Java复习要点整理

参考CyC2018、JavaGuide整理而成

Java基础

1. JVM、JDK、JRE

JVM是运行Java字节码文件（.class）的虚拟机，JVM有针对不同系统的实现，因此在不同平台下，相同字节码的运行结果相同（一次编译，处处执行）。Java程序通过编译器（javac）编译为字节码文件再经过JVM转变为机器可执行，由于字节码文件到机器码的过程是通过解释器逐行解释执行，所以速度比较慢。JVM引入了JIT编译器，对于热点代码执行编译之后，会将机器码保存，下次可直接使用。对于只执行一次的代码，解释器执行比JIT编译器要快，因此并不是所有地方都使用JIT编译器。

JDK是Java开发程序包，用有JRE、编译器（javac）和其他工具，能够创建和编译程序。

JRE是Java运行时环境，用来运行已经编译的Java程序，包括JVM、Java类库、Java命令和其他一些基础构建。

2. Java基本类型和包装类型

基本类型：

- byte/8
- char/16
- short/16
- int/32
- float/32
- long/64
- double/64
- boolean/1

包装类型：

- Byte
- Character
- Short
- Integer
- Float
- Long
- Double
- Boolean

自动装箱：

```
Integer b = 1;  #执行Integer.valueOf(1);
```

自动拆箱：

```
Integer a = new Integer(1);
int b = a;  #执行Integer.intValue()
```

不能执行隐式的向下转换（除 += 、*=、-=、\=、++）

3. 缓存池

Integer会缓存-128~127的值，调用Integer.valueOf(123)时会从缓存池中取对象

```
Integer x = new Integer(123);
Integer y = new Integer(123);
System.out.println(x == y);    // false
Integer z = Integer.valueOf(123);
Integer k = Integer.valueOf(123);
System.out.println(z == k);   // true
```

基本类型对应的缓冲池如下：

- boolean values true and false
- all byte values
- short values between -128 and 127
- int values between -128 and 127
- char in the range \u0000 to \u007F

3.String、StringBuilder、StringBuffer

String是不可变类型，被声明为final不可继承，内部使用char数组存储

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];
}
```

StringBuilder可变，线程不安全，性能比StringBuffer要好

StringBuffer可变，线程安全，内部用synchronized同步

字符串常量池

字符串常量池用来保存所有字符串字面量，字面量在编译时已经确定，可以使用intern方法，获取常量池中对象，若常量池中字符串值存在，则返回常量池中对象引用，否则，会先创建对象，在返回该对象引用。

```
String s1 = new String("aaa");
String s2 = new String("aaa");
System.out.println(s1 == s2);           // false
String s3 = s1.intern();
String s4 = s1.intern();
System.out.println(s3 == s4);           // true

String s5 = "bbb";
String s6 = "bbb";
System.out.println(s5 == s6);  // true
```

String a = new String("abc")创建几个对象

若常量池中没有字面量为“abc”的字符串对象，会创建2个对象，分别是常量池中字符串对象和堆中的字符串对象。若常量中存在，则只会创建堆中的字符串对象。

4. 继承

抽象类

抽象类中的抽象方法用abstract修饰，抽象类不可以被实例化，抽象类是类的模板设计，is a的关系

接口

接口中方法和字段默认为public修饰（不可以用private、protected修饰），字段默认都是static final，jdk1.8中允许接口中拥有默认的方法实现（default修饰），一个类只能继承一个父类，但可以实现多个接口。接口累死与一种like a的关系，是一种行为的抽象。优先使用接口。

重载和重写

重载发生在同一个类中， 参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。

重写发生在父类的子类中， 参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为 private 则子类就不能重写该方法。

super

- 调用父类构造函数

  当父类只有有参数的构造函数时，子类构造函数中必须显示调用父类的有参构造函数。

- 调用父类方法

  super.xxx()

this

- 类的当前实例
- 内部类用外部类的成员时

  外部类名.this.xxx

static

- **修饰成员变量和成员方法:** 被 static 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享，可以并且建议通过类名调用。被static 声明的成员变量属于静态成员变量，静态变量

存放在 Java 内存区域的方法区。调用格式：`类名.静态变量名` `类名.静态方法名()`

- **静态代码块:** 静态代码块定义在类中方法外, 静态代码块在非静态代码块之前执行(静态代码块—>非静态代码块—>构造方法)。 该类不管创建多少对象，静态代码块只执行一次.
- **静态内部类（static修饰类的话只能修饰内部类）：** 静态内部类与非静态内部类之间存在一个最大的区别: 非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围类，但是静态内部类却没有。没有这个引用就意味着：1. 它的创建是不需要依赖外围类的创建。2. 它不能使用任何外围类的非static成员变量和方法。
- **静态导包(用来导入类中的静态资源，1.5之后的新特性):** 格式为：`import static` 这两个关键字连用可以指定导入某个类中的指定静态资源，并且不需要使用类名调用类中静态成员，可以直接使用类中静态成员变量和成员方法。

final

- 修饰数据，则为编译时常量，对于基本类型，数值不变，对于引用类型，不能指向其他对象
- 修饰方法，不能被子类重写
- 修饰类，不能被继承

5. equals()和hashCode()

- 对于基本类型，==判断值是否相等，无equals()。
- 对于引用类型，==判断两个变量是否引用同一对象，equals()判断引用的对象是否等价（若没有重写和==一样）
- 重写equals()，必须重写hashCode()
  equals()的对象，散列值一定相等，反之不一定。
  在set中判断是否重复首先比较hashCode()，若相等再调用equals()。

6. Java序列化

将对象转换为字节序列便于存储传输，需要实现Serializable接口

- 序列化：ObjectOutputStream.writeObject()
- 反序列化：ObjectInputStream.readObject()

使用transient可以使一些关键字不被序列化

7. 反射

获取运行时的类信息，在运行时获取类的所有字段和方法，对象类型在编译期时是未知的，在运行时可以通过反射机制直接创建对象。

使用场景：JDBC连接数据库时使用Class.forName()来加载数据库的驱动程序。Spring框架的xml配置模式

8. 异常

Throwable分为两类，Error和Exception，Error表示不能被JVM处理，Exception可以被捕获并处理

9. 注解

附加在代码中的一些元信息，用于一些工具在编译、运行时进行解析和使用，起到说明、配置的功能。注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。

10. jdk8新特性

- Lambda表达式
- Stream函数式操作流元素集合
- 接口新增：默认方法与静态方法
- 方法引用,与Lambda表达式联合使用
- 引入重复注解
- 类型注解
- 最新的Date/Time API (JSR 310)
- 新增base64加解密API
- 数组并行（parallel）操作
- JVM的PermGen空间被移除：取代它的是Metaspace（JEP 122）元空间

Java容器

Collection

1. Set

- TreeSet:基于红黑树实现，支持有序性，查找插入均为O(logN)
- HashjSet:基于哈希表实现，不支持有序性，丢失插入顺序，查找效率O(1)
- LinkedHashSet:具有(1)查找效率，内部使用双向链表维护插入顺序

2.List

- ArrayList:基于动态数组实现，支持随机访问
- Vector:与ArrayList类似，线程安全
- LinkedList:基于双向链表实现，只能顺序访问，插入删除元素快

3.Queue

- LinkedList: 用它做双向队列
- PriorityQueue:基于堆结构实现，优先级队列

Map

- HashMap:基于数组+链表（红黑树）实现
- TreeMap:基于红黑树实现
- LinkedHashMap:使用双向链表维护插入顺序
- HashTable:线程安全，使用同一把锁，应用CocurrentHashMap替代

源码分析

List

1.ArrayList

- 实现了RandomAccess接口支持随机访问
- 数组的默认大小为10
- 添加元素时使用 ensureCapacityInternal() 方法来保证容量足够，不够时，使用grow()进行扩容，使用Arrays.copyOf()进行复制，新容量是旧容量的1.5倍。
- Fail-Fast迭代器，用modCount记录结构发生变化的次数（添加或删除元素），在迭代时需要比较操作前后的modCount是否改变，若改变抛出异常。

2.Vector

- 内部所有方法用synchronized同步，每次扩容容量为原来的2倍

3.CopyOnWriteArrayList

- 读写分离，写时复制原数组，写操作需要加锁，写入完成后指向新数组
- 读操作不能读取到实时性的数据，写操作的数据还未同步
- 内存占用大，每次写都需要复制原数组
- 适用于读多写少的场景

4.LinkedList

- 内部使用双向链表实现
- 不支持随机访问，插入删除元素快

Map

1.HashMap

- 内部基于数组（Node类型）+链表实现，jdk8之后当链表长度大于等于8之后会转变为红黑树
- HashMap数组长度为2的次幂，默认为16，因为桶下标为hashCode % length，若length为2的次幂则可以转变为位与运算x & (length - 1)，效率大大提升
- 当元素个数大于threshold时，需要进行扩容，新容量为旧容量的2倍，建立新数组，复制
- jdk8之前是链表是使用头插法，会造成并发插入导致扩容的情况下会造成死循环导致CPU100%，jdk8之后使用尾插法解决了这个问题
- 可以有null键

2.CocurrentHashMap

- jdk8之前使用分段锁，每个锁维护一部分桶，多个线程可以访问不同分段锁上的桶，默认并发级别是16， segment锁继承自Reentrantlock
- 执行size操作时会尝试不加锁，若两次不加锁得到结果一致则正确，若尝试次数超过3次，则对每个Segment加锁
- jdk8 之后使用CAS加内部synchronized（只锁定当前链表和红黑树首节点）实现

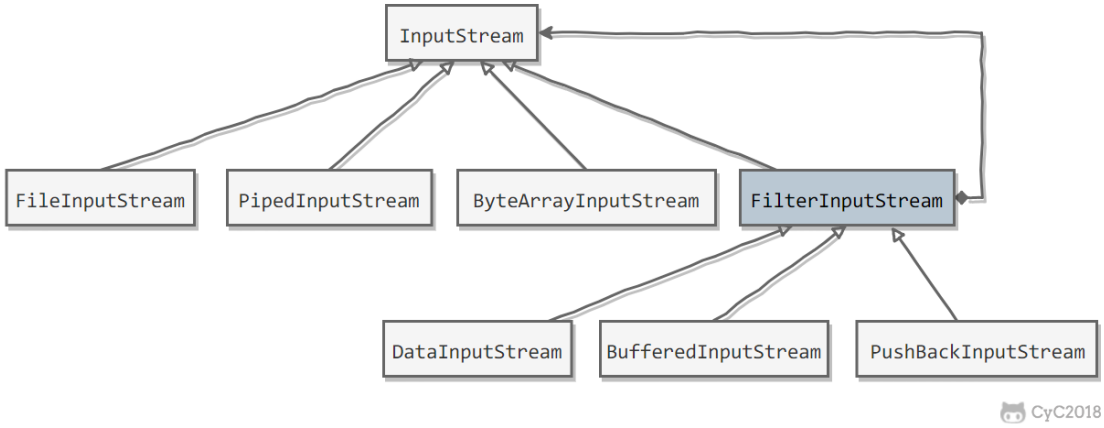
3.HashTable

- 内部使用数组加链表实现
- 使用的是同一把锁，效率低
- 不能插入null键
- 初始容量为11， 每次扩容2n + 1

Java IO

java IO内部使用了装饰者模式。

- InputStream 是抽象组件；
- FileInputStream 是 InputStream 的子类，属于具体组件，提供了字节流的输入操作；
- FilterInputStream 属于抽象装饰者，装饰者用于装饰组件，为组件提供额外的功能。例如 BufferedInputStream 为 FileInputStream 提供缓存的功能。



实例化一个具有缓存功能的字节流对象时，只需要在 FileInputStream 对象上再套一层 BufferedInputStream 对象即可。

```
FileInputStream fileInputStream = new FileInputStream(filePath);
BufferedInputStream bufferedInputStream = new
BufferedInputStream(fileInputStream);
```

DataInputStream 装饰者提供了对更多数据类型进行输入的操作，比如 int、double 等基本类型。

BIO

同步阻塞IO，数据的读取写入必须阻塞在一个线程内等待其完成，一请求一应答，在接收到客户端的连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回应答给客户端，线程销毁。采用 线程池实现，可以改善线程创建和回收的成本，有效控制线程最大数量。

NIO

同步非阻塞IO， 对应 java.nio 包，提供了 Channel , Selector，Buffer等抽象。 NIO提供了与传统BIO模型中的 `Socket` 和 `ServerSocket` 相对应的 `SocketChannel` 和 `ServerSocketChannel` 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发。

NIO与BIO的区别

- BIO是阻塞的，NIO是非阻塞的
- BIO是面向流的，NIO是面向缓冲区的（NIO读取写入都要通过Buffer）
- NIO通过Channel进行读写，Channel是双向的，而流是单向的
- NIO有选择器，IO没有，选择器用于单线程处理多个通道

IO多路复用

多个网络连接复用一个线程

selector、poll和epoll

- select，poll实现需要自己不断轮询所有fd集合，直到设备就绪 ， epoll采用了回调机制
- select，poll每次调用都要把fd集合从用户态往内核态拷贝一次 ， 而epoll只需要一次拷贝
- epoll数据结构是红黑树和双向链表

AIO

异步非阻塞IO，基于事件和回调机制，应用不广泛

JVM

1.Java内存区域

线程私有：

程序计数器：计数器的值用于选取下一条要执行的字节码指令

本地方法栈：为Naive方法服务

JVM栈：存放局部变量，调用方法时形成栈帧压入栈中

线程共享：

堆：存放对象实例

方法区：类信息、类静态变量、常量、JIT编译后的代码，jdk8之后放入元空间（使用直接内存）

运行时常量池：在方法区中，存放Class文件中的常量池（编译器生成的字面量和符号引用）

其他：

直接内存：不是运行时数据区一部分，NIO使用Native函数直接分配堆内存，避免了在Java堆和Native堆中来回复制数据

2. 创建对象

- 类加载检查：检查常量池中是否有该类的符号引用，检查该类是否被加载解析初始化，若没有进行类加载过程
- 分配内存：内存规整用指针碰撞，不规整用空闲列表
- 初始化零值：保证实例字段不赋值就可以直接使用（不包括对象头）
- 设置对象头：对象头中存放类的元数据信息，哈希吗、GC分代年龄、偏向锁、类型指针等
- 执行init方法：执行类的构造器方法

3. 对象的内存布局

- 对象头
- 实例数据
- 对齐填充
- 对象访问定位：使用句柄和直接指针

4. Java垃圾收集

判断对象是否可以回收

- 引用计数（由于循环引用不使用）
- 可达性分析（以GC Roots为起点进行搜索，能够到达的对象是存活的）

  GC Roots:

  1. JVM栈中变量引用的对象
  2. 本地方法栈中引用的对象
  3. 方法区中静态变量引用的对象
  4. 方法区中常量引用的对象

引用

1. 强引用：不会被回收
2. 软引用：若内存足够不会被回收
3. 弱引用：在下一次垃圾收集时被回收
4. 虚引用：在对象被回收时收到一个通知

对象回收前进行两次标记，第一次是可达性分析，第二是finalize方法中，若可以与引用链上对象建立关联，则存活

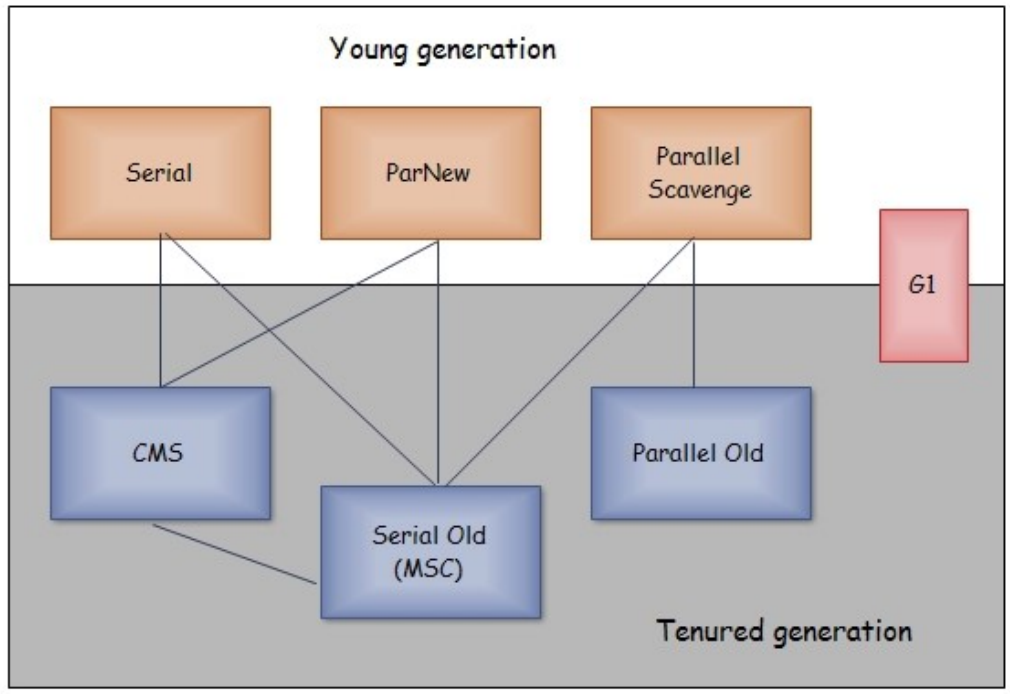
回收方法区

1. 回收常量：没有任何对象引用常量池中字面量
2. 回收无用的类：不存在任何类的实例、Classloader被回收、Class对象没有任何引用，无法通过反射机制访问

5.垃圾回收算法

- 标记-清除算法：对需要回收的对象进行标记，然后进行回收，会产生碎片
- 标记-整理算法：标记后，将存活对象移动至一端，直接清除之后的内存
- 复制算法：分为eden和survivor，每次使用其中一块，将存活对象复制至另外一块，交换。现在常用eden:survivor:survivor = 8:1:1，分配担保机制保证当survivor空间不足时，直接进入老年代
- 分代收集算法：对于新生代，只有少量对象存活，采用复制算法，对于老年代，存活率高，没有额外空间担保，采用标记-清除和标记-整理算法。

6. 垃圾收集器



Serial收集器

单线程，新生代复制，老年代标记-整理，暂停所有用户线程

ParNew收集器

多线程，新生代复制，老年代标记-整理，暂停所有用户线程

Paraller Scavenge收集器

多线程，注重吞吐量（CPU用于用户程序运行的时间占总时间的比值）

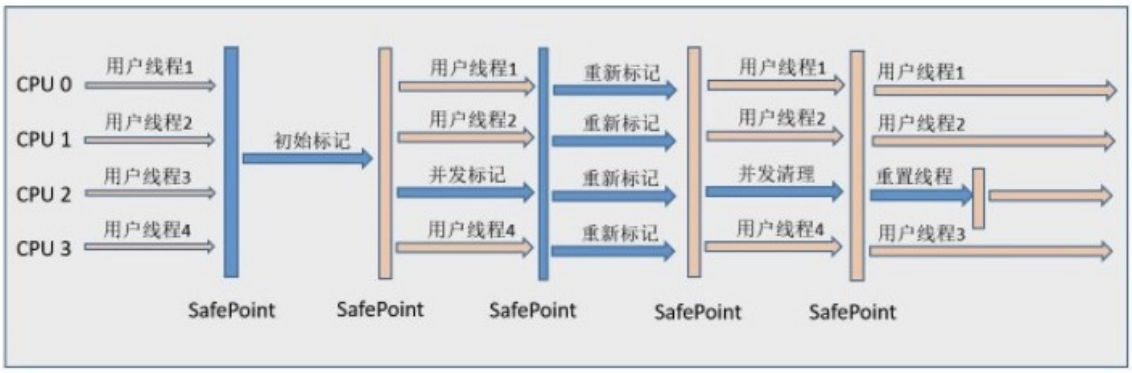
Serial Old收集器

Serial的老年代版本，作为CMS的后备方案，在并发收集发生Cocurrent Mode Failure时使用

Parallel Old收集器

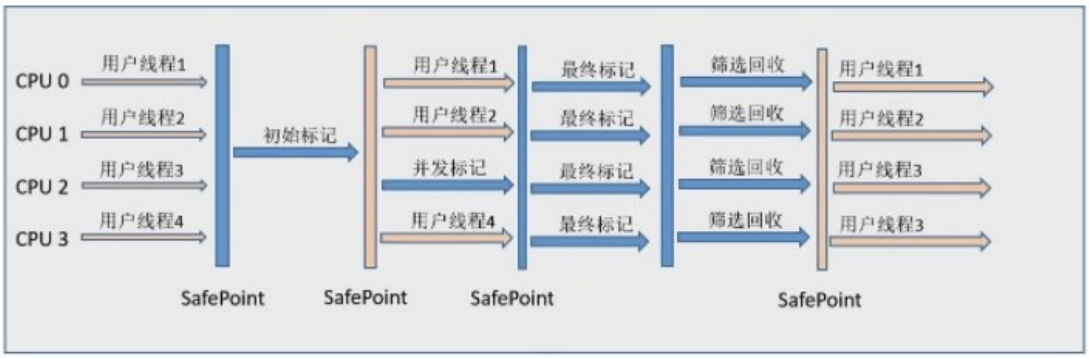
Paraller Scavenge的老年代版本

CMS收集器



- 初始标记：暂停所有线程，快速标记
- 并发标记：同时开启GC和用户线程
- 重新标记：标记那些并发标记过程中因为用户线程变动的对象，需要停顿
- 并发清理：开启用户线程和GC线程进行清理
  优点：并发收集，停顿时间短
  缺点：对CPU资源敏感（吞吐量低），无法处理浮动垃圾（并发清除阶段产生的垃圾，因此需要预留一部分内存，若内存，标记-清除算法会产生碎片

G1收集器



收集范围是整个老年代和新生代，把堆划分为大小相等的独立区域，新生代和老年代一起回收，记录每个区域垃圾回收时间和回收所获得的空间，维护一个优先列表，每次根据允许的收集时间，回收价值最大的区域

- 初始标记
- 并发标记
- 最终标记：记录在并发标记中产生变动的部分，记录在remember set logs中，合并到Remember Set中，需要停顿线程，可并行执行
- 筛选回收：对各Region回收成本和价值排序，根据用户期望GC停顿时间制定回收计划
  空间整合：整体上看是标记-清理，局部来看是复制算法，运行期间不产生碎片
  可预测的停顿：能让使用者明确指定一个长度为M毫秒的时间片段内，消耗在GC上的时间不超过N秒。

7. 内存分配策略

- 对象有现在eden分配
- 大对象直接进入老年代
- 长期存活的对象进入老年代
- 当相同年龄的对象大于survivor空间一半，大于等于该年龄的进入老年代

- 空间分配担保

8. Full GC的触发条件

- System.gc()
- 老年代空间不足
- Cocurrent Mode Failure
- 空间担保失败

9.常用的JDK命令行工具

- jps：查看Java进程
- jstat：监视运行时信息
- jinfo：实时查看虚拟机参数
- jmap：生成堆存储快照
- jhat：分析heapdump文件
- jstack：生成当前时刻的线程快照

10.类加载机制

类编译过程

- 词法分析，输入到符号表
- 注解处理
- 语义分析，生成字节码

类加载过程

- 加载：获取二进制字节流、将静态存储结构转变为方法区运行时数据结构，内存中生成Class对象，作为访问入口
- 验证：验证该二进制字节流是否符合虚拟机标准
- 准备：类变量分配内存并初始化为零值（常量直接为该值）
- 解析：将常量池中符号引用转变为直接饮用
- 初始化：执行clinit，类变量赋予初始值，执行静态语句块

执行初始化的情况：

- 遇到new，读取（修改）静态字段、调用静态方法
- 反射调用时
- 子类初始化时，先初始化父类
- 虚拟机启动时，初始化主类
- jdk1.7，如果一个MethodHandle实例解析后的句柄未初始化，则需要先触发初始化

11.类加载器

- 启动类加载器：加载lib下的jar包
- 扩展类加载器：加载lib/ext下的jar包和类
- 应用类加载器：加载classpath下的jar包和类

双亲委托模型

当进行类加载时，先委托父类加载器进行类加载，若父类加载无法进行类加载，再自己进行类加载

优点：
使得类具有类加载器的优先级的层次结构，基础类得到统一，避免类的重复加载，保证核心API不被篡改

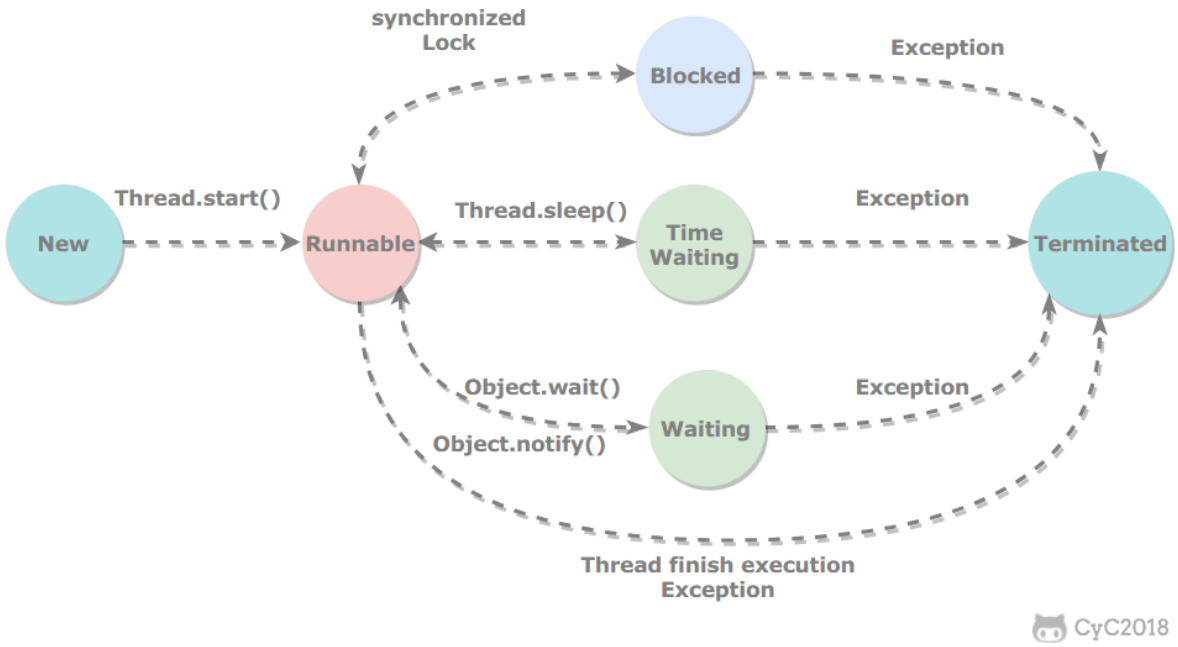
破坏：

自定义类加载器，重写loadclass()方法

Java并发

1. 线程的状态

- 新建
- 可运行
- 阻塞
- 无限期等待
- 限期等待
- 终止



2. 使用线程

- 继承Thread类
- 实现Runnable接口（无返回值）
- 实现Callable接口（有返回值），可以配合FutureTask使用

实现接口更好，Java不支持多重继承，但可以实现多个接口
继承Thread类开销太大

3. 基础线程机制

Executors

管理多个异步任务,主要有三种:

- CachedThreadPool
- FixedThreadPool
- SingleThreadPool

Daemon

守护线程是在后台提供服务的线程，当所有非守护线程结束时，程序终止，杀死所有守护线程

sleep

休眠当前线程，可能会跑出异常，因为异常不能跨线程传播，因此必须在本地处理。

yield

声明当前线程已经完成最重要的部分，可以进行切换

4. 中断

- 线程处于限期等待、无限期等待、阻塞，可以调用interrupt()进行中断，抛出异常，但不能中断IO阻塞和synchronized锁阻塞
- interrupt()方法会设置线程的中断标记（即使不能中断），调用interrupted()方法返回true
- Executors的shutdownNow方法相当于调用每个线程的interrupt()方法
- 可以使用submit方法返回FutureTask对象，调用cancel(true)来中断

5. 互斥同步

synchronized

- 同步代码块：monitorenter和monitorexit，实质上是获取monitor的持有权
- 同步方法：ACC_SYNCHRONIZED标志
- 同步类对象（静态方法）

ReentrantLock

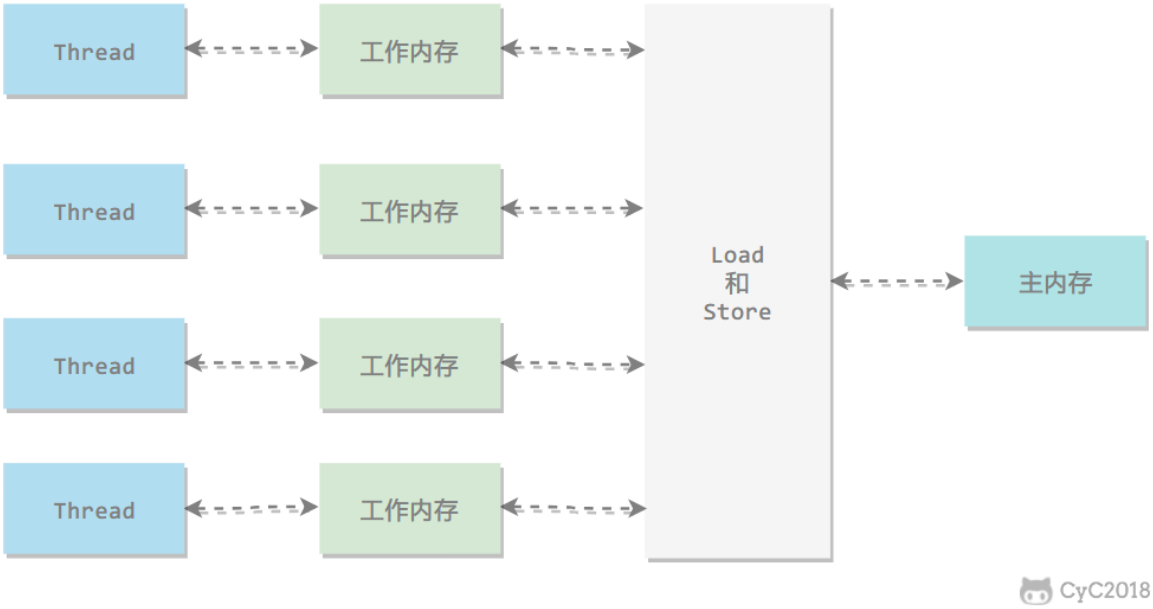
- JUC中的锁，lock.lock()上锁，finally中lock.unlock()

两者的区别

- synchronized是JVM实现，ReentrantLock是jdk实现
- synchronized等待不可中断，ReentrantLock等待可中断
- synchronized非公平锁，ReentrantLock支持公平锁（按请求锁的顺序获得锁）和非公平锁
- synchronized优化后（无锁，偏向锁，轻量级锁，重量级锁）和ReentrantLock性能差不多
- ReentrantLock支持绑定多个条件

6. Java内存模型

- 为了屏蔽各种硬件和操作系统的差异，实现Java程序在各平台达到一致访问效果
- 所有变量储存在主内存中，每个线程有自己的工作内存（高速缓存或寄存器中），线程只能读写自己的工作内存，不同线程之间的变量传递需要通过主内存完成。



三大特性

- 原子性：load、assign、store具有原子性（对于64位数据读写分为两次32位操作进行）
- 可见性：一个线程修改了共享变量的值，另一个线程能立即得知修改
- 有序性：从本线程内看，所有操作有序，在一个线程观察另外一个线程，所有操作是无序的（指令重排序），volatile通过添加内存屏障禁止指令重排序

happensbefore原则

前一个操作的结果对后一个操作可见，第一个操作的执行顺序在第二个操作之前。（重排序之后的结果与按happens-before一致，重排序不非法）

- 单一线程规则：单一线程中，每个操作happens-before任意后续的操作
- 管城锁定规则：lock的解锁happens-before加锁
- volatile变量规则：volatile写happens-before volatile读
- 传递性：A happens-before B ， B happens-before C， A happens-before C
- start()规则：start() happens-before 任意线程中的操作
- join()规则：A 中调用B.join()，B中所有操作happens-before A从B中返回
- 线程中断规则：调用interrupt() happens-before 检测到中断事件的发生
- 对象终结规则：对象初始化happens-before finalize()方法

7. volatile

- 保证可见性，读取变量时总是从主内存刷新到工作内存读取，写入变量时，工作内存立刻刷新到主内存
- 不保证原子性

volatile和synchronized区别

- volatile是线程同步的轻量级实现，性能比synchronized好，但只能用于变量
- 多线程访问volatile变量不会阻塞
- volatile只保证可见性，synchronized保证可见性和原子性
- volatile只要解决多个线程之间的可见性，synchronized解决多个线程之间的同步性

8. 线程间协作

join()

在线程a中调用b.join()，a线程挂起，等b线程返回后执行a线程

wait()notify()notifyAll()

wait使线程等待某个条件满足，线程在等待时挂起，当条件满足时其他线程调用notify()或者notifyAll()唤醒线程

wait()和sleep()的区别

- wait()属于Object,sleep()属于Thread的静态方法
- wait()挂起会释放锁，sleep()不会释放锁

await()signal()signalAll()

使用Lock获取condition对象，await()指定等待的条件，更加灵活

9. 非阻塞同步

CAS

比较并交换，CAS有三个操作数，内存地址V，旧的预期值A和新值B，只有当V的值等于A，才更新为B

问题：

- ABA问题：A更新为B后又更新为A，解决方案是添加版本号
- 循环时间过长：自旋CAS长时间不成功，降低CPU效率
- 只能保证一个变量的原子操作：JDK提供了引用类型原子类

原子类

- 基本类型
- 数组类型
- 引用类型
- 对象属性修改类型

10. 悲观锁和乐观锁

- 悲观锁：认为不采用同步措施，会出现问题，无论共享数据是否出现竞争，都会进行加锁，synchronized和ReentrantLock属于悲观锁，使用写多的场景
- 乐观锁：先操作，再进行冲突检测，认为拿数据的时候，别人都不会修改，所以不会上锁，只是在更新的时候判断是否有人更新这个数据，（非阻塞，提高吞吐量），原子类（基于CAS实现）属于乐观锁，适用于读多写少的场景

11. 无同步方案

- 栈封闭：使用局部变量
- ThreadLocal：每个线程内部维护一个ThreadLocalMap, ThreadLocal对象为键，值为该对象的值，可能出现内存泄漏，键为弱引用，垃圾回收时被回收后，key为null，value无法回收，手动调用remove()，清理键为null的记录
- 可重入代码：在任何时刻中断，转而执行另外一段代码，返回后不出现任何错误

12. 线程池

优点：

- 降低资源消耗：降低创建和销毁线程的资源消耗
- 提高响应速度：任务到达时，不用等待线程创建
- 提高线程的可管理性：可以对线程进行统一分配、调优和监控

ThreadPoolExecutor参数：

- corePoolSize：核心线程数，当新任务到达时，当前线程数小于核心线程数时，会新建线程
- maximumPoolSize：最大线程数，当工作队列已满时，并且已创建的线程数小于最大线程数，继续创建新线程，当前可同时运行线程数变为最大线程数
- workQueue：阻塞队列，当前运行线程数等于核心线程数，则将任务放入工作队列中
- ArrayBlockingQueue: 基于数组实现的有界阻塞队列
- LinkedBlockingQueue: 基于链表实现的阻塞队列
- SynchronousQueue:不储存元素的阻塞队列
- PriorityBlockingQueue:具有优先级的无限阻塞队列
- keepAliveTime:当前线程数大于核心线程数，若没有新任务提交，等到时间超过该值，被回收销毁
- unit:时间单位
- threadFactory:创建线程的工厂
- handler：饱和策略，当线程数等于最大线程数，工作队列已满时
    - AbortPolicy：抛出异常，拒绝新任务
    - CallerRunsPolicy：调用执行自己的线程处理任务，会降低新任务的提交速度，影响整体程序性能，会增加工作队列长度，若可以承受此延迟可以使用此策略
    - DiscardPolicy：不处理新任务，直接丢弃
    - DiscardOldestPolic：丢弃最早未处理的任务

FixedThreadPool

核心线程线程数和最大线程数一样，阻塞队列无界（队列容量Integer.MAX_VALUE)，请求堆积太多，发生OOM

SingleThreadExecutor

核心线程数和最大线程数均为1，使用无界队列（队列容量Integer.MAX_VALUE)，请求堆积太多，OOM

CachedThreadPool

核心线程数为0， 最大线程数Integer.MAX_VALUE，阻塞队列使用无容量的阻塞队列，创建的线程太多发生OOM

推荐使用ThreadPoolExecutor()构造函数构造线程池

13.锁优化（主要针对synchronized）

自旋锁

互斥同步进入阻塞状态，需要用户态与核心态的切换，十分消耗CPU性能，对于共享数据只锁定很短一段时间的情况，可以让一个线程在请求锁时执行忙循环（自旋），如果在这段时间获得锁，则不用进入阻塞状态。

若自旋时间过久，容易占用CPU时间，jdk1.6引入自适应自旋锁，自旋次数不再固定，由上一次获得锁的自旋次数和锁的拥有者状态决定

锁消除

对不存在竞争的共享数据的锁进行消除，主要通过逃逸分析来支持

锁粗化

增加上锁代码的范围，避免频繁对同意对象加锁解锁

偏向锁

对于第一个获取该对象锁的线程，无需进行同步操作，甚至不需要CAS操作，在对象头中设置偏向锁，当有另外一个线程尝试获取这个锁对象时，偏向锁状态结束

轻量级锁

使用CAS操作避免重量级锁使用互斥量的开销

14. AQS

队列同步器是构造锁和同步器的框架，若请求的共享资源处于空闲状态，就将当前线程设置有效的工作线程，将共享资源的状态设置为锁定状态，若请求的资源被占用，则需要一套线程阻塞等待以及被唤醒的机制，该机制是有CLH锁队列（虚拟的双向队列，只有节点之间的关联关系，不存在队列实例）实现的，将获取不到锁的线程放入队列中。

使用一个int变量state表示同步状态，有两种资源共享方式：

- 独占，如ReentrantLock，分为公平锁和非公平锁
- 共享，Semaphore，CountDownLatch，CyclicBarrier等

自定义的同步器的实现， 继承AbstractQueuedSynchronizer ，只需实现共享资源state状态的获取与释放方式即可。

Semaphore

- 允许多个线程同时访问
- acquire()获取一个许可证，release()释放一个许可证，也可以一次获取或释放多个许可证
- 有公平模式和非公平模式

CountDownLatch

一个线程或者多个线程一直等待，直到其他线程执行完成后再执行，只能使用一次

用法：

- 某一线程在开始运行前等待n个线程执行完毕
- 实现多个线程开始执行任务的最大并行性（共享一个CountDownLatch对象设置为1，当主线程调用countDown()，多个线程被同时唤醒
- 死锁检测

CyclicBarrier

循环屏障，所有线程都到达屏障时，一起被执行，用于多线程计算数据，提供了reset功能，可以多次使用。

操作系统

1 进程管理

并发与并行

并发是指多个进程指令被快速切换运行，在宏观上表现出一段时间内可以同时运行多个程序，微观上不是

并行是指同一时刻多条指令同时执行，需要多个CPU，并行是并发的真子集

进程、线程与协程

进程是资源分配的基本单位，是程序运行的实例

线程是独立调度的基本单位，一个进程中可以有多个线程，线程共享进程资源。

协程是用户态轻量级线程，协程的调度由用户控制，协程调度时，将上下文和栈保存在其他地方，切换时不需要内核切换的开销，不可以不加锁访问全局变量，进程和线程是同步机制，协程是异步。

进程与线程的区别：

- 线程不拥有资源，可以访问隶属于进程 的资源
- 同一进程中线程切换不会引起进程的切换，不同进程中线程切换会引起进程的切换
- 创建和撤销线程的系统开销远大于创建和撤销线程，线程又称为轻量级进程
- 线程间通信可以直接读写同一进程中的数据进行通信，而进程间通信需要借助IPC

进程间通信

- 管道，通过pipe函数创建，只支持半双工，只能在父子进程和兄弟进程中使用
- FIFO（命名管道），用作汇聚点
- 消息队列，可以独立于读写进程存在，避免了同步阻塞问题，可以根据消息类型接受消息
- 信号量
- 共享储存

- 套接字，用于不同机器间的进程通信

线程间通信

主要用于线程同步，无数据交换

- 锁机制
- 信号量机制
- 信号机制

进程调度算法

- 批处理系统（保证吞吐量和周转时间）：先来先服务、短作业优先、最短剩余时间优先
- 交互式系统（保证响应时间）：时间片轮转、优先级调度、多级反馈队列
- 实时系统

2. 死锁

死锁条件

- 互斥
- 不可抢占
- 占有和等待（占有锁时可以请求另一个锁）
- 环路条件

处理方法

- 鸵鸟策略
- 死锁检测和恢复
- 死锁预防
- 死锁避免

死锁检测

图算法（dfs），是否有环路

死锁恢复

- 回滚
- 抢占
- 杀死进程

死锁预防

- 破坏互斥条件
- 破坏占有和等待条件：进程开始前，一次获取所以的资源
- 破坏不可抢占条件
- 破坏环路等待：给资源同一编号，只能按编号获取资源

死锁避免

- 安全状态
- 银行家算法

3. 内存管理

- 虚拟内存的目的是将物理内存扩充为更大的逻辑内存，从而让程序获得更多的可用内存
- 每个进程拥有独立的地址空间，地址空间被分割为大小相等的块，称为页，这些页映射到物理内存
- 内存管理单元（MMU）管理着地址空间和物理内存的转换，其中的页表存储着页（地址空间）和页框（物理内存）的映射表

页面置换算法

- 最佳：换出页面是最长时间内不再被方位
- LRU、NRU（N,M）
- 先进先出（未考虑局部性原理）
- 第二次机会
- 时钟

段页式

程序的地址空间分为多个拥有独立地址空间的段，每个段上的地址空间划分为大小相同的页，既拥有分段的共享和保护，又拥有分页的虚拟内存功能

分段和分页

- 分段对程序员可见，分页透明
- 分页是一维地址空间，分段是二维
- 分页大小固定，分段大小可动态增长
- 分页只要用于扩充物理内存为更大的逻辑内存，分段用于数据的共享和保护

计算机网络

- 五层协议：应用层、传输层、网络层、链路层、物理层
- TCP/IP四层协议：应用层、传输层、网络层、链路层
- OSI七层协议：应用层、表示层、会话层、传输层、网络层、链路层、物理层

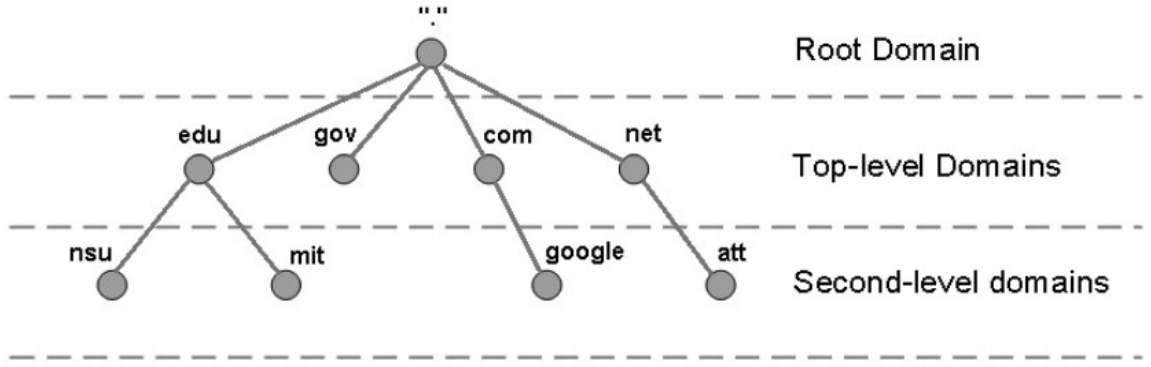
表示层：数据压缩、加密以及数据描述，使得应用程序不必关心在各台主机上数据内部格式的不同

会话层：建立及管理会话

1. 应用层（DNS、HTTP、SMTP、FTP、DHCP、POP3、IMAP、TELNET）

DNS协议

DNS是分布式系统，提供IP地址和域名的转换服务，域名具有层次结构，从上到下依次为：根域名、顶级域名、二级域名



DNS可以用TCP或者UDP传输,端口号为53,大多数使用UDP,需要服务器自己处理超时和重传从而保证可靠性,两种情况使用TCP传输:

- 返回的响应超过512字节
- 区域传送

FTP协议

FTP使用TCP传输,控制连接服务器打开端口21进行等待客户端连接, 客户端主动建立连接后,使用该连接将客户端命令传送给服务器,并传回服务器端的应答.数据连接,端口20(主动模式),传送文件数据

根据服务器端是否主动建立数据连接分为主动模式和被动模式

DHCP协议

配置IP地址,网关路由器的IP地址,DNS地址,子网掩码

远程登录协议

TELNET用于登录远程主机,可适应多计算机和操作系统的差异

电子邮件协议

- 发送协议:SMTP
- 读取协议:POP3、IMAP

POP3读取后，从邮件服务器删除，IMAP不会

常用端口

协议	端口	传输层协议
DNS	53	UDP/TCP
HTTP	80	TCP
FTP	21（控制）/20（数据）	TCP
SMTP	25	TCP
POP3	110	TCP
IMAP	143	TCP
TELNET	23	TCP
DHCP	67/68	UDP

Web页面请求过程

1. 使用DHCP配置主机信息，返回IP地址、DNS地址、网关地址、子网掩码（涉及UDP）
2. ARP协议获取网关路由器的MAC地址
3. DNS将域名转换为IP地址，其中涉及路由选择协议（OSPF），通过UDP传输
4. 建立TCP连接，发送HTTP GET报文，收到响应后显示HTML页面

涉及的协议：

- DHCP
- UDP
- ARP
- DNS
- TCP
- HTTP
- IP

2. 传输层（TCP、UDP）

UDP与TCP的区别

- UDP无连接，尽最大可能交付，不提供拥塞控制、流量控制，面向报文，支持一对一，一对多，多对多，用于语音、直播、DNS等
- TCP有连接，提供可靠的交付，有流量控制、拥塞控制，提供全双工通信，面向流，一对一，用于文件传输、邮件发送、远程登录等

TCP粘包

发送方发送的若干数据包到达时粘成了一包，后一个包的头紧接着前一个包的尾

原因：

- 发送方：使用了Nagle算法，将多次间隔较小、数据量较小的数据合并为一个数据量大的数据块进行了分包

- 接收方：从接受到缓存的速度大于程序从缓存中读取的速度，多个包被缓存，程序读到首尾相连的包

解决方法（多个包不相关，甚至并列）：

- 发送方：关闭Nagle算法
- 接收方：无法处理，交给应用层处理，应用层循环读取所有数据，判断开始结束位置

报文首部

- TCP 源端口号、目标端口号、序号、确认号、数据偏移、确认ACK、同步SYN、终止FIN、窗口
- UDP 8 字节 源端口、目标端口、长度、检验和

点对点基于IP地址或者MAC地址，端对端是通过TCP或者UDP，从发送端到接收端，其中跨越许多节点

三次握手

- 首先B处于监听状态，A发送SYN = 1, ACK = 0，初始序号为x的请求包
- B返回SYN = 1, ACK = 1, ack = x + 1, 序号为y的确认包
- A收到确认报文后，发送ACK = 1, ack = y + 1，序列号为x + 1的包（可以携带数据）
  SYN = 1表示连接请求或接收报文，不能懈怠数据

为什么是三次？两次可以吗

不可以，若客户端发送的连接请求在网络中滞留，隔很长一段时间到达服务端，最终服务端的确认报文还是会到达客户端，若两次握手，会打开两个连接，浪费资源，产生错误，三次握手，客户端会忽略重复的确认报文

SYN攻击

伪造大量IP地址，短时间向服务器发送大量SYN包，由于源地址不存在，因此不断超时重新发送，导致网络堵塞

- 缩短SYN Timeout时间
- 设置SYN cookie

四次挥手

- A发送关闭连接报文，FIN = 1, 序号为x
- B收到后，发送确认报文，ACK = 1, ack = x + 1，序号y，此时TCP属于半关闭状态，B可以向A发送数据，A不可以向B发送数据。服务器端进入close wait
- 当B不要连接时，发送关闭连接报文，FIN = 1, ACK = 1, ack = x + 1，序号为v
- 客户端收到后进入time wait,发出确认，等待2 MSL后释放连接
- B收到A的确认后关闭连接

为什么四次挥手

当服务器收到客户端关闭连接报文时，进入close wait，此时为了发送完未传输完成的数据，传输完毕之后，服务器端发送FIN连接释放报文

Time_Wait的作用

- 确认最后一个确认报文到达，若B没收到A发送的确认报文，A会重新发送

- 为了使得本持续时间内所有报文从网络中消失，使得下一个连接中不会出现旧的连接请求报文

滑动窗口与流量控制

流量控制是为了控制发送方的发送速率，使得接收方来得及接接收，在确认报文中的窗口字段可以控制发送方窗口的大小

拥塞控制

慢启动和拥塞避免

初始拥塞窗口大小为1，收到确认后翻倍，当窗口大小超过慢开始门限，进入拥塞避免，每次窗口增加1，若超时，门限设为当前窗口大小的一半，窗口设为1

快重传和快恢复

连续收到三次重复的去人，立即重传下一个报文段，2次可能是乱序，3次效率最高，此时执行快恢复，令门限等于窗口大小的一半，窗口大小等于门限，进入拥塞避免

3. 网络层（ICMP、IP、OSPF）

子网划分

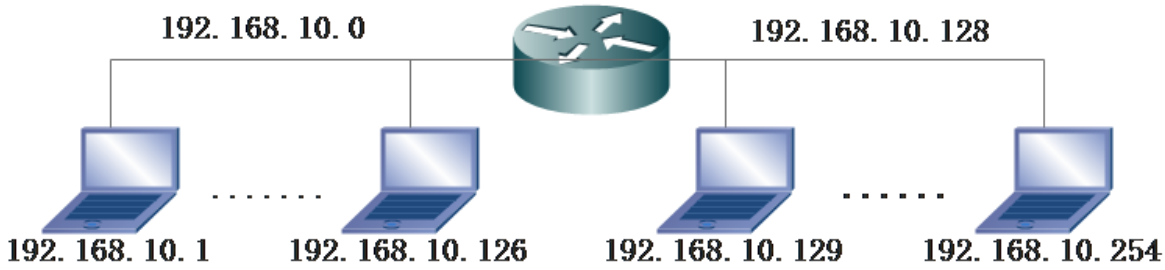
IP地址由网络号和主机号构成

子网掩码

用于知道哪一部分为子网地址

案列一：
255.255.255.128 (/25)
128的二进制表示为10000000，只有1位用于定义子网，余下7位用于定义主机。这里将对C类网络192.168.10.0进行子网划分。
网络地址=192.168.10.0
子网掩码=255.255.255.128
回答五大问题：

- 多少个子网?
  在128(10000000)中，取值为1的位数为1，借用了一位主机位，因此答案为2^1=2。
- 每个子网多少台主机?
  有7个主机位取值为o(10000000)，还剩下7位主机位，因此答案是2^7-2= 126台主机。
- 有哪些合法的子网?
  256 -128 = 128。也就是子网的增量是128.因此子网为0和128
- 每个子网的广播地址是什么?
  在下一个子网之前的数字中，所有主机位的取值都为1，是当前子网的广播地址。对于子网0，下一个子网为128，因此其广播地址为127
  每个子网包含哪些合法的主机地址?
  合法的主机地址为子网地址和广播地址之间的数字。要确定主机地址，最简单的方法是写出子网地址和广播地址，这样合法的主机地址就显而易见了。



Ping

使用ICMP协议，向目的主机发送ICMP Echo请求报文，目的主机收到后发送Echo回答报文，PING根据时间和成功响应次数估算数据包的往返时间和丢包率

4. 链路层（CSMA/CD、PPP协议）

接入ISP需要使用PPP协议

5. 物理层

- 单工通信：单向传输
- 半双工通信：双向交替传输
- 全双工通信：双向同时传输

6. HTTP

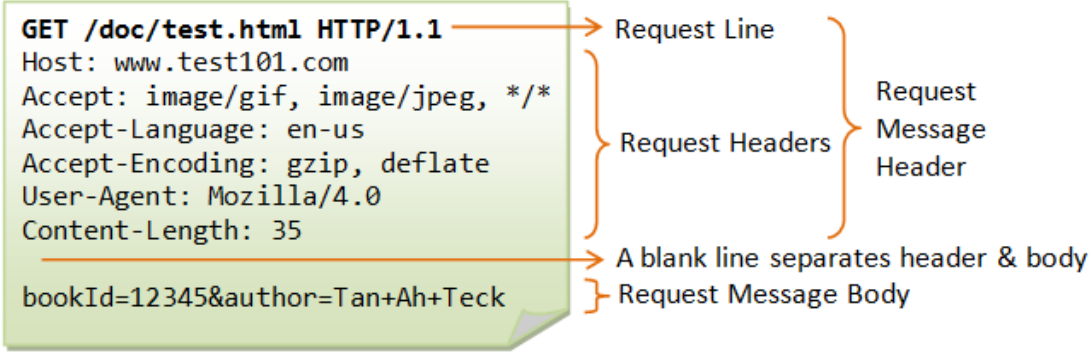
URI和URL

- URI是统一资源名称，相当于身份证号码
- URL是统一资源定位符，相当于某地的某人，是一种特殊的URI

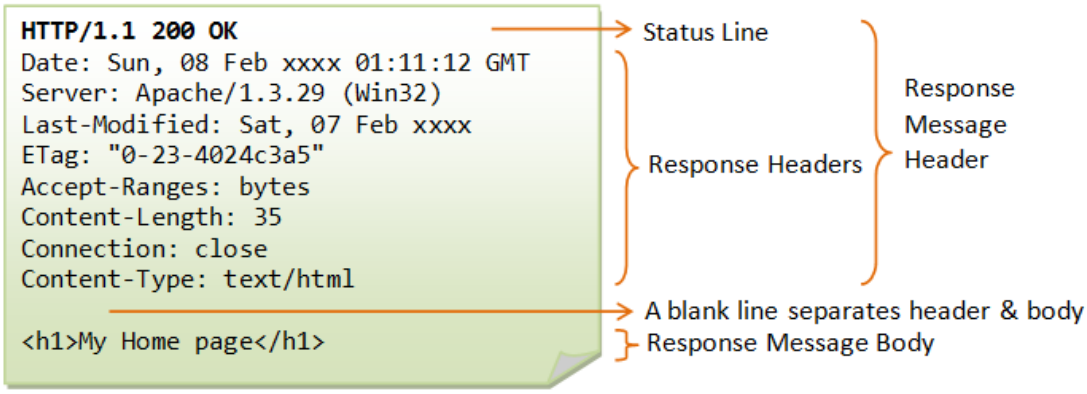
请求和响应报文

报文首部、空行、报文主体

- 请求报文



- 响应报文



HTTP首部：

- 通用首部字段
- 请求首部字段
- 响应首部字段
- 实体首部字段

HTTP方法（请求行中）

- GET：获取资源
- POST：传输实体主体
- HEAD：获取报文首部
- PUT：上传文件
- PATCH：对资源进行部分修改
- DELETE：删除文件
- OPTIONS：查询支持的方法
- CONNECT：要求在与代理服务器通信时建立隧道
- TRACE：追踪路径

HTTP状态码

1. 1xx信息状态码(请求正在处理)
2. 2xx成功状态码(请求成功处理完毕)
3. 3xx重定向吗(需要附加操作完成请求)
4. 4xx客户端错误码（服务器无法处理请求）
5. 5xx服务器错误状态码（服务器内部出错）

常用状态码：

- 100 ： 客户端必须继续发出请求
- 200：OK
- 301：永久重定向，隐式重定向
- 302：临时重定向，显式重定向
- 304：原有缓存有效，可以直接读取缓存内容
- 403：禁止访问
- 404：页面找不到
- 500：内部服务器错误
- 504：网关超时

转发和重定向的区别

- 转发是服务器行为，地址栏URL不变，可以共享request中的数据，效率高，用于账号登录
- 重定向是客户端行为，URL改变，不能共享数据，效率低，用于账号注销

cookie和session的区别

http是无状态的协议，所以需要用cookie或session跟踪浏览器用户身份

- cookie储存在客户端，cookie一般用来保存用户信息，比如用户登录，安全性低，通过在报文中添加cookie(请求报文）和set-cookie（响应报文）字段
- session存储在服务器，session一般用来保存用户状态信息，比如购物车状态，安全性高，通过在cookie中添加session ID实现

浏览器禁用cookie怎么办

URL重写技术，通过在URL后 添加sessionID

HTTP和HTTPS

- http端口80，https使用端口443
- 传统http传输使用明文，可能会被窃听，不验证通信方身份，可能伪装，不验证报文完整性，可能遭到篡改
- https是http和SSL/TLS组合使用
- SSL使用对称加密（只使用一把秘钥）和非对称加密（公钥和私钥）结合的方式
- 使用CA证书验证身份
- 使用报文摘要验证完整性

HTTP1.1新特性

- 使用长连接
- 支持分块传输
- 支持流水线
- 可以同时打开多个TCP连接
- 新增状态码
- 支持虚拟主机
- 新增缓存处理指令

HTTP2.0新特性

- 首部压缩
- 1.x解析基于文本，2.0采用二进制
- 多路复用，连接共享
- 服务端推送

数据库

1. InnoDB和MyISAM区别

- InnoDB支持事务及崩溃后的安全恢复， MyISAM不支持
- InnoDB支持行级锁，MyISAM只支持表级锁
- InnoDB支持外键（保持数据的一致性）， MyISAM不支持

- InnoDB支持MVCC，MyISAM不支持
- InnoDB适用于写密集， MyISAM适用于读密集

2. 事务

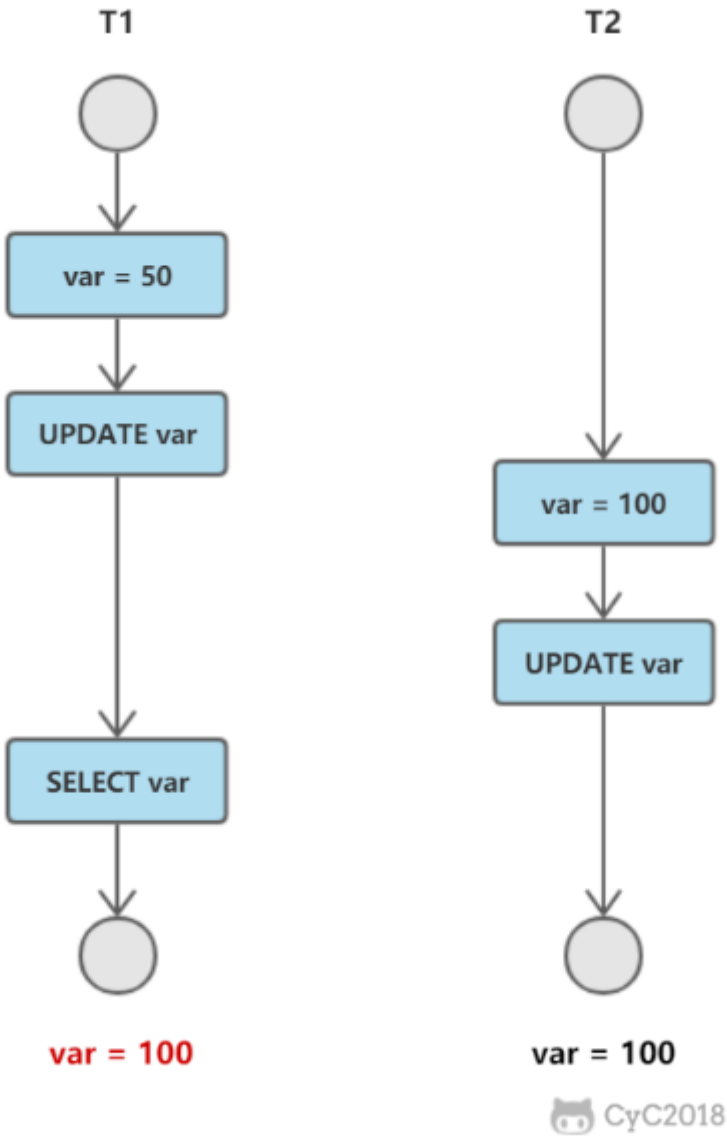
四大特性（ACID）

- 原子性：一个事务中要么全部成功，要么全部失败
- 一致性：事务提交前后，数据保持一致
- 隔离性：事务提交前对其他事务不可见
- 持久性：事务提交后，所做修改永远保存到数据库中，即使宕机，数据也不会丢失

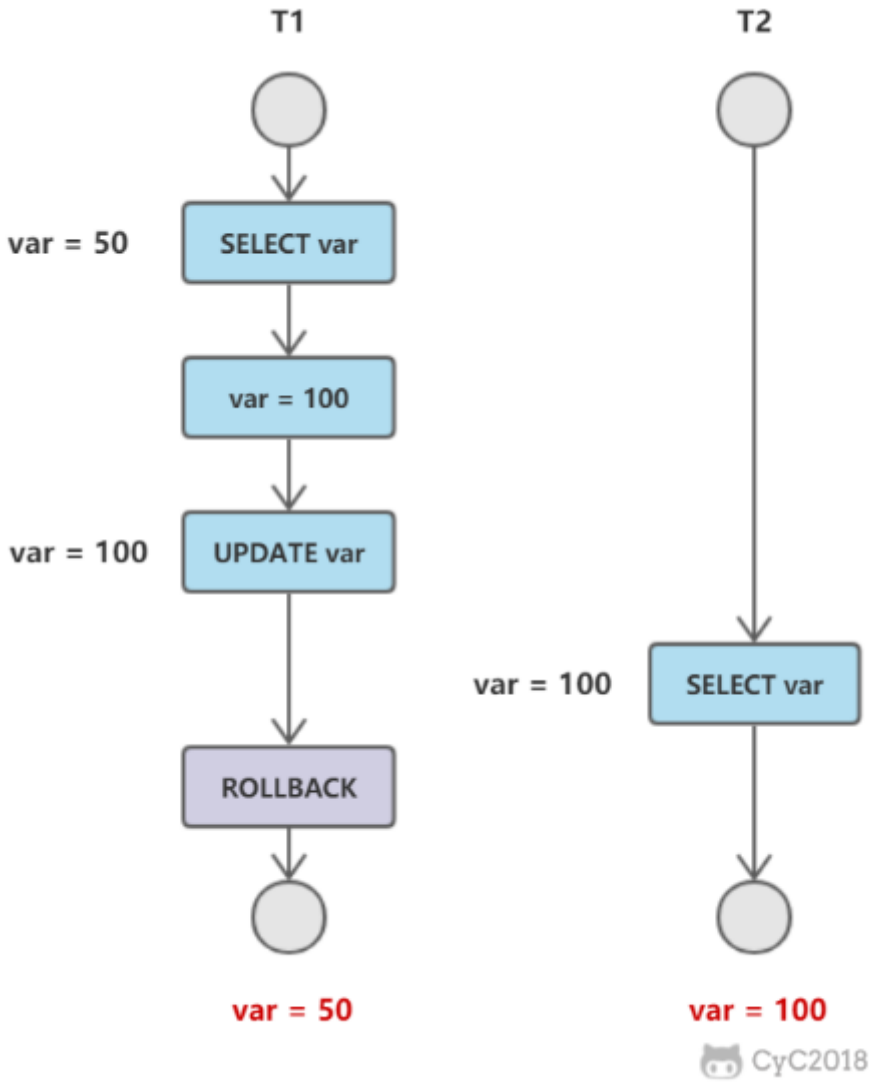
实现：通过redo日志（先写入redo日志再写入磁盘），和undo段（回滚）

3. 并发一致性问题

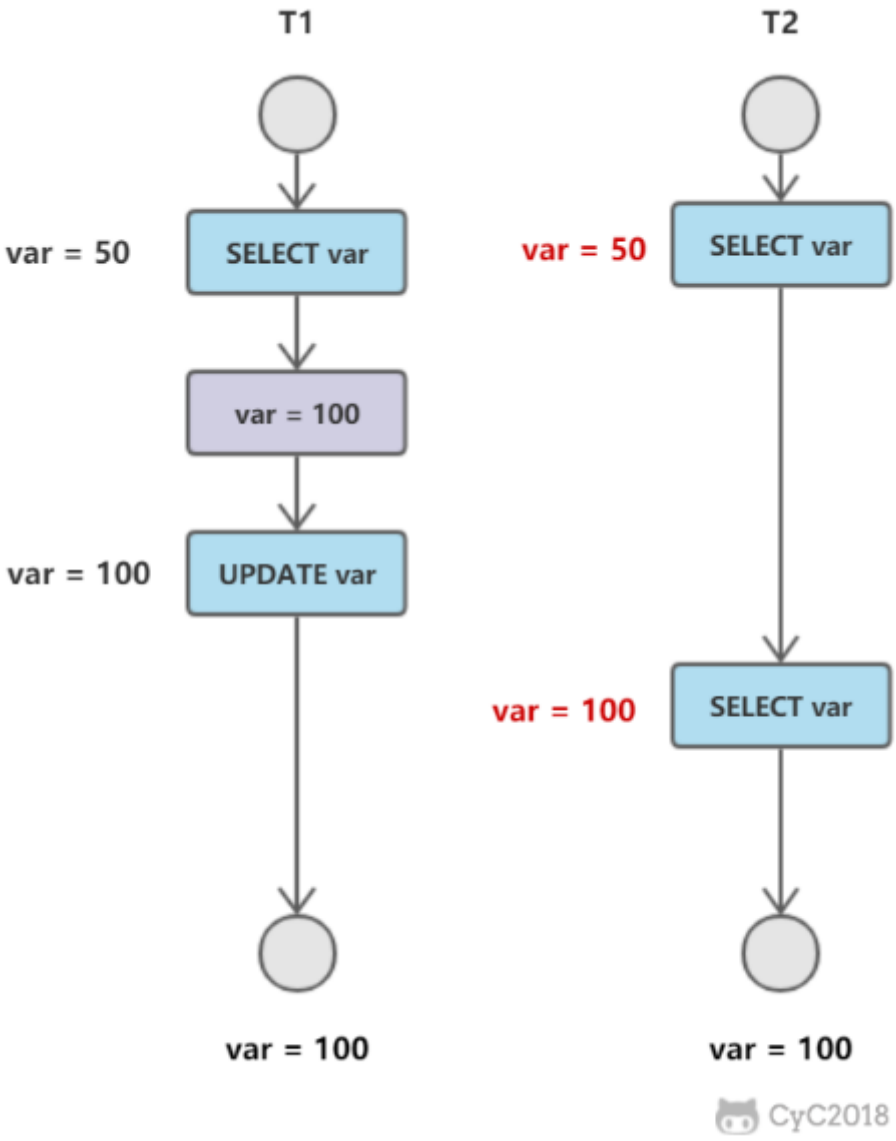
- 丢失修改



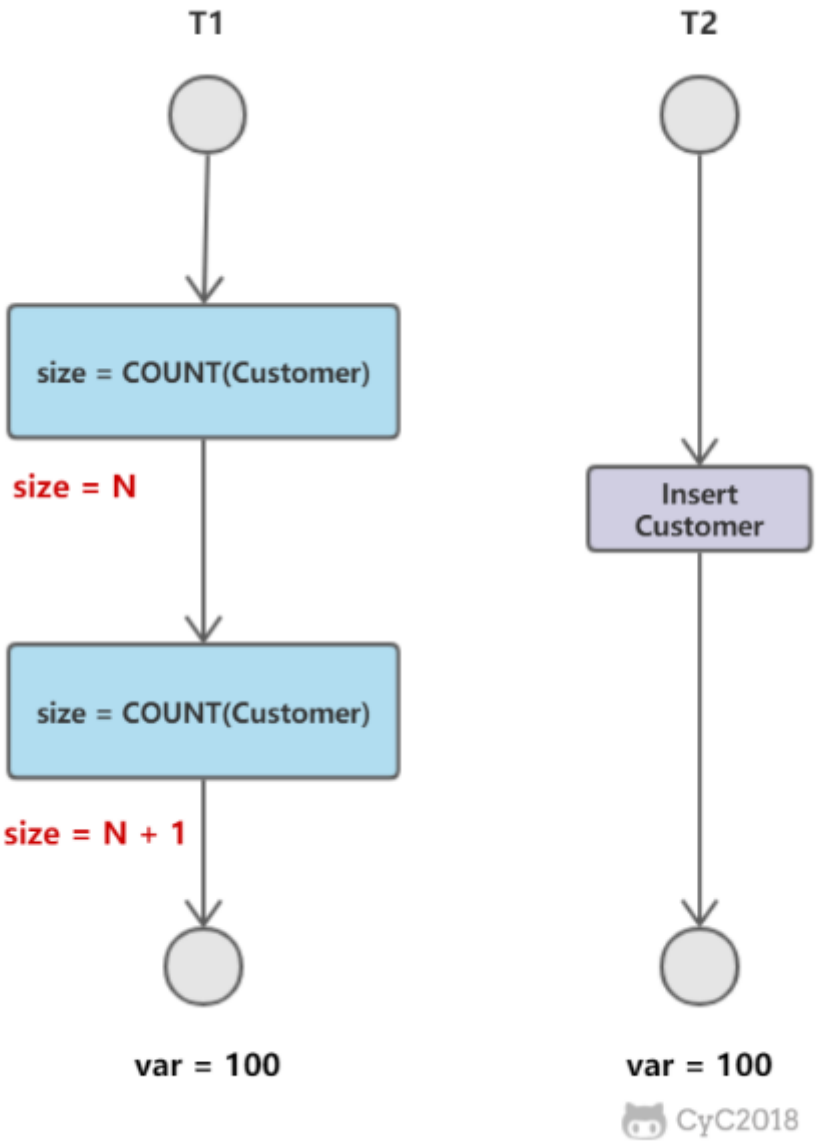
- 脏读


T1
T2
var = 50
SELECT var
var = 100
var = 100
UPDATE var
var = 100
SELECT var
ROLLBACK
var = 50
var = 100
CyC2018

- 不可重复读


T1
T2
var = 50
SELECT var
var = 50
SELECT var
var = 100
var = 100
UPDATE var
var = 100
SELECT var
var = 100
var = 100
CyC2018

- 幻读



4. 隔离级

- 未提交读：可以读未提交的数据，导致脏读、不可重复读、幻读
- 提交读：允许读已经提交的数据，导致不可重复读，幻读
- 可重复读：同一事务中多次读取数据一致，导致幻读
- 串行化

MySQL默认支持可重复读

5.MVCC(多版本并发控制)

非阻塞，通过保存某个时间的快照实现，只有在提交读和可重复读下工作。增加两个隐藏列，一个保存了行的创建时间，另一个保存了行的删除时间

- 非一致性读总是读取被锁定行的最新一份快照
- 可重复读总是读取事务开始时的行数据版本

MVCC快照是储存在undo段中，通过回滚指针把所有快照连接起来

不能解决幻读问题

6. Next-Key Lock

SELECT … FOR UPDATE 对读取的行加X锁

SELECT … LOCK IN SHARE MODE 对读取的行加S锁

锁定记录加范围

select * from t where a < 6 lock in share mode; 锁定（-∞，6）

select * from t where a = 7 lock in share mode; 只锁定7这个数据行

解决幻读问题

7. 连接池

启动时建立足够的数据库连接，组成连接池，尽可能多地重用资源，大大节省了内存，提高了服务效率，避免了频繁建立和关闭连接的步骤

常用的连接池

- C3P0
- DBCP
- Druid
- HikariCP

8. 索引

B+ Tree索引

数据结构

平衡树，是基于B Tree和叶子节点顺序访问指针进行实现的，相比B树，中间节点不保存数据，可以容纳更多节点元素，更加矮胖，减少查找次数，查找更加稳定，范围查找，只需遍历叶子节点链表

为什么不用红黑树

- 高度小于红何处，更少的查找次数（查找次数和树高有关）
- 利用磁盘预读特性

聚簇索引和辅助索引

- 聚簇索引中表中数据按主键存放，叶节点存放整张表的行记录数据，一张表只能有一个聚簇索引，当访问数据量适中时，适合聚簇索引
- 辅助索引叶节点存放聚簇索引键，所以需要再到聚簇索引中查找行记录，可以有多个辅助索引

哈希索引

能够以O（1）的时间查找，但失去了有序性，无法用于排序和分组，只支持精确查找，无法用于部分查找和范围查找。
在InnoDB中存在自适应哈希索引，当某个索引值使用非常频繁时，会在B+Tree索引上再创建一个哈希索引

全文索引

MyISAM支持，用于查找文本中的关键词，查找条件使用MATCH AGAINST

空间数据索引

用于地理数据存储

索引优化

- 独立的列：索引列不能是表达式的一部分
- 多列索引：多个列作为索引比单个列要好
- 索引列的顺序：把选择性（不重复的索引值和记录总数的比值，越高，区分度越高）最强的列放在最前面
- 前缀索引：对于BOLB、TEXT、VARCHAR类型的列，必须使用前缀索引，只索引开始的部分字符
- 覆盖索引：索引中包含需要查询的所有字段，（索引通常远小于数据行大小，大大减少访问，不需要再次访问聚簇索引，MyISAM中无需系统调用）

索引使用条件

- 数据量很小时，全表扫描
- 数据量中到大型，索引非常有效
- 特大型，分区技术

查询性能优化

- 使用Explain进行分析
- 优化数据访问（减少请求的数据量、减少扫描的行数使用覆盖索引）
- 切分大查询
- 分解大连接查询（对每个表进行一次单表查询）

大表优化

- 限定数据的范围
- 读/写分离
- 垂直分区
- 水平分区

一条SQL语句执行很慢的原因

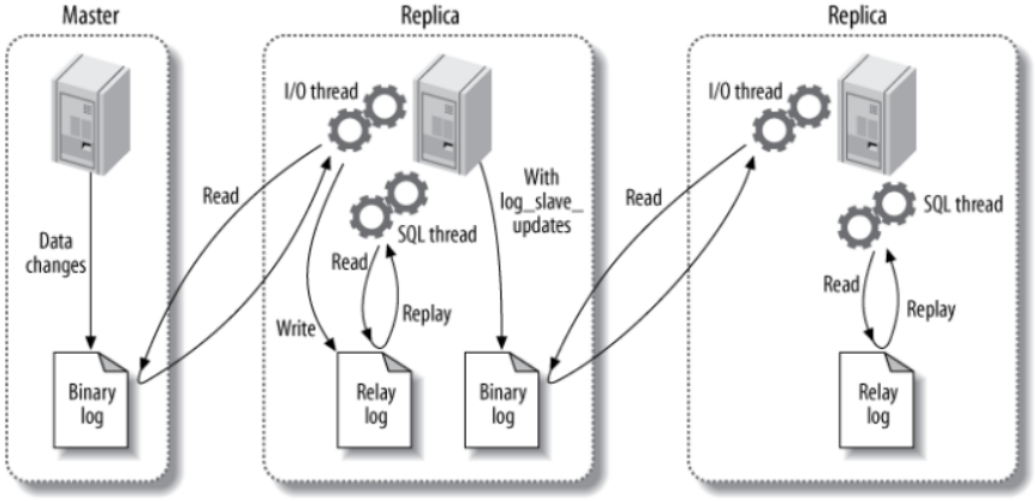
- 偶尔慢，可能是数据库在刷新脏页，拿不到锁
- 一直这么慢，字段没有索引或没有用到索引，数据库选错索引

9. 复制

主从复制

涉及三个线程：binlog线程、I/O线程、SQL线程

- binlog线程：将主服务器中的数据写入二进制日志binary log中
- I/O线程：从主服务器中读取二进制日志binary log，并写入从服务器的中继日志relay log
- SQL线程：负责读取中继日志，解析出主服务器已经执行的数据更改并在从服务器中重放



解决的问题

- 数据分布
- 负载均衡（读操作分布到多个服务器上）
- 备份
- 高可用性和故障切换（避免单点失败，缩短宕机时间）
- MySQL升级测试

可能存在的问题

- 主机宕机后，数据修改可能丢失
- 从库只有一个SQL线程，主库写压力大，复制可能延时

  解决：半同步复制、并行复制

读写分离（代理方式执行）

主服务器处理写操作以及实时性要求高的读操作，从服务器负责读操作

- 主从服务器负责各自的读和写，极大程度缓解了锁的争用
- 从服务器可使用MyISAM，提升查询性能
- 增加冗余，提高可用性

Spring

1. IOC

控制反转，通过DI（依赖注入），通过IOC容器（Spring框架）来管理创建对象的控制权。原来创建对象的主动权和时机由自己把握，现在这种权利转移到spring容器中。IOC容器实质上是一个Map，可以把应用从复杂的依赖关系中解放，实现解耦。

三种注入方式：

- 构造器注入
- setter方法注入
- 接口注入

2. AOP

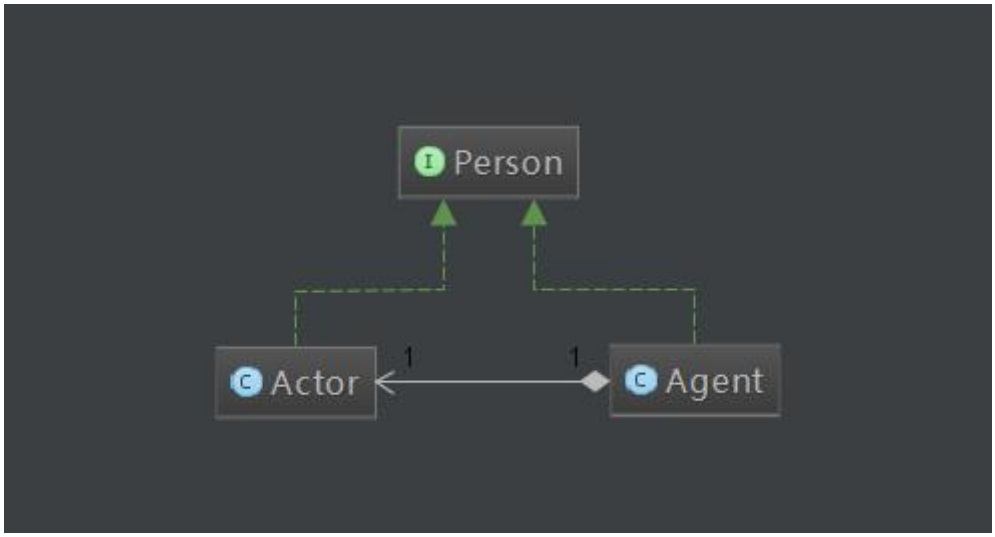
面向切面编程，能够将那些与业务无关，却为业务模块所共同调用的逻辑（例如事务处理、日志管理等）封装起来，便于减少系统的重复代码，降低模块间的解耦，有利于未来的可扩展性和可维护性。

基于动态代理实现：

- jdk自带的动态代理(实现接口的对象)
- 未实现接口则使用cglib

静态代理和动态代理

静态代理（编译时创建）



接口类：

```
interface Person {
    void speak();
}
```

真实实体类：

```
class Actor implements Person {
    private String content;
    public Actor(String content) {
        this.content = content;
    }

    @Override
    public void speak() {
        System.out.println(this.content);
    }
}
```

代理类：

```
class Agent implements Person {
    private Actor actor;
    private String before;
    private String after;
    public Agent(Actor actor, String before, String after) {
        this.actor = actor;
        this.before = before;
        this.after = after;
    }
    @Override
    public void speak() {
        //before speak
        System.out.println("Before actor speak, Agent say: " + before);
        //real speak
        this.actor.speak();
        //after speak
        System.out.println("After actor speak, Agent say: " + after);
    }
}
```

测试类：

```
public class StaticProxy {
    public static void main(String[] args) {
        Actor actor = new Actor("I am a famous actor!");
        Agent agent = new Agent(actor, "Hello I am an agent.", "That's all!");
        agent.speak();
    }
}
```

动态代理（运行时利用反射机制创建）

接口类：

```
public interface Fruit {
    public void show();
}
```

真实实体类：

```
public class Apple implements Fruit{
    @Override
    public void show() {
        System.out.println("<<<<show method is invoked");
    }
}
```

代理类：

```
public class DynamicAgent {

    //实现InvocationHandler接口，并且可以初始化被代理类的对象
    static class MyHandler implements InvocationHandler {
        private Object proxy;
        public MyHandler(Object proxy) {
            this.proxy = proxy;
        }

        //自定义invoke方法
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
            System.out.println(">>>>before invoking");
            //真正调用方法的地方
            Object ret = method.invoke(this.proxy, args);
            System.out.println(">>>>after invoking");
            return ret;
        }
    }

    //返回一个被修改过的对象
    public static Object agent(Class interfaceClazz, Object proxy) {
        return Proxy.newProxyInstance(interfaceClazz.getClassLoader(), new
Class[]{interfaceClazz},
                new MyHandler(proxy));
    }
}
```

测试类：

```
public class ReflectTest {
    public static void main(String[] args) throws InvocationTargetException,
IllegalAccessException {
        //注意一定要返回接口，不能返回实现类否则会报错
        Fruit fruit = (Fruit) DynamicAgent.agent(Fruit.class, new Apple());
        fruit.show();
    }
}
```

3. Spring Bean

Bean的生命周期

- Bean容器找到配置文件中 Spring Bean 的定义。
- Bean容器利用Java Reflection API创建一个Bean的实例。
- 如果涉及到一些属性值 利用set方法设置一些属性值。
- 如果Bean实现了BeanNameAware接口，调用setBeanName()方法，传入Bean的名字。
- 如果Bean实现了BeanClassLoaderAware接口，调用setBeanClassLoader()方法，传入ClassLoader对象的实例。
- 如果Bean实现了BeanFactoryAware接口，调用setBeanClassLoader()方法，传入ClassLoader对象的实例。
- 与上面的类似，如果实现了其他*Aware接口，就调用相应的方法。
- 如果有和加载这个Bean的Spring容器相关的BeanPostProcessor对象，执行postProcessBeforeInitialization()方法
- 如果Bean实现了InitializingBean接口，执行afterPropertiesSet()方法。
- 如果Bean在配置文件中的定义包含init-method属性，执行指定的方法。
- 如果有和加载这个Bean的Spring容器相关的BeanPostProcessor对象，执行postProcessAfterInitialization()方法
- 当要销毁Bean的时候，如果Bean实现了DisposableBean接口，执行destroy()方法。
- 当要销毁Bean的时候，如果Bean在配置文件中的定义包含destroy-method属性，执行指定的方法。

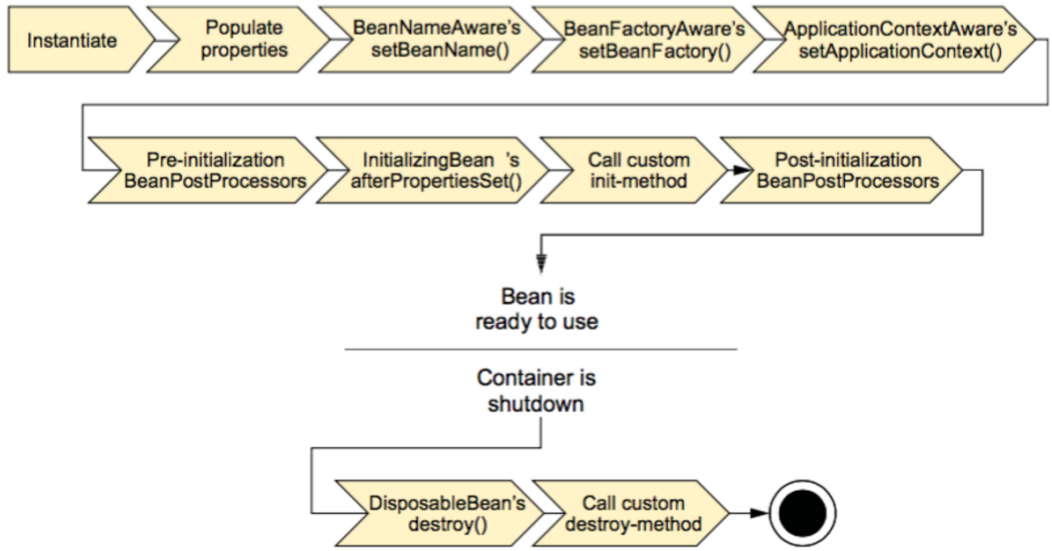


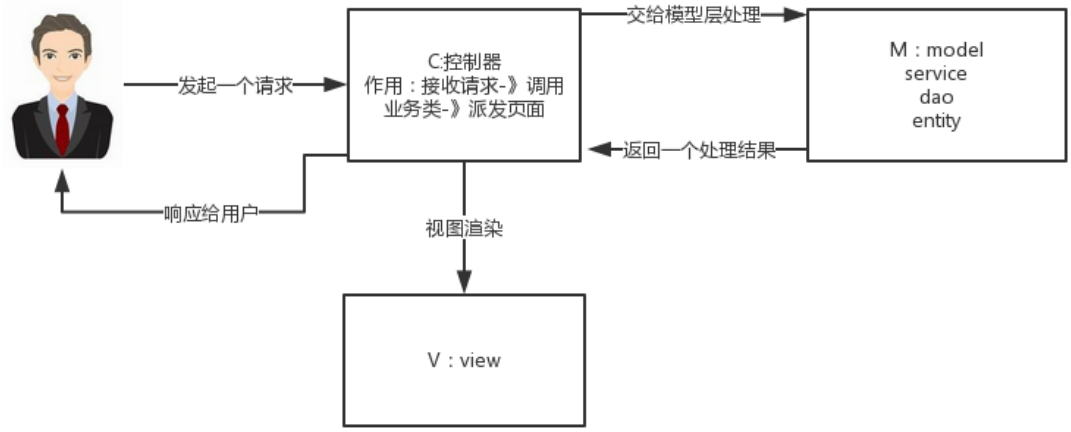
Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

Bean的作用域

- singleton：整个IOC容器只有一个bean对象
- prototype：每次请求bean都会new一个实例
- request：每次HTTP请求，都会创建一个新的bean
- session：同一个session共享一个bean，不同session使用不同的bean
- globalsession：同一个全局session共享一个bean

4. Spring MVC

M(model)V(view)C(control)



以请求为驱动，将请求发送给控制器，然后通过模型对象，分派器来展示请求结果视图，核心类为DispatcherServlet，它是一个servlet，顶层是实现的Servlet接口。

工作原理

（1）客户端（浏览器）发送请求，直接请求到 DispatcherServlet。

（2）DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。

（3）解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。

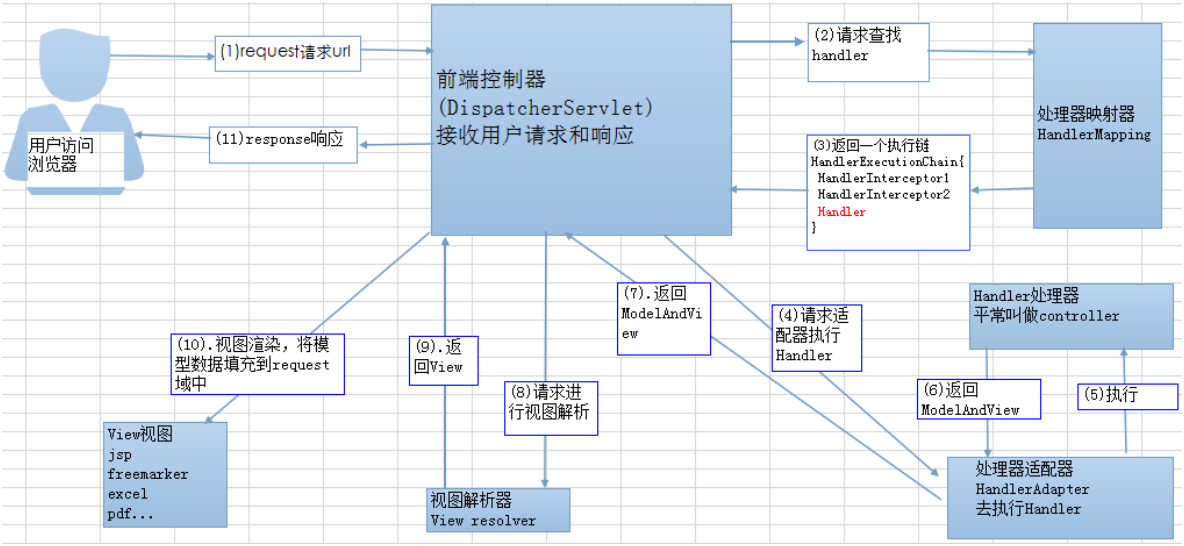
（4）HandlerAdapter 会根据 Handler 来调用真正的处理器开处理请求，并处理相应的业务逻辑。

（5）处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 View。

（6）ViewResolver 会根据逻辑 View 查找实际的 View。

（7）DispaterServlet 把返回的 Model 传给 View（视图渲染）。

（8）把 View 返回给请求者（浏览器）



5. Spring常用注解

- 声明为Bean：@Compoennt, @Controller, @Service, @Repository, @Bean
- 自动装配：@Autowired，@Resource
- MVC：@RequestMapping，@ResponseBody，@RequestHeader，@RequestParam

@Component和@Bean的区别

目的一样，注册Bean到Spring容器

- 作用对象不同，前者为类，后者为方法
- 前者表示组件类告诉Spring要创建一个该类的实例，后者表示会返回一个对象，该对象注册为上下文中的bean
- @Bean自定义性更强

@Autowired和@Resource区别

- 前者byType自动注入，后者默认按byName自动注入

6.MyBatis中$和#的区别

- #{}是编译好SQL语句后再取值，有一个占位符，再将参数值填充到SQL，可以防止SQL注入
- ${}会直接进行字符串替换

7. @SpringBootApplication 注解分析

可以看出大概可以把 @SpringBootApplication看作是 @Configuration、@EnableAutoConfiguration、@ComponentScan注解的集合。这三个注解的作用分别是：

- @EnableAutoConfiguration：启用 SpringBoot 的自动配置机制
- @ComponentScan： 扫描被@Component (@Service,@Controller)注解的bean，注解默认会扫描该类所在的包下所有的类。
- @Configuration：允许在上下文中注册额外的bean或导入其他配置类。

设计模式

面向对象

- 封装：利用抽象数据类型将数据和基于数据的操作封装在一起，减少耦合，提高重用性
- 继承：实现IS-A关系，
- 多态：编译时类型和运行时类型不同，使得相同编译类型在运行时表现出不同的行为，三个条件（继承、覆盖、向上转型）

设计原则

- S单一责任

- O开放封闭
- L里式替换
- I接口分离
- D依赖倒置

1. 单例模式

懒汉模式-线程不安全（安全加synchronized）

```
public class Singleton{
    private static Singleton instance;

    public static Singleton getInstance(){
        if (instance == null){
            instance =  new Singleton();
        }
        return instance;
    }
}
```

延迟初始化

饿汉模式

```
public class Singleton{
    private static Singleton instance = new Singleton();

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

双重检验锁定

```
public class Singleton{
    private static volatile Singleton instance;

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

使用volatile的原因：

new分为三个过程：

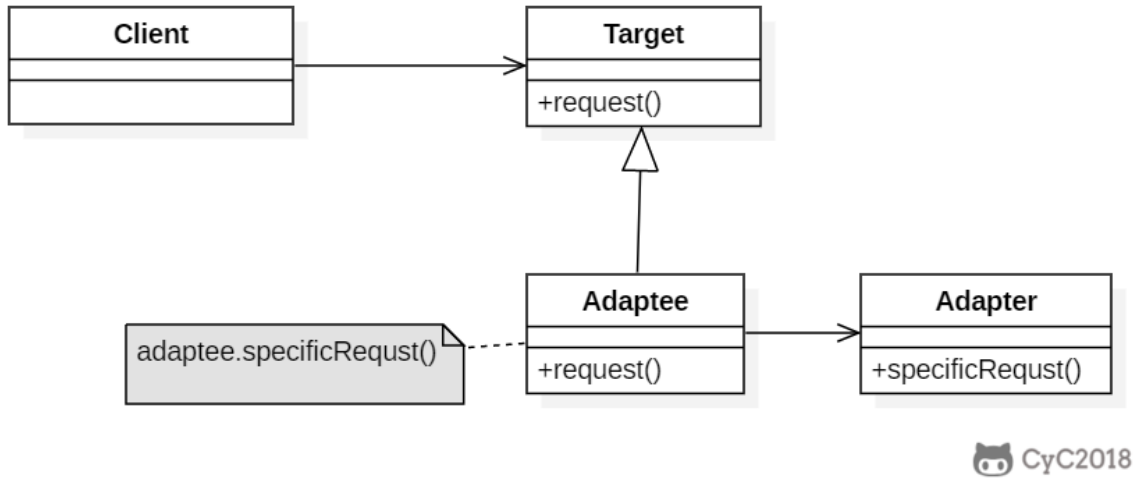
- 为对象分配内存地址
- 初始化

- 将引用指向该内存

volatile防止指令重排序

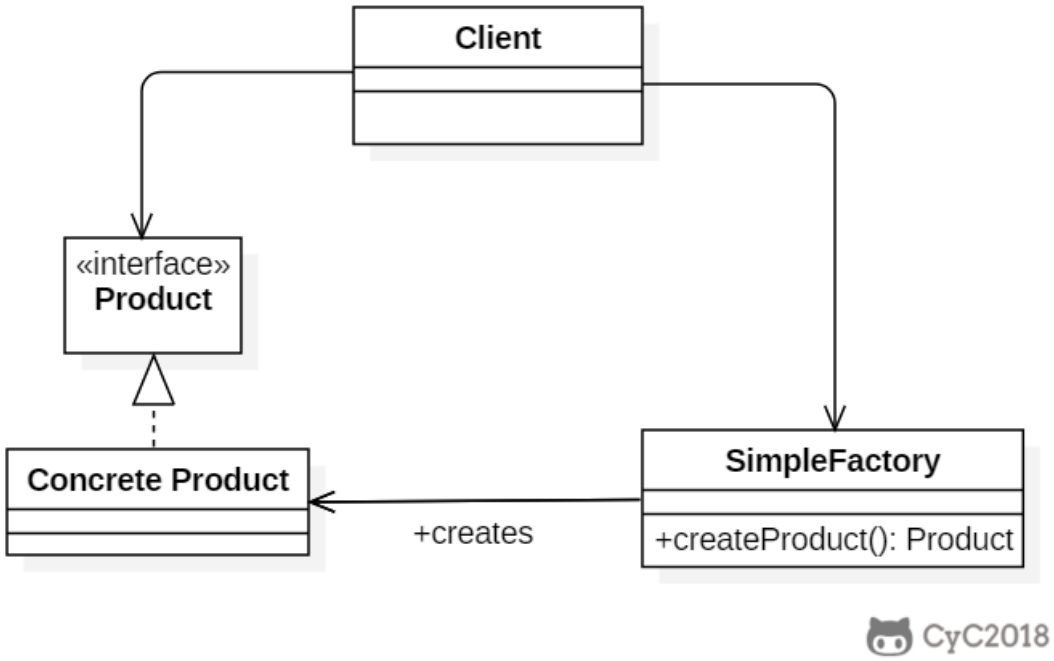
2. 适配器模式

把一个类接口转换为另一个用户需要的接口



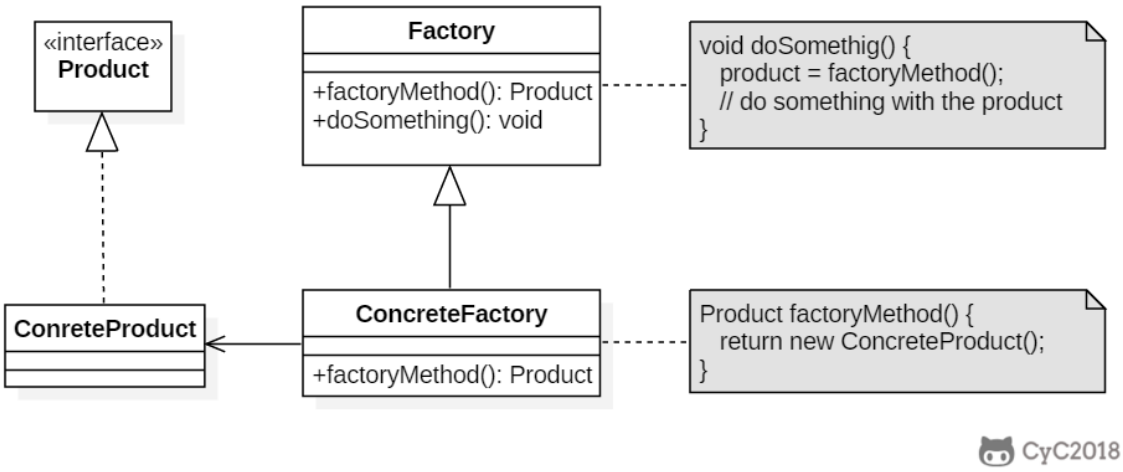
3. 工厂模式

简单工厂



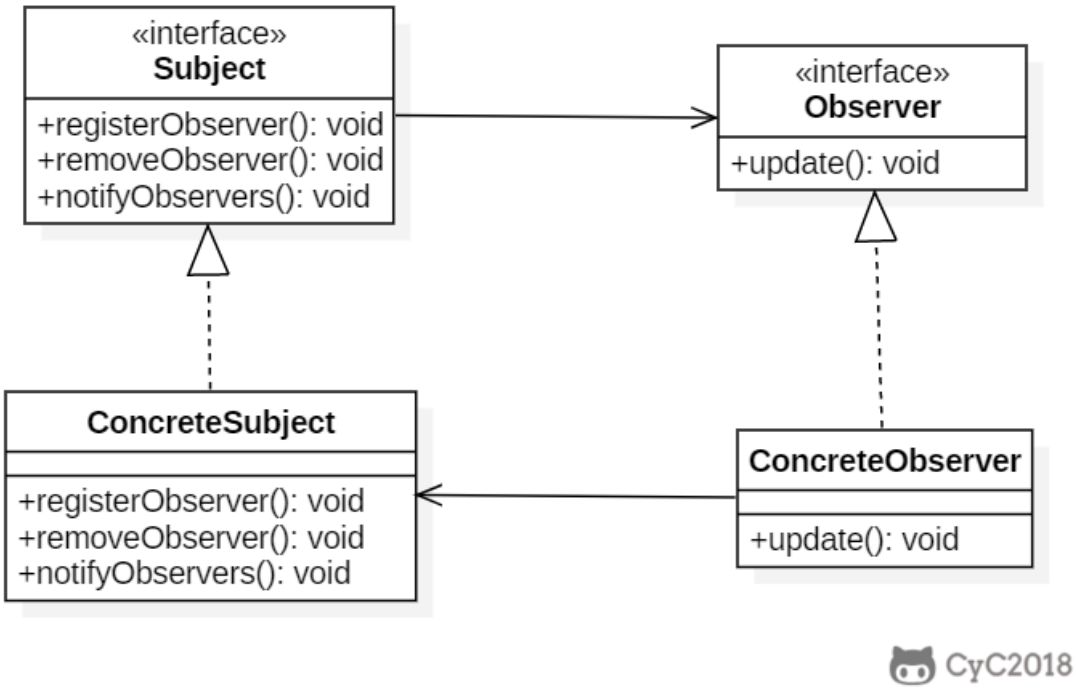
工厂方法

把实例化操作推迟到子类



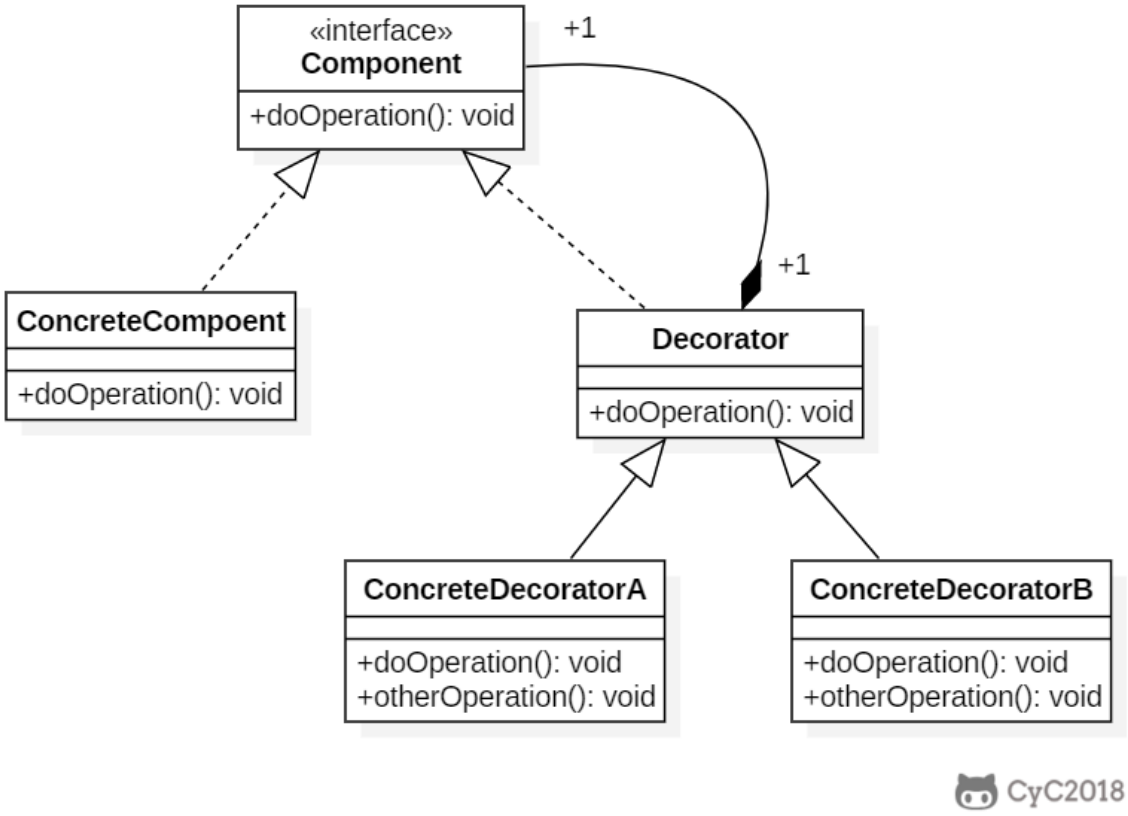
4. 观察者模式

主题具有注册和移除观察者以及通知所有观察者的功能，通过列表来维护观察者



5.装饰器模式

装饰者和具体组件都继承自组件，动态扩展装饰者功能


«interface»
Component
+doOperation(): void
+1
+1
ConcreteCompoent
+doOperation(): void
Decorator
+doOperation(): void
ConcreteDecoratorA
+doOperation(): void
+otherOperation(): void
ConcreteDecoratorB
+doOperation(): void
+otherOperation(): void
CyC2018