

学习笔记

Java



目录

I Java 基础知识	1
1 基本概念	2
1.1 基础知识	2
1.1.1 常用术语	2
1.1.2 Java 版本	3
1.1.3 JDK 安装	3
1.2 Java Tools	3
2 基本语法	7
2.1 基本规范	7
2.2 数据类型	7
2.2.1 整型	7
2.2.2 浮点数	8
2.3 运算符	9
2.3.1 位运算符	9
3 控制流程	12
3.1 switch	12
4 字符串	14
4.1 字符串对象	14
4.2 字符串连接操作符 +	15
4.3 字符串常量池	16
4.4 字符串拼接函数	16
4.5 特殊注意事项	16
4.5.1 String.format 和 MessageFormat.format 区别	16
4.5.2 MessageFormat.format 中传入 long 类型时问题	16

4.5.3 String.split() 使用单字符传入特殊符号时问题	16
5 对象	17
5.1 初始化	17
5.1.1 对象拷贝	17
6 内部类	19
6.1 .this 和 new	19
6.1.1 嵌套类	20
7 范型	21
7.1 基本概念	21
7.2 范型类	21
7.3 范型方法	21
7.4 通配符类型	21
7.5 范型代码和虚拟机	21
7.5.1 类型擦除	21
7.5.2 翻译泛型表达式	22
7.5.3 约束和局限性	22
8 集合	24
9 函数式接口	25
9.1 函数式接口	25
10 线程	26
10.1 线程基础知识	26
10.1.1 线程状态	26
10.1.2 创建线程的方式	28
10.1.3 捕获异常	32
10.1.4 常用函数	32
10.1.5 终止	33
10.2 synchronized	34
10.2.1 synchronized 使用与实现	34

10.2.2 原子性	35
10.2.3 可见性	35
11 Java Tools	37
11.1 javac 命令	38
11.2 javap 命令	39
11.2.1 Java 字节码	39
11.3 java 命令	41
11.3.1 选项	41
11.4 jps 命令	42
11.4.1 选项	42
11.5 jcmd 命令	44
12 内存模型	46
12.1 Java 内存模型与线程	46
12.1.1 Gustafson's law 古斯塔夫森定律	50
12.1.2 JAVA 内存模型 JMM	50
13 Linux	51
13.1 uptime	54
13.2 top	55
13.2.1 交互式命令	55
13.2.2 OPTIONS	55
13.2.3 列字段含义	58
13.2.4 示例	61
13.3 free	62
13.4 meminfo	63
13.5 slabinfo	64
13.6 vmstat	65
14 IPv6	66
14.1 基础知识	66

14.1.1 IPv6 格式	66
14.1.2 问题	66
15 Http/2	67
15.1 HTTP 历史	67
15.1.1 HTTP/0.9	67
15.1.2 HTTP/1.0	68
15.2 HTTP/1.1	69
15.2.1 持久连接	69
15.2.2 管道化连接	70
15.2.3 web 优化	71
15.3 HTTP/2	71
15.3.1 Connection	72
15.3.2 Frames	76
15.3.3 Stream	80
15.4 HPACK	87
15.4.1 Deflate	87
15.4.2 Huffman Coding	87
15.4.3 索引表	87
15.4.4 实现	88
15.5 参考	88

Part I

Java 基础知识

第1章 基本概念

1.1 基础知识

官方文档

[Java Documentation](#)

1.1.1 常用术语

JDK: Java Development Kit

编写 `java` 程序的程序员使用的软件

JRE: Java Runtime Environment

运行 `java` 程序的用户使用的软件

Server JRE:

在服务器上运行 `java` 程序的软件

SE: Standard Edition

用于桌面或者简单服务器应用的 `java` 平台

EE: Enterprise Edition

用于复杂服务器应用的 `java` 平台

ME: Micro Edition

用于手机和其他小型设备的 `java` 平台

Java FX:

用于图形化用户界面的一个替代工具包在 Oracle 的 Java SE 发布版本中提供

OpenJDK:

Java SE 的一个免费开源实现，不包含浏览器集成或者 JavaFX

JCP: Java Community Process

Java 社区进程

JSR: Java Specification Requests

Java 规范请求由 JCP 成员向委员会提交的 Java 发展议案，经过一系列流程后，如果通过最终会体现在未来的 Java 中

TCK: Technology Compatibility Kit

技术兼容性测试

1.1.2 Java 版本

Java SE 是相对于 Java EE 和 Java ME，它是 Java 的标准版。

Java SE8 对应“内部”版本号是 1.8.0。

版本号命名规则示例：

Java SE 8u31

Java SE 8 的第 31 次更新，内部版本号为 1.8.0_31

1.1.3 JDK 安装

Mac 可以通过 dmg 文件包安装

安装后相关路径

```
/Library/Java/JavaVirtualMachines/jdk-*.jdk/Contents/Home/  
/usr/libexec/java_home  
  
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/  
  
# 执行目录  
/usr/bin/java  
  
# 库源文件为压缩文件src.zip  
/Library/Java/JavaVirtualMachines/jdk-*.jdk/Contents/Home/lib/src.zip
```

src.zip 文件包含了所有的公共类的源代码。

设置环境变量

设置 JAVA_HOME

```
export JAVA_HOME="$(/usr/libexec/java_home -v 1.8.0_201)"      # 以版本1.8.0_201为例  
export PATH=$JAVA_HOME/bin:$PATH
```

1.2 Java Tools

Create and Build Applications

* javac

You can use the javac tool and its options to read Java class and interface definitions and compile them into bytecode and class files.

您可以使用 `javac` 工具及其选项来读取 Java 类和接口定义，并将它们编译成字节码和类文件。

具体使用可参照 [11.1 javac 章节](#)

* **javap**

You use the `javap` command to disassemble one or more class files.

您可以使用 `javap` 命令来反汇编一个或多个类文件。

具体使用可参照 [11.2 javap 章节](#)

* **java**

You can use the `java` command to launch a Java application.

您可以使用 `java` 命令启动 Java 应用程序。

具体使用可参照 [11.3 java 章节](#)

* **jar**

You can use the `jar` command to create an archive for classes and resources, and to manipulate or restore individual classes or resources from an archive.

您可以使用 `jar` 命令为类和资源创建存档，并从存档中操作或还原单个类或资源。

* **jdeps**

You use the `jdeps` command to launch the Java class dependency analyzer.

使用 `jdeps` 命令启动 Java 类依赖关系分析器。

* **javadoc**

You use the `javadoc` tool and its options to generate HTML pages of API documentation from Java source files.

您可以使用 `javadoc` 工具及其选项从 Java 源文件生成 API 文档的 HTML 页面。

Monitor Java Applications

* **jconsole**

You use the `jconsole` command to start a graphical console to monitor and manage Java applications.

您可以使用 `jconsole` 命令启动图形化控制台来监视和管理 Java 应用程序。

* **jvisualvm**

Monitor the JVM

* **jps**

Experimental You use the `jps` command to list the instrumented JVMs on the

target system.

实验性。您可以使用 `jps` 命令列出目标系统上已检测的 jvm。

具体使用可参照 [11.4 jps 章节](#)

* **jstat**

Experimental You use the `jstat` command to monitor JVM statistics.

实验性。使用 `jstat` 命令监视 JVM 统计数据。此命令是实验性的，不受支持。

* **jstatd**

Experimental You use the `jstatd` command to monitor the creation and termination of instrumented Java HotSpot VMs.

实验性。您可以使用 `jstatd` 命令监视仪表化 Java HotSpot vm 的创建和终止。

* **jmc**

You use the `jmc` command and its options to launch Java Mission Control. Java Mission Control is a profiling, monitoring, and diagnostics tools suite.

使用 `jmc` 命令及其选项启动 Java 任务控制。Java 任务控制是一个分析、监视和诊断工具套件。

Troubleshooting Tools

* **jcmd**

You use the `jcmd` utility to send diagnostic command requests to a running Java Virtual Machine (JVM).

* **jdb**

You use the `jdb` command and its options to find and fix bugs in Java platform programs.

* **jinfo**

Experimental You use the `jinfo` command to generate Java configuration information for a specified Java process.

* **jmap**

Experimental You use the `jmap` command to print details of a specified process.

实验性。您可以使用 `jmap` 命令打印指定进程的详细信息。

* **jstack**

Experimental You use the `jstack` command to print Java stack traces of Java threads for a specified Java process.

实验性。您可以使用 `jstack` 命令为指定的 Java 进程打印 Java 线程的 Java 堆栈跟踪。

Others

* **jlink**

You can use the `jlink` tool to assemble and optimize a set of modules and their dependencies into a custom runtime image.

您可以使用 `jlink` 工具将一组模块及其依赖项组装并优化到自定义运行时映像中。

* **jmod**

You use the `jmod` tool to create JMOD files and list the content of existing JMOD files.

使用 `jmod` 工具创建 JMOD 文件并列出现有 JMOD 文件的内容。

* **jdeprscan**

You use the `jdeprscan` tool as a static analysis tool that scans a jar file (or some other aggregation of class files) for uses of deprecated API elements.

您可以使用 `jdeprscan` 工具作为静态分析工具，它扫描一个 `jar` 文件（或一些其他类文件的聚合），以便使用废弃的 API 元素。

* **jshell**

You use the `jshell` tool to interactively evaluate declarations, statements, and expressions of the Java programming language in a read-eval-print loop (REPL).

您可以使用 `jshell` 工具在 read-eval-print 循环（REPL）中交互式地计算 Java 编程语言的声明、语句和表达式。



更详细的内容请参考
[tools-and-command-reference](#)

第 2 章 基本语法

2.1 基本规范

Java 区分大小写

Java 采用骆驼命名法，CamelCase。

2.2 数据类型

Java 包含 8 种基本类型。

- 4 种整型
- 2 种浮点类型
- 1 种表示 Unicode 编码的字符单元的字符类型 char
- 1 种表示真值的 boolean 类型

2.2.1 整型

类型	存储需求	取值范围
byte	1 字节	-128 ~ 127
short	2 字节	-32768 ~ 32767
int	4 字节	-2147483648 ~ 2147483647
long	8 字节	-9223372036854775808 ~ 9223372036854775807



Java 没有任何无符号 (unsigned) 形式的整型。

长整型值有一个后缀 l 或者 L。

```
long number = 123L; // number值为: 123
```

- 二进制带前缀 0b 或者 0B （从 java 7 开始）
- 八进制带前缀 0
- 十六进制带前缀 0x 或者 0X

```

int binaryNumber      = 0B101;           // 二进制表示 5
int hexadecimalNumber = 0x123;           // 十六进制表示 291
int octonaryNumber   = 0123;            // 八进制表示 83

```

同样从 Java7 开始为数字加下划线，Java 编译器会去除这些下划线

```
System.out.println(0_1_0);           // 010表示八进制，输出8
```

整数类型操作

- `toBinaryString` 将整数转为二进制位字符串
- `bitCount` 计算整数 bit 位为 1 的个数
- ...

```

System.out.println(Integer.toBinaryString(666));
System.out.println(Integer.toBinaryString(-1));
System.out.println(Integer.bitCount(0B10_10011010));

/**
 * 1010011010
 * 111111111111111111111111111111111111111111111
 * 5
 */

```



注意 JS INT 类型的值范围：

- [Integers in JavaScript](#)
- [Safe Integers](#)

2.2.2 浮点数

类型	存储需求	取值范围
<code>float</code>	4 字节	有效位数为 6–7 位
<code>double</code>	8 字节	有效位数为 15 位

`float` 类型的值要有后缀 `f` 或者 `F`。没有后缀默认为 `double` 类型。

`double` 类型也可以添加 `d` 或者 `D` 后缀。

用于表示溢出或者出错情况的三个特殊浮点值：

- 正无穷大
- 负无穷大
- NaN (不是一个数字)

例如：一个正整数除以 0 的结果为正无穷大；计算 $0/0$ 或者负数的平方根结果为 NaN。

对应常量为：

```
// float
Float.POSITIVE_INFINITY;
Float.NEGATIVE_INFINITY;
Float.NaN;
// double
Double.POSITIVE_INFINITY;
Double.NEGATIVE_INFINITY;
Double.NaN;
```

所有“非数值”的值都认为是不相同的。

```
if(Double.isNaN(x)) // check whether x is "not a number"
```

char 类型

2.3 运算符

2.3.1 位运算符

操作符	名称	说明
~	Not (按位取反)	一元运算符 0 变为 1、将 1 变为 0
&	And (按位与)	二元运算符 两值全为 1 则为 1 例: 1&1=1
	Or (按位或)	二元运算符 两值有 1 则为 1 例: 0 1=1
^	Xor (按位异或)	二元运算符 两值不相同则异或结果为 1 例: 1^0=1
<<	左移位	二元运算符
>>	右移位	二元运算符 符号运算符会填充高位
>>>	无符号右移位	二元运算符 运算符会用 0 填充高位

移位

```
/**  
 * 下一个2的幂: 普通性能
```

```

/*
 * @param numElements
 * @return
 */
public static int calculateSize(int numElements) {
    int initialCapacity = 8;
    while (initialCapacity < numElements) initialCapacity <<= 1;
    return initialCapacity;
}

/*
 * ArrayDeque 扩容原理；下一个2的幂；比上面的函数性能更好
*/
private static int calculateSize(int numElements) {
    int initialCapacity = 8;
    // Find the best power of two to hold elements.
    // Tests "<=" because arrays aren't kept full.
    if (numElements >= initialCapacity) {
        initialCapacity = numElements;
        initialCapacity |= (initialCapacity >>> 1);
        initialCapacity |= (initialCapacity >>> 2);
        initialCapacity |= (initialCapacity >>> 4);
        initialCapacity |= (initialCapacity >>> 8);
        initialCapacity |= (initialCapacity >>> 16);
        initialCapacity++;
    }

    if (initialCapacity < 0) // Too many elements, must back off
        initialCapacity >>>= 1;// Good luck allocating  $2^{30}$  elements
}
return initialCapacity;
}

```

与操作符

判断某个正数字是否为 2 的幂次方

如果 N 与 $N - 1$ 相与则此数为 2 的幂次方

```

/**
 * 判断非零正整数是否为2的幂次方
 * @param number
 * @return
 */
public boolean isPowerOfTwo(int number) {
    if (number <= 0) {
        return false;
    }

    return (number & (number - 1)) == 0;
}

```

字节顺序

对于 w 位的操作数 x 用位表示为 $[x_{w-1}, x_{w-2}, \dots, x_0, x_0]$ 。其中 x_{w-1} 是最高有效位，而 x_0 是最低有效位。

小端法 (little endian): 选择在内存中按照从最低有效字节到最高有效字节的顺序存储对象

大端法 (big endian): 选择在内存中按照从最高有效字节到最低字节的顺序存储对象

假设变量 `x` 的类型为 `int`, 位于地址 `0x100` 处, 它的十六进制值为 `0x1234567`。地址范围 `0x100 ~ 0x103` 的字节顺序。

大端法

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

小端法

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

移位运算符的右操作数会先完成取模运算。`int` 类型模 32, `long` 类型模 64。

避免时序攻击的字符串比较

```
static boolean equals(byte[] s1, byte[] s2) {
    if (s1.length != s2.length) {
        return false;
    }
    char c = 0;
    for (int i = 0; i < s1.length; i++) {
        c |= (s1[i] ^ s2[i]);
    }
    return c == 0;
}
```

第3章 控制流程

3.1 switch

Java

只支持基本类型 (byte, short, char, int)、Enum 类型以及 String, Character, Byte, Short 和 Integer 类。

只支持常量表达式，case 会进行整数范围验证。

```
/*
 * 枚举类型
 */
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

/*
 * 星期判断
 * @param day
 */
public static void switchTestWithEnum(Day day) {
    switch (day) {
        case MONDAY:
            System.out.println("Today is Mondays.");
            break;
        default:
            System.out.println("Today is ...");
            break;
    }
}

public static void switchTestWithCharacter(Character character) {
    switch (character) {
        case 'a':
            System.out.println("character is a");
            break;
        case 'b':
            System.out.println("character is b");
            break;
        default:
            System.out.println("nothing...");
    }
}

//调用
switchTestWithEnum(Day.SUNDAY);
switchTestWithCharacter(new Character('a'));
```

code snippet 3.1: switch

PHP

`case` 表达式可以是任何求值为简单类型的表达式，即整型或浮点数以及字符串。不能用数组或对象，除非它们被解除引用成为简单类型。

允许使用分号代替 `case` 语句后的冒号

`case` 比较执行的是松散比较 `==`

```
$beer = 'caseC';
$a = 'caseA';

switch($beer)
{
    case $a;
        echo "this is caseA";
        break;
    case 'caseB';
        echo "this is caseB";
        break;
    case 'case'.'C':
        echo "this is caseC";
        break;
    default;
        echo 'Please make a new selection...';
        break;
}
```

code snippet 3.2: php-swtich

第4章 字符串

Strings are constant; their values cannot be changed after they are created.

4.1 字符串对象

String 对象不可改变，具有恒定性。

每个 String 对象都有常量值。

字符串字面常量是对 String 类对象的引用。

在 Java 中 String 对象可以认为是 char 数组的延伸和进一步的封装。



这里所说的 char 数组不是 C 语言意义上的字符型数组，而是大致类似于 C 语言中的 char* 指针。

```
String str = "Java学习笔记";
```

```
char* str = "Java学习笔记";
```

```
public final class String implements java.io.Serializable, Comparable<String>, CharSequence {

    /** The value is used for character storage. */
    private final char value[];

    /*
     * @param value          Array that is the source of characters
     * @param offset         The initial offset
     * @param count          The length
     *
     * @throws IndexOutOfBoundsException
     *         If the {@code offset} and {@code count} arguments index
     *         characters outside the bounds of the {@code value} array
     */
    public String(char value[], int offset, int count) {
        if (offset < 0) {
            throw new StringIndexOutOfBoundsException(offset);
        }
        if (count <= 0) {
            if (count < 0) {
                throw new StringIndexOutOfBoundsException(count);
            }
            if (offset <= value.length) {
                this.value = "".value;
            }
        }
    }
}
```

```

        return;
    }
}

// Note: offset or count might be near -1>>1.
if (offset > value.length - count) {
    throw new StringIndexOutOfBoundsException(offset + count);
}
this.value = Arrays.copyOfRange(value, offset, offset+count);
}

}

```

通过上面 `String` 类的实现代码可以发现 `String` 类和 `value` 数组都是 `final` 类型，这就保证了 `String` 对象的不变性。这种不变性可以带来极大的好处。

- 保证对 `String` 对象的任意操作都不会改变原字符串
- 意味着操作字符串不会出现线程同步问题
- 成就了字符串驻留以及共享（常量池）

4.2 字符串连接操作符 +

如果字符串连接操作符运算的结果不是编译时的常量表达式，那么该操作符会隐式地创建新的 `String` 对象。

如果只有一个操作数表达式是 `String` 类型，那么就会在另一个操作数上执行字符串转换以在运行时产生字符串。对于简单类型 `Java` 还可以通过直接将简单类型转换为字符串而优化掉包装器对象的创建。

+操作字符在语法上是左结合。

例如：

```

String first = 1 + 2 + "Java";
String second = "Java" + 1 + 2;
String three = "Java" + null;
System.out.println(first);
System.out.println(second);
System.out.println(three);

/**
 * Output:
 *
 * 3Java
 * Java12
 * Javanull
 *
 */

```

在字符串转换方面要特别注意引用类型转行。

如果该引用是 `null`，那么它转换为字符串"null"。

如果其他引用类型，则调用其对象上的 `toString()` 方法；如果调用 `toString()` 方法的结果是 `null`，那么就用字符串"null" 代替。

4.3 字符串常量池

深入解析 [String#intern](#)

4.4 字符串拼接函数

4.5 特殊注意事项

4.5.1 String.format 和 MessageFormat.format 区别

4.5.2 MessageFormat.format 中传入 long 类型时间问题

4.5.3 String.split() 使用单字符传入特殊符号时问题

```
/**  
 * Splits this string around matches of the given regular expression.  
 */  
public String[] split(String regex);
```

regex 参数是正则表达式。如果使用单字符时并出现如下字符时一定要做转义，使用 \\ 转义。

```
. $ | ( ) [ { ^ ? * + \\
```

```
String origin = "a.b.c.d";  
//错误使用  
String[] errorArray = origin.split(".");  
for (int i = 0; i < errorArray.length; i++) {  
    System.out.println(i);  
}  
//正确使用；使用\\转义  
String[] rightArray = origin.split("\\.");  
for (String s : rightArray) {  
    System.out.println(s);  
}
```

第 5 章 对象

5.1 初始化

静态初始化只有在对象被创建或者第一次访问静态数据时才会被初始化。

初始化的顺序时先静态对象（如果他们尚未因前面的对象创建过程而被初始化），而后才是“非静态”对象。

以 Dog 类为示例总结一下对象的创建过程：

1. 当首次创建类型为 Dog 的对象或者 Dog 类的静态方法/静态域首次被访问时，java 解释器必须查找类路径，以定位 Dog.class 文件
2. 然后载入 Dog.class，有关静态初始化的所有动作都会执行。因此，静态初始化只在 Class 对象首次加载的时候进行一次。按照顺序。
3. 当用 new Dog() 创建对象的时候，首先将在堆上为 Dog 对象分配足够的存储空间。
4. 这块存储空间会被清零，这就自动将 Dog 对象中的所有基本类型数据都设置成了默认值（数字为 0，boolean 为 false），而引用则被设置成 null（例如 String）
5. 执行所有出现于字段定义处的初始化动作
6. 执行构造器（会涉及到继承的问题）

总结

基类静态代码块、基类静态成员字段并列优先级，按照代码中出现先后顺序执行（只有第一次加载类时执行）

派生类静态代码块、派生类静态成员字段并列优先级，按照代码中出现先后顺序执行（只有第一次加载类时执行）

基类普通代码块、基类普通成员字段并列优先级，按照代码中出现先后顺序执行

基类构造函数

派生类普通代码块、派生类普通成员字段并列优先级，按照代码中出现顺序执行

派生类构造函数

5.1.1 对象拷贝

对象的拷贝分为 shallow copy 和 deep copy 。

shallow copy 既浅拷贝。

<https://zhuanlan.zhihu.com/p/26964202>

clone

clone

PHP 与 Java 一样，对象的对象属性都只是引用拷贝。

php 的拷贝是通过 `clone` 关键字实现。

第6章 内部类

可以将一个类的定义放在另一个类的定义内部，这就是内部类。

内部类的对象与制造它的外围对象（enclosing object）之间就有一种联系，它能访问其外围对象的所有成员，而不要任何特殊条件。内部类还拥有其外围类的所有元素的访问权。

6.1 .this 和.new

Anonymous Inner Class

It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overloading methods of a class or interface, without having to actually subclass a class.

are mainly created in two ways:

- . Class (abstract or concrete) . interface

The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code. 匿名类表达式的语法类似于调用构造函数，只是在代码块中包含了一个类定义。

匿名内部类的创建格式

```
new class(arguments) | interface()
{
    //匿名内部类的类体部分
}
```

okHttp 中匿名类示例

```
public abstract class Internal {
    public abstract void addLenient(Headers.Builder builder, String line);
}

Internal.instance = new Internal() {
    @Override public void addLenient(Headers.Builder builder, String line) {
        builder.addLenient(line);
    }
}
```

Difference between Normal/Regular class and Anonymous Inner class:

A normal class can implement any number of interfaces but anonymous inner class can implement only one interface at a time. A regular class can extend a class and implement any number of interface simultaneously. But anonymous Inner class can extend a class or can implement an interface but not both at a time. For regular/normal class, we can write any number of constructors but we cant write

any constructor for anonymous Inner class because anonymous class does not have any name and while defining constructor class name and constructor name must be same.

6.1.1 嵌套类

将内部类声明为 `static`, 内部类对象与其外围类对象之间则没有联系。

1. 要创建嵌套类的对象，并不需要其外围类的对象。2. 不能从嵌套类的对象中访问非静态的外围类对象。

第7章 范型

7.1 基本概念

范型实现了参数化类型的概念。

7.2 范型类

7.3 范型方法

7.4 通配符类型

? extends T

? super T

无限定通配符

<?>

7.5 范型代码和虚拟机

虚拟机没有泛型类型对象——所有对象都属于普通类。

7.5.1 类型擦除

擦除类型变量并替换为限定类型（无限定的变量用 Object）。

原始类型用第一个限定的类型变量来替换，如果没有给定限定就用 Object 替换。

例如类 Pair<T> 中的类型变量没有显式的限定，因此原始类型用 Object 替换 T。为了提高效率应该将标签（tagging）接口（即没有方法的接口）放在边界列表的末尾。

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
    private T lower;
    private T upper;

    public Interval(T first, T second){

    }

    public class Interval implements Serializable
    {
        private Comparable lower;
        private Comparable upper;
    }
}
```

```
public Interval(Comparable first, Comparable second){
    }
}
```

7.5.2 翻译泛型表达式

当程序调用泛型方法时，如果擦除返回类型，编译器插入强制类型转换。

编译器会把方法调用翻译为两条虚拟机指令：

1. 对原始方法的调用
2. 将返回的 `Object` 类型（无限定类型为例）强制转化为特定类型。

桥方法被合成来保持多态。

7.5.3 约束和局限性

不能使用基本类型实例化类型参数

运行时类型查询只适用于原始类型

不能创建参数化类型的数组

可以声明通配类型的数组，然后进行类型转换：

```
Pair<String>[] table = new Pair<String>[10]; // Error
Pair<String>[] table = (Pair<String>[] ) new Pair<?>[10];
```

如果需要收集参数化类型对象，只有一种安全而有效的方法：使用 `ArrayList<ArrayList<Pair<String>>`。

Varargs 警告

```
public static <T> void addAll(Collection<T> coll, T... ts){
}
```

不能实例化类型变量

不能使用像 `new T(...)`, `new T[...]` 或者 `T.class` 这样的表达式中的类型变量。错误实例

```
public Pair(){
    first = new T();    second = new T(); // Errors
}
```

```
Pair<String> p = Pair.makePair(String::new);
public static <T> Pair<T> makePair(Supplier<T> constr){
```

```
        return new Pair<>(constr.get(), constr.get());
    }

Pair<String> otherPair = Pair.makePairOther(String.class);
public static <T> Pair<T> makePairOther(Class<T> cl){
    try{
        return new Pair<>(cl.newInstance(), cl.newInstance());
    }catch(Exception ex){
        return null;
    }
}
```

不能构造泛型数组

泛型类的静态上下文中类型变量无效

不能抛出或者捕获泛型类的实例

可以消除对受查异常的检查

第8章 集合

java 容器类可以划分为两个不同的概念

Collection

一个独立元素的序列。

- List 必须按照插入的顺序保存元素
- Set 不能有重复元素
- Queue 按照队列规则来确定对象产生的顺序

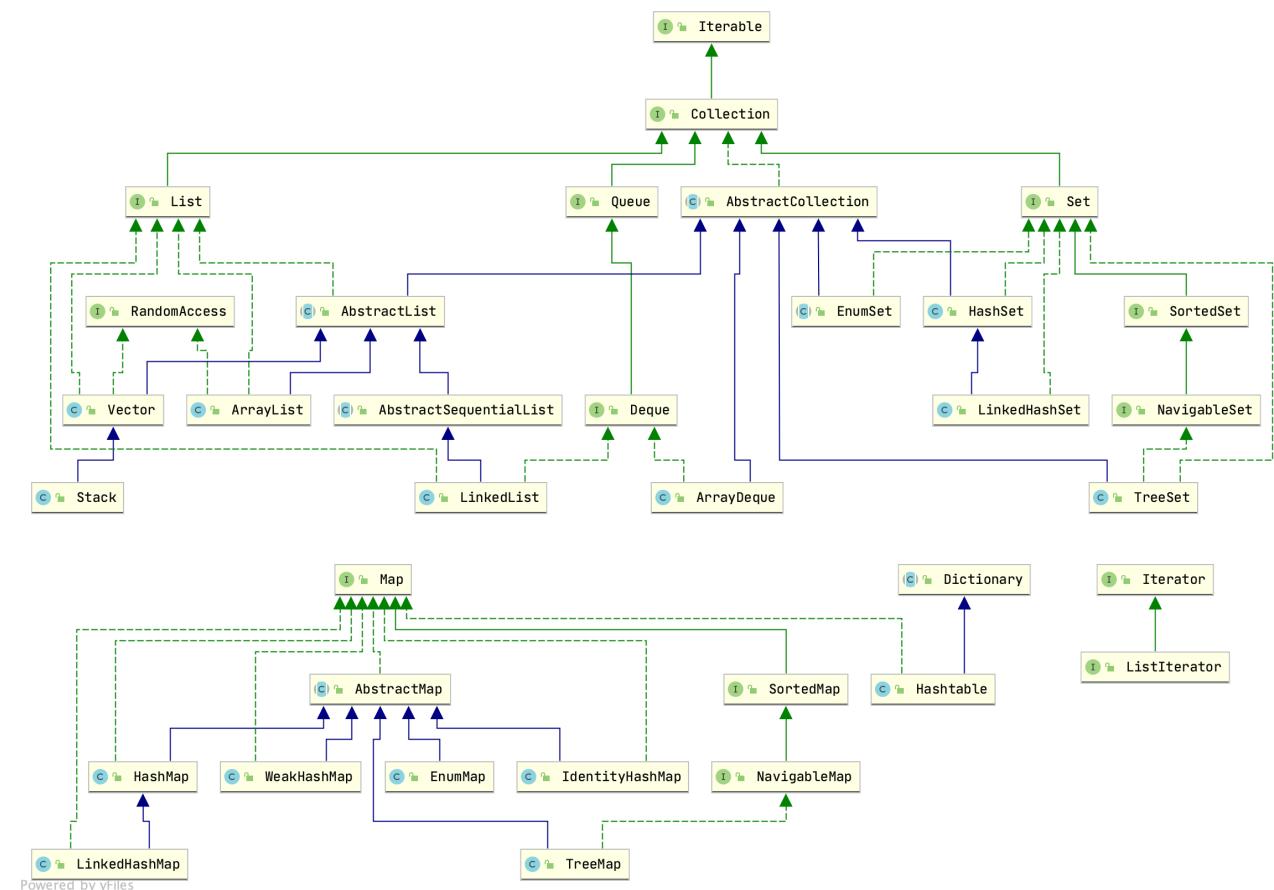
ArrayList、LinkedList、ArrayDeque、Vector、Stack

EnumSet、SortedSet、HashSet、TreeSet、LinkedHashSet

Map

一组成对的“键值对”对象，允许你使用键来查找值。

HashMap、TreeMap、LinkedHashMap、WeakHashMap、EnumMap、IdentityHashMap



ArrayList

`Arrays.asList()` 产生的 List 底层为数组，因此不能调整尺寸。

第9章 函数式接口

9.1 函数式接口

如果一个接口只有一个抽象方法，那么该接口就是一个函数式接口

如果我们在某个接口上声明了 `FunctionalInterface` 注解，那么编译器就会按照函数式接口的定义来要求该接口

如果某个接口只有一个抽象方法，但我们并没有给该接口声明 `FunctionalInterface` 注解，那么编译器依旧会将该接口看作是函数式接口

```
List<Integer> list = Arrays.asList(1,2,3,4,5);

list.forEach(new Consumer<Integer>(){
    @Override
    public void accept(Integer integer){
        System.out.println(integer);
    }
});
```

在 java 中 `Lambda` 表达式是对象，他们必须依附与一类特别的对象类型-函数式接口 (`functional interface`)

That instances of functional interfaces can be created with lambda expressions, method references, or constructor references.

函数式接口的实例可以通过 `lambda` 表达式，方法引用和构造函数引用创建。

外部迭代和内部迭代

Java `lambda` 表达式是一种匿名函数；它是没有声明的方法，即没有访问修饰符，返回值声明和名字。

`lambda` 操作符: `->` `lambda` 左边: 接口中抽象方法的形参列表 `lambda` 右边: 重写抽象方法的方法体

当只有一个参数，并且类型可推导时，圆括号 `()` 可省略。例如: `a-> return a*a`

`lambda` 表达式的主体可包含零条或多条语句

如果 `lambda` 表达式的主体只有一条语句，花括号 可省略。匿名函数的返回类型与该主体表达式一致

如果 `Lambda` 表达式的主体包含一条以上语句，则表达式必须包含在花括号中。匿名函数的返回类型与代码块的返回类型一致，若没有返回值则为空

`Function<T, R>`

`BiFunction<T, U, R>`

`Supplier<T>`

第 10 章 线程

10.1 线程基础知识

10.1.1 线程状态

JVM 线程的状态定义在 `Thread.State` 枚举中。包括：

- NEW 新建
- RUNNABLE 就绪
- BLOCKED 阻塞
- WAITING
- TIMED_WAITING
- TERMINATED

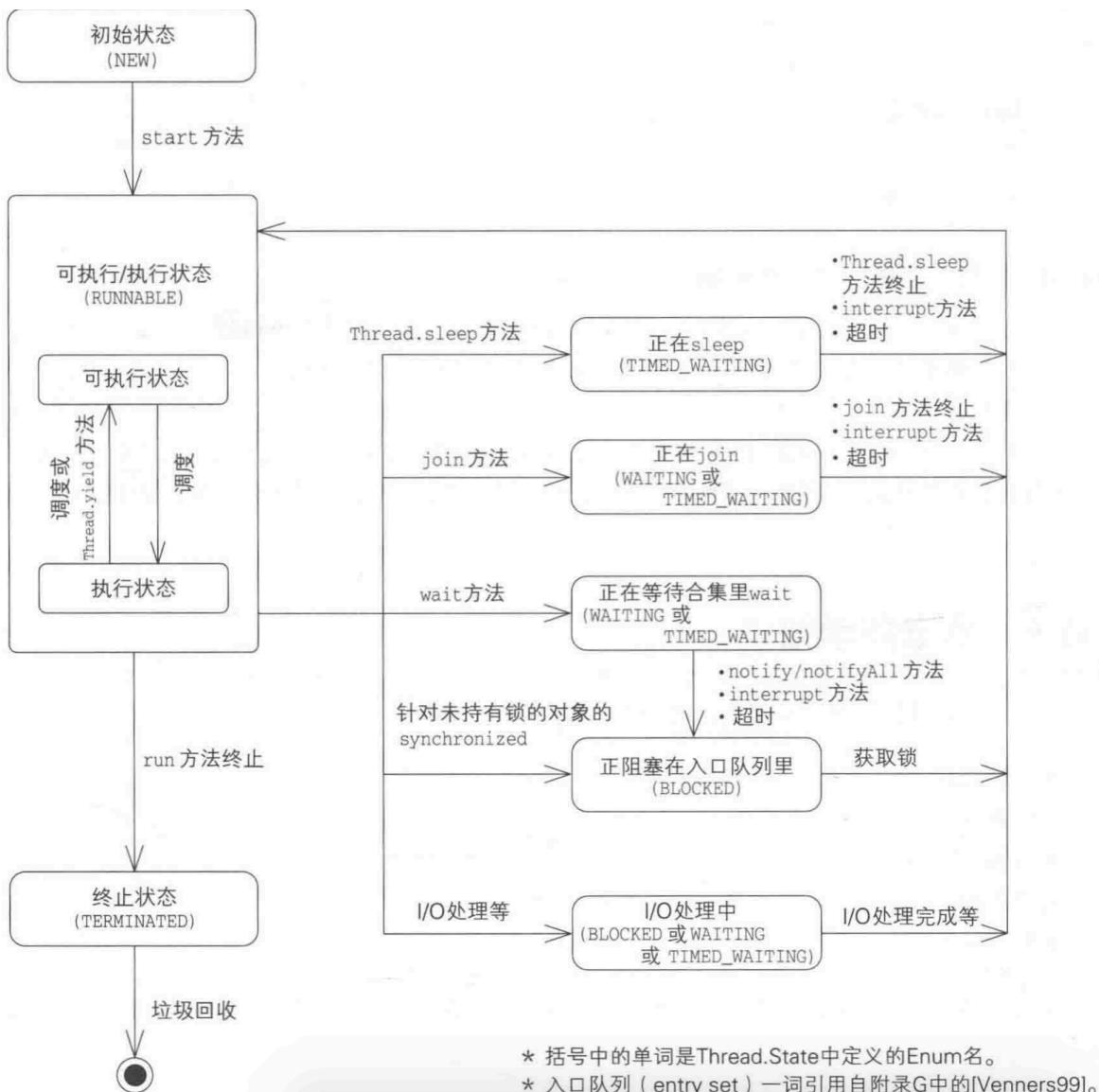


图 10.1: 线程状态迁移图 (摘自《图解 Java 多线程设计模式》)

NEW

当线程被创建时，它只会短暂地处于这种状态。此时它已经分配了必须的系统资源，并执行了初始化。

RUNNABLE

只要调度器把时间片分配给线程，线程就可以运行。

下面为 State 枚举

```
public enum State {
    /**
     * 尚未启动的线程的线程状态
     */
    NEW,
    /**
     * 可运行线程的线程状态。 处于可运行状态的线程正在Java虚拟机中执行,

```

```

    * 但是它可能正在等待来自操作系统（例如处理器）的其他资源。
    */
RUNNABLE,

/***
 * 线程的阻塞状态，等待监视器锁定。
 * 处于阻塞状态的线程正在等待一个监视器锁进入一个同步块/方法，
 * 或者在调用Object.wait()后重新进入一个同步块/方法。
 */
BLOCKED,

/***
 * 等待线程状态
 * 一个线程由于调用下列方法之一而处于等待状态：
 *
 * Object.wait() 未超时
 * Thread.join() 未超时
 * LockSupport.park()
 *
 * 处于等待状态的线程正在等待另一个线程执行特定的操作。
 * 例如：一个线程在一个对象上已经调用了Object.wait()，并正在等待在另一个线程在另一对象上调用
 * Object.notify() or Object.notifyAll()。
 * 调用 thread.join() 的线程正在等待指定的线程终止。
 */
WAITING,

/***
 * 具有指定等待时间的等待线程的状态。
 * 线程处于定时等待状态，因为调用以下方法之一与指定的正等待时间：
 *
 * Thread.sleep()
 * Object.wait(long)带超时参数
 * Thread.join(long)带超时参数
 * LockSupport.parkNanos
 * LockSupport.parkUntil
 *
 */
TIMED_WAITING,

/***
 * 终止状态。线程已经完成执行。
 */
TERMINATED;
}

```

10.1.2 创建线程的方式

创建线程的根本方法就是通过构造 `Thread` 类并且重写 `run()` 方法，然后调用 `start` 方法运行。

```

public Thread();
public Thread(String name);
public Thread(ThreadGroup group, String name);

public Thread(Runnable target);
public Thread(Runnable target, String name);
public Thread(ThreadGroup group, Runnable target, String name);
public Thread(ThreadGroup group, Runnable target, String name, long stackSize);

```

创建线程有四种方式

- 继承 `Thread` 类
- 实现 `Runnable` 接口
- 实现 `Callable` 和 `Future` 接口
- 使用线程池方式

实现 `Runnable` 接口

`Runnable` 接口

```
@FunctionalInterface  
public interface Runnable {  
    public abstract void run();  
}
```

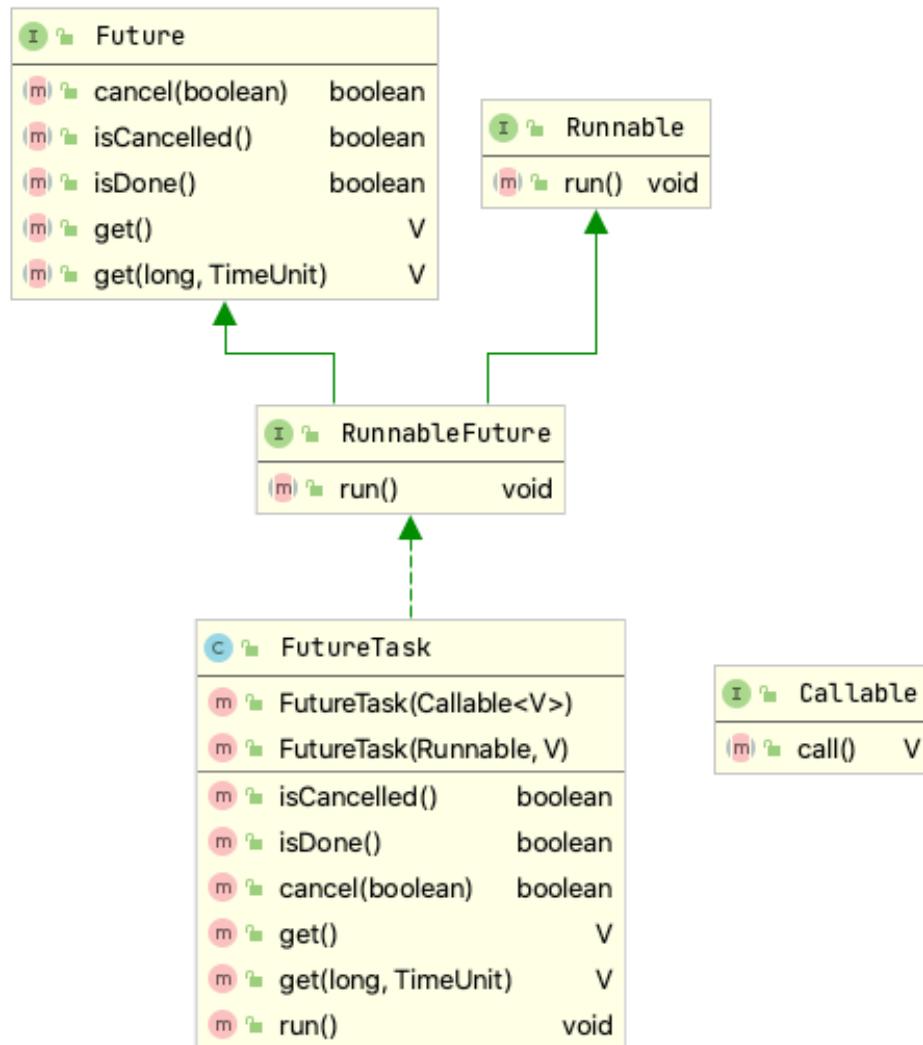
```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        new Thread(new HelloRunnable()).start();  
    }  
}
```

继承 `Thread` 类

`Thread` 类也继承了 `Runnable` 接口

```
public class MyThread extends Thread {  
    public void run() {  
        System.out.println(Thread.currentThread().getName() + ": Run!!!");  
    }  
  
    public static void main(String[] args) {  
        MyThread thread = new MyThread();  
        thread.start();  
    }  
}
```

实现 Callable 和 Future 接口



```

public class TaskWithResult implements Callable {

    private int id;
    public TaskWithResult(int id) {
        this.id = id;
    }

    @Override
    public Object call() throws Exception {
        return "Result of TaskWithResult is: " + id;
    }
}

public class FutureAndCallable {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        //不推荐使用Executors
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<String>> result = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            result.add(executorService.submit(new TaskWithResult(i)));
        }

        for (Future<String> future : result) {
            System.out.println(future.get());
        }
    }
}
  
```

```

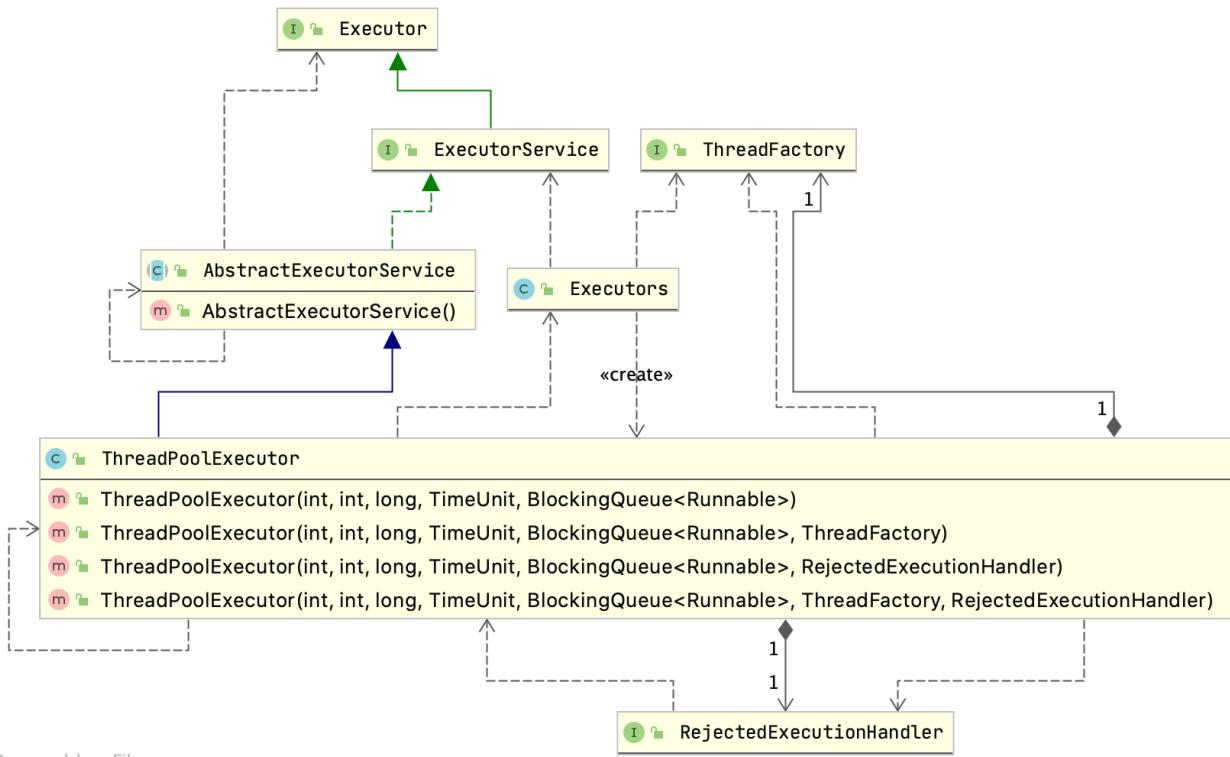
    }

    executorService.shutdown();
}

}

```

使用线程池方式



Powered by yFiles

线程池不允许使用 **Executors** 去创建，而是通过 **ThreadPoolExecutor** 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

Executors 提供了如下线程池

- `newSingleThreadExecutor`
- `newFixedThreadPool`
- `newCachedThreadPool`
- `newScheduledThreadPool`
- `newSingleThreadScheduledExecutor`
- `newWorkStealingPool`

**FixedThreadPool 和 SingleThreadExecutor:**

运行的请求队列长度为 `Integer.MAX_VALUE`, 可能会堆积大量的请求, 从而导致 OOM。

CachedThreadPool:

允许的创建线程数为 `Integer.MAX_VALUE`, 可能会创建大量的线程, 从而导致 OOM。

```
// 不推荐
ExecutorService executorService = Executors.newCachedThreadPool();

for (int i = 0; i < 5; i++) {
    executorService.execute(()->{
        System.out.println("Hello from a thread!");
    });
}

executorService.shutdown();

// 推荐: 指定最大线程数以及列队数
ExecutorService executor = new ThreadPoolExecutor(5, 5, 10L, TimeUnit.MILLISECONDS, new
    LinkedBlockingQueue<>(10));
for (int i = 0; i < 10; i++) {
    executor.execute(() -> System.out.println(Thread.currentThread().getName() + ":" +
        "ThreadPoolExecutor"));
}

executor.shutdown();
```

10.1.3 捕获异常

不能捕获从线程中逃逸的异常。一旦异常跳出任务的 `run()` 方法, 它就会向外传播到控制台。

可通过重写 `UncaughtExceptionHandler` 实现。使用 `setUncaughtExceptionHandler` 或者 `setDefaultUncaughtExceptionHandler` 方法。

10.1.4 常用函数

- `wait`
- `sleep`
- `join`
- `yield`
- `notify`
- `notifyAll`
- `interrupt`

wait

调用 `wait()` 会释放对象上的锁。

`wait()` 有两种形式。

第一种接受毫秒数做为参数，指“在此期间暂停”

在 `wait()` 期间对象锁是释放的可以通过 `notify()`,`notifyAll()` 或者令时间到期，从 `wait()` 中恢复执行。

第二种不接受任何参数

`wait()` 无限等待下去，直到线程接收到 `notify()` 或者 `notifyAll()` 的消息。

sleep

调用 `sleep()` 的时候锁并不会被释放。

yield

`yield()` 的时候锁并不会被释放。

join

`wait()`,`notify()`,`notifyAll()` 是基类 `Object` 的一部分，而不是属于 `Thread` 的一部分。只能在同步控制方法或者同步控制块里调用这三个方法，调用他们必须获取对象的锁。

为防止错失信号

```
synchronized(sharedMonitor){  
    while(someCondition){  
        shareMonitor.wait();  
    }  
}
```

10.1.5 终止

I/O 和在 `synchronized` 块上的等待是不可中断的。

10.2 synchronized

解决共享资源竞争

所有的并发模式在解决线程冲突问题时，都是采用序列化访问共享资源的方案。也就是说在给定时刻只允许一个任务访问共享资源。

因为锁语句产生了一种互相排斥的效果，所以这种机制常常称为互斥量（mutex）。

JDK 为什么要设计 synchronized？

要把共享资源包装进一个对象中。

多线程编程中，有可能出现多个线程同时访问同一个共享的，可变资源的情况。这种资源可能是对象、变量、文件等。

共享： 资源可以由多个线程同时访问

可变： 资源可以在其生命周期内被修改

引出的问题：

由于线程执行的过程是不可控的，所以需要采用同步机制来协同对对象可变状态的访问

加锁的目的：

序列号访问临界资源，即同一时刻只能有一个线程访问临界资源（同步互斥访问）



synchronized 属于隐式锁

隐式锁：不需要自己通过写代码去加锁跟解锁

10.2.1 synchronized 使用与实现

使用方式

- 同步实例方法，锁是当前实例对象
- 同步类方法，锁是当前类对象
- 同步代码块，锁是括号里面的对象

对于某个特定对象来说，其所有 synchronized 方法共享同一个锁。

实现方式

synchronized 是 JVM 内置锁，通过内部对象 Monitor（监视器锁）实现，基于进入与退出 Monitor 对象实现方法与代码块同步。监视器锁的实现依赖底层操作系统的 Mutex lock（互斥锁）实现。

JVM 对象加锁原理

认识对象的内存结构

oop.hpp

Object Header

实例对象内存存储是怎样的？对象的实例存在堆区，对象元数据存在元空间（方法区），对象的引用存在与栈空间

锁的种类

JDK1.6 之前是重量级锁，之后对 `synchronized` 对实现进行了优化，如适应性自旋锁，轻量级锁和偏向锁。默认开启偏向锁。

开启偏向锁：`-XX:+UseBiasedLocking -XX:BiasedLockingStartupDelay=0`

锁的升级过程

无锁 \rightarrow 偏向锁 \rightarrow 轻量级锁 \rightarrow 重量级锁

为什么不直接上重量级锁

上重量级锁会涉及到线程上下文切换？线程上下文切换会涉及到操作系统用户态到内核态的切换。

`volatile` 保证了原子性和可见性。

`volatile` 当一个域的值依赖它之前的值（例如递增一个计数器）`volatile` 就无法工作。

如果某个域的值受到其他域的值的限制，那么 `volatile` 也无法工作。

`volatile` 会告诉编译器不要执行任何移除读取和写入操作的优化。

10.2.2 原子性

原子性可以应用于除 `long` 和 `double` 之外的所有基本类型之上的“简单操作”。对于读取和写入除 `long` 和 `double` 之外的基本类型变量这样的操作，可以保证他们会被当作不可分（原子）的操作来操作内存。



JVM 将 64 位（`long` 和 `double` 变量）的读取和写入当作两个分离的 32 位操作来执行，这就导致了不同的任务可以看到不正确的结果的可能。（字撕裂）

10.2.3 可见性

JVM 原子操作

`lock, read, load, use, assign, store, write, unlock`

`read`：读取，从主内存读取数据

`load`：载入，将主内存读取到的数据写入工作内存

`use`：使用从工作内存读取数据来计算

assign: 赋值将计算好的值重新赋值到工作内存中

store: 存储将工作内存数据写入主内存

write: 写入将 **store** 过去的变量值赋值给主内存中的变量

lock: `锁定` 将主内存变量加锁，标识为线程独占状态

unlock: `解锁` 将主内存变量解锁，解锁后其他线程可以锁定该变量

MESI 缓存一致协议

<https://www.slideshare.net/cnbailey/memory-efficient-java>

<https://github.com/farmerjohngit/myblog>

<https://www.pdai.tech/>

第 11 章 Java Tools

11.1 javac 命令

Reads Java class and interface definitions and compiles them into bytecode and class files.

11.2 javap 命令

use the `javap` command to disassemble one or more class files.

用法: `javap <options> <classes>`

<code>-version</code>	版本信息
<code>-v -verbose</code>	输出附加信息
<code>-l</code>	输出行号和本地变量表
<code>-public</code>	仅显示公共类和成员
<code>-protected</code>	显示受保护的/公共类和成员
<code>-package</code>	显示程序包/受保护的/公共类和成员 (默认)
<code>-p -private</code>	显示所有类和成员
<code>-c</code>	对代码进行反汇编
<code>-s</code>	输出内部类型签名
<code>-sysinfo</code>	显示正在处理的类的系统信息 (路径, 大小, 日期, MD5 散列)
<code>-constants</code>	显示最终常量
<code>-classpath <path></code>	指定查找用户类文件的位置
<code>-cp <path></code>	指定查找用户类文件的位置
<code>-bootclasspath <path></code>	覆盖引导类文件的位置

示例:

```
javap -c Main.class
```

11.2.1 Java 字节码

Instruction set

instructions fall into a number of broad groups:

- 加载和存储指令 (Load and store) (e.g. `aload_0, istore`)
- 算术与逻辑指令 (Arithmetic and logic) (e.g. `ladd, fcmpl`)
- 类型转换指令 (Type conversion) (e.g. `i2b, d2i`)
- 对象创建与操作指令 (Object creation and manipulation) (`new, putfield`)
- 堆栈操作指令 (Operand stack management) (e.g. `swap, dup2`)
- 控制转移指令 (Control transfer) (e.g. `ifeq, goto`)
- 方法调用与返回指令 (Method invocation and return) (e.g. `invokespecial, areturn`)

大多数的指令有前缀和 (或) 后缀来表明其操作数的类型。如下表:

Many instructions have prefixes and/or suffixes referring to the types of operands they operate on.

表 11.1: 前缀/后缀类型对照表

Prefix/suffix	Operand type
i	integer
l	long
s	short
b	byte
c	character
f	float
d	double
a	reference

Java bytecode

表 11.2: Java 字节码

mnemonic	stack [before]->[after]	description
aload_0	objectref	load a reference onto the stack from local variable 0

bytecode**code****jvms**

11.3 java 命令

Launches a Java application.

```
java [options] classname [args]
```

```
java [options] -jar filename [args]
```

通过启动 JRE 来调用指定的类，调用此类的 `main()` 方法。

```
public static void main(String[] args)
```

11.3.1 选项

- 标准选项
- 非标准选项
- 高级运行时选项 (Advanced Runtime Options)
- 高级 JIT 编译器选项 (Advanced JIT Compiler Options)
- 高级可维护性选项 (Advanced Serviceability Options)
- 高级垃圾收集选项 (Advanced Garbage Collection Options)

标准选项

Java 虚拟机 (JVM) 的所有实现都保证支持的标准选项。

```
* -agentlib:libname[=options]
```

加载指定的本机代理库。在库名之后，可以使用特定于库的以逗号分隔的选项列表。

如果指定 `-agentlib:foo` 选项，则 JVM 尝试加载位于由系统环境变量名为 `LD_LIBRARY_PATH` (在 OS X 系统下变量名为 `DYLD_LIBRARY_PATH`) 指定位置下的 `libfoo.so` 的库。

下面的示例将展示如何加载堆分析工具 (HPROF) 库，并且获取堆栈深度为 3，每 20msCPU 的简单采样信息：

```
-agentlib:hprof=cpu=samples,interval=20,depth=3
```

下面这个示例将展示如何加载 Java 调试线协议库并且监听 8000 端口的套接字连接，在主类加载之前挂起 JVM：

```
-agentlib:jdwp=transport=dt_socket,server=y,address=8000
```

```
XX:+PrintStringTableStatistics
```

11.4 jps 命令

Lists the instrumented Java Virtual Machines (JVMs) on the target system.

只适用于 HotSpot 虚拟机。

```
jps [ options ] [ hostid ]
```

`hostid` 可以是进程的标识符或者类似于 URL 的 `[protocol:][[//]hostname][:port][[/servername]]` 地址

如果 `jps` 命令没有指定 `hostid` 参数，那么工具只搜索本机运行的 JVM。如果指定了 `hostid`，那么他将通过指定的地址来搜索 JVM。使用指定 `hostid` 的主机必须运行着 `jstatd` 进程。

11.4.1 选项

`-q`

只输出 JVM 标识符的列表，不输出类名、JAR 文件名以及传递给 `main` 方法的参数。

`-m`

输出传递给 `main` 方法的参数。嵌入式 JVM 可能输出为空。

`-l`

输出应用程序主类的完整包名或应用程序 JAR 文件的完整路径名。

`-v`

输出传递给 JVM 的参数。

`-V`

只输出本地 JVM 标识符，不输出类名、JAR 文件以及传递给 `main` 方法的参数。

`-Joption`

将选项传递给 JVM，其中的选项是 Java 应用程序启动程序参考页面中描述的选项之一。例如，`-J-Xms48m` 将启动内存设置为 48 MB。

输出格式：

```
lvmid [ [ classname | JARfilename | "Unknown" ] [ arg* ] [ jvmarg* ] ]
```

示例：

```
jps
```

```
18027 Java2Demo.JAR
18032 jps
```

18005 jstat

```
jps -m remote.domain:2002  
3002 /opt/jdk1.7.0/demo/jfc/Java2D/Java2Demo.JAR  
3102 sun.tools.jstatd.jstatd -p 2002
```

11.5 jcmd 命令

Sends diagnostic command requests to a running Java Virtual Machine (JVM).

```
jcmd [-l|-h|-help]
jcmd pid|main-class PerfCounter.print
jcmd pid|main-class -f filename
jcmd pid|main-class command[ arguments]
```

jcmd 工具向 JVM 发送诊断命令请求。必须在当前运行着 JVM 机器上执行，并且具有与 JVM 具有相同的 user 和 group 标识。

运行 jcmd 不带参数或者使用 l 选项时，相当于 jps (11.4)，会输出 java 进程标识符。

如果将进程标识符 (pid) 或主类 (main-class) 作为第一个参数，则 jcmd 将诊断命令请求发送给具有指定标识符或发具有指定 main-class 名称的 Java 进程。

如果使用 0 作为进程标识符，则会将诊断命令请求发送给所有可用的 Java 进程。

* `Perfcounter.print`

打印指定 Java 进程可用的性能计数器。性能计数器的列表可能因 Java 进程而异。

* `-f filename`

从文件中读取诊断命令。文件中每个命令必须单独一行。以 # 开头的命令会被忽略。如果命令行中包含 stop 关键字则结束命令行读取。

* `command [arguments]`

要发送到指定 Java 进程的命令。可以通过向该进程发送 help 命令来获得给定进程的可用诊断命令列表。

如果参数中包含空格，则必须使用单引号或者双引号扩起来。注意使用转义字符 (\) 转义单/双引号。

```
jcmd 15 help      # 15为java进程id
#####
15:
The following commands are available:
JFR.stop
JFR.start
JFR.dump
JFR.check
VM.native_memory
VM.check_commercial_features
VM.unlock_commercial_features
ManagementAgent.stop
ManagementAgent.start_local
ManagementAgent.start
VM.classloader_stats
```

```
GC.rotate_log
Thread.print
GC.class_stats
GC.class_histogram
GC.heap_dump
GC.finalizer_info
GC.heap_info
GC.run_finalization
GC.run
VM.uptime
VM.dynlibs
VM.flags
VM.system_properties
VM.command_line
VM.version
help
```

```
jcmd 15 help GC.heap_info
```

第12章 内存模型

12.1 Java 内存模型与线程

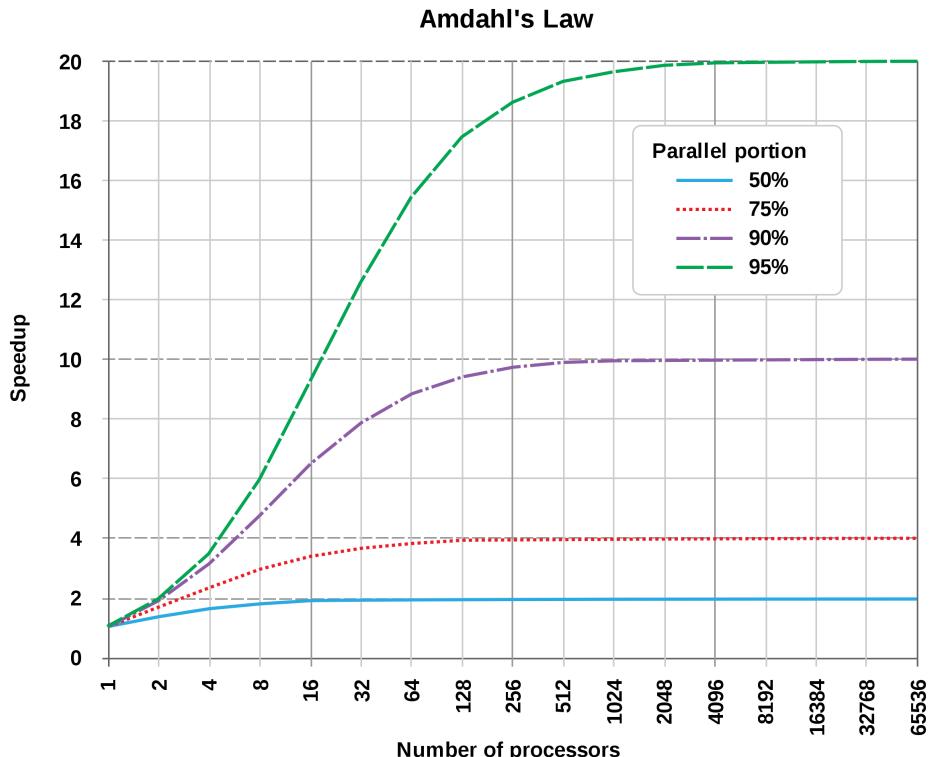


图 12.1: Amdahl's Law 阿姆达尔定律

并发不得不学的阿姆达尔定律。

一个程序（或者一个算法）可以按照是否可以被并行化分为下面两个部分：

- 可以被并行化的部分
- 不可以被并行化的部分

程序串行执行的总时间我们记为 T 。

时间 T 包括：不可以被并行和可以被并行部分的时间。不可以并行的部分记为 B ，可以被并行的部分就是 $T - B$ 。即 $T = B + (T - B)$

定义如下：

T = 串行执行的总时间

B = 不可以并行的总时间

$T - B$ = 并行部分的总时间

$T - B$ 是可并行化的部分，以并行的方式执行可以提高程序的执行速度。可以提速多少取决于有多少线程或者多少个 CPU 来执行。线程或者 CPU 的个数我们记为 N 。可并行化部分被执行的最快时间可以通过下面的公式计算出来：

$$(T - B) / N \text{ 或 } (1/N) * (T - B)$$

根据阿姆达定律，当可并行部分使用 N 个线程或 CPU 执行时，程序的总执行时间为：

$$T(N) = B + (T - B) / N$$



$T(N)$ 表示并行因子为 N 的总执行时间。 $T(1)$ 就是并行因子为 1 时的总执行时间。 $T(N) = B + (T(1) - B) / N$

示例

```
/*
 * 总时间为1。串行时间为0.4，那么并行时间为0.6。
 */

// 并行因子为2时
T(2) = 0.4 + ( 1 - 0.4 ) / 2
= 0.4 + 0.6 / 2
= 0.4 + 0.3
= 0.7

// 并行因子为5时
T(5) = 0.4 + ( 1 - 0.4 ) / 5
= 0.4 + 0.6 / 6
= 0.4 + 0.12
= 0.52
```

图示

并行因子分别为 1, 2, 3 时。

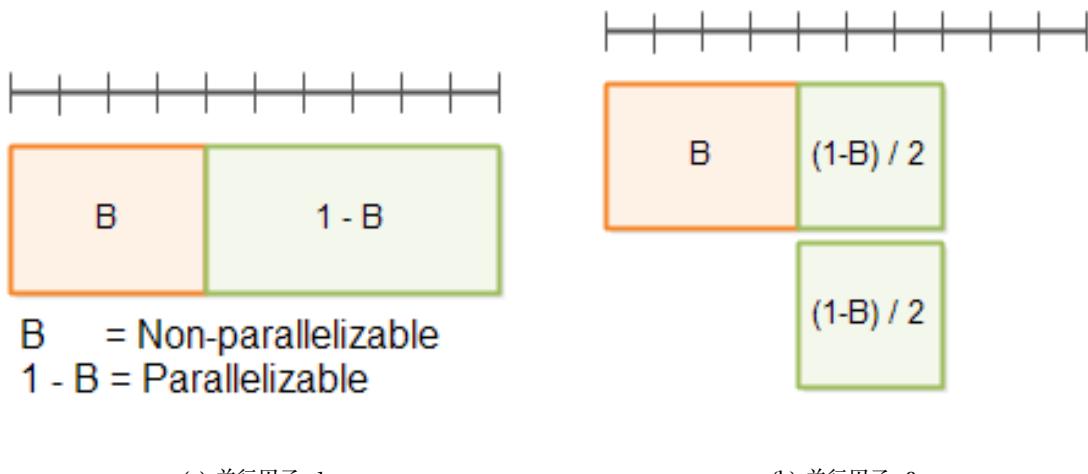


图 12.2: 并行因子分别为 1, 2 时

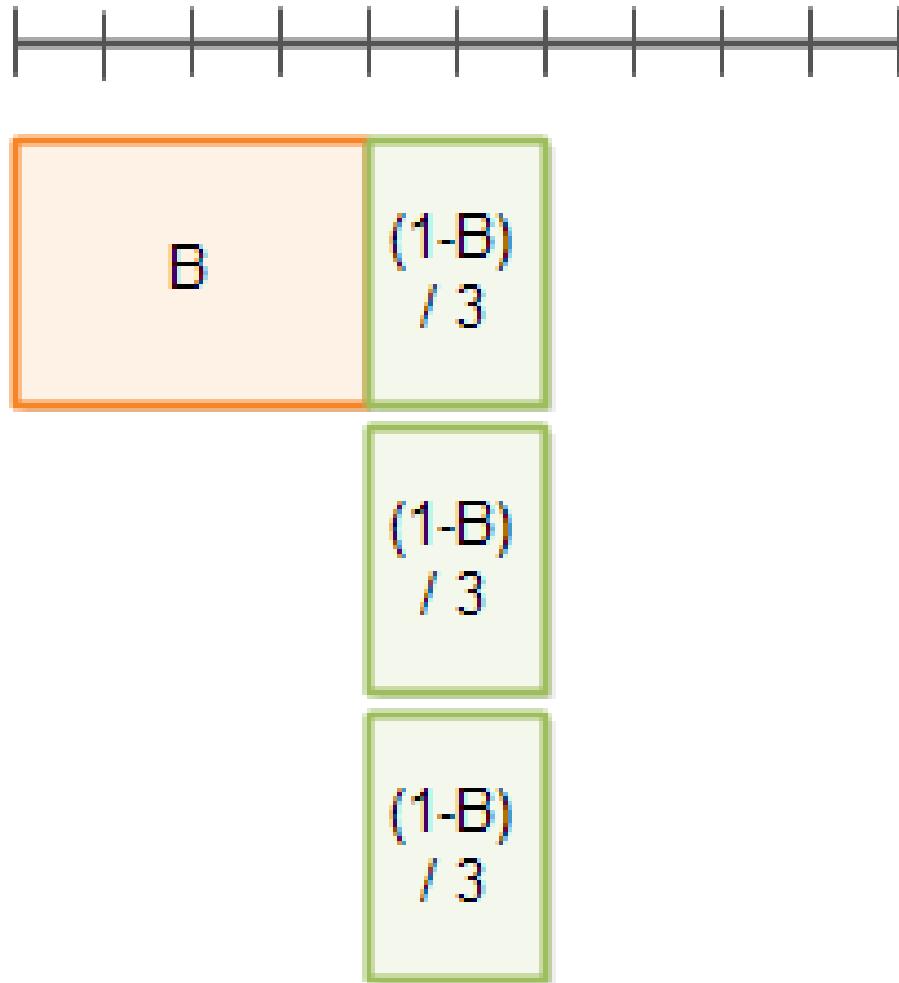


图 12.3: 并行因子 =3

定义

并行计算中的加速比是用并行前的执行速度和并行后的执行速度之比来表示的，它表示了在并行化之后的效率提升情况。

$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

图 12.4: 阿姆达尔定律

$S(\text{latency})$ 代表理论上的加速比

s 为并行处理结点个数

p 为并行计算部分所占比例

$1-p$ 为串行计算部分所占比例

这样：

当 $p = 1$ 时，最大加速比 $p = s$,

当 $p = 0$ 时，最小加速比 $S = 1$,

当 $s \rightarrow \infty$ 时，极限加速比 $S \rightarrow 1 / (1-p)$ ，这也就是加速比的上限。

例如，若加速前并行代码执行时间占整个代码的执行时间的 75% ($p=0.75$)，则加速后并行处理的总体性能的提升不可能超过原先的 4 倍。

因此可以推断出：

$$\begin{cases} S_{\text{latency}}(s) \leq \frac{1}{1-p} \\ \lim_{s \rightarrow \infty} S_{\text{latency}}(s) = \frac{1}{1-p}. \end{cases}$$

图 12.5: 阿姆达尔定律

阿姆达尔定律强调：当串行换比例一定时，加速比是有上限的，不管你堆叠多少个 CPU 参与计算，都不能突破这个上限。

12.1.1 Gustafson's law 古斯塔夫森定律

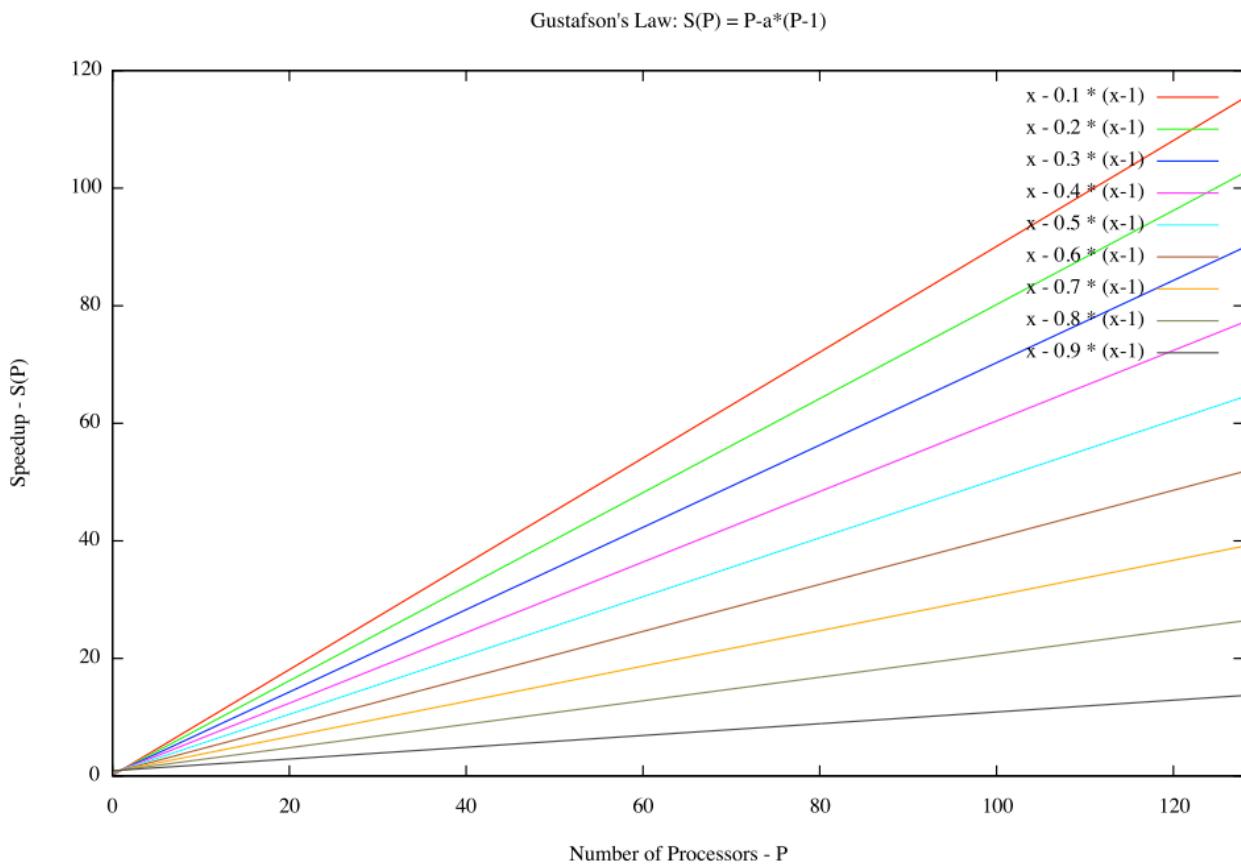


图 12.6: 古斯塔夫森定律

$$S_{\text{latency}}(s) = 1 - p + sp$$

图 12.7: 古斯塔夫森定律定义

古斯塔夫森定律强调的是：如果可被并行化的代码所占比例足够大，那么加速比就能随着 CPU 的数量线性增长。

12.1.2 JAVA 内存模型 JMM

JAVA 内存模型 Java Memory Model JMM

第13章 Linux

Linux Static Performance Tools

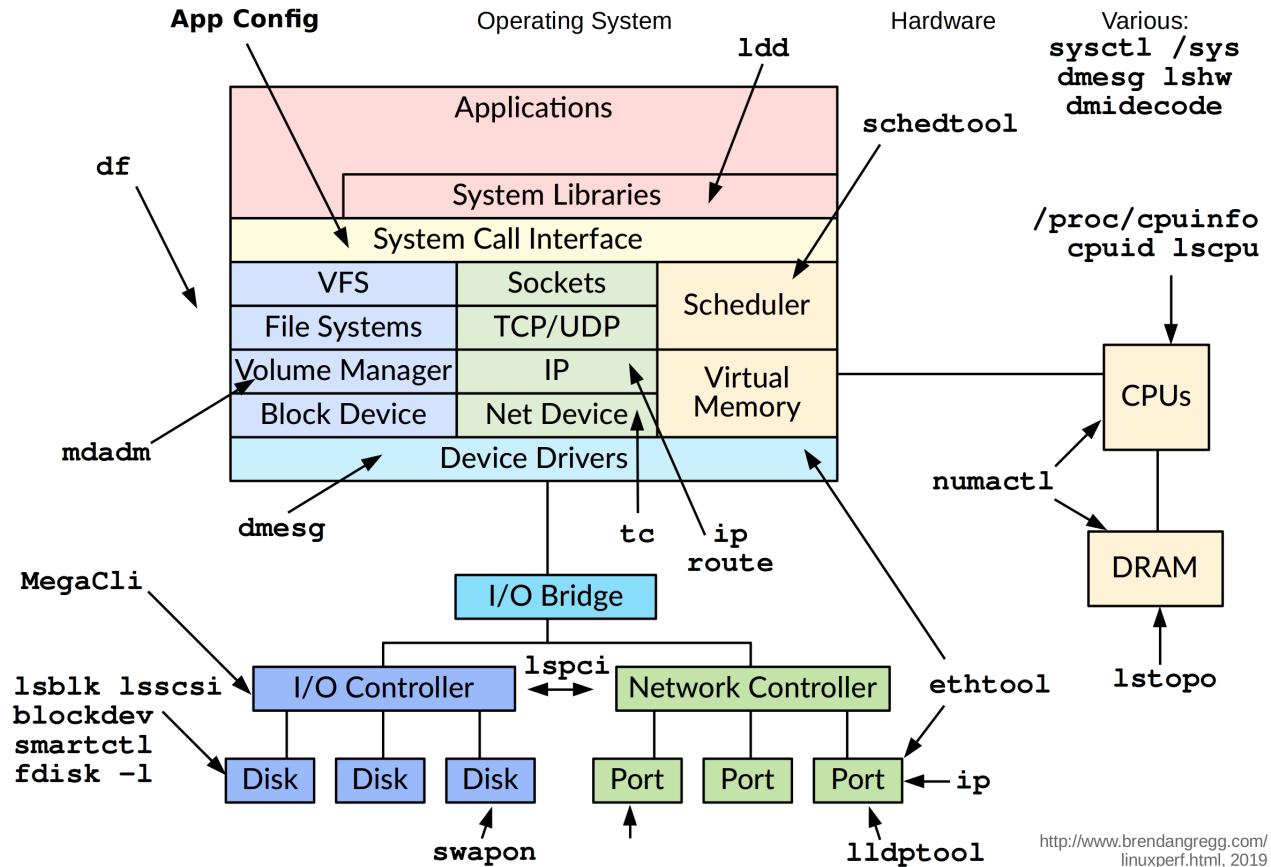


图 13.1: linux statis tools

Linux Performance Observability Tools

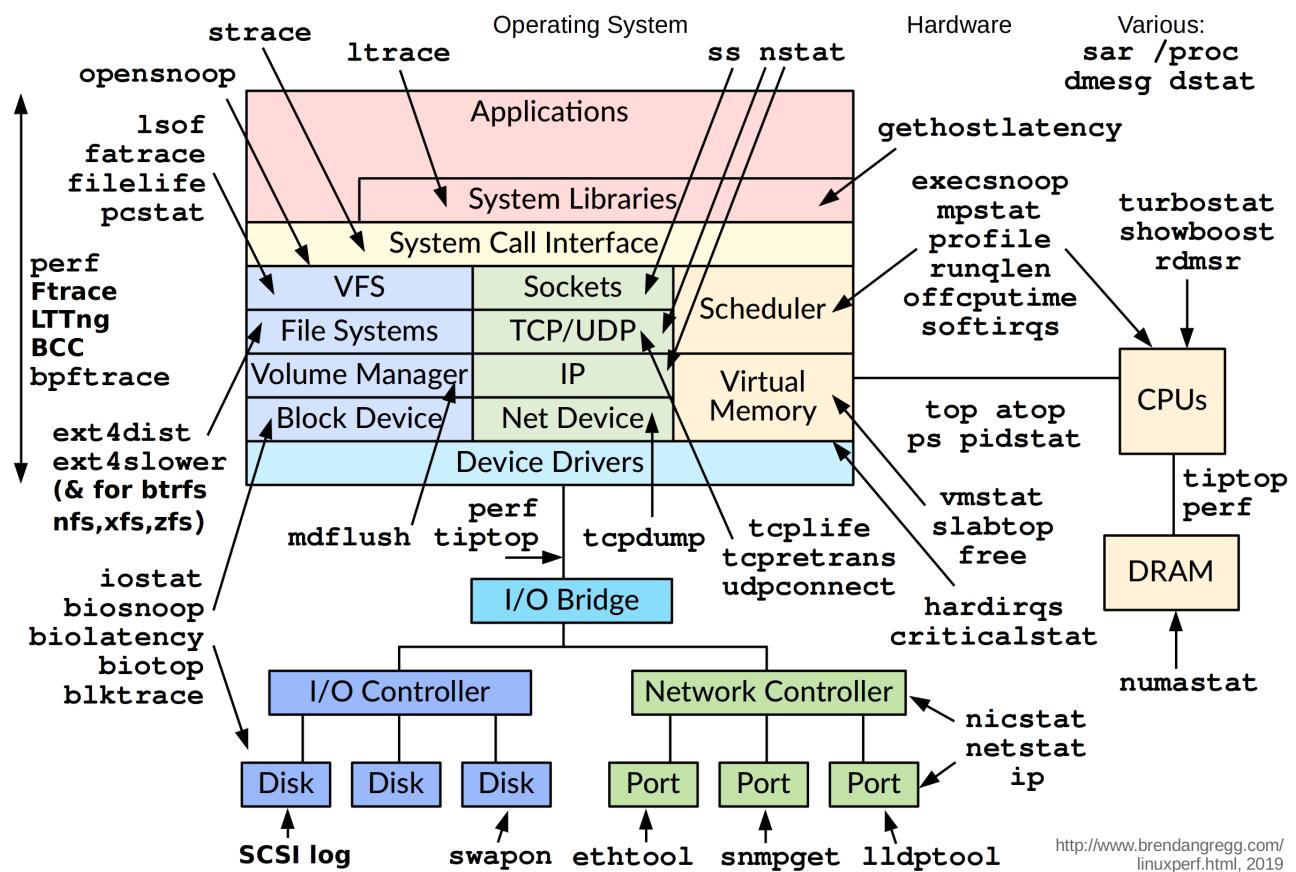


图 13.2: linux observability tools

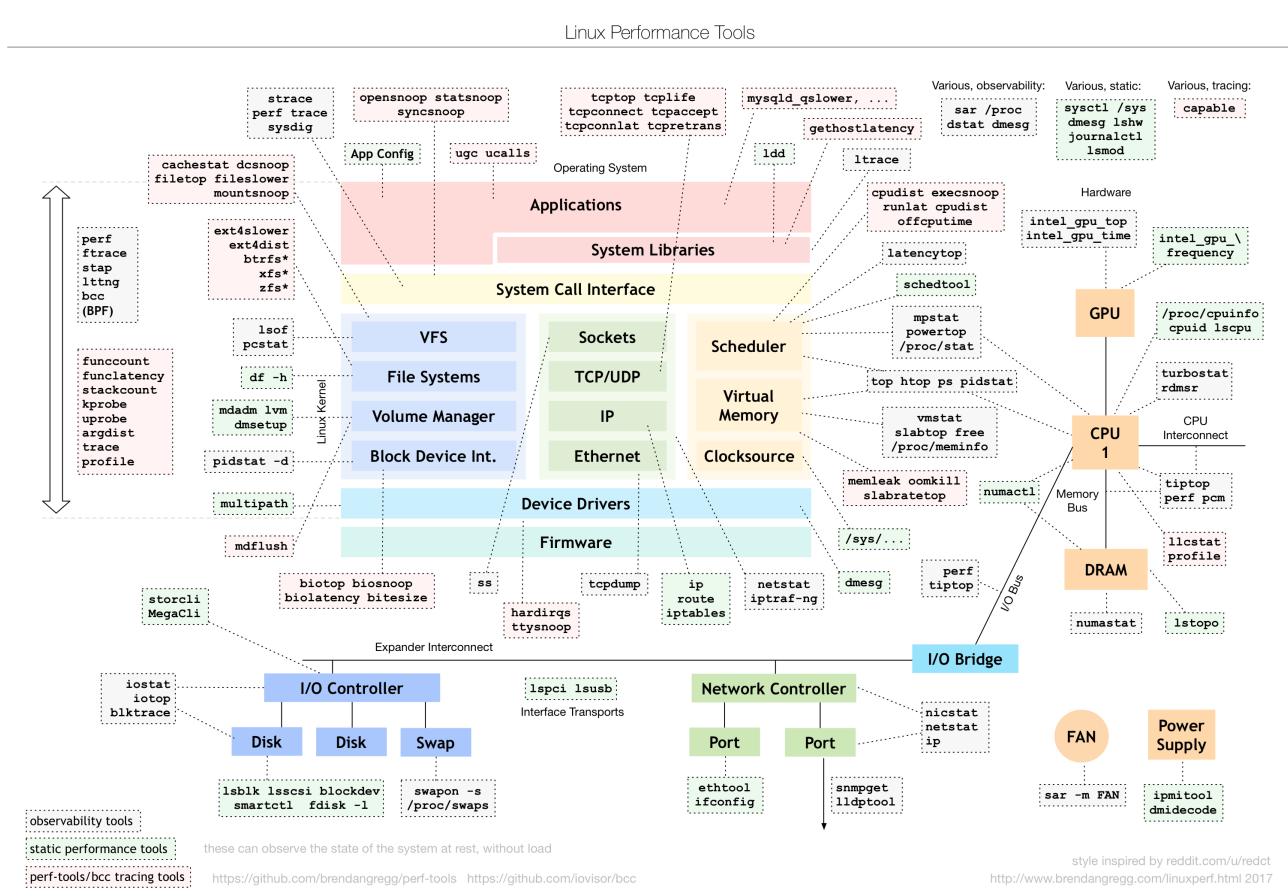


图 13.3: linux performance tools

13.1 uptime

uptime

当前时间系统运行了多长时间、多少用户登陆；以及过去 1 分钟、5 分钟、15 分钟的系统平均负载。

```
$ uptime  
  
# 20:08:43 up 11 days, 6:06, 0 users, load average: 21.76, 21.24, 19.93  
  
# 20:08:43      系统当前时间  
# up 11 days, 6:06      系统运行时间  
# 0 users      当前登陆用户数  
# load average: 21.76, 21.24, 19.93    过去1分钟、5分钟、15分钟的系统平均负载
```

系统负载

系统负载是处于可运行 (runnable) 或不可中断 (uninterruptable) 状态的进程的平均数。

可运行 (runnable) 状态的进程要么正在使用 CPU，要么在等待使用 CPU。不可中断状态的进程则正在等待某些 I/O 访问，例如等待磁盘 IO。

负载均值的意义根据系统中 CPU 的数量不同而不同，负载为 1 对于一个只有单 CPU 的系统来说意味着负载满了，而对于一个拥有 4 CPU 的系统来说则意味着 75% 的时间里都是空闲的。

OPTIONS

-p, --pretty

以漂亮的格式显示正常运行时间

```
uptime -p  
  
# up 17 weeks, 1 day, 11 hours, 37 minutes
```

-s, --since

以 yyyy-mm-dd HH:MM:SS 格式显示系统启动时间

```
uptime -s  
  
# 2019-11-18 23:30:43
```

13.2 top

显示 Linux 进程信息

top 提供了运行系统的动态实时视图。它可以显示系统摘要信息以及当前由 Linux 内核管理的进程或线程的列表。所显示的系统摘要信息的类型以及为流程显示的信息的类型、顺序和大小都是用户可配置的该配置可以在重启时保持不变。

下文以 3.3.10 版本为例。

13.2.1 交互式命令

进入 **top** 命令页面后输入的命令

l 切换是否显示系统平均负载

c 切换程序名或命令行的显示

d 修改屏幕刷新时间

i 切换显示所以线程或任务

H 切换为线程模式

t 切换 Task/Cpu 显示状态

m 切换内存模式显示状态

f 增加/删除/排序字段

1/2/3 切换展示所有 CPU/numa/node 使用情况

L 查找

E/e 概括/线程内存预览

S 切换累积模式

13.2.2 OPTIONS

-b

批处理模式操作，可用于将 **top** 输出内容发送到其他程序或者文件中

-c

显示完整命令行或者程序名

-d :Delay-time interval as: -d ss.t (secs.tenths)

指定屏幕更新时间间隔；也可以使用交互命令 **d** 或者 **s**。可以填写分数秒。

-H :Threads-mode operation

指定显示各个线程，如果没有此选项，则显示每个进程中所有线程的总和。也可以使用交互命令 **H**。

-i :Idle-process toggle

使 `top` 不显示任何闲置或者僵死进程。

-n :Number-of-iterations limit as: -n number

循环显示的次数

-o :Override-sort-field as: -o fieldname

指定将对那些任务字段进行排序。可以在字段名前加上 + 或者-用来覆盖排序方向。前置 + 将强制从高到低排序，而-是从低到高排序。可用字段名可以参选下面的-0 选项。

-0 :Output-field-names

打印可用的字段名

-p :Monitor-PIDs mode as: -pN1 -pN2 ... or -pN1,N2,N3 ...

仅监视具有指定进程 id 的进程。此选项最多可以使用 20 次，或者您可以提供最多 20 个 pid 的逗号分隔列表。如果需要恢复，可以使用 =,u 或 U。

-S :Cumulative-time toggle

当启用累积时间模式时，每个进程都会列出它和它死去的子进程所使用的 cpu 时间。

-u | -U :User-filter-mode as: -u | -U number or name

只显示与给定的用户 id 或用户名匹配的进程。`-u` 选项匹配有效用户，而`-U` 选项匹配任何用户（实际的、有效的、已保存的或文件系统）。在用户 id 或名称前面加上感叹号 ('!')，指示 `top` 只显示与提供的用户不匹配的进程。

输出内容解释

```
top - 15:44:32 up 120 days, 16:13, 2 users, load average: 0.00, 0.01, 0.05
Tasks: 83 total, 1 running, 82 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.5 us, 0.5 sy, 0.0 ni, 99.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 3881072 total, 252396 free, 575456 used, 3053220 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 3017932 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
9280	mysql	20	0	1844228	399192	16648	S	1.0	10.3	975:57.07	mysqld
1490	root	20	0	609124	13968	2336	S	0.7	0.4	793:26.93	barad_agent
1	root	20	0	125416	3796	2436	S	0.0	0.1	11:01.11	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.90	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:25.91	ksoftirqd/0
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H
7	root	rt	0	0	0	0	S	0.0	0.0	0:33.92	migration/0
8	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_bh
9	root	20	0	0	0	0	S	0.0	0.0	27:12.99	rcu_sched

图 13.4: top

第一行输出任务为 UPTIME and LOAD Averages

```
# top - 15:52:24 up 120 days, 16:21, 2 users, load average: 0.14, 0.10, 0.07
```

第二行为任务（进程）或者线程信息；进程或者线程的切换可以通过交互命令 H。分为：运行；休眠；停止；僵尸

```
# Tasks: 86 total, 1 running, 85 sleeping, 0 stopped, 0 zombie
# Threads: 149 total, 1 running, 148 sleeping, 0 stopped, 0 zombie
```

第三行为 CPU 状态信息

```
# %Cpu(s): 0.7 us, 0.7 sy, 0.0 ni, 98.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
#      a   b     c   d
# %Cpu(s): 0.3/0.5    1[|]
```

us, user : 用户空间占用 CPU 百分比 (time running un-niced user processes)

sy, system : 内核空间占用 CPU 百分比 (time running kernel processes)

ni, nice : 用户进程空间内改变过优先级的进程占用 CPU 百分比 (time running niced user processes)

id, idle : 空闲 CPU 百分比 (time spent in the kernel idle handler)

wa, IO-wait : 等待输入输出的 CPU 时间百分比 (time waiting for I/O completion)

hi : 硬中断 (Hardware IRQ) 占用 CPU 的百分比 (time spent servicing hardware interrupts)

si : 软中断 (Software Interrupts) 占用 CPU 的百分比 (time spent servicing software interrupts)

st : hypervisor vm 占用的 CPU 时间的百分比 (time stolen from this vm by the hypervisor)

在使用交互命令 t 切换到 cpu 状态模式时，还会额外显示一行 CPU 摘要信息，如上面展示第二行

- a) 是 us 和 ni 之和
- b) 是 sy 百分比
- c) 是总和
- d) 是可视化图像

内存使用情况

默认情况下，第 1 行反映物理内存，分类为：

total(物理内存总量), free(空闲内存总量), used(使用的物理内存总量) and buff/cache(用作内核缓存的内存量)

第 2 行反映的主要是虚拟内存，分为以下几类：

total(交换区总量), **free**(闲交换区总量), **used**(使用的交换区总量) and **avail**(可用物理内存)

此行的 **avail** 指启动新应用程序时可用的物理内存的估计值不包含 **swapping**。与 **free** 命令不同的是, **free** 试图分析易于回收的页面缓存和内存片。

可以通过交互命令 **E** 切换单位

```
# MiB Mem : 3790.1 total,    228.7 free,    568.1 used,   2993.2 buff/cache
# MiB Swap:    0.0 total,     0.0 free,     0.0 used.  2941.0 avail Mem
```

在内存模式（使用交互命令 **m**）下，会显示两行简短的摘要：

```
a      b      c
MiB Mem : 23.2/3790.1  [|||||] ]
MiB Swap: 0.0/0.0      [ ]
```

- a) 使用百分比
- b) 总物理内存
- c) 图形化展示

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
9280	mysql	20	0	1844228	399192	16648	S	0.7	10.3	990:08.75	mysqld
1490	root	20	0	609124	13968	2336	S	0.3	0.4	803:34.96	barad_agent

13.2.3 列字段含义

1. %CPU -- CPU 使用率

CPU 使用时间占比；

2. %MEM -- 内存使用 (RES)

进程当前使用的可用物理内存占比

3. CGROUPS -- Control Groups

4. CODE -- Code Size (KiB)

可执行代码占用物理内存大小，也称为文本常驻集 (Text Resident Set) 大小或 TRS。

5. COMMAND -- 命令名/命令行

6. DATA -- Data + Stack Size (KiB) 数据段 + 栈

可执行代码以外的物理内存的大小，也称为数据常驻集大小或 DRS。

7. ENVIRON -- 环境变量

8. Flags -- Task Flags

This column represents the task's current scheduling flags which are expressed in

hexadecimal notation and with zeros suppressed. These flags are officially documented in `<linux/sched.h>`.

9. GID -- 有效组 ID

10. GROUP -- 有效组名

11. NI -- Nice Value 进程的 nice 值。nice 值越小优先级越高。0 表示不调整优先级。

12. P -- Last used CPU (SMP)

A number representing the last used processor. In a true SMP environment this will likely change frequently since the kernel intentionally uses weak affinity. Also, the very act of running top may break this weak affinity and cause more processes to change CPUs more often (because of the extra demand for cpu time).

13. PGRP -- 进程组 ID

14. PID -- 进程 ID

15. PPID -- 父进程 ID

16. PR -- 优先级

可以使用 `chrt` 命令查看进程调度

进程调度分为 Normal Scheduling 和 Real-time Scheduling

Normal Scheduling: SCHED_OTHER SCHED_IDLE SCHED_BATCH

Real-time Scheduling: SCHE_FIFO SCHED_RR

Priority value: [1,99] 值越低优先级越高

Normal Scheduling priority 默认值为 0

SCHED_OTHER: 对应 time-sharing policy (RR)

PR 值和 NICE 仅对 Normal Scheduling 进程有效: PR = 20 + NICE(友好值 [-20,19]) = [0,39] PR 值越高优先级越低

SCHED_FIFO 和 SCHED_RR real-time process/thread

priority = [1,99] PR = -1 - priority = [-100, -2]

进程调度优先级。如果在字段中看到 rt, 则说明该线程是 Real-time Scheduling。

17. RES -- 进程占用内存大小 (KiB)

进程使用的未被 swap 物理内存大小 (The non-swapped physical memory a task is using.)

18. RUID -- 真实用户 id

19. RUSER -- 真实用户名

20. S -- 进程状态 (Process Status)

- D = 不可中断睡眠状态 (uninterruptible sleep)
- R = 运行状态 (running)
- S = 睡眠状态 (sleeping)
- T = stopped by job control signal
- t = stopped by debugger during trace
- Z = 僵尸状态 (zombie)

D 状态一般为等待 IO，例如磁盘 IO、网络 IO 等。

21. SHR -- 共享内存大小 (Shared Memory Size) (KiB)

进程可用的共享内存大小，通常不是所有的都驻留。它只能反应可能与其他进程共享的内存。

22. SID -- 会话 ID

23. SUID -- Saved User Id

24. SUPGIDS -- 附属组 ID

25. SUPGRPS -- 附属组名

26. SUSER -- Saved User Name

27. SWAP -- Swapped Size (KiB) 线程地址空间的非驻留部分

28. Tgid -- 线程组 ID (Thread Group Id)

任务所属的线程组的 ID。它是线程组领导的 PID。在内核术语中，它表示那些共享 mm_struct 的任务。

29. TIME -- CPU Time

线程启动后使用的总 CPU 时间。当启用累积模式时，将列出每个进程以及其死去的字进程所使用的 CPU 时间。可以使用交互命令 S 来切换累积模式。

30. TIME+ -- CPU Time, hundredths

与 TIME 相同，但通过百分之一秒反映出更细的粒度。

31. TPGID -- Tty Process Group Id

32. TTY -- Controlling Tty

33. UID -- 用户 ID; 进程所有者的真实用户 ID

34. USED -- Memory in Use (KiB)

此字段表示线程使用的非 swapped 物理内存 (RES) 及其地址空间的非驻留部分 (SWAP)。

35. USER -- 用户名; 进程所有者的真实用户名

36. VIRT -- 虚拟内存大小 (Virtual Memory Size)(KiB)

进程使用的虚拟内存大小。包括代码、数据以及共享库以及已换出的页和已映射但未使用的页。

37. WCHAN -- Sleeping in Function

若该进程在睡眠，则显示睡眠中的系统函数名。

38. nDRT -- 脏页数量 (Dirty Pages Count)

The number of pages that have been modified since they were last written to auxiliary storage. Dirty pages must be written to auxiliary storage before the corresponding physical memory location can be used for some other virtual page.

39. nMaj -- Major Page Fault Count

The number of major page faults that have occurred for a task. A page fault occurs when a process attempts to read from or write to a virtual page that is not currently present in its address space. A major page fault is when auxiliary storage access is involved in making that page available.

40. nMin -- Minor Page Fault count

The number of minor page faults that have occurred for a task. A page fault occurs when a process attempts to read from or write to a virtual page that is not currently present in its address space. A minor page fault does not involve auxiliary storage

access in making that page available.

41. nTH -- 线程数量

42. nsIPC -- IPC namespace

The Inode of the namespace used to isolate interprocess communication (IPC) resources such as System V IPC objects and POSIX message queues.

43. nsMNT -- MNT namespace

The Inode of the namespace used to isolate filesystem mount points thus offering different views of the filesystem hierarchy.

44. nsNET -- NET namespace

The Inode of the namespace used to isolate resources such as network devices, IP addresses, IP routing, port numbers, etc.

45. nsPID -- PID namespace

The Inode of the namespace used to isolate process ID numbers meaning they need not remain unique. Thus, each such namespace could have its own `init' (PID #1) to manage various initialization tasks and reap orphaned child processes.

46. nsUSER -- USER namespace

The Inode of the namespace used to isolate the user and group ID numbers. Thus, a process could have a normal unprivileged user ID outside a user namespace while having a user ID of 0, with full root privileges, inside that namespace.

47. nsUTS -- UTS namespace

The Inode of the namespace used to isolate hostname and NIS domain name. UTS simply means "UNIX Time-sharing System".

48. vMj -- Major Page Fault Count Delta

The number of major page faults that have occurred since the last update (see nMaj).

49. vMn -- Minor Page Fault Count Delta

The number of minor page faults that have occurred since the last update (see nMin).

13.2.4 示例

PID	USER	PR	NI	VIRT	RES	SHR	%CPU	%MEM	TIME+	COMMAND
290	root	20	0	39.0g	4.6g	6968	2677.2	3.6	1:20.61	java -Xbootcl
291	root	20	0	39.0g	4.6g	6968	2478.6	3.6	1:14.63	java -Xbootcl
289	root	20	0	39.0g	4.6g	6968	67.1	3.6	0:02.02	java -Xbootclas
292	root	20	0	39.0g	4.6g	6968	9.6	3.6	0:00.29	java -Xbootclas

图 13.5: top cpu 利用率超过 100%

如上图所示: CPU 使用率超过 100%

这里 CPU 使用率是所有 CPU 的总和。即你的 CPU 是多核处理器。可以使用交互式命令 1 来查看所有 CPU 的使用情况。

13.3 free

显示系统中空闲和使用的物理内存以及 swap 内存量。还有内核使用的缓冲区和缓存。这些信息来源于/proc/meminfo。

total

总内存大小 (MemTotal and SwapTotal in /proc/meminfo)

used

已使用的内存大小 (calculated as total - free - buffers - cache) free

未使用的内存 (MemFree and SwapFree in /proc/meminfo)

shared

(主要) 由 tmpfs 使用的内存

buffers

内核缓冲区使用的内存 (Buffers in /proc/meminfo)

cache

page cache and slabs (Cached and SReclaimable in /proc/meminfo)

buff/cache

buffers and cache 总和

available

Estimation of how much memory is available for starting new applications, without swapping. Unlike the data provided by the cache or free fields, this field takes into account page cache and also that not all reclaimable memory slabs will be reclaimed due to items being in use (MemAvailable in /proc/meminfo, available on kernels 3.14, emulated on kernels 2.6.27+, otherwise the same as free)



linux 系统内核为文件设置了一个缓存，对文件读写的数据内容都缓存在这里。
这个缓存称为 page cache(页缓存)。

13.4 meminfo

13.5 slabinfo

13.6 vmstat

第 14 章 IPv6

14.1 基础知识

14.1.1 IPv6 格式

IPv6 二进位制下为 128 位长度。

14.1.2 问题

1. IP 地址转为 int 实现
2. DNS 域名解析中 A、AAAA、CNAME、MX、NS、TXT、SRV、SOA、PTR 各项记录的作用

<https://blog.hackroad.com/operations-engineer/basics/13255.html>

DNS Tunneling

[DNS Tunneling 及相关实现](#)

第 15 章 Http/2

HTTP Timeline



图 15.1: HTTP 时间轴

i | HTTP 标准并未规定 TCP 就是唯一的传输协议。

15.1 HTTP 历史

15.1.1 HTTP/0.9

- 基于 TCP 协议
- 单行请求 GET 要请求的文档的路径 回车符 (CRLF) 结尾
- 连接在文档传输完毕后断开

重现 0.9 时代

```
telnet 192.168.31.59 80      # 命令

Trying 192.168.31.59...
Connected to 192.168.31.59.

GET /index.html                # 命令

output something...

Connection closed by foreign host.
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.00...	192.168.31.222	192.168.31.59	TCP	78	64848 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64
2	0.00...	192.168.31.59	192.168.31.222	TCP	74	80 → 64848 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 TSval=10 TStamp=0.000
3	0.00...	192.168.31.222	192.168.31.59	TCP	66	64848 → 80 [ACK] Seq=1 Ack=1 Win=131712 Len=0 TSval=10 TStamp=0.000
4	5.69...	192.168.31.222	192.168.31.59	TCP	83	64848 → 80 [PSH, ACK] Seq=1 Ack=1 Win=131712 Len=17 TSval=17 TStamp=5.690
5	5.70...	192.168.31.59	192.168.31.222	TCP	66	80 → 64848 [ACK] Seq=1 Ack=18 Win=29056 Len=0 TSval=22 TStamp=5.700
6	5.70...	192.168.31.59	192.168.31.222	TCP	165	80 → 64848 [PSH, ACK] Seq=1 Ack=18 Win=29056 Len=99 TSval=22 TStamp=5.700
7	5.70...	192.168.31.59	192.168.31.222	TCP	66	80 → 64848 [FIN, ACK] Seq=18 Ack=18 Win=29056 Len=0 TSval=22 TStamp=5.700
8	5.70...	192.168.31.222	192.168.31.59	TCP	66	64848 → 80 [ACK] Seq=18 Ack=100 Win=131648 Len=0 TSval=22 TStamp=5.700
9	5.70...	192.168.31.222	192.168.31.59	TCP	66	64848 → 80 [ACK] Seq=18 Ack=101 Win=131648 Len=0 TSval=22 TStamp=5.700
10	5.70...	192.168.31.222	192.168.31.59	TCP	66	64848 → 80 [FIN, ACK] Seq=18 Ack=101 Win=131648 Len=0 TSval=22 TStamp=5.700
11	5.70...	192.168.31.59	192.168.31.222	TCP	66	80 → 64848 [ACK] Seq=101 Ack=19 Win=29056 Len=0 TSval=22 TStamp=5.700

图 15.2: HTTP/0.9 TCP

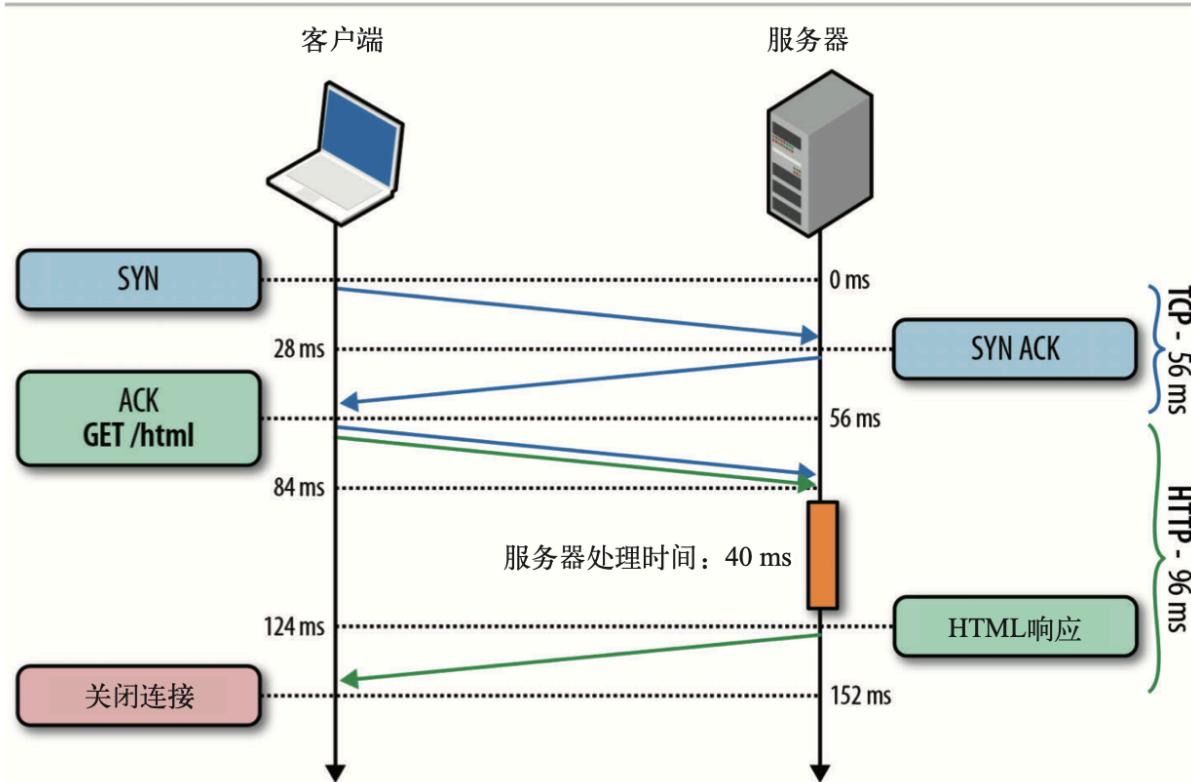


图 15.3: HTTP/0.9 TCP 请求流程图

15.1.2 HTTP/1.0

- 由起始行、首部、主体构成的请求和响应报文
- 响应对象不局限于超文本
- 服务器与客户端之间的连接在每次请求之后默认都会关闭
- 等等.....

HTTP/1.0 中反向移植了 HTTP/1.1 持久化连接，通过采用 `Connection: Keep-Alive` 可选首部参数。

重现 1.0 时代

```
telnet 192.168.31.59 80      # 命令
```

```
Trying 192.168.31.59...
Connected to 192.168.31.59.

GET /index.html HTTP/1.0          # 命令
User-Agent: This is from telnet  # 命令

HTTP/1.0 200 OK
Server: nginx/1.10.3 (Ubuntu)
Date: Thu, 02 Jul 2020 16:36:07 GMT
Content-Type: text/html
Content-Length: 99
Last-Modified: Thu, 02 Jul 2020 16:06:47 GMT
Connection: close
ETag: "5efe0617-63"
Accept-Ranges: bytes

output something...

Connection closed by foreign host.
```

15.2 HTTP/1.1

-
- 持久连接 persistent connection
- 管道化连接 pipeline connection

15.2.1 持久连接

* 在事务处理结束之后仍然保持在打开状态的 TCP

重用对目标服务器打开的空闲持久连接，就可以避免缓慢的连接建立阶段。而且已经打开的连接还可以避免慢启动的拥塞适用阶段。

HTTP/1.1 持久化连接在默认情况下激活的，除非特别指定，否则所有 HTTP/1.1 所有的连接都是持久化的。如果想要关闭连接需要添加 Connection: close

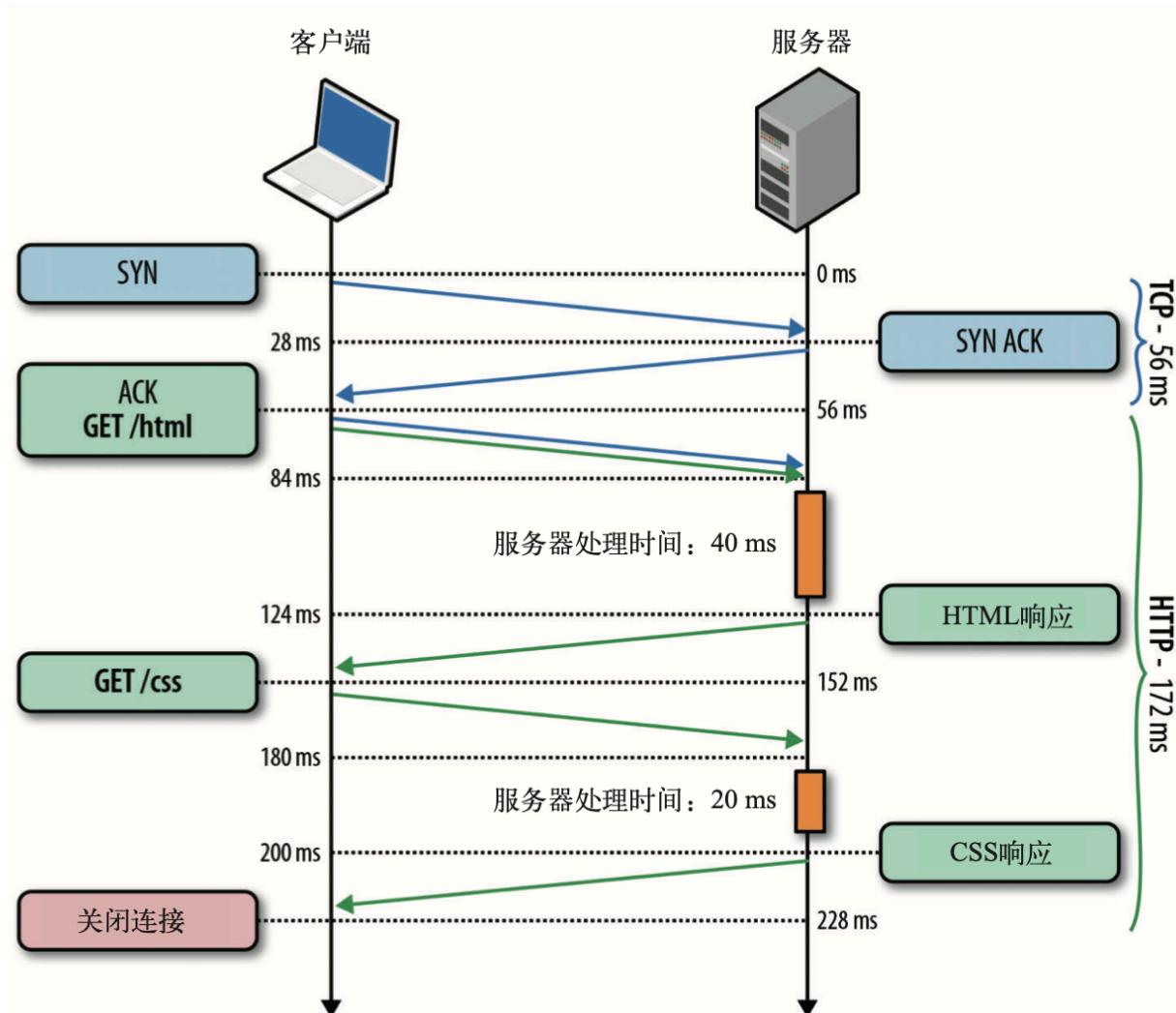


图 15.4: HTTP/1.1 持久化 TCP 请求流程图

但是我们的请求还是必须严格按照先进先出（FIFO）的队列顺序：发送请求，等待响应完成，再发送客户端队列中的下一个请求。

15.2.2 管道化连接

通过共享的 TCP 连接发起并发的 HTTP 请求

管道化连接是在持久连接的基础上进行的，算是对持久化的优化。把客户端的 FIFO 队列（请求队列）迁移到服务器（响应队列）。

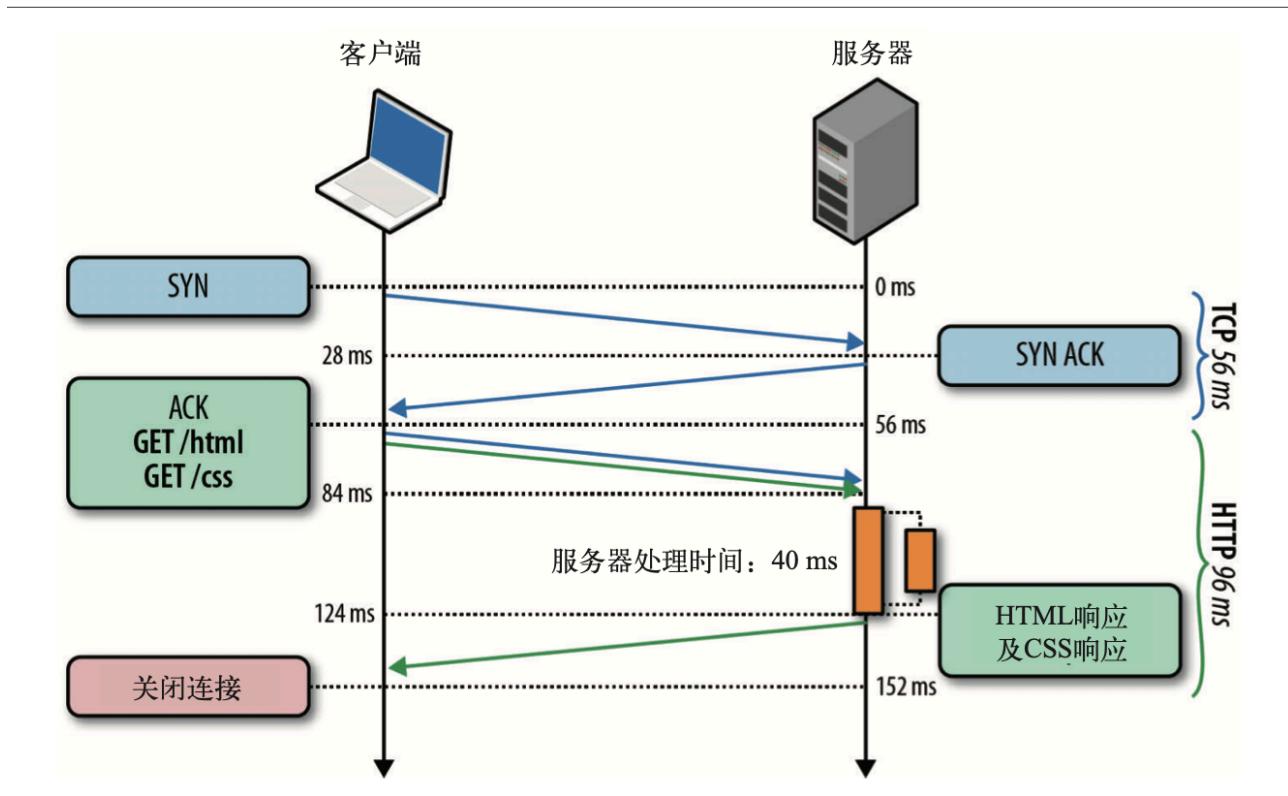


图 15.5: HTTP/1.1 持久化 TCP 请求流程图

严格串行响应，必须按照与请求相同的顺序回送 HTTP 响应，不支持交错到达（多路复用）

HTTP 报文中没有序列号标签，如果响应失序了，就没办法与请求匹配了。

按序交付就会导致队首阻塞问题，不能充分利用网络连接，而且会造成服务器缓冲开销等等。

不应该用管道的方式发送非幂等请求

例如一系列 POST 请求，出错的时候我们无法确定哪些被正确执行。

目前大部分浏览器是禁用管道技术的。

15.2.3 web 优化

并行连接；浏览器默认开启 6–8 个 TCP 连接

压缩

缓存

.....

15.3 HTTP/2

SPDY(speedy) 是 HTTP 2.0 的基础。

HTTP/2 要解决的问题：

提升性能；

解决 HTTP 中的“队首阻塞”问题；

并行操作无需与服务器建立多个连接，改进 TCP 的利用率，尤其是拥塞控制方面；

兼容 HTTP/1.1 的语义；

- One TCP connection
- Binary framing layer
- Streams and Multiplexing
- Header compression (HPACK)

HTTP/2 直观感受

[A grid of 180 tiled images is below. Compare](#)

15.3.1 Connection

- TCP 连接三次握手
- TCP 慢启动拥塞控制
- 用户捎带确认的 TCP 延迟确认算法
- TIME_WAIT 时延和端口耗尽

HTTP/2 两个标识符

- h2c
- h2

h2c

表示 HTTP/2 协议运行在明文 TCP 上。

标识通过明文 TCP 运行 HTTP/2 的协议。此标识符用于 HTTP/1.1 升级标头字段以及标识 HTTP/2 over TCP 的任何位置。

h2

表示 HTTP/2 使用传输层安全性 (TLS) 的协议。用于 TLS 应用层协议协商 (ALPN)。

通过 HTTP Uri 启动连接

发起请求

```
GET /index.html HTTP/1.1
host: nghttp2.org
connection: Upgrade, HTTP2-Settings
upgrade: h2c
http2-settings: AAMAAABkAAQAAP__
```

如果不支持 HTTP/2，服务忽略 h2c 首部字段，返回一个不包含升级的首部响应。

```
HTTP/1.1 200 OK
...
```

如果服务端支持 HTTP/2 则返回 101 协议来接受升级请求。101 协议响应结束后，服务端开始发送包含 HTTP/2 帧的请求，第一个被服务端发送的帧是设置帧。客户端在收到 101 协议响应后，也发送一个包含设置帧的请求。

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c
```

```
root@pluto:/# nghttp -nvu http://nghttp2.org
[ 0.179] Connected
[ 0.182] HTTP Upgrade request
GET / HTTP/1.1
host: nghttp2.org
connection: Upgrade, HTTP2-Settings
upgrade: h2c
http2-settings: AAMAAABkAAQAAAP__
accept: */*
user-agent: nghttp2/1.30.0

[ 0.302] HTTP Upgrade response
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ 0.302] HTTP Upgrade success
[ 0.303] recv SETTINGS frame <length=24, flags=0x00, stream_id=0>
          (niv=4)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):1048576]
          [UNKNOWN(0x08):1]
          [SETTINGS_HEADER_TABLE_SIZE(0x01):8192]
[ 0.303] send SETTINGS frame <length=12, flags=0x00, stream_id=0>
          (niv=2)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
[ 0.303] send SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
```

图 15.6: nghttp 模拟 HTTP/1.1 Upgrade

通过 HTTPS Uri 启动连接

ALPN(应用层协议协商)

```

TRANSMISSION CONTROL PROTOCOL, SRC PORT 57200, DST PORT 443, Seq. 1, ACK 552, Len 586
▼ Transport Layer Security
  ▼ TLSv1.3 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 586
  ▼ Handshake Protocol: Client Hello
    Handshake Type: Client Hello (1)
    Length: 582
    Version: TLS 1.2 (0x0303)
    Random: 86d0428c1e2ef6c83856f42a57f56e354c5e1d3f01e864f2...
    Session ID Length: 32
    Session ID: 22416391523c79c228fe3e702dc3fc3ecfa21e5f3596bda7...
    Cipher Suites Length: 32
    ▶ Cipher Suites (16 suites)
      Compression Methods Length: 1
    ▶ Compression Methods (1 method)
      Extensions Length: 477
    ▶ Extension: Reserved (GREASE) (len=0)
    ▶ Extension: server_name (len=16)
    ▶ Extension: extended_master_secret (len=0)
    ▶ Extension: renegotiation_info (len=1)
    ▶ Extension: supported_groups (len=10)
    ▶ Extension: ec_point_formats (len=2)
    ▶ Extension: session_ticket (len=0)
  ▼ Extension: application_layer_protocol_negotiation (len=14)
    Type: application_layer_protocol_negotiation (16)
    Length: 14
    ALPN Extension Length: 12
  ▼ ALPN Protocol
    ALPN string length: 2
    ALPN Next Protocol: h2
    ALPN string length: 8
    ALPN Next Protocol: http/1.1
  ▶ Extension: status_request (len=5)

```

图 15.7: 客户端发起 Client Hello 协议

```

TRANSMISSION CONTROL PROTOCOL, SRC PORT 443, DST PORT 57200, Seq. 1, ACK 552, Len 586
▼ Transport Layer Security
  ▶ TLSv1.3 Record Layer: Handshake Protocol: Server Hello
  ▶ TLSv1.3 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
  ▼ TLSv1.3 Record Layer: Handshake Protocol: Encrypted Extensions
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 32
    [Content Type: Handshake (22)]
  ▼ Handshake Protocol: Encrypted Extensions
    Handshake Type: Encrypted Extensions (8)
    Length: 11
    Extensions Length: 9
  ▼ Extension: application_layer_protocol_negotiation (len=5)
    Type: application_layer_protocol_negotiation (16)
    Length: 5
    ALPN Extension Length: 3
  ▼ ALPN Protocol
    ALPN string length: 2
    ALPN Next Protocol: h2
  ▶ TLSv1.3 Record Layer: Handshake Protocol: Finished

```

图 15.8: 服务端发起 Server Hello 协议

HTTP/2 Connection Preface

在 HTTP/2 中，每个端点都需要发送连接前奏作为正在使用的协议的最终确认，并建立 HTTP/2 连接的初始设置。客户端和服务器各自发送不同的连接前奏。

客户端连接前奏以 24 个八位字节的序列开始，以十六进制表示法为：

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

```
# 字符串格式: PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n
```

该序列必须后跟可为空的 SETTINGS 帧。

服务器连接前奏包含一个可为空的 SETTINGS 帧，该帧必须是服务器在 HTTP/2 连接中发送的第一帧。

服务器可以在客户端发送附加帧之前发送 SETTINGS，从而提供避免此问题的机会。

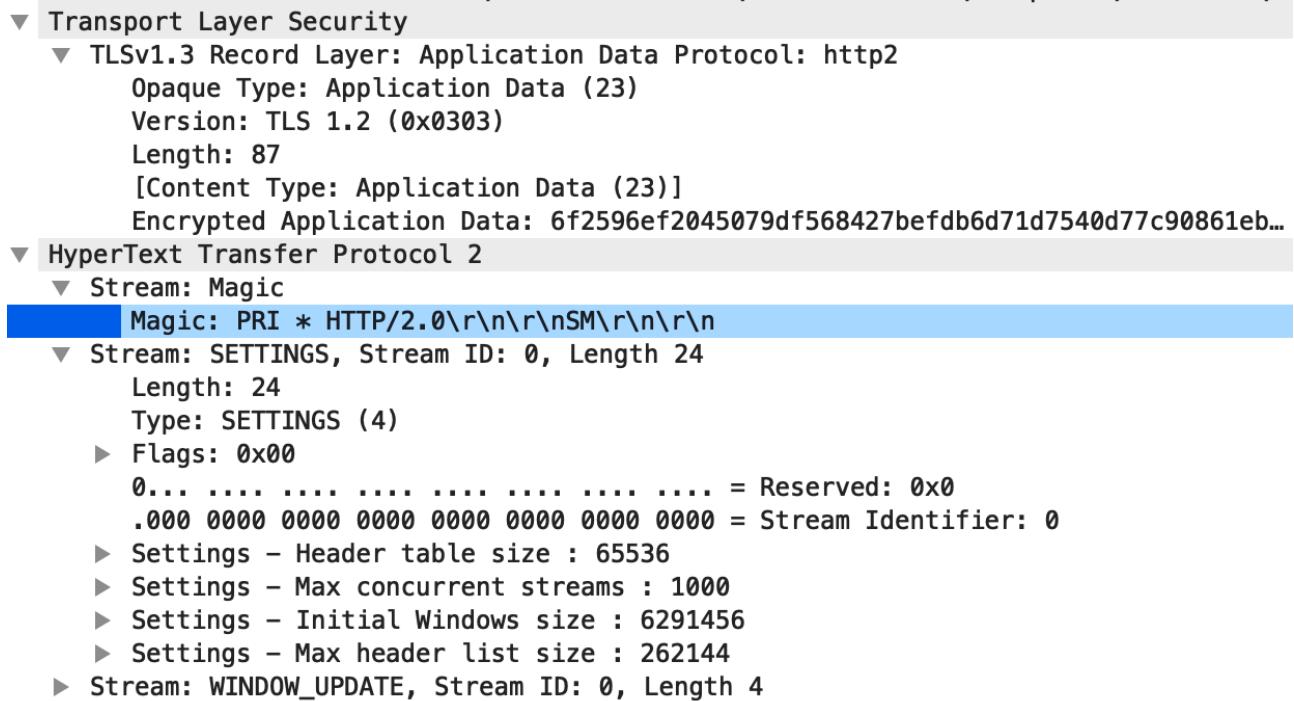


图 15.9: Connection Preface prism

15.3.2 Frames

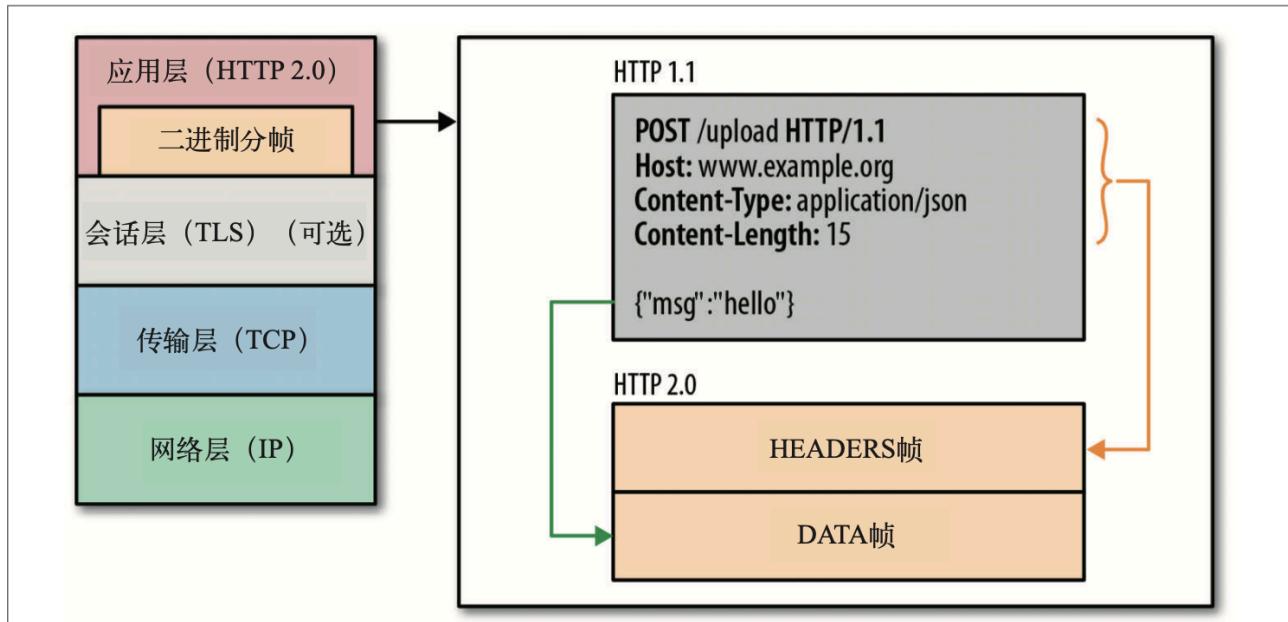


图 15.10: 二进制分帧层

所有 **HTTP 2.0** 通信都在一个连接上完成，这个连接可以承载任意数量的双向数据流。相应地，每个数据流以消息的形式发送，而消息由一或多个帧组成，这些帧可以乱序发送，然后再根据每个帧首部的流标识符重新组装。

HTTP 2.0 的二进制分帧机制解决了 **HTTP 1.x** 中存在的队首阻塞问题。

Frame Layout

帧就是 **HTTP 2.0** 通信的最小单位，每个帧包含帧首部，至少也会标识出当前帧所属的流。

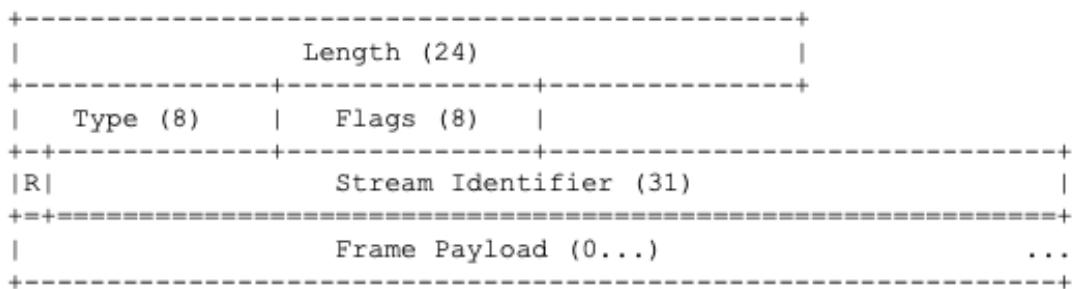


图 15.11: HTTP/2 帧结构

名称	长度	描述
Length	3 字节	表示帧负载的长度(取值范围为 $2^{14} \sim 2^{24}-1$)。默认最大帧大小是 2^{14} 。可以通过 SETTINGS 帧修改。
Type	1 字节	当前帧类型
Flags	1 字节	具体帧类型的标识
R	1 位	保留位
Stream Identifier	31 位	每个流的唯一 ID
Frame Payload	长度可变	真实帧内容；长度设置在 Length 中

表 15.1: 帧字段

前 9 个字节是帧的固定结构。解析时只需要读取这些字节。

名称	ID	描述
DATA	0x0	流核心内容
HEADERS	0x1	HTTP 首部以及可选的优先级参数
PRIORITY	0x2	指示或者更改流的优先级和依赖
RST_STREAM	0x3	允许一段停止流(通常由于错误导致的)
SETTINGS	0x4	协商连接级参数
PUSH_PROMISE	0x5	提示客户端，服务器要推送东西
PING	0x6	测试连接可用性和往返时延(RTT)
GOAWAY	0x7	告诉另一段，当前端已结束
WINDOW_UPDATE	0x8	用于流量控制；协商一段将要接受多少字节
CONTINUATION	0x9	用以扩展 HEADER 数据块

表 15.2: 帧类型

DATA

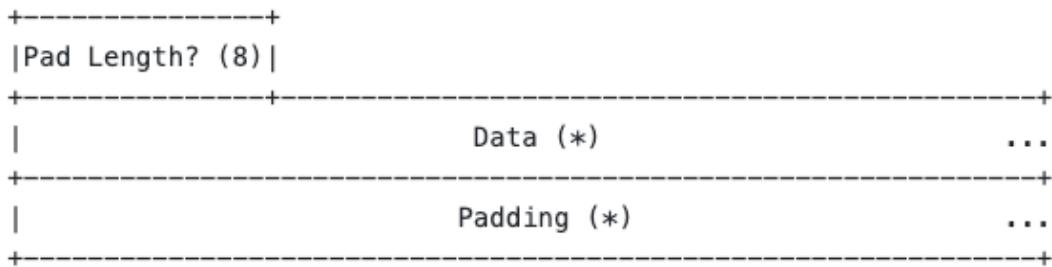


图 15.12: DATA 帧结构

DATA 帧包含以下几个字段：

Pad Length: 一个 8 位字段，包含以八位字节为单位的帧填充长度。该字段是有条件的，仅在设置了 PADDED 标志时才存在。

Data: 应用数据。在减去存在的其他字段的长度之后，**data** 的大小是帧有效载荷的剩余部分。

Padding: 填充的八位字节，它不包含应用程序语义值。发送时，填充的八位字必须设置为零。接收方没有义务验证填充，但可以将非零填充视为 PROTOCOL_ERROR 类型的连接错误。

DATA 帧定义了以下 flag 标识：

END_STREAM (0x1): 设置这个字段的时候，位 0 表示该帧是端点为将要发送的标识流的最后一帧。设置此标志会导致流进入“半关闭”状态或者“关闭”状态。

PADDED (0x8): 设置这个字段的时候，位 3 表示存在 Pad Length 字段及其描述的任何填充。

```

▼ HyperText Transfer Protocol 2
  ▼ Stream: DATA, Stream ID: 3, Length 691
    Length: 691
    Type: DATA (0)
    ▼ Flags: 0x01
      .... .1 = End Stream: True
      .... 0... = Padded: False
      0000 .00. = Unused: 0x00
      0.... .... .... .... .... .... = Reserved: 0x0
      .000 0000 0000 0000 0000 0000 0011 = Stream Identifier: 3
      [Pad Length: 0]
      Data: ffd8ffdb008400030202030203030304030304050805...
  
```

图 15.13: END_STREAM

DATA 帧必须与某一个流相互关联。如果接收到其流标识符字段为 0x0 的 DATA 帧，则接收方必须以 PROTOCOL_ERROR 类型的连接错误进行响应。

DATA 帧会受到流量控制，只能在流处于“打开”或“半关闭（远程）”状态时发送。整个 DATA 帧有效载荷包含在流量控制中，包括 Pad Length 和 Padding 字段（如果存在）。如果收到的数据帧的流不是“打开”或“半关闭（本地）”状态，则接收方必须以 STREAM_CLOSED 类型的流错误进行响应。

填充八位字节的总数由填充长度字段的值确定。如果填充的长度是帧有效负载的长度或更长，则接收方必须将其视为 PROTOCOL_ERROR 类型的连接错误。

HEADERS

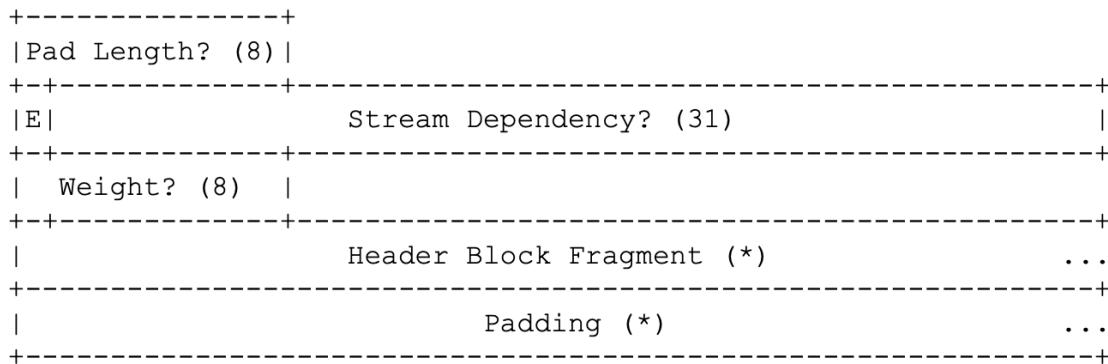


图 15.14: HEADERS 帧结构

Pad Length: 指定 Padding 长度, 存在则代表 PADDING flag 被设置

E: 一个比特位声明流的依赖性是否是排他的, 存在则代表 PRIORITY flag 被设置

Stream Dependency: 指定一个 stream identifier, 代表当前流所依赖的流的 id, 存在则代表 PRIORITY flag 被设置

Weight: 一个无符号 8 为整数, 代表当前流的优先级权重值 (1~256), 存在则代表 PRIORITY flag 被设置

Header Block Fragment: header 块片段

Padding: 填充字节, 没有具体语义, 作用与 DATA 的 Padding 一样, 存在则代表 PADDING flag 被设置

HEADERS 帧有以下标识 (flags):

END_STREAM: bit 0 设为 1 代表当前 header 块是发送的最后一块, 但是带有 END_STREAM 标识的 HEADERS 帧后面还可以跟 CONTINUATION 帧 (这里可以把 CONTINUATION 看作 HEADERS 的一部分)

END_HEADERS: bit 2 设为 1 代表 header 块结束

PADDED: bit 3 设为 1 代表 Pad 被设置, 存在 Pad Length 和 Padding

PRIORITY: bit 5 设为 1 表示存在 Exclusive Flag (E), Stream Dependency, 和 Weight

```

▼ HyperText Transfer Protocol 2
  ▼ Stream: HEADERS, Stream ID: 253, Length 68, GET /gophertiles?x=5&y=8&cachebust=1594096435592712014&latency=0
    Length: 68
    Type: HEADERS (1)
    ▼ Flags: 0x25
      .... .1. = End Stream: True
      .... .1.. = End Headers: True
      .... 0... = Padded: False
      ..1. .... = Priority: True
      00.0 ..0. = Unused: 0x00
      .... .... .... .... .... = Reserved: 0x0
      .000 0000 0000 0000 0000 1101 1101 = Stream Identifier: 253
      [Pad Length: 0]
      1.... .... .... .... .... .... = Exclusive: True
      .000 0000 0000 0000 0000 1101 1011 = Stream Dependency: 251
      Weight: 146
      [Weight real: 147]
      Header Block Fragment: 82cd870084b958d33fac6263d7396c49a82a3fcf3037f1ea...
      [Header Length: 587]
      [Header Count: 13]
    ▶ Header: :method: GET
    ▶ Header: :authority: http2.golang.org
    ▶ Header: :scheme: https

```

图 15.15: HEADERS 帧结构

SETTINGS

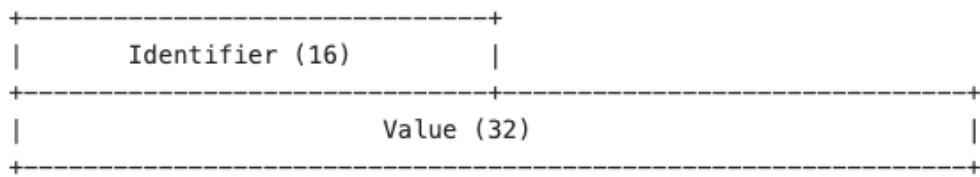


图 15.16: SETTINGS 帧结构

SETTINGS_HEADER_TABLE_SIZE(0x1)
SETTINGS_ENABLE_PUSH(0x2)
SETTINGS_MAX_CONCURRENT_STREAMS(0x3)
SETTINGS_INITIAL_WINDOW_SIZE(0x4)
SETTINGS_MAX_FRAME_SIZE(0x5)
SETTINGS_MAX_HEADER_LIST_SIZE(0x6)

15.3.3 Stream

HTTP/2 连接上独立的，双向的帧序列交换。

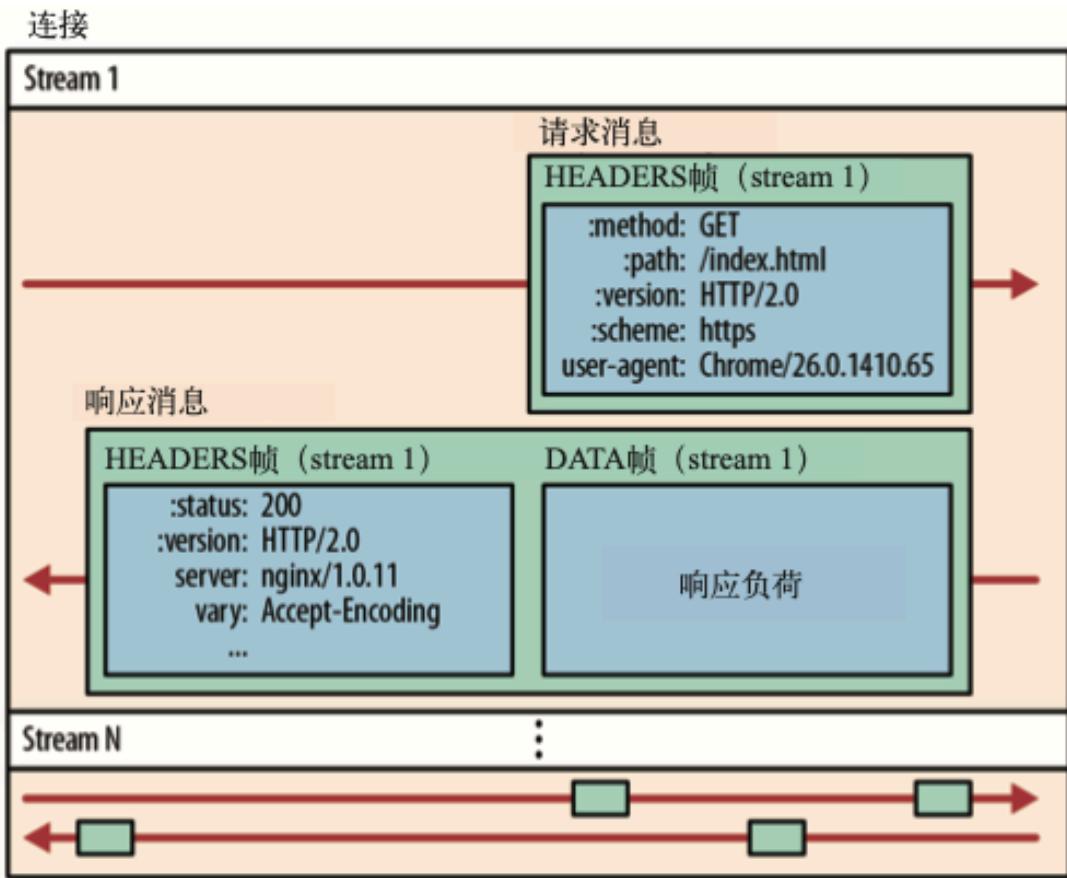


图 15.17: 流、消息和帧

客户端到服务端的 h2 连接建立之后，通过发送 HEADERS 帧来启动新的流，如果首部需要跨多个帧，可能还会发送 CONTINUATION 帧。后续流启动会发送一个带有自增 Stream Identifier 的新 HEADERS 帧。



HEADERS 和 CONTINUATION 帧必须是有序的。

stream 流使用无符号的 31 位整数标识。由客户端发起的流必须使用奇数编号的流标识符。

服务器发起的必须使用偶数编号的流标识符。

流标识符零 ($0x0$) 用于连接控制消息；零流标识符不能用于建立新的 **stream** 流。

HTTP/1.1 Upgrade to HTTP/2 时响应的流 ID 是 $0x1$ ，在升级完成之后，流 $0x1$ 在客户端会转为 half-closed (local) 状态，因此这种情况下客户端不能用 $0x1$ 初始化一个流。因为 HTTP/1.1 请求已经完成了。

流生命周期

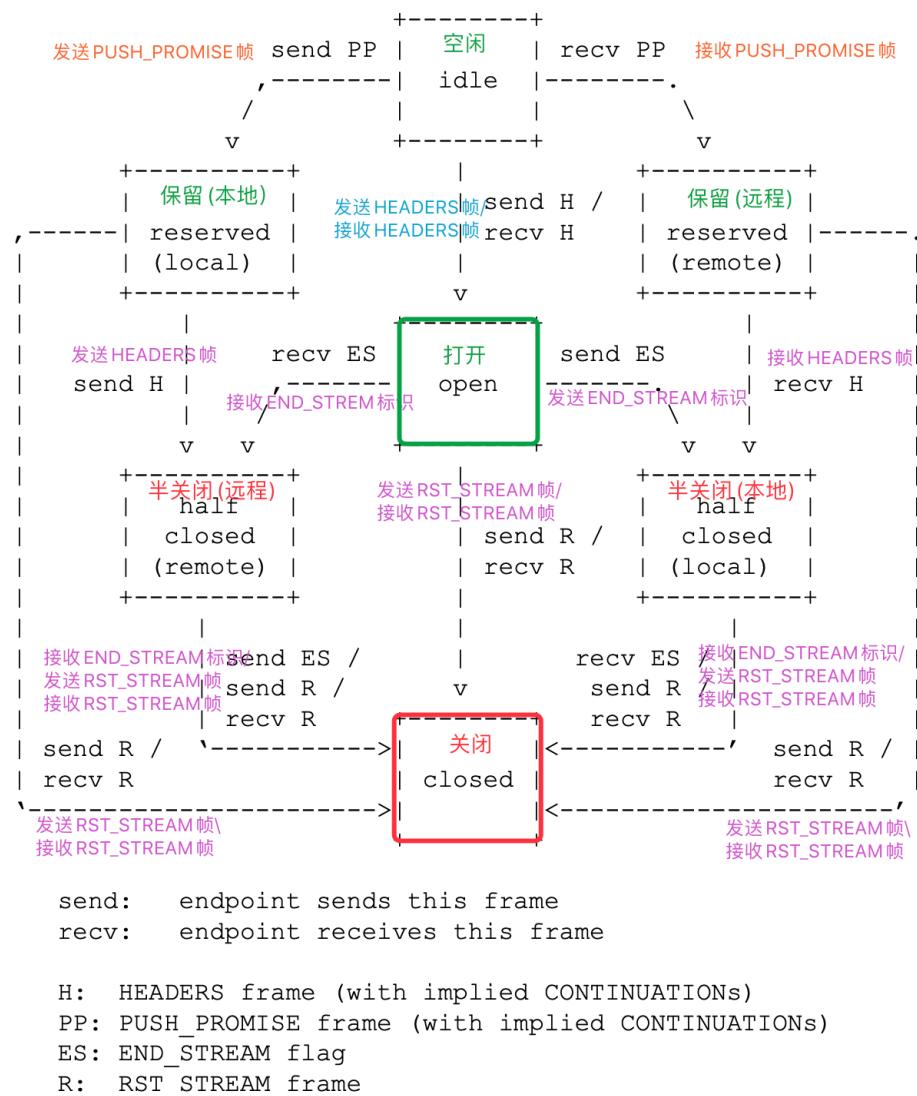


图 15.18: 流生命周期

idle

所有的 stream 流都是从空闲态开始的。

reserved

reserved, 为推送保留一个流稍后使用。

reserved(local)

服务器端发送完 PUSH_PROMISE 帧本地预留的一个用于推送流所处于的状态

只能发送 HEADERS、RST_STREAM、PRIORITY 帧

只能接收 RST_STREAM、PRIORITY、WINDOW_UPDATE 帧

reserved(remote)

客户端接收到 PUSH_PROMISE 帧, 本地预留的一个用于接收推送流所处于的状态

只能发送 WINDOW_UPDATE、RST_STREAM、PRIORITY 帧

只能接收 RST_STREAM、PRIORITY、HEADERS 帧

不满足条件，需要报 PROTOCOL_ERROR 类型连接错误

open

用于两端发送帧，需要发送数据的对等端需要遵守流量控制的通告。

每一端可以发送包含 END_STREAM 标志位的帧，导致流进入"half closed" 状态

每一端都可以发送 RST_STREAM 帧，流进入"closed" 状态

half closed

half closed(local)

发送包含有 END_STREAM 标志位帧的一端，流进入本地半关闭状态

不能发送 WINDOW_UPDATE、PRIORITY 和 RST_STREAM 帧

可以接收到任何类型帧

接收者可以忽略 WINDOW_UPDATE 帧，后续可能会马上接收到包含有 END_STREAM 标志位帧

接收到优先级 PRIORITY 帧，可用来变更依赖流的优先级顺序

一旦接收到包含 END_STREAM 标志位的帧，将进入"closed" 状态

half closed(remote)

接收到包含有 END_STREAM 标志位帧的一端，流进入远程半关闭状态

对流量控制窗口可不用维护

只能接收 RST_STREAM、PRIORITY、WINDOW_UPDATE 帧，否则报 STREAM_CLOSED 流错误

终端可以发送任何类型帧，但需要遵守对端的当前流的流量控制限制

一旦发送包含 END_STREAM 标志位的帧，将进入"closed" 状态

一旦接收或发送 RST_STREAM 帧，流将进入"closed" 状态

closed

流的最终关闭状态

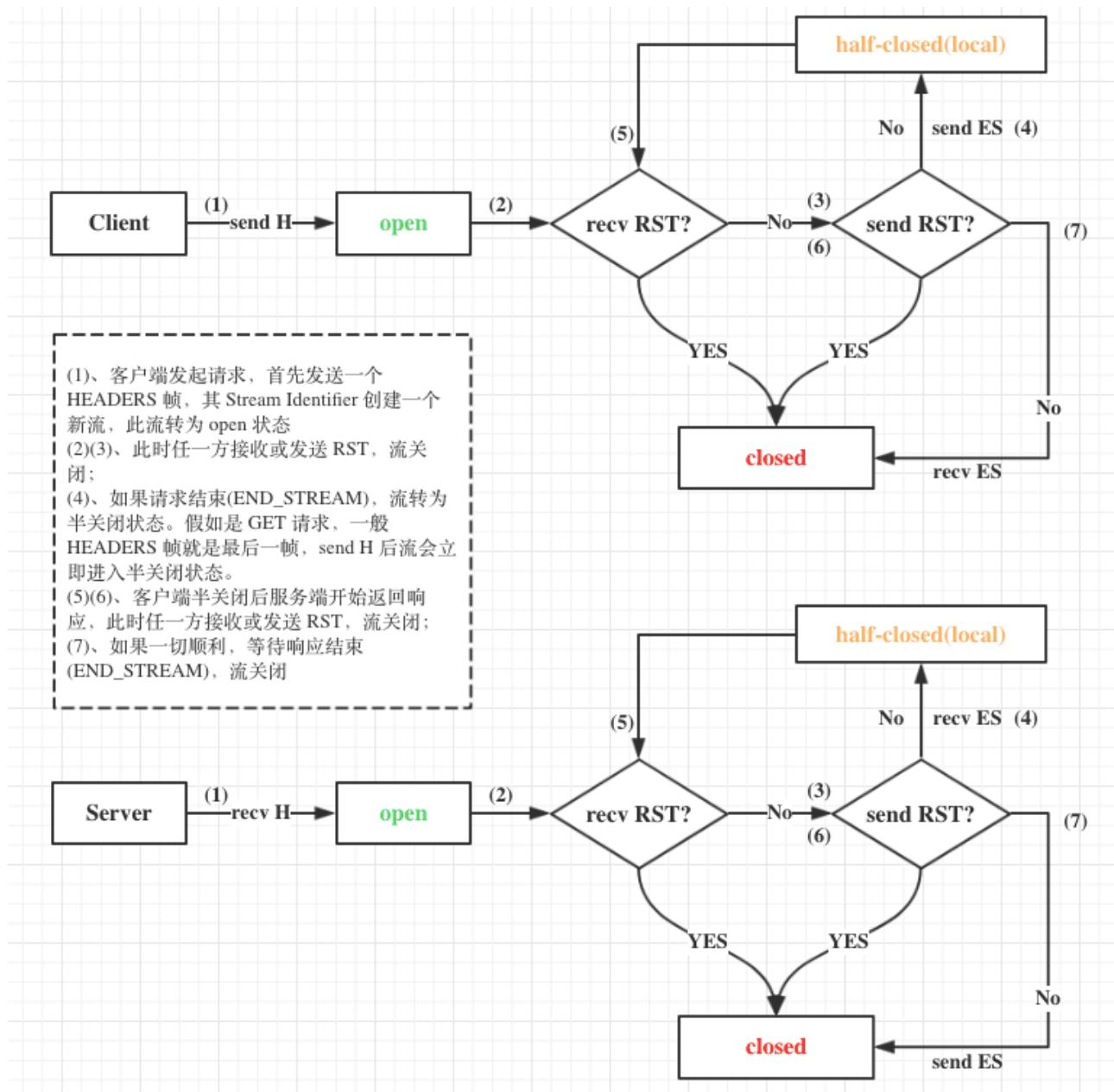


图 15.19: 请求/响应流状态转化

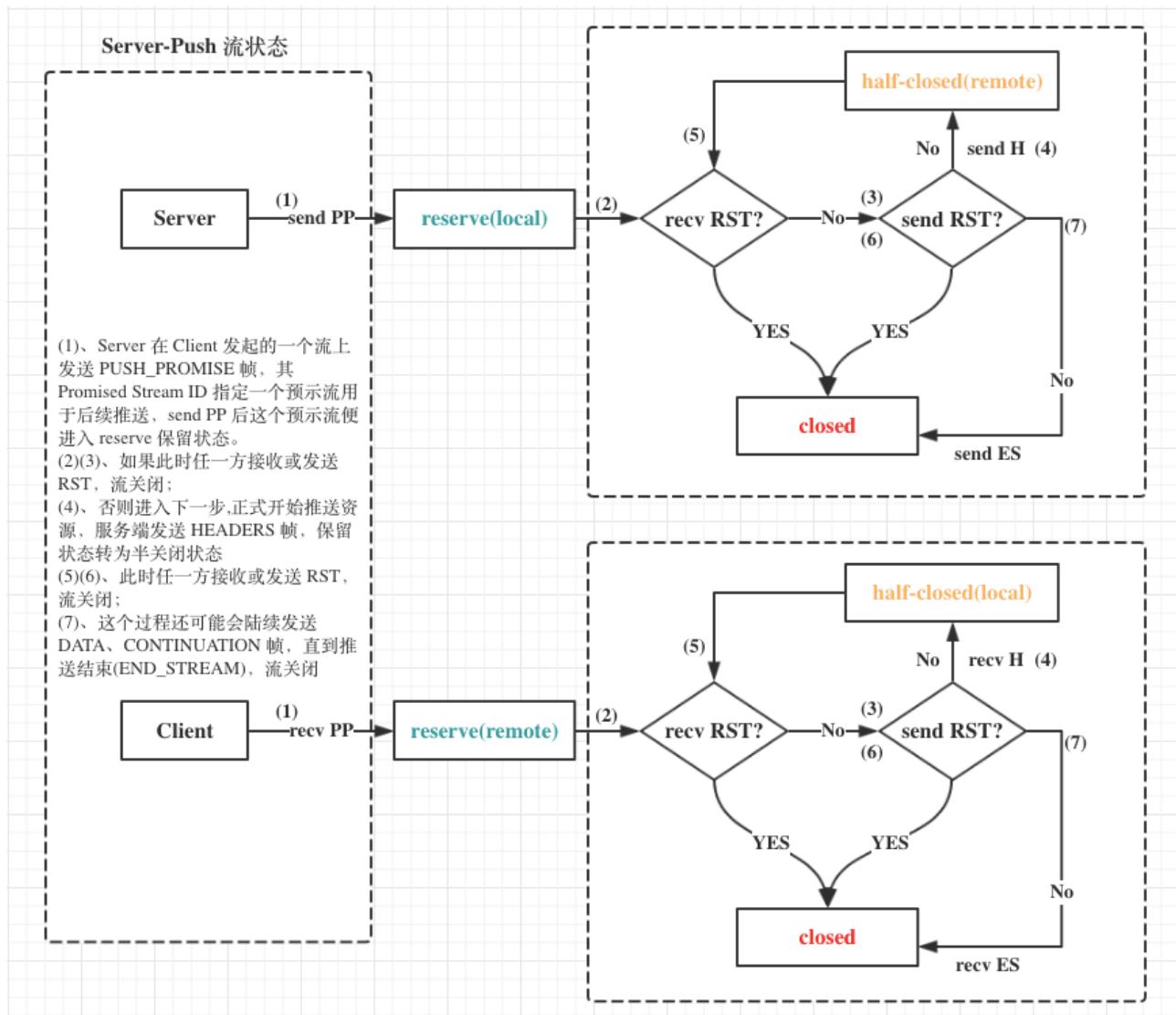


图 15.20: Server Push

流优先级

- 依赖关系
- 权重

客户端可以通过在打开流的 **HEADERS** 帧中包含优先级信息来为新的流分配优先级。在任何其他时间, **PRIORITY** 帧可用于更改流的优先级。

确定优先级的目的是允许端点在管理并发流时表达它希望其对端如何分配资源。最重要的是, 当发送容量有限时, 可以使用优先级来选择用于发送帧的流。

可以通过将流标记为依赖其他流的完成, 来确定流的优先级。为每个依赖项分配一个相对权重, 该数字用于确定分配给依赖于相同流的 **stream** 流的可用资源的相对比例。

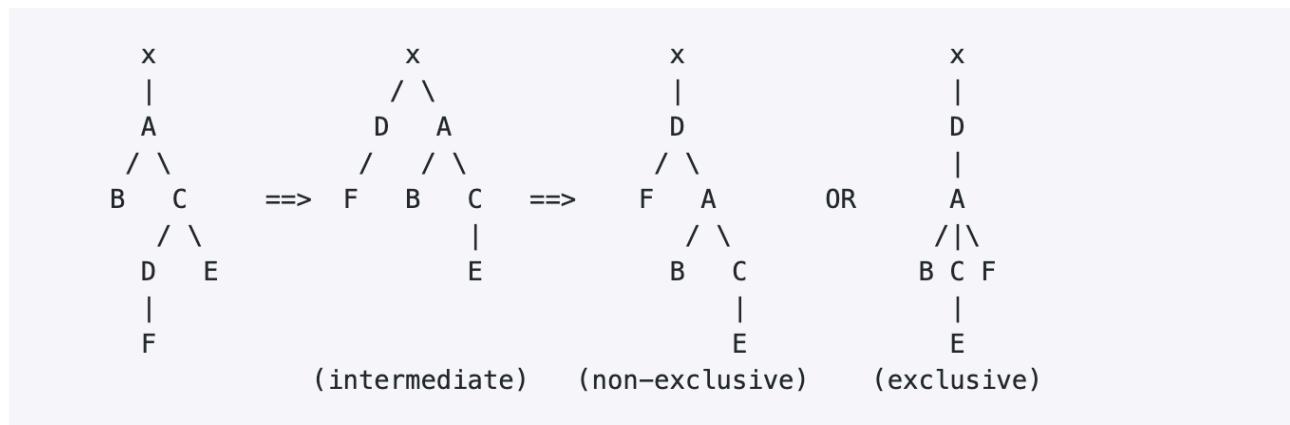


图 15.21: 依赖优先级调整

D 优先级变高

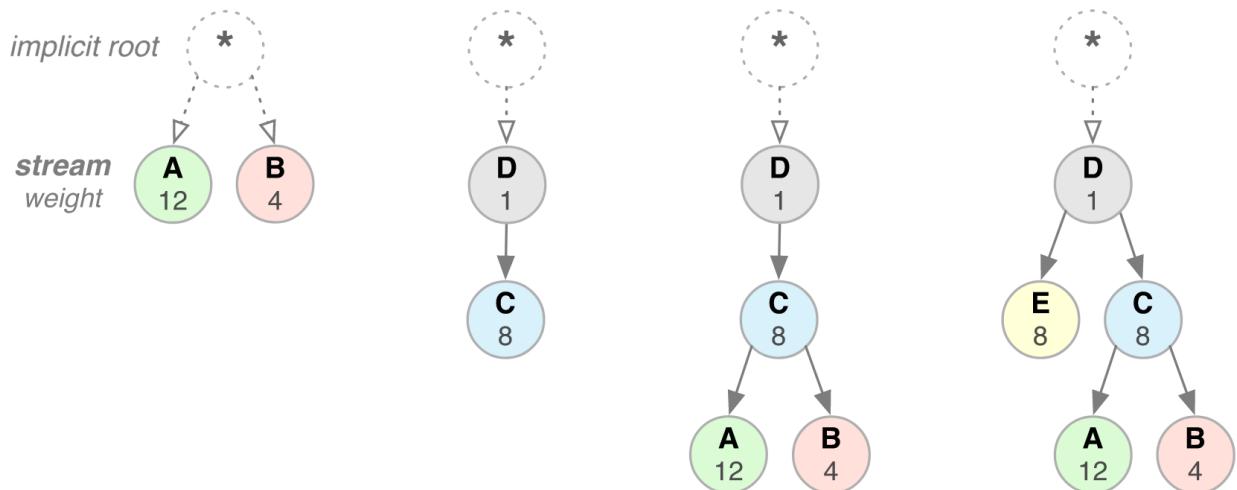


图 15.22: 依赖以及权重

流量控制

HTTP/2 通过使用 WINDOW_UPDATE 帧提供流量控制

只有 DATA 帧受流量控制

流控制窗口的默认值设为 65,535 字节

流量控制示例

在流建立的时候，窗口大小默认都是 $65\ 535$ ($2^{16}-1$) 字节。假设客户端 A 支持该默认值，它的另一端 (B) 发送了 10 000 字节，B 也会关注窗口大小 (现在有 55 535 字节了)。现在 A 花时间处理了 5000 字节，还剩下 5000 字节，然后它会发送一个 WINDOW_UPDATE 帧，说明它现在的窗口大小是 60 535 字节。B 收到这个帧之后，开始发送一个大文件 (比如 4GB 大小)。在这个场景下，在 B 等 A 准备好接收更多的数据之前，B 能发送的数据量就是当前窗口的大小，即 60 535 字节。通过这种方式，A 可以控制 B 发送数据的最大速率。

图 15.23: 流量控制示例

服务器推送

[http2-server-push](#)

15.4 HPACK

15.4.1 Deflate

LZ77 + Huffman

CRIME 漏洞

攻击者在请求中 (cookie) 添加数据，观察压缩加密后的数据量是否会小于预期，如果变小了，注入的文本和请求中的其他内容有重复。

15.4.2 Huffman Coding

[动画演示](#)

15.4.3 索引表

静态表 (Static Table) 和动态表 (Dynamic Table)

静态表

静态索引表由预定义的首部字段构成。包含 61 个预定义 Header key-value。

[静态索引表字段](#)

动态表

15.4.4 实现

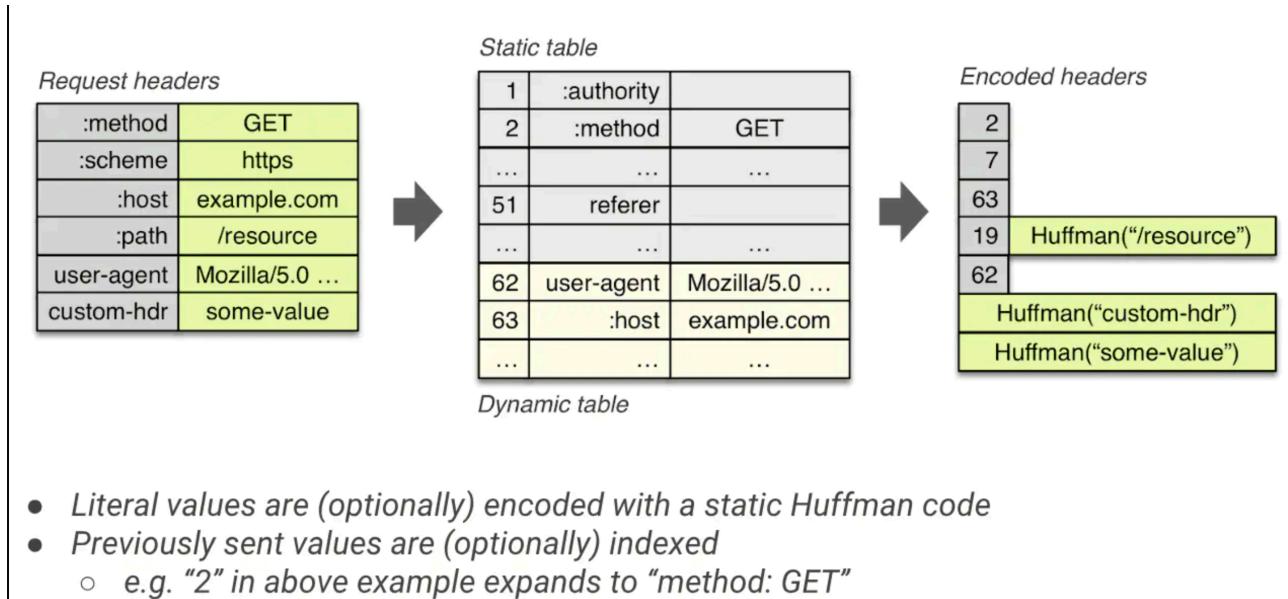


图 15.24: Header 压缩示例

.....

15.5 参考

[Hypertext Transfer Protocol Version 2 \(HTTP/2\)](#)

[HPACK: Header Compression for HTTP/2](#)

[nghttp2](#)

[http2.golang](#)

[mkcert](#)

[How to get HTTPS working on your local development environment](#)

[http2-spec](#)

[《web 性能权威指南》](#)

	standard font size		
command	10pt	11pt	12pt
\tiny	5pt	6pt	6pt
\scriptsize	7pt	8pt	8pt
\footnotesize	8pt	9pt	10pt
\small	9pt	10pt	11pt
\normalsize	10pt	11pt	12pt
\large	12pt	12pt	14pt
\Large	14pt	14pt	17pt
\LARGE	17pt	17pt	20pt
\huge	20pt	20pt	25pt
\Huge	25pt	25pt	25pt

article, report, book und letter standard font size