据量肯定比原始表要少很多）

候选者：思路大致的就是「以空间换时间」，相同的数据换别的地方也存储一份，提高查询效率



面试官：那我还想问下，除了读之外，写性能同样有瓶颈，怎么办？

候选者：你说到了这个，我就不困了。

候选者：如果在MySQL读写都有瓶颈，那首先看下目前MySQL的架构是怎么样的。

候选者：如果是单库的，那是不可以考虑升级至主从架构，实现读写分离。

候选者：简单理解就是：主库接收写请求，从库接收读请求。从库的数据由主库发送的binlog进而更新，实现主从数据一致（在一般场景下，主从的数据是通过异步来保证最终一致性）

MySQL 主从架构 (读写分离)

面试官：嗯...

候选者：如果在主从架构下，读写仍存在瓶颈，那就要考虑是否要分库分表了

候选者：至少在我前公司的架构下，业务是区分的。流量有流量数据库，广告有广告的数据
库，商品有商品的数据库。所以，我这里讲的分库分表的含义是：在原来的某个库的某个表
进而拆分。

候选者：比如，现在我有一张业务订单表，这张订单表在广告库中，假定这张业务订单表已
经有1亿数据量了，现在我要分库分表

候选者：那就会将这张表的数据分至多个广告库以及多张表中（：

候选者：分库分表的最明显的好处就是把请求进行均摊（本来单个库单个表有一亿的数据，
那假设我分开8个库，那每个库1200+W的数据量，每个库下分8张表，那每张表就150W的数
据量）。

分库分表
(将数据量打散至多库和多表)

面试官：你们是以什么来作为分库键的？

候选者：按照我们这边的经验，一般来说是按照userId的（因为按照用户的维度查询比较多），如果要按照其他的维度进行查询，那还是参照上面的思路（以空间换时间）。

面试官：那分库分表后的ID是怎么生成的？

候选者：这就涉及到分布式ID生成的方式了，思路有很多。有借助MySQL自增的，有借助Redis自增的，有基于「雪花算法」自增的。具体使用哪种方式，那就看公司的技术栈了，一般使用Redis和基于「雪花算法」实现用得比较多。

候选者：至于为什么强调自增（还是跟索引是有序有关，前面已经讲过了，你应该还记得）



面试官：嗯，那如果我要分库分表了，迁移的过程是怎么样的呢

候选者：我们一般采取「双写」的方式来进行迁移，大致步骤就是：

候选者：一、增量的消息各自往新表和旧表写一份

候选者：二、将旧表的数据迁移至新库

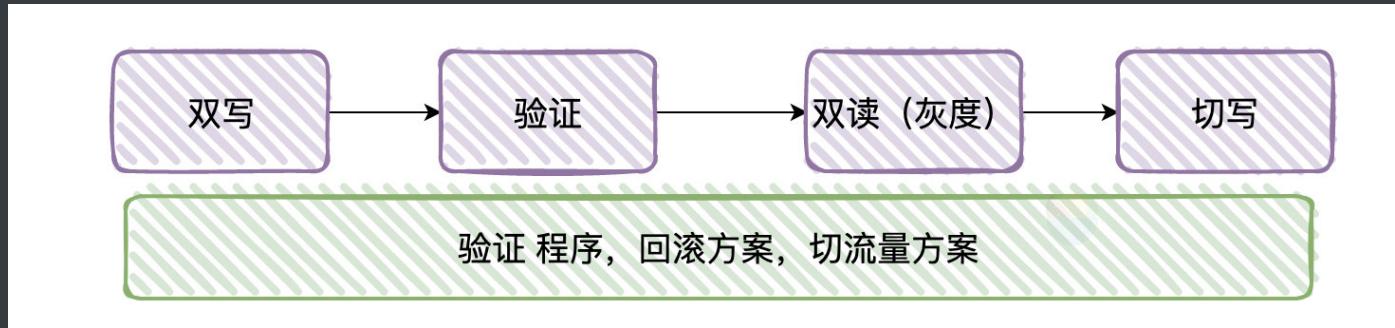
候选者：三、迟早新表的数据都会追得上旧表（在某个节点上数据是同步的）

候选者：四、校验新表和老表的数据是否正常（主要看能不能对得上）

候选者：五、开启双读（一部分流量走新表，一部分流量走老表），相当于灰度上线的过程

候选者：六、读流量全部切新表，停止老表的写入

候选者：七、提前准备回滚机制，临时切换失败能恢复正常业务以及有修数据的相关程序。



面试官：嗯...今天就到这吧

本文总结：

- 数据库表存在一定数据量，就需要有对应的索引
- 发现慢查询时，检查是否走对索引，是否能用更好的索引进行优化查询速度，查看使用索引的姿势有没有问题
- 当索引解决不了慢查询时，一般由于业务表的数据量太大导致，利用空间换时间的思想
- 当读写性能均遇到瓶颈时，先考虑能否升级数据库架构即可解决问题，若不能则需要考虑分库分表
- 分库分表虽然能解决掉读写瓶颈，但同时会带来各种问题，需要提前调研解决方案和踩坑

线上不是给你炫技的地方，安稳才是硬道理。能用简单的方式去解决，不要用复杂的方式



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



09-项目场景类

01、如何实现去重和幂等

面试官：要不你来讲讲你最近在看的点呗？可以拉出来一起讨论下

候选者：最近在看「去重」和「幂等」相关的内容

面试官：那你就先来聊聊你对「去重」和「幂等」的理解吧

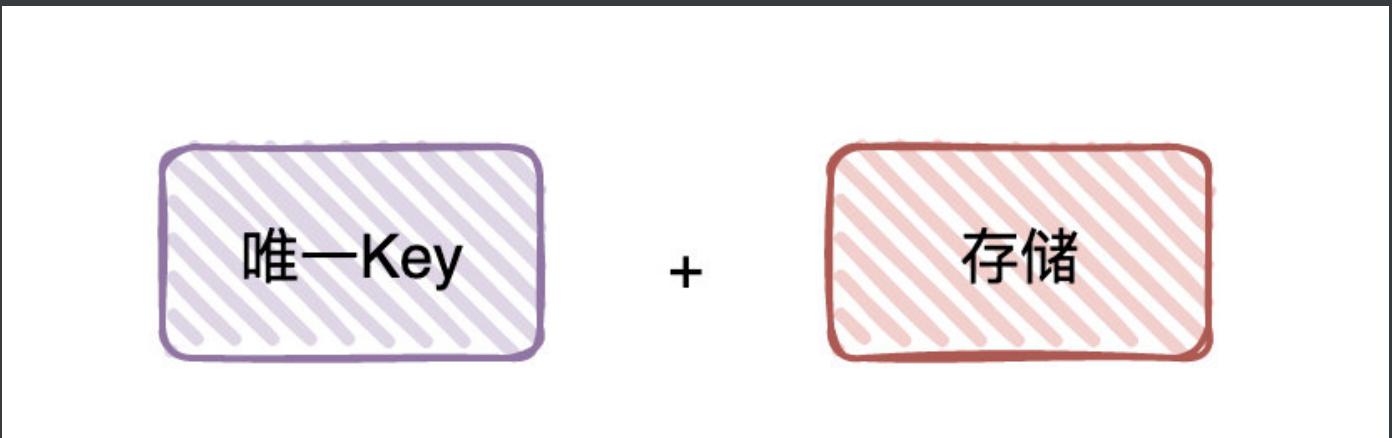
候选者：我认为「幂等」和「去重」它们很像，我也说不出他们之间的严格区别

候选者：我说下我个人的理解，我也不知道对不对

候选者：「去重」是对请求或者消息在「一定时间内」进行去重「N次」

候选者：「幂等」则是保证请求或消息在「任意时间内」进行处理，都需要保证它的结果是一致的

候选者：不论是「去重」还是「幂等」，都需要对有一个「唯一 Key」，并且有地方对唯一 Key 进行「存储」



候选者：以项目举例，我维护的「消息管理平台」是有「去重」的功能的：「5分钟相同内容消息去重」「1小时内模板去重」「一天内渠道达到N次阈值去重」...

候选者：再次强调下「幂等」和「去重」的本质：「唯一Key」 + 「存储」

面试官：那你是怎么做的呢

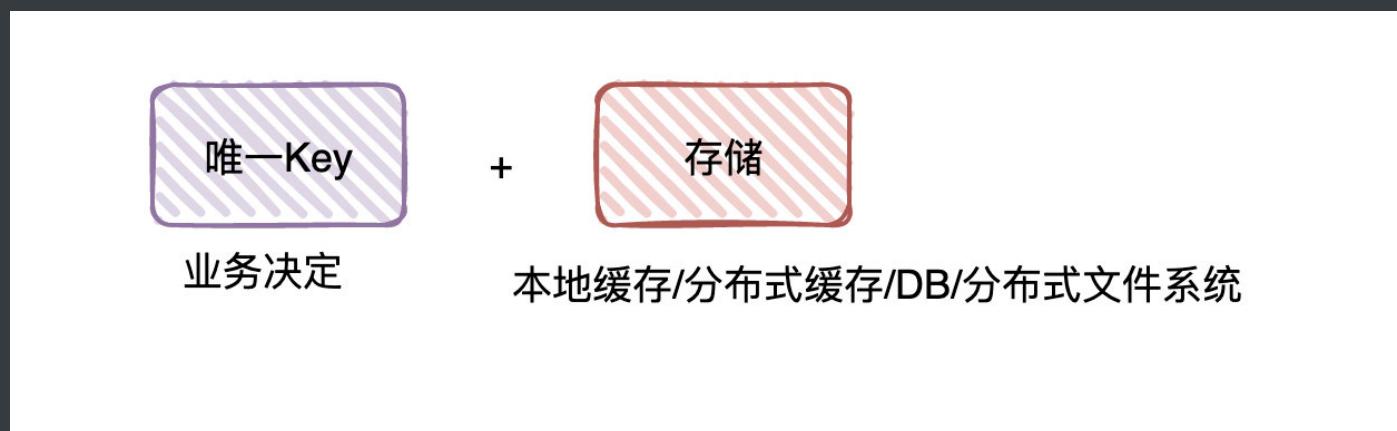
候选者：不同的业务场景，唯一Key是不一样的，由业务决定

候选者：存储选择挺多的，比如「本地缓存」 / 「Redis」 / 「MySQL」 / 「HBase」等等，具体选取什么，也跟业务有关

候选者：比如说，在「消息管理平台」这个场景下，我存储选择的「Redis」（读写性能优越），Redis也有「过期时间」方便解决「一定时间内」的问题

候选者：而唯一Key，自然就是根据不同的业务构建不同的。

候选者：比如说「5分钟相同内容消息去重」，我直接MD5请求参数作为唯一Key。「1小时模板去重」则是「模板ID+userId」作为唯一Key，「一天内渠道去重」则是「渠道ID+userId」作为唯一Key...



面试官：既然提到了「去重」了，你听过布隆过滤器吗？

候选者：自然是知道的啦

面试官：来讲讲布隆过滤器吧，你为什么不用呢？

候选者：布隆过滤器的底层数据结构可以理解为bitmap，bitmap也可以简单理解为是一个数组，元素只存储0和1，所以它占用的空间相对较小

候选者: 当一个元素要存入bitmap时，其实是要去看存储到bitmap的那个位置，这时一般用的就是哈希算法，存进去的位置标记为1

候选者: 标记为1的位置表示存在，标记为0的位置表示不存在

bitmap 标记 (占用空间小)

候选者: 布隆过滤器是可以以较低的空间占用来判断元素是否存在进而用于去重，但是它也有对应的缺点

候选者: 只要使用哈希算法离不开「哈希冲突」，导致有存在「误判」的情况

候选者: 在布隆过滤器中，如果元素判定为存在，那该元素「未必」真实存在。如果元素判定为不存在，那就肯定是不存在

候选者: 这应该不用我多解释了吧？（结合「哈希算法」和「标记为1的位置表示存在，标记为0的位置表示不存在」这两者就能得出上面结论）

候选者: 布隆过滤器也不能「删除」元素（也是哈希算法的局限性，在布隆过滤器中是不能准确定位一个元素的）

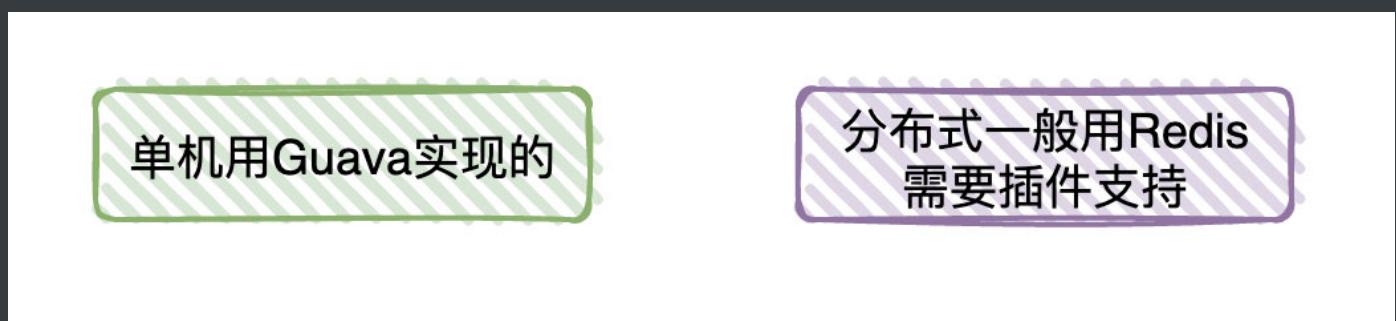
不能删除

存在误判 (hash冲突)

候选者: 如果要用的话，布隆过滤器的实现可以直接上Guava已经实现好的，不过这个是单机的

候选者：而分布式下的布隆过滤器，一般现在会用Redis，但也不是没个公司都会部署布隆过滤器的Redis版（还是有局限，像我以前公司就没有）

候选者：所以，目前我负责的项目都是没有用布隆过滤器的（：



候选者：如果「去重」开销比较大，可以考虑建立「多层过滤」的逻辑

候选者：比如，先看看『本地缓存』能不能过滤一部分，剩下「强校验」交由『远程存储』（常见的Redis或者DB）进行二次过滤

面试官：嗯，那我就想起你上一次回答Kafka的时候了

面试官：当时你说在处理订单时实现了at least one + 幂等

面试官：幂等处理时：前置过滤使用的是Redis，强一致校验时使用的是DB唯一索引，也是为了提高性能，对吧？

面试官：唯一Key 好像就是「订单编号 + 订单状态」

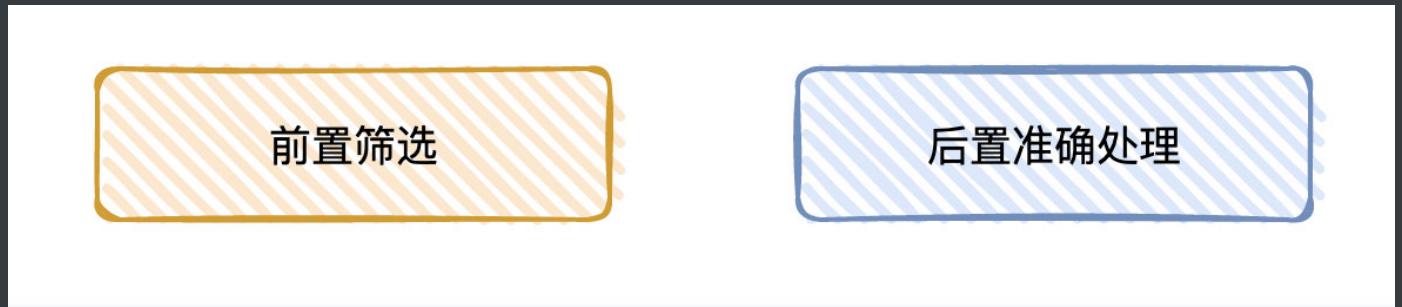
候选者：面试官你记性真的好！

候选者：一般我们需要对数据强一致性校验，就直接上MySQL（DB），毕竟有事务的支持

候选者：「本地缓存」如果业务适合，那可以作为一个「前置」判断

候选者：Redis高性能读写，前置判断和后置均可（：

候选者: 而HBase则一般用于庞大数据量的场景下 (Redis内存太贵, DB不够灵活也不适合单表存大量数据)



候选者: 至于幂等, 一般的存储还是「Redis」和「数据库」

候选者: 最最最常见的就是数据库「唯一索引」来实现幂等 (我所负责的好几个项目都是用这个)

候选者: 构建「唯一Key」是业务相关的事了 (: 一般是用自己的业务ID进行拼接, 生成一个"有意义"的唯一Key)

候选者: 当然, 也有用「Redis」和「MySQL」实现分布式锁来实现幂等的 (:

候选者: 但Redis分布式锁是不能完全保证安全的, 而MySQL实现分布式锁 (乐观锁和悲观锁还是看业务吧, 我是没用到过的)

候选者: 网上有很多实现「幂等」的方案, 本质上都是围绕着「存储」和「唯一Key」做了些变种, 然后取了个名字...

候选者: 总的来说, 换汤不换药 (:

面试官: �恩...了解了



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zhongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

02、系统需求多变如何设计

面试官：我想问个问题哈，项目里比较常见的问题

面试官：我现在有个系统会根据请求的入参，做出不同动作。但是，这块不同的动作很有可能是会发生需求变动的，这块系统你会怎么样设计？

面试官：实际的例子：现在有多个第三方渠道，系统需要对各种渠道进行订单归因。但是归因的逻辑很有可能会发生变化，不同的渠道归因的逻辑也不太一样，此时系统里的逻辑相对比较复杂。

面试官：如果让你优化，你会怎么设计？

候选者：我理解你的意思了

候选者：归根到底，就是处理的逻辑相对复杂，`if else`的判断太多了

候选者：虽然新的需求来了，都可以添加`if else`进行解决

候选者：但你想要的就是，系统的可扩展性和可维护性更强

候选者：想要我这边出一个方案，来解决类似的问题

候选者：对吧？

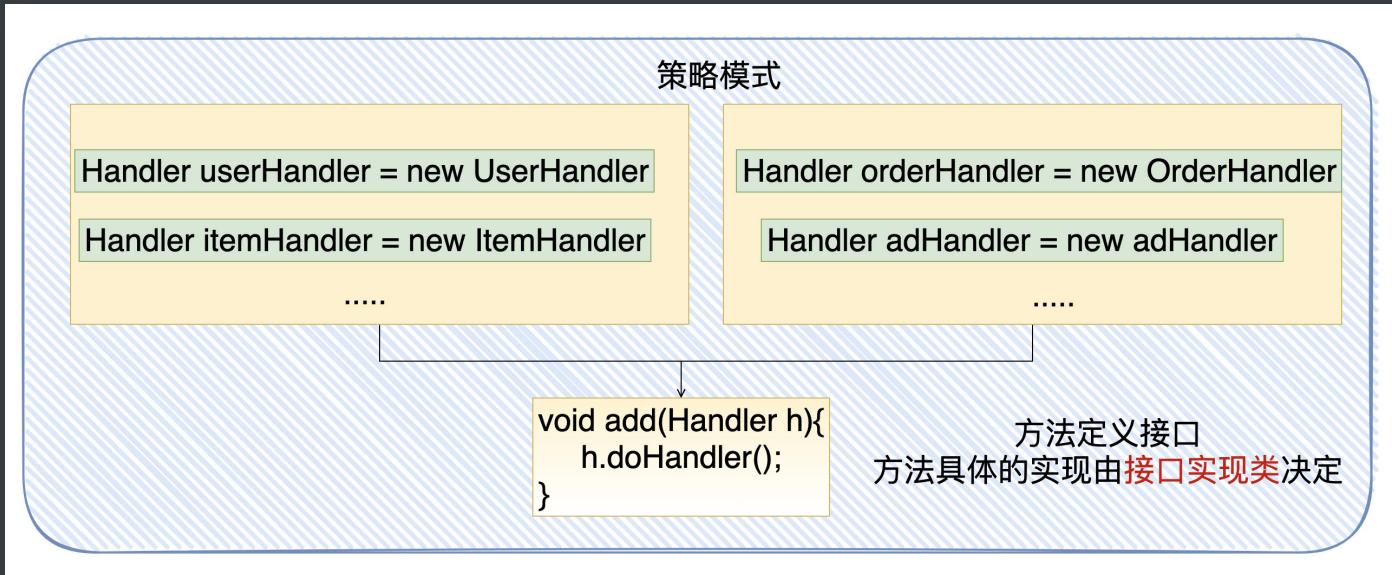
面试官：嗯...

候选者：在这之前，一般上网搜如何解决`if else`，大多数都说是策略模式

候选者：但是举的例子又没感同身受，很多时候看完就过去了

候选者：实际上，在项目里边，用策略模式还是蛮多的，可能无意间就已经用上了（毕竟面向接口编程嘛）

候选者：而我认为，策略模式不是解决`if else`的关键



候选者：这个问题，我的项目里的做法是：责任链模式

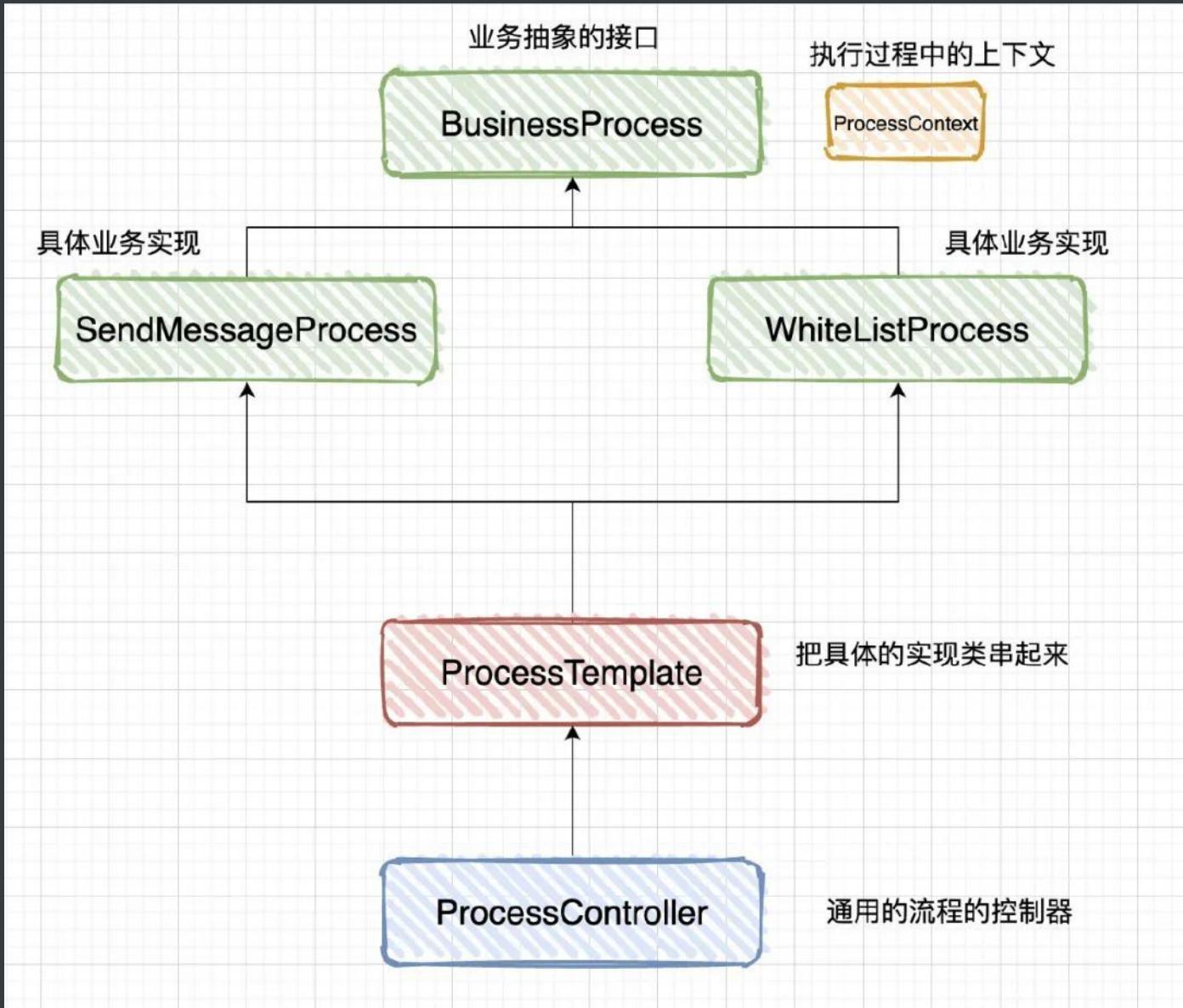
候选者：把每个流程单独抽取成一个Process(可以理解为一个模块或节点)，然后请求都会塞进Context中

候选者：比如，之前维护过一个项目，也是类似于不同的渠道走不同的逻辑

候选者：我们这边的做法是：抽取相关的逻辑到Process中，为不同的渠道分配不同的责任链

候选者：比如渠道A的责任链是： WhiteListProcess->DataAssembleProcess->ChannelAProcess->SendProcess

候选者：而渠道B的责任链是： WhiteListProcess->DataAssembleProcess->ChannelBProcess->SendProcess



候选者: 在责任链基础之上，又可以在代码里内嵌「脚本」

候选者: 比如在SendProcess上，内置发送消息的脚本（脚本可以选择不同的运营商进行发送消息）。有了「脚本」以后，那就可以做到对逻辑的改动不需要重启就可以生效。

候选者: 有人把这一套东西叫做「规则引擎」。比如，规则引擎中比较出名的实现框架「Drools」就可以做到类似的事

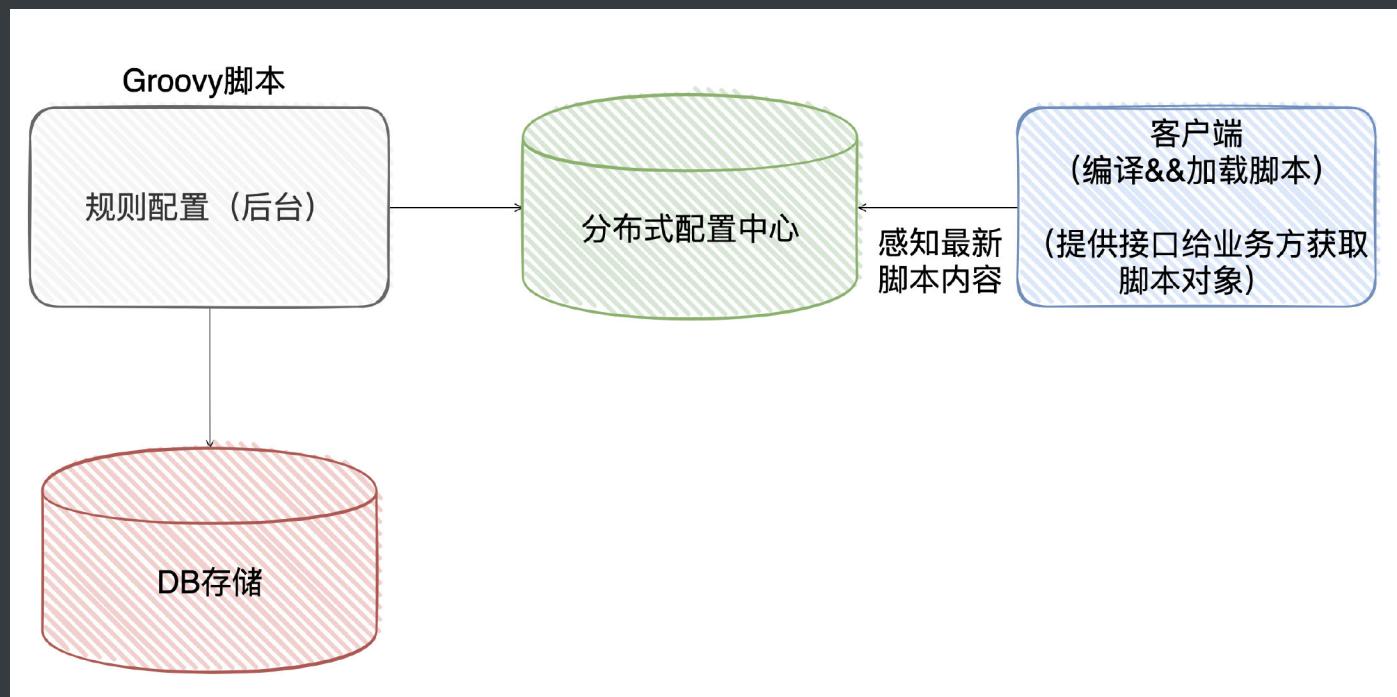
候选者: 把易改动的逻辑写在「脚本」上（至少我们认为，脚本和我们的应用真实逻辑是分离）

候选者: （脚本我这里指的是规则集，它可以是Drools的dsl，也可以是Groovy，也可以是aviator等等）

面试官：嗯...

候选者：在我之前的公司，使用的是Groovy脚本。大致的实现逻辑就是：有专门后台对脚本进行管理，然后会把脚本写到「分布式配置中心」（实时刷新），客户端监听「分布式配置中心」所存储的脚本是否有改动

候选者：如果存在改动，则通过Groovy类加载器重新编译并加载脚本，最后放到Spring容器对外使用



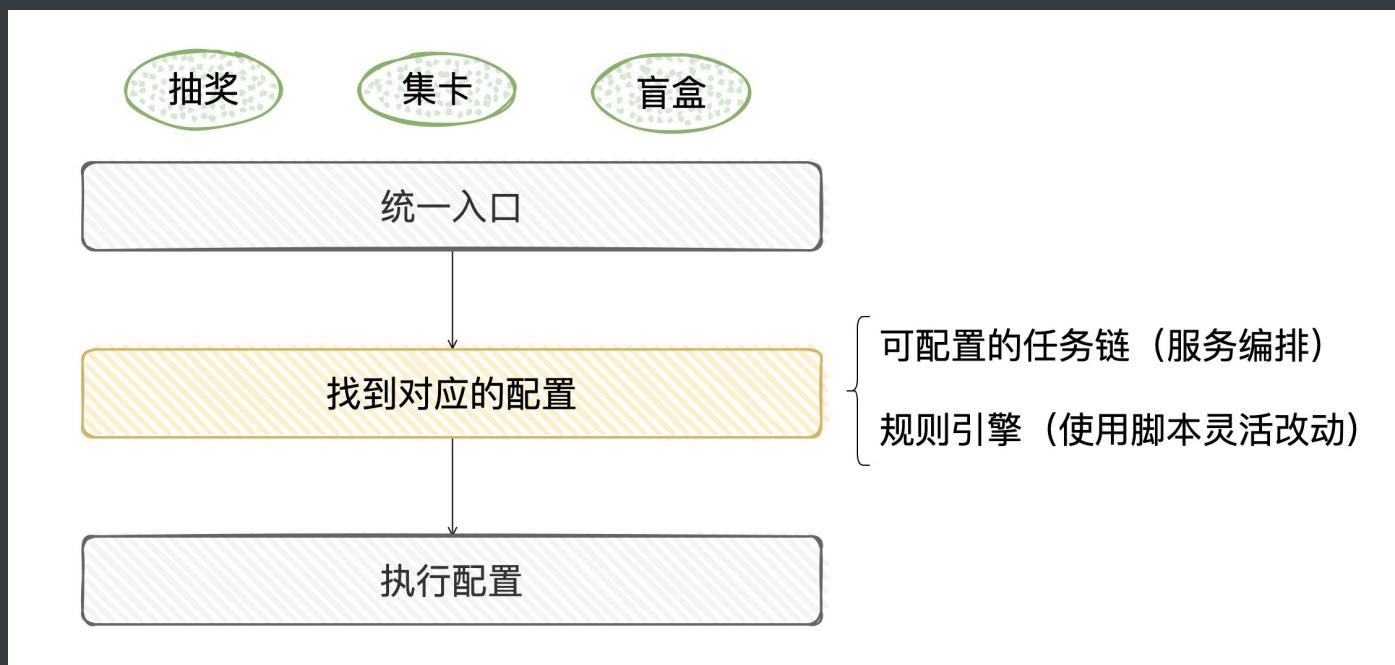
候选者：我目前所负责的系统就是这样处理 多变 以及需求变更频繁的业务（责任链+规则引擎）

候选者：不过据我了解，我们的玩法业务又在「责任链」多做了些事情

候选者：「责任链」不再从代码里编写，而是下沉到平台去做「服务编排」，就是由程序员去「服务编排后台」上配置信息（配置责任链的每一个节点）

候选者：在业务系统里使用「服务编排」的客户端，请求时只要传入「服务编排」的ID，就可以按「服务编排」的流程执行代码

候选者: 这样做的好处就是：业务链是在后台配置的，不用在系统业务上维护链，灵活性更高（写好的责任链节点可以随意组合）



面试官：那我懂了



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

03、设计模式

面试官：我看你的简历写着熟悉常见的设计模式，要不你来简单聊聊你熟悉哪几个吧？

候选者：常见的工厂模式、代理模式、模板方法模式、责任链模式、单例模式、包装设计模式、策略模式等都是有所了解的

候选者：项目手写代码用得比较多的，一般就模板方法模式、责任链模式、策略模式、单例模式吧

候选者：像工厂模式、代理模式这种，手写倒是不多，但毕竟Java后端一般环境下都用Spring嘛，所以还是比较熟悉的。

面试官：要不你来手写下单例模式呗？

候选者：单例模式一般会有好几种写法

候选者：饿汉式、简单懒汉式（在方法声明时加锁）、DCL双重检验加锁（进阶懒汉式）、静态内部类（优雅懒汉式）、枚举

候选者：所谓「饿汉式」指的就是还没被用到，就直接初始化了对象。所谓「懒汉式」指的就是等用到的时候，才进行初始化

候选者：那我就都写写吧，反正就那些代码

```
// 饿汉式
class Singleton {
    private static Singleton singleton = new Singleton();
    private Singleton(){}
    public static Singleton getInstance() {
        return singleton;
    }
}

// 简单懒汉式（方法加锁）
class Singleton {
    private static Singleton singleton = null;
```

```
private Singleton(){}  
}  
public static synchronized Singleton getInstance() {  
    if (singleton == null) {  
        singleton = new Singleton();  
    }  
    return singleton;  
}  
  
}  
  
// DCL懒汉式  
class Singleton {  
    // 这里一定要volatile修饰，防止指令重排导致线程不安全的问题  
    private static volatile Singleton singleton = null;  
  
    private Singleton(){  
    }  
    public static Singleton getInstance() {  
        if (singleton == null) {  
            synchronized (Singleton.class) {  
                if (singleton == null) {  
                    singleton = new Singleton();  
                }  
            }  
        }  
        return singleton;  
    }  
}  
  
}  
  
// 静态内部类 懒汉式  
class Singleton {  
  
    private Singleton(){  
    }  
  
    // 使用静态内部类的方式来实现懒加载(一定线程安全)  
    private static class LazyHolder {  
        private static final Singleton singleton = new Singleton();  
    }  
  
    public static final Singleton getInstance() {  
        return LazyHolder.singleton;  
    }  
}  
  
}  
  
// 枚举  
enum Singleton{  
    SINGLETON,  
}
```

面试官：那你们用的哪种比较多？

候选者：一般我们项目里用静态内部类的方式实现单例会比较多（如果没有Spring的环境下），代码简洁易读

候选者：如果有Spring环境，那还是直接交由Spring容器管理会比较方便（Spring默认就是单例的）

候选者：枚举一般我们就用它来做「标识」吧，而DCL这种方式也有同学会在项目里写（在一些源码里也能看到其身影），但总体太不利于阅读和理解（：

候选者：总的来说，用哪一种都可以的，关键我觉得要看团队的代码风格吧（保持一致就行），即便都用「饿汉式」也没啥大的问题（现在内存也没那么稀缺，我认为可读性比较重要）

面试官：嗯...

面试官：我看你在DCL的单例代码上，写了volatile修饰嘛？为什么呢？

候选者：你不记得我们曾经聊过volatile的了吗？指令是有可能乱序执行的（编译器优化导致乱序、CPU缓存架构导致乱序、CPU原生重排导致乱序）

候选者：在代码new Object的时候，不是一条原子的指令，它会由几个步骤组成，在这过程中，就可能会发生指令重排的问题，而volatile这个关键字就可以避免指令重排的发生。

面试官：那你说下你在项目里用到的设计模式吧？

候选者：嗯，比如说，我这边在处理请求的时候，会用到责任链模式进行处理（减免if else 并且让项目结构更加清晰）

```
● ● ●
```

```
// 责任链模式
<bean name="defaultPipeline" class="rye.apiService.core.pipeline.impl.DefaultPipeline">
    <constructor-arg>
        <list>
            <ref bean="parameterValve"/>
            <ref bean="rateValve"/>
            <ref bean="senderValve"/>
        </list>
    </constructor-arg>
</bean>
```

候选者：在处理公共逻辑时，会使用模板方法模式进行抽象，具体不同的逻辑会由不同的实现类处理（每种消息发送前都需要经过文案校验，所以可以把文案校验的逻辑写在抽象类上）

```
● ● ●
```

```
// 模板方法模式
public boolean handle(Task task) {

    // messyCodeCheck 是统一逻辑处理方法
    if (messyCodeCheck(task)) {
        return doHandle(task);
    } else {
        return false;
    }
}

// doHandle由子类实现
public abstract boolean doHandle(Task task);
```

候选者: 代理模式手写的机会比较少（因为项目一般有Spring环境，直接用Spring 的AOP代理就好了）

候选者: 我之前使用过AOP把「监控客户端」封装以「注解」的方式进行使用（不用以硬编码的方式来进行监控，只要有注解就行了）

```
/*
 * 原本的使用「监控告警」的姿势
 */
public void send(String userName) {
    try {
        // qps 上报
        qps(params);
        long startTime = System.currentTimeMillis();

        // 构建上下文(模拟业务代码)
        ProcessContext processContext = new ProcessContext();
        UserModel userModel = new UserModel();
        userModel.setAge("22");
        userModel.setName(userName);
        //...

        // rt 上报
        long endTime = System.currentTimeMillis();
        rt(endTime - startTime);
    } catch (Exception e) {

        // 出错上报
        error(params);
    }
}

-----
/** 
 * qps上报、rt上报、出错上报
 * 使用【代理模式】Spring AOP 进行抽取
 * 业务上只要方法上存在@Monitor 注解，即可有监控的功能
 */
@Around("@annotation(com.sanwai.service.openapi.monitor.Monitor)")
public Object antispan(ProceedingJoinPoint pjp) throws Throwable {

    String functionName = pjp.getSignature().getName();
    Map<String, String> tags = new HashMap<>();

    logger.info(functionName);

    tags.put("functionName", functionName);
}
```

```
tags.put("functionName", functionName),  
tags.put("flag", "done");  
  
monitor.sum(functionName, "start", 1);  
  
//方法执行开始时间  
long startTime = System.currentTimeMillis();  
  
Object o = null;  
try {  
    o = pjp.proceed();  
} catch (Exception e) {  
    //方法执行结束时间  
    long endTime = System.currentTimeMillis();  
  
    tags.put("flag", "fail");  
    monitor.avg("rt", tags, endTime - startTime);  
  
    monitor.sum(functionName, "fail", 1);  
    throw e;  
}  
  
//方法执行结束时间  
long endTime = System.currentTimeMillis();  
  
monitor.avg("rt", tags, endTime - startTime);  
  
if (null != o) {  
    monitor.sum(functionName, "done", 1);  
}  
return o;  
}
```

面试官：那你能聊聊Spring常见的设计模式嘛？

候选者：比如，Spring IOC容器可以理解为应用了「工厂模式」（通过ApplicationContext或者BeanFactory去获取对象）

候选者：Spring的对象默认都是单例的，所以肯定是用了「单例模式」（源码里对单例的实现是用的DCL来实现单例）

候选者：Spring AOP的底层原理就是用了「代理模式」，实现可能是JDK 动态代理，也可能是CGLIB动态代理

候选者：Spring有很多地方都用了「模板方法模式」，比如事务管理器（AbstractPlatformTransactionManager），getTransaction定义了框架，其中很多都由子类实现

候选者：Spring的事件驱动模型用了「观察者模式」，具体实现就是 ApplicationContextEvent、ApplicationListener

面试官：嗯，了解...



第一时间获取**BATJTMD一线互联网大厂**最新的面试资料以及内推机会关注公众号「**对线面试官**」



10-计算机网络

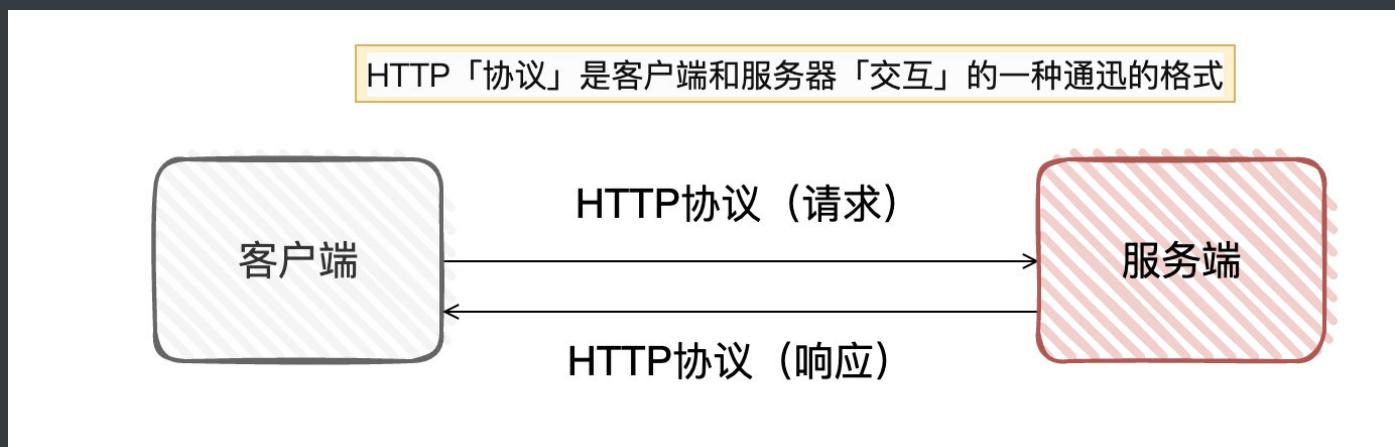
01、HTTP

面试官：今天要不来聊聊HTTP吧？

候选者：嗯，HTTP「协议」是客户端和服务器「交互」的一种通迅的格式

候选者：所谓的「协议」实际上就是双方约定好的「格式」，让双方都能看得懂的东西而已

候选者：所谓的交互实际上就是「请求」和「响应」



面试官：那你知道HTTP各个版本之间的区别吗？

候选者：HTTP1.0默认是短连接，每次与服务器交互，都需要新开一个连接

候选者：HTTP1.1版本最主要的是「默认持久连接」。只要客户端服务端没有断开TCP连接，就一直保持连接，可以发送多次HTTP请求。

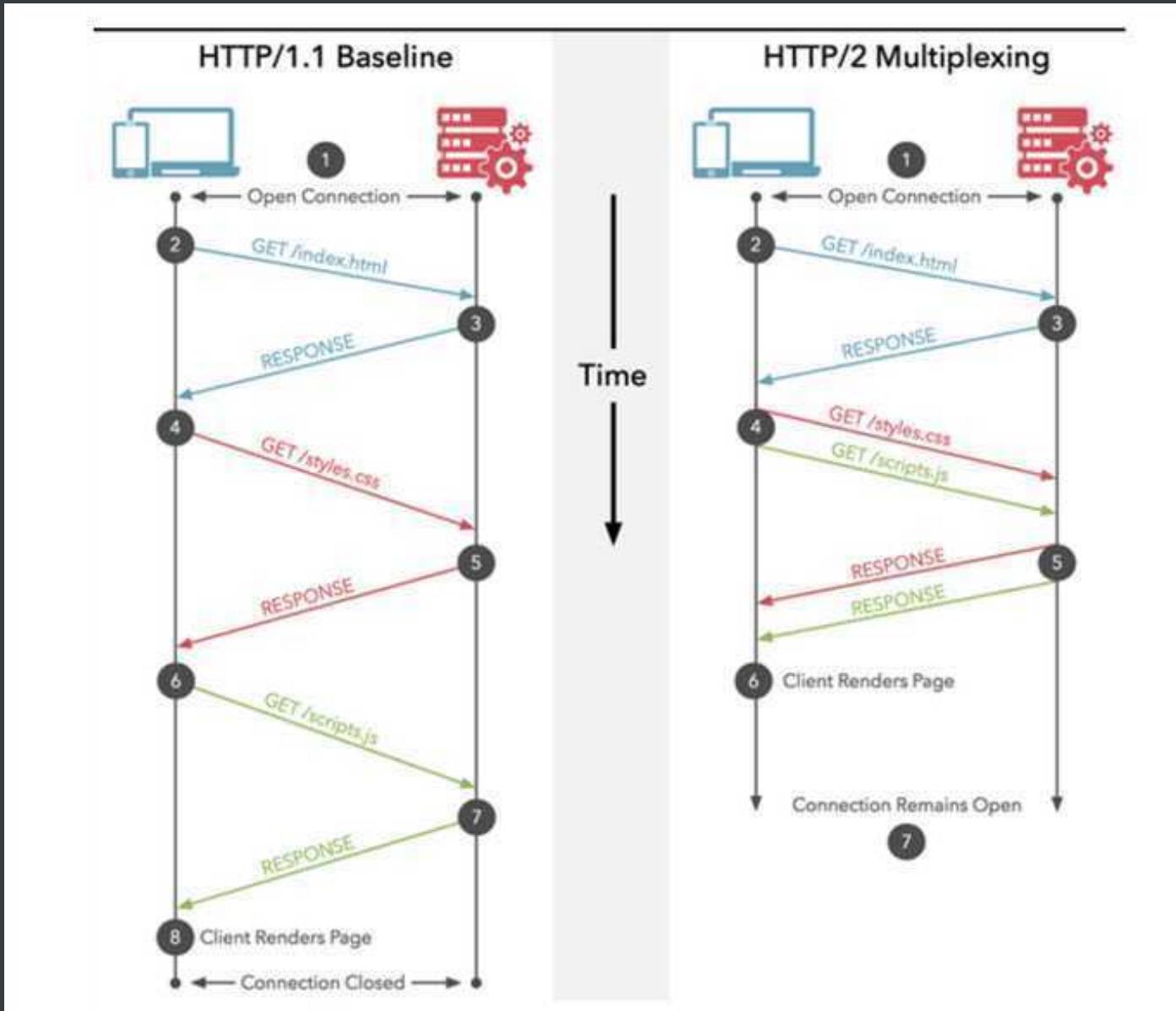
候选者：其次就是「断点续传」（Chunked transfer-coding）。利用HTTP消息头使用分块传输编码，将实体主体分块进行传输

候选者：HTTP/2不再以文本的方式传输，采用「二进制分帧层」，对头部进行了「压缩」，支持「流控」，最主要就是HTTP/2是支持「多路复用」的（通过单一的TCP连接「并行」发起多个的请求和响应消息）

面试官：嗯，稍微打断下。我知道HTTP1.1版本有个管线化(pipelining)理论，但默认是关闭的。管线化这个跟HTTP/2的「多路复用」是很类似的，它们有什么区别呀？

候选者：HTTP1.1提出的「管线化」只能「串行」（一个响应必须完全返回后，下一个请求才会开始传输）

候选者：HTTP/2多路复用则是利用「分帧」数据流，把HTTP协议分解为「互不依赖」的帧（为每个帧「标序」发送，接收回来的时候按序重组），进而可以「乱序」发送避免「一定程度上」的队首阻塞问题



候选者：但是，无论是HTTP1.1还是HTTP/2，response响应的「处理顺序」总是需要跟request请求顺序保持一致的。假如某个请求的response响应慢了，还是同样会有阻塞的问题。

候选者：这受限于HTTP底层的传输协议是TCP，没办法完全解决「线头阻塞」的问题

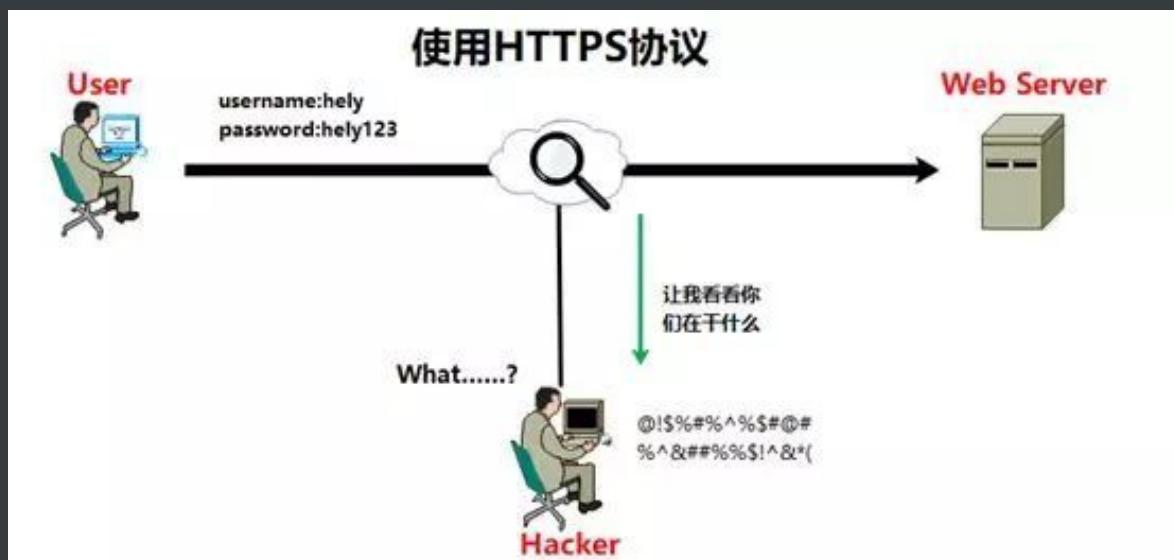
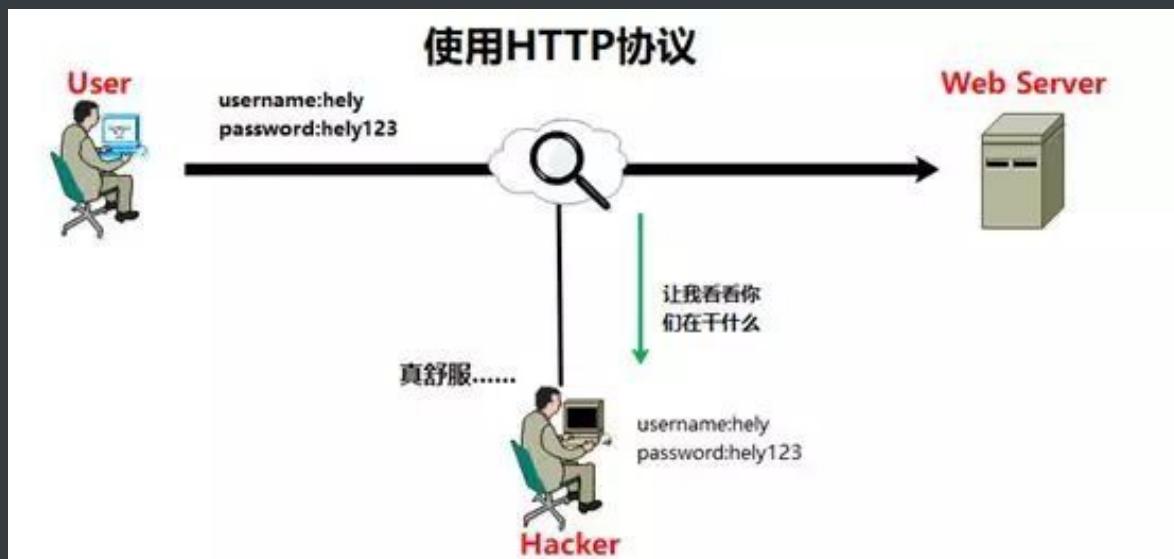
面试官：哦，好的。

候选者：HTTP/3 跟前面版本最大的区别就是：HTTP1.x和HTTP/2底层都是TCP，而HTTP/3底层是UDP。使用HTTP/3能够减少RTT「往返时延」（TCP三次握手，TLS握手）



面试官：你了解HTTPS的过程吗？

候选者：嗯啊， HTTPS在我的理解下，就是「安全」的HTTP协议（客户端与服务端的传输链路中进行加密）



候选者： HTTPS首先要解决的是：认证的问题

候选者： 客户端是需要确切地知道服务端是不是「真实」，所以在HTTPS中会有一个角色：CA（公信机构）

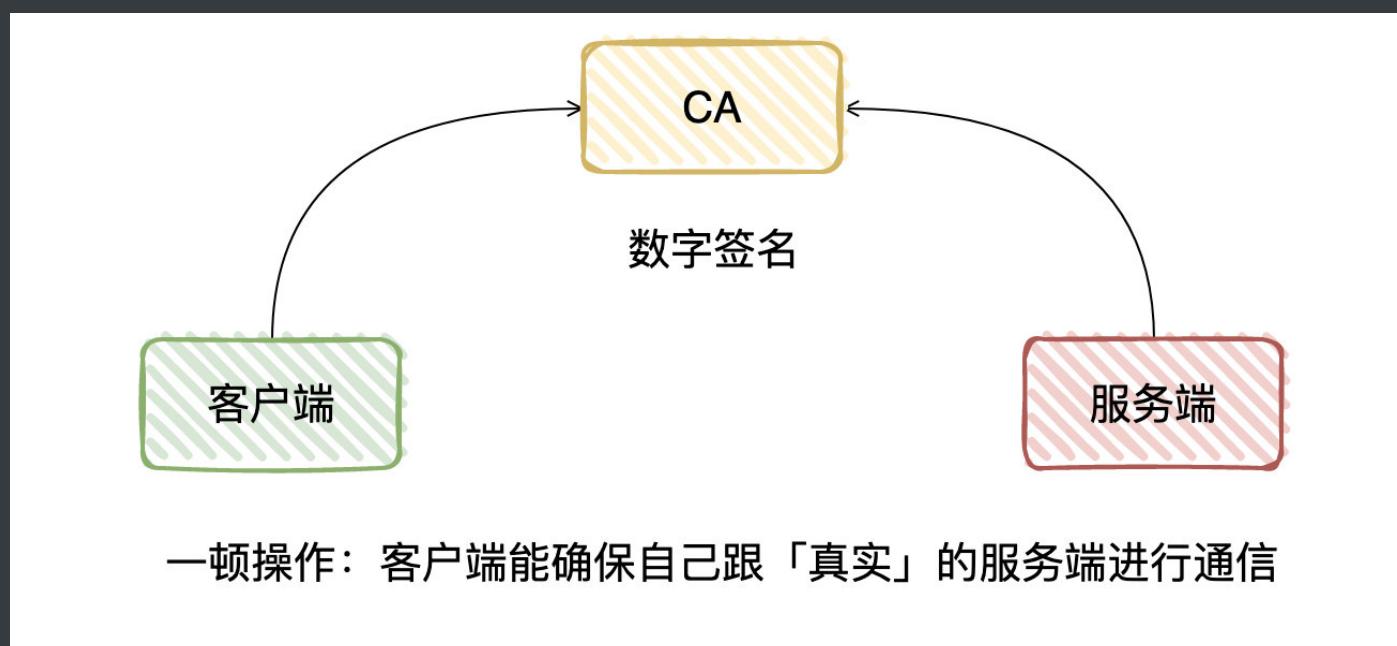
候选者： 服务端在使用HTTPS前，需要去认证的CA机构申请一份「数字证书」。数字证书里包含有证书持有者、证书有效期、「服务器公钥」等信息

候选者： CA机构也有自己的一份公私钥，在发布数字证书之前，会用自己的「私钥」对这份数字证书进行加密

候选者： 等到客户端请求服务器的时候，服务端返回证书给客户端。客户端用CA的公钥对证书解密（因为CA是公信机构，会内置到浏览器或操作系统中，所以客户端会有公钥）。这个时候，客户端会判断这个「证书是否可信/有无被篡改」

候选者： 私钥加密，公钥解密我们叫做「数字签名」（这种方式可以查看有无被篡改）

候选者： 到这里，就解决了「认证」的问题，至少客户端能保证是在跟「真实的服务器」进行通信。



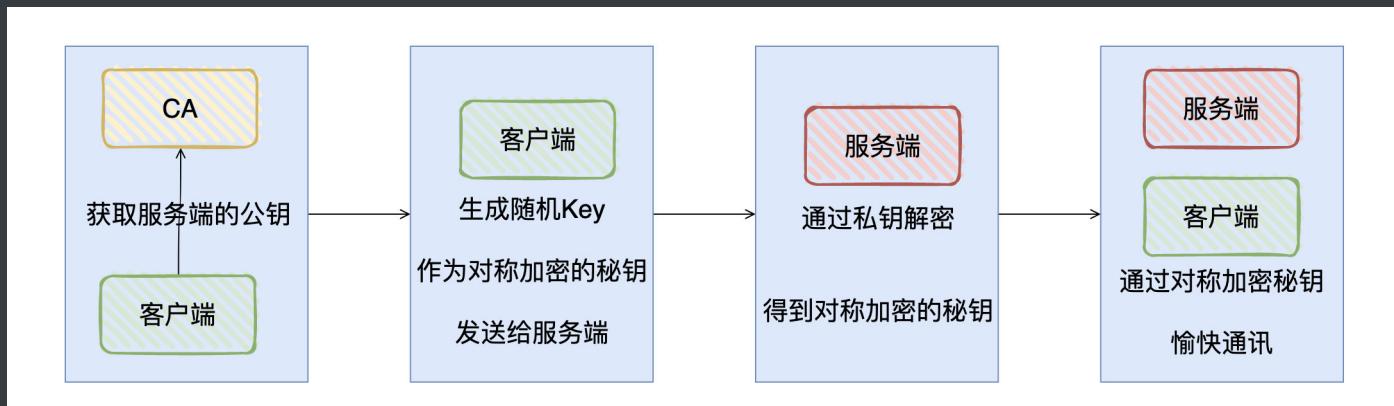
候选者： 解决了「认证」的问题之后，就要解决「保密」问题，客户端与服务器的通讯内容在传输中不会泄露给第三方

候选者： 客户端从CA拿到数字证书后，就能拿到服务端的公钥

候选者: 客户端生成一个Key作为「对称加密」的秘钥，用服务端的「公钥加密」传给服务端

候选者: 服务端用自己的「私钥解密」客户端的数据，得到对称加密的秘钥

候选者: 之后客户端与服务端就可以使用「对称加密的秘钥」愉快地发送和接收消息



面试官: 了解了



第一时间获取**BATJTMD**一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



02、TCP

候选者：面试官你好，请问面试可以开始了吗

面试官：嗯，开始吧

面试官：今天来聊聊TCP吧，TCP的各个状态还有印象吗？

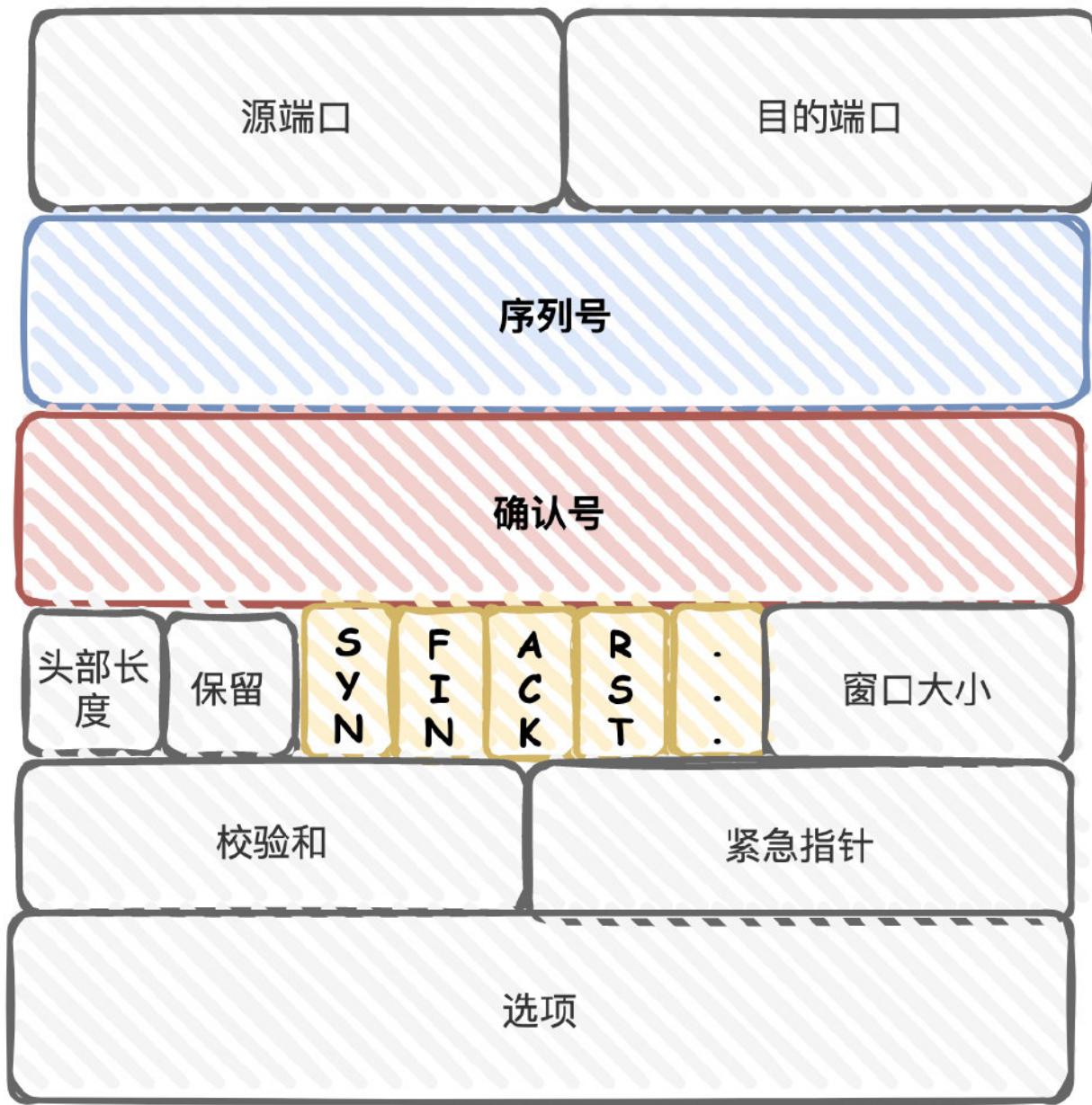
候选者：还有些印象的，要不我就来简单说下TCP的三次握手和四次挥手的流程吧

候选者：说完这两个流程，就能把TCP的状态给涵盖上了

面试官：可以吧

候选者：在说TCP的三次握手和四次挥手之前，我先给你画下TCP的头部格式呗（：

TCP头部



候选者：对于TCP三次握手和四次挥手，我们最主要的就是关注TCP头部的序列号、确认号以及几个标记位（SYN/FIN/ACK/RST）

候选者：序列号：在初次建立连接的时候，客户端和服务端都会为「本次的连接」随机初始化一个序列号。（纵观整个TCP流程中，序列号可以用来解决网络包乱序的问题）

候选者：确认号：该字段表示「接收端」告诉「发送端」对上一个数据包已经成功接收（确认号可以用来解决网络包丢失的问题）

候选者: 而标记位就很好理解啦。SYN为1时，表示希望创建连接。ACK为1时，确认号字段有效。FIN为1时，表示希望断开连接。RST为1时，表示TCP连接出现异常，需要断开。

序列号（解决网络包乱序）

确认号（解决网络包丢失）



不同的标记位有着不同的TCP的语义

候选者: 下面就先从三次握手开始吧，期间我也会在三次握手中涉及到的TCP状态也说一下的。

候选者: TCP三次握手的过程其实就是在：确认通信双方（客户端和服务端）的序列号



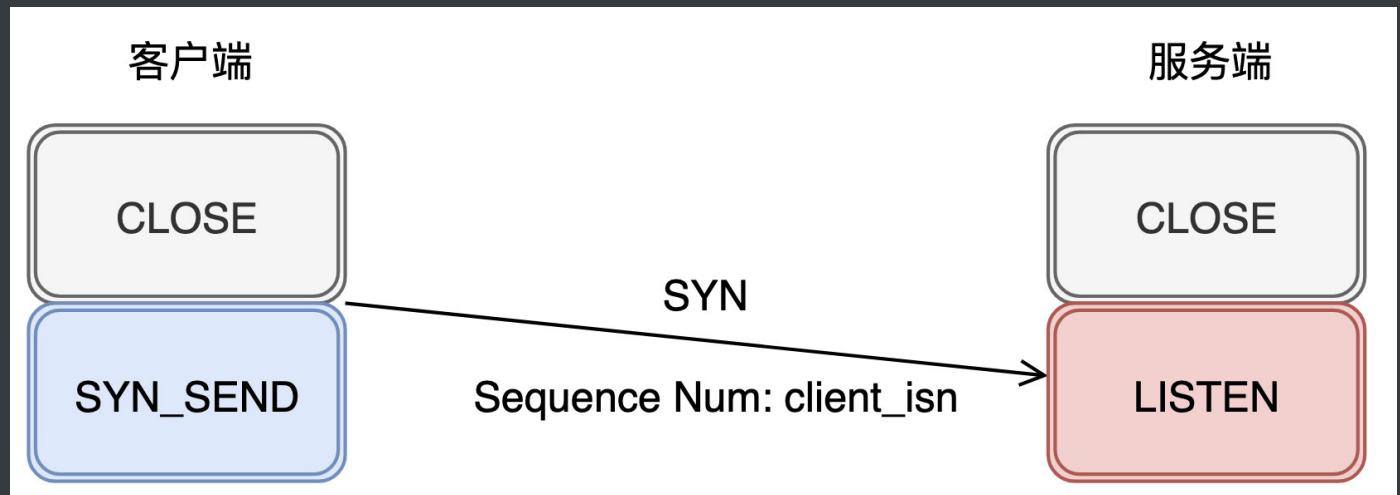
候选者: 它的过程是这样的

候选者: 在最开始的时候，客户端和服务端都处于 CLOSE 状态

候选者: 服务器主动监听某个端口，处于 LISTEN 状态

候选者: 客户端会随机生成出序列号（这里的序列号一般叫做client_isn），并且把标志位设置为SYN（意味着要连接），然后把该报文发送给服务端

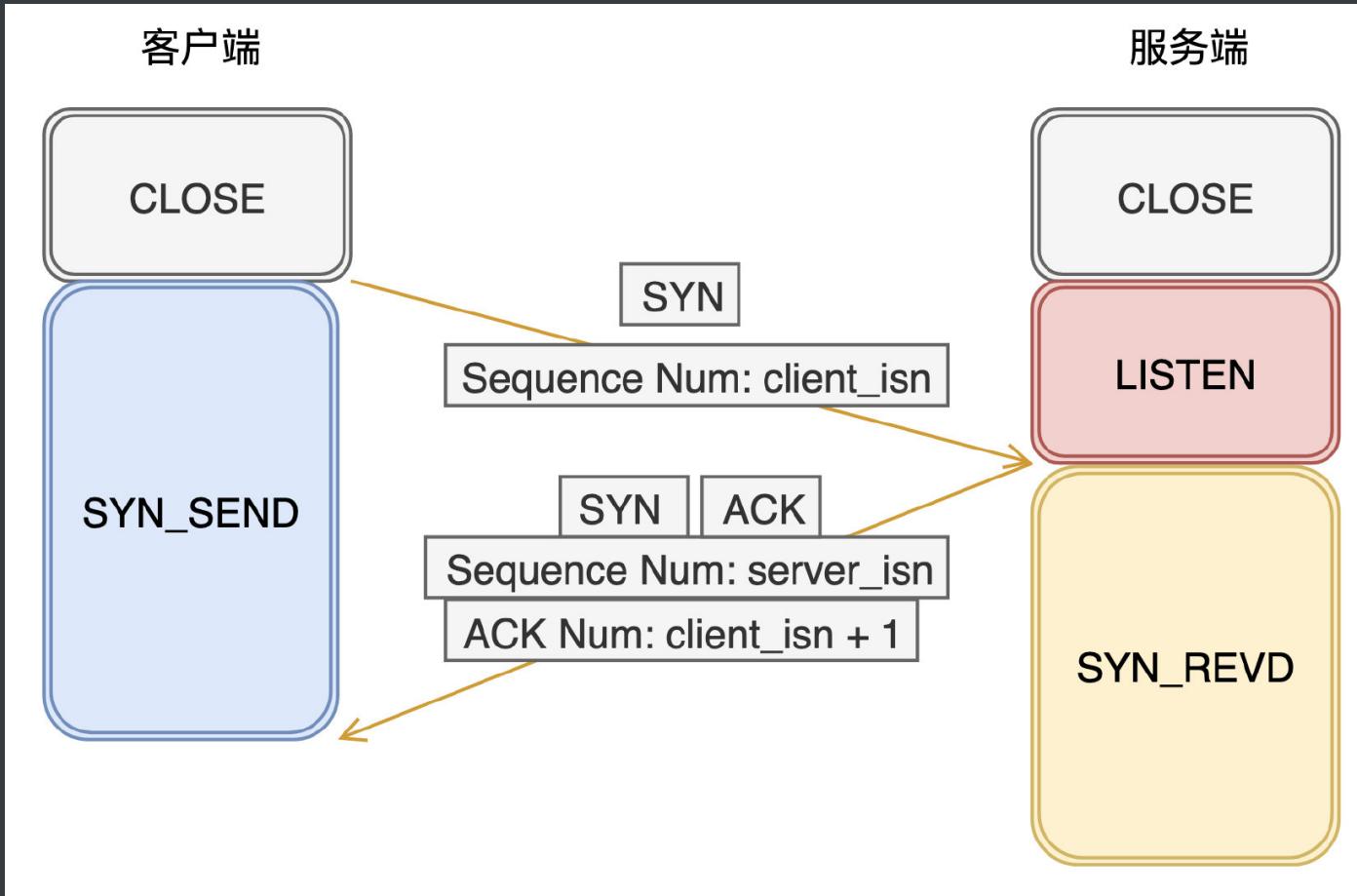
候选者: 客户端发送完SYN报文以后，自己便进入了 SYN_SEND 状态



候选者: 服务端接收到了客户端的请求之后，自己也初始化对应的序列号（这里的序列号一般叫做 server_isn）

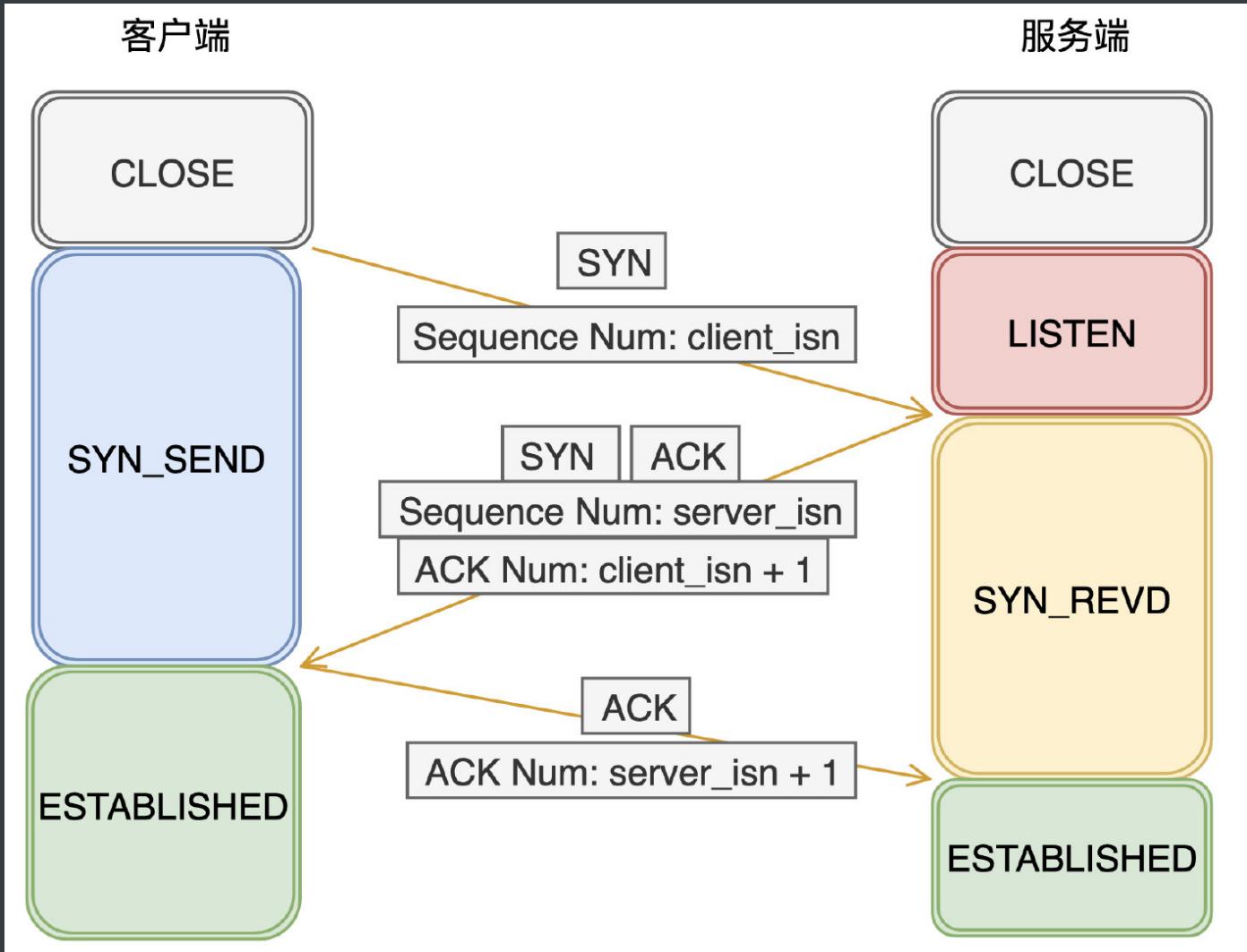
候选者: 在「确认号」字段里填上 $client_isn + 1$ （相当于告诉客户端，已经收到了发送过来的序列号了），并且把 SYN 和 ACK 标记位都点亮(置为1)

候选者: 把该报文发送客户端，服务端的状态变成 SYN-REVD 状态



候选者: 客户端收到服务端发送的报文后，就知道服务端已经接收到了自己的序列号（通过确认号就可以知道），并且接收到了服务端的序列号(server_isn)

候选者: 此时，客户端需要告诉服务端自己已经接收到了他发送过来的序列号，所以在「确认号」字段上填上server_isn+1，，并且标记位 ACK 为1



候选者：客户端在发送报文之后，进入 ESTABLISHED 状态，而服务端接收到客户端的报文之后，也进入 ESTABLISHED 状态

候选者：这就是三次握手的过程以及涉及到的TCP状态

候选者：总结下来，就是双方都把自身的序列号发给对方，看对方能不能接收到。如果「确认可以」，那就可以正常通信。（三次握手这个过程就可以看到双方都有接收和发送的能力）

面试官：那两次握手行吗？

候选者：两次握手只能保证客户端的序列号成功被服务端接收，而服务端是无法确认自己的序列号是否被客户端成功接收。所以是不行的（：

面试官：了解了，那我想问问序列号为什么是随机的？以及序列号是怎么生成的？

候选者: 一方面为了安全性（随机ISN能避免非同一网络的攻击），另一方面可以让通信双方能够根据序号将「不属于」本连接的报文段丢弃

候选者: 序列号怎么生成的？这...随便猜下就应该跟「时钟」和TCP头部的某些属性做运算生成的吧，类似于雪花算法（：具体我忘了。

面试官: 既然网络是不可靠的，那建立连接不是会经过三次握手吗？那要是在中途丢了，怎么办？

候选者: 假设第一个包丢了，客户端发送给服务端的 SYN 包丢了（简而言之就是服务端没接收到客户端的SYN包）

候选者: 客户端迟迟收不到服务端的ACK包，那会周期性超时重传，直到收到服务端的ACK

候选者: 假设第二个包丢了，服务端发送的SYN+ACK包丢了（简而言之就是客户端没接收到服务端的SYN+ACK包）

候选者: 服务端迟迟收不到客户端的ACK包，那会周期性超时重传，直到收到客户端的ACK

候选者: 假设第三个包丢了（ACK包），客户端发送完第三个包后单方面进入了 ESTABLISHED 状态，而服务端也认为此时连接是正常的，但第三个包没到达服务端

候选者: 一、如果此时客户端与服务端都还没数据发送，那服务端会认为自己发送的 SYN+ACK的包没发送至客户端，所以会超时重传自己的SYN+ACK包

候选者: 二、如果这时候客户端已经要发送数据了，服务端接收到了ACK + Data数据包，那自然就切换到 ESTABLISHED 状态下，并且接收客户端的Data数据包

候选者: 三、如果此时服务端要发送数据了，但发送不了，会一直周期性超时重传SYN + ACK，直到接收到客户端的ACK包

对方不回复你，继续问啊，愣着干什么？

面试官: 嗯，是不是要讲下四次挥手了？

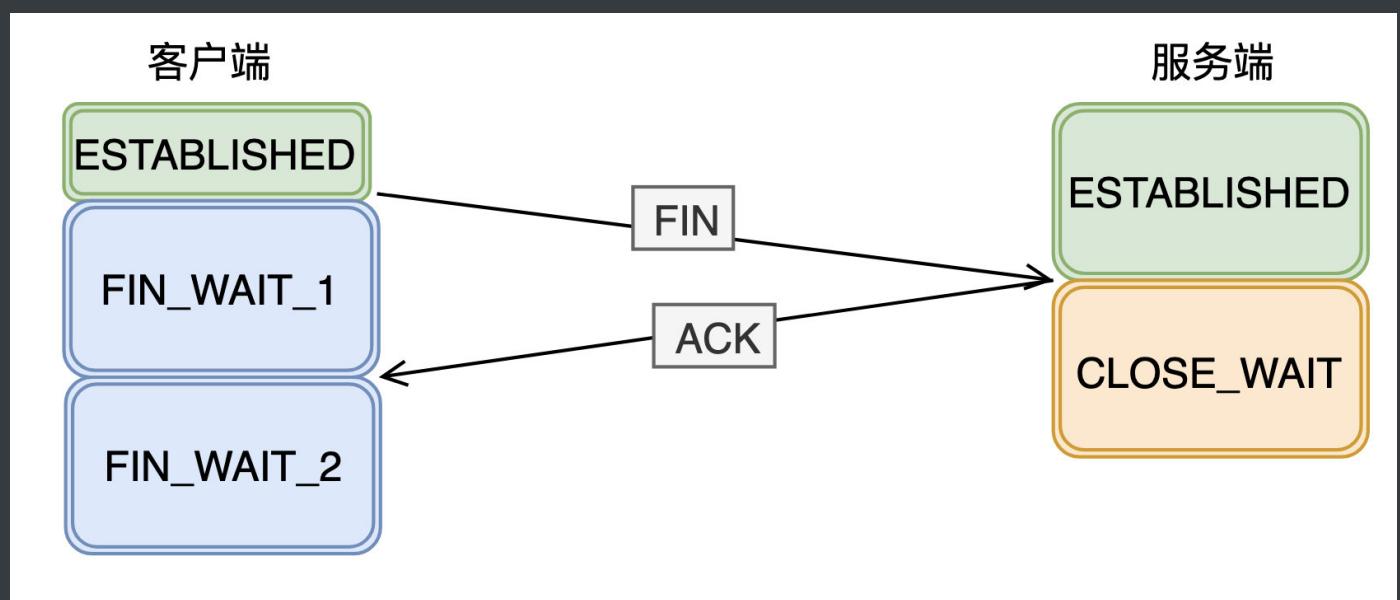
候选者: 嗯，在建立完连接之后，客户端和服务端双方都处于 ESTABLISHED 状态状态

候选者: 断开连接双方都有权利的，下面我还是以客户端主动断开为例好啦。

候选者: 客户端打算关闭连接，会发 FIN 报文给服务端（其实就是把标志位 FIN 点亮），客户端发送完之后，就进入 FIN_WAIT_1 状态

候选者: 服务端收到 FIN 报文之后，回复 ACK 报文给客户端（表示已经收到了），服务端发送完之后，就进入 CLOSE_WAIT 状态

候选者: 客户端接收到服务端的 ACK 报文，就进入了 FIN_WAIT_2 状态

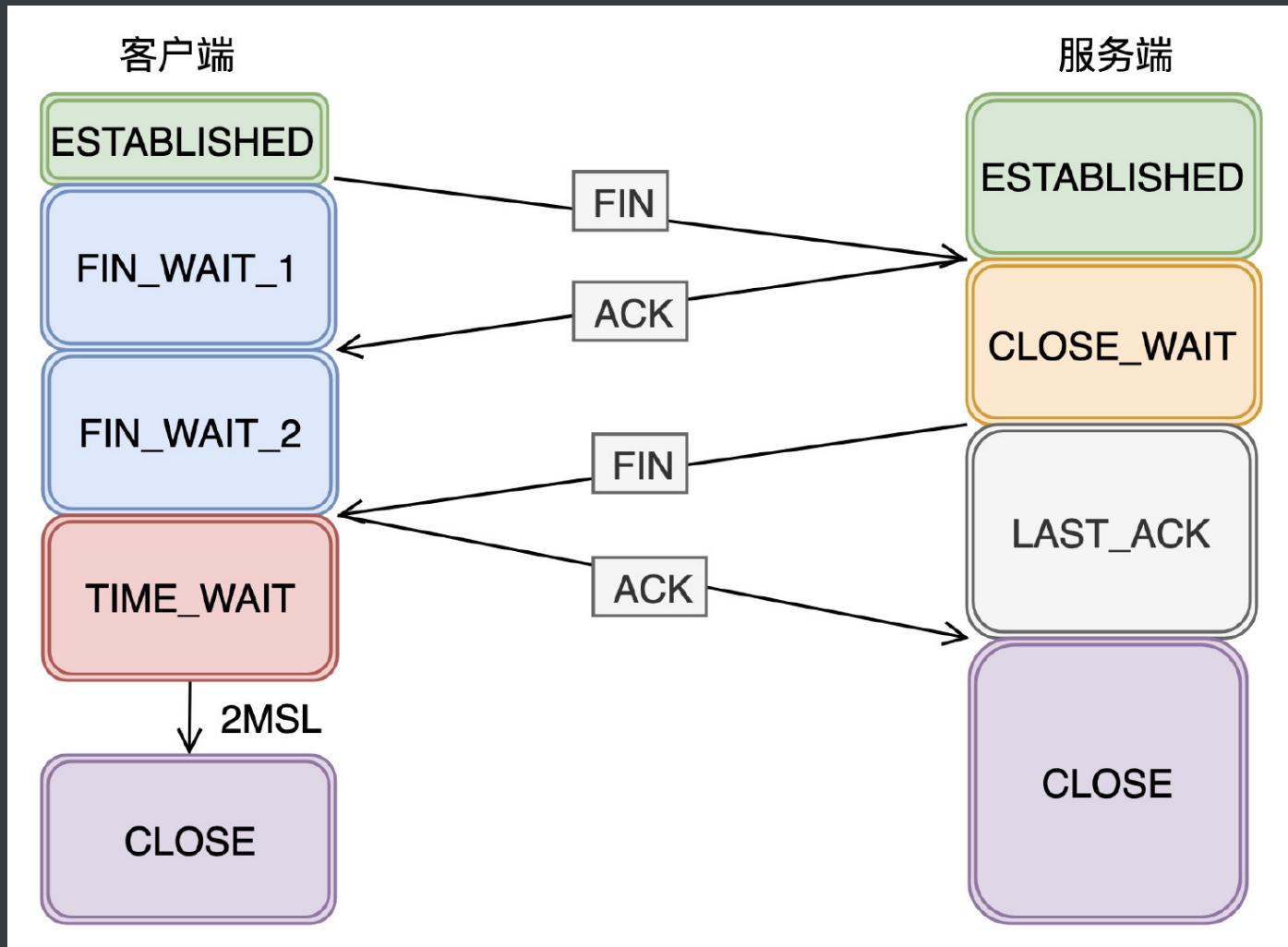


候选者: 这时候，服务器可能还有数据要发送给客户端，等服务端确认自己已经没有数据返回给客户端之后，就发送FIN报文给客户端了，自己进入 LAST_ACK 状态

候选者: 客户端收到服务端的FIN报文之后，回应ACK报文，自己进入 TIME_WAIT 状态

候选者: 服务端收到客户端的ACK报文之后，服务端就进入 CLOSE 状态

候选者: 客户端在TIME_WAIT等到2MSL，也进入了 CLOSE 状态



候选者：四次挥手的流程到这里就结束了，结合三次握手，TCP的各个状态也已经说完了。

面试官：嗯嗯，刚聊完四次挥手嘛，那你觉得为什么是四次呢？

候选者：其实很好理解，当客户端第一次发送 FIN 报文之后，只是代表着客户端不再发送数据给服务端，但此时客户端还是有接收数据的能力的。而服务端收到FIN报文的时候，可能还有数据要传输给客户端，所以只能先回复 ACK给客户端

候选者：等到服务端不再有数据发送给客户端时，才发送 FIN 报文给客户端，表示可以关闭了。

候选者：所以，一来一回就四次了。

你要结束恋爱了是吧？我知道了，但你先听我解释下

面试官：从四次挥手的流程上来看，有个 TIME_WAIT 状态，你知道这个状态干什么用的吗？（等待 2MSL）

候选者：主要有两个原因吧。1.保证最后的 ACK 报文「接收方」一定能收到（如果收不到，对方会重发 FIN 报文）2.确保在创建新连接时，先前网络中残余的数据都丢失了

候选者：其实也比较好理解的。就正如我们重启服务器一样，会先优雅关闭各种资源，再留有一段时间，希望在这段时间内，资源是正常关闭的，这样重启服务器（或者发布）就基本认为不会影响到线上运行了。

面试官：假设 TIME_WAIT 状态多过会有什么危害？怎么解决呢？

候选者：从流程上看，TIME_WAIT 状态只会出现在主动发起关闭连接的一方。危害就是会占用内存资源和端口呗（毕竟在等待嘛），解决的话，有Linux参数可以设置，具体忘了额。

面试官：今天最后再问个问题吧，我们常说TCP连接，那这个连接到底是什么？你是怎么理解的？

候选者：其实从三次握手可以发现的是，TCP建立连接无非就是交换了双方的状态（比如序列号）。然后就没有然后了...连接本质上「只是互相维持一个状态，有连接特性」

面试官：好吧。



求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zhongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

11-算法

01、排序算法

冒泡排序

思路：俩俩交换，大的放在后面，第一次排序后最大值已在数组末尾。因为俩俩交换，需要 $n-1$ 趟排序（比如10个数，需要9趟排序）

代码实现要点：两个for循环，外层循环控制排序的趟数，内层循环控制比较的次数。每趟过后，比较的次数都应该要减1

```
//装载临时变量
int temp;

//记录是否发生了置换， 0 表示没有发生置换、 1 表示发生了置换
int isChange;

//记录执行了多少趟
int num = 0;

//外层循环是排序的趟数
for (int i = 0; i < arrays.length -1 ; i++) {

    //每比较一趟就重新初始化为0
    isChange = 0;

    //内层循环是当前趟数需要比较的次数
    for (int j = 0; j < arrays.length - i - 1; j++) {

        //前一位与后一位与前一位比较，如果前一位比后一位要大，那么交换
        if (arrays[j] > arrays[j + 1]) {
            temp = arrays[j];
            arrays[j] = arrays[j + 1];
            arrays[j + 1] = temp;

            //如果进到这里面了，说明发生置换了
            isChange = 1;
        }
    }

    //如果比较完一趟没有发生置换，那么说明已经排好序了，不需要再执行下去了
    if (isChange == 0) {
```

```
        break;  
    }  
    num++;  
  
}
```

选择排序

思路：找到数组中最大的元素，与数组最后一位元素交换。当只有一个数时，则不需要选择了，因此需要 $n-1$ 趟排序

代码实现要点：两个for循环，外层循环控制排序的趟数，内层循环找到当前趟数的最大值，随后与当前趟数组最后的一位元素交换

```
//记录当前趟数的最大值的角标  
int pos ;  
  
//交换的变量  
int temp;  
  
//外层循环控制需要排序的趟数  
for (int i = 0; i < arrays.length - 1; i++) {  
  
    //新的趟数、将角标重新赋值为0  
    pos = 0;  
  
    //内层循环控制遍历数组的个数并得到最大数的角标  
    for (int j = 0; j < arrays.length - i; j++) {  
  
        if (arrays[j] > arrays[pos]) {  
            pos = j;  
        }  
  
    }  
}
```

```
//交换
temp = arrays[pos];
arrays[pos] = arrays[arrays.length - 1 - i];
arrays[arrays.length - 1 - i] = temp;

}

System.out.println("公众号Java3y" + arrays);
```

插入排序

思路：将一个元素插入到已有序的数组中，在初始时未知是否存在有序的数据，因此将元素第一个元素看成是有序的。与有序的数组进行比较，比它大则直接放入，比它小则移动数组元素的位置，找到个合适的位置插入。当只有一个数时，则不需要插入了，因此需要 $n-1$ 趟排序

代码实现：一个for循环内嵌一个while循环实现，外层for循环控制需要排序的趟数，while循环找到合适的插入位置(并且插入的位置不能小于0)

```
//临时变量
int temp;

//外层循环控制需要排序的趟数(从1开始因为将第0位看成了有序数据)
for (int i = 1; i < arrays.length; i++) {

    temp = arrays[i];

    //如果前一位(已排序的数据)比当前数据要大，那么就进入循环比较[参考第二趟排序]
    int j = i - 1;

    while (j >= 0 && arrays[j] > temp) {

        //往后退一个位置，让当前数据与之前前位进行比较
        arrays[j + 1] = arrays[j];
        j--;
    }

    arrays[j + 1] = temp;
}
```

```
arrays[j + 1] = arrays[j];

//不断往前，直到退出循环
j--;

}

//退出了循环说明找到了合适的位置了，将当前数据插入合适的位置中
arrays[j + 1] = temp;

}

System.out.println("公众号Java3y" + arrays);
```

快速排序

学习快速排序的前提：需要了解递归

思路：在数组中找一个元素(节点)，比它小的放在节点的左边，比它大的放在节点右边。一趟下来，比节点小的在左边，比节点大的在右边。不断执行这个操作....

代码实现：支点取中间，使用L和R表示数组的最小和最大位置。不断进行比较，直到找到比支点小(大)的数，随后交换，不断减小范围。递归L到支点前一个元素(j)。递归支点后一个元素(i)到R元素

```
/**
 * 快速排序
 *
 * @param arr
 * @param L    指向数组第一个元素
 * @param R    指向数组最后一个元素
```



```
//“右边”再做排序，直到右边剩下一个数(递归出口)
if (i < R)
    quickSort(arr, i, R);
System.out.println("关注公众号：Java3y 免费领取各种知识点的精美PDF");
}
```

归并排序

学习归并排序的前提：需要了解递归

思路：将两个已排好序的数组合并成一个有序的数组。**将元素分隔开来，看成是有序的数组，进行比较合并。**不断拆分和合并，直到只有一个元素

代码实现：在第一趟排序时实质是两个元素(看成是两个已有序的数组)来进行合并，不断执行这样的操作，最终数组有序，拆分左边，右边，合并...

```
/**
 * 归并排序
 *
 * @param arrays
 * @param L      指向数组第一个元素
 * @param R      指向数组最后一个元素
 */
public static void mergeSort(int[] arrays, int L, int R) {
```

```
//如果只有一个元素，那就不用排序了
if (L == R) {
    return;
} else {

    //取中间的数，进行拆分
    int M = (L + R) / 2;

    //左边的数不断进行拆分
    mergeSort(arrays, L, M);

    //右边的数不断进行拆分
    mergeSort(arrays, M + 1, R);

    //合并
    merge(arrays, L, M + 1, R);
    System.out.println("关注公众号：Java3y 免费领取各种知识点的精美
PDF");
}

}

/**
 * 合并数组
 *
 * @param arrays
 * @param L          指向数组第一个元素
 * @param M          指向数组分隔的元素
 * @param R          指向数组最后的元素
 */
public static void merge(int[] arrays, int L, int M, int R) {

    //左边的数组的大小
    int[] leftArray = new int[M - L];

    //右边的数组大小
    int[] rightArray = new int[R - M];
```

```
int[] rightArray = new int[R - M + 1];

//往这两个数组填充数据
for (int i = L; i < M; i++) {
    leftArray[i - L] = arrays[i];
}
for (int i = M; i <= R; i++) {
    rightArray[i - M] = arrays[i];
}

int i = 0, j = 0;
// arrays数组的第一个元素
int k = L;

//比较这两个数组的值，哪个小，就往数组上放
while (i < leftArray.length && j < rightArray.length) {

    //谁比较小，谁将元素放入大数组中，移动指针，继续比较下一个
    if (leftArray[i] < rightArray[j]) {
        arrays[k] = leftArray[i];

        i++;
        k++;
    } else {
        arrays[k] = rightArray[j];

        j++;
        k++;
    }
}

//如果左边的数组还没比较完，右边的数都已经完了，那么将左边的数抄到大数组中
//（剩下的都是大数字）
while (i < leftArray.length) {
    arrays[k] = leftArray[i];
```

```
i++;
k++;
}
//如果右边的数组还没比较完，左边的数都已经完了，那么将右边的数抄到大数组中
(剩下的都是大数字)
while (j < rightArray.length) {
    arrays[k] = rightArray[j];

    k++;
    j++;
}
System.out.println("关注公众号：Java3y 免费领取各种知识点的精美PDF");
}
```

堆排序

学习堆排序的前提：需要了解二叉树

思路：堆排序使用到了完全二叉树的一个特性，根节点比左孩子和右孩子都要大，完成一次**建堆**的操作实质上是比较根节点和左孩子、右孩子的大小，大的交换到根节点上，**直至最大的节点在树顶**。随后与数组最后一位元素进行交换

代码实现：只要左子树或右子树大于当前根节点，则替换。替换后会导致下面的子树发生了变化，因此同样需要进行比较，直至各个节点实现父>子这么一个条件

```
public class HeapifySort {  
  
    public static void main(String[] args) {  
  
        int[] arrays = {6, 3, 8, 5, 2, -1, -5, -2, -6, 345, 7, 5, 1, 2, 23,  
4321, 432, 3, 2, 34234, 2134, 1234, 5, 132423, 234, 4, 2, 4, 1, 5, 2,  
5};  
  
        // 完成一次建堆..  
        maxHeapify(arrays, arrays.length - 1);  
        int size = arrays.length - 1;  
  
        for (int i = 0; i < arrays.length; i++) {  
            //交换  
            int temp = arrays[0];  
            arrays[0] = arrays[(arrays.length - 1) - i];  
            arrays[(arrays.length - 1) - i] = temp;  
  
            // 调整位置  
            heapify(arrays, 0, size);  
            size--;  
        }  
        System.out.println("关注公众号: Java3y 免费领取各种知识点的精美PDF");  
    }  
  
    /**  
     * 完成一次建堆, 最大值在堆的顶部(根节点)  
     */  
    public static void maxHeapify(int[] arrays, int size) {  
        for (int i = size - 1; i >= 0; i--) {  
            heapify(arrays, i, size);  
        }  
    }  
  
    /**  
     * 建堆
```

```
* @param arrays 看作是完全二叉树
* @param currentRootNode 当前父节点位置
* @param size 节点总数
*/
public static void heapify(int[] arrays, int currentRootNode, int size) {

    if (currentRootNode < size) {
        //左子树和右子树的位置
        int left = 2 * currentRootNode + 1;
        int right = 2 * currentRootNode + 2;

        //把当前父节点位置看成是最大的
        int max = currentRootNode;

        if (left < size) {
            //如果比当前根元素要大，记录它的位置
            if (arrays[max] < arrays[left]) {
                max = left;
            }
        }
        if (right < size) {
            //如果比当前根元素要大，记录它的位置
            if (arrays[max] < arrays[right]) {
                max = right;
            }
        }
        //如果最大的不是根元素位置，那么就交换
        if (max != currentRootNode) {
            int temp = arrays[max];
            arrays[max] = arrays[currentRootNode];
            arrays[currentRootNode] = temp;

            //继续比较，直到完成一次建堆
            heapify(arrays, max, size);
        }
    }
}
```

```
        }
    }

}
```

希尔排序

思路：希尔排序实质上就是插入排序的增强版，希尔排序将数组分隔成n组来进行插入排序，**直至该数组宏观上有有序**，最后再进行插入排序时就不用移动那么多次位置了~

代码思路：希尔增量一般是 $gap = gap / 2$ ，只是比普通版插入排序多了这么一个for循环而已。

```
/*
 * 希尔排序
 *
 * @param arrays
 */
public static void shellSort(int[] arrays) {
    //增量每次都/2
    for (int step = arrays.length / 2; step > 0; step /= 2) {

        //从增量那组开始进行插入排序，直至完毕
        for (int i = step; i < arrays.length; i++) {

            int j = i;
            int temp = arrays[j];

            // j - step 就是代表与它同组隔壁的元素
            while (j - step >= 0 && arrays[j - step] > temp) {
                arrays[j] = arrays[j - step];
                j = j - step;
            }
        }
    }
}
```

```
        arrays[j] = temp;
    }
}

System.out.println("关注公众号: Java3y 免费领取各种知识点的精美PDF");
}
```

基数排序(桶排序)

思路：基数排序(桶排序)：将数字切割成个、十、百、千位放入到不同的桶子里，放一次就按桶子顺序回收一次，直至最大位数的数字放完～那么该数组就有有序了

代码实现：先找到数组的最大值，然后根据最大值/10来作为循环的条件(只要>0，那么就说明还有位数)。将个位、十位、...分配到桶子上，每分配一次就回收一次

```
/***
 * 基数排序 (桶排序)
 * @param arrays
 */
public static void radixSort(int[] arrays) {
    int max = findMax(arrays, 0, arrays.length - 1);

    //需要遍历的次数由数组最大值的位数来决定
    for (int i = 1; max / i > 0; i = i * 10) {

        int[][] buckets = new int[arrays.length][10];

        //获取每一位数字(个、十、百、千位...分配到桶子里)
        for (int j = 0; j < arrays.length; j++) {
            int num = (arrays[j] / i) % 10;
            //将其放入桶子里
            buckets[j][num] = arrays[j];
        }
    }
}
```

```
//回收桶子里的元素
int k = 0;

//有10个桶子
for (int j = 0; j < 10; j++) {
    //对每个桶子里的元素进行回收
    for (int l = 0; l < arrays.length; l++) {
        //如果桶子里面有元素就回收(数据初始化会为0)
        if (buckets[l][j] != 0) {
            arrays[k++] = buckets[l][j];
        }
    }
}
System.out.println("关注公众号: Java3y 免费领取各种知识点的精美PDF");
}

/**
 * 递归，找出数组最大的值
 *
 * @param arrays 数组
 * @param L      左边界，第一个数
 * @param R      右边界，数组的长度
 * @return
 */
public static int findMax(int[] arrays, int L, int R) {
    //如果该数组只有一个数，那么最大的就是该数组第一个值了
    if (L == R) {
        return arrays[L];
    } else {
        int a = arrays[L];
        int b = findMax(arrays, L + 1, R); //找出整体的最大值
    }
}
```

```
if (a > b) {  
    return a;  
} else {  
    return b;  
}  
}
```



排序算法有部分需要递归和树相关的基础知识，我都已经整理到PDF上了。

打造这么一个文档花了我不少的时间。为了防止白嫖，关注我的公众号回复「888」即可获取。

The screenshot shows a PDF document with a table of contents on the left and a detailed explanation of the second pass of bubble sort on the right.

Table of Contents:

- 前言
- 一、冒泡排序
- 二、选择排序
- 三、插入排序
 - 零、插入排序介绍
 - 一、第一趟排序
 - 二、第二趟排序
 - 三、简化代码
- 四、快速排序
- 五、归并排序
- 六、希尔排序
- 七、堆排序
 - 一、堆排序介绍
 - 二、堆排序体验
 - 三、堆排序代码实现
- 八、基数排序（桶排序）
 - 一、基数排序(桶排序)介绍
 - 二、基数排序体验
 - 2.1第一趟分配与回收
 - 2.2第二趟分配与回收
 - 2.3第三趟分配与回收
 - 2.4第四趟分配与回收
 - 三、基数排序代码编写
 - 四、桶排序（基数排序）总结
- 九、递归
- 十、链表
- 十一、栈
- 十二、队列

Section: 二、第二趟排序

用数组的第三个数与已是有序的数据 {2, 3} (刚才在第一趟排的)比较

- 如果比第二位的数大，那就不管它
- 如果比第二位的数小，那就将第二的位置退一个位置，让第三个数和第一位比较
 - 如果第三个数比第一位大，那么将第三个数插入到第二的位置上
 - 如果第三个数比第一位小，那么将第一位后退一步，将第三个数插入到第一的位置上

```
//第二趟排序-----  
if (arrays[2] > arrays[1]) {  
    //如果第三个数比第二个数大，直接跟上  
  
} else {  
    //如果第三个数比第二个数小，将第二个数往后退一个位置，让第三个数跟第一个数比  
    temp = arrays[2];  
  
    arrays[2] = arrays[1];  
  
    //如果第三个数比第一个大，那就插入到第二个数中  
    if (temp > arrays[0]) {  
        arrays[1] = temp;  
    } else {  
  
        //如果第三个数比第一个小，将第三个数插入到第一个数前面  
        int swapTemp = arrays[0];  
        arrays[0] = temp;  
        arrays[1] = swapTemp;  
  
    }  
  
}  
System.out.println("公众号Java3y" + arrays);
```

第一时间获取BATJTMD一线互联网大厂最新的面试资料以及内推机会关注公众号「对线面试官」



02、LeetCode Easy

LeetCode 1

问题：给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 `和` 为为目标值 `target` 的那 两个 整数，并返回它们的数组下标。你可以假设每种输入只会对应一个答案。但是， 数组中同一个元素在答案里不能重复出现。

<https://leetcode-cn.com/problems/two-sum/>

```
/**
 * 两数之和
 * 1. Map存储着val 对应的下标
 * 2. target - num[i] 在Map中有存储，则能算出结果
 */
public class LeetCode1 {

    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> memo = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int val = target - nums[i];
            if (memo.containsKey(val)) {
                return new int[]{i, memo.get(val)};
            }
            memo.put(nums[i], i);
        }
    }
}
```

```
        } else {
            memo.put(nums[i], i);
        }
    }
    return new int[2];
}

}
```

LeetCode 20

问题：给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串 s，判断字符串是否有效。

有效字符串需满足：

- 左括号必须用相同类型的右括号闭合。
- 左括号必须以正确的顺序闭合。

<https://leetcode-cn.com/problems/valid-parentheses/>

```
/*
 * 有效括号
 * 1. 借助Stack 栈先进先出的特性，将左侧括号进入栈
 * 2. 当前character不是左侧括号时，则pop出来比对是否匹配
 * 3. 注意要判断边界问题（在pop前 memo的大小是否为0， 在遍历后， memo的大小是否为0）
 */
public class LeetCode20 {
    public boolean isValid(String s) {

        LinkedList<Character> memo = new LinkedList<>();
        for (int i = 0; i < s.length(); i++) {
```

```
        if (s.charAt(i) == '[' || s.charAt(i) == '{' || s.charAt(i)
== '(') {
            memo.push(s.charAt(i));
        } else {
            if (memo.size() == 0) {
                return false;
            }
            Character pop = memo.pop();
            Character c = null;
            if (pop.equals('(')) {
                c = ')';
            }
            if (pop.equals('{')) {
                c = '}';
            }
            if (pop.equals('[')) {
                c = ']';
            }
            if (!c.equals(s.charAt(i))) {
                return false;
            }
        }
    }

    if (memo.size() != 0) {
        return false;
    }

    return true;
}
}
```

LeetCode 21

问题：将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

<https://leetcode-cn.com/problems/merge-two-sorted-lists/>

```
/*
 * 合并两个有序链表
 * 1. 关键在于 使用虚拟头节点
 * 2. 记得移动 cur指针
 */
public class LeetCode21 {

    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null) {
            return l2;
        }
        if (l2 == null) {
            return l1;
        }
        if (l1 == null && l2 == null) {
            return null;
        }

        ListNode dummyHead = new ListNode(0);
        ListNode cur = dummyHead;

        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                cur.next = l1;
                l1 = l1.next;
            } else {
                cur.next = l2;
                l2 = l2.next;
            }
            cur = cur.next;
        }

        if (l1 == null) {
            cur.next = l2;
        } else {
            cur.next = l1;
        }
        return dummyHead.next;
    }
}
```

```
        } else {
            cur.next = l2;
            l2 = l2.next;
        }
        cur = cur.next;
    }
    while (l1 != null) {
        cur.next = l1;
        l1 = l1.next;
    }
    while (l2 != null) {
        cur.next = l2;
        l2 = l2.next;
    }
    return dummyHead.next;
}
}
```

LeetCode 53

问题：给定一个整数数组 `nums`，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

<https://leetcode-cn.com/problems/maximum-subarray/>

```
/**
```

最大子序和(贪心算法)

`res`作为历史最佳解，
`sum`作为当前最佳解，

每一次遍历`nums`数组时，
都去动态更新`res`和`sum`。

动态更新的逻辑为： 如果`sum`为正数，在有`res`记录历史最佳值的条件下，可以有恃无恐地继续累加，创造新高；

如果`sum`为负数，不管下一次遍历值是多少累加后都不会大于它，见风使舵果断取下一个遍历值为当前最佳解。

每一轮遍历结束后，如果当前最佳解优于历史最佳解，就会升任历史最佳解。

*/

```
public class LeetCode53 {  
  
    public int maxSubArray(int[] nums) {  
  
        int res = nums[0];  
        int sum = 0;  
        for (int num : nums) {  
            if (sum > 0) {  
                sum = sum + num;  
            } else {  
                sum = num;  
            }  
            res = Math.max(res, sum);  
        }  
  
        return res;  
    }  
}
```

LeetCode 70

问题：假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

<https://leetcode-cn.com/problems/climbing-stairs/>

```
/**  
 * 爬楼梯  
 * 斐波拉契数列问题 (使用备忘录, 处理过的步数直接返回)  
 * 1. 当n=1和n=2时 直接添加至备忘录且返回  
 * 2. 当n在备忘录曾经计算过时, 直接返回  
 * 3. 备忘录添加 n的步数 (n-1) + (n-2)  
 * 4. 返回备忘录的n  
 */  
  
public class LeetCode70 {  
  
    Map<Integer, Integer> memo = new HashMap<>();  
  
    public int climbStairs(int n) {  
  
        if (n < 3) {  
            memo.put(n, n);  
            return memo.get(n);  
        }  
        if (memo.containsKey(n)) {  
            return memo.get(n);  
        }  
        memo.put(n, climbStairs(n - 1) + climbStairs(n - 2));  
  
        return memo.get(n);  
    }  
}
```

LeetCode 101

问题：给定一个二叉树，检查它是否是镜像对称的。

<https://leetcode-cn.com/problems/symmetric-tree/>

```
/*
 * 给定一个二叉树，检查它是否是镜像对称的。
 * <p>
 * 1. 假如左子树和右子树都为null(说明只有"root"节点 返回true)
 * 2. 左右子树的val 不相等， return false
 */
public class LeetCode101 {

    public boolean isSymmetric(TreeNode root) {

        if (root == null) {
            return true;
        }
        return check(root.left, root.right);
    }

    private boolean check(TreeNode left, TreeNode right) {

        if (left == null && right == null) {
            return true;
        }
        if (left == null || right == null || left.val != right.val) {
            return false;
        }
        return check(left.left, right.right) && check(left.right,
right.left);

    }
}
```

LeetCode 104

问题: 给定一个二叉树，找出其最大深度，二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

<https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/>

```
/*
 * 二叉树的最大深度
 * 递归找出左边和右边最大的深度 +1 就好了
 */
public class LeetCode104 {

    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
    }
}
```

LeetCode 136

问题: 给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明: 你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

<https://leetcode-cn.com/problems/single-number/>


```
}
```

LeetCode 155

问题：设计一个支持 push , pop , top 操作，并能在常数时间内检索到最小元素的栈。

push(x) —— 将元素 x 推入栈中。

pop() —— 删除栈顶的元素。

top() —— 获取栈顶元素。

getMin() —— 检索栈中的最小元素。

提示： pop 、 top 和 getMin 操作总是在 非空栈 上调用。

```
/**  
 * 借助 minStack 完成  
 *  
 * 首先扔进去一个MaxIntegerVal，这样就不用判空了  
 */  
  
class MinStack {  
  
    LinkedList<Integer> stack = new LinkedList<>();  
  
    LinkedList<Integer> minStack = new LinkedList<>();  
  
    /**  
     * initialize your data structure here.  
     */  
    public MinStack() {  
        minStack.push(Integer.MAX_VALUE);  
    }  
}
```

```
public void push(int x) {
    stack.push(x);
    minStack.push(Math.min(minStack.peek(), x));
}

public void pop() {
    stack.pop();
    minStack.pop();
}

public int top() {
    return stack.peek();
}

public int getMin() {
    return minStack.peek();
}

public static void main(String[] args) {

    MinStack obj = new MinStack();
    obj.push(2);
    obj.pop();
    int param_3 = obj.top();
    int param_4 = obj.getMin();

}

}
```

LeetCode 160

问题：给你两个单链表的头节点 headA 和 headB，请你找出并返回两个单链表相交的起始节点。如果两个链表没有交点，返回 null。

<https://leetcode-cn.com/problems/intersection-of-two-linked-lists/>

```
/**  
 * 编写一个程序，找到两个单链表相交的起始节点。  
 * 1. 两个节点相等，那就是相交；用Map来存储即可  
 */  
  
public class LeetCode160 {  
    public ListNode getIntersectionNode(ListNode headA, ListNode headB)  
{  
  
    HashMap<ListNode, ListNode> map = new HashMap<>();  
  
    if (headA == null || headB == null) {  
        return null;  
    }  
  
    while (headA != null) {  
        map.put(headA, headA);  
        headA = headA.next;  
    }  
    while (headB != null) {  
        if (map.get(headB) != null) {  
            return headB;  
        }  
        headB = headB.next;  
    }  
  
    return null;  
}  
}
```

LeetCode 169

问题：给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数 大于 $\lfloor n/2 \rfloor$ 的元素。你可以假设数组是非空的，并且给定的数组总是存在多数元素。

<https://leetcode-cn.com/problems/majority-element/>

```
/**  
 * 给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数 大于  $\lfloor n/2 \rfloor$  的元素。  
 *  
 * 你可以假设数组是非空的，并且给定的数组总是存在多数元素。  
 *  
 * 1. Map大法好，遍历直接存储出现的次数，找到最大的即可  
 * 2. 使用单个变量来记录，剩下的一定是"多数"的元素  
 *      (从第一个数开始count=1，遇到相同的就加1，遇到不同的就减1，减到0就重新换个数开始计数，总能找到最多的那个)  
 */  
  
public class LeetCode169 {  
    public int majorityElement(int[] nums) {  
        HashMap<Integer, Integer> keyVal = new HashMap<>();  
        for (int num : nums) {  
            if (keyVal.containsKey(num)) {  
                keyVal.put(num, keyVal.get(num) + 1);  
            } else {  
                keyVal.put(num, 1);  
            }  
        }  
        int max = Integer.MIN_VALUE;  
        int key = 0;  
        for (Map.Entry<Integer, Integer> entry : keyVal.entrySet()) {  
            if (entry.getValue() > max) {  
                max = entry.getValue();  
                key = entry.getKey();  
            }  
        }  
        return key;  
    }  
}
```

```
        }
    }
    return key;
}

// public int majorityElement(int[] nums) {
//     int cand_num = nums[0];
//     int count = 1;
//     for (int i = 1; i < nums.length; ++i) {
//         if (cand_num == nums[i]){
//             ++count;
//         } else if (--count == 0) {
//             cand_num = nums[i];
//             count = 1;
//         }
//     }
//     return cand_num;
// }

}
```

LeetCode 206

问题：给你单链表的头节点 `head`，请你反转链表，并返回反转后的链表。

<https://leetcode-cn.com/problems/reverse-linked-list/>

```
/**
 * 反转链表
 * 1. 头节点指向 NULL
 * 2. 头结点和下一个节点不断往后移
 */
```

```
public class LeetCode206 {
    public ListNode reverseList(ListNode head) {
        ListNode cur = head;
        ListNode pre = null;

        while (cur != null) {
            ListNode next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }

}
```

LeetCode 226

问题：翻转一棵二叉树。

<https://leetcode-cn.com/problems/invert-binary-tree/>

```
/**
 * 反转二叉树
 */
public class LeetCode226 {
    public TreeNode invertTree(TreeNode root) {
        if (root == null) {
            return null;
        }

        TreeNode left = invertTree(root.left);
        TreeNode right = invertTree(root.right);
        root.left = right;
        root.right = left;
        return root;
    }
}
```

```
    root.left = right;
    root.right = left;
    return root;
}

}
```

LeetCode 234

问题：给你一个单链表的头节点 `head`，请你判断该链表是否为回文链表。如果是，返回 `true`；否则，返回 `false`。

<https://leetcode-cn.com/problems/palindrome-linked-list/>

```
/**
 * 请判断一个链表是否为回文链表。
 *
 * 借助栈结构，进栈出栈再判断一次
 */

public class LeetCode234 {
    public boolean isPalindrome(ListNode head) {
        if (head == null || head.next == null) {
            return true;
        }

        LinkedList<ListNode> stack = new LinkedList<>();
        ListNode cur = head;
        while (cur != null) {
            stack.push(cur);
            cur = cur.next;
        }

        while (cur != null) {
            if (stack.pop().val != cur.val) {
                return false;
            }
            cur = cur.next;
        }

        return true;
    }
}
```

```

        while (!stack.isEmpty()) {
            if
(!Integer.valueOf(stack.pop().val).equals(Integer.valueOf(head.val))) {
                return false;
            }
            head = head.next;
        }
        return true;
    }

}

```

LeetCode 283

问题：给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

<https://leetcode-cn.com/problems/move-zeroes/>

```

/**
 * 283. 移动零
 * 遍历数据，只要不为0，则交换到index
 * 从index开始，复制为0
 */

public class LeetCode283 {

    public void moveZeroes(int[] nums) {
        int index = 0;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] != 0) {
                nums[index] = nums[i];

```

```
        index++;
    }
}
for (int i = index; i < nums.length; i++) {
    nums[i] = 0;
}
}
}
```

LeetCode 448

问题：给你一个含 n 个整数的数组 nums，其中 nums[i] 在区间 [1, n] 内。请你找出所有在 [1, n] 范围内但没有出现在 nums 中的数字，并以数组的形式返回结果。

<https://leetcode-cn.com/problems/find-all-numbers-disappeared-in-an-array/>

```
/**
 * 448. 找到所有数组中消失的数字
 * (不能用额外的空间)
 *
 * 数字范围1~n，而数组下标0~(n-1)，则可用数组原地标记数字是否出现过
 *
 */
public class LeetCode448 {
    public static List<Integer> findDisappearedNumbers(int... nums) {
        List<Integer> res = new ArrayList<>();
        for (int i = 0; i < nums.length; i++) {
            int index = Math.abs(nums[i]) - 1;
            if (nums[index] > 0) {
                nums[index] = -nums[index];
            }
        }
        List<Integer> ans = new ArrayList<>();
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] > 0) {
                ans.add(i + 1);
            }
        }
        return ans;
    }
}
```

```
        }

    for (int i = 0; i < nums.length; i++) {
        if (nums[i] > 0) {
            res.add(i + 1);
        }
    }

    return res;
}

}
```

求关注我最近在更新的Java项目！从零更新，看完绝对有收获，点进Git仓库看看绝对不亏！

消息推送平台Gitee链接：<https://gitee.com/zongfucheng/austin>

消息推送平台GitHub链接：<http://github.com/ZhongFuCheng3y/austin>

03 LeetCode Medium

LeetCode 3

问题：给定一个字符串 `s`，请你找出其中不含有重复字符的 **最长子串** 的长度。

<https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/>

```
/***
 * 无重复字符的最长子串
 * (滑动指针)
 * 1. 使用字符数组 记录已遍历过的值
 */
```

```
* 2. 只要 r 未越界且 字符数组不存在该值，移动r 且 添加至 字符数组
* 3. 否则 移动l 且 删除 字符数组
* 4. result 获取每次移动后的最大值
* 5. 当字符遍历完成，返回result
*/
public class LeetCode3 {

    public int lengthOfLongestSubstring(String s) {

        int result = 0;
        int l = 0;
        int r = -1;

        char[] memo = new char[666];

        while (l < s.length()) {
            if (r + 1 < s.length() && memo[s.charAt(r + 1)] == 0) {
                memo[s.charAt(++r)]++;
            } else {
                memo[s.charAt(l++)]--;
            }
            result = Math.max(result, r - l + 1);
        }
        return result;
    }
}
```